FRANCESCO RICCI, VINCENZO FIORENTINI

FONDAMENTI DI FISICA COMPUTAZIONALE

DRAFT - MAY 7, 2018

Indice

Nota

Questo testo è una sintesi di diversi anni di insegnamento di corsi di Fisica Computazionale di base di diverso livello. È diretto a un livello di conoscenza basso: smanettoni e teste d'uovo si adatteranno a ripassare e a far pratica su tecniche di programmazione semplici. Dopo aver visitato in passato territori complicati e remoti (fino ad esempio alle equazioni integrali...), col passare del tempo la tendenza al minimale ha prevalso, ed è stato inevitabile eliminare e semplificare drasticamente. Questo non significa che alcuni argomenti siano sottili e potenzialmente di interesse molto vasto.

La scelta del sistema operativo Unix (robusto, usatissimo, flessibile, e alla base di tutti i sistemi operativi commerciali) è obbligatoria. Ne esistono versioni open-source (i vari dialetti di Linux) o proprietarie (MacOSX), facilmente reperibili e ben mantenute. Benchè per i linguaggi la scelta sia più difficile, negli ultimi anni il python è emerso come una eccellente alternativa ai classici C e Fortran (anche nelle loro incarnazioni moderne). In particolare, i notebook python offrono una maniera potente per riassumere molto materiale che a stampa sarebbe difficilmente digeribile.

Il testo è quindi diviso in capitoli "sdoppiati" in una parte teorica generale e una di programmazione elementare in python; i tutorial relativi a queste ultime sezioni sono disponibili in rete. Vengono anche forniti in rete dei notebook python che riassumono ed elaborano il grosso degli argomenti svolti nel testo, e forniscono una quantità di esempi discussi solo brevemente nel testo.

Come noto, un libro non si finisce: lo si abbandona. Se avete interesse, suggerite modifiche, aggiunte, tagli. Eviteremo di abbandonarlo del tutto.

1. A - Introduzione

La descrizione di un sistema o fenomeno fisico tramite simulazione numerica richiede la costruzione di un modello, con tutte le relative semplificazioni e omissioni (che di solito elegantemente chiamiamo approssimazioni), e la sua traduzione in –o descrizione entro lo schema di– un algoritmo di calcolo. Ci sono numerose fonti di errore possibili (la rappresentazione dei numeri, l'accuratezza dei parametri, lo specifico algoritmo scelto) di cui bisogna essere consci.

Supponiamo di voler calcolare il periodo di un pendolo. Assumiamo che la massa sia sospesa a un filo indeformabile e che si possa trascurare la resistenza del mezzo in cui essi si muovono; queste ipotesi possono essere più o meno ragionevoli, o condurre a errori più o meno grandi a seconda dei casi. Assumiamo poi che la lunghezza del filo e la massa del peso siano note esattamente; questo non è vero, ovviamente, essendoci errori inevitabili nella loro misura. Sul peso agisce solamente la forza di gravità, che dipende dalla massa, di cui si è appena detto, e dalla accelerazione di gravità $g \simeq 9.81\ \mathrm{m/s^2}$. Quest'ultima dipende dalla distanza della massa dal centro della Terra e dalle fluttuazioni di densità entro la Terra stessa – tutte cose trascurate assumendola costante. (A rigore, poi, la distanza del peso dal centro della Terra varia durante l'oscillazione del pendolo, e con essa varia g.) Trascurare queste piccole correzioni non causa errori apprezzabili per un piccolo pendolo armonico, ma potrebbe essere sconsigliabile se si studiasse un pendolo anarmonico (quindi con grande elongazione) di grande estensione.

Usando la seconda legge della dinamica (equazione di Newton),

$$m\mathbf{a} = \mathbf{F} \tag{1.1}$$

l'equazione differenziale da cui ottenere l'elongazione angolare istantanea del pendolo rispetto alla verticale è

$$m\ell \frac{d^2\theta}{dt^2} = -mg \sin\theta, \tag{1.2}$$

dove, avendo assunto un filo indeformabile, si è considerata solo la componente della forza ortogonale al filo. Poichè la funzione cercata è argomento di una funzione trascendente, questa è una equazione differenziale ordinaria non-lineare del secondo ordine. In questo caso specifico esiste una complessa espressione analitica per il periodo (grazie al fatto che, per una data condizione iniziale, esso è una costante del moto), ma in generale, come vedremo, la soluzione numerica è l'unica strada possibile in simili casi. Per il momento, facciamo la cosiddetta approssimazione armonica, o di piccole oscillazioni. Sviluppiamo il secondo membro in serie di potenze per $\theta \sim 0$ ottenendo

$$\sin\theta \sim \theta. \tag{1.3}$$

Questa approssimazione è discretamente robusta: l'errore sulla forza per θ=0.1 rad=5.7° è circa 0.02%, e per un elongazione massima tripla (quindi oltre 15°) circa 0.5%. L'equazione diventa

$$\frac{d^2\theta}{dt^2} = -\frac{g}{\ell}\theta = -\omega^2\theta. \tag{1.4}$$

che si riconosce essere l'equazione di un oscillatore armonico di frequenza $\omega=\sqrt{g/\ell}$ e di periodo

$$T = \frac{2\pi}{\omega} = 2\pi\sqrt{\frac{g}{\ell}}. \tag{1.5}$$

Nel caso armonico, la frequenza, e con essa il periodo, non dipende dall'elongazione iniziale. (Vedremo in seguito come ottenere il periodo nel caso generale.)

Calcolando T da questa formula esatta (nell'approssimazione armonica) incorriamo inevitabilmente in vari errori. Oltre alle incertezze già evidenziate su g ed ℓ, dobbiamo rappresentare il numero irrazionale π con un numero (ovviamente) finito di cifre, calcolare la radice quadrata, e moltiplicare i vari fattori. Ognuna di queste operazioni *genera* e *propaga* errori, schematizzabili come dovuti a

- *dati*: incertezze sul valore o misura degli eventuali parametri e dei dati di input;
- *rappresentazione*: i numeri sono rappresentati con una precisione finita, come per esempio $\pi \simeq 3.14159265$;
- *aritmetica*: errori dovuti all'implementazione approssimata delle operazioni,

e noti genericamente –e impropriamente– come *errori di round-off* o di rappresentazione. In aggiunta, abbiamo *errori di approssimazione*, generati cioè dalle approssimazioni del modello fisico (p.es. quella armonica fatta prima). Infine, molto importanti, ci sono gli *errori algoritmici*, che originano dalla maniera in cui scegliamo di approssimare le operazioni di alto livello nel nostro algoritmo di simulazione, e che discuteremo in tutto il seguito del corso.

1.1 *Errori di round-off*

I numeri "umani" sono rappresentati in base 10, per ovvi motivi fisiologici. In un computer fisico, è più comodo rappresentare i nu-

meri in base 2, essendo tecnologicamente più abbordabile un sistema a due stati, come ad esempio una cella di memoria con o senza magnetizzazione o carica immagazzinata.

Con una *parola di memoria* composta da N oggetti a 2 stati, noti come *bit*, possiamo rappresentare 2^N numeri interi combinando i vari possibili stati di ognuno dei bit. Per esempio, se N=3, posso rappresentare 2^3=8 numeri, come mostrato in Tabella 1.1, con le tre cifre binarie a rappresentare i coefficienti di 2^0, 2^1, e 2^2.

Tabella 1.1: Numeri rappresentabili da una parola a 3 bit

0	0	0	$\rightarrow 0$
1	0	0	$\rightarrow 2^0$=1
0	1	0	$\rightarrow 2^1$=2
1	1	0	$\rightarrow 2^0+2^1$=3
0	0	1	$\rightarrow 2^2$=4
1	0	1	$\rightarrow 2^0+2^2$=5
0	1	1	$\rightarrow 2^1+2^2$=6
1	1	1	$\rightarrow 2^0+2^1+2^2$=7

Convenzionalmente, un numero reale in virgola mobile *a precisione singola* è rappresentato da *una parola* di memoria. Le parole di memoria dei computer moderni sono rappresentate 32 o 64 bit. Nelle architetture a 32 bit, riservando un bit per il segno, possiamo rappresentare i numeri interi da -2^{31} a $+2^{31}$ (circa $\pm 2\times 10^9$). Definito 1 byte=8 bit, un numero reale *in singola precisione* occupa 4 byte; uno che ne occupi 8, di dice a *doppia precisione*. Ad esempio, la designazione `REAL*8` indica i numeri reali in doppia precisione nei compilatori fortran a 32 bit. (Ovviamente se usiamo un architettura a 64 bit, dovremo raddoppiare il tutto, e la nomenclatura cambia.)

I numeri a virgola mobile, o *floating-point*, sono rappresentati come

$$x_{\text{float}} = (-1)^s \times m \times 2^{(E-b)} \tag{1.6}$$

I 32 bit vengono suddivisi in 1 per il segno (s=0 o 1), 8 per l'esponente E, e 23 per la mantissa m. Con 8 bit, l'esponente è $E\in$[0,255]. Scegliendo lo shift b=128, l'esponenziale cade nell'intervallo $[2^{-128},2^{127}]=[2.93\times 10^{-39},1.7\times 10^{38}]$. Una mantissa a 23 bit permette di rappresentare numeri da 0 a circa 8×10^6, e quindi il massimo e minimo numero rappresentabili stanno tra circa 10^{-40} e 10^{44}. (Con 64 bit, i bit dell'esponente sono 11, e i numeri rappresentabili sono enormi, all'incirca [10^{-320}, 10^{310}].) Altra deduzione che deriva da quanto sopra è che un processore che salvi gli indirizzi della memoria in parole a 32 bit potrà accedere al massimo a 2^{32} bit o circa 0.5×10^9 byte$\equiv$0.5 Gb (Giga-byte). L'accesso ormai universale ai processori a 64 bit ha reso possibile la gestione nativa di quantità di memoria pressochè illimitate (circa 2 miliardi di Gigabyte, o 2 Pb, o Petabyte).

Con grandi mantisse ed esponenti, non è ovvio a prima vista che la precisione dei numeri e dei risultati di operazioni tra numeri sia apprezzabilmente limitata agli effetti pratici. Per capire che questo è effettivamente (almeno potenzialmente) il caso, consideriamo un sistema che abbia una mantissa molto ridotta, tale da darci solo 3 cifre decimali, e facciamo la somma 7.000 10^0 + 1.000 10^{-2}. Aggiustati gli esponenti (1.000 10^{-2}=0.010 10^0) così da poter sommare direttamente le mantisse, si ottiene 7.010 10^0=7.010. Ma consideriamo la somma 7.000 10^0 + 1.000 10^{-4}: aggiustando gli esponenti, 1.000 10^{-4} diventa 0.000(1) 10^0, cioè 0.000, essendo la mantissa a tre cifre decimali. La somma è (7.000+0.000) 10^0=7.000. Si definisce *precisione di macchina* ϵ_m il minimo numero rappresentabile dalla mantissa; il numero che volevamo sommare a 7.000 è più piccolo della precisione di macchina del computer fittizio di questo esempio (circa 10^{-3}), e di fatto non viene sommato. Dun-

que, mentre nell'aritmetica l'elemento di identità additiva è lo zero, nell'aritmetica in rappresentazione finita che stiamo considerando

l'elemento di identità additiva non è unico,

poichè qualunque numero minore della precisione di macchina è equivalente allo zero. La precisione di macchina è facilmente stimabile come il minimo numero rappresentabile dalla mantissa. Quest'ultima in architettura a 32 bit è tipicamente di 23 bit, e in 64 bit è di 52, quindi

$$\epsilon_m = 2^{-23} \sim 10^{-7} \quad 32 \text{ bit}, \tag{1.7}$$

$$\epsilon_m = 2^{-52} \sim 10^{-16} \quad 64 \text{ bit}. \tag{1.8}$$

Più in generale, un numero reale x^* in rappresentazione finita è diverso dal vero numero x. Ad esempio π è diverso dalla sua rappresentazione approssimata π^*=3.14159. Possiamo associare a un numero un errore assoluto o relativo. Il primo è $e_{\text{abs}}=|x-x^*|$, e per un numero con n decimali $e_{\text{abs}}\leq 0.5\ 10^{-n}$; il secondo è $e_{\text{rel}}=e_{\text{abs}}/|x|\simeq e_{\text{abs}}/|x^*|$. L'errore si propaga, e potenzialmente si amplifica, combinando i numeri tramite operazioni. Ad esempio, se i numeri rappresentati sono $x^*=x+e_x$, $y^*=y+e_y$, con e i relativi errori, sommando o sottraendo si ha

$$x^* \pm y^* = x \pm y + e_x \pm e_y$$

e quindi l'errore sulla somma o differenza è

$$e = (x^* \pm y^*) - (x \pm y) = e_x \pm e_y.$$

Poichè $|e|=|e_x+e_y|\leq|e_x|+|e_y|$ si ottiene che

$$\max\{|e|\} = |e_x| + |e_y|,$$

cioè il massimo errore è la somma dei moduli dei singoli errori. Benchè questo sia il caso peggiore e siano possibili cancellazioni, è chiaro che in generale una somma o sottrazione propaga gli errori dei singoli addendi. Per una moltiplicazione o divisione si mostra facilmente la stessa cosa per gli errori relativi.

Un ulteriore errore è quello generato dalle operazione approssimate in macchina. In generale, la vera operazione $\otimes$ è fatta in modo algoritmicamente diverso da quella $\otimes^*$ fatta dal computer. La vera operazione con i veri numeri sarebbe rappresentata da $x \otimes y$, e quella approssimata sui numeri approssimati da $x^* \otimes^* y^*$. Si ha che

$$\begin{aligned} e_{\text{abs}} &= |x \otimes y - x^* \otimes^* y^*| \\ &= |x \otimes y - x^* \otimes y^* + x^* \otimes y^* - x^* \otimes^* y^*| \\ &\leq |x \otimes y - x^* \otimes y^*| + |x^* \otimes y^* - x^* \otimes^* y^*|. \end{aligned} \tag{1.9}$$

Nell'ultima linea si riconoscono l'errore propagato dalla vera operazione e quello generato dalla operazione approssimata sui numeri approssimati. Quindi vale come prima il limite superiore dell'errore dato dalla somma dei moduli di errore propagato e generato. Ad esempio, con una mantissa a tre cifre complessive

$$2.77 \times 10^2 + 7.56 \times 10^2 = [10.36 \times 10^2] = 1.04 \times 10^3.$$

L'errore generato è 0.004×10^3, e quello propagato il doppio dell'arrotondamento, ovvero 0.01×10^3. Quanto visto finora implica altre due proprietà dell'aritmetica a rappresentazione finita. La prima è che

non esiste inverso moltiplicativo.

Ad esempio, $1/a$=3.33×10^{-1} è l'inverso approssimato di a=3.00×10^0, ma non è il suo elemento inverso per l'operazione di moltiplicazione, dato che $a\cdot(1/a)$=$9.99\times10^{-1}\neq1$. La seconda proprietà è che

non vale la legge associativa dell'addizione.

Supponiamo che a=1 e b=c=$0.9\epsilon_m$. Chiaramente $(a+b)+c$=1, dato che, come visto in precedenza, sia b che c non arrivano a modificare a se gli sono sommati individualmente; per contro $a+(b+c)>a$, dato che $b+c$=$1.8\ \epsilon_m>\epsilon_m$ e quindi sommabile con successo ad a. Questo suggerisce di sommare tra loro addendi piccoli e poi sommare il risultato a numeri più grandi, piuttosto che sommarli singolarmente al numero grande.

Effettuando numerose operazioni, gli errori si propagano e si generano, componendosi in varie maniere. Come già detto, non è necessariamente detto che gli errori siano uguali ai loro limiti superiori. Se si assume che essi si accumulino in modo casuale, risulta che l'errore accumulato in N operazioni è

$$e = \epsilon_m\sqrt{N}, \quad \text{ovvero} \quad \log e = \frac{1}{2}\log N + C, \tag{1.10}$$

un aumento relativamente lento con il numero di operazioni. La seconda equazione è ottenuta prendendo il logaritmo dei due membri, che è equivalente a disegnare la funzione in scala log-log: si ottiene una relazione lineare tra i logaritmi, con pendenza pari all'esponente. Nel lavoro numerico, questo andamento tipico dei fenomeni stocastici (vedi Sez.6) è spesso verificato, ma non sempre: sono noti casi di errori che aumentano addirittura come $N!$, cioè sovraesponenzialmente.

1.2 *Errore algoritmico*

Si è visto che qualunque calcolo ha un errore associato crescente con il numero N di operazioni effettuate. D'altra parte, l'errore *algoritmico* commesso nel valutare una certa quantità diminuisce con N – se non lo fa, l'algoritmo non funziona e va cambiato. Ad esempio, supponiamo di voler calcolare numericamente l'integrale definito di una funzione di una variabile (analizzeremo il problema più in dettaglio nel Cap.**??**). Per determinare l'area limitata dalla funzione, un possibile approccio (Fig.1.1) è dividere l'intervallo di definizione (a,b) in N sotto-intervalli di ampiezza Δ=$(b–a)/N$, valutare la funzione f al centro x_i di ogni intervallo, e calcolare l'area come la somma delle aree di rettangolini di base Δ e altezza $f(x_i)$:

$$\int_a^b f(x)dx \simeq \sum_{i=1}^{N} f(x_i)\,\Delta.$$

Questo è la prima *discretizzazione* che incontriamo. È verosimile, e lo mostreremo in seguito, che l'integrale approssimi tanto meglio il

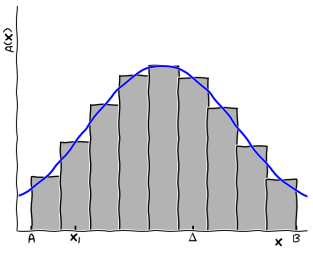


Figura 1.1: Schema dell'integrale numerico a istogrammi.

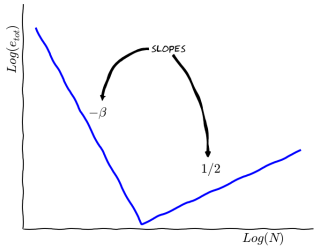


Figura 1.2: Errore algoritmico e di round-off.

valore vero quanto più i rettangolini approssimano la forma della funzione, cioè più sono stretti, cioè più N è grande. Supponiamo quindi che l'errore decresca come una potenza di N, ad esempio

$$e_{\text{algo}} \simeq \alpha / N^{\beta}.$$

L'errore totale è allora (ricordando, e assumendo che valga, Eq.1.10)

$$e_{\text{tot}} \simeq \frac{\alpha}{N^{\beta}} + \epsilon_m \sqrt{N}. \tag{1.11}$$

Come schematizzato in Fig.1.2, in scala log-log a bassi N l'errore algoritmico è grande perchè un campionamento rozzo non fornisce una stima buona dell'integrale; man mano che N cresce, l'errore totale cala, ed è dominato dalla discesa con pendenza $-\beta$ dell'errore algoritmico. Raggiunto un certo valore di N, che dipende dal problema e dall'algoritmo, l'errore algoritmico (che cala con N) diventa minore di quello di round-off (che sale con N). L'errore totale ha un minimo. A N abbastanza grandi, il round-off error diventa dominante. Concludiamo che c'è un valore ottimale di N che ci fornisce il minimo errore, possiamo valutarlo nel caso di errore algoritmico noto.

Ipotizziamo un errore algoritmico $e_{\text{algo}} \sim 1/N^2$. L'errore totale è $\sim 1/N^2 + \epsilon_m \sqrt{N}$. Per trovare il minimo annulliamo la derivata

$$\frac{de_{\text{tot}}}{dN} = -\frac{2}{N^3} + \frac{\epsilon_m}{2\sqrt{N}} = 0. \tag{1.12}$$

Riarrangiando,

$$N^{\frac{5}{2}} = \frac{4}{\epsilon_m}; \quad N_{\text{min}} = \left(\frac{4}{10^{-7}}\right)^{\frac{2}{5}} \simeq 1100.$$

L'errore totale commesso a N=N_{min} è

$$e_{\text{tot}} = \frac{1}{1100^2} + \epsilon_m \sqrt{1100} = 8\ 10^{-7} + 33\ 10^{-7} \sim 4\ 10^{-6}$$

da cui si vede che l'errore di round-off è il più grande dei due, e fa sì che l'errore totale sia dell'ordine di ben 40 volte la ϵ_m.

Per dimostrare l'importanza di un buon algoritmo –il quale, cioè, faccia scendere quanto possibile rapidamente l'errore– stimiamo le stesse grandezze per un errore algoritmico dell'ordine di $2/N^4$. Si ha

$$\frac{de_{\text{tot}}}{dN} = 0 = -\frac{8}{N^5} + \frac{\epsilon_m}{2\sqrt{N}}. \tag{1.13}$$

e quindi

$$N^{\frac{9}{2}} = \frac{16}{\epsilon_m}; \quad N_{\text{min}} = \left(\frac{16}{10^{-7}}\right)^{\frac{2}{9}} \simeq 67,$$

e un errore totale minimo

$$e_{\text{tot}} = 10^{-7} + 8\ 10^{-7} \sim 9\ 10^{-7}.$$

La migliore prestazione dell'algoritmo riduce perciò di un fattore 16 il numero delle operazioni e l'errore di un fattore 4.

Concludendo, notiamo che l'errore algoritmico può essere espresso in termini di un parametro grande proporzionale al numero di

operazioni, ad esempio il numero di divisioni di un intervallo come nel caso precedente, o al contrario in termini di un parametro piccolo, come ad esempio la dimensione $\Delta=(b–a)/N$ dell'intervallino, essenzialmente l'inverso di N. Ambedue questi tipi di parametri sono legati alla scelta fatta per la discretizzazione del problema.

1.3 *Derivate discretizzate*

Studiamo ora alcuni modi semplici per calcolare numericamente le derivate. Questo calcolo è potenzialmente rischioso in termini di accuratezza. L'integrazione, che discuteremo più oltre, è una operazione di media che tende a sopprimere gli effetti delle fluttuazioni locali della funzione in esame. Al contrario, la derivata misura ed evidenzia –anche drammaticamente– proprietà locali della funzione: la pendenza (derivata prima), la curvatura (derivata seconda) e così via. È difficile da calcolare sostanzialmente perchè, numericamente, è un rapporto tra quantità piccole (si veda la dettagliata discussione su Numerical Recipes, cap.5 [1]), come è evidente dalla definizione [1]

$$\frac{df}{dx}|_{x_0} = \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} \tag{1.14}$$

come rapporto incrementale destro o in avanti. (Qui assumiamo implicitamente che, come necessario per l'analiticità della derivata, esista anche il rapporto incrementale all'indietro

$$\frac{df}{dx}|_{x_0} = \lim_{h\to 0} \frac{f(x_0)-f(x_0-h)}{h} \tag{1.15}$$

e che i due siano uguali.)

Con tutto ciò, la discretizzazione delle derivate è inevitabile e fondamentale per la discretizzazione delle equazioni differenziali, oltre che più in generale per esprimere una approssimazione alla funzione nelle vicinanze di un punto del dominio (p.es. nell'integrazione numerica). In ogni caso, questo è un punto buono come un altro per familiarizzarci con la discretizzazione e la stima d'errore algoritmico.

Cominciamo con il discretizzare l'intervallo di definizione (o l'intervallo locale di nostro interesse) della funzione $f(x)$. Lo dividiamo in N sottointervalli di ampiezza

$$h = \frac{|b-a|}{N}, \tag{1.16}$$

e definiamo gli N+1 punti ("la griglia") dove valuteremo la funzione,

$$x_i = a + ih \quad i = 0, \ldots, N \tag{1.17}$$

come schematizzato in Fig.1.3. La funzione nei punti della griglia è indicata con f_i=$f(x_i)$. (Ovviamente si può scegliere di lavorare con N punti e quindi N–1 intervalli. Oppure l'intervallo potrebbe essere aperto a destra o a sinistra o ambedue. Quel che conta è che ogni punto è a distanza h dal punto adiacente.)

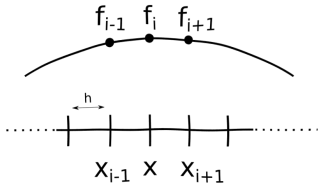


Figura 1.3: Schema di discretizzazione in 1D.

Lo sviluppo di f in serie di potenze vicino a un dato punto x_0 è

$$\begin{aligned} f(x) = f(x_0) &+ f'(x_0)(x-x_0) \\ &+ \frac{f''(x_0)}{2}(x-x_0)^2 \\ &+ \frac{f'''(x_0)}{6}(x-x_0)^3 + \dots \end{aligned} \tag{1.18}$$

La versione discretizzata si ottiene dalle corrispondenze $x \to x_{i\pm1}$, $x_0 \to x_i$, $f(x_i) \to f_i$, $f'(x_i) \to f'_i$, eccetera. Per costruzione, la minima distanza tra i punti della griglia è h; quindi ad esempio $x_{i+1} - x_i = h$ e $x_{i-1} - x_i = -h$. La funzione in un punto x_{i+1} adiacente a destra al punto x_i sarà allora (vedi Fig.1.3)

$$f_{i+1} = f_i + f'_i h + f''_i h^2/2 + f'''_i h^3/6 + o(h^4). \tag{1.19}$$

Con $o(h^4)$ indichiamo il fatto che l'errore che commettiamo omettendo ulteriori termini è dell'ordine di h^4, dato che il primo termine che trascuriamo è quello del quarto ordine. L'analogo di Eq.1.19 quando ci muoviamo verso sinistra sull'asse è

$$f_{i-1} = f_i - f'_i h + f''_i h^2/2 - f'''_i h^3/6 + o(h^4), \tag{1.20}$$

l'unica differenza essendo che i termini dispari sono negativi, perchè l'intervallino di cui ci si sposta è negativo ("verso $-\infty$") e pari a $-h$.

In queste espressioni conosciamo solamente le f_i e $f_{i\pm1}$, ma possiamo usarle per esprimere in modo approssimato le derivate, con diversa accuratezza a seconda del numero di termini che consideriamo. Supponiamo di trascurare tutti i termini a partire da quello di ordine h^2 incluso. In questo caso Eq.1.19 diventa

$$f_{i+1} = f_i + f'_i h + o(h^2) \tag{1.21}$$

da cui riarrangiando

$$f'_i = \frac{f_{i+1} - f_i}{h} + o(h), \tag{1.22}$$

dove l'errore, ovviamente, è $o(h)$ in quanto abbiamo diviso per h. Questa espressione della derivata è nota come *forward difference formula*, è equivalente al rapporto incrementale (Eq.1.14), ed ha un errore associato *lineare* in h. Come si capirà meglio in seguito, questo errore è piuttosto importante. Da Eq.1.20, si ottiene analogamente

$$f'_i = \frac{f_i - f_{i-1}}{h} + o(h), \tag{1.23}$$

la cosidetta *backward-difference formula*.

Si può fare molto meglio di così. Le due formule che faranno da cavallo da tiro nel seguito si ottengono usando le Eq.1.19 e 1.20 come stanno. Sottraendo Eq.1.20 da Eq.1.19, i termini pari (di ordine 0, 2, etc.) si cancellano, e rimangono solo quelli dispari:

$$f_{i+1} - f_{i-1} = 2h f'_i + \frac{h^3}{3} f'''_i. \tag{1.24}$$

Riarrangiando,

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} + o(h^2). \tag{1.25}$$

Questa formula, detta *centered-difference*, ha un errore *quadratico* in h, il che la rende molto più robusta e accurata di quelle forward e backward viste prima (semplicemente perchè se h=0.1, l'errore associato alla derivata nel punto i sarà 0.1 c per la formula forward, e 0.01 c per quella centrata, con c una costante).

Se ora sommiamo Eq.1.19 e 1.20, sono i termini dispari a cancellarsi, e otteniamo

$$f_{i+1} - f_{i-1} = 2f_i + h^2 f_i'' + \frac{h^4}{4!} f_i^{IV}, \tag{1.26}$$

da cui riarrangiando si ottiene una espressione per la derivata seconda,

$$f_i'' = \frac{f_{i+1} + f_{i-1} - 2f_i}{h^2} + o(h^2) \tag{1.27}$$

con errore quadratico in h. Questa formula è un altro cavallo da tiro nel contesto delle equazioni differenziali. È chiaro che conviene usare le Eq.1.25 e 1.27 con il loro errore quadratico, che permette di usare h non troppo piccoli e di evitare così la divisione tra due numeri troppo piccoli nel calcolo della derivata e la conseguente inaccuratezza.

1.4 *Integrali: trapezi, Simpson, e gli altri*

Analizziamo ora in maggior dettaglio un paio di tecniche per l'integrazione numerica. Maggiori dettagli in NR, Cap.4 [2]. Parliamo quasi solo di integrazione unidimensionale: l'applicazione delle tecniche che consideriamo a dimensionalità maggiori è possibile, anche se non sempre ovvia, ma diventa rapidamente molto costosa. Accenneremo più in dettaglio in seguito (Sez.6.3) ad alcune tecniche tecniche alternative.

[2]

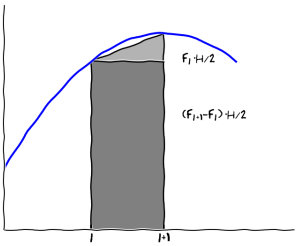


Figura 1.4: Schema di integrazione trapezoidale.

Consideriamo l'integrale unidimensionale

$$I = \int_a^b f(x)\, dx, \tag{1.28}$$

e discretizziamo l'intervallo di integrazione dividendolo come prima in N sottointervalli di ampiezza $h = |b-a|/N$, e definendo gli N+1 punti dove valuteremo la funzione come

$$x_i = a + ih \quad i = 0, \ldots, N. \tag{1.29}$$

Le due formule che esamineremo sono quella trapezoidale e quella di Simpson. Questi sono casi particolari della famiglia di formule di Newton-Cotes, che si ottengono approssimando la funzione in ogni intervallino con un polinomio. Trapezi e Simpson si ottengono rispettivamente con una approssimazione lineare e una quadratica. Supponiamo dunque di descrivere la funzione nei dintorni di x_i con una retta,

$$f = f_i + f'x = f_i + \frac{(f_{i+1} - f_i)}{h} x, \tag{1.30}$$

con errore di troncamento $o(h^2)$. L'integrale di questa funzione sull'intervallino h tra x_i e x_{i+1} è

$$I_i = (f_i + f_{i+1})h/2, \tag{1.31}$$

come si vede valutando le aree della Fig.1.4. L'errore sull'intervallino è ovviamente $\delta I_i \simeq f^* h \sim h^3$. L'integrale totale è la somma dei singoli integralini:

$$\begin{aligned} I = \sum_i I_i &= \frac{h}{2}[(f_1 + f_2) + \ldots (f_N + f_{N+1})] \\ &= \frac{h}{2}(f_1 + f_{N+1}) + h\sum_{i=2}^{N} f_i. \end{aligned} \tag{1.32}$$

L'errore associato all'integrale complessivo è N volte quello su un intervallino,

$$\delta I \simeq N * h^3 \simeq \frac{1}{h}h^3 \simeq h^2 \simeq \frac{1}{N^2}, \tag{1.33}$$

che decresce come una potenza di N come avevamo ipotizzato nella discussione in Sez.1.2.

Il caso di Simpson è analogo. Si approssima la funzione come quadratica nell'intervallino,

$$f = f_i + f_i' x + f_i'' x^2/2$$

e si utilizzano le espressioni delle derivate prima e seconda discretizzate ottenute in precedenza. Integrando su un intervallino doppio da $-h$ and h centrato nel punto x_i si ha

$$\begin{aligned} \int_{-h}^{+h} f dx &= f_i x|_{-h}^{+h} + (\frac{f_{i+1} + f_{i-1} - 2f_i}{2h^2})\frac{x^3}{3}|_{-h}^{+h} \\ &= 2hf_i + \frac{h}{3}(f_{i+i} + f_{i-1} - 2f_i) \\ &= \frac{h}{3}(f_{i+i} + f_{i-1} + 4f_i), \end{aligned} \tag{1.34}$$

dove il termine in f' cade in quanto dispari. Dunque su un intervallino doppio l'integrale è

$$\int_{x_1}^{x_3} = \frac{h}{3}(f_1 + 4f_2 + f_3), \tag{1.35}$$

ed estendendo la somma a tutti gli intervalli

$$\begin{aligned} \int_{x_1}^{x_{N+1}} f(x)dx = \frac{h}{3}(f_1 &+ 4f_2 + 2f_3 + \ldots \\ &+ 2f_{N-1} + 4f_N + f_{N+1}), \end{aligned} \tag{1.36}$$

cioè peso 1 per il primo punto, poi pesi alternati 4 e 2, e di nuovo 1 per l'ultimo. Chiaramente, il numero di punti dev'essere dispari per applicare questa formula. L'errore dell'integrale sull'intervallo è $\delta I_i \simeq h^* o(h^4) = h^5$, perchè i termini dispari (quindi anche quello cubico) non danno contributo su questo intervallino simmetrico. L'errore totale (sugli N intervalli) è perciò, analogamente a prima,

$\delta I \simeq N * h^5 \simeq h^4 \simeq 1/N^4$ – una caduta molto più rapida di quella del metodo trapezoidale.

In Fig.1.5 è rappresentato l'errore totale in scala log-log per i due metodi discussi. Si nota la caduta di tipo potenza uguale a quella predetta per l'errore algoritmico a basso N, e l'aumento $\propto N^{1/2}$ a grandi N. Anche nelle formule di Newton-Cotes generali (in cui il polinomio approssimante è di grado più alto di 2 e di conseguenza molto più complicate) l'errore totale è dell'ordine Nh^k, con k il primo dei termini troncati della serie di potenze che rappresenta la funzione nell'intervallino (salvo cancellazioni sistematiche come nel caso di Simpson). Siccome $h \sim 1/N$, l'errore è anche $\simeq 1/N^{k-1}$, cioè un ordine meno che in h. In ogni caso, l'errore cala con N come una potenza in tutti i casi.

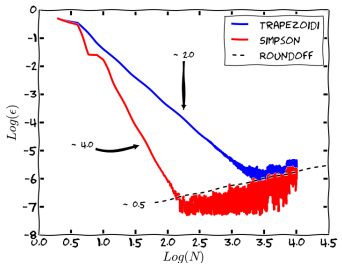


Figura 1.5: Errore totale vs N in scala log-log per trapezi e Simpson. Notare la caduta di tipo potenza uguale a quella predetta per l'errore algoritmico a basso

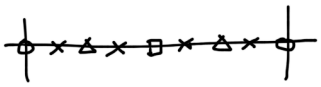


Figura 1.6: Schema di iterazione della regola trapezoidale: i primi punti sono i cerchi (estremi dell'intervallo); il successivo quello al centro dell'intervallo (quadrato); poi quelli al centro dei semi-intervalli destro e sinistro (triangoli), poi quelli al centro degli ulteriori sottointervallini, e così via.

La maniera naïf di monitorare la convergenza dell'integrale con N è ripetere il calcolo per N crescente, ma questo è ovviamente uno spreco. In alternativa si può invece chiamare un sottoprogramma che calcoli l'integrale in punti aggiuntivi rispetto a quelli già visitati. Per il metodo trapezoidale la cosa è relativamente semplice: si dividono successivamente in due gli intervallini dell'iterazione precedente (Fig.1.6); ad ogni suddivisione aggiuntiva si verifica l'andamento dell'integrale e si arresta l'iterazione sotto un dato errore relativo desiderato (vedi NR, cap.4 [3]).

Di passaggio, segnaliamo infine l'integrazione alla Romberg. Dato che I converge al suo valore esatto per grande N, o piccolo h, una volta raggiunta una decente convergenza, si calcola una funzione che interpoli gli ultimi valori calcolati in funzione di h, e la si estrapola ad $h \to 0$ (Fig.1.7). **QUI** info su romberg
`https://en.wikipedia.org/wiki/Romberg's_method`
`https://www.mathstat.dal.ca/~tkolokol/classes/2400/romberg.pdf`
, menzione notebook; Gaussian integration?

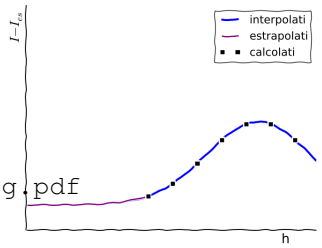


Figura 1.7: Schema di iterazione di Romberg. I converge al suo valore esatto per grande N, o piccolo h; interpolando gli ultimi valori calcolati, si estrapola la funzione interpolante ad $h \to 0$.

Notiamo infine che anche gli integrali impropri sono abbordabili con le tecniche viste o loro parenti stretti. Naturalmente si parla di integrali con a) integrando indefinito ai limiti d'integrazione, b) integrale su dominio infinito, c) integrando con singolarità integrabile al bordo, d) integrando con singolarità nota nel dominio. Integrali tipo

$$\int_0^{+\infty} \frac{dx}{x} \quad \text{o} \quad \int_{-\infty}^{+\infty} \cos x \, dx$$

non sono impropri: sono impossibili.

1. B - Introduzione al Python

In questa sezione vedremo le basi della programmazione in Python, molte delle quali, essendo similari ad altri linguaggi di programmazione, risulteranno utili in generale. Non è nostra intenzione essere esaustivi, ma soltanto dare una infarinatura delle caratteristiche di base della programmazione e di tutto ciò che ci sarà utile per la implementazione dei codici che useremo per la risolzione dei problemi di fisica. Sarà utile sia a chi è del tutto digiuno di programmazione, sia a chi sa già programmare e vuole imparare il Python. Inoltre alcuni dettagli sui vari aspetti della programmazione verranno ripetuti e ampliati nei successivi capitoli pratici dedicati all'implementazione di problemi. Una costante di questi capitoli sarà la presenza di codici e la loro spiegazione, ma non verrà quasi mai riportato l'output che questi codici restituiscono. Questo per stimolare il lettore a pensare quale sia l'output e confrontarlo con quello reale, provando direttamente sul proprio computer tutti i codici e sperimentando in autonomia.

Per chi vuole andare oltre si vedano le referenze sulla programmazione in generale e sul Python in particolare.

1.1 Installazione

Iniziamo il capitolo con alcune indicazioni su quali sono i software da installare per poter eseguire il codice presentato nelle sezioni pratiche che seguiranno.

Presenteremo principalmente due possibili strade da seguire per poter avere un ambiente python adatto all'esecuzione del codice presentato nelle sezioni pratiche: l'installazione di alcuni pacchetti in una distribuzione linux già installata e l'uso di una macchina virtuale linux.

- **Installazione di pacchetti su linux:** per chi possiede un pc con un linux installato e funzionante, è necessario soltanto cercare e installare i seguenti pacchetti, mediante il gestore del software

della propria distribuzione:
- python 2.7: quasi certamente già presente nelle recenti distro linux,
- numpy: libreria python per la gestione efficiente degli array,
- matplotlib: libreria python per la creazione di grafici di elevata qualità,
- ipython: shell python avanzata.

Le versioni disponibili attualmente per le due librerie sono 1.8.0 e 1.3.1, rispettivamente. La versione di Ipython è invece, attualmente, la 2.1.0. Versioni precedenti o successive di questi software non comportano sostanziali differenze del codice presentato.

[1] Se cosi fosse segnalatecelo ;)

[1]

Solitamente nelle distribuzioni linux, il software è spezzettato in più pacchetti, ma la gestione delle dipendenze del gestore pacchetti dovrebbe installare tutto ciò che serve senza doversi preoccupare troppo.

- **Macchina virtuale:** questo metodo è vivamente consigliato per gli utenti windows e mac. Si tratta di installare sul proprio computer un software di virtualizzazione e una macchina virtuale linux pronta all'uso, cioè con tutti pacchetti necessari (quelli visti precedentemente) già installati. I seguenti passaggi mostrano come procedere:
  - installare il programma di virtualizzazione free chiamato Virtualbox. http://www.virtualbox.org
  - scaricare la macchina virtuale (1GB) da questo indirizzo: http://www.
  - cliccare due volte sul file con estenzione (.ova) appena scaricato per installare la macchina virtuale,
  - si consiglia di modificate la ram da dedicare alla macchina dalle impostazioni (che trovate nella finestra principale di Virtualbox), portandola a 1GB.
  - avviare la macchina dalla finestra principale di Virtualbox,
  - all'interno della finestra avete la vostra macchina linux con tutto ciò che serve.
  - user e password per effettuare il login sono uguali a nugrid.

Note: avete la possibilità di mettere a tutto schermo nel menu visualizza; se avete la connessione ad internet attiva su win dovrebbe funzionare automaticamente anche all'interno della macchina virtuale linux; per spegnere cliccare sulla x della finestra e scegliere "salva stato": in questo modo al riavvio avete tutto come lo avete lasciato (molto comodo).

È doveroso dire che questi due metodi presentati non sono gli unici. Python e i software che useremo sono ormai molto diffusi e per questo non esclusivi del mondo linux infatti esistono anche pacchetti autoinstallanti sia per windows che per mac. Percui per gli utenti di questi sistemi è possibile anche l'utilizzo di questi pacchetti, ma vi rimandiamo alle numerose guide presenti nel web e alla pagina www.scipy.org.

1.2 *Shell interativa*

Il primo approccio al Python avviene aprendo la sua shell interativa, digitando nella riga di comando python in una shell linux. Ciò che si ottiene sarà questo:

```
Python 2.7.5 (default, May 29 2013, 02:28:51)
```

```
  [GCC 4.8.0] on linux2
  Type "help", "copyright", "credits" or "license" for
more information.
  >>>
```

La shell interattiva aspetta dei comandi che una volta inseriti, calcando il tasto invio, verranno eseguiti. Il modo più semplice per provare é fare qualche semplice calcolo:

```
  >>> 2 + 2
  4
```

Come si vede la shell legge i comandi che seguono le freccette `>>>`, esegue l'operazione chiesta e ci risponde con il risultato corretto, senza caratteri ad inizio riga.

È ovvio che dovremo inserire dei comandi che Python conosce, altrimenti otterremo un errore:

```
  >>> pippo
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError:  name 'pippò is not defined
```

Il comando `pippo` non è infatti riconosciuto, perché non definito da nessuna parte.

Tutto il codice che illustreremo in questa e in tutte le altre sezione del libro, lo si potrà inserire riga per riga direttamente della shell oppure lo si potrà scrivere in un file di testo, con il vostro editor preferito, ed eseguirlo in un secondo momento. Per eseguire un programma Python, meglio se salvato con estensione .py, dovremo semplicemente digitare nella shell linux `python file.py`. L'interprete leggerà riga per riga ed eseguirà il vostro programma, dandovi un output o meno a seconda di cosa avete scritto nel codice o un errore se c'é qualcosa che non va.

Tutto ciò che viene scritto nella shell o in un file di testo viene interpretato da Python. È utile però inserire nel codice dei commenti per spiegare cosa si sta facendo, sia a se stessi che a terzi che vogliono modificare il nostro codice. Per far si che Python non legga i nostri commenti, causando degli errori, possiamo postporre ad essi il carattere #, come nella seguente riga d'esempio:

```
  >>> 2+2 #usiamo Python per fare una addizione
  4
```

Il risultato sarà identico al precedente perché l'interprete non terrà in considerazione il commento che segue il cancelletto #.

Per concludere citiamo una shell interattiva molto usata: IPython. [2] Questa shell interativa avanzata ha numerose funzionalità utili come per esempio: un sistema di completamento automatico avanzato, un ampia gestione della storia dei comandi eseguiti, una serie di funzioni aggiuntive come per esempio quelle di aiuto in linea, la possibilità di eseguire un codice contenuto in un file all'interno della shell stessa, anche in modalità di debugging; la possibilità di modificare un codice, mediante un editor, eseguire comandi di sistema e salvarne in una variabile python l'output, e molto altro. Come vedremo, insieme con i pacchetti Numpy e Matplotlib, saremo in grado di ricreare un

[2] Per avere un idea delle potenzialità di IPython si veda http://www.ipython.org

ambiente di calcolo completo come il famoso Matlab.

1.3 Variabili

Lo strumento principale di un qualunque linguaggio di programmazione sono le variabili. Queste ci permettono di salvare in memoria dei dati, per esempio numeri, assegnandogli un nome di nostro comodo. Con esse possiamo compiere delle operazioni di vario tipo e modificarle quando ne abbiamo bisogno. Hanno dunque un significato del tutto simile a quello delle variabili in algebra. Vediamo subito come assegnare un valore numerico ad una variabile:

```
x=2
```

In questo modo abbiamo detto salva il valore 2 in memoria e chiamalo x. Ora potremo usare la variabile x per esempio per fare operazioni aritmetiche, per esempio:

```
x+x
Out:  4
```

La variabile x può essere perciò letta, usata e anche modificata nel programma ogni volta che ne abbiamo bisogno:

```
x=2
x+x
Out:  4
x=3
x*x
Out:  9
```

Il nome con cui definiamo una variabile può essere piu lungo di un carattere, può contenere maiuscole e "_", ma non spazi vuoti. È buona norma chiamare le variabili con un nome che sia descrittivo del dato che contengono, per esempio `g=9.81`, oppure `accel_gravit=9.81`, se vogliamo registrare l'accelerazione di gravità. Alcuni nomi non si possono usare per assegnare una variabile perché sono già usati da alcuni comandi del Python, per esempio `for`.

I tipi di dati che vogliamo registrare in una variabile possono essere molteplici. Il modo di assegnare un dato ad una variabile prescinde dal suo tipo, e sarà sempre uguale all'esempio precedente. A differenza di altri linguaggi, Python riconosce il tipo di dato inserito dentro una variabile ed assegna in maniera automatica il tipo corretto alla variabile.

I tre tipi di variabile di cui faremo maggiormente uso sono:

- interi: valori interi (o naturali) sia positivi che negativi
- float: valori a virgola mobile (floating-point) che rappresentano i numeri reali, sino ad un certo numero di cifre dopo la virgola. A differenza degli interi, possono rappresentare anche numeri interi (come 1.0) ovviamente consumando piu memoria.
- complex: valori complessi con parte reale e immaginaria, come per esempio `2 + 3j`, dove l'unità immaginaria è espressa dalla lettera j, e non la i usata solitamente in matematica.

Perché si ha bisogno di diversi tipi? Sia per occupare meno memoria, importantissimo sia in passato che tuttora, sia per diminuire il tempo di calcolo. Inoltre salvare una dato intero in una variabile intera permette di mantenere la sua precisione, al contrario di quanto si avrebbe salvandolo in una variabile float.

Di seguito vediamo come assegnare i tre tipi di variabile:

```
k = 2
x = 1.5
y = 3.2E-03
z = 2 + 3j
```

È abbastanza semplice capire che la variabile `k` sarà definita come intero, la `x` come float e la `z` come complex, in cui il 2 rappresenta la parte reale e 3j la parte immaginaria (la j sostituisce la i usata in fisica e segue senza spazi il numero). La variabile `y` sarà sempre un float ma può essere assegnato usando la notazione scientifica.

Notiamo inoltre come nella assegnazioni precedenti abbiamo usato uno spazio prima e dopo l'uguale. Questo serve solo per aumentare la leggibilà del codice, ma non altera il funzionamento. L'aggiunta di uno o più spazi non và fatta, però, all'inizio della riga, perche il Python interpreta questi spazi come un indentazione, come vedremo più avanti.

Un altro tipo di variabile, meno utile per i nostri scopi, ma comunque interessante, è il tipo stringa. Una variabile di tipo stringa contiene dei caratteri, per esempio:

```
x = "Questa è una stringà'
```

L'assegnazione di una stringa avviene racchiudendo del testo tra virgolette "..." o apici '...'.

1.4 Output e Input

Il principale comando per stampare a video il valore delle variabili del nostro programma e delle stringhe di testo è il comando `print`, per esempio:

```
x=1
print(x)
```

Come si osserva l'output del comando `print` è il valore della variabile x e non la stringa "x". Se volessimo stampare sia il valore delle variabili che del testo dovremo separarli con delle virgole come nel seguente esempio:

```
x=1
y=2.3
z=4+5j
print("La x vale:", x, "e la y vale:",y,)
print("La z complessa vale:",z)
Out:  La x vale 1 e la y vale 2
Out:  La z complessa vale (4+5j)
```

Come si vede tutti i tipi di variabile sono accettati dalla funzione print. È inoltre possibile scegliere come separare i diversi elementi

con qualcosa di diverso dal semplice spazio (usato di default, come in precedenza), nel seguente modo:

```
print(x,y,sep=" ...  ")
Out:  1 ...  2.3
```

Adesso invece vediamo come è possibile fornire un dato da memorizzare in una variabile in fase di esecuzione del programma, chiedendo all'utente di inserirlo. Questa è la forma standard:

```
x=input("Inserire il valore della x:  ")
```

Il programma quando incontra questa riga stampa a video la stringa che abbiamo inserito dentro la funzione input e aspetta che l'utente inserisca un comando interpretabile dal Python. Quindi può essere un semplice numero in qualunque formato (o una stringa racchiusa tra virgolette "...") o anche una serie di valori separati da virgola. [3] Dopo aver inserito il dato e premuto di invio il programma salva il dato nella variabile `x`, che sarà del tipo corretto, e potrà essere usata nelle righe di codice successive.

[3] In Python si chiamano liste. Vedremo in seguito questo tipo di variabile multivalore

1.5 Operazioni aritmetiche

Come abbiamo accennato in precedenza, possiamo usare le variabili per fare delle operazioni sui dati che esse contengono. Le operazioni matematiche che Python riconosce di default sono le seguenti:

- `x+y` addizione
- `x-y` sottrazione
- `x*y` moltiplicazione
- `x/y` divisione
- `x**y` elevamento a potenza
- `x//y` divisione intera, restituisce il numero intero più vicino al risultato della divisione di `x` per `y`
- `x%y` modulo, restituisce il resto della divisione di `x` per `y`. Utile, per esempio, per sapere se un numero è divisibile per un altro: per esempio `n%2` darà zero se n è pari e uno se dispari.

