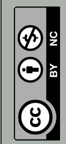
Appunti di calcolo numerico

Paolo Caressa

1999

Questa opera è distribuita con licenza *Creative Commons Attribuzione-Non commerciale 3.0 Unported.*
This work is licensed under a *Creative Commons Attribution-Non Commercial 3.0* Unported License.



Queste sono le note da me redatte ad uso personale per un corso di Calcolo Numerico nell'ambito del programma di Dottorato di Ricerca a Firenze; il corso è stato tenuto dal Prof. Aldo Pasquali usando essenzialmente il libro di Monegato *Fondamenti di calcolo numerico* consultato nella redazione di queste note; i programmi che implementano gli algoritmi sono stati da me sviluppati in ANSI C.

1 Eliminazione di Gauss

Supponiamo di voler risolvere il sistema lineare

$$Ax = b$$

ove $A \in M_n(\mathbb{R})$ è una matrice quadrata, $b \in \mathbb{R}^n$ un vettore costante e $x \in \mathbb{R}^n$ il vettore delle incognite del sistema; scriveremo sempre

$$A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}$$

e $b = (b_1, b_2, ..., b_n)^T$.

Ammettiamo che A sia invertibile: questo implica, ad esempio, che non esiste una colonna di soli zeri in A, e quindi che, a meno di permutare le righe (il che vuol dire scambiare le equazioni del sistema), possiamo sempre supporre $a_{ii} \neq 0$ per ogni $i = 1, 2, ..., n$.

Consideriamo per prima cosa il caso $n = 1$: questo è banale, poiché il sistema si riduce ad una sola equazione

$$a_{11}x_1 = b_1$$

la cui soluzione è ovviamente $x_1 = b_1/a_{11}$ ($a_{11} \neq 0$ perché la matrice (a_{11}) è appunto non singolare).

Nel caso $n = 2$ abbiamo

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases}$$

Allora, se $a_{11} \neq 0$ possiamo sostituire la seconda equazione con una sua combinazione lineare del tipo:

$$a_{21}x_1 + a_{22}x_2 = b_2 \mapsto (a_{21}x_1 + a_{22}x_2 = b_2) - \frac{a_{21}}{a_{11}}(a_{11}x_1 + a_{12}x_2 = b_1)$$

ottenendo cioè il sistema

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ \qquad\quad \left(a_{22} - \dfrac{a_{21}}{a_{11}}a_{12}\right)x_2 = b_2 - \dfrac{a_{21}}{a_{11}}b_1 \end{cases}$$

Ora la seconda equazione si risolve immediatamente come nel caso $n = 1$:

$$x_2 = \frac{a_{11}b_2 - a_{21}b_1}{a_{22}a_{11} - a_{21}a_{12}}$$

(di nuovo il denominatore è diverso da zero in quanto la matrice del sistema è invertibile: il denominatore è esattamente il determinante di questa matrice.)

Sostituendo questo valore di x_2 nella prima equazione ci riconduciamo ad una equazione in una sola incognita x_1 che sappiamo già risolvere.

Per $n > 2$ l'idea è di usare questo procedimento di eliminazione in modo ricorsivo: supponiamo cioè di saper risovere un sistema di $n - 1$ equazioni in $n-1$ incognite con matrice invertibile: allora dato il sistema $Ax = b$ possiamo trovare, dato che la prima colonna non può esser nulla in virtù dell'invertibilità della matrice A, un elemento $a_{1i} \neq 0$ per qualche $i \in \{1, 2, ..., n\}$; scambiare di posto l'equazione i-esima con la prima non cambia ovviamente il sistema, e quindi possiamo farlo.

Ora consideriamo i numeri

$$m_2 = -\frac{a_{21}}{a_{11}},\ m_3 = -\frac{a_{31}}{a_{11}},\ ...,\ m_n = -\frac{a_{n1}}{a_{11}}$$

Ovviamente $a_{i1} + m_i a_{11} = 0$ e quindi sommando alla i-esima equazione la prima moltiplicata per m_i troviamo una equazione il cui coefficiente della variabile x_1 è zero. Fatto questo il sistema diviene

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ... + a_{1n}x_n = b_1 \\ \qquad a'_{22}x_2 + a'_{23}x_3 + ... + a'_{2n}x_n = b'_2 \\ \qquad a'_{32}x_2 + a'_{33}x_3 + ... + a'_{3n}x_n = b'_3 \\ \qquad \dots\dots\dots\dots\dots\dots\dots\dots \\ \qquad a'_{n2}x_2 + a'_{n3}x_3 + ... + a'_{nn}x_n = b'_n \end{cases}$$

ove $a'_{ij} = a_{ij} + m_i a_{1j}$ e $b'_i = b_i + m_i$.

