Università degli studi di Padova

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Matematica

Tesi di Laurea

OTTIMIZZAZIONE A POSTERIORI DELLA SOLUZIONE NUMERICA DI PROBLEMI AI LIMITI

Relatore : ch.mo prof. Martino BARDI

Correlatore: ch.mo prof. Alberto BRESSAN

Studente: Mauro BRUNATO

Padova, 21 novembre 1994

"Tu lo sai bene: non ti riesce qualcosa, sei stanco, e non ce la fai piú. E d'un tratto incontri nella folla lo sguardo di qualcuno, ed è come se ti fossi accostato a un divino nascosto. E tutto diventa improvvisamente piú semplice."

Andrej Tarkovskij, *Andrej Rublëv.*

A mio nonno.

Ringraziamenti

Ringrazio il professor Alberto Bressan e il professor Martino Bardi per il paziente e indispensabile aiuto nella preparazione del presente lavoro e per la libertà concessami.

Un particolare ringraziamento alla professoressa Maria Morandi Cecchi, alla cui intraprendenza sono debitore di un proficuo anno di studi all'estero, per i suoi preziosi suggerimenti.

Sono grato alla mia famiglia, che in me ha sempre manifestato piena fiducia.

Ai miei amici, la cui assidua e operosa compagnia ha riempito di letizia ogni mia fatica: la nostra amicizia abbia la grazia di crescere negli anni a venire.

Indice

Tesi di laurea

OTTIMIZZAZIONE A POSTERIORI DELLA SOLUZIONE NUMERICA DI PROBLEMI AI LIMITI

Introduzione

A partire dal dopoguerra, ma soprattutto negli ultimi vent'anni, si è assistito a un rapido e consistente progresso delle discipline e dei metodi afferenti all'Analisi Numerica. Motore principale di tale sviluppo è stato, accanto al proliferare dei problemi posti dalle scienze fisiche (dalla balistica all'Astrofisica alla Chimica Cinetica), l'accrescimento letteralmente esponenziale della potenza di calcolo degli elaboratori elettronici, tanto da rendere possibile il trattamento di matrici con milioni di elementi, caratteristico dei metodi agli elementi finiti, impensabile fino a epoche molto recenti.

L'Analisi Numerica si dimostra oggi essenziale nella risoluzione delle equazioni differenziali che tanta parte hanno nelle teorie matematiche, fisiche e scientifiche in generale. Oggi disponiamo di una considerevole biblioteca di metodi di risoluzione approssimata, ciascuno applicabile a casi ben determinati per i quali l'errore e il costo della risoluzione possono essere stimati e risultano vantaggiosi. In generale, l'Analisi Numerica investe oggi tutti quei campi dove la manipolazione algebrica e il calcolo simbolico sono inapplicabili o troppo gravosi.

Un campo in cui questa disciplina s'è avventurata solo di recente è quello della dimostrazione rigorosa dell'esistenza di soluzioni di determinati problemi differenziali[1]. Ap-

[1] si veda [Nak] come esempio in tal senso.

parentemente, il calcolo approssimato e la dimostrazione rigorosa possono coesistere quanto la luce con il buio; in realtà, a patto di *maggiorare* opportunamente tutti quegli errori che l'Analisi Numerica preferisce per sua natura *stimare*, risulta abbastanza semplice applicare teoremi di esistenza e unicità locale come il *Teorema di Kantorovič*[2].

La dimostrazione mediante l'ausilio dell'elaboratore elettronico è uno dei temi piú controversi della matematica moderna, si pensi al clamore suscitato dall'annuncio della dimostrazione del *teorema dei quattro colori*: era stato isolato un insieme finito ed esaustivo, per quanto enorme, di casi la cui risoluzione era stata affidata a un programma per calcolatore. L'impossibilità umana a seguire tutti i passaggi, unita alla giustificata diffidenza verso la correttezza dei programmi per calcolatore ("*There's always one more bug*", dicono gli informatici...) spinse molti matematici a non ritenerla nemmeno una dimostrazione.

Per contro, l'uso del calcolatore da parte dell'Analisi Numerica è limitato al trattamento di vettori tramite costrutti iterativi e librerie matematiche di comprovata sicurezza, ed è quanto di piú lontano si possa concepire dalle arzigogolate strutture dati impiegate dai teorici della *dimostrazione automatica* (una branca dell'Intelligenza Artificiale) e dagli autori del programma di dimostrazione dei *quattro colori*. Un caso attualmente sotto studio è, per esempio, il problema di dimostrare l'esistenza di una soluzione strettamente positiva al problema ai limiti

$$\begin{cases} (y^p y')' - mxy' - \dfrac{1}{q-1}y + y^q = 0, & t \in [0, +\infty[\\ y'(0) = 0, \qquad \lim\limits_{x\to\infty} y(x) = 0, \end{cases} \tag{1}$$

con $p > 0$, $q > p - 1$ e $m = \dfrac{q-(p+1)}{2(q-1)}$, che nasce dallo studio delle leggi di conservazione.

La Programmazione Dinamica è anch'essa un potente strumento per la risoluzione di una vastissima gamma di problemi, nel discreto come nel continuo. Nata dal piú classico calcolo delle variazioni, è stata applicata, principalmente da Bellman (vedi per esempio [Bel]), dapprima a processi di controllo in tempo discreto (di tipo economico), poi è stata generalizzata a problemi continui.

(2) cfr. [Ked] e il primo capitolo della presente tesi.

Un impulso decisivo alla teoria fu dato dall'introduzione nei primi anni ottanta di una nuova nozione di *soluzione debole* per problemi differenziali con condizioni al bordo, che andava ad aggiungersi a quella definita da Sobolev negli anni trenta. La soluzione veniva ottenuta aggiungendo un termine di *viscosità* arbitrariamente piccolo alle equazioni, o in modo del tutto equivalente estendendo la definizione di differenziale a funzioni non di classe $\mathcal{C}^1$.

La teoria è in continua evoluzione: per avere un'idea della quantità di pubblicazioni si consulti l'ampia bibliografia in [Bar].

Questo lavoro consiste di due parti. Nella prima (capitolo 1) si descrive e si realizza una possibile tecnica per la dimostrazione numerica dell'esistenza di una soluzione a un problema differenziale ordinario con dati al contorno. A differenza del *problema di Cauchy* nel quale sono fissate le condizioni in un punto e per il quale Cauchy stesso diede teoremi di esistenza e unicità praticamente esaustivi, i problemi ai limiti richiedono, per dimostrazioni rigorose di esistenza e unicità, il ricorso a metodi non sempre comodi come la formulazione debole in opportuni spazi di Sobolev. Per inciso, finora il problema (1) ha resistito ad ogni assalto condotto con metodi analitici.

Realizzato l'algoritmo di dimostrazione, si nota come le stime d'errore siano in genere molto pessimiste, a meno di usare metodi complessi per i quali le maggiorazioni necessarie sono troppo difficoltose, oppure metodi a passo variabile.

Nella seconda parte (capitoli 2 e 3) si propone una tecnica applicata al metodo di Eulero che, data l'approssimazione desiderata, permette di determinare *a posteriori*, ovvero a soluzione approssimativamente nota, una suddivisione a passo variabile che minimizza il costo della risoluzione di un problema differenziale ordinario ai dati iniziali. Il metodo viene ricavato come soluzione di un problema di minimo lagrangiano a partire da una stima dell'errore finale della soluzione.

Il punto focale del capitolo 2 è la dimostrazione di ottimalità asintotica (per un errore tendente a zero) della suddivisione trovata; per questo il problema variazionale viene reimpostato come problema di controllo ottimo e trattato mediante le tecniche della Program-

mazione Dinamica: costruita un'opportuna funzione valore, si utilizza la sua equazione di Hamilton-Jacobi-Bellman. Da questo punto di vista, il lavoro vuole offrire un ulteriore esempio di applicazione dei metodi della Programmazione Dinamica, a riprova della vasta gamma di problemi affrontabili da questa disciplina.

Capitolo I
Sulla risoluzione numerica di problemi ai limiti: una dimostrazione numerica di esistenza

In questo capitolo si presenta una tecnica di dimostrazione numerica dell'esistenza di una soluzione a un problema differenziale ordinario del second'ordine con condizioni agli estremi di un intervallo $[a, b]$. La tecnica si rifà direttamente a un'estensione in spazi di Banach del metodo di Newton per la ricerca dello zero di una funzione in una variabile scalare. Oltre alla dimostrazione, il teorema permetterà di trovare un'approssimazione della soluzione cercata.

In seguito si descrive la realizzazione di un programma in linguaggio C che realizza questa tecnica, e se ne discute l'applicazione a un problema differenziale.

1 — Un problema ai limiti del second'ordine

Dati l'intervallo $[a, b]$ sull'asse dei tempi, la funzione $\mathbf{f} \in \mathcal{C}^1([a, b] \times \mathbb{R}^d \times \mathbb{R}^d, \mathbb{R}^d)$ (dove d è la dimensione del nostro problema) e fissati $\mathbf{x}_a, \mathbf{x}_b \in \mathbb{R}^d$, si vuole dimostrare l'esistenza di una soluzione $\mathbf{x} \in \mathcal{C}^2([a, b], \mathbb{R}^d)$ del seguente problema ordinario con condizioni al bordo:

$$\begin{cases} \ddot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}}) & t \in [a, b] \\ \mathbf{x}(a) = \mathbf{x}_a \\ \mathbf{x}(b) = \mathbf{x}_b. \end{cases} \tag{1}$$

Il problema (1) si riconduce immediatamente alla seguente formulazione:

Dimostrare l'esistenza di $\hat{\mathbf{x}}_a \in \mathbb{R}^d$ *tale che la soluzione* $\mathbf{x}_{\hat{\mathbf{x}}_a} \in \mathcal{C}^2([a, b], \mathbb{R}^d)$ *del problema di Cauchy*

$$\begin{cases} \ddot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}}) & t \in [a, b] \\ \mathbf{x}(a) = \mathbf{x}_a \\ \dot{\mathbf{x}}(a) = \hat{\mathbf{x}}_a \end{cases} \tag{2}$$

verifichi la condizione $\mathbf{x}_{\hat{\mathbf{x}}_a}(b) = \mathbf{x}_b$.

Supponiamo, fissato $\hat{\mathbf{x}}_a$, di avere la soluzione di (2). Sia $\mathbf{F} \in \mathcal{C}^1(\mathbb{R}^d, \mathbb{R}^d)$ il funzionale definito dalla posizione $\mathbf{F}(\hat{\mathbf{x}}_a) = \mathbf{x}_{\hat{\mathbf{x}}_a}(b) - \mathbf{x}_b$. Il problema si riconduce a determinare

l'esistenza di uno zero di $\mathbf{F}$ in un intorno di qualche opportuno punto $\hat{\mathbf{x}}_a$. Utilizzeremo a questo scopo la seguente generalizzazione del metodo di Newton dovuta a L. V. Kantorovič [Kan]:

1. Teorema — *Siano X_1 e X_2 spazi di Banach, $D \subset X_1$ aperto e $\mathbf{F} : D \longrightarrow X_2$ un operatore non lineare. Si supponga $\mathbf{F} \in \mathcal{C}^2(X_1, X_2)$ e siano $\mathbf{F}'$ e $\mathbf{F}''$ i differenziali primo e secondo di Fréchet di $\mathbf{F}$. Sia $\mathbf{x}_0 \in D$ e si assumano le seguenti ipotesi:*

i) *Esiste* $[\mathbf{F}'(\mathbf{x}_0)]^{-1}$;

ii) $\|[\mathbf{F}'(\mathbf{x}_0)]^{-1}\mathbf{F}(\mathbf{x}_0)\| \leq \eta$;

iii) $\|[\mathbf{F}'(\mathbf{x}_0)]^{-1}\| \leq B$;

iv) $\|\mathbf{F}''(\mathbf{x})\| \leq \kappa$ *per* $\|\mathbf{x} - \mathbf{x}_0\| \leq r$;

v) $h = \eta \cdot B \cdot \kappa \leq \frac{1}{2}$;

vi) $r \geq r_0 = \frac{1-\sqrt{1-2h}}{h} \cdot \eta$.

Allora $\mathbf{F}$ ha un unico zero $\mathbf{x}_*$ nella palla $\|\mathbf{x} - \mathbf{x}_0\| \leq r_0$, e lo schema iterativo $\mathbf{x}_{k+1} = \mathbf{x}_k - [\mathbf{F}'(\mathbf{x}_k)]^{-1}\mathbf{F}(\mathbf{x}_k)$ converge a $\mathbf{x}_*$.

Sia dunque $\mathbf{x}_{\hat{\mathbf{x}}_a} \in \mathcal{C}^1([a,b], \mathbb{R}^d)$ la soluzione di (2). Nelle notazioni del teorema poniamo $D = X_1 = X_2 = \mathbb{R}^d$ dotato per comodità della norma infinito $\|\cdot\|_\infty$. Possiamo applicare il teorema al funzionale prima descritto:

$$\begin{aligned} F : \mathbb{R}^d &\longrightarrow \mathbb{R}^d \\ \hat{\mathbf{x}}_a &\mapsto \mathbf{x}_{\hat{\mathbf{x}}_a}(b) - \mathbf{x}_b. \end{aligned}$$

Supponendo $\mathbf{f}$ sufficientemente regolare, i teoremi di dipendenza continua dal dato iniziale assicurano che $\mathbf{F} \in \mathcal{C}^2(\mathbb{R}^d, \mathbb{R}^d)$. Il problema si riconduce a stimare correttamente (e in modo non troppo pessimista) le due grandezze $\|[\mathbf{F}'(\hat{\mathbf{x}}_a)]^{-1}\|$ e $\|\mathbf{F}''(\hat{\mathbf{x}}_a)\|$.

2 — Calcolo di $\mathbf{F}'(\hat{\mathbf{x}}_a)$ e di $\mathbf{F}''(\hat{\mathbf{x}}_a)$

Introduciamo una perturbazione sulla derivata iniziale $\hat{\mathbf{x}}_a$ dipendente da un parametro $\eta \in \mathbb{R}$ variabile in un intorno di 0, tale che la perturbazione sia nulla per $\eta = 0$ e che la dipendenza dal parametro sia almeno $\mathcal{C}^3$. Se definiamo

$$\mathbf{v}_a = \frac{\mathrm{d}}{\mathrm{d}\eta}\hat{\mathbf{x}}_a(\eta)\bigg|_{\eta=0}, \qquad \mathbf{w}_a = \frac{\mathrm{d}^2}{\mathrm{d}\eta^2}\hat{\mathbf{x}}_a(\eta)\bigg|_{\eta=0},$$

potremo scrivere in serie di potenze la derivata iniziale:

$$\hat{\mathbf{x}}_a(\eta) = \hat{\mathbf{x}}_a + \mathbf{v}_a \cdot \eta + \mathbf{w}_a \cdot \frac{\eta^2}{2} + O(\eta^3).$$

Allo stesso modo, chiamando per brevità $\mathbf{x}^\eta(t)$ la soluzione corrispondente alla perturbazione η, poniamo:

$$\mathbf{v}(t) = \left.\frac{\mathrm{d}}{\mathrm{d}\eta}\mathbf{x}^\eta(t)\right|_{\eta=0}, \qquad \mathbf{w}(t) = \left.\frac{\mathrm{d}^2}{\mathrm{d}\eta^2}\mathbf{x}^\eta(t)\right|_{\eta=0}.$$

Sviluppando in serie di potenze di η la funzione

$$\begin{aligned}
&\mathbf{f}(t, \mathbf{x}^\eta(t), \dot{\mathbf{x}}^\eta(t)) = \\
&\quad = \mathbf{f}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) + \left[\mathbf{f}_{\mathbf{x}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{v}(t) + \mathbf{f}_{\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{v}}(t)\right] \cdot \eta + \\
&\qquad + \left[\begin{array}{c}
\mathbf{f}_{\mathbf{xx}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{v}(t) \cdot \mathbf{v}(t) + \mathbf{f}_{\dot{\mathbf{x}}\mathbf{x}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{v}}(t) \cdot \mathbf{v}(t) + \\
+ \mathbf{f}_{\mathbf{x}\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{v}(t) \cdot \dot{\mathbf{v}}(t) + \mathbf{f}_{\dot{\mathbf{x}}\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{v}}(t) \cdot \dot{\mathbf{v}}(t) + \\
+ \mathbf{f}_{\mathbf{x}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{w}(t) + \mathbf{f}_{\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{w}}(t)
\end{array}\right] \cdot \frac{\eta^2}{2} + \\
&\qquad + O(\eta^3)
\end{aligned}$$

e uguagliando i coefficienti delle potenze uguali in (2), otteniamo il seguente sistema di equazioni per le funzioni $\mathbf{v}, \mathbf{w} \in \mathcal{C}([a,b], \mathbb{R}^d)$:

$$\left\{\begin{array}{l}
\ddot{\mathbf{v}} = \mathbf{f}_{\mathbf{x}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{v} + \mathbf{f}_{\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{v}} \\
\ddot{\mathbf{w}} = \mathbf{f}_{\mathbf{xx}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{v} \cdot \mathbf{v} + \mathbf{f}_{\dot{\mathbf{x}}\mathbf{x}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{v}} \cdot \mathbf{v} + \mathbf{f}_{\mathbf{x}\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{v} \cdot \dot{\mathbf{v}} + \\
\qquad + \mathbf{f}_{\dot{\mathbf{x}}\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{v}} \cdot \dot{\mathbf{v}} + \mathbf{f}_{\mathbf{x}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \mathbf{w} + \mathbf{f}_{\dot{\mathbf{x}}}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t)) \cdot \dot{\mathbf{w}} \\
\mathbf{v}(a) = 0 \\
\dot{\mathbf{v}}(a) = \hat{\mathbf{v}}_a \\
\mathbf{w}(a) = 0 \\
\dot{\mathbf{w}}(a) = \hat{\mathbf{w}}_a.
\end{array}\right. \tag{3}$$

I differenziali che ci interessano si ottengono rispettivamente come matrice e tensore delle soluzioni fondamentali del sistema, ovvero risolvendo il sistema per $\hat{\mathbf{v}}_a$ e $\hat{\mathbf{w}}_a$ pari ai versori della base canonica.

3 — Approssimazioni numeriche e errori

Prima di passare alla costruzione di un algoritmo numerico in grado di trovare maggiorazioni delle grandezze in gioco, ovvero delle grandezze η, B e κ del teorema 1, è chiaramente necessario maggiorare con assoluta sicurezza ognuna delle norme considerate dal teorema.

Considereremo, per semplificare il ragionamento, la trasformazione del problema (2) in un problema del primo ordine. Usiamo le seguenti notazioni:

$$\mathbf{X}(t) = \begin{pmatrix} \mathbf{x} \\ \dot{\mathbf{x}} \end{pmatrix}, \quad \mathbf{G}(t, \mathbf{X}) = \mathbf{G}\left(t, \begin{pmatrix} \mathbf{x} \\ \dot{\mathbf{x}} \end{pmatrix}\right) = \begin{pmatrix} \dot{\mathbf{x}} \\ \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}}) \end{pmatrix}, \quad \mathbf{X}_a = \begin{pmatrix} \mathbf{x}_a \\ \hat{\mathbf{x}}_a \end{pmatrix}.$$

Il problema (2) diventa il seguente:

$$\begin{cases} \dot{\mathbf{X}} = \mathbf{G}(t, \mathbf{X}) & t \in [a, b] \\ \mathbf{X}(a) = \mathbf{X}_a. \end{cases} \tag{4}$$

Non abbiamo bisogno di stime molto precise della soluzione: basta che si verifichi l'ipotesi *v)* del teorema. Inoltre i calcoli saranno piú comodi se richiederanno la conoscenza di poche derivate per la maggiorazione degli errori. Come metodo di approssimazione della soluzione utilizzeremo dunque il *metodo di Eulero* a passo fisso.

