Calcolo Numerico

Sandro Tosi

Indice

Introduzione

In questa relazione vengono presentate le implementazioni in MATLAB degli algoritmi approfonditi durante il corso di calcolo numerico.

Ogni algoritmo viene presentato con una introduzione teorica che descrive il problema che andremo ad affrontare e che ci guiderà alla stesura del codice di una function Matlab corrispondente. Vengono inoltre forniti alcuni esempi di utilizzo per meglio comprendere il funzionamento del codice.

Ogni listato contiene all'inizio un commento che viene visualizzato dall'help in linea di Matlab e quindi descrive l'algoritmo all'utente.

Capitolo 1

Analisi degli errori

In questo capitolo esamineremo come un calcolatore memorizza i numeri, e come questo può influire sul calcolo e sulla rappresentazione dei numeri.

1.1 Rappresentazione dei numeri nel calcolatori

Ogni calcolatore ha a disposizione una memoria finita, incapace di contenere l'infinito insieme dei numeri reali. L'insieme $\mathbb{F} \subset \mathbb{R}$ è quello che si chiama insieme dei numeri macchina, l'insieme di quei numeri rappresentabili sul calcolatore.

Dato un numero $x \in \mathbb{R}$ perché sia possibile rappresentarlo sulla macchina in generale è necessario che questo venga approssimato con un valore $fl(x) \in \mathbb{F}$ sufficientemente vicino ad x.

Non è poi certo che, dati due numeri $x_1, x_2 \in \mathbb{F}$, applicando ad essi una operazione $\circ = (+, -, *, /)$, $x_1 \circ x_2 \in \mathbb{F}$. Per poter stabilire l'affidabilità del risultato è perciò necessario determinare:

- l'origine degli errori;
- la loro propagazione;
- una valutazione degli errori.

Quando un numero $x \in \mathbb{R}$ viene approssimato con $\hat{x} \in \mathbb{F}$ si definiscono due errori:

- l'errore assoluto, $|x - \hat{x}|$ e
- l'errore relativo, $\frac{|x-\hat{x}|}{|x|}$ $x \neq 0$, solitamente espresso come percentuale.

1.1.1 I numeri interi

Prendiamo come esempio i numeri interi. In questo caso l'errore di arrotondamento non si presenta, infatti, considerando un numero a caso

$$1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

o comunque più in generale, un numero in base 10 può essere visto come

$$d_n d_{n-1} \cdots d_0 = d_n \cdot 10^n + d_{n-1} \cdot 10^{n-1} \cdots + d_0 \cdot 10^0$$

poichè noi siamo soliti utilizzare il sistema decimale per esprimere i numeri ma se, sempre per generalità, chiamiamo $\beta > 1 \in \mathbb{N}$ la base generica, possiamo scrivere

$$d_n d_{n-1} \cdots d_0 = d_n \cdot \beta^n + d_{n-1} \cdot \beta^{n-1} \cdots + d_0 \cdot \beta^0$$

abbiamo così espresso la notazione posizionale in modo generico, posizionale perché la posizione del numero indica per quale potenza della base β debba essere moltiplicato.

Data la finitezza del calcolatore risulta necessario porre un limite superiore, $nmax$, ai numeri naturali rappresentabili o meglio ancora un intervallo di valori $[0, nmax]$. Per riportarci nel caso dei numeri interi, avremo un intervallo del tipo $[zmin, zmax]$. Se consideriamo, per esempio, di avere $\beta = 2$ e che la nostra rappresentazione dei numeri sia quella in complemento a due, allora l'intervallo dei valori assume questa forma

$$[-2^k, 2^{k-1} - 1]$$

e questo intervallo di valori viene rappresentato in modo esatto, senza introdurre alcun errore.

1.1.2 Aritmetica intera

Guardiamo ora alle operazioni di base per i numeri interi in un calcolatore. Siano allora $z_1, z_2 \in [zmin, zmax] = I$, allora $z_1 + z_2$ deve appartenere all'intervallo I, altrimenti significa che si è verificato un errore di overflow; ma se il valore appartiene a questo intervallo, il risultato sarà esatto e naturalmente questo vale anche per l'operazione di sottrazione e moltiplicazione.

1.1.3 I numeri reali

La rappresentazione che solitamente si usa sui calcolatori per i numeri reali è quella floating point, rappresentazione basata sul seguente

Teorema. *Sia $\beta > 1$, $\beta \in \mathbb{N}$ base del nostro sistema posizionale. Un qualunque $x \in \mathbb{R}$ $(x \neq 0)$ si può esprimere in modo univoco come*

$$x = sign(x) \underbrace{(d_1\beta^{-1} + d_2\beta^{-2} + \cdots)}_{mantissa} \beta^p$$

dove

1. $sign(x) = \begin{cases} 1 & x > 0 \\ -1 & x < 0 \end{cases}$

2. $p \in \mathbb{Z}$ *è detta caratteristica*

3. $0 \leq d_i \leq \beta - 1$, $d_1 \neq 0$.

Quella descritta nel teorema prende il nome di forma floating point normalizzata, in quanto viene richiesto $d_1 \neq 0$; il fattore β^p è detto parte esponenziale e le cifre d_i della mantissa sono le cifre significative.

Per poter definire i numeri che apparterranno all'insieme $\mathbb{F} \subset \mathbb{R}$ sarà necessario porre un intervallo per la caratteristica, $p \in [L, U]$ e solitamente $L < 0$ e $U > 0$; inoltre non è possibile rappresentare le infinite cifre significative dei numeri reali, si fissa allora t come massimo numero di cifre significative per un numero che appartiene all'insieme $\mathbb{F}$. Possiamo finalmente definire $\mathbb{F}$, l'insieme dei numeri macchina, come

$$\mathbb{F}(\beta, t, L, U) = \{0\} \cup \{x \in \mathbb{R} : x = sign(x)\beta^p \sum_{i=1}^{t} d_i\beta^{-i},$$

$$p \in [L, U](\cap\mathbb{Z}), \quad 0 \leq d_i \leq \beta - 1, \quad d_1 \neq 0, \}$$

1.1.4 Gli standard IEEE floating point

La IEEE è un'associazione ormai diventata un'autorità nel campo della tecnica, e quindi anche nel modo dell'informatica, che ha proposto due modi di rappresentazione dei numeri in formato floating point, in seguito divenuti due standard, i seguenti:

- singola precisione: $\beta = 2$, $t = 23$, $[L, U] = [-128, 127]$;

- doppia precisione: $\beta = 2$, $t = 52$, $[L, U] = [-1024, 1023]$

Il singola precisione utilizza 32 bit (4 byte) per rappresentare un numero, mentre la doppia precisione, come ci dice già il suo nome, utilizza il doppio dello spazio e cioè 64 bit (8 byte) per un numero floating point.

1.1.5 Le proprietà dell'insieme $\mathbb{F}$

Approfondiamo adesso alcune caratteristiche dell'insieme $\mathbb{F}$:

1. La cardinalità di $\mathbb{F}$ è minore di $+\infty$, di più
$$|\mathbb{F}| = 2(U - L + 1)(\beta - 1)\beta^{t-1}$$
vediamone le ragioni:

2	per il segno;
$(U - L + 1)$	sono i valori che puo' assumere p;
$(\beta - 1)\beta^{t-1}$	guardiamo quanti sono i $\sum_{i=1}^{t} d_i\beta^{-i}$: d_1 può assumere $\beta - 1$ valori: $(d_1 = 1, 2, \cdots \beta - 1)$ $d_i (i \neq 1)$ può assumere β valori $(d_i = 0, 1, \cdots \beta - 1)$ e di elementi d_i ce ne sono $t - 1$ allora si hanno β^{t-1} valori possibili. Riunendo si ottiene $\underbrace{(\beta - 1)}_{d_1}\underbrace{\beta^{t-1}}_{d_i}$

2. Se $x \in \mathbb{F}$, $x > 0$, allora $x \in [xmin, xmax]$ con $xmin = \beta^{L-1}$ e $xmax = \beta^U(1 - \beta^{-t})$:

   - xmin, $p = L$ $d_1 = 1$ per imposizione e $d_2 = \cdots = d_n = 0$, i valori minimi che possono assumere; si ottiene perciò $xmin = \beta^L \cdot \beta^{-1} = \beta^{L-1}$

   - xmax, $p = U$ e $d_1 = \cdots = d_n = \beta - 1$ i loro valori massimi, si ha quindi
$$\begin{aligned} xmax &= \beta^U \sum_{i=1}^{t} (\beta - 1)\beta^{-i} = \beta^U \left(\sum_{i=1}^{t} \beta^{1-i} - \sum_{i=1}^{t} \beta^{-i} \right) = \\ &= \beta^U \left(\beta^0 + \sum_{j=1}^{t-1} \beta^{-j} - \sum_{i=1}^{t} \beta^{-i} \right) = \\ &= \beta^U \left(1 + \sum_{j=1}^{t-1} \beta^{-j} - \sum_{i=1}^{t-1} \beta^{-i} - \beta^t \right) = \beta^U(1 - \beta^t) \end{aligned}$$
   Quello che abbiamo mostrato era basato sul fatto che $x > 0$, nel caso in cui $x < 0$ allora $x \in [-xmax, -xmin]$, perciò $\mathbb{F} \subset [-xmax, -xmin] \cup \{0\} \cup [xmin, xmax]$

3. I valori dell'insieme $\mathbb{F}$ non sono equidistanti tra loro, lo sono solo tra potenze successive.

1.2 Approssimazione

Dato un numero $x \in \mathbb{R}$ potrebbe avvenire che $x \notin \mathbb{F}$: in questo caso si dovrà trovare un'approssimazione di x in modo che possa essere rappresentato sul nostro calcolatore; si cerca quindi $fl(x) \in \mathbb{F} : fl(x) \approx x$. Esaminiamo le varie possibilità che ci si propongono:

1. $p \in [L, U] \quad d_i = 0 \; \forall i > t$; allora $x \in \mathbb{F}$ e $fl(x) = x$; viene quindi rappresentato in modo esatto sulla nostra macchina;

2. $p \notin [L, U]$, questo implica $x \notin \mathbb{F}$

   (a) $p < L$, si ottiene quello che si chiama underflow, poichè x viene a trovarsi nell'intervallo $]-xmin, xmin[-\{0\}$; in questo caso l'approssimazione esiste ed è $fl(x) = 0$;

   (b) $p > U$, si verifica un'errore di overflow, il numero risulta essere oltre i limiti della macchina, $x < -xmax$, $x > xmax$; il calcolo si arresta perché non esiste alcuna approssimazione atta a rappresentare il numero x.

3. $p \in [L, U]$, ma $\exists i > t : \; d_i \neq 0$. Ci troviamo nel caso in cui il numero di cifre significative di x è superiore al limite della macchina: ci si presentano allora due possibilità:

   (a) il troncamento;

   (b) l'arrotondamento.

1.2.1 Troncamento ed arrotondamento

Proviamo con un esempio a spiegare il funzionamento dei due metodi: consideriamo $\mathbb{F}(10, 3, -1, 1)$ ed $x > 0$, sia $x = 0.3426 \cdot 10^0$; naturalmente questo numero non può essere rappresentato in modo esatto con un elemento dell'insieme $\mathbb{F}$ (il numero di cifre significative è superiore al nostro limite); vediamo come si comportano allora troncamento ed arrotondamento:

troncamento: vengono eliminate le cifre in eccesso, ottenendo per il nostro esempio $fl(x) = 0.342 \cdot 10^0$;

arrotondamento: si considera la (t+1)-esima cifra e la si confronta con $\frac{\beta}{2}$: se questa cifre risulta maggiore di quella quantita si pone $d_t = d_t + 1$ altrimenti non si compie nessuna operazione; in ogni caso le

cifre in eccesso, dopo il passo precedente, vengono troncate. In pratica calcoliamo

$$y = x + \frac{\beta}{2} \cdot 10^{-(t+1)}$$

$$(0.3426 \cdot 10^0 + 0.0005 \cdot 10^0 = 0.3431 \cdot 10^0)$$

e successivamente se ne esegue il troncamento ($fl(x) = 0.343 \cdot 10^0$). Facciamo notare che la somma precedente può dal luogo ad overflow.

Riassumendo:

troncamento:

$$fl(x) = tronc(x) = \beta^p \sum_{i=1}^{t} d_i \beta^{-i}$$

arrotondamento:

$$fl(x) = arr(x) = \beta^p tronc\left(\sum_{i=1}^{t+1} d_i \beta^{-i} \pm \frac{\beta}{2}\beta^{t+1}\right)$$

Il significato del $\pm$ è che, nel caso $x > 0$ deve essere utilizzato il "+", mentre per $x < 0$ si deve utilizzare "$-$".

Proposizione. *Sia:*

$$x = \left(\sum_{i=1}^{+\infty} d_i \beta^{-i}\right) \beta^p \quad d_1 \neq 0,\ x \neq 0$$

allora, se non si verifica overflow, si ha

$$\left|\frac{fl(x) - x}{x}\right| < k\beta^{(1-t)} \quad con\ k = \begin{cases} 1 & se\ fl(x) = tronc(x) \\ \frac{1}{2} & se\ fl(x) = arr(x) \end{cases}$$

Il numero $k\beta^{(1-t)} = eps$ è detto precisione di macchina, ed è il più piccolo numero positivo tale che:

$$fl(1 + eps) > 1$$

$$fl(1 + \varepsilon) = 1, \quad 0 < \varepsilon < eps$$

infatti si verifica facilmente che, nel caso si utilizzi come metodo di approssimazione l'arrotondamento, si ha

$$1 + eps = 1 + \frac{1}{2}\beta^{1-t} = \beta\left(\beta^{-1} + \frac{\beta}{2}\beta^{-(t+1)}\right) \quad \{p = 1\}$$

$$\begin{aligned} arr(1+eps) &= \beta tronc\left(\beta^{-1} + \frac{\beta}{2}\beta^{-(t+1)} + \frac{\beta}{2}\beta^{-(t+1)}\right) = \\ &= \beta tronc \underbrace{\left(\beta^{-1} + \beta^{-t}\right)}_{\text{ha t cifre significative}} = \beta\left(\beta^{-1} + \beta^{-t}\right) > 1 \end{aligned}$$

Per verificare $fl(1+\varepsilon) = 1$ scegliendo $\varepsilon < \frac{\beta}{2}\beta^{-(t+1)}$ si otterrà, alla fine di una sequenza di espressioni simili a quelle sopra, un qualcosa di minore di β^{-t} che verrà scartato dall'operazione di troncamento.

1.2.2 Aritmetica floating point

Sull'insieme dei numeri macchina si definisce l'aritmetica di macchina che differisce dall'aritmetica esatta, vediamo come:

$$\begin{aligned} x \oplus y &= fl(x+y) = (x+y)(1+\varepsilon) \\ x \ominus y &= fl(x-y) = (x-y)(1+\varepsilon) \\ x \otimes y &= fl(x \times y) = (x \times y)(1+\varepsilon) \\ x \oslash y &= fl(x/y) = (x/y)(1+\varepsilon) \end{aligned}$$

Quanto scritto sopra è derivato dal fatto che

$$\left|\frac{fl(x) - x}{x}\right| \leq eps$$

da cui si ricava che $fl(x) = x(1+\varepsilon)$: infatti possiamo scrivere la precedente relazione come $|fl(x) - x| \leq |x| \cdot eps$ e da qui, prendendo $|\varepsilon| \leq eps$ si ottiene $|fl(x) - x| = |x||\varepsilon|$.

Il significato della precedente scrittura è che la singola operazione macchina commette al più un errore dell'ordine di *eps*: se $x, y, x \in \mathbb{F}$

$$x \oplus y \oplus z \neq (x+y+z)(1+\varepsilon) \quad |\varepsilon| \leq eps$$

1.3 Condizionamento del problema

Sia $y : \mathbb{R} \to \mathbb{R}$, $y = \varphi(x)$ una funzione con x soggetto ad errori casuali (diciamo che sia il risultato di prove sperimentali), ed ipotiziamo che φ venga calcolata in modo esatto.

Quello che ci proponiamo di fare adesso è, invece di calcolare $\varphi(x)$, è calcolare $\varphi(\hat{x})$ con $\hat{x}$ una perturbazione di x ma comunque un valore prossimo ad x, e poi osservare la relazione tra l'errore in partenza, tra x e $\hat{x}$, e l'errore in arrivo, tra y e $\hat{y}$. Definiamo allora:

εy: errore (relativo o assoluto) della soluzione y;

εx: errore (relativo o assoluto) della soluzione x.

Allora possiamo riunire questi errori in una relazione

$$|\varepsilon y| \leq k|\varepsilon x|$$

con k che viene detto numero di condizionamento del problema.

Se $k \approx 1$ il problema si dice ben condizionato, infatti si ha lo stesso ordine di grandezza per gli errori in partenza ed in arrivo; se invece $k \gg 1$ il problema si dice mal condizionato in quanto piccoli errori sui dati in partenza possono influenzare notevolmente il risultato finale.

1.3.1 Condizionamento delle operazioni fondamentali

Vogliamo adesso guardare come sono influenzate le operazioni fondamentali dell'aritmetica se applicate a dati perturbati, per fare questo abbiamo bisogno di alcune ipotesi preliminari: siano $x, y \in \mathbb{R}$ e siano $\hat{x} = x(1 + \varepsilon x)$, $\hat{y} = y(1 + \varepsilon y)$ i valori perturbati ed inoltre sia $\varepsilon = max\{|\varepsilon x|, |\varepsilon y|\}$.

1. La somma e la sottrazione.

$$z = x + y \quad \hat{z} = \hat{x} + \hat{y} = x(1 + \varepsilon x) + y(1 + \varepsilon y)$$

$$\hat{z} = \underbrace{x + y}_{z} + x\varepsilon x + y\varepsilon y$$

$$|\hat{z} - z| = |x\varepsilon x + y\varepsilon y|$$

per la disuguaglianza triangolare

$$|\hat{z} - z| \leq |x||\varepsilon x| + |y||\varepsilon y|$$

$$|\hat{z} - z| \leq (|x| + |y|)\varepsilon$$

possiamo allora porre

$$|\varepsilon z| = \frac{|\hat{z} - z|}{|z|} \leq \frac{|x| + |y|}{|x + y|}\varepsilon$$

per la somma abbiamo quindi un valore del numero di condizionamento pari a:

$$k = \frac{|x| + |y|}{|x + y|}$$

perciò il problema risulta mal condizionato quanto $|x + y| \approx 0$. Tutto quanto è stato detto per la somma vale in modo equivalente per la sottrazione, infatti non abbiamo posto alcun vincolo sul segno dei numeri x, y, perciò $x - y \equiv x + (-y)$.

2. Il prodotto.

$$z = xy \quad \hat{z} = \hat{x}\hat{y} = x(1+\varepsilon x)y(1+\varepsilon y)$$

$$\hat{z} = \underbrace{xy}_{z}(1+\varepsilon x+\varepsilon y+\varepsilon x\varepsilon y) \approx z(1+\varepsilon x+\varepsilon y)$$

$$\hat{z} = z + z(\varepsilon x+\varepsilon y)$$

$$|\hat{z}-z| = |z||\varepsilon x+\varepsilon y| \leq |z|(|\varepsilon x|+|\varepsilon y|)$$

e nuovamente possiamo ottenere

$$|\varepsilon z| = \frac{|\hat{z}-z|}{|z|} \leq 2\varepsilon$$

risulta perciò un problema sempre ben condizionato.

3. La divisione.

$$z = \frac{x}{y} \quad \hat{z} = \frac{\hat{x}}{\hat{y}} = \frac{x(1+\varepsilon x)}{y(1+\varepsilon y)} = \underbrace{\frac{x}{y}}_{z} \cdot \frac{(1+\varepsilon x)}{(1+\varepsilon y)}$$

ricordandosi adesso che la somma della serie geometrica è

$$\frac{1}{1+\varepsilon y} \approx 1-\varepsilon y(+\varepsilon y^2 - \cdots)$$

si ottiene

$$\hat{z} \approx z(1+\varepsilon x)(1-\varepsilon y)$$

$$|\varepsilon z| = \frac{|\hat{z}-z|}{|z|} \leq |\ |\varepsilon x|-|\varepsilon y|\ | \leq 2\varepsilon$$

e risulta ancora un problema sempre ben condizionato.

Quello che risulta conveniente fare è prima compiere le operazioni mal condizionate, per eseguire successivamente quelle ben condizionate; infatti, guardiamo il risultato di questo semplice esempio per vedere come questo può avere ripercussioni notevoli: siano $a = 0.23371258\times10^{-4}$, $b = 0.33678429\times 10^2$ e $c = -0.3367781\times10^2$ tre numeri di un sistema floating point con $\beta = 10$ e $t = 8$ proviamo allora sommare tra loro questi numeri:

$$\begin{aligned}(a\oplus b)\oplus c &= 0.33678452\times10^2 \ominus 0.3367781\times10^2 = 0.64100000\times10^{-3}\\ a\oplus(b\oplus c) &= 0.23371258\times10^{-4}\oplus 0.618000000\times10^{-3} = 0.64137126\times10^{-3}\end{aligned}$$

il risultato esatto è dato da: $a+b+c = 0.641371258\times10^{-3}$

Capitolo 2

Operazioni base dell'algebra lineare

In questo capitolo presenteremo alcune delle operazioni base dell'algebra lineare prestando particolare attenzione alla complessità ed al modo di accesso ai dati, ad esempio se si acceda alle matrici per righe oppure per colonne; potrebbe sembrare un aspetto secondario, ma conoscere il metodo di accesso ai dati del proprio linguaggio di programmazione e sfruttarlo può portare ad un notevole aumento prestazionale, praticamente a costo zero.

2.1 axpy: prodotto vettore-scalare

$$y \leftarrow y + \alpha x \qquad\qquad x, y \in \mathbb{R}^n$$

L'operazione è quella di prodotto tra un vettore (x) ed uno scalare (α) con la possibilità di inserire un vettore di valori iniziali (y) sul quale verrà poi trascritto il risultato.

2.1.1 Implementazione

```
for i=1:n
   y(i)=y(i)+alpha*x(i);
end
```

2.1.2 Complessità

Il ciclo che viene eseguito ha lunghezza n e le operazioni che si svolgono al suo interno sono una somma (tra le componenti omologhe dei due vettori) ed un prodotto (quello tra lo scalare e gli elementi del vettore x) quindi tra due numeri reali; in totale vengono eseguite $2n$ flops, dove con flops abbreviamo 'floating point operations'.

2.2 dot product: prodotto scalare

$$c = x^T y \qquad x, y, \in \mathbb{R}^n$$

L'operazione è quella di prodotto scalare tra due vettori, cioè la somma del prodotto delle componenti omologhe dei due vettori, formalmente:

$$c = \sum_{i=1}^{n} x_i y_i$$

2.2.1 Implementazione

```
c=0
for i=1:n
   c=c+x(i)*y(i);
end
```

2.2.2 Complessità

Come nel caso dell'operazione precedente, il ciclo ha lunghezza n ed al suo interno si svolgono due flops: il prodotto delle i-esime componenti dei vettori e la somma di tale prodotto con la somma parziale ricavata dai primi i-1 passi; in definitiva abbiamo ancora $2n$ flops.

2.3 matvect, gaxpy: prodotto matrice-vettore

$$y = Ax \qquad A \in \mathbb{R}^{m \times n}$$

$$y = y + Ax \qquad A \in \mathbb{R}^{m \times n}$$

Questa operazione esegue il prodotto tra una matrice ed un vettore (matvect appunto, il primo caso) eventualmente con la possibilità di un contenuto pregresso (il secondo caso, chiamato gaxpy, generalized axpy): nel caso $y \neq \underline{0}$ il valore di ogni componente verrebbe adeguatamente sommato al valore del vettore prodotto Ax.

2.3.1 Implementazioni

Implementazione (i,j)

```
for i=1:m
   for j=1:n
      y(i)=y(i)+A(i,j)*x(j)
   end
end
```

Implementazione (j,i)

```
for i=1:n
   for j=1:m
      y(j)=y(j)+A(i,j)*x(i)
   end
end
```

2.3.2 Complessità

(i,j): il ciclo interno contiene una somma ed un prodotto, 2 flops, e viene eseguito n volte, $2n$ flops; il ciclo interno è eseguito m volte ottenendo così $2nm$ flops;

(j,i): il ciclo interno esegue $2m$ flops, e viene eseguito n volte, la complessità totale risulta essere $2mn$ flops.

Risulta dunque un algoritmo di ordine quadratico, proporzionale al prodotto delle dimensioni significative; come si vede le due possibili implementazioni hanno medesima complessità, l'unica cosa che cambia è il modo di accesso ai dati.

2.3.3 Accesso ai dati

(i,j): nel ciclo più interno l'indice i è una costante, per cui si accede a tutta la riga i-esima e la si moltiplica per il vettore x: è come se si eseguissero m dot product fra le m righe di A ed il vettore x:

$$y_1 = a^1 x, \cdots, y_m = a^m x$$

(j,i): in questo caso, all'interno del ciclo annidato rimane costante l'indice i della colonna e si scorrono tutte le m righe di A; tenendo presente che $x(i)$ rimane costante nel ciclo interno, quello che si ottiene è:

$$y \leftarrow (\cdots((y + a_1 x_1) + a_2 x_2) + \cdots + a_n x_n)$$

che altri non sono che n axpy.

In conclusione, l'implementazione (i, j) accede ad i dati per righe, mentre la sua speculare accede ai dati per colonne. La scelta tra questi due algoritmi deve tenere presente il metodo di memorizzazione delle matrici del proprio linguaggio di programmazione.

2.4 Prodotto matrice-matrice

$$C \leftarrow A \cdot B + C \qquad A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times r}, C \in \mathbb{R}^{m \times r}$$

L'operazione che si intende ottenere è quella di prodotto tra matrici, nel caso $C \equiv \mathcal{O}$; nel caso in cui $C \neq \mathcal{O}$ avremo anche un contenuto pregresso, come già visto in precedenza. Come si vedrà, poichè sono necessari tre cicli per completare l'algoritmo, si avranno $3! = 6$ possibili permutazioni ed altrettanti algoritmi, come prima se ne avevano $2! = 2$ algoritmo.