Se ora escludiamo la prima equazione troviamo un sistema di $n-1$ equazioni in $n-1$ incognite, che abbiamo supposto di saper risolvere (la matrice di questo sistema è ancora non singolare perché il suo determinante non è cambiato se non nel segno, a causa dello scambio iniziale della riga i-sima con la prima, e per il fattore moltiplicativo a_{11}): dunque abbiamo la soluzione $(\xi_2, ..., \xi_n)$ di questo sotto-sistema, e, sostituendo questi valori al posto delle incognite $(x_2, ..., x_n)$ della prima equazione troviamo una equazione nella sola x_1, che possiamo risolvere come nel caso $n = 1$.

Infine, aggiungendo la soluzione di questa equazione alla soluzione del sotto-sistema $(n-1) \times (n-1)$ troviamo la soluzione del sistema di partenza.

Nello scambio di equazioni che si effettua per risolvere il sistema generale, l'idea è di sostituire uno zero con qualcosa di non nullo: sembra quindi naturale (ed in effetti garantisce maggior stabilità numerica) sostituire questo zero (cioè a_{11}) con il massimo numero (in valore assoluto) disponibile; poiché si tratta di scegliere un indice di riga i, sceglieremo l'equazione i-esima dove

$$|a_{i1}| = \max_{1 \le j \le n} |a_{j1}|$$

Questo modo di scegliere la riga da scambiare si chiama *pivoting parziale*.

Un altro metodo di effettuare la scelta è scegliere non solo per righe ma anche per colonne, cioè andare a cercare la coppia (i, j) tale che

$$|a_{ij}| = \max_{1 \le k, l \le n} |a_{kl}|$$

e scambiare la riga i-esima con la k-esima e la colonna j-esima con la l-esima: questo modo di procedere si chiama *pivoting completo*.

Volendo implementare l'algoritmo di eliminazione su una macchina numerica il *pivoting* completo, per quanto teoricamente migliore, può risultare assai più dispendioso del *pivoting* parziale.

Il condizionamento del problema della soluzione di un sistema di equazioni lineari dipende dalla matrice del sistema: supponiamo ad esempio di perturbare il vettore dei termini noti b di una quantità vettoriale δb e di risolvere il sistema $Ax' = b + \delta b$ in modo esatto: allora $x' = A^{-1}b + A^{-1}\delta b = x + \delta x$, cioè $\delta x = A^{-1}\delta b$. Per stimare gli ordini di grandezza scegliamo una norma matriciale compatibile con una norma vettoriale: allora

$$\|\delta x\| = \|A^{-1}\delta b\| \leq \|A^{-1}\|\,\|\delta b\|$$

cioè

$$\frac{\|\delta x\|}{\|\delta b\|} \leq \|A^{-1}\|$$

Ma $\|b\| \leq \|A\|\,\|x\|$, cioè $(\|A\|\,\|x\|)^{-1} \leq \|b\|^{-1}$, da cui

$$\frac{\|\delta x\|}{\|A\|\,\|x\|} \leq \|A^{-1}\|\frac{\|\delta b\|}{\|b\|}$$

Ne segue che

$$\frac{\|\delta x\|}{\|x\|} \leq K(A)\frac{\|\delta b\|}{\|b\|}$$

ove $K(A) = \|A\|\,\|A^{-1}\|$. Quindi il condizionamento del problema dipende dalla quantità $K(A)$; supponendo di perturbare anche la matrice A, avremo

$$\begin{aligned}(A + \delta A)(x + \delta x) = b + \delta b &\implies A\delta x + \delta Ax + \delta A\delta x = \delta b\\ &\implies (I + A^{-1}\delta A)\delta x = A^{-1}(\delta b - \delta Ax)\\ &\implies \delta x = (I + A^{-1}\delta A)^{-1}A^{-1}(\delta b - \delta Ax)\end{aligned}$$

(supponendo che $I + A^{-1}\delta A$ sia invertibile: poiché le matrici invertibili sono definite da un'equazione $\det A \neq 0$ sono un aperto denso nello spazio delle matrici, quindi esiste sempre una perturbazione dello stesso ordine di grandezza di δA che rende invertibile questa matrice.)

Ora vogliamo stimare

$$\begin{aligned}\frac{\|\delta x\|}{\|x\|} =&\frac{\|(I + A^{-1}\delta A)^{-1}A^{-1}(\delta b - \delta Ax)\|}{\|x\|} \leq\\ \leq&\|(I + A^{-1}\delta A)^{-1}\|\,\|A^{-1}\|\left(\frac{\|\delta b\|}{\|x\|} + \|\delta A\|\right)\end{aligned}$$

Supponiamo per questo che

$$\|\delta A\| \leq \frac{1}{\|A^{-1}\|}$$

Allora

$$\begin{aligned}\frac{\|\delta x\|}{\|x\|} \leq & \frac{\|A^{-1}\|}{\|I + A^{-1}\delta A\|}\left(\frac{\|\delta b\|}{\|x\|} + \|\delta A\|\right) \leq \frac{\|A^{-1}\|}{1 - \|A^{-1}\delta A\|}\left(\frac{\|\delta b\|}{\|x\|} + \|\delta A\|\right) \\ \leq & \frac{\|A^{-1}\|}{1 - \|A^{-1}\|\|\delta A\|}\left(\frac{\|\delta b\|}{\|x\|} + \|\delta A\|\right)\end{aligned}$$