In generale, se facciamo una operazione tra variabili dello stesso tipo e salviamo il risultato in una altra variabile, il tipo di questa sarà dello stesso tipo delle altre. Se invece i tipi delle variabili su cui operiamo sono differenti, la variabile su cui salviamo il risultato sarà del tipo più generale. È necessario riporre un pò di attenzione con la divisione, perché dividendo due interi si ottiene sempre un intero anche se il divisore non è multiplo intero del dividendo. Si ottiene infatti come risultato l'intero piu vicino al risultato reale. Se si vuole ottenere un float è necessario dunque far si che almeno uno tra divisore e dividendo sia float, anche temporaneamente, come nel seguente esempio:

```
x=3
y=2
z=x/y
print(z)
1
type(z)
```

```
Out:  <type 'int'>
z=x/float(y)
print(z)
Out:  1.5
type(z)
Out:  <type 'float'>
```

Come vediamo, si può trasformare il tipo di una variabile da intero a float mediante la funzione `float()`, ottenendo in questo modo una variabile z di tipo `float()`, come verificato usando la funzione `type()`, che restituisce il tipo di una variabile.

Di seguito riportiamo alcune espressioni che mostrano come si possano unire tra loro le operazioni in Python, in maniera del tutto simile a quanto succede in algebra.

Python	Algebra
`x+2*y`	$x+2y$
`x-y/2`	$x-\frac{1}{2}y$
`3*x**2`	$3x^2$
`x/2*y`	$\frac{1}{2}xy$
`(x+y)/2`	$\frac{x+y}{2}$
`a+2*b-0.5*(2.3**2+3/7.)`	$a+2b-\frac{1}{2}(2.3^2+\frac{3}{7})$

In generale, le moltiplicazioni e le divisioni vengono eseguite prima delle addizioni e sottrazioni. Le potenze vengono calcolate prima di tutte le altre. Si possono inoltre usare le parentesi per separare le operazioni allo stesso modo con cui si usano in algebra.

Alcune differenze con l'algebra ci sono. Per esempio, non potremo scrivere `2*x=y`, perché non significa nulla per Python, non essendo una assegnazione di variabile classica, non verrà perciò converita in `x=y/2` cosi come si può fare nella comune algebra. Un altro caso è quello di `x = x + 1`. Quì invece, mentre il Python assegna alla variabile x il suo precendente valore incrementato di 1, per l'algebra questa equazione non ha senso. Quindi ci si deve sempre ricordare che in Python (come nella stragrande maggiornza dei linguaggi) si possono fare delle assegnazioni ma non scrivere delle equazioni aspettandosi che vengano risolte in automatico, semplicemente perché si possono fare solamente delle assegnazioni di variabile. Un caso del tutto analogo è quello di $x = x^2 - 2$, che in algebra si risolve ottenendo due soluzioni, mentre in Python sarebbe come $x = 0^2 - 2 = -1$ se il valore iniziale della x è uguale a zero.

Di seguito mostriamo dei trucchi per modificare il valore delle variabili in maniera rapida:

- `x +=1` , incrementa di uno la variabile x
- `x -=4` , sottrae 4 al valore precendete di x
- `x *=-2.4` , moltiplica la x per -2.4
- `x /=5*y` , divide la variabile x per 5 moltiplicato y
- `x //= 3.4` , divide la x per 3.4 e arrotonda all'intero più vicino

Da ricordare che per poter modificare l'attuale valore della variabile, questa deve essere stata assegnata in precedenza, altrimenti si riceve un errore.

Possiamo assegnare il valore delle variabili anche nel modo seguente[4]:

```
x , y = 2.4 , 4
```

[4] questo metodo può essere usato anche insime alla funzione `input` per ricevere e memorizzare più di un valore contemporaneamente

che equivale alle seguenti due righe di codice

```
x=2.4
y=4
```

Si possono anche scrivere delle cose più sofisticate, come:

```
x , y = 2*z +1, (x+y)/3
```

e inoltre, sempre con lo stesso metodo, scambiare il valore di due variabili, cosa che in altri linguaggi non è cosi immediato:

```
x, y = y, x
```

Se non fosse possibile questo costrutto, come capita in altri linguaggi, si sarebbe dovuto procedere in questo modo:

```
temp=x
x=y
y=temp
```

Si deve salvare il valore della variabile x in un altra variabile ausialiaria `temp`, poi scrivere il valore di y dentro la x, ed infine scrivere il valore originale di x, che ora si trova in `temp`, nella variabile y.

1.6 Un primo esempio

Vediamo di seguito un piccolo programma che calcola l'altezza di una palla che cade ad un certo tempo, che ci permette di mettere in pratica tutte le cose viste sinora.

```
h = float(input("Enter the height of the tower: "))
t = float(input("Enter the time interval: "))
s = 9.81*t**2/2
print("The height of the ball is",h-s,"meters")
```

In questo codice chiediamo inizialmente all'utente di immettere i valori dell'altezza di partenza della palla e l'istante temporale in cui vogliamo sapere l'altezza della palla. Mediante la funione input() salviamo l'input dell'utente e lo salviamo nelle variabili h e t. Prima di salvare i valori inseriti li convertiamo in float mediante la funzione float(), in un'unica riga. A questo punto facciamo il calcolo mediante la formula del moto uniformemente accelerato e stampiamo a video il risultato.

Esercizi ulteriori da proporre.

1.7 Pacchetti e Funzioni aggiuntive

Per aggiungere funzioni che compiono operazioni aggiuntive a quelle di base in Python si usano delle librerie separate che possono essere caricate all'interno del nostro programma. Queste si trovano all'interno di pacchetti che possono essere anche molto complessi tanto da contenere una struttura ad albero composta da piu moduli, ognuno dei quali può avere uno o piu sottomoduli contenenti infine le funzioni che vengono chiamate dall'utente. L'esempio piu utile ai nostri scopi è quello del pacchetto math che contiene molte delle

funzioni matematiche piu usate come i logaritmi, radice quadrata, esponenziale, funzioni trigonometriche etc.

Vediamo di seguito come importare queste funzioni:

```
from math import log
x = log(2.5)
```

Nella prima riga tramite `from` specifichiamo il pacchetto da quale importiamo mendiante `import` la sola funzione log.

È possibile importare piu funzioni separandole con la virgola:

```
from math import log, sin
```

Oppure importarle tutte usando * dopo `import`.

Vediamo di seguito un altro piccolo programma che utilizza le funzioni trigonometriche per convertire le coordinate polari in cartesiane:

```
1 from math import sin,cos,pi
2 r = float(input("Enter r: "))
3 d = float(input("Enter theta in degrees: "))
4 theta = d*pi/180
5 x = r*cos(theta)
6 y = r*sin(theta)
7 print("x =",x," y =",y)
```

Per prima cosa importiamo dal pacchetto math le funzioni che ci servono, `sin` e `cos`, ma anche la costante `pi`. Successivamente, chiediamo all'utente di inserire le coordinate polari del punto da lui scelto, quindi il valore del raggio e dell'angolo, che salviamo nelle variabili apposite. Poiché le funzioni trigonometriche importate vogliono in ingresso un angolo espresso in radianti, mentre noi chiediamo all'utente di inserirli espressi in gradi, dovremmo fare la conversione (riga 4) prima di calcolare le due coordinate `x` e `y` (righe 5 e 6).

perche `sin` e `cos` vogliono in input angoli espressi in radianti e non in gradi.

1.8 Il costrutto IF

È spesso utile costruire un programma in modo che questo esegua operazioni differenti a seconda del valore di una o più variabili. Questo può fare mediante il costrutto IF, del tutto simile ad altri linguaggi, se non per la sintassi che viene semplificata. Vediamo subito con un esempio:

```
x = input(Inserisci un numero tra 0 e 10)
if x>5:
  print(il numero \'e maggiore di 5)
elif x<5:
  print(il numero \'e minore di 5)
else:
  print(il numero \'e uguale di 5)
```

In questo codice chiediamo all'utente di inserire un numero compreso nell'intervallo $[0, 10]$ e gli diciamo se è maggiore, minore o uguale a 5.

Nella riga 2 chiediamo di contralle se il valore di x sia maggiore di 5, se fosse vero allora vengono eseguite tutte le righe "indentate", cioé

quelle che hanno degli spazi all'inizio della riga, sino alla successiva riga non indentata, che nel nostro caso è il successivo cotrollo dato dalla istruzione elif.

Se non fosse vera allora viene fatto il controllo presente nella istruzione elif successiva, che nel nostro caso è `x<5`, e di nuovo vengono eseguite tutte le righe indentate successive (nel nostro caso solo quella contenente il comando `print`) solo se la verificha da esito positivo, altrimenti le righe indentate vengono ignorate e si passa alla successiva riga non indentata, che nel nostro caso è l'istruzione else. Quest'ultima serve se si vuole eseguire qualche operazione nel caso in cui tutte le condizioni precedenti fossero false, nel nostro caso se x fosse uguale a 5.

Facciamo notare che le istruzioni `elif` e `else` sono del tutto opzionali: dipende dal tipo di controlli e relative operazioni che vogliamo effettuare.

Riportiamo di seguito tutti le possili condizioni che si possono verificare:

Python	Condizione
`if x==1:`	controlla se $x = 1$
`if x>1:`	controlla se $x > 1$
`if x>=1:`	controlla se $x \geq 1$
`if x<1:`	controlla se $x < 1$
`if<=1:`	controlla se $x \leq 1$
`if x!=1:`	controlla se $x \neq 1$

È inoltre possibile unire due condizioni mediante le istruzioni logiche `and` e `or`:

Python	condizione
`if x>=0 and x<10:`	controlla se $x \in [0, 10)$
`if x<-10 or x>10:`	controlla se $x \in (\infty, -10) \cup (10, \infty)$

1.9 Il costrutto WHILE

Un altro utile contrutto per controllare il flusso del nostro programma é il ciclo `while`. Esso permette di eseguire ripetutamente tutte le righe contenute al suo interno, cioé indentate, sino a quando la condizione di controllo è verificata. Vediamo subito un esempio:

```
x = input(Inserisci un numero minore di 10)
while x>10:
  print(Hai inserito un numero maggiore di 10! Riprova.)
  x = input(Inserisci un numero minore di 10)
print(Ok, il numero inserito \'e minore di 10)
```

In questo programma viene chiesto all'utente di inserire un numero minore di 10 sinché questa condizione non viene verificata. Il ciclo while controlla la condizione, se questa è vero allora esegue le righe indentate successive e ricomincia sintanto che la x resta minore di

10. Una volta che si inserisce un numero minore di 10, il ciclo while salta le righe indentate e viene eseguita la prima riga non indentata successiva, nel nostro caso il `print` finale.

È possibile inoltre inserire delle altre condizioni all'interno del ciclo `while` che se soddisfatte permettano di uscire dal ciclo stesso. Questa utile possibilità la si realizza mediante l'istruzione break posta dentro un costrutto `if` opportunamente scritto. Per esempio, modifichiamo il precedente codice come segue:

```
while x>10:
  print(Il  numero \'e maggiore di 10! Riprova.)
  x = input(Inserisci un numero minore di 10)
  if x==0:
    break
```

In questo caso usciremo dal ciclo se il valore scelto per la x è minore di 10, oppure anche se la x assume il valore zero. È da precisare che l'istruzione break fa uscire dal ciclo in quel preciso punto, perciò tutte le righe indentate successive verranno saltate. Bisogna prestare attenzione pertanto a dove inserire il break.

1.10 Sequenza di Fibonacci

La famosa sequenza di Fibonacci è una serie di numeri interi in cui ognuno di essi è la somma dei due precedenti. I primi numeri sono infatti: 1, 1, 2, 3, 5, 8, 13, 21. Proviamo di seguito scrivere un piccolo codice per calcolare i primi numeri della sequenza minori di 1000. L'idea di base è quella di tenere traccia degli ultimi due numeri, sommarli e produrre cosi il terzo in un ciclo che stampi i primi numeri sino a raggiungere 1000. Vediamo il codice:

```
f1 = 1
f2 = 1
next = f1 + f2
while f1<=1000:
  print(f1)
  f1 = f2
  f2 = next
  next = f1 + f2
```

Nelle prime tre righe definiamo i primi due numeri, che son sempre uguali a 1, e il terzo dato dalla somma. Queste tre varibili ci permettono di far partire il ciclo e sono quelle con cui teniamo traccia degli ultimi tre numeri della sequenza, ad ogni iterazione. La condizione che farà terminare il ciclo `while` è che f1 sia maggiore di 1000. All'interno del ciclo abbiamo 4 righe indentate: la prima stampa il primo valore f1, poi lo sovrascriviamo con il successivo f2, sovrascriviamo anche f2 con il terzo valore contenuto in next, e poi aggiorniamo quest'ultimo con il nuovo dato dalla somma di f1 e f2. Questo meccanismo a cascata di sovrascrittura e aggiornamento delle tre variabili è piuttosto standard e generico: è utilizzabile infatti

anche in altri tipi di linguaggi. Questo che vedremo di seguito invece è una versione ristretta del codice, che sfrutta l'assegnazione multipla delle variabili tipica del Python:

```
f1, f2 = 1, 1

while f1<=1000:
  print(f1)
  f1, f2 = f2, f1+f2
```

Provate voi stessi entrambe i codici e verificatene il corretto funzionamento controllando l'output generato.

1.11 *Risoluzione di equazioni di secondo grado*

Proponiamo di seguito come implementare un programma per la risoluzione delle equazioni di 2° grado che costituisce un altro semplice esempio per sfruttare gli elementi visti sinora.

Come sappiamo dall'algebra le soluzioni di una equazione di 2° grado si trovano risolvendo la seguente espressione:

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a}$$

.

Esse dipendono quindi dal suo discriminante:

$$\Delta = b^2 - 4ac = \begin{cases} > 0 & \text{due soluzioni reali distinte} \\ = 0 & \text{due soluzioni reali coincidenti} \\ < 0 & \text{due soluzioni immaginarie distinte} \end{cases}$$

Abbiamo abbiamo tre possibilitá da distinguere e questo sembra dunque un ottimo caso in cui sfruttare il costrutto `if`.

Prima di scegliere uno dei tre casi dovremmo prima calcolare il discriminante a partire dai coefficienti a, b, c della equazione. Questi potranno essere inseriti chiesti all'utente mediante l'istruzione `inupt`. Vediamo di seguito come risulta l'implementazione:

```
a, b, c = input('Inserisci a, b, c ")

discr = b**2 - 4*a*c
```

Di seguito invece vediamo come implementare i tre casi e per ognuno calcolare le soluzioni corrispondenti:

```
if discr >= 0:
  x1 = (-b + sqrt(discr)) / (2*a)
  x2 = (-b - sqrt(discr)) / (2*a)
else:
  discr = abs(discr)
  x1 = (-b + 1j*sqrt(discr)) / (2*a)
  x2 = (-b - 1j*sqrt(discr)) / (2*a)
```

```
print( "Le soluzioni della eq. sono:\n
  x1 =" + str(x1) + "\n" + " x2 = " + str(x2) )
```

Come si nota abbiamo raggruppato insieme i due casi `discr >= 0` e l'altro, `> 0`, non è scritto esplicitamente ma usiamo il generico comando `else`, visto che non ci sono altri casi. Nel primo dei due casi abbiamo calcolato le due soluzioni in scivendo una espressione del tutto simile a quella matematica. Nel secondo caso, poiché sappiamo che le soluzioni sono immaginarie, modifichiamo il discriminante prendendone il suo valore assoluto e poi moltiplichiamo la sua radice quadrata per un numero immaginario puro `1j` [5].

[5] cosa succede se provate a fare sqrt(-1)?

Quello che stiamo quindi facendo è chiede all'utente i tre coefficienti con i quali calcoliamo il discriminante e poi mediante il costrutto `if` controliamo il suo valore e calcoliamo le soluzioni corrispondenti. Una volta che abbiamo le due soluzioni, in uno dei due casi, le facciamo scrivere mediante la funzione `print`.

Una nota importante prima di concludere. La funzione `sqrt` fa parte delle funzioni matematiche della libreria `math`, và perció importata all'inizio del programma mediante il solito comando `from math import sqrt`.

1.12 Liste e Array

Sinora abbiamo visto come definire e fare operazioni con variabili che contengono un singolo valore, che in fisica chiamiamo scalari. Sono però molto usate variabili a più valori come i vettori e le matrici e la loro generalizzazione a N dimensioni. Il Python soddisfa questa esigenza fornendoci delle variabili di diverso tipo che possono contenere più valori. Il tipo che vedremo per primo è la lista, di default nel linguaggio. A seguire parleremo anche degli array che però sono contenuti nel pachetto Numpy, insieme a tutta una serie di funzioni per compiere diversi tipi di operazioni su di essi, di cui le principali verrano anch'esse spiegate in seguito.

1.13 Liste

Una lista è una variabile che può contenere un numero (illimitato) di quantità anche diverse tra loro poste una di seguito all'altra. Per esempio, potremo conservare in una lista una serie di misure prese in laboratorio relative ad un certo esperimento.

Vediamo come si definisce un varibile lista:

```
r = [ 1, 2+3j, 3, 4.2, -5]
print(r)
```

Come si vede è molto semplice e si possono mettere in lista, se si vuole, anche qauntità diverse tra loro come interi, float e complessi.

Inoltre è possibile definire una lista partendo da delle variabili a singolo valore definite in precedenza:

```
x= 2.2
y=-3.1
```

```
z=4.9
r = [ x, y, z]
print(r)
```

In questo modo avremo una lista che contiene i valori delle tre variabili. Da notare che se i valori delle variabili dovessero cambiare in un secondo momento nel nostro programma, la lista r non viene aggiornata in automatico, ma conserva i valori che le variabili avevano nel momento in cui è stata assegnata. Se vogliamo aggiornare la lista dovremo assegnarla nuovemente.

Mostriamo anche come si possano usare delle espressioni per definire i singoli elementi di una lista, come per esempio:

```
r = [2*x, y+z-x, z*sqrt(x**2 - y)]
```

Verranno in questo caso calcolati i valori delle espressioni e memorizzati nella lista.

Ora che abbiamo la nostra sequenza di valori, vediamo le operazioni più utili che si possono effettuare sulle liste e sui singoli elementi:

```
print(r[0], r[1], r[2])
lenght = sqrt(r[0]**2 + r[1]**2 + r[2]**2)
print(lenght)
r[1] = 3.4
print(r)
total = sum(r)
print(total)
mean = sum(r)/len(r)
print(mean)
print( min(r), max(r) )
```

Come si intuisce possiamo richiamare il valore di ogni elemento della lista con `[i]` dove i è l'indice della posizione [6], sapendo che il primo elemento ha posizione 0. Oltre a leggere i singoli elementi, possiamo anche cambiarne il valore, come vediamo in linea 4, in maniera del tutto simile a quanto facciamo per una variabile semplice.

[6] cosa ottenete se mettete un indice negativo, per esempio -1 ?

Scopriamo inoltre che esistono delle funzioni di default: `sum()`, `len()`, `min()`, `max()`, che fanno rispettivamente la somma degli elementi della lista, calcola la sua lunghezza, trova il minimo e il massimo valore della lista.

Un altra utile caratteristica delle liste in Python è la possibilità di aggiugere elementi quando si vuole ad una lista precedentemente creata. Supponendo di avere la precedente lista r ancora definita, possiamo aggiunere un elemento in questo modo:

```
r.append(7.2)
```

La funzione append() aggiunge alla fine della lista `r` il valore indicato. Di nuovo potremo utilizzare una espressione matematica al posto di un valore preciso. Vediamo, perciò, di seguito un altro modo di assegnare una lista:

```
r=[]
r.append(3.7)
r.append(-2.3)
```

```
r.append(83.3)
print(r)
```

Nella prima riga assegnamo a `r` una lista vuota, successivamente aggiungiamo due valori usando la funzione `append()`. Come potete immaginare è possibile anche eliminare elementi dalla lista:

```
r.pop()
print(r)
r.pop(1)
```

La funzione `pop(n)` elimina l'elemento di posto n della lista. Se n non è specificato, come in riga 1, viene eliminato l'ultimo elemento.

La generazione dei numeri di Fibonacci di cui si è parlato può essere riformulata tramite liste nel modo seguente:

```
lst=[0,1]
numeroF=40    # massimo numero desiderato
for i in range(1,numeroF+1):
    s=lst[-1]+lst[-2]
    lst.append(s)
    print i,lst[-1]
```

Pensateci su.

1.14 Array

Le variabili di tipo array sono molto simili a quelle di tipo lista, infatti contengono anch'esse una serie di valori, hanno però alcune importanti differenze rispetto alle liste:

1. Il numero degli elementi è fissato in fase di assegnazione, perciò non si potranno ne aggiungere ne rimuovere elementi.
2. Gli elementi di un array devono essere dello stesso tipo, per esempio tutti float, e non si può cambiare il tipo degli elementi dopo l'assegnazione della variabile.
3. Gli array possono avere piu dimesioni. In particolare noi avremo a che fare con array 1D e 2D, vettori e matrici.
4. Con gli array potremo eseguire semplicemente le operazioni matematiche che sono tipiche di vettori e matrici, cosa che non si può fare in maniera altretanto immediata con le liste.
5. Gli array e le funzioni che operano su di essi sono molto efficienti e più veloci, specialmente se si lavora con array molto grandi.

Per poter creare degli array dovremo per prima cosa importare dal pacchetto numpy le funzioni necessarie. Di seguito riportiamo alcuni esempi di come creare degli array:

```
1  from numpy import array, empty, zeros, ones
2  r = array([1,2,3])
3  print(r)
4  r = array([[1,2,3],[4,5,6]],float)
5  print(r)
6  r = empty(3,int)
7  print(r)
```

```
8   r = zeros([3,4],int)
9   print(r)
10  r = ones(4,float)
11  print(r)
```

Dunque, nella prima riga importiamo la funzione di base per creare gli array e altre tre che rispettivamente generano array vuoti, di elementi tutti uguali a zero e tutti uguali a 1. Nella seconda e quarta riga creiamo un array scrivendo esplicitamente i singoli elementi usando la sintassi usata per creare una lista, quindi elenchiamo gli elementi usando le virgole e racchiudendoli tra parentesi quadrate. Inoltre notiamo come si possa definire il tipo a prescindere da quello degli elementi forniti: nella quarta riga infatti gli elementi della lista sono interi, ma l'opzione float crea un array di float, come si nota con il `print` successivo. Sempre nella quarta riga notiamo come si possano definire le matrici esplicitamente, creando in pratica una lista di liste: nel nostro caso abbiamo due liste, quindi due righe, e tre elementi per lista, quindi tre colonne.

Nelle righe successive usiamo da prima la funzione `empty()`, specificando la lunghezza pari a 3 e che gli elementi sono degli interi. In maniera simile usiamo la funzione `zeros()`, specificando sta volta che vogliamo una matrice $3x4$, sempre di interi. Stesso approccio per la creazione di un vettore di quattro elementi float pari a 1.0, mediante la funzione `ones()`.

In generale, questo tipo di funzioni che creano array, accettano come primo paramentro (o argomento) la dimensione che può essere un singolo numero, se vogliamo un vettore, o una lista con N dimensioni se vogliamo un array generico.

Una volta creati, gli array vengono modificati in maniera del tutto simile alle liste. Vediamo un esempio chiarificatore:

```
from numpy import zeros
z = zeros([2,2],float)
a[0,1]=-1.5
a[1,0]=1.5
print(z)
```

Da tenere a mente che il primo indice è quello delle righe, mentre il secondo è quello delle colonne e che la numerazione inizia sempre da 0.

Un altro modo di creare un array, molto utile se si vuole analizzare dei dati contenuti in files, è quello di importare dei dati da un file e metterli al suo interno. Il pacchetto numpy contiene la funzione `loadtxt()` che ci semplifica parecchio il lavoro, come si nota dal seguente codice:

```
from numpy import loadtxt
dati = loadtxt('miei_dati.dat',float)
print dati
```

Quello che è necessario è fornire un file contenente una o piu colonne di dati specificandone il percorso, o semplicemente il nome se la shell che avete aperto o il programma che state eseguendo si

trovano nella stessa directory del file dati. La funzione si occuperà di creare un vettore, se il file contiene una sola colonna, o una matrice, se avete più colonne, con tutti i valori contenuti nel file nello stesso identico ordine.

1.15 Aritmetica degli array

Vediamo di seguito che tipo di operazioni matematiche possiamo eseguire sugli array.

Ovviamente possiamo usare i singoli elementi per fare operazioni, come per esempio:

```
from numpy import zeros
z = ones(3,float)
z[0] = z[2]*3 + z[1]**2
```

e operazioni similari che coinvolgano i singoli elementi.

Di seguito invece riportiamo quelle che sono le operazioni più strettamente legate all'algebra dei array:

```
from numpy import array
# moltiplicazione di uno scalare per un vettore
a = array([1,2,3,4],int)
b = 2*a
print(b)

# somma di uno scalare per un vettore
print (a+1)

# somma tra due vettori
b = array([1,2,3,4],int)
print(a + b)

# moltiplicazione di due vettori per elementi
print(a * b)

#prodotto scalare
from numpy import dot
print(dot(a,b))
```

Come si nota è tutto abbastanza intuitivo, ma andiamo con ordine. La moltiplicazione di uno scalare per un vettore (o matrice) avviene allo stesso modo in cui averrebbe se a fosse una variabile a singolo valore: Python si occupa autonomamente di moltiplicare lo scalare per tutti gli elementi. È da notare che la variabile b è in automatico un vettore e non è necessario crearlo in precedenza.

La somma tra vettori avviene anchessa usando l'operatore matematico + che usiamo con le variabili semplici. Stesso funzionamento per la somma di uno scalare per un vettore o matrice. Un comportamento inatteso lo abbiamo nella moltiplicazione di due vettori usando il semplice *. Infatti, questa operazione esegue una moltiplicazione tra gli elementi dello stesso indice dei due vettori, ottenendo come

risultato ancora un vettore. Stesso funzionamento nel caso di matrici. Per ottenere il prodotto scalare è dunque necessario importare la specifica funzione dot() che esegue correttamente questo prodotto, sia nel caso di due vettori, sia nel caso di due matrici, sia nel caso misto vettore e matrice.

Vediamo un esempio di come si possa svolgere una operazione matriciale anche complicata come questa:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} 3 \\ 5 \end{pmatrix} + 2 * \begin{pmatrix} 3 \\ 5 \end{pmatrix} - \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ -3 & 1 \end{pmatrix} = \begin{pmatrix} 32 & 31 \\ 34 & 28 \end{pmatrix}$$

con un codice semplice come il seguente:

```
from numpy import dot
a=array([[1,3],[2,4]],int)
b=array([3,5],int)
c=array([[1,2],[-3,1]],int)
print(dot(a,b) + 2*b - dot(a,c))
```

Esistono moltissime funzioni che riguardano gli array, le più utili le vedremo più avanti nel libro quando presenteremo i vari codici proposti per la soluzione di problemi di fisica.

Una nota che facciamo ora è quella che rigurda la copia degli array. Vediamo il codice seguente:

```
a = ones(3,int)
b = a
a[1] = 2
print(a)
print(b)
```

Nella seconda riga noi tentiamo di copiare il vettore a in un altro chiamato b. Ma come si osserva dall'output, modificando l'elemento 1 di a abbiamo modificato anche quello di b. Questo avviene perché nella secondo riga abbiamo semplicemente cambiato il nome del vettore a in b, non abbiamo una vera e propria copia. La variabile b in realtà identifica sempre l'array a ma con un altro nome. Per avere una vera e propria copia, quindi indipendente dal vettore originale, dovremo usare la funzione apposita copy() che vediamo di seguito:

```
from numpy import copy
a = ones(3,int)
b = copy(a)
a[1] = 2
print(a)
print(b)
```

Dove abbiamo sostituito la riga `b = a` con la `b = copy(a)`. Ora i due vettori a e b sono del tutto indipendenti.

1.16 Slicing

In questa breve sezione vedremo un utilissimo trucco applicabile sia alle liste che agli array per poter selezionare e utilizzare solo parte di essi. Il metodo è molto semplice: se `r` è una lista (o array), `r[m:n]`

è un'altra lista che contiene gli elementi dall'indice m all'indice n-1 della lista r originale. Vediamo un esempio:

```
a = [ 1, 20, 3, 40, 5, 60, 7, 80, 9 ]
s = a[3:7]
print(s)
[40, 5, 60, 7]
```

Come vediamo una nuova lista s viene creata con gli elementi di posto 3, 4, 5 e 6 della lista a. Stiamo attenti che l'indice iniziale é sempre lo 0 e che scrivendo 7 come indice finale, la nuova lista avrà come ultimo elemento quello di posto 6 (7-1).

Ci sono anche altri modi più rapidi:

`a[:4]` se non specifichiamo il primo indice significa che vogliamo partire dal primo elemento, quello con indice 0

`a[5:]` se non specifichiamo il secondo indice significa che vogliamo arrivare sino all'ultimo elemento, quello con indice n-1

`a[1:8:2]` si può anche specificare uno step diverso da 1: prendiamo gli elementi dall'indice 1 al 7 con passo 2, quindi 1, 3 , 5.

`a[2,3:6]` nel caso avessimo una matrice possiamo selezionare per esempio la riga di indice 2 (quindi la terza!) e prendere solo gli elementi 3, 4, 5, ottenendo quindi un vettore.

`a[2:4,3:6]` possiamo anche estrarre una matrice da un'altra, come in questo caso: prendiamo le righe 2 e 3, ma selezionando solo gli elementi 3, 4, 5.

Questo dello slicing è uno strumento molto utile, ma necessita di un po di pratica e molta attenzione quando lo si utilizza perché è facile sbagliare!

1.17 Il ciclo `For`

Ora vediamo un altra istruzione, simile al `while`, che permette di eseguire una serie di righe di codice in maniera ripetitiva. La differenza con il `while` è che decidiamo in anticipo quante volte ripetere il ciclo. Vediamo subito un primo esempio:

```
r = [ 1, 3, 5]
for n in r:
  print(n*2)
print("Ciclo finito")
```

Mediante l'istruzione `for` eseguiamo le righe indentate successive, in questo caso solo il `print`, per ogni elemento della lista r. Inoltre ad ogni iterazione, la variabile n assume il valore dell'elemento della lista, e quindi è possibile utilizzarla, come facciamo nella riga indentata. Quello che otteniamo sarà perciò una sequenza di valori pari a n*2, quindi 1, 6, 10 e infine la stringa "Ciclo finito". Lo stesso vale se r fosse un array. Inoltre l'istruzione break vista per il ciclo `while`

può essere utilizzata anche per il ciclo for, messa all'interno di un opportuno costrutto `if`.

Una funzione molto utile per l'utilizzo del ciclo `for` è `range(m, n, p)` che genera una lista di valori da m a n-1 con passo p. Il modo più semplice e standard è specificare solo un valore dei tre che é la n, per esempio:

```
r = range(5)
print(r)
for n in r:
  print("Hello")
```

Come si nota dall'output del primo `print()` la lista r contiene i valori da 0 a 4, quindi 5 valori a partire dallo 0. Quindi il ciclo for esegue la riga indentata 5 volte, stampando a video 5 "Hellò'. Facciamo notare come non sia necessario creare la lista r, infatti potremo anche scrivere direttamente:

```
for n in range(5):
  print("Hello")
```

Ora vediamo altre due utili funzioni che creano sequenze di numeri float, contrariamente a range che genera sequenze di interi:

`arange(m, n, p)` ci permette di creare un array di valori float che vanno da m a n-1 con passo p, anche non intero;

`linspace(i, k, n)` ci permette di creare un array di n valori equispaziati tra i valori i e k, inclusi.

Vediamo per esempio gli array prodotti con le seguenti righe:

```
arange(0, 1, 0.1)
linspace(0, 1, 10)
```

Come si nota dall'output son del tutto simili tranne che per l'ultimo valore: nel primo caso non è incluso il 10, come previsto, mentre nel secondo caso è compreso anche il 10.

Di seguito riportiamo un semplice esempio di come il ciclo `for` poss essere utilizzato per calcolare la somma de numeri contenuti in una lista (o array):

```
s = 0
for k in range(1,101):
  s += k
print(s)
print(sum(range(1,101)))
```

Per prima cosa creiamo una variabile s con valore 0, che conterrà la somma parziale e finale degli elementi. Il ciclo assegna a k i valori generati dal la funzione range(1,101) ed esegue la riga indentata nella quale viene aggiornato il valore di s sommando il nuovo valore k al valore precedente di s. Una volta finiti i valori del range, viene stampato il risultato e quello che si ottiene con la funzione di default sum(). Come si nota dall'output i risultati sono identici.

Ovviamente questo è solo un esempio ed è ovvio che è molto più conveniente usare la funzione sum() che è piu facile da scrivere ed è più veloce. L'utilità del ciclo for però è facilemente comprensibile anche semplicemente eseguendo una variante del codice precedente [7]:

```
s = 0.0
ss = 0.0
for k in arange(1,101):
  s += 1/k
  ss += k**2
print(s)
print(ss)
```

In questo caso facciamo la somma degli inversi e il quadrato dei numeri generati dalla funzione arange() direttamente nello stesso ciclo for. Da notare che abbiamo definito s come un float e usiamo la funzione arange perché dobbiamo fare delle divisioni tra numeri non interi.

[7] In verità anche questo ciclo for puó essere sostituito da operazioni fatte direttante usado gli array. Riuscite a trovarle?

1.18 Prodotto tra matrici

Vediamo di seguito come potremo implementare il prodotto tra matrici mediante l'uso dei cicli `for`.

Il programma sarà solo a scopo didattico. Il prodotto tra matrici abbiamo visto essere già implementato nella libreria Numpy. Questa implementazione è fatta piú a basso livello perciò molto piú efficiente di quella che realizzeremo noi. Resta comunque un utile esercizio per capire come implementare una relazione algebrica non banale.

Il prodotto tra due matrici A, B di dimensioni $m \times n$ e $n \times p$ rispettivamente è dato dalla seguente relazione:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}.$$

Essa definisce come calcolare l'elemento di posto i, j della matrice risultato C, mediante una moltiplicazione degli elementi della riga i della matrice A con quelli della riga j della matrice B e una somma finale su tutti i prodotti ottenuti.

Per un singolo elemento della matrice C, l'implementazione è abbastanza semplice una volta scelti gli indici i, j:

```
for k in range(n):
  c[i,j] += a[i,k]*b[k,j]
```

Quello che dovremmo aggiungere per poter eseguire questa operazione per tutte le possibili coppie di indici i, j sono due cicli `for`, uno per indice, in questo modo:

```
for i in range(m):
  for j in range(p):
    for k in range(n):
      c[i,j] += a[i,k]*b[k,j]
```

Riusciamo cosí a generare tutte le coppie possibili e usarle all'interno del ciclo sull'indice k per identificare l'elemento della matrice *C* su cui salvare il prodotto. Per rendersi conto dell'ordine in cui vengono generate le coppie *i*, *j* potete aggiungere un `print(str(i),str(j))` all'interno del ciclo sui *j*. A parole, il primo ciclo seleziona un valore per l'indice *i* e il secondo genera tutti quelli per l'indice *j*. Una volta finiti, l'indice *i* cambia e si ripete la generazione degli indici *j*, e cosi via.

Concludiamo il programma definendo le matrici `a` e `b`, di cui vogliamo fare il prodotto e la matrice `a` su cui salvare il risultato:

```
a = array([ [1,2,3], [1,2,3], [1,2,3] ])
b = eye(3)
c = zeros((3,3))
```

Queste righe vanno ovviamente messe all'inizio del nostro programma, prima dei tre cicli `for`. Come si intuisce, stiamo semplicemente facendo il prodotto tra una matrice per la matrice identità in modo da verificare facilmente che il risultato sia corretto. Il nostro programma è comunque piú generico e può moltiplicare qualunque matrice, anche non quadrate, salvo che l'ordine della moltiplicazione sia corretto.

Si invita il lettore a calcolare il prodotto di due matrici di grande dimenzione, 1000×1000 per esempio, usando questo codice e generando le matrici in maniera random mediante[8]:

[8] la funzione `rand` genera matrici di qualunque dimensione contenenti numeri random compresi in [0,1). La libreria random contiene tante altre funzioni relative alla generazione di numeri random...

```
from numpy.random import rand
N = 1000
a = rand(N,N)
b = rand(N,N)
```

e confrontare i tempi di esecuzione rispetto alla funzione `dot(a,b)` implementata in NumPy.

1.19 *Funzioni definite dall'utente*

Come abbiamo visto sinora esistono molte funzioni già preimpostate in Python, e nei pacchetti aggiuntivi come Numpy, che ci permettono di eseguire calcoli anche complicati chiamando semplicemente la funzione e passandogli delle variabili su cui operare. Le funzioni in generale sono dei pezzi di codice che eseguono delle determinate operazioni a partire da alcuni argomenti (variabili) in ingresso. Ogni volta che in un codice si vuole eseguire quelle operazioni, non c'é bisogno di riscriverle per intero, ma basta richiamare la funzione che le svolge.

Oltre alle funzioni di default possiamo definire noi stessi delle nuove funzioni da utilizzare nei nostri codici quando vogliamo. Vediamo subito come nel seguente esempio [9]:

[9] anche in questo caso, è possibile usare le operazioni sugli array?

```
def factorial(n):
  f=1.0
```

```
  for i in range(1,n+1):
    f *= i
  return f

a = factorial(10)
print(a)
```

In questo esempio abbiamo definito una funzione chiamata factorial(n) che calcola il fattoriale di n, variabile che viene passata quando si chiama la funzione. Ogni volta che richiamiamo la funzione, questa esegue il codice al suo interno (in pratica tutte le righe indentate) sino a che raggiunge l'istruzione `return` che fa uscire dalla funzione e restituisce, se presente, il valore di una variabile, come nel nostro caso, oppure anche un array o più variabili separati da virgola.

È inoltre possibile anche passare alla funzione piu di una variabile. Vediamo un altro caso:

```
1  def cart_coord(r,theta):
2    x = r*cos(theta)
3    y = r*sin(theta)
4    return x,y
5    #return [x,y]
6    #return array([x,y])
7
8  X, Y = cart_coord(3, pi/12)
9  #coords = cart_coord(3, pi/12)
10
11 print(x,y)
12 #print(coords)
```

Questa funzione come si intuisce, prende in ingresso le coordinate polari, raggio e angolo, di un punto e ci restituisce quelle cartesiane. Il tipo di variabile che restituisce può essere differente: con il primo return (riga 4) resituiamo due variabili, una per coordinata, quindi quando richiamiamo la funzione dobbiamo usare l'assegnazione a due variabili della riga 8. Gli altri due return (commentati, nelle righe 5, 6) restituiscono una singola variabile di tipo lista o array, rispettivamente, perciò l'assegnazione sarà a singola variabile (riga 9).

Una nota riguarda le variabili usate all'interno della funzione: queste sono dette locali perché è possibile utilizzarle solo all'interno della funzione stessa ma non fuori. Guardando l'ultimo codice, le variabili locali della funzione cart_coord() sono x, y, r e theta: infatti se mettessimo un `print(r,theta)` fuori dalla funzione, per esempio dopo l'ultima riga, otterremo un errore, perché quelle variabili non sono definite nel resto del programma. Viceversa, all'interno della funzione non possiamo utilizzare delle variabili definite fuori da essa: se ci servono dovremo passarle come argomento.

1.20 Pratiche di buona programmazione

In questa sezione conclusiva vogliamo dare una serie di consigli e regole usate quando si programma, valide in generale per tutti i linguaggi. L'obiettivo è quello di scrivere un programma che sia il più semplice, comprensibile e veloce possibile. Per otterenere queste caratteristiche bisogna seguire delle regole e fare ovviamente tanta pratica.

1. Aggiungere i commenti al codice. Questa pratica, che risulta tediosa, si rivela però molto utile. Capita spesso che si debba riprendere in mano un codice scritto in passato, neanche tanto lontano, e di non ricordare più il suo funzionamento. Se abbiamo scritto dei commenti al suo interno faremo molta meno fatica. Stessa cosa quando dobbiamo condividere il nostro codice con altre persone: sarà piu facile anche per loro capirne il funzionamento e apportare le modifiche che desiderano.
2. Utilizzare nomi esplicativi per le variabili. Il nome delle variabili deve far capire la quantità che rappresentano, ma dovremo anche contenere la loro lunghezza. Per esempio potremo usare nomi come E, t per energia e tempo, simili quindi a quelli usati nelle formule matematiche, o anche alpha, beta per gli angoli. Potremo anche essesere meno sintetici e scrivere per esempio e_mass per la massa dell'elettrone, angular_velocity per la velocità angolare, usando il trattino basso per separare le parole.
3. Usare il corretto tipo di variabile. Se una quantità è intera è sempre preferibile salvarla in una variable intera, e cosi pure per le quantità reali e complesse usare float e complex. Questo rende più efficiente l'uso delle memoria da parte del vostro codice.
4. Importare le funzioni all'inizio. Le prime righe del codice sono di solito usate per importare i vari pacchetti, moduli e funzioni di cui abbiamo bisogno nel resto del codice. Sarà piu facile verificare che ci siano tutte eaggiungere quelle mancanti o correggere eventuali errori.
5. Definire le costanti. Sempre nelle prime righe di codice è utile definire le variabili che rappresentano costanti, in modo da poterle usare ogni volta che vogliamo, nel resto del codice. Inoltre sarà piu facile individuarle e modificarle nel caso volessimo eseguire il codice con valori diversi di queste costanti.
6. Sfruttare le funzione. Quando nel nostro programma abbiamo bisogno di eseguire un certo numero di operazioni, ripetutamente in diverse parti del codice, è conveniente racchiudere quelle righe di codice all'interno di una funzione e richiamarla. Questo aumenta la leggibilità del programma e diminuisce il numero di righe da scrivere. Tra l'altro le funzioni possono essere utilizzate anche in programmi diversi. Queste vanno scritte sempre nelle righe iniziali, dopo l'importazione dei pacchetti e la definizione delle costanti.
7. Stampare dei messaggi durante l'esecuzione del programma. È utile in generale, ma ancor di piu se il programma impiega parecchio tempo per araggiungere il risulato finale, stampare ogni tanto delle

righe di testo che dicano all'utente lo stato di avanzamento del calcolo, fornendo se possibile dei risultati parziali. Questo aiuta l'utente a capire se il suo programma sta funzionando correttamente senza dover aspettare la fine dell'esecuzione. Questa pratica è utile anche in fase di debugging del codice, per poter scoprire errori e funzionamenti non previsti.

8. Scrivere in maniera chiara. Le righe di codice complesse vanno scritte in maniera da aumentarne la leggibilità. Quindi è buona norma usare accuratamente gli spazi tra variabili e operatori, righe vuote per separare parti diverse di programma, spezzare delle linee troppo lunghe mettendo un backlash \ alla fine della riga e continuando a capo:

```
energy = mass*(vx**2 + vy**2)/2 + mass*g*y \
  + moment_inertia*omega**2/2
```

9. Semplificare il programma. Si deve sempre cercare di rendere il programma semplice, minimizzando il numero di righe, trovando degli algoritmi efficienti. Qeusto renderà il programma piu facile da leggere, spesso più rapido nell'esecuzione e sarà piu facile trovare gli errori.

1.21 Errori comuni

In questa breve sezione si vuole portare l'attenzione sui più comuni errori che si commettono durante la scrittura di un codice e che l'interprete puntualmente ci segnalerà. Vi invito a leggere attentamente ciò che ci viene mostrato dall'interprete: questo infatti ci mostra sempre il tipo di errore che si è verificato e la riga del codice che lo ha generato.

- **IndexError:**

```
v=arange(10)
for i in range(10):
  v[i]=v[i+1]
```

l'output sarà il seguente:

```
---------------------------------------------------
IndexError         Traceback (most recent call last)
<ipython-input-5-4114e34fddca> in <module>()
      1 for i in range(10):
----> 2         v[i]=v[i+1]
      3

IndexError: index out of bounds
```

In questo esempio, come in tutti i casi in cui otteniamo un "index out of bounds", il valore dell'indice i eccede il massimo indice del vettore `v`: infatti all'ultima iterazione, la `i` vale 9, ma nella riga 2 chiediamo di leggere il valore di posto `i+1`, quindi 10, del vettore `v` la cui lunghezza è si pari a 10, ma l'indice più grande è il 9.

Dobbiamo perciò prestare molta attenzione al valore degli indici che usiamo per leggere gli elementi degli array.

- **SyntaxError:**

```
v=arange(10)
for i in range(10):
  v[i]=v[i+1
  print v[i]
```

l'output che si ottiene è:

```
  File "<ipython-input-7-d4c3c358329e>", line 4
    print v[i]
SyntaxError: invalid syntax
```

Questo errore può creare qualche grattacapo: ci viene segnalato un errore di tipo `SyntaxError` nella riga 4. Un `SyntaxError` è un errore che si verifica quando scriviamo qualcosa che l'interprete python non conosce. In questo caso però la riga segnalata è scritta correttamente! Quando capita qualcosa del genere, si deve andare a vedere la riga precedente a quella segnalata. Nella riga 3 infatti troviamo che la parentesi quadra non è stata chiusa correttamente e questo genera l'errore segnalato.

Bisogna cercare di stare attenti a chiudere correttamente ogni parentesi (tonda o quadra) che viene aperta, soprattutto quando ci hanno righe piuttosto complesse.

Altri casi frequenti di `SyntaxError` sono:

```
if a = 2: # questo confronto si fa usando ==
  print a

if => a:   # questo confronto si fa usando >=
  print a

if a != 3 # mancano i : che chiudono il costrutto if
  print a


for i in range(10):
print i

# dopo i : l'interprete si aspetta una riga indentata
# questo produrr\`a una forma specifica di SyntaxError:
# IndentationError: expected an indented block

max([1,2,3,4) # manca la parentesi chiusa quadra
```

- **TypeError:**

```
t=1,2
N=100
print t/N
```

l'output sarà il seguente:

```
---------------------------------------------------
TypeError         Traceback (most recent call last)
<ipython-input-5-67cf2cc85008> in <module>()
----> 1 print t/float(N)

TypeError: unsupported
     operand type(s) for /: 'tupl\'e and 'float'
```

In questo caso abbiamo definito due variabili di cui poi chiediamo di fare la divisione e otteniamo un `TypeError` che ci informa che non è definita l'operazione di divisione (/) tra una variabile di tipo tuple e una di tipo float. Osservando meglio la prima riga, notiamo che stiamo definendo male la variabile `t`: noi vogliamo un numero float perciò dovremo usare il punto e non la virgola! Usando la virgola definiamo una variabile di tipo tuple, cioè una lista non mutabile di cui ovviamente non possiamo fare la divisione.

- **NameError:**

```
t=10
a=9.8
v0=1.0
x=0.5*g*t + v0*t
```

l'output sarà il seguente:

```
---------------------------------------------------
NameError         Traceback (most recent call last)
<ipython-input-10-50dccf3f255e> in <module>()
----> 1 x=0.5*g*t + v0*t

NameError: name 'g' is not defined
```

In questo semplice esempio inizialmente salviamo l'accelerazione di gravità in una variabile chiamata `a`, poi però nella riga 4 usiamo una variabile `g` che non è stata definita producendo l'errore di tipo `NameError`. Questo tipo di errore si ottiene anche in caso si chiami una funzione non definita. Capita spesso che errori del genere si verifichino quando sbagliamo a scrivere il nome di una variabile o funzione in fase di definizione o di lettura.

1.22 Grafici con matlplotlib

In questa sezione vedremo le istruzioni di base della libreria matplotlib per poter ottenere dei grafici perfetti per la pubblicazione dei vostri dati. Per chi volesse approfondire rimandiamo alla pagina ufficiale della libreria.

Ci sono due modalità per comporre un grafico: una interattiva

```
interactive(True) # oppure ion()
```

e una non interattiva

```
interactive(False) # oppure ioff()
```

Nella prima la shell resta attiva e potremo fare tutte le modifiche che vorremo e queste verranno immediatamente mostrate nella finestra del grafico. Nella seconda, invece, comporremo il grafico senza vederlo, soltanto quando è pronto potremo mostrare la finestra che lo contiene o salvarlo direttamente.

Iniziamo nella modalità interattiva a realizzare il grafico delle seguenti tre funzioni sinusoidali con ampiezza e frequenze differenti:

$$y_1 = sin(x)$$
$$y_2 = 3 * sin(x)$$
$$y_3 = 2 * sin(0.5 * x)$$

La prima cosa che dovremo fare è importare tutte le funzioni della libreria e attivare la modalità interattiva, mediante:

```
from pylab import *
ion()
```

In questo modo avremo caricato anche le librerie numpy, in un colpo solo.

Di seguito invece costruiamo i vettori `x` comune e i tre vettori `y` per le tre funzioni:

```
x=arange(0,2*pi*10,0.1)
y1 = sin(x)
y2 = 3*sin(x)
y3 = 2*sin(0.5*x)
```

ora per avere il grafico delle funzioni tutte insieme, usiamo la funzione `plot()`

```
plot(x,y1)
plot(x,y2)
plot(x,y3)
```

La prima funzione `plot()` apre una finestra che mostra il grafico della prima funzione. Le successive chiamate della funzione `plot()` stampano nella stessa figura le altre due funzioni.

Dunque se volessimo semplicemente visualizzare i nostri dati potremo anche fermarci qua, ma andiamo avanti.