Chiamiamo N il numero di intervalli in cui viene suddiviso l'intervallo $[a, b]$, sia $h = \frac{b-a}{N}$ l'ampiezza di ciascun intervallo e chiamiamo $t_i = a + ih \quad (i = 0, \ldots, N)$ i punti della suddivisione. Per ogni i fissato chiamiamo *errore di troncamento locale* la differenza

$$\tau_i(h) = (\mathbf{X}(t_{i+1}) - \mathbf{X}(t_i)) - \mathbf{G}(t_i, \mathbf{X}_i)$$

tra il salto effettivo compiuto dalla funzione tra i punti t_i e t_{i+1} e il salto causato dal corrispondente passo del metodo di Eulero. Essendo quest'ultimo semplicemente un troncamento al termine lineare della serie di Taylor della soluzione, la differenza è agilmente maggiorata nel seguente modo:

$$\|\tau_i(h)\|_\infty \leq \frac{h^2}{2} \sup_{\substack{t_i \leq t \leq t_{i+1} \\ \mathbf{x} \in \mathbb{R}^{2d}}} \|\mathbf{G}_t(t, \mathbf{X}) + \mathbf{G}_\mathbf{X}(t, \mathbf{X}) \cdot \mathbf{G}(t, \mathbf{X})\|_\infty .$$

D'ora in poi supporremo che esista una costante $\bar{\mathbf{X}} \in \mathbb{R}^{2d}$ tale che la soluzione di (4) resta sempre dentro la palla di centro 0 e raggio $\bar{\mathbf{X}}$. Quest'ulteriore ipotesi permette di fissare le seguenti costanti:

$$S = \sup_{(t,\mathbf{X}) \in \mathcal{B}} \|\mathbf{G}(t, \mathbf{X})\|_\infty ,$$

$$L_t = \sup_{(t,\mathbf{X}) \in \mathcal{B}} \|\mathbf{G}_t(t, \mathbf{X})\|_\infty , \qquad L_\mathbf{X} = \sup_{(t,\mathbf{X}) \in \mathcal{B}} \|\mathbf{G}_\mathbf{X}(t, \mathbf{X})\|_\infty ,$$

$$M_{t\mathbf{X}} = \sup_{(t,\mathbf{X}) \in \mathcal{B}} \|\mathbf{G}_{\mathbf{X}t}(t, \mathbf{X})\|_\infty , \qquad M_{\mathbf{XX}} = \sup_{(t,\mathbf{X}) \in \mathcal{B}} \|\mathbf{G}_{\mathbf{XX}}(t, \mathbf{X})\|_\infty ,$$

dove

$$\mathcal{B} = [a, b] \times B(0, \bar{\mathbf{X}}).$$

Ricordiamo che la norma matriciale indotta dalla norma infinito è, data la matrice $A = (a_{ij})_{1 \le i,j \le 2d}$,

$$\|A\|_\infty = \max_{i=1,\dots,2d} \sum_{j=1}^{2d} |a_{ij}|,$$

e per il tensore di rango 3 $S = (s_{ijk})_{1 \le i,j,k \le 2d}$ vale

$$\|S\|_\infty = \max_{i=1,\dots,2d} \sum_{j,k=1}^{2d} |s_{ijk}|.$$

L'errore di troncamento locale è dunque maggiorato così:

$$\|\tau_i(h)\|_\infty \le \frac{h^2}{2} \cdot (L_t + L_{\mathbf{X}} S).$$

Si noti che $L_{\mathbf{X}}$ vale anche come costante di Lipschitz uniforme per la seconda variabile della funzione $\mathbf{G}(t, \mathbf{X})$. In virtú di questo e in base a noti teoremi sui metodi alle differenze finite a passo singolo (vedi [Gea], [Bur]) l'errore sul dato finale $\|\mathbf{X}(b) - \mathbf{X}_N\|_\infty$ è maggiorato da

$$E_{\mathbf{X}} = \frac{1}{2}(L_t + L_{\mathbf{X}} S) \cdot \frac{e^{L_{\mathbf{X}}(b-a)} - 1}{L_{\mathbf{X}}} \cdot h.$$

Allo stesso modo, il sistema (3) si riconduce ai due sistemi (la notazione è ovvia):

$$\begin{cases} \dot{\mathbf{V}} = \mathbf{G}_{\mathbf{X}}(t, \mathbf{X}(t)) \cdot \mathbf{V} & t \in [a, b] \\ \mathbf{V}(a) = \mathbf{V}_a, \end{cases} \tag{5}$$

$$\begin{cases} \dot{\mathbf{W}} = \mathbf{G}_{\mathbf{XX}}(t, \mathbf{X}(t)) \cdot \mathbf{V} \cdot \mathbf{V} + \mathbf{G}_{\mathbf{X}}(t, \mathbf{X}(t)) \cdot \mathbf{W} & t \in [a, b] \\ \mathbf{W}(a) = \mathbf{W}_a. \end{cases} \tag{6}$$

Per quanto riguarda l'approssimazione numerica del sistema (5), è necessario tener conto che all'errore di troncamento locale dovuto alla discretizzazione del passo (che chiameremo $\tau_i'(h)$) si aggiunge quello dovuto all'approssimazione con cui conosciamo il valore della soluzione $\mathbf{X}(t)$ di (4), che induce un'approssimazione sul calcolo del termine lineare in h. Supponendo di disporre di un limite superiore $\bar{\mathbf{V}}$ per $\mathbf{V}(t)$ (l'analogo di $\bar{\mathbf{X}}$), l'errore di troncamento dovuto alla discretizzazione è maggiorato nel modo seguente:

$$\|\tau_i'(h)\|_\infty \le \sup_{(t,X) \in \mathcal{B}} \left\| \begin{array}{c} [\mathbf{G}_{\mathbf{X}t}(t, \mathbf{X}) + \mathbf{G}_{\mathbf{XX}}(t, \mathbf{X}) \cdot \mathbf{G}(t, \mathbf{X})] \cdot \bar{\mathbf{V}} + \\ + \mathbf{G}_{\mathbf{X}}(t, \mathbf{X}) \cdot \mathbf{G}_{\mathbf{X}}(t, \mathbf{X}) \cdot \bar{\mathbf{V}} \end{array} \right\|_\infty,$$

cioè

$$\|\tau_i'(h)\|_\infty \leq \frac{h^2}{2}(M_{t\mathbf{X}} + M_{\mathbf{XX}}S + L_{\mathbf{X}}^2) \cdot \bar{\mathbf{V}}.$$

La seconda componente dell'errore va cercata nel termine lineare in h e, maggiorata, vale

$$\|\tau_i''(h)\|_\infty \leq hM_{\mathbf{XX}}E_{\mathbf{X}}\bar{\mathbf{V}}.$$

L'errore finale nella valutazione di $\mathbf{V}$, $\|\mathbf{V}(b) - \mathbf{V}_N\|_\infty$ va calcolato in base alla somma $\tau_i'(h) + \tau_i''(h)$, ed è maggiorato da

$$E_{\mathbf{V}} = \frac{1}{2}\bar{\mathbf{V}}\left[M_{t\mathbf{X}} + M_{\mathbf{XX}}\left(L_t\frac{e^{L_{\mathbf{X}}(b-a)}-1}{L_{\mathbf{X}}} + Se^{L_{\mathbf{X}}(b-a)}\right) + L_{\mathbf{X}}^2\right]e^{L_{\mathbf{X}}(b-a)}.$$

Per quanto riguarda il sistema (6), ci accontenteremo di una maggiorazione grezza del dato finale a partire da questa disuguaglianza (lecita, in quanto per funzioni vettoriali di classe $\mathcal{C}^2$ la norma infinito è continua, derivabile in tutti i punti tranne al piú un numero finito e, dove esiste, la derivata della norma è maggiorata dalla norma della derivata):

$$\begin{cases} \dfrac{\mathrm{d}}{\mathrm{d}t}\|\mathbf{W}(t)\|_\infty \leq \|\mathbf{G}_{\mathbf{XX}}(t,\mathbf{X}(t))\cdot\mathbf{V}(t)\cdot\mathbf{V}(t) + \mathbf{G}_{\mathbf{X}}(t,\mathbf{X}(t))\cdot\mathbf{W}(t)\|_\infty \\ \qquad\qquad \leq M_{\mathbf{XX}}\bar{\mathbf{V}}^2 + L_{\mathbf{X}}\|\mathbf{W}(t)\|_\infty \\ \|\mathbf{W}(a)\|_\infty = 1 \end{cases}$$

che dà luogo alla maggiorazione di $\|\mathbf{W}(b)\|_\infty$:

$$M_{\mathbf{W}} = e^{L_{\mathbf{X}}(b-a)} + M_{\mathbf{XX}}\bar{\mathbf{V}}^2\frac{e^{L_{\mathbf{X}}(b-a)}-1}{L_{\mathbf{X}}}.$$

L'errore sulla valutazione di $\mathbf{F}(\hat{\mathbf{x}}_a)$ è abbondantemente maggiorato da $E_{\mathbf{X}}$ (quest'ultimo è una maggiorazione anche sull'errore della derivata $\dot{\mathbf{x}}$). La matrice del differenziale primo $\mathbf{F}'(\hat{\mathbf{x}}_a)$ è composta da d colonne ciascuna delle quali è calcolata con un errore anch'esso largamente maggiorato da $E_{\mathbf{V}}$; è dunque nota a meno di un errore pari a $dE_{\mathbf{V}}$. Il tensore del differenziale secondo $\mathbf{F}''(\hat{\mathbf{x}}_a)$ è composto da $d \times d$ colonne ciascuna delle quali è maggiorata in norma da $M_{\mathbf{W}}$. Dunque, $\|\mathbf{F}''(\hat{\mathbf{x}}_a)\|_\infty \leq d^2M_{\mathbf{W}}$.

Una volta note la matrice (chiamiamola $\bar{A}$) del differenziale approssimato, la sua inversa $\bar{A}^{-1}$ e una maggiorazione dell'errore rispetto alla matrice esatta $\|A - \bar{A}\|_\infty \leq dE_{\mathbf{V}}$, vogliamo calcolare l'errore $\|A^{-1} - \bar{A}^{-1}\|_\infty$ con cui è nota l'inversa:

$$\bar{A}(\bar{A}^{-1} - A^{-1}) + (\bar{A} - A)A^{-1} = 0$$

$$
\begin{aligned}
(\bar{A}^{-1} - A^{-1}) &= -\bar{A}^{-1}(\bar{A} - A)A^{-1} \\
\left\|\bar{A}^{-1} - A^{-1}\right\|_\infty &\leq \left\|\bar{A}^{-1}\right\|_\infty \left\|\bar{A} - A\right\|_\infty \left\|A^{-1}\right\|_\infty \\
&\leq \left\|\bar{A}^{-1}\right\|_\infty \left\|\bar{A} - A\right\|_\infty \left(\left\|\bar{A}^{-1}\right\|_\infty + \left\|\bar{A}^{-1} - A^{-1}\right\|_\infty\right) \\
\left\|\bar{A}^{-1} - A^{-1}\right\|_\infty \left(1 - \left\|\bar{A}^{-1}\right\|_\infty \left\|\bar{A} - A\right\|_\infty\right) &\leq \left\|\bar{A}^{-1}\right\|_\infty^2 \left\|\bar{A} - A\right\|_\infty .
\end{aligned}
$$

Se $\left\|\bar{A}^{-1}\right\|_\infty \left\|\bar{A} - A\right\|_\infty < 1$, ha senso effettuare la divisione e otteniamo la maggiorazione cercata:

$$
\begin{aligned}
\left\|A^{-1} - \bar{A}^{-1}\right\|_\infty &\leq \frac{\left\|\bar{A}^{-1}\right\|_\infty^2}{1 - \left\|\bar{A}^{-1}\right\|_\infty \left\|\bar{A} - A\right\|_\infty} \left\|\bar{A} - A\right\|_\infty \\
&\leq \frac{\left\|\bar{A}^{-1}\right\|_\infty^2}{1 - \left\|\bar{A}^{-1}\right\|_\infty dE_{\mathbf{V}}} dE_{\mathbf{V}}.
\end{aligned}
\tag{7}
$$

Chiaramente, se $\left\|\bar{A}^{-1}\right\|_\infty \left\|\bar{A} - A\right\|_\infty \geq 1$ non siamo piú sicuri che il differenziale sia invertibile (la stima dell'errore sull'inversa va ad infinito).

Per applicare il teorema 1 dobbiamo ora fissare il valore delle tre costanti η, B e κ, e sperare che il loro prodotto sia minore di $\frac{1}{2}$:

$$
B = \left\|\bar{A}^{-1}\right\|_\infty + \frac{\left\|\bar{A}^{-1}\right\|_\infty^2}{1 - \left\|\bar{A}^{-1}\right\|_\infty dE_{\mathbf{V}}} dE_{\mathbf{V}}, \qquad \eta = B \cdot (\|\mathbf{x}_N - \mathbf{x}_1\|_\infty + E_{\mathbf{X}}), \qquad \kappa = d^2 M_{\mathbf{W}}.
$$

Nel caso in cui $\eta B \kappa > \frac{1}{2}$, si procederà a un'iterazione (ovviamente approssimata) del metodo:

$$
\hat{\mathbf{x}}_a \longleftarrow \hat{\mathbf{x}}_a - \bar{A}^{-1} \cdot (\mathbf{x}_N - \mathbf{x}_b)
$$

e si ripeteranno i calcoli (le maggiorazioni $E_{\mathbf{X}}$, $E_{\mathbf{V}}$ e $M_{\mathbf{W}}$ restano sempre le stesse).

4 — Descrizione del programma

Descriveremo ora il programma C da noi costruito per la dimostrazione dell'esistenza di una soluzione del problema differenziale (1). Il programma è suddiviso in piú blocchi funzionali:

- Il file `kantorov.c` con il suo header `kantorov.h` contiene la funzione principale del programma e consiste nell'applicazione del teorema 1.
- Il file `diffeq.c` (header `diffeq.h`) contiene varie routines per la risoluzione di problemi differenziali ordinari ai dati iniziali, scalari o vettoriali, del primo o del second'ordine.

- Il file `matinv.c` (header `matinv.h`) contiene la procedura di inversione di una matrice quadrata.
- Il file `f-kan.c` (header `f-kan.h`) contiene la definizione della funzione $\mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}})$ e della sua derivata $\mathbf{f}_{\mathbf{x}}(t, \mathbf{x}, \dot{\mathbf{x}})$.
- Il file `p-kan.c` contiene la funzione `main`.
- L'header `errori.h` definisce tutti i possibili errori che le varie funzioni possono comunicare.
- Infine, il file `p-kan.dat` contiene in forma leggibile i dati che definiscono il problema e che saranno descritti in seguito.

4.1. *Descrizione del file* `kantorov.c`

Descriviamo innanzitutto il blocco relativo all'applicazione del teorema di Kantorovič. Vediamo subito l'header (file `kantorov.h`):

```
1. /* kantorov.h  (Mauro Brunato, ottobre 1994)
2.    applicazione del teorema di Kantorovich alla ricerca di una
3.    soluzione di un problema differenziale ai limiti del
4.    secondo ordine.
5. */
6. #ifndef _KANTOROV_H_
7. #define _KANTOROV_H_
8.
9. #include "errori.h"
10.
11. typedef void (*funzione) (double, double[], double[], double[]);
12. typedef void (*derivata) (double, double[], double[],
13.                                 double[], double[], double[]);
14.
15. typedef struct                       /* Dati della funzione          */
16. {int       dimensione;               /* dimensione del problema      */
17.  funzione  f;                        /* funzione x" = f(t,x,x')      */
18.  derivata  fxv;                      /* derivata v" = fx(t,x,x')*v   */
19.  double    X_, V_, a, b, S, Lt, Lx, Mxt, Mxx;
20. } dati_funzione;
21.
22. typedef struct                       /* Dati del problema            */
23. {double *dato_iniziale,             /* valore iniziale xa           */
24.         *dato_finale,               /* valore finale xb             */
25.         *derivata_iniziale;         /* primo tentativo per xa'      */
26.  int    tentativi;                  /* numero massimo di iterazioni */
27.  int    fino_in_fondo;              /* compiere tutte le iterazioni? */
28.  long   numero_nodi;                /* passi del metodo di Eulero   */
29. } dati_problema;
```

```
30.
31. typedef struct                          /* Risultati del problema        */
32. {int       iterazioni;                  /* iterazioni compiute           */
33.  long      consiglio;                   /* numero di nodi consigliato    */
34.  double    *xp0,                        /* ultimo valore di xa[}          */
35.            eta, B, kappa, raggio;       /* costanti definite dal teorema */
36. } risultati;
37.
38. esito Kantorovic (dati_funzione*, dati_problema*, risultati*);
39.
40. #endif
```

In questo file sono dichiarati i tipi di dato e le funzioni *pubbliche* del file `kantorov.c`. Per evitare di dover considerare anche gli errori di arrotondamento introdotti a causa dell'aritmetica finita del calcolatore, sono utilizzati sempre e soltanto reali in doppia precisione (tipo `double` a 64 bit).

$6 \div 7$ Il file si apre con una metaistruzione condizionale comune a tutti gli header volta a impedire che il file venga letto piú di una volta durante la compilazione.

9 Viene poi incluso l'header `errori.h` contenente la definizione del tipo per enumerazione `esito` e il cui listato verrà riportato piú avanti.

11 Subito dopo è definito il tipo `funzione` per la funzione, definita dal programma chiamante, che deve restituire nel vettore puntato dal quarto parametro il valore di $\mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}})$ calcolata sui primi tre parametri.

$12 \div 13$ Analogamente viene definito il tipo `derivata`. La funzione corrispondente deve restituire nel sesto parametro il valore del prodotto

$$\mathbf{f}_{\mathbf{x}}(t, \mathbf{x}, \dot{\mathbf{x}}) \cdot \mathbf{v} + \mathbf{f}_{\dot{\mathbf{x}}}(t, \mathbf{x}, \dot{\mathbf{x}}) \cdot \dot{\mathbf{v}},$$

dove lo scalare t e i vettori $\mathbf{x}$, $\dot{\mathbf{x}}$, $\mathbf{v}$ e $\dot{\mathbf{v}}$ sono i primi cinque parametri.

$15 \div 20$ `dati_funzione`, il primo record definito, serve a descrivere la funzione: innanzitutto la dimensione del problema, poi i puntatori alle due funzioni descritte prima, infine le costanti $\bar{\mathbf{X}}$, $\bar{\mathbf{V}}$, gli estremi dell'intervallo a e b, le costanti L_t, $L_{\mathbf{X}}$, $M_{t\mathbf{X}}$ e $M_{\mathbf{XX}}$.

22 ÷ 29 Il secondo record è `dati_problema`, e contiene i dati che descrivono il problema differenziale e le richieste del programma chiamante: `*dato_iniziale` e `*dato_finale` puntano a due vettori, ciascuno di lunghezza d (il campo `dimensione` del record precedente), contenenti le condizioni al bordo; `*derivata_iniziale`, anch'esso di dimensione d, contiene il valore di $\hat{\mathbf{x}}_a$ da cui iniziare le iterazioni; `tentativi` è un intero che limita il numero di iterazioni per evitare cicli infiniti nel caso in cui non si riesca a trovare una soluzione; `fino_in_fondo` è un flag che, se diverso da zero, dice alla routine di non fermarsi una volta che l'esistenza è dimostrata, ma di proseguire finché il numero di iterazioni ha raggiunto il massimo, in modo da trovare un'approssimazione migliore per $\hat{\mathbf{x}}_a$. Se `fino_in_fondo` vale zero, la routine si ferma non appena i calcoli assicurano l'esistenza di uno zero. L'intero a 32 bit `numero_nodi` riporta il numero di nodi per il calcolo della soluzione approssimata.