Formalmente quello che ci apprestiamo a fare è qualcosa di simile a

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

2.4.1 Implementazioni (i,j,k) e (j,i,k)

(i,j,k)

```
for i=1:m
   for j=1:r
      for k=1:n
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end
   end
end
```

(j,i,k)

La sua implementazione riveste scarso interesse, ne vedremo le ragioni nel seguente paragrafo.

2.4.2 Complessità di (i,j,k) e (j,i,k)

La complessità rimarrà sempre costante, e senza ripetere i calcoli degli algoritmi precedenti, vediamo che è $2mnr$ ancora il doppio del prodotto delle dimensioni significative. Rimane costante perché ciò che facciamo è permutare operazioni indipendenti tra loro e che quindi non si influenzano reciprocamente. L'algoritmo si può considerare un algoritmo di ordine cubico, piuttosto sgradevole, ma purtroppo meglio di così non si può fare.

Come accennato più sopra, l'implementazione (j,i,k) non presenta sostanziali variazioni rispetto a (i,j,k) dal momento che il nostro interesse è concentrato sul ciclo più interno, e questo non cambia.

2.4.3 Accesso ai dati di (i,j,k) e (j,i,k)

All'interno del ciclo più annidato $C(i,j)$ rimane costante, mentre al variare di k abbiamo $A(i,k) = a^i$ e $B(k,j) = b_j$, perciò ad A accediamo per righe mentre a B accediamo per colonne; precisamente all'interno di ogni ciclo ciò che eseguiamo è

$$c_{ij} = c_{ij} + a^i b_j$$

Come indicazione diciamo che se il linguaggio che utiliziamo accede alle matrici per righe, si deve fare in modo di rendere A la matrice più grande: questo comporta un minor degrado delle prestazioni, dal momento che risulta un accesso più ottimizzato per la matrice di dimensioni maggiori.

2.4.4 L'implementazione (j,k,i)

```
for j=1:r
   for k=1:n
      for i=1:m
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end
   end
end
```

2.4.5 Accesso ai dati di (j,k,i)

Nel ciclo interno $B(k,j)$ è una costante ed al variare di i $C(i,j)$ ed $A(i,k)$ sono due vettori colonna

$$c_j = c_j + a_k * b_{kj}$$

e quindi ci troviamo ad eseguire delle axpy accedendo alle matrici C ed A per colonna.

2.4.6 L'implementazione (i,k,j)

```
for i=1:m
   for k=1:n
      for j=1:r
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end
   end
end
```

2.4.7 Accesso ai dati di (i,k,j)

Questa volta a rimanere costante nel ciclo interno è $A(i,k)$ mentre $B(k,j) = b^k$ e $C(i,j) = c^i$ sono entrambi vettori riga

$$c^i = c^i + a_{ik}b^j$$

e quindi otteniamo di accedere a C e B per riga, mentre A rimane una costante.

2.4.8 L'implementazione (k,i,j) e (k,j,i)

(k,i,j)

```
for k=1:n
   for i=1:m
      for j=1:r
         C(i,j)=C(i,j)+A(i,k)*B(k,j)
      end
   end
end
```

(k,j,i)

Anche in questo caso non presentiamo l'implementazione poichè non introdurrebbe alcun elemento di interesse, come risulterà chiaro più avanti.

2.4.9 Accesso ai dati di (k,i,j) e (k,j,i)

Concentriamoci sull'implementazione (k,i,j) per poi mostrare le differenze tra le due; questa volta prendiamo però in esame i due cicli più interni, quelli eseguiti su i e j; possiamo notare che quello che andiamo a fare è

$$C \leftarrow C + a_k b^k$$

per un fissato k, per cui possiamo riscrivere $C = C + AB$ come

$$C = C + \sum_{k=1}^{n} a_k b^k;$$

abbiamo così ottenuto un'altra definizione di prodotto matriciale: costruiamo cioè la matrice C tramite correzioni di rango 1. Guardiamo adesso le differenze tra (k,i,j) e (k,j,i):

(k,i,j): il ciclo più interno può essere scritto come

$$c^i \leftarrow c^i + a_{ik}b^k$$

dove $A(i,k)$ è una costante e $C(i,j) = c^i$ e $B(k,j) = b^k$ sono due vettori riga; dunque questo algoritmo accede alle matrici per riga;

(k,j,i): in questo caso il ciclo interno si può scrivere nella forma:

$$c_j \leftarrow c_j + a_k b_{kj}$$

dove $B(k, j)$ è una costante ed i vettori $C(i, j) = c_j$ ed $A(i, k) = a_k$ sono due vettori colonna; dunque non facciamo altro che eseguire delle axpy accedendo alle matrici C ed A per colonna.

2.5 Scelta dell'algoritmo giusto

Si vuole puntualizzare un aspetto particolarmente importante perché fonte di ottimizzazione dell'esecuzione dell'algoritmo, a volte anche in modo considerevole. Mentre esponevamo gli algoritmi si faceva sempre notare il modo di accesso alle matrici per consentire di scegliere l'algoritmo giusto a seconda del metodo di memorizzazione delle matrici del nostro linguaggio di programmazione.

Questo non basta: esiste anche un'altra accortezza applicabile per la scelta dell'algoritmo, e cioè il principio secondo cui il ciclo più interno deve essere il più lungo possibile. Cerchiamo di vederne le ragioni: prima che un ciclo venga eseguito è necessario salvare alcuni registri, impostare alcune variabili ed altre operazioni che richiedono quello che è detto tempo di start-up; una volta completata questa fase, ad esclusione delle operazioni proprie del ciclo, le operazioni al contorno si limiteranno all'incremento del contatore del ciclo. Rendere il ciclo interno il più lungo possibile consente di diminuire questi tempi di startup (non produttivi dal il punto di vista dell'algoritmo, ma comunque necessari) ed a ottenere una maggiore efficienza.

Capitolo 3

Risoluzione di sistemi lineari

Il problema che andiamo ad affrontare adesso è quello di risolvere un sistema lineare, trovare cioè quei valori $x_1, \cdots, x_n$ tali da soddisfare il seguente sistema di equazioni:

$$\begin{cases} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & \vdots & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \cdots & + & a_{nn}x_n & = & b_n \end{cases}$$

che scritto nel compatto e semplice linguaggio matematico significa trovare quel vettore reale x tale che vaga l'uguaglianza:

$$Ax = b \qquad\qquad A \in \mathbb{R}^{n\times n}, \quad b, x \in \mathbb{R}^n.$$

L'ipotesi che faremo da qui in avanti sarà che $det(A) \neq 0$ cioè che la matrice A sia non singolare. Formalmente, e sotto questa ipotesi, potremmo trovare il vettore x come $x = A^{-1}b$ ma questo comporta il calcolo dell'inversa di A che risulta eccessivamente oneroso e per questo è una possibilità che non viene quasi mai utilizzata.

Dapprima vediamo casi semplici, in qui la soluzione del sistema lineare risulta quasi immediata grazie a forme particolari che A può assumere, per poi passare nel caso generale.

3.1 A diagonale

Se la matrice A è una matrice diagonale, allora assume la forma

$$\begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}.$$

Il determinante di una matrice siffatta è dato dal prodotto degli elementi diagonali

$$det(A) = \prod_{i=1}^{n} d_i$$

e risulta diverso da zero se e solo se $\forall i = 1, \cdots, n \; d_i \neq 0$.

La struttura a diagonale facilita molto il calcolo del vettore x, perché equivale ad avere n equazioni disaccoppiate, infatti

$$\begin{pmatrix} d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \Rightarrow \begin{cases} d_1 x_1 = b_1 \\ d_2 x_2 = b_2 \\ \vdots \quad \vdots \quad \vdots \\ d_n x_2 = b_n \end{cases}$$

e quindi ciascuna equazione specifica una componente della soluzione

$$x_i = \frac{b_i}{d_i} \qquad i = 1, \cdots, n$$

e sono tutte espressioni ben definite perché sappiamo che gli elementi diagonali sono tutti non nulli.

Invece di un problema di dimensione n, abbiamo risolto n problemi di dimensione 1, e quindi impieghiamo n flops.

Anche l'occupazione di memoria risulta lineare: non è necessario utilizzare una matrice $n \times n$ perché se sappiamo già che questa sarà diagonale ci basterà memorizzare gli elementi non nulli (quelli diagonali) in un vettore.

3.2 A ortogonale

Come sappiamo, se A è una matrice ortogonale, allora rispetta la proprietà che $AA^T = I$ ed in questo caso abbiamo già la matrice inversa disponibile: basta moltiplicare a sinistra per A^{-1} per ottenere la relazione $A^{-1} = A^T$. Questo è dunque un caso in cui il vettore x è calcolato tramite la matrice inversa:

$$Ax = b \quad \rightarrow \quad x = A^{-1}b \equiv A^T b$$

La soluzione la troviamo al costo di un prodotto matrice-vettore, che come sappiamo esegue $2n^2$ flops.

Naturalmente la scelta dell'algoritmo (i,j) oppure (j,i) risulta rilevante: per quanto detto prima dobbiamo optare per l'algoritmo che si adatta al nostro linguaggio di programmazione.

3.3 A triangolare inferiore

Se la matrice A è triangolare inferiore si presenta nella forma

$$A = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

caratterizzata da $a_{ij} = 0 \; \forall j > i$. Anche in questo caso il determinante è facilmente calcolabile, infatti si ottiene ancora come il prodotto degli elementi diagonali della matrice, allora

$$det(A) \neq 0 \quad \Leftrightarrow \quad \prod_{i=1}^{n} a_{ii} \neq 0 \quad \Leftrightarrow \quad a_{ii} \neq 0 \; \forall i = 1, \cdots, n$$

Proviamo con un esempio di una matrice triangolare inferiore 3×3 a vedere come funziona il modo di risoluzione:

$$\begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

$$\Downarrow$$

$$\begin{cases} a_{11}x_1 & & & & & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & & & = & b_2 \\ a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & = & b_3 \end{cases}.$$

Si vede che la prima componente del vettore soluzione x, x_1, è già disponibile, per cui possiamo trovare

$$x_1 = \frac{b_1}{a_{11}}$$

ed il fatto che gli elementi diagonali della matrice siano diversi da zero rende l'espressione ben formata.

Noto x_1, possiamo adesso ricavare x_2 nel seguente modo

$$x_2 = \frac{b_2 - a_{21} \overbrace{\frac{b_1}{a_{11}}}^{x_1}}{a_{22}}$$

ed adesso siamo in grado di trovare l'ultima componente del vettore soluzione come

$$x_3 = \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}}.$$

Quello che abbiamo cercato di fare con questo semplice esempio era mostrare come si poteva ottenere il vettore x: tramite un algoritmo iterativo si ottengono le componenti del vettore soluzione tramite la relazione

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j}{a_{ii}}$$

e partendo da $i = 1$ si arriva alla soluzione desiderata.

3.3.1 Implementazione in Matlab

Di seguito presentiamo la function Matlab che risolve un sistema lineare con la matrice dei coefficienti triangolare inferiore

```
function x=solveLT(A,b)
%SOLVELT Risolve il sistema lineare Ax=b con A matrice triangolare
%   inferiore
%
%   x=SOLVELT(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
%
%   See Also SOLVELTCS, SOLVEUT
  n=length(b);
  x=b;
  for i=1:n
     if A(i,i)==0
        disp('Matrice non risolvibile')
        break
     end
     for j=1:i-1
        x(i)=x(i)-A(i,j)*x(j);
     end
     x(i)=x(i)/A(i,i);
  end
```

3.3.2 Complessità, occupazione di memoria ed accesso ai dati

Il ciclo su j è eseguito $i-1$ volte ed al suo interno ci sono due operazioni, una somma ed un prodotto, e questo ciclo è seguito da una divisione; in definitiva

per ogni ciclo su i le operazioni sono $2(i-1)+1 = 2i-1$. Sommando al variare di i si ottiene

$$\sum_{i=1}^{n}(2i-1) = \frac{2n(n+1)}{2} - n = n^2$$

perciò vengono eseguite n^2 flops, ed anche l'occupazione è n^2, risulta dunque un algoritmo proporzionale alla dimensione della matrice.

Il ciclo interno accede alla matrice A per righe ma se, come già ampiamente discusso, il nostro linguaggio di programmazione memorizza le matrici per colonne risulta necessario rivedere il nostro algoritmo in modo opportuno.

3.3.3 Metodo column sweep

Con questo metodo, traducibile come "spazzolata a colonna", ciò che viene fatto è calcolare una componente (x_i) del vettore soluzione e poi sottrarre dal vettore dei termini noti gli elementi $a_{i+1,i}x_i, \cdots, a_{ni}x_i$.

3.3.4 Implementazione in Matlab

```
function x=solveLTcs(A,b)
%SOLVELTCS Risolve il sistema lineare Ax=b con A matrice triangolare
%   inferiore utilizzando il metodo column sweep
%
%   x=SOLVELTCS(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
%
%   See Also SOLVELT
  n=length(b);
  x=b;
  for j=1:n
     if A(j,j)==0
        disp('Matrice non risolvibile')
        break
     end
     x(j)=x(j)/A(j,j);
     for i=j+1:n
        x(i)=x(i)-A(i,j)*x(j);
```

```
    end
end
```

3.4 A triangolare superiore

Nel caso la matrice A sia triangolare superiore avrà una forma simile a questa:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix}$$

ed anche in questo caso, la non-nullità degli elementi diagonali è garanzia per la non singolarità della matrice; dunque la risoluzione del sistema lineare $Ax = b$ è equivalente a risolvere il seguente sistema di n equazioni:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ \quad + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \qquad\qquad \ddots \quad \vdots \quad \vdots \quad \vdots \\ \qquad\qquad\qquad a_{nn}x_n = b_n \end{cases}$$

Come nel caso precedente, quello che bisogna fare è ottenere passo passo le soluzioni, questa volta partendo dall'ultima equazione per risalire verso la prima, che contiene tutte le variabili. Il generico passo i-esimo si può esprimere come:

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{ij}x_j}{a_{ii}}$$

3.4.1 Implementazione in Matlab

Vediamo un algoritmo simile alla risoluzione della matrice triangolare inferiore, solo che in questo caso le sostituzioni avverranno all'indietro.

```
function x=solveUT(A,b)
%SOLVEUT Risolve il sistema lineare Ax=b con A matrice triangolare
%   superiore
%
%   x=SOLVEUT(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
```

```
%
%   See Also SOLVEUTCS, SOLVELT
  n=length(b);
  x=b;
  for i=n:-1:1
     if A(i,i)==0
        disp('Matrice non risolvibile')
        break
     end
     for j=i+1:n
        x(i)=x(i)-A(i,j)*x(j);
     end
     x(i)=x(i)/A(i,i);
  end
```

3.4.2 Complessità, occupazione di memoria ed accesso ai dati

Il metodo di risoluzione è del tutto simile a quanto visto per le matrici triangolari inferiori, per cui tutti i risultati che abbiamo ottenuto in precedenza per quanto riguarda complessità e occupazione di memoria possono essere applicati anche in questo caso; e lo stesso vale per le modifiche column sweep. Naturalmente anche qui accediamo alla matrice per righe, si dovra dunque prevedere una modifica per consentire l'accesso per colonne.

3.4.3 Metodo column sweep

Nel caso il linguaggio che stiamo utilizzando acceda alle matrici per colonne possiamo utilizzare nuovamente il metodo già visto di column sweep per fare in modo che la risoluzione acceda alla matrice dei coefficienti non per righe bensì per colonne.

In modo speculare a quanto visto per le matrici triangolari inferiori una volta ottenuta una componente del vettore soluzione, diciamo x_j, al vettore dei termini noti verrà sottratto quel fattore $a_{ij}x_j$ che nelle equazioni rimaste ancora da esaminare contiene appunto la soluzione appena trovata.

3.4.4 Implementazione in Matlab

```
function x=solveLTcs(A,b)
%SOLVEUTCS Risolve il sistema lineare Ax=b con A matrice triangolare
%   superiore utilizzando il metodo column sweep
%
%   x=SOLVEUTCS(A,b)
%
%   I parametri della funzione sono:
```

```
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
%
%   See Also SOLVEUT
  n=length(b);
  x=b;
  for j=n:-1:1
     if A(j,j)==0
        disp('Matrice non risolvibile')
        break
     end
     x(j)=x(j)/A(j,j);
     for i=1:j-1
        x(i)=x(i)-A(i,j)*x(j);
     end
  end
```

3.5 A matrice generica

Naturalmente non possiamo certo limitarci a risolvere sistemi lineari solo nei casi particolari appena visti: questi ci servivano come esempi di risoluzioni abbastanza semplici. Ma nel caso di A generica come procediamo? Riuscendo a fattorizzare la matrice data in matrici di cui conosciamo la risoluzione del sistema lineare associato, e cioè se riusciamo a scrivere

$$A = F_1 \cdot F_2 \cdot F_3 \cdot \cdots \cdot F_p \qquad F_i = \begin{cases} triangolare \\ diagonale \\ ortogonale \end{cases}$$

riuscendo a ridurre anche di molto l'ordine di grandezza del problema.

La non singolarità di A ci porta a dire che

$$0 \neq det(A) = det(F_1 \cdot F_2 \cdot \cdots \cdot F_p) = \prod_{i=1}^{p} det(F_i) \Rightarrow det(F_i) \neq 0 \forall i$$

e dunque possiamo risolvere il sistema lineare $Ax = b$ come

$$\begin{aligned} Ax = b \quad \equiv \quad & (F_1 \cdot \cdots \cdot F_p)x = b \quad \rightarrow \quad F_1(\underbrace{F_2 \cdot \cdots \cdot F_p x}_{x^{(1)}}) = b \\ & F_1 x^{(1)} = b \\ & F_2(F_3 \cdot \cdots \cdot F_p x) = x^{(1)} \\ & \vdots \end{aligned}$$

e dunque otteniamo la successione

$$\begin{aligned} &F_1 x^{(1)} = b \\ &F_2 x^{(2)} = x^{(1)} \\ &F_3 x^{(3)} = x^{(2)} \\ &\vdots \\ &F_p x^{(p)} = x^{(p-1)} \\ &x \equiv x^{(p)} \end{aligned}$$

Abbiamo così ridotto il problema iniziale $Ax = b$ nel trovare una fattorizzazione di A e risolvere poi i corrispondenti sistemi lineari.

Capitolo 4

Fattorizzazioni di matrici

Come accennato nel capitolo precedente, per poter risolvere un sistema lineare

$$Ax = b$$

nel caso la matrice dei coefficienti non sia in una forma particolare diventa necessaria la fattorizzazione di A, che consiste nel trovare quella scomposizione in fattori con i quali poi risulterà facile risolvere il sistema dato.

Quello che ci proponiamo di fare adesso è vedere come è possibile fattorizzare una matrice in modo da facilitare la risoluzione del problema iniziale.

4.1 Fattorizzazione $A = LU$

La somma ed il prodotto di matrici triangolari, inferiori o superiori, sono ancora triangolari, inferiori o superiori; in particolare per gli elementi diagonali abbiamo

1. $C = A + B \rightarrow c_{ii} = a_{ii} + b_{ii}$

2. $C = A \cdot B \rightarrow c_{ii} = a_{ii} \cdot b_{ii}$

3. $C = A^{-1} \rightarrow c_{ii} = \frac{1}{a_{ii}}$ (A non singolare)

Consideriamo adesso un particolare caso di matrice triangolare, quella triangolare inferiore a diagonale unitaria

$$L = (l_{ij}) = \begin{cases} l_{ij} = 0 & j > i \\ l_{ij} = 1 & j = i \end{cases}$$

$$L = \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ * & & 1 \end{pmatrix}.$$

In questo caso il determinante di L è sicuramente diverso da zero, ed inoltre abbiamo che tutti gli autovalori della matrice sono uguali ad uno: $\lambda_i = 1 \quad \forall i = 1, \cdots, n$. Dalle proprietà (2) e (3) si possono derivare queste due proprietà delle matrici triangolari a diagonale unitaria:

P1: il prodotto di matrici triangolari a diagonale unitaria è una matrice triangolare a diagonale unitaria (superiore o inferiore);

P2: l'inversa di una matrice triangolare a diagonale unitaria è una matrice dello stesso tipo.

Se possiamo scrivere $A = LU$ con

- L triangolare inferiore a diagonale unitaria;
- U triangolare superiore

allora la matrice A si dirà fattorizzabile LU. Non è assicurato che tutte le matrici siano fattorizzabili LU ma il seguente teorema ci fornisce un importante risultato:

Teorema. *Se A è fattorizzabile LU, tale fattorizzazione è unica.*

Dimostrazione. Supponiamo che $A = L_1U_1 = L_2U_2$, vedremo che questo implicherà $L_1 = L_2 \wedge U_1 = U_2$.

A è una matrice non singolare per ipotesi, allora

$$0 \neq det(A) = \begin{cases} det(L_1U_1) = \underbrace{det(L_1)}_{1} det(U_1) = det(U_1) \\ det(L_2U_2) = \overbrace{det(L_2)}^{1} det(U_2) = det(U_2) \end{cases}$$

Dunque tutti i fattori sono non singolari, L per costruzione, U come conseguenza e perciò ammettono inversa:

$$L_1U_1 = L_2U_2$$

$$\underbrace{{L_1}^{-1}L_1}_{I} U_1{U_2}^{-1} = {L_1}^{-1}L_2\underbrace{U_2{U_2}^{-1}}_{I}$$

$$U_1{U_2}^{-1} = {L_1}^{-1}L_2$$

Sia U_1 che ${U_2}^{-1}$ sono matrici triangolari superiori, e tale è anche il loro prodotto; lo stesso vale per ${L_1}^{-1}$ e L_2 che sono entrambe triangolari inferiori a diagonale unitaria e della stessa forma è il loro prodotto. L'uguaglianza tra queste due quantità implica che la matrice a destra ed a sinistra dell'uguale sia diagonale, e ricordandosi che ${L_1}^{-1}L_2$ è a diagonale unitaria, questo implica che siano uguali all'identità; siamo perciò giunti alla conclusione che

$$U_1{U_2}^{-1} = {L_1}^{-1}L_2 = I \Rightarrow \begin{cases} U_1{U_2}^{-1} = I & \Rightarrow \quad U_1 = U_2 \\ {L_1}^{-1}L_2 = I & \Rightarrow \quad L_1 = L_2 \end{cases}$$

□

4.1.1 Gauss ci da una mano

Cerchiamo di risolvere il nostro problema della fattorizzazione risolvendone prima uno più semplice: dato un vettore

$$v = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ \vdots \\ v_n \end{pmatrix} \in \mathbb{R}^n \quad v_k \neq 0$$

cerchiamo una matrice L triangolare inferiore a diagonale unitaria tale che

$$Lv = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

che quindi annulli gli elementi dal (k+1)-esimo in poi lasciando inalterati gli altri, ed in modo che sia la più semplice possibile.

Definiamo allora un vettore, detto vettore elementare di Gauss nel seguente modo:

$$g = \frac{1}{v_k}(\underbrace{0 \cdots 0}_{k}, v_{k+1}, \cdots, v_n)^T$$

Adesso possiamo definire la matrice L come

$$L = I - ge_k^T \qquad e_k = \begin{pmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \leftarrow \text{in k-esima posizione}$$

cioè come la matrice identità a cui viene sommata una correzione di rango 1 (infatti $dim(g) = n \times 1$ mentre $dim(e_k^T) = 1 \times n$ e dunque il loro prodotto sarà una matrice $n \times n$ le cui colonne saranno tutte nulle tranne la k-esima che contiene il vettore g; questa matrice aggiungerà contributi solo alla parte strettamente inferiore di L):

$$L = \begin{pmatrix} 1 & & & & & & & \\ 0 & \ddots & & & & & & \\ \vdots & \ddots & \ddots & & & & & \\ \vdots & & 0 & 1 & & & & \\ \vdots & & \vdots & -\frac{v_{k+1}}{v_k} & \ddots & & & \\ \vdots & & \vdots & \vdots & 0 & \ddots & & \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & -\frac{v_n}{v_k} & 0 & \cdots & 0 & 1 \end{pmatrix}$$

con elementi sottodiagonali non nulli solo in corrispondenza della k-esima colonna.

Una matrice siffatta si dice matrice elementare di Gauss, e soddisfa le nostre richieste, proviamo a verificarlo:

$$Lv = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$Lv = (I - ge_k^T)v = Iv - ge_k^T v = v - g(e_k^T v) = v - gv_k =$$

$$= \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ \vdots \\ v_n \end{pmatrix} - \frac{1}{v_k} v_k \begin{pmatrix} 0 \\ \vdots \\ 0 \\ v_{k+1} \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

proprio il vettore che volevamo ottenere.

La matrice L è stata trovata solo grazie al fatto che $v_k \neq 0$ che è condizione necessaria e sufficiente affinchè il problema abbia soluzione.

Vediamo adesso che forma ha la matrice L^{-1}. Sappiamo per certo che è triangolare inferiore a diagonale unitaria, ed è ottenuta come l'identità a cui viene aggiunto lo stesso termine di correzione di rango uno di L:

$$L^{-1} = I + ge_k^T$$

come verifica possiamo calcolare

$$L^{-1}L = (I + ge_k^T)(I - ge_k^T) = I + ge_k^T - ge_k^T - ge_k^T ge_k^T =$$

$$= I - g(\underbrace{e_k^T g}_{g_k = 0})e_k^T = I$$

4.1.2 Il metodo di eliminazione di Gauss

Adesso andremo a compiere la vera e propria fattorizzazione. Quello che faremo sarà trovare una matrice triangolare superiore ottenendola da A attraverso opportune trasformazioni. Innanzitutto poniamo

$$A = \begin{pmatrix} a_{11}^{(1)} & \cdots & a_{1n}^{(1)} \\ \vdots & \ddots & \vdots \\ a_{n1}^{(1)} & \cdots & a_{nn}^{(1)} \end{pmatrix} \equiv A^{(1)}$$

dove con l'indice (1) stabiliamo l'ultimo passaggio in cui quel determinato elemento della matrice è stato modificato. In modo iterativo, al passo i-esimo ci concentreremo sulla colonna i-esima ed il compito del passo corrente sarà quello di fare in modo che quella colonna diventi la colonna di una matrice triangolare superiore e quindi annulli gli elementi dall'(i+1)-esima posizione in poi.

