Appunti di Metodi Numerici

Vincenzo Carbone, Leonardo Primavera & Fedele Stabile

Dipartimento di Fisica, Università della Calabria,

Rende (CS), Italy

Parte I

Tecniche numeriche per la soluzione di alcuni problemi di fisica

Capitolo 1

Derivata di una funzione

1.1 Introduzione

Il problema che ci poniamo in questa sezione é quello classico nella analisi di calcolare la derivata $f'(x) = df/dx$ di una funzione $f(x)$, supponendo ovviamente che la funzione sia nota per punti. Dato quindi un insieme di punti $\{x_i\}$ $(i = 0, 1, \ldots, N+1)$ che costituiscono una griglia di passo $\Delta x_i = x_{i+1} - x_i$, in corrispondenza di ogni punto della griglia conosciamo il valore di una funzione[1] $f(x_i)$ supposta analiticamente differenziabile in un intervallo $[a, b]$ $(x_0 = a,\ x_{N+1} = b)$. Se i punti della griglia sono equispaziati, allora il passo Δx_i sará costante $\Delta x = (b-a)/(N+1)$, ed i punti della griglia saranno dati da $x_i = a + i\Delta x$. Dall'analisi matematica classica sappiamo che per definizione

$$\left.\frac{df}{dx}\right|_x = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

per cui il calcolo della derivata é fatto usando un incremento infinitesimo. Da un punto di vista numerico, in una primissima approssimazione possiamo fare il seguente ragionamento. Supponiamo che, fissato l'intervallo $[a, b]$, i punti della griglia siano tanti, ossia $N \gg 1$. Allora, non potendo considerare intervalli infinitesimi ma solo intervalli finiti di passo al piú Δx, la derivata della funzione in uno dei punti della griglia sará data semplicemente dalla ovvia espressione

$$\left.\frac{df}{dx}\right|_{x_i} \simeq \frac{f(x + \Delta x) - f(x)}{\Delta x} \equiv \frac{f(x_{i+1}) - f(x_i)}{\Delta x}$$

[1]Per brevitá in genere useremo la notazione $f(x_i) \equiv f_i$ per indicare il valore della funzione $f(x)$ calcolato nel punto $x = x_i$.

Ci si chiede quale sia grado di approssimazione della funzione derivata ricavata in questo caso. Intuitivamente si capisce che la precisione dipenderá dal valore del passo della griglia. Tanto piú piccolo sará Δx tanto piú precisa sará l'approssimazione per la derivata. E' comunque possibile calcolare la derivata di una funzione con precisioni differenti, usando differenti punti della griglia. Le formule di basso ordine che consentono di calcolare le derivate sono presentate nella sezione seguente.

1.2 Metodi alle differenze finite

Cosí come si é approssimata una funzione $f(x)$ con un polinomio, l'idea dei metodi alle differenze finite consiste nell'approssimare la funzione $f^{(n)}(x)$, che rappresenta la derivata n–esima di $f(x)$, con un polinomio di interpolazione di grado $k-1$, ossia $f_i^{(n)} \simeq L_k(x_i)$. La funzione derivata coincide con il polinomio sui punti della griglia che sono noti. L'idea principale dei metodi alle differenze finite consiste nello scrivere il polinomio $L_k(x_i)$, che intérpola la derivata della funzione, come una combinazione lineare dei $(J+1)$ valori della funzione stessa $f(x_j)$ che sono vicini al punto $f(x_i)$

$$f_i^{(n)} \equiv \left.\frac{d^n f}{dx^n}\right|_{x=x_i} \simeq \sum_{j=-J_1}^{J_2} c_j f_{i+j} \tag{1.1}$$

L'equazione (1.1) costituisce la base per il calcolo delle derivate. Per ogni scelta dei punti J_1 e J_2, ossia per ogni scelta del numero minimo di punti della griglia da coinvolgere nel calcolo della derivata, si calcola il valore dei coefficienti c_j. Ovviamente é possibile aumentare il grado di precisione della derivata, oppure aumentare il grado n della derivata da calcolare, aumentando il numero di punti della griglia coinvolti.

Consideriamo innanzitutto il calcolo della derivata prima, e scegliamo $J_1 = J_2 = 1$, ottenendo dalla (1.1) l'espressione seguente

$$\left.\frac{df}{dx}\right|_{x=x_i} \simeq c_{-1} f_{i-1} + c_0 f_i + c_1 f_{i+1} \tag{1.2}$$

Quindi la derivata prima é calcolata usando tre differenti punti della griglia spaziale, come indicato nella figura 1.1. Poiché in genere il numero di punti é grande, e quindi Δx é piccolo, per calcolare i coefficienti della equazione (1.2) e per definire il grado di precisione della derivata, usiamo una espansione in serie di Taylor della funzione calcolata nei punti $x_{i\pm1}$, nell'intorno del punto x_i

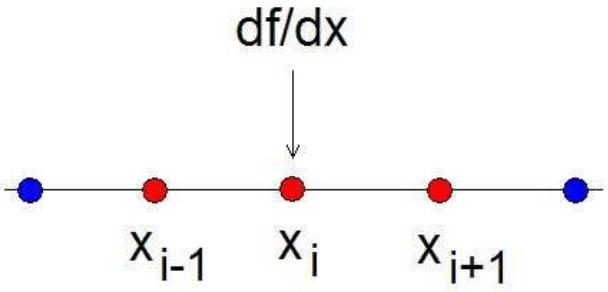


Figura 1.1: Usando un metodo alle differenze finite di ordine basso, per calcolare la derivata prima della funzione in un punto, bisoga considerare il punto stesso ed i due punti adiacenti.

$$\begin{aligned} f_{i\pm1} &\simeq f_i + (x_{i\pm1} - x_i)\left.\frac{df}{dx}\right|_{x_i} + \frac{(x_{i\pm1} - x_i)^2}{2}\left.\frac{d^2f}{dx^2}\right|_{x_i} + \\ &+ \frac{(x_{i\pm1} - x_i)^3}{6}\left.\frac{d^3f}{dx^3}\right|_{x_i} + \dots \end{aligned} \tag{1.3}$$

Sostituendo queste espressioni nella (1.2), e ricordando che $\Delta x = x_{i+1} - x_i = x_i - x_{i-1}$, trascurando per il momento il termine in derivata terza, si ottiene l'equazione seguente

$$\begin{aligned} \left.\frac{df}{dx}\right|_{x_i} &\simeq (c_{-1} + c_0 + c_1) f_i + (c_1 - c_{-1})\Delta x \left.\frac{df}{dx}\right|_{x_i} + \\ &+ (c_1 + c_{-1})\frac{\Delta x}{2}\left.\frac{d^2f}{dx^2}\right|_{x_i} + \dots \end{aligned} \tag{1.4}$$

Per avere una approssimazione della derivata prima bisogna quindi che siano verificate le due condizioni seguenti:

$$\begin{cases} c_{-1} + c_0 + c_1 = 0 \\ c_{-1} - c_1 = 1/\Delta x \end{cases}$$

Poiché siamo in presenza di due equazioni per tre incognite c_j, non é possibile determinare tutti e tre i coefficienti, ma soltanto due, che risultano funzione

del terzo. Introducendo quindi per semplicitá il parametro libero $\xi = c_0 \Delta x$, si ottiene finalmente

$$\left.\frac{df}{dx}\right|_{x_i} \simeq \frac{(1-\xi) f_{i+1} + 2\xi f_i + (1+\xi) f_{i-1}}{2\Delta x} + E(\Delta x) \tag{1.5}$$

Il termine $E(\Delta x)$ é l'errore che si commette, ossia tutto quello che resta nello sviluppo in serie di Taylor, e quindi é un parametro che definisce il grado di accuratezza con cui si stima la derivata:

$$E(\Delta x) \simeq -\frac{\xi \Delta x}{2} \left.\frac{d^2 f}{dx^2}\right|_{x_i = z} - \frac{\Delta x^2}{6} \left.\frac{d^3 f}{dx^3}\right|_{x_i = z} + \dots \tag{1.6}$$

(z é un punto interno all'intervallo $x + \Delta x$). Dalla (1.5) si vede allora che per stimare la derivata in un punto bisogna usare oltre al punto stesso, i due punti ad esso adiacenti. Per ottenere le formule standard della derivata bisogna peró precisare un valore per il parametro libero ξ. In genere ξ assume tre differenti valori da cui si ottengono tre differenti formule per la stima delle derivate (vedi figura 1.2).

Derivate in avanti: ($\xi = -1$)

$$\begin{aligned} \left.\frac{df}{dx}\right|_{x_i} &\simeq \frac{f_{i+1} - f_i}{\Delta x} + E(\Delta x) \\ E(\Delta x) &\simeq \frac{\Delta x}{2} \left.\frac{d^2 f}{dx^2}\right|_z \sim 0(\Delta x) \end{aligned} \tag{1.7}$$

Derivate indietro: ($\xi = 1$)

$$\begin{aligned} \left.\frac{df}{dx}\right|_{x_i} &\simeq \frac{f_i - f_{i-1}}{\Delta x} + E(\Delta x) \\ E(\Delta x) &\simeq -\frac{\Delta x}{2} \left.\frac{d^2 f}{dx^2}\right|_z \sim 0(\Delta x) \end{aligned} \tag{1.8}$$

Derivate centrate: ($\xi = 0$)

$$\begin{aligned} \left.\frac{df}{dx}\right|_{x_i} &\simeq \frac{f_{i+1} - f_{i-1}}{2\Delta x} + E(\Delta x) \\ E(\Delta x) &\simeq \frac{\Delta x^2}{6} \left.\frac{d^3 f}{dx^3}\right|_z \sim 0(\Delta x^2) \end{aligned} \tag{1.9}$$

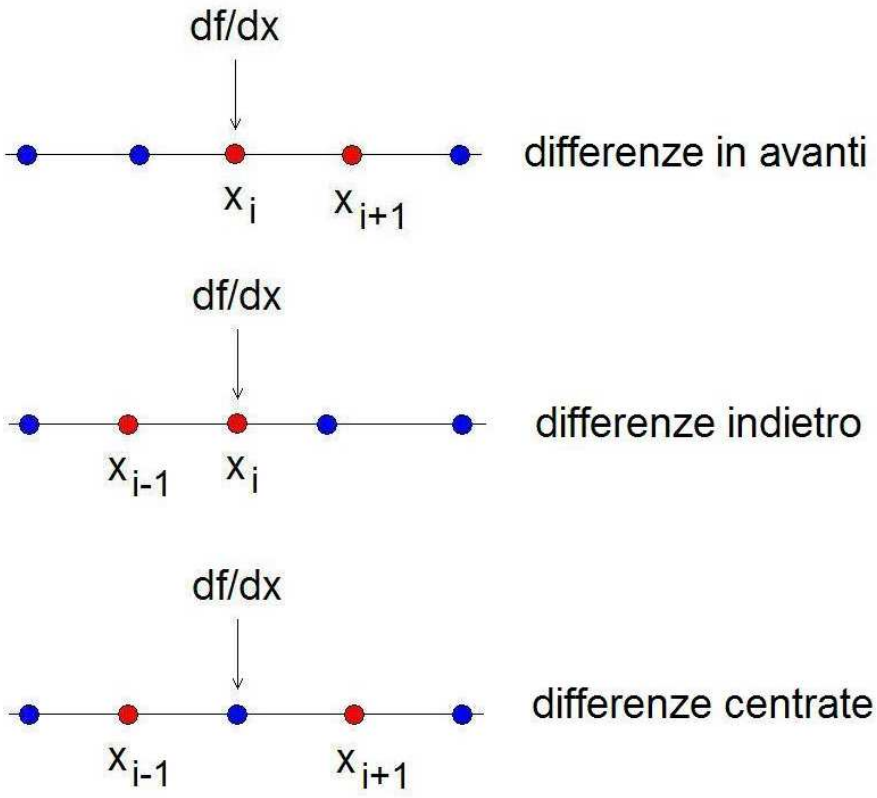


Figura 1.2: Riportiamo schematicamente le tre formule di derivazione che si ottengono usando schemi alle differenze finite di ordine basso.

Possiamo notare che tutte e tre le formule usano la funzione calcolata in due punti della griglia, peró la formula per le derivate centrate consente di calcolare la derivata con una precisione piú alta di quella delle altre due. Il motivo numerico é che il parametro ξ compare nell'espressione per l'errore, per cui se $\xi = 0$ il termine corrispondente di errore si elimina. Tuttavia, come é facile rendersi conto, il motivo fisico é che la tangente (derivata) alla curva nel punto x_i ha una pendenza piú vicina al segmento che unisce i due punti adiacenti ad x_i, rispetto alle pendenze delle rette che uniscono i punti x_i ed $x_{i\pm1}$.

L'espressione (1.9) pur essendo piú precisa come stima per la derivata, non puó essere usata per calcolare la derivata ai bordi dell'intervallo, ossia $f'(a)$ ed $f'(b)$. Poiché le altre due formule hanno una precisione inferiore, non é neanche utile calcolare le derivate ai bordi (usando le (1.7) e (1.8) per questo scopo) con precisione inferiore. Ci occorrono allora differenti formule per questo scopo. Ció puó essere fatto usando i valori $J_1 = 2$ e $J_2 = 0$ nella (1.1), per cui si ottiene

$$\left.\frac{df}{dx}\right|_{x_i} \simeq \frac{3f_i - 4f_{i-1} + f_{i-2}}{2\Delta x} + 0(\Delta x^2) \tag{1.10}$$

oppure ponendo $J_1 = 0$ e $J_2 = 2$ nella (1.1), ottenendo cosí

$$\left.\frac{df}{dx}\right|_{x_i} \simeq \frac{-3f_i + 4f_{i+1} - f_{i+2}}{2\Delta x} + 0(\Delta x^2) \tag{1.11}$$

Ovviamente la (1.10) si usa per calcolare la derivata sul bordo destro mentre la (1.11) quella sul bordo sinistro. Queste due equazioni, accoppiate con la (1.9) per i punti centrali dell'intervallo, danno una approssimazione della derivata fino all'ordine $0(\Delta x^2)$.

Allo stesso modo si definiscono le derivate di ordine superiore. Per esempio una formula classica per la derivata del secondo ordine, che coinvolge solo tre punti sulla griglia é data da

$$\begin{aligned} \left.\frac{d^2f}{dx^2}\right|_{x_i} &\simeq \frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2} + E(\Delta x) \\ E(\Delta x) &\simeq \frac{\Delta x^2}{12}\left.\frac{d^3f}{dx^3}\right|_z \sim 0(\Delta x^2) \end{aligned} \tag{1.12}$$

In definitiva, tanto piú alta é l'accuratezza che si vuole ottenere, tanto piú grande é il numero di punti sulla griglia che bisogna considerare. Cosí come tanto piú alto é l'ordine della derivata che si vuole calcolare, tanto piú grande é il numero di punti sulla griglia da considerare.

1.2.1 Errori di arrotondamento e scelta del passo Δx opportuno

Accanto agli errori inerenti al metodo numerico usato, ci sono anche gli errori di arrotondamento che non possono essere trascurati. Nel caso delle derivate prime in avanti e indietro, se ϵ é il grado di accuratezza dei numeri, allora l'errore di arrotondamento al piú sará dato da $E_a \simeq 2\epsilon/\Delta x$, mentre nel caso delle derivate centrate sará $E_a \simeq \epsilon/\Delta x$. Allora é possibile in linea di principio minimizzare l'errore totale scegliendo un valore opportuno per Δx. Per esempio per le differenze in avanti oppure indietro l'errore totale sará $E_{Tot} \simeq \Delta x f^{(2)}/2 + 2\epsilon/\Delta x$ (dove $f^{(2)}$ indica una stima per la derivata seconda). Differenziando l'errore totale, rispetto ad una variazione di Δx si ottiene l'espressione $dE_{Tot} \simeq (f^{(2)}/2 - 2\epsilon/\Delta x^2)d\Delta x$ per cui l'errore totale si minimizza scegliendo un passo

$$\Delta x \simeq \left[\frac{4\epsilon}{f^{(2)}}\right]^{1/2}$$

Nel caso delle differenze centrali invece si ottiene $E_{Tot} \simeq \Delta x^2(f^{(2)}/6) - \epsilon/\Delta x$ per cui differenziando $dE_{Tot} \simeq (\Delta x f^{(3)}/3 - \epsilon/\Delta x^2)d\Delta x$, e quindi un minimo dell'errore si ottiene da

$$\Delta x \simeq \left[\frac{3\epsilon}{f^{(3)}}\right]^{1/3}$$

Infine per la derivata seconda l'errore é $E_{Tot} \simeq \Delta x^2(f^{(4)}/12) + 4\epsilon/\Delta x^2$ per cui $dE_{Tot} \simeq (\Delta x f^{(4)}/6 - 8\epsilon/\Delta x^3)d\Delta x$, e quindi l'errore totale é minimo quando vale la relazione

$$\Delta x \simeq \left[\frac{48\epsilon}{f^{(4)}}\right]^{1/4}$$

1.3 I punti di collocazione

Un metodo particolarmente efficace, anche se molto piú difficoltoso da implementare rispetto alla tecnica delle differenze finite per rappresentare la derivata di una funzione, consiste nel metodo dei punti di collocazione. In questo caso, invece di approssimare la derivata con un polinomio di ordine basso, la approssimiamo con un polinomio ortogonale di grado elevato. Questo consente di stimare la derivata di una funzione con una precisione molto alta.

Sia $f(x)$ una funzione definita in un intervallo $[a, b]$ e sia $\Phi_k(x)$ un polinomio ortogonale, rispetto al peso $p(x)$, di grado k che approssima la funzione

$$f(x) = \sum_{k=0}^{N} a_k \Phi_k(x) \tag{1.13}$$

dove

$$\int_a^b \Phi_m(x)\Phi_k^\star(x)p(x)dx = \lambda_k \delta_{m,k}$$

Allora moltiplicando la $f(x)$ per $\Phi_k^\star(x)$ ed integrando si ottiene

$$\int_a^b f(x)\Phi_k^\star(x)dx = a_k \int_a^b |\Phi_k(x)|^2 \, dx = \lambda_k a_k$$

e quindi é possibile calcolare i coefficienti dello sviluppo

$$a_k = \frac{1}{\lambda_k} \int_a^b f(x)\Phi_k^\star(x)dx \tag{1.14}$$

Sulla base di questo schema, possiamo sviluppare anche la funzione derivata m–esima tramite lo stesso polinomio ortogonale

$$\frac{d^m f}{dx^m} = \sum_{k=0}^{N} a_k^{(m)} \Phi_k(x) \tag{1.15}$$

e poiché lo sviluppo della derivata (1.15) ha lo stesso polinomio dello sviluppo (1.13), é possibile ottenere un legame fra i coefficienti dello sviluppo della derivata $a_k^{(m)}$ ed i coefficienti dello sviluppo della funzione a_k.

In altri termini, data la funzione $f(x)$ ed ottenuti i coefficienti a_k tramite una routine che calcola la trasformata della funzione, i coefficienti $a_k^{(m)}$ dello sviluppo della derivata si possono ottenere in funzione degli a_k. Infine tramite una trasformata inversa, dal set di coefficienti $a_k^{(m)}$ si ottiene l'espressione della derivata m–esima. Anche se tutto ció puó sembrare a prima vista complicato é invece abbastanza usuale, nel caso di calcoli numerici in cui si richiede una certa precisione nel calcolo delle derivate, ottenere una stima per le derivate usando questo metodo. Ovviamente tutto si basa sulla possibilitá di effettuare *rapidamente* le trasformate diretta ed inversa richieste dal metodo. La complicazione viene dal fatto che l'inversione $a_k^{(m)} = F(a_k)$ puó risultare abbastanza complicata per alcuni polinomi ortogonali. Diamo ora le formule relative ai tre polinomi piú usati.

1.3.1 Serie di Fourier

Se la funzione $f(x)$ é periodica, allora la coppia di trasformate di Fourier sará data da

$$\begin{aligned} f(x) &= \sum_{k=-\infty}^{\infty} a_k e^{ikx} \\ a_k &= \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-ikx}dx \end{aligned}$$

e quindi per i coefficienti dello sviluppo delle derivate prima e seconda si ottiene immediatamente

$$\begin{aligned} a_k^{(1)} &= ika_k \\ a_k^{(2)} &= -k^2 a_k \end{aligned} \tag{1.16}$$

In questo caso entrambi i coefficienti di ordine k delle derivate sono legati al coefficiente di ordine k della funzione in modo molto semplice.

1.3.2 Serie di Legendre

Una funzione $f(x)$ puó essere sviluppata in serie di polinomi di Legendre di grado k. La coppia di trasformate sará

$$\begin{aligned} f(x) &= \sum_{k=-\infty}^{\infty} a_k P_k(x) \\ a_k &= \frac{2k+1}{2}\int_{-1}^{1} f(x)P_k(x)dx \end{aligned}$$

mentre il polinomio di Legendre é dato da

$$P_k(\cos\vartheta) = \sqrt{\frac{2}{k\pi\sin\vartheta}}\,[\sin(k+1/2)\vartheta + \pi/4]$$

Dalla relazione di ricorrenza per i polinomi di Legendre

$$(k+1)P_{k+1}(x) = (2k+1)xP_k(x)kP_{k-1}(x)$$

si ottengono i coefficienti della derivata prima:

$$a_k^{(1)} = (2k+1) \sum_{p=k+1}^{\infty} a_p \tag{1.17}$$

dove la somma é estesa a tutti i valori tali che $p+k$ sia un numero dispari, e quelli della derivata seconda

$$a_k^{(2)} = \left(k + \frac{1}{2}\right) \sum_{p=k+2}^{\infty} \left[p(p+1) - k(k+1)\right] a_p \tag{1.18}$$

dove la somma é estesa a tutti i valori tali che $p+k$ sia un numero pari. A differenza delle serie di Fourier, in questo caso i coefficienti dello sviluppo delle derivate non sono legati in modo semplice ai coefficienti dello sviluppo della funzione.

1.3.3 Serie di Chebyshev

Una funzione $f(x)$ puó essere sviluppata in serie di polinomi di Chebyshev di grado k. La coppia di trasformate é data da

$$\begin{aligned} f(x) &= \sum_{-\infty}^{\infty} a_k T_k(x) \\ a_k &= \frac{2}{\pi c_{k-1}} \int_{-1}^{1} \frac{f(x) T_k(x)}{\sqrt{1-x^2}} dx \end{aligned}$$

ed il polinomio di Chebyshev sará

$$T_k(\cos\vartheta) = \cos(k\vartheta)$$

Dalla relazione di ricorrenza per i polinomi di Chebyshev

$$T_{k+1}(x) = 2kT_k(x)T_{k-1}(x)$$

e definendo i parametri $c_0 = 0$ e $c_n = 1$ per $n > 1$, si ottengono i coefficienti della derivata prima:

$$a_k^{(1)} = \frac{2}{c_k} \sum_{p=k+1}^{\infty} p a_p \tag{1.19}$$

dove la somma é estesa a tutti i valori tali che $p+k$ sia un numero dispari, e quelli della derivata seconda

$$a_k^{(2)} = \frac{1}{c_k} \sum_{p=k+2}^{\infty} p\left(p^2 - k^2\right) a_p \tag{1.20}$$

dove la somma é estesa a tutti i valori tali che $p + k$ sia un numero pari. Anche in questo caso, a differenza delle serie di Fourier, i coefficienti dello sviluppo delle derivate non sono legati in modo semplice ai coefficienti dello sviluppo della funzione.

Da questi esempi abbiamo visto che le derivate espresse come combinazioni di polinomi ortogonali possono essere calcolate con una accuratezza praticamente infinita rispetto all'accuratezza ottenuta usando le differenze finite. Tuttavia l'implementazione numerica di questa tecnica é piuttosto complicata. In pratica, mentre nel caso delle differenze finite il calcolo della derivata si riduce poco piú che a dieci righe di un qualsiasi programma, nel caso dello sviluppo in polinomi bisogna assolutamente avere una efficiente routine numerica che permette di calcolare coefficienti dello sviluppo. Il calcolo quindi si compone di tre differenti fasi, come schematicamente descritto di seguito:

1. Tramite una trasformata diretta si calcolano i coefficienti dello sviluppo della funzione: $f(x_i) \rightarrow a_k$.

2. Si calcolano i coefficienti della trasformata delle derivate tramite l'inversione della relazione $a_k^{(m)} = F(a_k)$.

3. Tramite la trasformata inversa si calcola la derivata della funzione: $a_k^{(m)} \rightarrow d^m f(x_i)/dx^m$.

In genere le routine di trasformate veloci (Fast Fourier Transform) permettono di calcolare la funzione, e quindi le derivate, solo su determinati punti spaziali, detti punti di collocazione x_i. A complicare ulteriormente la cosa c'é il fatto che, tranne che nel caso dei coefficienti di Fourier, lo sforzo computazionale per calcolare i coefficienti dello sviluppo delle derivate a partire dai coefficienti dello sviluppo della funzione (fase 2.) non é banale. Come sempre nel calcolo numerico, un aumento di accuratezza é sempre accompagnato da un aumento dello sforzo computazionale.

Capitolo 2

Equazioni differenziali ordinarie

2.1 Introduzione

Il problema che ci poniamo in questa sezione, che é anche il problema centrale del libro, riguarda l'integrazione numerica delle equazioni differenziali alle derivate ordinarie. Cominciamo con la definizione di un problema di Cauchy. Sia $y(t)$ una funzione della variabile indipendente $t \in [t_0, \infty)$, che assume un valore noto y_0 nel punto t_0, e sia $F(t, y)$ una funzione qualsiasi[1]. Allora il sistema

$$\begin{cases} \dot{y} = F(t, y) \\ y(t_0) = y_0 \end{cases}$$

dove $\dot{y} = dy/dt$, rappresenta un problema di Cauchy. La soluzione di questo sistema, come é noto dall'analisi, dipende dall'integrale della funzione $F(t, y)$, per cui non é sempre possibile ottenere una soluzione analitica $y(t)$. La soluzione numerica del sistema (2.1) coinvolge una tecnica di approssimazione delle derivate su una griglia data, ed una stima della funzione $F(t, y)$ sui punti della griglia. A seconda delle differenti approssimazioni si parlerá allora di differenti schemi di risoluzione numerica della (2.1).

2.2 Le differenze finite

Sia data allora la funzione $y(t_i) \equiv y_i$, definita su un insieme di punti di una griglia $\{t_i\}$ ordinati, e sia Δt il passo costante della griglia. Allora il modo piú

[1] Useremo la variabiile temporale come variabile indipendente in quanto, lo scopo didattico del libro, é quello di presentare metodi numerici di risoluzione delle equazioni differenziali, che siano utilizzabili immediatamente per risolvere problemi di dinamica classica che non si possono risolvere in maniera analitica.

comodo per risolvere l'equazione (2.1) é quello di approssimare la derivata tramite una delle formule alle differenze finite. Si parlerá allora di schemi alle differenze finite per la risoluzione.

Definizione: Dato il problema di Cauchy (2.1), e data una approssimazione alle differenze finite per la derivata, ossia definito un polinomio di interpolazione $L_n(t_i)$, si definisce errore di troncamento la quantitá $E = |L_n(t_i) - F(t_i, y_i)|$, mentre l'ordine di accuratezza p di uno schema numerico resta definito tramite $E(\Delta t) \sim 0(\Delta t^{p+1})$. Inoltre uno schema si dirá consistente se

$$\lim_{\Delta t \to 0} |L_n(t_i) - F(t_i, y_i)| = 0$$

Il problema principale che si incontra nella risoluzione numerica delle equazioni differenziali é il problema delle instabilitá numeriche che nascono nel corso del calcolo e che sono dovute alla matematica discreta, quindi allo schema numerico, che si usa per risolvere numericamente l'equazione. In molti casi questo é dovuto ad un passo Δt troppo grande, ma in altri casi le instabilitá nascono anche con un passo piccolissimo, e sono inerenti allo schema usato.

Consideriamo allora l'approssimazione (1.2) a tre punti della derivata, e sostituiamola nella (2.1), usando una equazione lineare $F(t, y) = cy$ con $c = cost$. Si ottiene una equazione alle differenze

$$(1 - \xi)y_{n+1} + 2\xi y_n - (1 + \xi)y_{n-1} - 2c\Delta t y_n = 0 \tag{2.1}$$

Consideriamo una soluzione di questa equazione nella forma $y_n = Y_n \exp(ikt_n)$, per cui sostituendo nella (2.1), si ottiene l'equazione alle differenze

$$G_1 Y_{n+1} + G_0 Y_n + G_{-1} Y_{n-1} = 0 \tag{2.2}$$

con opportune definizioni dei nuovi coefficienti G_i. L'equazione (2.2) puó essere scritta in forma matriciale $\Xi_{n+1} = G\Xi_n$, definendo una nuova variabile $V_{n+1} = Y_n$ ed il vettore $\Xi_n = (Y_n, V_n)$. La matrice G, definita da:

$$G = \begin{bmatrix} \frac{2(c\Delta t - \xi)}{(1-\xi)} & \frac{1+\xi}{1-\xi} \\ 1 & 0 \end{bmatrix} \tag{2.3}$$

é detta matrice di amplificazione. Si puó allora dimostrare il seguente teorema:

Teorema di Von Neumann: Lo schema (2.1) é stabile se la matrice di amplificazione G ha un raggio spettrale non superiore all'unitá.

In altri termini, siano p_1 e p_2 gli autovalori della matrice di amplificazione G, ossia le due radici dellequazione $|G-pI| = 0$ (dove I é la matrice identitá). Usando la (2.3) si vede che i due autovalori si ottengono dalla equazione

$$(1-\xi)p^2 - (c\Delta t - \xi)p - (1+\xi) = 0$$

da cui immediatamente si ottengono i due valori

$$(1-\xi)p_{1,2} = (c\Delta t - \xi) \pm \sqrt{(c\Delta t - \xi)^2 + (1-\xi^2)} \tag{2.4}$$

In accordo con il Teorema di Von Neuman, lo schema é stabile se

$$\max\left[|p_1|, |p_2|\right] \leq 1 \tag{2.5}$$

Come é facile intuire osservando la (2.4), fissato il valore di ξ per definire la derivata, la relazione (2.5) lega il passo della griglia Δt ai parametri fisici dell'equazione da integrare (nel nostro semplice caso alla costante c). Allora se la disequazione (2.5) é verificata qualsiasi sia il valore di Δt, lo schema si dirá *incondizionatamente stabile*, se la (2.5) non é mai verificata per nessun valore di Δt lo schema si dirá *incondizionatamente instabile*. Il caso piú comune é che dalla (2.5) si ottenga una maggiorazione per Δt, ossia una relazione $\Delta t \leq g$ (con g fissato dai parametri fisici). In questo caso lo schema sará *condizionatamente stabile*, ossia stabile a condizione di usare passi della griglia minori di un certo valore.

2.2.1 L'equazione equivalente

L'origine fisica delle instabilitá risiede nel fatto che stiamo approssimando la soluzione di un problema analitico, che richiederebbe quindi una matematica infinitesimale, con una approssimazione ottenuta usando una matematica discreta. Infatti, come abbiamo ripetuto piú volte, quando stimiamo una derivata con le differenze finite, e quindi quando risolviamo numericamente una equazione differenziale, commettiamo un errore $E(\Delta t)$. Si puó dimostrare che se Δt é finito, la soluzione numerica della equazione (2.1) é piú simile all'equazione differenziale

$$\frac{dy}{dt} \simeq F(t,y) + E(\Delta t)$$

che chiameremo *equazione equivalente*, che all'equazione originaria (2.1). In altri termini la presenza della matematica discreta rende l'equazione da risolvere una equazione di grado superiore, perché nel termine di errore ci sono

derivate di ordine superiore. In questo caso allora é facile convincersi dell'esistenza di soluzioni spurie che non sono fisiche, ma sono dovute solamente alla presenza del termine finito $E(\Delta t)$.

2.3 Lo schema di Eulero

Questo é lo schema principale di risoluzione di una equazione differenziale, per cui si puó usare come esempio di tutti gli schemi piú complicati. Lo schema di Eulero deriva dall'usare la definizione (1.7) per le derivate, e dall'approssimare il termine di destra della (2.1) sui punti della griglia. Sostituendo la derivata nella (2.1) si ottiene

$$y_{n+1} = y_n + \Delta t F(t_n, y_n) \tag{2.6}$$

Lo schema (2.6) ha una interpretazione semplice, infatti esso corrisponde ad approssimare la funzione $F(t, y) \simeq const.$ in ogni sotto–intervallo infinitesimo Δt. Infatti se integriamo l'equazione (2.1) nell'intervallo $[t, t+\Delta t]$, con $F(t, y)$ costante, si ottiene

$$y_{n+1} - y_n = \int_{t_n}^{t_{n+1}} F(t, y) dt \simeq \Delta t F(t_n, y_n)$$

Questo schema é accurato fino all'ordine $0(\Delta t)$, come si vede subito sostituendo una espansione in serie di Taylor di y_{n+1} nell'intorno di y_n, da cui si ottiene

$$\left.\frac{dy}{dt}\right|_{t_n} - F(t_n, y_n) \simeq -\frac{\Delta t}{2} \left.\frac{d^2 y}{dt^2}\right|_{t_n} \tag{2.7}$$

Come si vede lo schema é consistente perché passando al continuo, ossia nel limite $\Delta t \to 0$, l'errore dello schema si annulla. Per una funzione lineare $F(t, y) = cy$ si ottiene $y_{n+1} = (1 + c\Delta t) y_n$, per cui in questo caso si parlerá di fattore di amplificazione $G = (1 + c\Delta t)$ piuttosto che di matrice di amplificazione. Ovviamente lo schema é stabile solo se il fattore di amplificazione soddisfa alla disequazione $|1 + c\Delta t| \leq 1$. La soluzione di questo é che, se $c > 0$ lo schema é sempre instabile, mentre se $c < 0$ esiste una relazione di stabilitá a cui deve soddisfare Δt che é $\Delta t \leq 2/|c|$. Se $|c|$ é molto grande la relazione di stabilitá puó portare a dovere considerare passi di integrazione molto piccoli.

Esempio: Studiare la soluzione del seguente problema di Cauchy

Tabella 2.1: Riportiamo i valori ottenuti dalla soluzione del problema di Cauchy $\dot{y} = ty^{1/3}$ con $y(1) = 1$, con uno schema di Eulero usando due differenti valori del passo Δt, ossia $\Delta t = 0.1$ e $\Delta t = 0.01$, ed il valore esatto $y(t_n)$. Riportiamo inoltre gli errori, rispetto al valore esatto, ottenuti tramite le soluzione numeriche con i due passi.

t_n	$\Delta t = 0.1$	$\Delta t = 0.01$	Soluzione esatta	Errore con $\Delta t = 0.01$	Errore con $\Delta t = 0.1$
1.0	1.000	1.000	1.000	0.000	0.000
2.0	2.72	2.82	2.8884	0.01	0.11
3.0	6.71	6.99	7.0211	0.03	0.31
4.0	14.08	14.63	14.6969	0.07	0.62
5.0	25.96	26.89	27.0000	0.11	1.04

$$\dot{y} = ty^{1/3} \;\; ; \quad y(1) = 1$$

L'equazione é a variabili separabili, per cui si integra facilmente in modo analitico. Lo schema di Eulero applicato alla equazione differenziale risulta invece essere

$$y_{n+1} = y_n + \Delta t \left[t_n y_n^{1/3} \right]$$

che stato risolto usando due differenti valori del passo Δt. I risultati, assieme al valore di $y(t)$ calcolato in modo esatto, sono riportati nella figura. Per maggiore comoditá nella Tabella 2.1 sono riportati i valori ottenuti con i due passi, la soluzione esatta e l'errore ottenuto nei due casi:

Come si vede al decrescere di Δt la soluzione numerica tende verso la soluzione esatta, nel senso che la differenza fra la soluzione numerica e quella esatta (ultime due colonne) decresce.

Si potrebbe pensare che il fatto che per avere uno schema stabile deve essere $c < 0$, dal momento che una equazione lineare con $c > 0$ ha come soluzione una funzione esponenziale che tende ad infinito, per cui qualsiasi schema sar instabile per principio. In realtá il problema non é cosí ovvio e risiede nella equazione equivalente della (2.6) che la (2.7). Come dicevamo in precedenza, l'equazione equivalente é di ordine superiore, in questo caso di secondo ordine, e questo fa si che esiste una soluzione spuria rispetto a quella reale. Nel caso in cui l'equazione é lineare $F(t, y) = cy$, l'equazione equivalente risulta essere a coefficienti costanti, ossia:

$$\frac{d^2y}{dt^2} + \left(\frac{2}{\Delta t}\right)\frac{dy}{dt} - \left(\frac{2c}{\Delta t}\right)y = 0$$

la cui soluzione é $y(t) = A\exp(r_1 t) + B\exp(r_2 t)$, dove A e B sono costanti, e dove r_i sono le due soluzioni dell'equazione caratteristica associata $r^2 + (2/\Delta t)r(2c/\Delta t) = 0$. Le due soluzioni, per $c > 0$, sono quindi

$$\begin{aligned} r_1 &= -\frac{1}{\Delta t}\left[1 + \sqrt{1 + 2c\Delta t}\right] < 0 \\ r_2 &= -\frac{1}{\Delta t}\left[1 - \sqrt{1 + 2c\Delta t}\right] > 0 \end{aligned}$$

per cui nel secondo caso esiste una soluzione r_2 che é il tasso di crescita della soluzione esponenziale instabile.

2.4 Lo schema di Runge–Kutta

Questo schema consiste in una modifica sofisticata del metodo di Eulero, perché come vedremo, il passaggio dalla soluzione al passo t_n a quello al passo t_{n+1} é fatto usando una media pesata dei gradienti ad un set di punti (t_i, y_i) nelle vicinanze di (t_n, y_n). L'idea dello schema é quella di esprimere y_{n+1} come combinazione lineare della forma

$$y_{n+1} = y_n + aK_1(t_n, y_n) + bK_2(t_n, y_n) + E(\Delta t) \tag{2.8}$$

dove le funzioni incognite sono delle medie pesate

$$\begin{aligned} K_1 &= \Delta t F(t_n, y_n) \\ K_2 &= \Delta t F(t_n + \vartheta\Delta t, y_n + \varphi K_1) \end{aligned}$$

ed i parametri a, b, ϑ e φ sono calcolati in modo che l'accuratezza dello schema sia del secondo ordine $E(\Delta t) \sim 0(\Delta t^3)$. Usando una espansione in serie di Taylor

$$K_2 \simeq \Delta t\left[F(t_n, y_n) + \vartheta\Delta t\left.\frac{\partial F}{\partial t}\right|_{t_n,y_n} + \varphi K_1\left.\frac{\partial F}{\partial y}\right|_{t_n,y_n}\right] + 0(\Delta t^3)$$

nell'equazione (2.8) si ottiene

$$y_n + \Delta t F(t_n, y_n) \quad + \quad \frac{\Delta t^2}{2} \left.\frac{\partial F}{\partial t}\right|_{t_n} + \frac{\Delta t^2}{2} \left.\frac{\partial F}{\partial y}\right|_{y_n} = y_n + a\Delta t F(t_n, y_n) +$$
$$+ \quad b\Delta t F(t_n, y_n) + b\vartheta \Delta t^2 \left.\frac{\partial F}{\partial t}\right|_{t_n} + b\varphi K_1 \Delta t \left.\frac{\partial F}{\partial y}\right|_{y_n}$$

per cui i quattro parametri liberi devono soddisfare alle tre equazioni seguenti: $a + b = 1$; $b\vartheta = 1/2$ e quindi $b\varphi = 1/2$. Ovviamente un parametro deve restare libero, scegliendo ϑ si ottiene lequazione finale

$$y_{n+1} = y_n + \left(1 - \frac{1}{2\vartheta}\right) \Delta t F_n + \frac{\Delta t}{2\vartheta} F(t_n + \vartheta \Delta t, y_n + \vartheta \Delta t F_n) \tag{2.9}$$

dove si é definito $F_n = F(t_n, y_n)$. Differenti valori di ϑ danno origine a schemi differenti. Di particolare interesse é il semplice schema di Eulero modificato dove $\vartheta = 1/2$, che risulta essere molto usato

$$y_{n+1} = y_n + \Delta t F(t_n + \Delta t/2, y_n + F_n \Delta t/2) \tag{2.10}$$

mentre se $\vartheta = 1$ lo schema si dice di Henn.

Ovviamente é possibile costruire schemi di Runge–Kutta di ordine piú elevato, basta aggiungere altri termini all'equazione (2.8) ed usare la tecnica degli sviluppi in serie di Taylor. Lo svantaggio principale dello schema di Runge–Kutta rispetto a quello di Eulero, é costituito dal fatto che nel primo caso bisogna calcolare la funzione $F(t, y)$ un certo numero di volte per ogni step Δt. Se $F(t, y)$ é una funzione complicata questo calcolo porta un evidente rallentamento di tempo di calcolo numerico. Il vantaggio ovvio consiste in un aumento della precisione del calcolo.

Tutti gli schemi di Runge–Kutta sono convergenti, nel senso che l'errore $E(\Delta t) \to 0$ nel limite $\Delta t \to 0$, tuttavia anche lo schema di Runge–Kutta é soggetto ad instabilitá se si usano valori troppo elevati di Δt. Consideriamo per esempio l'equazione lineare $\dot{y} = cy$, con $y(0) = 1$, che ha soluzione $y(t) = \exp(ct)$. L'equazione (2.10) posto $F = cy$, dá come risultato

$$y_{n+1} = \left(1 + c\Delta t + \frac{c^2 \Delta t^2}{2}\right) y_n \tag{2.11}$$

Allora lo schema é stabile solo se $|1 + c\Delta t + c^2 \Delta t^2/2| \leq 1$, per cui se $c > 0$ lo schema é sempre instabile, mentre se $c < 0$ lo schema é stabile solo se $\Delta t \leq 2/|c|$. L'equazione (2.11) puó essere risolta iterativamente, per cui la soluzione che soddisfa a $y(0) = 1$, é data da

$$y_n = \left(1 + c\Delta t + \frac{c^2\Delta t^2}{2}\right)^n = \left(1 + c\Delta t + \frac{c^2\Delta t^2}{2}\right)^{t_n/\Delta t}$$

da cui si ottiene

$$\log y_n = \left(\frac{t_n}{\Delta t}\right) \log\left(1 + c\Delta t + \frac{c^2\Delta t^2}{2}\right)$$

Si vede quindi immediatamente che nel limite $\Delta t \to 0$ si ottiene $\log y_n = ct_n$, ossia lo schema é convergente verso la soluzione esatta.

2.5 Schemi di tipo multi–step

I due tipi di schemi che abbiamo visto finora sono di tipo one–step, ossia la soluzione al passo temporale $(n+1)$–esimo é ottenuta in funzione soltanto della soluzione al passo n–esimo. Esistono tuttavia schemi di tipo multi–step, in cui la soluzione al passo $(n+1)$–esimo é ottenuta in funzione di un certo numero di passi precedenti n–esimo, $(n-1)$–esimo, etc. In questo caso ovviamente gli schemi sono complicati dal punto di partenza, ossia diventa piú difficile innescare lo schema.

2.5.1 Lo schema Leap–Frog

Lo schema multi–step piú semplice é il Leap–Frog, che come al solito consiste in una modifica dello schema di Eulero. Questo schema infatti consiste nello stimare la derivata con le differenze centrali invece che con le derivate di tipo forward, per cui

$$y_{n+1} = y_{n-1} + 2\Delta t F(t_n, y_n) \tag{2.12}$$

per cui la precisione di questo schema é del primo ordine $E \sim 0(\Delta t^2)$. Infatti la solita espansione in serie di Taylor dá come risultato l'equazione equivalente

$$\frac{dy}{dt} - F(t,y) \simeq -\frac{\Delta t^2}{2}\frac{d^3y}{dt^3} \tag{2.13}$$

Come si vede in questo caso la derivata di ordine superiore dell'errore é del terzo ordine. Nel caso della solita equazione lineare $F(t,y) = cy$, la matrice di amplificazione dello schema é data da

$$G = \begin{bmatrix} 2c\Delta t & 1 \\ 1 & 0 \end{bmatrix} \tag{2.14}$$

per cui gli autovalori si ottengono dalla equazione $p^2 - 2c\Delta t p 1 = 0$. Nel caso in cui $c > 0$, il piú grande autovalore $p = c\Delta t + (\sqrt{1 + c^2\Delta t^2})$ é maggiore di 1 e quindi lo schema é incondizionatamente instabile. Nel caso in cui $c < 0$, si ha che contemporaneamente per i due segni, deve valere la relazione $| - |c|\Delta t \pm \sqrt{(1 + c^2\Delta t^2)}| \leq 1$. Allora nel caso del segno positivo devono valere le due disequazioni seguenti

$$\begin{aligned} -1 &< -|c|\Delta t + \sqrt{1 + c^2\Delta t^2} \\ 1 &> -|c|\Delta t + \sqrt{1 + c^2\Delta t^2} \end{aligned}$$

che sono sempre soddisfatte per $|c| > 0$. Nel caso del segno negativo le disequazioni sono

$$(1 - |c|\Delta t) > \sqrt{1 + c^2\Delta t^2}$$

che non é mai soddisfatta se $|c| > 0$, e

$$-(1 - |c|\Delta t) < \sqrt{1 + c^2\Delta t^2}$$

che é sempre soddisfatta. Come si vede facilmente l'intersezione fra le soluzioni é nulla, e quindi lo schema Leap–Frog é incondizionatamente instabile.