31 ÷ 36 L'ultimo record, `risultati`, contiene tutti i dati che debbono essere restituiti dalla routine: `iterazioni` contiene il numero di iterazioni effettivamente compiute; `consiglio` serve nel caso in cui il programma chiamante abbia dato un numero di nodi troppo piccolo per garantire un errore sufficientemente basso, e in tal caso viene riportato nel campo il numero di nodi consigliato per raggiungere lo scopo. Nel vettore `*xp0` la routine riporrà l'ultimo valore calcolato per $\hat{\mathbf{x}}_a$; negli ultimi quattro campi verranno riposti gli ultimi valori di η, B, κ e r_0.

38 Infine viene fornito il prototipo della funzione `kantorovich`, l'unica funzione pubblica definita nel file `kantorov.c`. La funzione accetta come parametri i puntatori ai tre vettori prima dichiarati, che dovranno essere già stati allocati dal programma chiamante, con tutti i puntatori legati a vettori già definiti. La funzione restituisce un valore di tipo `esito`, un tipo per enumerazione definito nell'header `errori.c`, i cui valori corrispondono ai diversi risultati dell'algoritmo.

Il file `kantorov.c` definisce la funzione dichiarata nell'header precedente. La funzione

`kantorovich` funziona nel seguente modo:

Passo 1: (linee $50 \div 71$) Dati i parametri d'ingresso descritti in `kantorov.h`, calcolare le variabili `errore_F` (maggiorazione dell'errore sul calcolo di $\|\mathbf{F}(\hat{\mathbf{x}}_a)\|_\infty$), `errore_Fp` (errore sul calcolo di $\|\mathbf{F}'(\hat{\mathbf{x}}_a)\|_\infty$) e `norma_Fs` (maggiorazione di $\|\mathbf{F}''(\hat{\mathbf{x}}_a)\|_\infty$ per ogni $\hat{\mathbf{x}}_a$ ammissibile). Allocare la memoria necessaria per immagazzinare i vettori usati dall'algoritmo e inizializzarli.

Passo 2: (linee $74 \div 86$) Inizia l'iterazione. Calcolare il valore approssimato di $\mathbf{F}(\hat{\mathbf{x}}_a)$ nel vettore `xf` e quello di $\mathbf{F}'(\hat{\mathbf{x}}_a)$ nella matrice `M1`. Invertire la matrice `M1` ponendo l'inversa in `M2`. Se la procedura di inversione restituisce un errore (cioè se la matrice non è invertibile), interrompere trasmettendo l'errore alla procedura chiamante.

Passo 3: (linee $88 \div 102$) Calcolare le norme infinito di `xf`, di `M1` e di `M2`. Calcolare inoltre il prodotto `M2` $\cdot$ `xf` (approssimazione di $[\mathbf{F}'(\hat{\mathbf{x}}_a)]^{-1} \cdot \mathbf{F}(\hat{\mathbf{x}}_a)$) e porlo nella prima riga di `M1`, che non è piú necessaria. Le norme calcolate non sono ancora maggiorazioni.

Passo 4: (linee $103 \div 126$) Verificare se è possibile maggiorare la norma dell'errore su `M2` per mezzo della formula (7) (se non è possibile, non si ha la sicurezza che il vero differenziale sia invertibile). Se sì, una volta calcolato `errore_Fp1F` (l'errore su $[\mathbf{F}'(\hat{\mathbf{x}}_a)]^{-1} \cdot \mathbf{F}(\hat{\mathbf{x}}_a)$) tutti gli errori necessari sono noti, altrimenti andare al passo 5.

Se il prodotto `errore_Fp1F` $\cdot$ `errore_Fp1` $\cdot$ `norma_Fs` è maggiore di $\frac{1}{2}$, gli errori sono troppo grossi per ottenere un risultato: interrompere la procedura e riferire al programma chiamante la necessità di un numero di nodi maggiore.

Una volta sommati gli errori `errore_Fp1F` e `errore_Fp1` alle rispettive norme, abbiamo a disposizione le costanti η, B e κ. Se il loro prodotto è minore di $\frac{1}{2}$, porre nel record dei risultati le costanti e il raggio r_0 da queste ricavato.

se il campo `fino_in_fondo` del record dei dati vale 0, terminare il procedimento.

Passo 4: (linee $127 \div 137$) Se il numero di iterazioni ha raggiunto il massimo, ritornare al programma chiamante, altrimenti aggiornare secondo lo schema iterativo del teorema 1 la derivata iniziale e tornare al Passo 2.

Ecco, tradotto in linguaggio C, l'algoritmo appena dscritto:

```
1. /* kantorov.c  (Mauro Brunato, ottobre 1994)
2.    Applicazione del metodo di Kantorovic alla ricerca di condizioni iniziali
3.    (vedere kantorov.h)
4. */
5. #include <stdlib.h>
6. #include <math.h>
7. #include "kantorov.h"
8. #include "diffeq.h"
9. #include "matinv.h"
10.
11. /* Alloca memoria per dati double */
12. #define DALLOC(n) (double*) malloc ((n) * sizeof(double))
13.
14. /* Rendi il dato compatibile con le procedure */
15. #define ADATTA double(*)[]
16.
17. /* Errore se l'istruzione non riesce */
18. #define PROVA(istruzione) if(ERRORE(err=istruzione)){free(KA_X);return err;}
19.
20. /* Ripetizione indefinita */
21. #define RIPETI while ( 1 )
22.
23. /* Ripetizione per il numero di equazioni */
24. #define PER(i) for ( i = 0; i < KA_D; i++ )
25.
26. /* Massimo tra due grandezze */
27. #define MAX(x,y) (((x)>(y))?(x):(y))
28.
29. /* Minimo tra due grandezze */
30. #define MIN(x,y) (((x)<(y))?(x):(y))
31.
32. /* Variabili globali */
33. int      KA_D;
34. double   KA_A, KA_H, *KA_X, *KA_Xp, KA_XM_, KA_VM_;
35. long     KA_N;
36. funzione KA_F;
37. derivata KA_FXV;
38.
39. /* Prototipi delle funzioni interne */
40. void   KA_F_con_controllo   (double, double[], double[], double[]);
41. void   KA_FXV_con_controllo (double, double[], double[], double[]);
42.
43. esito Kantorovic (dati_funzione *df, dati_problema *dp, risultati *r)
44. {double S, Lt, Lx, Mxt, Mxx, KA_B, X_, V_, esp, molt,
```

```
45.         *M1, *M2, *v, *vp, *xf, *vf,
46.         errore_F, errore_Fp, errore_Fp1, errore_Fp1F,
47.         norma_F, norma_Fp, norma_Fp1, norma_Fp1F, norma_Fs,
48.         somma;
49.  int i, j, err, dentro, riuscito = 0;
50.  KA_D = df -> dimensione;      X_     = df -> X_;
51.  KA_A = df -> a;              KA_B   = df -> b;
52.  S    = MAX (X_, df -> S);     Lt     = df -> Lt;
53.  Lx   = MAX (1, df -> Lx),     Mxt    = df -> Mxt;
54.  Mxx  = df -> Mxx;            esp    = exp (Lx * (KA_B - KA_A));
55.  V_   = MIN (df -> V_, esp);   molt   = (esp - 1) / Lx;
56.  KA_N = dp -> numero_nodi;     KA_H   = (KA_B - KA_A) / KA_N;
57.  KA_F = df -> f;               KA_FXV = df -> fxv;
58.  errore_F  = (Lt + Lx*S) * molt * KA_H / 2;
59.  errore_Fp = KA_D * V_ * (Mxt + Mxx*(Lt*molt+S*esp) + Lx*Lx) * molt * KA_H / 2;
60.  norma_Fs  = KA_D*KA_D * (esp + Mxx * V_ * V_ * molt);
61.  if ( !(KA_X = DALLOC (2*(KA_N+1)*KA_D + 2*KA_D*KA_D + 3*KA_D)) )
62.   return ka_allocazione_fallita;
63.  KA_Xp = KA_X + (KA_N+1)*KA_D; v = KA_Xp + (KA_N+1)*KA_D;
64.  vp = v + KA_D; vf = vp + KA_D;
65.  M1 = vf + KA_D; M2 = M1 + KA_D*KA_D;
66.  xf = KA_X + KA_N*KA_D;
67.  PER(i)
68.  {KA_X[i]  = dp -> dato_iniziale[i];
69.   KA_Xp[i] = dp -> derivata_iniziale[i];
70.   v[i]  = vp[i] = 0;
71.  }
72.  r -> iterazioni = 1;
73.  RIPETI                    /* Itera fino alla conclusione dell'algoritmo */
74.  {KA_XM_ = KA_VM_ = 0;
75.   PROVA                                              /* Calcola F */
76.   ( Eulero_s2 (KA_F_con_controllo, KA_A, KA_B,
77.                (ADATTA)KA_X, (ADATTA)KA_Xp, KA_N, KA_D) )
78.   PER(i)                                             /* Calcola F' */
79.   {vp[i] = 1;
80.    PROVA
81.    ( Eulero_s2n (KA_FXV_con_controllo, KA_A, KA_B,
82.                  v, vp, vf, KA_N, KA_D) )
83.    PER(j) M1[j*KA_D+i] = vf[j];
84.    vp[i] = 0;
85.   }
86.   PROVA ( inversione ((ADATTA)M1, (ADATTA)M2, KA_D) ) /* Calcola F'^{}-1 */
87.   PER(i) xf[i] -= dp -> dato_finale[i];               /* Normalizza F */
88.   norma_F = norma_Fp1 = norma_Fp1F = norma_Fp = 0;    /* Calcola le norme */
89.   PER(i)                                             /* di F, F' e F'^{}-1 */
90.   {somma = 0;
91.    PER(j) somma += fabs(M2[i*KA_D+j]);
92.    if( somma > norma_Fp1 ) norma_Fp1 = somma;
93.    somma = 0;
94.    PER(j) somma += fabs (M1[i*KA_D+j]);
95.    if ( somma > norma_Fp ) norma_Fp = somma;
96.    somma = 0;
97.    PER(j) somma += M2[i*KA_D+j] * xf[j];
```

```
98.    if ( fabs (vf[i] = somma) > norma_Fp1F )
99.     norma_Fp1F = fabs (somma);
100.   if ( fabs (xf[i]) > norma_F )
101.    norma_F = fabs (xf[i]);
102.  }
103.  if ( (molt = norma_Fp1 * errore_Fp) < 1 &&          /* verifica se il dif- */
104.       (dentro = (KA_XM_ <= X_ && KA_VM_ <= V_)) )    /* ferenziale e' vera- */
105.  {norma_F += errore_F;                               /* mente invertibile.  */
106.   errore_Fp1 = molt * norma_Fp1 / (1 - molt);        /* Se si' prosegue il  */
107.   norma_Fp1 += errore_Fp1;                           /* calcolo degli       */
108.   errore_Fp1F = norma_Fp1 * errore_F +               /* errori.             */
109.                 norma_F * errore_Fp1 +
110.                 errore_F * errore_Fp;
111.   norma_Fp1F += errore_Fp1F;
112.   if ( errore_Fp1F * errore_Fp1 * norma_Fs >= 0.5 )
113.   {r -> consiglio =
114.         (long) (KA_N * sqrt (2 * errore_Fp1F * errore_Fp1 * norma_Fs)) + 1;
115.    free (KA_X);
116.    return ka_poche_suddivisioni;
117.   }
118.   if ( (molt = norma_Fp1F * norma_Fp1 * norma_Fs) < 0.5 ) /* Successo:      */
119.   {PER(i) r -> xp0[i] = KA_Xp[i];                        /* riporta i dati */
120.    r -> eta    = norma_Fp1F;                             /* nel record     */
121.    r -> B      = norma_Fp1;
122.    r -> kappa  = norma_Fs;
123.    r -> raggio = (1 - sqrt(1-2*molt)) * norma_Fp1F / molt;
124.    if ( dp -> fino_in_fondo ) riuscito = 1;              /* se e' il caso  */
125.    else {free(KA_X); return esito_ok;}                   /* smette         */
126.  }}
127.  if ( r -> iterazioni++ >= dp -> tentativi )       /* Massimo di iterazioni */
128.  {free (KA_X);                                     /* raggiunto: riporta il */
129.   r -> iterazioni --;                              /* successo o il         */
130.   return                                           /* fallimento.           */
131.    riuscito ?
132.     esito_ok :
133.     dentro?
134.      ka_iterazioni_insufficienti :
135.      ka_estremi_superati;
136.  }
137.  PER(i) KA_Xp[i] -= vf[i];    /* Calcola la nuova derivata iniziale e itera */
138. }}
139.
140. void KA_F_con_controllo (double t, double x[], double xp[], double xs[])
141. {int i;
142.  PER(i)     /* controlla se x o xp oltrepassano il limite previsto KA_X_ */
143.  {if ( fabs(x[i]) > KA_XM_ ) KA_XM_ = fabs(x[i]);
144.   if ( fabs(xp[i]) > KA_XM_ ) KA_XM_ = fabs(xp[i]);
145.  }
146.  KA_F (t, x, xp, xs);
147. }
148.
149. void KA_FXV_con_controllo (double t, double v[], double vp[], double vs[])
150. {int i;
```

```
151. PER(i)     /* controlla se v o vp oltrepassano il limite previsto KA_V_ */
152. {if ( fabs(v[i]) > KA_VM_ ) KA_VM_ = fabs(v[i]);
153.  if ( fabs(vp[i]) > KA_VM_ ) KA_VM_ = fabs(vp[i]);
154. }
155. i = ((int)((t-KA_A) / KA_H + 0.5)) * KA_D; /* cerca l'elemento di KA_X */
156. KA_FXV (t, KA_X+i, KA_Xp+i, v, vp, vs);    /* al tempo t e calcola FXV */
157. }
```

4.2. *Descrizione del file* `diffeq.c`

La funzione `kantorovich` contiene una chiamata alla funzione `Eulero_s2` e una a `Eulero_s2n`. La prima procedura approssima mediante il metodo di Eulero la soluzione di un problema differenziale i cui dati vengono passati come parametri, e restituisce in due vettori i valori dell'approssimazione e della sua derivata prima ai nodi della suddivisione dell'intervallo. La seconda restituisce solamente il valore finale dell'approssimazione. L'header contenente la descrizione delle due funzioni è il seguente:

```
1. /* diffeq.h  (Mauro Brunato, 21/9/94)
2.    Libreria di funzioni per la risoluzione numerica di equazioni
3.    differenziali ordinarie di primo o di secondo ordine.
4.    Sintassi:
5.      <Nome> (f, a, b, x, n);
6.     dove:
7.      - <Nome> puo' essere
8.          - Eulero (metodo di Eulero semplice)
9.          - Runge_Kutta (metodo di Runge - Kutta del quarto ordine)
10.         - Adams (metodo previsore - correttore del quarto ordine)
11.     - double f (double t, double x) e' la funzione tale che x' = f(t,x);
12.     - a e b sono gli estremi di integrazione;
13.     - x[n+1] e' il vettore double della soluzione, contenente come primo
14.        elemento il dato iniziale;
15.     - n e' il numero di passi;
16.   oppure, per sistemi di equazioni differenziali:
17.     <Nome>_s (f, a, b, x, n, d);
18.    dove:
19.     - <Nome> e' uno dei nomi visti sopra;
20.     - void f (double t, double x[d], double xp[d])
21.        rappresenta la funzione a valori vettoriali del sistema tale che
22.        xp = f (t, x);
23.     - a e b sono gli estremi di integrazione;
24.     - x[n+1][d] e' il vettore di d-uple della soluzione, mentre x[0] e' il
25.        vettore iniziale;
26.     - n e' il numero di passi;
27.     - d e' la dimensione del sistema.
```

```
28.    Oppure, per equazioni differenziali scalari del second'ordine,
29.      <nome>_2 (f, a, b, x, xp, n, d);
30.     dove:
31.      - <nome> e' uno di quelli visti sopra;
32.      - double f (double t, double x, double xp) e' la funzione f
33.         dell'equazione x" = f (t, x, x');
34.      - a e b sono gli estremi;
35.      - x[n+1] e' il vettore che conterra' la soluzione. x[0] e' il dato
36.         iniziale;
37.      - xp e' la derivata prima iniziale;
38.      - n e' il numero di suddivisioni.
39.    Infine, per sistemi di equazioni del second'ordine:
40.      <nome>_s2 (f, a, b, x, xp, n, d);
41.     dove:
42.      - <nome> come sopra;
43.      - void f (double t, double x[d], double xp[d], double xs[d]) restituisce
44.         in xs[d] il valore di f (t, x, x');
45.      - a e b sono gli estremi;
46.      - x[n+1][d] conterr il vettore di d-uple della soluzione; x[0] e' il
47.         dato iniziale;
48.      - xp[n+1][d] e' il vettore di d-uple delle derivate;
49.      - n e' il numero di passi;
50.      - d e' la dimensione del problema.
51.    E' definita inoltre la funzione
52.      Eulero_s2n (f, a, b, x, xp, xf, n, d)
53.     dove i parametri x, xp e xf sono di dimensione d. La funzione pone in
54.     xf l'approssimazione finale e restituisce soltanto quella.
55.    Le funzioni restituiscono il valore per enumerazione esito_ok in
56.    caso di esecuzione corretta, altrimenti l'errore
57.    de_allocazione_fallita (possibile nel caso di grossi sistemi di equazioni
58.    o per n molto grande).
59. */
60. #ifndef _DIFFEQ_H_
61. #define _DIFFEQ_H_
62.
63. #include "errori.h"
64.
65. /* Metodo di Eulero */
66. esito Eulero     (double(*)(double,double), double, double, double[], int);
67. esito Eulero_s   (void(*)(double,double[],double[]),
68.                   double, double, double[][], int, int);
69. esito Eulero_2   (double(*)(double,double,double), double, double,
70.                   double[], double, int);
71. esito Eulero_s2  (void(*)(double,double[],double[],double[]),
72.                   double, double, double[][], double[][], int, int);
73. esito Eulero_s2n (void(*)(double,double[],double[],double[]),
74.                   double, double, double[], double[], double[], int, int);
75.
76. /* Metodo di Runge-Kutta di ordine 4 */
77. esito Runge_Kutta    (double(*)(double,double), double, double, double[], int);
78. esito Runge_Kutta_s  (void(*)(double,double[],double[]),
79.                       double, double, double[][], int, int);
80. esito Runge_Kutta_2  (double(*)(double,double,double),
```

```
81.                         double, double, double[], double, int);
82. esito Runge_Kutta_s2 (void(*)(double,double[],double[],double[]),
83.                         double, double, double[][], double[][], int, int);
84.
85. /* Previsore-correttore di ordine 4 */
86. esito Adams    (double(*)(double,double), double, double, double[], int);
87. esito Adams_s  (void(*)(double,double[],double[]),
88.                  double, double, double[][], int, int);
89. esito Adams_2  (double(*)(double,double,double),
90.                  double, double, double[], double, int);
91. esito Adams_s2 (void(*)(double,double[],double[],double[]),
92.                  double, double, double[][], double[][], int, int);
93. #endif
```