Al primo passo si devono azzerare gli elementi $a_{21}, \cdots, a_{n1}$, quindi se $a_{11} \neq 0$ possiamo definire

$$g_1 = \frac{1}{a_{11}}(0, a_{21}^{(1)}, \cdots, a_{n1}^{(1)})^T$$

$$L_1 = I - g_1 e_1^T$$

tali che

$$L_1 \begin{pmatrix} a_{11}^{(1)} \\ \vdots \\ a_{n1}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Possiamo adesso moltiplicare a sinistra la matrice di partenza A per L_1 per ottenere

$$L_1 A = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix} \equiv A^{(2)}$$

Come si può vedere l'indice all'interno della sottomatrice di A è stato aggiornato: infatti applicando la matrice L_1 ad A gli elementi sulla prima riga rimangono inalterati (il vettore g_1 è stato scelto in modo che lasciasse inalterata proprio la prima componente del vettore) e si ottiene l'annullamento degli elementi sotto la diagonale della prima colonna; come effetto secondario otteniamo la modifica degli elementi della sottomatrice di A: infatti mentre per costruzione della matrice L_1 la prima colonna viene modificata come volevamo, applicando detta matrice alle altre colonne, queste verranno modificate senza un preciso schema, da qui l'aggiornamento dell'indice. Infatti, sia v un vettore qualsiasi:

$$L_1 v = \begin{pmatrix} 1 & & & \\ * & \ddots & & \\ \vdots & & \ddots & \\ * & & & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ \vdots \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} v_1 \\ * \\ \vdots \\ * \end{pmatrix}$$

dove con gli elementi $*$ intendiamo degli elementi che non vogliamo caratterizzare ma solitamente diversi da quelli precedenti.

Abbiamo così concluso il primo passo.

Durante il generico passo i-esimo vediamo come prosegue l'algoritmo. Se $a_{ii} \neq 0$ possiamo definire

$$g_i = \frac{1}{a_{ii}^{(i)}}(\underbrace{0 \cdots 0}_{i}, a_{i+1,i}^{(i)}, \cdots, a_{ni}^{(i)})^T$$

$$L_i = I - g_i e_i^T$$

dai quali possiamo ottenere:

$$L_i L_{i-1} \cdots L_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & \cdots & \cdots & \cdots & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & & & & & \vdots \\ \vdots & & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & a_{ii}^{(i)} & \cdots & \cdots & a_{in}^{(i)} \\ \vdots & & & & 0 & a_{i+1,i+1}^{(i+1)} & \cdots & a_{i+1,n}^{(i+1)} \\ \vdots & & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a_{n,i+1}^{(i+1)} & \cdots & a_{nn}^{(i+1)} \end{pmatrix} \equiv A^{(i+1)}$$

Dopo $n-1$ passi otteniamo:

$$L_{n-1} \cdots L_i \cdots L_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & \cdots & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & & & \vdots \\ \vdots & & \ddots & a_{ii}^{(i)} & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a_{nn}^{(n)} \end{pmatrix} \equiv A^{(n)} \equiv U$$

Abbiamo dunque ottenuto $L_{n-1} \cdots L_1 A = U$, con le matrici L_i triangolari inferiori a diagonale unitaria; se chiamiamo $L^{-1} = L_{n-1} \cdots L_1$, L sarà anch'essa una matrice triangolare inferiore a diagonale unitaria e quindi possiamo scrivere:

$$L^{-1}A = U \quad \rightarrow \quad LL^{-1}A = LU \quad \rightarrow \quad A = LU$$

la fattorizzazione che cercavamo.

4.1.3 Requisiti algebrici della fattorizzazione

Prestiamo attenzione ai requisiti algebrici di questa fattorizzazione. Scriviamo la matrice A come

$$A = \left(\begin{array}{c|c} A_i & B_i \\ \hline C_i & D_i \end{array}\right) \quad \text{con } A_i = \begin{pmatrix} a_{11} & \cdots & a_{1i} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ii} \end{pmatrix}$$

i-esima sottomatrice principale di A.

Adesso possiamo scrivere

$$A = LU$$

come

$$\left(\begin{array}{c|c} A_i & B_i \\ \hline C_i & D_i \end{array}\right) = \left(\begin{array}{c|c} l_i & \mathcal{O} \\ \hline F_i & G_i \end{array}\right)\left(\begin{array}{c|c} U_i & H_i \\ \hline \mathcal{O} & M_i \end{array}\right) = \left(\begin{array}{c|c} l_iU_i & l_iH_i \\ \hline F_iU_i & F_iH_i + G_iH_i \end{array}\right)$$

e questo significa che $A_i = l_iU_i \quad \forall i = 1, \cdots, n$ con l_i matrice triangolare inferiore a diagonale unitaria e U_i matrice triangolare superiore; definiamo ora il minore principale di ordine i come il determinante della sottomatrice principale di ordine i, allora

$$det(A_i) = det(l_iU_i) = det(l_i)det(U_i) = det(U_i)$$

$$\forall i \quad det(A_i) = det(U_i) = \prod_{j=1}^{i} a_{jj}^{(j)} = \left(\prod_{j=1}^{i-1} a_{jj}^{(j)}\right) a_{ii}^{(i)} = det(A_{i-1})a_{ii}^{(i)}$$

Quanto appena scritto ci consente un'utile interpretazione: la condizione $det(A_{i-1}) \neq 0$ ci consente di arrivare fino al passo i-esimo, mentre $a_{ii}^{(i)} \neq 0$ ci consente di eseguire proprio il passo i-esimo.

La fattorizzazione LU è quindi definita se e solo se tutti i minori principali sono non nulli, e questa è una condizione molto forte: il problema originario $Ax = b$ è ben posto solo se il minore principale di ordine massimo è non nullo (la matrice dei coefficienti è non singolare). Questo vincolo può essere formulato sotto forma di teorema che sancisce la condizione necessaria e sufficiente affinchè la fattorizzazione esista:

Teorema. *A è fattorizzabile LU se e solo se tutti i minori principali di A sono non nulli*

Guardiamo adesso come è fatta la matrice L. Dall'applicazione dell'algoritmo precedente quello che otteniamo è

$$L_{n-1}L_{n-2} \cdot \cdots \cdot L_2L_1 = L^{-1}$$

ma naturalmente eseguire esplicitamente il prodotto delle $n-1$ matrici per poi calcolare l'inversa, ed ottenere così L, risulta eccessivamente complesso; ricordandoci le proprietà delle matrici inverse otteniamo che

$$L = (L_{n-1} \cdot \cdots \cdot L_2L_1)^{-1} = L_1^{-1}L_2^{-1} \cdot \cdots \cdot L_{n-1}^{-1}$$

decisamente più semplice da ottenere, vediamone le ragioni:

$$L_i^{-1} = I + g_ie_i^T$$

e quindi otteniamo

$$\begin{aligned} L &= (I + g_1 e_1^T) \cdots (I + g_{n-1} e_{n-1}^T) = \\ &= I + g_1 e_1^T + \cdots + g_{n-1} e_{n-1}^T + \text{ qualcosa nella forma} \\ &\quad (g_i e_i^T g_j e_j^T \quad j > i) \end{aligned}$$

ma $e_i^T g_j = 0$ per $j > i$ e quindi quel "qualcosa" si riduce a zero, perciò la matrice L che cerchiamo è

$$L = I + g_1 e_1^T + \cdots + g_{n-1} e_{n-1}^T$$

$$L = \begin{pmatrix} 1 & & & & & & 0 \\ & \ddots & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & -\frac{a_{i+1,i}^{(i)}}{a_{ii}^{(i)}} & \ddots & & \\ & & & \vdots & & \ddots & \\ & & & \vdots & & & \ddots \\ * & & & -\frac{a_{ni}^{(i)}}{a_{ii}^{(i)}} & & * & 1 \end{pmatrix}$$

dove in colonna i-esima troviamo l'i-esimo vettore elementare di Gauss.

Questo metodo è detto metodo di eliminazione di Gauss visto l'utilizzo delle matrici e dei vettori elementari dello stesso Gauss

4.1.4 Costruzione dell'algoritmo di fattorizzazione LU

Cerchiamo adesso di costruire un algoritmo per fattorizzare la matrice A come LU prestando particolare attenzione al costo computazionale ed all'occupazione di memoria.

Dal punto di vista dell'occupazione di memoria questo algoritmo risulta particolarmente efficiente perché riusciamo a riscrivere sopra alla matrice A le informazioni necessarie alla fattorizzazione e cioè la parte triangolare superiore di U è la parte strettamente triangolare inferiore di L: essendo una matrice a diagonale unitaria non c'è necessità di memorizzarne anche la diagonale. Lo spazio di memoria necessario è dunque quello occupato dalla matrice A.

Durante il primo passo vengono azzerati gli elementi $a_{21}^{(1)} \cdots a_{n1}^{(1)}$, possiamo allora utilizzare questi spazi lasciati vuoti per memorizzarci qualcosa. Ricordandoci come abbiamo costruito la matrice elementare di Gauss

$$L_1 = I - g_1 e_i^T$$

$$\text{con } g_1 = \frac{1}{a_{11}^{(1)}}(0 a_{21}^{(1)} \cdots a_{n1}^{(1)})^T$$

le informazioni necessarie per ricostruire L_1 sono gli elementi non nulli del vettore g_1 che altri non sono che gli elementi da azzerare divisi per l'elemento diagonale, per cui quello che andremo a fare sarà dividere gli elementi in loco per $a_{11}^{(1)}$.

Applicando questo passaggio nella parte strettamente inferiore di A vi otterremo gli elementi non nulli dei vettori g_i, che risulterà essere proprio la parte strettamente inferiore di L.

4.1.5 Implementazione in Matlab

L'algoritmo he implementa il metodo di eliminazione di Gauss per la fattorizzazione della matrice A in LU è il seguente:

```
function A=fattLU(A)
%FATTLU Fattorizza la matrice A come LU tramite il metodo di
%   eliminazione di Gauss
%
%   A=FATTLU(A)
%
%   I parametri della funzione sono:
%       A -> la matrice quadrata da fattorizzare
%
%   I valori di ritorno sono:
%       A -> la matrice modificata contenente nella parte
%            triangolare superiore U e nella parte strettamente
%            triangolare inferiore L
%
%   See Also SOLVELU, FATTPALU, FATTLDLT, FATTQR
  n=length(A);
  for i=1:n-1
     if A(i,i)==0
        disp('Non è possibile fattorizzare la matrice come LU')
        return
     end
(1)  A(i+1:n,i)=A(i+1:n,i)/A(i,i);
(2)  A(i+1:n,i+1:n)=A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
  end
```

4.1.6 Analisi dell'algoritmo e costo computazionale

La riga (1) divide gli elementi sottodiagonali per $a_{ii}^{(i)}$, mentre la riga (2) necessita di maggiori chiarimenti. Tale riga modifica gli elementi che si

trovano sotto la riga i-esima e a destra della colonna i-esima escluse; vediamo cosa succede nel caso $i = 1$: moltiplichiamo L_1 ad $A \equiv A^{(1)}$

$$L_1 A^{(1)} = (I - g_1 e_1^T) A^{(1)} = A^{(1)} - g_1 e_1^T A^{(1)} = A^{(1)} - g_1 (e_1^T A^{(1)})$$

si nota che $(e_1^T A^{(1)})$ altri non è che la prima riga di A, a^1, e quindi scriviamo

$$L_1 A^{(1)} = A^{(1)} - g_1 a^1.$$

Concentriamoci adesso su $g_1 a^1$: la matrice che otteniamo dal prodotto ha la prima riga composta di elementi uguali a zero (la prima componente di g_1 è nulla) e quindi possiamo considerare g_1 dal primo elemento non nullo in poi; inoltre la prima colonna di $g_1 a^1$ risulta essere uguale ad a_1 dal secondo elemento in poi (il primo è zero): se la matrice così fatta fosse sottratta direttamente ad $A^{(1)}$ teoricamente andrebbe ad azzerare gli elementi sottodiagonali della prima colonna, ma proprio in quelle posizioni abbiamo memorizzato gli elementi non nulli del vettore g_1 che quindi verrebbero modificati; possiamo allora limitarci alla porzione suddetta.

Passiamo adesso al costo computazionale: durante il passo i-esimo vengono eseguite:

1. $(n - i)$ operazioni nella riga (1), le divisioni;

2. $(n - i)^2$ moltiplicazioni ed altrettante sottrazioni per la riga (2) per un totale di $2(n - i)^2$ operazioni.

Sommando tra 1 ed $n - 1$ otteniamo

$$\begin{aligned}
\sum_{i=1}^{n-1} ((n-i) + 2(n-i)^2) &= \sum_{i=1}^{n-1} (n-i) + 2 \sum_{i=1}^{n-1} (n-i)^2 \\
&= \sum_{i=1}^{n-1} i + 2 \sum_{i=1}^{n-1} i^2 \\
&= \frac{n(n-1)}{2} + 2 \left(\frac{n(n-1)(2n-1)}{6} \right) \\
&= \frac{n(n-1)}{2} + \frac{n(n-1)(2n-1)}{3} \\
&\approx \frac{2}{3} n^3
\end{aligned}$$

La fattorizzazione ha un costo proporzionale al cubo della dimensione significativa; per risolvere il sistema lineare associato ci sarà anche un costo proporzionale ad n^2.

4.1.7 Sperimentazioni dell'algoritmo

Faremo alcuni esempi di funzionamento dell'algoritmo di fattorizzazione $A = LU$ per osservarne il comportamento.

Per verificare la bontà dell'algoritmo abbiamo generato casualmente una matrice triangolare inferiore a diagonale unitaria ed una triangolare superiore e le abbiamo moltiplicate tra loro ed abbiamo poi applicato l'algoritmo di fattorizzazione al prodotto, ottenendo

```
» l1=round(10*(tril(rand(10,10),-1)))+eye(10)

l1 =

     1     0     0     0     0     0     0     0     0     0
     4     1     0     0     0     0     0     0     0     0
     1     5     1     0     0     0     0     0     0     0
     4     6     1     1     0     0     0     0     0     0
     9     1     5     5     1     0     0     0     0     0
     4     5     0     5     4     1     0     0     0     0
     3     5     2     3     4     8     1     0     0     0
     4     9     3     4     5     8     8     1     0     0
     7     9     9     9    10     9     4     6     1     0
     7     5     3     8     3     6     0     5     6     1

» u1=round(10*(triu(rand(10,10))))

u1 =

     8     0     7     3     5     2     0     8    10    10
     0     4     4    10     1     6     4     3     4     5
     0     0     5     5     7    10     7     2     1     5
     0     0     0     4     0     3     9     7     7     1
     0     0     0     0     6     5     5     3     9     5
     0     0     0     0     0     6     1     5     2     4
     0     0     0     0     0     0     5     8     0     3
     0     0     0     0     0     0     0     9     5     3
     0     0     0     0     0     0     0     0     1     8
     0     0     0     0     0     0     0     0     0     1

» a1=l1*u1

a1 =

     8     0     7     3     5     2     0     8    10    10
```

```
    32     4    32    22    21    14     4    35    44    45
     8    20    32    58    17    42    27    25    31    40
    32    24    57    81    33    57    40    59    72    76
    72     4    92    82    87    94    89   123   143   130
    32    20    48    82    49    79    86    99   133    94
    24    20    51    81    58   133    94   124   125   123
    32    36    79   133    80   177   166   221   173   188
    56    36   130   192   167   289   259   325   317   293
    56    20    84   118    79   149   134   217   219   221
```

```
» A=fattlu(a1)

A =

     8     0     7     3     5     2     0     8    10    10
     4     4     4    10     1     6     4     3     4     5
     1     5     5     5     7    10     7     2     1     5
     4     6     1     4     0     3     9     7     7     1
     9     1     5     5     6     5     5     3     9     5
     4     5     0     5     4     6     1     5     2     4
     3     5     2     3     4     8     5     8     0     3
     4     9     3     4     5     8     8     9     5     3
     7     9     9     9    10     9     4     6     1     8
     7     5     3     8     3     6     0     5     6     1

»
```

dove, nella parte strettamente triangolare inferiore di A abbiamo la parte strettamente triangolare inferiore di $l1$ e nella parte triangolare superiore abbiamo $u1$.

Proviamo adesso l'algoritmo su una matrice completamente casuale:

```
» a1

a1 =

     2     6     5     1     9     6     1     9     9    10
     3     1     9     2    10     6     8     3     5     5
     6     9     6     1     8     2     5     3     7     5
     8     7     4     5     9     7     6     2     7     4
     0     0     8     8     4     1     5     6    10     8
     4     7     7     2     7     5     1     6     6     5
     8     3     4     0     3     3     3     9     5     8
     2     8     5     1     8     6     9     2     6     5
```

```
     3     1     9     5     2     3     4     3    10     1
     6     2     6     9     4     8     1     6     4     1

≫ A=fattlu(a1)

A =

  Columns 1 through 4

   2.00000000000000   6.00000000000000   5.00000000000000   1.00000000000000
   1.50000000000000  -8.00000000000000   1.50000000000000   0.50000000000000
   3.00000000000000   1.12500000000000 -10.68750000000000  -2.56250000000000
   4.00000000000000   2.12500000000000   1.79532163742690   4.53801169590643
                   0                  0  -0.74853801169591   1.34020618556701
   2.00000000000000   0.62500000000000   0.36842105263158   0.13917525773196
   4.00000000000000   2.62500000000000   1.86549707602339  -0.11726804123711
   1.00000000000000  -0.25000000000000  -0.03508771929825   0.00773195876289
   1.50000000000000   1.00000000000000                   0   0.66108247422680
   3.00000000000000   2.00000000000000   1.12280701754386   1.73582474226804

  Columns 5 through 8

   9.00000000000000   6.00000000000000   1.00000000000000   9.00000000000000
  -3.50000000000000  -3.00000000000000   6.50000000000000 -10.50000000000000
 -15.06250000000000 -12.62500000000000  -5.31250000000000 -12.18750000000000
   7.47953216374269  12.04093567251462  -2.27485380116959  10.19298245614035
 -17.29896907216495 -24.58762886597938   4.07216494845361 -16.78350515463917
   0.24880810488677   3.96811680572110  -3.80184743742551   1.80989272943981
  -0.29849523241955   1.13394533303296  -2.89220545167831  17.43144101524367
   0.14228247914184   0.55752046256664  -3.80207968220586  57.52323090703740
   0.74828665077473   1.87457760756927  -0.54753281146551   0.20812412887897
   0.69778009535161   1.62072914320042   0.61062429411536  -0.10213668385438

  Columns 9 through 10

   9.00000000000000  10.00000000000000
  -8.50000000000000 -10.00000000000000
 -10.43750000000000 -13.75000000000000
   7.80116959064327   9.93567251461988
  -8.26804123711340 -15.60824742268041
  -1.87067938021454  -1.18355184743743
  11.35173087031614  17.74881730119396
  39.82797834638001  62.30377380535107
   7.46266388810654   0.08094519791148
```

```
  -0.25256005115097  -2.45256300872870

≫
```

come riprova possiamo

```
≫ a2=(tril(A,-1)+eye(10))*triu(A)

a2 =

     2     6     5     1     9     6     1     9     9    10
     3     1     9     2    10     6     8     3     5     5
     6     9     6     1     8     2     5     3     7     5
     8     7     4     5     9     7     6     2     7     4
     0     0     8     8     4     1     5     6    10     8
     4     7     7     2     7     5     1     6     6     5
     8     3     4     0     3     3     3     9     5     8
     2     8     5     1     8     6     9     2     6     5
     3     1     9     5     2     3     4     3    10     1
     6     2     6     9     4     8     1     6     4     1

≫
```

4.1.8 Risoluzione di $Ax = b$ tramite fattorizzazione LU

Non dobbiamo dimenticarci che il nostro problema di partenza era la risoluzione del sistema lineare $Ax = b$; vediamo come questo problema cambia dopo la fattorizzazione della matrice A: dal momento che possiamo scrivere

$$A = LU$$

il nostro problema si traduce in

$$Ax = b \qquad \Rightarrow \qquad LUx = b$$

e quindi dobbiamo risolvere il sistema

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

che equivale a risolvere prima un sistema lineare la cui matrice dei coefficienti è triangolare inferiore a diagonale unitaria, per poi sfruttare la soluzione appena trovata per identificare la soluzione cercata risolvendo un sistema con matrice triangolare superiore.

4.1.9 Implementazione in Matlab risolvere $LUx = b$

Il codice che risolve il problema suddetto è il seguente:

```
function x=solveLU(A,b)
%SOLVELU Risolve il sistema lineare Ax=b fattorizzando la matrice
%   A come LU ed infine risolvendo i sistemi
%                Ly=b
%                Ux=y
%
%   x=SOLVELU(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
%
%   See Also FATTLU
  A=fattLU(A);
  n=length(b);
  x1=b;
  for i=2:n
     for j=1:i-1
         x1(i)=x1(i)-A(i,j)*x1(j);
     end
  end
  x=solveUT(A,x1);
```

4.1.10 Sperimentazioni dell'algoritmo

Proviamo il nostro algoritmo generando casualmente una matrice 10×10 ed un vettore per verificare la soluzione:

```
≫ a1=round(10*rand(10))

a1 =

     8     1     2     9     3     4     7     1     6     1
     3     7     7     6     4     6     9     3     2     7
     6    10     4     2     6     3     0    10     5     0
     1     1     1     1     6     2     6     4     2     3
     4     5     7     8     7     2     6     8     7     1
     1     1     9     4    10     2     7     9     5     4
```

```
     8     7     1     9     3     1     8     1     6     4
     6     3     4    10     2     1     8     5     0     3
     8     1     4     4     9     5     5     8     3     9
    10     2     5     3     3     7     3     5     4     2

» b1=round(10*rand(10,1))

b1 =

     1
     3
     2
     7
     8
     3
     8
     8
     1
     3

» x=solvelu(a1,b1)

x =

  -2.48449679544613
  -1.57729575627816
  -2.90299977582920
  -1.60498736801344
  -9.46718019044310
   0.07933652351090
   3.35772199201200
   7.37058528117625
   6.07222618703308
   3.48035411345352

» inv(a1)*b1

ans =

  -2.48449679544607
  -1.57729575627817
  -2.90299977582912
  -1.60498736801348
  -9.46718019044303
```

```
0.07933652351086
3.35772199201199
7.37058528117619
6.07222618703305
3.48035411345350

≫
```

come riprova della bontà dell'algoritmo abbiamo calcolato formalmente il vettore soluzione come $x = A^{-1}b$ ottenendo un risultato pressappoco uguale a quello del nostro metodo.

4.2 Fattorizzazione $PA = LU$

Come già accennato precedentemente, il teorema che stabilisce la condizione necessaria e sufficiente affinchè la fattorizzazione $A = LU$ esista impone un vincolo molto stringente. Cerchiamo allora un metodo per ovviare a questo problema.

4.2.1 Pivoting

Se, per esempio, avessimo che $a_{11}^{(1)} = 0$ con il metodo precedente non potremmo proseguire, ma siamo sicuri che all'interno della stessa colonna esiste un elemento non nullo (altrimenti avremmo una colonna tutta nulla che comporterebbe la singolarità della matrice, in opposizione alle ipotesi). Cerchiamo allora un altro elemento diverso da zero nel seguente modo, limitandoci al caso reale:

$$a_{k1,1}^{(1)} = max_i |a_{i1}^{(1)}|$$

prendiamo cioè l'elemento che ha modulo massimo sulla prima colonna. Individuato il suo indice di riga, scambiamo allora la prima riga con la k1-esima, in modo da ottenere un elemento valido sulla diagonale. Per effettuare questo scambio utiliziamo una matrice di permutazione così costruita:

$$P_1 = \begin{pmatrix} 0 & \cdots & \cdots & 0 & 1 & 0 & \cdots & \cdots & 0 \\ \vdots & 1 & & & 0 & & & & \vdots \\ \vdots & & \ddots & & \vdots & & & & \vdots \\ 0 & & & 1 & \vdots & & & & \vdots \\ 1 & 0 & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & & & & \vdots & 1 & & & \vdots \\ \vdots & & & & \vdots & & \ddots & & \vdots \\ 0 & & & & \vdots & & & \ddots & 0 \\ 1 & \cdots & \cdots & \cdots & 0 & \cdots & \cdots & \cdots & 1 \end{pmatrix}$$

che non è altro che la matrice identità con la prima e la k1-esima riga scambiate. P_1 è una matrice simmetrica ed ortogonale ($P_1 = P_1^T = P_1^{-1}$) che scambia la prima riga con la k1-esima:

$$P_1 \begin{pmatrix} 1 \\ 2 \\ \vdots \\ k1-1 \\ k1 \\ k1+1 \\ \vdots \\ n \end{pmatrix} = \begin{pmatrix} k1 \\ 2 \\ \vdots \\ k1-1 \\ 1 \\ k1+1 \\ \vdots \\ n \end{pmatrix}$$

Dopo aver costruito la matrice di permutazione elementare, possiamo eseguire $P_1 A^{(1)}$ che scambia nella matrice A le righe 1 e $k1$. Ristrutturata in questo modo, la matrice $P_1 A^{(1)}$ possiede tutti i requisiti richiesti dal metodo di eliminazione di Gauss, che dunque possiamo applicare; ottenuta dunque la matrice L_1 come visto per l'algoritmo precedente, portiamo a termine il primo passo ottenendo come risultato:

$$L_1 P_1 A^{(1)} = \begin{pmatrix} a_{k1,1}^{(1)} & a_{k1,2}^{(1)} & \cdots & a_{k1,n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix} \equiv A^{(2)}$$

Un fatto interessante da notare è che, calcolando il vettore elementare di Gauss come già visto, e cioè come

$$g = \frac{1}{a_{k1,1}^{(1)}} (0, a_{21}^{(1)}, \cdots, a_{n1}^{(1)})^T$$

i suoi elementi avranno modulo massimo pari ad uno: infatti abbiamo scelto $a_{k1,1}$ come il massimo della prima colonna, e come favorevole conseguenza abbiamo che gli errori sono ben limitati in aritmetica finita; questa considerazione è valida per la matrice L mentre per U non possiamo dire niente.