2.5.2 Lo schema Adams–Bashforth

Questo schema multi–step consiste nell'effettuare una media pesata della funzione $F(t, y)$ su due passi temporali:

$$y_{n+1} = y_n + \frac{\Delta t}{2}\left[3F(t_n, y_n) - F(t_{n-1}, y_{n-1}\right] \tag{2.15}$$

e la solita espansione in serie di Taylor mostra che l'equazione equivalente é data dalla espressione

$$\frac{dy}{dt} - F(t, y) \simeq -\frac{\Delta t^2}{12}\frac{d^3y}{dt^3} - \frac{\Delta t^2}{4}\frac{\partial^2 F}{\partial t^2} \tag{2.16}$$

per cui anche questo schema é preciso al primo ordine. Usando una equazione lineare l'equazione agli autovalori é $p^2(1 + 3/2c\Delta t)p + c\Delta t/2 = 0$, per cui si vede che se $c < 0$, esiste un intervallo di stabilitá dello schema se $\Delta t < 1/|c|$.

E' interessante confrontare lo schema Adams–Bashforth con lo schema Leap–Frog. Nel caso di una equazione lineare $F = cy$, l'equazione equivalente (2.13) dello schema Leap–Frog é una equazione del tipo

$$\frac{\Delta t^2}{12}\frac{d^3y}{dt^3} + \frac{dy}{dt} - cy = 0$$

mentre per lo schema Adams–Bashforth l'equazione equivalente é

$$\frac{\Delta t^2}{12}\frac{d^3y}{dt^3} + \frac{\Delta t^2 c}{4}\frac{d^2y}{dt^2} + \frac{dy}{dt} - cy = 0$$

Si vede che, anche se l'errore é proporzionale a Δt^2, nel primo caso manca il termine di derivata seconda che invece esiste nel secondo caso. Questa é la differenza della stabilitá dello schema Adams–Bashforth rispetto allo schema Leap–Frog.

2.5.3 Lo schema predictor–corrector

Questo schema é molto usato rispetto ai due schemi precedenti. Lo schema predictor–corrector consiste nell'effettuare due passi distinti con differenti Δt. Il primo passo é fatto in un punto virtuale t_{n+a}, dove $0 < a < 1$, e consiste nel calcolare un valore di predizione tramite uno step con uno schema di Eulero. Questo passo viene poi corretto con un passo ulteriore fino al tempo t_{n+1}. I due passi che costituiscono lo schema sono quindi dati da:

$$\begin{aligned} y_{n+a} &= y_n + a\Delta t F(t_n, y_n) \quad \text{step predictor} \\ y_{n+1} &= y_n + \Delta t F(t_{n+a}, y_{n+a}) \quad \text{step corrector} \end{aligned} \tag{2.17}$$

Da notare che la caratteristica dello schema é il termine y_n a secondo membro dello step corrector. Infatti se ci fosse y_{n+a} al posto di y_n, lo schema si ridurrebbe a due passi semplici di Eulero. Invece il risultato dello schema a due step predictor–corrector é differente. Usando una equazione lineare $F = cy$ si ottiene l'errore dello schema, che risulta essere

$$E(\Delta t) \simeq -\frac{\Delta t^2}{6}\frac{d^2}{dt^2}\left[\frac{dy}{dt} - 3a^2 cy\right] \tag{2.18}$$

Come si vede lo schema predictor–corrector ha una precisione maggiore dello schema di Eulero. Usando l'equazione lineare lo schema si riscrive come one–step

$$y_{n+1} = (1 + c\Delta t + ac^2\Delta t^2)y_n$$

e si vede immediatamente che lo schema risulta stabile se $c < 0$ e $\Delta t < 1/a|c|$. Nel caso piú comune in cui si sceglie $a = 1/2$ il criterio di stabilitá é lo stesso degli schemi one–step.

Tabella 2.2: Riportiamo i valori ottenuti dalla soluzione del problema di Cauchy $\dot{y} = 2y$ con $y(0) = 1$, con uno schema di Eulero usando tre differenti valori del passo Δt, ossia $\Delta t = 0.1$, $\Delta t = 0.05$ e $\Delta t = 0.01$, ed il valore $Y(t_n)$ che si ottiene tramite l'estrapolazione di Richardson. Per confronto riportiamo anche il valore della soluzione esatta $y(t_n) = \exp(2t_n)$.

t_n	$\Delta t = 0.1$	$\Delta t = 0.05$	$\Delta t = 0.01$	$Y(t_n)$	$\exp(2t_n)$
0.0	1.000	1.000	1.000	1.000	1.000
0.2	1.440	1.464	1.486	1.492	1.492
0.4	2.074	2.144	2.208	2.225	2.226
0.6	2.986	3.138	3.281	3.320	3.320
0.8	4.300	4.595	4.876	4.953	4.953
1.0	6.192	6.727	7.245	7.387	7.389

2.6 L'estrapolazione di Richardson

Per aumentare la precisione di uno schema si puó ricorrere alla cosiddetta estrapolazione di Richardson. Questa tecnica si basa sul fatto che, per uno schema stabile, la diminuzione del passo porta ad un risultato convergente punto per punto al valore esatto. Per aumentare la precisione allora si calcola uno step di integrazione con tre passi differenti, e quindi si estrapola il risultato al valore esatto, che si ottiene nel limite in cui $\Delta t \to 0$. Per esempio sia $y_{n+1}(\Delta t)$ il risultato ottenuto al passo $(n+1)$–esimo con uno passo Δt ed uno schema qualsiasi. Calcoliamo questo stesso risultato suddividendo il passo Δt per due volte, per esempio calcoliamo $y_{n+1}(\Delta t/2)$, usando un passo $\Delta t/2$, ed $y_{n+1}(\Delta t/4)$ usando un passo $\Delta t/4$. Allora se lo schema é stabile si puó estrapolare un valore esatto Y_{n+1} all'istante t_{n+1}, ossia il risultato che si otterrebbe nel limite $\Delta t \to 0$, tramite la soluzione del seguente di tre equazioni

$$\begin{cases} y_{n+1}(\Delta t) = Y_{n+1} + A\Delta t + B\Delta t^2 \\ y_{n+1}\left(\frac{\Delta t}{2}\right) = Y_{n+1} + A\frac{\Delta t}{2} + B\left(\frac{\Delta t^2}{2}\right)^2 \\ y_{n+1}\left(\frac{\Delta t}{4}\right) = Y_{n+1} + A\frac{\Delta t}{4} + B\left(\frac{\Delta t^2}{4}\right)^2 \end{cases} \tag{2.19}$$

(A e B sono coefficienti).

La soluzione $y(t) = \exp(2t)$ della equazione $y' = 2y$ (con valore iniziale $y(0) = 1$), per tre valori di Δt é riportata nella tabella 2.2. E' riportato anche il valore Y_{n+1} ottenuto tramite l'estrapolazione di Richardson.

2.7 Schemi impliciti

Abbiamo visto finora che la difficoltá principale nel calcolo computazionale delle equazioni differenziali é rappresentata dalle instabilitá dei vari schemi che, quando sono eliminabili, richiedono che il passo di integrazione Δt sia minore di una certa quantitá che coinvolge i parametri fisici dell'equazione. Per ovviare a questo inconveniente é consigliabile usare gli schemi impliciti che, almeno in alcuni casi, riescono a migliorare il problema, ossia fanno in modo che il passo da usare possa essere piú grande.

In generale, data l'equazione (2.1) $\dot{y} = F(t, y)$, uno schema implicito consiste nello stimare il secondo membro $F(t, y)$ nel punto t_{n+1} che si vuole determinare, o, piú in generale, si stima la funzione $F(t, y)$ come una media pesata fra i punti t_n e $_{tn+1}$. Questo é il significato dello schema, ossia la soluzione y_{n+1}, al passo t_{n+1}, si calcola implicitamente in funzione della soluzione allo stesso passo. Data quindi l'equazione, lo schema implicito in generale si scrive nel modo seguente:

$$y_{n+1} = y_n + \Delta t \left[\phi F(t_{n+1}, y_{n+1}) + (1 - \phi) F(t_n, y_n)\right] \tag{2.20}$$

dove $0 \leq \phi \leq 1$ é un parametro libero. Se $\phi = 1$ lo schema é puramente implicito, mentre se $\phi = 0$ lo schema diventa un semplice schema di Eulero visto in precedenza. Se si usa $\phi = 1/2$ lo schema si dice di Crank–Nicolson.

Per vedere i vantaggi di uno schema implicito consideriamo l'equazione lineare $F(t, y) = cy$ ed usiamo uno schema implicito

$$y_{n+1} = y_n + c\Delta t \left[\phi y_{n+1} + (1 - \phi) y_n\right]$$

che si riscrive nel modo seguente

$$y_{n+1} = \left[\frac{1 + c(1 - \phi)\Delta t}{1 - c\phi\Delta t}\right] y_n \tag{2.21}$$

con un fattore di amplificazione dato da $G = [1+c(1-\phi)\Delta t]/[1-c\phi\Delta t]$. Come si vede, lo schema é incondizionatamente stabile, ossia $|G| \leq 1$ per ogni valore di Δt, se $\phi > 0$ e $c < 0$. Questo risultato é abbastanza stupefacente se si pensa che lo schema di Eulero, che non differisce molto dalla (refimplicito1), é stabile solo se il passo soddisfa alla condizione $\Delta t \leq 2/|c|$. Ovviamente la maggiore stabilitá, da un punto di vista puramente aritmetico, consiste nel fatto che l'espressione per G contiene un termine al denominatore che é sicuramente minore di 1. Quindi il fattore di amplificazione G é diminuito in questo caso implicito, e piú facilmente soddisfa alla condizione di stabilitá. Se inoltre sviluppiamo come al solito y_{n+1} in serie di Taylor attorno ad y_n e sostituiamo lo sviluppo nella (2.21), otteniamo l'espressione seguente

$$\left(\frac{dy}{dt} - cy\right) \simeq -\frac{\Delta t}{2}\frac{d}{dt}\left(\frac{dy}{dt} - 2c\phi y\right) - \frac{\Delta t^2}{6}\frac{d^2}{dt^2}\left(\frac{dy}{dt} - 3c\phi y\right) \qquad (2.22)$$

Se quindi si usa uno schema di Crank–Nicolson con $\phi = 1/2$, il primo termine dell'errore scompare identicamente perché per definizione $\dot{y} = cy$, e l'errore dello schema é dato da

$$E(\Delta t) \simeq \frac{5c\Delta t^2}{12}\frac{d^2y}{dt^2}$$

Questo significa che gli schemi impliciti di Crank–Nicolson sono precisi fino all'ordine $O(\Delta t^2)$.

Gli schemi impliciti sono particolarmente complicati da applicare nel caso in cui si abbia un sistema di N equazioni differenziali lineari, ossia una espressione del tipo

$$\frac{dy_i}{dt} = \sum_j c_{ij} y_j \qquad (2.23)$$

con $i = 1, 2, \ldots, N$, e dove c_{ij} sono coefficienti numerici. In questo caso l'applicazione di uno schema implicito porta alla soluzione del sistema seguente di equazioni

$$\left[I - \Delta t\phi \sum_j c_{ij}\right] y_i^{(n+1)} = \left[I + \Delta t(1-\phi)\sum_j c_{ij}\right] y_i^{(n)} \qquad (2.24)$$

(I é la matrice identitá), che in genere non é semplice da risolvere. In pratica si tratta di invertire la matrice a primo membro, e quindi ottenere la soluzione

$$y_i^{(n+1)} = \left[I - \Delta t\phi \sum_j c_{ij}\right]^{-1} \left[I + \Delta t(1-\phi)\sum_j c_{ij}\right] y_i^{(n)}$$

ma questo implica uno sforzo computazionale notevole.

Esempio. Risolvere con uno schema implicito il seguente sistema di equazioni: $\dot{y} = ay + bz$, $\dot{z} = cy + dz$, dove a, b, c, d sono costanti. Lo schema implicito si traduce nelle due espressioni seguenti:

$$\begin{aligned} y_{n+1} &= y_n + a\Delta t[\phi y_{n+1} + (1-\phi)y_n] + b\Delta t[\phi z_{n+1} + (1-\phi)z_n] \\ z_{n+1} &= z_n + c\Delta t[\phi y_{n+1} + (1-\phi)y_n] + d\Delta t[\phi z_{n+1} + (1-\phi)z_n] \end{aligned}$$

ovvero in forma matriciale

$$\begin{pmatrix} 1-a\phi\Delta t & -b\phi\Delta t \\ -c\phi\Delta t & 1-d\phi\Delta t \end{pmatrix} \begin{pmatrix} y_{n+1} \\ z_{n+1} \end{pmatrix} =$$
$$= \begin{pmatrix} 1+a(1-\phi)\Delta t & b(1-\phi)\Delta t \\ c(1-\phi)\Delta t & 1+d(1-\phi)\Delta t \end{pmatrix} \begin{pmatrix} y_n \\ z_n \end{pmatrix}$$

La soluzione si ottiene dall'inversione della matrice a primo membro. Si puó quindi capire che la difficoltá principale sia questa, soprattutto per sistemi con un gran numero di equazioni.

Un'altra difficoltá nell'applicare lo schema implicito consiste nel fatto che in genere le equazioni non lineari sono difficili da esplicitare. Ossia, data una equazione in cui $F(t, y)$ é una funzione non lineare, l'inversione non é affatto scontata, ed in genere si ricorre a soluzioni numeriche per calcolare dalla (2.20) una espressione esplicita per y_{n+1}.

Esempio (banale!). Risolvere con uno schema implicito l'equazione non lineare $\dot{y} = \sin y$.

Lo schema implicito, per esempio di Crank–Nicolson, si riduce a risolvere l'equazione alle differenze

$$y_{n+1} - \frac{\Delta t}{2} \sin y_{n+1} = y_n + \frac{\Delta t}{2} \sin y_n$$

la cui soluzione, ovvero esplicitare y_{n+1}, non é assolutamente banale.

Vista l'importanza degli schemi impliciti, é comunque utile implicitare sempre la parte lineare di una qualsiasi equazione. Questo in genere stabilizza gli schemi classici. Ossia, data l'equazione $y = cy + f(t, y)$, dove $f(t, y)$ é una funzione non lineare, é utile scrivere uno schema in cui la parte lineare del secondo membro cy é implicitata, mentre la funzione $f(t, y)$ é calcolata usando uno schema qualsiasi visto finora. Per esempio usando uno schema implicito di Crank–Nicolson per la parte lineare, ed uno schema di Eulero per la parte non lineare, si ottiene l'equazione alle differenze seguente

$$\left(1 - \frac{c\Delta t}{2}\right) y_{n+1} = -\frac{c\Delta t}{2} y_n + \Delta t f(t_n, y_n)$$

Nel caso di uno schema Adams–Bashforth per la parte non lineare si ottiene l'equazione

$$\left(1 - \frac{c\Delta t}{2}\right) y_{n+1} = -\frac{c\Delta t}{2} y_n + \frac{\Delta t}{2} \left[3f(t_n, y_n) - f(t_{n-1}, y_{n-1}\right]$$

e cosí via negli altri casi.

2.8 Equazioni di ordine superiore

Una equazione di ordine superiore puó sempre ridursi ad un sistema di equazioni del primo ordine. Per esempio consideriamo l'equazione del secondo ordine

$$\frac{d^2y}{dt^2} = F(t, y, \dot{y})$$

con le condizioni iniziali $y(0) = y_0$ ed $y(0) = V$. Questa equazione puó sempre ridursi ad un sistema di due equazioni

$$\begin{aligned} \frac{dy}{dt} &= u \\ \frac{du}{dt} &= F(t, y, u) \end{aligned}$$

che puó essere risolto con i metodi usuali che abbiamo descritto finora.

Quanto detto vale in generale, mentre per equazioni speciali si possono studiare metodi numerici di tipo diverso che sono piú efficaci. Per esempio consideriamo l'equazione $y'' = F(t, y)$ in cui manca il termine in derivata prima y' nella funzione F, con condizioni iniziali date $y(0) = y_0$ ed $y'(0) = V$. Allora un metodo migliore per la risoluzione numerica é lo schema di Störmer che consiste nello stimare la derivata seconda su tre punti usando l'equazione (35) e nello stimare la funzione $F(t, y)$ usando una espansione in serie

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta t^2} = \sum_{i=j}^{k} b_i F(t_{n-i}, y_{n-i}) \tag{2.25}$$

I coefficienti b_i si fissano imponendo che l'errore commesso nello schema sia di ordine elevato. Se $j = 0$ il metodo é esplicito, mentre se $j = -1$ il metodo 'e implicito. Un esempio di schema di questo tipo é la formula di Cowell–Numerov–Crommelin

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta t^2} = \frac{1}{12}\left(F_{n+1} + 10F_n + F_{n-1}\right) + E(\Delta t) \tag{2.26}$$

(dove $F_n = F(t_n, y_n)$) che consente di ottenere un errore $E(\Delta t) \sim 0(\Delta t^6)$, molto piú elevato dei metodi usuali.

2.9 Problemi con condizioni al contorno

Finora abbiamo visto un certo tipo di problemi dinamici. I problemi con condizioni al contorno in genere sono differenti. Infatti quando si studiano problemi di questo tipo, il problema principale della risoluzione della equazione differenziale non é la stabilitá dello schema, ma ci sono ulteriori difficoltá che emergono, dovute al fatto che, in questo caso, bisogna risolvere contemporaneamente un sistema di equazioni lineari o non lineari, generalmente tridiagonale.

Consideriamo una funzione $y(x)$ che soddisfa alla seguente equazione non omogenea del secondo ordine a coefficienti non costanti

$$\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} + w(x)y = f(x) \tag{2.27}$$

dove $p(x)$, $w(x)$ ed $f(x)$ sono funzioni qualsiasi. La soluzione sia definita in un intervallo $x \in [a, b]$ con due condizioni al contorno, che in generale saranno espresse dalle relazioni che legano la funzione e la derivata prima nei due punti estremi dell'intervallo, ossia $y(a) + \alpha_a y'(a) = \beta_a$ ed $y(b) + \alpha_b y'(b) = \beta_b$. Supponiamo di dividere l'intervallo $[a, b]$ in passi finiti di ampiezza Δx, in modo che ogni punto é fissato da $x_i = i\Delta x$ $(i = 0, 1, \ldots, N)$, ed usiamo schemi del secondo ordine in Δx per le derivate della equazione (2.27). Allora si otterrá il sistema

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{\Delta x^2} + p(x_i)\frac{y_{i+1} - y_{i-1}}{2\Delta x} + w(x_i)y_i = f(x_i) \tag{2.28}$$

che in generale porta ad un sistema di N equazioni per y_i del tipo: $a_i y_{i-1} + b_i y_i + c_i y_{i+1} = d_i$, dove a_i, b_i, c_i e d_i sono coefficienti che dipendono in generale dal punto x_i. Come si vede il sistema é un sistema di N equazioni in $N + 2$ incognite, per cui 2 delle incognite devono essere ricavate dalle condizioni al contorno. In genere le condizioni al contorno saranno espresse dalle due relazioni seguenti

$$\begin{aligned} y_0 + \alpha_a \frac{y_2 - y_0}{2\Delta x} &= \beta_a \\ y_{N+1} + \alpha_b \frac{y_{N+1} - y_{N-1}}{2\Delta x} &= \beta_b \end{aligned}$$

da cui é possibile ottenere le due incognite y_0 ed y_{N+1} in funzione di y_2 e di y_{N-1}

$$
\begin{aligned}
y_0 &= \frac{\beta_a - \alpha_a y_2/2\Delta x}{1 - \alpha_a/2\Delta x} \\
y_{N+1} &= \frac{\beta_b - \alpha_b y_{N-1}/2\Delta x}{1 - \alpha_b/2\Delta x}
\end{aligned}
\tag{2.29}
$$

e quindi sostituendo nell'espressione (2.28) si ottiene il sistema seguente

$$
\begin{pmatrix}
b_1 & c_1 - \frac{a_1\alpha_a}{2\Delta x - \alpha_a} & 0 & 0 & - \\
a_2 & b_2 & c_3 & 0 & -- \\
0 & a_3 & b_3 & c_4 & 0 \\
- & - & - & - & - \\
- & 0 & 0 & a_N - \frac{c_N\alpha_b}{2\Delta x - \alpha_b} & b_N
\end{pmatrix}
\begin{pmatrix} y_1 \\ y2 \\ - \\ y_{N-1} \\ y_N \end{pmatrix} =
$$

$$
= \begin{pmatrix} d_1 - \frac{\alpha_1\beta_a 2\Delta x}{2\Delta x - \alpha_a} \\ d_2 \\ - \\ d_{N-1} \\ d_N - \frac{\alpha_N\beta_b 2\Delta x}{2\Delta x - \alpha_a} \end{pmatrix}
\tag{2.30}
$$

Come si vede la matrice al primo membro é una matrice *tridiagonale*, ossia gli unici elementi differenti da zero stanno sulla diagonale e sulle due diagonali sopra e sotto la diagonale principale. L'inversione di una tale matrice, pur presentando qualche difficoltá, puó essere trattata abbastanza facilmente. Quindi lo sforzo computazionale nella soluzione di un problema con condizioni al contorno sta nella inversione della matrice tridiagonale a primo membro della (2.30).

Parte II

Programmazione in Fortran 77

2.10 Introduzione

Diamo qui di seguito un breve sommario delle istruzioni fondamentali del linguaggio FORTRAN 77, che sarà il linguaggio che utilizzeremo negli esempi pratici che seguiranno. Il FORTRAN (da FORmula TRANslator, cioé "traduttore di formule") è un linguaggio di programmazione altamente efficiente per l'esecuzione di programmi di tipo matematico. Ovviamente il linguaggio permette la scrittura anche di programmi di tipo differente, ma risulta particolarmente efficace per il calcolo numerico, essendo ottimizzato per eseguire funzioni o operazioni matematiche. Per questo motivo, il FORTRAN 77 è tuttora utilizzato nella scrittura di programmi di tipo scientifico. Un secondo motivo è rappresentato dal fatto che esistono centinaia di funzioni e subroutines (vedi oltre), cioé pezzi di programma che eseguono operazioni particolari, che sono liberamente disponibili a tutti i programmatori e che possono essere inclusi nei programmi da sviluppare, velocizzando la scrittura dei codici ed evitando di riscrivere brani di programma molto complessi.

La prima versione del linguaggio FORTRAN risale al 1966. Tale versione del linguaggio è ormai considerata obsoleta perchè consentiva l'utilizzo di costrutti ormai abbandonati nei linguaggi di programmazione moderni. Il FORTRAN 77 è il diretto discendente di quella prima versione, che è stata chiamata FORTRAN 4. È stato sviluppato nel 1977 ed è a sua volta considerato un linguaggio non moderno in quanto non consente, ad esempio, la programmazione ad oggetti, la definizione di tipi complessi (strutture), il dimensionamento dinamico delle variabili, ecc.. Inoltre, il FORTRAN 77 standard conserva ancora una formattazione rigida, come vedremo in seguito, risalente all'epoca in cui si utilizzavano le schede perforate per introdurre i dati nel computer. Per ovviare a tali inconvenienti, nel 1990 è stata introdotta una nuova versione del linguaggio chiamata FORTRAN 90. Esiste una versione addirittura successiva, il FORTRAN 95, creata appunto nel 1995, che cerca di automatizzare il passaggio dei programmi da macchine a singolo processore a macchine a più processori. In ogni caso, tutte queste versioni del linguaggio sono compatibili "verso l'alto", nel senso che un compilatore FORTRAN 90 compilerà senza problemi un programma scritto in FORTRAN 77, sebbene potrebbe protestare leggermente (ma lo compilerà comunque) se il programma è scritto in FORTRAN 4.

A questo punto ci si puó chiedere perché non utilizzare direttamente il FORTRAN 90 per scrivere i programmi di esempio. Una prima ragione è che il FORTRAN 77, con i suoi costrutti più semplici, rappresenta un approccio più "naturale" per chi si appresta ad imparare un linguaggio di programmazione senza avere precedenti esperienze. Inoltre, un aspetto non trascurabile, è rappresentato dal fatto che, mentre il compilatore FORTRAN 77 è distribuito

gratuitamente sulle varie distribuzioni di Linux, il compilatore `FORTRAN 90` andrebbe acquistato a parte (anche se alcuni compilatori, tipo il compilatore INTEL, sono gratuitamente scaricabili da Internet purché si dichiari che l'uso che se ne farà è di tipo non commerciale), cosa non raccomandabile dal punto di vista didattico. Del resto, il `FORTRAN 77` è più che sufficiente per i programmi di esempio che andremo a realizzare.

In generale, un programma `FORTRAN 77` consiste delle seguenti parti: *a)* una **parte dichiarativa**, in cui si definiscono il nome assegnato al programma, le variabili che saranno utilizzate, cioé quelle entità su cui il programma opererà come numeri, lettere, nomi, ecc., ed eventuali costanti; le **istruzioni** vere e proprie, cioé la sequenza di operazioni sulle variabili che rappresentano il programma stesso; le istruzioni `STOP` ed `END`, che indicano la fine del programma. Come vedremo, mentre l'istruzione `END` indica la fine vera e propria del codice, l'istruzione `STOP` può apparire anche all'interno del programma stesso se, per qualche particolare ragione, questo deve terminare prima della sua fine "naturale". Vedremo in seguito esempi in questo senso. Dopo la sequenza di istruzioni `STOP-END`, possono trovarsi **subroutines** o **funzioni**: vedremo in dettaglio nel seguito cosa sono queste entità e come si utilizzano.

2.10.1 Formattazione del codice

Poiché il `FORTRAN 77` è un linguaggio che risale, appunto, al 1977 e a quell'epoca l'immissione dei dati nei computers veniva effettuata non tramite tastiera, come in epoca moderna, ma attraverso schede perforate, occorreva una formattazione rigida del testo che consentisse al lettore di schede di distinguere etichette, righe di istruzione, ecc.. Ancora oggi il linguaggio risente di queste sue origini remote ed ha mantenuto la formattazione fissata anni fa. I moderni compilatori consentono di ovviare a questo problema con opportune opzioni di compilazione, mentre il Fortran 90 ha del tutto abbandonato la formattazione rigida del testo e ne consente una assolutamente libera. Tuttavia, a meno di non star utilizzando il Fortran 90, noi suggeriamo sempre di attenersi allo standard del linguaggio, che consente una portabilità quasi assoluta sulla maggior parte delle macchine esistenti.

In `FORTRAN 77`, ogni riga di dichiarazione o di istruzioni deve iniziare dalla settima colonna. Le prime 6 colonne del file che contiene il programma sono infatti riservate a contenere eventuali etichette numeriche che identificano le righe. Inoltre le righe di dichiarazione delle variabili o di istruzioni non debbono superare la settantaduesima colonna. Nel caso si sia costretti a scrivere una riga che superi tale colonna, occorrerà "andare a capo" interrompendo la linea prima della settantaduesima colonna, quindi porre sulla riga seguente, alla sesta colonna, un *caratterre di continuazione* e continuare

la scrittura della riga precedente. Tale operazione può essere ripetuto un numero di volte che dipende dal compilatore, ma in genere è molto grande (alcuni compilatori moderni accettano centinaia di righe di continuazione).

Il *carattere di continuazione* può essere qualunque carattere alfanumerico (lettere o numeri o simboli rappresentabili sulla tastiera) tranne lo zero. In genere, sia per motivi pratici (evitare confusioni col resto della riga, introducendo caratteri strani) che per motivi di stile, si utilizza come carattere di continuazione il carattere: **&**. In ogni caso, ciò non è obbligatorio.

Una linea di codice cha abbia un carattere "**C**" o "*****" nella prima colonna inizia una riga di commento. Alcuni moderni compilatori accettano anche altri caratteri come commento ma, al solito, è sempre opportuno attenersi allo standard del linguaggio nella programmazione. È inoltre possibile inserire brevi commenti sulle linee di istruzione inserendo il carattere "!" e il relativo commento alla fine della linea. Ovviamente, Le linee di commento sono utili solo per le persone che leggono i programmi, di conseguenza sono sempre ignorate dai compilatori. Di conseguenza, le linee di commento, anche se seguono linee di istruzioni, non debbono necessariamente rispettare la formattazione del programma. In altre parole è, ad esempio, possibile inserire linee di commento che vadano oltre la 72-esima colonna, ecc..

Come detto, le colonne dalla prima alla quinta (la sesta è riservata al carattere di continuazione), venivano utilizzate nei vecchi compilatori per contenere delle etichette numeriche per identificare le linee. Infatti, l'antenato del `FORTRAN 77`, vale a dire il `FORTRAN 4`, consentiva l'utilizzo, all'interno del programma, di salti incondizionati a righe identificate da tali etichette numeriche. Inoltre, era possibile utilizzare delle istruzioni condizionali (tipo l'attuale `IF`) con delle condizioni per cui il programma "saltava" alla linea specificata a seconda del valore (positivo, nullo o negativo) di una variabile (`IF` calcolato). Le linee a cui saltare venivano indicate tramite una etichetta numerica. Analogamente, esistevano anche altri usi di tali etichette.

Con l'introduzione della programmazione strutturata, tale modo di programmare venne (giustamente) giudicato obsoleto, in quanto i salti incondizionati diminuiscono la leggibilità del codice ed aumentano il rischio di errori. Il `FORTRAN 77` è un linguaggio strutturato, di conseguenza sono stati introdotti costrutti del linguaggio che consentano di evitare l'uso delle etichette. La possibilità di definire etichette numeriche è però rimasta per garantire la compatibilità con i vecchi programmi in `FORTRAN 4`. L'uso di etichette è tuttavia sconsigliato, tranne un solo caso, cioè con l'istruzione `FORMAT`, che consente di stabilire in quale formato numeri o insiemi di caratteri debbano essere stampati o letti (ad esempio, con quante cifre decimali, nel caso dei numeri). Approfondiremo comunque in seguito questi aspetti.

Infine, notiamo che il `FORTRAN 77` non fa differenza tra lettere maiuscole

o minuscole, quindi tutte le istruzioni o le variabili possono essere indicate con letter maiuscole o minuscole indifferentemente. È uso comune in molti testi di indicare le istruzioni del linguaggio con le maiuscole e le variabili con le minuscole: cercheremo, nel seguito, ove possibile, di seguire anche noi tale convenzione, ma non in maniera rigorosa.

2.11 Parte dichiarativa

Come detto, la parte dichiarativa di un programma può contenere diverse istruzioni, che cercheremo ora di discutere in maggiore dettaglio.

2.11.1 Nome del programma

La prima istruzione `FORTRAN 77`, che si incontra nella parte dichiarativa, è l'istruzione `PROGRAM Nome_Programma`, che serve semplicemente a dire al compilatore che il programma si chiamerà `Nome_Programma`. Tale istruzione non è assolutamente obbligatoria, e può essere tranquillamente omessa, tuttavia può essere utile aggiungerla al codice per indicare brevemente che cosa fa il programma stesso. Noi useremo sempre tale istruzione nei programmi di esempio.

2.11.2 Dichiarazione di variabili

Una variabile è un segmento di celle di memoria (*locazioni*) consecutive, in cui il computer va a memorizzare i dati su cui dovrà operare. Tali locazioni di memoria vengono identificate all'interno dei programmi tramite nomi mnemonici (di preferenza, che servano a comprendere a quale uso tale parte della memoria è destinato!) chiamate, appunto, variabili. Il nome di una variabile sarà quindi una sequenza di lettere o numeri (anche altri simboli sono ammessi, ma in numero limitato; ad esempio il simbolo "`_`" è usato comunemente). Esistono però dei vincoli: il primo carattere di un nome di variabile deve essere una lettera (ad esempio, "`var_1`" è un nome ammissibile, mentre "`1_var`" non lo è!); inoltre, secondo lo standard del linguaggio, solo i primi sei caratteri del nome sono riconosciuti come distinti, vale a dire che un nome di variabile può essere più lungo di sei caratteri, ma due variabili che abbiano i primi sei caratteri uguali sono visti come la stessa entità nel programma (ad esempio, "`pippo_1`" e "`pippo_2`" sono visti come la stessa variabile). Tale limitazione è stata ormai superata da tutti i compilatori moderni che accettano molti più caratteri per distinguere le variabili,

in ogni caso è bene tenere presente che, su alcune macchine, certi programmi potrebbero dare problemi e non funzionare per questo motivo.

Ritornando alle variabili, esse servono quindi a conservare dei valori, su cui verranno effettuate delle operazioni, nella memoria del computer. Tali valori possono essere di tipo numerico, quando si eseguono calcoli, come in un programma di calcolo numerico, o anche caratteri o nomi quando, ad esempio, i programmi debbono operare su nomi, cognomi, indirizzi, ecc., come un programma che gestisce un elenco telefonico. Il `FORTRAN 77` accetta i seguenti tipi di variabile:

- **integer**: serve a memorizzare numeri interi con segno. Occupa due locazioni in memoria e può contenere numeri compresi nel seguente intervallo: `[-32768,+32767]`;

- **integer*4**: serve a memorizzare numeri interi, con segno, che non possono essere contenuti in un `integer` standard. Occupa 4 bytes in memoria e l'intervallo di variabilità è: `[-2147483648,+2147483647]`. Molti compilatori moderni usano `integer*4` come standard per i numeri interi, invece di `integer` semplice, ma questo non è standard, quindi bisogna porre molta attenzione a verificare quale è lo standard per gli interi del compilatore che si sta utilizzando, oppure dichiarare gli interi come `integer*4` ove necessario;

- **real**: serve a memorizzare numeri reali con una precisione di 7 cifre decimali. Occupa 4 bytes e l'intervallo di variabilità dipende dal compilatore che si sta utilizzando. In genere i numeri rappresentabili vanno da 10^{-38} a 10^{+38} circa;

- **real*8** (o **double precision**): serve a memorizzare reali con una precisione fino a 15 cifre decimali. Occupa 8 bytes in memoria e l'intervallo di variabilità (che dipende ancora dal compilatore) include numeri compresi tra 10^{-308} a 10^{+308} circa;

- **complex**: serve a memorizzare numeri complessi. Come nell'ordinaria matematica, è rappresentato da una coppia di numeri reali (parte reale ed immaginaria), di conseguenza occuperà $2 \times 4 = 8$ bytes in memoria;

- **complex*16** (o **double complex**): serve a memorizzare numeri complessi con maggiore precisione ed equivale ad una coppia di reali in doppia precisione, quindi occupa $2 \times 8 = 16$ bytes in memoria;

- **character**: serve a memorizzare singoli caratteri. Occupa un singolo byte in memoria;

- **character*num**: memorizza sequenze di **num** caratteri consecutivi, anche chiamate **stringhe di testo**;
- **logical**: sono variabili particolari, utilizzate nelle espressioni condizionali, come vedremo in seguito, che possono assumere esclusivamente due valori: `.TRUE` (o `.T.`), cioé **vero**, oppure `.FALSE.` (o `.F.`), cioé **falso**.

2.11.3 Dichiarazioni implicite

A differenza di quanto accade nella maggior parte dei linguaggi moderni, le variabili non devono essere necessariamente dichiarate! Il `FORTRAN 77` accetta infatti la cosiddetta *dichiarazione implicita*, cioé tutte le variabili il cui nome inizia con una lettera nell'insieme da `A` ad `H` e da `O` fino a `Z` sono automaticamente interpretate dal compilatore (se non vengono dichiarate all'inizio del programma!) come reali. Tutte le variabili i cui nomi iniziano con lettere nell'intervallo da `I` ad `N` vengono considerate intere (se non altrimenti dichiarate). Ad esempio, una variabile non dichiarata il cui nome sia `esponente` viene considerata automaticamente un reale, mentre una variabile di nome `numero` viene considerata automaticamente intera. Tuttavia, come detto, questo vale solo per le variabili non espressamente dichiarate all'inizio del programma. Questo modo di programmare è però spesso fonte di errori grossolani, specialmente in programmi lunghi e complessi, di conseguenza, specialmente per chi inizia a programmare, è buona norma non utilizzare questa possibilità. Esiste un modo di dire al compilatore che si vuole rinunciare a questa opportunità di non dichiarare esplicitamente le variabili. Basta porre, all'inizio della parte dichiarativa del programma, subito dopo il comando `PROGRAM`, il comando:

```
IMPLICIT NONE
```

che dice al compilatore che nessuna variabile deve essere supposta come implicitamente dichiarata. Se si utilizza quindi nel programma una variabile non esplicitamente dichiarata, il compilatore segnalerà il problema all'atto della compilazione.

È possibile invece anche cambiare la regola di default del compilatore e, per esempio, dichiarare implicitamente reali tutte le variabili il cui nome inizia per una lettera differente da `A-H` oppure `O-Z`, tramite l'istruzione:
`IMPLICIT` *tipo (carattere1, carattere2, carattere3-carattere4, ...)*
che dichiara le variabili che iniziano per *carattere1* del tipo *tipo*, e lo stesso per quelle che iniziano per *carattere2* o le cui iniziali stanno nell'intervallo tra *carattere3* fino a *carattere4*. Ad esempio:

```
IMPLICIT CHARACTER*80 (C), COMPLEX (Q,X-Z)
```

dichiarerà implicitamente come stringhe di 80 caratteri le variabili che iniziano per "C" e come numeri complessi quelle che iniziano per "Q" o per "X", "Y", "Z".

Tuttavia, la dichiarazione implicita delle variabili è considerata obsoleta e pericolosa, quindi nei nostri programmi utilizzeremo il più possibile l'istruzione IMPLICIT NONE.

2.11.4 Dichiarazione di costanti

È spesso utile poter definire delle quantità che non variano all'interno del programma, cioè delle costanti. Questo è utile non solo per identificare costanti numeriche tipo π (pi greco), e (numero di Neper), ecc., ma anche per identificare tutte quelle quantità che all'interno del programma attuale debbono mantenersi costanti, ma che invece si dovrà far variare quando il programma verrà eseguito nuovamente. In questo caso, invece di andare a ricercare il valore da cambiare dappertutto all'interno del programma, basterà cambiare il valore della costante nella parte dichiarativa. Questo accade sovente per parametri che appaiono nel programma o per le dimensioni dei vettori e delle matrici, come vedremo meglio in seguito. Per dichiarare una costante, bisogna dapprima dichiararne il tipo come se fosse una variabile, quindi utilizzare l'istruzione:

```
PARAMETER (nome\_costante=valore\_costante, ...)
```

Ad esempio, se vogliamo dichiarare π una costante e assegnargli il suo valore dovremo usare:

```
REAL PI
PARAMETER (PI=3.1415926)
```

Più parametri possono essere dichiarati all'interno della singola istruzione PARAMETER, mentre ovviamente si possono definire più linee con l'istruzione PARAMETER.

La parte dichiarativa può contenere ulteriori istruzioni complesse, ma quelle date finora sono le principali ed anche quelle che utilizzeremo intensivamente nei programmi di esempio che vedremo in seguito.

2.12 Istruzioni

Dopo la parte dichiarativa, in un programma FORTRAN 77 dobbiamo inserire le istruzioni, cioé dobbiamo indicare al computer cosa deve fare con i dati per i quali abbiamo riservato una parte della memoria. Le istruzioni fondamentali si possono dividere in:

- **Espressioni aritmetiche e di assegnazione**
- **Istruzioni di Input/Output (I/O)**
- **Espressioni condizionali**
- **Cicli**

Analizziamo quindi singolarmente le istruzioni.

2.12.1 Espressioni aritmetiche e di assegnazione

Fra i numeri o le variabili si possono definire operazioni aritmetiche di calcolo, indicate con i simboli: `+` per la somma, `-` per la differenza, `*` per la moltiplicazione, `/` per la divisione, `**` per l'elevamento a potenza. Il simbolo "`=`" non rappresenta l'uguaglianza, come nella matematica ordinaria, ma un'assegnazione, cioé assegna alla quantità che sta a sinistra, il valore che sta a destra. Ad esempio, la relazione:

```
x = x**2 + 5.0
```

non rappresenta una equazione nell'incognita `x`, bensì vuole dire: assegna alla variabile `x` il valore della stessa variabile elevato al quadrato (`x**2`) e somma a tale valore il numero 5.

La priorità delle operazioni può essere modificato tramite l'uso delle parentesi tonde, come nell'ordinaria matematica (ma solo le tonde, non le quadre o graffe!). Normalmente, a meno che non si utilizzino le parentesi, l'operatore `**` ha la priorità su `*` e `/`, che a loro volta hanno la priorità su tt + e `-`. Ad esempio:

```
2.0*x**2+x/y*3
```

eseguirà prima `x**2`, cioé x^2, quindi moltiplicherà per 2 tale valore, quindi eseguirà x/y, moltiplicherà il risultato per 3 e infine sommerà i due termini così calcolati.

2.12.2 Istruzioni di Input/Output (I/O)

Sono istruzioni che servono a leggere dei dati dalla tastiera o da un file, ovvero a stampare qualche risultato o informazione sullo schermo o su un file. Per poter scrivere o leggere su un file, tale file deve essere "aperto" tramite l'istruzione `OPEN`, che ha il seguente formato:

```
OPEN (FILE='nome_file', UNIT=numero, STATUS='status')
```

apre il file identificato dalla stringa `nome_file` (che può essere una variabile o una costante), assegnandogli un numero di unità `numero`, tramite il quale le istruzioni di scrittura o lettura faranno successivamente riferimento al file. Lo `STATUS` può assumere i valori: `NEW`, che indica che il file con quel nome ancora non esiste, `OLD`, che indica che esiste già una vecchia versione del file che deve essere eventualmente sovrascritta, `UNKNOWN` che indica che non si sa se il file è nuovo o esiste già. Noi utilizzaremo quasi sempre lo status `UNKNOWN` nei programmi che seguono, tuttavia gli altri stati possono essere molto utili per evitare errori. Ad esempio, se si apre un file esistente per leggerlo, specificare `STATUS='OLD'` può essere utile, perché se per qualche motivo il file non ci fosse il programma si bloccherebbe. Analogamente, se si vuole aprire un file per scriverci sopra utilizzando lo `STATUS='NEW'`, il programma si arresterebbe se esiste già un file con lo stesso nome, evitando una sovrascrittura magari non voluta. Tuttavia, sebbene meno sicuro, l'utilizzo di `STATUS='UNKNONW'` semplifica di molto la vita. L'istruzione `OPEN` consente l'utilizzo di molti altri parametri, ma noi ci limiteremo a quelli che utilizzeremo correntemente.

Una volta che il file sia stato aperto, possiamo andare a scriverci sopra o a leggere da esso, tramite le due istruzioni:

```
WRITE(numero,formato) var1, var2, 'stringa', ecc.
```

oppure:

```
READ(numero,formato) var1, var2, 'stringa', ecc.
```

che, rispettivamente, scrive le variabili `var1`, ecc., ovvero la stringa `'stringa'`, sul file specificato dall'unità `numero`, oppure le leggono. Il termine `formato` può rappresentare il simbolo "`*`", ovvero una *etichetta numerica*. Il simbolo "`*`" indica che le variabili verranno scritte in *formato libero*, cioé con tutte le cifre significative se si tratta di numeri o con tutti i caratteri della stringa, se si tratta di stringhe. Alternativamente, l'etichetta numerica indicata dal numero `formato` indica che la maniera in cui le variabili andranno scritte si trova in una riga etichettata (tra la prima e quinta colonna) con il numero `formato` e seguita da un comando: `FORMAT(formato_var1, formato_var2, ecc.)`.

Una discussione, anche superficiale, sui formati di scrittura richiederebbe molto spazio e molti paragrafi, di conseguenza il lettore che voglia approfondire la questione è invitato a leggere un qualunque manuale del linguaggio `FORTRAN 77`, dove troverà spiegazioni dettagliate in merito. Per quanto ci concerne, noi eviteremo sempre di utilizzare formati particolari negli esempi che vedremo in seguito e invece utilizzeremo sempre il *formato libero* di scrittura, cioé l' "`*`". Notiamo tuttavia che per leggere con l'istruzione

READ una stringa da un file è SEMPRE necessario stabilire il formato di lettura per le stringhe. Tale necessità non si presenterà mai nei programmi che esamineremo nel seguito della trattazione.

Ovviamente, non è sempre necessario scrivere o leggere da un file, ma potrebbe essere necessario scrivere sullo schermo o leggere da tastiera il valore di una variabile. A questo scopo si possono utilizzare le stesse istruzioni WRITE e READ appena citate, con l'accortezza di indicare "6" come numero dell'unità di scrittura (che rappresenta lo schermo), e "5" per l'unità di lettura (che rappresenta la tastiera). Naturalmente, non trattandosi di una scrittura/lettura su file, non è necessario utilizzare preventivamente l'istruzione OPEN. Quindi una sequenza di istruzioni del tipo:

```
PROGRAM KEY_INPUT IMPLICIT NONE REAL A,B

WRITE(6,*) 'Scrivi due numeri A e B:'
READ(5,*) A,B
WRITE(6,*) 'Risultato del prodotto A*B: ',A*B
STOP
END
```

leggerà dalla tastiera i due numeri (dichiarati reali nella parte parte dichiarativa) A e B e ne stamperà il prodotto A*B sulla stessa linea della stringa di testo: "Risultato del prodotto A*B: ". Notiamo che i numeri "5" e "6" possono essere sostituiti entrambi dal simbolo convenzionale *, visto che comunque la scrittura si intende fatta sullo schermo e la lettura dalla tastiera senza possibilità di confusione. Così: WRITE(*,*) è del tutto equivalente a: WRITE(6,*, mentre: READ(*,*) equivale a: READ(5,*).

Come si vede, il FORTRAN 77 assume che esistano delle unità, in particolare la "5" e la "6" che sono destinate a scopi particolari. In generale quindi è sconsigliato, nell'apertura di un file con l'istruzione OPEN di utilizzare numeri di unità inferiori a "10", in quanto ciò potrebbe portare a conflitti con le unitá implicitamente "aperte" dal compilatore. Il numero massimo di unità consentito dipende dal compilatore e dal sistema operativo, in ogni caso tipicamente si utilizzano numeri di unità compresi tra "10" e "255", che non dovrebbero dare problemi.

Notiamo inoltre che l'istruzione WRITE(6,*) var può anche essere sostituita dall'istruzione PRINT *,var, che stampa esclusivamente sullo schermo. L'asterisco rappresenta, anche in questo caso, il formato libero e può essere sostituito dall'etichetta numerica di una linea di formato.