Come si può notare, il pacchetto contiene piú algoritmi di risoluzione: i prototipi di quelli da noi adottati sono definiti alle linee $71 \div 74$. Alle linee $60 \div 89$ del file `diffeq.c` sono definite le funzioni vere e proprie:

```
1. /* diffeq.c  (Mauro Brunato, settembre 1994)
2.    Libreria di funzioni per la risoluzione numerica di
3.    equazioni differenziali.
4. */
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include "diffeq.h"
8.
9. /* Alloca un array di n double */
10. #define DALLOC(n) (double*) malloc (n * sizeof(double))
11.
12. /* Ripetizione per n passi */
13. #define PASSO(i) for ( i = 0; i < n; i++ )
14.
15. /* Ripetizione per d passi */
16. #define PER(i) for ( i = 0; i < d; i++ )
17.
18. /* Ripetizione indefinita */
19. #define RIPETI while ( 1 )
20.
21. /* Controllo dell'allocazione */
22. #define CONTROLLA(p) if ( !p ) return de_allocazione_fallita
23.
24. esito Eulero (double(*f)(double, double),
25.               double a, double b, double x[], int n)
26. {double h = (b-a) / n;
27.  int i;
28.  PASSO(i) x[i+1] = x[i] + h * f(a+i*h, x[i]);
29.  return esito_ok;
```

```
30. }
31.
32. esito Eulero_s (void(*f)(double, double[], double[]),
33.                   double a, double b, double xx[][],
34.                   int n, int d)
35. {double h = (b-a) / n,
36.         *x = (double*)xx,
37.         *val = DALLOC (d);
38.  int i, j;
39.  CONTROLLA (val);
40.  PASSO(i)
41.  {f (a+i*h, x+i*d, val);
42.   PER(j) (x+(i+1)*d)[j] = (x+i*d)[j] + h * val[j];
43.  }
44.  free (val);
45.  return esito_ok;
46. }
47.
48. esito Eulero_2 (double(*f)(double, double, double),
49.                 double a, double b, double x[], double xp, int n)
50. {double h = (b-a) / n;
51.  int i = 0;
52.  n--;
53.  RIPETI
54.  {x[i+1] = x[i] + h * xp;
55.   if ( i == n ) return esito_ok;
56.   xp += h * f(a+i*h, x[i], xp);
57.   i++;
58. }}
59.
60. esito Eulero_s2 (void(*f)(double, double[], double[], double[]),
61.                      double a, double b, double xx[][], double xxp[][],
62.                      int n, int d)
63. {double h = (b-a) / n, *x = (double*)xx, *xp = (double*)xxp,
64.         *val = DALLOC (d), *x1, *xp1;
65.  int i = 0, j;
66.  CONTROLLA (val);
67.  PASSO(i)
68.  {f (a+i*h, x, xp, val);
69.   x1 = x + d; xp1 = xp + d;
70.   PER(j) {x1[j] = x[j] + h * xp[j]; xp1[j] = xp[j] + h * val[j];}
71.   x = x1; xp = xp1;
72.  }
73.  free (val); return esito_ok;
74. }
75.
76. esito Eulero_s2n (void(*f)(double, double[], double[], double[]),
77.                    double a, double b, double x[], double xxp[],
78.                    double xf[], int n, int d)
79. {double h = (b-a) / n, *val = DALLOC (2*d), *xp;
80.  int i = 0, j;
81.  CONTROLLA (val);
82.  xp = val + d;
```

```
 PER(j) {xf[j] = x[j]; xp[j] = xxp[j];}
 PASSO(i)
 {f (a+i*h, xf, xp, val);
  PER(j) {xf[j] += h * xp[j]; xp[j] += h * val[j];}
 }
 free (val); return esito_ok;
}

esito Runge_Kutta (double(*f)(double, double),
                   double a, double b, double x[], int n)
{double k, t = a, x0 = *x, incr, h = (b-a) / n, h2 = h/2, h6 = h/6;
 int i = 0;
 while ( i < n )
 {incr = f (t, x0);
  incr += 2 * (k = f (t+h2, x0+h2*incr));
  incr += 2 * (k = f (t+h2, x0+h2*k));
  *++x = x0 += h6 * (incr + f (t = a+(++i)*h, x0 + h*k));
 }
 return esito_ok;
}

esito Runge_Kutta_s (void(*f)(double, double[], double[]),
                     double a, double b, double xxx[][], int n, int d)
{double *k = DALLOC (3*d), *incr = k+d, *xx = incr+d, *x1,
        *x = (double*) xxx, h = (b-a)/n, h2 = h/2, h6 = h/6, t = a, t1;
 int i = 0, j;
 CONTROLLA (k);
 while ( i < n )
 {f (t, x, incr);
  PER(j) xx[j] = x[j] + h2 * incr[j];
  f (t1 = t+h2, xx, k);
  PER(j) {incr[j] += 2 * k[j]; xx[j] = x[j] + h2 * k[j];}
  f (t1, xx, k);
  PER(j) {incr[j] += 2 * k[j]; xx[j] = x[j] + h * k[j];}
  f (t = a+(++i)*h, xx, k);
  x1 = x + d;
  PER(j) x1[j] = x[j] + h6 * (incr[j] + k[j]);
  x = x1;
 }
 free (k);
 return esito_ok;
}

esito Runge_Kutta_2 (double(*f)(double, double, double),
                     double a, double b, double x[], double xp, int n)
{double t = a, x0 = *x, h = (b-a) / n, h2 = h/2, h6 = h/6,
        t1, k, incr, incrp, xp2, xp3;
 int i = 0;
 RIPETI
 {incr = f (t, x0, incrp = xp);
  incrp += 2 * (xp2 = xp + h2*incr);
  incr += 2 * (k = f (t1 = t + h2, x0 + h2*xp, xp2));
  incrp += 2 * (xp3 = xp + h2*k);
```

```
136.   incr += 2 * (k = f (t1, x0 + h2*xp2, xp3));
137.   ***x = x0 += h6 * (incrp + (xp2 = xp + h*k));
138.   if ( ++i == n ) return esito_ok;
139.   xp += h6 * (incr + f (t = a + i*h, x[-1] + h*xp3, xp2));
140. }}
141.
142. esito Runge_Kutta_s2 (void(*f)(double, double[], double[], double[]),
143.                       double a, double b, double xxx[][], double xxp[][],
144.                       int n, int d)
145. {double t = a, t1, *x = (double*)xxx, *xp = (double*)xxp,
146.         h = (b-a) / n, h2 = h/2, h6 = h/6,
147.         *k = DALLOC(6*d), *incr = k+d, *incrp = incr+d,
148.         *xp2 = incrp+d, *xp3 = xp2+d, *xx = xp3+d;
149.  int i = 0, j;
150.  CONTROLLA (k);
151.  PASSO(i)
152.  {PER(j) incrp[j] = xp[j];
153.   f (t, x, xp, incr);
154.   PER(j)
155.   {incrp[j] += 2 * (xp2[j] = xp[j] + h2*incr[j]);
156.    xx[j] = x[j] + h2*xp[j];
157.   }
158.   f (t1 = t + h2, xx, xp2, k);
159.   PER(j)
160.   {incr[j] += 2 * k[j];
161.    incrp[j] += 2 * (xp3[j] = xp[j] + h2*k[j]);
162.    xx[j] = x[j] + h2*xp2[j];
163.   }
164.   f (t1, xx, xp3, k);
165.   PER(j)
166.   {incr[j] += 2 * k[j];
167.    x[j+d] = x[j] + h6 * (incrp[j] + (xp2[j] = xp[j] + h*k[j]));
168.   }
169.   PER(j) xx[j] = x[j] + h*xp3[j];
170.   f ( t = a + i*h, xx, xp2, k);
171.   PER(j) xp[j+d] = xp[j] + h6 * (incr[j] + k[j]);
172.   x += d;
173.  }
174.  free (k); return esito_ok;
175. }
176.
177. esito Adams (double(*f)(double, double), double a, double b, double x[], int n)
178. {double h = (b-a) / n, h24 = h/24,
179.         fun[4], xx, t;
180.  int i, j;
181.  Runge_Kutta (f, a, t = a+3*h, x, 3);
182.  for ( i = 0; i < 3; i++ ) fun[i] = f (a+i*h,x[i]);
183.  RIPETI
184.  {xx = x[i] + h24 * (55 * (fun[3] = f(t,x[i]))
185.                      - 59 * fun[2] + 37 * fun[1] - 9 * fun[0]);
186.   x[i+1] = x[i] + h24 * (9 * f(t=a+(i+1)*h,xx)
187.                          + 19 * fun[3] - 5 * fun[2] + fun[1]);
188.   if ( ++i == n ) return esito_ok;
```

```
189.   for ( j = 0; j < 3; j++ ) fun[j] = fun[j+1];
190. }}
191.
192. esito Adams_s (void(*f)(double, double[], double[]),
193.                double a, double b, double xxx[][], int n, int d)
194. {double h = (b-a) / n, h24 = h/24,
195.         *xx = DALLOC (5*d), *fun = xx+d, t,
196.         *x = (double*) xxx, *xi, *xi1;
197.  int i, j, k;
198.  CONTROLLA (xx);
199.  if ( (i = Runge_Kutta_s (f, a, a+3*h, xxx, 3, d)) < 0 )
200.  {free (xx);
201.   return i;
202.  }
203.  for ( i = 0; i < 4; i++ ) f (a+i*h, x+i*d, fun+i*d);
204.  i = 3; xi = x + 3*d; xi1 = xi + d;
205.  RIPETI
206.  {PER(j) xx[j] = xi[j] + h24 * (  55 * fun[3*d+j] - 59 * fun[2*d+j]
207.                                 + 37 * fun[d+j]   -  9 * fun[j]);
208.   f (t=a+(i+1)*h, xx, fun);
209.   PER(j) xi1[j] = xi[j] + h24 * (  9 * fun[j]     + 19 * fun[3*d+j]
210.                                  - 5 * fun[2*d+j] +      fun[d+j]);
211.   if ( ++i == n ) {free (xx); return esito_ok;}
212.   for ( k = 0; k < 3*d; k++ )
213.    fun[k] = fun[k+d];
214.   f (t, xi1, fun+3*d);
215.   xi += d; xi1 += d;
216. }}
217.
218. esito Adams_2 (double(*f)(double, double, double),
219.                double a, double b, double x[], double xxp, int n)
220. {double h = (b-a) / n, h2 = h/2, h6 = h/6, h24 = h/24,
221.         fun[4], xp[4], incr, incrp, k, x0 = *x, t = a, t1, xp2, xp3;
222.  int i, j;
223.  *xp = xxp;
224.  for ( i = 0; i < 3; i++ )
225.  {incr = fun[i] = f (t, x0, incrp = xxp);
226.   incrp += 2 * (xp2 = xxp + h2*incr);
227.   incr += 2 * (k = f (t1 = t + h2, x0 + h2*xxp, xp2));
228.   incrp += 2 * (xp3 = xxp + h2*k);
229.   incr += 2 * (k = f (t1, x0 + h2*xp2, xp3));
230.   *++x = x0 += h6 * (incrp + (xp2 = xxp + h*k));
231.   xp[i+1] = xxp += h6 * (incr + f (t = a + (i+1)*h, x[-1] + h*xp3, xp2));
232.  }
233.  RIPETI
234.  {x0  = *x + h24 * (55 * xp[3] - 59 * xp[2] + 37 * xp[1] - 9 * xp[0]);
235.   xp3 = xp[3] + h24 * (55 * (fun[3] = f(t, *x, xp[3]))
236.                          - 59 * fun[2] + 37 * fun[1] - 9 * fun[0]);
237.   x[1] = *x + h24 * (9 * xp3 + 19 * xp[3] - 5 * xp[2] + xp[1]);
238.   xxp += h24 * (9 * f(t = a + (i+1)*h, x0, xp3)
239.                  + 19 * fun[3] - 5 * fun[2] + fun[1]);
240.   if ( ++i == n ) return esito_ok;
241.   for ( j = 0; j < 3; j++ ) {fun[j] = fun[j+1]; xp[j] = xp[j+1];}
```

```
242.   xp[3] = xxp; x++;
243. }}
244.
245. esito Adams_s2 (void (*f)(double, double[], double[], double[]),
246.                 double a, double b, double xxx[][], double xxxp[][],
247.                 int n, int d)
248. {double h = (b-a) / n, h2 = h/2, h6 = h/6, h24 = h/24, t = a, t1,
249.         *k = DALLOC(14*d), *fun = k+d, *xp = fun+4*d, *incr = xp+4*d,
250.         *incrp = incr+d, *xp2 = incrp+d, *xp3 = xp2+d, *xx = xp3+d,
251.         *x = (double*)xxx, *salvaxp = xp, *xxp = (double*)xxxp;
252.  int i, j, l;
253.  CONTROLLA (k);
254.  PER(j) xp[j] = xxp[j];
255.  for ( i = 0; i < 3; i++, x+=d )
256.  {PER(j) incrp[j] = xp[j];
257.   f (t, x, xp, incr);
258.   PER(j)
259.   {fun[i*d+j] = incr[j];
260.    incrp[j] += 2 * (xp2[j] = xp[j] + h2*incr[j]);
261.    xx[j] = x[j] + h2*xp[j];
262.   }
263.   f (t1 = t + h2, xx, xp2, k);
264.   PER(j)
265.   {incr[j] += 2 * k[j];
266.    incrp[j] += 2 * (xp3[j] = xp[j] + h2*k[j]);
267.    xx[j] = x[j] + h2*xp2[j];
268.   }
269.   f (t1, xx, xp3, k);
270.   PER(j) incr[j] += 2 * k[j];
271.   PER(j)
272.   {x[d+j] = x[j] += h6 * (incrp[j] + (xp2[j] = xp[j] + h*k[j]));
273.    xx[j] = x[j] + h*xp3[j];
274.   }
275.   f (t = a + (i+1)*h, xx, xp2, k);
276.   PER(j) xp[d+j] = xp[j] + h6 * (incr[j] + k[j]);
277.   xp += d;
278.  }
279.  xp = salvaxp;
280.  RIPETI
281.  {f (t, x, xp+3*d, fun+3*d);
282.   PER(j)
283.   {xx[j]  = x[j] + h24 * (  55 * xp[3*d+j] - 59 * xp[2*d+j]
284.                          + 37 * xp[d+j]   -  9 * xp[j]);
285.    xp3[j] = xp[3*d+j] + h24 * (  55 * fun[3*d+j] - 59 * fun[2*d+j]
286.                               + 37 * fun[d+j]   -  9 * fun[j]);
287.   }
288.   f (t = a + (i+1)*h, xx, xp3, k);
289.   PER(j)
290.   {x[j+d] = x[j] + h24 * (  9 * xp3[j]    + 19 * xp[3*d+j]
291.                          - 5 * xp[2*d+j] +      xp[d+j]);
292.    xxp[j+d] = xxp[j] + h24 * (  9 * k[j]       + 19 * fun[3*d+j]
293.                               - 5 * fun[2*d+j] +      fun[d+j]);
294.   }
```

```
295.   if ( ++i == n ) return esito_ok;
296.   for ( l = 0; l < 3*d; l++ ) {fun[l] = fun[l+d]; xp[l] = xp[l+d];}
297.   PER(j) xp[3*d+j] = xxp[j];
298.   x+=d;
299. }}
```

4.3. *I files* `matinv.h` *e* `matinv.c`

Ecco di seguito l'header e il corpo della funzione di inversione matriciale usata dalla funzione `kantorovich`.

```
1. /* matinv.h  (Mauro Brunato, ottobre 1994)
2.    Algoritmi d'inversione di matrici.
3.    Sintassi:
4.                inversione (A, B, d)
5.      dove
6.       - A[d][d] e' la matrice da invertire;
7.       - B[d][d] e' lo spazio per l'inversa;
8.       - d e' l'ordine.
9.    La matrice A non viene distrutta.
10.   La funzione restituisce i messaggi:
11.    - mi_allocazione_fallita nel caso in cui l'allocazione della memoria
12.       necessaria non riesca;
13.    - mi_matrice_singolare se la matrice non puo' essere invertita;
14.    - esito_ok se l'inversione e' riuscita.
15. */
16. #ifndef _MATINV_H_
17. #define _MATINV_H_
18.
19. #include "errori.h"
20.
21. esito inversione (double[][], double[][], int);
22.
23. #endif
```

```
1. /* matinv.c  (Mauro Brunato, ottobre 1994)
2.    Algoritmo di inversione matriciale.
3. */
4. #include <stdlib.h>
5. #include <math.h>
6. #include "matinv.h"
7.
8. #define QUASI_ZERO 1e-10
```

```
 9.
10. esito inversione (double A[][], double B[][], int d)
11. {int i, j, k;
12.  double *b = (double*)B, q,
13.         *m = (double*) malloc (d*d*sizeof(double));
14.  if ( !m ) return mi_allocazione_fallita;
15.  for ( i = 0; i < d*d; i++ ) m[i] = ((double*)A)[i];
16.  for ( i = 0; i < d; i++ )
17.   for ( j = 0; j < d; j++ )
18.    b[i*d+j] = (i == j) ? 1 : 0;
19.  for ( i = 0; i< d; i++ )
20.  {if ( fabs(m[i*d+i]) < QUASI_ZERO ) {free (m); return mi_matrice_singolare;}
21.   q = 1 / m[i*d+i];
22.   m[i*d+i] = 1;
23.   for ( j = i + 1; j < d; j++ ) m[i*d+j] *= q;
24.   for ( j = 0; j < d; j++ ) b[i*d+j] *= q;
25.   for ( j = 0; j < d; j++ )
26.    if ( j != i )
27.    {q = -m[j*d+i];
28.     m[j*d+i] = 0;
29.     for ( k = i+1; k < d; k++ ) m[j*d+k] += q * m[i*d+k];
30.     for ( k = 0; k < d; k++ ) b[j*d+k] += q * b[i*d+k];
31.  } }
32.  free (m); return esito_ok;
33. }
```

4.4. *Il file di definizione degli errori*

L'header `errori.h` definisce il tipo per enumerazione `esito` i cui valori sono tutti i possibili messaggi di errore riportabili dalle procedure sopra descritte (si noti che tutti gli headers riportano la metaistruzione "`#include <errori.h>`"). Gli errori sono abbastanza autoesplicativi. Si veda la descrizione di `p-kan.c` per un esempio di interpretazione degli errori da parte del `main()`.

```
 1. /* errori.h  (Mauro Brunato, ottobre 1994)
 2.    Raccolta dei codici di errore dei moduli diffeq, integra, matinv e kantorov.
 3. */
 4. #ifndef _ERRORI_H_
 5. #define _ERRORI_H_
 6.
 7. typedef enum
 8. {/* esito positivo, uguale per tutti i moduli */
 9.  esito_ok,
10.
11.  /* errori di diffeq */
```

```
12.  de_allocazione_fallita,
13.
14.  /* errori di matinv */
15.  mi_allocazione_fallita,
16.  mi_matrice_singolare,
17.
18.  /* errori di kantorov */
19.  ka_estremi_superati,
20.  ka_allocazione_fallita,
21.  ka_iterazioni_insufficienti,
22.  ka_poche_suddivisioni
23. } esito;
24.
25. #define ERRORE(e) ((e) != esito_ok)
26.
27. #endif
```

4.5. *I files di definizione del problema*

L'ultima libreria che descriveremo è quella relativa alla definizione delle due funzioni i cui puntatori andranno caricati nel record di tipo `dati_funzione`:

```
1. /* f-kan.h  (Mauro Brunato, ottobre 1994)
2.    Funzione vettoriale ad uso di P-KAN.C.
3. */
4. #ifndef _F_KAN_H_
5. #define _F_KAN_H_
6.
7. extern char nome_della_funzione[];
8. extern int dimensione;
9.
10. void   Funzione (double, double[], double[], double[]);
11. void   Derivate (double, double[], double[], double[], double[], double[]);
12.
13. #endif
```

Un esempio di definizione è contenuto nel file `f-kan.c`, che andrà modificato a seconda del problema da studiare. Si veda la prossima sezione per una discussione dell'esempio cui questo file si riferisce:

```
/* f-kan.c  (Mauro Brunato, ottobre 1994)
   Definizione della funzione da utilizzare in p-kan.c
   (vedere f-kan.h)
*/
#include <math.h>
#include "f-kan.h"

char nome_della_funzione[] =
 " x0\" = (x0x1 - 1)/6 + e^t\n x1\" = (x0 + x1')/2 - e^t,   0 < t < 1";

int dimensione = 2;

void Funzione (double t, double x[2], double xp[2], double xs[2])
{double p = exp(t);
 xs[0] = (x[0]*x[1] - 1) / 6 + p;
 xs[1] = (x[0] + xp[1]) / 2 - p;
}

void Derivate (double t, double x[2], double xp[2],
                         double v[2], double vp[2], double vs[2])
{vs[0] = (x[1]*v[0] + x[0]*v[1]) / 6;
 vs[1] = (v[0] + vp[1]) / 2;
}
```

Gli altri dati (gli estremi di integrazione, i valori al bordo e le varie costanti) sono definiti su un file a parte (`p-kan.dat`), in modo da evitare per quanto possibile la ricompilazione del programma durante lo studio di un problema. Il file testo ha un formato abbastanza libero. Il programma principale che lo legge si limita a cercare tutte le linee contenenti un uguale e a leggere il numero che segue. L'ordine in cui debbono comparire i dati è prefissato: se lo si vuole modificare, bisogna riordinare anche le linee di `main()` preposte alla lettura. Un esempio di questo file è il seguente:

```
p-kan.dat  (Mauro Brunato, ottobre 1994)

Dati per l'esecuzione del programma p-kan


       Questo file puo' contenere qualunque cosa, a patto che siano
       presenti 17 righe contenenti un segno di "uguale", e che
       l'uguale sia seguito da un numero. Il significato dei numeri
       e' quello riportato riga per riga, e il loro ordine
       NON DEV'ESSERE MODIFICATO.