Quello che abbiamo descritto ora è il metodo di pivoting parziale, in quanto viene scelto il pivot (l'elemento diagonale utilizzato per costruire il vettore elementare di Gauss) non in modo statico come avveniva per la fattorizzazione $A = LU$ ma viene ricercato all'interno della colonna in esame l'elemento di modulo massimo, e con quello si procede con l'algoritmo di eliminazione di Gauss.

Ottenuta la matrice $A^{(2)}$ possiamo continuare ad applicare l'algoritmo di scambio e di fattorizzazione. Ripetuto questo passo per $n-1$ volte otteniamo

$$L_{n-1}P_{n-1}\cdots L_1P_1A^{(1)} = \begin{pmatrix} a^{(1)}_{k1,1} & \cdots & \cdots & \cdots & \cdots & a^{(1)}_{k1,n} \\ 0 & a^{(2)}_{k2,2} & \cdots & \cdots & \cdots & a^{(2)}_{k2,n} \\ \vdots & \ddots & \ddots & & & \vdots \\ \vdots & & \ddots & a^{(i)}_{ki,i} & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & 0 & a^{(n)}_{kn,n} \end{pmatrix} \equiv U$$

Applicando il pivoting, siamo dunque ancora in grado di eseguire l'eliminazione di Gauss. Se l'elemento di massimo modulo di una colonna avesse valore pari a zero, questo implicherebbe l'esistenza di un blocco diagonale con una colonna nulla e questo implicherebbe la singolarità di A in opposizione alle nostre ipotesi: infatti sia

$$B = \left(\begin{array}{c|c} B_1 & C \\ \hline \mathcal{O} & B_2 \end{array}\right)$$

a blocchi quadrati, allora

$$det(B) = det(B_1)det(B_2)$$

e

$$det(B) \neq 0 \qquad \Rightarrow \qquad \begin{cases} det(B_1) & \neq & 0 \\ det(B_2) & \neq & 0 \end{cases}$$

cioè la non singolarità di B è derivata dalla non singolarità dei suoi blocchi diagonali.

Abbiamo ottenuto la matrice U, ma come possiamo ottenere L? Facciamo un esempio con $n = 4$ tenendo a mente che P_i è simmetrica ed ortogonale:

$$L_3P_3L_2P_2L_1P_1A = U$$

$$L_3P_3L_2\overbrace{P_3P_3}^{I}P_2L_1\overbrace{P_2P_2}^{I}P_1A =$$

$$L_3(P_3L_2P_3)P_3P_2L_1P_2\overbrace{P_3P_3}^{I}P_2P_1A =$$

$$L_3(P_3L_2P_3)(P_3P_2L_1P_2P_3)(P_3P_2P_1)A$$

da cui con opportune sostituzioni si ha

$$\hat{L}_3\hat{L}_2\hat{L}_1PA = U.$$

In generale definiamo

$$P = P_{n-1} \cdots P_1$$

$$\hat{L}_i = P_{n-1} \cdots P_{i+1} L_i P_{i+1} \cdots P_{n-1} \qquad (\hat{L}_{n-1} = L_{n-1})$$

tali che

$$L_{n-1} P_{n-1} \cdots L_2 P_2 L_1 P_1 A = \hat{L}_{n-1} \cdots \hat{L}_1 PA (= U)$$

Se le matrici $\hat{L}_i$ avessero la stessa struttura delle matrici elementari di Gauss avremmo trovato la nostra fattorizzazione poichè potremmo nuovamente definire $L^{-1} = \prod_{i=1}^{n-1} \hat{L}_I$ per poi ottenere la fattorizzazione $PA = LU$; cerchiamo di verificare allora che $\hat{L}_i$ ed L_i hanno la stessa struttura: partendo dal vettore elementare di Gauss

$$g_i = (\underbrace{0 \cdots 0}_{i}, g_{i+1}^{(i)}, \cdots, g_n^{(i)})^T$$

cerchiamo di vedere com'è fatta $\hat{L}_i$

$$\begin{aligned} \hat{L}_i &= P_{n-1} \cdots P_{i+1} (I - g_i e_i^T) P_{i+1} \cdots P_{n-1} = \\ &= I - (P_{n-1} \cdots P_{i+1} g_i)(e_i^T P_{i+1} \cdots P_{n-1}) \end{aligned}$$

prestiamo attenzione alla forma di $e_i^T P_j$ per $j > i$. e_i^T applicato ad una matrice seleziona l'i-esima riga, ma P_{i+1}, ad esempio, ha le prime i righe dell'identità, quindi

$$e_i^T P_j = e_i^T \quad \forall j > i$$

$$e_i^T P_{i+1} \cdots P_{n-1} = e_i^T$$

Ci siamo ridotti a

$$\hat{L}_i = I - (P_{n-1} \cdots P_{i+1} g_i) e_i^T$$

guardiamo adesso alla forma di $P_{i+1} g_i$: P_{i+1} è la matrice elementare di permutazione che scambia $i+1$ con $ki (> i+1)$ che applicata a g_i produce una permutazione degli elementi del vettore dal'indice $i+1$ in poi, e così per tutte le altre matrici P_k; chiamando $\hat{g}_i$ il nuovo vettore, questo ha forma

$$\hat{g}_i = (\underbrace{0 \cdots 0}_{i}, \hat{g}_{i+1}^{(i)}, \cdots, \hat{g}_n^{(i)})^T.$$

L'applicazione delle matrici di permutazione al vettore g_i ha prodotto un vettore con gli elementi $g_{i+1}^{(i)}, \cdots, g_n^{(i)}$ permutati, ma che mantiene ancora la struttura del vettore elementare di Gauss. Unendo i due risultati ottenuti si perviene a

$$\hat{L}_i = I - \hat{g}_i e_i^T$$

proprio una matrice elementare di Gauss, da cui possiamo scrivere:

$$L^{-1}PA = U \qquad \longrightarrow \qquad PA = LU$$

la fattorizzazione che stavamo cercando. Il metodo di pivoting ci consente di applicare l'eliminazione di Gauss anche in caso di elementi diagonali nulli, e quindi con qualsiasi matrice non singolare: basta applicare una matrice di permutazione che la renda conforme alle richieste della fattorizzazione LU.

4.2.2 Implementazione in Matlab

L'algoritmo che implementa il metodo del pivoting parziale per fattorizzare la matrice come $PA = LU$.

```
function [A,P]=fattPALU(A)
%FATTPALU Fattorizza la matrice A come LU tramite il metodo di
%   eliminazione di Gauss applicando il pivoting
%
%   [A,P]=FATTPALU(A)
%
%   I parametri della funzione sono:
%       A -> la matrice quadrata da fattorizzare
%
%   I valori di ritorno sono:
%       A -> la matrice modificata contenente nella parte
%            triangolare superiore U e nella parte strettamente
%            triangolare inferiore L
%       P -> vettore di permutazione
%
%   See Also SOLVEPALU, FATTLU, FATTLDLT, FATTQR
  n=length(A);
  P=1:n;
  for i=1:n-1
     [m,ki]=max(abs(A(i:n,i)));
     if m==0
        disp('Matrice non fattorizzabile perché singolare')
        return
     end;
     ki=ki+i-1;
     if ki>i
        P([i,ki])=P([ki,i]);
        A([i,ki],:)=A([ki,i],:);
     end
     A(i+1:n,i)=A(i+1:n,i)/A(i,i);
     A(i+1:n,i+1:n)=A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
```

```
end
```

Invece di utilizzare una matrice di permutazione è stato scelto di utilizzare un vettore di permutazione che contiene al suo interno gli scambi effettuati durante l'algoritmo perché possano poi essere applicati al vettore dei termini noti.

Per quanto riguarda il costo computazionale, oltre a quanto già visto per $A = LU$, il pivoting parziale introduce un costo dovuto ai confronti che sono $n - i$ al passo i-esimo, perciò

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx n^2$$

4.2.3 Sperimentazioni dell'algoritmo

In questo caso non possiamo generare le due matrici L ed U, moltiplicarle ed applicare l'algoritmo sul loro prodotto, perché il riordinamento delle righe dalla matrice da fattorizzare sicuramente non produrrebbe come risultato le due matrici di partenza. Abbiamo perciò condotto la prova solo su una matrice casuale non fattorizzabile LU:

```
» a1=round(10*rand(10))

a1 =

     0     5     6     6     5     1     4     1     3    10
     2     8     1     9     9     9     9     8     9     1
     9     1    10     5     2     5     8     1     8     8
    10     5     8     0     5     9    10     1     1     4
     9     7     9     5     6     7     3     5     6     7
     1     9     2     5     4     1     7     4     4     1
     2     6     7     7     4     7     4     9     9     2
    10    10     3     5     3     5     1     7    10     4
     5     5     6     1     4     7     6     8     3     8
     2     8     2     5     5     9     4     3     8    10

» fattlu(a1)

Non è possibile fattorizzare la matrice come LU

» A=fattpalu(a1)

A =
```

```
Columns 1 through 4

10.00000000000000   5.00000000000000   8.00000000000000                   0
 0.10000000000000   8.50000000000000   1.20000000000000   5.00000000000000
 1.00000000000000   0.58823529411765  -5.70588235294118   2.05882352941176
 0.90000000000000  -0.41176470588235  -0.57731958762887   8.24742268041237
 0.20000000000000   0.82352941176471   0.27835051546392   0.52250000000000
 0.20000000000000   0.58823529411765  -0.82268041237113   0.69750000000000
 0.90000000000000   0.29411764705882  -0.25360824742268   0.49125000000000
 0.20000000000000   0.82352941176471   0.10309278350515   0.08125000000000
 0.50000000000000   0.29411764705882  -0.28865979381443   0.01500000000000
                0   0.58823529411765  -0.92783505154639   0.60250000000000

Columns 5 through 8

 5.00000000000000   9.00000000000000  10.00000000000000   1.00000000000000
 3.50000000000000   0.10000000000000   6.00000000000000   3.90000000000000
-4.05882352941176  -4.05882352941176 -12.52941176470588   3.70588235294118
-3.40206185567010  -5.40206185567010  -5.76288659793814   3.84536082474227
 8.02500000000000  11.07000000000000   8.55750000000000   1.54750000000000
-0.00311526479751   5.60448598130841  -7.79084112149533   6.87732087227415
 0.13862928348910  -0.18549893276414 -10.74276280683031   3.06495686588403
 0.22585669781931   0.97685432230523  -0.41856068249765  -6.89103342222609
-0.08099688473520   0.40621664887940   0.04068783742250  -0.66346419968185
 0.15264797507788  -0.22478655282818   0.99974853389049   0.27963579696932

Columns 9 through 10

 1.00000000000000   4.00000000000000
 3.90000000000000   0.60000000000000
 6.70588235294118  -0.35294117647059
12.57731958762886   4.44329896907217
-2.85000000000000  -2.51750000000000
 3.24112149532710  -2.55034267912773
 0.47131803628602   0.82716337602277
 0.54986324589970  11.78737557343069
 1.89819903280251  14.27392821149857
-0.05866281861715   3.16772986439264

≫
```

4.2.4 Risoluzione di $Ax = b$ tramite fattorizzazione $PA = LU$

Anche in questo caso dobbiamo ritornare al problema di partenza, la risoluzione del sistema lineare $Ax = b$: fattorizzando la matrice tramite pivoting possiamo risolvere il problema iniziale in questo modo:

$$\begin{array}{l} Ax = b \\ PAx = Pb = \hat{b} \\ LUx = \hat{b} \end{array} \quad \Rightarrow \quad \begin{cases} Ly = \hat{b} \\ \\ Ux = y \end{cases}$$

4.2.5 Implementazione in Matlab per risolvere $LUx = Pb$

Il codice che risolve il sistema lineare $Ax = b$ tramite fattorizzazione con pivoting parziale:

```
function x=solvePALU(A,b)
%SOLVEPALU Risolve il sistema lineare Ax=b fattorizzando la matrice
%   A come LU applicando il pivoting ed infine risolvendo i sistemi
%                Ly=b
%                Ux=y
%
%   x=SOLVEPALU(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
%
%   See Also FATTPALU
  [A,p]=fattPALU(A);
  n=length(b);
  x1=b(p);
  for i=2:n
     for j=1:i-1
         x1(i)=x1(i)-A(i,j)*x1(j);
     end
  end
  x=solveUT(A,x1);
```

4.2.6 Sperimentazioni dell'algoritmo

Anche in questo caso abbiamo generato casualmente una matrice ed un vettore ($n = 10$) e su questi abbiamo applicato l'algoritmo di risoluzione del sistema lineare $Ax = b$ tramite fattorizzazione $PA = LU$:

```
» a1=round(10*rand(10))

a1 =

     0     4     3     4     2     4     5     5     1     4
    10     5     3     3     2     2     5     8     1     1
     7     0     8     0     2     2     3     1     5     9
     3     9     1     3     7     9     4     3     8     2
     2     4     9     1     8     1    10     6     6     5
     5     2     6     9     8     0     4     2     1    10
     5     6     1     6     5     1    10     9     3     5
     9     3     1     7    10     5     6     1     4     6
     4     9     5     4     4     1     6     9     6     4
     4     6     4     1     6     3     2    10     8     3

» b1=round(10*rand(10,1))

b1 =

     4
     0
     1
     9
     4
     4
     9
     7
     5
     8

» x=solvepalu(a1,b1)

x =

  -0.26198945735416
   1.00112154037766
  -1.40256923272077
  -1.54752006386472
   1.10934597753791
   0.08535565618920
  -0.08686034535773
   0.44663778751295
  -0.76296543964648
   1.69931636028708
```

```
» inv(a1)*b1

ans =

  -0.26198945735416
   1.00112154037766
  -1.40256923272077
  -1.54752006386472
   1.10934597753791
   0.08535565618920
  -0.08686034535773
   0.44663778751295
  -0.76296543964648
   1.69931636028708

»
```

e per riprova abbiamo anche eseguito $A^{-1}b$ per verificare la bontà del metodo.

4.3 Matrici sicuramente fattorizzabili $A = LU$

Esistono alcuni tipi di matrici per cui la fattorizzazione LU è definita senza bisogno di ricorrere al pivoting, e queste sono:

1. A è a diagonale dominante (per righe).

$$\forall i = 1, \cdots, n \quad |a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|$$

   (in modo analogo per le colonne)

2. A è simmetrica (hermitiana se definita in $\mathbb{C}$) e definita positiva

$$\begin{array}{l} A = A^T \quad \text{per simmetria} \\ \forall x \neq \underline{0} \in \mathbb{R}^n \ (\mathbb{C}^n) \quad \Rightarrow \quad x^T A x \ (x^* A x) > 0 \end{array}$$

   La simmetria riduce di un fattore $\frac{1}{2}$ le operazioni di fattorizzazione, in quanto conoscendo metà matrice è comunque nota tutta A.

4.3.1 Matrici a diagonale dominante

In caso di matrici di questo tipo abbiamo la proprietà che ogni sottomatrice di ordine k, A_k, è a diagonale dominante: infatti

$$\forall i = 1, \cdots, n \quad |a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{k(\leq n)} |a_{ij}|$$

e quindi A_k è a diagonale dominante.

Abbiamo dimostrato che se A è a diagonale dominante, allora tali sono anche le sue sottomatrici principali; per dimostrare che A è fattorizzabile LU quello che dobbiamo dimostrare è che A è non singolare, e cioè che i suoi minori principali sono non nulli. Dimostriamo dunque che se A è a diagonale dominante, allora A è non singolare.

Per assurdo supponiamo A essere a diagonale dominante e singolare, allora esiste $x \neq \underline{0}$ tale che $Ax = \underline{0}$ e questo vettore è definito a meno di una costante. Assumiamo allora

$$|x_i| = max_j |x_j| = 1$$

altrimenti potremmo dividere x per la sua norma infinito, che è proprio definita come il massimo delle componenti di x. Allora prendiamo la i-esima componente dei due termini moltiplicandoli per e_i; l'indice i è quello in corrispondenza dell'elemento massimo di x:

$$\begin{aligned} e_i A x &= e_i \underline{0} = 0 \\ (e_i A) x &= (a_{i1}, \cdots, a_{in}) \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \sum_{j=1}^{k} a_{ij} x_j \\ \sum_{j=1}^{k} a_{ij} x_j &= 0 \qquad |x_j| \leq |x_i| = 1 \\ a_{ii} x_i &= -\sum_{j \neq i} a_{ij} x_j \end{aligned}$$

possiamo dunque scrivere

$$\begin{aligned} |a_{ii}| &= |a_{ii} x_i| = \left| -\sum_{j \neq i} a_{ij} x_j \right| \leq \sum_{j \neq i} |a_{ij} x_j| = \\ &= \sum_{j \neq i} |a_{ij}||x_j| \leq \sum_{j \neq i} |a_{ij}| \end{aligned}$$

avendo maggiorato $|x_j|$ con 1; ma allora quello che abbiamo ottenuto è che

$$|a_{ii}| \leq \sum_{j \neq i} |a_{ij}|$$

cioè non è a diagonale dominante, un assurdo in contrapposizione alle nostre ipotesi.

Dunque se una matrice è a diagonale dominante è non singolare ed allora è fattorizzabile LU come si voleva dimostrare.

4.3.2 Matrici simmetriche e definite positive

Sia A una matrice simmetrica e definita positiva, allora verifica:

- $A = A^T$ per simmetria;

- $\forall x \neq \underline{0} \in \mathbb{R}^n \Rightarrow x^T A x > 0$.

la matrice A ha gli elementi diagonali diversi da zero: infatti scegliendo $x = e_i$, che è diverso dal vettore nullo, allora $e_i^T A e_i = a_{ii} > 0$

Se A è simmetrica e definita positiva, allora tali saranno le sue sottomatrici principali A_k: la simmetria discende immediatamente dalla simmetria di A, ma A_k è anche definita positiva? Cioè, $\forall y \neq \underline{0} \in \mathbb{R}^k \; y^T A_k y > 0$ è verificato? Prendiamo un vettore x così costruito:

$$x = \begin{pmatrix} y \\ \underline{0} \end{pmatrix} \in \mathbb{R}^n \wedge x \neq \underline{0}$$

per ipotesi abbiamo che $x^T A x > 0$ allora scomponendo la matrice a blocchi

$$A = \left(\begin{array}{c|c} A_k & B \\ \hline B^T & D \end{array}\right)$$

si può scrivere

$$(y \; \underline{0}) \left(\begin{array}{c|c} A_k & B \\ \hline B^T & D \end{array}\right) \begin{pmatrix} y \\ \underline{0} \end{pmatrix} = (y \; \underline{0}) \begin{pmatrix} A_k y \\ B^T y \end{pmatrix} = y^T A_k y > 0$$

Allora se A è simmetria e definita positiva, A è anche non singolare; infatti se avessimo, per assurdo, A singolare, questo implicherebbe che $\exists x \neq \underline{0} : \quad Ax = \underline{0}$ e quindi

$$0 < x^T A x = x^T \underline{0} = 0$$

un assurdo evidentemente.

4.3.3 Ottenere matrici simmetriche e definite positive

Vediamo come si può ottenere una matrice simmetrica e definita positiva: sia B una matrice non singolare, allora $A = BB^T$ è una matrice simmetrica e definita positiva. Infatti BB^T è simmetrica per costruzione e preso $v \neq \underline{0}$:

$$v^T A v = v^T B B^T v = (v^T B)(B^T v)$$

detto ora $w = B^T v$ abbiamo che $w \neq \underline{0}$ ed inoltre $w^T = v^T B$ possiamo allora scrivere

$$(v^T B)(B^T v) = w^T w = \|w\|_2^2 > 0$$

poiche il vettore w è non-nullo, ed è la condizione che cercavamo.

Per le matrici simmetriche e definite positive esiste anche una fattorizzazione, che non vedremo, come RR^T dove con R indichiamo una matrice triangolare inferiore; questa fattorizzazione ci sarà utile per la costruzione di una matrice simmetrica e definita positiva da utilizzare per le sperimentazioni del prossimo metodo.

4.4 Fattorizzazione $A = LDL^T$

Sappiamo che se A è una matrice simmetrica e definita positiva allora A è fattorizzabile LU. Ciò che ci proponiamo adesso è scrivere U come $D\hat{U}$ con D matrice diagonale

$$D = \begin{pmatrix} u_{11} & & 0 \\ & \ddots & \\ 0 & & u_{nn} \end{pmatrix} \qquad d^i = u_{ii} e_i^T$$

e quindi $\hat{U}$ risulta essere la matrice U di partenza scalata per gli elementi $u_{11}, u_{22}, \cdots$, dove per scalata intendiamo che i suoi fattori di scala sono gli u_{ii}. Posto $U = D\hat{U}$, per la prima riga si ha

$$\begin{aligned} (u_{11} \cdots u_{1n}) &= u_{11}(e_1^T \hat{U}) = \\ &= u_{11}(\hat{u}_{11} \cdots \hat{u}_{1n}) \end{aligned}$$

ed uguagliando componente per componente otteniamo

$$\begin{aligned} \hat{u}_{11} &= 1 \\ \hat{u}_{1j} &= \tfrac{u_{1j}}{u_{11}} \end{aligned} \quad .$$

Questo risultato è valido per ogni riga, perciò la matrice $\hat{U}$ avrà elementi diagonali pari ad uno

$$\hat{U} = \begin{pmatrix} 1 & & * \\ & \ddots & \\ 0 & & 1 \end{pmatrix}$$

ed è una matrice triangolare superiore a diagonale unitaria. L'operazione appena eseguita è lecita solo se $u_{ii} \neq 0$, ma ciò è garantito dalla possibilità di fattorizzazione LU.

$$A = LU = LD\hat{U}$$

ricordandosi che A è simmetrica

$$A = A^T = (LD\hat{U})^T = \hat{U}^T DL^T$$

ma DL^T che forma ha? E' una matrice triangolare superiore con diagonale u_{ii}. Unendo le due espressioni sopra otteniamo

$$LD\hat{U} = \hat{U}^T DL^T \qquad \Longleftrightarrow \qquad L = \hat{U}^T, \quad \hat{U} = L^T.$$

Perciò se A è simmetrica definita positiva conviene cercare una fattorizzazione nella forma LDL^T; la matrice U non ci serve, necessitiamo solo della sua diagonale e di L.

Quello che ci proponiamo di fare è riorganizzare i passi della fattorizzazione LU senza calcolare U e supporremo di avere a disposizione solo la porzione triangolare inferiore di A, in modo che

$$A = (a_{ij}) \quad \text{noti per } j \leq i.$$

Sapendo che $j \leq i$

$$a_{ij} = e_i^T A e_j$$

e dal momento che cerchiamo la fattorizzazione LDL^T, che sappiamo esistere, possiamo scrivere:

$$\begin{aligned}
a_{ij} &= e_i^T LDL^T e_j = (e_i^T L) D (L^T e_j) \\
a_{ij} &= (l_{i1} \cdots l_{ii} 0 \cdots 0) \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix} \begin{pmatrix} l_{j1} \\ \vdots \\ l_{jj} \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \\
&= (l_{i1} \cdots l_{ii} 0 \cdots 0) \begin{pmatrix} l_{j1} d_1 \\ \vdots \\ l_{jj} d_j \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \sum_{k=1}^{min(i,j)=j} l_{ik} l_{jk} d_k \\
a_{ij} &= \sum_{k=1}^{j} l_{ik} l_{jk} d_k
\end{aligned}$$

$$\forall j \leq i \quad a_{ij} = \sum_{k=1}^{j} l_{ik} l_{jk} d_k = \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k + l_{ij} \underbrace{l_{jj}}_{1} d_j$$

Si presentano dunque due casi:

$$i = j \rightarrow d_j = a_{jj} - \sum_{k=1}^{j-1} (l_{jk})^2 d_k \qquad (l_{jj} = 1)$$

$$i = j+1 \cdots n \rightarrow l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k}{d_j} \qquad d_j \neq 0$$

Quindi, durante il primo passo viene calcolato d_1 e poi tutta la prima colonna di L, e poi l'algoritmo procede nello stesso modo.

4.4.1 Implementazione in Matlab

Di seguito proponiamo il codice che fattorizza la matrice A come LDL^T.

```
function A=fattLDLT(A)
%FATTLDLT Fattorizza la matrice A come LDLT
%
%   A=FATTLDLT(A)
%
%   I parametri della funzione sono:
%       A -> la matrice quadrata da fattorizzare
%
%   I valori di ritorno sono:
%       A -> la matrice modificata contenente nella parte
%            strettamente triangolare inferiore L, sulla
%            diagonale D e nella parte strettamente superiore
%            la parte strettamente superiore di A
%
%   See Also SOLVELDLT, FATTLU, FATTPALU, FATTQR
  n=length(A);
  for j=1:n
     if j==1
        v=A(j,1:j-1)';
     else
(1)     v=(A(j,1:j-1).*diag(A(1:j-1,1:j-1))')';
     end
(2)  A(j,j)=A(j,j)-A(j,1:j-1)*v;
(3)  A(j+1:n,j)=(A(j+1:n,j)-A(j+1:n,1:j-1)*v)/A(j,j);
  end
```

4.4.2 Analisi del codice e costo computazionale

Cerchiamo adesso di dare una spiegazione del codice appena visto. Come si vede dall'introduzione teorica, durante il passo j-esimo sia per il calcolo di d_j che degli l_{ij} $(i > j)$ viene utilizzata la quantità $l_{jk}d_k$ all'interno di una sommatoria per $k = 1, \cdots, j-1$; risulta perciò utile memorizzare questi valori in una forma comoda per il calcolo della sommatoria, e questa forma ci viene fornita dall'algebra lineare: se si memorizzasse, per esempio per il calcolo di d_j, gli elementi l_{jk} in un vettore riga e il fattore $l_{jk}d_k$ in un vettore colonna (chiamato da qui in avanti v), e se ne facesse il prodotto scalare quello che si otterrebbe sarebbe proprio la sommatoria da sottrarre ad a_{ij}:

$$d_j = a_{jj} - \sum_{k=1}^{j-1} (l_{jk})(l_{jk}d_k)$$

$$(l_{j1}, \cdots, l_{jk}) \begin{pmatrix} l_{j1}d_1 \\ \vdots \\ l_{jk}d_k \end{pmatrix} = l_{j1}l_{j1}d_1 + \cdots + l_{jk}l_{jk}d_k = \sum_{k=1}^{j-1} (l_{jk})^2 d_k$$

proprio quello che stavamo cercando. Ma come costruire v? Gli elementi l_{jk} sono gli elementi a_{jk} e quindi $A(j, 1 : j-1)$, mentre gli elementi d_k sono la diagonale della matrice $A(1 : j-1, 1 : j-1)$, facendo il prodotto di queste due quantità si ottiene un vettore riga, facendone il trasposto otteniamo il vettore che cercavamo; quanto descritto viene eseguito nella riga (1). Ora che abbiamo ottenuto il vettore v possiamo calcolare l'elemento d_j tramite la riga (2):

$$d_j \equiv A(j,j) = A(j,j) - \underbrace{A(j, 1 : j-1) * v}_{\sum_{k=1}^{j-1} (l_{jk})^2 d_k}.$$

Per il calcolo della j-esima colonna di L seguiamo le indicazioni che ci vengono della teoria che tradotte in codice portano alla riga (3):

$$A(j+1 : n, j) = (\underbrace{A(j+1 : n, j)}_{a_{ij}} - \overbrace{\underbrace{A(j+1 : n, 1 : j-1)}_{l_{ik}} * v}^{\sum_{k=1}^{j-1} l_{ik}l_{jk}d_k}) / \underbrace{A(j,j)}_{d_j}$$

Costo computazionale

Vediamo adesso il costo di questo algoritmo, analizzando le righe significative per la complessità:

(1) viene eseguito un prodotto scalare di dimensione $j-1$, quindi $2(j-1)$ flops;

(2) si esegue una somma ed un prodotto scalare di dimensione $j-1$, quindi ancora $2(j-1)$ flops;

(3) vi sono una somma ed una divisione, due operazioni trascurabili, ma viene anche eseguito un prodotto matrice-vettore di dimensioni $n-j \times j-1$, quindi $2(n-j)(j-1)$.