Infine, si può anche scrivere su un file che non sia stato precedentemente aperto con l'istruzione OPEN. In questo caso, quando il computer incontra una istruzione del tipo: WRITE(num,*), aprirà automaticamente un file che

si chiamerà: `fort.num` (il nome dipende in realtà dal sistema operativo utilizzato, ma per Unix/Linux `fort.num` è lo standard). Analogamente, una istruzione di lettura potrà essere effettuata da un file non aperto con l'istruzione: `READ(num,*)` che leggerá dal file: `for.num` che ovviamente deve essere stato preventivamente creato.

2.12.3 Istruzioni condizionali

Le istruzioni condizionali consentono al programma di operare in maniera che dipende dal valore di una espressione, detta appunto *espressione condizionale*, che può essere vera oppure falsa. Un tipico esempio è costituito dall'istruzione `IF-THEN-ELSE-ENDIF`, che ha la forma:

```
IF (espressione_condizionale) THEN
   blocco_istruzioni_1
ELSE
   blocco_istruzioni_2
ENDIF
```

in cui, se l'espressione `espressione_condizionale` è verificata, la sequenza `blocco_istruzioni_1` viene eseguita, mentre il `blocco_istruzioni_2` viene ignorato. Viceversa, se la `espressione_condizionale` risulta falsa. Esistono anche forme più complesse di condizioni, come la:

```
IF (espressione_condizionale_1) THEN
   blocco_istruzioni_1
ELSE IF (espressione_condizionale_2) THEN
   blocco_istruzioni_2
ELSE IF (espressione_condizionale_3) THEN
   blocco_istruzioni_3
...
ELSE
   blocco_istruzioni_n
ENDIF
```

che esegue `blocco_istruzioni_1` se `espressione_condizionale_1` è vera, esegue `blocco_istruzioni_2` se `espressione_condizionale_2` è vera, esegue `blocco_istruzioni_3` se `espressione_condizionale_3` è vera, ecc., ovvero `blocco_istruzioni_n` se tutte le precedenti sono false. Ovviamente si può utilizzare un numero rilevante di sequenze `ELSE IF (...) THEN`. Notiamo che, in tutti i casi, quando una delle istruzioni condizionali è vera e quindi il relativo blocco di istruzioni viene eseguito, alla fine del blocco il

programma prosegue dalla istruzione seguente l'`ENDIF`. Ciò vuol dire che anche se, ad esempio, due istruzioni condizionali della sequenza sono vere, solo il blocco di istruzioni che si riferisce alla prima di esse viene eseguito, perché subito dopo il programma va all'istruzione seguente l'`ENDIF` e le ulteriori possibilità di confronto logico non vengono analizzate.

Analogamente, se si deve eseguire un singolo `blocco_istruzioni`, l'istruzione `ELSE` deve essere eliminata. Se addirittura si deve eseguire una singola istruzione se l'espressione condizionale è verificata, basta la seguente linea:

```
IF (espressione_condizionale) istruzione
```

Cos'è l'espressione condizionale? Può essere o una variabile logica (che può appunto assumere i valori vero o falso), oppure dev'essere una istruzione di confronto logico tra variabili. Il confronto logico tra variabili si effettua tramite i seguenti operatori:

- `A.EQ.B`: vero se la variabile `A` è uguale alla variabile `B` ($A = B$), falso se il contrario;
- `A.NE.B`: vero se $A \neq B$, falso se $A = B$;
- `A.GT.B`: vero se $A > B$, falso se il contrario;
- `A.LT.B`: vero se $A < B$, falso se il contrario;
- `A.GE.B`: vero se $A \geq B$, falso se il contrario;
- `A.LE.B`: vero se $A \leq B$, falso se il contrario.

Naturalmente, le variabili `A` e `B` devono essere dello stesso tipo per poter essere confrontate. Si possono, allo stesso modo delle variabili numeriche, confrontare due stringhe di testo o caratteri. In questo caso, le relazioni di maggiorazione o minorazione si riferiscono all'ordine alfabetico dei caratteri che costituiscono le stringhe di testo.

Le variabili logiche possono essere utilizzate per contenere il risultato di una espressione condizionale, quindi essere inserite nell'argomento di un `IF`. Ad esempio, il seguente programma:

```
PROGRAM CONDIZIONI
IMPLICIT NONE
REAL A,B
LOGICAL LLL

B=0.5
```

```
READ(5,*) A
LLL=A.GT.B
IF (LLL) THEN
   PRINT *,'La variabile A = ', A
   PRINT *,'risulta maggiore della variabile'
   PRINT *,'B = ',B
ELSE
   PRINT *,'A < B'
ENDIF
IF (A.EQ.0.0) PRINT *,'Hai scritto A=0!'
STOP
END
```

riceve in input la variabile `A`; la variabile logica `LLL` contiene il risultato vero (`.TRUE.`) se $A > 0.5$, falso (`.FALSE.`) se $A \leq 0.5$. Nel primo caso, viene stampato un messaggio che avverte che $A > B$, il contrario se `LLL` risulta falsa. Infine, se per caso si è inserito un valore di `A` pari a 0.0, il programma avverte anche di questo con l'`IF` finale (`IF (A.EQ.0.0) ...`).

Più istruzioni condizionali possono essere combinate tra loro tramite operatori logici. Alcuni degli operatori ammessi sono i seguenti:

- `expr1.AND.expr2` che dà come risultato "vero" se `expr1` ed `expr2` sono entrambe "vere", dà come risultato "falso" se almeno una delle due espressioni risulta falsa;
- `expr1.OR.expr2` che dà come risultato "vero" se almeno una tra `expr1` ed `expr2` è "vera"; dà come risultato "falso" se entrambe le espressioni sono "false";
- `.NOT.expr` che dà come risultato "falso" se `expr` è "vera", mentre dà "vero" se è "falsa".

Per ulteriori operatori logici si rimanda ad un manuale di `FORTRAN 77`.

Esistono ulteriori forme per l'istruzione `IF`, ma sono da considerarsi obsolete e da evitare, presenti nel linguaggio solo per ragioni di compatibilità con il `FORTRAN 4`, quindi non ci curiamo di descriverle.

Un'altra istruzione condizionale del `FORTRAN 77` è la `DO WHILE-ENDDO`, che esegue ciclicamente un blocco di istruzioni mentre una certa condizione, espressa al solito da una *istruzione condizionale*), è verificata. Il formato dell'istruzione è il seguente:

```
DO WHILE (espressione_condizionale)
   blocco_istruzioni
ENDDO
```

In questo caso, il blocco istruzioni viene eseguito quando l'espressione condizionale risulta vera, altrimenti non viene eseguito. Ad esempio, il seguente programma calcola i valori dei quadrati dei primi 10 numeri interi:

```
PROGRAM QUADRATI
IMPLICIT NONE
INTEGER I,I2

I=1
DO WHILE (I.LE.10)
   I2=I*I
   PRINT *,'Numero I = ',I,'     Quadrato I*I = ',I2
   I=I+1
ENDDO
STOP
END
```

Il programma esegue ripetutamente il blocco di istruzioni all'interno delle istruzioni `DO WHILE-ENDDO` fin quando la condizione specificata $I \leq 10$ è verificata. Quando `I` risulterà uguale ad 11, il ciclo verrà abbandonato. Notiamo che, a differenza dell'istruzione `IF`, il blocco di istruzioni viene eseguito ripetutamente mentre la condizione è verificata. Nel caso dell'`IF` il blocco di istruzioni viene eseguito una sola volta. Da questo punto di vista, l'istruzione `DO WHILE` potrebbe includersi anche tra le istruzioni di ciclo che vedremo tra breve.

Notiamo infine che l'istruzione condizionale deve dipendere dalle variabili contenute nel blocco di istruzioni. Se ciò non accade, il ciclo verrà ripetuto all'infinito! Un errore tipico, nell'esempio appena illustrato, sarebbe di "dimenticare" di incrementare la variabile `I` all'interno del blocco_istruzioni. Se ciò accade, cioé se si omettesse la linea: `I=I+1`, il ciclo stamperebbe il quadrato di `I=1` infinite volte!

2.12.4 Cicli

Abbiamo esaminato nel paragrafo precedente l'istruzione `DO WHILE-ENDDO` che esegue ciclicamente una sequenza di istruzioni purché una certa fissata condizione sia soddisfatta. Esiste la possibilità di eseguire in maniera ciclica un blocco di istruzioni un numero fissato di volte, attraverso la sequenza `DO-ENDDO`. Il formato di tale istruzione è il seguente:

```
DO var = e1,e2,e3
   blocco_istruzioni
ENDDO
```

che esegue il blocco istruzioni all'interno della sequenza `DO-ENDDO` facendo assumere alla variabile `var` valori compresi tra `e1` (incluso) ed `e2`, a passi di `e3`. Ad esempio, il ciclo:

```
DO i = 1,21,2
   PRINT *,i
ENDDO
```

stampa i numeri dispari tra 1 e 21 (inclusi). Notiamo che lo svolgimento del ciclo suddetto è il seguente: dapprima il compilatore calcola quante volte il ciclo verrà eseguito come: $N =$`(e2-e1)/e3+1` (eventualmente troncando il rapporto alla parte intera); quindi pone la variabile `i` al primo valore della sequenza, cioé 1; si esegue il blocco di istruzioni (la stampa del numero `i`), quindi si fa assumere alla `i` il valore precedente più l'incremento (cioé il nuovo valore di `i` risulta: 1+2=3). Quindi si riesegue il blocco di istruzioni per il nuovo valore di `i` per un numero di volte totale pari ad N. Dopodichè, l'esecuzione del programma passa all'istruzione seguente. In maniera analoga, il ciclo:

```
DO i = 2,21,2
   PRINT *,i
ENDDO
```

stamperà i numeri pari da 2 fino a 20. Notiamo che quando `i` assume il valore 20, tale valore è l'ultimo stampato in quanto il numero di volte che bisogna eseguire il ciclo è dato da: $N = (21 - 2)/2 + 1 = 10$.

Se l'incremento del ciclo è uguale a 1, esso può essere omesso (e in genere lo si omette sempre). Notiamo inoltre che l'incremento può essere negativo (`e3`< 0), cioé il ciclo va alla rovescia, ma in tal caso l'estremo inferiore (`e1`) dev'essere necessariamente maggiore di quello superiore (`e2`), altrimenti il ciclo non verrà eseguito. Ad esempio:

```
DO i=21,1,-2
   PRINT *,i
ENDDO
```

stampa i numeri dispari da 21 a 1 in ordine inverso, mentre:

```
DO i=1,21,-2
   PRINT *,i
ENDDO
```

non stampa nulla! Questo perché nel primo caso il numero di volte che il ciclo deve essere eseguito risulta essere: $N = (1-21)/(-2)+1 = 11$, mentre nel secondo: $N = (21-1)/(-2)+1 = -9$ che significa mai, visto che non si può eseguire un ciclo un numero negativo di volte! Alcuni compilatori (ad esempio il compilatore GNU incluso nelle distribuzioni Linux) avvertono di questo fatto, altri no. In ogni caso, se `e1`, `e2` o `e3` sono rappresentate da variabili e non da valori numerici costanti, il compilatore non avrebbe modo di accorgersi dell'errore, quindi conviene essere attenti quando si utilizza un ciclo con incremento negativo.

I cicli possono essere nidificati, cioé inseriti l'uno dentro l'altro, ovviamente a patto che le variabili su cui il ciclo viene eseguito siano diverse tra loro. Ad esempio:

```
DO i = 1,10
   DO j = 1,10
      PRINT *,i*j
   ENDDO
ENDDO
```

stamperà il valore del prodotto `i*j` per 100 volte, cioé la tabellina dei primi 10 numeri, solo in formato di una singola colonna. Il numero di cicli nidificati dipende dal compilatore, ma in genere è abbastanza elevato.

In generale, la variabile di ciclo (`var`) e gli estremi dell'intervallo del ciclo e l'incremento (`e1, e2, e3`) possono anche essere dei numeri reali, invece che interi. Tuttavia l'utilizzo dei reali potrebbe dare dei risultati inattesi a causa degli arrotondamenti che il computer effettuerà nell'incrementare la variabile. Ad esempio, il programma:

```
      PROGRAM TEST
      IMPLICIT NONE
      REAL A

      DO A=1.0,2.0,0.2
         PRINT *,A
      ENDDO
      STOP
      END
```

produrrà il seguente output:

```
 1.
 1.20000005
 1.4000001
```

```
1.60000014
1.80000019
2.00000024
```

che significa che il ciclo verrà giustamente eseguito $N = (2.0-1.0)/0.2+1 = 6$ volte, tuttavia nell'effettuare le somme si hanno degli arrotondamenti indesiderati e il ciclo viene eseguito anche se l'ultimo valore è maggiore dell'estremo superiore del ciclo (2.0), cosa che potrebbe non essere voluta. In generale, a meno che non ci siano dei motivi particolari, converrà sempre utilizzare variabili di ciclo intere.

Infine, notiamo che esiste una forma di ciclo, detta *DO implicito*, che viene utilizzata in operazioni di stampa per dare un aspetto più gradevole ad un testo che deve essere messo sotto forma di tabella. Il DO implicito si realizza, nel caso di una scrittura sullo schermo, nel seguente modo:

```
WRITE(6,*) (quantita_che_dipende_da_i, i = e1,e2,e3)
```

cioé una quantità che dipende da una variabile di ciclo `i` viene scritta per tutti i valori tra `e1` a `e2` a passi di `e3` su una singola linea. Ad esempio, il programma che stampa la tabellina dei primi 10 numeri tramite due cicli nidificati, può essere riscritto come:

```
PROGRAM TABELLINA
IMPLICIT NONE
INTEGER I,J

DO j = 1,10
   PRINT *,(i*j,i=1,10)
ENDDO
STOP
END
```

che stamperà, sotto forma di una tabella di 10 righe (i 10 valori di `j`) e 10 colonne (i 10 valori di `i`), il prodotto `i*j`. Analogamente, il DO implicito può essere eseguito anche nel caso di una istruzione di lettura.

2.13 Vettori e matrici

Finora abbiamo esaminato l'uso di variabili scalari, cioè di numeri e semplici sequenze di caratteri, cioé stringhe di testo. Talvolta, specialmente nel calcolo scientifico, si ha a che fare con quantità che non hanno natura scalare, ma sono vettori o matrici definiti su un opportuno spazio vettoriale di dimensione

fissata. Quindi è opportuno che il linguaggio di programmazione fornisca la possibilità di operare su enti di questo tipo. Ad esempio, se vogliamo eseguire un prodotto scalare tra vettori o un prodotto riga per colonna di un vettore per una matrice, conviene avere a disposizione delle variabili che rappresentino enti di questo tipo.

Inoltre, spesso si hanno delle quantità il cui risultato non dipende da un singolo valore numerico, ma da un insieme di valori. Nasce quindi l'esigenza di poter memorizzare, invece che variabili singole in memoria, degli insiemi discreti di variabili indicizzate. Un esempio tipico è costituito da un insieme di misure di cui si vogliano calcolare, ad esempio, la media e la deviazione standard. In tal caso, mentre sarebbe ancora possibile calcolare la media di N numeri inserendoli uno ad uno in memoria sovrascrivendo una variabile x e calcolando la somma, come nel programma che segue:

```
PROGRAM MEDIA
IMPLICIT NONE
REAL X,XN
INTEGER I,N

WRITE(6,*) 'Inserire il numero dei valori:'
READ(5,*) N
X=0.0
DO I=1,N
   READ(5,*) XN
   X=X+XN
ENDDO
PRINT *,'Media dei numeri: ',X/N
STOP
END
```

dove abbiamo usato la variabile `X` per contenere la somma e la variabile `XN` come variabile temporanea per contenere la sequenza di valori. Se volessimo calcolare la deviazione standard, dovremmo re-inserire tutti i numeri una seconda volta, poichè i valori di `XN` sono sovrascritti ad ogni nuovo inserimento, il che sarebbe estremamente poco pratico. Questo succede perchè, per il calcolo della deviazione standard, abbiamo bisogno di conoscere preventivamente il valore medio. Conviene invece avere uno strumento che consenta di inserire una volta per tutte la sequenza di valori e quindi operare su tale sequenza con tutti i calcoli voluti.

Per aiutarci in tutti questi problemi intergono alcune variabili particolari, dette appunto *vettori e matrici*, che si comportano proprio come gli analoghi

dell'algebra lineare. Un **vettore** di dimensione N è una sequenza di N valori di un tipo definito (reali, interi, complessi, o anche caratteri, stringhe o logical), memorizzati consecutivamente nella memoria del computer. Un vettore deve essere dichiarato, come una qualsiasi variabile, con il suo tipo, poi bisogna usare l'istruzione `DIMENSION` (sempre all'interno della parte dichiarativa) per indicare al computer il numero di elementi del vettore per cui bisogna riservare spazio in memoria. Ad esempio:

```
REAL A,B
DIMENSION A(100),B(20)
```

indica che `A` e `B` sono due vettori, di dimensioni 100 e 20, rispettivamente. Ciò vuol dire che il computer allocherà nella sua memoria una sequenza di 100 valori reali, indirizzati con indice da 1 a 100, per la variabile `A`, e una sequenza di 20 valori reali, indicizzati da 1 a 20, per la variabile `B`. Invece di utilizzare due righe, si può condensare il tutto nella singola istruzione:

```
REAL A(100),B(20)
```

Al contrario, se si utilizzano tipi impliciti, si può indicare solo la dimensione di un vettore con l'istruzione `DIMENSION`, ma al solito sconsigliamo l'uso di tale tecnica.

Notiamo che, al contrario di altri linguaggi, in `FORTRAN 77` NON è possibile dimensionare i vettori in maniera dinamica, cioè durante l'esecuzione del programma, indicando ad esempio la dimensione con una variabile, ma la dimensione deve essere SEMPRE una costante il cui valore è noto all'inizio dell'esecuzione del programma. Questo fatto presenta lo svantaggio di dover sempre "abbondare" nello stabilire la dimensione, a meno di non volerla cambiare ogni volta, ricompilando il programma. In altri termini, si dovrà sempre scegliere la dimensione dei vettori esagerando un po', per essere sicuri di avergli dato "abbastanza spazio" per contenere tutti i valori necessari. D'altronde, tale "esegerazione" non deve essere eccessiva per evitare di esaurire la memoria a disposizione sulla macchina inutilmente. Spesso risulta utile confrontare la dimensione massima stimata con quella necessaria desunta dai dati del programma, avvertendo l'utente di modificare la dimensione dei vettori se necessario. Ad esempio, nell'esempio seguente dimensioniamo il numero dei dati in ingresso a 100 tramite una costante `Ndim`, quindi chiediamo di inserire da tastiera il numero di dati in ingresso e lo confrontiamo con `Ndim`: se risulta maggiore, il programma si ferma (nota l'uso dell'istruzione `STOP` dentro l'`IF`!) e dice di modificare la costante `Ndim` e ricompilare e rieseguire il programma. Ad esempio:

```
      PROGRAM ESEMPIO
      IMPLICIT NONE
      INTEGER Ndim, Ndati
      PARAMETER (Ndim=100)
      REAL X(Ndim)

      PRINT *,'Inserisci il numero di dati in ingresso:'
      READ(5,*) Ndati
      IF (Ndati.GT.Ndim) THEN
         PRINT *,'Attenzione! Il numero di dati in ingresso:'
         PRINT *,'Ndati = ',Ndati,'    e'' maggiore'
         PRINT *,'della dimensione massima consentita:'
         PRINT *,'Ndim = ',Ndim
         PRINT *,'Cambiare il valore della costante'
         PRINT *,'Ndim, quindi ricompilare e '
         PRINT *,'rieseguire il programma...'
         STOP
      ELSE
         Legge i dati...
      ENDIF
      ecc.
      STOP
      END
```

Negli esempi che studieremo, a meno che non sia espressamente indicato, supporremo sempre che le dimensioni dei vettori siano sufficienti a contenere i dati.

La potenza dei vettori si comprende dal fatto che, essendo indicizzati tramite una variabile, essi possono facilmente essere utilizzati efficacemente nei cicli. Ad esempio, il seguente programma calcola la media e la varianza di un insieme di dati immesso da tastiera:

```
      PROGRAM STATISTICA
      IMPLICIT NONE
      REAL X(100), XMEDIO, VARIANZA
      INTEGER NDATI,I

      WRITE(6,*) 'Inserire il numero di dati in ingresso'
      WRITE(6,*) '(Ndati<100):'
      READ(5,*) NDATI
C - Legge i dati in ingresso
      DO I=1,NDATI
```

```
         READ(5,*) X(I)
      ENDDO
C - Calcola la somma, poi la media, dei dati in ingresso
      XMEDIO=0.0
      DO I=1,NDATI
         XMEDIO=XMEDIO+X(I)
      ENDDO
      XMEDIO=XMEDIO/NDATI
C - Calcola la somma degli scarti al quadrato,
C   quindi la varianza
      VARIANZA=0.0
      DO I=1,NDATI
         VARIANZA=VARIANZA+(X(I)-XMEDIO)**2.0
      ENDDO
      VARIANZA=VARIANZA/NDATI
      STOP
      END
```

Come si vede, una volta immessa la sequenza dei dati in ingresso ed averla memorizzata nel vettore `X`, i calcoli divengono estremamente semplici.

Talvolta puó essere utile avere degli indici del vettore che assumano valori che non partono da 1 o addirittura negativi. Se si vuole indicizzare un vettore `A` con indici da -10 a 20, ad esempio, si può utilizzare la dichiarazione: `DIMENSION A(-10:20)`.

Come spiegato in precedenza, può a volte essere necessario utilizzare variabili che rappresentano matrici, cioè enti a due indici, o a più dimensioni. In questo caso, si possono dimensionare enti a più indici con la stessa istruzione `DIMENSION`, come nell'esempio seguente:

```
REAL CC,ALFA
DIMENSION CC(-5:5,20,-10:10),ALFA(5,5,5,5)
```

che dichiara `CC` come un ente a 3 indici, i cui valori vanno da -5 a +5 il primo, da 1 fino a 20 il secondo, da -10 a +10 il terzo, quindi un ente a quattro indici, `ALFA`, i cui indici vanno da 1 a 5 in tutti i casi. Si possono definire, secondo lo standard del linguaggio, enti che possiedono fino ad un massimo di 7 indici. Notiamo infine però che la memoria occupata da tali enti è uguale al prodotto delle dimensioni per ogni indice, cioé, ad esempio, la variabile `CC` contiene: 11x20x21=4620 elementi ciascuno memorizzato come una sequenza di 4 bytes (perché si tratta di un numero reale), mentre la variabile `ALFA` contiene 625 elementi, ciascuno a sua volta costituito da 4 bytes. Quindi se si hanno molti indici e si assegna a ciascun indice una dimensione abbastanza grande, si

rischia di saturare la memoria della macchina. Bisogna quindi fare attenzione a limitare il numero delle dimensioni negli enti a più indici a quelle davvero indispensabili.

2.14 Subroutines e funzioni

I computers eseguono operazioni matematiche (e non solo!), spesso ripetendo operazioni simili su medesimi enti, al variare di alcuni parametri. Cioé spesso ci si trova nella condizione di poter distinguere, all'interno di un programma, alcune operazioni di "routine" che debbono essere eseguite al variare di certi parametri. Risulta conveniente, in tal caso, isolare il pezzo di programma che esegue tali operazioni in una entità a sé stante che possa essere "richiamata" dal programma all'occorrenza, passandogli i parametri che servono. Le *subroutines* (anche chiamate "sottoprogrammi" o "procedure") e le *funzioni* servono a questo scopo. Ad esempio, supposto di aver scritto un programma che calcola l'integrale definito di una funzione tra due estremi a e b, avendo messo i valori della funzione integranda in un vettore `f` di dimensione `Ndim`, si può pensare di isolare il pezzo di programma che calcola l'integrale in una entità a parte, una subroutine o una funzione che accetti come parametri la funzione integranda (cioé il vettore `f`), gli estremi di integrazione, quindi restituisca il risultato. Tale operazione è utile, sia per mantenere un certo ordine logico nella programmazione, isolando le parti del codice che effettuano le diverse operazioni in entità separate, sia perché le subroutines o le funzioni, una volta astratte dal resto del codice, possono essere inserite in altri programmi che effettuano simili operazioni, con uno sforzo minimo. Addirittura esistono diverse collezioni di subroutines e funzioni di varia natura, per lo più numeriche, riunite in genere in *librerie*, che possono essere compilate insieme ad un programma, risparmiandosi così la fatica di riscriverle da zero.

Nel seguito utilizzeremo il più possibile sia le subroutines che le funzioni. La differenza principale tra le due sta nel fatto che le funzioni restituiscono un valore (necessariamente scalare) che dipenderà dai parametri della funzione, mentre le subroutines non possono ritornare un valore direttamente. Tuttavia possono farlo in maniera indiretta se il valore di ritorno è passato anch'esso alla subroutine come un parametro. Ad esempio, il pezzo di programma che calcola l'integrale definito, è conveniente scriverlo come funzione, in quanto deve ritornare il valore dell'integrale, che è una semplice quantità scalare. Invece, dovendo ad esempio calcolare l'inversa di una matrice data, converrà utilizzare una subroutine, in quanto le funzioni in `FORTRAN 77` (a differenza di altri linguaggi di programmazione) non consentono di ritornare vettori o matrici.

Il formato di una subroutine è il seguente:

```
SUBROUTINE nome_subroutine(param1, param2, ...)
IMPLICIT NONE
Dichiarazione costanti
Dichiarazione parametri variabili param1, param2, ecc.
Dichiarazione variabili locali
PARAMETER (...)

Istruzioni

RETURN
END
```

Come si vede, una subroutine assomiglia in tutto e per tutto ad un programma standard. È costituito da una *parte dichiarativa*, in cui vengono specificate: eventuali costanti, le variabili che vengono passate durante l'invocazione della subroutine (*parametri della subroutine*) e le variabili *locali* cioé quelle variabili che vengono utilizzate all'interno della subroutine e non sono passate tra i parametri. Quindi seguono le istruzioni. Infine, la coppia di istruzioni `RETURN-END` che chiude la subroutine e che sono gli analoghi della coppia `STOP-END` che chiude un programma. Come nel caso dell'istruzione `STOP`, si può scegliere di terminare una subroutine prima della sua fine naturale, ad esempio se è soddisfatta una condizione particolare, nel qual caso `RETURN` può trovarsi fra le istruzioni.

Analogamente, una funzione che ritorna un valore `VAL` ha la struttura seguente:

```
tipo_VAL FUNCTION VAL(param1, param2, ...)
IMPLICIT NONE
Dichiarazione tipo VAL
Dichiarazione costanti
Dichiarazione parametri variabili param1, param2, ecc.
Dichiarazione variabili locali
PARAMETER (...)

Istruzioni
VAL=...

RETURN
END
```

cioé all'interno della FUNCTION deve sempre trovarsi una variabile che ha lo stesso nome della funzione stessa (VAL nel nostro caso). Notiamo che il nome della subroutine deve essere dichiarato col suo tipo corretto, come se fosse una variabile, sia all'interno della subroutine stessa che nel programma principale.

Tutte le variabili che sono "locali" alla subroutine, cioé che sono indicate nella parte dichiarativa ma non sono passate tra i parametri, assumono valori che non dipendono in alcun modo dal programma principale. Quindi, ad esempio, se nel programma è dichiarata una variabile VAR e una variabile con lo stesso nome è dichiarata all'interno della subroutine, le due variabili non hanno nulla in comune tra loro, in quanto dichiarate in entità diverse del programma. Di conseguenza, per far si che una subroutine possa cambiare il valore di una variabile, tale variabile deve essere passata alla subroutine come parametro.

Esiste un altro modo ottenere lo stesso risultato, vale a dire dichiarare una variabile come *globale*, tramite l'istruzione COMMON. Una variabile globale è un variabile dichiarata nel programma principale (o in una subroutine) e il cui valore può essere conosciuto o cambiato da un'altra subroutine senza che sia espressamente passato tra i parametri. Per fare questo, bisognerà dichiarare la variabile che si vuole sia globale, sia nella parte dichiarativa del programma che della subroutine, e indicare che quella variabile è in comune fra il programma e la subroutine tramite il comando COMMON. Ad esempio, se VAR è la variabile che si vuole sia globale, avremo:

```
PROGRAM GLOBALE
IMPLICIT NONE
REAL VAR
COMMON VAR

Istruzioni...
STOP
END

SUBROUTINE TEST(...)
IMPLICIT NONE
REAL VAR
COMMON VAR

Istruzioni...
STOP
END
```

Esistono forme più complesse dell'istruzione `COMMON`, che permettono essenzialmente di raggruppare le variabili assegnandogli una etichetta, ecc., ma per un uso semplice, la procedura descritta funziona senz'altro.

Sottolineiamo il fatto che i parametri delle subroutines o delle funzioni non devono necessariamente avere lo stesso nome delle variabili che li rappresentano nell'invocazione della subroutine o funzione. Tuttavia, i **tipi** delle variabili debbono essere identici, così come il numero delle variabili passate deve essere lo stesso di quello dichiarato nella subroutine o funzione.

Notiamo infine che, come tutti i linguaggi di programmazione, esistono nel `FORTRAN 77`molte funzioni predefinite che consentono il calcolo veloce di molte espressioni matematiche o operazioni sulle stringhe di testo o altre operazioni di diversa natura. Tali funzioni sono dette `FUNZIONI INTRINSECHE` ed accettano valori che possono essere di tipo diverso, restituendo altri valori di tipo differente. Ad esempio, la funzione `SIN(x)` restituisce il valore reale semplice della funzione seno della variabile x reale, mentre `DSIN(x)` restituisce il valore del seno in doppia precisione, a dove x è anch'esso in doppia precisione. Ovviamente sono permesse diverse combinazioni di nomi, che dipendono sia dal tipo delle variabili argomento, sia dal tipo del risultato. Qui diamo, sotto forma di tabella, le funzioni intrinseche di uso più comune e che coinvolgono il tipo reale e intero o stringa. Per varianti relativi ai differenti tipi, rimandiamo ad un buon manuale di `FORTRAN 77`. Le seguenti, sono le funzioni intrinseche di uso comune:

- `I=INT(R)`: converte la variabile reale `R` in intero (troncamento);
- `R=REAL(I)`: converte la variabile intera `I` in reale a singola precisione;
- `I=MIN(I1,I2,...)`: ad `I` (intero) viene assegnato il valore minimo tra `I1`, `I2`, ecc.;
- `I=MAX(I1,I2,...)`: come sopra, ma assegna il valore massimo;
- `R=SQRT(V)`: assegna ad `R` il valore della radice quadrata della variabile `V`;
- `R=EXP(V)`: assegna ad `R` l'esponenziale della variabile `V`;
- `R=ALOG(V)`: assegna ad `R` il valore del logaritmo naturale di `V`;
- `R=ALOG10(V)`: assegna ad `R` il valore del logaritmo decimale di `V`;
- `R=SIN(X)`: assegna ad `R` il valore del seno della variabile `X`;
- `R=COS(X)`: assegna ad `R` il valore del coseno della variabile `X`;

- `R=TAN(X)`: assegna ad `R` il valore della tangente della variabile `X`;
- `R=ASIN(X)`: assegna ad `R` il valore dell'arco seno della variabile `X`;
- `R=ACOS(X)`: assegna ad `R` il valore dell'arco coseno della variabile `X`;
- `R=ATAN(X)`: assegna ad `R` il valore dell'arco tangente della variabile `X`;
- `R=SINH(X)`: assegna ad `R` il valore del seno iperbolico della variabile `X`;
- `R=COSH(X)`: assegna ad `R` il valore del coseno iperbolico della variabile `X`;
- `R=TANH(X)`: assegna ad `R` il valore della tangente iperbolica della variabile `X`;
- `R=ABS(X)`: assegna ad `R` il valore assoluto della variabile `X`;
- `L=LEN(STR)`: assegna ad `L` (intero) il valore della lunghezza della variabile stringa `STR`;

2.15 Conclusioni

Abbiamo quindi esaminato le istruzioni e i comandi del `FORTRAN 77` che utilizzeremo intensivamente all'interno dei programmi di esempio che vedremo nel seguito. Ovviamente non pretendiamo di aver illustrato né in dettaglio e né in maniera completa tutte le istruzioni del linguaggio. Per ulteriori approfondimenti ed informazioni, raccomandiamo la lettura di uno dei molti manuali di `FORTRAN 77` disponibili sul mercato.

2.16 Un codice per lo schema di Runge–Kutta al secondo ordine

Presentiamo adesso, a titolo di esempio delle nozioni sul `FORTRAN 77`appena esposte, un sottoprogramma per la soluzione di un sistema di equazioni differenziali ordinarie che usa lo schema di Runge–Kutta al secondo ordine. Questo programma verrá usato ampiamente in seguito per la soluzione di specifici problemi di dinamica classica. Chiameremo RK2 il sottoprogramma, che é listato di seguito.

```
      subroutine RK2(Neq,t,Dt,y,dy)
      dimension y(Neq),dy(Neq),ys(Neq)
c
c
c  Calcolo del secondo membro all'istante t
c
      call dery(Neq,t,y,dy)
c
c  Definizione delle variabili ausiliarie
c
      do j=1,Neq
        ys(j)=y(j)+0.5*Dt*dy(j)
      end do
      ts=t+0.5*Dt
c
c  Calcolo del secondo membro all'istante t*
c
      call dery(Neq,ts,ys,dy)
c
c  Calcolo dello step Runge-Kutta
c
      do j=1,Neq
        y(j)=y(j)+Dt*dy(j)
      end do
c
      return
      end
```

In questo sottoprogramma Neq é il numero di equazioni del sistema da risolvere (in input), t é l'istante di tempo t_n (in input), Dt é lo step temporale (in input), y é contemporaneamente il vettore delle incognite al tempo t (in input), ed al tempo t+Dt (in output), mentre dy é il vettore dei secondi membri del sistema (in output). La subroutine dery(Neq,t,y,dy) é fornita dall'esterno e calcola il secondo membro del sistema di equazioni da risolvere, ossia fornisce il vettore dy(j) in output. Finalmente ys(j) é un vettore ausiliario. E' facilmente riconoscibile nella subroutine la struttura dello schema di Runge–Kutta con i due calcoli del secondo membro delle equazioni.

2.17 Un programma per la generazione di numeri aleatori

Riportiamo qui una funzione[2] che genera numeri aleatori con distribuzione uniforme nell'intervallo $[0, 1]$, a partire da un seme idum che rappresenta un numero intero qualsiasi. Questa funzione é utile nei programmi di calcolo Monte Carlo.

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       FUNCTION ran3(idum)
       INTEGER idum
       INTEGER MBIG,MSEED,MZ
C      REAL MBIG,MSEED,MZ
       DOUBLE PRECISION ran3,FAC
       PARAMETER (MBIG=1000000000,MSEED=161803398,MZ=0,FAC=1./MBIG)
C      PARAMETER (MBIG=4000000.,MSEED=1618033.,MZ=0.,FAC=1./MBIG)
       INTEGER i,iff,ii,inext,inextp,k
       INTEGER mj,mk,ma(55)
C      REAL mj,mk,ma(55)
       SAVE iff,inext,inextp,ma
       DATA iff /0/
       if(idum.lt.0.or.iff.eq.0)then
         iff=1
         mj=MSEED-iabs(idum)
         mj=mod(mj,MBIG)
         ma(55)=mj
         mk=1
         do 11 i=1,54
           ii=mod(21*i,55)
           ma(ii)=mk
           mk=mj-mk
           if(mk.lt.MZ)mk=mk+MBIG
           mj=ma(ii)
 11      continue
         do 13 k=1,4
           do 12 i=1,55
             ma(i)=ma(i)-ma(1+mod(i+30,55))
             if(ma(i).lt.MZ)ma(i)=ma(i)+MBIG
 12        continue
```

[2]La function RAN3(k) é stata ricopiata da Numerical Recipes.

```
13      continue
        inext=0
        inextp=31
        idum=1
      endif
      inext=inext+1
      if(inext.eq.56)inext=1
      inextp=inextp+1
      if(inextp.eq.56)inextp=1
      mj=ma(inext)-ma(inextp)
      if(mj.lt.MZ)mj=mj+MBIG
      ma(inext)=mj
      ran3=mj*FAC
      return
      END
```

Proviamo a generare un certo numero N di numeri aleatori, per esempio tramite un semplice programma che richiama la funzione RAN3(idum)

```
program random
dimension r(100000)
open(unit=11,file='random.dat',status='unknown')
idum=123456
N=5
do j=1,N
  r(j)=ran3(idum)
  write(11,*) j,r(j)
end do
close(11)
stop
end
```

Il programma genera $N = 5$ numeri aleatori, come si vede nella tabella seguente che riproduce l'elenco del file *random.dat*. Possiamo generare $N = 1000$ numeri aleatori e calcolare la funzione di distribuzione, suddividendo l'intervallo $[0, 1]$ in 10 classi R_i $(i = 1, 2, \ldots, 10)$ di ampiezza fissata $\Delta = 0.1$, e calcolando il numero di volte $n(R)$ che r cade in una data classe. Per esempio si puó usare il programma seguente

```
      program random
      dimension r(100000),n(100)
      open(unit=11,file='random.dat',status='unknown')
c
      idum=123456
      Np=1000
      do j=1,Np
        r(j)=ran3(idum)
      end do
c
      D=0.1
      do i=1,10
        rinf=(i-1)*D
        rsup=i*D
        n(i)=0          ! Inizializzazione
        do j=1,Np
          if(r(j).ge.rinf .and. r(j).lt.rsup) then
            n(i)=n(i)+1
          end if
        end do
        write(11,*) i,n(i)
      end do
      close(11)
c
      stop
      end
```

Facendo un grafico di $n(R)$ in funzione delle classi R, si vede che essa é uniforme, a meno di errori statistici dell'ordine di $\sqrt{n}$.

Parte III

Alcuni problemi risolti e commentati di meccanica classica

Capitolo 3

Problemi di dinamica classica

3.1 Introduzione

Come primo esempio di applicazione dei concetti di base studiati nella prima parte del libro, iniziamo lo studio pratico di alcuni problemi che si incontrano sovente nello studio della fisica.

Alcuni di questi problemi, come vedremo, sono molto semplici e se ne conosce la soluzione analitica. Quindi è lecito chiedersi se valga la pena preoccuparsi di approfondire gli aspetti del calcolo numerico applicato, vale a dire cercare una soluzione approssimata utilizzando gli strumenti dell'analisi numerica, a tali problemi quando se ne conosce la soluzione esatta. La risposta è senz'altro positiva, per una serie di motivi.

Il primo motivo è che lo studio di problemi di cui si conosca la soluzione analitica consente di comprendere meglio quello che accade nei casi in cui tale soluzione non è, invece, nota. Il secondo motivo, a sua volta estremamente importante, è costituito dalla possibilità che tali problemi ci offrono di confrontare la soluzione numerica con quella esatta, stimando gli errori che si commettono ed evidenziando i vantaggi e gli svantaggi di uno schema rispetto ad un altro dal punto di vista della precisione, stabilità, ecc.. Infine, non per tutti i problemi che verranno illustrati la soluzione analitica è nota. Questo vale specialmente per problemi che richiedono la soluzione di una equazione differenziale non lineare. Tuttavia, anche in questi casi, il confronto con la soluzione analitica dell'equazione linearizzata fornisce il test più efficace per valutare l'efficacia e l'affidabilità dello schema numerico utilizzato.

3.2 Punto materiale in movimento in un fluido viscoso

Iniziamo lo studio di una equazione del tipo:

$$\frac{dv}{dt} = -kv \tag{3.1}$$

Tale equazione è stata più volte presa a modello per analizzare le proprietà di precisione e di stabilità dei vari schemi introdotti nella parte teorica. In questo capitolo analizzeremo la (3.1) come il modello di una equazione fisica da risolvere, confrontando la soluzione analitica nota con la soluzione numerica ottenuta tramite schemi di tipo diverso. Questo ci fornirà delle indicazioni sulle caratteristiche di precisione e stabilità di tali schemi e sulla loro valutazione tramite esempi pratici diretti.

Una equazione del tipo (3.1) si incontra nello studio di molti problemi di fisica. Un esempio tipico è costituito dal moto di un punto materiale all'interno di un fluido con attrito. Supponiamo di avere un punto materiale di massa m che si muove all'interno di un fluido viscoso. Si osserva sperimentalmente che la resistenza che il fluido oppone al moto del punto materiale è proporzionale, tramite un coefficiente $\mu > 0$ all velocità con cui il fluido si muove, cioé la forza agente sulla particella, per effetto della presenza del fluido è data (supposto che il moto avvenga in una direzione fissata, in modo da ridursi ad una singola equazione scalare) da:

$$F = -\mu v$$

dove v è la velocità del punto ed il segno $-$ indica il fatto che la forza si oppone sempre al moto. Dalla legge di Newton si ha:

$$F = ma = m\frac{dv}{dt}$$

e posto: $k = \mu/m$, che rappresenta una quantità sempre positiva, si arriva ad una equazione del tipo (3.1).

Di tale equazione si può trovare banalmente la soluzione analitica. Supposto di avere fissato come condizione iniziale:

$$v(t = 0) = V_0$$

la soluzione analitica dell'equazione è data da:

$$v(t) = V_0 e^{-kt} \tag{3.2}$$

come si verifica facilmente sostituendo la soluzione direttamente nell'equazione (3.1).

Notiamo, come si vede immediatamente sia dalla (3.1) che dalla sua soluzione (3.2), che il parametro k ha le dimensioni dell'inverso di un tempo. La soluzione rappresenta uno smorzamento esponenziale nel tempo della velocità iniziale. Dopo un tempo $\tau = 1/k$, la velocità iniziale si sarà ridotta di un fattore $e^{-1} \sim 0.37$, cioé avrà un valore che è circa il 37% di quello iniziale. Dopo un tempo 2τ la velocità sarà smorzata di un fattore e^{-2}, cioè si sarà ridotta a circa il 14% rispetto al valore iniziale, ecc. Quindi in sostanza il parametro $\tau = 1/k$ rappresenta un *tempo caratteristico* per il fenomeno, cioé ci dà un'idea del tempo trascorso, rispetto all'istante iniziale, perchè la soluzione sia smorzata di un certo fattore. Come vedremo fra breve, questa interpretazione fisica del parametro k fornisce una giustificazione intuitiva della condizione di stabilità degli schemi numerici che andremo a studiare.

Calcoliamo ora la soluzione numerica dell'equazione (3.1) con alcuni dei metodi illustrati nella parte teorica del libro.

3.2.1 Metodo Eulero Forward

Come visto nella prima parte del testo, una soluzione semplice di una equazione tipo la (3.1) si può ottenere discretizzando la variabile indipendente continua t, nell'intervallo tra l'istante iniziale e un generico istante T_f, tramite un insieme discreto di N tempi: $t_n = n\Delta t$, con $n = 0, \ldots, N$ e $\Delta t = T_f/N$. Si cercherà la soluzione dell'equazione non per qualsiasi istante di tempo t, ma solo per alcuni valori discreti t_n. L'equazione (3.1) si scriverà quindi in forma discreta, come:

$$\left|\frac{dv}{dt}\right|_{t=t_n} = -kv_n \tag{3.3}$$

dove abbiamo indicato per brevità con v_n il valore della soluzione v al tempo discreto t_n: $v_n = v(t_n)$.

La soluzione a ciascun istante t_n è quindi ottenuta approssimando la derivata prima della funzione incognita (v nel nostro caso) nella (3.3), tramite una opportuna espressione della derivata ottenuta con un metodo alle differenze finite. Per esempio, approssimando la derivata tramite il rapporto incrementale della funzione "in avanti" (*forward*), si ottiene:

$$\left|\frac{dv}{dt}\right|_{t=t_n} = \frac{v_{n+1} - v_n}{\Delta t} + O(\Delta t).$$

Il termine $O(\Delta t)$ indica il fatto che in tale approssimazione si commette un errore di ordine Δt. Lo schema di Eulero "in avanti" (*forward*) si traduce

quindi nell'espressione:

$$\frac{v_{n+1} - v_n}{\Delta t} = -kv_n$$

dove l'uguaglianza vale a meno di termini di ordine Δt. A questo punto si può ottenere l'espressione che permette, nota la soluzione al tempo t_n, di calcolare la soluzione al passo successivo (cioé al tempo t_{n+1}):

$$v_{n+1} = v_n(1 - k\Delta t). \tag{3.4}$$

Quindi, nota la condizione iniziale: $v_0 = v(t_0) = V_0$, si può calcolare la soluzione al passo t_1 tramite la (3.4): $v_1 = V_0(1 - k\Delta t)$; quindi la $v_2 = v_1(1 - k\Delta t)$, ecc. fino all'ultimo istante temporale $t_N = N\Delta t = T_f$.

Il codice seguente risolve l'equazione (3.1) implementando uno schema del tipo (3.4):

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione:
C dv/dt = -kv
C con uno schema di Eulero forward.
C L'output del programma si trova nel file "eulf.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica, la terza colonna la soluzione esatta.
C Infine, la quarta colonna rappresenta il valore assoluto della differenza
C tra la soluzione esatta e quella numerica, vale a dire l'errore numerico.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM eulero_forward

C - Parte dichiarativa

      IMPLICIT NONE

      REAL k,t,dt,Tfin,Tprint,V0
      REAL v_num,v_ex,v_diff
      INTEGER n,Nsteps,Nprints

C - Main program

C - Immissione dati iniziali
```

```
      PRINT *,'Inserire il valore di k (k>0)'
      READ (5,*) k
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Vo:'
      READ (5,*) V0

C - Definizione variabili utilizzate nel programma

      t=0.0                    ! Tempo iniziale
      Nsteps=nint(Tfin/dt)     ! Numero passi temporali totali
      Nprints=nint(Tprint/dt)  ! Stampa ogni Nprints passi temporali
      v_num=V0                 ! A t=0 la soluzione e' data
      v_ex=V0                  ! dalla condizione iniziale
      v_diff=0.0

C - Apertura file per la scrittura della soluzione

      OPEN (FILE='eulf.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,v_num,v_ex,v_diff

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         v_num=v_num*(1.0-k*dt)
         v_ex=V0*exp(-k*t)
         v_diff=abs(v_num-v_ex)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,v_num,v_ex,v_diff
         END IF
      END DO

C - Chiusura file di output e fine programma

      CLOSE (UNIT=16)
      STOP
```

```
END
```

Il programma riceve in input il valore del parametro k, il tempo finale T_f dell'integrazione, il passo temporale Δt, l'intervallo di tempo tra le uscite da stampare (questo evita di stampare il risultato ad ogni passo temporale, ottenendo file che occupano troppo spazio inutilmente!) e la condizione iniziale V_0. Quindi calcola la soluzione numerica approssimata tramite uno schema Eulero forward, la soluzione esatta nonché il valore assoluto della differenza fra le due, che fornisce l'errore nell'integrazione. Il tempo, oltre alle tre quantità appena citate, vengono trascritti dal programma nel file `eulf.dat`.