```

```
12.
13. Estremi di integrazione:
14.
15.   a = 0
16.   b = 1
17.
18. Valori massimi delle soluzioni delle equazioni in gioco:
19.
20.   Massimo ipotizzato per x e x': X_ = 2
21.   Massimo ipotizzato per v e v': V_ = 2
22.
23. Valori massimi della funzione f e delle sue derivate nel dominio [a,b]*B(0,X_):
24.
25.   Massimo di f:   S   = 5
26.   Massimo di ft:  Lt  = 3
27.   Massimo di fx:  Lx  = 1
28.   Massimo di fxt: Mxt = 0
29.   Massimo di fxx: Mxx = 0.33333333333333333
30.
31. Condizioni al contorno:
32.
33.   x(a) = 0
34.   y(a) = 0
35.
36.   x(b) = 0
37.   y(b) = 0
38.
39. Derivate iniziali per il primo tentativo:
40.
41.   x'(a) = 0
42.   y'(a) = 0
43.
44. Numero massimo di tentativi prima del fallimento = 10
45.
46. Numero di suddivisioni dell'intervallo di integrazione = 1000
47.
48. Flag per la prosecuzione dell'iterazione fino al compimento
49. delle iterazioni previste (0 se il programma si deve fermare
50. appena trovato un intorno accettabile, 1 se il programma
51. deve proseguire le iterazioni):
52.
53. fino in fondo = 1
```

4.6. *Il programma principale*

Il programma principale utilizza, direttamente o indirettamente, tutte le librerie di funzioni viste finora. Il suo scopo è di inizializzare i record per la chiamata a `kantorovich` con i puntatori alle funzioni definite in `f-kan.c` e con i dati di `p-kan.dat`, che deve leggere e interpretare per mezzo delle due funzioni `leggi_prossimo_reale` e `leggi_prossimo_intero`. Una volta chiamata la funzione `kantorovich` (alla linea 61), deve interpretare il codice di ritorno e, se la funzione ha avuto successo, scrivere i risultati. Ecco il listato del file `p-kan.c`:

```
1. /* p-kan.c  (Mauro Brunato, ottobre 1994)
2.    [standard + diffeq + matinv + kantorov + f-kan]
3.    Cerca di determinare la soluzione (se c'e') al problema differenziale
4.    ordinario del secondo ordine descritto in f-kan.c
5.    dove le condizioni al contorno sono definite nel file p-kan.dat.
6. */
7. #include <stdio.h>
8. #include <stdlib.h>
9. #include <math.h>
10. #include "kantorov.h"
11. #include "f-kan.h"
12.
13. /* Nomi dei files */
14. #define NOME_INPUT "p-kan.dat"
15. #define NOME_OUTPUT "p-kan.ris"
16.
17. #define PER(i) for ( i = 0; i < dimensione; i++ )
18.
19. /* Prototipi delle funzioni */
20. double leggi_prossimo_reale (FILE*);
21. long   leggi_prossimo_intero (FILE*);
22.
23. int main (void)
24. {double *x0, *x1, *xp0i, *xp0;
25.  int i, j;
26.  dati_funzione df;
27.  dati_problema dp;
28.  risultati r;
29.  FILE *f;
30.  if ( !(x0 = (double*)malloc(4*dimensione*sizeof(double))) )
31.  {puts ("Errore di allocazione."); return 1;}
32.  x1 = x0+dimensione; xp0i = x1+dimensione; xp0 = xp0i+dimensione;
33.  if ( !(f = fopen(NOME_INPUT, "r")) )   /* apertura del file dati */
34.  {puts ("File dati non trovato."); return 1;}
35.  df.a    = leggi_prossimo_reale (f);
36.  df.b    = leggi_prossimo_reale (f);
37.  df.X_   = leggi_prossimo_reale (f);
```

```
38. df.V_   = leggi_prossimo_reale (f);
39. df.S    = leggi_prossimo_reale (f);
40. df.Lt   = leggi_prossimo_reale (f);
41. df.Lx   = leggi_prossimo_reale (f);
42. df.Mxt  = leggi_prossimo_reale (f);
43. df.Mxx  = leggi_prossimo_reale (f);
44. PER(i) x0[i]   = leggi_prossimo_reale (f);
45. PER(i) x1[i]   = leggi_prossimo_reale (f);
46. PER(i) xp0i[0] = leggi_prossimo_reale (f);
47. dp.tentativi     = leggi_prossimo_intero (f);
48. dp.numero_nodi   = leggi_prossimo_intero (f);
49. dp.fino_in_fondo = leggi_prossimo_intero (f);
50. fclose (f);
51. df.dimensione = dimensione; df.f = Funzione; df.fxv = Derivate;
52. dp.dato_iniziale = x0; dp.dato_finale = x1;
53. dp.derivata_iniziale = xp0i; r.xp0 = xp0;
54. puts ("Risoluzione del problema");
55. puts (nome_della_funzione);
56. PER(i) printf (" x%1i(%lf) = %lf,", i, df.a, x0[i]);
57. putchar ('\n');
58. PER(i) printf (" x%1i(%lf) = %lf,", i, df.b, x1[i]);
59. printf ("\n(%ld suddivisioni, Max. %d iterazioni)\n",
60.         dp.numero_nodi, dp.tentativi);
61. switch ( Kantorovic (&df, &dp, &r) )
62. {case ka_allocazione_fallita:
63.  case de_allocazione_fallita:
64.  case mi_allocazione_fallita:
65.   puts ("--- Allocazione fallita. ---");
66.   return 1;
67.  case mi_matrice_singolare:
68.   puts ("--- Differenziale non invertibile. ---");
69.   return 1;
70.  case ka_estremi_superati:
71.   puts ("\n --- Funzione oltre gli estremi programmati. ---");
72.   return 1;
73.  case ka_iterazioni_insufficienti:
74.   puts ("\n --- Numero massimo di iterazioni superato. ---");
75.   return 1;
76.  case ka_poche_suddivisioni:
77.   puts ("\n --- Suddivisione troppo grossolana. ---");
78.   printf ("Sono consigliati %ld intervalli.\n",
79.            r.consiglio);
80.   return 0;
81.  case esito_ok:
82.   printf ("\n --- Dimostrazione riuscita in %d iterazioni. ---\n",
83.           r.iterazioni);
84.   printf ("Soluzione: ( ");
85.   PER(i) printf ("%le ", xp0[i]);
86.   printf (") +/- %le\n(eta = %le, B = %le, kappa = %le).\n",
87.           r.raggio, r.eta, r.B, r.kappa);
88.   return 0;
89. }}
90.
```

```
91. double leggi_prossimo_reale (FILE *f)
92. {char s[80];
93.  int i;
94.  do
95.  {fgets (s, 80, f);
96.   for ( i = 0; s[i] != '=' && s[i] >= ' '; i++ );
97.  } while ( s[i] != '=' && !feof(f) );
98.  return feof(f)? (double)0.0 : atof (s+i+1);
99. }
100.
101. long leggi_prossimo_intero (FILE *f)
102. {char s[80];
103.  int i;
104.  do
105.  {fgets (s, 80, f);
106.   for ( i = 0; s[i] != '=' && s[i] >= ' '; i++ );
107.  } while ( s[i] != '=' && !feof(f) );
108.  return feof(f)? 0L : atol (s+i+1);
109. }
```

5 — Un esempio d'uso del programma

Il programma è stato verificato con alcuni sistemi di equazioni del secondo ordine in dimensione 2. I files `f-kan.c` e `p-kan.dat` riportati sopra si riferiscono al sistema

$$\begin{cases} \ddot{x} = \dfrac{xy-1}{6} + e^t & t \in [0,1] \\ \ddot{y} = \dfrac{x+\dot{y}}{2} - e^t & \\ x(0) = y(0) = 0 & \\ x(1) = y(1) = 0, & \end{cases} \tag{8}$$

da risolvere con un massimo di dieci iterazioni e con l'intervallo $[0,1]$ diviso in mille parti uguali. L'algoritmo ha l'indicazione di non fermarsi una volta ottenuta la dimostrazione, ma di proseguire fino alla decima iterazione allo scopo di fornire un risultato piú preciso. Il valore di partenza per entrambe le derivate iniziali è zero.

La tabella 1 riporta l'output del programma. Il raggio che descrive l'errore con il quale è nota la soluzione è in realtà una maggiorazione molto pessimista.

Sostituendo il numero di nodi (1000) con un valore molto piú piccolo (ad esempio 50) il programma verifica un errore troppo alto (vedi tabella 2) e suggerisce l'impiego di un

```
Risoluzione del problema
 x0" = (x0x1 - 1)/6 + e{}^t
 x1" = (x0 + x1')/2 - e{}^t,   0 < t < 1
 x0(0.000000) = 0.000000, x1(0.000000) = 0.000000,
 x0(1.000000) = 0.000000, x1(1.000000) = 0.000000,
(1000 suddivisioni, Max. 10 iterazioni)

 --- Dimostrazione riuscita in 10 iterazioni. ---
Soluzione: ( -6.320880e-01 6.748392e-01 ) +/- 8.050135e-03
(eta = 7.385221e-03, B = 1.024119e+00, kappa = 2.003730e+01).
```

Tabella 1: Output del programma `p-kan.c`*, 1000 nodi.*

```
Risoluzione del problema
 x0" = (x0x1 - 1)/6 + e{}^t
 x1" = (x0 + x1')/2 - e{}^t,   0 < t < 1
 x0(0.000000) = 0.000000, x1(0.000000) = 0.000000,
 x0(1.000000) = 0.000000, x1(1.000000) = 0.000000,
(50 suddivisioni, Max. 10 iterazioni)

 --- Suddivisione troppo grossolana. ---
Sono consigliati 363 intervalli.
```

Tabella 2: Output del programma `p-kan.c`*, 50 nodi.*

```
Risoluzione del problema
 x0" = (x0x1 - 1)/6 + e{}^t
 x1" = (x0 + x1')/2 - e{}^t,   0 < t < 1
 x0(0.000000) = 0.000000, x1(0.000000) = 0.000000,
 x0(1.000000) = 0.000000, x1(1.000000) = 0.000000,
(363 suddivisioni, Max. 10 iterazioni)

 --- Dimostrazione riuscita in 10 iterazioni. ---
Soluzione: ( -6.301141e-01 6.726968e-01 ) +/- 4.140547e-02
(eta = 2.299049e-02, B = 1.072128e+00, kappa = 2.003730e+01).
```

Tabella 3: Output del programma `p-kan.c`*, 363 nodi.*

numero maggiore di nodi. L'euristica utilizzata per determinare il numero di nodi necessari non è infallibile (si tratta di una semplice proporzione), ma in questo caso particolare funziona, anche se restituisce un intorno piú ampio (tabella 3).

Capitolo II
Equazioni del primo ordine: Considerazioni sul metodo di Eulero

Come s'è visto, le maggiorazioni trovate nel capitolo precedente sono molto pessimiste. Il bisogno di un errore abbastanza basso costringe il programma ad utilizzare un numero di passi molto elevato, talora proibitivo (ci sembra opportuno confessare che i coefficienti del problema (I.8) sono stati impostati *ad hoc*) in termini di tempo o, piú facilmente, di memoria.

In questo capitolo si propone una tecnica per la determinazione *a posteriori* della migliore suddivisione a passo variabile per un problema differenziale ordinario del primo ordine. Lo scopo è quello di minimizzare il costo del calcolo una volta che è stato fissato l'errore che si desidera ottenere. Si dimostrerà che questa tecnica è asintoticamente ottimale.

La tecnica verrà poi esemplificata tramite un semplice programma FORTRAN che la applica a un problema di cui è nota la soluzione, allo scopo di verificarne, almeno in un caso concreto, l'effettivo funzionamento.

1 — Il Metodo di Eulero: una stima a posteriori dell'errore

Si vuole risolvere l'equazione del primo ordine in dimensione $d \geq 1$:

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}) & t \in [a, b] \\ \mathbf{x}(a) = \mathbf{x}_0 \end{cases} \tag{1}$$

dove, dati lo spazio di Hilbert $H = \mathbb{R}^d$, l'intervallo sull'"asse dei tempi" $[a, b]$, la funzione $\mathbf{f} \in \mathcal{C}^1([a, b] \times H, H)$ e il dato iniziale $\mathbf{x}_0 \in H$, si cerca un'approssimazione di $\mathbf{x} \in \mathcal{C}^1([a, b], H)$ per mezzo dell'algoritmo di Eulero.

Data una suddivisione $\{t_k\}_{k=0,\ldots,N}$ dell'intervallo $[a, b]$, tale cioè che $a = t_0 < t_1 < \cdots < t_N = b$, si costruisce la successione $\{\mathbf{x}_k\}_{k=0,\ldots,N}$ per induzione a partire dal dato iniziale x_0:

$$\mathbf{x}_n = \mathbf{x}_{n-1} + \mathbf{f}(t_{n-1}, \mathbf{x}_{n-1})\Delta t_n \qquad (n = 1, \ldots, N).$$

con $\Delta t_n = t_n - t_{n-1} \quad (n = 1, \ldots, N)$.

Introduciamo ora la seguente notazione: sia $\mathbf{x}(t)$ la soluzione del problema (1) e sia $\mathbf{x}_n(t)$ la soluzione esatta del seguente problema:

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}) & t \in [t_n, t_{n+1}] \\ \mathbf{x}(t_n) = \mathbf{x}_n \end{cases}$$

Sia inoltre

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{x}) = \mathbf{f}_t(t, \mathbf{x}) + \mathbf{f}_{\mathbf{x}}(t, \mathbf{x}) \cdot \mathbf{f}(t, \mathbf{x}).$$

Siamo interessati a valutare l'errore indotto dal metodo di Eulero sulla soluzione all'istante finale $b = t_N$. Vogliamo cioè stimare la grandezza $|\mathbf{x}(b) - \mathbf{x}_N|$. All'$n$-esimo passo, il metodo di Eulero introduce l'errore $\mathbf{x}_n(t_{n+1}) - \mathbf{x}_{n+1}$, che poi si propagherà, in generale amplificandosi, fino all'istante finale.

Se Δt_{n+1} è abbastanza piccolo, l'errore introdotto da un singolo passaggio del metodo di Eulero può essere stimato mediante il termine di primo grado della serie di Taylor trascurato da tale metodo. Scriviamo cioè:

$$\begin{aligned} \mathbf{x}_n(t_{n+1}) &\approx \mathbf{x}_n(t_n) + \dot{\mathbf{x}}_n(t_n)\Delta t_{n+1} + \frac{1}{2}\ddot{\mathbf{x}}_n(t_n)(\Delta t_{n+1})^2 = \\ &\approx \mathbf{x}_n + \mathbf{f}(t_n, \mathbf{x}_n)\Delta t_{n+1} + \frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n)(\Delta t_{n+1})^2. \end{aligned} \tag{2}$$

L'errore introdotto all'n-esimo passo sarà dunque

$$\mathbf{x}_n(t_{n+1}) - \mathbf{x}_{n+1} \approx \frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n)(\Delta t_{n+1})^2. \tag{3}$$

Supponendo dunque che la suddivisione sia abbastanza fitta, e che Δt_n non sia troppo variabile su piccola scala, ovvero che intervalli contigui abbiano ampiezze simili, introduciamo la funzione scalare $u(t)$, continua, tale che $u(t_n) \approx \Delta t_n$. Il numero di suddivisioni per unità di tempo sarà dunque circa $1/u(t)$. Introduciamo inoltre la funzione $d\mathbf{v}(t)$ che rappresenta una stima dell'errore di discretizzazione dell'intervallo $[t, t + \mathrm{d}t]$, supponendo che la suddivisione utilizzata sia più piccola del lasso di tempo dt fissato.

$$\mathrm{d}\mathbf{v}(t) \approx \overbrace{\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{x}(t))u^2(t)}^{\text{errore in un passo}} \cdot \overbrace{\frac{1}{u(t)}\mathrm{d}t}^{\substack{\text{numero} \\ \text{di passi}}} \tag{4}$$

Vediamo ora come l'errore $d\mathbf{v}(t)$, introdotto all'istante generico t, si propaga fino all'istante finale b. Siano $\mathbf{x}$ e $\mathbf{x}_1$ due soluzioni di $\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$ con valori iniziali diversi, e sia $\mathbf{v}(t) = \mathbf{x}_1(t) - \mathbf{x}(t)$. Studiamo la variazione di $\mathbf{v}$:

$$\begin{aligned}\dot{\mathbf{v}}(t) = \dot{\mathbf{x}}_1(t) - \dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}_1(t)) - \mathbf{f}(t, \mathbf{x}(t)) = \\ = \mathbf{f}(t, \mathbf{x}(t) + \mathbf{v}(t)) - \mathbf{f}(t, \mathbf{x}(t)).\end{aligned}$$

Se $\mathbf{v}(t)$ è molto piccolo (come nel nostro caso):

$$\mathbf{f}(t, \mathbf{x}(t) - \mathbf{v}(t)) - \mathbf{f}(t, \mathbf{x}(t)) \approx \mathbf{f}_{\mathbf{x}}(t, \mathbf{x}(t)) \cdot \mathbf{v}(t).$$

Chiamiamo $A(t) = \mathbf{f}_{\mathbf{x}}(t, \mathbf{x}(t))$. $A(t)$ è una matrice $n \times n$ dipendente dal tempo. Per $\mathbf{v}$ piccolo, esso risponderà all'equazione differenziale lineare

$$\dot{\mathbf{v}}(t) = A(t)\mathbf{v}(t). \tag{5}$$

Chiamiamo v il modulo di $\mathbf{v}$. Siccome $v^2 = (\mathbf{v}, \mathbf{v})$, si ha

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}t}v^2 = \frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{v}, \mathbf{v}) = 2(\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}, \mathbf{v}) \\ \approx 2(A(t)\mathbf{v}, \mathbf{v}) = 2(A(t)\frac{\mathbf{v}}{v}, \frac{\mathbf{v}}{v})v^2 \leq 2a(t)v^2,\end{aligned}$$

dove $a(t) = \sup_{|\mathbf{v}|=1}(A(t)\mathbf{v}, \mathbf{v}) = \|A(t)\|_\rho$, e si dimostra essere una norma tale che $\|A\|_\rho \leq n\rho(A)$, essendo n la dimensione dello spazio e $\rho(A)$ il raggio spettrale della matrice A.