Sommando il costo del prodotto matrice-vettore (che è predominante sul costo dei prodotti scalari) su j si perviene a:

$$\begin{aligned} 2\sum_{j=1}^{n}(n-j)(j-1) &= 2\sum_{j=1}^{n}(nj-j^2-n+j) \approx \\ &\approx 2\sum_{j=1}^{n} nj - 2\sum_{j=1}^{n} j^2 = 2n\sum_{j=1}^{n} j - 2\sum_{j=1}^{n} j^2 = \\ &= 2n\frac{n(n+1)}{2} - 2\left(\frac{n^3}{3}+\frac{n^2}{2}+\frac{n}{6}\right) \approx \\ &\approx n^3 - \frac{2n^3}{3} = \frac{n^3}{3} \\ 2\sum_{j=1}^{n}(n-j)(j-1) &\approx \frac{n^3}{3} \end{aligned}$$

un costo proporzionale al cubo della dimensione significativa, un costo sicuramente importante ma la metà di quello necessario alla fattorizzazione LU: il fatto di avere una matrice simmetrica ci consente di ridurre di un fattore $\frac{1}{2}$ il costo della fattorizzazione.

4.4.3 Sperimentazioni dell'algoritmo

Poichè una matrice simmetrica e definita positiva si può fattorizzare come RR^T con R matrice triangolare inferiore, allora per generare le nostre matrici di prova, utilizzeremo proprio questa proprietà

```
» r=tril(round(5*rand(10)))

r =

     5     0     0     0     0     0     0     0     0     0
     4     2     0     0     0     0     0     0     0     0
     2     1     1     0     0     0     0     0     0     0
     1     1     2     2     0     0     0     0     0     0
     5     3     2     5     4     0     0     0     0     0
```

```
     5     3     0     1     1     2     0     0     0     0
     4     1     4     0     0     1     2     0     0     0
     3     4     4     4     4     0     0     2     0     0
     3     1     2     5     3     5     2     1     1     0
     1     4     4     2     2     3     2     3     4     4

» a=r*r'

a =

    25    20    10     5    25    25    20    15    15     5
    20    20    10     6    26    26    18    20    14    12
    10    10     6     5    15    13    13    14     9    10
     5     6     5    10    22    10    13    23    18    17
    25    26    15    22    79    43    31    71    59    43
    25    26    13    10    43    40    25    35    36    27
    20    18    13    13    31    25    38    32    30    31
    15    20    14    23    71    35    32    77    55    57
    15    14     9    18    59    36    30    55    79    57
     5    12    10    17    43    27    31    57    57    95

» A=fattldlt(a)

A =

  Columns 1 through 4

  25.00000000000000  20.00000000000000  10.00000000000000   5.00000000000000
   0.80000000000000   4.00000000000000  10.00000000000000   6.00000000000000
   0.40000000000000   0.50000000000000   1.00000000000000   5.00000000000000
   0.20000000000000   0.50000000000000   2.00000000000000   4.00000000000000
   1.00000000000000   1.50000000000000   2.00000000000000   2.50000000000000
   1.00000000000000   1.50000000000000                  0   0.50000000000000
   0.80000000000000   0.50000000000000   4.00000000000000                  0
   0.60000000000000   2.00000000000000   4.00000000000000   2.00000000000000
   0.60000000000000   0.50000000000000   2.00000000000000   2.50000000000000
   0.20000000000000   2.00000000000000   4.00000000000000   1.00000000000000

  Columns 5 through 8

  25.00000000000000  25.00000000000000  20.00000000000000  15.00000000000000
  26.00000000000000  26.00000000000000  18.00000000000000  20.00000000000000
  15.00000000000000  13.00000000000000  13.00000000000000  14.00000000000000
  22.00000000000000  10.00000000000000  13.00000000000000  23.00000000000000
```

```
  16.00000000000000  43.00000000000000  31.00000000000000  71.00000000000000
   0.25000000000000   4.00000000000000  25.00000000000000  35.00000000000000
                  0   0.50000000000000   4.00000000000000  32.00000000000000
   1.00000000000000                   0                  0   4.00000000000000
   0.75000000000000   2.50000000000000   1.00000000000000   0.50000000000000
   0.50000000000000   1.50000000000000   1.00000000000000   1.50000000000000

  Columns 9 through 10

  15.00000000000000   5.00000000000000
  14.00000000000000  12.00000000000000
   9.00000000000000  10.00000000000000
  18.00000000000000  17.00000000000000
  59.00000000000000  43.00000000000000
  36.00000000000000  27.00000000000000
  30.00000000000000  31.00000000000000
  55.00000000000000  57.00000000000000
   1.00000000000000  57.00000000000000
   4.00000000000000  16.00000000000000

≫ l=tril(A,-1)+eye(10)

l =

  Columns 1 through 4

   1.00000000000000                   0                  0                   0
   0.80000000000000   1.00000000000000                  0                   0
   0.40000000000000   0.50000000000000   1.00000000000000                   0
   0.20000000000000   0.50000000000000   2.00000000000000   1.00000000000000
   1.00000000000000   1.50000000000000   2.00000000000000   2.50000000000000
   1.00000000000000   1.50000000000000                  0   0.50000000000000
   0.80000000000000   0.50000000000000   4.00000000000000                   0
   0.60000000000000   2.00000000000000   4.00000000000000   2.00000000000000
   0.60000000000000   0.50000000000000   2.00000000000000   2.50000000000000
   0.20000000000000   2.00000000000000   4.00000000000000   1.00000000000000

  Columns 5 through 8

                  0                   0                  0                   0
                  0                   0                  0                   0
                  0                   0                  0                   0
                  0                   0                  0                   0
   1.00000000000000                   0                  0                   0
```

```
   0.25000000000000   1.00000000000000                   0                   0
                   0   0.50000000000000   1.00000000000000                   0
   1.00000000000000                   0                   0   1.00000000000000
   0.75000000000000   2.50000000000000   1.00000000000000   0.50000000000000
   0.50000000000000   1.50000000000000   1.00000000000000   1.50000000000000

  Columns 9 through 10

                   0                   0
                   0                   0
                   0                   0
                   0                   0
                   0                   0
                   0                   0
                   0                   0
                   0                   0
   1.00000000000000                   0
   4.00000000000000   1.00000000000000

≫ d=diag(A)

d =

    25
     4
     1
     4
    16
     4
     4
     4
     1
    16

≫ l*diag(d)*l'

ans =

    25    20    10     5    25    25    20    15    15     5
    20    20    10     6    26    26    18    20    14    12
    10    10     6     5    15    13    13    14     9    10
     5     6     5    10    22    10    13    23    18    17
    25    26    15    22    79    43    31    71    59    43
    25    26    13    10    43    40    25    35    36    27
```

```
    20    18    13    13    31    25    38    32    30    31
    15    20    14    23    71    35    32    77    55    57
    15    14     9    18    59    36    30    55    79    57
     5    12    10    17    43    27    31    57    57    95

≫ a

a =

    25    20    10     5    25    25    20    15    15     5
    20    20    10     6    26    26    18    20    14    12
    10    10     6     5    15    13    13    14     9    10
     5     6     5    10    22    10    13    23    18    17
    25    26    15    22    79    43    31    71    59    43
    25    26    13    10    43    40    25    35    36    27
    20    18    13    13    31    25    38    32    30    31
    15    20    14    23    71    35    32    77    55    57
    15    14     9    18    59    36    30    55    79    57
     5    12    10    17    43    27    31    57    57    95

≫
```

esattamente la matrice di partenza.

4.4.4 Risoluzione di $Ax = b$ con fattorizzazione $A = LDL^T$

Come sempre questa fattorizzazione ci deve consentire di risolvere il sistema lineare, problema di partenza, $Ax = b$; sfruttando il fatto che $A = LDL^T$ possiamo semplificare il nostro problema risolvendo:

$$\begin{array}{l} Ax = b \\ LDL^T x = b \end{array} \quad \Rightarrow \quad \left\{ \begin{array}{l} Ly = b \\ Dz = y \\ L^T x = z \end{array} \right.$$

4.4.5 Implementazione in Matlab per risolvere $LDL^T x = b$

Di seguito presentiamo il codice che risolve il sistema lineare $Ax = b$ fattorizzando A come LDL^T:

```
function x=solveLDLT(A,b)
%SOLVELDLT Risolve il sistema lineare Ax=b fattorizzando la matrice
%   A come LDLT ed infine risolvendo i sistemi
%                Ly=b
%                Dz=y
%                LTx=z
```

```
%
%   x=SOLVELDLT(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
%
%   See Also FATTLDLT
  n=length(b);
  A=fattLDLT(A);
  % Ly=b
  y=b;
  for i=2:n
     for j=1:i-1
         y(i)=y(i)-A(i,j)*y(j);
     end
  end
  % Dz=y
  z=y ./ diag(A);
  % LTx=z
  x=z;
  for i=n:-1:1
     for j=1:i-1
         x(j)=x(j)-A(i,j)*x(i);
     end
  end
```

4.4.6 Sperimentazioni dell'algoritmo

Generiamo una matrice simmetrica definita positiva come visto in precedenza ed un vettore anch'esso casuale per poi risolvere il sistema lineare ad essi associato:

```
» r=tril(round(5*rand(10)))

r =

     3     0     0     0     0     0     0     0     0     0
     3     5     0     0     0     0     0     0     0     0
     3     1     3     0     0     0     0     0     0     0
     3     3     2     2     0     0     0     0     0     0
```

```
     3     4     2     3     5     0     0     0     0     0
     3     1     3     2     5     2     0     0     0     0
     3     1     3     4     3     4     4     0     0     0
     2     4     5     3     1     3     4     4     0     0
     1     1     2     5     1     4     1     3     4     0
     5     3     5     1     3     4     1     4     4     5

» a=r*r'

a =

     9     9     9     9     9     9     9     6     3    15
     9    34    14    24    29    14    14    26     8    30
     9    14    19    18    19    19    19    25    10    33
     9    24    18    26    31    22    26    34    20    36
     9    29    19    31    63    50    46    46    31    55
     9    14    19    22    50    52    50    42    33    58
     9    14    19    26    46    50    76    68    53    66
     6    26    25    34    46    42    68    96    60    85
     3     8    10    20    31    33    53    60    74    71
    15    30    33    36    55    58    66    85    71   143

» b=round(5*rand(10,1))

b =

     1
     4
     1
     0
     4
     1
     1
     4
     5
     3

» x=solveldlt(a,b)

x =

   0.07316284722222
   1.00544262152778
   4.15498611111111
```

```
  -5.06072526041667
   2.52293489583333
  -2.57046093750000
   0.55198958333333
  -0.32498437500000
   1.13964583333333
  -0.43766666666667

≫ inv(a)*b

ans =

   0.07316284722222
   1.00544262152778
   4.15498611111112
  -5.06072526041668
   2.52293489583334
  -2.57046093750001
   0.55198958333334
  -0.32498437500000
   1.13964583333334
  -0.43766666666667

≫
```

e come al solito abbiamo confrontato il risultato da noi ottenuto con la definizione formale di soluzione di un sistema lineare.

4.5 Fattorizzazione $A = QR$

Quello che facciamo adesso è cercare una fattorizzazione di A come un fattore ortogonale ed un fattore triangolare superiore.

Data $A \in \mathbb{R}^{m \times n}$ $m \geq n$ (in generale ha più righe che colonne) e richiediamo che abbia rango massimo, cioè $rank(A) = n$ (e quindi se la matrice è quadrata, sarà anche non singolare). Allora ciò che cerchiamo è

$$A = QR \quad Q \in \mathbb{R}^{m \times m},\ R \in \mathbb{R}^{m \times n}$$

$$R = \left(\frac{R_1}{\mathcal{O}} \right) \quad R_1 \in \mathbb{R}^{n \times n} \text{ triangolare superiore}$$

Come osservazione possiamo notare che se $m = n$, R è triangolare superiore ed $R = R_1$.

Mostreremo l'esistenza di questa fattorizzazione mediante una dimostrazione costruttiva che ci guiderà nella costruzione dell'algoritmo.

Dalle proprietà del rango abbiamo che

$$n = rank(A) = rank(QR) = rank(R)$$

visto che Q ha rango massimo; R ha rango massimo, e per costruzione

$$rank(R) = rank(R_1)$$

ed essendo $R_1 \in \mathbb{R}^{n\times n}$ questo implica che R_1 è non singolare.

Dunque, come già visto durante l'algoritmo di eliminazione di Gauss, dato un vettore x cerchiamo una matrice ortogonale $P \in \mathbb{R}^{n\times n}$ tale che:

$$Px = \begin{pmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \alpha e_1,$$

che azzeri tutti gli elementi di x da un certo punto in poi. Inoltre, prendendo il quadrato della norma euclidea di Px si ottiene:

$$\begin{aligned} &x^T P^T P x = x^T x = \|x\|_2^2 \\ &\text{con } Px = \alpha e_1 \\ &x^T P^T P x = \alpha^2 e_1^T e_1 = \alpha^2 \|e_1\|_2^2 = \alpha^2 \end{aligned}$$

da cui otteniamo $\alpha = \pm\|x\|_2$.

Cerchiamo dunque una matrice P nella forma

$$P = I - 2\frac{vv^T}{v^T v} \qquad v \in \mathbb{R}^n \text{ assegnato}$$

con v vettore da determinare; notiamo inoltre che $\frac{2}{v^T v}$ è uno scalare. P è una matrice simmetrica, perché sia I che vv^T lo sono ($vv^T = (vv^T)^T = (v^T)^T v^T = vv^T$) ed è detta matrice di Householder; sapendo che $P = P^T$ verifichiamo l'ortogonalità di P:

$$\begin{aligned} P^2 &= P^T P = I \\ &= \left(I - 2\frac{vv^T}{v^T v}\right)\left(I - 2\frac{vv^T}{v^T v}\right) = \\ &= I - 4\frac{vv^T}{v^T v} + 4\frac{v(v^T v)v^T}{(v^T v)^2} = I - 4\frac{vv^T}{v^T v} + 4\frac{vv^T}{v^T v} = I \end{aligned}$$

Come per la matrice elementare di Gauss, anche P è ottenuta come la matrice identità a cui viene sommata una matrice di correzione di rango 1, e sommare matrici di rango basso significa eseguire pochi conti.

Eravamo rimasti che il vettore v doveva essere determinato, verifichiamo allora che

$$v = x - \alpha e_1$$

è proprio ciò di cui abbiamo bisogno, cioè che:

$$\begin{aligned}
\left(I - 2\frac{vv^T}{v^Tv}\right)x &= \alpha e_1 \\
\left(I - 2\frac{vv^T}{v^Tv}\right)x &= x - 2\frac{v^Tx}{v^Tv}v = x - 2\frac{v^Tx}{v^Tv}(x - \alpha e_1) = \\
&\qquad (v^Tx) \text{ è uno scalare, quindi } (v^Tx)v = v(v^Tx) \\
&= x\left(1 - 2\frac{v^Tx}{v^Tv}\right) + \alpha 2\frac{v^Tx}{v^Tv}e_1.
\end{aligned}$$

Se si riuscisse a dimostrare che il coefficiente del vettore x è pari a zero, e quello del vettore e_1 è pari ad α avremmo terminato; per fare questo bisogna dimostrare che

$$2\frac{v^Tx}{v^Tv} = 1 \qquad 2v^Tx = v^Tv$$

$$\begin{aligned}
2v^Tx &= \left(\begin{array}{l} v = x - \alpha e_1 \\ \\ \alpha = \pm\|x\|_2 \Leftrightarrow \alpha^2 = x^Tx \end{array} \right) = 2(x - \alpha e_1)^Tx = \\
&= 2x^Tx - 2\alpha x_1 = 2\alpha^2 - 2\alpha x_1 \\
v^Tv &= (x - \alpha e_1)^T(x - \alpha e_1) = x^Tx - 2\alpha x_1 + \alpha^2 \overbrace{e_1^Te_1}^{1} = \\
&= 2\alpha^2 - 2\alpha x_1
\end{aligned}$$

Le due quantità sono uguali e quindi abbiamo verificato che una matrice P così costruita soddisfa le nostre richieste; il vettore v visto è detto vettore di Householder.

La scelta del segno di α, cioè se scegliere $\alpha = +\|x\|_2$ oppure $\alpha = -\|x\|_2$, è sottesa alla riduzione degli errori nell'aritmetica finita del calcolatore (la matrice ortogonale non pone grossi problemi di calcolo poichè l'imposizione che la somma dei quadrati sia pari ad uno la rende tale che gli elementi siano ben limitati in modulo e quindi che anche gli errori sono limitati). Come detto

$$v = x - \alpha e_1 = \begin{pmatrix} x_1 - \alpha \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

è quindi opportuno che x_1 ed α siano di segno concorde così da eliminare il problema della cancellazione (soprattutto nel caso x_1 fosse l'elemento di

modulo massimo), dunque

$$\text{se } x_1 \geq 0 \quad , \quad \alpha = -\|x\|_2$$

$$\text{se } x_1 < 0 \quad , \quad \alpha = +\|x\|_2$$

Siamo ora in grado di dimostrare il teorema di esistenza della fattorizzazione QR

Teorema. *Se A è una matrice $\mathbb{R}^{m\times n}$, di rango massimo, allora $\exists Q \in \mathbb{R}^{m\times m}$ ortogonale ed $\exists R = \begin{pmatrix} R_1 \\ \hline \mathcal{O} \end{pmatrix} \in \mathbb{R}^{m\times n}, R_1 \in \mathbb{R}^{n\times n}$ triangolare superiore e non singolare tali che $A = QR$*

La dimostrazione di questo teorema definirà un algoritmo che ci consentirà di generare la fattorizzazione. Ragioniamo come abbiamo fatto con l'eliminazione di Gauss: quindi poniamo

$$A = \begin{pmatrix} a_{11}^{(0)} & \cdots & a_{1n}^{(0)} \\ \vdots & \ddots & \vdots \\ a_{m1}^{(0)} & \cdots & a_{mn}^{(0)} \end{pmatrix} \equiv A^{(0)}$$

considerando adesso

$$x = \begin{pmatrix} a_{11}^{(0)} \\ \vdots \\ a_{m1}^{(0)} \end{pmatrix}$$

possiamo costruire la matrice P_1 tale che

$$P_1 \begin{pmatrix} a_{11}^{(0)} \\ \vdots \\ a_{m1}^{(0)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

con $a_{11}^{(0)} \neq a_{11}^{(1)}$; in questo caso non abbiamo bisogno di pivoting come con l'algoritmo di Gauss: per come è costruito il vettore di Householder questo sarà sempre diverso dal vettore nullo ($x - \alpha e_1 = \underline{0} \iff x = \underline{0}$) e dunque la matrice P sarà sempre ben definita.

Ottenuta la matrice P_1 possiamo operare come già esaminato con Gauss

$$P_1 A^{(0)} = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m2}^{(1)} & \cdots & a_{mn}^{(1)} \end{pmatrix} \equiv A^{(1)}$$

come si può notare gli elementi sottodiagonali della prima colonna vengono annullati ma gli altri vengono comunque modificati, anche quelli della prima riga.

Consideriamo la seconda colonna, dall'elemento diagonale in poi e definiamo la matrice $P^{(2)}$ in modo tale che

$$P^{(2)} \begin{pmatrix} a_{22}^{(1)} \\ \vdots \\ a_{m2}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{22}^{(2)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad P^{(2)} \in \mathbb{R}^{m-1 \times m-1}$$

possiamo adesso definire la matrice P_2 come

$$P_2 = \left(\begin{array}{c|c} 1 & \\ \hline & P^{(2)} \end{array}\right) \qquad P_2 \in \mathbb{R}^{m \times m}$$

ed è ancora simmetrica ed ortogonale. Otteniamo infine

$$P_2 P_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m2}^{(2)} & a_{m3}^{(2)} & \cdots & a_{mn}^{(2)} \end{pmatrix} \equiv A^{(2)}.$$

La prima riga non viene modificata poichè la prima riga di P_2 è la prima riga dell'identità, vengono modificate solo le $m-1$ righe sotto la prima.

In generale, al passo i-esimo

$$P^{(i)} \begin{pmatrix} a_{ii}^{(i-1)} \\ \vdots \\ a_{mi}^{(i-1)} \end{pmatrix} = \begin{pmatrix} a_{ii}^{(i)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad P^{(i)} \in \mathbb{R}^{m-i+1 \times m-i+1}$$

$$P_i = \left(\begin{array}{c|c} I_{i-1} & \\ \hline & P^{(i)} \end{array}\right) \qquad P_i \in \mathbb{R}^{m \times m}$$

ortogonale e simmetrica. Alla fine, dopo n passi, otteniamo:

$$\underbrace{P_n \cdots P_2 P_1}_{Q^T} A = \left(\begin{array}{cccc} a_{11}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{nn}^{(n)} \\ \hline 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 \end{array}\right) \equiv \left(\frac{R_1}{\mathcal{O}}\right) \equiv R.$$

Il prodotto di matrici ortogonali è ancora una matrice ortogonale, da cui

$$Q^TQ = (P_n \cdots P_1)(P_1 \cdots P_n) = I$$

$$QQ^TA = QR$$

$$A = QR$$

Nel caso di matrici quadrate, se la matrice in esame è non singolare, allora è anche fattorizzabile QR, mentre per la fattorizzazione LU, come ci ricordiamo, ci sono ben altri vincoli, molto più stringenti che la semplice non singolarità.

Soffermiamoci sullo spazio di memoria di cui abbiamo bisogno. Ci riferiremo al primo passo, dal momento che gli altri sono simili. Per conoscere P_1 l'unica cosa di cui abbiamo bisogno è il vettore v, che ha dimensioni m: infatti noto v, possiamo ricavare P_1.

Dato che $rank(A) = n$ è massimo

$$x \neq \underline{0} \qquad \alpha^2 = \|x\|_2^2 > 0$$

$$v = x - \alpha e_1 = \begin{pmatrix} x_1 - \alpha \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

e sicuramente $v_1 \neq 0$ poichè x_1 ed α sono di segno concorde. Possiamo dunque definire

$$\tilde{v} = \frac{v}{v_1} \qquad \tilde{v}_1 \neq 0$$

e quindi possiamo evitare di memorizzare la prima componente del vettore $\tilde{v}$ in quanto sappiamo che è pari ad uno. Vediamo cosa comporta questa scelta di vettore per la matrice di Householder:

$$\tilde{P} = I - 2\frac{\tilde{v}\tilde{v}^T}{\tilde{v}^T\tilde{v}} = I - 2\frac{(\frac{v}{v_1})(\frac{v}{v_1})^T}{(\frac{v}{v_1})^T(\frac{v}{v_1})} = I - 2\frac{v_1^2}{v_1^2}\frac{vv^T}{v^Tv} = P$$

la matrice di Householder è perciò invariante per moltiplicazioni per scalari. In conclusione, invece di dover utilizzare $m - i$ componenti, ne abbiamo bisogno solo di $m - i - 1$, proprio quelle che andiamo ad azzerare; possiamo riscrivere la matrice A con tutte e sole le informazioni necessarie alla fattorizzazione: nella parte triangolare superiore metteremo R_1 mentre nelle parti che andiamo ad azzerare metteremo i vettori $\tilde{v}$ a partire dalla seconda componente in poi.