La seconda cosa che solitamente uno vorrebbe aggiungere ad un grafico, poichè di fondamentale importanza, sono le etichette degli assi. Queste le si aggiunge cosi:

```
xlabel('Time (sec)') # label dell'asse X
ylabel('Angle (degree)') # label dell'asse Y
```

Le due funzioni prendono come primo argomento una stringa che contiene la nostra etichetta. Eventualmente si può anche aggiungere un titolo alla figura, mediante:

```
title('Funzioni sinusoidali')
```

La figura 1.1 mostra quanto abbiamo ottenuto finora.

Se questo è quello che volevamo, possiamo salvare cliccando sul dischetto nel pannello delle icone nella parte alta della finestra del grafico.

Ora se chiudiamo la finestra il grafico andrà perduto. Se volessimo modificarlo dovremo prima ricrearlo eseguendo nuovamente le funzioni di plot e quelle per le etichette.

Introduciamo a questo punto la possibilità di avere più finestre aperte con grafici differenti. Apriamo perciò una nuova finestra, mediante:

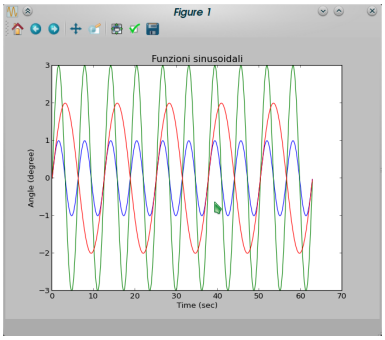


Figura 1.1: Etichette e titolo.

```
figure(2) # al posto di 2 possiamo mettere anche una stringa
```

comparirà una nuova finestra dal titolo contenente l'argomento della funzione. Da questo momento tutti i successivi comandi andranno a modificare il grafico mostrato un questa ultima finestra. La precedente non verrà modificata. Se volessimo modificare una delle precedenti finestre aperte, dovremo prima richiamarla, nel nostro caso:

```
figure(1) # oppure al posto di 1 il titolo della finestra
```

Cogliamo l'occasione dunque e rifacciamo il plot delle funzioni nella nuova finestra, in questo modo:

```
xlabel('Time (sec)') # label dell'asse X
ylabel('Angle (degree)') # label dell'asse Y
plot(x,y1,label='sin(x)')
plot(x,y2,label='3*sin(x)')
plot(x,y3,label='2*sin(0.5*x)')
```

Come notiamo abbiamo specificato un argomento in più chiamato label: questo specifica l'etichetta di ogni singolo grafico che andrà nella legenda che creiamo semplicemente con :

```
legend()
```

Di default la leggenda sarà posizionata in alto a destra, ma possiamo scegliere un altra posizione specificando l'argomento loc:

```
legend(loc="upper center")
```

per esempio in questo modo la legenda sara ridisegnata in alto al centro.[10]

I tool ingrandimento e sposta presenti sempre nel pannello superiore della finestra del grafico permettono di ingrandire o rimpiciolire e spostare il grafico come vogliamo. È pero possibile anche farlo dalla linea di comando specificando i limiti degli assi, per esempio:

```
ylim(-3,6)
xlim(0,10)
```

In questo modo lasciamo dello spazio vuoto in alto per la legenda e prendiamo soltanto le prime oscillazioni. Il risultato ottenuto fino a qui è mostrato in figura 1.2.

[10] si veda help(legend) per una lista di tutte le possibili posizioni delle legenda.

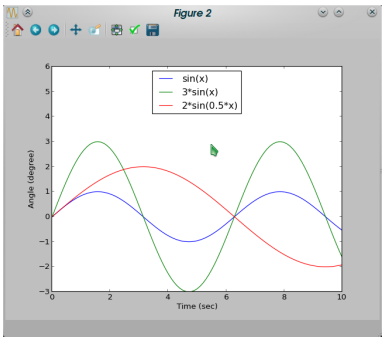


Figura 1.2: Legenda e limiti delle scale degli assi.

Ora vediamo una serie di altre opzioni utili:

```
plot(x,y1, '--',linewidht=3.0)
```

la stringa passata come terzo argomento è un modo rapido per specificare il colore, il tipo di linea e il tipo di punto, per esempio:

- `'g'`, indichiamo semplicemente il colore verde (green), si ottiene anche usando color="green"
- `'--'` , indica che vogliamo la linea tratteggiata
- `'-s'` , indica che vogliamo linea continua e punto quadrato
- `'s'` , indica che vogliamo solo punti quadrati senza linea che li congiunge

L'argomento linewidth invece specifica la grossezza della linea e/o del punto usati. La stessa grossezza verrà usata nella legenda.

Possiamo anche variare la dimensione dei caratteri per le etichette degli assi, per i numeri degli assi e per la legenda:

```
xlabel('Time (sec)',fontsize=20)
xticks(fontsize=10)
legend(fontsize=15)
```

come si nota il parametro da impostare comune fontsize permette di modifica la dimensione del carattere per i tre tipi di testo.

Se volessimo impostare dei parametri globali per il nostro grafico potremo usare la funzione `rc()`, per esempio:

```
rc('font',size=20)
rc('lines',linewidth=3, markersize=3)
```

in questo modo, cambiamo la dimensione di tutto il testo che compare nel grafico (prima riga) e impostiamo la grossezza delle linee e dei punti di tutti i grafici che stampiamo in eguito alla chiamata di `rc()`. Questa funzione `rc()` è utile se usata all'inizio della composizione di un grafico per impostare dei parametri che vogliamo che siano comuni per esempio per il testo o per le linee. Segnaliamo che usando la funzione `rcParams` ci viene restituita tutta la lista di parametri e rispettivi valori che sono impostati di default e che potremo modificare.

Mediante la funzione `grid()` specifichiamo se vogliamo la griglia o meno nel nostro grafico:

```
grid(True) # o False
```

Per finire citiamo anche:

```
axis('equal')
```

per poter rendere la scala dei due assi uguale: utile nel caso del grafico delle orbite.

Passiamo ora alla modalità non interattiva. Questa modalità è utile quando sappiamo già come vogliamo il nostro grafico e magari dopo un certo calcolo vogliamo mostrare il risultato finale, senza dover salvare i dati su file e andarli poi a farne il grafico son programmi

esterni. È molto utile anche per salvare in maniera automatica molteplici grafici senza doverli visualizzare singolarmente. Per esempio, usando le cose imparate nella modalita interattiva, potremo decidere di scrivere uno script come questo:

```
from pylab import *

#ioff() la modalit\'a \'e di default non interattiva

rc('font',size=20)
rc('lines',linewidth=3, markersize=3)

xlabel('Time (sec)') # label dell'asse X
ylabel('Angle (degree)') # label dell'asse Y

x=arange(0,2*pi*10,0.1)

for w in [1,2,4,8]:
  y = 1./w * sin(w*x)
  plot(x,y,label="A="+str(1./w)+"; f="+str(w))

ylim(-3,6)
xlim(0,10)
legend()
show()
```

In questo codice impostiamo inizialmente la dimensione dei font e la grandezza di linee e punti del grafico e impostiamo le etichette degli assi. Poi creiamo l'array dei valori di `x` e dentro il ciclo `for` creiamo il vettore `y` che rappresenta la funzione $y = 1/w \sin(w \cdot x)$ e subito dopo ne facciamo grafico chiamando la funzione `plot()`, indicando inoltre la corretta label. Una volta usciti dal ciclo for impostiamo i limiti degli assi, disegniamo la legenda e facciamo apparire infine la finestra con il nostro grafico mediante la funzione `show()`.

Il risultato è mostrato in figura 1.3.

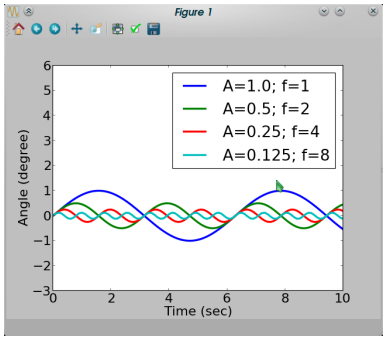


Figura 1.3: Grafico ottenuto in modalita non interattiva.

Ricordiamo che nella modalità non interattiva, sintanto che la finestra non viene chiusa la shell dei comandi non è piu operativa, perciò non potremo inserire nessun comando. Per poter avere di nuovo la shell attiva, dovremo chiudere la finestra del grafico.

Chiudiamo menzionando la funzione che permette di salvare su file il nostro grafico:

```
savefig('nomefile.png',dpi=300)
```

dove il primo argomento è il percorso più il nome del file con la sua estensione su cui vogliamo salvare il file (in questo caso abbiamo specificato solo il nome, perciò il file sarà salvato nella stessa directory dove viene eseguito il codice); il secondo argomento è la risoluzione dell'immagine finale che vogliamo. Ricordiamo che il salvataggio del grafico nella modalità non interattiva, si ottiene chiamando questa funzione prima (o al posto) della funzione `show()`.

1.23 B - Trapezoidi e Simpson

Proviamo ora a implementare i metodi di integrazione dei trapezoidi e di Simpson per calcolare l'integrale di una funzione di nostra scelta in un certo intervallo. Confronteremo inoltre l'accuratezza dei due metodi rispetto al numero di intervalli di integrazione, come descritto in Fig.1.2. Inizieremo anche a familiarizzare con la programmazione in Python e con gli strumenti della libreria `Numpy`. Inoltre vedremo come creare il nostro primo grafico dei dati che abbiamo memorizzato nelle nostre variabili, direttamente dalla shell Python, sfruttando il modulo `Matplotlib`. Un primo assaggio degli elementi che in seguito saranno la base di tutti i codici successivi.

Le prime due righe che dovranno essere presenti in ogni nostro codice saranno:

```
from numpy import *
from matplotlib import *
#from pylab import *
```

queste ci permettono di caricare tutte le funzioni presenti nelle librerie `Numpy` e `Matplotlib`. La terza riga é un modo rapido per caricare entrambe le librerie. Questo modo potrebbe essere criticato, perché sarebbe meglio caricare singolarmente solo le funzioni di cui abbiamo bisogno. Al momento però partiamo caricando tutto insieme, quando saremo diventati più esperti ci preoccuperemo anche di questo.

Ora possiamo proseguire, creando un array che contenga i valori di una funzione in un certo intervallo, per esempio $f(x) = x^4 - 2x + 1$, $x\epsilon[0,2]$.

In Python+NumPy questo lo possiamo fare in questo modo:

```
N=100
x = linspace(0,2,N+1)
y = x**4 - 2*x + 1
```

Nella seconda riga sfruttiamo la funzione `linspace` che genera un array di $N+1$ valori equispaziati nell'intervallo $[0,2]$. Nella seconda riga, sfruttiamo il fatto che ogni operazione fatta su un array viene in automatico eseguita su ogni elemento di esso, ottenendo cosi l'array della funzione che volevamo.

Il metodo dei trapezoidi è espresso dalla seguente espressione:

$$I = \frac{h}{2}(f_1 + f_{N+1}) + h\sum_{i=2}^{N} f_i$$

che possiamo tradurre in codice in maniera diretta:

```
h = abs( x[1] - x[0] )
trap = 0.5*h*(y[0] + y[-1])
trap += sum(y[1:-1]) * h
```

Definiamo la lunghezza dell'intervallino h di integrazione come differenza tra due valori consecutivi di x; scriviamo nella variabile `trap` la somma del primo e del'ultimo valore della funzione divisi

per due; successivamente, aggiungiamo al valore appena registrato la somma di tutti i valori della funzione esclusi il primo e l'ultimo.

Soffermiamoci per capire un pò meglio alcune cose importanti, utili sempre, nascoste in queste apparentemente innocenti righe:

- per avere un singolo elemento di un array usiamo `x[i]`, con i la posizione dell'elemento partendo da zero; si possono usare anche indici negativi per partire dall'ultimo elemento dell'array, come abbiamo fatto in `y[-1]` per prendere appunto l'ultimo elemento di `y`;
- per poter scegliere alcuni elementi consecutivi di un array è possibile usare una *slice* : con `x[i:j]` creiamo un array con tutti gli elementi di `x` che vanno da quello di posto `i` a quello di posto `j-1`;
- mentre con `trap =` assegniamo alla variabile un valore (quindi se `trap` non esiste la creiamo, se contiene già un valore, questo viene sovrascritto), con `trap +=` sommiamo al valore già contenuto in `trap` il valore che segue.
- la funzione `abs(num)` calcola il valore assoluto di num; la funzione `sum(arr)` restituisce la somma di tutti gli elementi del vettore `arr`

Ora in maniera del tutto analoga, partiamo dalla definizione matematica del metodo di Simpson:

$$I = \frac{h}{3}\left(f_1 + f_{N+1}\right) + \frac{h}{3}\sum_{i=2}^{N} 4f_{2i} + \frac{h}{3}\sum_{i=1}^{N} 2f_{2i+1}$$

e scriviamo il codice relativo:

```
simp = y[0] + y[-1]
simp += 4*sum(y[1:-1:2]) + 2*sum(y[2:-1:2])
simp *= h/3
```

Notare l'uso avanzato delle slice per poter selezionare gli elementi di posto pari e dispari:

- con `y[1:-1:2]` prendiamo dal secondo all'ultimo (escluso) saltando di due in due, quindi gli elementi di posto pari;
- con `y[2:-1:2]` prendiamo dal terzo all'ultimo (escluso) saltando di due in due, quindi gli elementi di posto dispari;

Torniamo al nostro problema: l'integrale della funzione scelta. Dopo aver eseguito il codice visto sin ora, dentro le variabili trap e simp abbiamo salvato il valore dell'integrale calcolato con i due metodi. Poichè conosciamo il valore esatto dell'integrale pari a 4.4, possiamo fare una prima verifica della bontà del nostro codice e del metodo usato:

```
esatto = 4.4
print "Valore esatto, Trapezoidi, Simpson: ", \
  esatto, trap, simp
```

Potremo ora ripetere la serie di comandi precedenti aumentando però il numero di intervallini con $N = 1000$.

Osserviamo che il metodo di Simpson è notevolmente più preciso di quello dei Trapezoidi.

Vediamo ora come fare un analisi più estesa dell'andamento dell'errore dei due metodi in funzione del numero N di intervallini. L'idea di base è eseguire le righe di codice precedenti per valori di N compresi in un certo intervallo di valori. Ogni volta che un nuovo valore dell'integrale viene calcolato, salviamo in un elemento di un array l'errore, per entrambe i metodi. Questo lo si può fare in automatico utilizzando un ciclo `for`, in questo modo:

```
esatto = 4.4
Nmax=1000
trap_err=zeros((Nmax-1)/2.)
simp_err=zeros((Nmax-1)/2.)
i=0

for N in range(2,Nmax,2):
  x = linspace(0,2,N+1)
  y = x**4 - 2*x + 1
  h = abs( x[1] - x[0] )
  trap = h* ( 0.5*y[0] + 0.5*y[-1] + sum(y[1:-1]) )
  simp = h/3 *( y[0] + y[-1] + 4*sum(y[1:-1:2]) \
    + 2*sum(y[2:-1:2]) )

  trap_err[i] = abs(trap - esatto)
  simp_err[i] = abs(simp - esatto)
  i=i+1
```

Oltre al codice già visto troviamo delle novità:

- la funzione `zeros(num)` genera un array con `num` elementi di valore zero; `trap_err` e `simp_err` saranno dunque i vettori dove salveremo gli errori dei due metodi per ogni N scelto;
- la funzione `range(i,j,k)` è spesso usata nei cicli `for` per generare una sequenza di valori nell'intervallo $[i, j-1]$ con passo k
- il ciclo `for` esegue ripetutamente tutte le righe indentate e ad ogni ripetizione la variabile `N` assume i valori della sequenza fornita da `range`;
- il numero di volte che vengono eseguite è dato dalla lunghezza del sequenza fornita da `range`, pari a `(Nmax-1)/2.`, valore con cui abbiamo dimensionato gli array che conterranno gli errori;
- in generale, per scrivere un valore in uno specifico elemento di posto i di un array facciamo come nel codice sopra: `trap_err[i] =`.

Ora che abbiamo degli array contenenti gli errori dei due metodi per un numero crescente di intervallini, potremo farne il grafico in questo modo:

```
N_arr = range(2,Nmax,2)
loglog(N_arr, trap_err)
loglog(N_arr, simp_err)
```

Creiamo prima un array con la sequenza di valori usata nel ciclo `for` e poi usiamo la funzione `loglog(x_arr,y_arr)` che crea un grafico in scala bi-logaritmica, con `x_arr` e `y_arr` i vettori delle coordinate x,y dei punti.

Il grafico che otteniamo è del tutto analogo a quello mostrato in Fig.1.5. Una differenza significativa è che, poichè si sono usati numeri rappresentati a 64 bit, l'errore di round-off non è ancora visibile a questi valori di N relativamente piccoli. Andando, per esempio, a N=10000, esso puntualmente ricompare. In effetti l'errore minimo è circa 10^{-12}, mentre ci aspettiamo (come discusso in precedenza) un errore dell'ordine di 10-100 volte la precisione di macchina, quindi intorno a 10^{-14}. Come notiamo la pendenza delle due curve è diversa, e dovrebbe darci l'ordine dell'errore commesso con i due metodi. È quindi interessante verificarlo, calcolandola in questo modo:

```
deltaY = log(trap_err[350])-log(trap_err[300])
deltaX = log(N_arr[350])-log(N_arr[300])
slope_trap = deltaY / deltaX
deltaY = log(simp_err[350])-log(simp_err[300])
deltaX = log(N_arr[350])-log(N_arr[300])
slope_simp = deltaY / deltaX

print slope_simp, slope_trap
```

La pendenza la calcoliamo geometricamente scegliendo opportunamente due punti per ogni grafico e facendo il rapporto tra la differenza delle coordinate y e quella delle coordinate x. Per esempio abbiamo scelto gli elementi 300 e 350 degli array `trap_err`, `simp_err` e `N_arr` e, poichè la relazione è lineare in scala Log-Log, ne abbiamo fatto il logaritmo mediante la funzione `log()`.

Le pendenze calcolate (−4.0001382837,−1.99999952468) sono effettivamente in ottimo accordo con quello che sappiamo essere l'ordine di grandezza dell'errore dei due metodi.

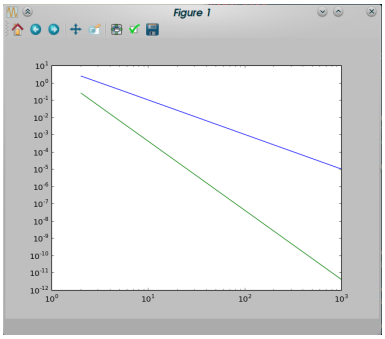
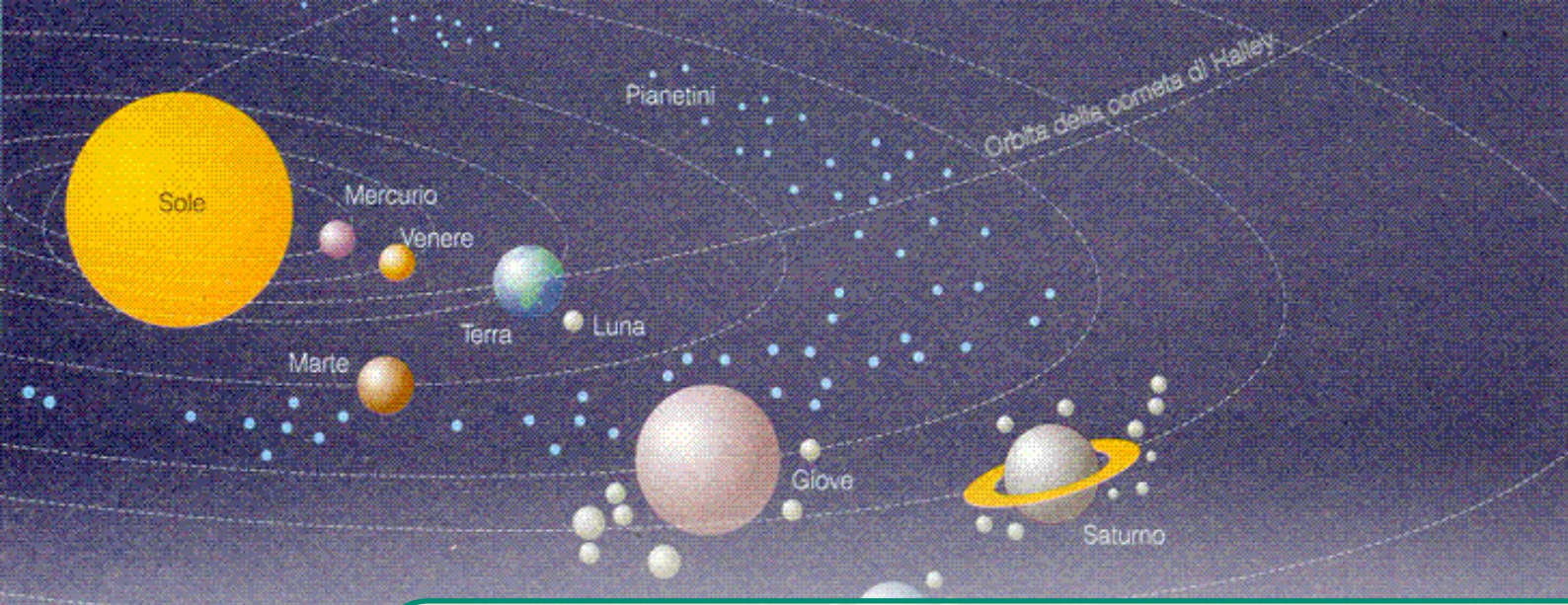


Figura 1.4: Errore vs N in scala log-log per i due metodi di integrazione Trapezoidi e Simpson. Poichè si sono usati numeri rappresentati a 64 bit, l'errore di round-off non è ancora visibile a questi valori di N relativamente piccoli. Andando, per esempio, a N=10000, esso puntualmente ricompare.
Per abbellire il grafico si veda la sezione 1.22



2. A - Equazioni Differenziali Ordinarie

2.1 *Generalità*

Le equazioni differenziali sono relazioni che legano una funzione incognita alle sue derivate; come si vedrà, in aggiunta all'equazione, sono essenziali le condizioni al contorno. Si distinguono due grandi gruppi: le equazioni ordinarie, che coinvolgono una (o più di una) funzione di una variabile e le sue derivate; e le equazioni a derivate parziali, in cui la funzione incognita dipende da più variabili e quindi sono coinvolte le derivate parziali. In questa Sezione ci occupiamo di equazioni ordinarie (ODE); delle PDE ci occuperemo in Sez.4.

Un elemento tassonomico importante è l'ordine dell'equazione, cioè l'ordine massimo delle derivate presenti. Ad esempio

$$\frac{df(x)}{dx} = -f(x) \tag{2.1}$$

è una equazione del primo ordine; invece,

$$m\frac{d^2x}{dt^2} = G(x,t) \tag{2.2}$$

è una equazione del secondo ordine, che riconosciamo essere l'equazione di Newton per una massa m che si muove in una dimensione ed è soggetta a una forza G. Ci occuperemo essenzialmente solo di equazioni del tipo di Eq.2.2, molto frequenti e rilevanti in fisica e contenenti tutti gli aspetti che vogliamo trattare (al nostro livello).

Come menzionato, le condizioni al contorno sono parte integrante del problema. Risolvendo una ODE otteniamo in generale una famiglia di funzioni. Per ottenere una specifica soluzione serve fornire una o più condizioni che la identifichino. Eq.2.1, ad esempio, ha soluzione f=C exp($-x$); C è una costante arbitraria, che può essere determinata imponendo il valore di f in un punto del dominio di definizione; p.es. $f(0)$=1 seleziona la soluzione f=exp($-x$). Le ODE del secondo ordine, come Eq.2.2, hanno invece bisogno di due condizioni. Qui

si distingue tra condizioni al bordo e condizioni iniziali; la scelta dipende dal problema in questione. Per l'equazione di Schrödinger stazionaria unidimensionale

$$[-\frac{d^2}{dx^2} + v(x) - E]\,\psi(x) = 0 \tag{2.3}$$

(il primo termine è l'energia cinetica p^2 tradotta in operatore secondo la procedura di "prima quantizzazione", v l'operatore moltiplicativo di energia potenziale ed E la costante del moto "energia"), è frequente che le due condizioni richieste sono ottenute specificando la soluzione in due punti (tipicamente, i bordi) del dominio; la soluzione è la forma spaziale della funzione d'onda da utilizzare nell'interpretazione di Born come ampiezza di probabilità di osservare una particella in x. La scelta di condizioni al bordo porta a utilizzare metodi specializzati, come il cosiddetto "shooting", di cui probabilmente non ci occuperemo. L'equazione di Newton (Eq.2.2), invece, descrive l'evoluzione dinamica –il moto– di una particella classica soggetta a una forza. La particella parte da una data posizione (valore iniziale della funzione) con una data velocità (=valore della derivata prima), e l'integrazione dell'equazione produce la traiettoria a tempi successivi. La generazione di una traiettoria sufficientemente accurata (secondo opportuni criteri: ad esempio quello di conservare l'energia o altre costanti del moto) sarà il nostro principale obiettivo.

Come discutiamo in più dettaglio oltre, una equazione del secondo ordine (ma anche di ordine diverso) ai valori iniziali iniziali può essere integrata direttamente tramite una discretizzazione della derivata seconda come Eq.1.27, oppure, con minori difficoltà numeriche, nella forma di sistema di equazioni accoppiate del primo ordine. Infatti l'equazione è facilmente spacchettata in due equazioni accoppiate del primo ordine, ognuna con le sue condizioni iniziali. Definita la velocità

$$\frac{dx}{dt} = v \tag{2.4}$$

si ha

$$m\frac{d^2x}{dt^2} = G(x,t) \Rightarrow \frac{d(mv)}{dt} = \frac{dp}{dt} = G(x,t). \tag{2.5}$$

Con le condizioni iniziali, ad esempio, $x(0)=x_0$ e $v(0)=v_0$, queste due equazioni del primo ordine sono equivalenti a Eq.2.2. Eq.2.4 e l'ultima eguaglianza in Eq.2.5 sono note in meccanica analitica come equazioni di Hamilton, e sono (chiaramente) equivalenti a quella di Newton.

2.2 *I metodi di Eulero*

Un buon motivo per integrare numericamente le equazioni del primo ordine piuttosto che quelle del secondo ordine è che a parità di precisione gli schemi di integrazione di equazioni di ordine minore richiedono, per predire il punto successivo, meno operazioni e la conoscenza di meno punti della traiettoria già ottenuta – in casi fortunati, solo il punto corrente (l'ultimo della traiettoria finora).

Consideriamo allora i metodi di Eulero ed Eulero-Cromer. Il primo è decisamente sconsigliato per l'uso, ma interessante didatticamente; in seguito formuleremo e useremo altri metodi più solidi. Le equazioni da risolvere (ponendo F=G/m) sono

$$\frac{dv}{dt} = F; \quad \frac{dx}{dt} = v. \tag{2.6}$$

(Abbiamo assunto che m sia costante.) Usando la formula forward-difference per la derivata prima, Eq.1.22, su un intervallino di tempo τ, detto *time step*, abbiamo

$$\begin{aligned} \frac{v(t+\tau) - v(t)}{\tau} + o(\tau) &= F(x,t) \\ \frac{x(t+\tau) - x(t)}{\tau} + o(\tau) &= v(t). \end{aligned} \tag{2.7}$$

Discretizziamo come in precedenza il problema ponendo (per n=0, 1, ...)

$$t_n = t_0 + n\tau; \quad v(t_n) = v_n \tag{2.8}$$

$$x(t_n) = x_n; \quad F(t_n, x_n) = F_n \tag{2.9}$$

Riarrangiando otteniamo lo schema di integrazione di Eulero,

$$\begin{aligned} v_{n+1} &= v_n + \tau F_n \\ x_{n+1} &= x_n + \tau v_n, \end{aligned} \tag{2.10}$$

che date posizione e velocità al tempo attuale (n) predice la velocità e posizione al tempo successivo (n+1), con le condizioni iniziali

$$v_0 = v(t_0) = \text{costante}, \quad x_0 = x(t_0) = \text{costante}.$$

Naturalmente l'indice usato nei codici potrà partire da 0 o da 1 a seconda dei gusti o delle richieste del linguaggio.

Abbiamo omesso il termine di errore locale, che è o(τ^2) per passo temporale. Ovviamente l'errore globale sulla traiettoria sarà maggiore. Per N_τ passi temporali τ il tempo totale di simulazione sarà T=$N_\tau\tau$ (e più piccolo è τ, più grande dovrà essere N_τ per ottenere lo stesso tempo di simulazione). Se l'errore locale è $o(\tau^n)$, l'errore globale sulla traiettoria sarà N_τ $o(\tau^n)$=T $o(\tau^{n-1})$. Nel caso di Eulero, perciò, l'errore globale è lineare in τ. È grave? Dipende. (Eulero, come vedremo, ha comunque altri seri problemi.) Come ricorderete, il termine di errore quadratico dovuto al troncamento della serie di potenze della funzione approssimata è proporzionale alla derivata seconda. Quindi nel caso presente (a meno di fattori dell'ordine di 1)

$$\text{errore} \sim h^2 f'' \equiv \tau^2 a.$$

Se a=g e scegliamo τ=0.1 s, l'errore è dell'ordine di 0.1 m per passo; se vogliamo seguire il moto per T=1 s dobbiamo fare N_τ=10 steps, e quindi l'errore globale sulla posizione è circa 1 m. Se questo è accettabile per i nostri scopi, bene – altrimenti devo ridurre τ e aumentare N_τ. Questo ovviamente ha i suoi limiti: lo step dev'essere (parecchio)

maggiore della precisione di macchina, e il tempo di esecuzione non deve essere eccessivo. L'altra cosa da fare è scegliere metodi con minor errore locale. Se quest'ultimo fosse $o(\tau^4)$, nell'esempio precedente il nostro errore globale scenderebbe a 1 mm.

Il metodo di Eulero-Cromer è in effetti una semplicissima modifica di Eulero:

$$\begin{aligned} v_{n+1} &= v_n + \tau F_n \\ x_{n+1} &= x_n + \tau v_{n+1}. \end{aligned} \tag{2.11}$$

Mentre in Eulero l'ordine del calcolo è irrilevante, qui la velocità va calcolata prima. Mentre il metodo di Eulero usa la forward difference per ambedue le derivate prime, la seconda equazione del metodo di Eulero-Cromer deriva dalla backward difference

$$v_{n+1} = \frac{x_{n+1} - x_n}{\tau}. \tag{2.12}$$

L'errore locale sulla traiettoria è $o(\tau^3)$, dato che sostituendo la prima equazione nella seconda si ha

$$x_{n+1} = x_n + \tau v_n + \tau^2 F_n \tag{2.13}$$

Ne deriva una cancellazione d'errore che rende E-Cromer molto migliore di Eulero (anche se l'errore locale nella prima equazione è sempre $o(\tau^2)$).

2.3 *Algoritmo di Verlet*

Come vedremo tra poco con qualche applicazione, Eulero non può essere usato in pratica. E-Cromer è già meglio, ma in genere (e in particolare per sistemi a tante coordinate) è indispensabile utilizzare metodi migliori. Nei testi di analisi numerica, il passo successivo è tipicamente analizzare metodi con a) errore locale migliore e omogeneo nelle diverse equazioni coinvolte, e b) con time step adattivo. Il punto b) è sofisticato per i nostri scopi e lo considereremo brevemente in seguito. Il punto a) è tipicamente sviluppato descrivendo gli algoritmi noti come Runge-Kutta (tipicamente quello del quarto ordine, cioè
[1] con errore globale $o(\tau^4)$, vedi dettagli in NR [1]). Qui scegliamo un bordo diverso: l'algoritmo di Verlet, introdotto da Loup Verlet nel contesto della dinamica molecolare a molte particelle nel 1969, eccelle
[2] nel contesto della dinamica classica [2]. Come vedremo, esso deriva direttamente dal principio di minima azione di Hamilton, il che garantisce (entro certi limiti) la "verità" delle traiettorie dinamiche che genera; inoltre è un algoritmo simplettico, ovvero conserva il volume dello spazio delle fasi del sistema dinamico in esame, e cioè (come sappiamo dalla meccanica) l'energia.

L'idea è utilizzare direttamente la formula per la derivata seconda, Eq.1.27, nella

$$\frac{d^2x}{dt^2} = a = F \tag{2.14}$$

(sempre nella convenzione che F sia la forza diviso la massa). Sostituendo si ottiene

$$x_{n+1} = 2x_n - x_{n-1} + \tau^2 F + o(\tau^4) \tag{2.15}$$

che genera la traiettoria al tempo n+1 a partire dai valori della coordinata ai tempi n e n–1, e della forza al tempo n. Come si vede, la procedura non è auto-avviante, nel senso che la condizione iniziale x=x_0 non è sufficiente a predire x_1, e così via. Per iniziare con Verlet a partire da x_2, x_1 potrebbe essere generato con un passo di Eulero-Cromer dalla condizione iniziale x_0. Le velocità non servono a generare la traiettoria, ma possono essere ottenute p.es. come

$$v_n = \frac{x_{n+1} + x_{n-1}}{2\tau} + o(\tau^2). \tag{2.16}$$

Le espressioni per v e x sono asimmetriche, e in aggiunta l'errore locale è diverso per x e v, ambedue caratteristiche non desiderabili. Notiamo che si può ottenere Eq.2.15 anche sviluppando la traiettoria al terzo ordine per un tempo immediatamente successivo o precedente al corrente:

$$\begin{aligned} x_{n+1} &= x_n + x'_n\tau + x''_n\tau^2/2 + x'''_n\tau^3/6 + o(\tau^4) \\ x_{n-1} &= x_n - x'_n\tau + x''_n\tau^2/2 - x'''_n\tau^3/6 + o(\tau^4) \end{aligned}$$

sommando le quali si ottiene subito Verlet. Si può poi mostrare che una espressione equivalente a Verlet è

$$\begin{aligned} x_{n+1} &= x_n + v_n\tau + F_n\frac{\tau^2}{2} \\ v_{n+1} &= v_n + \frac{\tau}{2}(F_n + F_{n+1}), \end{aligned} \tag{2.17}$$

il cosidetto "velocity-Verlet", l'algoritmo standard della dinamica molecolare, e quello che useremo prevalentemente. Più semplice e simmetrico del Verlet originale, il v-V è simplettico, esplicito, e self-starting. Può tuttavia presentare delle difficoltà se la forza dipende dalla velocità (ad esempio nel moto smorzato viscoso). Per contro, è semplice implementare passi temporali variabili nel caso di forze centrali coulombiane o gravitazionali.

2.4 *Alle basi della meccanica: Verlet e minima azione*

L'algoritmo di Verlet, nonostante l'apparenza di un trucco di discretizzazione, è connesso alle più profondi principi di base della meccanica. In questa Sezione, mostriamo che esso deriva dalla versione discretizzata del principio di minima azione, o principio di Hamilton. Usiamo la meccanica lagrangiana invece di quella newtoniana, ovvero partiamo dal funzionale delle coordinate e velocità detto Lagrangiana, la differenza dell'energia cinetica e di quella potenziale. Per una particella in una dimensione, la Lagrangiana è

$$\mathcal{L}(x(t), \dot{x}(t)) = \frac{m\dot{x}^2}{2} - U(x). \tag{2.18}$$

Si definisce poi l'azione

$$S \equiv \int_{t_1}^{t_2} dt\, \mathcal{L}(x(t)), \tag{2.19}$$

che è a sua volta un funzionale della traiettoria, nel senso che il valore dell'integrale su tutti i tempi cambia al variare della funzione x che entra in $\mathcal{L}$. Il principio di minima azione di Hamilton (o di Maupertuis) afferma che la vera traiettoria $X(t)$ di un sistema meccanico, tra tutte quelle possibili $x(t)$ nell'intervallo (t_0,t_1) per cui $x(t_0)=x_0$ e $x(t_1)=x_1$, è quella che rende stazionaria l'azione. Se cioè consideriamo una traiettoria che devii poco da quella vera, $x=X+\delta x(t)$, con $\delta x(t_0)=\delta x(t_1)=0$, si ha che la variazione del funzionale rispetto alla traiettoria è nulla:

$$\frac{\delta S}{\delta x(t)} = 0, \quad \forall\, t \tag{2.20}$$

dove la derivata è di un tipo particolare detto derivata funzionale (di cui qui non ci interessa il dettaglio). Si dimostra infatti che la stazionarietà dell'azione implica che la traiettoria obbedisce le equazioni di Lagrange

$$\frac{d}{dt}\frac{d\mathcal{L}}{d\dot{q}} - \frac{d\mathcal{L}}{dq} = 0, \tag{2.21}$$

dove q e $\dot{q}$ sono le coordinate generalizzate e le loro derivate temporali. Per la Lagrangiana Eq.2.18,

$$\begin{aligned} \frac{d\mathcal{L}}{d\dot{q}} &= m\dot{x} \\ \frac{d}{dt}\frac{d\mathcal{L}}{d\dot{q}} &= m\ddot{x} \\ -\frac{d\mathcal{L}}{dq} &= \frac{dU}{dx} \end{aligned}$$

e quindi Eq.2.21 diventa

$$m\ddot{x} + \frac{dU}{dx} = 0 \tag{2.22}$$

cioè l'equazione di Newton

$$ma = -\frac{dU}{dx} \equiv G. \tag{2.23}$$

Vediamo dunque cosa succede imponendo il principio di minima azione alla versione discretizzata dell'azione

$$S_{\text{discr}} = \tau \sum_{n=0}^{N_\tau} \mathcal{L}(t_n) \tag{2.24}$$

(come al solito n è l'indice "orologio": $t=t_0+n\tau$). La nostra Lagrangiana Eq.2.18 discretizzata è

$$\mathcal{L}(t_n) = \frac{m}{2}\frac{(x_{n+1}-x_n)^2}{\tau^2} - U(x_n), \tag{2.25}$$

dove si è usata la derivata prima forward-difference. Quindi

$$S_{\text{discr}} = \sum_{n=0}^{N_\tau} \left(\frac{m(x_{n+1}-x_n)^2}{2\tau} - U(x_n)\tau \right). \tag{2.26}$$

Per ottenere l'estremo, annulliamo la derivata dell'azione rispetto alla coordinata

$$\frac{\partial S_{\text{discr}}}{\partial x_n} = 0 = \frac{\partial}{\partial x_n}\left[\sum_{n=0}^{N_\tau}\left(\frac{m(x_{n+1}-x_n)^2}{2\tau} - U(x_n)\tau\right)\right], \quad \forall\, n \qquad (2.27)$$

dove usiamo la derivata parziale perchè S_{discr} può dipendere da altre variabili. Si intuisce che questo produrrà uno "shadowing" punto per punto della traiettoria vera da parte di quella discretizzata (Fig.2.1). Nella derivata sopravvivono solo i due termini della somma dove compare x_n, cioè l'n-esimo e l'$(n-1)$-esimo, e quindi

$$m\frac{(x_n - x_{n-1}) - (x_{n+1} - x_n)}{\tau} - \frac{\partial U(x_n)}{\partial x_n}\tau = 0. \qquad (2.28)$$

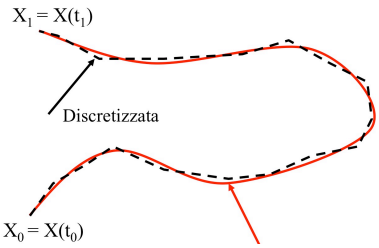


Figura 2.1: Shadowing della traiettoria vera da parte di quella discretizzata.

Riarrangiando si ottiene

$$\begin{aligned} m\frac{2x_n - x_{n+1} - x_{n-1}}{\tau} + G\tau &= 0 \\ 2x_n - x_{n+1} - x_{n-1} + F\tau^2 &= 0 \end{aligned}$$

e infine (F è sempre la forza diviso la massa)

$$x_{n+1} = 2x_n - x_{n-1} + F\tau^2$$

cioè l'algoritmo di Verlet – che deriva dunque dal principio di azione estremale. Le traiettorie generate da Verlet soddisfano le condizioni temporali al contorno della traiettoria reale. Ne risulta che Verlet è temporalmente reversibile, conserva il volume dello spazio di fase (cioè è simplettico), e di conseguenza conserva con buona accuratezza l'energia totale, ovvero non soffre di apprezzabile drift in energia.

Una ulteriore considerazione interessante riguarda i sistemi a molti corpi. La dinamica di un sistema a molti corpi non-patologico è caotica, ovvero mostra sensibilità estrema alle condizioni iniziali, ovvero è soggetta all'instabilità di Lyapunov: traiettorie con condizioni iniziali diverse, ma vicinissime, divergono esponenzialmente per tempi lunghi (Fig.2.2).

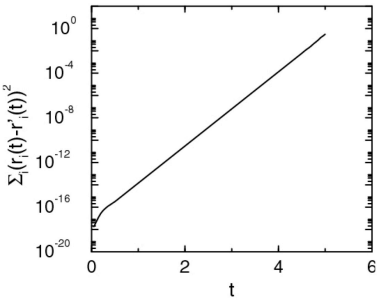


Figura 2.2: Divergenza delle traiettorie di un sistema many-body (instabilità di Lyapunov).

Lo "shadow theorem" (in realtà un'ipotesi) afferma che algoritmi "buoni" generano traiettorie "vicine" (come ombre, appunto) a quelle reali. Le traiettorie di Verlet, essendo associate all'azione estremale, sono buone traiettorie "shadow" anche in un sistema many-body. Questo è uno dei motivi della popolarità dell'algoritmo in dinamica molecolare.

2.5 *Un esempio: il pendolo anarmonico*

Ci sono infinite applicazioni possibili delle tecniche appena discusse. Alcune di esse sono riassunte nel seguito e sviluppate in notebooks python separati. Qui consideriamo esplicitamente il pendolo anarmonico. Verificheremo anche (Sez.2.5.2) che Eulero non è utile in pratica altro che per scopi dimostrativi (di quanto, appunto, un algoritmo possa andare male). Sostanzialmente vogliamo integrare l'Eq.1.2 numericamente, con condizioni iniziali sulla posizione θ=θ_0 e velocità

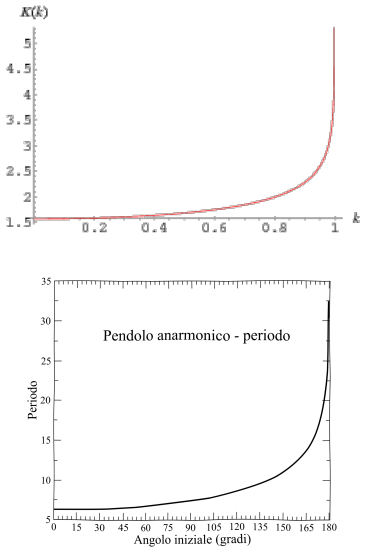


Figura 2.3: Sopra, l'integrale ellittico del primo tipo. Sotto, il periodo del pendolo anarmonico in unità di $\sqrt{\ell/g}$ calcolato con l'algoritmo vV. Il limite per piccoli angoli è 2π.

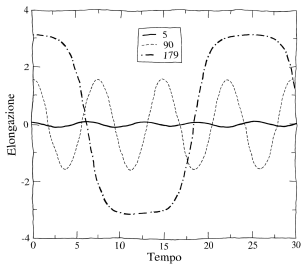


Figura 2.4: Le traiettorie del pendolo anarmonico per θ_0=5°, 90°, 179° generate con vV.

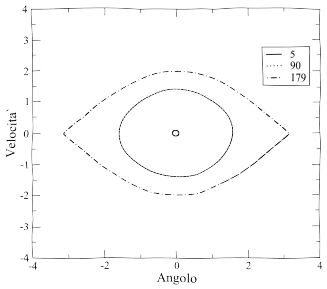


Figura 2.5: Traiettorie di fase del pendolo anarmonico per θ_0=5°, 90°, 179°.

angolare ω=ω_0. Come visto in Sez.1, per piccole elongazioni il pendolo è un oscillatore armonico, e ha quindi frequenza indipendente dalle condizioni iniziali; fuori dal regime armonico, cioè per elongazioni grandi, la frequenza, e quindi il periodo, rimangono costanti del moto, ma il loro valore dipende dalle condizioni iniziali.

L'energia è una costante del moto che, supponendo di partire da una elongazione θ_0 con velocità angolare ω_0=0, vale

$$E = E_{\text{cin}} + E_{\text{pot}} = \frac{m\ell^2\omega^2}{2} - mg\ell\cos\theta = -mg\ell\cos\theta_0. \tag{2.29}$$

Risolvendo per ω^2 si ottiene

$$\left(\frac{d\theta}{dt}\right)^2 \equiv \omega^2 = \frac{2g}{\ell}(\cos\theta - \cos\theta_0) \tag{2.30}$$

e quindi

$$dt = \frac{d\theta}{\sqrt{2g(\cos\theta - \cos\theta_0)/\ell}}. \tag{2.31}$$

Ora integriamo sull'angolo tra elongazione nulla ed elongazione massima, e notiamo che nel suo periodo il pendolo percorre quattro volte questo angolo; dunque il periodo è

$$T = 4\int_0^{\theta_0} dt = 4\sqrt{\frac{\ell}{2g}}\int_0^{\theta_0}\frac{d\theta}{\sqrt{(\cos\theta - \cos\theta_0)}} \tag{2.32}$$

L'integrale è noto come integrale ellittico completo del primo tipo, che per piccole elongazioni fornisce

$$T \simeq 2\pi\sqrt{\frac{\ell}{g}}\,(1 + \frac{1}{16}\theta_0^2 + \dots) \tag{2.33}$$

che tende quadraticamente al limite armonico per $\theta_0 \to 0$. In Fig.2.3 si nota che l'integrale ellittico e il periodo calcolato con l'algoritmo vV (con τ=0.001) sono, in effetti, visivamente uguali.

Fisicamente, all'aumentare dell'elongazione iniziale il periodo T aumenta, e $T \to \infty$ per il punto di equilibrio instabile $\theta_0 \to \pi$. Per analizzare il sistema in maggior dettaglio, esaminiamo la traiettoria $\theta(t)$, la traiettoria di fase $\omega(\theta)$, e le componenti cinetica e potenziale dell'energia in funzione del tempo, per tre angoli iniziali corrispondenti ai casi armonico, anarmonico, e anarmonico estremo (θ_0=5°, 90°, 179°). Notiamo che abbiamo scelto per semplicità g=ℓ=m=1. Angoli e velocità angolari, dentro il codice, sono convertite in rad e rad/s.

Iniziamo con le traiettorie, che sono mostrate in Fig.2.4. A sinistra, sono evidenti l'ampiezza e il periodo crescenti da angoli piccoli a angoli grandi; in quella a destra, dove si sono riscalate le curve per confrontarne la forma, si nota il diverso andamento della legge oraria, con una traiettoria più "squadrata" ai punti stazionari man mano che l'angolo aumenta. Torneremo su questo punto nel contesto dell'oscillatore anarmonico con potenziale di tipo potenza crescente.

Le traiettorie di fase sono mostrate in Fig.2.5. Poichè il moto del sistema è unidimensionale, la traiettoria di fase è racchiusa in uno

spazio bidimensionale. Essendo il moto periodico, tale traiettoria è chiusa. Per il caso armonico, la traiettoria di fase è un cerchio di equazione $R^2=x^2+p^2 \propto E_{tot}$; per i casi anarmonici, rimane vero che l'area inclusa nella traiettoria è proporzionale all'energia totale. Come si vede, la traiettoria non accede a tutti i punti dello spazio della fasi, ed è quindi non-ergodica. La traiettoria armonica è un cerchio di piccola area, data la piccola energia totale; le altre curve hanno area (energia totale) corrispondentemente sempre maggiore. A elongazione 90° la traiettoria è visibilmente ellissoidale, dato che l'angolo a cui la velocità si annulla è maggiore. A 179°, dove il pendolo parte quasi verticale, la curva di fase ha quasi una cuspide ad elongazione massima e ω=0; in realtà un ingrandimento mostra che la curva di fase ha, in quel punto, curvatura finita, e la cuspide esisterebbe solo come caso limite per $\theta_0 \to 180°$.

Come si vede in Fig.2.6, che mostra la traiettorie di fase e del moto per θ_0=179° e velocità iniziale non nulla, quello menzionato è il punto di "comunicazione" tra branche di una traiettoria di fase illimitata, dato che il pendolo lascia il bacino intorno a θ=0 da cui parte, per ruotare sempre nello stesso verso attorno al punto di sospensione.

Veniamo ora al comportamento delle componenti dell'energia. In Fig.2.7 mostriamo, a sinistra, l'energia cinetica e potenziale per il caso armonico (θ_0=5°), e a destra le stesse quantità per il caso anarmonico estremo (θ_0=179°). Nel caso armonico, l'energia totale è $-\cos(5°)=-0.9962$, ed è inizialmente tutta potenziale (si è scelta velocità iniziale nulla in ambedue i casi). Poi, le due componenti dell'energia oscillano in controfase con comportamente pressochè identico; integrandole su un periodo, si verifica che il teorema del viriale $\langle E_{kin}\rangle=\langle E_{pot}\rangle$ è ben soddisfatto.