Per provare il funzionamento del codice si possono utilizzare i seguenti valori dei parametri di input:

Esempio 1

```
k = 1.0; Tfin = 2.0; dt = 0.001; Tprint = 0.1; V0 = 1.0.
```

Questi parametri produrranno `Tfin/Tprint`=2.0/0.1=20 stampe, corrispondenti agli istanti discreti $t_1 = 0.1$, $t_2 = 0.2$, ..., $t_{20} = 2.0$, oltre alla condizione iniziale corrispondente a $t_0 = 0.0$.

Per avere un'idea più precisa dell'andamento della soluzione, si può utilizzare il programma `gnuplot` per tracciare un grafico della soluzione numerica e della soluzione esatta in funzione del tempo. Il risultato di tale operazione dovrebbe essere simile a quello di fig. 3.2.1a, dove le croci rappresentano la soluzione numerica e la curva la soluzione esatta dell'equazione.

Come si vede dalla figura, la soluzione trovata riproduce abbastanza fedelmente la soluzione esatta. La fig. 3.2.1b rappresenta la differenza (in valore assoluto) tra la soluzione numerica e quella esatta, cioé quello che viene chiamato *errore globale* della soluzione. Il massimo valore di tale errore risulta essere di ordine $\sim 1.9 \cdot 10^{-4}$, che è abbastanza più piccolo dell'errore atteso dallo schema che è di ordine $\Delta t = 10^{-3}$.

Analizziamo la stabilità di questo schema. Secondo il criterio di Von Neumann, lo schema (3.4) risulta essere stabile se vale la condizione:

$$\left| \frac{v_{n+1}}{v_n} \right| \leq 1$$

Dalla (3.4) risulta:

$$\left| \frac{v_{n+1}}{v_n} \right| \leq 1 \Rightarrow |1 - k\Delta t| \leq 1 \Rightarrow \begin{cases} -1 \leq 1 - k\Delta t \\ 1 - k\Delta t \leq 1 \end{cases} \Rightarrow$$

$$\begin{cases} \Delta t \leq 2/k \\ -k\Delta t \leq 0 \quad \text{sempre verificata se } k > 0 \text{ e } \Delta t > 0 \end{cases}$$

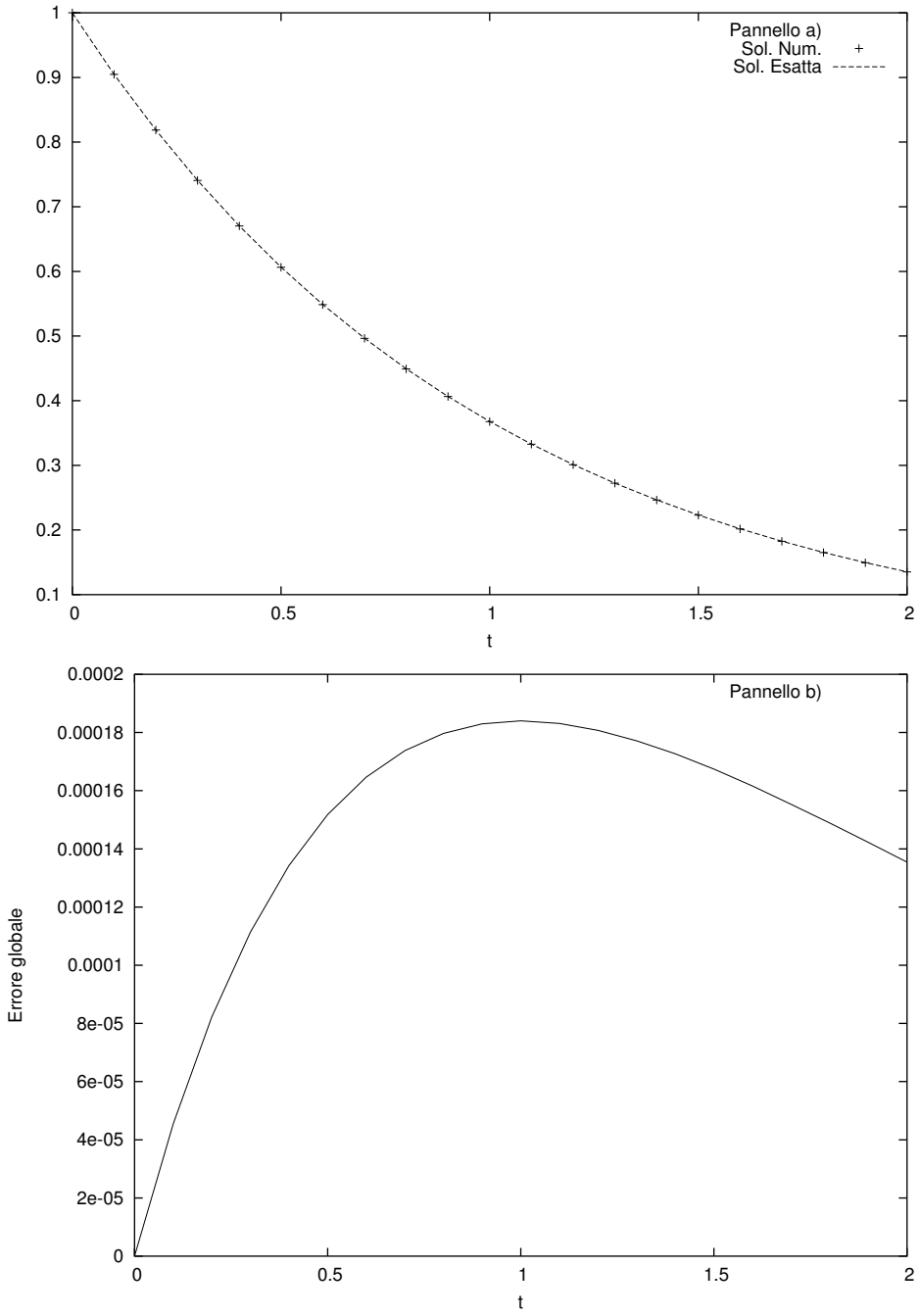


Figura 3.1: Pannello a): Risultati forniti dal codice `eulf.f` utilizzando i parametri di input dell'**Esempio 1**. Pannello b): errore globale per lo stesso caso.

La condizione Von Neumann quindi impone che $\Delta t \leq 2/k$. In teoria, se k è molto grande questa condizione puó essere abbastanza stringente. Tuttavia, quasi sempre in problemi fisici k risulta essere molto piccolo; nel caso del punto materiale in moto in un fluido viscoso, ad esempio, k è proporzionale al coefficiente di viscosità del fluido che è usualmente un numero molto piccolo.

Notiamo che la condizione: $\Delta t \leq 2/k$ assume un significato fisico notevole quando si tenga conto dell'interpretazione del parametro $\tau = 1/k$ data all'inizio del capitolo. Si era detto che τ rappresenta il *tempo caratteristico* del fenomeno in esame, cioé dello smorzamento della velocità iniziale. La condizione di stabilità: $\Delta t \leq 2/k$ può quindi scriversi anche come: $\Delta t \leq 2\tau$. Questo vuol dire che, quando stiamo studiando un fenomeno che evolve su un tempo caratteristico τ, dobbiamo utilizzare un passo temporale dell'ordine di o inferiore a τ per poterlo descrivere correttamente, altrimenti incorriamo in una "perdita di informazioni" che produce instabilità.

Tanto per fare un esempio di natura diversa, se si vuol misurare lo spessore di uno spago largo un millimetro, non si può utilizzare un righello graduato al decimetro, in quanto la stima dello spessore dello spago sarebbe completamente priva di senso. Dovrei usare almeno un righello graduato al millimetro, meglio ancora se con una graduazione più fine, per poter risolvere lo spessore in maniera opportuna. Allo stesso modo, studiando un fenomeno che evolve su un tempo scala τ, si debbono utilizzare degli step temporali piú piccoli o confrontabili con τ, altrimenti il risultato non ha senso. Quest'ultima affermazione, dal punto di vista numerico, si traduce non soltanto nel fatto che la soluzione è completamente sbagliata, nel senso che l'errore risulta molto grande, ma anche nella presenza nella soluzione di oscillazioni completamente artificiali che crescono col passare del tempo finché l'ampiezza di tali oscillazioni risulta maggiore della possibilità di rappresentazione dei numeri reali della macchina, producendo un "overflow".

Nel caso dell'**Esempio 1**, la scelta $\Delta t = 10^{-3}$ e $k = 1$ assicura che stiamo lavorando molto addentro al margine di stabilità previsto dalla condizione di Von Neumann: $2/k = 2$. Per dare un esempio di instabilità, mettiamoci ora invece in un caso in cui la condizione di Von Neumann **non** è rispettata, come ad esempio:

Esempio 2

```
k = 20.0; Tfin = 1.0; dt = 0.2; Tprint = 0.2; V0 = 100.0.
```

Il valore di V_0 è stato scelto particolarmente alto perché la soluzione, per $k = 20$ decresce molto velocemente. In questo caso, la condizione di stabilità *non* è rispettata, in quanto $2/k = 0.1$, mentre $\Delta t = 0.2$, quindi $\Delta t > 2/k$!

La fig. 3.2.1 mostra i risultati della simulazione effettuata utilizzando i parametri dell'**Esempio 2**. In questo caso, nel pannello a) della figura,

l'asse y è stato messo in scala logaritmica. In questo caso, la soluzione (esponenziale decrescente) risulta essere una retta, visibile nella figura, senonché i punti che rappresentano la soluzione numerica sono completamente sparsi nel grafico e non c'è nessuna tendenza, per la soluzione numerica, a somigliare seppur lontanamente alla soluzione analitica. Anzi, guardando i dati contenuti nel file, non tutti i punti della soluzione numerica sono visualizzati nel grafico in quanto alcuni valori risultano addirittura negativi e quindi non sono rappresentabili in scala logaritmica. Il pannello b) della figura descrive invece l'errore globale, che come si vede cresce esponenzialmente e assume valori enormi molto velocemente. In sostanza, si ha una soluzione numerica che presenta forti oscillazioni completamente innaturali e viene amplificata esponenzialmente, invece di decrescere esponenzialmente. Questo é l'effetto della instabilità numerica. Se avessimo prolungato l'intervallo di calcolo abbastanza a lungo, il programma non sarebbe nemmeno stato capace di arrivare al termine dell'esecuzione, ma avrebbe, come si suol dire, "abortito" e dato un errore di "overflow".

3.2.2 Schema Eulero backward

Analizziamo ora cosa succede alla nostra equazione nel caso volessimo utilizzare lo schema di Eulero "all'indietro", o *backward*. In questo caso, la derivata viene approssimata nello schema alle differenze finite tramite il rapporto incrementale calcolato col punto indietro, cioé all'istante t_{n-1} invece che con quello in avanti, t_{n+1}, cioé:

$$\left|\frac{dv}{dt}\right|_{t=t_n} = \frac{v_n - v_{n-1}}{\Delta t} + O(\Delta t).$$

Anche in questo caso, l'errore che si commette nell'approssimazione è di ordine Δt. Inserendo la derivata nell'equazione si ha:

$$\frac{v_n - v_{n-1}}{\Delta t} = -kv_n \Rightarrow v_n = \frac{v_{n-1}}{1 + k\Delta t}$$

Per ottenere lo schema numerico, dobbiamo esplicitare la dipendenza di v_{n+1} da v_n. Iterando l'espressione appena trovata, cioé sostituendo $n+1$ a n ed n al posto di $n-1$ si ottiene lo schema Eulero all'indietro:

$$v_{n+1} = \frac{v_n}{1 + k\Delta t} \tag{3.5}$$

Il seguente programma, chiamato `eulb.f` implementa il nuovo schema in una struttura del tutto identica a quella del programma `eulf.f`, l'unica differenza è in pratica nella linea che implementa lo schema numerico. L'output del codice viene scritto in questo caso nel file `eulb.dat`.

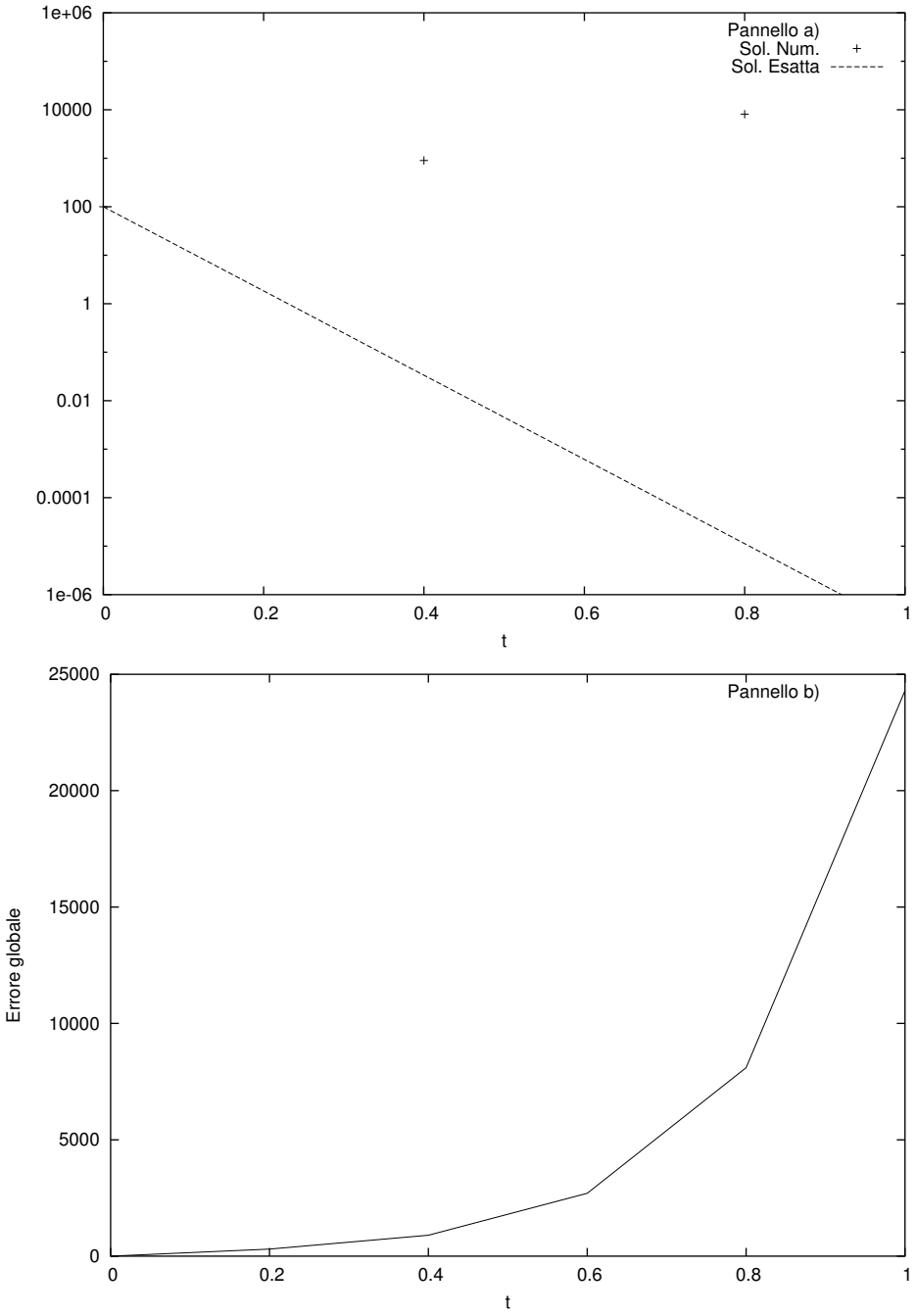


Figura 3.2: Pannello a): Risultati forniti dal codice `eulf.f` utilizzando i parametri di input dell'**Esempio 2**. Pannello b): errore globale per lo stesso caso.

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione:
C dv/dt = -kv
C con uno schema di Eulero backward.
C L'output del programma si trova nel file "eulb.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica, la terza colonna la soluzione esatta.
C Infine, la quarta colonna rappresenta il valore assoluto della differenza
C tra la soluzione esatta e quella numerica, vale a dire l'errore numerico.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM eulero_backward

C - Parte dichiarativa

      IMPLICIT NONE

      REAL k,t,dt,Tfin,Tprint,V0
      REAL v_num,v_ex,v_diff
      INTEGER n,Nsteps,Nprints

C - Main program

C - Immissione dati iniziali

      PRINT *,'Inserire il valore di k (k>0)'
      READ (5,*) k
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Vo:'
      READ (5,*) V0

C - Definizione variabili utilizzate nel programma

      t=0.0                    ! Tempo iniziale
```

```
      Nsteps=nint(Tfin/dt)      ! Numero passi temporali totali
      Nprints=nint(Tprint/dt)   ! Stampa ogni Nprints passi temporali
      v_num=V0                  ! A t=0 la soluzione e' data
      v_ex=V0                   ! dalla condizione iniziale
      v_diff=0.0

C - Apertura file per la scrittura della soluzione

      OPEN (FILE='eulb.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,v_num,v_ex,v_diff

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         v_num=v_num/(1.0+k*dt)
         v_ex=V0*exp(-k*t)
         v_diff=abs(v_num-v_ex)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,v_num,v_ex,v_diff
         END IF
      END DO

C - Chiusura file di output e fine programma

      CLOSE (UNIT=16)
      STOP
      END
```

Inserendo come parametri di input i valori dell'**Esempio 1** si ottengono risultati assolutamente simili a quelli ottenuti con lo schema Eulero "in avanti", sia per quanto concerne l'accordo tra la soluzione numerica e la soluzione esatta, sia per quanto concerne l'entità dell'errore numerico. Tuttavia, il discorso della stabilità dello schema in questo caso assume caratteristiche totalmente differenti. La condizione di Von Neumann per lo schema (3.5) fornisce:

$$\left|\frac{v_{n+1}}{v_n}\right| \leq 1 \Rightarrow \left|\frac{1}{1+k\Delta t}\right| \leq 1$$

ma questa condizione, per $k > 0$ e $\Delta t > 0$, è sempre soddisfatta, quindi lo schema di Eulero "all'indietro", a differenza di quello "in avanti", è incondizionatamente stabile, cioé risulta stabile qualunque sia il passo tem-

porale Δt scelto. Ovviamente non ha senso scegliere dei Δt troppo grandi comunque, in quanto l'errore, proporzionale a Δt, sarebbe talmente grande da falsare la soluzione numerica. Però è possibile, per esempio, notare la differenza con lo schema precedente utilizzando i parametri dell'**Esempio 2**. I risultati della simulazione con questi parametri sono mostrati in fig. 3.2.2.

Come è visibile dal pannello a) della figura, dove l'asse y è posto in scala logaritmica per amplificare le differenze, la soluzione numerica si discosta abbastanza da quella esatta, a causa della scelta di Δt abbastanza grande, ma lo schema non è instabile: la soluzione numerica ha ancora una forma esponenziale (una retta, in scala logaritmica), sebbene con un esponente differente, e non presenta oscillazioni. L'errore globale, graficato nel pannello b) della figura, non cresce esponenzialmente come nel caso dello schema "in avanti", tuttavia presenta un picco abbastanza grande. Questo valore grande è spiegabile per il fatto che l'errore è proporzionale a Δt tramite la derivata seconda della funzione soluzione. La derivata seconda di e^{-kt} dá un termine di ordine k^2 che per $k = 20$ vale $k^2 = 400$. Poiché $\Delta t = 0.2$, il prodotto risulta di ordine 10, all'incirca, il che spiega il massimo valore dell'errore, pari circa a 20.

Per finire, notiamo che lo schema di Eulero backward può anche scriversi nella forma seguente:

$$\frac{v_n - v_{n-1}}{\Delta t} = -kv_n \Rightarrow \frac{v_{n+1} - v_n}{\Delta t} = -kv_{n+1},$$

avendo iterato lo schema di un valore (cioé essendo passati da $n-1$ a n e da n ad $n+1$). Ma questo schema è in effetti identico allo schema Eulero forward, eccetto per il fatto che la funzione a secondo membro è valutata al passo t_{n+1} invece che al passo t_n. Questo è un esempio di implicitazione di uno schema che, come detto nella prima parte del libro, ha sempre effetti stabilizzanti sugli schemi. In effetti, anche in questo caso, non solo l'intervallo di stabilità è più ampio, ma addirittura lo schema diventa sempre stabile.

3.2.3 Schema di Runge-Kutta

Infine, risolviamo l'equazione (3.1) utilizzando un metodo Runge-Kutta al secondo ordine. Lo schema Runge-Kutta fa parte dei metodi multi-step, cioè la variabile incognita al passo successivo viene calcolata dividendo lo step in più parti e valutando la quantità a tali passi intermedi per migliorare la stima dell'incognita. Questo valore viene quindi usato alla fine per calcolare il secondo membro dell'equazione con una precisione maggiore di quanto non si possa fare con uno schema semplice, ad esempio di Eulero. Esistono schemi Runge-Kutta di ordine molto elevato, ma nel nostro caso ci limiteremo ad

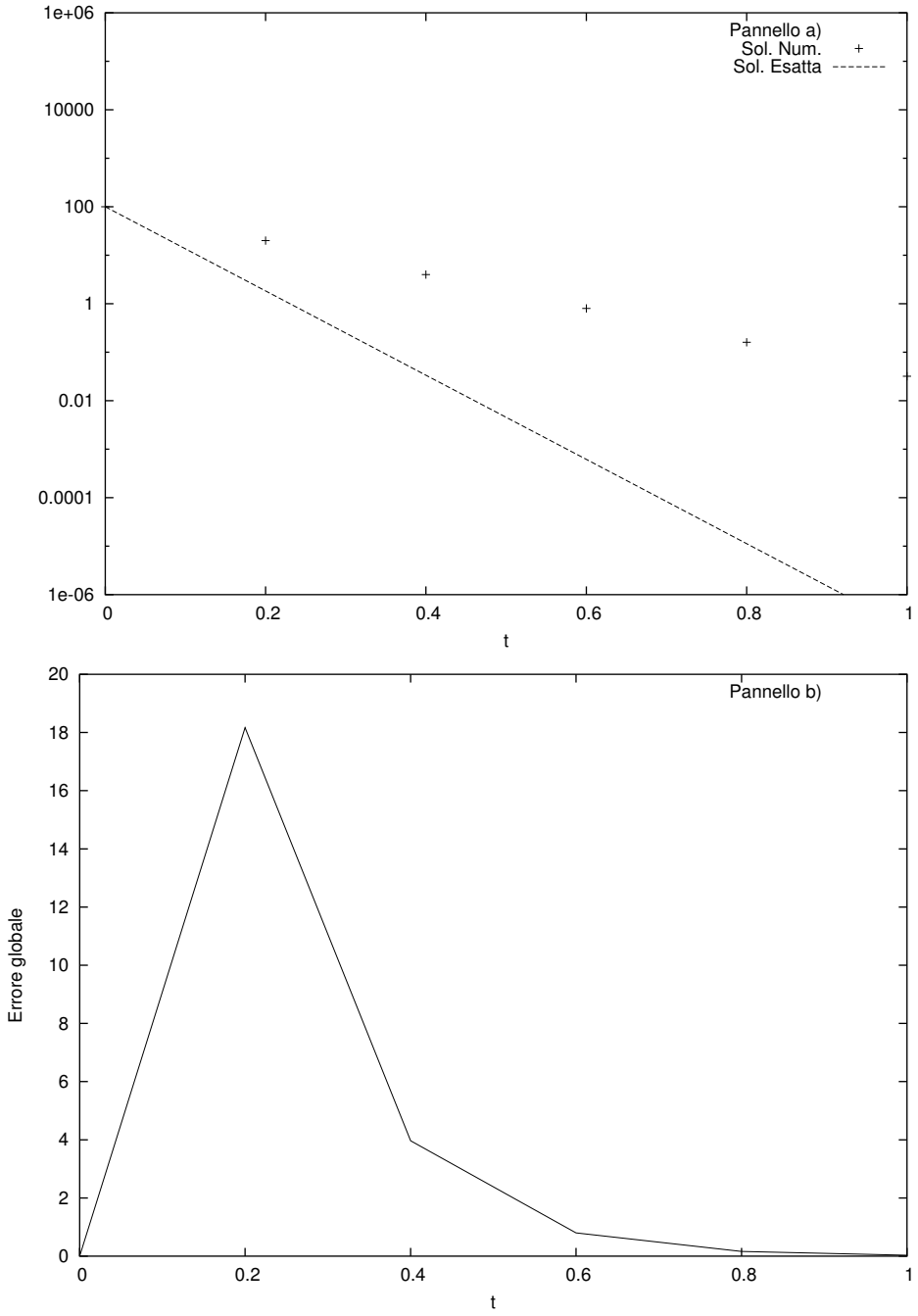


Figura 3.3: Pannello a): Risultati forniti dal codice `eulb.f` utilizzando i parametri di input dell'**Esempio 2**. Pannello b): errore globale per lo stesso caso.

utilizzare uno schema del secondo ordine. In questo caso lo step temporale di integrazione viene diviso a metà, quindi si valuta dall'equazione un valore approssimato della variabile v in questo punto intermedio, diciamo v^*, infine tale valore viene utilizzato per valutare in secondo membro dell'equazione con una precisione più elevata. In pratica:

$$\frac{v^* - v_n}{\Delta t/2} = -kv_n \tag{3.6}$$

$$\frac{v_{n+1} - v_n}{\Delta t} = -kv^* \tag{3.7}$$

Cioé si calcola prima il valore approssimato di v^* alpasso intermedio, quindi tale valore viene utilizzato per calcolare il secondo membro dell'equazione nel passo definitivo da v_n a v_{n+1}.

Il seguente programma `rk2.f`, calcola la soluzione dell'equazione specificata utilizzando lo schema (3.6):

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione:
C dv/dt = -kv
C con uno schema Runge-Kutta del secondo ordine.
C L'output del programma si trova nel file "rk2.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica, la terza colonna la soluzione esatta.
C Infine, la quarta colonna rappresenta il valore assoluto della differenza
C tra la soluzione esatta e quella numerica, vale a dire l'errore numerico.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM Runge_Kutta

C - Parte dichiarativa

      IMPLICIT NONE

      REAL k,t,dt,Tfin,Tprint,V0
      REAL v_num,v_ex,v_diff,v_star
      INTEGER n,Nsteps,Nprints

C - Main program
```

```
C - Immissione dati iniziali

      PRINT *,'Inserire il valore di k (k>0)'
      READ (5,*) k
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Vo:'
      READ (5,*) V0

C - Definizione variabili utilizzate nel programma

      t=0.0                     ! Tempo iniziale
      Nsteps=nint(Tfin/dt)      ! Numero passi temporali totali
      Nprints=nint(Tprint/dt)   ! Stampa ogni Nprints passi temporali
      v_num=V0                  ! A t=0 la soluzione e' data
      v_ex=V0                   ! dalla condizione iniziale
      v_diff=0.0

C - Apertura file per la scrittura della soluzione

      OPEN (FILE='rk2.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,v_num,v_ex,v_diff

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         v_star=v_num-0.5*dt*k*v_num     ! Primo step Runge-Kutta
         v_num=v_num-dt*k*v_star         ! Secondo step Runge-Kutta
         v_ex=V0*exp(-k*t)
         v_diff=abs(v_num-v_ex)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,v_num,v_ex,v_diff
         END IF
      END DO

C - Chiusura file di output e fine programma
```

```
CLOSE (UNIT=16)
STOP
END
```

Nel programma la variabile `v_star` rappresenta l'incognita al passo intermedio v^*. Le quantità vengono stampate sul file di output `rk2.dat` nello stesso formato degli altri files di output.

In fig. 3.2.3 vengono mostrati, nei due pannelli, il valore della soluzione numerica e della soluzione esatta (pannello a)) e dell'errore globale (pannello b)). Come si vede, il massimo dell'errore globale in questo caso vale all'incirca $2.5 \cdot 10^{-7}$, molto più piccolo di quello ottenuto con lo schema di Eulero.

Per assicurarsi della correttezza dei risultati ottenuti, possiamo analizzare come cambia l'errore globale per i tre schemi al variare dello step temporale Δt. La tabella 3.2.3 mostra l'errore globale al tempo finale per i tre schemi studiati per tre differenti valori di Δt; gli altri parametri sono identici a quelli dell'**Esempio 1**. Il valore di Δt viene progressivamente dimezzato. Quindi ci aspettiamo che per gli schemi di tipo Eulero, per i quali l'errore è proporzionale a Δt, l'errore globale dovrebbe dimezzarsi, mentre per lo schema di Runge-Kutta, per il quale l'errore è proporzionale a Δt^2, l'errore dovrebbe diminuire di un fattore 4 ad ogni dimezzamento di Δt. Questo è effettivamente quello che succede: per le prime due colonne, rappresentanti gli schemi Eulero forward e backward, l'errore si dimezza al diminuire di Δt, mentre per la terza colonna, che rappresenta l'errore del Runge-Kutta, l'errore diminuisce di volta in volta di un fattore 4.

3.3 Oscillatore armonico in una dimensione.

Un altro esempio di equazione che si incontra sovente in fisica è l'equazione dell'oscillatore armonico monodimensionale:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0 \tag{3.8}$$

Δt	Eulero forward	Eulero backward	Runge-Kutta
0.1	0.0123	0.0138	4.9×10^{-4}
0.05	0.0067	0.0068	1.2×10^{-4}
0.025	0.0034	0.0034	0.29×10^{-4}

Tabella 3.1: Errore globale al tempo finale della simulazione al variare del passo temporale Δt per i tre schemi studiati: Eulero forward, backward e Runge-Kutta al secondo ordine.

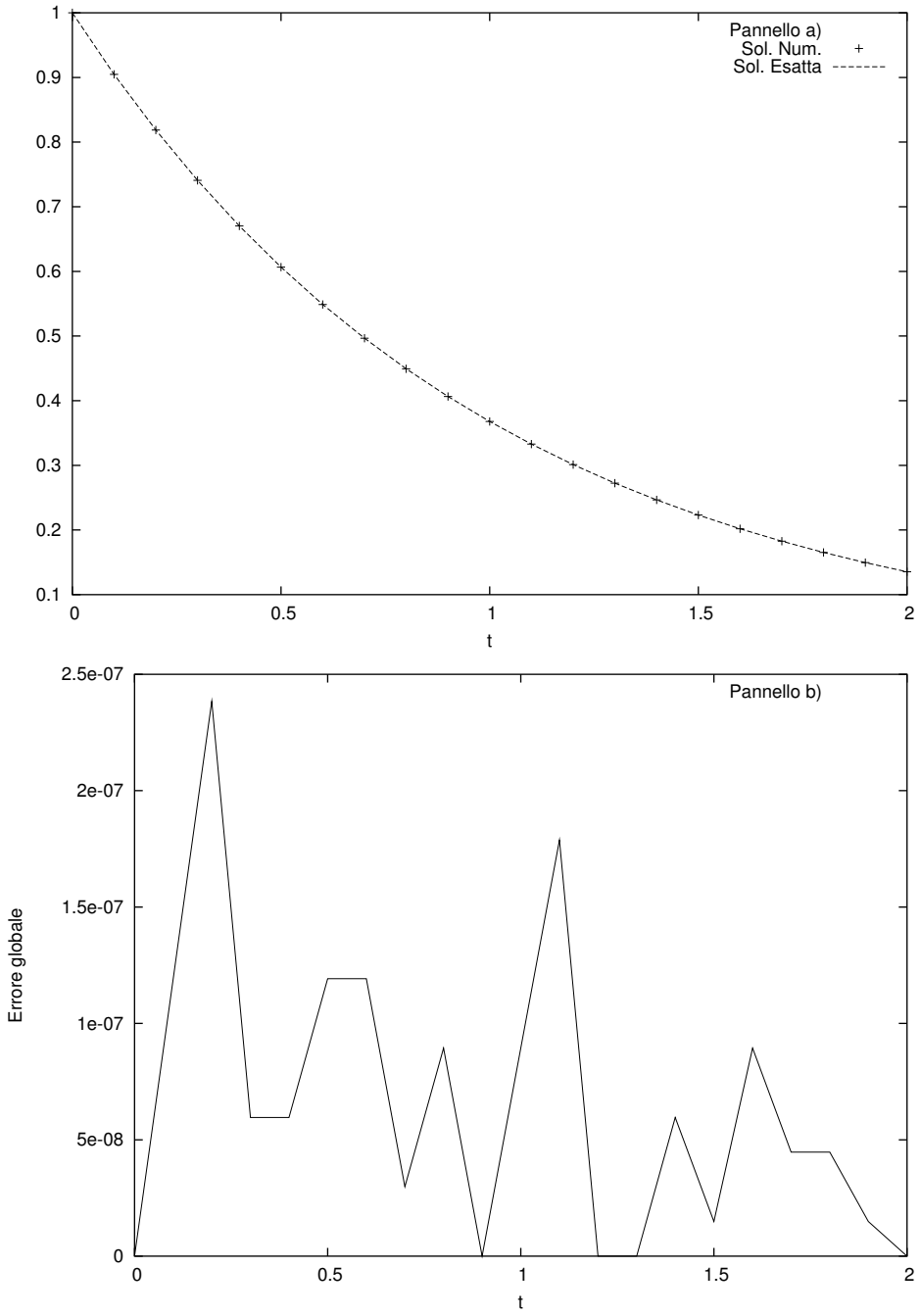


Figura 3.4: Pannello a): Risultati forniti dal codice `rk2.f` utilizzando i parametri di input dell'**Esempio 1**. Pannello b): errore globale per lo stesso caso.

Tale equazione si ottiene, ad esempio, studiando il moto di un punto materiale in moto su un piano orizzontale e collegato all'estremità di una molla. L'altra estremità della molla è tenuta fissa. Scelto un sistema di riferimento con l'origine nel punto di equilibrio della molla e l'asse x lungo la direzione del moto sul piano orizzontale, se m è la massa del punto e k la costante elastica della molla, la legge di Newton fornisce l'equazione:

$$ma = -kx \Rightarrow \frac{d^2x}{dt^2} + \frac{k}{m}x = 0.$$

Poiché sia k che m sono costanti positive, si può senz'altro scrivere: $\omega^2 = k/m$, da cui si ottiene l'equazione (3.8). La soluzione dell'equazione è del tipo:

$$x(t) = A\sin(\omega t + \phi)$$

con A e ϕ costanti da determinare tramite le condizioni iniziali. Possiamo supporre, per esempio, che a $t = 0$ la molla sia nella posizione di equilibrio, cioé $x(t = 0) = 0$, e al punto materiale venga impressa una velocità iniziale fissata $v(t = 0) = V_0$. Con queste condizioni iniziali, la soluzione dell'equazione è data da:

$$x(t) = \frac{V_0}{\omega}\sin(\omega t). \tag{3.9}$$

Al fine di risolvere l'equazione con gli schemi già discussi nell'esempio precedente, bisogna prima di tutto trasformare l'equazione di partenza in un sistema di due equazioni in due incognite. Questo è sempre possibile introducendo la velocitá v del punto materiale come nuova variabile incognita:

$$\begin{cases} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\omega^2 x. \end{cases} \tag{3.10}$$

A questo punto, ognuna delle due equazioni può essere trattata con gli schemi studiati.

3.3.1 Schema di Eulero forward.

Supposto di aver effettuato la solita discretizzazione dell'intervallo temporale $[0, T_f]$ tramite N intervalli discreti di lunghezza Δt, lo schema viene implementato scrivendo la derivata prima delle due quantità incognite x e v come il rapporto incrementale fra il passo temporale t_n e t_{n+1}. Lo schema diventa quindi:

$$\begin{cases} \frac{x_{n+1}-x_n}{\Delta t} &= v_n & \Rightarrow & \quad x_{n+1} = x_n + \Delta t\ v_n \\ \frac{v_{n+1}-v_n}{\Delta t} &= -\omega^2 x_n & \Rightarrow & \quad v_{n+1} = v_n - \omega^2 \Delta t\ x_n \end{cases} \tag{3.11}$$

Il seguente programma implementa la soluzione numerica dell'equazione (3.8) tramite lo schema (3.11). I parametri iniziali richiesti dal programma sono la frequenza ω di oscillazione (identificata con `w` nel codice), oltre ai soliti parametri già visti nell'esempio precedente. La condizione iniziale `V0` si riferisce alla velocità iniziale V_0. Il programma genera in output due files: `eulfx.dat` contenente il tempo, la soluzione numerica, quella esatta e l'errore globale per la posizione x, mentre `eulfv.dat` contiene le analoghe quantità per la velocità. Il listato del codice è il seguente:

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione:
C d^2 x/dt^2 + w^2 x = 0
C con uno schema di Eulero forward.
C L'output del programma si trova nel file "eulfx.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica per lo spazio (x_num), la terza colonna la
C soluzione esatta per lo spazio (x_ex). La quarta colonna rappresenta il
C valore assoluto della differenza tra la soluzione esatta e quella numerica,
C vale a dire l'errore numerico per la x.
C Analogamente, il file "eulfv.dat" contiene le analoghe quantita' per la
C velocita'.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM eulero_forward

C - Parte dichiarativa

      IMPLICIT NONE

      REAL w,w2dt,t,dt,Tfin,Tprint,V0
      REAL x_num,x_ex,v_num,v_ex,x_diff,v_diff
      REAL xnp1,vnp1
      INTEGER n,Nsteps,Nprints

C - Main program

C - Immissione dati iniziali

      PRINT *,'Inserire il valore di w (omega)'
```

```
      READ (5,*) w
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Vo:'
      READ (5,*) V0

C - Definizione variabili utilizzate nel programma

      t=0.0                     ! Tempo iniziale
      Nsteps=nint(Tfin/dt)      ! Numero passi temporali totali
      Nprints=nint(Tprint/dt)   ! Stampa ogni Nprints passi temporali
      x_num=0.0                 ! Condizione iniziale per x:
      x_ex=0.0                  ! x(t=0) = 0.0
      x_diff=0.0
      v_num=V0                  ! A t=0 la soluzione  per la velocita'
      v_ex=V0                   ! e' data dalla condizione iniziale
      v_diff=0.0
      w2dt=w*w*dt

C - Apertura files per la scrittura delle soluzioni

      OPEN (FILE='eulfx.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,x_num,x_ex,x_diff
      OPEN (FILE='eulfv.dat',UNIT=17,STATUS='UNKNOWN')
      WRITE(17,*) t,v_num,v_ex,v_diff

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         xnp1=x_num+dt*v_num
         vnp1=v_num-w2dt*x_num
         x_num=xnp1          ! Aggiornamento variabili
         v_num=vnp1
         x_ex=(V0/w)*sin(w*t)
         v_ex=V0*cos(w*t)
         x_diff=abs(x_num-x_ex)
```

```
         v_diff=abs(v_num-v_ex)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,x_num,x_ex,x_diff
            WRITE(17,*) t,v_num,v_ex,v_diff
         END IF
      END DO

C - Chiusura files di output e fine programma

      CLOSE (UNIT=16)
      CLOSE (UNIT=17)
      STOP
      END
```

Tuttavia ci si accorge facilmente che lo schema di Eulero forward risulta sempre **instabile** per qualsiasi valore di Δt. Al fine di analizzare la stabilità di un sistema di equazioni come in (3.11) il criterio di Von Neumann deve essere applicato nella maniera che segue, come spiegato nella prima parte del libro: lo schema numerico deve essere espresso in forma matriciale, con il vettore delle incognite al passo $n+1$ scritto come il prodotto di una matrice per lo stesso vettore al passo n. Una volta ricavata la matrice del prodotto, essa deve essere diagonalizzata; quindi l'autovalore (che sarà in generale un numero complesso) di modulo massimo deve essere posto minore o uguale a uno. Nel caso specifico:

$$\left\{ \begin{array}{c} x_{n+1} = x_n + \Delta t\ v_n \\ v_{n+1} = v_n - \omega^2 \Delta t\ x_n \end{array} \right. \Rightarrow \left(\begin{array}{c} x_{n+1} \\ v_{n+1} \end{array} \right) = \left(\begin{array}{cc} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{array} \right) \left(\begin{array}{c} x_n \\ v_n \end{array} \right)$$

Diagonalizzando la matrice, si ottiene il polinomio caratteristico:

$$\lambda^2 - 2\lambda + (1 + \omega^2 \Delta t^2) = 0$$

ed i corrispondenti autovalori: $\lambda = 1 \pm i\omega\Delta t$, essendo i l'unità immaginaria. I due autovalori sono complessi coniugati, di conseguenza il loro modulo è identico e vale: $|\lambda| = \sqrt{1 + \omega^2 \Delta t^2}$.

La condizione di Von Neumann afferma che lo schema è stabile se: $|\lambda| \leq 1$. È evidente che tale condizione non è mai soddisfatta in questo caso, di conseguenza lo schema di Eulero forward risulta sempre instabile. Questa instabilità può essere osservata anche direttamente sui risultati del codice numerico. Eseguiamo il codice con i parametri seguenti:

Esempio 1