D'altra parte,

$$\frac{\mathrm{d}}{\mathrm{d}t}v^2 = 2v\frac{\mathrm{d}}{\mathrm{d}t}v,$$

dunque si ha la maggiorazione

$$\dot{v}(t) \leq a(t)v(t), \tag{6}$$

che porta a una stima dell'entità della propagazione dell'errore commesso all'istante t:

$$v(b) \leq v(t) \exp\left(\int_t^b a(s)\mathrm{d}s\right). \tag{7}$$

Dalla (7), integrata con la (4), otteniamo una stima *a posteriori* dell'errore di discretizzazione sul dato finale causato da una suddivisione descrivibile tramite la funzione $u(t)$:

$$|\mathbf{x}(b) - \mathbf{x}_N| \approx \frac{1}{2}\int_a^b \exp\left(\int_t^b a(s)\mathrm{d}s\right)\left|\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{x}(t))\right| u(t)\mathrm{d}t.$$

Un ultimo appunto: sia $v(t)$ una stima superiore dell'errore al tempo t. L'equazione differenziale che tiene conto dell'errore di discretizzazione compiuto nell'intervallo $[t, t+\mathrm{d}t]$ e della propagazione degli errori precedenti è

$$\dot{v}(t) = a(t)v(t) + \gamma(t)u(t), \tag{8}$$

dove

$$\gamma(t) = \frac{1}{2}\left|\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{x}(t))\right|.$$

2 — Minimizzazione dell'errore di discretizzazione

Ci proponiamo ora di determinare, fissato un errore massimo E, la suddivisione piú opportuna per mantenere l'errore di discretizzazione sul dato finale al di sotto di E:

$$|\mathbf{x}(b) - \mathbf{x}_N| \leq E.$$

Date opportune condizioni sulla funzione u, che studieremo in una prossima sezione, il numero di elementi di una suddivisione è stimato dall'integrale $\int_a^b 1/u(t)\mathrm{d}t$, per cui il problema diviene, fissato E,

$$\text{minimizzare} \qquad \int_a^b \frac{\mathrm{d}t}{u(t)} \tag{9}$$

$$\text{soggetto al vincolo} \qquad \int_a^b \beta(t)u(t)\mathrm{d}t = E \tag{10}$$

con

$$\beta(t) = \frac{1}{2}\exp\left(\int_t^b a(s)\mathrm{d}s\right)\left|\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{x}(t))\right|.$$

Si tratta di un problema di minimo lagrangiano, ove la soluzione è quella funzione $u_E(t)$ che rende stazionario l'integrale $\int_a^b L(t, u(t))\mathrm{d}t$, con

$$L(t, u(t)) = \frac{\lambda_1}{u(t)} + \lambda_2\beta(t)u(t) \tag{11}$$

per qualche λ_1, λ_2. Si ottiene

$$0 = \frac{\partial L}{\partial u} = -\frac{1}{u_E^2(t)} + \lambda\beta(t),$$

da cui, supponendo che esista una costante $c > 0$ tale che $\forall t \in [a,b] \quad \beta(t) \geq c$,

$$u_E(t) = \frac{hE}{\sqrt{\beta(t)}}, \tag{12}$$

dove h è una costante indipendente dall'errore imposto E, e si ricava dalla (10):

$$h = \frac{1}{\int_a^b \sqrt{\beta(t)}\mathrm{d}t}. \tag{13}$$

Si osservi che, chiamando $u_1(t)$ la funzione corrispondente a $E = 1$, si ha

$$u_E(t) = E \cdot u_1(t).$$

In corrispondenza di questa u_E ottimale l'integrale da minimizzare, ovvero il numero di passi da effettuare, vale

$$N_E = \int_a^b \frac{\mathrm{d}t}{u(t)} = \int_a^b \frac{\sqrt{\beta(t)}}{hE}\mathrm{d}t = \frac{1}{h^2 E}.$$

Segue che, nella speranza di ottenere un errore vicino a quello imposto, si può determinare la funzione $u_E(t)$ come sopra e imporre $t_0 = a$, $t_i = t_{i-1} + u_E(t_{i-1})$ fino al primo indice, che chiameremo N_E^*, per il quale $t_{N_E^*} \geq b$. Imporremo infine $t_{N_E^*} = b$. Nel seguito, ci riferiremo a questo metodo come al metodo $\mathcal{A}$.

Questo problema ha un duale naturale: la minimizzazione a costo prefissato dell'errore di discretizzazione. Risultando ovviamente la stessa lagrangiana (11), la soluzione è la stessa, a meno di una costante. Per un costo K si ottiene:

$$\begin{aligned} u^{(K)} &= \frac{\int_a^b \sqrt{\beta(t)}\mathrm{d}t}{K} \cdot \frac{1}{\sqrt{\beta(t)}} \\ &= \frac{u^{(1)}}{K}, \end{aligned}$$

con ovvio significato dei termini.

3 — Bontà delle stime di errore e di costo

Vogliamo verificare se, almeno asintoticamente (per E piccolo a piacere), la suddivisione trovata è quella ottimale.

3.1. *Correttezza della stima del numero di passi*

Chiamiamo

$$N_E = \int_a^b \frac{\mathrm{d}t}{u_E(t)}$$

la quantità minimizzata nella (9), e sia N_E^* il numero ricavato alla fine della sezione 2, ovvero il numero effettivo di intervalli ricavati a partire dalla funzione u_E. Dobbiamo verificare che, almeno per $E \to 0$, N_E descrive correttamente N_E^*. In altri termini, vogliamo che

$$\lim_{E\to 0} \frac{N_E}{N_E^*} = 1.$$

Innanzitutto, supponiamo che $\sqrt{\beta(t)}$ sia lipschitziana con costante B. Allora, per $a', b' \in [a, b]$, si ha

$$\left| \frac{1}{u_E(a')} - \frac{1}{u_E(b')} \right| = \frac{1}{hE} \left| \sqrt{\beta(a')} - \sqrt{\beta(b')} \right| \leq \frac{B}{hE} |a' - b'|.$$

Siano $m = \inf_{t\in[a,b]} (\beta(t))^{-1/2}$ e $M = \sup_{t\in[a,b]} (\beta(t))^{-1/2}$. Per le condizioni imposte nella sezione precedente, si ha $0 < m \leq M < \infty$. Inoltre, $\forall t \in [a, b] \quad mhE \leq u_E(t) \leq MhE$, ovvero[3] $\forall t \quad u_E(t) = \Theta(E)$ (d'ora in poi sottintenderemo "per $E \to 0$"). Otteniamo infine

$$\frac{b-a}{MhE} \leq N_E^* \leq \frac{b-a}{mhE}, \qquad \text{cioè} \qquad N_E^* = \Theta\left(\frac{1}{E}\right).$$

Sia ora $\xi \in [t_{i-1}, t_i]$. Allora si ha

$$\begin{aligned} \left| \frac{1}{u_E(\xi)} - \frac{1}{u_E(t_{i-1})} \right| &\leq \frac{B}{hE} |\xi - t_{i-1}| \leq \frac{B}{hE} (t_i - t_{i-1}) \leq \frac{B}{hE} u_E(t_{i-1}) \\ &\leq \frac{B}{hE} MhE = MB = O(1). \end{aligned}$$

Dunque,

$$\frac{1}{u_E(\xi)} = \frac{1}{u_E(t_{i-1})} + O(1).$$

Possiamo dunque concludere che:

$$N_E = \int_a^b \frac{\mathrm{d}t}{u_E(t)} = \sum_{i=1}^{N_E^*} \int_{t_{i-1}}^{t_i} \frac{\mathrm{d}t}{u_E(t)} = \sum_{i=1}^{N_E^*} (t_i - t_{i-1}) \frac{1}{u_E(\xi_i)} \quad \text{con} \quad \xi_i \in [t_{i-1}, t_i]$$

[3] Come di consueto, per $x \to x_0$ scriviamo $f = O(g)$ se esiste una costante $C > 0$ tale che $|f(x)| \leq C|g(x)|$ in un intorno di x_0. Scriviamo inoltre $f = \Theta(g)$ se $f = O(g)$ e $g = O(f)$.

$$= \sum_{i=1}^{N_E^*} \frac{u_E(t_{i-1})}{u_E(\xi_i)} = \sum_{i=1}^{N_E^*} u_E(t_{i-1}) \left[\frac{1}{u_E(t_{i-1})} + O(1) \right]$$
$$= \sum_{i=1}^{N_E^*} [1 + O(E)] = N_E^* + N_E^* O(E) = N_E^* + \Theta(1/E) O(E)$$
$$= N_E^* + O(1).$$

N_E e N_E^* sono dunque asintotiche.

Per valori di E abbastanza piccoli, dunque, la grandezza minimizzata dalle (9) e (10) rappresenta correttamente il costo della risoluzione del problema differenziale.

3.2. *Correttezza della stima dell'errore*

Allo stesso modo dobbiamo verificare l'asintoticità di E, errore stimato dalla (10), con l'errore effettivamente valutato con la suddivisione trovata a partire da u_E, vale a dire:

$$E^* = \sum_{i=0}^{N_E^*} \beta(t_i)(t_{i+1} - t_i)^2.$$

Per questo, fatte buone le considerazioni del paragrafo precedente e sapendo che anche u'_E è dello stesso ordine di E abbiamo, posto $\xi_i \in [t_{i-1}, t_i]$, la seguente uguaglianza:

$$u_E(\xi_i) = u_E(t_{i-1}) + u'_E(\psi_i)(\xi_i - t_{i-1}) = u_E(t_{i-1}) + O(E^2).$$

Con un calcolo del tutto simile al precedente otteniamo:

$$E = \int_a^b \beta(t) u_E(t) \mathrm{d}t = \sum_{i=1}^{N_E^*} \int_{t_{i-1}}^{t_i} \beta(t) u_E(t) \mathrm{d}t$$
$$= \sum_{i=1}^{N_E^*} (t_i - t_{i-1}) \beta(\xi_i) u_E(\xi_i) \qquad \text{con} \quad \xi_i \in [t_{i-1}, t_i]$$
$$= \sum_{i=1}^{N_E^*} u_E(t_{i-1}) [\beta(t_{i-1}) + O(E)][u_E(t_{i-1}) + O(E^2)]$$
$$= \sum_{i=1}^{N_E^*} \beta(t_{i-1}) u_E^2(t_{i-1}) + \sum_{i=1}^{N_E^*} \beta(t_{i-1}) u_E(t_{i-1}) O(E^2) +$$
$$+ \sum_{i=1}^{N_E^*} u_E^2(t_{i-1}) O(E^2) + \sum_{i=1}^{N_E^*} u_E(t_{i-1}) O(E^3)$$

$$= \sum_{i=1}^{N_E^*} \beta(t_{i-1}) u_E^2(t_{i-1}) + O(E^2) + O(E^2) + O(E^3)$$
$$= \sum_{i=1}^{N_E^*} \beta(t_{i-1})(t_i - t_{i-1})^2 + O(E^2) = E^* + O(E^2).$$

Segue che anche E e E^* sono asintotici. Il problema è stato impostato correttamente: è dunque giustificato l'uso delle (9) e (10).

4 — Assenza di suddivisioni "strane"

Vediamo ora come, nel discreto, la suddivisione ottenuta con il metodo $\mathcal{A}$ della sezione 2 approssimi quella ottima. Ci resta da dimostrare che anche cercando tra le suddivisioni non rappresentabili da una funzione come la $u_E(t)$ e il metodo $\mathcal{A}$ è impossibile diminuire l'errore o il costo. Un esempio di suddivisione non rappresentabile e di finezza arbitrariamente piccola è una i cui intervalli sono ampi alternativamente δ e 2δ. Nella sezione 2 abbiamo infatti ipotizzato che $u(t)$, e quindi le ampiezze degli intervalli della suddivisione, fossero lentamente variabili nel tempo. Nel caso in questione, il numero di punti di suddivisione nell'intervallo $[t, t+\mathrm{d}t]$ non sarebbe piú stimato con sufficiente correttezza da $\mathrm{d}t/u(t)$.

Siamo dunque alle prese con un problema in tempo discreto la cui approssimazione continua, pur dotata di una soluzione "facile", sembra imporre una condizione troppo restrittiva su quest'ultima. Per dimostrare che la soluzione è asintoticamente ottima, ricorreremo ai metodi della programmazione dinamica.

Introduciamo, a tale scopo, la funzione valore $V(t, e)$, definita come il "numero minimo di passi per arrivare, da un errore e al tempo t, a un errore E al tempo finale b":

$$V(t, e) = \min_{u \in \mathcal{U}} \int_t^b \frac{\mathrm{d}s}{u(s)}, \tag{14}$$

dove

$$\mathcal{U} = \left\{ v \in \mathcal{C}([a, b], \mathbb{R}_{>0}) \,\middle|\, \int_t^b \beta(s) v(s) \mathrm{d}s + e \cdot \exp\left(\int_t^b a(s) \mathrm{d}s \right) = E \right\}$$

generalizzazione del problema (9)-(10) per un istante iniziale generico t e per un errore "iniziale" non nullo e. Chiaramente, fissato E considereremo ammissibili soltanto quei

valori di e per cui

$$e \exp\left(\int_t^b a(s)\mathrm{d}s\right) < E, \tag{15}$$

altrimenti $\mathcal{U} = \emptyset$, e la funzione valore perde significato.

2. Teorema — *Supponendo che $V(t,e)$ sia di classe $\mathcal{C}^1$, per ogni t nell'intervallo $[a,b]$ e per ogni valore ammissibile di e la funzione valore $V(t,e)$ soddisfa l'equazione di Hamilton-Jacobi-Bellman*

$$V_t(t,e) + \min_{u\in\mathcal{U}}\left[V_e(t,e)\cdot(a(t)e + \gamma(t)u(t)) + \frac{1}{u(t)}\right] = 0. \tag{16}$$

Dimostrazione: Applichiamo il *principio di Bellman*[4]. Supponiamo cioè che nell'intervallo $[t, t+\mathrm{d}t]$ si utilizzi una suddivisione $u \in \mathcal{U}$ non ottimale, giungendo a un errore $e + \mathrm{d}e$ all'istante $t+\mathrm{d}t$, usando in seguito nell'intervallo $[t+\mathrm{d}t, b]$ una funzione quasi ottimale per il sottoproblema di determinare $V(t+\mathrm{d}t, e+\mathrm{d}e)$. "Quasi ottimale" nel senso che, essendo $V(t,e)$ un estremo inferiore, non è detto che esista la funzione che realizza il minimo. Supponiamo di vederlo realizzato a meno di una costante positiva arbitrariamente piccola ε. Il costo complessivo nell'intervallo $[t,b]$ sarà stimabile in

$$\frac{\mathrm{d}t}{u(t)} + V(t+\mathrm{d}t, e+\mathrm{d}e) + \varepsilon, \tag{17}$$

dove $\mathrm{d}e$ è l'aumento dell'errore stimato a causa della suddivisione non ottimale. Il costo (17) è maggiore o uguale del costo minimo rappresentato dalla funzione valore V:

$$\frac{\mathrm{d}t}{u(t)} + V(t+\mathrm{d}t, e+\mathrm{d}e) + \varepsilon \geq V(t,e). \tag{18}$$

In base alle stime calcolate nella sezione 1 e precisamente per la (9) abbiamo:

$$\mathrm{d}e = [a(t)e + \gamma(t)u(t)]\mathrm{d}t.$$

Segue

$$V_t(t,e) + V_e(t,e)\cdot[a(t)e + \gamma(t)u(t)] + \frac{1}{u(t)} + \varepsilon \geq 0.$$

(4) "An optimal policy has the property that whatever the initial state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." [Bel, pag. 57]

Vista l'arbitarietà di ε, e siccome l'uso della funzione "quasi ottimale" (a meno di ε) in $[t,b]$ impone l'uguaglianza nella (18), abbiamo la (16). ■

La regolarità della funzione valore è assicurata dal fatto che ne conosciamo una formulazione esplicita:

3. Lemma — *La funzione valore (14) è di classe* $\mathcal{C}^1$.

Dimostrazione: Nel nostro caso, la funzione u è facilmente calcolabile come soluzione della generalizzazione del problema di minimo lagrangiano (9)-(10):

$$\text{minimizzare} \qquad \int_t^b \frac{\mathrm{d}s}{u(s)} \tag{19}$$

$$\text{soggetto al vincolo} \qquad \int_t^b \beta(s)u(s)\mathrm{d}t = E - e\exp\left(\int_t^b a(s)\mathrm{d}s\right) \tag{20}$$

con

$$\beta(s) = \frac{1}{2}\exp\left(\int_s^b a(\tau)\mathrm{d}\tau\right)\left|\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(s,\mathbf{x}(s))\right|.$$

La soluzione è

$$u_{E,t,e}(s) = \frac{E - e\exp\left(\int_t^b a(\tau)\mathrm{d}\tau\right)}{\sqrt{\beta(s)}\displaystyle\int_t^b \sqrt{\beta(\tau)}\mathrm{d}\tau},$$

il che ci permette di ottenere la formulazione esplicita di $V(t,e)$ desiderata:

$$V(t,e) = \int_t^b \frac{\mathrm{d}s}{u_{E,t,e}(s)} = \frac{\left(\displaystyle\int_t^b \sqrt{\beta(s)}\mathrm{d}s\right)^2}{E - e\exp\left(\displaystyle\int_t^b a(s)\mathrm{d}s\right)}.$$

Viste le ipotesi fatte sulle funzioni in gioco e la condizione (15), questa formulazione esplicita assicura la continuità e la derivabilità di $V(t,e)$. ■

Il prossimo teorema assicura che il costo di un passo, nel passaggio dal continuo al discreto, è valutato come prossimo a 1:

4. Teorema — *Per ogni* $\sigma > 0$ *arbitrariamente piccolo esiste* $\delta > 0$ *tale che per ogni suddivisione* $\{t_k\}_{k=0,\ldots,N}$ *di finezza minore di* δ *tale che* $e_N = E$ *si ha*

$$V(t_k,\varepsilon_k) < V(t_{k+1},e_{k+1}) + (1+\sigma).$$

Dimostrazione: Per assurdo: Supponiamo che per ogni $\sigma > 0$ esista una suddivisione $\{t_k\}_{k=0,\dots,N_\sigma}$ di finezza arbitrariamente piccola tale che

$$V(t_k, e_k) \geq V(t_{k+1}, e_{k+1}) + (1 + \sigma).$$

L'ipotesi ci permette di andare al limite (per assurdo esistono suddivisioni piccole a piacere che realizzano la disuguaglianza), ottenendo per $t_{k+1} - t_k$ abbastanza piccolo

$$V_t(t_k, e_k) + V_e(t_k, e_k)\,[a(t_k)e_k + \gamma(t_k)(t_{k+1} - t_k)] + \frac{1+\sigma}{t_{k+1} - t_k} \leq 0,$$

e si ha contraddizione con la (16) del lemma 2. ■

Infine:

5. Corollario — *Per suddivisioni abbastanza fini, la suddivisione calcolata con il metodo $\mathcal{A}$ è arbitrariamente vicina all'ottimo.*

Dimostrazione: Fissato σ, troviamo una suddivisione $\{t_k\}_{k=1,\dots,N}$ che realizza la tesi del teorema 4. Ragionando per induzione otteniamo

$$V(a, 0) < V(t_k, e_k) + k(1 + \sigma).$$

Infatti, applicando l'enunciato del teorema 4 per $k = 1$

$$V(a, 0) < V(t_1, e_1) + (i + \sigma)$$

e fatta l'ipotesi induttiva

$$V(a, 0) < V(t_k, e_k) + k(1 + \sigma)$$

otteniamo, applicando il teorema a $V(t_k, e_k)$:

$$V(a, 0) < [V(t_{k+1}, e_{k+1}) + (1 + \sigma)] + k(1 + \sigma) = V(t_{k+1}, e_{k+1}) + (k + 1)(1 + \sigma).$$

Infine, per $k = N$ si ha $V(a, 0) < V(t_N, e_N) + N(1 + \sigma)$. Ricordiamo che $V(a, 0)$ altro non è che il costo del calcolo con la soluzione fornita dal problema variazionale, che avevamo chiamato N_E, e che $V(t_N, e_N) = 0$. Si ottiene

$$N > \frac{N_E}{1 + \sigma}.$$

Per l'arbitrarietà di σ si conclude. ■

L'ottimalità asintotica del metodo descritto è dunque dimostrata.