4.5.1 Implementazione in Matlab

Di seguito presentiamo l'implementazione del metodo di fattorizzazione QR.

```
function A=fattQR(A)
%FATTQR Fattorizza la matrice A come QR con Q matrice ortogonale
%   ed R triangolare superiore
%
%   A=FATTQR(A)
%
%   I parametri della funzione sono:
%       A -> la matrice quadrata da fattorizzare
%
%   I valori di ritorno sono:
%       A -> la matrice modificata contenente nella parte
%            strettamente triangolare inferiore gli elementi
%            significativi dei vettori di Householder nella
%            parte superiore R
%
%   See Also SOLVEQR, FATTLU, FATTPALU, FATTLDLT
  [m,n]=size(A);
  for i=1:n
(1)  alpha=norm(A(i:m,i));
     if alpha==0
        disp('No rango MAX!!');
        return
     end
     v1=A(i,i);
     if v1>=0
        v1=v1+alpha;
        s=1;
        A(i,i)=-alpha;
     else
        v1=v1-alpha;
        s=-1;
        A(i,i)=alpha;
     end
(2)  A(i+1:m,i)=A(i+1:m,i)/v1;
     vt=[1;A(i+1:m,i)];
     beta=s*v1/alpha;
(3)  A(i:m,i+1:n)=A(i:m,i+1:n)-(beta*[1;A(i+1:m,i)])*
                      ([1 A(i+1:m,i)']*A(i:m,i+1:n));
  end
```

4.5.2 Analisi dell'algoritmo e costo computazionale

Come si vede nel codice sopra abbiamo introdotto la quantità β per un fatto di efficienza: infatti quello che vogliamo rappresentare è

$$\frac{2}{v^T v}$$

che sappiamo essere uno scalare. Tramite alcuni passi algebrici si giunge al risultato che

$$\beta = \frac{2}{v^T v} = \frac{|x_1| + |\alpha|}{|\alpha|} \rightarrow \frac{s * v_1}{\alpha}$$

e l'aver introdotto questo fattore ci consente di spezzare il calcolo del prodotto tra la matrice di Householder e la matrice A in modo da ridurre la complessità: infatti calcolando esplicitamente vv^T avremmo ottenuto una matrice, che moltiplicata per A avrebbe aumentato considerevolmente la complessità dell'algoritmo; in questo modo riusciamo a calcolarci βv e successivamente $v^T A$ che è un vettore e quindi vanno a moltiplicarsi due vettori, un costo decisamente minore del prodotto matrice-matrice.

Per quanto riguarda il costo computazionale dividiamo il calcolo per le righe significative:

(1) si calcola una norma su un vettore di lunghezza $m - i + 1$ che prevede il quadrato di ogni componente, la loro somma ed infine estrarne la radice quadrata: un totale di $2(m - i + 1) + 1$ flops;

(2) si eseguono $m - i$ divisioni;

(3) la parte più corposa dell'algoritmo si trova qua dentro, infatti vengono eseguite circa $4(m-i)(n-i)$ flops, metà per il calcolo di $v^T A$ e le altre per l'aggiornamento della matrice.

Allora, sommando su i il costo della riga (3) si ottiene

$$\sum_{i=1}^{n} 4(m-i)(n-i) \approx \frac{4}{3}(n^3 + \frac{3}{2}n^2(m-n))$$

che nel caso quadrato $m = n$ diventa un costo proporzionale a $\frac{4}{3}n^3$, non proprio l'ideale per risolvere sistemi lineari.

4.5.3 Sperimentazioni dell'algoritmo

Come sempre verifichiamo che il nostro algoritmo funzioni, e per fare questo prendiamo una matrice ortogonale (nel nostro esempio la matrice identità 2×2 con le colonne invertite, che sappiamo essere ortogonale) ed una matrice triangolare superiore, le moltiplichiamo tra loro ed applichiamo la fattorizzazione QR al risultato:

```
» q=[0 1;1 0]

q =

     0     1
     1     0

» r=[5 6; 0 7]

r =

     5     6
     0     7

» a=q*r

a =

     0     7
     5     6

» a1=fattqr(a)

a1 =

    -5    -6
     1     7

» r1=triu(a1)

r1 =

    -5    -6
     0     7

» v1=[1;1]

v1 =

     1
     1

» p1=eye(2)-2*(v1*v1')/(v1'*v1)
```

```
p1 =

     0    -1
    -1     0

≫ v2=[1]

v2 =

     1

≫ p2=eye(1)-2*(v2*v2')/(v2'*v2)

p2 =

    -1

≫ p2=[eye(1) 0; 0 p2]

p2 =

     1     0
     0    -1

≫ q1=p1*p2

q1 =

     0     1
    -1     0

≫ q1*r1

ans =

     0     7
     5     6

≫
```

come risultato abbiamo ottenuto la matrice di partenza. Possiamo allora passare ad un esempio con una matrice casuale:

```
≫ a=round(10*rand(4))
```

```
a =

     4     4     5     2
     3     6     9     9
     3     1     9     2
     4     0     3     6

» fattqr(a)

ans =

  -7.07106781186548  -5.23259018078045 -12.16223663640862  -9.19238815542512
   0.27097657163519  -5.06162029393751  -3.82486217371702  -3.14128659928206
   0.27097657163519  -0.17544975548535  -5.78363461432936  -0.55069379758375
   0.36130209551359  -0.38969980181225  -0.00131662244093  -5.50718211451862

»
```

che contiene nella parte triangolare superiore la matrice R.

4.5.4 Implementazione in Matlab per risolvere $QRx = b$

Come sempre ci riportiamo al nostro problema di partenza, dunque alla risoluzione del sistema lineare $Ax = b$ tramite fattorizzazione QR. Dal momento che $Q^T A = R$ possiamo scrivere $Q^T Ax = Q^T b$ e dunque $Rx = \hat{b}$; il vettore $\hat{b}$ è il vettore dei termini noti modificato applicando la matrice Q^T, ma questa modifica può essere fatta mentre si esegue l'algoritmo, moltiplicando anch'esso per le matrici di Householder:

```
function x=solveQR(A,b)
%SOLVEQR Risolve il sistema lineare Ax=b fattorizzando la matrice
%   A come QR ed infine risolvendo il sistema
%                Rx=bˆ
%   dove bˆ è il vettore dei termini noti aggiornato come bˆ=QˆTb
%
%   x=SOLVEQR(A,b)
%
%   I parametri della funzione sono:
%       A -> la matrice dei coefficienti del sistema lineare
%       b -> il vettore dei termini noti
%
%   I valori di ritorno sono:
%       x -> il vettore soluzione del sistema lineare
```

```
%
%   See Also FATTQR
  [m,n]=size(A);
  x=b;
  for i=1:n
     alpha=norm(A(i:m,i));
     if alpha==0
        disp('No rango MAX!!');
        return
     end
     v1=A(i,i);
     if v1>=0
        v1=v1+alpha;
        s=1;
        A(i,i)=-alpha;
     else
        v1=v1-alpha;
        s=-1;
        A(i,i)=alpha;
     end
     A(i+1:m,i)=A(i+1:m,i)/v1;
     vt=[1;A(i+1:m,i)];
     beta=s*v1/alpha;
     A(i:m,i+1:n)=A(i:m,i+1:n)-(beta*[1;A(i+1:m,i)])*
                      ([1 A(i+1:m,i)']*A(i:m,i+1:n));
     b(i:n)=b(i:n)-(beta*[1 A(i+1:m,i)']*b(i:n))*[1;A(i+1:m,i)];
  end
  x=solveUT(A,b);
```

4.5.5 Sperimentazioni dell'algoritmo

Proviamo l'algoritmo di risoluzione dei sistemi lineari tramite fattorizzazione QR con la matrice dell'esempio precedente:

```
≫ a

a =

     0     7
     5     6

≫ x=solveQR(a,[1;1])

x =
```

```
   0.02857142857143
   0.14285714285714

» inv(a)*[1;1]

ans =

   0.02857142857143
   0.14285714285714

»
```

ed anche in questo caso abbiamo verificato il risultato ottenuto co nostro algoritmo con il calcolo esplicito del vettore soluzione x.

Capitolo 5

Soluzione di equazioni non lineari

In questa parte ci occupiamo dell'approssimazione numerica delle radici (anche dette zeri) di una funzione di una variabile reale. Definendo in generale il problema come, si ha

$$f : \mathcal{D} \subseteq \mathbf{R}^n \to \mathbf{R}^m$$

ed un vettore

$$y \in \mathbf{R}^m$$

e vorremmo risolvere

$$f(x) = y$$

e quindi trovare quel vettore x che renda vera l'uguaglianza. A priori noi non possiamo sapere se questa soluzione esista oppure sua unica, infatti vediamo negli esempi successivi come siano varie le situazioni in cui possiamo imbatterci:

$f(x) \equiv x^2 + 1 = 0$	non esiste soluzione	$S = \emptyset$
$f(x) \equiv (x-1)(x^2+1) = 0$	esiste ed è unica	$S = \{1\}$
$f(x) \equiv x^4 - 1 = 0$	esiste ma non è unica	$S = \{\pm 1\}$
$f(x) \equiv sinx = 0$	un insieme numerabile	$S = \{x = k\pi \vert k \in \mathbb{Z}\}$
$f(x_1, x_2) \equiv x_1^2 - x_2 = 0$	un insieme continuo	$S = \{(x_1, x_2) \in \mathbb{R}^2 \vert x_2 = x_1^2\}$

Possiamo tranquillamente considerare $y = \underline{0}$ senza perdita di generalità: infatti risolvere $g(x) = y$ è del tutto equivalente a risolvere $f(x) \equiv g(x) - y = \underline{0}$.

La nostra analisi si concentrerà nel caso unidimensionale, e cioè quando $m = n = 1$ e svilupperemo dei metodi numerici che ci consentiranno di individuare uno zero di funzione.

5.1 Il metodo di Bisezione

Presentiamo qui il metodo di bisezione che, data una funzione continua

$$f : \mathcal{I} = (a, b) \subseteq \mathbb{R} \to \mathbb{R}$$

e tale che

$$f(a)f(b) < 0$$

genera una successione $\{x_k\}$ che converge verso la soluzione $\overline{x}$. Infatti, sotto le ipotesi fatte prima, sappiamo dal Teorema di esistenza degli zeri che

$$\exists \overline{x} \in (a, b) \text{ t.c. } f(\overline{x}) = 0$$

.

L'algoritmo procede come segue:

$$x = \frac{a + b}{2}$$

1. se $f(x) = 0 \Rightarrow \overline{x} = x$

2. se $f(a)f(x) < 0 \Rightarrow$ si riapplica nell'intervallo $[a, x]$

3. se $f(a)f(x) > 0 \Rightarrow$ si riapplica nell'intervallo $[x, b]$

Difficilmente l'algoritmo potrà convergere dato che, in aritmetica finita, la condizione di uscita $f(x) = 0$ non si verificherà mai, se non in casi molto rari. Si devono quindi cercare criteri di arresto alternativi a quello proposto sopra. Ricordiamoci che quello che interessa a noi non è la soluzione esatta, ma una buona approssimazione di essa.

Indicando con $L_k = |b_k - a_k|$ la larghezza dell'intervallo nel quale attualmente stiamo cercando la soluzione, otteniamo la seguente relazione ricorsiva

$$L_k = \frac{1}{2} L_{k-1}.$$

Sviluppando fino al termine di indice 0 avremo:

$$L_k = \frac{1}{2} L_{k-1} = \cdots = \frac{1}{2^k} L_0.$$

Allora fissata la nostra tolleranza ε, cioè la bontà con cui cerchiamo la soluzione, basterà arrestarsi quando l'ampiezza dell'intervallo sarà minore di questo valore. Infatti per poter avere $x_k \approx \overline{x}$ deve verificarsi

$$\frac{1}{2^k} L_0 < \varepsilon \quad \Leftrightarrow \quad 2^k \geq \frac{L_0}{\varepsilon} \quad \Leftrightarrow \quad k \ln 2 \geq \ln L_0 - \ln \varepsilon$$

$$k \geq \left\lceil \frac{\ln L_0 - \ln \varepsilon}{\ln 2} \right\rceil = \nu$$

e questo ν indica il numero massimo di iterazioni necessarie per ottenere una approssimazione di $\overline{x}$ con la precisione ε richiesta. Al diminuire di ε, cioè più si avvicina allo zero, e più il metodo di bisezione diventa inefficiente.

Un altro criterio di arresto potrebbe basarsi sul valore che assume la funzione in esame nel punto di approssimazione corrente. Potremmo difatti decidere di arrestare il nostro algoritmo quando $|f(x_k)| \leq tolf$ ossia quando la funzione è inferiore ad una certa tolleranza. Sviluppando in serie di Taylor la funzione in un intorno di $\overline{x}$ ed arrestandoci al primo ordine si ottiene:

$$f(x_k) = f(\overline{x}) + f'(\overline{x})(x_k - \overline{x})$$

e sapendo che per definizione $f(\overline{x}) = 0$ possiamo riscrivere tutto come:

$$f(x_k) = f'(\overline{x})(x_k - \overline{x}) \longrightarrow (x_k - \overline{x}) = \frac{f(x_k)}{f'(\overline{x})}$$

portandoci ai valori assoluti

$$|x_k - \overline{x}| = \left| \frac{f(x_k)}{f'(\overline{x})} \right| \leq \frac{|f(x_k)|}{|f'(\overline{x})|} \leq \frac{tolf}{|f'(\overline{x})|}$$

Naturalmente vorremmo che, arrestandoci perché il valore della funzione è inferiore alla tolleranza, sia verificato anche che $|x_k - \overline{x}| \leq \varepsilon$ quindi imponendo:

$$\frac{tolf}{|f'(\overline{x})|} \approx \varepsilon \quad \rightarrow \quad tolf \approx \varepsilon |f'(\overline{x})|.$$

La derivata, ovviamente, non la abbiamo a disposizione ed il suo calcolo richiederebbe uno sforzo computazionale superiore allo stesso metodo di bisezione, quello che si può fare è approssimare la derivata con il rapporto incrementale dei punti estremi dell'intervallo attualmente preso in considerazione, alla fine si ottiene:

$$f'(\overline{x}) \approx \frac{f(b_k) - f(a_k)}{b_k - a_k} \quad \Rightarrow \quad tolf \approx \varepsilon \left| \frac{f(b_k) - f(a_k)}{b_k - a_k} \right|$$

Questo ultimo metodo di arresto è generale e non è stato utilizzato niente che riguardi il metodo di bisezione, possiamo perciò utilizzarlo anche con i metodi che vedremo in seguito avendo l'accortezza di utilizzare la derivata od una sua approssimazione a seconda che sia disponibile oppure no.

5.1.1 Ordine di convergenza

Questo dato ci indica quanto velocemente la successione di valori generata da un metodo numerico converge verso la soluzione.

Considerando $e_k = x_k - \overline{x}$ come l'errore relativo commesso al passo k, si dice che un metodo ha convergenza p ($p \in \mathbb{R} \geq 1$) se e solo se:

$$\lim_{k \to +\infty} \frac{|e_{k+1}|}{|e_k|^p} = c \neq 0 \text{ e positivo}$$

con c che è detta costate asintotica dell'errore e rappresenta quanto è più piccolo $|e_{k+1}|$ rispetto a $|e_k|$. E' utile notare che, nel caso $p = 1$ è necessario che $0 < c < 1$ altrimenti il metodo non convergerebbe, come si vede

$$\text{per } k \gg 1 \quad |e_{k+1}| \approx c|e_k|^p$$

e se $p = 1$ perché converga deve essere $c < 1$.

Nel caso del metodo si bisezione, il limite diviene

$$\lim_{k \to +\infty} \frac{|e_{k+1}|}{|e_k|} = \frac{1}{2}$$

infatti ad ogni passo di dimezza il campo di ricerca. Il metodo di bisezione è quello che si dice un metodo lineare ($p = 1$), è lento ma ha il pregio di avere un limite superiore al numero di iterazioni necessarie per ottenere un'approssimazione della soluzione ed in più converge sempre, ha quella che si chiama convergenza totale: qualsiasi sia l'intervallo scelto, che rispetti le ipotesi iniziali, il metodo di bisezione ci porterà verso la soluzione; vedremo che questo comportamento non è comune a tutti gli algoritmi.

5.1.2 Implementazione in Matlab

Ecco il codice Matlab che implementa il metodo di bisezione

```
function [x,i,tolf,nu]=bisezione(a,b,f,tolx)
%BISEZIONE Esegue il metodo di bisezione per il calcolo della radice
%   di una funzione non lineare
%
%   [i,x,tolf,nu]=BISEZIONE(a,b,f,tolx)
%
%   I parametri della funzione sono:
%       f -> funzione di cui valutare uno zero
%       a,b -> estremi dell'intervallo in cui si ricerca lo zero di f;
%              si richiede che f(a)f(b)<0
%       tolx -> tolleranza sulla x
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
```

```
%       nu -> il numero massimo di iterazioni necessarie per ottenere una
%             soluzione con la precisione tolx
%
%   See Also NEWTON
  nu=ceil(log2(b-a)-log2(tolx));
  fa=feval(f,a);
  fb=feval(f,b);
  if fa*fb>0
     disp('La funzione deve soddisfare f(a)f(b)<0!')
     break
  end
  for i=1:nu
      c=(a+b)/2;
      fc=feval(f,c);
      if abs(b-a)<tolx
          break
      end
      tolf=tolx*abs((fb-fa)/(b-a));
      if abs(fc)<=tolf
          break
      end
      if (fa*fc)<0
          b=c;
          fb=fc;
      else
          a=c;
          fa=fc;
      end
  end
  x=c;
```

5.1.3 Sperimentazioni dell'algoritmo

Proviamo il nostro algoritmo su una semplice funzione: $f(x) = x - cosx$; dal momento che siamo sicuri della convergenza del metodo, utilizzeremo la soluzione trovata con bisezione per verificare anche gli altri metodi; dapprima verifichiamo se l'intervallo scelto rispetta le richieste del metodo

```
≫ fxcosx(0)*fxcosx(1)

ans =

  -0.45969769413186
```

```
≫
```

possiamo utilizzare $[0, 1]$ come intervallo per la ricerca del nostro zero, in quanto $f(x)$ è continua in quell'intervallo ed ammette valori discordi agli estremi. Vediamo come si comporta l'algoritmo:

```
≫ [x,it,tolf,nu]=bisezione(0,1,'fxcosx',1e-15)

x =

   0.73908513321516


it =

    49


tolf =

    1.687500000000000e-015


nu =

    50
```

L'algoritmo arriva ad un'approssimazione della funzione con un passo in meno del previsto; verifichiamo la bontà di tale approssimazione:

```
≫ fxcosx(x)

ans =

    4.440892098500626e-016

≫
```

l'approssimazione trovata è sicuramente buona. Lentamente ma il metodo di bisezione arriva ad un valore molto prossimo alla soluzione esatta.

Non sempre il metodo di bisezione è da disprezzare: inseriamo un esempio che ci farà comodo in seguito:

```
≫ type fx5
```

```
function y=fx5(x)
  y=(x-5)^5;

» [x,it,tolf,nu]=bisezione(4,7,'fx5',1e-15)

x =

   5.00000000000011


it =

    43


tolf =

    2.940041181101415e-065


nu =

    52

»
```

5.2 Metodo di Newton

Per questo metodo si richiede che $f \in \mathbb{C}^1$ che significa che la funzione deve essere continua, derivabile e a derivata continua ma non si suppone l'esistenza della soluzione come si faceva nel metodo di bisezione.

Il metodo di Newton approssima la funzione con la retta tangente ad essa nel punto di coordinate $(x_0, f(x_0))$ e si prende il nuovo punto di approssimazione come l'intersezione di questa retta con l'asse delle x, reiterando nuovamente il procedimento. Se sviluppiamo nuovamente la funzione in serie di Taylor arrestandoci al primo ordine si ottiene:

$$f(x) \approx fa(x) = f(x_k) + f'(x_k)(x - x_k)$$

scegliendo x_{k+1} in modo tale che $fa(x_{k+1}) = 0$ quello che si ottiene è

$$\begin{aligned} fa(x_{k+1}) &= f(x_k) + f'(x_k)(x_{k+1} - x_k) = 0 \\ x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} \end{aligned}$$

otteniamo quindi la relazione ricorsiva

$$\begin{cases} x_0 \text{ assegnato} \\ x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \end{cases}$$

La convergenza del metodo di Newton è locale, cioè si deve scegliere x_0 abbastanza vicino ad $\overline{x}$ perché il metodo converga; questo inconveniente non si aveva con il metodo di bisezione, che garantiva la convergenza qualsiasi fosse l'intervallo scelto (che rispettasse le richieste).

Teorema (del punto fisso). *Sia $\phi : \mathbb{R} \to \mathbb{R}$ una funzione continua ($\in \mathbb{C}^0$) e tale che $\overline{x} = \phi(\overline{x})$, allora $\overline{x}$ è detto punto fisso di ϕ. Se $\exists 0 < L < 1$ ed un intorno di $\overline{x}$ di raggio r ($I(\overline{x}, r)$) tale che*

$$\forall x, y \in I : |\phi(x) - \phi(y)| \leq L|x - y|$$

allora la successione definita da

$$\begin{cases} x_0 \in I \text{ assegnato} \\ x_{k+1} = \phi(x_k) \end{cases}$$

è tale che

$$\lim_{k \to +\infty} x_k = \overline{x}$$

Dimostrazione. Dimostreremo che se $x_0 \in I \Rightarrow \{x_k\} \subset I$. Supponiamo $x_k \in I$ e guardiamo se anche $x_{k+1} \in I$.

$$\begin{aligned} &|x_{k+1} - \overline{x}| = |\phi(x_k) - \phi(\overline{x})| \leq L|x_k - \overline{x}| \leq |x_k - \overline{x}| \\ &0 \leq |x_k - \overline{x}| \leq L|x_{k-1} - \overline{x}| \leq L^2|x_{k-2} - \overline{x}| \leq \cdots \leq L^k|x_0 - \overline{x}| \\ &\lim_{k \to +\infty} L^k|x_0 - \overline{x}| = 0 \end{aligned}$$

e quindi, grazie al confronto, abbiamo che $x_k \to \overline{x}$. □

Dividiamo adesso lo studio della convergenza nel caso di radici semplici e radici multiple, definendo prima cosa si intenda per molteplicità di una radice:

Definizione. *Si dice che $\overline{x}$ è radice di f con molteplicità p ($p \in \mathbb{Z} \geq 1$) se e solo se*

$$f(\overline{x}) = f'(\overline{x}) = \cdots = f^{(p-1)}(\overline{x}) = 0 \text{ e } f^{(p)}(\overline{x}) \neq 0.$$

Se $p = 1$ $\overline{x}$ si dice radice semplice.

5.2.1 Convergenza di Newton con radici semplici

Nel caso di radici semplici si può enunciare la seguente proposizione che sancisce la convergenza del metodo di Newton:

Proposizione. *Se $f \in \mathbb{C}^2$ e $|x_0 - \overline{x}| < \delta$ con δ sufficientemente piccolo allora il metodo di Newton converge.*

Dimostrazione. Sia $\phi(x) = x - \frac{f(x)}{f'(x)}$ allora sappiamo che:

1. $\phi(\overline{x}) = \overline{x}$

2. $\phi'(x) = 1 - \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2} = \frac{f(x)f''(x)}{(f'(x))^2}$

allora questo implica che:

$$\begin{cases} \phi'(\overline{x}) = 0 \\ \phi' \text{ continua in un intervallo di } \overline{x} \end{cases}$$

possiamo perciò affermare che

$$\lim_{x \to \overline{x}} \phi'(x) = \phi'(\overline{x}) = 0$$

seguendo adesso la definizione di limite

$$\forall 0 < L < 1 \quad \exists \delta : \quad \forall x \in [\overline{x} - \delta, \overline{x} + \delta] \quad |\phi'(x) - \phi'(\overline{x})| = |\phi'(x)| < L.$$

Ora, $\forall x, y \in I$ (si può considerare $x > y$ senza perdita di generalità), sviluppando $\phi(x)$ in serie di Taylor arrestandoci al primo ordine con resto di Lagrange si ottiene

$$\phi(x) = \phi(y) + \phi'(\xi)(x - y) \qquad \xi \in [y, x] \subset I$$

$$|\phi(x) - \phi(y)| = |\phi'(\xi)||x - y| < L|x - y|.$$

Abbiamo così soddisfatto le ipotesi del Teorema del punto fisso avendo avuto l'accortezza di scegliere $\phi(x)$ come la successione del metodo di Newton. □

Inoltre si ottiene anche il seguente importante risultato:

Proposizione. *Sia $f \in \mathbb{C}^2$ e $\{x_k\}$ la successione di approssimazioni generata dal metodo di Newton, allora*

$$\lim_{k \to +\infty} \frac{|e_{k+1}|}{|e_k|^2} = c \neq 0.$$

Dimostrazione.

$$0 = f(\overline{x}) = f(x_k) + f'(x_k)(\overline{x} - x_k) + f''(\xi_k)(\overline{x} - x_k)^2 =$$

$$= f'(x_k) \left[\underbrace{\frac{f(x_k)}{f'(x_k)} - x_k}_{-x_{k+1}} + \overline{x} \right] + f''(\xi_k) \underbrace{(\overline{x} - x_k)^2}_{e_k}$$

$$= f'(x) \overbrace{e_{k+1}}^{-x_{k+1}+x_k} + f''(\xi_k) {e_k}^2 = 0$$

$$\left| \frac{e_{k+1}}{{e_k}^2} \right| = \left| \frac{f''(\xi_k)}{f'(x_k)} \right| \overset{k \to +\infty}{\longrightarrow} \frac{|f''(x_k)|}{|f'(\overline{x})|}$$

ed in generale è un valore diverso da zero (cioè $|f''(x_k)| \neq 0$), se fosse $|f''(x_k)| = 0$ saremmo fortunati perché $p \geq 3$ e potremmo continuare nello sviluppo; inoltre

$$0 \leq |\overline{x} - \xi_k| \leq |\overline{x} - x_k|$$

ed essendo $\xi_k \in I(\overline{x}, x_k)$ per $k \to +\infty$

$$|x - x_k| \to 0 \quad \Rightarrow \quad |x - \xi_k| \to 0$$

□

5.2.2 Criteri di arresto

Come abbiamo già visto, un criterio di arresto è

$$|f(x_k)| \leq tolf.$$

Inoltre vorremmo imporre una condizione del tipo:

$$|x_k - \overline{x}| \leq tolx,$$

ma non siamo in grado di valutarla in quanto non sappiamo qual'è il valore di $\overline{x}$; cerchiamo allora una maggiorazione della quantità $|x_k - \overline{x}|$ per arrestarci quando questa è minore od uguale a $tolx$.