Nel caso anarmonico l'energia totale è $-\cos(179°)=0.9998$; all'inizio del moto essa è quasi tutta potenziale, e tale rimane a lungo (1/4 di periodo) dato che la velocità aumenta lentamente all'inizio del moto. L'energia cinetica aumenta bruscamente al passaggio a θ=0, ma le regioni di alta e bassa velocità sono del tutto asimmetriche. Infine, in ambedue i casi, la linea pressochè sovrapposta all'asse dei tempi è la differenza dell'energia totale calcolata ed esatta. In Fig.2.8 è mostrata (in alto) una visione ingrandita dell'errore relativo dell'energia totale calcolata rispetto a quella esatta per θ_0=179° ("grandi" fluttuazioni) e θ_0=5° (piccole fluttuazioni). Va notata la scala: l'errore relativo è sempre minuscolo anche nel caso estremo della grande anarmonicità. (Il calcolo è fatto in doppia precisione a 32 bit, $\epsilon_m \sim 10^{-14}$.) Questo conferma che l'algoritmo vV conserva bene l'energia. La figura di destra mostra invece l'errore per due simulazioni della durata di 100 unità di tempo, una (in nero) di 10^6 passi e $\tau=10^{-4}$ e l'altra (in rosso, grigio in b/w) di 10^5 passi e $\tau=10^{-3}$. Si nota intanto che l'errore è piatto nella regione tra i massimi di energia cinetica, dove ha invece degli spike transienti dovuti alla rapida variazione della velocità. Questi spike portano all'accumulo di errore su tempi lunghi, benchè solo al livello di 10 ppm. È poi interessante notare che la riduzione del time step di un fattore 10 non porta particolari vantaggi (almeno

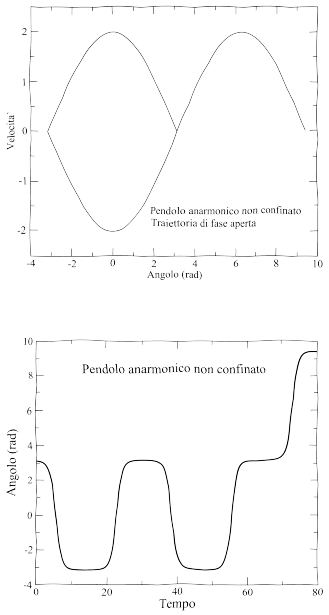


Figura 2.6: Traiettoria di fase (sinistra) e legge del moto (destra) del pendolo anarmonico per θ_0=179° e velocità iniziale non nulla ω_0=–0.0124 rad/s (un grado al secondo). Il pendolo fa un singolo periodo intorno a θ=0, poi inizia a ruotare (sempre nello stesso verso, deciso dal segno della velocità iniziale) attorno al punto di sospensione.

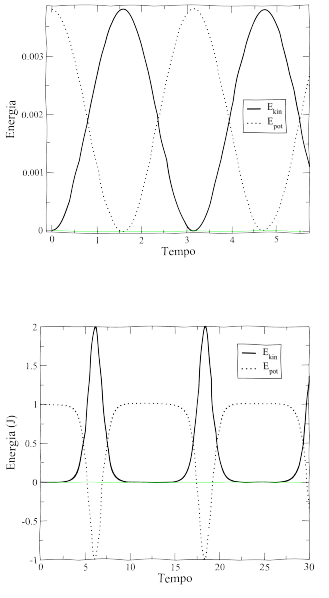


Figura 2.7: Energia cinetica e potenziale per il pendolo armonico (a sinistra, θ_0=5°, ω_0=0) e anarmonico (a destraθ_0=179°, ω_0=0). L'energia potenziale armonica è la deviazione rispetto all'equilibrio.

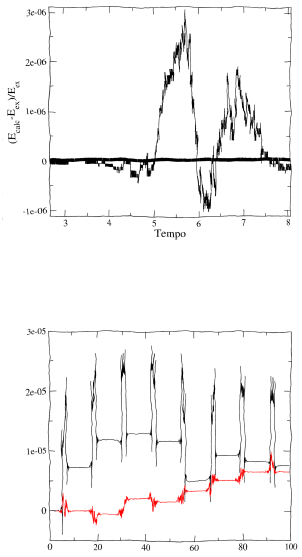


Figura 2.8: Sopra, visione ingrandita dell'errore relativo dell'energia totale calcolata rispetto a quella esatta per θ_0=179° ("grandi" fluttuazioni) e θ_0=5° (piccole fluttuazioni) su un breve intervallo di tempo. Il time step è 0.001. Sotto, errore su un intervallo lungo con time step 0.0001 (nero) e 0.001 (rosso, grigio in b/w).

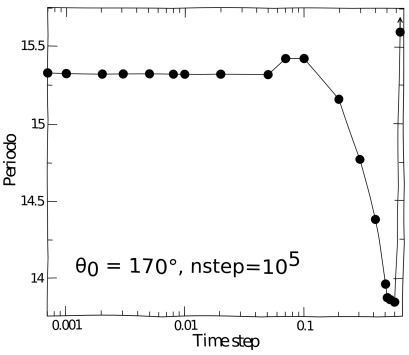


Figura 2.9: Periodo calcolato per il pendolo anarmonico con θ_0=170° in funzione del time step τ. Sopra $\tau\sim$0.6 il periodo diventa erratico e molto grande.

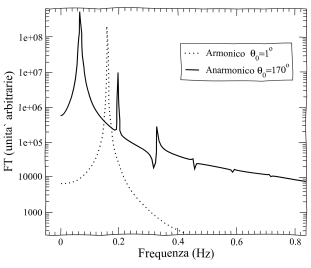


Figura 2.10: Trasformata di Fourier della traiettoria del pendolo per θ_0=170° e θ_0=1°. Si vedono, nella prima, segnali corrispondenti alle armoniche dalla 0 alla 5.

in questa finestra temporale); sembrerebbe che si stia avvicinando il limite della performance di vV. Infine, è probabile che un algoritmo a passo variabile (breve dove le velocità variano molto, più lungo altrove) possa migliorare ulteriormente l'errore.

2.5.1 *Considerazioni sul passo temporale*

Come discusso in precedenza, il passo temporale τ è un parametro aggiustabile importante in qualunque simulazione del tipo discusso. Passo piccolo implica grandi tempi di simulazione, e viceversa passo grande rende più probabili inacuratezza e instabilità. Ogni metodo di integrazione delle equazioni ha la sua ricetta per il passo temporale. È anche possibile, ma con un overhead computazionale, rendere variabile (e quindi potenzialmente adattabile alla specifica fase della dinamica) il passo temporale; in presenza di potenziali centrali di tipo coulombiano o gravitazionale questo è particolarmente semplice (discussione più oltre).

Nel caso di Verlet, dato il pedigree prestigioso dell'algoritmo e la sua alta accuratezza, si possono usare time step piuttosto grandi. Ne facciamo una breve analisi sempre per il pendolo anarmonico. In Fig.2.9 è mostrato il periodo del pendolo anarmonico per θ_0=170° in funzione del τ usato in vV. Il valore per piccolo τ è T=15.3271±0.0003. Si vede che fino a circa τ=0.05 l'errore è sotto lo 0.01%. Il massimo time step è primariamente determinato empiricamente in base alla richiesta che la dinamica sia stabile e che il drift dell'energia totale conservata sia sotto una soglia. Come orientamento, è anche utile ricordare che, in base all'analisi di Fourier (Cap.5), la risoluzione temporale minima necessaria a descrivere la traiettoria, cioè il massimo time step $\tau_{\min}$, è legata alla massima frequenza nello spettro armonico (supponendo che ne abbia uno) del sistema dalla relazione di Nyquist

$$\tau < \tau_{\min} = \frac{1}{2f_{\max}}.$$

Per una oscillazione armonica con m=g=ℓ=1, che ha $f_{\max}$=1/(2π)≃0.159 Hz, il time step limite sarebbe τ=π. Nel caso di oscillazioni anarmoniche, la traiettoria è una sovrapposizione di armoniche dispari, con frequenze del tipo (2k+1)f con k=0, 1, 2, ... e f la frequenza minima: il τ massimo ammissibile dovrà quindi essere certamente più piccolo (di un fattore 3, 5, 7, ...) rispetto a quello massimo di campionamento dell'armonica fondamentale.

La Fig.2.9 mostra che il massimo time step che dà risultati plausibili è circa 0.6-0.7, cioè circa 5 volte minore del limite teorico per il pendolo armonico. In effetti lo spettro di Fourier della traiettoria dello stesso pendolo anarmonico (θ_0=170°, confrontato in Fig.2.10 con quella del pendolo armonico) mostra, oltre a una frequenza fondamentale $f\simeq$0.065≃1/T, segnali di armoniche superiori almeno fino a k=5, con frequenza $f_5\simeq$0.72 (cioè appunto circa 2k+1=11 volte quella fondamentale). La f_5 è circa 4.6 volte quella armonica, quindi il time step limite sarà π/4.6∼0.68, in accordo con quanto osservato nella stima del periodo. Naturalmente ci sono forti inaccuratezze anche

a time step minori di quello limite; la traiettoria dev'essere stabile e conservativa, soprattutto per una stima accurata del periodo basata, come questa, sull'osservazione dei tempi di transito all'equilibrio, e questo non è garantito da considerazioni di analisi di Fourier. Anche il pendolo armonico con τ=3 diventa instabile dopo pochi cicli.

2.5.2 *Confronto con Eulero ed E-Cromer*

Abbiamo promesso un confronto di vV con Eulero ed E-Cromer. Consideriamo il pendolo anarmonico con θ_0=170°. Scegliamo un time step τ=0.01 che, essendo abbastanza grande, dovrebbe evidenziare facilmente le difficoltà di Eulero (grosse) e di E-Cromer (molto minori), già per relativamente pochi step.

In Fig.2.11 sono mostrate le energie totali ottenute dai vari metodi. Come si vede, con Eulero il pendolo acquista energia in modo spurio. Con E-Cromer, invece, anche se con errori relativi cospicui (∼1% intorno ai punti di inversione del moto), l'energia fluttua attorno al valore corretto. Questo si spiega con il fatto che (risulta) anche E-Cromer è un algoritmo simplettico. Verlet ha fluttuazioni massime dell'ordine dello 0.001%, dato che oltre ad essere simplettico ha errore globale molto minore di E-Cromer.

Si può valutare il drift relativo dell'energia calcolando la media integrale dell'energia totale (variabile istante per istante, poichè non esattamente conservata) calcolata su tutta simulazione (di durata T), e dividendo per l'energia totale:

$$D \simeq \frac{\langle E_{\rm tot}\rangle}{E_{\rm esatta}}, \quad \langle E_{\rm tot}\rangle = \frac{1}{T}\int_0^T dt\, E_{\rm tot}(t).$$

Sia per Verlet che per E-Cromer il drift relativo è un molto modesto 10^{-5} (calcolato su 10^5 passi). Naturalmente, soprattutto per E-Cromer, questo è in parte dovuto al fatto che il sistema è periodico. Un sistema aperiodico o, a maggior ragione, ergodico verosimilmente causerebbe più problemi a E-Cromer che Verlet. [Per mostrarlo abbiamo provato ad integrare le equazioni del moto per una forza casuale di valor medio nullo –banalmente, prodotta da un generatore di numeri casuali uniformi, come in Sez.6, inizializzato identicamente nei due casi. Ci si aspetterebbe su tempi lunghi una energia media nulla, per cui la deviazione dell'energia da zero è una misura del drift. Per velocità iniziale nulla abbiamo trovato una deviazione del 3% per Verlet contro il 15% di E-Cromer (fluttuazioni maggiori per velocità iniziale non nulla, ma sempre a favore di Verlet). Incidentalmente, questo calcolino è una rozza realizzazione di una versione non dissipativa dell'equazione stocastica di Langevin in una dimensione, per la quale Verlet è ritenuto gravemente inadatto.]

La traiettoria di fase (Fig.2.12, in alto) conferma quanto detto: Eulero ha una traiettoria non limitata, e con velocità che aumenta. Per E-Cromer e Verlet, che hanno traiettorie simili, è rappresentata la differenza delle traiettorie di fase (x_d,v_d), definita come $x_d \equiv x_{EC} - x_V$, $v_d \equiv v_{EC} - v_V$. Ci sono differenze, ma sono piccole e sembrano tendere ad essere periodiche.

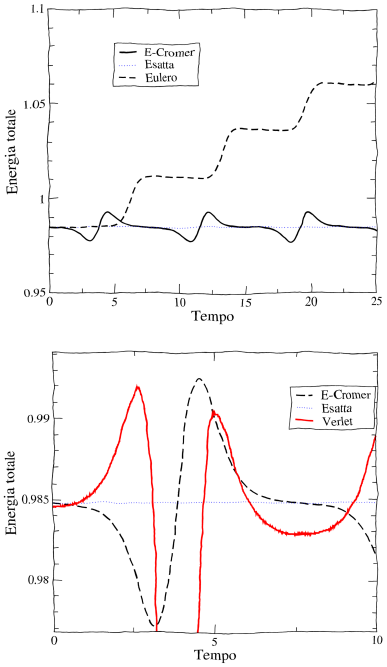


Figura 2.11: Energie totali per il pendolo anarmonico, θ_0=170°, τ=0.01. Sopra, Eulero (linea tratteggiata) vs E-Cromer (continua); sotto, E-Cromer (tratteggiata) e Verlet (continua). Importante: Verlet è amplificata di un fattore 1000 !

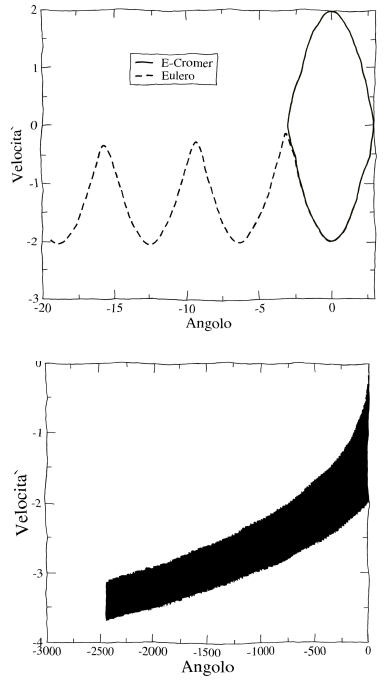


Figura 2.12: Traiettorie di fase per il pendolo anarmonico, θ_0=170°, τ=0.01. Sopra, 2500 step: Eulero (tratteggiata) e E-Cromer (continua); sotto, 10^5 step, solo Eulero.

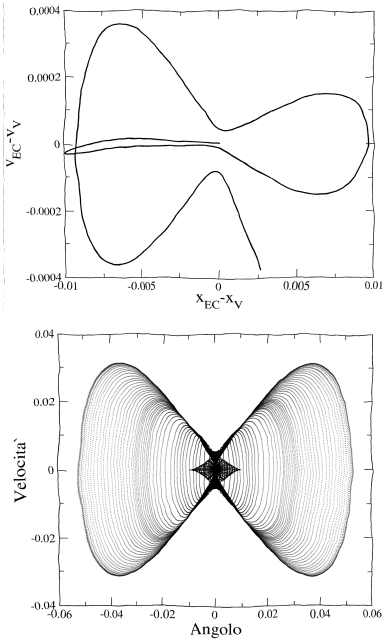


Figura 2.13: Differenza di traiettoria di fase per il pendolo anarmonico, θ_0=170°, τ=0.01, a tempi brevi (sopra) e lunghi (sotto).

Questa supposizione è confermata dall'analisi delle traiettorie di fase per tempi lunghi (Fig.2.12, in basso). Eulero continua il suo drift spurio verso grandi velocità e angoli (a tempi lunghi, la velocità sembra tendere a saturare – chissà). E-Cromer aumenta la sua separazione da Verlet fino a circa l'1% in ambedue le variabili. A tempi lunghi, però (verificato simulando fino a 5×10^6 step), la figura di differenza di fase si ripete sempre uguale: E-Cromer e Verlet hanno ambedue un ciclo limite, leggermente diverso tra loro. Come si vede, la traiettoria di fase sta tra circa $\pm\pi$ in angolo e ±2 in velocità, e le differenze massime sono dell'ordine del $\pm1\%$. In sintesi, E-Cromer emerge come un buon algoritmo anche in confronto a Verlet; Eulero è invece sempre da scartare.

2.6 *Applicazioni*

2.6.1 *Palle: gravità, attrito, rotazione*

2.6.2 *Applicazioni - Oscillatore smorzato, forzato e anarmonico*

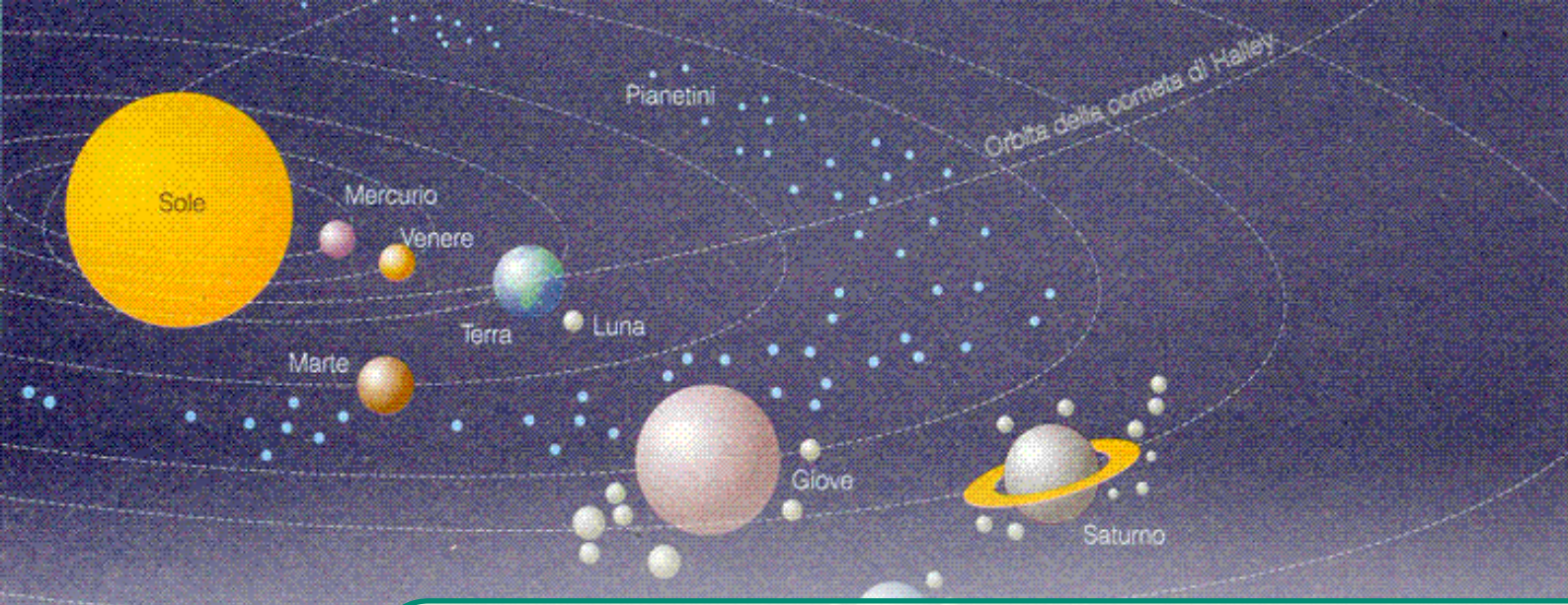
2.6.3 *Applicazioni - Moto planetario: orbite*

2.6.4 *Algoritmi a passo variabile*

2.6.5 *Applicazioni - Moto planetario: problema a "tre" corpi*

2.6.6 *Applicazioni - Moto planetario: precessione relativistica*

2.6.7 *Applicazioni - Moto planetario: fascia dei meteoriti*



3. B - Equazioni differenziali ordinarie

3.1 Oscillatore

In questa sezione scriveremo un codice per risolvere l'equazione differenziale dell'oscillatore armonico generica, implementando i tre metodi presentati nel Cap. 2.A : Eulero, Eulero-Cromer, Verlet.

Gli schemi di integrazione abbiamo visto essere i seguenti:

- Eulero:

$$v_{n+1} = v_n + \tau F_n;\ x_{n+1} = x_n + \tau v_n$$

- Eulero-Cromer:

$$v_{n+1} = v_n + \tau F_n;\ x_{n+1} = x_n + \tau v_{n+1}$$

- Verlet:

$$x_{n+1} = x_n + \tau v_n + F_n \frac{\tau^2}{2};\ v_{n+1} = v_n + \frac{\tau^2}{2}(F_n + F_{n+1})$$

dove con τ indichiamo il passo temporale e con F_n indichiamo l'accelerazione, o la forza diviso la massa, a cui è sottoposto il nostro sistema. Nel nostro caso avremo:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta = -\omega^2 \sin\theta$$

La traduzione in codice di questi schemi è immediata. Vediamola di seguito e poi ci costruiamo il programma intorno:

```
1 #Accelerazione
2 F0=-g/l*sin(x0)
3
4 #Eulero
5 v1=v0+tau*F0
6 x1=x0+tau*v0
7
8 #Eulero-Cromer
```

```
9  v1=v0+tau*F0
10 x1=x0+tau*v1
11
12 #velocity-Verlet
13 x1=x0+v0*tau+0.5*F0*tau**2
14 F1=-g/l*sin(x1)
15 v1=v0+0.5*tau*(F0+F1)
```

Come notiamo facciamo uso di due variabili una denominata con o l'altra con 1 che si riferiscono al tempo n e $n+1$ rispettivamente. Se ora si definiscono le costanti necessarie (g, l, τ) e le condizioni iniziali (x_0,v_0), la serie di righe scritte sopra ci permettono di calcolare la posizione e la velocità dopo un tempo τ, con il metodo che desideriamo. Vediamo di seguito tali definizioni:

```
1 g=1
2 l=1
3 x0=deg2rad(5)
4 v0=0.0
5 tau=0.01
```

Facciamo notare quà l'uso della funzione `deg2rad(alpha)` che permette di convertire l'angolo alpha espresso in gradi sessagesimali in radianti. Questo è necessario perchè la funzione sin(alpha) che troviamo per calcolare la forza prende in input un angolo alpha in radianti. Esiste anche la funzione inversa rad2deg().

Ovviamente vorremo prolungare la nostra simulazione, calcolando la posizione e la velocità del nostro oscillatore, per un periodo di tempo piu lungo di in solo passo τ, per esempio per $T = N \cdot \tau$, con N a piacere. E facile capire che dovremo ripetere lo schema di integrazione scelto per un numero N di volte, mediante un ciclo for, come vediamo di seguito:

```
1 N=3000
2
3 for t_step in range(N):
4   #Eulero-Cromer
5   F0=-g/l*sin(x0)
6   v1=v0+tau*F0
7   x1=x0+tau*v1
8   x0=x1
9   v0=v1
```

Nelle ultime due righe ci ricordiamo di aggiornare la variabile della posizione e della velocità denominata con o con quelle appena calcolate, denominate con 1.

A questo punto il nostro programma calcola la posizione e la velocità dell'oscillatore per gli N istanti di durata τ, ma è ancora poco utile, perchè dopo l'uscità dal ciclo for avremo a disposizione solo l'ultimi valori di posizione e velocità. Sarebbe utile quindi registrare

l'intera dinamica per poi studiarne l'andamento mediante un grafico, per esempio. Facciamo in questo modo:

- Nelle righe iniziali del programma, quindi prima del ciclo for, creiamo due arrays:

```
x_arr=zeros(N+1)
v_arr=zeros(N+1)
```

- registriamo nel primo elemento degli array le condizioni iniziali:

```
x_arr[0]=x0
v_arr[0]=v0
```

- all'interno del ciclo for registriamo ogni nuovo valore di x e v all'interno dei rispettivi arrays:

```
1 for t_step in range(N):
2   ...
3   x_arr[t_step+1]=x0
4   v_arr[t_step+1]=v0
```

Ora una volta eseguite le righe di codice precedenti potremo fare il grafico della posizione (velocità) in funzione del tempo, creando prima un array con tutti gli istanti temporali usati nella integrazione, mediante la funzione linspace():

```
1 time_arr=linspace(0,N*tau,N+1)
2 plot(time_arr,x_arr)
```

Il grafico che compare è mostrato in Fig. **??**.

Se ora vogliamo riprodurre la Fig. 2.4, dovremo lasciare aperta la finestra con il grafico appena creato, cambiare l'angolo da 5 a 90 gradi ed eseguire nuovamente il programma. Cambiando ancora da 90 a 179, otteniamo il grafico mostrato in Fig. 3.2.

Provate a fare il grafico mostrato in Fig. 2.5, dove si mostra la velocità in funzione dell'angolo (la nostra posizione x).

Passiamo ora al calcolo dell'energia cinetica e potenziale dell'oscillatore. L'implementazione è del tutto analoga a quanto abbiamo fatto per salvare la dinamica:

- aggiungiamo all'inizio del codice due array:

```
Ep_arr=zeros(N+1)
Ek_arr=zeros(N+1)
```

- dentro il ciclo for calcoliamo i valori delle due energie usando la posizione e la velocità ad ogni iterazione:

```
1 for t_step in range(N):
2   ...
3   Ep_arr[t_step+1]=-cos(x0)
4   Ek_arr[t_step+1]=0.5*v0**2
```

Ora quindi, ogni volta che eseguiamo il nostro programma, avremo a disposizione l'energia potenziale e cinetica per tutti gli istanti considerati e potremo facilmente farne il grafico:

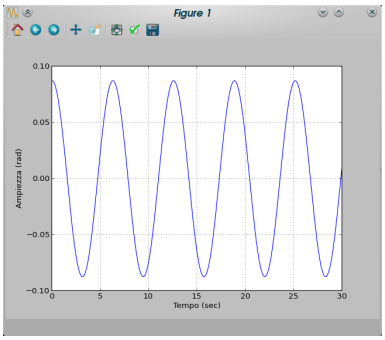


Figura 3.1: Ampiezza del pendolo in funzione del tempo per i primi 30 secondi.

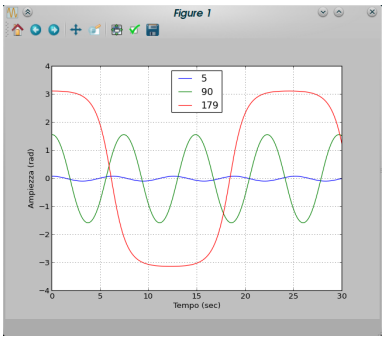


Figura 3.2: Ampiezza in funzione del tempo per diverse ampiezze iniziali.

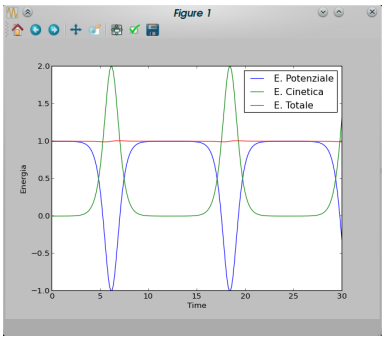


Figura 3.3: Energia totale, cinetica e potenziale dell'oscillatore anarmonico in funzione del tempo.

```
plot(time_arr,Ek_arr)
plot(time_arr,Ep_arr)
plot(time_arr,Ep_arr + Ek_arr)
```

In questo modo si ottiene uno dei due grafici di Fig. 2.7, a seconda dell'angolo scelto. Notiamo come nell'ultima riga possiamo fare il grafico dell'energia totale sommando i vettori direttamente dentro la funzione plot() La figura 3.3 mostra il risultato.

Il programma sin qui costruito permette di fare tutti i confronti mostrati nella dispensa, tra metodi differenti considerando lo stesso angolo di partenza (quindi oscillatore armonico o anarmomico) oppure tra oscillatori diversi usando lo stesso metodo.

Per evitare di dover cambiare il codice ogni volta che vogliamo provare un metodo o delle condizioni iniziali differenti, mostriamo di seguito come inserire dei parametri dall'esterno, subito dopo che il programma viene eseguito:

```
x0 = input('Inserisci angolo di partenza in gradi ')
```

La funzione input('testò) mostra il testo scelto e aspetta che inseriate un valore e premiate invio; il valore inserito verrà memorizzato nella variabile `x0`. Lo stesso metodo lo si può usare per inserire tutti i parametri che si vogliono.

Di seguito, invece, vediamo come usare il costrutto condizionale if per scegliere quale dei tre schemi di integrazione usare:

```
method = input('0 - Eulero; 1 - Cromer; 2 - Verlet ')

for t_step in range(N):
  if method == 1:
    #codice Eulero
  else if method == 2:
    #codice Eulero-Cromer
  else if method == 3:
    #codice Verlet

  x0=x1
  v0=v1
  ...
  ...
```

- nella prima riga supponiamo di chiedere all'utente un numero per identificare il metodo da usare;
- dentro il ciclo for usiamo il costrutto `if method == valore:`, vogliamo che il programma confronti la variabile method con un valore e se questa è uguale allora esegue le righe di codice indentate che si trovano sotto;
- poichè `method` ha un solo valore, una sola delle tre condizioni sarà verificata, quindi uno solo dei tre metodi verrà di fatto eseguito;
- al posto del commento andranno inserite le righe di codice relative al metodo viste all'inizio del tutorial;

- subito dopo il costrutto `if`, possiamo lasciare tutte le righe gia usate in precedenza, poichè sono comuni ai tre metodi.

3.2 *Orbite dei Pianeti*

In questa sezione vediamo come calcolare l'orbita di un pianeta attorno al sole. Le equazioni da cui partire sono quelle che descrivono le componenti della forza di attrazione gravitazionale a cui è soggetto il pianeta in esame da parte del sole, considerato fisso in uno dei due fuochi dell'orbita:

$$\frac{d^2x}{dt^2} = \frac{F_x}{M_E}; \frac{d^2y}{dt^2} = \frac{F_y}{M_E};$$

$$F_x = -\frac{GM_EM_S}{r^2}\cos\theta, F_y = -\frac{GM_EM_S}{r^2}\sin\theta$$

Anche in questo caso abbiamo a che fare con le equazioni differenziali ordinarie, come nel caso dell'oscillatore. La novità è che trattandosi di un sistema a due dimensioni, ne dovremo risolvere due contemporaneamente, una per componente.

Vediamo di seguito il corpo principale del programma, constituito da un ciclo for che contiene uno dei metodi di integrazione visti in precedenza e lo esegue iterativamente:

```
for istep in range(Nstep):
  Fx=-GM*x/r**3
  Fy=-GM*y/r**3

  #Eulero-Cromer
  #vx+=tau*Fx
  #vy+=tau*Fy
  #x+=tau*vx
  #y+=tau*vy
  #r=sqrt(x**2+y**2)

  #velocity-Verlet
  x+=vx*tau+0.5*Fx*tau**2
  y+=vy*tau+0.5*Fy*tau**2
  r=sqrt(x**2+y**2)
  F1x=-GM*x/r**3
  F1y=-GM*y/r**3
  vx+=0.5*tau*(Fx+F1x)
  vy+=0.5*tau*(Fy+F1y)

  x_arr[istep+1]=x
  y_arr[istep+1]=y
  vx_arr[istep+1]=vx
  vy_arr[istep+1]=vy
```

Spieghiamo i punti di novità rispetto a quanto già visto per il caso 1D:

- abbiamo scritto anche qui due metodi, quello di Cromer e quello di Verlet, in modo che commentando uno dei due possiamo provarne l'accuratezza e l'affidabilità;
- poichè il sistema è bidimensionale abbiamo due componenti per la posizione (x, y), la velocità (v_x, v_y), e l'accelerazione (F_x, F_y), ognuna delle quali si calcola allo stesso modo di quello previsto dallo schema di integrazione;
- calcoliamo il raggio a partire dalle coordinate (x, y) appena calcolate, perchè esso è richiesto per il calcolo dell'accelerazione;
- al posto della forma `var1=var0+...`, con due variabili, denominate con 0 e 1, che indicavano la variabile allo tempo n e $n+1$, usiamo la forma `var+=...` che aggiunge al valore gia contenuto nella variabile il nuovo valore; questo ci permette di non dover aggiornare la variabile 0 con quella 1 dopo averla calcolata (`var0=var1`);
- anche qui salviamo la posizione e la velocità in appositi arrays.

Affinchè il programma sia completo ci servono le definizioni delle costanti usate dentro il ciclo e delle condizioni iniziali, vediamole di seguito:

```
#Costanti moto
GM = 4*pi**2 # G*M in unita astronomiche (UA)
m=5.9722E+24 # massa terra (kg)

#Posizioni e velocit\'a iniziali in UA, al perielio
x=0.98
y=0.0
r=sqrt(x**2+y**2)
vx=0.0
vy=2*pi

# Nstep*tau \'e il tempo totale  in anni
Nstep=1000 # Numero di passi temporali
tau=0.001 # lunghezza singolo passo temporale

#arrays posizione e velocit\'a
x_arr=zeros(Nstep+1)
y_arr=zeros(Nstep+1)
vx_arr=zeros(Nstep+1)
vy_arr=zeros(Nstep+1)

#Salvataggio delle condizioni iniziali
x_arr[0]=x
y_arr[0]=y
vx_arr[0]=vx
vy_arr[0]=vy
```

Notiamo come si siano usate le unità astronomiche: quindi le lunghezze espresse in termini di distanza media Terra-Sole e il tempo in anni. Queste unità sono più adatte al sistema che stiamo trattando

e ci permettono di ridurre il numero di iterazioni da compiere, quindi il tempo di attesa e il costo computazionale.

Le condizioni iniziali proposte sono relative al pianeta Terra. Ci aspettiamo quindi di osservare una singola orbita (`Nstep*tau=1` anni) quasi circolare. Facciamo il grafico al solito modo, poi eguagliamo la scala dei due assi in maniera che l'orbita non venga distorta:

```
plot(x_arr,y_arr)
axis('equal')
```

La Fig. 3.4 mostra il grafico atteso.

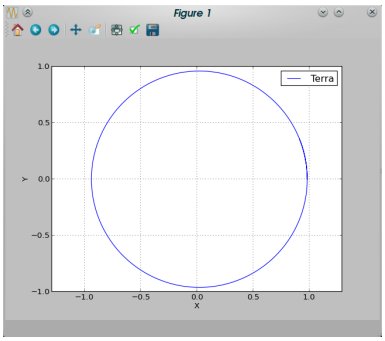


Figura 3.4: Orbita terrestre, quasi circolare.

Per generare le orbite di altri pianeti basterà impostare la posizione e la velocità iniziali come fatto per la Terra, regolare il numero di step, eseguire nuovamente il programma e infine fare il grafico. Potremo per comodità chiedere all'utente di inserire questi parametri, come abbiamo gia visto per l'oscillatore:

```
x=input('Posizione iniziale perieli\'o)
vy=input('Velocit\'a iniziale perieli\'o)
Nstep=input('Numero di passi')
```

Eseguendo il programma più volte, lasciando la prima finestra del grafico aperta, inseriamo i seguenti paremetri: `Nstep=1000`, $x = 0.98$, $v_y = 2pi, 2pi+1, 2pi+0.5, 2pi-0.5, 2pi-1$; quindi lasciamo fisso il numero di passi e la posizione iniziale e cambiamo la velocità iniziale lungo y. Il grafico che si ottiene è mostrato in Fig. 3.5.

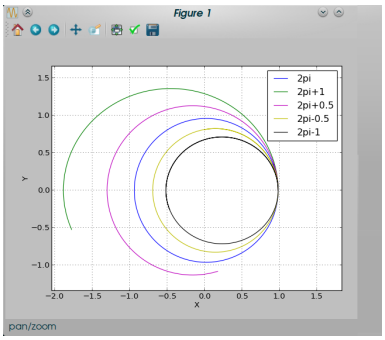


Figura 3.5: Orbite ottenute variando la velocità iniziale rispetto a quella terrestre di 2π

Come osserviamo, le due orbite più grandi della blu (terrestre) non si chiudono per via del fatto che `Nstep=1000` non è sufficiente. Viceversa le orbite più piccole della blu sono piu lunghe di un singolo giro.

Se volessimo calcolare altre quantità legate al moto del pianeta in orbita attorno al sole, come ad esempio il raggio, l'angolo θ tra raggio e asse x, velocità angolare, energia cinetica e potenziale, il metodo è sempre lo stesso: aggiungiamo un array per la grandezza scelta, impostiamo la prima componente in base alle condizioni iniziali e poi scriviamo ad ogni iterazione il nuovo valore nelle componenti successive, come di seguito mostrato per l'angolo θ:

```
...
theta_arr=zeros(Nstep+1)
theta_arr[0]=arctan2(y,x)
...
for istep in range(Nstep):
  ...
  theta_arr[istep+1]=arctan2(y,x)

time_arr=linspace(0,Nstep*tau,Nstep+1)
plot(time_arr,theta_arr)
```

La funzione `arctan2(y,x)` ci permette di calcolare l'angolo compreso tra raggio e asse x in base alle coordinate (x, y) della posizione del pianeta. Il grafico che si ottinene per il caso della terra e per il

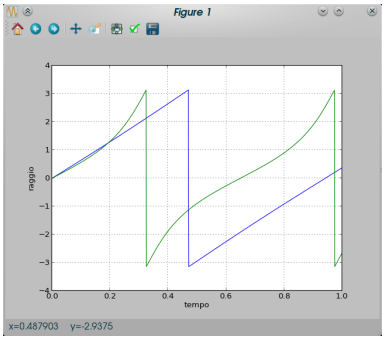
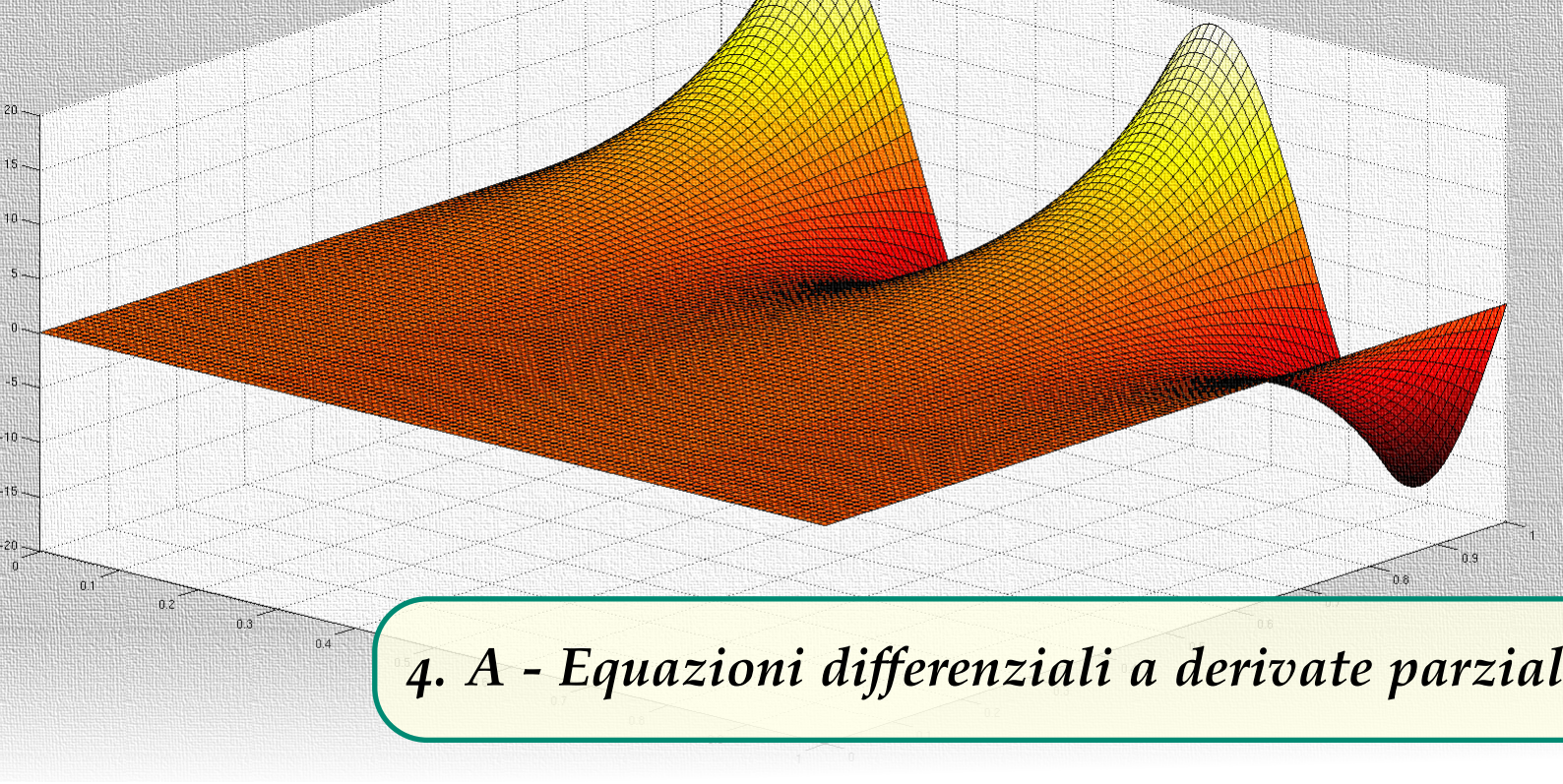


Figura 3.6: Raggio dell'orbita terrestre e di una più eccentrica in funzione del tempo.

caso di maggiore eccentricità dell'orbita (ottenuto con $v_y = 2pi - 1$) è mostrato in Fig. 3.6.

L'angolo cresce linearmente nel caso di orbita circolare, come nel caso approsimato della terra. I salti che si vedono son dovuti al fatto che la funzione `artan2()` restituisce solo angoli compresi tra $-\pi, \pi$. Quando l'orbita è ellitica, la velocità del pianeta non è piu costante per cui l'angolo varia piu velocemente in corrispondenza del afelio.



4. A - Equazioni differenziali a derivate parzial

4.1 Introduzione

Le equazioni differenziali a derivate parziali sono relazioni tra una funzione incognita di più variabili e le sue derivate parziali. Come si sa, le derivate parziali sono normali derivate, rispetto a una data variabile, fatte mantenendo le altre variabili fisse (come schematizzato in Fig.4.1).

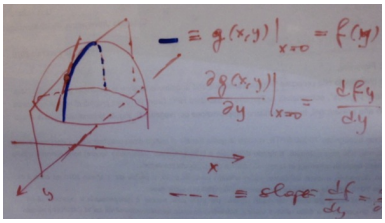


Figura 4.1: Schema mnemonico della definizione di derivata parziale.

Come le ODE, le PDE sono caratterizzate, tra l'altro, dal proprio ordine e carattere lineare o non-lineare. Utilizzando una notazione del tipo

$$\frac{\partial f}{\partial x} = f_x; \quad \frac{\partial^2 f}{\partial x \partial y} = f_{xy}; \quad \dots$$

una PDE è per esempio

$$u_x + u_y + u = 0,$$

che è del primo ordine e lineare –ovvero non vi compare nessuna potenza o funzione trascendente della funzione incognita u o delle sue derivate. L'insieme dei termini che contengono derivate sono la *parte principale*. Un esempio di PDE in 2 variabili è

$$u_{xx} + 3u_{xy} + u_{yy} + u_x - u = \cos(x - y)$$

che è lineare (gli argomenti del coseno sono le variabili e non la funzione) e a coefficienti costanti. La PDE

$$u_{xx} + 3x^2 u_{xy} - u = 0$$

è sempre lineare, ma a coefficienti variabili. Una equazione non-lineare, ad esempio, è

$$u_{xx} + 3x^2 u_{xy} + u_x^2 - u = 0$$

perchè vi compare il quadrato della derivata prima.

Contrariamente alle ODE, le cui soluzioni sono garantite esistere dal teorema di Picard-Lindelöf (o di Cauchy-Lipschitz), l'esistenza delle soluzioni delle PDE non è generalmente garantita. Negli anni '50, si è trovato che il teorema di Cauchy-Kowalevski ("*Il problema di Cauchy per una PDE i cui coefficienti siano analitici nella funzione incognita e nelle sue derivate ha una soluzione analitica e localmente unica.*") ha delle eccezioni. Alcune di queste sono nascoste nella definizione delle condizioni al contorno, che sono ancora più determinanti e stringenti che per le ODE: qui si parla infatti di definire, in generale, non solo i valori ma anche la forma funzionale della soluzione e di alcune delle sue derivate sui bordi del dominio.

Per quanto ci riguarda ci limiteremo a PDE di ordine massimo 2, in una o due variabili, con coefficienti costanti delle derivate, che tipicamente non presentano difficoltà patologiche. Il caso del secondo ordine permette inoltre una classificazione delle PDE in *paraboliche*, *ellittiche*, e *iperboliche*. Nella notazione introdotta sopra, una PDE del secondo ordine è del tipo

$$A f_{xx} + 2B f_{xy} + C f_{yy} + \cdots = 0, \tag{4.1}$$

dove abbiamo omesso termini di ordine inferiore (tipo f_x e simili) e tenuto conto del teorema di Schwartz (f_{xy}=f_{yx}). I coefficienti A, B, e C possono in generale essere funzioni di x e y. Una PDE è del secondo ordine in un dominio nel piano (x,y) se

$$A^2 + B^2 + C^2 > 0 \tag{4.2}$$

in quel dominio. Una PDE del secondo ordine può essere convertita tramite trasformata di Fourier in una *equazione caratteristica*,

$$AX^2 + b\,XY + CY^2 + \cdots = 0, \tag{4.3}$$

analoga all'equazione di una sezione conica, sostituendo $X \to f_x$, $X^2 \to f_{xx}$, etc. [Per coefficienti costanti, infatti, si ha

$$\mathrm{FT}\left(\frac{d^2 f}{dx^2}\right) \propto X^2 \overline{f}, \tag{4.4}$$

con $\overline{f}$ è la trasformata di f, e X, etc., le variabili duali di x, etc.] Le sezioni coniche sono classificate in ellissi, parabole, o iperboli a seconda del valore del discriminante

$$b^2 - 4AC = 4B^2 - 4AC = 4(B^2 - AC),$$

e in particolare del suo segno; questa classificazione (almeno in 2 variabili) si può estendere alle PDE:

B^2–AC<0 equazioni *ellittiche*. Esempio classico, l'equazione di Laplace,

$$u_{xx} + u_{yy} = 0, \tag{4.5}$$

che ha B=0, AC=1, e quindi B^2–AC=–1<0. Le equazioni ellittiche sono definite in un dominio di due variabili spaziali, con condizioni al contorno definite sul bordo del dominio (vedi sotto). Le soluzioni sono ben comportantisi nella regione di definizione.

B^2–AC=0 equazioni *paraboliche*. Il classico esempio è la equazione di diffusione (o del calore, o di Newton-Fourier)

$$u_{xx} = u_t, \tag{4.6}$$

per cui B=C=0, e quindi B^2–AC=0. Altro esempio è l'equazione di Schrödinger dipendente dal tempo

$$-u_{xx} + Vu = u_t,$$

dove V è una funzione di x. Questi problemi sono definiti su un dominio bidimensionale misto spaziale e temporale. Le soluzioni sono ben comportantisi soprattutto a tempi grandi (se lo schema di integrazione è stabile).

B^2–AC>0 : equazioni *iperboliche*. Definite anch'esse su dominio misto spazio-tempo, propagano qualunque discontinuità o forte variazione presente nelle condizioni iniziali, e sono quindi generalmente più difficili da integrare degli altri due tipi. Esempio principe, l'equazione delle onde di d'Alembert,

$$u_{xx} - u_{tt} = 0,$$

che ha B=0, C=$-A$=1, e quindi B^2–AC=1>0.

La difficoltà delle equazioni iperboliche risalta nel caso di n variabili $x_1, \ldots, x_n$, in cui una PDE del secondo ordine ha una forma del tipo

$$\tilde{A}f = \sum_i^n \sum_j^n a_{ij} f_{x_i x_j},$$

e ora la classificazione dipende dal segno degli autovalori della matrice $\tilde{A}$ dei coefficienti. Nel caso ellittico, gli autovalori sono tutti negativi o tutti positivi; nel caso parabolico, sono tutti negativi o tutti positivi tranne uno che è nullo; nel caso iperbolico, un solo autovalore è negativo (positivo) e gli altri sono positivi (negativi); infine, esiste il caso ultraiperbolico, in cui c'è più di un autovalore positivo o negativo (e nessuno nullo).

Un problema di PDE è "ben posto secondo Hadamard" se esiste una soluzione unica (il che è garantito nei casi che ci interessano dal teorema di Cauchy-Kowalevski) e con dipendenza continua dai dati ausiliari (le condizioni al contorno): questi devono essere in numero idoneo, e adeguati al tipo di PDE. L'approccio numerico alle PDE può essere poi basato su diversi tipi di rappresentazioni o discretizzazioni. Si può discretizzare per differenze finite, cioè su griglie con direzioni di solito mutuamente ortogonali, con spacing opportuni nelle diverse dimensioni (è quel che faremo qui, usando di norma griglie quadrate in 2D con spacing uguali); per elementi finiti, cioè suddividendo il dominio in elementi di forma e numero che variano localmente adattandosi al tipo di problema e di dominio; oppure non discretizzare affatto, come nel caso dei metodi spettrali, che rappresentano le soluzioni tramite trasformate generalizzate di Fourier (questo è vero solo formalmente, se la trasformata va calcolata numericamente).

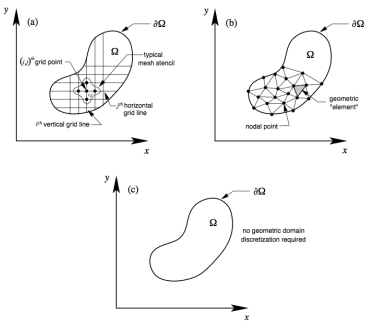


Figura 4.2: Tipi di discretizzazione delle PDE: differenze finite (a), elementi finiti (b), metodi spettrali (c).