5 — Esempi

A parziale esemplificazione di quanto visto nel presente capitolo, costruiamo e proviamo un programma FORTRAN che, data un'equazione differenziale ordinaria, scalare e del primo ordine, cerchi di trovare la suddivisione ottimale e la impieghi per trovare una buona approssimazione della soluzione.

Il programma dovrà dapprima trovare una soluzione approssimata del problema differenziale in esame, applicando un metodo poco dispendioso (Eulero a passo costante con una suddivisione molto grossolana), restituendo in un vettore la soluzione approssimata. I valori di questo vettore saranno poi utilizzati per calcolare la funzione $\beta(t)$. In base a quest'ultima serie di valori sarà deciso il passo di integrazione punto per punto.

5.1. *Descrizione del programma* `OTTIMO.F`

Ecco il testo del programma `OTTIMO.F`:

```
1. C*********************************************************************
2. C*                                                                   *
3. C*        OTTIMO.F  (MAURO BRUNATO, OTTOBRE 1994)                    *
4. C*                                                                   *
5. C*********************************************************************
6. C
7. C
8. C  ESEMPIO DI CALCOLO DELLA SUDDIVISIONE OTTIMALE SULLA FUNZIONE
9. C
10. C     X'   = F(T,X)
11. C     X(0) = A
12. C
13.       PROGRAM OTTIMO
14. C
15. C**********************************************************************
16. C
17. C  DEFINIZIONE DELLE VARIABILI: SI LAVORA IN DOPPIA PRECISIONE (64 BIT)
18. C
19.       IMPLICIT REAL*8    (A-H,O-Z)
20.       IMPLICIT INTEGER*4 (I-K,M-N)
21.       IMPLICIT LOGICAL   (L)
22. C
23. C**********************************************************************
24. C
25. C       ********************************************
26. C       * INIZIO DELLA PARTE DIPENDENTE DAL PROBLEMA *
27. C       ********************************************
28. C
29. C  DEFINIZIONE DELLE COSTANTI:
```

```
30. C
31. C     A E B SONO GLI ESTREMI DELL'INTERVALLO DI INTEGRAZIONE,
32. C     NAPPR E' IL NUMERO DI PASSI DELLA PRIMA APPROSSIMAZIONE,
33. C     XA E' IL VALORE INIZIALE DI X
34. C     XB E' IL VALORE FINALE ESATTO (SE E' CONOSCIUTO)
35. C
36.       PARAMETER(A    = 0.0D0,
37.      :          B    = 2.0D1,
38.      :          XA   = 0.0D0,
39.      :          XB   = 0.761594155090133285E+00,
40.      :          NAPPR= 100)
41. C
42. C  DEFINIZIONE DELLE FUNZIONI IMPLICITE:
43. C
44. C     FUNZIONE DEL PROBLEMA DIFFERENZIALE E SUE DERIVATE
45. C
46.       F(T,X)  = (1.0D0-X*X)*DEXP(-T)
47.       FT(T,X) = (X*X-1.0D0)*DEXP(-T)
48.       FX(T,X) = -2.0D0*X*DEXP(-T)
49. C
50. C        ****************************************
51. C        * FINE DELLA PARTE DIPENDENTE DAL PROBLEMA *
52. C        ****************************************
53. C
54. C***********************************************************************
55. C
56. C                ***************************
57. C                * INIZIO DELLA PARTE FISSA *
58. C                ***************************
59. C
60. C  SPAZIO DI LAVORO: ARRAY DELLA PRIMA APPROSSIMAZIONE
61. C
62.       DIMENSION XAPPR(0:NAPPR)
63. C
64. C  DEFINIZIONE DELLE FUNZIONI IMPLICITE:
65. C
66. C   - PASSO ALL'ISTANTE T
67. C
68.       U(E,T) = H*E*XAPPR((T-A)/PAPPR+1)
69. C
70. C   - NUMERO DI PASSI NECESSARI PER UN ERRORE PARI AD E
71. C
72.       PASSI(E) = 1/(E*H*H)
73. C
74. C***********************************************************************
75. C
76. C  PARTE ESEGUIBILE:
77. C
78. C                      ***********
79. C                      * PARTE 1 *
80. C                      ***********
81. C
82. C  CALCOLI PRELIMINARI:
```

```
C
C   - PRIMA APPROSSIMAZIONE DELLA SOLUZIONE NELL'ARRAY XAPPR
C
      XAPPR(0) = XA
      PAPPR = (B-A)/NAPPR
      DO 100 I=0,NAPPR-2
 100    XAPPR(I+2) = XAPPR(I+1)+PAPPR*F(A+I*PAPPR,XAPPR(I+1))
C
C   - CALCOLO (RETROGRADO) DELL'INTEGRALE AD ESPONENTE IN ESPO
C     E SUO UTILIZZO NEL CALCOLO DELLA FUNZIONE 1/DSQRT(BETA(T));
C     INSERIMENTO DI QUESTA NEL VETTORE XAPPR E CALCOLO DEL
C     COEFFICIENTE H
C
      ESPO = 0.0D0
      H = 0.0D0
      DO 110 I=NAPPR-1,0,-1
        T = A+I*PAPPR
        X = XAPPR(I+1)
        ESPO = ESPO+PAPPR*DABS(FX(T,X))
        AUX = DSQRT(DEXP(ESPO)*DABS(FT(T,X)+FX(T,X)*F(T,X))/2)
        H = H+PAPPR*AUX
 110    XAPPR(I+1) = 1/AUX
      H = 1/H
C
C   - INTRODUZIONE DELL'ERRORE DESIDERATO
C
      WRITE (*,*)'ERRORE DESIDERATO?'
      READ (*,*)ERRORE
C
C  CALCOLO DELLA SOLUZIONE CON EULERO A PASSO VARIABILE
C
      T = A
      N = 0
      SOL1 = XA
      LDENTRO = .TRUE.
      DO WHILE (LDENTRO)
        PASSO = U(ERRORE,T)
        IF (T+PASSO .GE. B) THEN
          PASSO = B-T
          LDENTRO = .FALSE.
        END IF
        SOL1 = SOL1 + PASSO*F(T,SOL1)
        T = T+PASSO
        N = N+1
      END DO
      ERROR1 = DABS(SOL1-XB)
C
C                     ***********
C                     * PARTE 2 *
C                     ***********
C
C  CALCOLO  DELLA SOLUZIONE CON EULERO A PASSO COSTANTE
C  E CON LO STESSO NUMERO DI PASSI
```

```
136. C
137.       PASSO = (B-A)/N
138.       SOL2 = XA
139.       DO 120 I=0,N-1
140.  120    SOL2 = SOL2 + PASSO*F(A+I*PASSO,SOL2)
141.       ERROR2 = DABS(SOL2-XB)
142. C
143. C                       ***********
144. C                       * PARTE 3 *
145. C                       ***********
146. C
147. C  STAMPA DEI RISULTATI
148. C
149.       WRITE (*,10)ERRORE,SOL1,ERROR1,SOL2,ERROR2,PASSI(ERRORE),N
150.  10   FORMAT(1X//
151.      + 1X,'ERRORE DESIDERATO: ',D12.5//
152.      + 1X,'RISULTATO CON PASSO VARIABILE: ',D20.12/
153.      + 1X,'ERRORE OTTENUTO:               ',D13.5//
154.      + 1X,'RISULTATO CON PASSO FISSO:     ',D20.12/
155.      + 1X,'ERRORE OTTENUTO:               ',D13.5//
156.      + 1X,'NUMERO PREVISTO DI PASSI:  ',D13.5/
157.      + 1X,'NUMERO EFFETTIVO DI PASSI: ',I10//)
158. C
159. C                   FINE DEL PROGRAMMA
160. C
161.       STOP
162. C
163.       END
164. C
165. C*****************************************************************
```

Come si vede dalle linee $19 \div 22$, il programma utilizza reali in doppia precisione (64 bit, tipo `REAL*8`, corrispondente al tipo `double` del C), mentre gli interi sono a 32 bit (tipo `INTEGER*4`, corrispondente al tipo `long int` del C). I dati relativi al problema da risolvere sono quelli descritti dalle linee $36 \div 48$, e consistono nella dichiarazione di alcune costanti e di alcune funzioni:

- `A` è l'estremo iniziale dell'intervallo di definizione;
- `B` è l'estremo finale;
- `XA` è il dato iniziale del problema;
- `XB` è il dato finale, all'istante `B`, se è noto: il suo valore serve per calcolare l'errore compiuto dal programma.

`NAPPR` è il numero di passi per l'approssimazione grezza iniziale. Il suo valore non dev'essere molto elevato, dato che il programma dichiarerà in seguito un vettore di reali contenente `NAPPR` elementi, e soprattutto l'approssimazione iniziale non deve incidere troppo sul costo complessivo del calcolo;

`F(T,X)` è la funzione $f(t,x)$ in doppia precisione;

`FX(T,X)` è la derivata parziale $f_x = \dfrac{\partial f}{\partial x}$;

`FT(T,X)` è la derivata rispetto al tempo.

In seguito inizia la parte relativa alle dichiarazioni comuni a tutti i problemi trattati:

62 All'inizio, viene definito l'array `XAPPR` contenente `NAPPR` elementi; in questo vettore verrà calcolata dapprima la soluzione approssimata, poi direttamente la funzione $\frac{1}{\sqrt{\beta(t)}}$ da questa desunta.

68 Viene poi definita la funzione implicita `U(E,T)`, che restituirà il valore approssimato di $u_E(t)$. Questa funzione utilizza il valore di `H`, corrispondente alla costante h definita alla fine della sezione 2.

72 La funzione implicita `PASSI(E)` restituisce il numero stimato di passi per la soluzione definitiva corrispondente a un determinato errore `E`.

Dopo le dichiarazioni, comincia la parte eseguibile.

$86 \div 89$ All'inizio della parte eseguibile il programma calcola immediatamente la soluzione approssimata in `NAPPR` passi, ponendola nel vettore `XAPPR`.

$96 \div 105$ In seguito, nella variabile `ESPO` viene calcolato il valore approssimato dell'integrale

$$\int_t^b |f_x(s, x(s))| \, \mathrm{d}s,$$

all'indietro, cioè a partire da $t = b$. Ad ogni passo, il valore corrente di `ESPO` è utilizzato per calcolare la funzione $\sqrt{\beta(t)}$, ponendo il valore in `XAPPR` stesso. Contemporaneamente, in `H` viene accumulato l'integrale di $\sqrt{\beta(t)}$. Una volta invertito alla linea 105, H contiene un'approssimazione del coefficiente h.

$109 \div 110$ Nella variabile `ERRORE` viene richiesto all'utente il valore desiderato di E.

$114 \div 127$ Inizia il calcolo vero e proprio, con il metodo di Eulero a passo variabile, dove il passo è determinato dalla funzione implicita `U(E,T)`. Quest'ultima resti-

tuisce il valore dell'array XAPPR corrispondente al nodo dell'approssimazione a passo costante immediatamente a sinistra dell'istante T. Ciò comporta che parecchi intervalli contigui hanno la stessa ampiezza, in particolare tutti quelli contenuti in uno stesso intervallo a passo fisso. Nella variabile N viene contato il numero di passi effettuati. Le linee $120 \div 123$ servono a controllare che l'ultima iterazione non oltrepassi il punto finale b. La soluzione finale viene posta in SOL1.

La parte principale del programma termina qui. Le istruzioni successive servono a confrontare il nostro procedimento con il metodo di Eulero a passo costante e con lo stesso numero di iterazioni.

128 In ERROR1 viene posto l'errore rispetto al valore della soluzione esatta all'istante b. Se il valore di XB non è stato correttamente definito alla linea 39, ad esempio se la soluzione vera non è nota, ERROR1 non ha nessun significato.

$137 \div 141$ Calcola in SOL2 l'approssimazione con il metodo di Eulero a passo costante con lo stesso numero di passi (precedentemente memorizzato in N).

$149 \div 157$ Infine, scrive i risultati ottenuti: l'errore desiderato dall'utente, il risultato del procedimento a passo variabile, l'errore commesso, il risultato del procedimento a pari costo con passo costante, l'errore commesso, il numero previsto di passi e il numero di passi effettivamente compiuti.

5.2. *Un'applicazione del programma* OTTIMO.F

Le linee dalla 36 alla 48 del listato precedente definiscono il seguente problema:

$$\begin{cases} \dot{x} = (1 - x^2)e^{-t} & t \in [0, 20] \\ x(0) = 0, \end{cases}$$

che ha per soluzione la funzione

$$x(t) = \frac{e^2 - e^{2e^{-t}}}{e^2 + e^{2e^{-t}}}.$$

Alle righe 36 e 37 sono stati definiti gli estremi dell'intervallo $[a, b]$, la riga 38 contiene il valore iniziale 0, mentre essendo nota la soluzione possiamo calcolarla per $t = 20$ e porne il valore in XB alla riga 39. Le righe 46, 47 e 48 contengono le definizioni delle funzioni:

$$f(t, x) = (1 - x^2)e^{-t}, \qquad f_t(t, x) = (x^2 - 1)e^{-t} \qquad \text{e} \qquad f_x(t, x) = -2xe^{-t}.$$

```
ERRORE DESIDERATO:  0.10000E+00

RISULTATO CON PASSO VARIABILE:   0.798218424438E+00
ERRORE OTTENUTO:                 0.36624E-01

RISULTATO CON PASSO FISSO:       0.919712584092E+00
ERRORE OTTENUTO:                 0.15812E+00

NUMERO PREVISTO DI PASSI:    0.29049E+02
NUMERO EFFETTIVO DI PASSI:         33
```

Tabella 4: Output di `OTTIMO.F` *($E = 10^{-1}$).*

```
ERRORE DESIDERATO:  0.10000E-01

RISULTATO CON PASSO VARIABILE:   0.765586562694E+00
ERRORE OTTENUTO:                 0.39924E-02

RISULTATO CON PASSO FISSO:       0.780130459369E+00
ERRORE OTTENUTO:                 0.18536E-01

NUMERO PREVISTO DI PASSI:    0.29049E+03
NUMERO EFFETTIVO DI PASSI:        295
```

Tabella 5: Output di `OTTIMO.F` *($E = 10^{-2}$).*

```
ERRORE DESIDERATO:  0.10000E-02

RISULTATO CON PASSO VARIABILE:   0.761998845811E+00
ERRORE OTTENUTO:                 0.40467E-03

RISULTATO CON PASSO FISSO:       0.763477378850E+00
ERRORE OTTENUTO:                 0.18832E-02

NUMERO PREVISTO DI PASSI:    0.29049E+04
NUMERO EFFETTIVO DI PASSI:       2910
```

Tabella 6: Output di `OTTIMO.F` *($E = 10^{-3}$).*

La nostra tecnica è particolarmente vantaggiosa sul problema appena illustrato. Infatti, l'esponenziale e^{-t} fa svanire rapidamente la funzione f, rendendo così vantaggioso l'infittimento dei nodi della suddivisione laddove la funzione è grande, all'inizio dell'intervallo di integrazione. Il procedimento a passo costante restituisce infatti un errore piú di tre volte maggiore. Le tabelle 4, 5 e 6 riportano gli output del programma per un errore pari a 10^{-1}, 10^{-2} e 10^{-3} rispettivamente.

5.3. *Limitazioni d'uso*

Non sempre il metodo descritto in questo capitolo risulta di qualche utilità: per molte equazioni che non presentano grosse variazioni dell'ordine di grandezza della funzione $f(t, x(t))$ il costo supplementare della valutazione delle funzioni ausiliarie $u_E(t)$, $\beta(t)$ e dei vari integrali, oltre alla prima approssimazione di $x(t)$, non vale il piccolo miglioramento nella precisione.

Un altro problema sorge quando $f_t(t,x) - f_x(t,x)f(t,x)$ è molto piccolo in prossimità di un punto della soluzione. In tal caso, l' approssimazione (2) dell'errore non vale piú, prevalendo il termine in $(\Delta t_i)^3$. In tal caso, la funzione $u_E(t)$ può diventare troppo grande in alcuni punti pregiudicando il funzionamento dell'algoritmo. Provando a sostituire nel programma `OTTIMO.F` i dati dell'equazione

$$\begin{cases} \dot{x} = (1+x)e^{-t} & t \in [0,1] \\ x(0) = 0, \end{cases}$$

la cui soluzione è $x(t) = e^{1-e^{-t}} - 1$, il programma restituisce un errore di overflow numerico. In questo caso, infatti,

$$|f_t(t,x(t)) + f_x(t,x(t))f(t,x(t))| = (1-e^{-t})e^{-t}e^{1-e^{-t}}$$

si annulla nel punto iniziale.

In generale dovremo imporre che esistano due costanti $c_1 > c_2 > 0$ tali che

$$\forall t \in [a,b] \qquad c_2 \leq \|\mathbf{f}_t(t,\mathbf{x}(t)) + \mathbf{f}_{\mathbf{x}}(t,\mathbf{x}(t)) \cdot \mathbf{f}(t,\mathbf{x}(t))\| \leq c_1.$$

Capitolo III
Estensioni

Introduciamo qui di seguito due possibili estensioni della tecnica esaminata al capitolo precedente.

La prima estensione riguarda il caso in cui la valutazione della funzione $\mathbf{f}(t, \mathbf{x})$ è molto costosa, con un costo variabile a seconda della precisione con cui la si vuole determinare. In tal caso, è possibile determinare *a posteriori* un controllo asintoticamente ottimale della precisione.

La seconda estensione riguarda i problemi del secondo ordine. La tecnica descritta in precedenza è applicabile, ma con una semplice modifica al metodo di Eulero è possibile ottenere in questo caso stime di errore molto piú vantaggiose.