```
w = 1.0; Tfin = 25.12; dt = 0.01; Tprint = 0.1; V0 = 1.0
```

Essendo $\omega = 1.0$ il periodo del moto vale: $T = 2\pi/\omega \sim 6.28$. Abbiamo scelto un tempo finale $T_f = 25.12$, che corrisponde a circa 4 periodi. Di conseguenza, dovremmo osservare nella soluzione circa 4 oscillazioni.

Nella fig. 3.3.1a mostriamo il confronto tra la soluzione numerica e la soluzione esatta per la posizione $x(t)$. Si vede che la soluzione numerica è inizialmente molto vicina a quella esatta ma, man mano che il tempo avanza, mentre la soluzione esatta è una oscillazione di ampiezza costante, la soluzione numerica, pur mantenendo il periodo corretto, ha una ampiezza crescente nel tempo. Questo effetto può essere visto ancora più chiaramente sul pannello b) della figura, dove è mostrato l'errore globale della soluzione: l'errore, seppur oscillando, cresce nel tempo in maniera secolare. Notiamo che, in questo caso, l'instabilità non si manifesta in maniera "violenta", come si è visto nel caso dell'equazione del punto materiale in moto nel fluido viscoso, cioé con oscillazioni esponenzialmente crescenti. La crescita della soluzione numerica è lenta e proporzionale a Δt, cosicché si potrebbe pensare, scegliendo un valore di Δt molto piccolo, di poter aver una soluzione accettabile anche utilizzando lo schema Eulero forward. Questo è possibile però solo se si cerca la soluzione per periodi di tempo molto limitati. In caso contrario, dobbiamo provare ad ottenere uno schema stabile.

3.3.2 Schema Eulero backward

Il problema della stabilità dello schema per l'equazione (3.8) può essere risolta introducendo lo schema di Eulero backward su una sola delle due equazioni (3.10), utilizzando invece lo schema forward per l'altra equazione. Si può far vedere inoltre che l'effetto stabilizzante dell'implicitazione viene perso se lo schema backward viene applicato ad entrambe le equazioni: in questo caso, lo schema risulta ancora sempre instabile come quello forward. Invece si può ottenere uno schema stabile per certi valori di Δt se si utilizza lo schema forward su una equazione e quello backward sull'altra. Scegliamo di applicare Eulero forward alla prima equazione del sistema (3.10) e lo schema backward alla seconda. Analoghi risultati si hanno se si utilizza la scelta contraria.

Assumiamo quindi che:

$$\begin{aligned} \tfrac{x_{n+1}-x_n}{\Delta t} &= v_n \qquad \Rightarrow \quad x_{n+1} = x_n + \Delta t\ v_{n+1} \\ \tfrac{v_n-v_{n-1}}{\Delta t} &= -\omega^2 x_n \quad \Rightarrow \quad v_n = v_{n-1} - \omega^2 \Delta t\ x_n \end{aligned}$$

Iterando di un passo la seconda equazione si ottiene:

$$\left\{ \begin{array}{rcl} x_{n+1} &=& x_n + \Delta t\ v_{n+1} \\ v_{n+1} &=& v_n - \omega^2 \Delta t\ x_{n+1} \end{array} \right. \tag{3.12}$$

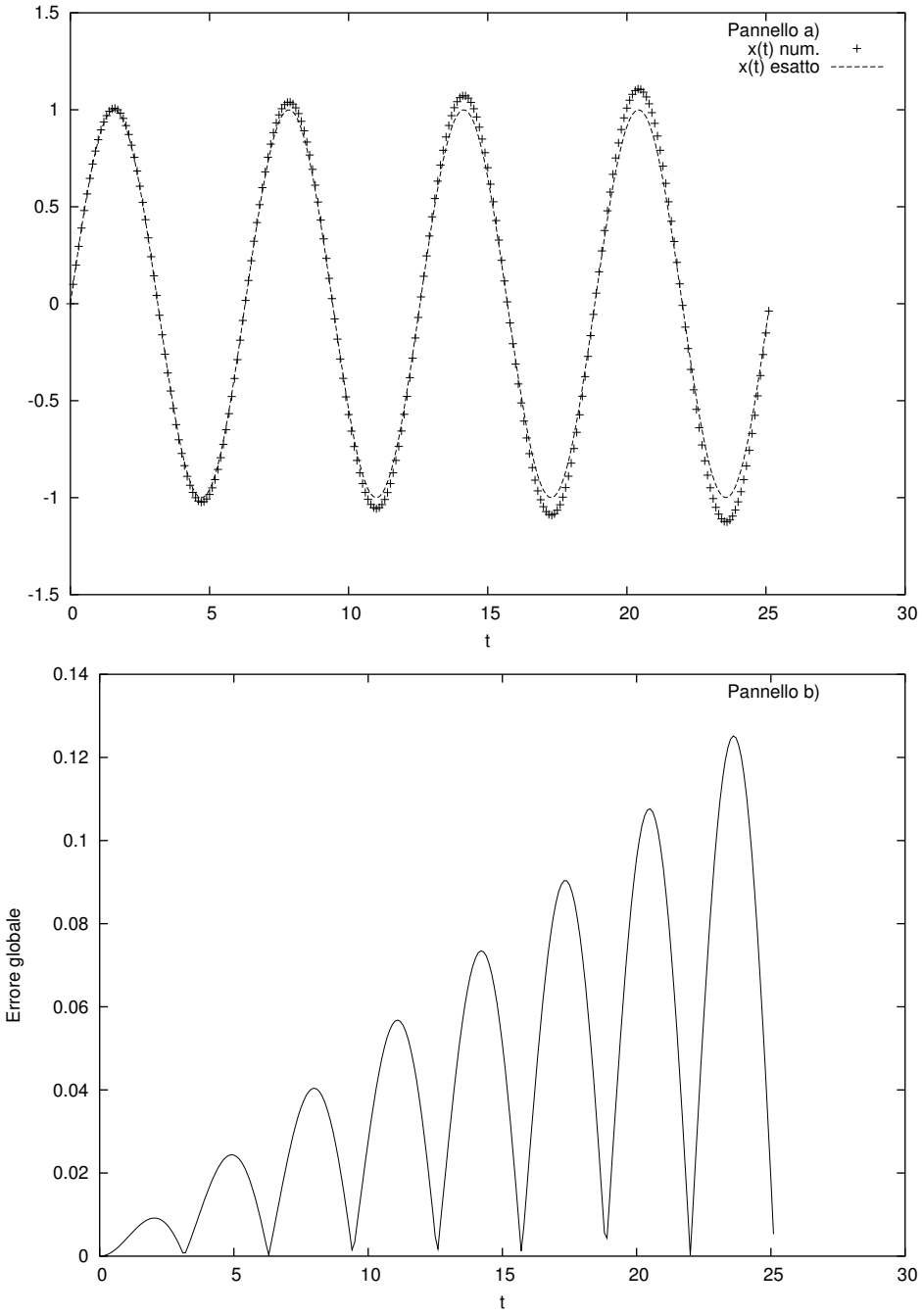


Figura 3.5: Pannello a): Risultati forniti dal codice `eulf.f` utilizzando i parametri di input dell'**Esempio 1**. Pannello b): errore globale per lo stesso caso.

A noi serve di ottenere tutte le quantità al passo $n+1$ al membro destro delle equazioni, di conseguenza possiamo sostituire il valore di x_{n+1} dalla prima equazione nel corrispondente termine della seconda, ottenendo così lo schema:

$$\begin{aligned} x_{n+1} &= x_n + \Delta t\ v_{n+1} \\ v_{n+1} &= -\omega^2 \Delta t\ x_{n+1} + (1 - \omega^2 \Delta t^2)\ v_n \end{aligned}$$

ovvero, in forma matriciale:

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 - \omega^2 \Delta t \end{pmatrix} \begin{pmatrix} x_n \\ v_n \end{pmatrix}$$

Diagonalizzando la matrice a secondo membro si ottiene l'equazione secolare:

$$\lambda^2 - \lambda(2 - \omega^2 \Delta t^2) + 1 = 0$$

che ammette come soluzioni:

$$\lambda = 1 - \frac{\omega^2 \Delta t^2}{2} \pm \frac{\omega \Delta t}{2} \sqrt{\omega^2 \Delta t^2 - 4}$$

Possiamo distinguere due casi, corrispondenti al caso di autovalori reali o complessi:

Caso 1 : $\omega^2 \Delta t^2 - 4 \geq 0$, corrispondente al caso di due autovalori reali. In questo caso l'autovalore massimo vale:

$$1 - \frac{\omega^2 \Delta t^2}{2} + \frac{\omega \Delta t}{2} \sqrt{\omega^2 \Delta t^2 - 4}$$

e la condizione di Von Neumann impone che:

$$\left| 1 - \frac{\omega^2 \Delta t^2}{2} \pm \frac{\omega \Delta t}{2} \sqrt{\omega^2 \Delta t^2 - 4} \right| \leq 1$$

che si può scrivere nella forma di due disequazioni:

$$\begin{cases} -1 \leq 1 - \frac{\omega^2 \Delta t^2}{2} + \frac{\omega \Delta t}{2} \sqrt{\omega^2 \Delta t^2 - 4} \\ 1 - \frac{\omega^2 \Delta t^2}{2} + \frac{\omega \Delta t}{2} \sqrt{\omega^2 \Delta t^2 - 4} \geq 1 \end{cases}$$

La prima disequazione può scriversi:

$$\frac{\omega^2 \Delta t^2}{2} - 2 \leq \frac{\omega \Delta t}{2} \sqrt{\omega^2 \Delta t^2 - 4}$$

ed elevando al quadrato ambo i membri si ottiene: $4 \leq \omega^2 \Delta t^2$ che è soddisfatta per qualunque Δt, poiché per ipotesi siamo nel caso: $\omega^2 \Delta t^2 - 4 \geq 0$. Analogamente, la seconda disequazione fornisce, dopo aver semplificato il termine $\omega \Delta t/2$ (che è sempre positivo per definizione):

$$-\omega \Delta t + \sqrt{\omega^2 \Delta t^2 - 4} \leq 0.$$

Elevando al quadrato si ottiene infine la relazione: $\omega^2 \Delta t^2 - 4 \leq \omega^2 \Delta t^2$ che è sempre soddisfatta. In definitiva, la condizione di Von Neumann risulta sempre soddisfatta nel caso di autovalori reali.

Caso 2 : $\omega^2 \Delta t^2 - 4 < 0$, corrispondente al caso di due autovalori complessi coniugati. Sotto questa ipotesi avremo:

$$\lambda = 1 - \frac{\omega^2 \Delta t^2}{2} \pm \frac{\omega \Delta t}{2} \sqrt{-(4 - \omega^2 \Delta t^2)} = 1 - \frac{\omega^2 \Delta t^2}{2} \pm i \frac{\omega \Delta t}{2} \sqrt{4 - \omega^2 \Delta t^2}$$

I due autovalori hanno identico modulo, che vale:

$$|\lambda| = \sqrt{(1 - \frac{\omega^2 \Delta t^2}{2})^2 + \frac{\omega^2 \Delta t^2}{4}(4 - \omega^2 \Delta t^2)} = 1$$

In altre parole, la condizione di Von Neumann è soddisfatta anche in questo caso.

Concludendo, lo schema Eulero backward risulta stabile per qualunque valore di Δt.

Il seguente programma risolve l'equazione dell'oscillatore armonico unidimensionale utilizzando lo schema (3.12). La struttura del programma è identica alla precedente, l'output dei risultati è scritto nei files `eulbx.dat` e `eulbv.dat`. Il listato del programma è:

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione:
C d^2 x/dt^2 + w^2 x = 0
C con uno schema di Eulero forward per la prima equazione del sistema
C corrispondente e lo schema Eulero backward per la seconda.
C L'output del programma si trova nel file "eulbx.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica per lo spazio (x_num), la terza colonna la
C soluzione esatta per lo spazio (x_ex). La quarta colonna rappresenta il
C valore assoluto della differenza tra la soluzione esatta e quella numerica,
C vale a dire l'errore numerico per la x.
```

```
C Analogamente, il file "eulbv.dat" contiene le analoghe quantita' per la
C velocita'.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM eulero_backward

C - Parte dichiarativa

      IMPLICIT NONE

      REAL w,w2dt,t,dt,Tfin,Tprint,V0
      REAL x_num,x_ex,v_num,v_ex,x_diff,v_diff
      REAL xnp1,vnp1
      INTEGER n,Nsteps,Nprints

C - Main program

C - Immissione dati iniziali

      PRINT *,'Inserire il valore di w (omega)'
      READ (5,*) w
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Vo:'
      READ (5,*) V0

C - Definizione variabili utilizzate nel programma

      t=0.0                    ! Tempo iniziale
      Nsteps=nint(Tfin/dt)     ! Numero passi temporali totali
      Nprints=nint(Tprint/dt)  ! Stampa ogni Nprints passi temporali
      x_num=0.0                ! Condizione iniziale per x:
      x_ex=0.0                 ! x(t=0) = 0.0
      x_diff=0.0
      v_num=V0                 ! A t=0 la soluzione  per la velocita'
      v_ex=V0                  ! e' data dalla condizione iniziale
```

```
      v_diff=0.0
      w2dt=w*w*dt

C - Apertura files per la scrittura delle soluzioni

      OPEN (FILE='eulbx.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,x_num,x_ex,x_diff
      OPEN (FILE='eulbv.dat',UNIT=17,STATUS='UNKNOWN')
      WRITE(17,*) t,v_num,v_ex,v_diff

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         xnp1=x_num+dt*v_num
         vnp1=v_num-w2dt*xnp1
         x_num=xnp1          ! Aggiornamento variabili
         v_num=vnp1
         x_ex=(V0/w)*sin(w*t)
         v_ex=V0*cos(w*t)
         x_diff=abs(x_num-x_ex)
         v_diff=abs(v_num-v_ex)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,x_num,x_ex,x_diff
            WRITE(17,*) t,v_num,v_ex,v_diff
         END IF
      END DO

C - Chiusura files di output e fine programma

      CLOSE (UNIT=16)
      CLOSE (UNIT=17)
      STOP
      END
```

In fig. 3.3.2 è mostrato l'andamento della soluzione ottenuta con lo schema di Eulero backward. A differenza dello schema forward, la soluzione numerica e la soluzione esatta rimangono molto vicine e non si osserva instabilità.

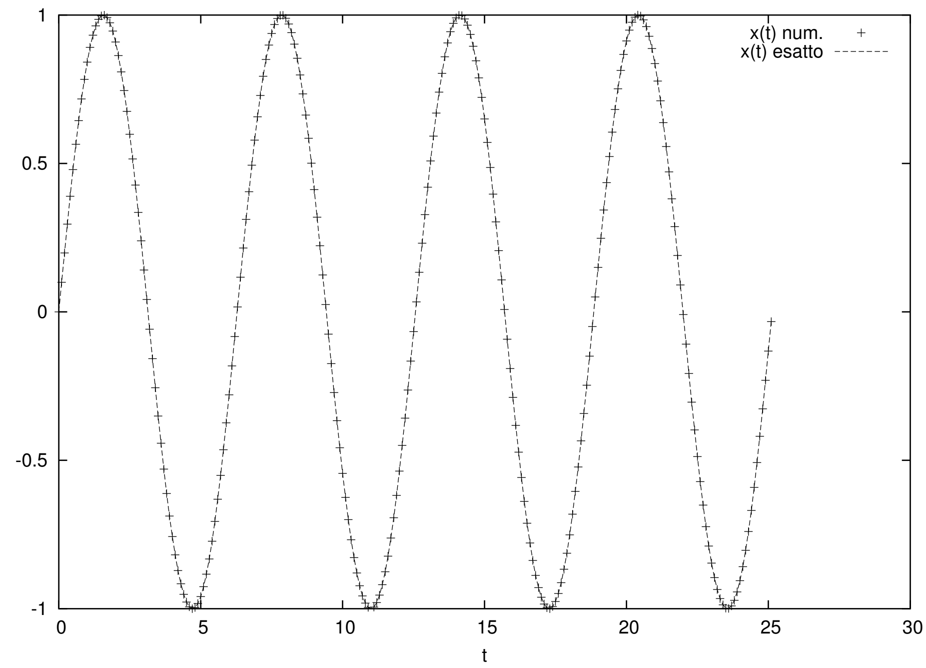


Figura 3.6: Risultati forniti dal codice `eulb.f` utilizzando i parametri di input dell'**Esempio 1**.

3.3.3 Schema Runge-Kutta del secondo ordine.

Per finire, vediamo invece come trattare il sistema di equazioni (3.10) utilizzando uno schema Runge-Kutta al secondo ordine. In questo caso, ogni equazione si sdoppia in una relazione per la x e la v al passo intermedio $\Delta t/2$ e altre due relazioni per lo step totale. In definitiva, lo schema temporale si scriverà come:

$$\begin{cases} x^* = x_n + \frac{\Delta t}{2} v_n \\ v^* = v_n - \omega^2 \frac{\Delta t}{2} x_n \\ x_{n+1} = x_n + \Delta t \ v^* \\ v_{n+1} = v_n - \omega^2 \Delta t \ x^* \end{cases} \quad (3.13)$$

Il seguente programma implementa lo schema (3.13): i files di uscita, chiamati `rk2x.dat''` ed `rk2v.dat` contengono il tempo, la soluzione numerica e la soluzione analitica per la posizione e la velocità rispettivamente:

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
C
C Il programma risolve l'equazione:
C d^2 x/dt^2 + w^2 x = 0
C con uno schema Runge-Kutta del secondo ordine.
```

```
C L'output del programma si trova nel file "rk2x.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica per lo spazio (x_num), la terza colonna la
C soluzione esatta per lo spazio (x_ex). La quarta colonna rappresenta il
C valore assoluto della differenza tra la soluzione esatta e quella numerica,
C vale a dire l'errore numerico per la x.
C Analogamente, il file "rk2v.dat" contiene le analoghe quantita' per la
C velocita'.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM Runge_Kutta

C - Parte dichiarativa

      IMPLICIT NONE

      REAL w,w2dt,t,dt,Tfin,Tprint,V0
      REAL x_num,x_ex,v_num,v_ex,x_diff,v_diff
      REAL x_star,v_star
      INTEGER n,Nsteps,Nprints

C - Main program

C - Immissione dati iniziali

      PRINT *,'Inserire il valore di w (omega)'
      READ (5,*) w
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Vo:'
      READ (5,*) V0

C - Definizione variabili utilizzate nel programma

      t=0.0                    ! Tempo iniziale
      Nsteps=nint(Tfin/dt)     ! Numero passi temporali totali
```

```
      Nprints=nint(Tprint/dt)  ! Stampa ogni Nprints passi temporali
      x_num=0.0                ! Condizione iniziale per x:
      x_ex=0.0                 ! x(t=0) = 0.0
      x_diff=0.0
      v_num=V0                 ! A t=0 la soluzione  per la velocita'
      v_ex=V0                  ! e' data dalla condizione iniziale
      v_diff=0.0
      w2dt=w*w*dt

C - Apertura files per la scrittura delle soluzioni

      OPEN (FILE='rk2x.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,x_num,x_ex,x_diff
      OPEN (FILE='rk2v.dat',UNIT=17,STATUS='UNKNOWN')
      WRITE(17,*) t,v_num,v_ex,v_diff

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         x_star=x_num+0.5*dt*v_num
         v_star=v_num-0.5*w2dt*x_num
         x_num=x_num+dt*v_star
         v_num=v_num-w2dt*x_star
         x_ex=(V0/w)*sin(w*t)
         v_ex=V0*cos(w*t)
         x_diff=abs(x_num-x_ex)
         v_diff=abs(v_num-v_ex)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,x_num,x_ex,x_diff
            WRITE(17,*) t,v_num,v_ex,v_diff
         END IF
      END DO

C - Chiusura files di output e fine programma

      CLOSE (UNIT=16)
      CLOSE (UNIT=17)
      STOP
      END
```

Notiamo che, come nel caso del moto del punto materiale nel fluido viscoso, anche qui la condizione di stabilità per lo schema di Runge-Kutta è soggetta ad una condizione di stabilità analoga a quella dello schema di Eulero forward, cioé, in questo caso, lo schema risulta, come quello di Eulero forward, sempre instabile. Sviluppiamo i calcoli nel dettaglio, anche per acquisire un po' di pratica con l'analisi di stabilità di Von Neumann. Sostituendo il valore al passo intermedio dello schema (3.13) nelle equazioni corrispondenti al passo $n+1$, si ha:

$$\begin{cases} x_{n+1} &= x_n + \Delta t(v_n - \omega^2 \frac{\Delta t}{2} x_n) &= (1 - \frac{\omega^2 \Delta t^2}{2} x_n + \Delta t\ v_n \\ v_{n+1} &= v_n - \omega^2 \Delta t(x_n + \frac{\Delta t}{2} v_n) &= -\omega^2 \Delta t\ x_n + (1 - \frac{\omega^2 \Delta t^2}{2}) v_n \end{cases}$$

ovvero, scritto in forma matriciale:

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} 1 - \frac{\omega^2 \Delta t^2}{2} & \Delta t \\ -\omega^2 \Delta t & 1 - \frac{\omega^2 \Delta t^2}{2} \end{pmatrix} \begin{pmatrix} x_n \\ v_n \end{pmatrix}.$$

La matrice diagonalizzata fornisce l'equazione secolare:

$$\lambda^2 + \lambda(\omega^2 \Delta t^2 - 2) + (1 + \frac{\omega^4 \Delta t^4}{4}) = 0$$

da cui si ottengono gli autovalori:

$$\lambda = 1 - \frac{\omega^2 \Delta t^2}{2} \pm i\omega \Delta t$$

I due autovalori hanno entrambi lo stesso modulo che vale: $|\lambda| = \sqrt{1 + \frac{\omega^4 \Delta t^4}{4}}$. La condizione di Von Neumann impone: $|\lambda| \leq 1$, ma dall'espressione di λ si vede che tale condizione non può mai essere soddisfatta per alcun valore di Δt, cioé lo schema Runge-Kutta al secondo ordine è incondizionatamente **instabile** per l'equazione (3.8).

Tuttavia, bisogna tenere presente che, come nel caso di Eulero forward, l'instabilità non cresce in maniera esponenziale e "violenta", ma assai lentamente. Ad esempio, in fig. 3.3.3, è mostrata l'evoluzione temporale della soluzione ottenuta dall schema Runge-Kutta, confrontata con la soluzione analitica, per i seguenti parametri di input:

Esempio 2

```
w = 1.0; Tfin = 1004.80; dt = 0.01; Tprint = 0.1; V0 = 1.0
```

cioé gli stessi parametri di **Esempio 1** a meno del fatto che il tempo finale è molto più grande: $T_f = 1004.80$, corrispondente a circa 160 periodi di oscillazione. L'output della figura rappresenta solo l'intervallo della soluzione

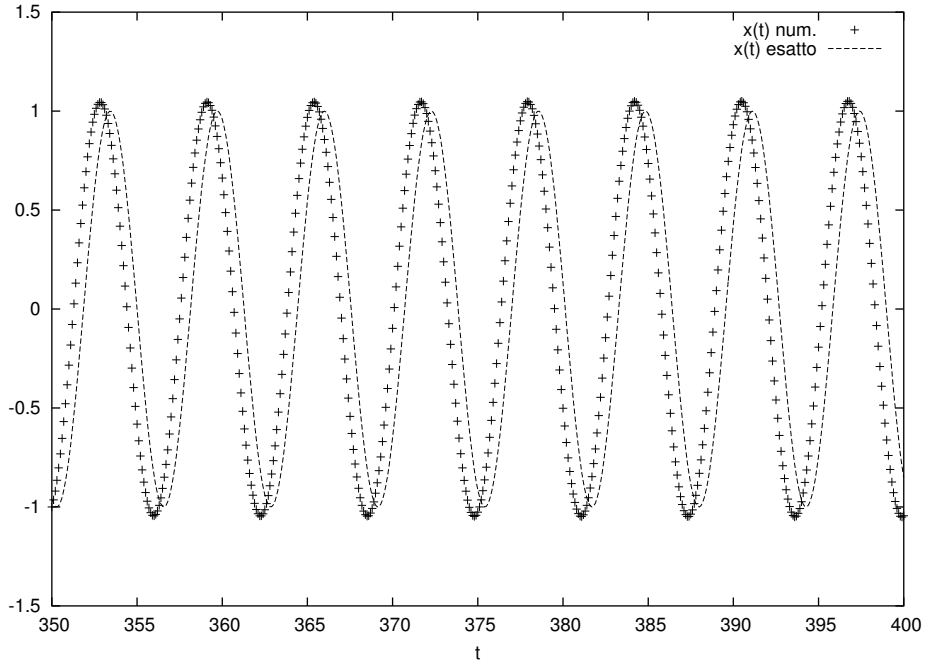


Figura 3.7: Risultati forniti dal codice `rk2.f` utilizzando i parametri di input dell'**Esempio 2**. L'intervallo temporale visualizzato è limitato a $t \in [350, 400]$

tra il tempo 350 e 400. Come si vede la soluzione numerica è costantemente più grande della soluzione esatta ed anche si nota uno sfasamento tra le due (tale sfasamento è una caratteristica generale degli schemi alle differenze finite, dove l'errore numerico viene generalmente diviso in *errore di ampiezza* ed *errore di fase*). Tuttavia bisogna effettivamente andare a tempi molto lunghi per notare la differenza tra le curve, a tempi più brevi le due soluzioni sono praticamente sovrapposte. Si può quindi concludere che, a meno di voler studiare il problema dell'oscillatore armonico per un numero enorme di oscillazioni, per i casi pratici anche lo schema Runge-Kutta può essere tranquillamente utilizzato per ottenere una soluzione accettabile su tempi brevi di evoluzione. Per tempi lunghi, l'applicazione di una schema Eulero backward diventa necessaria, sebbene l'errore numerico del Runge-Kutta (secondo ordine) sia inferiore a quello di Eulero backward (primo ordine). In molti esempi che seguono, lo schema Runge-Kutta verrà applicato allo studio concreto di equazioni tipo la (3.8) su tempi limitati.

3.4 Oscillatore anarmonico in una dimensione.

Abbiamo affrontato in dettaglio, nell'esempio precedente, il caso importante di risoluzione dell'equazione dell'oscillatore armonico. Tuttavia, tale equazione descrive fenomeni fisici solo in prima approssimazione, vale a dire quando si supponga che l'oscillatore, ad esempio la molla con il corpo attaccato, sia perfettamente elastica e indeformabile e gli spostamenti attorno alla posizione di equilibrio molto piccoli, caso non sempre realistico. Questa approssimazione porta ad una equazione differenziale *lineare*. In natura spesso si ha a che fare con fenomeni *non lineari*, cioé occorre studiare il problema fisico proprio mettendosi al di fuori delle approssimazioni suddette, sia perché l'approssimazione lineare è troppo stringente, sia perché le non linearità presentano una maggior (seppur estremamente complessa) ricchezza di comportamenti e di caratteristiche rispetto al caso lineare. In questo esempio cercheremo di introdurre gli effetti delle non linearità in maniera semplice, riservandoci di approfondirne le caratteristiche in esempi successivi.

Introducendo l'equazione dell'oscillatore armonico abbiamo supposto di applicare la legge di Newton:

$$m\ddot{x} = F(x) \tag{3.14}$$

dove $\ddot{x}$ indica la derivata seconda rispetto al tempo della coordinata $x(t)$ del punto attaccato alla molla. Quindi abbiamo applicato la *Legge di Hooke*, supponendo la molla perfettamente elastica ed indeformabile e piccoli spostamenti della massa attorno alla posizione di equilibrio, ottenendo per la forza l'espressione: $F(x) = -kx$ con k costante elastica della molla. Tale espressione della forza elastica si ottiene da uno sviluppo in serie di Taylor per la forza di richiamo della molla:

$$\begin{aligned} F(x) &= -\left[F(x_0) + \left.\frac{dF}{dx}\right|_{x=x_0} (x-x_0) + \left.\frac{d^2F}{dx^2}\right|_{x=x_0} \frac{(x-x_0)^2}{2!} \right. \\ &\quad \left. + \left.\frac{d^3F}{dx^3}\right|_{x=x_0} \frac{(x-x_0)^3}{3!} + \dots \right] \end{aligned}$$

dove il segno negativo viene dal fatto che si tratta di una forza di richiamo e la forza si oppone quindi allo spostamento. Supposto che x_0 rappresenti la posizione di equilibrio della molla e di aver scelto il sistema di riferimento in tale posizione, si avrà $x_0 = 0$. Inoltre, poiché x_0 rappresenta la posizione di equilibrio della molla, la risultante delle forze in tale posizione deve essere nulla a ciascun istante di tempo, vale a dire: $F(x_0) = F(0) = 0$. Posto

quindi:

$$k = BP\frac{dF}{dx}_{x=0}; \qquad \alpha' = \frac{1}{2!}\left.\frac{d^2F}{dx^2}\right|_{x=0}; \qquad \beta' = \frac{1}{3!}\left.\frac{d^3F}{dx^3}\right|_{x=0}; \qquad \ldots$$

si ottiene l'espressione della forza nel caso generale:

$$F(x) = -\left[kx + \alpha' x^2 + \beta' x^3 + \ldots\right].$$

Fermando l'approssimazione all'ordine lineare si ottiene la classica legge di Hooke, mentre tenendo anche gli ordini superiori nello spostamento x si ottiene, dopo aver diviso per la massa m del corpo, l'espressione:

$$\ddot{x} + \omega_0^2 x = -\alpha x^2 - \beta x^3 - \ldots \tag{3.15}$$

dove si è posto: $\omega_0^2 = k/m$, $\alpha = \alpha'/m$, $\beta = \beta'/m$, e così via.

La (3.15) rappresenta l'equazione dell'*oscillatore anarmonico*. In questa relazione, l'incognita x appare elevata a potenze di ordine superiore al primo, l'equazione risulta quindi essere *non lineare*. La teoria delle equazioni differenziali non ci consente di risolvere equazioni di questo tipo, addirittura il teorema di esistenza e unicità della soluzione non è applicabile, quindi a priori non si sa nemmeno se tale soluzione esiste ed è unica. Tuttavia, facendo opportune ipotesi semplificative, si possono ricavare delle soluzioni approssimate per ordini delle potenze superiori al primo grado nella (3.15). Sul libro di meccanica di Landau e Lifŝits (vedi le letture consigliate) è riportata la soluzione della (3.15) come correzioni successive alla soluzione del caso dell'oscillatore armonico. Ad esempio, se si suppone $\alpha \neq 0$ e nulli tutti gli altri coefficienti (β, ecc.) a destra della (3.15), si trova una soluzione del tipo:

$$x(t) = -\frac{\alpha}{2\omega_0^2} + A_1 \cos(\omega_0 t) + \frac{\alpha}{6\omega_0^2}\cos(2\omega_0 t) \tag{3.16}$$

dove si sono ignorate le fasi nella soluzione per semplicità.

Come si vede, alla soluzione classica dell'oscillatore armonico a frequenza ω_0, della forma $\cos(\omega_0 t)$, appaiono sommati altri due termini, uno costante rispetto al tempo e l'altro ad una frequenza $2\omega_0$. Quello che succede in pratica è che la non linearità quadratica accoppia la soluzione lineare dell'oscillatore a frequenza ω_0 con se' stessa producendo due nuove frequenze caratteristiche di oscillazione: la $\omega_0 - \omega_0 = 0$, cioé la parte della soluzione indipendente dal tempo e la $\omega_0 + \omega_0 = 2\omega_0$, cioé la parte della soluzione a frequenza doppia $2\omega_0$, rispetto alla ω_0. Per questo motivo, la frequenza ω_0 viene in genere chiamata *frequenza fondamentale*, mentre le altre *frequenze combinatorie*.

Si può far vedere in maniera del tutto analoga, ma con calcoli molto complessi, che la presenza del termine $\beta \neq 0$ comporterebbe nella soluzione,

oltre ad un termine a frequenza nulla e doppia rispetto alla fondamentale, anche di un termine del tipo $\cos(3\omega_0 t)$, cioé con frequenza tripla rispetto a quella dell'oscillatore armonico semplice.

Come vedremo fra breve, la presenza di tali termini altera profondamente la natura della soluzione. Ad esempio, considerando la (3.16), la soluzione mostra una asimmetria rispetto all'assse dei tempi, cioé la molla impiega tempi differenti a contrarsi e ad espandersi e l'oscillazione risulta "deformata". Per questo, i termini a frequenza diversa dalla fondamentale che appaiono ad esempio nella (3.16) vengono detti *termini anarmonici*.

Il seguente programma calcola la soluzione numerica dell'equazione dell'oscillatore anarmonico quando un numero arbitrario di parametri anarmonici è presente a secondo membro dell'equazione. Il programma riceve in input la condizione iniziale, la frequenza fondamentale ω_0, e il numero di parametri anarmonici da inserire, nonché finalmente le ampiezza di tali parametri (α, β, ecc.). Quindi il programma integra l'equazione utilizzando uno schema Eulero backward, che risulta sempre stabile per quanto detto negli esempi precedenti. Notiamo che l'equazione in questione è non lineare, a causa della presenza dei parametri anarmonici. In generale, nulla può essere detto circa la stabilità di una equazione non lineare, dato che l'analisi di stabilità risulterebbe eccessivamente complicata, se non impossibile. La condizione di Von Neumann, infatti, richiede di scrivere esplicitamente la matrice di amplificazione associata allo schema temporale, il che equivale a supporre che il sistema che lo rappresenta sia un sistema lineare. Il meglio che si possa fare in tali casi è linearizzare il sistema e ipotizzare che se lo schema risulta stabile per l'equazione linearizzata dovrebbe esserlo anche nel caso del sistema non lineare. Non c'è alcuna garanzia che ciò sia vero, tuttavia è ragionevole pensare che, viceversa, se lo schema non è stabile nel caso lineare, allora tanto meno lo sarà nel caso non lineare.

Il listato seguente rappresenta il programma "anarmonico.f":

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione di un oscillatore anarmonico per
C studiare gli effetti del termine non-lineare presente nell'equazione.
C Possono essere introdotti fino a 10 termini anarmonici nell'equazione.
C L'output del programma viene scritto nel file "anarmonico.dat", il quale
C contiene tre colonne: il tempo, la soluzione per la posizione (x) e la
C velocita' (v). Il programma stampa le soluzioni a ciascun passo temporale,
C fermandosi laddove c'e' uno zero nella soluzione, cioe' dopo un periodo.
C Alla fine, il programma scrive alcune informazioni, fra cui il numero di
C punti totali stampati. Questa informazione, in particolare, viene richiesta
```

```
C come dato in ingresso dal programma "spettro.f".
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM anarmonico
      IMPLICIT NONE

C - Parte dichiarativa

      REAL v0,t,dt,x,v,w,w2,smv,xb
      REAL anarm(10)
      INTEGER i,ia,Nanarm,nosc

C - Inserimento parametri iniziali

      WRITE(6,*) 'Inserire velocita'' iniziale:'
      READ(5,*) v0
      WRITE(6,*) 'Inserire il parametro w (sqrt(k/m)):'
      READ(5,*) w
      WRITE(6,*) 'Passo temporale Delta t:'
      READ(5,*) dt
      WRITE(6,*) 'Quanti parametri anarmonici vuoi inserire? (Max. 10)'

C -Controllo sul numero di parametri inserito

      Nanarm=11
      DO WHILE (Nanarm.GT.10)
         READ(5,*) Nanarm
         IF (Nanarm.GT.10) THEN
            WRITE(6,*) 'Il numero massimo di parametri e'' 10!'
            WRITE(6,*) 'Sono stati richiesti ',Nanarm,' parametri!'
            WRITE(6,*) 'Riprova...'
         ENDIF
      ENDDO

C - Inserimento parametri anarmonici

      DO ia=1,Nanarm
         WRITE(6,*) 'Inserire parametro anarmonico N. ',ia
         READ(5,*) anarm(ia)
      ENDDO
```

```
C - Parametri della simulazione

      w2=w*w
      x=0.0
      v=v0
      t=0.0

C - Apre il file per il salvataggio dei dati e scrittura condizione iniziale

      OPEN(UNIT=16,FILE='anarmonico.dat',STATUS='UNKNOWN')
      WRITE(16,*) t,x,v

C - Inizio ciclo temporale

      i=1
      xb=1.0
      nosc=0
      DO WHILE (nosc.LT.2)
         xb=x
         t=i*dt

C - Secondi membro per v e passo Eulero per calcolo v_n+1

         smv=w2*x
         DO ia=1,Nanarm
            smv=smv+anarm(ia)*(x**(ia+1))
         END DO
         v=v-smv*dt

C - Secondi membro per theta e passo Eulero per calcolo theta_n+1
C - Nota che la v e' calcolata al passo n+1, per Eulero backward!

         x=x+v*dt

C - Stampa su file

         WRITE(16,*) t,x,v

C - Fine ciclo temporale
```

```
         IF (xb*x.LT.0.0) nosc=nosc+1
         i=i+1
      END DO

C - Fine programma

      CLOSE(UNIT=16)
      WRITE(6,*) 'Numero totale di passi: ',i
      WRITE(6,*) 'Durata totale della simulazione: ',t
      STOP
      END
```

Notiamo che la presenza dei termini anarmonici fa sì che il periodo di oscillazione del sistema non sia più semplicemente stabilito dall'inverso della frequenza fondamentale: $2\pi/\omega_0$ ma in generale dipenderà dal numero di parametri anarmonici inserito. Quindi il programma non richiede in input il tempo finale, che rappresenta una stima ragionevole della durata totale del programma, ma calcola da solo questo periodo analizzando quando la soluzione passa per zero due volte. Il programma implementa le equazioni nella forma seguente:

$$\begin{cases} \frac{dv}{dt} &= -\omega^2 x + \sum_{k=1}^{N_a} \alpha_k x^{k+1} \\ \frac{dx}{dt} &= v \end{cases}$$

dove N_a rappresenta il numero di parametri anarmonici richiesti (identificato dalla variabile `Nanarm` nel programma), gli α_k sono i parametri stessi, v è la velocità del punto materiale e x la posizione.

Per analizzare l'effetto dei parametri anarmonici sulla soluzione, abbiamo eseguito il programma diverse volte con un numero via via crescente di parametri anarmonici $N_a = 0,\ 1,\ 2,\ 3,\ 4,\ 5$, diminuendo il valore di α_k di 0.1 all'aumentare di k. Il caso $N_a = 0$ rappresenta il caso dell'oscillatore armonico semplice, quindi il caso di riferimento. Per distinguere le varie prove effettuate, chiamiamo `Run0`, ..., `Run5` i programmi eseguiti con $N_a = 0$, ..., $N_a = 5$, rispettivamente. Di conseguenza, i parametri iniziali del programma nei vari "runs" sono i seguenti:

Esempio 1

```
Run0: v0 = 1.0; w = 1.0; dt = 0.01; Nanarm=0
Run1: v0 = 1.0; w = 1.0; dt = 0.01; Nanarm=1; anarm(1)=0.5
Run2: v0 = 1.0; w = 1.0; dt = 0.01; Nanarm=2; anarm(1)=0.5;
                                              anarm(2)=0.4
Run3: v0 = 1.0; w = 1.0; dt = 0.01; Nanarm=3; anarm(1)=0.5;
```

```
                                                    anarm(2)=0.4;
                                                    anarm(3)=0.3
Run4: v0 = 1.0; w = 1.0; dt = 0.01; Nanarm=4; anarm(1)=0.5;
                                                    anarm(2)=0.4;
                                                    anarm(3)=0.3;
                                                    anarm(4)=0.2
Run5: v0 = 1.0; w = 1.0; dt = 0.01; Nanarm=5; anarm(1)=0.5;
                                                    anarm(2)=0.4;
                                                    anarm(3)=0.3;
                                                    anarm(4)=0.2;
                                                    anarm(5)=0.1
```

I risultati delle varie simulazioni sono mostrati in figura 3.4. Come si vede, il periodo delle oscillazioni varia al variare del numero di termini anarmonici presenti dell'equazione. La forma dell'oscillazione, inoltre, risulta essere asimmetrica, cioé l'oscillazione nella fase di espansione dura meno che nella fase di compressione o viceversa e anche l'ampiezza risulta inferiore nella parte positiva che nella parte negativa. In generale, la forma della soluzione non è più chiaramente una sinusoide come nel caso del `Run0`, cioé dell'oscillatore armonico.

I parametri anarmonici inseriti nell'**Esempio 1** erano tutti positivi, ma ovviamente è possibile inserirne anche di negativi. Notiamo tuttavia che inserendo parametri troppo grandi (più grandi di 0.5, in particolare) può portare la crescita di instabilità numeriche nel codice. Questo conferma quanto detto sopra, cioé che lo schema Eulero backward, pur risultando incondizionatamente stabile per qualunque valore dei parametri nel caso lineare, non lo è più necessariamente nel caso non lineare.

3.4.1 Spettro di Fourier

Una tecnica largamente utilizzata nello studio delle equazioni non lineari e nell'analisi della turbolenza nei fluidi in particolare, utile per individuare il contributo di diverse frequenze (o lunghezze d'onda) in un segnale che ne contenga molte sovrapposte, è lo sviluppo di Fourier e il calcolo del relativo spettro di potenza. Fourier dimostrò che, data una qualunque funzione $f(t)$, periodica su un intervallo T, essa può venire scomposta come una successione infinita di funzioni esponenziali complesse (quindi funzioni oscillanti) nella forma:

$$f(t) = \sum_{k=-\infty}^{\infty} a_k \exp(i\frac{2\pi}{T}kt)$$

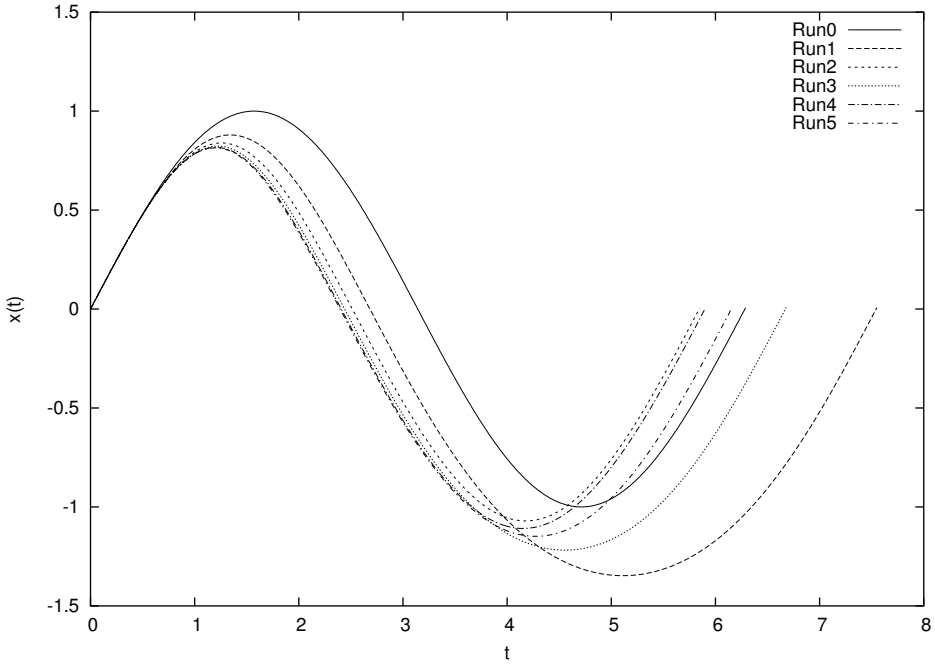


Figura 3.8: Risultati forniti dal codice `anarmonico.f` utilizzando i parametri di input di `Run0`, ..., `Run5` descritti nell'**Esempio 1**.

dove i rappresenta l'unità immaginaria e a_k è un numero (in generale complesso) detto k-esimo coefficiente di Fourier. Tale sviluppo è possibile se gli a_k hanno la forma:

$$a_k = \frac{1}{T} \int_0^T f(t) e^{-i\frac{2\pi}{T}kt} dt \tag{3.17}$$

Quindi, nota la funzione $f(t)$ si possono calcolare i coefficienti a_k dello sviluppo tramite la relazione suddetta.

L'utilità pratica dello sviluppo di Fourier viene dal fatto che i coefficienti a_k rappresentano proprio il contributo che le armoniche k =0, 1, 2, ecc. danno nella funzione $f(t)$. Di conseguenza, noto che una generica funzione periodica $f(t)$ è costituita dalla sovrapposizione di più armoniche, si puó individuare il contributo di ogni singola armonica calcolando i coefficienti di Fourier relativi a quelle armoniche. Questo fatto può essere reso quantitativamente ancora più evidente tenendo conto del **teorema di Parceval**, espresso dalla seguente formula:

$$\int_0^T f^2(t) dt = T \sum_{k=-\infty}^{+\infty} |a_k|^2$$

cioé l'integrale del quadrato della funzione $f(t)$ su tutto l'intervallo di periodicità è uguale, a meno del fattore T, alla somma dei moduli quadri dei coefficienti di Fourier dello sviluppo di f. Se la $f(t)$ rappresenta una quantità fisica, ad esempio la velocità o l'ampiezza di una oscillazione, allora l'integrale di f^2 rappresenta **l'energia** associata al moto o all'oscillazione. Di conseguenza, il teorema di Parceval ci dice che il quadrato del k-esimo coefficiente di Fourier di f rappresenta il contributo dell'armonica k-esima all'energia del segnale.