Le condizioni al contorno sono, per i nostri scopi, di due tipi: al bordo (di Cauchy o Neumann) e miste ai valori iniziali e al bordo. Nel primo caso, si fissano i valori della funzione (o delle derivate di ordine inferiore a quello massimo) sul bordo del dominio: ad esempio, nell'equazione di Laplace bidimensionale, i valori del potenziale sono fissati su tutto il bordo unidimensionale del dominio. Sui punti del bordo la funzione non varia. Il passo di discretizzazione della griglia, così come il numero delle divisioni, è diverso nelle due variabili, anche se noi useremo per semplicità solo griglie equispaziate in tutte le variabili. Nel secondo caso, la funzione è fissata all'istante iniziale su tutto il dominio, ma solo i valori sul bordo (che si riduce ai due punti estremi del domino spaziale unidimensionale) rimangono costanti durante l'evoluzione temporale.

4.2 *Equazioni paraboliche: l'equazione del calore*

L'equazione del calore è la equazione parabolica paradigmatica. In una variabile spaziale x e una temporale t, chiamando la funzione incognita T come temperatura,

$$\kappa \frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}, \tag{4.7}$$

con κ un coefficiente legato alla conduttività termica. Se la temperatura è molto alta in una regione particolare del dominio rispetto ai punti adiacenti, T avrà una grande curvatura negativa; l'equazione asserisce che in quella regione la variazione temporale di T sarà grande e negativa, e quindi T tenderà ad equalizzarsi sul dominio x, raccordandosi alle condizioni al bordo, fino a che curvatura e variazione temporale si annullino entrambe. È utile considerare la soluzione particolare per la condizione iniziale $T(x,t{=}0){=}\delta(x - x_0)$, con $T{=}0$ come condizione al bordo:

$$T(x,t) = \frac{1}{\sqrt{2\pi}\sigma(t)} \exp\left(-\frac{(x-x_0)^2}{2\sigma^2(t)}\right). \tag{4.8}$$

Lo spike iniziale si allarga nel tempo come

$$\sigma(t) = \sqrt{2\kappa t}. \tag{4.9}$$

Questo è in accordo con la discussione appena fatta. L'andamento $\sqrt{t}$ è tipico dei fenomeni stocastici, di cui la diffusione, come sintetizzata in Eq.4.7, è un esempio. Abbiamo incontrato questo comportamento negli errori di arrotondamento ($\propto\sqrt{N}$) e lo ri-incontreremo nel random walk Montecarlo (Sez.6).

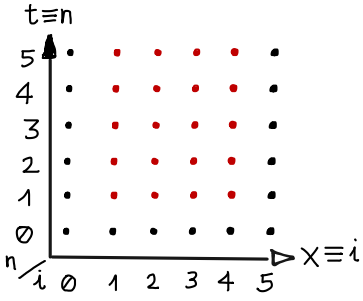


Figura 4.3: Griglia mista 1+1-D per l'equazione di diffusione ($N{=}5$; la griglia è aperta per n crescenti). Le condizioni iniziali e al contorno sono definite sui punti in nero per, rispettivamente, $n{=}0$ $\forall i$ (iniziali) e $i{=}1$ e $i{=}N$, $\forall n$ (bordo).

Per discretizzare l'equazione, dividiamo il dominio spaziale di ampiezza L in $N{+}1$ intervallini di ampiezza $h{=}L/N$, e poniamo

$$\begin{aligned} t \to t_n &= t_0 + n\tau, \quad n = 0, 1, \dots \\ x \to x_i &= -\frac{L}{2} + hi, \quad i = 0, N \end{aligned} \tag{4.10}$$

così che x varia da $-L/2$ a $L/2$, e t da 0 in avanti. Rappresentiamo, d'accordo con la convenzione di Eq.4.10, la funzione $T(t,x)$ come T_i^n.

Naturalmente l'apice n indica il passo temporale e non un elevamento a potenza. Utilizziamo poi le espressioni Eq.1.22 e 1.27 per le derivate prima e seconda. Si ottiene

$$\kappa \frac{T_{i+1}^n + T_{i-1}^n - 2T_i^n}{h^2} = \frac{T_i^{n+1} - T_i^n}{\tau}$$

e riarrangiando

$$T_i^{n+1} = T_i^n + \frac{\tau\kappa}{h^2}[T_{i+1}^n + T_{i-1}^n - 2T_i^n]. \tag{4.11}$$

In Fig.4.3 è esemplificata la griglia 1+1-dimensionale mista per N=6; ovviamente, è inteso che la griglia continua indefinitamente in n. Questo schema di integrazione *esplicito*, noto come FTCS (forward-time–centered-space, dalle due formule usate per le derivate), predice T nel punto i al tempo n+1, usando solo valori di T al tempo n (nei punti i, i+1, i–1). Questa operazione è sempre possibile a partire dai valori di T_i^0, che sono forniti $\forall i$ dalle condizioni iniziali, e considerando che T_0^n e T_N^n sono i valori al bordo e restano costanti nel tempo $\forall n$. Non si usano le differenze centrate nella derivata temporale perchè causerebbero una dipendenza da t_{n-1} (in realtà si usano eccome, ma noi qui non lo facciamo).

Come si nota, il coefficiente del termine che "aggiorna" T lega il passo temporale e quello spaziale – il che è ragionevole dato che la variazione in t del profilo di T dev'essere legato alla sua velocità di allargamento in x. Ne risulta un vincolo di stabilità che lega mutuamente τ e h. Considerata Eq.4.9, stipuliamo che t_s sia il tempo necessario alla soluzione per "allargarsi" di h nello spazio, cioè

$$\sigma(t_s) = \sqrt{2\kappa t_s} = h,$$

da cui

$$t_s = \frac{h^2}{2\kappa}. \tag{4.12}$$

Possiamo quindi riscrivere Eq.4.11 come

$$T_i^{n+1} = T_i^n + \frac{\tau}{2t_s}[T_{i+1}^n + T_{i-1}^n - 2T_i^n]. \tag{4.13}$$

È intuibile che osservare la soluzione ad intervalli temporali più lunghi di t_s, cioè usare $\tau > t_s$, è sconsigliabile, perchè la soluzione ha, nel frattempo, già percorso un tratto di spazio maggiore di h, e questo potrebbe causare inaccuratezze e instabilità. Dunque (lo mostreremo formalmente per un caso più semplice) per il metodo FTCS c'è la condizione di stabilità

$$\tau \leq t_s = \frac{h^2}{2\kappa}. \tag{4.14}$$

Un h piccolo implica direttamente (e quadraticamente !) un τ piccolo. Sulla questione dell'analisi di stabilità torneremo in Sez.4.3.

In Fig.4.4 è mostrata, per valori di τ minore, uguale, e maggiore maggiore della soglia di stabilità t_s, l'evoluzione di un profilo di T dato inizialmente da $T(x, t = 0) = \delta(x)$ al centro del dominio. Come si vede la soluzione si evolve in modo smooth nella regione di stabilità,

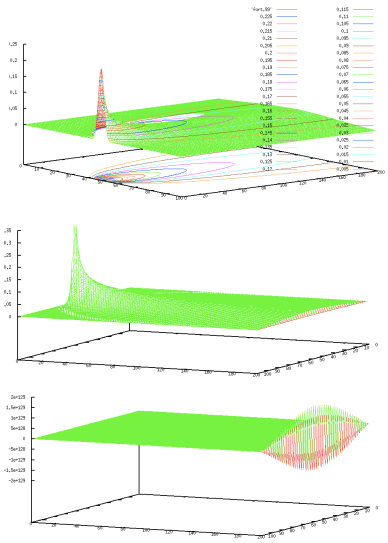


Figura 4.4: Evoluzione di T per $T(x, t = 0) = \delta(x)$ per valori τ minore di (in alto), uguale a (al centro), e maggiore (in basso) di t_s.

e invece diverge (notare la scala) al di fuori di essa. Al confine, la situazione è stabile ma mostra un'oscillazione (visibile dalle linee di connessione a grandi t) che suggerisce incipiente instabilità. Notiamo che, avendo discretizzato il dominio come in Eq.4.10, il punto centrale (se N è pari; altrimenti sarà subito a destra o a sinistra del centro geometrico dell'intervallo) ha indice $N/2$. La condizione iniziale può quindi essere discretizzata come

$$T_i^0 = \frac{\delta_{i,N/2}}{h} \equiv \Delta_{N/2}$$

dove δ_{lk} è il simbolo di Kronecker. Infatti l'integrale è

$$\sum_i T_i^0 h = \sum_{i=0}^{N} \Delta_i h = \delta_{N/2,N/2} = 1.$$

Con il semplice metodo analizzato si può sperimentare con condizioni al contorno diverse, e con pozzi o sorgenti di calore (T è mantenuta a valori diversi all'interno del dominio).

Un esempio è il semplice modello per una reazione autosostenuta, ad esempio in un reattore a fissione, basato sull'equazione

$$n_t = Dn_{xx} + Cn$$

per la densità di neutroni nel reattore. Oltre alla fissione spontanea, i nuclei di uranio (ad esempio) possono subire fissione per assorbimento di neutroni, i quali sono peraltro emessi dall'evento stesso della fissione. È perciò presente un termine sorgente proporzionale alla densità di neutroni: infatti, tanti più neutroni sono presenti, tanti più eventi di fissione indotta si realizzeranno. I neutroni, inoltre, diffondono nel sistema con un certo coefficiente caratteristico del materiale. Le condizioni al contorno di zero densità ai bordi descrivono l'assorbimento dei neutroni da parte delle pareti e quindi la loro scomparsa agli effetti della reazione. La presenza del termine sorgente fa sì che la densità totale possa tuttavia conservarsi o aumentare nel tempo in opportune condizioni. Supponiamo che sia possibile separare le variabili,

$$n = X(x)\, T(t).$$

L'equazione diventa

$$X\frac{\partial T}{\partial t} = DT\frac{\partial^2 X}{\partial x^2} + CXT \tag{4.15}$$

e quindi

$$\frac{1}{T}\frac{\partial T}{\partial t} = \frac{D}{X}\frac{\partial^2 X}{\partial x^2} + C. \tag{4.16}$$

Essendo il primo membro funzione solo di t, e il secondo solo di x, l'unica possibilità è che siano entrambi costanti. Dunque

$$\frac{1}{T}\frac{dT}{dt} = \alpha, \tag{4.17}$$

che ha soluzione

$$T(t) = T(0)\, e^{\alpha t}.$$

La parte temporale della soluzione è quindi esponenzialmente amplificata se α>0. Per contro il secondo membro dà un'equazione tipo oscillatore armonico,

$$\frac{d^2X}{dx^2} = \frac{\alpha - C}{D}X,$$

che ha una soluzione oscillante generale del tipo

$$X(x) = \sum_{j=1}^{\infty} a_j \sin\left[\frac{j\pi}{L}(x + \frac{L}{2})\right] \tag{4.18}$$

dove

$$-\omega^2 \equiv= -(\frac{j\pi}{L})^2 = \frac{\alpha - C}{D} \tag{4.19}$$

Dunque, risolvendo per α, la condizione di amplificazione α>0 si traduce in

$$\alpha = C - D(\frac{j\pi}{L})^2 > 0 \quad \forall j \tag{4.20}$$

ovvero

$$C > \frac{Dj^2\pi^2}{L^2} \Rightarrow \frac{D\pi^2}{CL^2} < \frac{1}{j^2}. \tag{4.21}$$

Poichè

$$\max_{j\in(1,\infty)} \frac{1}{j^2} = 1,$$

la condizione di amplificazione è

$$L > L_c \equiv \pi\sqrt{\frac{D}{C}}. \tag{4.22}$$

Si vede che un grande tasso di produzione di neutroni C diminuisce le dimensioni critiche (la quantità di materiale), mentre una alta diffusività D le aumenta, perchè i neutroni viaggiano più a lungo prima di essere coinvolti in un evento di fissione o assorbimento.

In Fig.4.5 è rappresentata la densità media dei neutroni per diversi valori di L. Avendo assunto D=C=1, L=π. Si notano i diversi regimi di smorzamento, critico, e di espansione per L sotto, vicino a, e sopra π. Naturalmente l'instabilità intesa come aumento indefinito della densità a grande t) sopra L_c non ha nulla a che fare con la stabilità numerica, che è sempre imposta nell'integrazione dell'equazione.

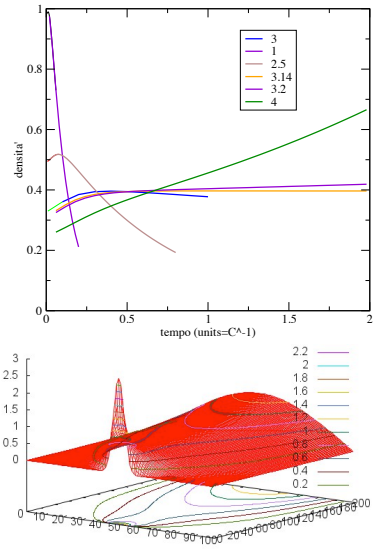


Figura 4.5: Densità media dei neutroni per diverse dimensioni del sistema intorno al valore critico (a destra), e andamento della densità locale nel sistema in funzione del tempo (a sinistra) per L=5.

4.3 *Analisi di stabilità*

Torniamo ora a studiare la stabilità numerica del metodo FTCS con una analisi matriciale. Usiamo la notazione $u_i^n \to \mathbf{u}_n$, poichè per ogni dato n i valori u_i^n sono le componenti di un vettore di dimensione N, e indichiamo n al pedice per distinguerlo dall'elevamento a potenza. Il passo di evoluzione temporale da n a n+1 genera la funzione $\mathbf{u}_{n+1}$ a partire da $\mathbf{u}_n$ tramite moltiplicazione per una matrice di iterazione $\tilde{A}$:

$$\mathbf{u}_{n+1} = \tilde{A}\mathbf{u}_n. \tag{4.23}$$

Come si può verificare direttamente, per FTCS la matrice (qui per N=6) è

$$\tilde{A} = \tilde{\mathcal{I}} + \frac{\tau}{2t_s} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.24}$$

con $\mathcal{I}$ la matrice identità. Poichè Eq.4.23 vale $\forall n \geq 0$, si ha

$$\begin{aligned} \mathbf{u}_1 &= \tilde{A}\mathbf{u}_0 \\ \mathbf{u}_2 &= \tilde{A}\mathbf{u}_1 = \tilde{A}(\tilde{A}\mathbf{u}_0) = \tilde{A}^2\mathbf{u}_0 \\ \mathbf{u}_3 &= \tilde{A}\mathbf{u}_2 = \tilde{A}(\tilde{A}\mathbf{u}_1) = \tilde{A}(\tilde{A}(\tilde{A}\mathbf{u}_0)) = \tilde{A}^3\mathbf{u}_0 \\ &\vdots \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ \mathbf{u}_k &= \tilde{A}\mathbf{u}_{k-1} = \tilde{A}(\tilde{A}\mathbf{u}_{k-2}) = \tilde{A}(\tilde{A}(\tilde{A}\mathbf{u}_{k-3})) = \cdots = \tilde{A}^k\mathbf{u}_0. \end{aligned}$$

(La potenza k-esima di una matrice è semplicemente il prodotto matriciale ripetuto k volte.) Poichè gli N autovettori $\mathbf{v}$ di $\tilde{A}$ sono una base del dominio di definizione di $\mathbf{u}_n$, posso sviluppare

$$\mathbf{u}_0 = \sum_{m=1}^{N} c_m \mathbf{v}_m.$$

Applicando $\tilde{A}^k$ a $\mathbf{u}_0$ si ha

$$\mathbf{u}_k = \tilde{A}^k\mathbf{u}_0 = \sum_{m=1}^{N} c_m \tilde{A}^k \mathbf{v}_m = \sum_{m=1}^{N} c_m \lambda_m^k \mathbf{v}_m \leq \lambda_{\max}^k \sum_{m=1}^{N} c_m \mathbf{v}_m.$$

Ora se il massimo autovalore di $\tilde{A}$ è $|\lambda_{\max}|>1$ la sequenza diverge per grandi k; se $|\lambda_{\max}|\leq 1$, invece, essa converge. Il massimo autovalore è detto anche raggio spettrale. La matrice e i suoi autovalori dipendono dal valore di τ/t_s. Scegliendo il valore limite τ/t_s=1, la matrice diventa (sempre per N=6 per semplicità)

$$\tilde{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.25}$$

Gli autovalori di questa matrice, come si può verificare con una routine `lapack` di diagonalizzazione o power iteration, sono pari a ($\pm$0.809,$\pm$0.309, 1, 1), e quindi l'iterazione converge. Risulta che il massimo autovalore positivo (doppio) è 1 per qualunque N o τ. Ma se $\tau>t_s$, quindi sopra la soglia di stabilità, compaiono autovalori negativi minori di –1 e si ha quindi instabilità. Si conferma quindi che il raggio spettrale della matrice di iterazione causa instabilità sopra la soglia euristica τ=t_s.

4.3.1 *Metodi stabili impliciti: l'equazione di Schrödinger*

Un metodo per le equazioni paraboliche alternativo a FTCS che risulta sempre stabile è quello di Crank-Nicolson, un metodo implicito (la soluzione al tempo n+1 dipende da sé stessa, quindi serve una inversione di matrice), ma incondizionatamente stabile. Lo discutiamo nel contesto della meccanica quantistica, applicandolo alla equazione di Schrödinger dipendente dal tempo. Essa si ottiene dall'hamiltoniana (il funzionale energia) classica non-relativistica di una particella massiva,

$$H = \frac{p^2}{2m} + V(x) = E,$$

con le regole (dette di prima quantizzazione)

$$p \to i\hbar\partial_x, \quad E \to i\hbar\partial_t, \quad x \to x,$$

($\hbar$ è la costante ridotta di Planck) discusse per esempio nel Capitolo 1 di *Conceptual foundations of quantum mechanics* di B. d'Espagnat. Ne risulta

$$\hat{H}\psi \equiv -\frac{\hbar^2}{2m}\psi_{xx} + v(x)\psi = i\hbar\psi_t, \tag{4.26}$$

che è formalmente una equazione parabolica, e specificamente una equazione di diffusione con sorgente. L'Hamiltoniana del sistema classico è ora un operatore (misto differenziale e moltiplicativo).

La soluzione di questa equazione è la funzione d'onda ψ, il cui modulo quadrato $|\psi^2|$ è la densità di probabilità di osservare in x una particella di massa m e soggetta a un potenziale $V(x)$, ovvero la probabilità di questa osservazione in un volumetto Ω_x è

$$P(\Omega_x) = \int_{\Omega_x} |\psi|^2 dx.$$

La soluzione formale di Eq.4.26 è

$$\psi(x,t) = \exp\left[-i\frac{\hat{H}t}{\hbar}\right]\psi^0(x), \tag{4.27}$$

cioè risulta dall'azione del cosiddetto *propagatore* su una funzione ψ_0 normalizzata. Se questa è soluzione del problema ausiliare stazionario

$$\hat{H}\psi^s \equiv -\frac{\hbar^2}{2m}\psi^s_{xx} + v(x)\psi^s = E\psi^s, \tag{4.28}$$

(su cui si basa gran parte della meccanica quantistica "applicata" ad atomi, molecole, solidi, nanostrutture), la soluzione è semplicemente

$$\psi(x,t) = \exp\left[-i\frac{Et}{\hbar}\right]\psi^s(x),$$

cioè lo stato ha forma fissa spaziale e una fase oscillante temporale. Gli oggetti fisici descritti da questa e analoghe equazioni hanno comportamento duale onda-particella: il loro momento p (mv nel caso non relativistico) è legato ad una lunghezza d'onda λ (o a un vettore d'onda k=2π/λ) dalla relazione di de Broglie

$$p = \frac{h}{\lambda} = \frac{h}{2\pi}\frac{2\pi}{\lambda} = \hbar k,$$

con h la costante di Planck.

En passant, notiamo che l'equazione stazionaria 4.28 deriva direttamente dall'equazione di Helmholtz

$$\psi^s_{xx} = -k^2\psi^s,$$

o più precisamente dalla sua versione non-omogenea

$$\psi^s_{xx} + k^2\psi^s = -f(x).$$

L'equazione omogenea deriva banalmente dall'equazione d'onda di d'Alembert

$$\left(\partial_{xx} - \frac{1}{c^2}\partial_{tt}\right)\psi = 0$$

nell'ipotesi di fattorizzabilità

$$\psi(x,t) = \psi^s(x)\,T(t).$$

L'equazione di Helmholtz descrive quindi onde stazionarie di vettore d'onda k, e di forma dipendente dalle condizioni al contorno.

Tornando all'evoluzione temporale, la ψ^s è solo una delle possibili condizioni iniziali: è possibile fare evolvere anche cosiddetti "pacchetti d'onda" costruiti sovrapponendo più ψ^s diverse. Per qualunque scelta iniziale, vale il principio di indeterminazione di Heisenberg

$$p \geq \frac{h}{\lambda} = \hbar k,$$

e in particolare l'uguaglianza vale per un autostato stazionario ψ^s. Se il potenziale V è nullo (per semplicità), la soluzione stazionaria è

$$\psi^s = \frac{1}{\sqrt{L}}\mathrm{e}^{ikx},$$

normalizzata su un dominio lungo L, che ha un momento ben definito e quindi, per il principio di indeterminazione, una posizione completamente indefinita. Infatti la densità di probabilità,

$$\psi^{s*}\psi^s = \frac{1}{L},$$

è uguale dappertutto. Un pacchetto è una sovrapposizione del tipo

$$\psi = \sum_i \psi^s_i = \frac{1}{\sqrt{L}}\sum_k c_k \mathrm{e}^{ikx},$$

dove vale $\sum_k |c_k|^2 = 1$, in modo che la norma sia

$$\int dx\,\psi^*\psi = \sum_{kk'} c^*_{k'}c_k \int dx\,\mathrm{e}^{i(k-k')x} = \sum_{kk'} c^*_{k'}c_k\,\delta(k-k') = \sum_k |c_k|^2 = 1.$$

Sotto l'azione dell'operatore hamiltoniano, il pacchetto non resterà invariato durante l'evoluzione a tempi lunghi, ma si decomporrà –avvicinandosi allo stato stazionario–nelle sue componenti ψ_s; questo perchè ognuna delle componenti ha diversa energia e si propaga nel tempo con una diversa fase temporale, e in particolare i coefficienti c cambiano nel tempo. Il pacchetto è costruito proprio per far sì che la

sua $|\psi|^2$ sia localizzata nello spazio (probabilità grande in una certa regione x), consistentemente con la larghezza finita della regione di momento in cui vivono i valori dei momenti delle onde componenti, cioè in breve l'indeterminazione sul momento.

Notiamo che la dispersione del pacchetto è dovuta alla struttura diffusiva dell'equazione. Se avessimo quantizzato con le stesse regole una energia relativistica (che contiene un termine E^2), avremmo ottenuto una equazione delle onde non dispersiva, del tipo

$$\partial_{xx}\psi = \partial_{tt}\psi + \dots.$$

Sono di questo tipo le prime (incomplete) generalizzazioni relativistiche, per esempio le equazioni di Pauli o di Klein-Gordon. Notiamo anche che partendo da una espressione dell'energia del tipo

$$cp = E$$

con c costante, otterremmo

$$ic\hbar\partial_x u = i\hbar\partial_t u \quad \Rightarrow \quad c\,\partial_x u = \partial u_t,$$

cioè l'equazione di advezione (discussa in Sez.4.5.1), che propaga una forma d'onda senza dispersione (se integrata opportunamente). L'analogo quantistico di questo caso è la equazione di Dirac per particelle senza massa.

4.3.2 *Crank-Nicolson come propagatore unitario*

Le diverse discretizzazioni dell'equazione dipendente dal tempo corrispondono a diverse approssimazioni per il propagatore. Ponendo $z{=}i\hat{H}t/\hbar$, nel limite di piccoli tempi possiamo usare uno sviluppo di Taylor, uno sviluppo per l'inverso, o un approssimante di Padé:

$$e^{-z} \underset{t\to 0}{\simeq} 1 - z; \tag{4.29}$$

$$e^{-z} = \frac{1}{e^z} \simeq \frac{1}{1+z}; \tag{4.30}$$

$$e^{-z} \simeq \frac{1 - z/2}{1 + z/2}. \tag{4.31}$$

Poiché il propagatore è unitario (cioè $A^\dagger A{=}I$), dovrebbero esserlo anche le sue approssimazioni. Posto a reale, osserviamo che

$$(1 - ia)^* = (1 + ia) \neq (1 - ia) \tag{4.32}$$

$$\left(\frac{1}{1+ia}\right)^* = \frac{1}{1-ia} \neq \frac{1}{1+ia} \tag{4.33}$$

$$\left(\frac{1-a}{1+a}\right)^* = \frac{1+a}{1-a} = \left(\frac{1-a}{1+a}\right)^{-1}, \tag{4.34}$$

cioè che solo la terza forma conserva l'unitarietà. (L'analogo, ad esempio, nel contesto delle ODE sono gli integratori simplettici.) La prima approssimazione corrisponde al metodo FTCS. Discretizzando le derivate, e rappresentando come al solito $\psi(x,t)\to\psi_j^n$ si ottiene

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\tau} = \frac{-\hbar^2}{2m}\left[\frac{\psi_{j+1}^n + \psi_{j-1}^n - 2\psi_j^n}{h^2}\right] + V_j\psi_j^n = \sum_{k=1}^{N} H_{jk}\psi_k^n \tag{4.35}$$

dove la matrice hamiltoniana è

$$H_{jk} = (\tilde{H})_{jk} = \frac{-\hbar^2}{2m}\left[\frac{\delta_{j+1,k} + \delta_{j-1,k} - 2\delta_{jk}}{h^2}\right] + V_j\delta_{jk} \tag{4.36}$$

con δ_{jk} il simbolo di Kronecker. Quindi, passando a una notazione vettoriale (i valori di ψ_j^n indicizzati da j sono le componenti di un vettore, come già visto nel contesto delle stabilità matriciale) e riarrangiando,

$$\vec{\psi}^{n+1} = \vec{\psi}^n + \frac{\tau}{i\hbar}\tilde{H}\vec{\psi}^n = \left[\mathcal{I} - \frac{i\tau}{\hbar}\tilde{H}\right]\vec{\psi}^n, \tag{4.37}$$

con $\mathcal{I}$ la matrice identità. Questa espressione, come si vede, corrisponde ad approssimare il propagatore con Eq.4.29, cioè in modo non unitario. Inoltre ha, come si è visto per l'equazione del calore, stabilità condizionata. Proviamo allora a usare un "BTCS", o, che è lo stesso, applichiamo $\hat{H}$ a ψ nel futuro:

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\tau} = \sum_{k=1}^{N} H_{jk}\psi_k^{n+1} \tag{4.38}$$

$$\vec{\psi}^{n+1} = \vec{\psi}^n + \frac{\tau}{i\hbar}\tilde{H}\vec{\psi}^{n+1}, \tag{4.39}$$

un'espressione implicita, dato che ψ^{n+1} dipende da sé stessa. Possiamo però trasformare come

$$(\mathcal{I} + \frac{i\tau}{\hbar}\tilde{H})\vec{\psi}^{n+1} = \vec{\psi}^n \tag{4.40}$$

e infine (indicando con $\tilde{A}^{-1}$ la matrice inversa di $\tilde{A}$)

$$\vec{\psi}^{n+1} = (\mathcal{I} + \frac{i\tau}{\hbar}\tilde{H})^{-1}\vec{\psi}^n \tag{4.41}$$

che chiaramente è l'approssimazione Eq.4.30 del propagatore, e quindi, di nuovo, non conserva l'unitarietà. Tuttavia risulta che lo schema Eq.4.41 è stabile incondizionatamente. Eq.4.41 è equivalente alla FTCS se $\tau \to 0$, dato che sviluppando si ha

$$\frac{1}{1+\epsilon} \simeq 1 - \frac{\epsilon}{[(1+\epsilon)^2]_{\epsilon=0}} + \cdots = 1 - \epsilon + \ldots$$

La differenza numerica tra Eq.4.37 e Eq.4.41 è che nel primo caso si fanno solo prodotti matrice×vettore, nel secondo caso anche una inversione di matrice. Questo non è un grosso problema, dato che l'inversione è fatta una sola volta all'inizio del calcolo (e la matrice è tridiagonale per un sistema finito; in condizioni periodiche al contorno ha elementi fuori-diagonale, anche se rimane comunque sparsa). Potrebbe diventarlo se la matrice, cioè l'operatore hamiltoniano, variasse nel corso del calcolo. Ad esempio, il potenziale potrebbe includere una parte elettrostatica dipendente dalla densità di carica $\rho(x)=e|\psi|^2$, che dipende dalla funzione d'onda.

Per ottenere uno schema che conservi l'unitarietà dell'evoluzione, si combinano i due approcci appena visti facendo la media dei secondi membri:

$$i\hbar\frac{\psi_j^{n+1} - \psi_j^n}{\tau} = \frac{1}{2}\sum_{k=1}^{N} H_{jk}[\psi_k^{n+1} + \psi_k^n] \tag{4.42}$$

$$\vec{\psi}^{n+1} = \vec{\psi}^n + \frac{\tau}{2i\hbar}\tilde{H}[\vec{\psi}^{n+1} + \vec{\psi}^n]. \tag{4.43}$$

Anche questo schema è implicito. Raccogliendo le ψ ai due tempi n e $n+1$ si ha

$$(\mathcal{I} + \frac{i\tau}{\hbar}\tilde{H})\vec{\psi}^{n+1} = (\mathcal{I} - \frac{i\tau}{\hbar}\tilde{H})\vec{\psi}^{n} \tag{4.44}$$

e infine

$$\vec{\psi}^{n+1} = (\mathcal{I} + \frac{i\tau}{2\hbar}\tilde{H})^{-1}(\mathcal{I} - \frac{i\tau}{2\hbar}\tilde{H})\vec{\psi}^{n} \tag{4.45}$$

che si riconoscere essere l'approssimante unitario di Padé del propagatore. Rispetto ai casi precedenti, oltre all'inversione della matrice c'è un prodotto matrice×matrice. Di nuovo, non troppo problematico per potenziale fisso e dimensioni non esagerate, dato che viene fatta una volta sola all'inizio dell'evoluzione.

Riassumendo, e a scanso di equivoci (dato che i segni dei vari termini possono confondere per via delle unità immaginarie nello specifico caso di Schrödinger), lo schema di Crank-Nicolson per un generica equazione di diffusione quale Eq.4.7 è definito dall'iterazione

$$\mathbf{u}_{n+1} = (\mathcal{I} - \mathcal{C})^{-1}(\mathcal{I} + \mathcal{C})\mathbf{u}_n \tag{4.46}$$

con

$$\mathcal{C} = \frac{\tau\kappa}{h^2}\left[\delta_{j+1,k} + \delta_{j-1,k} - 2\delta_{jk}\right].$$

Per l'eventuale inclusione di una κ dipendente dalla posizione è consigliabile riesaminare l'iterazione originale, considerando che $\kappa(x)$ deve moltiplicare $\partial_{xx}u(x)$ punto per punto.

4.3.3 *Pacchetto d'onda*

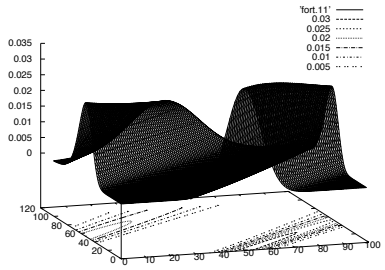
A titolo di esempio applichiamo il metodo di Crank-Nicolson alla propagazione di un pacchetto d'onda gaussiano

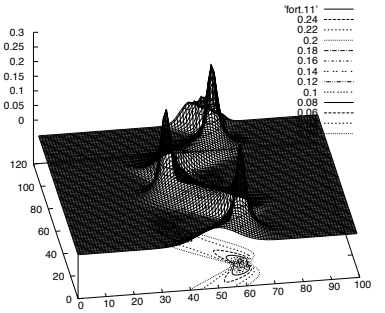
$$\psi^0 = \frac{1}{\sqrt{\sigma_0\sqrt{\pi}}}\exp\left[-\frac{(x-x_0)^2}{2\sigma_0^2} + ik_0x\right]$$

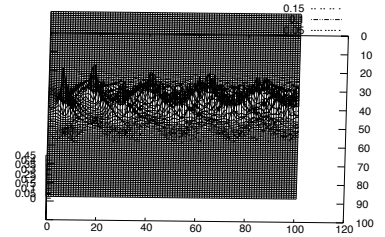
con k_0=$mv/\hbar$. Scegliamo v=0.5 e σ_0=0.1L con L la dimensione del dominio. Applichiamo condizioni periodiche e aggiungiamo un potenziale armonico

$$V(x) = A(x-d)^2, \qquad x \in (-L/2, L/2)$$

con d=0 e A variabile. Per A=0, il pacchetto è libero e si attenua gradualmente. Per piccolo A, il pacchetto si riflette sulle pareti di potenziale mantenendo la sua forma; per A crescente, la riflessione diventa più frequente, e ad alto A è praticamente un confinamento. Analogamente si può, ad esempio, studiare il moto in presenza di una barriera rettangolare e osservare la transizione da regime di riflessione a quello di tunneling in funzione dell'energia $\hbar k_0/2m$ del pacchetto. (Anche i fenomeni simili che compaiono nel caso di una buca sono interessanti.)







4.4 *Equazioni ellittiche e metodi iterativi*

Le equazioni ellittiche sono, come accennato, definite su domini chiusi e con condizioni sui bordi. La classica equazione di questo tipo è quella di Laplace-Poisson per il potenziale u generato da una densità di carica ρ, eventualmente nulla:

$$u_{xx} + u_{yy} = -\rho(x, y). \tag{4.47}$$

Il segno di ρ significa che se, ad esempio, $\rho>0$ in un dato punto la curvatura del potenziale è negativa in quel punto, cioè il potenziale ha un massimo come dev'essere.

La discretizzazione per le semplici geometrie 2D che consideriamo è

$$x_\ell = h_x \ell \quad (\ell = 1, \ldots, N_x); \quad y_m = h_y m, \quad (m =, \ldots 1, N_y),$$

ma qui assumiamo che il domino sia quadrato con passi uguali nelle due dimensioni, cioè h_x=h_y e N_x=N_y=N. Discretizzando le due derivate seconde, usando la discretizzazione appena menzionata, si ha

$$\frac{u_{i-1,j} + u_{i+1,j} - 2u_{ij}}{h^2} + \frac{u_{i,j-1} + u_{i,j+1} - 2u_{ij}}{h^2} = -\rho_{ij} \tag{4.48}$$

cioè

$$u_{i-1,j} + u_{i+1,j} - 4u_{ij} + u_{i,j-1} + u_{i,j+1} = -h^2\rho_{ij} \tag{4.49}$$

con i, j=2,N–1, e con condizioni al contorno descritte da

$$u_{ij} = U_{ij} \text{ per } i = 1 \text{ e } N, j = 1, \ldots, N, \text{ e per } i = 1, \ldots, N \text{ e } j = 1, N,$$

cioè valori della funzione fissati a certi valori **U** sui bordi destro, sinistro, alto, e basso della griglia. Eq.4.49 e le condizioni al bordo definiscono un problema matriciale (tridiagonale a bande) risolubile con metodi specializzati di algebra lineare, che costano tipicamente K^3 operazioni; qui K=N_xN_y, e in tre dimensioni K=$N_xN_yN_z$. Questo approccio può essere molto costoso o impossibile dato che K può essere molto grande (K=10^6 per una griglia equispaziata di 100 punti per direzione in 3D).

Una alternativa efficiente sono i metodi iterativi. Per formulare l'analogo di Eq.4.49 (ritroveremo in effetti esattamente la stessa espressione) per tali metodi, notiamo che le soluzioni dell'equazione del calore, Eq.4.7, per tempi lunghi tendono allo stato stazionario (p.es. a zero, come la soluzione Eq.4.8) e quindi si ha

$$\frac{\partial T}{\partial t} \underset{t\to\infty}{\longrightarrow} 0,$$

cioè allo stato stazionario l'equazione di diffusione diventa quella di Laplace (o se è presente un termine "sorgente" o "pozzo", tipo la densità, quella di Poisson). Questo suggerisce di usare uno schema di integrazione temporale dell'equazione del calore in 2D, per esempio una variante del metodo FTCS, per raggiungere iterativamente, a partire da una qualche (anche fantasiosa) soluzione approssimata, lo

stato stazionario. Il tempo in questo caso si riduce a un parametro senza effettivo significato fisico.
Discretizziamo dunque l'equazione di Poisson con termine temporale in 2D

$$u_{xx} + u_{yy} + \rho(x,y) = u_t,$$

indicando con n il parametro di iterazione (il "tempo"). Usando le solite formule FTCS (sempre per griglia quadrata ed equispaziata) otteniamo

$$\frac{u^n_{i-1,j} + u^n_{i+1,j} - 2u^n_{ij}}{h^2} + \frac{u^n_{i,j+1} + u^n_{i,j-1} - 2u^n_{ij}}{h^2} + \rho_{ij} = \frac{u^{n+1}_{i,j} - u^n_{ij}}{\tau} \tag{4.50}$$

e riarrangiando

$$u^{n+1}_{ij} = u^n_{ij} + \frac{\tau}{h^2}[u^n_{i-1,j} + u^n_{i+1,j} - 2u^n_{ij}] + \frac{\tau}{h^2}[u^n_{i,j-1} + u^n_{i,j+1}] + \tau\rho_{ij}. \tag{4.51}$$

Abbiamo visto che la condizione di stabilità è $\tau/h^2 \leq 1/2$ in 1D, ma in 2D risulta essere $\tau/h^2 \leq 1/4$. Scegliendo il massimo valore possibile si ha perciò

$$\begin{aligned} u^{n+1}_{ij} &= u^n_{ij} + \frac{1}{4}[u^n_{i-1,j} + u^n_{i+1,j} - 4u^n_{ij} + u^n_{i,j-1} + u^n_{i,j+1} + h^2\rho_{ij}]. \\ u^{n+1}_{ij} &= \frac{1}{4}[u^n_{i-1,j} + u^n_{i+1,j} + u^n_{i,j-1} + u^n_{i,j+1} + h^2\rho_{ij}]. \end{aligned} \tag{4.52}$$

Eq.4.52 è nota come iterazione di Gauss-Jacobi; la funzione in un dato punto all'iterazione successiva è (a parte l'azione della densità) la media della funzione nei punti laterali dello stencil (vedi p.es. Fig.4.7). Quindi data una approssimazione della funzione su tutta la griglia e le condizioni al bordo, è immediato generare la funzione all'iterazione successiva. Data la funzione a n=0 $\forall(i,j)$ si ottiene iterativamente una approssimazione sempre migliore: la convergenza è infatti garantita dall'analisi di stabilità matriciale (discussa in Sez.4.3 per il metodo FTCS): si può infatti mostrare che la matrice di iterazione è simmetrica e positiva definita, e ha massimo autovalore $\lambda_{\max} \propto 1/h^2 < 1$ "sempre".

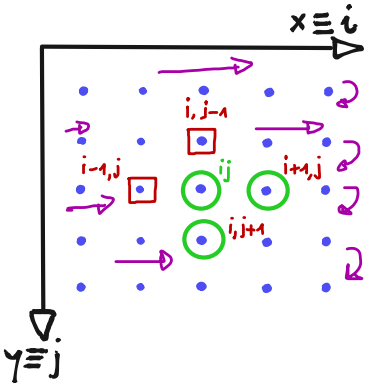


Figura 4.7: L'iterazione di Gauss-Seidel sfrutta il fatto che, scansionando la griglia dall'alto a sinistra verso destra e verso il basso, la funzione è già aggiornata nei punti rossi dello stencil, ma non in quelli verdi.

Vediamo ora il metodo di Gauss-Seidel. È un'iterazione accelerata che si ottiene osservando (Fig.4.7) che quando, durante la scansione della griglia (supponiamo, dall'alto a sinistra verso destra e verso il basso) all'iterazione n+1, si arriva ad aggiornare la funzione nel punto (i,j), la soluzione è già quella nuova (la n+1) nei punti $(i,j-1)$ e $(i-1,j)$, mentre è quella vecchia negli altri due punti dello stencil $(i+1,j)$ e $(i,j+1)$, oltre che nel punto (i,j) stesso. Questo suggerisce di modificare l'iterazione di Jacobi come

$$u^{n+1}_{ij} = \frac{1}{4}[u^{n+1}_{i-1,j} + u^n_{i+1,j} + u^{n+1}_{i,j-1} + u^n_{i,j+1} + h^2\rho_{ij}], \tag{4.53}$$

cioè usare le soluzioni nuove dove sono disponibili, "tirando" così la soluzione verso quei valori. Gauss-Seidel accelera notevolmente la convergenza rispetto a Jacobi, come si vede in Fig.4.8 (linea tratteggiata vs linea dotted) per una equazione di Laplace con condizioni di potenziale nullo su tre lati del dominio e non nullo sul quarto lato. Inoltre l'aggiornamento dei valori della funzione con quelli nuovi

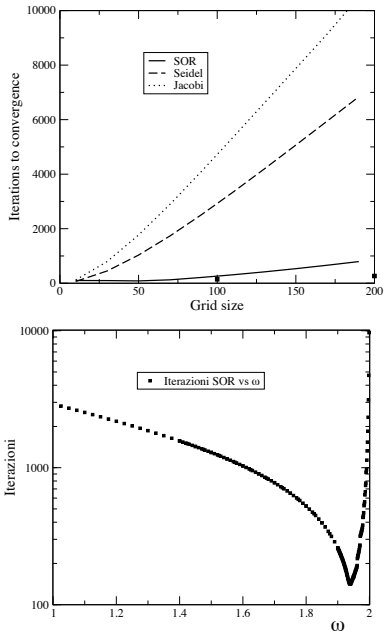


Figura 4.8: A sinistra, iterazioni per raggiungere convergenza in funzione delle dimensioni della griglia, per i metodi Jacobi (dotted), Seidel (tratteggio), e SOR (continuo; ω=1.9); i quadrati neri sono le iterazioni per N=100 e 200 ai rispettivi ω ottimali. A destra: iterazioni a convergenza per SOR in funzione del parametro di sovrarilassamento ω per una griglia 100×100, da cui risulta che il valore ottimale è $\omega_{\rm opt}$=1.94.

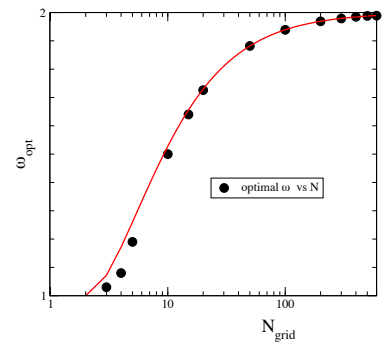
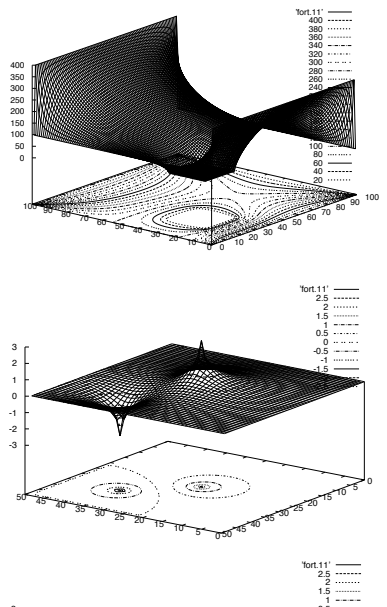


Figura 4.9: Parametro di sovrarilassamento ottimale in funzione della dimensione della griglia (calcolato: punti neri; analitico: curva continua).



può avvenire "in place". Alla locazione di memoria dove si salva la funzione u_{ij} si accede infatti solo tre volte per iterazione: una con u "vecchia", una con u "nuova", e una per aggiornarla.

In Fig.4.8, a sinistra, abbiamo presentato insieme a Jacobi e Seidel il risultato, chiaramente lusinghiero, del metodo SOR (successive over-relaxation). L'idea è "tirare" ulteriormente la soluzione attuale verso la convergenza "sovra-pesando" il termine di update (il secondo membro di Eq.4.53) rispetto al valore attuale:

$$u_{ij}^{n+1} = \frac{\omega}{4}[u_{i-1,j}^{n+1} + u_{i+1,j}^{n} + u_{i,j-1}^{n+1} + u_{i,j+1}^{n} + h^2\rho_{ij}] + (1-\omega)u_{ij}^{n} \quad (4.54)$$

dove il parametro ω misura il sovrarilassamento. I vari casi sono

- ω=0: non succede niente perchè u_{ij}^{n+1}=u_{ij}^{n};
- ω<1: sottorilassamento; la parte di update è minore di quella di Seidel e si reintroduce un po' della funzione "vecchia" in quella "nuova";
- ω=1: cade l'ultimo termine e si torna al caso di Seidel;
- $\omega\in$(1,2): sovrarilassamento, perchè sottraiamo una porzione della funzione u_{ij}^{n}, e sovra-pesiamo la parte di update;
- ω=2: divergenza.

Un calcolo diretto conferma la miglior prestazione di SOR, come mostra Fig.4.8, a destra: le iterazioni SOR (linea continua) sono un ordine di grandezza in meno di quelle del metodo di Seidel. Tuttavia, SOR va ancora meglio di così. Infatti, in Fig.4.8 (a sinistra) si è usato ω=1.9 per tutte le griglie, ma l'ω ottimale dipende da N. Nel caso della griglia 100×100 mostrato in Fig.4.8 a sinistra, risulta essere 1.94. Poichè il numero di iterazioni cala bruscamente vicino a $\omega_{\rm opt}$, in Fig.4.8 a sinistra le iterazioni SOR sono sovrastimate apprezzabilmente. I quadrati neri mostrano le (molto minori) iterazioni per N=100 e 200 al rispettivo $\omega_{\rm opt}$. Lo speed-up rispetto a Seidel per questo problema satura velocemente in N, ed è intorno a 34 (cioè SOR fa 34 volte meno iterazioni di Seidel). Metodi ancora migliori, ma più complessi, del SOR sono quelli ADI (Alternate Direction Iteration) e multigrid in uso ad esempio in fluidodinamica computazionale.

Aumentando le dimensioni della griglia, le iterazioni minime $I_{\rm min}$ di SOR salgono linearmente con N ad un tasso $\delta I_{\rm min}/\delta N \simeq 1.15$. Allo stesso tempo, $\omega_{\rm opt}$ converge asintoticamente a 2; in Fig.4.9 i valori calcolati (neri) sono confrontati con l'espressione analitica di $\omega_{\rm opt}$ per griglie quadrate ed equispaziate

$$\omega_{\rm opt} = \frac{2}{1+\sqrt{(1-\cos^2(\pi h))}} \left\{ \begin{array}{c} \stackrel{\longrightarrow}{h\to 0}\ 2 \\ \stackrel{\longrightarrow}{h\to \frac{1}{2}}\ 1 \end{array} \right\}$$

L'estremo superiore e inferiore in N sono equivalenti, rispettivamente, a infinite divisioni e, rispettivamente, a due divisioni dell'intervallo (nell'esempio, l'intervallo è unitario, quindi h=1/N). A piccoli N, il vantaggio di SOR su Seidel diventa marginale; tuttavia, $\omega_{\rm opt}$ sale molto rapidamente con N e così fa il guadagno di SOR – notiamo che la scala è logaritmica in N. Naturalmente in generale le prestazioni,

sia assolute che relative, dei diversi metodi sono dipendenti dal problema (occasionalmente, anche se molto raramente, SOR va peggio degli altri due).

A titolo di esempio mostriamo in Fig.6.9 il potenziale generato con i metodi iterativi per alcuni casi di interesse. Tutti questi grafici sono ottenuti con una griglia 50×50 per chiarezza. La convergenza richiede nell'ordine di 100 iterazioni SOR.