1 — L'errore di troncamento

La discussione fatta al capitolo precedente riguarda il caso ideale, in cui la macchina esegue i calcoli con precisione infinita. Quest'ipotesi è ragionevole nel caso in cui $\mathbf{f}(t, \mathbf{x})$ sia calcolabile con le routines di libreria con buona precisione. Nel caso generale, $\mathbf{f}(t, \mathbf{x})$ può essere, ad esempio, un integrale molto costoso, che potrà essere calcolato a meno di un errore di troncamento che chiameremo $\varepsilon(t_n)$, ε essendo una funzione che ad ogni istante $t \in [a, b]$ associa l'errore col quale viene eseguito il calcolo in quel punto. Ovviamente, a precisioni diverse corrispondono costi di calcolo diversi. Supporremo di disporre di una funzione $c(\varepsilon)$ che descrive tale corrispondenza.

Per tener conto dell'arrotondamento, la (II.3) va modificata così:

$$|\mathbf{x}_n(t_{n+1}) - x_{n+1}| \approx \frac{1}{2} \left| \frac{\mathrm{d}}{\mathrm{d}t} \mathbf{f}(t_n, \mathbf{x}_n) \right| (\Delta t_{n+1})^2 + \varepsilon(t_n),$$

mentre la (II.4) diventa:

$$\mathrm{d}v(t) = \left(\frac{1}{2} \left| \frac{\mathrm{d}}{\mathrm{d}t} \mathbf{f}(t_n, \mathbf{x}_n) \right| u^2(t) + \varepsilon(t) \right) \frac{1}{u(t)} \mathrm{d}t.$$

Supponendo unitario il costo di un'iterazione e chiamando $c(\varepsilon)$ il costo di un calcolo di $\mathbf{f}(t, \mathbf{x}(t))$ con precisione ε, il costo totale dell'integrazione sarà approssimato dalla funzione

$$\int_a^b \frac{1 + c(\varepsilon(t))}{u(t)} \mathrm{d}t. \tag{1}$$

L'errore sarà

$$E(u, \varepsilon) = \int_a^b \alpha(t) \left[\gamma(t) u^2(t) + \varepsilon(t)\right] \frac{1}{u(t)} \mathrm{d}t, \tag{2}$$

dove

$$\alpha(t) = \exp\left(\int_t^b a(s)\mathrm{d}s\right) \qquad \text{e} \qquad \gamma(t) = \frac{1}{2}\left|\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t, \mathbf{x}(t))\right|.$$

Il problema di minimizzare l'integrale (1) imponendo l'errore E richiede di manipolare due controlli, u ed ε. La lagrangiana diventa (per motivi di opportunità chiamiamo λ^2 il moltiplicatore):

$$L(t, u, \varepsilon, \dot{u}, \dot{\varepsilon}) = \frac{1 + c(\varepsilon)}{u} + \lambda^2 \alpha (\gamma u^2 + \varepsilon)\frac{1}{u}.$$

Cerchiamo innanzitutto il controllo $\varepsilon(t)$ che minimizza il costo ad errore prefissato:

$$\frac{\partial L}{\partial \varepsilon} = 0$$

$$\frac{c'(\varepsilon)}{u} + \lambda^2 \alpha \frac{1}{u} = 0$$

$$c'(\varepsilon) = -\lambda^2 \alpha.$$

Per procedere occorre dunque ipotizzare che la derivata prima sia iniettiva e negativa; l'ipotesi è ragionevole, in quanto $c(\varepsilon)$ è presumibilmente funzione decrescente, positiva e dunque presumibilmente convessa di ε. Con queste ipotesi si ottiene:

$$\varepsilon(t) = c'^{-1}(-\lambda^2 \alpha(t)).$$

Cerchiamo $u(t)$:

$$\frac{\partial L}{\partial u} = -\frac{1 + c(\varepsilon)}{u^2} + \lambda^2 \alpha \left(\gamma - \frac{\varepsilon}{u^2}\right) = 0$$

$$u(t) = \sqrt{\frac{1 + c(\varepsilon(t)) + \lambda^2 \alpha(t) \varepsilon(t)}{\lambda^2 \alpha(t) \gamma(t)}}. \tag{3}$$

Infine, λ si ottiene sostituendo $\varepsilon(t)$ e $u(t)$ nella (2) e imponendo l'uguaglianza con E.

Osserviamo che la funzione $\varepsilon(t)$ non dipende in alcun modo dalla $u(t)$ scelta, ma solo indirettamente attraverso il parametro λ. Un altro risultato è che $\varepsilon(t)$ non dipende in alcun modo da $\gamma(t)$.

Un caso piú semplice a trattarsi è quello in cui l'errore di troncamento è costantemente uguale a ε. Naturalmente, il fatto di porre ε diverso da zero pone un limite inferiore all'accuratezza con cui possiamo calcolare il risultato pur accrescendo il lavoro. L'errore complessivo dovuto all'arrotondamento cresce al crescere del numero di suddivisioni.

Per tener conto dell'arrotondamento costante, la (II.3) va modificata così:

$$|\mathbf{x}_n(t_{n+1}) - x_{n+1}| \approx \frac{1}{2} \left| \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n) \right| (\Delta t_{n+1})^2 + \varepsilon,$$

mentre la (II.4) diventa:

$$\mathrm{d}v(t) = \left(\frac{1}{2} \left| \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n) \right| u^2(t) + \varepsilon \right) \frac{1}{u(t)} \mathrm{d}t.$$

La formula di propagazione non viene invece alterata (il costo di un passo è costante e viene quindi incorporato nel costo dell'iterazione). La stima finale dell'errore complessivo dovuto alla suddivisione imposta da $u(t)$ è il funzionale

$$E(u) \approx \int_a^b \alpha(t)[\gamma(t)u^2(t) + \varepsilon] \frac{1}{u(t)} \mathrm{d}t.$$

Minimizzare E vuol dire trovare quella funzione u tale che per ogni funzione v e per ogni λ reale $E(u + \lambda v) \geq E(u)$. In altri termini,

$$\forall v \qquad \frac{\partial}{\partial \lambda} E(u + \lambda v) \bigg|_{\lambda = 0} = 0.$$

Dopo gli opportuni calcoli si ottiene

$$\forall v \qquad \int_a^b \alpha(t) v(t) \left(\gamma(t) - \frac{\varepsilon}{u^2(t)} \right) \mathrm{d}t = 0,$$

da cui

$$u(t) = \sqrt{\frac{\varepsilon}{\gamma(t)}}, \tag{4}$$

corrispondente a un errore

$$E(u) = \int_a^b 2\alpha(t) \sqrt{\varepsilon \gamma(t)} \mathrm{d}t. \tag{5}$$

Vediamo ora come si determina u una volta imposto l'errore E. Il procedimento è lo stesso della sezione 2, ma porta a una lagrangiana piú complicata:

$$\begin{aligned} &\text{minimizzare} && \int_a^b \frac{\mathrm{d}t}{u(t)} \\ &\text{soggetto al vincolo} && E(u) = E. \end{aligned}$$

La lagrangiana è

$$L(t, u, \dot{u}) = \frac{1}{u} + \lambda\alpha(t)[\gamma(t)u^2 + \varepsilon]\frac{1}{u}.$$

L'equazione

$$\frac{\partial L}{\partial u} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{u}}$$

porta a determinare

$$u_E(t) = \sqrt{\frac{1 + \lambda\varepsilon\alpha(t)}{\lambda\alpha(t)\gamma(t)}}, \tag{6}$$

dove λ è il parametro tale che

$$\int_a^b (1 + 2\lambda\varepsilon\alpha(t))\sqrt{\frac{\alpha(t)\gamma(t)}{\lambda(1 + \lambda\varepsilon\alpha(t))}}\mathrm{d}t = E.$$

Si noti che le espressioni appena ricavate si riconducono per $\lambda \to \infty$ a quelle già ottenute ricercando l'errore minimo, la (4) e la (5). La (3), invece si riconduce all'espressione (6) per $\varepsilon(t)$ costante e costo del calcolo incorporato nel costo (unitario) dell'iterazione.

2 — Problemi del secondo ordine: un miglioramento

Si debba ora risolvere l'equazione del second'ordine in dimensione $d \geq 1$:

$$\begin{cases} \ddot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}}) \qquad t \in [a, b] \\ \mathbf{x}(a) = \mathbf{x}_0 \\ \dot{\mathbf{x}}(a) = \hat{\mathbf{x}}_0, \end{cases} \tag{7}$$

dove $\mathbf{x}_0, \hat{\mathbf{x}}_0 \in \mathbb{R}^d$, $\mathbf{x} \in \mathcal{C}^2([a, b], \mathbb{R}^d)$ e $\mathbf{f} \in \mathcal{C}^1([a, b] \times \mathbb{R}^d \times \mathbb{R}^d, \mathbb{R}^d)$.

Il problema può essere ricondotto al sistema del primo ordine in dimensione $2d$ ed essere risolto mediante il procedimento di Eulero studiato finora, calcolando le due successioni $\{\mathbf{x}_n\}_{n=0,\ldots,N}$ e $\{\hat{\mathbf{x}}_n\}_{n=0,\ldots,N}$, la prima approssimante la funzione $\mathbf{x}(t)$, l'altra la sua derivata:

$$\begin{cases} \mathbf{x}_{n+1} = \mathbf{x}_n + \hat{\mathbf{x}}_n \Delta t_{n+1} \\ \hat{\mathbf{x}}_{n+1} = \hat{\mathbf{x}}_n + \mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)\Delta t_{n+1}. \end{cases} \tag{8}$$

L'errore stimabile in un passo vale in questo caso

$$\frac{1}{2}\mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)(\Delta t_{n+1})^2 \tag{9}$$

per il calcolo di $\mathbf{x}_{n+1}$ e

$$\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)(\Delta t_{n+1})^2 \tag{10}$$

per il calcolo di $\hat{\mathbf{x}}_{n+1}$.

Possiamo però affinare il metodo. Studieremo l'approssimazione seguente e vedremo che garantisce un errore di un ordine di infinitesimo maggiore:

$$\begin{cases} \mathbf{x}_{n+1} = \mathbf{x}_n + \hat{\mathbf{x}}_n \Delta t_{n+1} + \dfrac{1}{2}\mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)(\Delta t)^2 \\ \hat{\mathbf{x}}_{n+1} = \hat{\mathbf{x}}_n + \dfrac{\mathbf{f}(t_{n+1}, \mathbf{x}_{n+1}, \hat{\mathbf{x}}_n + \mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)) + \mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)}{2}\Delta t_{n+1}. \end{cases} \tag{11}$$

L'approssimazione della prima riga sfrutta la conoscenza che noi abbiamo della derivata seconda di $\mathbf{x}(t)$. L'errore di discretizzazione in un singolo passo diventa dunque

$$|\mathbf{x}_n(t_{n+1}) - \mathbf{x}_{n+1}| \approx \frac{1}{6}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)(\Delta t_{n+1})^3. \tag{12}$$

Per giustificare la seconda riga della (11), modifichiamo opportunamente le definizioni date all'inizio del capitolo II: sia $\mathbf{x}_n(t)$ la soluzione del problema parziale

$$\begin{cases} \ddot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}, \dot{\mathbf{x}}) \qquad t \in [t_n, t_{n+1}] \\ \mathbf{x}(t_n) = \mathbf{x}_n \\ \dot{\mathbf{x}}(t_n) = \hat{\mathbf{x}}_n. \end{cases}$$

L'approssimazione data per la $\hat{\mathbf{x}}_{n+1}$ nella (11) differisce da quella della (8) per il fatto che si è calcolata una media della $\mathbf{f}$ in due punti diversi, quello iniziale e uno molto prossimo a quello finale (dove il punto $\hat{\mathbf{x}}_{n+1}$ è stato approssimato al primo ordine come nella (8)). Si ha infatti:

$$\dot{\mathbf{x}}_n(t_{n+1}) = \hat{\mathbf{x}}_n + \mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)\Delta t_{n+1} + \frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)(\Delta t_{n+1})^2 + \delta(\Delta t_{n+1}), \tag{13}$$

dove

$$\delta(\Delta t_{n+1}) \approx \frac{1}{6}\frac{\mathrm{d}^2}{\mathrm{d}t^2}\mathbf{f}(\xi, \mathbf{x}_n(\xi), \hat{\mathbf{x}}_n(\xi))(\Delta t_{n+1})^3,$$

dove ξ è qualunque punto nell'intervallo $[t_n, t_{n+1}]$ (noi useremo t_n).

La derivata $\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)$ è approssimata dal rapporto incrementale di $\mathbf{f}$ calcolato tra i punti $(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n)$ e $(t_{n+1}, \mathbf{x}_{n+1}, \hat{\mathbf{x}}_n + \mathbf{f}(t_n, \mathbf{x}_n, \hat{\mathbf{x}}_n))$. Quest'ultimo approssima il vero punto finale $(t_{n+1}, \mathbf{x}_n(t_{n+1}), \dot{\mathbf{x}}_n(t_{n+1}))$ a meno della quantità (12) nella seconda componente e a meno della quantità (10) nella terza.

In definitiva, il nostro rapporto incrementale approssima la vera derivata di $\mathbf{f}$ della (13) a meno di

$$\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f} \cdot \mathbf{f}_{\dot{\mathbf{x}}} \Delta t_{n+1}.$$

L'errore totale di discretizzazione nel calcolo di $\hat{\mathbf{x}}_{n+1}$ è dunque

$$|\dot{\mathbf{x}}_n(t_{n+1}) - \hat{\mathbf{x}}_{n+1}| \approx \frac{1}{2}\left|\mathbf{f}_{\dot{\mathbf{x}}} \cdot \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f} + \frac{1}{3}\frac{\mathrm{d}^2}{\mathrm{d}t^2}\mathbf{f}\right| (\Delta t_{n+1})^3.$$

Per studiare la propagazione dell'errore, chiamiamo $\mathbf{v}_n$ una piccola alterazione del dato $\mathbf{x}_n$, e così $\hat{\mathbf{v}}_n$ sia una alterazione di $\hat{\mathbf{x}}_n$. Siano i loro moduli v_n e $\hat{v}_n$. Dalle (11) otteniamo:

$$\begin{cases} \mathbf{v}_{n+1} \approx \mathbf{v}_n + \hat{\mathbf{v}}_n \Delta t_{n+1} + \ldots (\Delta t_{n+1})^2 \\ \hat{\mathbf{v}}_{n+1} \approx \hat{\mathbf{v}}_n + \dfrac{\mathbf{f}_{\mathbf{x}}\mathbf{v}_{n+1} + \mathbf{f}_{\dot{\mathbf{x}}}(\hat{\mathbf{v}}_n + \mathbf{f}_{\mathbf{x}}\mathbf{v}_n + \mathbf{f}_{\dot{\mathbf{x}}}\hat{\mathbf{v}}_n) + \mathbf{f}_{\mathbf{x}}\mathbf{v}_n + \mathbf{f}_{\dot{\mathbf{x}}}\hat{\mathbf{v}}_n}{2} \Delta t_{n+1}, \end{cases}$$

da cui, passando al continuo con le funzioni $v(t)$ e $\hat{v}(t)$, che per suddivisioni abbastanza fitte approssimano i moduli di $\mathbf{v}_n$ e di $\hat{\mathbf{v}}_n$, otteniamo il sistema differenziale:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} v \\ \hat{v} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \left|\mathbf{f}_{\mathbf{x}} + \frac{\mathbf{f}_{\mathbf{x}}\mathbf{f}_{\dot{\mathbf{x}}}}{2}\right| & \left|\mathbf{f}_{\dot{\mathbf{x}}} + \frac{\mathbf{f}_{\dot{\mathbf{x}}}^2}{2}\right| \end{pmatrix} \begin{pmatrix} v \\ \hat{v} \end{pmatrix}.$$

A questo punto si procede come visto nel capitolo precedente. Chiamiamo

$$a_1(t) = \left|\mathbf{f}_{\mathbf{x}} + \frac{\mathbf{f}_{\mathbf{x}}\mathbf{f}_{\dot{\mathbf{x}}}}{2}\right| (t, \mathbf{x}(t), \dot{\mathbf{x}}(t)),$$

$$a_2(t) = \left|\mathbf{f}_{\dot{\mathbf{x}}} + \frac{\mathbf{f}_{\dot{\mathbf{x}}}^2}{2}\right| (t, \mathbf{x}(t), \dot{\mathbf{x}}(t)).$$

Possiamo calcolare la funzione $a(t) = \|A(t)\|_\rho$:

$$a(t) = \left(a_2(t) + \sqrt{a_2^2(t) + 4a_1(t)}\right).$$

Sia poi per brevità definita la funzione

$$\beta(t) = \frac{1}{2}\exp\left(\int_t^b a(s)\mathrm{d}s\right) \cdot \left|\mathbf{f}_{\dot{\mathbf{x}}} \cdot \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{f} + \frac{1}{3}\frac{\mathrm{d}^2}{\mathrm{d}t^2}\mathbf{f}\right| (t, \mathbf{x}(t), \dot{\mathbf{x}}(t)).$$

La stima dell'errore finale è:

$$|\mathbf{x}(b) - \mathbf{x}_N| \approx \int_a^b \beta(t) u^2(t) \mathrm{d}t.$$

Risulta il problema variazionale

$$\text{minimizzare} \quad \int_a^b \frac{\mathrm{d}t}{u(t)}$$

$$\text{soggetto al vincolo} \quad \int_a^b \beta(t) u^2(t) \mathrm{d}t = E.$$

La soluzione è un pò piú complessa di prima:

$$u(t) = \frac{h}{\beta^{\frac{1}{3}}(t)},$$

con

$$h = 4^{-\frac{1}{3}} \left(E \cdot \int_a^b \beta^{\frac{1}{3}}(t) \mathrm{d}t \right)^{-\frac{1}{2}}.$$

Conclusioni

Il metodo discusso al capitolo II potrebbe essere esteso in altre direzioni, in particolar modo verso l'ottenimento di maggiorazioni piuttosto che di stime d'errore. Questo in vista dell'applicazione di metodi di dimostrazione numerica, come quello sviluppato al primo capitolo. La necessità di maggiorazioni "ottimiste" e poco costose è discussa nell'introduzione di [Ked].

La dimostrazione di ottimalità asintotica dovrebbe essere applicabile senza difficoltà a tutte le modifiche del metodo presentate nell'ultimo capitolo; dovrebbe servire altresì come nuovo esempio applicativo della Programmazione Dinamica e come spunto — ci auguriamo — per future tecniche di dimostrazione.

Bibliografia

[BaC] M. Bardi e I. Capuzzo Dolcetta, *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman Equations*, Birkhäuser, Boston (in preparazione) .

[Bar] M. Bardi, *Viscosity Solutions of Isaac's Equations and Existence of a Value (*in *Lectures on Games)*, G. Ricci & C. Torricelli editors, Springer 1991.

[Bel] R. Bellman, *Adaptive Control Processes: A Guided Tour*, Princeton University Press 1961.

[Bre] C. Brézinski, *Résolution des systèmes d'équations linéaires*, Publications du Laboratoire de Calcul de l'USTL, Lille 1992.

[Bur] R. L. Burden e J. D. Faires, *Numerical Analysis — Fourth Edition*, PWS-KENT Publishing Company, Boston 1988.

[Gea] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall Series in Automatic Computation, Englewood Cliffs (NJ) 1971.

[Kan] L. V. Kantorovič e G. P. Akilov, *Functional Analysis in Normed Spaces*, Pergamon Press, New York 1964.

[Ked] G. Kedem, *A Posteriori Error Bounds for Two–Point Boundary Value Problems*, Siam J. Numer. Anal. **18** 1981, pp. 431–448 .

[Mor] M. Morandi Cecchi, *Introduzione al Calcolo Numerico*, Coop. Alfasessanta, Padova 1989.

[Nak] M. T. Nakao, *A Numerical Verification Method for the Existence of Weak Solutions for Nonlinear Boundary Value Problems*, Journal of Mathematical Analysis and Applications **164** 1992, pp. 489–507 .