In genere, per analizzare il contributo delle varie frequenze presenti in un segnale si usa graficare lo spettro del segnale stesso, vale a dire fare un grafico dell'energia associata alla k-esima armonica di Fourier: $|a_k|^2$ in funzione di k. Il seguente programma calcola lo spettro della soluzione ottenuta dal programma `anarmonico.f`. Il programma richiede in input il numero dei punti nel file `anarmonico.dat`, scritto in output alla fine del programma `anarmonico.f`, quindi elabora lo spettro e produce in output il file `spettro.dat`. In pratica, il programma calcola i coefficienti di Fourier, parte reale e parte immaginaria, valutando l'integrale nella (3.17) tramite la regola dei trapezi, quindi ne calcola il modulo quadro e lo stampa in funzione di k. Il listato del programma `spettro.f` è il seguente:

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Programma Spettro:
C calcola lo spettro di Fourier per l'output del programma anarmonico.f.
C Lo spettro viene calcolato trovando dapprima i coefficienti di Fourier
C delle 10 frequenze richieste in anarmonico.f, quindi viene applicato
C il teorema di Parceval per il calcolo dell'energia. L'output del
C programma si trova nel file "spettro.dat" che contiene una colonna
C rappreentante le frequenze e una le energie associate ai modi di
C Fourier della soluzione. Il coefficiente di Fourier viene calcolato
C utilizzando la regola dei trapezi per lo svolgimento dell'integrale
C (funzione fint_trap).
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM Spettro

      IMPLICIT NONE
      INTEGER Ndim
      PARAMETER (Ndim=10000)
      REAL t(Ndim),f(Ndim),ff(Ndim)
```

```
      INTEGER npt,i,k
      REAL energ,w0,Tper,dt,pi,fint_trap,v,akr,aki

      pi=acos(-1.0)          ! Pi greco
      WRITE(6,*) 'Numero di punti da leggere nel file:'
      READ(5,*) npt
      IF (npt.GT.Ndim) THEN
         WRITE(6,*) 'Errore! Il numero dei punti indicato e'''
         WRITE(6,*) 'maggiore della dimensione Ndim assegnata'
         WRITE(6,*) 'al vettore dei dati in ingresso.'
         WRITE(6,*) 'Modificare tale parametro e riprovare!'
         STOP
      END IF
      OPEN(FILE='anarmonico.dat',UNIT=15,STATUS='UNKNOWN')
      DO i=1,npt
         READ(15,*) t(i),f(i),v
      END DO
      CLOSE(UNIT=15)
      w0=2.0*pi/t(npt)
      dt=t(2)-t(1)
      Tper=t(npt)-t(1)
      WRITE(6,*) 'Frequenza dell''oscillazione:'
      WRITE(6,*) w0
      WRITE(6,*) 'Passo temporale dt:'
      WRITE(6,*) dt
      WRITE(6,*) 'Lunghezza di periodicita'':'
      WRITE(6,*) Tper

C - Ciclo sulle armoniche e calcolo dello spettro

      OPEN(FILE='spettro.dat',UNIT=16,STATUS='UNKNOWN')
      DO k=0,10
         DO i=1,npt
            ff(i)=f(i)*cos(k*w0*t(i))   ! Coefficiente di Fourier reale
         END DO
         akr=fint_trap(ff,npt,dt)/Tper
         DO i=1,npt
            ff(i)=f(i)*sin(k*w0*t(i))   ! Coefficiente di Fourier immaginario
         END DO
         aki=fint_trap(ff,npt,dt)/Tper
         IF (k.EQ.0) THEN
```

```
      energ=Tper*(akr*akr+aki*aki)
   ELSE
      energ=2.0*Tper*(akr*akr+aki*aki)
   ENDIF
   WRITE(16,*) k,energ
END DO

CLOSE(UNIT=16)
STOP
END
```

```
REAL FUNCTION fint_trap(f,n,dt)
IMPLICIT NONE
INTEGER n,i
REAL f(n),dt,sum

sum=0.5*(f(1)+f(n))
DO i=2,n-1
   sum=sum+f(i)
END DO
fint_trap=dt*sum

RETURN
END
```

Il calcolo dell'integrale col metodo dei trapezi è implementato sotto forma di funzione `fint_trap`. Abbiamo plottato il grafico dello spettro per i vari runs citati nell'**Esempio 1**. I risultati sono mostrati in figura 3.4.1.

Per il caso del `Run 0`, cioé dell'oscillatore armonico semplice, l'unica frequenza presente è la $k = 1$, mentre l'energia presente sulle altre armoniche è ad un livello di rumore. All'aumentare della quantità `Nanarm`, un livello più alto di energia è presente anche sulle armoniche superiori. Notiamo che mettere, ad esempio, tre termini anarmonici, implica un contributo, seppur minimo, all'energia anche su modi maggiori di tre, dovuto all'accoppiamento non lineare tra le armoniche più energetiche.

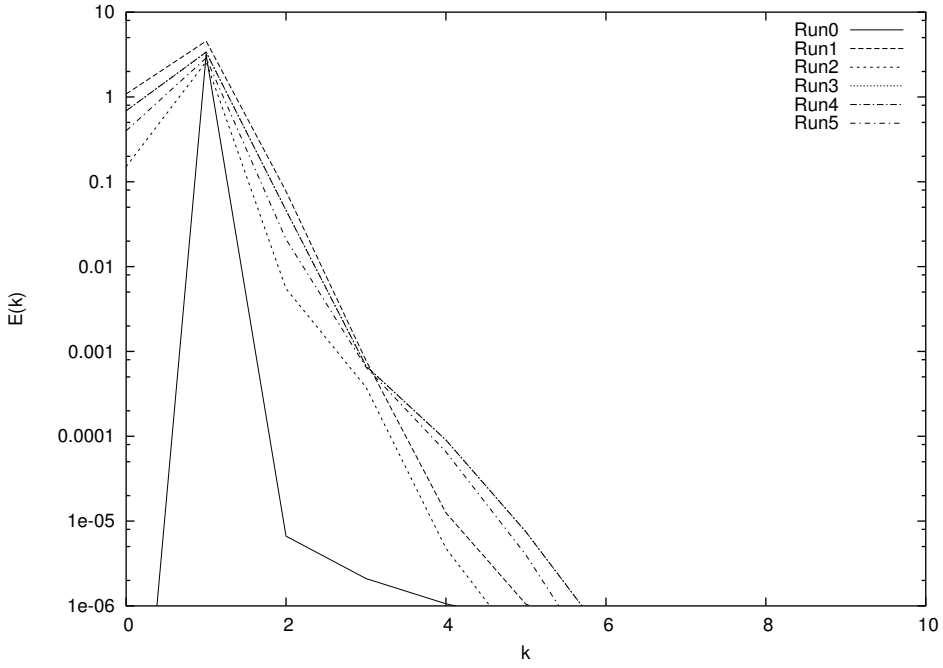


Figura 3.9: Grafico degli spettri prodotti tramite il programma `spettro.f` per i parametri di input di `Run0`, ..., `Run5` descritti nell'**Esempio 1**.

3.5 Pendolo semplice unidimensionale.

Abbiamo esaminato finora l'equazione dell'oscillatore armonico ed anarmonico. Quest'ultimo, essendo descritto da una equazione contenente termini non lineari, presenta una fenomenologia molto più ricca rispetto al caso dell'oscillatore armonico semplice. Molto spesso, in fisica, si incontrano problemi descritti da equazioni non lineari, quindi risulta opportuno poter padroneggiare degli strumenti che ci consentano di studiare quello che accade in questi casi. Un esempio tipico è fornito dall'equazione del pendolo semplice.

Con tale nome si indica una massa attaccata ad un filo inestensibile o ad una bacchetta di massa trascurabile, a sua volta sospeso ad un punto fisso. La massa, spostata dalla posizione di equilibrio, descriverà un moto oscillatorio. Indicando con θ l'angolo formato tra il filo e la verticale passante per il punto di sospensione, l'equazione caratteristica del pendolo semplice è data da:

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

dove g è l'accelerazione di gravità ed L la lunghezza del filo. Indicando con:

$$\omega_0 = \sqrt{g/L} \tag{3.18}$$

si ottiene l'equazione:

$$\frac{d^2\theta}{dt^2} = -\omega_0^2 \sin\theta. \tag{3.19}$$

Tale equazione è non lineare, poiché la funzione incognita appare come argomento della funzione seno. Sviluppando tale funzione in serie di Taylor nell'intorno della posizione di equilibrio $\theta = 0$, si ottiene:

$$\frac{d^2\theta}{dt^2} + \omega_0^2\theta = -\omega_0^2 \left(-\frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \dots \right)$$

cioé per angoli molto piccoli (approssimazione lineare) i termini a secondo membro sono trascurabili e l'equazione diventa quella di un oscillatore armonico unidimensionale con frequenza ω_0. Se l'angolo di oscillazione non è piccolo, il pendolo si comporta come un oscillatore anarmonico e quindi appariranno frequenze di ordine superiore rispetto alla frequenza principale ω_0.

Studiamo quindi numericamente l'equazione (3.19). Per poter implementare uno schema numerico atto a risolvere l'equazione, al solito possiamo riscrivere l'equazione differenziale del secondo ordine come un sistema di due equazioni del primo ordine ponendo: $\Omega = \dot{\theta}$. L'equazione (3.19) può quindi essere riscritta come:

$$\begin{cases} \frac{d\Omega}{dt} &= -\omega_0^2 \sin\theta \\ \frac{d\theta}{dt} &= \Omega \end{cases} \tag{3.20}$$

Per formalizzare il problema dal punto di vista matematico occorre inoltre fornire due condizioni iniziali. Possiamo supporre, ad esempio, che il pendolo si trovi inizialmente nella posizione di equilibrio $\theta(t = 0) = 0$ e che venga messo in movimento conferendogli una velocità iniziale $v(t = 0) = v_0$. Poiché la velocità tangenziale del punto materiale ad un generico istante di tempo è legata all'angolo θ tramite la relazione:

$$v(t) = L\dot{\theta}(t) = L\Omega(t)$$

si avrà che: $\Omega(t = 0) = \Omega_0 = v_0/L$.

Dovremo quindi scrivere uno schema numerico, possibilmente stabile per qualche valore di Δt. Anche in questo caso, trattandosi di una equazione non lineare, nulla può essere detto a priori circa la stabilità di uno schema qualsiasi. Il meglio che possiamo fare è utilizzare uno schema che almeno nel caso lineare risulti stabile, sperando che si mantenga tale anche nel caso non lineare. A questo punto, siccome l'equazione linearizzata coincide con l'equazione di un oscillatore armonico a frequenza ω_0 data dalla (3.18), l'unico schema che risulti stabile, come visto negli esempi precedenti, è quello in

cui una delle due equazioni implementa uno schema di Eulero backward. Per quanto detto nell'esempio 2, potremmo utilizzare anche uno schema Runge-Kutta del secondo ordine in cui, pur essendo instabili, le oscillazioni crescono molto lentamente. In ogni caso, qualunque sia lo schema utilizzato, studiando questo esempio ci concentreremo, più che sulle proprietà dello schema numerico in se', sulla possibilità di riprodurre le caratteristiche generali della soluzione dell'equazione e di confrontarle con le previsioni teoriche. Di conseguenza, scegliamo di studiare l'equazione utilizzando lo schema di Eulero backward, cioé:

$$\begin{cases} \Omega_{n+1} &= \Omega_n - \omega_0^2 \Delta t \sin\theta_n \\ \theta_{n+1} &= \theta_n + \Delta t \Omega_{n+1} \end{cases} \tag{3.21}$$

Un programma che implementa uno schema di questo tipo per l'equazione (3.20) con le condizioni iniziali specificate è il seguente:

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Il programma risolve l'equazione:
C d^2 theta/dt^2 = - w^2 sin(theta)
C con uno schema di Eulero forward per la prima equazione del sistema
C corrispondente e lo schema Eulero backward per la seconda.
C L'output del programma si trova nel file "eulb.dat" nella forma
C di quattro colonne. La prima colonna rappresenta il tempo t, la seconda
C colonna la soluzione numerica per l'angolo theta (th_num), la terza
C colonna rappresenta la soluzione numerica per la Omega (derivata di
C theta rispetto al tempo) (Om_num). La quarta colonna nel file rappresenta
C la soluzione esatta dell'equazione linearizzata per l'angolo theta
C (th_lin).
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      PROGRAM eulero_backward

C - Parte dichiarativa

      IMPLICIT NONE

      REAL w,w2dt,t,dt,Tfin,Tprint,Om_0
      REAL th_num,th_lin,Om_num
      REAL th_np1,Om_np1
      INTEGER n,Nsteps,Nprints
```

```
C - Main program

C - Immissione dati iniziali

      PRINT *,'Inserire il valore di w (omega)'
      READ (5,*) w
      PRINT *,'Inserire il tempo finale della simulazione:'
      READ (5,*) Tfin
      PRINT *,'Inserire passo temporale dt:'
      READ (5,*) dt
      PRINT *,'Ogni quanti tempi stampare i risultati?'
      READ (5,*) Tprint
      PRINT *,'Inserire condizione iniziale Omega_0:'
      READ (5,*) Om_0

C - Definizione variabili utilizzate nel programma

      t=0.0                    ! Tempo iniziale
      Nsteps=nint(Tfin/dt)     ! Numero passi temporali totali
      Nprints=nint(Tprint/dt)  ! Stampa ogni Nprints passi temporali
      th_num=0.0               ! Condizione iniziale per theta:
      th_lin=0.0               ! theta(t=0) = 0.0
      Om_num=Om_0              ! e per Omega.
      w2dt=w*w*dt

C - Apertura file per la scrittura delle soluzioni

      OPEN (FILE='eulb.dat',UNIT=16,STATUS='UNKNOWN')
      WRITE(16,*) t,th_num,Om_num,th_lin

C - Main loop

      DO n=1,Nsteps
         t=n*dt
         Om_np1=Om_num-w2dt*sin(th_num)
         th_np1=th_num+dt*Om_np1
         th_num=th_np1         ! Aggiornamento variabili
         Om_num=Om_np1
         th_lin=(Om_0/w)*sin(w*t)
         IF (mod(n,Nprints).EQ.0) THEN
            WRITE(16,*) t,th_num,Om_num,th_lin
```

```
         END IF
      END DO

C - Chiusura file di output e fine programma

      CLOSE (UNIT=16)
      STOP
      END
```

Il programma implementa lo schema (3.21) e calcola la soluzione per l'angolo θ e la sua derivata Ω in funzione del tempo. La soluzione viene scritta sul file "eulb.dat" sotto forma di quattro colonne: la prima contiene il tempo t, la seconda e la terza le soluzioni numeriche per l'angolo θ e la sua derivata temporale Ω, infine la quarta colonna rappresenta la soluzione lineare calcolata a parità di condizioni iniziali.

3.5.1 Test lineare

Come primo test per verificare che il programma fornisca risultati corretti, possiamo effettuare il confronto tra la soluzione numerica nel caso lineare e la soluzione esatta, cioé la soluzione dell'equazione dell'oscillatore armonico con frequenza ω_0. Per testare il codice possiamo ad esempio inserire differenti valori della velocità angolare iniziale Ω_0, in modo da esaminare quello che accade quando siamo nel caso lineare e man mano passiamo al caso non lineare. I parametri inseriti sono i seguenti:
Esempio 1

```
w = 1.0; Tfin = 125; dt = 0.01; Tprint = 0.1;
Omega_0 = 0.01, 0.1, 0.5
```

dove `w` rappresenta ω_0, la frequenza lineare di oscillazione del pendolo semplice (fissare tale valore, corrisponde in pratica a fissare L ed il periodo dell'oscillazione: $T = 2\pi/\omega_0 = 2\pi$ in unità arbitrarie); T_{fin} rappresenta il tempo finale della simulazione (circa 20 periodi, per il valore calcolato di T); gli altri parametri hanno il significato usuale ed `Omega_0` rappresenta invece il valore iniziale di Ω, vale a dire Ω_0. Graficando i risultati del programma, non mostrati, in particolare la seconda e quarta colonna in funzione del tempo per i vari casi, si nota come la soluzione numerica e quella lineare siano praticamente indistinguibili per $\Omega_0 = 0.01$, siano realmente poco differenti per $\Omega_0 = 0.1$ e siano sempre più separate per $\Omega_0 = 0.5$. Il fatto che la

soluzione numerica approssimi in maniera davvero ottima la soluzione lineare ci rassicura sulla correttezza del metodo utilizzato nel programma. Per valori più grandi di Ω_0, quando siamo lontani dal caso lineare, la presenza dei termini anarmonici nell'equazione diventa importante e, come visto nell'esempio 3, la curva che rappresenta la soluzione non è più una sinusoide perfetta e anche il periodo delle oscillazioni risulta cambiato a causa della presenza delle armoniche superiori.

3.5.2 Traiettorie di fase

Un metodo molto utilizzato per studiare l'andamento delle soluzioni di una equazione rappresentante un fenomeno fisico quando la soluzione stessa non possa essere calcolata è di tentare di disegnare le traiettorie di fase della soluzione. Cioé si introduce uno spazio particolare, detto "piano di fase" dove, invece di graficare i valori della soluzione per la posizione o la velocità del punto materiale, si disegnano i grafici della curve, parametrizzate attraverso il tempo t, della soluzione $x(v)$, ovvero $v(x)$, cioè lo spazio rappresentato in funzione della velocità o viceversa. Il vantaggio di tale metodo consiste nel fatto che, per sistemi conservativi, la posizione e la velocità del punto materiale sono legate tra loro dalla relazione che esprime la conservazione dell'energia meccanica e da tale equazione si può ricavare la forma delle traiettorie di fase in maniera semplice, anche quando non si conosca la traiettoria del punto materiale in maniera analitica.

A titolo di esempio, applichiamo il metodo al caso del pendolo semplice. Per scrivere la conservazione dell'energia per il pendolo bisogna tenere conto del fatto che l'unica forza agente sul sistema che compia lavoro è la forza peso (la tensione del filo agisce perpendicolarmente allo spostamento), a cui è associata una energia potenziale: $U = mgL(1 - \cos\theta)$, dove m rappresenta la massa del punto materiale, g l'accelerazione di gravità, L la lunghezza del filo e θ l'angolo tra il filo e la verticale, come in precedenza. In questo caso abbiamo assunto di aver posto il sistema di riferimento nel punto più basso della traiettoria, in modo che $U = 0$ quando il punto passa per la posizione di equilibrio $\theta = 0$. Analogamente, se v rappresenta la velocità tangenziale del punto materiale lungo la traiettoria ad un istante t generico, l'energia cinetica del punto materiale sarà data dall'espressione: $K = mv^2/2$. Poiché le forze agenti sono conservative, supposto che il corpo parta dalla posizione di equilibrio con una certa velocità v_0, si avrà che l'energia meccanica totale (cioé la somma dell'energia cinetica più l'energia potenziale) a ciascun istante di tempo eguaglierà l'energia meccanica iniziale, cioè:

$$\frac{1}{2}mv_0^2 = \frac{1}{2}mv^2 + mgL(1 - \cos\theta)$$

Poiché $v = L\dot{\theta} = L\Omega$, si ha, semplificando le masse e la lunghezza L:

$$\Omega = \sqrt{\Omega_0^2 - \frac{2g}{L}(1 - \cos\theta)} = \sqrt{\Omega_0^2 - 2\omega_0^2(1 - \cos\theta)} \tag{3.22}$$

In altri termini, anche se non siamo in grado di calcolare le soluzioni analitiche $\theta(t)$ e $\Omega(t)$ per tutti i tempi, tuttavia la (3.22) ci fornisce una relazione analitica (una volta fissata la condizione iniziale Ω_0) che ci permette di calcolare Ω in funzione dell'angolo θ, cioé ci fornisce proprio l'espressione analitica della *traiettoria del punto nello spazio delle fasi*. Notiamo che, a rigore e per come lo abbiamo definito, lo spazio delle fasi ha sugli assi la posizione e la velocità, ma θ ed Ω sono proporzionali a tali quantitá tramite la stessa costante di proporzionalità, la lunghezza L del filo.

Possiamo cercare di vedere come sono fatte tali traiettorie (dette appunto *traiettorie di fase*) per diversi valori del parametro Ω_0. Ad esempio, se si pone: $\Omega_0 = 0$, la relazione (3.22) ha soluzioni reali solo se $\theta = 0$. La traiettoria di fase, in questo caso, si riduce ad un singolo punto, l'origine del piano di fase corrispondente al punto $(\theta, \Omega) = (0, 0)$, che corrisponde al fatto ovvio che se il punto parte con velocitá nulla dalla posizione di riposo permane in tale posizione indefinitamente, essendo tale posizione di equilibrio stabile.

Le traiettorie di fase per diversi valori di Ω_0 sono graficate raggruppate in figura 3.5.2. Per dare una interpretazione fisica a tali curve, consideriamo cosa succede praticamente per valori di Ω_0 crescenti. Consideriamo il caso in cui partiamo da valori Ω_0 molto piccoli, cioé ci mettiamo sull'asse delle Ω molto vicino all'origine. In questo caso, che corrisponde al caso lineare, si può usare per il coseno di θ lo sviluppo di Taylor al secondo ordine: $\cos\theta \sim 1 - \frac{1}{2}\theta^2$ e la (3.22) in questo caso fornisce:

$$\Omega = \sqrt{\Omega_0^2 - \omega_0^2\theta^2}$$

ed elevando al quadrato e dividendo per Ω_0^2:

$$\frac{\Omega^2}{\Omega_0^2} + \frac{\omega_0^2\theta^2}{\Omega_0^2} = 1$$

che rappresenta, nello spazio (θ, Ω) una ellisse di semiassi Ω_0^2 ed Ω_0^2/ω_0^2 rispettivamente. Nel caso particolare i cui $\omega_0 = 1$, come nell'**Esempio 1**, i semiassi sono uguali e l'ellisse degenra in una circonferenza.

Aumentando il valore di Ω_0 ci allontaneremo ben presto dal caso lineare, quindi le curve risulteranno comunque delle traiettorie chiuse (questo è dovuto al fatto che, se la velocità iniziale non è cosiì elevata da far arrivare il pendolo nella posizione di equilibrio instabile, in ogni caso la massa ripasserà

dalla posizione di equilibrio con la stessa velocitá iniziale, dovendosi conservare l'energia), ma non più di forma ellittica, bensì di un ovale schiacciato (vedi figura 3.5.2). La Ω decresce dal valore iniziale Ω_0 fino ad un valore nullo che si ha per $\theta = \arccos(1 - \frac{\Omega_0^2}{2\omega_0^2})$, corrispondente a quando il pendolo reggiunge la massima ampiezza di oscillazione (la velocità tangenziale si annulla). Quindi la velocità diventa negativa diminuendo fino al valore $-\Omega_0$, corrispondente a metà oscillazione del pendolo, dopodiché il processo si ripete con i segni scambiati per θ e Ω finchè il punto materiale non ripassa per la posizione di partenza e la traiettoria si chiude.

Aumentando ancora il valore di Ω_0 fino a: $\Omega_0 = 2\omega_0$ si osserva un comportamento diverso nella curva che rappresenta la soluzione: in tal caso la Ω si annulla per $\theta = \pi$, che corrisponde al caso in cui il punto materiale ha energia sufficiente per portarsi fino alla posizione di *equilibrio instabile*. In tal caso, come si vede chiaramente in figura 3.5.2, il punto materiale ha due possibilità distinte: o continuare lungo la traiettoria iniziata e completare il giro per valori di θ sempre crescenti, ovvero può ritornare indietro sulla traiettoria per cui è salito, ma con velocità negativa. Tale situazione corrisponde, nella figura 3.5.2, alle due traiettorie di fase che si intersecano in $\theta = \pi$.

Per valori ulteriormente più elevati di Ω_0 si ha un cambiamento radicale nella natura delle soluzioni: si passa dalle traiettorie chiuse, osservate finora a delle traiettorie aperte, corrispondenti al fatto che per valori di Ω_0 più grandi di $2\omega_0$ il pendolo avrà energia sufficiente per superare la posizione di equilibrio instabile e continuare sulla traiettoria senza che la sua velocità (cioé Ω) si annulli per qualunque valore di Ω. Questa situazione corrisponderà, in figura 3.5.2, al caso delle curve oscillanti che **non** hanno intersezione con l'asse $\Omega = 0$ (ad esempio la curva corrispondente ad $\Omega_0 = 2.5$).

Possiamo riprodurre le curve teoriche di figura 3.5.2 utilizzando il programma di cui abbiamo dato il listato sopra. Dobbiamo porre anche in questo caso $\omega_0 = 1$ e plottare le curve $\Omega(\theta)$ per gli stessi valori di Ω_0. In pratica, utilizzeremo i seguenti dati di input:

Esempio 2

```
w = 1.0; Tfin = 25; dt = 0.01; Tprint = 0.1;
Omega_0 = -2.5, -2.0, -1.5, -1.0, -0.5, -0.1,
          0.1, 0.5, 1.0, 1.5, 2.0, 2.5.
```

Per ogni run con un differente valore di Ω_0, i risultati sono stati memorizzati in un file con nome differente. La curva $\Omega(\theta)$ corrispondente si ottiene semplicemente graficando la terza colonna del file (Ω) in funzione della seconda (θ) anziché in funzione del tempo, come al solito. Il risultato finale è

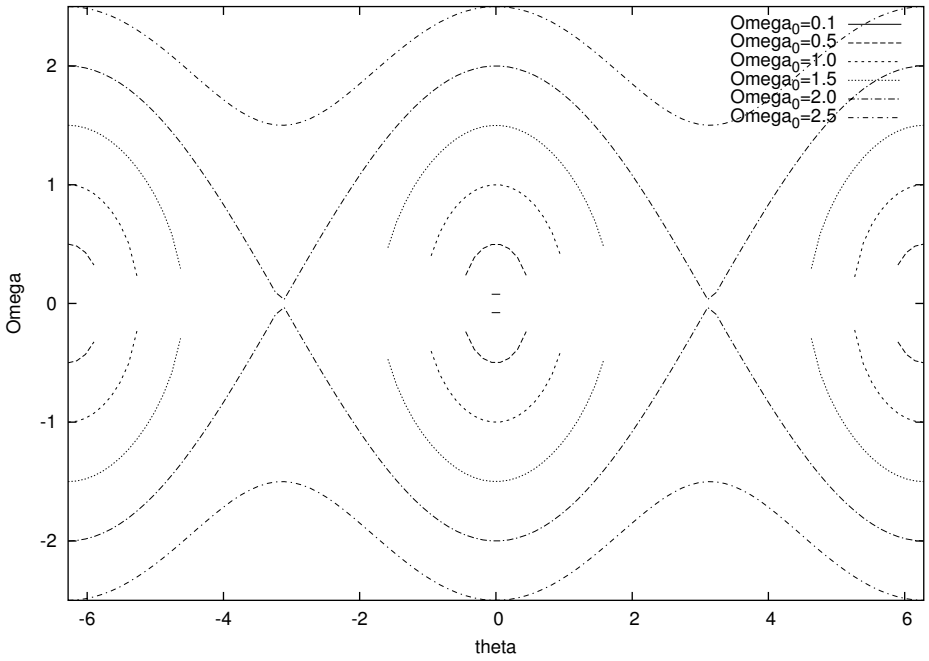


Figura 3.10: Curve rappresentanti le traiettorie di fase del pendolo semplice ottenute analiticamente dalla relazione (3.22) per i valori di Ω_0 dati nell'**Esempio 1**.

mostrato in figura 3.5.2. Notiamo che, nella figura, abbiamo rappresentato solo la parte positiva del grafico 3.5.2 in quanto nel nostro programma il tempo parte da zero. Inoltre, le curve ottenute per valori negativi di Ω_0, che normalmente sarebbero apparse nel terzo quadrante del sistema di assi scelto nel grafico, sono state riportate nel secondo quadrante per garantire una rappresentazione esteticamente migliore. Notiamo infine che le traiettorie chiuse a destra di $\theta = \pi$ non appaiono in quanto dovremmo spostare l'origine dei tempi nel programma per ottenerle.

Al di là delle curve che non appaiono a causa della scelta particolare dell'origine dei tempi che abbiamo fatto, le curve presenti nel grafico sono assolutamente equivalenti nei valori numerici, a meno degli errori introdotti dallo schema numerico, ai valori del grafico in figura 3.5.2. Naturalmente, una analisi più approfondita può essere effettuata andando a calcolare, ad esempio, il valore analitico esatto di Ω per alcuni valori di θ fornito dalla (3.22) e confrontarlo coi valori numerici dati nei risultati del programma (o addirittura modificare il programma stesso per stampare una ulteriore colonna di risultati nel file di output, contenente i valori della soluzione analitica (3.22) per i valori di θ calcolati dal programma). In ogni caso si troverà che il valore numerico differisce da quello analitico per quantità di ordine Δt, che è l'errore che si commette nello schema temporale.

3.6 Il moto dei pianeti

Consideriamo un pianeta di massa m che si muove attorno al sole che ha massa M. Il moto del pianeta avviene sul piano dell'eclittica, e supponiamo che questo sia il piano (x, z). La forza che agisce fra il sole ed il pianeta é una forza di tipo centrale, ossia é

$$\mathbf{F} = -GMm\frac{\mathbf{e}_i}{r^2} \tag{3.23}$$

dove G é la costante di gravitazione universale, ed $\mathbf{e_i}$ é il versore nella direzione i–esima, mentre $r = \sqrt{x^2 + z^2}$. Posto per comoditá $GM = 1$ le equazioni del moto del pianeta sono date da

$$\frac{dv_x}{dt} = -\frac{x}{r^3}\ ; \qquad \frac{dv_z}{dt} = -\frac{z}{r^3}$$

dove le velocitá sul piano sono definite da $dx/dt = v_x$ e $dz/dt = v_z$.

Abbiamo quindi da risolvere un sistema di 4 equazioni differenziali. Per costruire un codice attribuiamo alle variabili fisiche un elemento di un vettore $y(j)$ nel modo seguente: $x \to y(1)$, $v_x \to y(2)$, $z \to y(3)$ e $v_z \to y(4)$. La parte principale e la subroutine dei secondi membri é listata di seguito

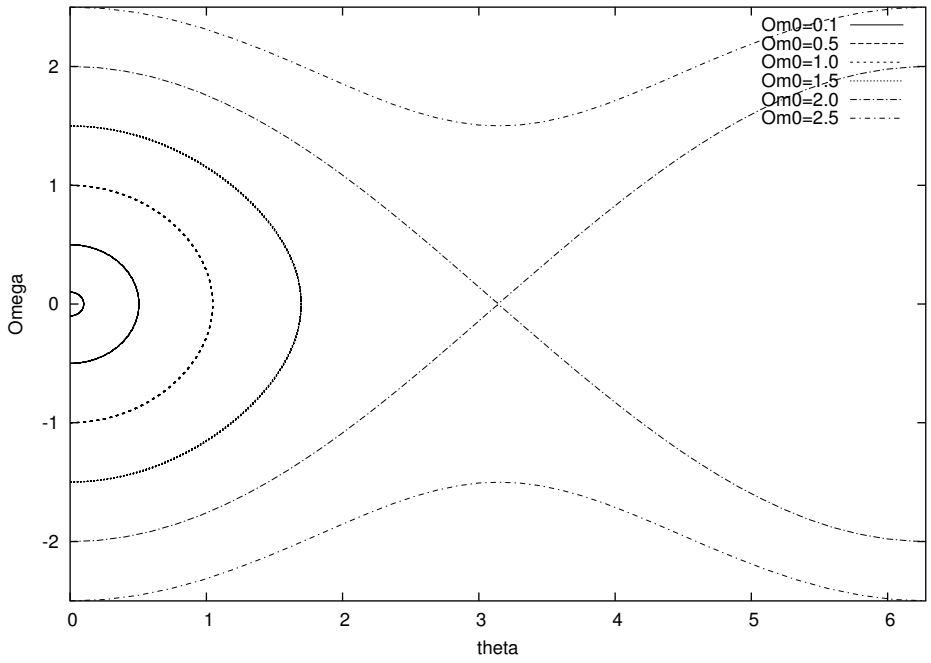


Figura 3.11: Curve rappresentanti le traiettorie di fase del pendolo semplice ottenute numericamente per i valori di Ω_0 dati nell'**Esempio 2**.

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Programma Principale
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      Program pianeti
c
      parameter (Neq=4)
      dimension y(Neq),dy(Neq)
      external dery
      character*50 fileout
c
      print *,'File di uscita'
      read(5,'(a)') fileout
      open(unit=16,file=fileout,status='unknown')
      print *,'Inserisci i valori di time  dt  e tprint'
      read(5,*) time,Dt,tprint
      nmax=int(time/Dt)
      nprint=int(tprint/Dt)
c
```

```
c   Condizioni iniziali
c
      y(1)=0.5     ! variabile x
      y(2)=0.0     ! variabile Vx
      y(3)=0.0     ! variabile y
      y(4)=1.63    ! variabile Vy
c
      write(16,*) 0.0,y(1),y(2),y(3),y(4)
c
      do n=1,nmax
        t0=(n-1)*Dt
        t1=n*Dt
        call RK2(Neq,t0,Dt,y,dy)
        if(mod(n,nprint).eq.0) then
          write(16,*) t1,y(1),y(2),y(3),y(4)
        end if
      end do
      close(16)
      stop
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c  Subroutine che calcola il secondo membro dell'equazione
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine Dery(Neq,t,y,dy)
      dimension y(Neq),dy(Neq)
c
      R=sqrt(y(1)*y(1)+y(3)*y(3))
      R3=R*R*R
      dy(1)=y(2)
      dy(2)=-y(1)/R3
      dy(3)=y(4)
      dy(4)=-y(3)/R3
c
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Facciamo un run del programma, con passo temporale $\Delta t = 10^{-3}$, fino al tempo $t = 100$ passi temporali e scrivendo su file ogni $t_p = 10^{-1}$. In figura 3.12 é riportato x in funzione di z, che rappresenta l'orbita del pianeta attorno al sole, le due variabili interne associate sono $y(1)$ ed $y(3)$. Come si

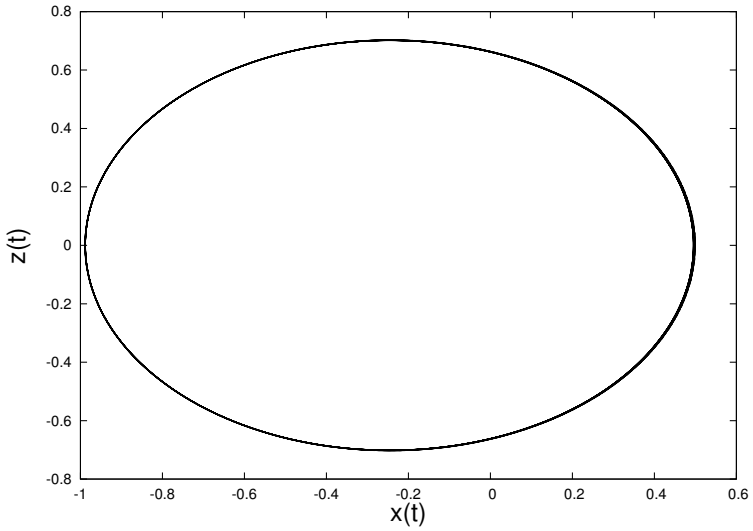


Figura 3.12: L'evoluzione nel tempo di $z(t)$ in funzione di $x(t)$, che descrive l'orbita del pianeta sul piano dell'eclittica, come risulta dal programma con un passo temporale di $\Delta t = 10^{-3}$.

vede l'orbita ellittica é ben descritta dal programma, ed é chiusa, ossia ritorna su se stessa dopo una rivoluzione completa. Questo é quanto é rappresentato dalle equazioni che abbiamo integrato numericamente. Possiamo anche fare un grafico di x in funzione di v_x (ossia $y(1)$ in funzione di $y(2)$), riportato in figura 3.13 e di z in funzione di v_z (ossia $y(3)$ in funzione di $y(4)$), riportato in figura 3.14 per vedere lo spazio delle fasi del sistema.

Facciamo adesso una integrazione numerica con un passo temporale piú grande, ossia usiamo $\Delta t = 10^{-1}$, lasciando invariati gli altri parametri. Come si vede dalla figura 3.15, in questo caso l'orbita non si chiude, perché l'errore commesso nel corso del calcolo numerico é grande, e quindi l'equazione che integriamo é affetta da errore. L'orbita si muove sul piano dell'eclittica, ma questo é un errore puramente numerico, nel senso che l'effetto fisico della migrazione dell'orbita non é descritta nelle equazioni che abbiamo usato, e quindi non puó essere ritrovata.

Se si vuole verificare le quantitá che si conservano, o vedere l'errore commesso nello schema, allora si modifica il programma nel modo seguente

```
              .....
              .....
c
      y(1)=0.5      ! variabile x
      y(2)=0.0      ! variabile Vx
```

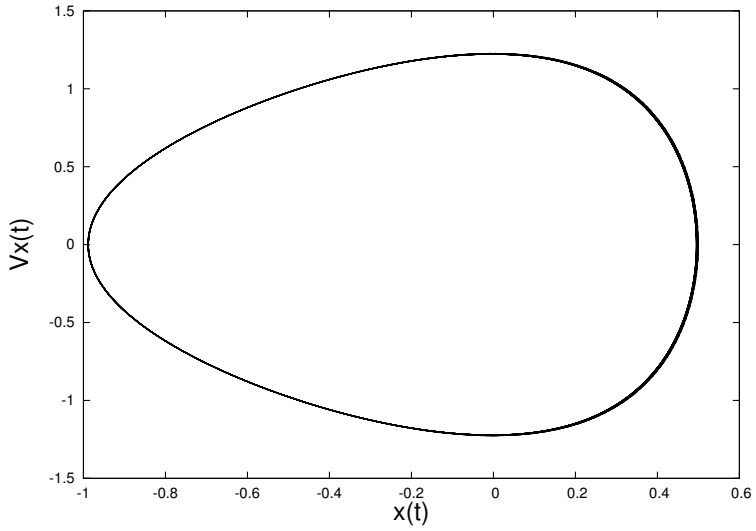


Figura 3.13: E' rappresentata l'evoluzione temporale di $v_x(t)$ in funzione di $x(t)$.

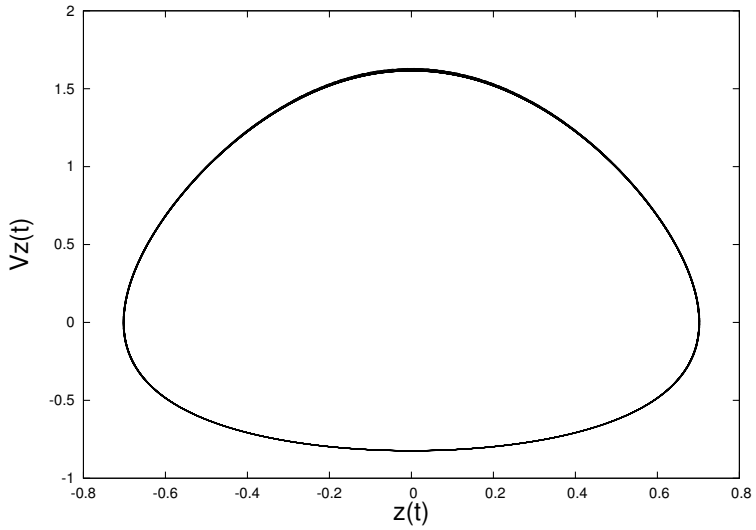


Figura 3.14: E' rappresentata l'evoluzione temporale di $v_z(t)$ in funzione di $z(t)$.

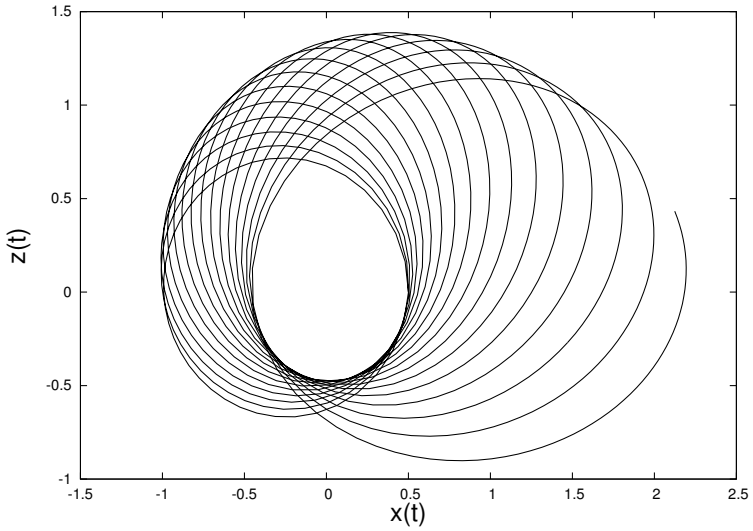


Figura 3.15: L'evoluzione nel tempo di $z(t)$ in funzione di $x(t)$, che descrive l'orbita del pianeta sul piano dell'eclittica, come risulta dal programma con un passo temporale di $\Delta t = 10^{-1}$.

```
      y(3)=0.0     ! variabile y
      y(4)=1.63    ! variabile Vy
c
      Va0=y(1)*y(4)-y(2)*y(3)          ! Velocita' aereolare
      r=sqrt(y(1)*y(1)+y(3)*y(3))
      Ek=0.5*(y(2)*y(2)+y(4)*y(4))     ! Energia cinetica
      Ep=-1.0/r                        ! Energia potenziale
      Energy0=Ek+Ep                    ! Energia totale
c
      write(16,*) 0.0,Va0,Ek,Ep,0.0,0.0
c
      do n=1,nmax
            .....
            .....
        if(mod(n,nprint).eq.0) then
          Va=y(1)*y(4)-y(2)*y(3)   ! Velocita' aereolare
          r=sqrt(y(1)*y(1)+y(3)*y(3))
          Ek=0.5*(y(2)*y(2)+y(4)*y(4))
          Ep=-1.0/r
          Energy=Ek+Ep
          dE=Energy-Energy0
          dVa=Va-Va0
```

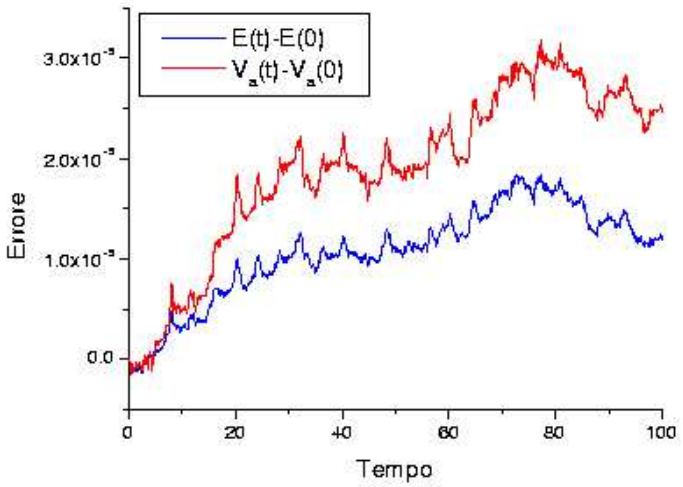


Figura 3.16: E' riportata l'evoluzione temporale degli errori commessi nellla conservazione dei due invarianti, nel caso in cui il passo di integrazione sia $\Delta t = 10^{-3}$.

```
      write(16,*) t1,Va,Ek,Ep,dVa,dE
    end if
  end do
        .....
        .....
```

Le quantitá che si conservano durante il moto sono l'energia totale $E = E_p + E_k$ e la velocitá aerolare. L'energia totale, usando le variabili interne, é data da

$$E = -\frac{1}{r} + \frac{1}{2}\left[v_x^2 + v_y^2\right] = -\frac{1}{r} + 0.5\left[y(2)^2 + y(4)^2\right]$$

dove ovviamente $r = [y(1)^2 + y(3)^2]^{1/2}$. La velocitá aereolare é invece definita come $V_a = xv_y - yv_x$, ossia nelle variabili interne $V_a = y(1) * y(4) - y(2) * y(3)$. Se facciamo un grafico della variazione di energia rispetto al suo valore all'istante iniziale, ossia $\Delta E = E(t) - E(0)$ troviamo che $\Delta E \simeq 10^{-5}$, che puó essere considerato come l'errore commesso nel corso del calcolo numerico, come si vede dalla figura 3.16. Dello stesso ordine di grandezza é la variazione della velocitá aerolare $\Delta V_a = V_a(t) - V_a(0)$ (vedi figura 3.16. Quando invece usiamo $\Delta t = 10^{-1}$, vediamo che l'errore commesso, ossia la differenza ΔE cresce nel tempo, e quindi il calcolo numerico é errato (figura 3.17).

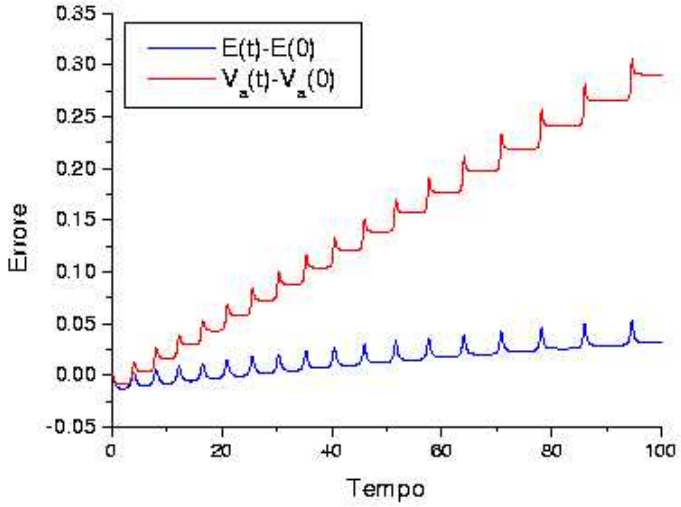


Figura 3.17: E' riportata l'evoluzione temporale degli errori commessi nellla conservazione dei due invarianti, nel caso in cui il passo di integrazione sia $\Delta t = 10^{-1}$.

3.7 Il sistema preda–predatore

Questo é un esempio classico di un sistema di equazioni non lineari, chiamato sistema di Lotka–Volterra, ed é un primo esempio di come si possano ottenere, con una modellizzazione che parte dalla fisica, informazioni su sistemi biologici. Infatti il sistema descrive l'evoluzione temporale, in un sistema isolato, di due popolazioni di individui che possono interagire fra loro: le prede $x(t)$ ed i predatori $z(t)$.

Si suppone che le prede nascano in continuazione con un tasso costante positivo a, e muoiano solo quando interagiscono con i predatori, con un tasso $\gamma(t) = -cz(t)$ proporzionale al numero di predatori esistenti. Di conseguenza

$$\frac{dx}{dt} = ax - cxz \tag{3.24}$$

I predatori muoiono continuamente con un tasso costante negativo $-b$, mentre aumentano se interagiscono con le prede, con un tasso $\gamma(t) = ex(t)$ proporzionale al numero di prede $x(t)$, per cui

$$\frac{dz}{dt} = -bz + exz \tag{3.25}$$

Il sistema, di 2 equazioni ordinarie, possiede quindi 4 parametri arbitrari

positivi, ossia a, b, c ed e che vanno fissati in modo arbitrario a seconda delle condizioni che si vuole osservare[1]. Per costruire un codice attribuiamo alle variabili fisiche un elemento di un vettore $y(j)$ nel modo seguenet: $x \to y(1)$ e $z \to y(2)$. La parte principale e la subroutine dei secondi membri é listata di seguito

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Programma Principale
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
       Program preda-predatore
c
       parameter (Neq=2)
       dimension y(Neq),dy(Neq)
       external dery
       common a,b,c,e
       character*50 fileout
c
       print *,'File di uscita'
       read(5,'(a)') fileout
       open(unit=16,file=fileout,status='unknown')
       print *,'Inserisci i valori di time  dt  e tprint'
       read(5,*) time,Dt,tprint
       nmax=int(time/Dt)
       nprint=int(tprint/Dt)
c
c   Condizioni iniziali
c
       y(1)=0.5
       y(2)=1.0
       a=0.5
       b=0.5
       c=0.5
       e=0.5
       write(16,*) 0.0,y(1),y(2)
c
      do n=1,nmax
        t0=(n-1)*Dt
        t1=n*Dt
```

[1]Si noti che le due costanti c ed e non sono uguali, perché la sparizione di una preda non necessariamente implica la nascita di un predatore.

```
        call RK2(Neq,t0,Dt,y,dy)
        if(mod(n,nprint).eq.0) then
          write(16,*) t1,y(1),y(2)
        end if
      end do
      close(16)
c
      stop
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Subroutine che calcola il secondo membro dell'equazione
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine Dery(Neq,t,y,dy)
      dimension y(Neq),dy(Neq)
      common a,b,c,e
c
      dy(1)=a*y(1)-c*y(1)*y(2)
      dy(2)=-b*y(2)+e*y(1)*y(2)
c
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Facciamo un run del programma fino ad un tempo $t = 50$, con passi temporali di $\Delta t = 10^{-3}$ e scrivendo su file ogni $t_p = 10^{-1}$. Se facciamo un grafico di $x(t)$ e di $z(t)$ in funzione del tempo (figura 3.18), vediamo che le prede ed i predatori hanno un moto oscillante nel tempo, che risulta essere periodico ma non armonico (ossia le funzioni $x(t)$ e $z(t)$ non sono proporzionali a funzioni seno o coseno). Lo sfasamento fra le due funzioni dipende dal valore delle costanti caratteristiche dell'equazione. Se infatti facciamo il grafico di x in funzione di z (figura 3.19, vediamo che la caratteristica principale del sistema di Lotka–Volterra é quello di avere soluzioni cicliche, ossia il sistema preda–predatore é in una situazione di equilibrio, le prede ed i predatori si sostengono ciclicamente (questa é la *prima legge* di Volterra). Il ciclo nello spazio delle fasi (x, z) non é circolare perché le funzioni $x(t)$ e $z(t)$ sono periodiche ma non armoniche. Quando le prede aumentano nel tempo, anche i predatori aumentano, fino al punto (estremo destro A sul grafico) in cui le prede decrescono perché ci sono troppi predatori, mentre i predatori continuano ad aumentare. A partire dal punto B in alto nel grafico i predatori cominciano a decrescere perché le prede sono ridotte, e fino al punto C sia

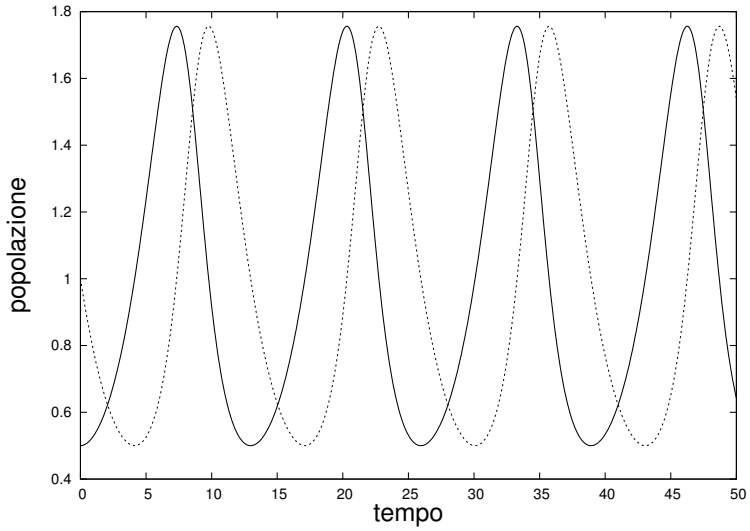


Figura 3.18: Evoluzione nel tempo delle variabile $x(t)$ (prede, linea continua) e $z(t)$ (predatori, linea tratteggiata) nel sistema di Lotka–Volterra.

le prede che i predatori decrescono. Finalmente le prede, per mancanza di predatori, riescono a ricrescere fino al punto D, da cui ricomincia il ciclo.