4.5 *Equazioni iperboliche*

L'equazione iperbolica più importante è naturalmente quella delle onde. Le onde sono oscillazioni temporali o spaziali (o ambedue) di quantità varie, ad esempio il livello dell'acqua attorno al livello medio del mare in presenza di onde in acqua profonda, o la posizione di una corda di chitarra pizzicata, o la densità dell'aria fatta vibrare dalle corde vocali o da un pistone. Consideriamo una porzione di corda spostata meccanicamente dall'equilibrio. La posizione lungo la corda è x, e il suo spostamento laterale dalla posizione di equilibrio è y, che è funzione di x e del tempo t. Ai capi di un suo tratto generico infinitesimo dx, la corda è soggetta a tensioni, cioè forze di richiamo, che non si cancellano in generale (lo fanno se il tratto di corda è localmente e momentaneamente diritta). La forza netta è

$$F_{\text{net}} = T(\sin\theta_1 - \sin\theta_2).$$

Sviluppando in serie per piccolo angolo, $\sin\theta\sim\theta$. Ma poichè anche la tangente ha lo stesso sviluppo al primo ordine, $\sin\theta\sim\tan\theta$. Poichè $\tan\theta$ è la pendenza locale di $y(x)$, cioè la sua derivata,

$$F_{\text{net}} \simeq T(\tan\theta_1 - \tan\theta_2) = T\left(\frac{\partial y}{\partial x}|_{x+dx} - \frac{\partial y}{\partial x}|_x\right) \simeq T\frac{\partial^2 y}{\partial x^2}dx,$$

dove nell'ultima equaglianza abbiamo linearizzato o, equivalentemente, applicato la definizione di derivata a $\partial y/\partial x$. Definita una densità lineare di massa μ, la massa del tratto di corda dx è $m{=}\mu\ dx$; l'equazione di Newton $F{=}ma$ ci dà direttamente l'equazione delle onde:

$$F = T\frac{\partial^2 y}{\partial x^2}dx = ma = \mu\, dx\frac{\partial^2 y}{\partial t^2} \Rightarrow \frac{\partial^2 y}{\partial x^2} = \frac{\mu}{T}\,\frac{\partial^2 y}{\partial t^2}$$

che riscriviamo come

$$\frac{\partial^2 y}{\partial x^2} = \frac{1}{c^2}\,\frac{\partial^2 y}{\partial t^2}. \tag{4.55}$$

La costante

$$v = \sqrt{\frac{T}{\mu}}$$

ha le dimensioni di una velocità, ed effettivamente è la velocità con cui la perturbazione si propaga sulla corda. La soluzione dell'equazione d'onda Eq.4.55 deve infatti soddisfare una sola proprietà, e precisamente deve dipendere da x e t come

$$y(x,t) \stackrel{\text{def}}{=} y(x-ct).$$

Definiamo y=G e z=x–ct, e notiamo che

$$\frac{\partial y}{\partial x} = \frac{\partial G}{\partial z}\frac{\partial z}{\partial x} = \frac{\partial G}{\partial z} \;\Rightarrow\; \frac{\partial^2 y}{\partial x^2} = \frac{\partial^2 G}{\partial z^2}$$

e d'altra parte

$$\frac{\partial y}{\partial t} = \frac{\partial G}{\partial z}\frac{\partial z}{\partial t} = \frac{\partial G}{\partial z}(-c) \;\Rightarrow\; \frac{\partial^2 y}{\partial t^2} = \frac{\partial^2 G}{\partial z^2}(-c)^2 \;\Rightarrow\; \frac{1}{v^2}\frac{\partial^2 y}{\partial t^2} = \frac{\partial^2 G}{\partial z^2},$$

ovvero: una funzione generica di x e t del tipo $y(x–ct)$ soddisfa l'equazione d'onda. Ad esempio

$$y = \exp\left[-\frac{(x-ct)^2}{x_0^2}\right] \tag{4.56}$$

è una possibile soluzione, ma

$$y \stackrel{?}{=} \exp\left[-\frac{x^2-(ct)^2}{x_0^2}\right]$$

non lo è. Dato che solo c^2 compare nell'equazione, ambedue i segni di v, cioè i due versi di moto, sono validi. Scegliendo $+v$ la forma d'onda si muoverà verso destra, e viceversa per $–c$. Una soluzione generale sarà una combinazione lineare (l'equazione, ricordiamo, è lineare) del tipo $Ay(x-vt)$+$By(x+vt)$. Notiamo che la velocità è costante e indipendente dalla lunghezza d'onda: questo è il caso delle onde non-dispersive. Se c=$c(k)$, che si chiama appunto relazione di dispersione, l'onda sarà invece dispersiva. Abbiamo in effetti già discusso implicitamente questo caso a proposito dell'equazione di Schrödinger, dove la velocità è $c\simeq k$, accennando che il caso "para-schrödingeriano" con velocità costante fornisce l'equazione di advezione, discussa in dettaglio qui sotto.

Dunque, la nostra equazione è del tipo

$$u_{tt} = c^2 u_{xx}, \tag{4.57}$$

dove c è la velocità. Poichè l'equazione contiene due derivate seconde è necessario imporre le condizioni iniziali per la funzione e per la derivata prima temporale,

$$u(x,0) = f(x), \quad u_t(x,0) = g(x), \tag{4.58}$$

dove f e g sono funzioni della posizione (discretizzabili quindi come f_i, g_i per i=0,N). Se il dominio spaziale è $x\in(-\infty,+\infty)$ la soluzione è

$$u(x,t) = \frac{1}{2}[f(x+ct) + f(x-ct) + \int_{x+ct}^{x-ct} g(s)\,ds] \tag{4.59}$$

Una proprietà di questa soluzione è che a ogni (x,t) finito, il valore di u è determinato dai valori iniziali entro l'intervallo [x–ct,x+ct]. Questo intervallo, detto domino di dipendenza, è in Fig.4.11. Le linee (x–ct,0)→(x,t) e (x+ct,0)→(x,t), sono la caratteristica destra e sinistra del punto (x,t). Le stesse linee si estendono oltre (x,t), e definiscono la regione di influenza di (x,t): il valore di u fuori dalla regione ombreggiata è indipendente da $u(x,\,t)$.

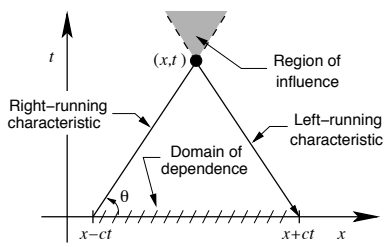


Figura 4.11: Dominio di dipendenza e regione di influenza in una equazione iperbolica.

Discretizzando le derivate seconde, si ha

$$\frac{1}{\tau^2}(u_m^{n-1} - 2u_m^n + u_m^{n+1}) = \frac{c^2}{h^2}(u_{m-1}^n - 2u_m^n + u_{m+1}^n), \tag{4.60}$$

da cui riarrangiando

$$u_m^{n+1} = 2u_m^n + \frac{c^2\tau^2}{h^2}(u_{m-1}^n - 2u_m^n + u_{m+1}^n) - u_m^{n-1} \tag{4.61}$$

e infine, posto ρ=τ/h,

$$u_m^{n+1} = 2(1 - c^2\rho^2)u_m^n + c^2\rho^2(u_{m+1}^n + u_{m-1}^n) - u_m^{n-1} \tag{4.62}$$

Chiaramente questo è uno schema esplicito a tre livelli, cioè con dipendenza al tempo n+1 dai tempi n e $n-1$. Se n=1, otteniamo i valori per n=2 in funzione dei dati iniziali u_0 e di u_1, che però non sappiamo. Con una procedura abbastanza semplice si trova [1] che

$$u_m^1 = (1 - c^2\rho^2)u_m^0 + \frac{c^2\rho^2}{2}(u_{m+1}^0 + u_{m-1}^0) + \tau g_m$$

dove $u_m^0 \equiv f_m$ e g_m sono le due condizioni iniziali Eq.4.58 in forma discretizzata. Ora abbiamo u^0 e u^1 e possiamo ottenere direttamente u^2.

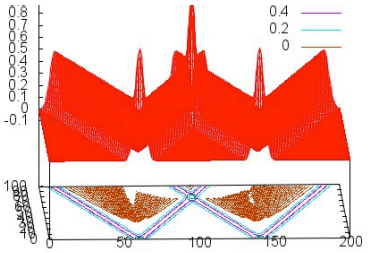


Figura 4.12: La soluzione dell'equazione d'onda per una forma d'onda iniziale costituita da due gaussiane. Le si vede separarsi in due e attraversarsi interferendo.

[1]

Si può mostrare [2], con una analisi (detta di von Neumann) dimostrata sotto per l'equazione di advezione, che la stabilità è determinata dalla condizione di Courant-Friedrichs-Lewy

[2]

$$c\tau \leq h, \tag{4.63}$$

che in sostanza prescrive la consistenza della risoluzione temporale e di quella spaziale per una data velocità dell'onda (C=$c\tau/h$ è detto numero di Courant). Si può mostrare anche che questo schema di discretizzazione è consistente con l'equazione originale, nel senso che *a*) nel limite τ, $h \to 0$ si riottiene l'equazione originale, e *b*) la regione di influenza della discretizzazione e della equazione sono uguali. Chiaramente questo criterio è lo stesso che abbiamo derivato euristicamente per il metodo FTCS per l'equazione di diffusione.

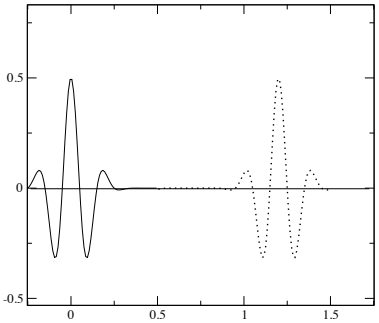


Figura 4.13: Soluzione teorica dell'equazione di advezione: la forma d'onda iniziale si propaga invariata.

4.5.1 *Stabilità di von Neuman: l'equazione di advezione*

Veniamo ora all'equazione del primo ordine detta di advezione,

$$u_t + vu_x = 0 \tag{4.64}$$

con v>0, costante, che è importante, oltre che dimostrativamente, perchè sue varianti non-lineari, l'equazione di Burgers (discussa più oltre) e l'equazione di Korteveg-de Vries, mostrano interessanti effetti come lo svilupparsi di onde d'urto e di onde solitarie non-dispersive (solitoni). Definita una forma d'onda iniziale $u(x,0)$=$f(x)$, la soluzione è

$$u(x,t) = f(x - vt), \tag{4.65}$$

cioè la forma d'onda si propaga invariata (Fig.4.13). Se discretizziamo, come per l'equazione del calore, usando la formula forward per la

derivata temporale e la formula centrata per quella spaziale, abbiamo

$$\frac{a_i^{n+1} - a_i^n}{\tau} = -v\frac{a_{i+1} - a_{i-1}}{2h}, \tag{4.66}$$

da cui l'espressione esplicita FTCS

$$a_i^{n+1} = a_i^n - \frac{v\tau}{2h}(a_{i+1} - a_{i-1}). \tag{4.67}$$

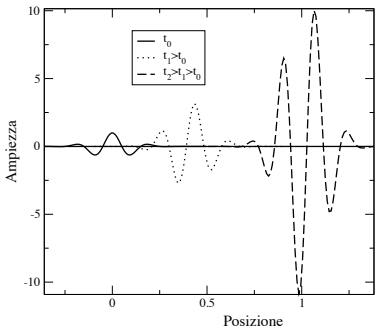


Figura 4.14: Soluzione FTCS dell'equazione di advezione: la forma d'onda iniziale (a sinistra) è amplificata in modo spurio.

In Fig.4.14 sono rappresentate le forme d'onda ottenute da FTCS a intervalli di tempo successivi (da sinistra a destra). Si osserva che questo schema di iterazione amplifica la forma d'onda in modo spurio, ed è quindi instabile. Questo succede per qualunque combinazione di τ e h.

Per analizzare questa instabilità usiamo l'analisi di von Neumann. Esprimiamo la soluzione, separando le variabili, come

$$a(x,t) = A(t)e^{ikx} \rightarrow a_j^n = A_n e^{ikjh} \tag{4.68}$$

dove j è un intero. La forma d'onda propagata sarà

$$a_j^{n+1} = A_{n+1}e^{ikjh} = \xi A_n e^{ikjh}, \tag{4.69}$$

dove i coefficienti A sono dipendenti dal tempo. Questo definisce implicitamente il fattore di amplificazione $\xi = A_{n+1}/A_n$. Se $\xi > 1$, l'integrazione è instabile. Inserendo la soluzione di prova Eq.4.68 nella formula FTCS otteniamo

$$\xi A_n e^{ikjh} = A_n e^{ikjh} - \frac{v\tau}{2h}A_n[e^{ik(j+1)h} - e^{ik(j-1)h)}]. \tag{4.70}$$

Raccogliendo a fattore la soluzione Eq.4.68, si ha una espressione per il fattore di amplificazione:

$$\xi = 1 - \frac{v\tau}{2h}(e^{ikh} - e^{-ikh}) = 1 - \frac{v\tau}{h}i\sin(kh) \tag{4.71}$$

il cui modulo è ($\forall\, k > 0$, τ, v, h)

$$|\xi| = \sqrt{1 + \left(\frac{v\tau}{h}\right)^2 \sin^2(kh)} > 1. \tag{4.72}$$

Quindi il metodo FTCS è sempre instabile. È possibile ottenere stabilità condizionata con la discretizzazione di Lax (vincitore anni fa del premio Wolf – il premio Nobel della matematica– proprio per i suoi contributi alle equazioni differenziali discretizzate):

$$a_i^{n+1} = \frac{1}{2}\left(a_{i+1}^n + a_{i-1}^n\right) - \frac{v\tau}{2h}(a_{i+1} - a_{i-1}). \tag{4.73}$$

Applicando come prima l'analisi di von Neumann, si ha

$$\xi A_n e^{ikjh} = \frac{A_n}{2}\left(e^{ik(j+1)h} + e^{ik(j-1)h}\right) - \frac{v\tau}{2h}A_n[e^{ik(j+1)h} - e^{ik(j-1)h)}],$$

e quindi

$$\xi = \cos(kh) - \frac{v\tau}{h}i\sin(kh) \tag{4.74}$$

il cui modulo ($\forall\ k{>}0$, τ, v, h)

$$|\xi| = \sqrt{\cos^2(kh) + \left(\frac{v\tau}{h}\right)^2 \sin^2(kh)} \tag{4.75}$$

soddisfa la condizione di convergenza $|\xi|{\leq}1$ se $\beta{=}c\tau/h{\leq}1$: di nuovo la condizione di Courant-Friedrichs-Levy. Usando questa discretizzazione si osserva che per $\beta{<}1$ si ha damping, mentre per $\beta{=}1$ la forma d'onda di propaga correttamente.

4.6 *Equazione di Burgers e onde d'urto*

L'equazione di Burgers

$$u_t + cu \cdot u_x = 0 \tag{4.76}$$

è una equazione non lineare, come si vede dalla forma equivalente

$$u_t + \epsilon[u^2/2]_x = 0, \tag{4.77}$$

e dà luogo, come sempre nei casi non-lineari, a fenomeni compicati tra cui le onde d'urto. In sostanza, si tratta di una equazione di tipo advezione, ma con velocità che dipende dall'ampiezza, cosicchè porzioni di onda più "alte" tendono ad accelerare rispetto a quelle più basse. Usiamo un metodo analogo a quello di Lax, mantenendo le derivate seconde nello sviluppo

$$u(x, t+\Delta t) \simeq u(x,t) + \frac{\partial u}{\partial t}\Delta t + \frac{1}{2}\frac{\partial^2 u}{\partial t^2}\Delta t^2 \tag{4.78}$$

della funzione per piccoli tempi. Le derivate temporali possono essere ottenute dall'espressione dell'equazione stessa, la prima direttamente, e la seconda con un po' di lavoro:

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} &= \frac{\partial}{\partial t}\left[-c\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right)\right] = -c\frac{\partial}{\partial x}\frac{\partial}{\partial t}\left(\frac{u^2}{2}\right) \\ &= -c\frac{\partial}{\partial x}\left[u\frac{\partial u}{\partial t}\right] = -c^2\frac{\partial}{\partial x}\left[u\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right)\right]. \end{aligned} \tag{4.79}$$

Sostituendo in Eq.4.78 si ottiene

$$u(x, t+\Delta t) \simeq u(x,t) + c\Delta t\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) + \frac{(\Delta t)^2}{2}c^2\frac{\partial}{\partial x}\left[u\frac{\partial}{\partial x}\left(\frac{u^2}{2}\right)\right]. \tag{4.80}$$

Si discretizza rimpiazzando le derivate esterne con differenze centrate di passo $\Delta x/2$, e quelle interne con differenze in avanti di passo Δx (dettagli in [3], Eq.19.11 sgg.). Raccogliendo tutti i pezzi si ha alla fine [3]

$$\begin{aligned} u_i^{n+1} &= u_i^n - \frac{\beta}{4}\left((u_{i+1}^n)^2 - (u_{i-1}^n)^2\right) \\ &+ \frac{\beta^2}{8}\left(u_{i+1}^n + u_i^n\right)\left((u_{i+1}^n)^2 - (u_i^n)^2\right) \times \\ &\times \left(u_i^n + u_{i-1}^n\right)\left((u_i^n)^2 - (u_{i-1}^n)^2\right), \end{aligned} \tag{4.81}$$

dove $\beta{=}ch/\tau$ è sempre il numero di Courant. Utilizzando una condizione iniziale gaussiana si ottiene la soluzione in Fig. 4.15. Si nota che la forma d'onda si distorce nella direzione del moto formando una parete verticale.

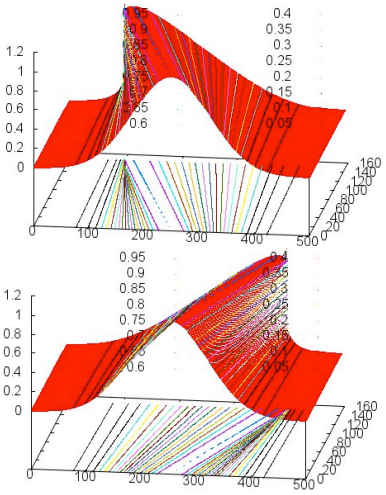
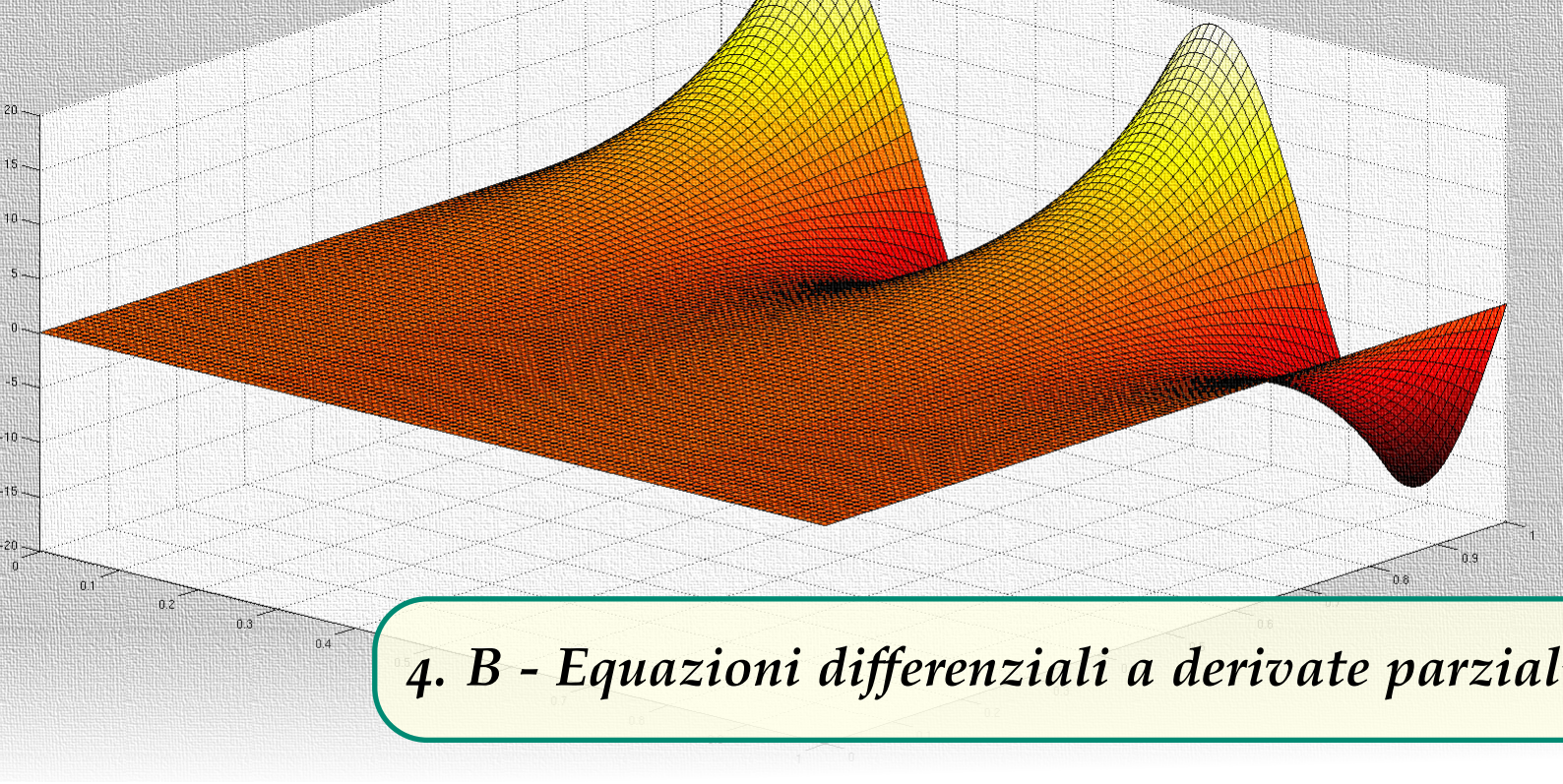


Figura 4.15: Una condizione iniziale gaussiana dell'equazione di Burgers si sviluppa in una onda d'urto (con velocità negativa, a sinistra, e positiva, a destra).



4. B - Equazioni differenziali a derivate parzial

4.1 Flusso di calore

In questo tutorial scriveremo un codice per risolvere una equazione differenziale alle derivate parziali di tipo parabolico, come quella che descrive l'andamento nello spazio e nel tempo della temperatura di un certo sistema unidemensionale:

$$\kappa \frac{d^2T}{dx^2} = \frac{dT}{dt}$$

Come visto nelle dispense (pag.) il metodo FTCS si ottiene discretizzando le derivate della precedente equazione e riordinando i termini si arriva alla seguente relazione:

$$T_{n+1} = AT_n, t_s = \frac{h^2}{2\kappa}$$

Come si nota, é quindi possibile trovare la temperatura in tutti i punti i del nostro sistema unidimensionale ad un certo istante $n+1$, conoscendo la temperatura al tempo precedente nello stesso punto i e in quelli adiacenti $i-1, i+1$. È perciò necessaria la conoscenza della temperatura in certo istante iniziale, per calcolare la sua evoluzione ad istanti successivi e dobbiamo inoltre fissare le condizioni al contorno, quindi decidere cosa accade alle estremità del nostro sistema. L'implementazione dello schema FTCS consiste nel tradurre in codice la relazione precedente e iterarla all'interno di un ciclo `for` un certo numero di volte. Vediamo il codice di seguito:

```
for time in range(1,tstep):
  for i in range(1,N-1):
  T[i,time]=T[i,time-1]+0.5*tau/ts*(T[i-1,time-1]+T[i+1,time-1]-2*T[i,time-1])
```

Vediamo in dettaglio queste righe di codice:

- la temperatura del sistema ai vari istanti è memorizzata nella matrice `T`, dove l'indice di riga i rappresenta la posizione mentre l'indice di colonna time rappresenta l'istante temporale;

- abbiamo due cicli for: uno sul tempo che parte da 1 quindi dall'istante successivo all'istante iniziale (per il quale conosciamo già `T`); l'altro sullo spazio che va dalla posizione 1 alla N-1, quindi esclude le posizioni ai bordi che stiamo cosi fissando costanti al valore iniziale, implementando di fatto le condizioni di Dirichlet.

Al solito, per poter eseguire correttamente le righe precedenti dobbiamo definire le condizioni iniziali:

```
N=100
tstep=1000
T=zeros((N,tstep))

#Temperatura iniziale
T[50,0]=10.

ts=1
tau=0.9*ts
```

Fissiamo quindi le dimensioni del sistema con `N` e la durata temporale `tstep` della nostra simulazione, che risultano anche essere le dimensioni della matrice `T`. Decidiamo anche la costante `ts` e l'intervallo temporale tau in maniera che esso sia leggermente piu piccolo, per avere la stabilità dello schema di integrazione.

La temperatura iniziale è stata posta a 10 in un singolo punto al centro del sistema, simulando cosi uno spike di temperatura. Tutto il resto del sistema è a temperatura nulla.

Siamo ora pronti per eseguire il nostro programma e vedere come evolve la temperatura del nostro sistema. Una volta eseguito dovremo fare un grafico della temperatura in funzione del tempo e dello spazio perciò necessitiamo di un grafico 3D. Per otternere questo ci servono le seguenti linee di codice:

```
from mpl_toolkits.mplot3d import Axes3D

ax = gca(projection='3d')
gridx , gridy = meshgrid(range(tstep),range(N))
ax.plot_wireframe(gridx,gridy,T,cstride=10,rstride=2)
```

Vediamo in dettaglio il significato delle precedenti linee:

- importiamo il tool Axes3D per poter disegnare grafici in 3D;
- creiamo la figura con i 3 assi X,Y,Z;
- mediante `meshgrid()` creiamo due matrici `gridx` e `gridy` che ci danno le coordinate x,y di tutti i punti di cui vogliamo rapresentare la coordinata z; in input alla funzione diamo due sequenze di lunghezza pari al numero di righe e colonne, `N` e `tstep` nel nostro caso, che rappresentano i punti negli assi x e y.
- infine, disegnamo il grafico mediante `plot_wireframe()` dando in input le matrici `gridx`, `gridy` e `T`. Se la matrice da rappresentare è grande come nel nostro caso, è conveniente limitare il numero di punti realmente disegnati usando cstride e rstride che fissano ogni quante x e y disegnare.

Per regolare meglio il range di valori sia in x che in y, quindi posizione e tempo nel nostro caso, e avere un grafico più comprensibile, ho usato le seguenti righe:

```
from mpl_toolkits.mplot3d import Axes3D

ax = gca(projection='3d')
gridx , gridy = meshgrid(range(2,200),range(0,N,2))
ax.plot_wireframe(gridx,gridy,Temp[::2,2:200])
```

Con `range(2,200)` ho creato selezionato un range temporale che mi esclude i due istanti iniziali e che considera solo i successivi 200 istanti. Con `range(0,N,2)` disegno un solo punto su due lungo l'asse x.

Eseguendo queste righe otteniamo il grafico mostrato in 4.1.

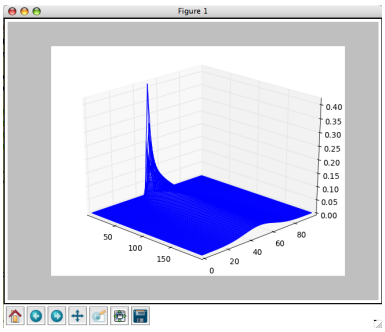


Figura 4.1:

Si nota come il picco di temperatura iniziale si abbassa e si allarga lungo x con il passare del tempo, segno che il calore inialmente generato al centro del sistema, diffonda in entrambe i lati ed fuoriesca dal sistema ai bordi (condizioni di Dilichlet). Perciò, per tempi molto lunghi, la temperatura torna a zero su tutto il sistema.

Proviamo ora ad implementare le condizioni di Neumann, per le quali la derivata della temperatura rispetto a x negli estremi e' nulla:

$$\left.\frac{dT}{dx}\right|_{x=0,L} = 0.$$

Queste condizioni impongono un flusso di calore nullo agli estremi, per cui se supponiamo di avere un certa quantità di calore all'istante iniziale concentrata in una zona del sistema in esame, questa si distribuisce sino a che la temperatura raggiunge un valore costante per tutto il sistema. Per implementare queste condizioni al contorno dobbiamo imporre che$T_0^n = T_1^n; T_N^n = T_{N-1}^n$, aggiungendo queste due righe al codice precedente:

```
for time in range(1,tstep):
  #ciclo for su i
  ...
  T[0,time]=T[1,time]
  T[99,time]=T[98,time]
```

Quindi una volta calcolata la temperatura su tutti i punti interni, aggiorniamo gli estremi imponendo la temperatura uguale a quella dei loro punti adiacenti.

Ora vediamo cosa succede se imponiamo una temperatura iniziale più alta in una zona centrale del sistema lunga 30 punti e lasciando andare la simulazione per 3000 passi:

```
N=100
tstep=3000
Temp=zeros((N,tstep))
Temp[35:65,0]=500.
```

Il risultato è mostrato in Fig. 4.2

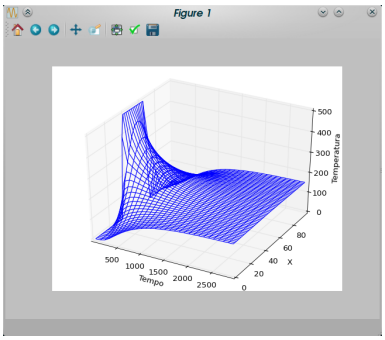


Figura 4.2:

Come vediamo il calore iniziale si distribuisce aumentanto la temperatura delle zone inizialmente a zero gradi, sino a che questa non diventa uniforme per tutto il sistema.

Per migliorare il grafico si può usare la funzione `plot_surface()` come ho fatto qui:

```
ax.plot_surface(gridx,gridy,Temp,cmap=cm.coolwarm,
      vmax=250,linewidth=0,rstride=2, cstride=100)
```

Vediamo le opzioni usate per questa funzione:

- `cmap=cm.coolwarm`, permette di aggiungere un colore alle singole caselle della superficie in base al valore della loro coordinata z; la particolare colorazione è data da `cm.collwarm`, una delle moltissime palette di colori a disposizione;
- `vmax=250`, il valore massimo sopra il quale il colore non cambia;
- `linewidth=0`, le linee di confine delle caselle della superficie hanno spessore o (appaiono in bianco);
- `cstride` e `rstride`, limitano i punti realmente disegnati e quindi la grandezza delle caselle.

In figura 4.3 è riportato il risultato.

Invito a sperimentare valori diversi per capire meglio il significato dei parametri usati.

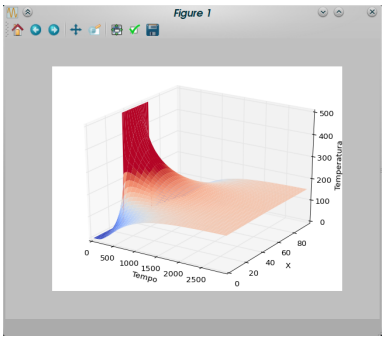


Figura 4.3:

4.2 *Potenziali elettrostatici*

In questo tutorial scriveremo un codice per la risoluzione di equazioni differenziali ellitiche come quella nota in fisica come eq. di Laplace-Poisson per un potenziale u generato da una carica $\rho(x,y)$:

$$u_{xx} + u_{yy} = \rho(x,y)$$

I metodi iterativi costituiscono un modo efficiente per calcolare la soluzione di queste eqs. Come visto nelle dispense, si tratta di applicare il metodo FTCS alla eq. di Poisson eguagliata ad un termine temporale:

$$u_{xx} + u_{yy} + \rho(x,y) = u_t;\ u_t = \frac{\partial u}{\partial t} \underset{t\to\infty}{\longrightarrow} 0.$$

Discretizzando otteniamo la seguente espressione nota come iterazione di Gauss-Jacobi:

$$u_{ij}^{n+1} = \frac{1}{4}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n} + h^2\rho_{ij}\right);$$

come notiamo ci permette di calcolare il potenziale all'iterazione successiva in un certo punto, facendo la media del potenziale nei punti attorno all'iterazione precedente. Questo ci dice anche che per iniziare l'iterazione abbiamo bisogno di sapere le condizioni al contorno della griglia e il potenziale iniziale su tutta la griglia.

```
for i in range(1,N-1):
  for j in range(1,N-1):
    #Jacobi
    U1[i,j] = 0.25*(U[i-1,j] + U[i+1,j] + U[i,j+1] + U[i,j-1])
```

Questa rappresenta la prima iterazione. Poichè noi cerchiamo una soluzione al limite, per la quale la variazione del potenziale sia nulla, dovremo implementare un modo per definire la convergenza. Un metodo rapido è quello di calcolare la media del poteziale su tutta la griglia ad ogni iterazione e confrontarla con il valore medio trovato alla iterazione precedente. Quando questa differenza è minore di un certo valore potremo considerare la soluzione trovata quella finale. Il codice seguente esegue ciò che vogliamo:

```
conv=1
mean_U0=mean(U)
niter=0

while conv > 1E-04:
  #ciclo for i
    #ciclo for j
  conv = abs(mean(U1)-mean_U0)
  mean_U0=mean(U1)
  U=U1.copy()
  niter+=1
```

Vediamo in dettaglio il significato di queste righe:

- le iterazioni vengono fatte mediante l'uso di un ciclo while, il quale esegue le righe al suo interno (quelle indentate) sintanto che la condizione conv>1E-04 è verificata;
- è necessario quindi aggiornare il valore di conv, quindi la differenza tra le medie a iterazioni successive;
- la funzione `mean(arr)` calcola la media su tutti i valori dell'array in input;
- conserviamo il valore attuale della media in `mean_U0`;
- conserviamo il potenziale attuale in `U` copiando `U1` con la funzione `copy()`;
- poichè non sappiamo a priori quante iterazioni saranno eseguite le contiamo incrementando la variabile niter;
- nelle prime righe impostiamo i valori iniziali per la media, il parametro di convergenza e il numero di iterazioni, `mean_U0`, `conv`, `niter` rispetivamente.

Non ci crederete ma questo è quanto basta. Anzi, possiamo anche semplificare ulteriormente, se utilizziamo il metodo di Gauss-Seidel:

$$u_{ij}^{n+1} = \frac{1}{4}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n} + u_{i,j-1}^{n+1} + h^2\rho_{ij}\right);$$

che sfrutta il fatto che nei punti i-1 e j-1 il potenziale è gia stato calcolato. Questo oltre a velocizzare la convergenza, semplifica anche il codice perche possiamo usare la stessa matrice leggendo le nove componenti e scrivendo le nuove. Vediamo come il codice precedente diventa:

```
conv=1
mean_U0=mean(U)
niter=0
```

```
while conv > 1E-04:
  for i in range(1,N-1):
    for j in range(1,N-1):
      #Seidel
      U[i,j] = 0.25*(U[i-1,j] + U[i+1,j] +
               U[i,j+1] + U[i,j-1])
  conv = abs(mean(U)-mean_U0)
  mean_U0=mean(U)
  niter+=1
```

Resta quasi tutto invariato, tranne l'uso della sola matrice U nei cicli i,j, e non abbiamo quindi bisogno di salvare il potenziale ad ogni iterazione.

A questo punto implementiamo anche il metodo SOR, semplicemente traducendo in codice la seguente iterazione:

$$u_{ij}^{n+1} = \frac{\omega}{4}\left(u_{i+1,j}^{n} + u_{i-1,j}^{n+1} + u_{i,j+1}^{n} + u_{i,j-1}^{n+1} + h^2\rho_{ij}\right) + (1-\omega)\,u_{ij}^{n};$$

in questo modo:

```
#SOR
U[i,j] = W*0.25*(U[i-1,j] + U[i+1,j] + U[i,j+1] +
     U[i,j-1])+ (1-W)*U[i,j]
```

che andrà a sostituire la riga n del codice precedente. Per questo metodo dovremo definire il peso W nel range $[1,2]$, solitamente vicino a 2.

Ora che abbiamo implentato i tre metodi, fissiamo le dimensioni del sistema, le condizioni al contorno e un valore iniziale del potenziale per tutta la griglia e confrontiamo il numero di iterazioni necessarie per la convergenza dei tre metodi:

```
N=100
U=zeros((N,N))
U[:,0]=10.

#Jacobi
#U1=U.copy()
#SOR
#W.19
```

Usiamo una griglia 100x100 e supponiamo di avere un potenziale fisso pari a 10 lungo un lato e zero lungo gli altri tre lati. Stiamo anche partendo da un potenziale nullo su tutto il resto della griglia. Eseguendo i tre metodi separatamente, con queste condizioni iniziali i tre metodi eseguono il seguente numero di iterazioni:

```
#Jacobi

#Seidel

#SOR
```

che possono essere facilmente confrontati con il grafico in figura 27. Se volessimo replicare interamente quel grafico, basterebbe racchiudere il nostro codice dentro un ciclo `for` come questo:

```
niter_list=[]
for N in range(20,200,20):
  #codice precedente per uno dei 3 metodi
  niter_list.append(niter)

plot(range(20,200,20),niter_list)
```

Salviamo il numero di iterazioni per ogni valore di N mediante la lista `niter_list`, su cui aggiungiamo i nuovi valori mediante la funzione `append()`; il plot è eseguito al solito modo.

Sfruttando un ciclo simile ma fatto sul peso W e mantenendo la griglia fissa su 100, possiamo riprodurre il secondo grafico in fig 27, dove si mostra l'andamento del numero di terazioni con il peso:

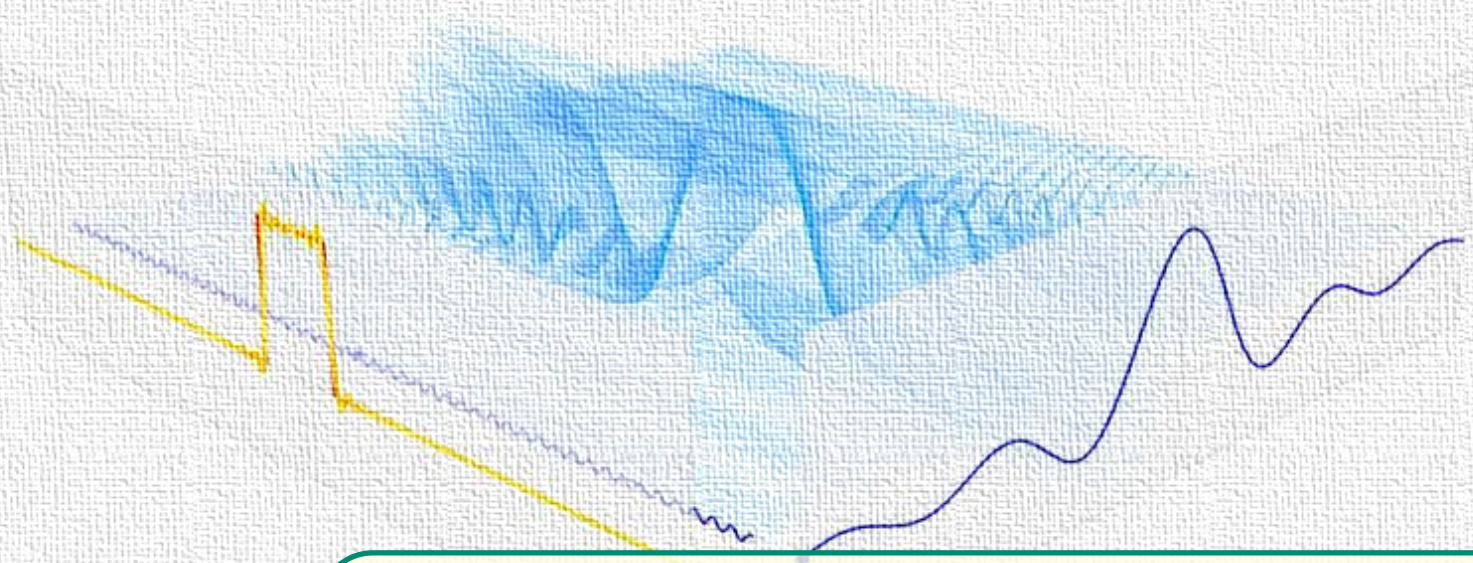
```
niter_list=[]
for W in arange(1,2,0.02):
  N=100
  #codice precedente per SOR
  niter_list.append(niter)

plot(arange(1,2,0.02),niter_list)
```

In questo caso usiamo la funzione `arange(start,end,step)` che in maniera simile alla funzione `range()` fornisce un array con valori nel range $[start, end]$, ma con step che in questo caso può essere un numero non intero.

I tre metodi, seppur con diverso numero di iterazioni producono la stessa soluzione finale. Vediamone qualcuna prodotta con le seguenti condizioni iniziale e usando il metodo piu rapido, come visto il SOR:

FIGURE MANCANTI DI ALCUNI POTENZIALI DI ESEMPIO



5. A - Fourier transform

Molti processi fisici (e le funzioni che li rappresentano) possono essere utilmente descritti in dominî duali, le cui variabili sono il reciproco l'una dell'altra – tipicamente tempo t e frequenza f, o posizione x e vettore d'onda k. Ad esempio, un segnale che varia nel tempo, o una funzione della posizione, possono essere rappresentati come sovrapposizione di onde, di opportuna ampiezza, oscillanti nel tempo con specifica frequenza o nello spazio con specifici vettori d'onda, etc. La trasformata di Fourier (FT nel seguito per brevità) è una operazione che trasforma una funzione della variabile di un dominio in una funzione della variabile del dominio duale. La funzione e la sua trasformata di fatto sono due diverse maniere di rappresentare la stessa informazione. La definizione è

$$H(f) = \int_{-\infty}^{\infty} h(t)\, e^{2\pi i f t}\, dt \tag{5.1}$$

$$h(t) = \int_{-\infty}^{\infty} H(f)\, e^{-2\pi i f t}\, df \tag{5.2}$$

dove H è la trasformata e h la funzione originale, o antitrasformata (anti-FT). La convenzione che usiamo qui è la più simmetrica, oltre che comoda in termini di fattori numerici da ricordare. Se la variabile è il tempo in secondi, la variabile duale è la frequenza in Hz$\equiv$s^{-1}, e similmente per la lunghezza vs vettore d'onda. Si intuisce dalla definizione che, nel nostro contesto di analisi di segnale, la FT dal dominio del tempo a quello della frequenza costituisce una *analisi* in frequenza di un dato segnale, mentre l'anti-FT è la sua *sintesi*, poichè ricostruisce il segnale a partire dalle sue componenti oscillanti. Nel dominio delle frequenze, il segnale può essere modificato, filtrato, etc., ad esempio privandolo delle componenti di frequenza sotto un dato valore o simili.

Qui considereremo gli aspetti teorici strettamente indispensabili a discutere la trasformata in versione discreta, senza pretesa di rigore o completezza. Tra i molti testi disponibili, il migliore, più graduale, e completo è probabilmente Ref.[1]. La FT è in un certo senso il limite

continuo della rappresentazione di una funzione in serie di Fourier di funzioni oscillanti,

$$g(t) = \sum_{f=-\infty}^{\infty} G_f e^{2\pi i f t} \tag{5.3}$$

nella quale G_f, cioè la FT, definisce il peso della specifica frequenza nel determinare la forma del segnale. Formalmente le espressioni si somigliano, ma va ricordato il diverso significato e contesto. La FT è lineare,

$$\mathcal{F}(f+g) = \mathcal{F}(f) + \mathcal{F}(g)$$

ed ha una quantità di utili simmetrie; ad esempio una funzione reale pari ha trasformata reale pari, e una funzione reale dipari ha trasformata immaginaria dispari. Un concetto particolarmente utile per capire la FT è la convoluzione, definita

$$g * f = \int_{-\infty}^{\infty} g(\tau) h(t-\tau) d\tau. \tag{5.4}$$

[2] La costruzione grafica in [2], Cap.4 (specialmente p.81), chiarisce bene il concetto per il caso che ci interessa di più, quello della convoluzione di un segnale con un array di funzioni impulso (delle δ di Dirac) corrispondenti al campionamento discreto (ovviamente ubiquo e inevitabile) del segnale: il risultato è un array di segnali *replicati* nella posizione delle funzioni impulso–niente paura, riprendiamo la discussione sotto. La FT della convoluzione è un prodotto

$$g * f \Longleftrightarrow G(f)H(f) \tag{5.5}$$

La convoluzione modificata, detta correlazione,

$$g * f = \int_{-\infty}^{\infty} g(\tau + t) h(t) d\tau \tag{5.6}$$

ha trasformata

$$\text{Corr}\,(g * f) \Longleftrightarrow G(f)H^*(f) \tag{5.7}$$

Ovviamente (teorema di Wiener-Khinchin)

$$\text{Corr}\,(g * g) \Longleftrightarrow |G(f)|^2, \tag{5.8}$$

anche detta densità di potere spettrale. Il potere spettrale integrato è lo stesso in tutti e due i domini (teorema di Parseval):

$$\int_{-\infty}^{\infty} dt\, |h(t)|^2 = \int_{-\infty}^{\infty} df\, |H(f)|^2. \tag{5.9}$$

Il potere spettrale permette di analizzare uno spettro in modo semplice senza confondersi con le parti reale e immaginaria della FT.

5.1 Campionamento discreto e finito

È un fatto della vita (pensateci un secondo e ne converrete) che qualunque funzione può essere rappresentata o misurata o osservata nella realtà solo in modo discreto e finito. La funzione è cioè definita

solo in un insieme discreto di valori e su un intervallo finito della variabile indipendente (proprio come nelle discretizzazioni fatte in precedenza). Questa ovvietà ha, come vedremo, delle conseguenze interessanti nel contesto della FT. Ferme restando le simmetrie della FT continua, la FT discreta deve sottostare a ulteriori vincoli dovuti a

a) campionamento discreto, cioè su N punti a intervalli di tempo τ tra un punto del campione e il successivo, e
b) finitezza del campione, che inizia a un tempo finito t e finisce a un tempo finito t+T. Vale naturalmente $N\tau$=T.

Discutiamo prima il punto a) con riferimento alla Fig.5.1, presa da Ref.[3], che illustra il processo graficamente. Il campionamento discreto è equivalente a moltiplicare il segnale per un array infinitamente esteso di funzioni $\delta(t–i\tau)$ con i intero relativo. La forma d'onda campionata contiene cioè solo valori discreti nel tempo. Notiamo ora che le coppie FT-antiFT sono definite in modo quasi del tutto simmetrico, per $t\rightarrow f$ come per $f\rightarrow t$. La convoluzione ha per FT il prodotto delle trasformate delle funzioni convolute, e la antiFT del prodotto è una convoluzione. Analogamente, la FT di un prodotto di due funzioni è la loro convoluzione.

Dunque la nostra funzione campionata, che è il prodotto della funzione per un array di δ, ha come trasformata la convoluzione della FT della funzione con la FT di un array di δ. Notiamo (potremmo dimostrarlo con un po' di lavoro analitico) che la FT dell'array è a sua volta un array $\delta(f–i/\tau)$ nello spazio delle frequenze; ricordiamo poi che convolvere una funzione con un array di impulsi equivale a replicare la funzione nelle posizioni degli impulsi. Il segnale originale è dunque trasformato, invece che nella sua vera FT, in un array periodico di repliche della FT in questione. Le repliche sono centrate nei punti i/τ, con i intero relativo. La distanza tra repliche adiacenti è $1/\tau$, e metà di questa distanza in frequenza, f_N=$1/(2\tau)$, è la frequenza di Nyquist. Supponiamo che la funzione sia *a banda limitata*, cioè abbia FT zero sopra una certa frequenza f_c. Un campionamento con τ abbastanza piccolo, tale che $f_N>f_c$, produrrà una FT con repliche ben separate tra loro, ognuna contenente tutta e sola l'informazione originale. Se però τ è troppo grande e tale che $f_N<f_c$, non solo la frequenza massima f_c non può essere campionata (ovviamente): le repliche si sovrappongono, e le frequenze superiori a f_N nel segnale si sovrappongono artificialmente con la replica adiacente. Questo fenomeno è detto *aliasing*.

Veniamo al punto b), l'effetto della finitezza del campione, con riferimento alla Fig.5.2. Il segnale originale viene osservato, ad intervallini τ, per un tempo finito T=$N\tau$, e può essere quindi visto come il prodotto del vero segnale e di un'onda quadra di larghezza T e ampiezza unitaria. Il segnale trasformando è quindi il prodotto del segnale campionato (quello vero moltiplicato per l'array di impulsi a intervallini τ) e dell'onda quadra che delimita il campione. La FT è dunque la convoluzione della FT campionata con la FT di un onda

[3]

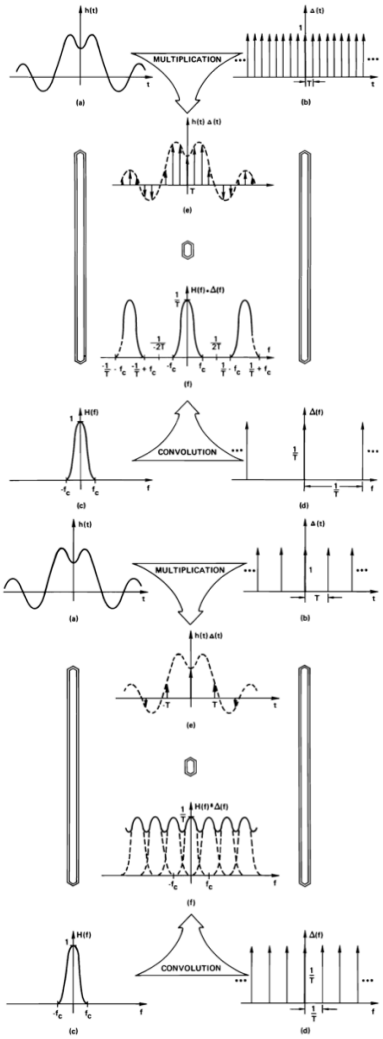


Figura 5.1: Schema grafico delle conseguenze del campionamento discreto sulla FT di un segnale; a sinistra, buon campionamento; a destra campionamento insufficiente. (I simboletti a forma di losanga indicano la FT.)

quadra di ampiezza unitaria, ovvero

$$\frac{\sin(\pi T f)}{\pi f}. \tag{5.10}$$

Ne risulta che le repliche della FT sono modulate dalla FT dell'onda quadra, che aumenta o diminisce il peso spettrale localmente con i propri lobi lentamente decrescenti. Ovviamente più piccolo è T, più lento il decay, e quindi maggiore la regione di spettro adulterata dal campionamento.

Questo effetto si chiama *leakage*, cioè 'spandimento' del potere spettrale, dato che il peso spettrale vero viene travasato da certe zone di frequenza ad altre. Si può attenuare l'effetto del leakage principalmente estendendo la lunghezza del campione; secondariamente, qualche giovamento viene dal cosiddetto *windowing*, di cui vedremo un esempio sotto: si moltiplica cioè il segnale per una funzione che faccia le veci dell'onda quadra, ma che aumenti (e diminuisca) meno bruscamente dell'onda quadra all'inizio (e alla fine) dell'intervallo totale di campionamento. Tutto quel che si è detto vale anche per la FT inversa, che in effetti genera una funzione periodica anche se il segnale originale era aperiodico.