Come visto in precedenza, prendendo $\phi(x)$ in modo che descriva il metodo di Newton, e cioè nella forma

$$\phi(x) = x - \frac{f(x)}{f'(x)}$$

si può scrivere

$$\forall 0 < L < 1 \quad \exists \delta > 0 \text{ tale che } \forall x, y \in [\overline{x} - \delta, \overline{x} + \delta] \quad |\phi(x) - \phi(y)| \leq L|x - y|.$$

Cerchiamo allora di scrivere in modo più utile $|x_k - \overline{x}|$:

$$|x_k - \overline{x}| = |x_k - x_{k+1} + x_{k+1} - \overline{x}| \leq |x_k - x_{k+1}| + |x_{k+1} - \overline{x}| =$$

$$= |\phi(x_{k-1}) - \phi(x_k)| + |\phi(x_k) - \phi(\overline{x})|.$$

Scegliendo $x_0 \in [\overline{x} - \delta, \overline{x} + \delta] = I$ tutta la successione generata apparterrà ad I, quindi:

$$|\phi(x_{k-1}) - \phi(x_k)| + |\phi(x_k) - \phi(\overline{x})| \leq L|x_{k-1} - x_k| + L|x_k - \overline{x}|$$

$$|x_k - \overline{x}| \leq L|x_{k-1} - x_k| + L|x_k - \overline{x}|$$

$$(1-L)|x_k - \overline{x}| \leq L|x_{k-1} - \overline{x}| \quad \Rightarrow \quad |x_k - \overline{x}| \leq \frac{L}{1-L}|x_{k-1} - x_k| \leq tolx.$$

Il fattore $\frac{L}{1-L}$ dipende dal punto iniziale e dalla funzione, ma sicuramente

$$\exists x_0 : L = \frac{1}{2} \quad \Rightarrow \quad \frac{L}{1-L} = 1 \quad \Rightarrow \quad |x_{k-1} - x_k| \leq tolx$$

che è la nostra condizione di arresto.

5.2.3 Implementazione in Matlab (radici semplici)

Il codice di seguito implementa il metodo di Newton classico, quello cioè per radici semplici

```
function [x,i,tolf]=newton(x0,f,df,tolx,nmax)
%NEWTON Esegue il metodo di Newton per il calcolo della radice
%   di una funzione non lineare
%
%   [x,i,tolf]=NEWTON(x0,f,df,tolx,nmax)
%
%   I parametri della funzione sono:
%       x0 -> il punto iniziale
%       f -> funzione di cui valutare uno zero
%       df -> la derivata della funzione f
%       tolx -> tolleranza per la radice
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
%
%  See Also NEWTONMOLT, AITKEN, BISEZIONE, CORDE, SECANTI, STEFFENSEN
  i=0;
```

```
  err=tolx+1;
  x=x0;
  while(i<nmax & err>tolx)
      fx=feval(f,x);
      dfx=feval(df,x);
      tolf=tolx*abs(dfx);
      if abs(fx)<=tolf
         break
      end
      x1=x-(fx/dfx);
      err=abs(x1-x);
      i=i+1;
      x=x1;
  end
```

5.2.4 Sperimentazioni dell'algoritmo

Eseguiamo l'algoritmo sulla stessa funzione del metodo di bisezione e guardiamo il risultato:

```
≫ [x,it,tolf]=newton(0,'fxcosx','dfxcosx',1e-15,2000)

x =

   0.73908513321516


it =

     5


tolf =

    1.673612029183215e-015
≫
```

impiega circa un decimo delle iterazioni di bisezione. Si deve comunque tenere presente che ad ogni iterazione si devono valutare due funzioni, e da non trascurare è il costo del calcolo analitico della derivata della funzione.

Proviamo adesso ad applicare l'algoritmo a due funzioni a radici multiple, per mostrare quanto lentamente converge verso la soluzione; la prima funzione che vedremo è $f(x) = (x-5)^5$ che ammette come soluzione $x = 5$:

```
≫ [x,it,tolf]=newton(0,'fx5','dfx5',1e-15,2000)
```

```
x =

   4.99999999999999


it =

   155


tolf =

    7.470729841072302e-072

≫
```

impiega ben 155 iterazioni per raggiungere un'approssimazione neanche troppo precisa della soluzione: vedremo come si riuscirà ad ottenere risultati molto migliori (perfino il metodo di bisezione ottiene risultati migliori, impiegando soltanto 43 iterazioni).

La seconda funzione che andremo a vedere è leggermente più complessa ed è $f(x) = (x-1)^4(x-2)$ e otteniamo come risultato

```
≫ [x,it,tolf]=newton(0,'fxm','dfxm',1e-15,2000)

x =

   1.00000000000000


it =

   117


tolf =

    2.772655121618958e-058

≫
```

certamente un risultato non esaltante per un metodo così oneroso come Newton.

Quello che abbiamo intenzione di mostrare adesso è come la scelta del punto iniziale possa influenzare la convergenza del metodo verso una soluzione oppure verso un'altra; la funzione che prenderemo in esame è la semplice $f(x) = sinx$ e cambiando punto di partenza otteniamo due delle sue infinite soluzioni:

```
≫ [x,it,tolf]=newton(1,'sin','cos',1e-15,2000)

x =

     0


it =

     5


tolf =

    1.000000000000000e-015

≫ [x,it,tolf]=newton(2,'sin','cos',1e-15,2000)

x =

   3.14159265358979


it =

     6


tolf =

    1.000000000000000e-015

≫
```

in un caso abbiamo scelto $x_0 = 1$ ed il metodo converge verso la soluzione $\overline{x} = 0$, nel secondo caso, invece, si è scelto $x_0 = 2$ e la successione di approssimazioni converge verso la soluzione $\overline{x} = \pi$

Come la scelta del punto iniziale può portare a convergere verso una soluzione piuttosto che verso un'altra (nel caso ce ne siano più d'una), questa

può anche portare a non convergere il metodo di Newton come vedremo adesso. Consideriamo la funzione $f(x) = arctgx$ che è continua e derivabile su tutto $\mathbb{R}$. Cerchiamo il punto iniziale tale che

$$arctg|x_0| \geq \frac{2|x_0|}{1+x_0^2}$$

questo punto esiste e per trovarlo possiamo utilizzare il metodo di bisezione, che sappiamo convergere:

```
≫ type fnonew

function y=fnonew(x);
  y=atan(abs(x))-2*abs(x).*dfatanx(x);

≫ x0=bisezione(0.1,3,'fnonew',1e-15)

x0 =

   1.39174520027073

≫ fnonew(x0)

ans =

    1.110223024625157e-016

≫
```

$x0$ è il punto che stavamo cercando; ci si presentano due scelte: porre $x_0 = x0$ oppure $x_0 > x0$ entrambe interessanti, vediamo perché:

```
≫ [x,it]=newton(x0,'fatanx','dfatanx',1e-10,2000)

x =

   1.39174520027073


x =

   -1.39174520027073


x =
```

```
   1.39174520027073

......

it =

        2000

≫

≫ newton(3,'fatanx','dfatanx',1e-10,2000)
x1 =

  -9.49045772398254

x1 =

    1.239995111788842e+002

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

x1 =

    1.554292211858608e+146

x1 =

   -3.794767904961391e+292

x1 =

   Inf
```

Nel primo caso vediamo come la funzione continui ad alternarsi tra $-x_0$ ed x_0 senza mai convergere verso la soluzione; nel secondo caso, invece, si vede come scegliendo un valore maggiore di quell'$x0$ la successione delle approssimazioni diverga verso infinito. Scegliendo un valore appena inferiore ad $x0$ e precisamente $x0 - eps$, dove con eps indichiamo la precisione di macchina, il metodo di Newton converge, come si vede

```
» [x,it,tolf]=newton(x0-eps,'fatanx','dfatanx',1e-15,2000)

x =

   -2.073113138404900e-019


it =

    41


tolf =

    1.000000000000000e-015

»
```

Vogliamo dare anche un esempio di come a volte le approssimazioni possano creare non pochi problemi. Prendiamo nuovamente in esame la funzione $f(x) = sinx$ e cerchiamo come punto iniziale

$$x_0 : \quad tanx_0 = 2x_0.$$

Proviamo a vedere analiticamente i primi due passi del metodo di Newton:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = x_0 - \frac{sinx_0}{\underbrace{cosx_0}_{tanx_0}} = x_0 - 2x_0 = -x_0$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = -x_0 - \frac{-sinx_0}{cosx_0} = -x_0 + tanx_0 = -x_0 + 2x_0 = x_0$$

In due passi siamo tornati al punto di partenza. Calcoliamo il punto iniziale $x0$ ancora tramite il metodo di bisezione:

```
» f

f =
```

```
     Inline function:
     f(x) = tan(x)-2*x

» x0=bisezione(1,pi/2-eps,f,1e-15s)

x0 =

   1.16556118520721

» f(x0)

ans =

   -4.440892098500626e-016

»
```

proviamo dunque ad applicare il metodo di Newton innescato con questo punto:

```
» [x,it,tolf]=newton(x0,'sin','cos',1e-15,2000)

x1 =

  -1.16556118520721


x1 =

   1.16556118520721


x1 =

  -1.16556118520719


x1 =

   1.16556118520712


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

```
x =

     0


it =

    26


tolf =

    1.000000000000000e-019

≫
```

Al contrario di quanto ci si aspettava l'algoritmo converge! E questo è proprio dovuto alle approssimazioni fatte; se si guarda le prime due iterazioni, l'algoritmo si composta come previsto, ma già dalla terza inizia a perdere precisione sull'ultima cifra significativa, ed è questo fatto che porta alla convergenza il metodo.

5.2.5 Il metodo di Newton per radici multiple

Nel caso la molteplicità della radice sia nota e questa sia r avremo

$$\lim_{k\to+\infty} \frac{|e_{k+1}|}{|e_k|} = 1 - \frac{1}{r} = c$$

L'algoritmo prima visto può essere modificato nel modo seguente

$$x_{k+1} = x_k - r\frac{f(x_k)}{f'(x_k)}$$

consentendoci di recuperare pienamente l'efficienza del metodo di Newton ottenendo nuovamente una convergenza quadratica.

Nel caso in cui la molteplicità della radice non sia nota, ci sarà bisogno di sviluppare un nuovo algoritmo, noto come metodo di accelerazione di Aitken, che vedremo più avanti.

5.2.6 Implementazione in Matlab (radici multiple)

Quello che segue è il codice per la modifica del metodo di Newton in caso di radici multiple

```
function [x,i,tolf]=newtonmolt(x0,r,f,df,tolx,nmax)
%NEWTONMOLT Esegue il metodo di Newton modificato per la risoluzione di
%   f(x)=0 anche in presenza di radici multiple, nota la loro
%   molteplicità
%
%   [x,i,tolf]=NEWTONMOLT(x0,r,f,df,tolx,nmax)
%
%   I parametri della funzione sono:
%       x0 -> il punto iniziale
%       r -> molteplicità della radice cercata
%       f -> funzione di cui valutare uno zero
%       df -> la derivata della funzione f
%       toll -> tolleranza per la radice
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
%
%   See Also NEWTON, AITKEN
  i=0;
  err=tolx+1;
  x=x0;
  while(i<nmax & err>tolx)
      fx=feval(f,x);
      dfx=feval(df,x);
      tolf=tolx*abs(dfx);
      if abs(fx)<=tolf
         break
      end
      x1=x-r*(fx/dfx);
      err=abs(x1-x);
      x=x1;
      i=i+1;
  end
```

5.2.7 Sperimentazioni dell'algoritmo

Riprendiamo gli esempi con radici multiple già visti durante la sperimentazione per il metodo di Newton, mostrando come, conoscendo la molteplicità della radice, si riesca ad ottenere ancora un algoritmo molto efficiente:

```
» [x,it,tolf]=newtonmolt(0,5,'fx5','dfx5',1e-15,2000)
```

```
x =

     5


it =

     1


tolf =

     0
≫
```

in questo caso, essendo la funzione in esame del tipo $(x-\alpha)^r$ e conoscendo la molteplicità r, l'algoritmo riesce a convergere in una sola iterazione verso la soluzione $x = \alpha \equiv 5$.

```
≫ [x,it,tolf]=newtonmolt(0,4,'fxm','dfxm',1e-15,2000)

x =

     1


it =

     5


tolf =

     0
≫
```

essendo una funzione leggermente più complessa della precendente, il metodo di Newton modificato impiega qualche iterazione per convergere verso una buona approssimazione, niente però di paragonabile al metodo classico.

5.3 Metodo di accelerazione di Aitken

Il metodi di accelerazione di Aitken viene utilizzato per ottenere ancora una convergenza quadratica in caso di radici multiple, questa volta senza conoscerne la molteplicità. Tutto il metodo si basa sul seguente limite:

$$\lim_{k\to+\infty} \frac{\overline{x} - x_{k+1}}{\overline{x} - x_k} = c$$

a partire da questo limite si possono ottenere le seguenti espressioni:

$$k \gg 1$$

$$\begin{aligned} \overline{x} - x_{k+1} &\approx c(\overline{x} - x_k) \\ \overline{x} - x_k &\approx c(\overline{x} - x_{k-1}) \end{aligned} \tag{5.1}$$

sottraendo tra loro queste due quantita si ricava

$$\begin{aligned} x_k - x_{k+1} &\approx c(x_k - x_{k-1}) \\ c &\approx \frac{x_k - x_{k+1}}{x_{k-1} - x_k}. \end{aligned}$$

Considerando l'espressione 5.1 e sviluppando

$$\begin{aligned} \overline{x} - x_k &\approx c(\overline{x} - x_k) \\ (1-c)\overline{x} &\approx x_{k+1} - cx_k \\ \overline{x} \approx \frac{x_{k+1} - cx_k}{1-c} &\approx \frac{x_{k+1}x_{k-1} - x_k^2}{x_{k+1} - 2x_k + x_{k-1}} = \hat{x}_k. \end{aligned}$$

una volta ottenute le approssimazioni x_{k-1}, x_k e x_{k+1} si può generare $\hat{x}_k$ tramite il metodo di Aitken. Uno schema riassuntivo dell'algoritmo è simile al seguente:

$$\left.\begin{array}{c} x_0 \\ \downarrow \\ x_1 \\ \downarrow \\ x_2 \end{array}\right\} \Rightarrow \left.\begin{array}{c} \hat{x}_1 \\ \downarrow \\ \tilde{x}_2 \\ \downarrow \\ \tilde{x}_3 \end{array}\right\} \Rightarrow \begin{array}{c} \hat{x}_2 \\ \vdots \end{array}$$

Se $\{x_k\}$ converge linearmente, allora la successione $\{\hat{x}_k\}$ converge quadraticamente.

5.3.1 Implementazione in Matlab

Di seguito presentiamo l'implementazione del metodo di accelerazione di Aitken

```
function [x,i,tolf]=aitken(x0,f,df,tolx,nmax)
%AITKEN Esegue il metodo di accelerazione Aitken, una modifica del metodo
%   di Newton per il calcolo di f(x)=0 in presenza di radici multiple,
%   senza conoscerne la molteplicità
%
%   [x,i,tolf]=AITKEN(x0,f,df,tolx,nmax)
%
%   I parametri della funzione sono:
%       x0 -> il punto iniziale
%       f -> funzione di cui valutare uno zero
%       df -> la derivata della funzione f
%       tolx -> tolleranza per la radice
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
%
%   See Also NEWTON, NEWTONMOLT
  i=0;
  err=tolx+1;
  x=x0;
  while (i<nmax & err>tolx)
      x0=x;
      fx0=feval(f,x0);
      dfx0=feval(df,x0);
      tolf=tolx*abs(dfx0);
      if abs(fx0)<=tolf
         break
      end
      x1=x0-(fx0/dfx0);
      err=abs(x1-x0);
      if err<tolx
         break
      end
      fx1=feval(f,x1);
      dfx1=feval(df,x1);
      tolf=tolx*abs(dfx1);
      if abs(fx1)<=tolf
         break
      end
      x2=x1-(fx1/dfx1);
      err=abs(x2-x1);
```

```
      if err<tolx
         break
      end
      x=(x2*x0-(x1)^2)/(x2-2*x1+x0);
      err=abs(x0-x);
      i=i+1;
  end
```

5.3.2 Sperimentazioni dell'algoritmo

Proviamo anche il metodo di Aitken sugli stessi esempio visti per il metodo di Newton classico e modificato:

```
≫ [x,it,tolf]=aitken(0,'fx5','dfx5',1e-15,2000)

x =

   5.00000000000000


it =

     1


tolf =

    3.111507638930571e-075

≫
```

anche il metodo di Aitken impiega una sola iterazione per giungere alla soluzione, come il metodo di Newton modificato.

```
≫ [x,it,tolf]=aitken(0,'fxm','dfxm',1e-15,2000)

x =

     1


it =

     4
```

```
tolf =

     0

≫
```

il metodo di accelerazione di Aitken converge verso la soluzione in sole 4 iterazioni. Si tenga però presente che il metodo di Aitken è molto oneroso computazionalmente: infatti richiede due passi del metodo di Newton per poter generare un'approssimazione, è come se ad ogni passo del metodo di Aitken ne corrispondessero tre del metodo di Newton.

5.4 Metodi Quasi-Newtoniani

Da questo punto in avanti vedremo quelli che sono detti i metodi quasi-newtoniani. Un grande problema del metodo di Newton è dovuto al costo del calcolo della derivata prima della funzione in esame, derivata non sempre ottenibile per via analitica (magari per la sua eccessiva complessità). Si definisce allora un'altra funzione

$$\varphi(x) \approx f'(x)$$

che approssima la derivata prima di f ma che sia decisamente più semplice da calcolarsi.

I metodi quasi-newtoniani sono detti così proprio perché utilizzano un algoritmo simile a quello di Newton, solo che al posto della derivata utilizzano questa nuova funzione:

$$x_{k+1} = x_k - \frac{f(x_k)}{\varphi(x_k)}$$

in modo da diminuire la complessità.

5.5 Metodo delle corde

Nel metodo delle corde $\varphi(x)$ è una costante che può anche essere scelta come la derivata prima nel punto x_0 ($f'(x_0)$). Utilizzando questo valore costante come coefficiente angolare per le rette approssimatrici queste verranno ad essere tutte parallele fra loro.

E' questo un metodo lineare ed è molto più sensibile del metodo di Newton per quanto riguarda la scelta del punto iniziale.

5.5.1 Implementazione in Matlab

Ecco come è stato tradotto in codice Matlab il metodo delle corde

```
function [x,i,tolf]=corde(x0,m,f,tolx,nmax)
%CORDE Esegue il metodo delle corde, per la risoluzione di f(x)=0
%
%   [x,i,tolf]=CORDE(x0,m,f,tolx,nmax)
%
%   I parametri della funzione sono:
%       x0 -> il punto iniziale
%       m -> il coefficiente angolare che verrà mantenuto costante durante
%            tutto l'algoritmo
%       f -> funzione di cui valutare uno zero
%       tolx -> tolleranza per la radice
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
%
%   See Also NEWTON, SECANTI, STEFFENSEN
  i=0;
  err=tolx+1;
  x=x0;
  while (i<nmax & err>tolx)
      fx=feval(f,x);
      tolf=tolx*abs(m);
      if abs(fx)<=tolf
         break
      end
      x1=x-fx/m;
      err=abs(x1-x);
      x=x1;
      i=i+1;
  end
```

5.5.2 Sperimentazioni dell'algoritmo

Vediamo come la scelta del coefficiente angolare utilizzato dall'algoritmo influenzi la velocità di convergenza del metodo:

```
» [x,it,tolf]=corde(0,1,'fxcosx',1e-15,2000)

x =

   0.73908513321516
```

```
it =

    87


tolf =

    1.000000000000000e-015

≫
```

Figura 5.1: Grafico del metodo delle corde con $f(x) = x - cosx$, $m = 1$, $x_0 = 0$

```
≫ [x,it,tolf]=corde(0,2,'fxcosx',1e-15,2000)

x =

   0.73908513321516


it =

    20
```

```
tolf =

    2.000000000000000e-015

≫
```

Figura 5.2: Grafico del metodo delle corde con $f(x) = x - cosx$, $m = 2$, $x_0 = 0$

La scelta di $m = 2$ anzichè $m = 1$ porta ad una riduzione del numero delle iterazioni di un fattore superiore a 4; questo fatto è ben evidenziato dai grafici che mostrano l'andamento dell'algoritmo.

5.6 Metodo delle secanti

In questo metodo la derivata prima viene approssimata con il rapporto incrementale tra i due ultimi punti di approssimazione, x_{k-1} e x_k

$$\varphi(x) = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

E' un metodo in due passi: avendo bisogno di due approssimazioni per calcolare la funzione $\varphi(x)$ per far partire l'algoritmo si necessitano dei punti x_0 e x_1 e solitamente x_1 viene calcolato utilizzando il metodo di Newton (il primo passo) e vengono poi utilizzati questi due valori per far iniziare il metodo dell secanti (il secondo passo).

La convergenza del metodo è superlineare, in quanto

$$1 < p = \frac{1+\sqrt{5}}{2} < 2.$$

Potrebbero nascere dei problemi dal fatto che i fattori $f(x_k) - f(x_{k-1})$ e $x_k - x_{k-1}$ sono la differenza di quantità molto vicine tra loro e quindi risulta essere un problema mal condizionato, ed è proprio quello che impedisce al metodo di raggiungere un'efficienza elevata.

5.6.1 Implementazione in Matlab

Il codice Matlab che implementa il metodo appena visto

```
function [x,i,tolf]=secanti(x0,x1,f,tolx,nmax)
%SECANTI Esegue il metodo delle secanti, per la risoluzione di f(x)=0
%
%   [x,i,tolf]=SECANTI(x0,x1,f,tolx,nmax)
%
%   I parametri della funzione sono:
%       x0 -> il punto iniziale e prima approssimazione di x
%       x1 -> la seconda approssimazione della soluzione x
%       f -> funzione di cui valutare uno zero
%       tolx -> tolleranza per la radice
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
%
%   See Also NEWTON, CORDE, STEFFENSEN
  i=0;
  fx0=feval(f,x0);
  err=abs(x1-x0);
  while (i<nmax & err>tolx)
      fx1=feval(f,x1);
      dfx1=(fx1-fx0)/(x1-x0);
      tolf=tolx*abs(dfx1);
      if abs(fx1)<=tolf
         break
      end
      x2=x1-(fx1/dfx1);
      err=abs(x2-x1);
      x0=x1;
```

```
      x1=x2;
      fx0=fx1;
      i=i+1;
  end
  x=x1;
```

5.6.2 Sperimentazioni dell'algoritmo

Vediamo un esempio di utilizzo del metodo delle secanti dove come valore di x_1 viene passato l'approssimazione generata dal metodo di Newton:

```
» [x,it,tolf]=secanti(0,1,'fxcosx',1e-15,2000)

x =

   0.73908513321516


it =

     6


tolf =

    1.673398328690808e-015

»
```

5.7 Metodo di Steffensen

Dal momento che sappiamo $f(x_k) \to 0$ per $x_k \to \overline{x}$ possiamo utilizzare questo fatto per calcolare l'approssimazione della derivata come, da definizione, limite per $h \to 0$ del rapporto incrementale calcolato in $f(x_k)$, ed in questo caso possiamo utilizzare $f(x_k)$ al posto di h per ottenere

$$\varphi(x_k) = \frac{f(x_k + f(x_k)) - f(x_k)}{f(x_k)} \simeq \frac{f(x_k + h) - f(x_k)}{h}$$

si utilizza quindi la $f(x_k)$ come quel fattore infinitesimale utilizzato nel calcolo della derivata.

La convergenza di questo metodo è quadratica, ma ad ogni iterazione devono essere valutate due funzione $f(x_k)$ e $f(x_k + f(x_k))$ che lo porta ad avere un costo simile a quello del metodo di Newton il che, unito a prestazioni poco brillanti, lo rendono un metodo poco utilizzato.

5.7.1 Implementazione in Matlab

L'implementazione del metodo di Steffensen in Matlab

```
function [x,i,tolf]=steffensen(x0,f,tolx,nmax)
%STEFFENSEN Esegue il metodo di Steffensen, per la risoluzione di f(x)=0
%
%   [x,i,tolf]=STEFFENSEN(x0,f,tolx,nmax)
%
%   I parametri della funzione sono:
%       x0 -> il punto iniziale
%       f -> funzione di cui valutare uno zero
%       tolx -> tolleranza per la radice
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       x -> la soluzione trovata
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
%       tolf -> la tolleranza sulla funzione
%
%   See Also NEWTON, CORDE, SECANTI
  i=0;
  err=tolx+1;
  x=x0;
  phi=0;
  while(i<nmax & err>tolx)
      xx=x;
      fxk=feval(f,x);
      tolf=tolx*abs(phi);
      if abs(fxk)<=tolf
         break
      end
      fxk2=feval(f,x+fxk);
      phi=(fxk2-fxk)/fxk;
      x=xx-fxk/phi;
      err=abs(x-xx);
      i=i+1;
  end
```

5.7.2 Sperimentazioni dell'algoritmo

Vediamo all'opera il metodo di Steffensen sulla funzione $f(x) = x - cosx$:

```
» [x,it,tolf]=steffensen(0,'fxcosx',1e-15,2000)
```

```
x =

   0.73908513321516


it =

     8


tolf =

    1.673612064912526e-015

≫
```

5.8 Conclusioni

Vogliamo qui riassumere le sperimentazioni effettuate sui vari algoritmi in modo da poterne confrontare i risultati, un confronto altrimenti poco agevole.

Presentiamo prima il grafico della funzione e successivamente una tabella riassuntiva dei risultati.

Figura 5.3: Grafico della funzione $f(x) = x - cosx$

	Soluzione	Iterazioni	Tolleranza
Bisezione	0.739085133215	49	1.68750000000e-15
Newton	0.739085133215	5	1.67361202918e-15
Corde ($m = 1$)	0.739085133215	87	1.00000000000e-15
Corde ($m = 2$)	0.739085133215	20	2.00000000000e-15
Secanti	0.739085133215	6	1.67339832869e-15
Steffensen	0.739085133215	8	1.67361206491e-15

Come si vede tutti i metodi esaminati convergono verso la stessa soluzione (qui presentata con solo 12 cifre significative), peraltro molto prossima alla soluzione esatta, ed anche la tolleranza sulla funzione (anch'essa presentata con un ridotto numero di cifre decimali) risulta pressappoco la stessa; ciò che cambia, anche in modo vistoso, sono il numero di iterazioni necessarie perché l'algoritmo converga: se ad un estremo abbiamo il metodo delle corde con 87 iterazioni, causate da una cattiva scelta del coefficiente angolare, all'altro estremo abbiamo il metodo di Newton che si dimostra essere molto efficiente; non scordiamoci però che ogni passo di Newton ha costo doppio rispetto, ad esempio, a bisezione dovuto alla doppia valutazione di funzione. Come già ampiamente discusso, il metodo di bisezione è un metodo lento (lo testimoniano le sue 49 iterazioni) ma di sicura convergenza; il metodo delle secanti si candida come buon sostituto del metodo di Newton, per la sua efficacia e per la sua semplicità computazionale, nel caso il calcolo analitico della derivata della funzione in esame sia decisamente complesso. Un ultima parola per il metodo di Steffensen che, sebbene abbia un costo paragonabile al metodo di Newton, impiega molte più iterazioni (in questo caso oltre il

60% in più) di Newton per giungere alla soluzione; proprio questi due fatti lo rendono un metodo scarsamente utilizzato.

Figura 5.4: Grafico della funzione $f(x) = (x-5)^5$

	Soluzione	Iterazioni	Tolleranza
Bisezione	5.000000000000	43	2.94004118110e-65
Newton	4.999999999999	155	7.47072984107e-72
NewtonMolt	5	1	0
Aitken	5.000000000000	1	3.11150763893e-75

Si nota bene come il metodo di Newton classico risulti molto inefficiente in caso di radici multiple (impiega più di 3 volte le iterazioni del metodo di bisezione per arrivare all'approssimazione); conoscendo la molteplicità della radice, in questo caso 5, la modifica apportata al metodo di Newton, con una sola iterazione, arriva alla soluzione. Il metodo di Aitken, da parte sua, impiega sempre una sola iterazione, ma non necessita uno studio preventivo per determinare la molteplicità della radice.

Figura 5.5: Grafico della funzione $f(x) = (x-1)^4(x-2)$

	Soluzione	Iterazioni	Tolleranza
Newton	1.000000000000	117	2.77265512161e-58
NewtonMolt	1	5	0
Aitken	1	4	0

Questa tabella mostra ancora una volta l'inefficienza del metodo di Newton; nuovamente conoscendo la molteplicità 4 della radice $x = 1$ possiamo ripristinare la velocità del metodo di Newton che pertanto ritorna a valori accettabili di convergenza. Potrebbe sembrare che il metodo di Aitken sia decisamente performante, ma non dobbiamo dimenticare che ogni sua iterazione ha un costo superiore a tutti gli altri metodi a confronto.

Capitolo 6

Ricerca dell'autovalore dominante

Data una matrice $A \in \mathbf{C}^{n\times n}$ si dice autovalore di A ogni numero $\lambda \in \mathbf{C}$ tale che

$$Av = \lambda v \qquad v \in \mathbf{C}^n$$

con $v \neq 0$; ogni tale vettore v è detto autovettore associato all'autovalore λ.

Come sappiamo (Teorema Rouche-Capelli) un sistema lineare omogeneo ha soluzioni non nulle se e solo se la matrice dei coefficienti del sistema è singolare e cioè ha determinante nullo; poichè l'equazione precedente è equivalente al sistema

$$(A - \lambda I)x = 0$$

segue che gli autovalori siano tutti e soli i valori λ che soddisfano l'equazione

$$det(A - \lambda I) = 0$$

e dal calcolo esplicito del determinante abbiamo

$$det(A - \lambda I) = p_a(\lambda)$$

quello che si chiama polinomio caratteristico della matrice A; gli autovalori di una matrice $A \in \mathbf{C}^{n\times n}$ coincidono con le radici del polinomio $p_a(\lambda)$, che sono n e che perciò verranno indicate con $\lambda_1, \lambda_2, \cdots, \lambda_n$.

Calcolarsi gli autovalori risolvendo il polinomio caratteristico comporta il calcolo del determinante di una matrice, un'operazione molto costosa e spesso non necessaria. Esiste ad esempio il metodo QR che consente di calcolare l'intero spettro della matrice e che genera una successione di matrici che trasformano per similitudine la matrice data per ottenere una matrice triangolare superiore. Dato il suo costo molto elevato e la complessità dell'algoritmo non ce ne occuperemo, preferendo orientarci verso un problema simile.

Ci concentriamo infatti su un metodo per ottenere quello che è detto l'autovalore dominante, cioè l'autovalore di modulo massimo. Anche se non risolviamo il problema originario, e quindi non troviamo tutti gli autovalori della matrice A, la soluzione a questo problema è di grande interesse in molti problemi di applicazione reale, ad esempio per la geosismica, per lo studio delle vibrazioni di macchine e strutture ed addirittura in meccanica quantistica.

6.1 Il metodo delle potenze

Questo metodo è utilizzato proprio per risolvere il problema di determinare l'autovalore dominante; a questo scopo viene richiesto che

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|.$$

Un'altra ipotesi richiesta è che A sia diagonalizzabile, cioè che $\exists x_1, \cdots, x_n$ autovettori corrispondenti agli λ_i autovalori di A tali che siano linearmente indipendenti. In questo caso la matrice è detta diagonalizzabile e $x_1, \cdots, x_n$ formano una base per $\mathbb{C}^n$.

Il metodo procede come segue: dato

$$z_0 \in \mathbb{C}^n$$

un vettore qualunque, per quanto detto sopra, potrà essere scritto come

$$z_0 = \sum_{i=1}^{n} \alpha_i x_i \qquad \text{per} \quad \alpha_i \in \mathbb{C} \quad \text{opportuni;}$$

a questo punto, si genera la successione

$$\begin{cases} y_0 = z_0 \\ y_k = A y_{k-1} \end{cases}$$

Le proprietà di questa successione sono:

1. $y_k = A^k z_0$;

2. $y_k = \sum_{i=1}^{n} \alpha_i \lambda_i^k x_i$ infatti, vediamo per induzione che

$$k = 0 \qquad y_0 = \sum_{i=1}^{n} \alpha_i x_i = z_0$$

ora supponiamo vero l'asserto per $k-1$ e dimostriamolo per k:

$$\begin{aligned} y_k &= A y_{k-1} = A \sum_{i=1}^{n} \alpha_i \lambda_i^{k-1} x_i = \sum_{i=1}^{n} \alpha_i \lambda_i^{k-1} A x_i = \\ &= \sum_{i=1}^{n} \alpha_i \lambda_i^{k-1} \lambda x_i = \sum_{i=1}^{n} \alpha_i \lambda_i^k x_i \end{aligned}$$

Con $\alpha_1 \neq 0$ e sfruttando il fatto che λ_1 è dominante, si ottiene

$$y_k = \lambda_1^k \left[\alpha_1 x_1 + \sum_{i=2}^{n} \alpha_i \left(\frac{\lambda_i}{\lambda_1} \right)^k x_i \right] \overset{k \gg 1}{\longrightarrow} \lambda_1^k \alpha_1 x_1$$

($|\lambda_1| \neq 0$ poiche maggiore di qualcosa che al minimo è zero)

Notiamo che il vettore y_k tende ad allinearsi nella direzione di x_1, autovettore dominante.

Osserviamo adesso che

$$y_k^* y_k \simeq \left(\bar{\alpha}_1 \bar{\lambda}_1^k x_1^* \right) \left(\alpha_1 \lambda_1^k x_1 \right) = |\alpha_1|^2 |\lambda_1^k|^2 \|x\|_2^2$$

$$y_k^* y_{k+1} \simeq |\alpha_1|^2 |\lambda_1|^{2k} \lambda_1 \|x\|_2^2 = \lambda_1 y_k^* y_k$$

a questo punto possiamo definire

$$\sigma_k = \frac{y_k^* y_{k+1}}{y_k^* y_k} \to \lambda_1 \quad \text{per } k \to +\infty$$

Questo algoritmo oltre all'autovalore dominante trova anche l'autovettore associato, infatti y_k tende ad essere un autovettore corrispondente a λ_1.

Il metodo procede come segue:

$$(z_0 =) \underbrace{y_0 \longrightarrow y_1 = A}_{\sigma_0} \underbrace{y_0 \longrightarrow y_2 = Ay_1}_{\sigma_1} \quad \begin{matrix} \cdots \\ \longrightarrow \lambda_1 \end{matrix}$$

Implementando così l'algoritmo potrebbe dare origine ad errori di underflow ed overflow. Questi inconvenienti derivano dal fatto che per calcolare σ_k dobbiamo calcolare y_k e questo tende a $+\infty$ se $\lambda_1 > 1$ mentre tende a 0 se $\lambda_1 < 1$; l'underflow si può dire che sia un problema nascosto poichè in quel caso $y_k \to 0$ ma poi si vedrebbe che l'algoritmo non converge ad una approssimazione accettabile.

Allora si cercano delle modifiche all'algoritmo per ovviare a questi inconvenienti:

$$\begin{cases} \hat{y}_0 = z_0 \\ t_k = \frac{\hat{y}_k}{\|\hat{y}_k\|_2} & (\|t_k\|_2 = 1) \\ \hat{y}_{k+1} = At_k \end{cases}$$

La successione adesso procede come segue:

$$\begin{array}{ccc} \hat{y}_0 & & \hat{y}_1 \\ \downarrow & \nearrow & \downarrow \\ t_0 & & t_1 \end{array}$$

Aver normalizzato il vettore $\hat{y}_k$ rende At_k il prodotto di una matrice per un vettore di norma uno che non diventa mai troppo grande. Quello che interessa a noi è un'approssimazione di λ_1; si nota allora che

$$t_k = \frac{A^k z_0}{\|A^k z_0\|_2}$$

$$y_{k+1} = \frac{A^{k+1} z_0}{\|A^k z_0\|_2}$$

A partire da questi due valori si può scrivere

$$t_k^* A t_k = \frac{\left(A^k z_0\right)^* A \left(A^k z_0\right)}{\|A^k z_0\|_2^2}$$

$$t_k^* t_k = \frac{\left(A^k z_0\right)^* \left(A^k z_0\right)}{\|A^k z_0\|_2^2}$$

a questo punto siamo in grado di definite

$$\hat{\sigma}_k = \frac{t_k^* \hat{y}_{k+1}}{t_k^* t_k} = \frac{t_k^* A t_k}{t_k^* t_k} = \sigma_k \qquad \hat{\sigma}_k \to \lambda_1$$

esattamente uguale a σ_k di prima; ma per ottenere $\hat{\sigma}_k$ ho bisogno di t_k e $\hat{y}_{k+1}$, che numericamente non danno problemi. A questo punto il nuovo algoritmo è il seguente:

$$\left.\begin{array}{c} \hat{y}_0 = z_0 \\ t_0 = \frac{\hat{y}_0}{\|\hat{y}_0\|_2} \end{array}\right\} \longrightarrow \left.\begin{array}{c} \hat{y}_1 = A t_0 \\ t_1 = \frac{\hat{y}_1}{\|\hat{y}_1\|_2} \end{array}\right\} \longrightarrow \quad \cdots$$

e questo può essere implementato.

Come criteri di arresto possiamo prendere indifferentemente

$$|\hat{\sigma}_{k+1} - \hat{\sigma}_k| < toll \qquad \text{oppure}$$

$$\left|\frac{\hat{\sigma}_{k+1} - \hat{\sigma}_k}{\hat{\sigma}_k}\right| < toll$$

ma inoltre dobbiamo porre un limite massimo alle iterazioni; questo è dovuto al fatto che è richiesto che la matrice ammetta autovalore dominante, se non lo ammette l'algoritmo potrebbe non convergere. Se usciamo dall'algoritmo perché si è raggiunto il massimo numero di iterazioni concesse, si può dire che non si è raggiunta un'approssimazione di λ_1.

6.1.1 Implementazione in Matlab

Qui di seguito mostriamo il codice che implementa il metodo delle potenze:

```
function [lambda,i]=potenze(A,vett,toll,nmax)
%POTENZE Esegue il metodo delle potenze per il calcolo dell'autovalore
%   dominante della matrice A, se questo esiste
%
%   [lambda,i]=POTENZE(A,vett,toll,nmax)
%
%   I parametri della funzione sono:
%       A -> matrice di cui vogliamo calcolare l'autovalore dominante
%       vett -> vettore di partenza; solitamente rand(n,1)
%       toll -> tolleranza per l'autovalore
%       nmax -> limite superiore al numero di iterazioni
%
%   I valori di ritorno sono:
%       lambda -> l'autovalore dominante
%       i -> il numero di iterazioni impiegate per ottenere la soluzione
  i=0;
  err=toll+1;
  sig=0;
  y=vett;
  while (i<nmax & err>toll*abs(sig))
      i=i+1;
      t=y/norm(y);
      y=A*t;
      sig1=(t'*y)/(t'*t);
      err=abs(sig1-sig);
      sig=sig1;
  end
  lambda=sig;
```

6.1.2 Sperimentazioni dell'algoritmo

Ecco alcuni esempi di utilizzo del metodo delle potenze:

Proviamo innanzitutto che l'algoritmo funziona: lo proviamo cioè su una matrice di cui conosciamo lo spettro, per esempio una matrice diagonale: gli autovalori sono infatti gli elementi sulla diagonale.

```
» A

A =

    20     0     0
```

```
     0     5     0
     0     0     3

» [l,i]=potenze(A, rand(3,1), 10e-15, 2000)

l =

  20.00000000000000


i =

    14
»
```

Chiaramente l'autovalore dominante era l'elemento $a_{1,1}$ con valore 10; l'algoritmo è riuscito a trovare questo elemento in 14 passi e con la precisione richiesta.

La prova successiva consiste in una matrice i cui elementi sono interi, ma scelti casualmente:

```
» A

A =

    45     2     9     0
     0     9     8    50
     1     2     3     4
     6     9     1     3

» [l,i]=potenze(A, rand(4,1), 10e-15, 2000)

l =

  46.05467285995372


i =

    55

» eig(A)

ans =
```

```
  46.05467285995431
 -15.19630313772274
  26.92844782249211
   2.21318245527628

≫
```

Vista la casualità della matrice per avere una riprova che l'algoritmo trovasse effettivamente l'autovalore di modulo massimo, si è utilizzato il comando "eig" che Matlab fornisce per il calcolo dello spettro di una matrice; come si vede il metodo delle potenze trova una buona approssimazione dell'autovalore dominante. Si tenga anche presente che il numero di passi necessari per avere una soluzione e la precisione di questa dipendono dal vettore iniziale: essendo scelto casuale questi valori possono cambiare, a volte anche in modo rilevante.

Naturalmente il metodo delle potenze funziona anche in caso di matrici e vettori iniziali complessi, come si nota nel seguente esempio:

```
≫ A

A =

 10     0     0         0
  0     3     0         0
  0     0     0 + i     0
  0     0     0         0 - i

≫ [l,i]=potenze(A, rand(4,1)+i*rand(4,1), 10e-15, 2000)

l =

  10.00000000000000


i =

    16

≫
```

Abbiamo provato un semplice esempio di una matrice diagonale con elementi complessi, ed anche il vettore casuale di partenza è stato scelto complesso.

Proviamo adesso un caso in cui l'algoritmo non converge:

```
≫ A

A =

     4   141  -576   432
     1     0     0     0
     0     1     0     0
     0     0     1     0

≫ [l,i]=potenze(A, rand(4,1), 10e-15, 2000)

l =

   3.09646771834061


i =

        2000

≫ eig(A)

ans =

 -12.00000000000000
  12.00000000000000
   3.00000000000000
   1.00000000000000

≫
```

Osservando l'output del comando "eig" ci accorgiamo subito dove risiede il problema: ci sono due autovalori che hanno modulo massimo; in questo caso "potenze" impiega tutte le 2000 iterazioni concesse senza raggiungere un'approssimazione accettabile. Questo è dovuto al fatto che durante la costruzione dell'algoritmo avevamo supposto che esistesse uno ed un solo autovalore dominante, qui ce ne sono due e l'algoritmo non riesce a convergere. Un'altra cosa da notare è la forma della matrice che abbiamo utilizzato per quest'ultimo test: una matrice siffatta si chiama matrice di compagnia e vedremo tra poco come si possono costruire matrici di questo tipo.

6.2 Costruzione di matrici noto lo spettro

Si vuole costruire una matrice $A \in \mathbb{R}^{n\times n}$ che abbia uno spettro noto, $\sigma(A) = \{\lambda_1, \cdots, \lambda_n\}$. A questo punto si presentano due casi:

1. $\sigma(A) \subset (R)$;
2. $\sigma(A) \not\subset (R)$..

Analizziamoli separatamente.

6.2.1 Il caso $\sigma(A) \subset \mathbb{R}$

Se $\sigma(A) \subset (R)$ possiamo costruire la matrice A come una matrice simile a quella diagonale che contiene sulla diagonale proprio gli autovalori, vediamo come:

$$A = Q \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} Q^T \qquad \text{con } Q \text{ ortogonale}$$

e per calcolare Q utiliziamo una tecnica già vista: sia v un vettore qualsiasi, allora

$$Q = I - 2\frac{vv^T}{v^Tv}$$

è proprio la matrice di Householder che sappiamo essere ortogonale.

Un altro modo per calcolare Q potrebbe essere quello di generare una matrice casuale e poi applicare su di essa la fattorizzazione QR e da qui prendere la nostra Q, ma come metodo è decisamente costoso.

6.2.2 Sperimentazioni nel caso $\sigma(A) \subset \mathbb{R}$

Quello che andremo a fare è seguire passo passo quanto indicato sopra per ottenere una verifica che il metodo sia corretto:

```
» v=[10;7;12], Q=eye(3)-2*v*v'/(v'*v)

v =

    10
     7
    12


Q =

   0.31740614334471  -0.47781569965870  -0.81911262798635
```

```
  -0.47781569965870   0.66552901023891  -0.57337883959044
  -0.81911262798635  -0.57337883959044   0.01706484641638

» Q*Q'

ans =

   1.00000000000000                   0   0.00000000000000
                  0   1.00000000000000  -0.00000000000000
   0.00000000000000  -0.00000000000000   1.00000000000000

» A=Q*diag([40;20;10])*Q'

A =

  15.30547822339223  -7.72985125045137  -5.06004729233887
  -7.72985125045137  21.27852391990588   7.92554368717166
  -5.06004729233887   7.92554368717166  33.41599785670188

» [l,i]=potenze(A, rand(3,1), 10e-15, 2000)

l =

  39.99999999999992


i =

    24

»
```

Si è costruito un vettore v di tre elementi, dopo di che si è costruito la matrice Q di Householder, che sappiamo essere ortogonale, e quindi come riprova abbiamo moltiplicato Q per la sua trasposta per verificare che produca come risultato la matrice identita, e così è; successivamente abbiamo prodotto la matrice A come trasformazione per similitudine della matrice diagonale che contiene i nostri autovalori, ed infine abbiamo applicato il metodo delle potenze a questa matrice. Il risultato, come si vede, è pienamente soddisfacente.

6.2.3 Il caso $\sigma(A) \not\subset \mathbb{R}$ e le matrici di compagnia

In questo caso si usano quelle che sono dette matrici di compagnia, vediamo la loro struttura:

$$A = \begin{pmatrix} -a_{n-1} & -a_{n-2} & \cdots & \cdots & -a_0 \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix}$$

praticamente una matrice con tutti valori nulli tranne la prima sottodiagonale e la prima riga, che contiene dei valori generici.

Proposizione. *Se A è una matrice di compagnia allora*

$$P_A(\lambda) = (-1)^n \left(\lambda^n + a_{n-1}\lambda^{n-1} + \cdots + a_1\lambda + a_0\right)$$

Facciamo un esempio: considerando la seguente matrice

$$A^{(1)} = \begin{pmatrix} -a_2 & -a_1 & -a_0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

calcoliamone adesso il polinomio caratteristico come il determinante di $A^{(1)}$ rispetto alla terza colonna:

$$\begin{aligned} P_{A^{(1)}}(\lambda) &= det \begin{pmatrix} -a_2-\lambda & -a_1 & -a_0 \\ 1 & -\lambda & 0 \\ 0 & 1 & -\lambda \end{pmatrix} = (-1)^4(a_0)\begin{vmatrix} 1 & -\lambda \\ 0 & 1 \end{vmatrix} + \\ &+ (-1)^6(-\lambda)\begin{vmatrix} -a_2-\lambda & -a_1 \\ 1 & -\lambda \end{vmatrix} = -a_2 + (-\lambda)((-a_2-\lambda)(-\lambda)+a_1) = \\ &= -(\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0) \end{aligned}$$

Detto questo possiamo costruire una matrice A con $P_A(\lambda)$ fissato: infatti dato $P(\lambda)$ di grado n, e cioè nella forma

$$P(\lambda) = a_n\lambda^n + a_{n-n}\lambda^{n-1} + \cdots + a_0$$

possiamo scrivere la matrice

$$A = \begin{pmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \cdots & \cdots & -\frac{a_0}{a_n} \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{pmatrix} \quad \text{con} \quad P_A(\lambda) = \frac{(-1)^n}{a_n}P(\lambda)$$

Un paio di osservazioni sono d'obbligo:

1. Con la matrice A sopra definita, se avessimo $B = A - \mu I$ conosceremmo anche gli autovalori di B, infatti sia $\lambda \in \sigma(A)$ e sia v l'autovettore corrispondente allora

$$Bv = (A - \mu I)v = Av - \mu v = (\lambda - \mu)v$$

e quindi v e autovettore anche di B, oltre che di A, e l'autovalore corrispondente è $\lambda - \mu$.

2. Se avessimo $B = A^k$ allora sia ancora $\lambda \in \sigma(A)$ e v l'autovettore corrispondente allora

$$Bv = \lambda^k v$$

facilmente verificabile per induzione.

6.2.4 Sperimentazioni per le matrici di compagnia

Faremo qui vedere alcuni esempi che riguardano l'utilizzo delle matrici di compagnia.

Matlab fornisce alcune funzioni che ci sono utili per ottenere una matrice di compagnia: la funzione "poly", che riceve un vettore contenente gli autovalori, restituisce un vettore contenente i coefficienti del polinomio che ha come radici i valori contenuti nel vettore di input; la funzione "compan" genera la matrice di compagnia utilizzando i valori del vettore in ingresso per calcolare il valore degli elementi della prima riga.

Vediamo un utilizzo:

```
≫ v=poly([17;3;12])

v =

     1   -32   291  -612

≫ A=compan(v)

A =

    32  -291   612
     1     0     0
     0     1     0

≫ eig(A)

ans =

  16.99999999999995
```

```
  12.00000000000003
   3.00000000000000

» [l,i]=potenze(A, rand(3,1), 10e-15, 2000)

l =

  17.00000000000039


i =

    88

»
```

Quello che abbiamo fatto è stato generare il vettore "v" dei coefficienti del polinomio avente come radici $17, 12, 3$ tramite la funzione "poly", dopo di che abbiamo costruito la matrice di compagnia con la funzione "compan"; per essere certi che gli autovalori della matrice A fossero proprio gli elementi del vettore parametro di "poly" abbiamo richiamato nuovamente la funzione "eig" che ci ha confermato la bontà delle operazioni svolte fin ora; infine abbiamo eseguito il nostro codice ottenendo il risultato desiderato. Si vuol far notare anche come l'algoritmo, eseguito numerose volte sullo stesso esempio, abbia mantenuto praticamente sempre lo stesso comportamento: converge verso la medesima soluzione $\lambda = 17.00000000000039$ in un numero costante di passi, 88, anche con valori del vettore iniziale sempre diversi.

Ora vogliamo evidenziare una ben precisa caratteristica delle matrici, ma per fare questo proponiamo prima due esempi per poi trarne le conclusione solo dopo un loro esame:

Primo esempio:

```
» v=poly([-1,2,-4,3,1+i,1])

v =

  Columns 1 through 2

  1.00000000000000                      -2.00000000000000 - i

  Columns 3 through 4

-14.00000000000000 + 1.00000000000000i 40.00000000000000 + 15i
```

```
  Columns 5 through 6

-11.00000000000000 -25.000000000000000i-38.00000000000000 -14i

  Column 7

 24.00000000000000 +24.000000000000000i

≫ A=compan(v)

A =

  Columns 1 through 2

  2.00000000000000 + 1.000000000000000i 14.00000000000000 - i
  1.00000000000000                                         0
                 0                       1.00000000000000
                 0                                         0
                 0                                         0
                 0                                         0

  Columns 3 through 4

-40.00000000000000 -15.000000000000000i 11.00000000000000 +25i
                 0                                         0
                 0                                         0
  1.00000000000000                                         0
                 0                       1.00000000000000
                 0                                         0

  Columns 5 through 6

 38.00000000000000 +14.000000000000000i-24.00000000000000 -24i
                 0                                         0
                 0                                         0
                 0                                         0
                 0                                         0
  1.00000000000000                                         0

≫ [l,i]=potenze(A, rand(6,1), 10e-15, 2000)

l =

 -4.00000000000001 + 0.000000000000001i
```

```
i =

   121

≫
```

Il secondo esempio:

```
≫ v=poly([-1,2,-4,3,1+i,1-i])

v =

     1    -2   -13    40   -26   -28    48

≫ A=compan(v)

A =

     2    13   -40    26    28   -48
     1     0     0     0     0     0
     0     1     0     0     0     0
     0     0     1     0     0     0
     0     0     0     1     0     0
     0     0     0     0     1     0

≫ [l,i]=potenze(A, rand(6,1), 10e-15, 2000)

l =

  -3.99999999999998


i =

   113

≫
```

Tralasciando la brutta forma del primo esempio, veniamo a quanto ci interessa: se si guarda con attenzione il vettore che contiene gli autovalori nel primo esempio si nota un numero complesso, mentre nel secondo ce ne sono due; di più, i due numeri complessi del secondo esempio non sono

slegati tra di loro, ma anzi, sono uno il complesso coniugato dell'altro. Se si riguarda ancora la matrice del primo esempio si nota come questa sia una matrice complessa, mentre la matrice del secondo esempio è una matrice reale. In ogni caso l'algoritmo converge alla soluzione attesa. Quello che si voleva far notare è che la presenza di valori complessi, uniti ai loro complessi coniugati, tra gli autovalori rende la matrice reale, mentre la presenza di numeri complessi senza i coniugati rende la matrice complessa.

Abbiamo utilizzato le matrici di compagnia per verificare quello che può essere formulato come

Teorema. *Se A è reale e λ è un suo autovalore, allora $\overline{\lambda} \in \sigma(A)$*