Nel seguito riportiamo un esempio del file di uscita, con i valori delle tre colonne, per calcolare il periodo e la frequenza del moto oscillante. Nella prima colonna é riportato il tempo, a passi di t_p, nella seconda colonna la variabile (t) e nella terza colonna la variabile $z(t)$.

```
  ......
  ......

 7.02900028   1.74165964   0.891678572
 7.12800026   1.74958229   0.925214767
 7.22700024   1.75455701   0.9603194
 7.32600021   1.7564255   0.996925592        <---
 7.42500019   1.7550503   1.03494
 7.52400017   1.75032282   1.07424068
 7.62300014   1.74216509   1.11468005

  ......
  ......

19.9980011   1.74117804   0.889993906
```

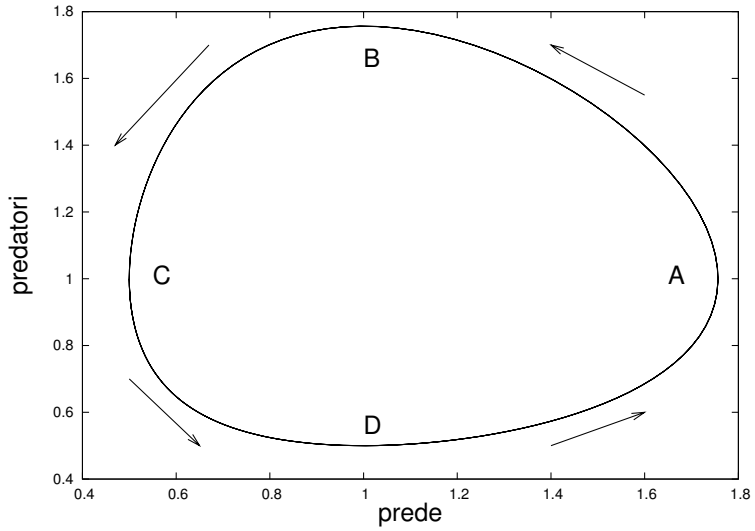


Figura 3.19: Lo spazio delle fasi $z(t)$ in funzione di $x(t)$ nel sistema di Lotka–Volterra. Le frecce indicano l'evoluzione temporale, i punti A, B, C, e D sono descritti nel testo.

```
20.0970001   1.74924719   0.923447967
20.1960011   1.75437796   0.958473265
20.2950001   1.75641012   0.995003879       <---
20.394001   1.75520515   1.03294861
20.493   1.75065255   1.07218719
20.592001   1.74267304   1.11257231
20.6910019   1.73122299   1.15392363

  ......
  ......
```

Possiamo calcolare il periodo della oscillazione osservata nello spazio delle fasi del sistemza, utilizzando questi dati. Usiamo la prima colonna e selezioniamo due massimi successivi, pari a $x_{max} \simeq 1.756$ (da leggere nella seconda colonna) indicati con il simbolo $<--$), che si ottengono negli istanti di tempo $t_1 = 7.3$ e $t_2 = 20.3$ rispettivamente (da leggere nella prima colonna). Allora il periodo del moto sará dato da $P = t_2 - t_1 = 13$, e quindi otteniamo la frequenza del moto circolare $\omega = 2\pi/P \simeq 0.48$ nello spazio delle fasi del sistema. Se si calcola il valore medio del numero di prede e di predatori su un periodo si scopre che questi sono sempre uguali ai rapporti fra i coefficienti, ossia

$$\frac{1}{P}\int_0^P x(t)dt = \frac{a}{c}$$
$$\frac{1}{P}\int_0^P z(t)dt = \frac{b}{e}$$

Questo é il contenuto della *seconda legge di Volterra*. Questa seconda legge puó essere facilmente verificata nella nostra simulazione.

Fino ad oggi il solo esempio di ecosistema che soddisfa alle equazioni di Lotka–Volterra, e quindi presenta una dinamica ciclica, é quello che concerne le lepri e le linci del Canada. Dal 1847 al 1903 la Hudson's Bay Company controlló la caccia alle linci ed al loro alimento primario, ossia le lepri, registrando il numero di esemplari cacciati. Questo numero rappresenta approssimativamente le due popolazioni. I dati ottenuti, estendendosi per un periodo molto lungo, hanno permesso di effettuare una statistica che, forse per il carattere chiuso dell'ecosistema, si é visto che presenta un carattere ciclico come descritto dal sistema. A parte l'interesse del modello di Lotka–Volterra come modello di ecosistema ottenuto da analogie con la fisica, il successo del modello é dovuto al fatto che esso permette di predire un fenomeno osservato da U. D'Ancona relativo alla interazione fra due tipi di pesci nell'Adriatico, ed alla variazione della loro popolazione durante la prima guerra mondiale. Questa osservazione costituisce la *terza legge di Volterra*.

3.7.1 I pesticidi in agricoltura

Il modello di Lotka–Volterra, ed in particolare la terza legge, puó essere usato per spiegare in modo brillante alcuni fenomeni apparentemente contraddittori osservati in ecologia. Nel 1868 l'insetto *Icerya Purchasi* parassita del cotone, entró negli Stati Uniti distruggendo le coltivazioni. Per ovviare in parte al problema si pensó di introdurre il predatore naturale, ossia la coccinella *Novius Cardinalis*, che infatti riuscí a contenere la crescita dell'insetto. Dopo un certo numero di anni la scoperta del DDT suggerí di ovviare definitivamente al problema. Il risultato fu peró opposto a quello sperato perché il numero degli insetti aumentó!

Se usiamo il sistema di Lotka–Volterra per descrivere gli insetti $x(t)$ e le coccinelle $z(t)$, l'intervento del DDT rappresenta una decrescita indiscriminata delle due popolazioni, decrescita che si puó modellizzare tramite un termine lineare proporzionale alla popolazione esistente, per cui

$$\begin{aligned} \frac{dx}{dt} &= ax - cxz - \epsilon x \\ \frac{dz}{dt} &= -bz + exz - \epsilon z \end{aligned} \tag{3.26}$$

dove ϵ é una costante arbitraria. Se usiamo lo stesso programma usato nella sezione precedente, ed aggiungiamo un termine proporzionale ad ϵ alle due equazioni possiamo fare una simulazione di questo nuovo caso. Per osservare il cambiamento usiamo una strategia del genere. Ossia facciamo un run fino ad un tempo $T = 100$ a passi di $\Delta t = 10^{-3}$, ma usiamo un valore di *epsilon* variabile. Ossia $\epsilon = 0.0$ per i primi 50 tempi, mentre $\epsilon = 0.25$ per i secondi 50 tempi. Un esempio di programma é riportato nel seguito.

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      Program pesticidi
c
      dimension y(4),dy(4)
      external dery
      common a,b,c,e,epsilon
c
      open(unit=16,file='pesticidi.dat',status='unknown')
      print *,'Inserisci i valori di time  dt  e tprint'
      read(5,*) time,dt,tprint
      nmax=int(time/dt)
      nprint=int(tprint/dt)
      neq=2
c
c   Condizioni iniziali
c
      y(1)=1.0
      y(2)=0.5
      a=0.5
      b=0.5
      c=0.5
      e=0.5
      epsilon=0.0
      write(16,*) 0.0,y(1),y(2)
c
      do n=1,nmax
        t0=(n-1)*Dt
```

```
          t=n*dt
          call RK2(neq,t0,dt,y,dy)
          if(mod(n,nprint).eq.0) then
            if(t.ge.50.0) epsilon = 0.25  ! TERMINE AGGIUNTO
            write(16,*) t,y(1),y(2)
          end if
        end do
        close(16)
        stop
        end
c
        subroutine Dery(t,y,dy)
        dimension y(4),dy(4)
        common a,b,c,e,epsilon
        dy(1)=a*y(1)-c*y(1)*y(2)-epsilon*y(1)
        dy(2)=-b*y(2)+e*y(1)*y(2)-epsilon*y(2)
        return
        end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

La soluzione numerica, ossia la dinamica di $x(t)$ e di $z(t)$ é mostrata nella figura 3.20. Come si vede chiaramente, a partire dal tempo $t = 50$ le due popolazioni cambiano drasticamente, ed in particolare il numero delle prede *aumenta di colpo*, come risulta dalle osservazioni dell'insetto, mentre i predatori decrescono. La spiegazione di ció consiste nel fatto che la situazione di equilibrio, in un periodo, varia con l'aggiunta del termine proporzionale ad ϵ. In particolare il valore medio delle prede diventa $(a+\epsilon)/c)$, e quindi aumenta di una quantitá ϵ/c, mentre quello dei predatori diventa $(b-\epsilon)/e)$, e quindi diminuisce di una quantitá ϵ/e. Lo stesso tipo di fenomeno, interpretato brillantemente da Volterra, era stato osservato da D'Ancona per la dinamica dei pesci dell'Adriatico, con la pesca indiscriminata che ha la stessa funzione dell'azione del DDT nell'esempio appena spiegato.

3.8 Il pendolo semplice smorzato e forzato

Il caso di un pendolo semplice soggetto ad una forza esterna periodica e ad una forza viscosa proporzionale alla velocit, é descritto dalla seguente equazione del moto per l'angolo $\vartheta(t)$

$$\frac{d^2\vartheta}{dt^2} + 2\gamma\frac{d\vartheta}{dt} + \omega_0^2 \sin\vartheta = F\cos\omega t \tag{3.27}$$

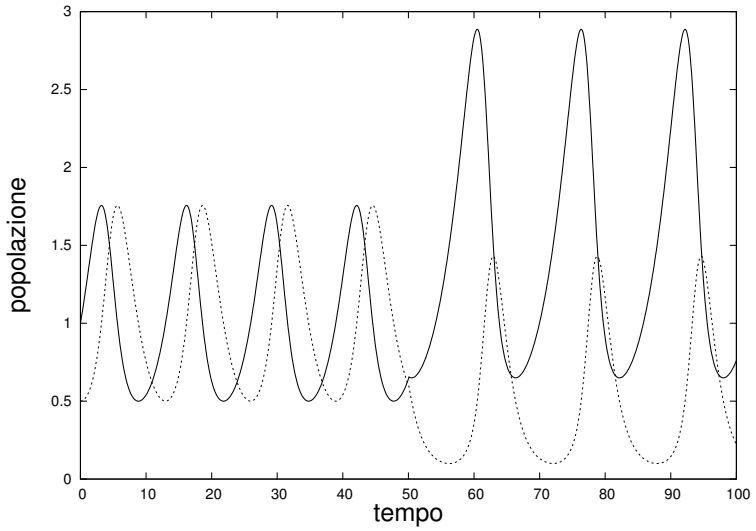


Figura 3.20: Evoluzione nel tempo delle variabile $x(t)$ (prede, linea continua) e $z(t)$ (predatori, linea tratteggiata) nel sistema di Lotka–Volterra con un termine aggiuntivo proporzionale ad ϵ (vedi testo).

dove γ é un coefficiente di smorzamento, ω_0 é la frequenza di oscillazione propria del sistema mentre ω é la frequenza della forza esterna.

E' ben noto che, nel caso lineare in cui si fa l'approssimazione di piccoli angoli $\sin\vartheta \simeq \vartheta$, la soluzione generale del problema é data analiticamente da

$$\vartheta(t) = Ae^{-\gamma t}\cos(\omega_0 t + \alpha) + B\cos(\omega t - \epsilon) \tag{3.28}$$

dove A e B sono due costanti di integrazione. Il moto sará quindi inizialmente smorzato con tempo caratteristico pari a $1/\gamma$, e quindi la forza esterna guida le oscillazioni. In questo caso lineare, dato un valore di F, l'ampiezza delle oscillazioni é massima alla risonanza, ossia quando la frequenza della forza esterna vale

$$\omega = \sqrt{\omega_0^2 - 2\gamma^2}$$

e l'ampiezza e la fase sono date da

$$\begin{aligned} B &= \frac{F}{\sqrt{(\omega^2 - \omega_0^2)^2 + 4\gamma^2\omega^2}} \\ \tan\epsilon &= \frac{\omega^2 - \omega_0^2}{2\gamma\omega} \end{aligned}$$

ossia dipendono dalla frequenza della forza esterna. Di seguito riportiamo un programma che simula l'equazione (3.27)

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Programma Principale
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      Program oscillatore_forzato
c
      parameter(neq=2)
      dimension y(neq),dy(neq)
      external dery
      common gamma2,W,W02,F
c
      open(unit=16,file='forzato.dat',status='unknown')
      print *,'Inserisci i valori di time dt e tprint'
      read(5,*) time,dt,tprint
      nmax=int(time/dt)
      nprint=int(tprint/dt)
c
c   Condizioni iniziali
c
      y(1)=1.0
      y(2)=0.0
      gamma=0.05
      gamma2=2.0*gamma
      W0=0.8                          ! Frequenza propria
      W02=W0*W0
      F=0.1
      W=0.8                           ! Frequenza forza esterna
      write(16,*) 0.0,y(1),y(2)
c
c   Ciclo temporale
c
      do n=1,nmax
        t0=(n-1)*Dt
        t=n*dt
        call RK2(neq,t0,dt,y,dy)
        if(mod(n,nprint).eq.0) then
           write(16,*) t,y(1),y(2)
        end if
```

```
      end do
      close(16)
c
      stop
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Subroutine che calcola il secondo membro dell'equazione
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine Dery(neq,t,y,dy)
      dimension y(neq),dy(neq)
      common gamma2,W,W02,F
c
      dy(1)=y(2)
      dy(2)=-gamma2*y(2)-W02*sin(y(1))+F*cos(W*t)
c
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Abbiamo eseguito un run del programma innanzitutto escludendo la forza esterna, ossia ponendo $F = 0$. Usiamo come valori $\gamma = 0.05$ e frequenza propria $\omega_0 = 0.8$, mentre il passo di integrazione é $\Delta t = 10^{-3}$. Facciamo andare il programma fino ad un tempo di $t = 100$. In figura 3.21 riportiamo l'andamento di $\vartheta(t)$ in funzione del tempo. Come si vede il sistema viene smorzato con un tempo caratteristico pari ad $\tau \simeq 1/0.05$. Infatti le ampiezze del moto sono ben descritte dalla legge $\exp(-t/\tau)$, come si vede dalla figura 3.21 in cui sono riportate anche le curve $\pm\exp(-0.05t)$. L'andamento nello spazio delle fasi $(\vartheta, d\vartheta/dt)$, che é rappresentato dal piano $(y(1), y(2))$ secondo le variabili interne al programma, é mostrato nella figura 3.22. Si osserva un moto a spirale, che tende verso il punto $(0, 0)$. Infatti cosí come l'angolo si smorza nel tempo, la velocitá del pendolo decresce verso zero.

Applichiamo adesso la forza esterna, pari a $F = 0.1$, ed usiamo come frequenza di oscillazione della forza esterna il valore $\omega = 1.5$, lasciando inalterati gli altri valori (facciamo andare il programma fino ad un tempo pari a $t = 200$). Nella figura 3.23 si vede come il sistema inizialmente viene smorzato. Per tempi lunghi peró esso viene guidato verso una oscillazione armonica perché l'oscillatore smorzato entra in risonanza con la forza esterna. Dall'analisi dei dati, come visto in precedenza, si puó misurare l'ampiezza e la frequenza. Facendo variare il parametro ω si puó vedere come l'ampiezza della oscillazione che si instaura per tempi lunghi dipende dalla frequenza

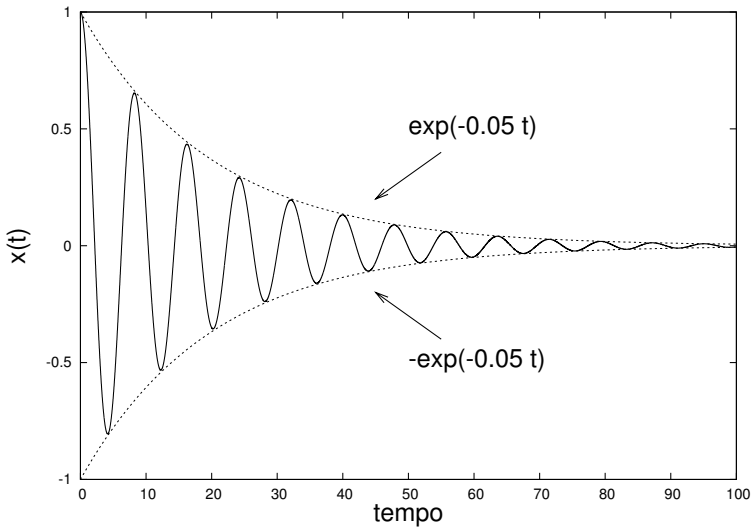


Figura 3.21: E' rappresentata l'evoluzione temporale di $x(t)$ in funzione del tempo per l'oscillatore armonico smorzato. Sono riportate anche le due curve analitiche $\pm \exp(-\gamma t)$ relative alla variazione dell'ampiezza.

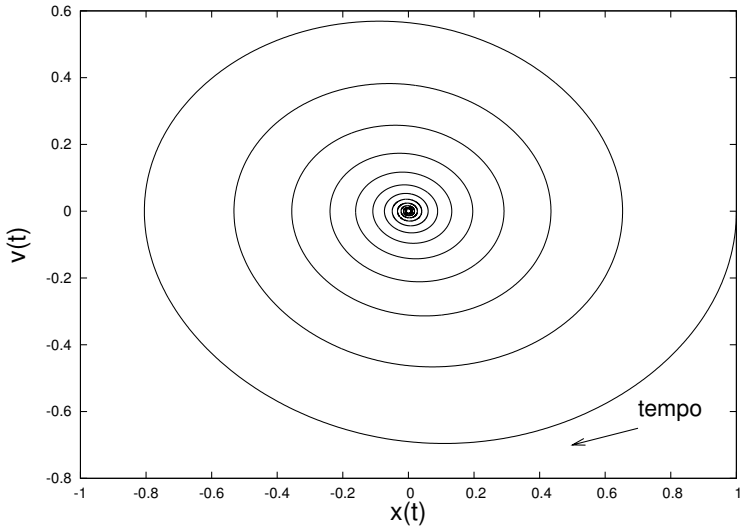


Figura 3.22: Evoluzione temporale nello spazio delle fasi $v = dx/dt$ in funzione di $x(t)$ nel caso dell'oscillatore armonico smorzato. Con una freccia é indicato l'andamento temporale.

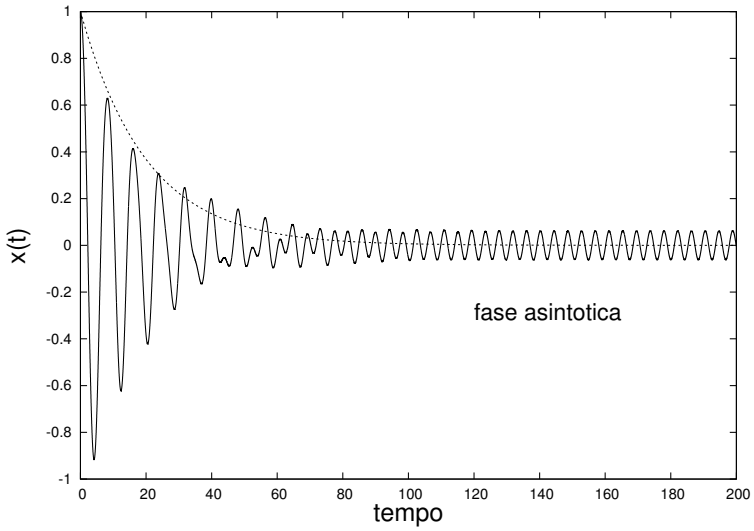


Figura 3.23: Evoluzione temporale di $x(t)$ per l'oscillatore armonico forzato e smorzato. Si nota la fase asintotica oscillante. E' anche riportato l'andamento della funzione $\exp(-\gamma t)$.

esterna del moto, ed ottenere una curva di risonanza simile a quella che si puó studiare analiticamente nel caso lineare.

3.9 Oscillatori accoppiati

Un altro classico problema riguarda gli oscillatori accoppiati che si risolve in modo analitico. Ha un valore storico perché stato il primo problema, usato da Enrico Fermi, per effettuare un test per il computer costruito a Chicago, e per questo é indicato come il sistema di Fermi–Pasta–Ulam. Questo consiste in un certo numero di masse accoppiate da molle. Vediamo il caso in cui ci siano due masse m_1 ed m_2, accoppiate da tre molle di costanti elastiche k, k_1 e k_2, come in figura. Le equazioni del moto per gli spostamenti x_1 ed x_2 dalle rispettive posizioni di equilibrio delle masse, saranno date dal seguente sistema

$$\begin{aligned} \frac{d^2x_1}{dt^2} &= -\left(\frac{k+k_1}{m_1}\right)x_1 + \left(\frac{k}{m_1}\right)x_2 \\ \frac{d^2x_2}{dt^2} &= -\left(\frac{k+k_2}{m_2}\right)x_2 + \left(\frac{k}{m_2}\right)x_1 \end{aligned} \tag{3.29}$$

Nel caso in cui le tre molle abbiano la stessa costante elastica k e le masse siano uguali ad m, i modi propri sono dati da $\omega^2 = k/m$ e da $\omega^2 = 3k/m$, rispettivamente se inizialmente gli spostamenti sono concordi in segno o discordi.

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Programma Principale
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      Program Fermi_Pasta_Ulam
c
      dimension y(4),dy(4)
      external dery
      real k,k1,k2,m1,m2
      common k,k1,k2,m1,m2
c
      open(unit=16,file='fermi_pasta_ulam.dat',status='unknown')
      print *,'Inserisci i valori di time  dt  e tprint'
      read(5,*) time,dt,tprint
      nmax=int(time/dt)
      nprint=int(tprint/dt)
      neq=4
      k=1.0
      k1=1.0
      k2=1.0
      m1=1.0
      m2=1.0
c
c   Condizioni iniziali
c
      y(1)=0.5
      y(2)=0.0
      y(3)=-1.0
      y(4)=0.0
c
      do n=1,nmax
        t0=(n-1)*Dt
        t=n*dt
        call RK2(neq,t0,dt,y,dy)
        if(mod(n,nprint).eq.0) then
          write(16,*) t,y(1),y(2),y(3),y(4)
```

```
        end if
      end do
      close(16)
c
      stop
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
c   Subroutine che calcola il secondo membro dell'equazione
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      subroutine Dery(ti,y,dy)
      dimension y(4),dy(4)
      real k,k1,k2,m1,m2
      common k,k1,k2,m1,m2
c
      dy(1)=y(2)
      dy(2)=-((k+k1)/m1)*y(1)+(k/m1)*y(3)
      dy(3)=y(4)
      dy(4)=-((k+k2)/m2)*y(3)+(k/m2)*y(1)
c
      return
      end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Abbiamo risolto numericamente il sistema di equazioni nel caso semplificato in cui le costanti elastiche e le masse sono tutte uguali $k = k_1 = k_2 = m_1 = m_2 = 1.0$, utilizzando come condizioni iniziali due casi. Nel primo (caso A) abbiamo posto $x_1(0) = 0.5$ ed $x_2(0) = 1.0$, mentre nel secondo caso abbiamo invertito il moto della seconda molla, ossia abbiamo posto $x_1(0) = 0.5$ ed $x_2(0) = -1.0$ (caso B). In entrambi i casi le velocitá iniziali sono state poste uguali a zero.

Nella figura 3.24 é rappresentata l'evoluzione temporale di $x_1(t)$ nei casi in cui le condizioni iniziali per gli spostamenti hanno lo stesso segno (curva punteggiata), e nel caso in cui i segni sono opposti (curva continua). Come si vede l'evoluzione nel tempo é oscillante in entrammbi i casi, ma la frequenza varia nei due casi. La frequenza nel caso in cui le condizioni iniziali sono dello stesso segno é maggiore che nel caso opposto. Se ω é questa frequenza (nel caso A), allora si puó dimostrare (anche numericamente) che la frequenza nel caso B é 3ω. La stessa cosa capita per $x_2(t)$, come si vede dalla figura 3.25.

Nella figura 3.26 é rappresentato lo spazio delle fasi (x_1, v_1). Si vede che l'oscillazione ha una ampiezza non costante. E' ovviamente possibile studiare

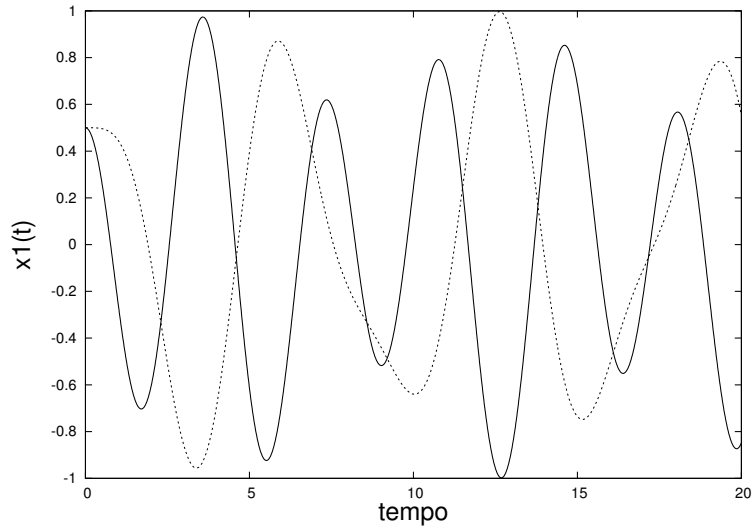


Figura 3.24: E' rappresentata l'evoluzione temporale di $x_1(t)$ nei casi A (curva punteggiata) e nel caso B (curva continua).

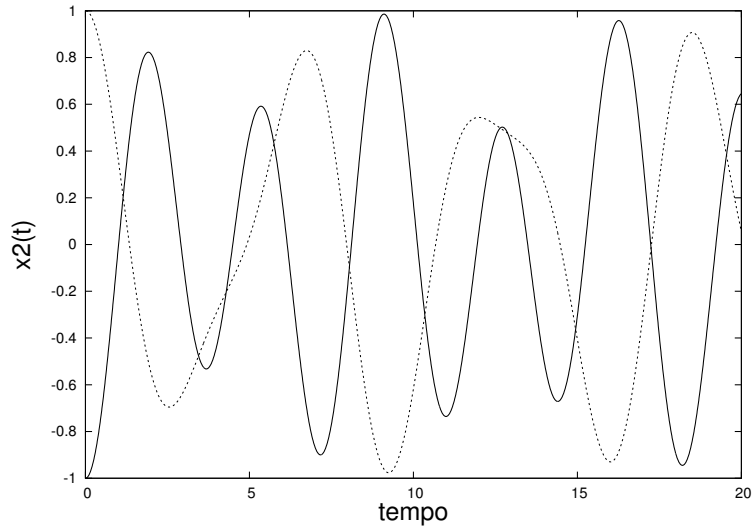


Figura 3.25: E' rappresentata l'evoluzione temporale di $x_2(t)$ nei casi A (curva punteggiata) e nel caso B (curva continua).

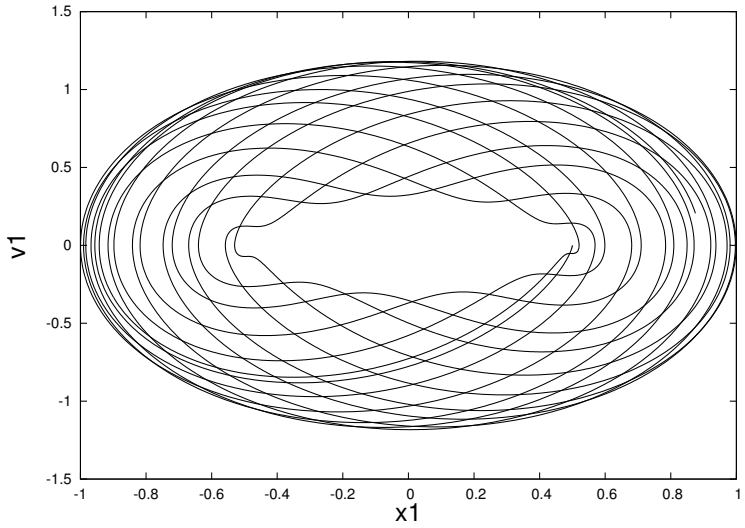


Figura 3.26: E' rappresentata l'evoluzione temporale dello spazio delle fasi $v_1(t)$ in funzione di $x_1(t)$, nel caso A.

altri tipi di configurazione con condizioni iniziali qualsiasi, a discrezione dello studente.

3.10 Particella in una doppia buca di potenziale

Questo sistema fisico é ben noto. Il moto unidimensionale di una particella di massa unitaria in una doppia buca simmetrica di potenziale é descritto dalla seguente equazione del moto

$$\frac{d^2x}{dt^2} = \alpha x \left(1 - \frac{x^2}{x_0^2}\right) \tag{3.30}$$

dove α ed x_0 sono due costanti. Il problema si studia se si considera l'equazione del potenziale associato

$$U(x) = \frac{\alpha}{2}x^2 \left(1 - \frac{x^2}{2x_0^2}\right) \tag{3.31}$$

da cui si ottiene immediatamente che i punti critici sono $x = 0$ (punto instabile) e le due buche a $\pm x_0$, che sono i punti stabili. Inoltre $U(x) = 0$ quando $x = \pm x_0\sqrt{2}$. Fissata allora l'energia E della particella si ottiene che una particela con velocitá iniziale $\dot{x}(0) > \sqrt{2E}$ compie un moto oscillatorio nella

doppia buca, mentre se $\dot{x}(0) < \sqrt{2E}$ il moto avviene in una sola delle buche. Questo si puó osservare bene se usiamo il seguente programma per integrare le equazioni del moto.

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
      program doppiabuca
      dimension y(2),dy(2)
      external dery
      common alfa,x0
c
      open(unit=11,file='doppiabuca.dat',status='unknown')
      print *,'Inserisci i valori di time   dt   e tprint'
      read(5,*) time,dt,tprint
c
      nmax=int(time/dt)
      nprint=int(tprint/dt)
      neq=2
c
c   Costanti
c
      alfa=1.0
      x0=1.0
c
c   posizione e velocita' iniziale
c
      extr=sqrt(2.0)*x0
      print *,'Dai la posizione iniziale nella buca'
      print *,'gli estremi sono'
      print *,-extr,extr
      read(5,*) y(1)
      U=0.5*y(1)*y(1)*alfa*(1.0-y(1)*y(1)/(2.0*x0*x0))
      ucr=sqrt(U)
      print *,'Dai il valore della velocita iniziale'
      print *,'La velocita critica e',ucr
      read(5,*) y(2)
c
      open(11,file='bistabile.dat',status='unknown')
      write(11,*) 0.d0,y(1),y(2)
c
      do n=1,nmax
        t0=(n-1)*dt
```

```
          t=n*dt
          call RK2(neq,t0,dt,y,dy)
          if(mod(n,nprint).eq.0) then
            write(11,*) t,y(1),y(2)
          end if
        end do
        close(11)
c
        stop
        end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
        subroutine Dery(t,y,dy)
        dimension y(2),dy(2)
        common alfa,x0
c
        term=y(1)*y(1)/(x0*x0)
c
        dy(1)=y(2)
        dy(2)=alfa*y(1)*(1.d0-term)
c
        return
        end
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Infatti usiamo i parametri $x_0 = 1.0$, ed $\alpha = 1.0$, si vede che i punti in cui si annulla il potenziale sono $\pm\sqrt{2}$. Partiamo allora al di sotto di uno di questi punti, ossia $x(0) = -1.0$, e con una velocitá iniziale pari a quella critica, ossia $\dot{x}(0) = 0.5$. Si vede una oscillazione semplice (figura 3.27). In figura 3.28 é mostrato lo spazio delle fasi (x, v) di questo caso. Se diminuiamo $x(0)$ ed aumentiamo la velocitá, si vede che la particella si avvicina sempre di piú ad $x = 0$ con velocitá nulla. Se usiamo $x(0) = -\sqrt{2}$ ed una velocitá iniziale superiore a quella critica, ossia $\dot{x}(0) = 0.1$, si vede un moto nella doppia buca (figura 3.29). Anche in questo caso in figura 3.30 é mostrato lo spazio delle fasi (x, v)

Ma la cosa interessante, per il calcolo numerico é partire dal punto $x_0 = -\sqrt{2}$ con velocitá iniziale leggermente differente dalla velocitá critica, ossia $\dot{x}(0) = \pm\epsilon$, con $\epsilon \ll 1$ e un segno qualsisi. Si vede che, a causa degli errori inerenti al calcolo numerico la aggiunta della ϵ consente alla particella di transire da una buca all'altra, anche se fisicamente non dovrebbe farlo. Se per esempio usiamo $\epsilon = -10^{-6}$ si ottiene un moto oscillatorio che durante

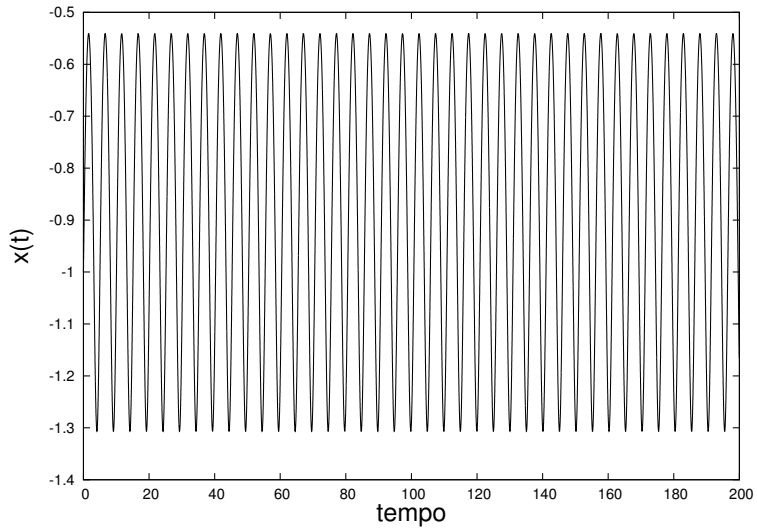


Figura 3.27: E' rappresentata l'evoluzione temporale di $x(t)$ nel caso in cui le condizioni iniziali $x(0) = -1.0$ e $\dot{x}(0) = 0.5$, ossia tali per cui il moto avviene nella singola buca.

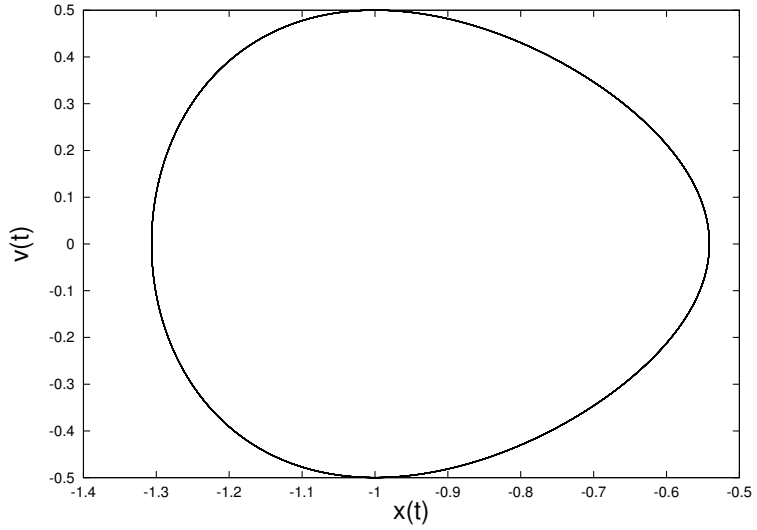


Figura 3.28: E' rappresentata l'evoluzione temporale dello spazio delle fasi $\dot{x}(t) = v(t)$ in funzione di $x(t)$, nel caso in cui le condizioni iniziali sono tali per cui il moto avviene nella singola buca.

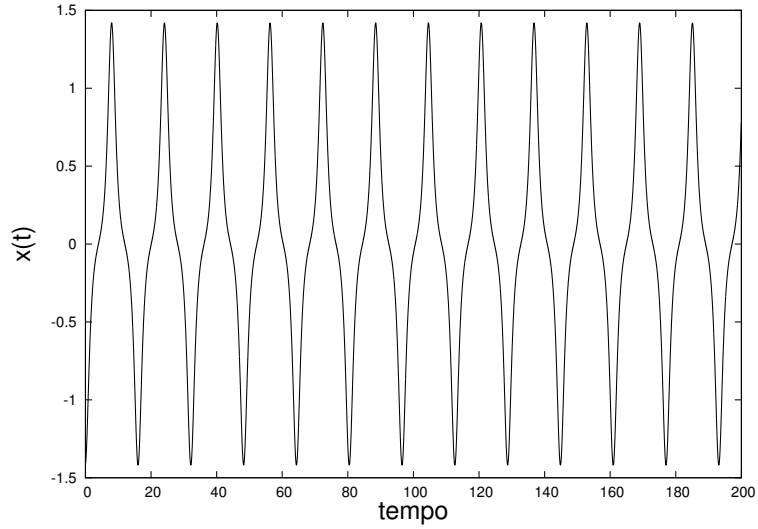


Figura 3.29: E' rappresentata l'evoluzione temporale di $x(t)$ nel caso in cui le condizioni iniziali sono $x(0) = -\sqrt{2}$ e $\dot{x}(0) = 0.1$, tali per cui il moto avviene nella doppia buca.

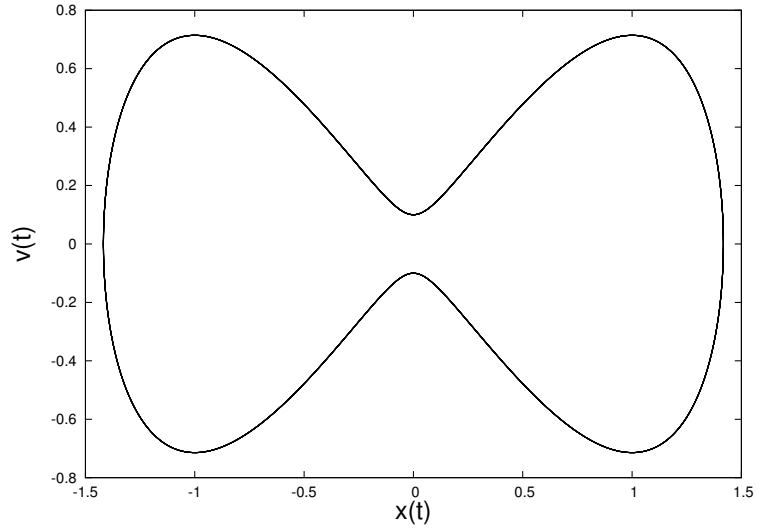


Figura 3.30: E' rappresentata l'evoluzione temporale dello spazio delle fasi $v(t)$ in funzione di $x(t)$, nel caso in cui le condizioni iniziali sono tali per cui il moto avviene nella doppia buca.

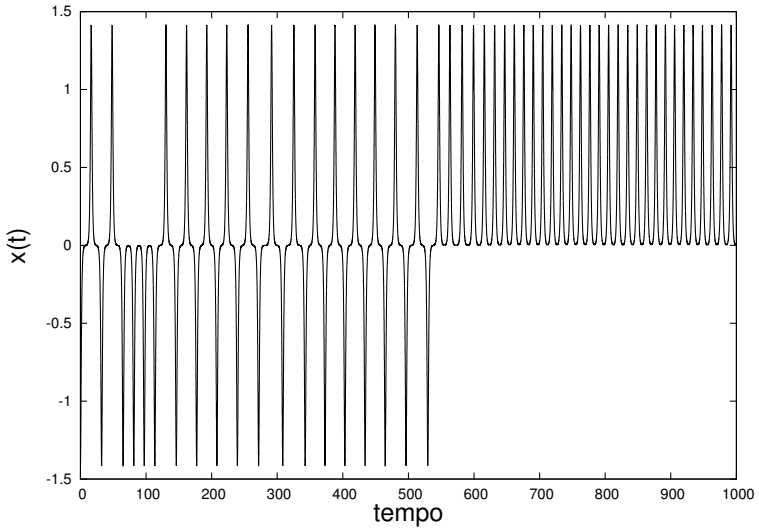


Figura 3.31: E' rappresentata l'evoluzione temporale di $x(t)$ nel caso in cui le condizioni iniziali sono $x(0) = -\sqrt{2}$, mentre la velocitá iniziale é posta pari a $\dot{x}(0) = 10^{-6}$.

alcuni periodi di tempo si svolge in una sola buca, mentre durante altri periodi si svolge nelle due buche, come si vede dalla figura 3.31.

La cosa impressionante é che anche se si usa un valore *numerico* $\dot{x}(0) = 0.0$ si ottiene il passaggio nell'altra buca a causa degli errori di calcolo (vedi figura 3.32).

Questo é vero anche se si fa il programma in doppia precisione e si parte da $x_0 = -\sqrt{2}$ ed $\dot{x}(0) = 0.0$, basta aspettare un tempo sufficientemente lungo (dell'ordine di 1500) ed il sistema comincerá ad oscillare nella doppia buca (vedi figura 3.33).

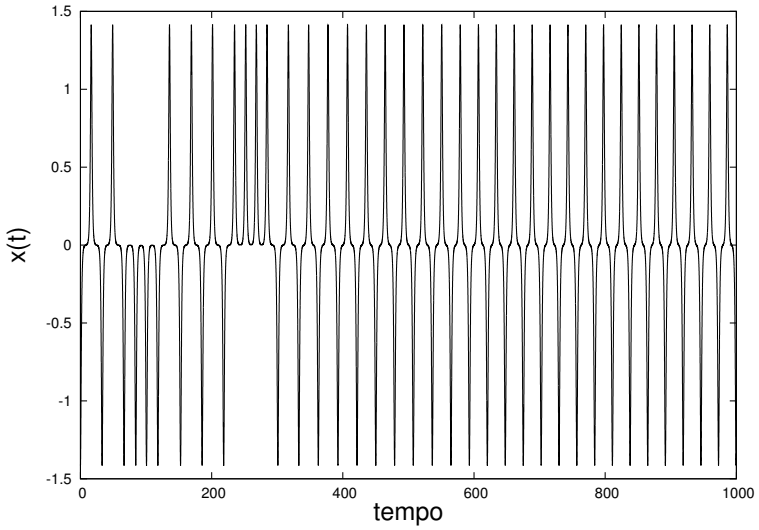


Figura 3.32: E' rappresentata l'evoluzione temporale di $x(t)$ nel caso in cui le condizioni iniziali sono $x(0) = -\sqrt{2}$, mentre la velocitá iniziale é posta pari a $\dot{x}(0) = 0$. E' rappresentato l'output del programma in singola precisione.

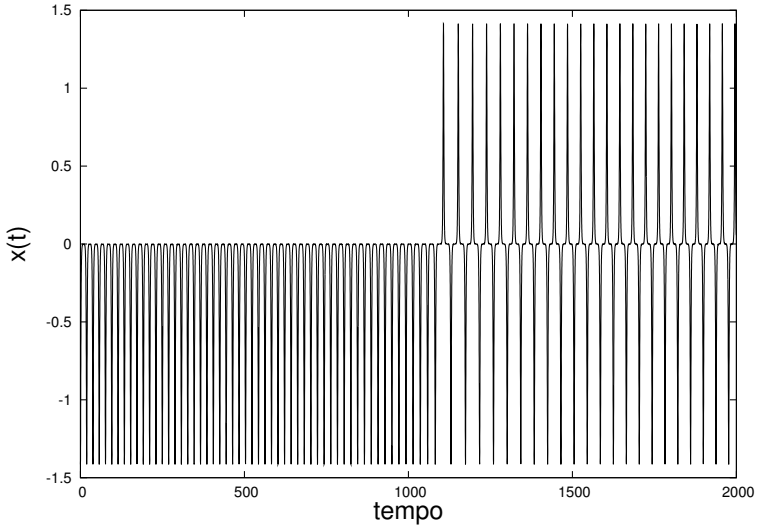


Figura 3.33: E' rappresentata l'evoluzione temporale di $x(t)$ nel caso in cui le condizioni iniziali sono $x(0) = -\sqrt{2}$, mentre la velocitá iniziale é posta pari a $\dot{x}(0) = 0$. E' rappresentato l'output del programma in doppia precisione.

3.11 Letture consigliate relative al calcolo numerico

La seguente bibliografia può essere di ausilio per approfondire gli argomenti di calcolo numerico trattati durante il corso:

1. Landau, L. D., Lifŝits, E. M., "Meccanica", III edizione, Editori Riuniti — Edizioni Mir, Roma — Mosca, Italia — URSS, 1991.

2. Scheid F., "Analisi Numerica", I edizione, Mc Graw Hill Ed., Milano, Italia, 1994.

3. Press, W. H., Teukolsky, S. A., Vetterling, W. T., Brian, P. F., "Numerical Recipes in FORTRAN: The Art of Scientific Computing", II edizione, Cambridge University Press, Cambridge, USA, 1992.