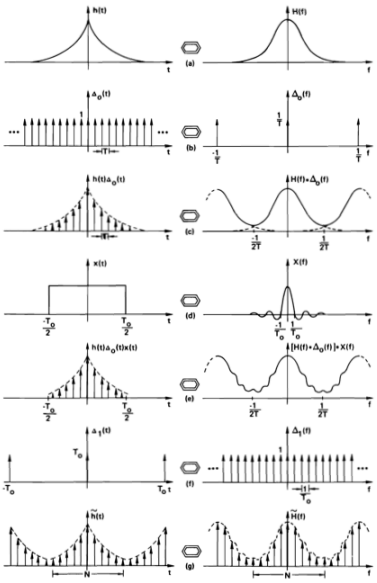


Figura 5.2: Costruzione grafica della trasformata discreta. Dalla quarta alla settima figuretta, l'effetto del campione finito.
[4]

La FT della funzione h nota su una griglia di N valori di t spaziati di τ si rappresenta in forma discreta approssimando l'integrale della definizione (o alternativamente con una serie di considerazioni riportate in Appendice o in Ref.[4]):

$$H(f_n) = \tau \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \equiv \tau H_n, \tag{5.11}$$

con H_n la FT discreta. L'antiFT discreta è

$$h(t_k) = \phi \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N} \equiv \phi h_k = \frac{1}{N\tau} h_k, \tag{5.12}$$

dove ϕ=$1/N\tau$ è l'intervallino di campionamento nel dominio delle frequenze (gli indici n e k di fatto sono intercambiabili, purchè in modo consistente). Il teorema di Parseval si scrive

$$\sum_{i=0}^{N-1} |h_i|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |H_k|^2$$

In sostanza, la antiFT è identica alla FT salvo la fase dell'esponenziale e una divisione per N; con questa procedura, come vedremo, si analizza il segnale e lo si ricostruisce pressochè perfettamente. Questa procedura ha un costo dell'ordine di N^2 operazioni. Ne esiste una variante detta Fast FT, che abbassa considerevolmente il costo portandolo a $o(N\log N)$. Data la complicazione della FFT, notiamo però che se la dimensione del set di dati non è troppo grande può essere conveniente usare la formula qui sopra.

La FT prodotta da Eq.5.11 è salvata in maniera un po' particolare. È importante precisare il dettaglio della discretizzazione. Il tempo ha N valori

$$t_i = i\tau \quad i = 0, N-1.$$

La k-esima componente della FT discreta corrisponde alla frequenza

$$f_k = \frac{k}{N\tau} \qquad k = 0, \frac{N}{2} \tag{5.13}$$

$$f_k = -\frac{k}{N\tau} \qquad k = \frac{N}{2}+1, N-1 \tag{5.14}$$

con la precisazione che la locazione $N/2$ contiene in realtà una combinazione delle componenti alla frequenza critica f_N=$\pm$1/(2τ). Gli N valori della FT corrispondono quindi alle frequenze positive da 0 alla frequenza f_N–ϕ (ϕ=1/[Nτ]) sono salvati in locazioni da 0 a ($N/2$)–1; quelli corrispondenti alle frequenze negative da –ϕ a –f_N+ϕ nelle locazioni (in discesa) da N–1 a ($N/2$)+1. La cosa sembra più complicata di quel che è, e la si capisce facilmente nello schema di Fig.5.3.

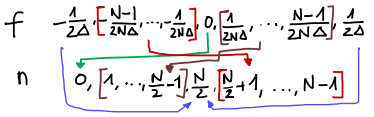


Figura 5.3: Schema dello storage e rappresentazione del segnale e della sua FT per valori complessi

Disegnando la trasformata come esce dalla routine che la calcola, la si vede quindi con la frequenza zero a sinistra estrema del dominio di frequenza, che copre l'intervallo di periodicità $1/\tau$ della FT del segnale campionato. In Fig.5.4, a sinistra, è mostrata questa rappresentazione convenzionale per la funzione

$$y(t) = A\left[\sin(2\pi f_0 t) + 2\sin(2\pi[2f_0]t) + \frac{1}{2}\sin(2\pi[4f_0]t)\right] + 1 \tag{5.15}$$

con A=1 e f_0=2 Hz, che risulta avere periodo P=0.5. (Importante: questo è il potere spettrale e non la trasformata !) Essendo f_N=8, il massimo time step ammissibile è τ_N=1/16=0.0625. Si è usato uno step 0.5 τ_N=1/32 e N=32, e quindi stiamo campionando un numero intero di periodi (due). Questo elimina completamente il leakage per costruzione. A frequenza zero c'e' una componente, dato che la funzione ha media non nulla. Ce n'è poi una a frequenza 2, 4, e 8, come previsto. Le frequenze negative –2, –4, e –8 sono alle locazioni 30, 28, e 24. Alla locazione 32, se ci fosse (ma non c'è, perchè N va da 0 a 31), ci sarebbe di nuovo la componente a frequenza nulla. Poichè la FT discreta è periodica in f per costruzione, traslando tutto il grafico a destra di $N/2$ si ottiene la visione simmetrica naturale. Comunque, con un po' di pratica, anche questa rappresentazione diventa facile da usare.

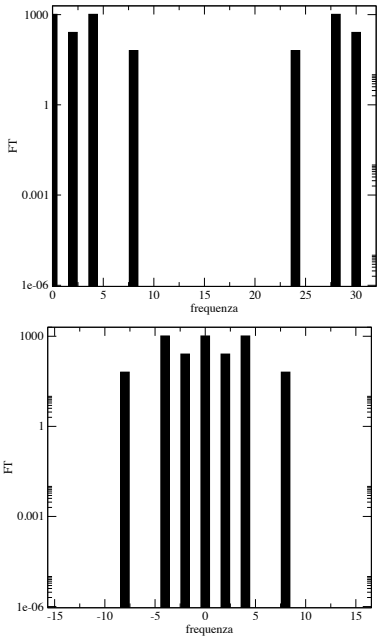


Figura 5.4: Rappresentazione della FT discreta periodica. A sinistra quella convenzionale, a destra quella "naturale".

Fig.5.4 mostra che la FT rileva correttamente le frequenze presenti nel segnale che abbiamo inventato. In Fig.5.5, mostriamo che la antiFT ri-sintetizza correttamente il segnale originale. Sono mostrate la funzione "continua" (linea), la funzione campionata (cerchi) e la funzione ri-sintetizzata dalla FT discreta, ovvero la anti-FT della FT della funzione campionata. Notiamo che sono pressochè indistinguibili, e che la componente immaginaria (non mostrata) del segnale ottenuto dalla anti-FT è minuscola (6-7 ordini di grandezza minore della parte reale).

In questo esempio abbiamo campionato esattamente un numero intero di periodi. Si può mostrare analiticamente che in questo caso non si ha leakage. Infatti i picchi della FT risultano particolarmente netti, e non c'è segnale ad altre frequenze su questa scala. Se il cam-

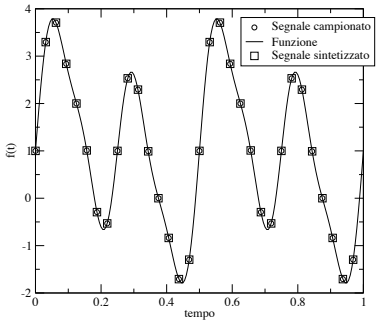


Figura 5.5: Funzione Eq.5.15 campionata e risintetizzata

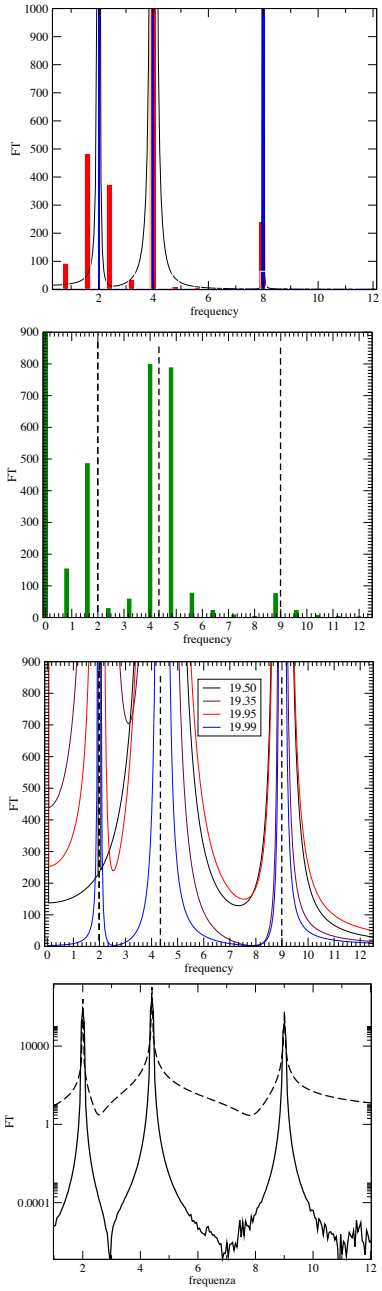


Figura 5.6: Leakage e dintorni. In alto: frequenze caratteristiche 0 (non mostrata), 2, 4, 8; durata del segnale 2.5 P (barre rosse), 39.2P (curva nera), 40P (barre blu). Secondo dall'alto: durata 1.25, ma con frequenze 0, 2, 4.4, 9. L'inviluppo $\sin f/f$ sposta il peso spettrale dalle vere frequenze (linee tratteggiate verticali) ad altre spurie. Terzo dall'alto: effetto del campionamento allungato: le frequenze sono tutte identificate, ma il leakage è importante, e viene soppresso (e non del tutto) solo per lunghezza del campione molto vicina a un multiplo del periodo. In basso: effetto del windowing. Il leakage è drammaticamente ridotto per la funzione cui è stato applicato il windowing, come mostrato in Fig.5.7.

[5]

pione non è multiplo del periodo della funzione campionata, tuttavia, il leakage è sempre significativo, e spesso inficia qualitativamente le conclusioni ottenibili dalla FT. In Fig.5.6, le barre rosse mostrano la FT per lo stesso segnale di Eq.5.15, ma campionato per un totale di 1.25 s=2.5 P. Le quattro frequenze (0, 2, 4, 8) sono difficili da distinguere; 0 (non mostrata per chiarezza), 4 e 8 ci sono, ma 2 no (e ci sono 1.4 e 2.7 spurie).

Il motivo è abbastanza chiaro: la risoluzione in frequenza è troppo bassa, con un intervallino elementare ϕ=1/T=0.795: quelle che si vedono sono tutte e sole le frequenze campionate, così non sorprende che non si becchi la frequenza 2. La 4 e la 8, invece, sono riprodotte bene perchè sono multipli quasi interi di ϕ. Se aumentiamo la durata del segnale a 19.6 s=39.2 P, con la stessa risoluzione temporale, otteniamo la curva nera, che mostra le frequenze corrette, sia pure con una larghezza apprezzabile, dovuta appunto al leakage. Con lo stesso segnale allungato a 20 s=40 P, si hanno le barre verticali blu, cioè le frequenze sono perfettamente riprodotte, senza nessun leakage.

È interessante modificare leggermente le frequenze a 4 e a 8 in frequenze a 4.4 e 9, in modo che τ non sia, almeno nelle intenzioni, sottomultiplo del periodo di alcuna componente. In questo caso, mostrato in Fig.5.6, secondo pannello dall'alto, troviamo varii picchi spurii che vanno attribuiti al leakage. Aumentando il periodo di campionamento (Fig.5.6, in basso a sinistra) le frequenze sono tutte ben identificate, ma il leakage è comunque importante, e viene soppresso in parte solo per lunghezza del campione molto vicina a un multiplo del periodo. Occasionalmente, per esempio per la frequenza f_0=2 per T=19.50, si trova un picco molto stretto e praticamente privo di leakage: di nuovo, questo corrisponde a un campionamento (accidentale, in questo caso) di un numero intero, o quasi, di periodi di quella componente dell'oscillazione. Nel caso specifico, questa frequenza è un multiplo intero della risoluzione in frequenza:

$$\frac{1}{T_0} \equiv f_0 = k\phi = \frac{k}{N\tau} \longrightarrow \mathrm{T}_0 = \frac{N}{k}\tau,$$

e capita che $k \simeq 40$ e N=3198, cioè $N/k \simeq 80$, cosicchè, appunto, stiamo campionando un numero intero di cicli di quella frequenza.

Infine, è utile analizzare gli effetti di riduzione del leakage ottenuti tramite windowing. Si moltiplica il segnale per una funzione che accompagni, meno bruscamente dell'onda quadra, l'inizio e la fine dell'intervallo totale di campionamento. Esistono varie versioni di window-function (l'articolo in proposito su Wikipedia è inusualmente bello [5]); usiamo per semplicità la funzione di Hanning

$$w(t) = \frac{1}{2}(1.0 - \cos r), \tag{5.16}$$

con

$$r = \frac{2\pi t}{T} = \frac{2\pi i}{N},$$

che ha proprietà più che discrete. La funzione originale e la sua versione "windowed", le cui brusche variazioni iniziali e finali sono

raccordate con continuità a zero, sono mostrate in Fig.5.7; la funzione di Hanning è nell'inset in alto a destra. L'effetto del windowing sulla FT è mostrato in Fig.5.6, ultimo pannello in basso. Si vede che il leakage è drammaticamente ridotto (notare la scala log-lin).

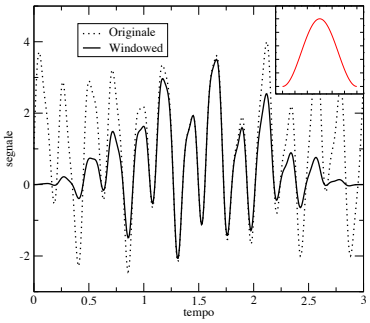


Figura 5.7: Segnale originale e segnale "windowed". Inset: funzione di window di Hanning.

5.2 *Applicazione: modi normali*

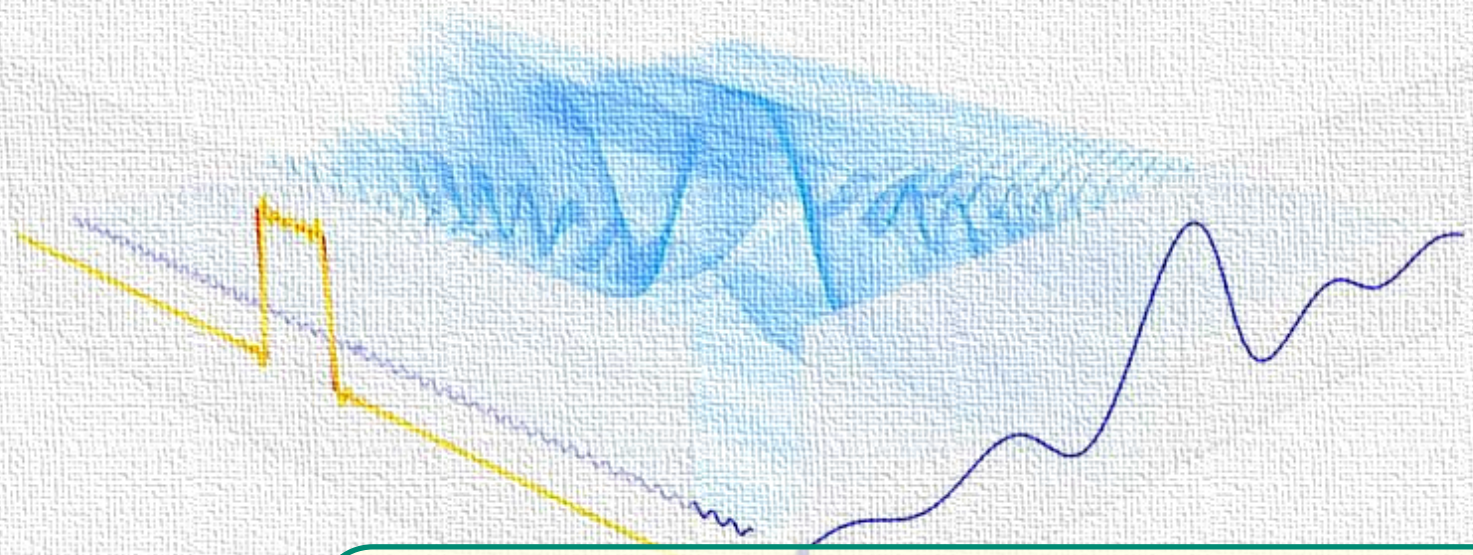
5.3 *Applicazione: analisi di segnali*

5.4 *Applicazione: filtro di componenti indesiderate di un segnale*

5.5 *Applicazione: deconvoluzione e image processing*

5.6 *Applicazione: PDE con l'analisi spettrale*

5.7 *Applicazione: "colori" del rumore*



5. B - Fourier transform

5.1 Analisi di segnali

In questo tutorial vedremo come implementare la routine per il calcolo della trasformata di Fourier discreta. La useremo poi per riprodurre l'analisi della funzione 161, fatta nelle dispense.

Partiamo dalla formula della trasformata discreta:

$$H(f_n) = \tau \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \equiv \tau H_n$$

Come vediamo è costituita da una sommatoria di termini esponenziali complessi dipendenti da due indici k, n moltiplicati per i valori della funzione di cui si vuole fare la trasformata h_k. L'indice k è anche quello su cui si esegue la sommatoria, per ogni valore dell'indice n fissato, rappresenterà i valori discreti della trasformata H_n. Il tempo di campionamento invece è rappresentato da τ. Vediamo di seguito le righe di codice che permettono di imprementare questa formula:

```
1  #definisco una funzione h di prova
2  tau=0.05
3  t=arange(-10,10,tau)
4  T=2.
5  h=sin(2*pi*t/T)
6
7  N=len(h)
8  H=zeros(N,dtype='complex')
9
10 for n in range(N):
11   for k in range(N):
12     H[n] += h[k]*exp(2j*pi*k*n/N)
13 H*=tau
```

Vediamo qualche dettaglio sulle precedenti linee di codice:

- nelle righe 2-5 definiamo una funzione di prova molto semplice con una frequenza nota. Solitamente la funzione h di cui si vuol fare la trasformata è data da un segnale proveniente da una qualche sorgente che quindi verrà inizializzata partendo da dati registrati su un file (vedremo meglio in seguito un caso di questo tipo).
- nelle due righe successive, definiamo il numero `N` di componenti della funzione e poi il vettore `H` che conterrà la trasformata. Poiché la trasformata in genere restituisce dei valori complessi, definiamo come tali le componenti di `H`, usando `dtype='complex'`.
- nelle righe successive troviamo i due cicli `for` che permettono il calcolo della traasformata: fissato l'indice n dal relativo ciclo, per ogni valore dell'indice k calcoliamo l'esponenziale complesso, lo moltiplichiamo per la componente k-sima della funzione h e aggiungiamo il risultato al valore calcolato per l'indice k precedente. Una volta eseguite tutte le iterazioni, moltiplichiamo tutte le componenti del vettore `H` per il tempo `tau`.
- soffermiamoci sul termine `2j` usato dentro l'esponenziale: è la forma rapida per rappresentare il numero immaginario $2i$ ed è del tutto uguale a scrivere 2*1j.

Poiché la routine che calcola la trasformata è indipendente dal tipo di funzione di partenza possiamo racchiuderla all'interno di una funzione python che potremo chiamare ogni volta che vogliamo dandogli in ingresso la funzione di cui vogliamo calcolare la trasformata. Vediamo dunque come definire una funzione in python:

```
1  def my_dft(h,tau):
2    N=len(h)
3    H=zeros(N,dtype='complex')
4    for n in range(N):
5      for k in range(N):
6        H[n] += h[k]*exp(2j*pi*k*n/N)
7    H *= tau
8    return H
9
10 #uso la funzione appena definita
11 Transf_f1 = my_dft(f1,tau1)
12 Transf_f2 = my_dft(f2,tau2)
```

Vediamo in dettaglio cosa abbiamo fatto:

- con `def nome_funzione(arg1,arg2,...):` si definisce una funzione in python, avente come argomenti in ingresso le variabili (di qualunque tipo) `arg1`, `arg2`,... etc.
- Le variabili in ingresso saranno usate all'interno della funzione, cioè in tutte le righe che dopo il `def` sono indentate sino alla riga `return`.
- Come si nota nelle ultime due righe, il vantaggio di definire una funzione come `my_dft` é quello che, una volta fatto, potremo richiamarla tutte le volte che vorremo dandogli in ingresso le funzioni che vogliamo trasformare senza dover cambiare ogni volta nella routine il nome delle delle variabili che contengono la

funzione stessa.

- La routine della trasformata viene quindi scritta una sola volta, all'interno della funzione `my_dft` e posta solitamente all'inizio del file (o anche in un file a parte contenente tutte le funzioni che definiamo).
- la riga `return H` permette di restituire la variabile `H` calcolata all'interno della funzione, quando questa finisce, e quindi di poterla salvare in un'altra variabile, come si fa nelle ultime due righe di codice.

In maniera del tutto analoga a quanto fatto per la trasformata, possiamo definire una funzione che calcoli l'anti-trasformata usando la formula 163 delle dispense.

```
1  def my_antidft(H,tau):
2    N=len(H)
3    h=zeros(N,dtype='complex')
4    for n in range(N):
5      for k in range(N):
6        h[n] += H[k]*exp(-2j*pi*k*n/N)
7    H *= 1/tau/N
8    return h
9
10 #uso la funzione appena definita
11 Transf_f1 = my_dft(f1,tau1)
12 f2 = my_antidft(Transf_f1,tau1)
13 plot(f1)
14 plot(f2)
```

Come vediamo la routine eà del tutto simile a quella scritta per la trasformata. Nelle ultime quattro righe facciamo la trasformata di una funzione `f1` e la salviamo nel vettore `Transf_f1`. Poi facciamo l'anti-trasformata di quest'ultima e salviamo il risultato nel vettore `f2`. Cosa vi aspettate dal plot delle funzioni `f1` e `f2`?

Ora che abbiamo gli strumenti necessari possiamo studiare la funzione 166 delle dispense nella configurazione di partenza, cioe con tempo di campionamento $\tau = 1/32$ e numero totale di punti $N = 32$, quindi periodo $P = 1$, e frequenze multipli di 2. Vediamo il codice:

```
1  #supponiamo che siano state definite
2  #sopra le funzioni my_dft e my_antidft
3
4  tau=1/32.
5  T=1.
6  A=1.
7  f0=2.
8  f1=2*f0
9  f2=4*f0
10
11 t=arange(0,T,tau)
12 N=len(t) #o anche N=T/tau
13
```

```
14 funct = 1 + A*(sin(2*pi*f0*t)+2*sin(2*pi*f1*t)+0.5*sin(2*pi*f2*
15 Transf = my_dft(funct,tau)
16
17 freq=arange(N)/tau/N
18 bar(freq,abs(Transf)**2,width=0.5,align='center',color='b')
19
20 figure(2)
21 bar(freq,Transf.real,width=0.5,align='center',color='b',label='
22 bar(freq,Transf.imag,width=0.5,align='center',color='r',label='
```

Come vediamo, una volta definite le variabili utili al calcolo della funzione, definiamo un vettore con i tempi, la funzione funct che vogliamo analizzare e ne facciamo la trasformata chiamando la nostra funzione definita all'inizio del file e salviamo il risultato dentro la variabile `Transf`.

Definiamo inoltre un vettore contenente i valori delle frequenze mediante la formula 164 delle dispense. Da notare che in questo modo, però, saranno corrette solo le prime $N/2$, che sono comunque quelle di interesse. Le successive frequenze sono delle repliche traslate di N (vedasi dispense e grafici seguenti).

I plot che riportiamo di seguito sono stati eseguiti mediante la funzione bar che disegna delle barre verticali in corrispondenza di ogni frequenza con altezza pari al valore della trasformata. I primi due argomenti di ingresso sono gli array delle frequenze e del potere spettrale, o della parte reale e immaginaria. Notiamo come per fare il modulo di un vettore complesso si possa ancora usare la funzione abs, già vista nei precedenti tutorials. Inoltre per estrarre dall'array complesso `Transf` le due componenti usiamo la forma `Transf.real/.imag` che di fatto genera due array.

Gli altri parametri di ingresso sono abbastanza espliciti: larghezza, allineamento centrato rispetto ai valori dell'asse x e colore delle barre.

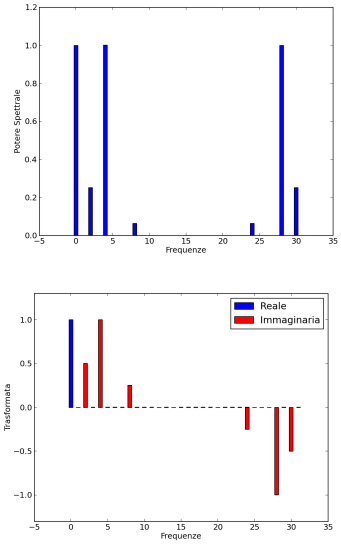


Figura 5.1:

Il grafico 5.1 a sinistra rappresenta il potere spettrale, mentre in quello a destra troviamo la parte reale e immaginaria della trasformata. Osservando i grafici notiamo come le tutte le frequenze che compongono il segnale vengano identificate con precisione e si trovano nella parte immaginaria della trasformata. Nella parte reale invece troviamo la sola componente a frequenza nulla che non è altro che la costante $+1$ che abbiamo nel segnale. Svolgendo analiticamente la trasformata si ha una conferma dei diversi contributi che si osservano.

Vediamo di passaggio un interessante esempio, sicuramente il più semplice, di come si possa modificare il segnale nello spazio delle frequenze. Supponiamo di voler togliere la componente a frequenza nulla dalla parte reale e tornando indietro allo spazio dei tempi, mediante l'anti-trasformata, vediamo come il segnale viene modificato. Di seguito il codice necessario:

```
1 #eliminiamo la componente a frequenza zero
2 Transf.real[0] = 0
3
4 antiTransf = my_antidft(Transf,tau)
```

```
5
6 plot(t,antiTransf,'-s',label='funct modificata')
7 plot(t,funct,'-d',label='funct originale')
```

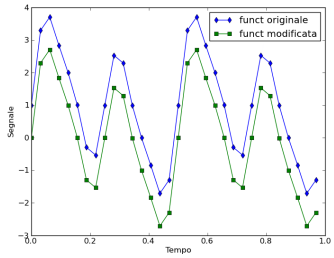


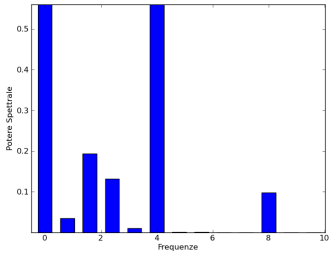
Figura 5.2:

In Figura 5.2 vediamo i grafici generati dalle due chiamate plot.

Come osserviamo la funzione modificata è stata soltanto traslata in basso di una quantità pari a 1. Questo è in linea con quanto abbiamo accennato in precedenza: la componente reale della trasformata contiene la parte costante della funzione, perciò eliminare quella componente nello spazio delle frequenze, significa eliminare la costante nello spazio dei tempi. Cosa ci aspettiamo dunque se dovessimo togliere dalla trasformata le qualcuna delle altre componenti?

Vediamo di seguito i seguenti casi che riassumono un pò quelli mostrati nelle dispense:

- cambiamo il periodo di campionamento da 1a 1.25;
- aumentiamo ancora il periodo portandolo a 19.6;
- modifichiamo ora le frequenze in $2, 4.4, 9$ e proviamo diversi periodi di campionamento.



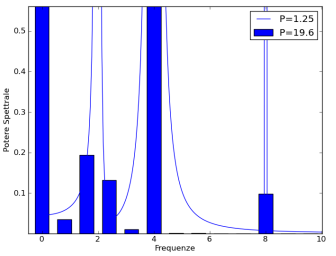


Figura 5.3:

In Figura 5.3 mostriamo i grafici relativi ai primi due punti, dove abbiamo fatto andare il programma per due volte, modificando il solo periodo, usando la funzione `bar(freq,abs(Transf)**2)` per il primo grafico e la funzione `plot(freq,abs(Transf)**2)` per aggiungere le linee nel secondo. Quello che osserviamo nel primo grafico è che la frequenza 2 non è più identificata e il potere spettrale si è spostato su frequenze spurie ad essa affiancate. Questo spostamento del peso spettrale si nota anche per la frequenza 4, anche se questa resta identificata insieme alla 8. Questo è dovuto al semplice fatto che il periodo 1.25 non è più un multiplo della frequenza 2, mentre lo è ancora per le frequenze 4 e 8. Si provi a modificare in maniera ancora diversa il periodo e si osservi come cambia lo spettro delle frequenze.

Nel secondo grafico abbiamo fatto il grafico del potere spettrale per il caso di un periodo pari a 19.6. In questo caso osserviamo che aumentando il periodo di quasi venti volte si ottengono dei picchi in corrispondenza delle frequenze corrette, ma come effetto negativo il peso spettrale si sposta anche su frequenze adiacenti a quelle reali, in maniera differente come si nota dalla diversa larghezza dei picchi. Questo è il fenomeno del leakage. Si provi a modificare il periodo di campionamento con valori vicini a 20 (multiplo del periodo) e si osservi come varia lo spettro delle frequenze.

Passiamo ora al terzo caso, in cui abbiamo intenzionalmente posto delle frequenze che difficilmente saranno contemporaneamente suttomultipli di un periodo di campionamento.

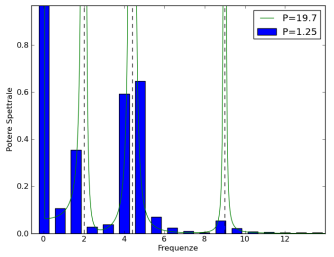


Figura 5.4:

Come osserviamo dalla Figura 5.4 se il periodo di campionamento è corto (1.25 per esempio) le frequenze che compongono il segnale non vengono identificate correttamente, mentre al loro posto ne compaiono delle altre. Aumentando il periodo di campionamento (19.7 per esempio) riusciamo a identificare le corrette frequenze anche se un pò di leakage è presente. Questo mostra dunque l'importanza di avere un tempo di campionamento il più lungo possibile per

assicurarsi di poter trovare le corrette frequenze di un certo segnale.

Una tecnica usata per ridurre il leakage è quella del windowing: moltiplicare il segnale per una funzione finestra che vada a zero lentamente. La funzione finestra che usiamo di seguito, tra le tante che si possono usare, è la 167 delle dispense. Di seguito riportiamo il codice per implementare il windowing:

```
#definiamo la funzione finestra
r=2*pi*t/T
han=0.5-0.5*cos(r)

funct_windowed=funct*han
Transf_window=my_dft(funct_windowed,tau)

#facciamo i grafici
plot(t,han)

figure(2)
plot(t,funct)
plot(t,funct_windowed)

figure(3)
plot(freq,abs(Transf)**2,label='Senza window')
plot(freq,abs(Transf_window)**2,label='Con window')
```

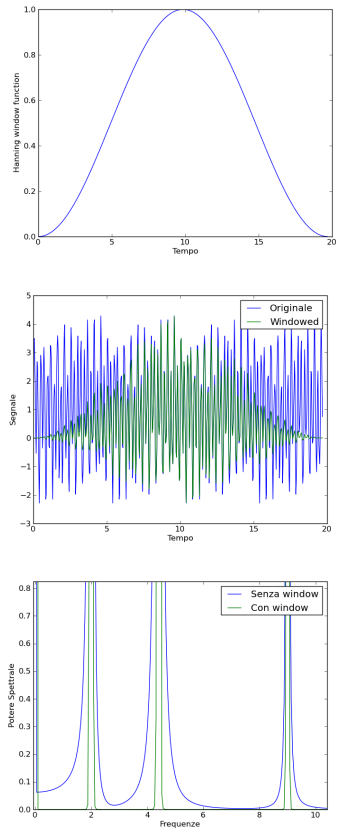


Figura 5.5:

I grafici che otteniamo sono mostrati in Figura 5.5.

Abbiamo applicato la finestra all'ultimo segnale usato, quello con periodo di campionamento di 19.7 e frequenze $2, 4.4, 9$. Come osserviamo nella terza figura, i picchi che identificano le frequenze sono notevolmente più stretti nel caso di windowing rispetto al caso senza.

5.2 *Macchie solari*

In questo tutorial faremo l'analisi delle frequenze di segnali provenienti da due sorgenti differenti: il primo rappresenta in numero di macchie solari monitorate mensilmente dal 1749 al 2011; il secondo è una nota musicale proveniente da due strumenti differenti, un piano e una tromba.

Partiamo con il primo segnale. Di seguito riporto le righe di codice utili a importare dei dati presenti su un file e salvarli in arrays python:

```
data=loadtxt('sunspots.txt')
time=data[:,0]
nspots=data[:,1]

plot(time,nspots)
```

La funzione `loadtxt('nomefile')` legge i dati contenuti all'interno del file indicato come primo argomento e li salva un array mantenendo la stessa struttura che hanno nel file. In questo caso, il file contiene due colonne e 3143 colonne perciò l'array data avrà

dimensioni 3143x2. Nelle righe successive estraiamo dall'array data la prima e la seconda colonna, il tempo e in numero di macchie solari, rispettivamente, e le salviamo in due array diversi, per comodità. Il grafico è mostrato in Figura 5.6

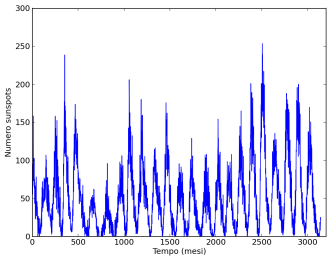


Figura 5.6:

Dal grafico si può notare una certa periodicità nell'andamento degli sunspots. Prendendo due picchi a caso, la loro distanza è di circa 130 mesi, quindi circa 11 anni. Proviamo a calcolare lo spettro delle frequenze facendo la trasformata del segnale con le righe di codice seguenti:

```
1 tau=time[1]-time[0]
2 T=time[-1]
3 N=len(time)
4 freq=arange(N)/T
5 Transf = my_dft(nspots,tau)
6
7 plot(freq,abs(Transf)**2,'-d')
```

Dopo aver definito alcune costanti a partire dal segnale campionato, creiamo l'array delle frequenze e poi facciamo la trasformata, supponendo di aver definito la funzione `my_dft` precedentemente. Nella Figura 5.7 è riportato il grafico del potere spettrale.

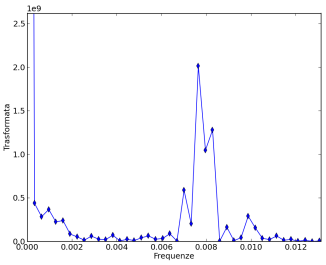


Figura 5.7:

Come si nota effettuando uno zoom troviamo un picco dominante a frequenze di cira 0.008. Per ridurre il leakage si è provato a usare la finestra di Hanning, in questo modo:

```
1 r=2*pi*time/T
2 han=0.5-0.5*cos(r)
3 Transf_w=my_dft(nspots*han)
4
5 plot(freq,abs(Hw)**2,'-d')
```

ottenendo una riduzione del leakage apprezzabile, come si nota dal grafico in Figura 5.8

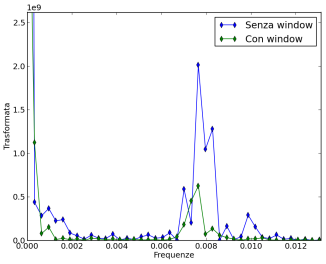


Figura 5.8:

Il picco ora è più facilemente identificabile e risulta in corrispondenza della frequenza 0.007638. Il periodo corrispondente a tale frequenza è dunque $1/0.007638 = 130.92\,mesi = 10.91\,anni$.

Abbiamo dunque trovato in maniera più precisa ed elaborata il periodo di variazione delle macchie solari, noto per essere di cira 11 anni apppunto, utilizzando la trasformata di fourier di un segnale proveniente da una sorgente naturale, in questo caso il sole.

5.3 *Strumenti musicali*

Passiamo ora al caso di segnale proveniente da due strumenti musicali. Cominciamo importando i dati e visualizzandoli:

```
1 samples_piano=loadtxt('piano.txt')
2 samples_trumpet=loadtxt('trumpet.txt')
3
4 N=len(samples_piano)
```

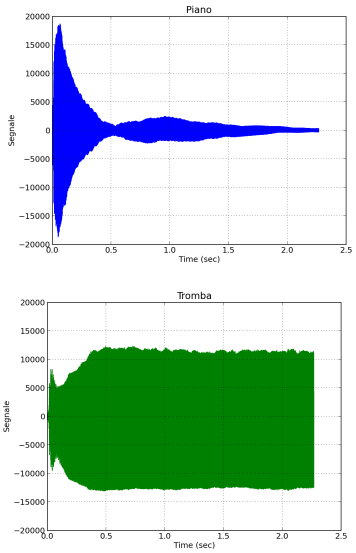


Figura 5.9:

```
5  tau=1./44100
6  T=N*tau
7  time=linspace(0,T,N)
8
9  plot(time,samples_piano)
10 plot(time,samples_trumpet)
```

In questo caso i files contenenti il suono proveniente dai due strumenti hanno una sola colonna, ma sapendo che la frequenza di campionamento è quella standard di 44100 Hz, possiamo ricavare l'array tempo e fare il grafico di entrambi i suoni registrati, mostrati in Figura 5.9

Le forme sono notevolemente differenti anche nello spazio dei tempi: il suono del piano tende a decrescere rapidamente, mentre quello della tromba sale rapidamente ad una certa soglia che poi mantiene per tutto il periodo di campionamento.

Vediamo come appaiono questi due segnali nello spazio delle frequenze, usando le seguenti righe di codice:

```
1  Transf_piano=my_dft(samples_piano,tau)
2  Transf_trumpet=my_dft(samples_trumpet,tau)
3  freq=arange(N)/T
4
5  plot(freq,abs(Transf_piano)**2)
6  xlim(0,10000/T) # imposto il range da visualizzare lungo x
7  ylim(0,3E+16)   # e lungo y
```

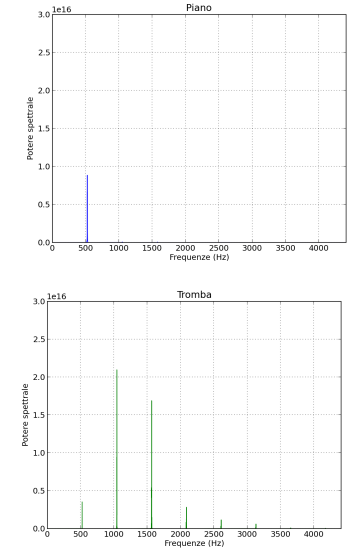


Figura 5.10:

Come osserviamo dalle figure 5.10, la frequenza dominante per per il segnale prodoto dal piano si trova a circa 522 Hz e rappresenta un $Do5$, cioé il Do successivo al $La4$ (440 Hz) presa convenzionalmente come nota di riferimento. Ingrandendo, compaiono frequenze corrispondenti a note superiori in frequenza, ma con peso ordini di grandezza via via inferiore. Nello spettro delle frequenze del segnale della tromba, notiamo che la prima frequenza è la stessa trovata nel piano, ma ci sono altre due frequenze successive che pesano di più e corrispondono al $Do7$ ($\sim$ 1044 Hz) e al $Sol7$ ($\sim$ 1566 Hz) , e un altra che ha lo stesso peso e che corrisponde al $Do8$ ($\sim$ 2088 Hz).

Potremo concludere che il suono del piano, contenendo una singola componente predominante, è più "puro" rispetto a quello della tromba, che al contrario mescola la stessa nota del piano, il $Do5$, con Do di due ottave superiori e con una nota differente, il $Sol7$, non presente nel piano.

6. A - Metodo Montecarlo

Il metodo Montecarlo (MC) calcola quantità fisiche in sistemi con grande numero di gradi di libertà tramite medie su un campione stocastico scorrelato di coordinate o configurazioni. In sostanza si tratta, alla base, di una sequenza di numeri o eventi casuali (o quantità che da essi dipendono): stòqos in greco significa anche "caso", e il metodo prende nome dal casinò di Montecarlo nel principato di Monaco.

La caratteristica chiave del MC è che il suo errore algoritmico è proporzionale a $1/\sqrt{N}$, con N la dimensione del campione, *indipendentemente dalla dimensionalità del problema*, e quindi il suo costo è proporzionale a $\sqrt{N}$. Questo fa la differenza tra ciò che si può o non si può fare: un integrale MC calcolato come media su un campione casuale di N ascisse in D dimensioni costa $o(\sqrt{N})$, mentre, come discusso in Sez.**??**, un integrale su una griglia equispaziata (tipo regola trapezoidale) ha un proibitivo costo esponenziale $o(N^D)$. Il comportamento $\sqrt{N}$ è caratteristico di molti fenomeni intrinsecamente stocastici, come ad esempio il cammino quadratico medio nel moto browniano o in generale nel moto aleatorio,

$$\sqrt{\langle r^2 \rangle} \simeq \sqrt{N},$$

con N il numero di passi. Torneremo su questo punto più oltre. Tipici problemi impossibili che diventano trattabili con il MC sono quelli della meccanica statistica, dove le quantità fisiche medie si calcolano con integrali in altissima dimensionalità del tipo già visto in precedenza,

$$\langle A \rangle = \int d\mathbf{q}_1 \dots d\mathbf{q}_k \, A(\mathbf{q}_1, \dots \mathbf{q}_k) \exp[-\beta E(\mathbf{q}_1, \dots \mathbf{q}_k)], \qquad (6.1)$$

con k dell'ordine delle centinaia o migliaia. Per generare, e quindi campionare, stocasticamente le configurazioni di un sistema (o di una funzione, come vedremo) si utilizzano numeri pseudo-casuali (che per brevità chiameremo casuali, con linguaggio un po', ehm, casual).

Il motore del MC è perciò il generatore di numeri casuali. Scrivere un generatore affidabile è apparentemente semplice, ma assolutamente non banale, come discusso ad esempio in [1], Cap.7, e [2], Cap.5.

6.1 *Numeri pseudocasuali e loro test. Decadimento*

I numeri casuali sono caratterizzati dalla distribuzione $P(x)$; P è definita positiva e normalizzata su un certo dominio Ω, e fornisce la probabilità che la sequenza generi un dato numero casuale:

$$P(x) > 0; \qquad \int_\Omega P(x)\,dx = 1; \qquad P(r)\,dr = \text{Prob}\{r \in (r, r+dr)\}.$$

La distribuzione più importante è quella uniforme (da cui, con tecniche che discuteremo, si estraggono le altre distribuzioni di interesse), che è uguale a 1 sull'intervallo [0,1]. Se si genera una grande quantità di numeri casuali uniformi e si costruisce un istogramma che conti quanti numeri sono usciti in ogni dato intervallino dx si ottiene (o si dovrebbe ottenere) un'onda quadra in [0,1) come in Fig.6.1 a sinistra; lo stesso processo con una distribuzione gaussiana dovrebbe dare il risultato a destra.

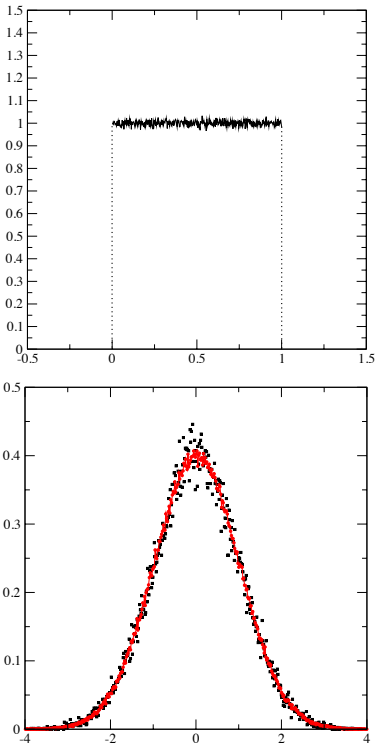


Figura 6.1: Distribuzione uniforme e gaussiana.

I numeri casuali sono generati (in pratica) sequenzialmente, e si cerca di far sì che la sequenza sia scorrelata da un passo al successivo (catena markoviana di ordine 1). Il più semplice generatore di numeri casuali è quello lineare congruente

$$r_i \equiv (a r_{i-1} + c) \bmod M = \text{resto di } \left(\frac{a r_{i-1} + c}{M}\right) \tag{6.2}$$

che genera, a partire da un numero r_1 detto seed, una sequenza di numeri tra 0 e M–1, che ovviamente può essere riportata in [0,1) dividendo per M. La speranza è che per opportuni a e c, e soprattutto per grande e opportuno M, questo processo evidenzi la parte "meno significativa" del numeratore, in sostanza amplificando la "casualità di arrotondamento" che causa (di norma) l'errore di arrotondamento proporzionale a $\sqrt{N}$ di cui in Sez.1.2.

Non vogliamo cimentarci con la scrittura di un buon generatore di numeri casuali, impresa al confine tra la scienza, l'arte, e la magia. Qui dimostriamo solo che scelte volutamente cattive dei parametri del generatore producono sequenze altamente correlate. In Fig.6.2 mostriamo tre array di punti le cui coordinate x e y sono (nelle intenzioni) casuali. Se scegliamo r_1=13, a=57, c=1, e M=25 (Fig.6.2, a sinistra) si ottiene un array di punti visibilmente correlati. Passando a c=13, M=4097 (Fig.6.2, al centro) la sequenza sembra meno correlata; in effetti, però, i punti sono a occhio un centinaio, mentre dovrebbero essere due migliaia circa: gli stessi punti sono visitati varie volte, indice di forte correlazione. Usando c=17 e M=4111 (che è primo) si ottiene una sequenza molto meno correlata delle altre due, almeno visualmente.

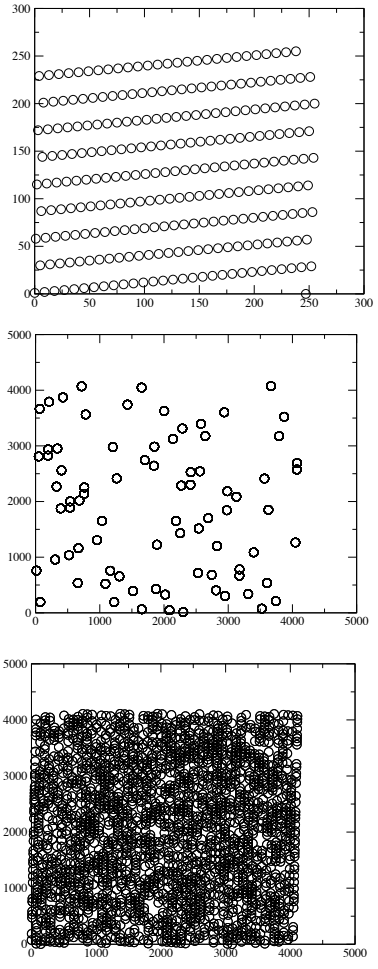


Figura 6.2: Sequenze random correlate.

L'apparenza non è però un buon criterio per valutare la casualità e la correlazione di una sequenza. Per decidere se una sequenza sia random e scorrelata, esistono alcuni possibili semplici test, almeno per la

distribuzione uniforme, che iniziano ad introdurci alla manipolazione del classico integrale MC.

Il core del MC è appunto, in una forma o nell'altra, il calcolo di medie, e quindi piuttosto naturalmente l'integrazione. Per una funzione in 1D, il teorema della media integrale dice che

$$I = \int_a^b dx\, f(x) = (b-a)\langle f\rangle, \tag{6.3}$$

dove la media al terzo membro va calcolata appunto nel punto della media integrale. Nello stesso spirito, assumendo di non sapere (come di solito è il caso) il punto di media, si approssima l'integrale come media su un campione stocastico come

$$I = \int_a^b dx\, f(x) = (b-a)\langle f\rangle \simeq (b-a)\frac{1}{N}\sum_i^N f(x_i) \tag{6.4}$$

dove gli x_i sono numeri casuali nel dominio $\Omega \equiv [a,b]$, e N va scelto come sempre in modo da portare l'errore relativo sotto una soglia desiderata. L'integrale, essendo una media, ha associata una varianza totale

$$\sigma_I^2 = \frac{1}{N}\sigma_f^2 = \frac{1}{N}\left[\frac{1}{N}\sum_i^N f^2(x_i) - \left(\frac{1}{N}\sum_i^N f(x_i)\right)^2\right], \tag{6.5}$$

la media delle fluttuazioni della funzione f, espresse in modo classico come differenza tra "media del quadrato" e "quadrato della media"; l'errore caratteristico σ_I è quindi proporzionale a $1/\sqrt{N}$, ma è tanto più grande quanto lo è la varianza della funzione sul dominio. Tra le strategie per ridurre l'errore, la riduzione della varianza è molto importante, e ne riparleremo in Sez.6.3. Notiamo di passaggio che un integrale in D dimensioni non presenta maggiori difficoltà. Infatti

$$\int_\Omega f(\vec{x}) d^D\vec{x} \simeq \frac{1}{N}\sum_i^N f(\vec{x}_i) \tag{6.6}$$

dove gli $\vec{x}_i$ sono vettori con D componenti casuali che fungono da coordinate di un punto in D dimensioni. N sarà dipendente dalla funzione, ma non dipende dalla dimensionalità.

Il primo integrale che calcoliamo è legato appunto ai test di casualità di una sequenza. È il momento k-esimo della distribuzione. Se la distribuzione è uniforme,

$$\int_0^1 dx\, x^k\, P(x) = \int_0^1 dx\, x^k = \frac{x^{k+1}}{k+1}|_0^1 = \frac{1}{k+1} \simeq \frac{1}{N}\sum_{i=1}^N x_i^k, \tag{6.7}$$

e se la distribuzione è random, l'ultima eguaglianza vale a meno di un errore $o(\sqrt{1/N})$.