4. Turchetti G., "Dinamica classica dei sistemi fisici", I edizione, Zanichelli Ed., Bologna, Italia, 1998.

5. Queen, N. M., "Methods of applied mathematics", I edizione, Thomas Nelson & Sons Ed., Hong Kong, 1980.

Parte IV

Il sistema operativo LINUX

Capitolo 4

LINUX

4.1 Introduzione

Linux é un sistema operativo versatile: puó essere usato da studenti, ricercatori e programmatori per scrivere documenti, eseguire brevi calcoli oppure sviluppare programmi che fanno simulazioni complesse, é anche installato sui computer di imprese e pubbliche amministrazioni ed usato per esempio per gestire database o per realizzare server web.

Questo sistema operativo può essere perciò installato sia su workstation che su server di calcolo e può inoltre essere usato per realizzare servizi su Internet (siti web, posta elettronica, firewall, ecc) o sistemi integrati per l'automazione (sistemi di controllo presenze ed allarme, ecc.). Insomma, conoscere Linux è diventato una necessità per chi vuol proporsi nel mondo del lavoro in modo da essere più appetibile di altri concorrenti.

In questo capitolo spiegheremo come sfruttare al meglio le potenzialità di Linux nello sviluppo di applicazioni, cercando di utilizzare il meno possibile l'interfaccia grafica: è questa a mio parere la marcia in più che deve avere chi conoscerealmente il sistema operativo. Lasceremo al lettore il compito di studiare le applicazioni comerciali.

4.2 Per iniziare subito

La prima cosa da procurarsi è sicuramente una *username* ed una *password* che bisogna chiedere all'amministratore di sistema. LINUX permette di usare il computer (in gergo si dice *fare il login*) solo se hai un username, che dev'essere inserita alla richiesta di **login:** e di seguito bisogna fornire la password.

A questo punto in molti sistemi si aprirà una schermata grafica che mostrerà un desktop molto simile a quello di Windows, altrimenti, se state inizian-

do il lavoro da una console, appariranno dei messaggi di benvenuto e poi il prompt dei comandi.

Non sempre è possibile accedere ad un terminale grafico del computer, perciò da questo momento in poi ci occuperemo della modalità testo per l'utilizzo della macchina LINUX, questo ci permetterà anche di conoscere meglio il sistema operativo.

4.3 Interprete dei comandi o SHELL

Non appena si è fatto il login, il sistema mostrerà il prompt della *shell* che è generalmente un segno di dollaro($) oppure un percento (%). A questo punto la fase di *login* è terminata con successo ed il sistema è pronto a ricevere un comando. In realtà d'ora in poi, quando si digita qualcosa da tastiera e poi si preme il tasto di *INVIO* la stringa immessa verrà interpretata dalla *shell* e, nel caso venga riconosciuta come comando, verrà eseguito il programma corrispondente del sistema operativo.

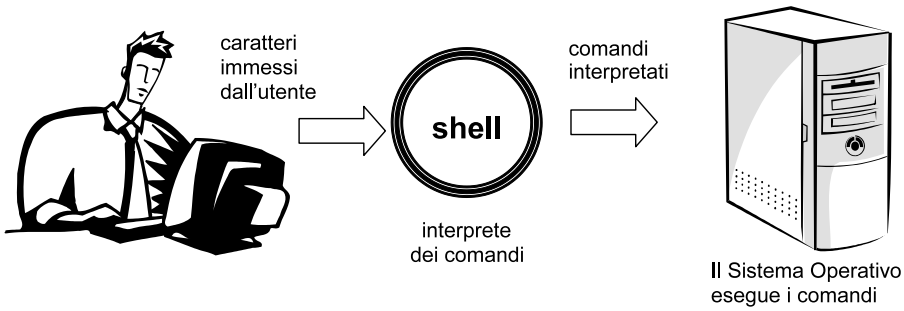


Figura 4.1: La shell: interfaccia fra utente e sistema operativo

Esistono quattro *shell* principali e per sapere quale si sta usando basta eseguire il comando

```
$ echo \$SHELL
```

se l'output di `echo $SHELL` è	allora si usa
/bin/sh	Bourne Shell
/bin/bash	Bash (Bourne Again Shell)
/bin/csh	C Shell
/bin/tcsh	Enhanced C Shell

Ogni volta che si digita un comando:

1. la shell interpreta la linea di comando e cerca nel disco finchè non trova il programma richiesto

2. la shell ancora chiede al sistema operativo di eseguire il programma e trasferisce il controllo al sistema operativo stesso

3. il sistema operativo copia il programma dal disco in memoria ed incomincia ad eseguirlo, così facendo crea un processo,

4. ad ogni processo il sistema operativo assegna un numero di identificazione (Process Identifier o PID)

   Per vedere quali sono i processi attualmente in esecuzione su un sistema usare il comando `ps --ef`

5. quando un programma completa l'esecuzione il controllo viene trasferito nuovamente alla shell ed il processo muore.

4.4 Incominciamo con due comandi

Il primo comando da usare dopo aver completato il login è **passwd**. L'uso della password aiuta a impedire l'uso del computer da parte di utenti non autorizzati, perciò è conveniente non usarne di banali (es. il proprio cognome oppure il nome o la data di nascita) ed è consigliato cambiarla al primo collegamento sulla stazione di lavoro.

Questo comando chiede di inserire la vecchia password, se ne avete una, e poi la nuova password che viene richiesta due volte per questioni di sicurezza.

```
$passwd
Enter old password:
Enter new password:
Re-enter new password:
```

Il secondo comando è quello che permette la chiusura della sessione di lavoro, anche questo è molto importante. Ci sono in genere due possibilità: usare il comando **exit** oppure premere simultaneamente i tasti **CTRL** e **d**.

4.5 File e directory (documenti e cartelle)

Un file è un contenitore posto in una memoria del computer, generalmente un disco, in esso sia l'utente che il sistema operativo possono registrare informazioni che possono essere di natura variabile (numeri, testi, programmi, immagini, suoni) queste informazioni sono caratterizzate dal fatto di essere espresse in formato binario. Alcune di queste informazioni sono leggibili

dall'utente, generalmente costituiscono file di testo, altre invece sono informazioni illegibili, e sono contenute ad esempio nei file eseguibili; tutti i file sono però scritti su disco facendo uso della codifica binaria. E' possibile, tramite i comandi del sistema operativo oppure tramite una applicazione creare files che contengono una gran varietà di cose.

Anche le directories sono contenitori, servono però a contenere files o altre directories in modo tale da permettere di organizzare le informazioni in gruppi logicamente correlati.

Le directories hanno una struttura gerarchica, cioè ognuna ha una directory genitore localizzabile "sopra di essa" e può avere una directory figlia o sotto-directory. Allo stesso modo, ogni sotto-directory può contenere files e può anche avere più sotto-directory.

Questa struttura è molto simile a quella di un albero rovesciato: al primo livello c'è la directory radice (*root directory*) che si indica con / ed ogni suo ramo rappresenta invece una sotto-directory. Si chiama cammino di ricerca (*path name*) l'elenco dei nomi di directory per giungere ad un file in una sotto-directory

Ecco il significato delle directory principali, che sono mostrate in figura 4.2:

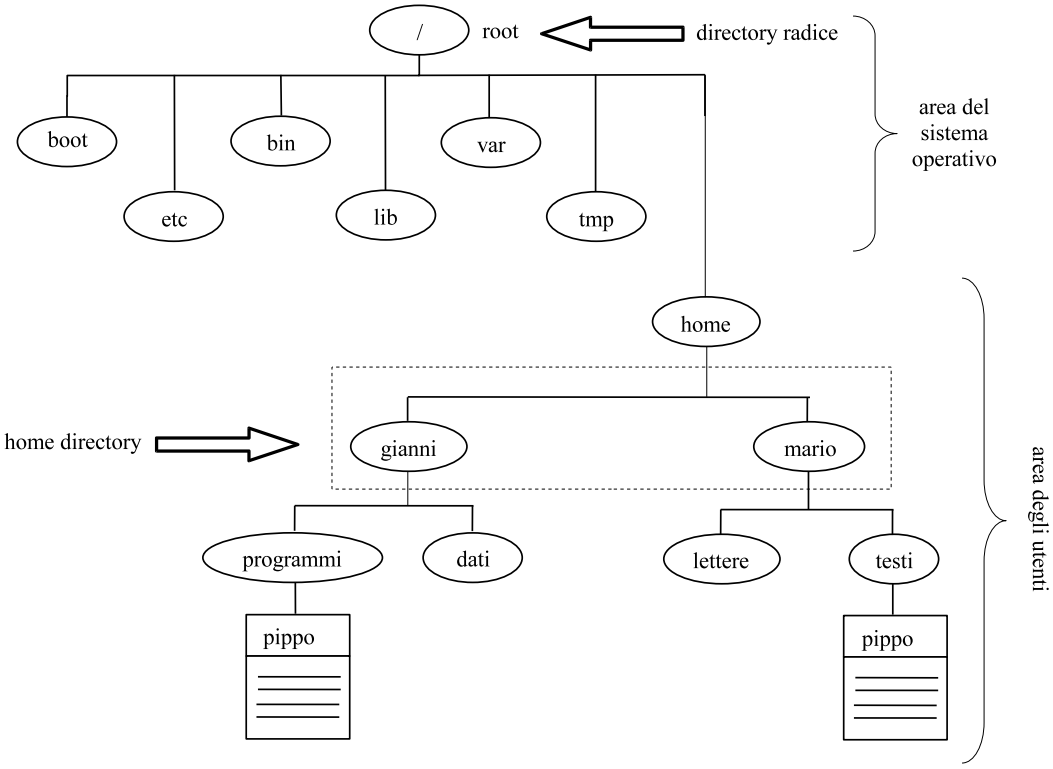


Figura 4.2: Albero delle directory di un tipico sistema LINUX

/	è la radice dell'albero delle directory e si chiama *root directory*.
bin	contiene i comandi di base
sbin	contiene comandi per la gestione del sistema, di solito usati solo dall'utente root
dev	qui si trovano i file di descrizione delle periferiche: dischi rigidi, stampanti, modem ed altro
etc	contiene i file per la configurazione del sistema
lib	qui ci sono le librerie condivise dai programmi
tmp	contiene file temporanei che possono essere cancellati
usr	contiene la maggior parte dei programmi esistenti nel sistema, perciò è organizzata in sottodirectory: *bin*, *doc*, *include*, *lib*, ecc. Contiene anche il software per l'ambiente grafico *X11*
var	contiene vari tipi di dati, tra cui anche le code di stampa
home	qui ci sono tutte le directory di base (*home directory*) degli utenti

Al termine delle operazioni di login il sistema operativo, se non ci sono altre disposizioni, ogni volta che bisogna accedere alle informazioni dei file farà riferimento ad una directory che per default è la *home directory* che puó essere indirizzata usando il carattere ˜.

Per poter indicare un file bisogna specificare il cammino di ricerca nell'albero delle directory (*path name*) perciò */home/mario/testi/pippo* farà riferimento al *file* di nome pippo, il cui cammino di ricerca è */home/mario/testi* questo permette di distinguerlo da un altro file di nome *pippo* che ha però il cammino di ricerca */home/gianni/programmi/file*.

Il comando per conoscere il *path name* della directory in cui si sta lavorando è **pwd** mentre per ottenere la lista dei file contenuti in essa si usa il comando **ls**

Esempio:

```
$ ls
pippo              topolino.txt
pluto.exe          paperone.dat
prova              a.lettura
```

Esistono due file nascosti, che non vengono mostrati dal solo comando *ls*: il file . (punto) che è un puntatore alla directory corrente ed il file .. (punto punto)è il puntatore alla directory genitore che è immediatamente sopra alla directory corrente.

Per muoversi nell'albero delle directory si usa il comado `cd` *(change directory)*

esempio:

```
$ cd /home/mario/testi
```

comando alternativo:

```
$ cd /home/mario
$ cd testi
```

questo esempio permette di posizionarsi nella sottodirectory */home/mario/testi*
`cd ..` permette invece di salire di un livello nell'albero delle directory
i comandi `cd` oppure `cd ~` permettono di spostarsi nella propria home directory.

Ad esempio, se la home directory è */home/mario*

`$ cd ~/testi`	accederà alla directory */home/mario/testi*, invece
`$ ls`	permette di visualizzare la lista dei file contenuti nella directory corrente
`$ ls testi`	se la *working directory* è */home/mario* permette di avere la lista dei file contenuti nella directory *testi*
`$ ls ..`	permette di ottenere la lista dei file contenuti nella directory di livello superiore

Il comando per creare directory è *mkdir*, così per creare la subdirectory *progetti* bisogna eseguire

```
$ mkdir progetti
```

il comando `ls` permette di verificare che il comando abbia funzionato

```
$ ls
pippo progetti/
```

Per spostare file da una directory ad un'altra, oppure per cambiarne il nome, il comando è *mv*, mentre per copiare i file si usa *cp*
Esempio

`$ mv pippo progetti` sposta il file *pippo* nella directory *progetti*

`$ mv pippo nuovopippo` cambia il nome di *pippo* in *nuovopippo*

altro esempio

`$ cp pippo progetti/pippo1` copia il file *file* nella directory progetti e gli assegna il nome *file1*

Nota: quando si spostano o copiano file bisogna far attenzione a non sovrascrivere file preesistenti.

Per cancellare un file si usa il comando *rm*, mentre per una directory si usa *rmdir*
Esempio:

```
$ rm progetti/nuovopippo progetti/pippo1
$ rm -R progetti
```

si puó usare, in alternativa, il comando

```
$ rmdir progetti
```

4.5.1 Il wildcard *

Il simbolo * serve a sostituire uno o più caratteri: questo permette di selezionare con un singolo riferimento più file o directory.

Ad esempio può essere usato per per elencare tutti i file il cui nome comincia per *pippo* all'interno di una directory:

```
$ ls
pappa/      miopippo    paperone    pippo       pippo1
pippo2      pippomio
$ ls pappa*
pippo     pippo1    pippo2    pippomio
```

Riassumiamo i comandi discussi fin'ora:

`$ cd`	sposta nella home directory
`$ ls` `file1 file2 progetti/`	mostra i file e le directory

`$ rmdir progetti`
 	
`rmdir: progetti not empty`
	prova a cancellare una directory non vuota

`$ cd progetti`	si sposta nella directory progetti
`$ ls` `testo1 testo2 prova3`	
`$ rm *`	rimuove tutti i fles (si noti l'uso del wildcard *)
`$ cd ..`	sale di un livello nell'albero delle directory
`$ rmdir progetti`	ora può cancellare la directory progetti

4.5.2 I permessi dei file e delle directory

Nel sistema operativo Linux ogni utente è associato ad un gruppo (inteso come gruppo di lavoro), perciò esistono tre classi di utenti che possono accedere ai file ed alle directory: *owner* (il proprietario), *group* (gli utenti che fanno parte dello stesso gruppo del proprietario) e *other* (tutti gli altri). I tipi di accesso sono:
in lettura (*read*), in scrittura (*write*) ed in esecuzione (*execute*).

Ogni file/directory ha uno o più modi di accesso per ogni classe di utenti.

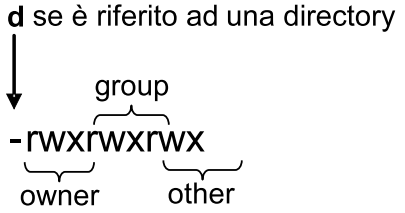
Se l'utente *gianni* facente parte del gruppo *users* immette il comando

```
$ ls -l
```

mostrerà

```
 -rw-r---r--  1  gianni  users   154   mar 12 11:13  pippo
 -rw-r---r--  1  gianni  users   154   mar 12 11:13  pippo1
 drwxr-x---   1  gianni  users   154   mar 12 11:13  progetti
 permessi        owner   group  size  data          filename
```

I permessi sono riferiti a file e directory e permettono un accesso differenziato per *owner*, *group* ed *other*. L'accesso può essere in lettura *r*, scrittura *w* ed esecuzione *x*:



Se un accesso non è consentito apparirà un trattino (-), perciò

- il file *pippo* può essere letto e scritto, ma non eseguito dal proprietario (*user owner*), solo letto dal gruppo cui fa parte il proprietario del file, solo letto da tutti gli altri utenti.
- il file *pippo1* può essere letto e scritto, ma non eseguito dal proprietario (*user owner*), solo letto dal gruppo cui fa parte il proprietario del file, solo letto da tutti gli altri utenti.
- la directory *progetti* può essere acceduta dal proprietario per leggere e scrivere file ed anche per fare una ricerca (*execute*) cioè per ottenere la lista dei file contenuti, mentre il gruppo del proprietario può solo leggere i file e ottenere la lista dei contenuti e tutti gli altri utenti non hanno accesso alla directory.

Si noti che i permessi d'accesso ad un file dipendono dai permessi della directory in cui è locato il file.

Se per esempio si permette l'accesso ad un file in lettura, scrittura ed esecuzione a tutti col comando

```
$ chmod u=rwx, g=rwx, o=rwx pippo
```

questo file non potrà essere acceduto da un utente finchè questi non avrà accesso in lettura ed esecuzione anche alla directory che contiene il file.

È possibile cambiare le protezioni di file e direttory usando il comando *chmod*

Ad esempio, per permettere a tutti di scrivere il file *pippo* dell'esempio, il comando è

```
$ chmod o+w pippo
$ ls -l
-rw-r--rw-  1 pippo users   154 Mar 12 11:13 pippo
```

Un altro esempio di utilizzo di *chmod* è il seguente:

```
$ chmod u=rwx, o-w file
```

questo comando imposta i permessi di accesso del proprietario del file a *read*, *write* ed *execute* ed elimina l'accesso in scrittura per *other*, gli utenti che non fanno parte del gruppo del proprietario del file.

4.5.3 File con più nomi (i link)

Un link è una connessione tra un nome di file ed un file generico. Di solito un file ha un solo link (connessione) al suo nome, tuttavia è possibile usare il comando *ln* per dare ad un file più nomi contemporaneamente. I link sono utili, ad esempio, per evitare di replicare il contenuto di file inutilmente, ma si tenga presente che le modifiche relative ad un *file name* saranno fatte in realtà sul file originale. I textitsoft link, chiamati anche link simbolici permettono di connettere sia file che directory e contengono un puntatore ad un altro file o directory. Questo puntatore è il pathname che porta al file/directory originale.
La sintassi del comando per il link simbolico è la seguente:

```
$ n -s /path_1/file_originale /path_2/nome_del_link
```

dove
`/path_1/file_originale` è il rifermento al file esistente,

mentre
`/path_2/nome_del_link` si riferisce al collegamento da creare.

È possibile rimuovere un link simbolico e non il rifermento originale, usando il comando *rm* seguito dal nome del link

Facciamo un esempio:

```
$ pwd
/home/mario
$ ls -l testi
-rw-rw-r-- 1 mario users 20 giu 13 13:59 pippo
$ ls -l lettere
totale 0
$ ln -s testi/pippo lettere/prova
$ ls -l lettere
lrwxrwxrwx 1 mario users 21 giu 14 10:12 prova -> testi/pippo
$ rm lettere/prova
$ ls -l lettere
totale 0
$ ls -l testi
-rw-rw-r-- 1 mario users 20 giu 13 13:59 pippo
```

4.6 Gli editor di testo: il vi

Per creare un file di testo è necessario usare qualche programma di manipolazione testi. In ambiente grafico X11 sono disponibili programmi come *gedit* o *xemacs* il cui uso è piuttosto intuitivo, mentre con un terminale alfanumerico si potrà usare ad esempio il *vi* (*Visual Interactive editor*) e noi discuteremo proprio di quest'ultimo programma.

Quando viene eseguito il comando *vi* si entra nel modo comandi dell'editor ed a questo punto si digita uno dei tasti che fanno accedere al modo testo in cui si può scrivere il testo del file, mentre premendo `ESC` si ritorna al modo comandi.

Perciò ad esempio se si esegue il comando *vi nomefile* si entra in modalitá comandi finchè non si digita il tasto *i* che permette di inserire il testo prima del cursore, mentre la pressione del tasto *a* permette di aggiungere il testo a partire dalla posizione del cursore e con il tasto *x* si cancella un carattere. I tasti *h*, *j*, *k*, *l* permettono di spostare il cursore nel testo:

- *l* muove il cursore di un carattere a destra
- *h* muove il cursore di un carattere a sinistra

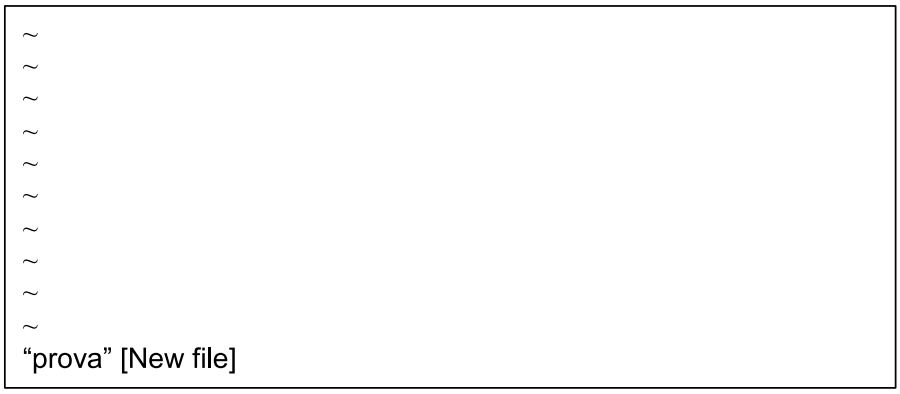


Figura 4.3: Schermata di output dopo il comando `vi prova`

- k muove il cursore una linea sopra
- j muove il cursore una linea sotto

Infine nella modalitá comando, se si digita *:wq* si salva il file (w) e poi si esce dall'editor (q)

Per fare un esempio, eseguiamo il comando:

```
$ vi prova
```

la schermata che il sistema ci presenta è quella di figura 4.3. Premendo il tasto i si entra in modalitá *inserimento* perció si può immettere la seguente linea di testo:

```
Esempio di testo
```

e la schermata ci apparirà come in figura 4.4, nella parte bassa dello schermo il *vi* evidenzia il fatto che è in modalità di inserimento testo.

Per salvare il file si deve cambiare modalità premendo il tasto **ESC**, assicuriamoci che in basso allo schermo non appaia più nulla e poi possiamo digitare due caratteri **ZZ** debbono essere zeta maiuscole. Altri modi di salvare sono elencati nella tabella mostrata piú sotto in cui vengono mostrati alcuni comandi dell'editor *vi*.

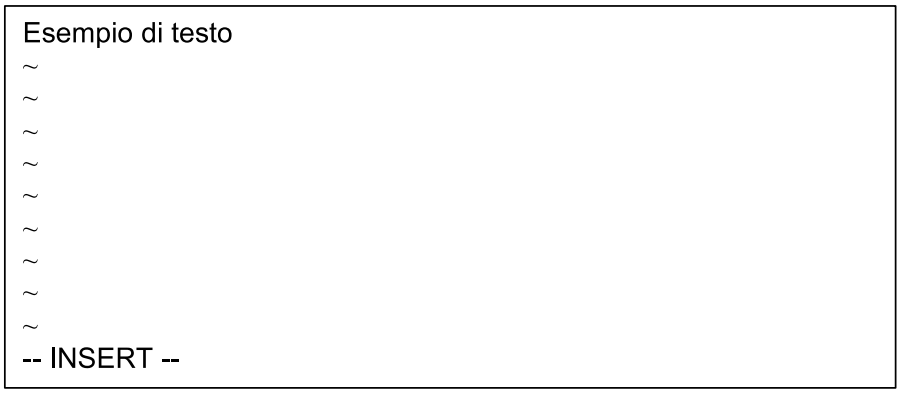


Figura 4.4: Schermata di output del comando `vi` dopo avere inserito del testo

sintassi	**azione**
:q!	uscire senza salvare
:w	salvare il contenuto
:wq	salvare il contenuto prima di uscire
:help	consultare la guida del programma vi
:+*n*	spostarsi alla riga *n* (numero)
:s/*old*/*new*/g	sostituzione della parola *old* con *new* in tutto il contenuto
:r *file*	inserire il file nel testo aperto
:r! *comando*	eseguire un comando di sistema

Infine, al termine di questa discussione su file e directory, ecco due comandi che permettono di leggere il contenuto di un file senza possibilità di modificarlo: *more* e *cat*.

Con *more* è possibile visualizzare il file una pagina alla volta, mentre *cat* lo stampa sullo schermo senza interruzioni.

4.7 Come ottenere aiuto sui comandi

E' possibile consultare la documentazione per tutti i comandi di sistema, tale facility è conosciuta come *man pages*.
Il comando usato per accedere alla documentazione è *man*.
Se si è dunque interessati a cercare informazioni sulle opzioni supportate dal comando *cp* sarà sufficiente digitare

```
$ man cp
```

Le *man pages* sono divise in un certo numero di sezioni che riguardano i comandi, le chiamate di sistema ed altro. Nella letteratura ogni riferimento

comando	**descrizione**	**esempio**
cp	copia un file	cp *questofile.txt /home/questadirectory*
mv	sposta file	mv *thisfile.txt /home/questadirectory*
ls	elenca file	ls
clear	cancella una schermata	clear
exit	chiude una shell	exit
date	visualizza o impostare la data	date
rm	cancella un file	rm *questofile.txt*
echo	copia l'output sullo schermo	echo *questo messaggio*
diff	paragona il contenuto dei file	diff *file1 file2*
grep	trova una stringa di testo in un file	grep *questa parola o frase questofile.txt*
mke2fs o mformat	formatta un dischetto	/sbin/mke2fs /dev/fd0 (/dev/fd0 è l'equivalente Linux di A: in DOS)
man	visualizza l'help di un comando	man *comando*
mkdir	crea una directory	mkdir *directory*
less	visualizza un file	less *questofile.txt*
mv	rinomina un file	mv *questofile.txt quelfile.txt*
pwd	visualizza la propria posizione nel filesystem	pwd
cd *path*	cambia directory con un percorso specifico (*absolute path*)	cd */directory/directory*
cd ..	cambia directory con un *percorso relativo*	cd ..
date	visualizza l'ora	date
free	mostra la quantità di RAM utilizzata	free
sort	riordina le linee di un file di testo	sort *questofile.txt*

Tabella 4.1: Tabella riassuntiva dei principali comandi LINUX

del tipo *entry(n)* indica un rimando alla pagina di manuale *entry* nella sezione numero *n*.
Ecco la lista delle sezioni:

sezione	descrizione	esempio
1	comandi	passwd(1)
2	chiamate di sistema	open(2)
3	funzioni di libreria	printf(3)
4	periferiche	console(4)
5	fomato dei files	inittab(5)
6	giochi	chess(6)
7	miscellanea	ascii(7)
8	comandi di amministrazione	shutdown(8)

Un ulteriore aiuto viene fornito dal comando *apropos*

```
$ apropos stringa
```

Che cerca le occorrenze della stringa *stringa* all'interno della documentazione tra le descrizioni dei comandi

Il comando `apropos` è perciò considerato un buon punto di partenza se non si ha idea di quale comando usare.

Ecco un esempio:

```
$ apropos logged
last (1) - show listing of last logged in users
lastb [last] (1) - show listing of last logged in users
users (1) - print the user names of users currently logged in to
the current host
w (1) - Show who is logged on and what they are doing
who (1) - show who is logged on
```

4.8 L'input e l'output dei processi

Quando inizia l'esecuzione di un processo, il sistema operativo apre tre file dedicati al processo:

- *stdout* (standard output) in cui il programma va a scrivere i messaggi d'uscita. Per default, i processi scrivono il file *stdout* sul terminale.
- *stdin* (standard input) da cui il programma si aspetta di leggere i dati (messaggi) d'ingresso (input). Per default, i processi leggono il file *stdin*

dalla tastiera del computer.
Esempio:

`$ sort`	Il comando sort crea un processo che legge il file stdin
`pippo`	Lo stdin viene creato dall'input della tastiera
`topolino`	
`pluto`	
`CTRL - D`	Termina l'input da tastiera
`pippo`	Il processo scrive il file stdout sullo schermo
`pluto`	
`topolino`	
`$`	Il controllo ritorna alla shell

- *stderr* (standard error) in cui il programma scrive i messaggi di errore. Per default, i processi scrivono il file *stderr* sul terminale.

`$ moer pippo`	è stato immesso un comando sbagliato
`moer: not found`	il processo scrive il file stderr sullo schermo
`$`	Il controllo ritorna alla shell

E' possibile riindirizzare (*redirect*) i file standard tramite i quali un processo comunica. Questo permette di memorizzare, ad esempio, il testo generato da un programma in un file, oppure permette di impartire delle istruzioni ad un programma tramite un file e perciò in modalità non interattiva

La forma più semplice di ridirezione dell'*output* ha questa sintassi:

```
$ programma > outputfile
```

Il segno > permette di redirigere l'output in un file di cui viene soprascritto ogni precedente contenuto.

Per aggiungere dati al file di output si usa invece il doppio maggiore >>

Per ridirezionare l'*input* si usa il segno di minore < secondo la seguente sintassi:

```
$ programma < inputfile
```

Ed ecco un esempio:

`$ who`	mostra gli utenti collegati
`pippo` `poo`	output su terminale
`$ who > collegati`	ridirige l'output del comando nel file collegati
`$ more collegati`	visualizza il contenuto del file
`$ wc -l < collegati`	scrive il numero degli utenti
`$ date >>collegati`	aggiunge la data al file
`$ more collegati`	

`pippo`
`poo`
`2 collegati`
`Lun Lug 18 14:00:00 2005`
	output del file su terminale

4.9 Pipe o come collegare i comandi

La shell permette di connettere due o più processi cosicché l'output di un processo sia usato come input di un altro processo. Questo tipo di connessione che unisce i processi è chiamata *pipe*. Per mandare l'output di un processo ad un altro processo, si devono separare i comandi con la barra verticale |

La sintassi generale sarà dunque

```
programma1 | programma2 | ... | programmaN
```

Riprendendo l'esempio precedente, per ottenere il numero di utenti collegati al sistema si può usare la seguente *pipe*

```
$ who | wc -- l
```

Un altro esempio di utilizzo di pipe è

```
$ ps -ef | more
$ ps -ef | grep pippo
```

4.10 Esecuzione in background dei processi

Un processo in esecuzione viene chiamato *job*, si noti che i termini processo e job sono intercambiabili. Non è detto che un *job* in esecuzione debba interagire con l'utente, inolte abbiamo visto che l'input e l'output possono essere riindirizzati in file, per questo motivo alcuni *job* possono essere eseguiti in *background*.

Alcuni job richiedono molto tempo per finire e non producono risultati intermedi utili per la prosecuzione del lavoro: è il caso della compilazione di programmi, oppure la decompressione di grandi file, oppure le operazioni di manutenzione automatiche. Non c'è ragione di fermare il proprio lavoro per attendere il completamento di questi job: si possono benissimo eseguire in background mentre si fanno altre cose sul terminale.

Per eseguire in background un job bisogna aggiungere il carattere `&` alla fine del comando.

Esempio:

```
$ pippo > /home/gianni/dati/risultati.txt &
[1] 164
$
```

Come si vede la shell ritorna il prompt di comandi. Il valore `[1]` rappresenta il *job number* che la shell assegna al processo generato, mentre `164` rappresenta il *PID* che è stato assegnato al processo

Il comando *jobs* permette di conoscere lo stato dei processi in background:

```
$ jobs
[1]+ Running comando > /home/gianni/dati/risultati.txt &
$
```

Per terminare l'esecuzione del job si usa il comando

```
$ kill %1
```

dove `%1` indica il job number, mentre

```
$ kill 164
```

viene usato per uccidere un processo dal PID generico generato dall'utente.

La sequenza di caratteri `CTRL-C` invece permette di uccidere un processo in esecuzione che sta impegnando il terminale in foreground.

È anche possibile sospendere l'esecuzione di un processo in foreground in modo da farlo ripartire in background, per questo si usa la sequenza *CTRL-Z*

```
$ comando > /home/gianni/dati/risultati.txt
CTRL-Z
Suspended
$
```

Per far ripartire il programma in foreground si usa il comando *fg*

```
$ fg
$ comando > /home/gianni/dati/risultati.txt
```

per farlo eseguire in background si usa invece il comando *bg*

```
$ bg
[1] comando > /home/gianni/dati/risultati.txt &
$
```

4.11 Le code per l'esecuzione batch

A volte è necessario eseguire un programma ad un orario prefissato, oppure semplicemente si vuole avviare il programma ed andarsene a casa senza preoccuparsi di lasciare incustodito il proprio account.

Ad esempio se si vuole eseguire il programma `BuonCompleanno` il 20 aprile alle ore 8:00, si puó usare il seguente comando

```
$ at 0800 apr20 BuonCompleanno
```

cosí il programma sottomesso in coda verrá eseguito in background.
Anche il comando *batch* permette di usare le code di lavoro, ma senza specificare l'orario di partenza. I programmi sottomessi con *batch* verranno eseguiti uno dopo l'altro cosí si evita di sovracaricare il sistema con l'esecuzione in concorrenza di job. Per usarlo, si immette il comando *batch*, poi si digitano i comandi come se fosse una linea di comando ed alla fine della sottomissione si preme la sequenza di tasti `CTRL` e `d` (che verrá visualizzata come `^d`):

```
$ batch
BuonCompleanno 2>& > outfile
^d
```

Quando il programma sarà terminato, il sistema manderà un mail di avviso.

Il comando *atq* mostra tutti i programmi in coda *batch* e tutti i programmi sottomessi in attesa di essere processati.

Il comando `time` misura il tempo impiegato ad eseguire un programma viene usato molto spesso per misurare le prestazioni di un programma.
Per esempio:

`$ time BuonCompleanno`	fornisce il tempo impiegato ad eseguire BuonCompleanno

4.12 Ed ora parliamo della shell

La shell è l'interfaccia fra utente ed sistema operativo Unix/Linux, come si é giá detto nel paragrafo 1.3, perciò è naturale che ne siano state sviluppate diverse ognuna avente delle caratteristiche peculiari. Proprio per sfruttare queste caratteristiche l'ambiente di ogni shell, il cosiddetto *environment*, può essere modificato in modo tale da venire incontro alle richieste dell'utente ed aumentare l'efficienza con cui avviene l'interazione col sistema operativo.

Qui esamineremo le due shell principali sotto Linux: la **bash** e la **tcsh**. In realtà queste sono evoluzioni delle shell più utilizzate sotto UNIX: la Bourne shell (**sh**), che é diretta discendente delle prime shell dei sistemi UNIX, e la **csh** la cui sintassi è simile a quella del linguaggio C.

Perciò possiamo dire che, per quanto riguarda la sintassi, le shell **sh** e **bash** si somigliano, così come si somigliano la **csh** e la **tcsh**.

Inoltre il software per le shell bash e tcsh é di tipo public domain, cioé ne é permessa la libera distribuzione. In particolare la bash viene fornita insieme al sistema Linux, mentre csh e sh sono supportate dai fornitori dei sistemi operativi Unix e quindi sono soggete a copyright.

Esistono altre shell, ma lasciamo al lettore il compito di studiarne le caratteristiche.

Riuscire a comparare le diverse shell è difficile, basta sapere che la bourne/bash è quella con minori funzionalità, peró è quella più largamente utilizzata per gli script del sistema operativo, mentre la csh/tcsh é la piú semplice da programmare.

Le caratteristiche principali di una shell sono

- Il richiamo dei comandi (history): è possibile rieditare i comandi dati in precedenza usando i tasti cursore (non supportato nella bourne).
- Completamento automatico dei comandi e dei nomi di file: si possono digitare i primi caratteri di un comando oppure del nome di un file o di una directory e poi premere il tasto di tabulazione. La shell cercherà di completare il comando oppure il nome. Se ci sono più possibilità di completamento, la shell permetterà di scegliere tra i vari nomi (esiste solo nelle versioni avanzate di bash e tcsh).
- Alias dei comandi: è possibile sostituire ad una stringa anche molto lunga, un nome più semplice
- permettono di modificare facilmente il prompt

  È inoltre possibile cambiare la shell di lavoro in modo permanente la propria shell col comando

```
$ chsh username full_shell_name
```

dove *full_shell_name* è il nome della shell preceduto dal cammino di ricerca.

Le caratteristiche della shell sono definite dalle variabili d'ambiente (*environment variables*) che consistono di un nome a cui è associato un valore.

Per esempio, prendiamo in considerazione la directory in cui si inizia ogni sessione, cioè la *home directory*, la variabile d'ambiente associata si chiama `HOME` ed il suo valore è assegnato durante il processo di login.

Ecco le altre variabili impostate durante il processo di login:

Variabile	**Descrizione**	**Valore di default**
HOME	indica la *home directory*	assegnata durante il login
LOGNAME	contiene lo username	*username*
MAIL	determina dove il sistema registra l'e-mail dell'utente	*/usr/mail/username*
PATH	imposta le directory attraverso cui il sistema cerca i comandi da eseguire	/bin: /usr/bin: /usr/local/bin
SHELL	determina quale shell esegue l'utente	/bin/bash
TERM	tipo di terminale usato per il collegamento	
TZ	TimeZone: indica il fuso orario della locazione	

Alcuni comandi utili sono:

```
$ echo $<nome variabile>
```

che mostra il contenuto della variabile. Provate, ad esempio, il comando `echo $HOME`. Inoltre

```
$ setenv <nome variabile> <valore>  (vale per csh e tcsh)
```

imposta il valore della variabile.

csh e tcsh

Se vogliamo aggiungere una nuova directory al cammino di ricerca dei comandi (*path name*) dovremo eseguire il comando:

```
$ setenv PATH ${PATH}:/home/mario/bin
```

Con ${PATH} si fa riferimento al vecchio valore di PATH a cui viene aggiunta la nuova directory /home/mario/bin.

```
$ unsetenv <variable>
```

cancella il contenuto della variabile.

sh e bash

Per impostare il valore di una variabile d'ambiente si usano i comandi:

```
$ nome_variabile=valore; export nome_variabile
```

Nota che non ci debbono essere spazi vicino l'uguale (=). Esempio:

```
$ PATH=${PATH}:/home/mario/bin; export PATH
```

Per eliminare l'impostazione della variabile invece basta assegnarle un valore nullo:

```
$ nome_variabile=; export nome_variabile
```

Nota sull'importanza della variabile PATH

Quando si immette un comando da terminale, il sistema operativo dev'essere in grado di trovare la directory dove è contenuto. La variabile d'ambiente PATH contiene la lista di directories che viene usata per cercare il comando.

E' possibile modificare localmente il valore di PATH in modo che possa contenere anche directory non-standard, i comandi da usare sono già stati mostrati.

Normalmente il valore predefinito di PATH è

```
$ echo $PATH
/bin:/usr/bin:/usr/local/bin
```

Se il comando non si trova in una di queste directory viene visualizzato il messaggio d'errore

```
comando:  Command not found.
```

Si noti che la ricerca del comando viene effettuata seguendo l'ordine con cui sono elencate le directory nella variabile PATH, perciò la shell esegue la prima istanza del comando che trova lungo il cammino di ricerca e poi procede a seguire.

Se la shell non può trovare il comando, sarà possibile modificare la variabile PATH in modo da includere anche la directory in cui il file di comando è registrato.

4.13 Login script

Durante le fasi di accesso al sistema, la shell esegue i comandi di configurazione contenuti in appositi file chiamati **login script**. Questi file di comandi permettono di modificare i parametri della shell in modo da soddisfare le necessità dell'utente.

Ci sono due tipi di login script:

- script di sistema, questi vengono eseguiti da tutti gli utenti che usano una particolare shell
- script locali che risiedono nella home directory dell'utente e contengono le impostazioni personali.

Gli script di sistema possono esser modificati dall'amministratore e contengono impostazioni di default comuni a tutti gli utenti.

L'utente invece può modificare gli script locali in modo da soddisfare le proprie richieste.

In tabella sono elencati gli script caratteristici delle shell fondamentali:

shell	**programma**	**login script di sistema**	**login script locali**
Bourne	/bin/sh	/etc/profile	$HOME/.profile
bash	/bin/bash	/etc/profile	$HOME/.bash_profile e $HOME/.bashrc
csh e tcsh	/bin/csh e /bin/tcsh	/etc/login	$HOME/.login e $HOME/.cshrc

Nota: i file .bash_profile e .login sono eseguiti ad ogni login e contengono informazioni globali, mentre i file .bashrc e .cshrc vengono eseguiti ogni volta che viene creata una shell. Ogni volta che si esegue un comando viene creata una nuova shell che importa le impostazioni del processo padre e poi esegue gli script .cshrc e .bashrc.

4.14 Come eseguire uno shell script

Ci sono due modi per eseguire uno script:

- dandone semplicemente il nome, con eventuali argomenti. In questo caso viene attivata una nuova shell che esegue i comandi dello script e poi termina, le modifiche fatte alle variabili d'ambiente verranno di conseguenza perse. Lo script deve essere marcato eseguibile attraverso il comando *chmod*. Esempio:

```
$ sh ./comando.sh (in Bourne shell)
$ csh ./comando.csh (in C-Shell)
```

- attraverso il comando *source* in csh e tcsh oppure usando il punto (.) seguito dal nome dello script, se si usa la Bourne shell. In questo caso i comandi sono eseguiti all'interno della shell corrente. Esempio:

```
$ source ./comando.csh (in C-Shell)
$ . ./comando.sh (in Bourne-Shell)
```

4.15 Abbreviazione di comandi: alias

Si possono definire nomi diversi per i comandi, ad esempio:

```
alias del='rm -i'
alias dir='ls -l'
```

Per conoscere la lista degli alias si usa il comando `alias` senza argomenti.
`alias nome` fornisce l'alias definito per quel nome.
Il comando *set* stampa le variabili che sono state definite.
Il valore di una variabile si referenzia facendola precedere dal carattere *$*.

4.16 Prompting

Il prompt dei comandi `PS1` puo' essere modificato per fornire informazioni utili all'utente.
Per fare questo si usano stringhe di caratteri come quelle mostrate di seguito:

```
    significato:
 \t tempo corrente
 \d data corrente
 \w current working directory
 \u username
 \h hostname
 \# command number
 \! history number
```

Ad esempio, se si esegue il seguente comando:

```
PS1='\u@\h:\w > '
```

si cambierá il prompt in

```
mario@localhost:/home/mario >
```

4.17 Uso di ftp, sftp, telnet ed ssh

I programmi che servono ad utilizzare i collegamenti di rete sono gli stessi usati sotto Windows, perciò verranno solamente accennati:

- **ftp** ed **sftp** servono per trasferire file da un computer ad un altro, infatti ftp significa File Transfer Protocol. La sintassi è:

  ```
  $ ftp nome_host
  ```

  nome _host è il nome che in internet viene associato al computer a cui ci si vuol collegare. Una volta collegati, al solito bisogna fornire una *username* ed una *password* di accesso e poi il programma presenta una sua shell con un prompt di comandi.

  Per poter trasferire un file si usano i comandi:

  `put nome_file`, per spedire un file che abbiamo nel nostro computer locale al computer remoto a cui ci siamo collegati tramite ftp

  `get nome_file`, per ricevere e registrare sul nostro computer locale un file presente nel computer remoto

  è possibile trasferire più di un file facendo uso dei *wildcard* con i comandi *mput* ed *mget*, si può cambiare directory con *cd* e visualizzarne il contenuto con *ls*. Per conoscere tutti gli altri comandi ed averne una descrizione è possibile usare il comando *help* nella shell di ftp, oppure *man ftp* nella shell di Linux.

  Il comando **sftp** ha le stesse funzionalità di **ftp**, ma utilizza un *nuovo* protocollo con connessioni crittografate (infatti la *s* sta per *secure*). I due programmi sono perfettamente compatibili: **ftp** viene usato per il collegamento a file server pubblici dove non avrebbe senso proteggere le comunicazioni, mentre **sftp** si usa nei collegamenti in cui vengono scambiate password ed altri tipi di informazioni *sensibili*.

  Nota: per specificare uno username diverso da quello che si ha localmente per collegarsi al computer con sftp si usa la sintassi:

  ```
  $ sftp altro_username@nome_host
  ```

- **telnet** ed **ssh** sono comandi che permettono di collegarsi ad un computer remoto e creare una nuova *shell*. Il comando è

  ```
  $ telnet nome_host
  ```

  oppure

  ```
  $ ssh nome_host -l username
  ```

4.18 Lo sviluppo di applicazioni

Linux mette a disposizione i due compilatori **gcc** ed **f77** sviluppati nell'ambito del progetto GNU per la diffusione libera del software e disponibili su diverse piattaforme anche non Unix.

Per questo motivo il codice sviluppato con questi compilatori è estremamente portabile.

Il *gcc* è anche compilatore per C++ e Objective C, mentre esiste una versione GNU del compilatore per il Fortran 90. Per compilare un programma si usano i comandi mostrati in tabella:

programma C	**programma Fortran**	
$ gcc hello.c	$ f77 hello.f	genera l'eseguibile del programma su **a.out**
$ gcc -o hello hello.c	$ f77 -o hello hello.f	genera l'eseguibile del programma su **hello**

La compilazione si compone di due parti:

1. prima viene generato un file oggetto (hello.o)
2. poi il linker (ld) genera l'eseguibile.

Queste due fasi si possono separare:

```
$ f77 -c hello.f          genera hello.o
$ f77 -o hello hello.o    genera l'eseguibile hello a partire da hello.o.
```

4.19 Letture consigliate

Per scrivere questa parte del capitolo ho fatto riferimento a vari libri o appunti disponibili su internet. Il lettore desideroso di approfondire potrá sicuramente consultarli proficuamente.

1. *A beginner's guide to HP-UX*, manuale del sistema Unix di HP
2. B. W. Kerninghan, R. Pike, *UNIX*, Zanichelli
3. M. Welsh, *LINUX*, Infomagic
4. M. Missiroli, *Linux da Zero*,libro scaricabile dal sito: `http://www.pclinux.it/linuxdazero`
5. D. Medri, *Linux Facile*, presente sul sito: `http://linuxfacile.medri.org`