Un altro test interessante concerne la correlazione. La funzione di correlazione è

$$C(k) = \int_0^1 dx \int_0^1 dy\, P(x,y) = \int_0^1 dx \int_0^1 xy = \int_0^1 dx\, \frac{y^2 x}{2}|_0^1 = \frac{1}{4} \tag{6.8}$$

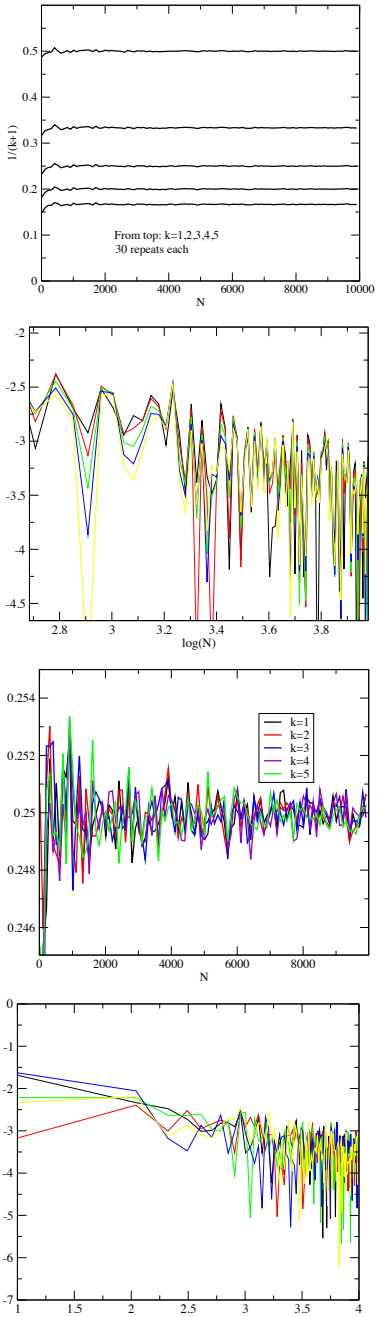


Figura 6.3: Test di correlazione (in alto) e dei momenti (in basso) per la distribuzione uniforme, effettuati in 30 prove ripetute per ogni valore di N con seed iniziale diverso per ogni sequenza.

dove la seconda uguaglianza vale se la distribuzione è uniforme, e assumendo nessuna correlazione tra x e y (cioè che la probabilità congiunta sia il prodotto delle singole probabilità). I numeri di una sequenza, perciò, non sono correlati a corto raggio (a distanza di k numeri) se l'integrale MC fa $1/4$. Di nuovo, un errore $o(\sqrt{1/N})$ indica una sequenza random.

Applichiamo i due test al generatore `ran1` del Cap.7 di Numerical Recipes [3], che è un buon generatore per uso generico e dimostrativo. I risultati sono in Fig.6.3. Si nota in alto a sinistra che i valori dei momenti convergono correttamente a $1/(k+1)$, e in basso a sinistra che la correlazione è quella attesa per una distribuzione uniforme e scorrelata. Gli errori sono lentamente decrescenti, e in ambedue i casi se ne nota, in scala doppia-log, un calo di mezza unità per ogni unità di aumento di N, cioè pendenza circa $1/2$, ovvero la legge $1/\sqrt{N}$. Ognuno dei calcoli per un dato N è effettuato su 30 ripetizioni. Questo suggerisce domande del tipo "Quanto dev'essere grande N ? E quante volte va ripetuto il campione con sequenze random indipendenti ? Meglio tanti N una volta sola, o meno N e tanti campioni ? In che proporzione ?" Purtroppo non ci sono risposte univoche – dipende dal problema. Naturalmente N dev'essere grande, ma le ripetizioni con campioni (sequenze) diversi sono altamente desiderabili. Gli esempi successivi forniranno qualche indicazione.

[3]

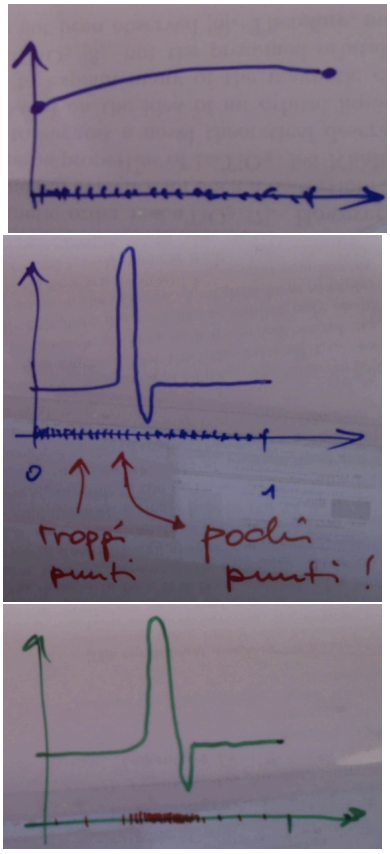


Figura 6.4: Schema di sampling di una funzione lentamente variabile (sinistra) e di una rapidamente variabile (centro) con numeri uniformi. A destra, il principio dell'importance sampling.

6.2 *Random walk*

6.3 *Importance sampling nell'integrazione MC*

Come menzionato in relazione ad Eq.6.5, la riduzione della varianza è una importante strategia per convergere più velocemente le medie MC. La tecnica principe a questo fine è l'importance sampling. Come schematizzato in Fig.6.4, per una funzione lentamente variabile il campionamento uniforme delle coordinate può essere adeguato, dato che la funzione fluttua poco. Ma qualunque forte variazione risulterà mal campionata. L'ideale sarebbe avere tanti punti campione dove la funzione varia molto, e pochi dove varia poco, cioè appunto un campionamento pesato per importanza (peso nell'integrale) in ogni regione. Ne seguirebbe una varianza ridotta e quindi una convergenza per N minore, sia pure con la stessa legge $\sqrt{N}$. Ovviamente è necessario generare numeri casuali distribuiti non uniformemente: idealmente, generarli distribuiti come la funzione integranda, o comunque in modo simile. Spesso questo non è possibile perchè non si sa generare numeri casuali con quella distribuzione o perchè la funzione non è nota a priori (in questo caso si utilizza ad esempio il metodo di Metropolis, Sez.6.4).

Per il momento formuliamo il principio, e scegliamo un integrale di una funzione semplice per dimostrarlo, specificamente

$$\int_0^1 \frac{dx}{1+x^2} = \frac{\pi}{4} = 0.78540. \tag{6.9}$$

Per ridurre la varianza decidiamo di moltiplicare e dividere l'integrando per una funzione peso positiva e normalizzata $w(x)$. L'integrale diventa allora

$$I = \int_0^1 dx\, f(x) = \int_0^1 dx\, w(x) \frac{f(x)}{w(x)} \tag{6.10}$$

Ora facciamo un cambio di variabile da x a una y tale che dy=$w(x)dx$, e cioè

$$\frac{dy}{dx} = w(x), \quad \text{o anche :} \quad y(x) = \int_0^x dt\, w(t). \tag{6.11}$$

Imponendo y(0)=0 e y(1)=1 l'integrale diventa

$$I = \int_0^1 dx\, f(x) = \int_0^1 dy \frac{f(x(y))}{w(x(y))} \tag{6.12}$$

Se scegliamo una w che somiglia a f, l'integrando sarà lentamente variabile e quindi la sua varianza sarà ridotta, cosicchè l'integrale convergerà più rapidamente. Notiamo che se y è uniforme, x è distribuito come w, e quindi i punti di integrazione x sono concentrati nelle regioni dove w è maggiore (e sperabilmente così è f). Naturalmente è necessario trovare una w adeguata e saper invertire la relazione Eq.6.11. Per l'integrale Eq.6.9, una buona scelta di funzione peso è

$$w(x) = \frac{4 - 2x}{3}, \tag{6.13}$$

che è positiva e normalizzata. Integrando Eq.6.11 si ottiene

$$y = \frac{x(4 - x)}{3}, \quad \text{ovvero} \quad x = 2 - \sqrt{4 - 3y}. \tag{6.14}$$

Ora si procede alla stima MC dell'integrale Eq.6.12, che è

$$\frac{1}{N} \sum_{i=1}^{N} \frac{f(x(y_i))}{w(x(y_i))} \tag{6.15}$$

con y_i numeri casuali uniformi, w e $x(y)$ le funzioni appena viste. Il vantaggio dell'uso di questa procedura è evidente analizzando in Fig.6.5 l'errore nei due casi con importance sampling o senza: nel primo caso l'errore è un ordine di grandezza minore. Si nota anche che l'errore scende come $\sqrt{N}$. Nella stessa Figura sono mostrate le funzioni f, w, e f/w. Visibilmente quest'ultima è più piatta di f; notiamo comunque che al di fuori dell'intervallo [0,1] la scelta di w diventa molto meno buona.

Generare distribuzioni diverse da quella uniforme analiticamente o con semplici manipolazioni, come appena visto, è possibile solo in pochi casi. Uno di questi è la distribuzione esponenziale

$$w(x) = \frac{e^{-x/\lambda}}{\lambda} \qquad x \geq 0. \tag{6.16}$$

L'integrale è

$$y(x) = \int_0^x dt\, \frac{e^{-x/\lambda}}{\lambda} = 1 - e^{-x/\lambda} \tag{6.17}$$

e quindi

$$x = -\lambda \log(1 - y), \tag{6.18}$$

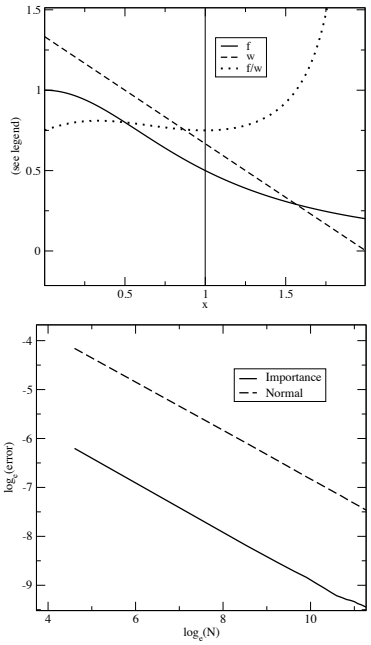


Figura 6.5: Integrando e funzione peso Eq.6.13 (a sinistra); errore normale e importance-sampled.

con y distribuito uniformemente in [0,1). Un altro esempio ovvio è quello dei numeri uniformi in [a,b) dati numeri uniformi in [0,1),

$$x = a + (b - a)y, \tag{6.19}$$

che lasciamo come esercizio. Un altro caso semplice è il metodo Box-Müller per generare numeri (due alla volta, in effetti) secondo la distribuzione gaussiana

$$w(x) = e^{-x^2/2}. \tag{6.20}$$

La probabilità che un numero distribuito secondo w sia generato in un intervallino $dx_1 dx_2$ in due dimensioni è

$$P = e^{-(x_1^2+x_2^2)/2} dx_1 dx_2. \tag{6.21}$$

Passando a coordinate polari

$$r = (x_1^2 + x_2^2)^{1/2}, \qquad \theta = \arctan x_2/x_1, \tag{6.22}$$

Eq.6.21 diventa, ponendo u=$r^2/2$,

$$P = e^{-r^2}\, r\, dr\, d\theta = e^{-u}\, du\, d\theta. \tag{6.23}$$

Quindi se generiamo dei numeri θ uniformi e dei numeri u esponenziali, otterremo due numeri gaussiani come

$$x_1 = \sqrt{2u}\cos\theta \quad \text{e} \quad x_2 = \sqrt{2u}\sin\theta. \tag{6.24}$$

A parte casi particolari, la generazione di distribuzioni arbitrarie è in genere impossibile, soprattutto in alta dimensionalità. Per questo si utilizza il metodo di Metropolis.

6.4 *Importance sampling: Metropolis*

L'algoritmo di Metropolis è il più popolare dei metodi di importance sampling nel Montecarlo, e certamente il più potente in rapporto alla sua semplicità. È uno dei 10 top algorithms del ventesimo secolo– in realtà, proprio il primo [4]. Ha una storia curiosa: una versione accreditata [5] è che il tema dell'articolo originale (N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, ed E. Teller, J. Chem. Phys. 21, 1087 (1953), che è peraltro di una chiarezza disarmante), l'equazione di fase di un fluido di sfere rigide, originalmente proposto da Edward Teller, sia stato sviluppato e formalizzato da Marshall Rosenbluth, che inventò anche l'algoritmo; che Arianna Rosenbluth abbia programmato il calcolatore (primitivo) con cui si fecero i conti; che Augusta Teller sia finita sull'articolo nonostante non avesse fatto niente perchè il marito voleva convincerla a ritornare al lavoro; e infine, ironia delle ironie, che Nicholas "Nick" Metropolis finisse sull'articolo, pur non avendo contribuito, perchè era il capo del centro di calcolo di Los Alamos. Curiosamente nessuno degli autori utilizzò più quella tecnica, o quasi: solo due articoli, a firma dei Rosenbluth, del 1954-55 [6].

[4]

[5]

[6]

Metropolis è una ricetta per generare sequenze casuali con distribuzione arbitraria. Queste sequenze possono consistere di punti in

uno spazio delle fasi o di configurazioni geometriche di un sistema, o semplicemente in un certo intervallo di numeri reali; questi punti sono spesso denominati "walker". Risulta che i walker tendono ad essere correlati a breve distanza (due successivi, o uno ogni due), ma non troppo fortemente, come discuteremo poi. Metropolis ha interessanti interpretazioni e applicazioni di tipo meccanico statistico su cui torneremo. La ricetta per generare una sequenza con distribuzione w è estremamente semplice. In una singola frase:

accettare la mossa casuale dal punto x_i al punto x_{i+1} con probabilità $w(x_{i+1})/w(x_i)$.

Elaboriamo un po' per chiarire il concetto, riferendoci semplicemente a punti (numeri) casuali sull'asse reale (anche se il discorso vale allo stesso modo per spazi complicati e multidimensionali); ricordiamo che il nostro scopo è generare più punti dove w è grande e meno dove è piccola, precisamente in numero proporzionale a w. Supponiamo di avere un punto x_i cui corrisponde il valore $w(x_i)$ della distribuzione desiderata. Generiamo casualmente un punto x_t (t per *trial*, prova); ad esempio, potremmo decidere di spostarci a caso da x_i di $\pm\delta$. Calcoliamo $w(x_t)$. Se $w(x_t)/w(x_i)>1$, il punto appena generato è più probabile del precedente in termini della distribuzione w. Accettiamo quindi sicuramente questo nuovo punto, ponendo $x_{i+1}=x_t$. Ora, se $w(x_t)/w(x_i)<1$, stiamo andando verso regioni meno desiderabili secondo la misura fornita da w; tuttvia *non rifiutiamo la mossa a priori: la accettiamo condizionalmente*. (In questo senso, Metropolis è un processo bayesiano.) Estraiamo un numero casuale uniforme $\xi\in[0,1]$, che funziona da generatore di probabilità uniforme. Se $w(x_t)/w(x_i) > \xi$ accettiamo la mossa, e poniamo $x_{i+1}=x_t$. Se

$$w(x_t)/w(x_i) < \xi$$

rifiutiamo la mossa, e poniamo $x_{i+1}=x_i$, cioè contiamo di nuovo il punto precedente. Ottenuta, dopo molti di questi passi, una lunga sequenza di punti sull'asse reale, possiamo ordinarli dal più piccolo al più grande, e rappresentarli in un istogramma secondo il numero di punti che cade in ogni intervallino. L'istogramma riprodurrà fedelmente la funzione w (normalizzata). Ovviamente, se si vuole utilizzarli per calcolare un integrale o simili, lo si può fare on the fly e non occorre ordinare e istogrammare. Come esempio, in Fig.6.6 sono mostrate una distribuzione gaussiana (nera) e una data da una sovrapposizione di gaussiane.

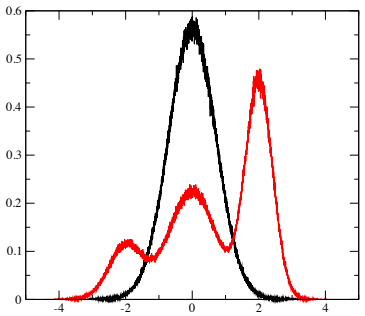


Figura 6.6: Sequenze generate con Metropolis, ordinate e istogrammate, per una distribuzione gaussiana (nero), e una distribuzione data da una sovrapposizione di tre gaussiane di peso 0.5, 1, 2 centrate in –2, 0, 2. Tutte e due le distribuzioni sono normalizzate.

Per dimostrare che Metropolis fa effettivamente quel che abbiamo detto, consideriamo una *media di insieme*, cioè su tanti "walkers" che si muovono indipendentemente nello spazio X di nostro interesse (ad esempio, come prima, punti sull'asse reale). Definiamo $N(x)$ la densità di walkers al punto x, che misura il numero di walkers (cioè di sequenze indipendenti) che si trovano attualmente in x. La variazione di questa densità causata dal moto dei walker da x ad y è

$$N(x)P(x\to y) - N(y)P(y\to x) = N(x)P(x\to y)\left[\frac{N(x)}{N(y)} - \frac{P(y\to x)}{P(x\to y)}\right] \tag{6.25}$$

dove P è la probabilità che il walker passi da x a y nel suo vagabondare. Questa variazione è nulla in equilibrio, quando cioè tanti walker vanno da x a y quanti da y a x. In quel caso

$$\frac{N_e(x)}{N_e(y)} \equiv \frac{N(x)}{N(y)} = \frac{P(y \to x)}{P(x \to y)}. \tag{6.26}$$

È quindi plausibile (e può essere dimostrato) che la distribuzione tenderà all'equilibrio dopo molti passi. Ora dobbiamo mostrare che la distribuzione di equilibrio $N_e \simeq w(x)$. La probabilità di salto da x ad y è il prodotto

$$P(x \to y) = T(x \to y)A(x \to y) \tag{6.27}$$

della probabilità T di scegliere y come punto di arrivo e della probabilità A di accettare la mossa. Si intuisce che se y può essere raggiunto a partire da x in un passo, così anche x potrà esserlo a partire da y, cioè

$$T(x \to y) = T(y \to x), \tag{6.28}$$

e quindi

$$\frac{N_e(x)}{N_e(y)} = \frac{A(y \to x)}{A(x \to y)}. \tag{6.29}$$

Ora si ha

$$w(x) > w(y) \;\Rightarrow A(y \to x) = 1,\; A(x \to y) = \frac{w(x)}{w(y)}, \tag{6.30}$$

oppure

$$w(x) < w(y) \;\Rightarrow A(x \to y) = 1,\; A(y \to x) = \frac{w(y)}{w(x)}. \tag{6.31}$$

In ambedue i casi la popolazione di equilibrio dei walkers (o se preferiamo la popolazione accumulata da una unica lunghissima sequenza) soddisfa

$$\frac{N_e(x)}{N_e(y)} = \frac{w(y)}{w(x)}, \tag{6.32}$$

e quindi i walkers (cioè i nostri punti) sono distribuiti secondo w come desiderato.

Due punti importanti riguardano la correlazione della sequenza e la modalitàà di scelta delle configurazioni. Quanto al primo punto, è chiaro che se punti adiacenti nella sequenza sono correlati tra loro, non si può a rigore calcolare un integrale del tipo

$$I = \int w(x) f(x) dx \tag{6.33}$$

come media su un campione di x. Si potrebbe campionare su punti non immediatamente adiacenti, ma separati da una certa distanza nella sequenza tale da sopprimere la correlazione, che quantificheremo nel contesto della Sezione successiva. Per il secondo punto, nel caso della sequenza di punti sull'asse reale, ad esempio, potremmo ad esempio scegliere il successivo punto di trial come

$$x_t = x_i + 2\delta(\texttt{ran1} - 0.5),$$

dove `ran1` genera numeri casuali uniformi. Il parametro δ va scelto in modo oculato; se supponiamo che x sia vicino a un massimo di w (il posto dov'è più probabile che sia), una mossa "avventurosa", cioè di grande δ, produrrebbe un y di bassa probabilità che verosimilmente verrebbe rifiutato il più delle volte; per contro una mossa conservativa (piccolo δ) otterrebbe più accettazioni, ma rimanendo in una regione limitata intorno al punto iniziale. Di norma si consiglia di aggiustare δ in modo che circa metà delle mosse siano accettate.

6.5 *Applicazioni: meccanica statistica*

L'applicazione del metodo a problemi di meccanica statistica è naturale. In ossequio al postulato della meccanica statistica secondo cui ogni microstato di un sistema è realizzato con probabilità data dal peso di Boltzmann $\exp(-\beta E(x))$ dove la costante β=1/k_BT ed $E(x)$ è l'energia del sistema, la funzione w in questo caso è appunto

$$w(x) = \exp(-\beta E(x)). \tag{6.34}$$

Evidentemente generando questa w si generano stati che appartengono all'ensemble canonico. Se ad esempio $E(x)$=$x^2/2$, cioè l'energia potenziale di un oscillatore armonico (assunte pari a 1 le costanti dimensionali), la distribuzione è una semplice gaussiana. Naturalmente possiamo immaginare potenziali più complicati, ad esempio

$$E(x) = x^4 - 2x^2 + 0.5x^3, \tag{6.35}$$

(la curva sottile in Fig.6.7) nel qual caso la distribuzione sarà

$$w(x) = \exp(-\beta(x^4 - 2x^2 + 0.5x^3)). \tag{6.36}$$

La presenza di β permette di modificare la distribuzione senza cambiarne la forma analitica. La distribuzione può essere interpretata come la distribuzione di probabilità canonica della presenza di una particella classica in una certa superficie di energia potenziale, in contatto con un bagno termico a temperatura T. Come si vede in Fig.6.7, la probabilità di visitare la valle di energia più alta diminuisce, come ci si aspetta, al diminuire di T (all'aumentare di β), come se il sistema si "congelasse" nel minimo assoluto di energia.

Può capitare che, per T abbastanza basse e per speciali condizioni iniziali, il sistema rimanga confinato in uno stato metastabile. In Fig.6.7, a destra, sono rappresentate le distribuzioni per β=30 (molto alto, cioè T molto bassa) ottenute con una sequenza partita da x=0.8, quindi nel bacino metastabile, con 5 milioni di passi Montecarlo (linea continua) come nella stessa Figura a sinistra, e 50 milioni di passi (linea tratteggiata). Si vede che solo con moltissimi passi ("moltissimo tempo") il sistema riesce ad uscire dal minimo metastabile. Questo qualitativamente simula sistemi metastabili separati dallo stato stabile da una barriera di energia abbastanza alta da rendere impossibile la transizione in tempi ragionevoli.

Nell'interpretazione termico-statistica che stiamo utilizzando, è possibile studiare la termodinamica di vari sistemi. In Fig.6.8 sono

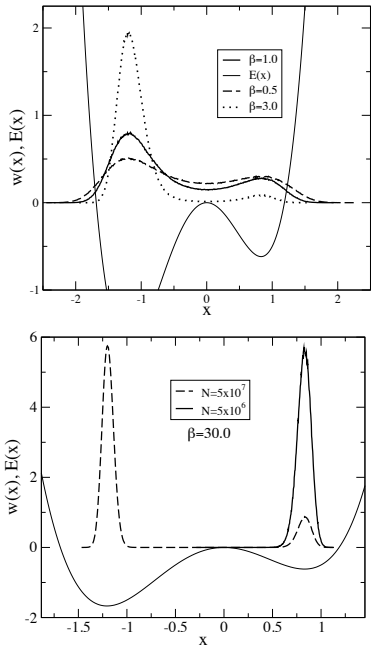


Figura 6.7: A sinistra, distribuzioni Eq.6.36 per diversi β e 5×10^6 passi. Si vede che la probabilità si localizza quando la temperatura scende (β sale). A destra, le stesse distribuzioni, ma per β=30 con x_0=0.8 nel bacino metastabile. Con un numero "normale" di passi il sistema resta confinato nel bacino metastabile.

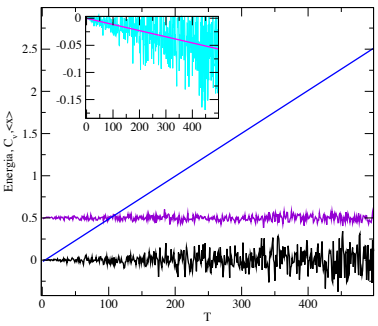


Figura 6.8: Quantità per l'oscillatore armonico (inset: posizione media nel caso anarmonico cubico).

riportate energia termica media, posizione media, e calore specifico per l'oscillatore anarmonico con energia $E{=}x^2/2$. La costante di Boltzmann è posta uguale a 1. Le prime due quantità sono ottenute tramite integrale MC sulla sequenza di Metropolis, e la terza dalle fluttuazioni dell'energia:

$$C_v = k_B\beta^2(\langle E^2\rangle - \langle E\rangle^2). \tag{6.37}$$

L'energia aumenta linearmente con T come previsto dall'equipartizione; il calore specifico ottenuto come derivata dell'energia media rispetto a T risulta essere $1/2$; in accordo con questo risultato, il calore specifico è costante e uguale a $1/2$ anche in forma di fluttuazione come qui sopra. Notiamo che questo succede perchè nell'energia abbiamo il solo termine di energia potenziale. Se avessimo usato l'energia totale, $p^2/2 + x^2/2$, ovvero x^2, dato che sia p che x sarebbero numeri casuali, l'energia termica sarebbe aumentata con pendenza 1. Infine la posizione media è zero, come atteso per l'oscillatore armonico. Nell'inset è mostrata la posizione media per un caso anarmonico (un termine cubico negativo) che decresce linearmente con T come previsto teoricamente [7].

[7]

È importante ricordare che, benchè qui facciamo esempi in cui l'energia è nota, questo non è necessario in generale. Ad esempio, in un sistema a molte particelle, l'energia potenziale di ognuna di esse risulta dalla somma delle interazioni con le altre; questa cambia dinamicamente con il cambiare della configurazione collettiva, ed è quindi impossibile da predire o esprimere analiticamente. Il bello del metodo di Metropolis è che genera una distribuzione (e quindi quantità medie generate su di essa) che in media tiene conto di tutte queste variazioni.

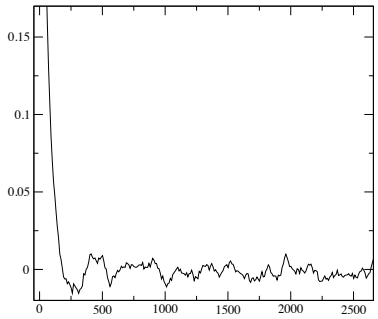


Figura 6.9: Autocorrelazione dell'energia di un oscillatore integrata su una sequenza di Metropolis (β=0.3).

In chiusura di questa Sezione, invece, esaminiamo rapidamente la correlazione della sequenza. Per quantificarla, si usa la funzione di autocorrelazione della funzione integranda,

$$C(k) = \frac{\langle f_i f_{i+k}\rangle - \langle f_i\rangle^2}{\langle f_i^2\rangle - \langle f_i\rangle^2}, \tag{6.38}$$

con

$$\langle f_i f_{i+k}\rangle = \frac{1}{N-k}\sum_{i=1}^{N-k} f(x_i)f(x_{i+k}). \tag{6.39}$$

Chiaramente $C(0){=}1$, e una C apprezzabilmente non nulla per $k{\neq}0$ indica correlazione a distanza k nella sequenza. Il problema è delicato e molto dipendente dal problema. In Fig.6.9 è mostrata a titolo di esempio la autocorrelazione dell'energia media di un oscillatore armonico

$$\langle E\rangle = \int x^2 \exp\left(-\beta x^2\right) dx,$$

con β=0.3 su una sequenza di Metropolis con $w{=}\exp\left(-\beta x^2\right)$. In questo caso la sequenza si scorrela abbastanza rapidamente, benchè le fluttuazioni siano grandi anche a tempi lunghi (500 step). Alzando β (abbassando la temperatura) fino ad un certo punto la fluttuazione viene soppressa; forti cali di temperatura, in cui il sistema resta fortemente confinato, tendono a produrre più forti correlazioni.

6.6 *Applicazione: minimizzazione "globale" con simulated annealing*

Finora abbiamo scelto una energia e valutato la distribuzione statistica della coordinata in funzione della temperatura, e abbiamo visto che la distribuzione si localizza nei minimi dell'energia quanto T scende. Possiamo invertire il ragionamento e trovare invece il minimo dell'energia (che supponiamo non sia nota esplicitamente in toto) generando la distribuzione per temperature decrescenti: partiremo da una sequenza che viaggia su tutto lo spazio disponibile, e gradualmente la localizzeremo. Si possono anche fare cicli di "annealing" (dal nome di un trattamento termico che si usa per eliminare i difetti nei materiali), cioè riscaldare e raffreddare il sistema, facendolo uscire da un minimi dell'energia e ricongelandolo (spiegabilmente) in un altro. Questa procedura si chiama "simulated annealing".

Il SA è capace di determinare il minimo assoluto di una funzione multidimensionale (per esempio un landscape di energia); come può far sospettare l'uso del termine "assoluto", esistono di norma molti altri minimi più alti di quello assoluto, detti relativi o anche locali (nel senso che sono dei minimi solo per configurazioni del sistema confinate nelle vicinanze dei minimi stessi). La minimizzazione locale, ad esempio seguendo la derivata in discesa, raggiunge in modo efficiente il minimo locale più vicino, ma per il minimo globale questo sarà il caso solo se ci troviamo nelle sue vicinanze già dall'inizio (Fig.6.10, sinistra). Il Montecarlo invece, proprio per i salti "improbabili" che a volte fa, potrebbe essere in grado di sfuggire dal minimo locale e trovare quello globale (Fig.6.10, destra). Come esempio, in Fig.6.11 mostriamo la distribuzione di Boltzmann $w(x)=e^{-\beta E(x)}$ con

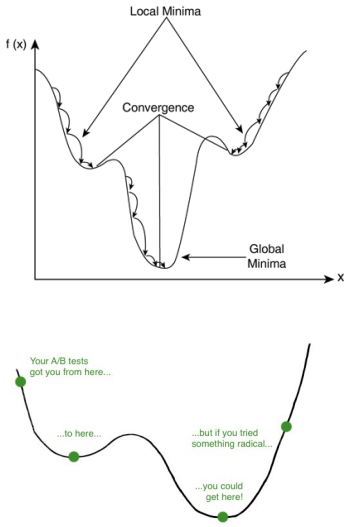
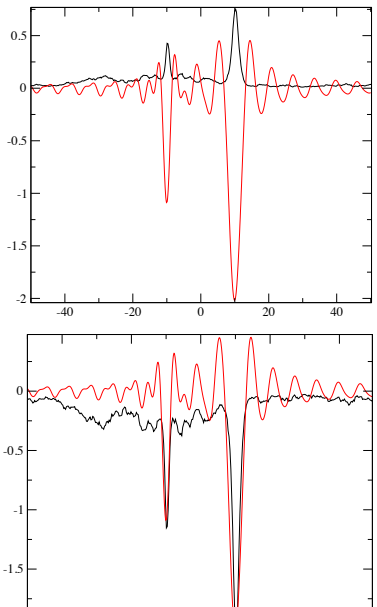


Figura 6.10: Minimi locali e globali.

$$E(x) = -\frac{\sin 2(x-x_1)}{2(x-x_1)} - 2\frac{\sin 2(x-x_0)}{2(x-x_0)} \tag{6.40}$$

con x_1=–x_0=10 generata dal simulated annealing. Il potenziale (curva rossa) non è particolarmente semplice da campionare. È interessante notare che la distribuzione segue fedelmente il potenziale (in figura a destra): invertendola e riscalando opportunamente si vede come segua discretamente bene il potenziale, soprattutto in corrispondenza ai minimi. La distribuzione non è quella che avremmo generato a T fissa, ma invece una specie di sovrapposizione a diverse T. Nel caso di Fig.6.11, la schedule termica è 15 passi a β= 3, 2.4, 1.9, 1.4, 1, 0.5, 0.25, 0.1, e poi risalita fino a 3. Come si vede, l'abbassamento della temperatura fa (con un po' di fortuna) congelare il sistema nel minimo assoluto.



6.7 Traveling salesman problem; minimizzazione e annealing simulato

6.8 Applicazioni: Ising

6.8.1 Transizioni di fase: generalità e mean-field

6.8.2 Dualità e esponenti di ising

kramers wannier duality and TC; onsager; mean field

6.8.3 Metropolis per Ising

6.8.4 Stima di T_c via magnetizzazione, calore specifico, suscettività, cumulante di Binder

6. B - Metodo Montecarlo

6.1 Generazione e test numeri casuali

Proviamo a implementare il semplice generatore di numeri casuali che sfrutta la formula 6.2 della Sezione 6.1, mediante le seguenti righe di codice:

```
1  def myrandgen(a,c,M,r0,n=1):
2    if n==1:
3      return mod(a*r0+c,M)
4    else:
5      r=zeros(n)
6      r[0]=r0
7      for i in range(1,n):
8        r[i]=mod(a*r[i-1]+c,M)
9      return r
10
11 i=0
12 sets=((57,1,25,13,'s'), (57,13,4097,13,'sg'), (57,17,4111,13,'sr'))
13 for a,c,M,r0,p in sets:
14   i+=1
15   r=myrandgen(a,c,M,r0,1000)
16   figure(i)
17   plot(r[:-1],r[1:],p)
18   xlabel(r'$r_i$',fontsize=25)
19   ylabel(r'$r_{i+1}$',fontsize=25)
20   title('a='+str(a)+', c='+str(c)+', M='+str(M)+', r0='+str(r0))
```

Vediamo prima un pò in dettaglio le precedenti linee di codice che contengono alcune cose ancora non viste riguardo la programmazione in Python:

- Per prima cosa, definiamo una funzione che partendo da 5 parametri di input, ci restituisce uno o piu valori random. I parametri `a, c, M, r0` sono quelli necessari per poter calcolare la funzione **??**

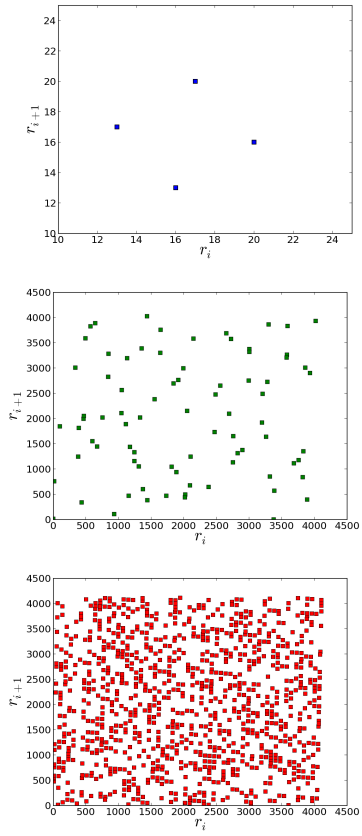


Figura 6.1:

che genera il numero random successivo. Abbiamo inserito anche il parametro n con il quale specifichiamo quanti numeri random vogliamo generare. Poichè abbiamo posto `n=1`, se non viene specificato 1 sarà preso come valore di default. A seconda che n sia uguale o diverso da 1 creiamo uno o piu numeri random, ma la funzione di base è `mod(n,m)`: essa restituisce il resto della divisione n diviso m. Nel caso che n sia diverso da uno creiamo un array di n zeri e mediante un ciclo `for` creiamo l'i-simo numero random partendo dal precedente e lo registriamo nella posizione i-sima. Una volta usciti dal ciclo chiediamo alla funzione di restituirci il vettore r appena creato.

- Ora vogliamo provare i tre differenti sets di variabili `a, c, M, r0` proposti nelle dispense, in maniera smart. Creiamo una lista sets che contiene tre liste contenenti ognuna un set diverso di valori delle variabili. Sfruttiamo il fatto che in Python si possa fare un ciclo su una lista di liste, come la nostra sets, ed inoltre possiamo attribuire ogni singolo valore di ogni singola lista ad una variabile diversa. Quindi ad ogni ciclo le variabili `a, c, M, r0` avranno i valori contenuti nelle liste di sets.
- All'interno del ciclo chiamiamo la nostra funzione `myrandgen`, che genera numeri random, in cui impostiamo a 1000 il numero n, mentre gli altri parametri cambieranno.
- Creiamo una figura e facciamo il plot di r_{i+1} in funzione di r_i: partendo dall'array `r` ne creiamo un altro che contenga tutti i valori di r dal primo al penultimo (`r[:-1]`) e un ancora altro che contenga tutti i valori di r dal secondo all'ultimo (`r[1:]`).
- Infine, impostiamo i label per gli assi usando una stringa simile al TEX, ponendo `r` prima degli apici; ingrandiamo anche un pochino la dimensione dei font modificando la variabile fontsize. Impostiamo anche il titolo scrivendo il set di variabili che stiamo usando per quel grafico.
- Nella funzione plot, notiamo anche la variabile in ingresso `p`, che viene impostata uguale al quinto valore di ogni lista. Questo, come vediamo, è una stringa di due caratteri: la `s` sta per square e le `b,g,r` stanno per i colori. Stiamo usando una forma contrattata per specificare che tipo di simbolo usare e di che colore vogliamo che sia.

Nella Figura 6.1 troviamo i tre grafici che il nostro, fin troppo elaborato, programma genera:

Come discusso nelle dispense, i primi sets di variabili generano numeri random che dopo poco si ripetono, quindi delle sequenze con alta correlazione: nel primo grafico i punti son solo 4, nel secondo sono un centinaio, ma ne stiamo chiedendo 1000. Ne terzo grafico, la distribuzione uniforme dei punti ci fa essere più ottimisti che la sequenza generata sia abbastanza buona.

Aggiungere Test sui momenti e sulla correlazione delle sequenze appena descrite e su quelle generate dal generatore di Python.

6.2 *Importance sampling: metodo analitico.*

Di seguito proviamo a verificare graficamente le formule trovate analiticamente nelle dispense per ottenere una sequenza di numeri casuali che segua una certa distribuzione $w(x)$. Il primo esempio è quello della seguente funzione peso $w(x) = \frac{4-2x}{3}$, che integrata e invertendo ci fornisce la relazione $x = 2 - \sqrt{4-3y}$, con y numeri casuali distribuiti uniformemente mentre le x saranno invece distribute come la funzione peso $w(x)$. Mediante le poche righe di codice seguente proviamo a verificare che sia effettivamente cosi:

```
N=10000
y=rand(N)
x=2-sqrt(4-3*y)
w=(4-2*x)/3

hist(x,10,normed=True)
plot(x,(4-2*x)/3,'s')

#sorting
x.sort() #sul posto, x viene sostituito da quello ordinato
x_sorted = sort(x) #il risultato va memorizzato su un altro array
```

Come notiamo dal grafico in Figura 6.2, la distribuzione dei valori dell'array x e la funzione peso analitica sono in buon accordo. L'istogramma è stato realizzato mediante la funzione `hist`: il primo argomento è l'array d'interesse, il secondo è il numero di bin, intervalli, all'interno dei quali contare quanti valori di x si hanno, con il terzo chiediamo che l'istogramma sia normalizzato.

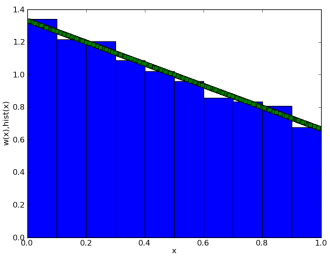


Figura 6.2:

Il plot della funzione lo si fa per punti (notare il 's') perchè la sequenza x non è ordinata. L'ordinamento è utile in qualche caso e lo si fa mediante la funzione `sort()`.

Ora che abbiamo generato la nostra sequenza random distribuita come $w(x)$, possiamo calcolare la somma 182 e confrontarla con la somma fatta usando una distribuzione uniforme, in questo modo:

```
fx=1/(1+x**2)
f=1/(1+y**2)

I_esatto = pi/4
I_imp = sum(fx/w)/N
I_unif = sum(f)/N

err_imp=log10(abs(I_imp - I_esatto))
err_unif=log10(abs(I_unif - I_esatto))

print 'Importance sampling: ' + str(I_imp) + ' Log10 error: ' + str(err_imp)
print 'Uniform sampling: ' + str(I_unif) + ' Log10 error: ' + str(err_unif)

#Output
Importance sampling: 0.785375419649 Log error: -4.64313796351
```

```
16 Uniform sampling: 0.787076269334 Log error: -2.77518062628
```

Abbiamo creato un array `fx` che contiene i valori della funzione campionata secondo i valori di x, da dividere poi per la funzione `w`. L'array f invece contiene i valori della stessa funzione campionata in maniera uniforme pero usando la sequenza random y. Come si può vedere dall'output, l'integrale calcolato usando l'importance sampling è piu vicino a quello vero, avendo un errore di due ordini di grandezza inferiore rispetto all'altro caso.

Volendo riprodurre il grafico dell'andamento dell'errore in funzione del numero di punti random di campionamento, mostrato nelle dispense in fig. 46, è necessario eseguire le righe di codice viste all'interno di un ciclo for nel quale modifichiamo il valore della variabile `N`. Questo però non basta a causa delle normali fluttuazioni nei valori dell'integrale, percui non otterremo degli andamenti quali lineari come in figura ma delle nuvole di punti. Per ridurre le fluttuazioni, dobbiamo eseguire piu volte il calcolo dell'integrale e fare la media per ogni valore di `N`; ci servirà perciò un ulteriore ciclo `for`. Lasciamo come esercizio la scrittura di questo codice e la produzione del grafico in figura 46. Inoltre, lasciamo a voi la verifica delle altre distribuzioni ottenute per via analitica, come la esponenziale (formula 185), uniforme su intervallo generico (186) e gaussiana (191).

6.3 *Importance sampling: algoritmo Metropolis.*

Quando non si conosce la forma analitica della funzione distribuzione che si vuole genenerare o non si può applicare il metodo analitico, si ricorre all'algoritmo Metropolis, uno dei più importanti algoritmi mai inventati.

L'algoritmo si riassume nei seguenti passi:

1. si sceglie una posizione di prova $x_t = x_i + \delta_i$, dove x_i è la posizione precedente e δ_i è un numero random nell'intervallo $[-\delta, \delta]$;
2. si calcola la probabilità $w = p(x_t)/p(x_i)$;
3. se $w \geq 1$, si accetta la nuova posizione, quindi $x_{i+1} = x_t$;
4. se $w < 1$, si genera un numero random tra 0 e 1;
   (a) se $r < w$, si accetta la nuova posizione, quindi $x_{i+1} = x_t$;
   (b) altrimenti la posizione viene rifiutata

Questi semplici passaggi, che si traducono in poche righe di codice, ci permettono di avere una sequenza di numeri random distribuiti come la funzione $p(x)$. Vediamo di seguito una implementazione che segue i passi appena visti:

```
1 xt=xi+delta*(2*rand()-1)
2 w=p(xt)/p(xi)
3
4 if w >=1:
5   xi=xt
6 elif w < 1:
7   r=rand()
8   if r < w:
```

```
9       xi=xt
```

Ovviamente, per poter funzionare vanno definite le variabili `delta`, `xi` e la funzione `p(x)`. Inoltre queste righe genererebbero un solo numero casuale, `xt` appunto, a partire dal precedente `xi`. Prima di completare la nostra routine, vediamo di seguito una versione più efficente:

```
1  xt=xi+delta*(2*rand()-1)
2  w=p(xt)/p(xi)
3  r=rand()
4  if w > r:
5    xi=xt
```

Può sembrare strano ma queste righe fanno esattamente la stessa cosa di quelle scritte in precedenza. Il trucco sta nel fatto che il numero random è sempre compreso tra 0 e 1, perciò quando $w > 1$ sarà sempre verificata anche la condizione $w > r$, quindi la mossa viene sempre accettata; l'altra condizione è che $r < w < 1$, che è quella che usiamo e che di fatto racchiude in se entrambe i due casi.

Quello che dovremmo aggiungere è un ciclo sul numero di mosse accettate e una lista/array che conservi tutte le mosse accettate, che costituiscono la sequenza di numeri random che stiamo cercando di generare. Vanno inoltre definite alcune parametri come delta e la posizione iniziale e la funzione peso che vogliamo usare. Completiamo quindi il nostro codice come segue.

```
1  def p(x):
2    return 2*exp(-(x)**2)+exp(-(x+3)**2)+exp(-(x-3)**2)
3
4  delta=3
5  x=[0.]
6  N=5000
7  naccept=0
8  counter=0
9
10 while naccept < N:
11   counter+=1
12   xt=x[-1]+delta*(2*rand()-1)
13   w=p(xt)/p(x[-1])
14   r=rand()
15   if w > r:
16     x.append(xt)
17     naccept+=1
18
19 print naccept/counter
```

Per poter salvare tutte le posizioni accettate usiamo una lista x di cui poniamo a 0 il primo elemento che rappresenta la posizione di partenza. All'interno del ciclo while usiamo `x[-1]` per leggere l'ultima posizione registrata/accettata e usiamo la funzione `append()`

per aggiungere alla lista la posizione xt in caso questa sia stata accettata. Il ciclo è eseguito sintanto che la variabile naccept raggiunge il valore desiderato, 1000 in questo caso, che sarà anche la lunghezza finale della lista `x`. Per trovare un valore accettabile di delta è utile controllare il rapporto tra tutti i passi generati e quelli accettati, come facciamo nell'ultima riga: questo rapporto normalmente dovrebbe essere minore di 0.5 per avere delle sequenze accettabili. Ora vediamo se la nostra sequenza random `x` è distribuita come vogliamo, cioè come la funzione.

```
hist(x,100,normed=True)
```

Cosi creiamo direttametne un istogramma normalizzato con 100 intervalli. Altrimenti, per fare un grafico per punti:

```
h,b=histogram(x,100,normed=True)
plot(b[:-1],h)
```

Di seguito i grafici che si ottengono:

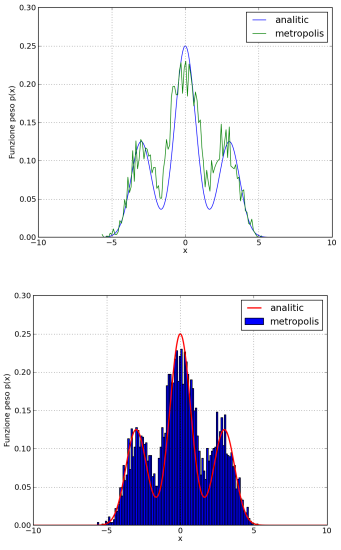


Figura 6.3:

Come si nota, anche con una sola esecuzione del programma e usando non tantissimi passi, l'istogramma è piuttosto simile alla funzione analitica che volevamo riprodurre. Se volessimo avere una maggiore precisione dovremmo aumentare la lunghezza della sequenza e fare delle medie su più istogrammi.

Quando si a che fare con funzioni periodiche, è bene limitare la generazione di nuove posizioni all'interno di un intervallo stabilito. Altrimenti il walker andrà in giro per tutto lo spazio cercando di riprodurre tutte le copie periodiche della funzione. Per esempio volendo riprodurre la funzione di hanning con periodo 10, senza limiti e un delta di 1 e 5, otteniamo quanto mostrato in Figura 6.4

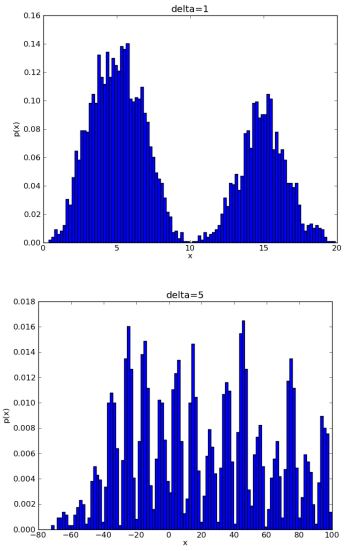


Figura 6.4:

Nella prima figura si riconoscono 2 periodi positivi della funzione, non riprodotti al meglio visto che hanno diversa altezza. Molte più copie si hanno se il `delta` è 5, ma l'accuratezza diminuisce ulteriormente. E' perciò doveroso porre un limite alle posizioni generate per replicare accuratamente l'intervallo che si desidera. Potremo modificare il ciclo `while`, introducendo una condizione if sui valori della `xt`, come segue:

```
def p(x):
  return 0.5-0.5*cos(2*pi*x/10)

delta=1.
N=50000
x=[5.]
naccept=0
counter=0.

while naccept < N:
  xt=x[-1]+delta*(2*rand()-1)
  counter+=1
  if xt <10 and xt > 0:
    w=p(xt)/p(x[-1])
```

```
15      r=rand()
16      if w > r:
17        x.append(xt)
18        naccept+=1
19
20  print naccept/counter
21  hist(x,100,normed=True)
22  xx=arange(0,10,0.1)
23  plot(xx,p(xx)/4.5)
```

In questo modo escludiamo le posizioni che sono fuori dall'intervallo $[0,10]$, in questo modo tutti le `N` posizioni generate saranno al suo interno. Facciamo notare come la posizione iniziale sia in questo caso 5, sempre in corrispondenza del massimo della funzione. Con le ultime due righe facciamo un grafico anche della funzione analitica (riscalata). In Figura 6.5 è mostrato il risultato.

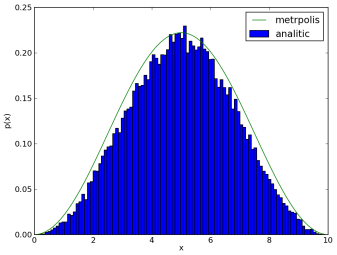


Figura 6.5:

6.4 *Simulated annealing*