Ora teniamo conto che $\|b\| \leq \|A\|\|x\|$:

$$\begin{aligned}\frac{\|\delta x\|}{\|x\|} \leq & \frac{\|A^{-1}\|}{1 - \|A^{-1}\|\|\delta A\|}\left(\|A\|\frac{\|\delta b\|}{\|b\|} + \|\delta A\|\right) \\ \leq & \frac{\|A^{-1}\|\|A\|}{1 - \|A^{-1}\|\|\delta A\|}\left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|}\right) \\ \leq & \frac{K(A)}{1 - K(A)\frac{\|\delta A\|}{\|A\|}}\left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|}\right)\end{aligned}$$

Quindi il condizionamento del problema dipende essenzialmente da $K(A)$: ad esempio vale il

Teorema 1.1 *Se A è una matrice invertibile allora $1/K(A)$ è la distanza relativa (rispetto ad una norma compatibile con una vettoriale) di A dall'insieme delle matrici singolari, cioè*

$$\frac{1}{K(A)} = \min\left\{\frac{\|A - B\|}{\|A\|} \mid B \text{ singolare}\right\}$$

2 Decomposizione di Gauss

La tecnica dell'eliminazione consiste nel trasformare un sistema in un sistema la cui matrice sia triangolare: il legame fra la matrice di partenza e quella così ottenuta fa parte di un profondo risultato della teoria dei gruppi di Lie, noto come *decomposizione di Iwasawa*, ma che, nel caso da noi considerato di $A \in GL_n(\mathbb{R})$, risale appunto a Gauss.

L'idea è semplice: per risolvere il sistema $Ax = b$ abbiamo manipolato la matrice A (ed il vettore b) sostituendo a delle sue righe combinazioni lineari di righe, e scambiando delle righe. Queste operazioni si possono esprimere in termini di moltiplicazione di matrici.

Ad esempio supponiamo di voler scambiare la riga i-esima di una matrice (quadrata) A con la riga j-esima di A: la matrice che ne risulta è $P_{ij}A$ ove

$$P_{ij} = \begin{pmatrix} 1 & 0 & 0 & ... & 0 & ... & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 & ... & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 & ... & 0 & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots & ... & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & 0 & ... & 1 & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots & ... & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & 1 & ... & 0 & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots & ... & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & 0 & ... & 0 & ... & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \\ \end{matrix}$$
$$\begin{matrix} \uparrow & \quad & \uparrow \\ i & & j \end{matrix}$$

(le frecce indicano le colonne e le righe relative agli indici riportati.)

La proprietà che caratterizza la matrice P_{ij} è esattamente che quando si moltiplica a sinistra una matrice A per P_{ij} il risultato è la matrice $P_{ij}A$ ottenuta da A scambiando la riga i-esima con la riga j-esima; se invece si moltiplica A per P_{ij} a destra, la matrice AP_{ij} che ne risulta è ottenuta da A scambiando la colonna i-esima con la colonna j-esima.

Allora l'operazione di scambiare l'equazione i-esima con la j-esima nel sistema equivale a considerare il nuovo sistema

$$P_{ij}A = P_{ij}b$$

le cui soluzioni coincidono con quelle di $Ax = b$.

Oltre a scambiare equazioni, abbiamo anche sostituito una equazione i-esima con se stessa sommata al multiplo scalare m di un'equazione j-esima: questa trasformazione si ottiene moltiplicando per la matrice

$$M_{ij}(m) = \begin{pmatrix} 1 & 0 & 0 & ... & 0 & ... & 0 & ... & 0 \\ 0 & 1 & 0 & ... & 0 & ... & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 & ... & 0 & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots & ... & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & 1 & ... & 0 & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots & ... & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & m & ... & 1 & ... & 0 \\ \vdots & \vdots & \vdots & ... & \vdots & ... & \vdots & ... & \vdots \\ 0 & 0 & 0 & ... & 0 & ... & 0 & ... & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \\ \end{matrix}$$
$$\begin{matrix} \uparrow & \quad & \uparrow \\ i & & j \end{matrix}$$

Questa matrice si può sinteticamente esprimere come

$$M_{ij}(m) = I + mE_{ij}$$

ove E_{ij} è la matrice che all'incrocio fra la riga i-esima e la riga j-esima ha un 1, mentre in tutte le altre entrate ha zero (cioè un elemento della base canonica dello spazio delle matrici).

Sia P_{ij} che $M_{ij}(m)$ sono invertibili (per ogni i, j, m): infatti

$$P_{ij}P_{ij} = I_n$$

perché scambiando due volte i con j non si compie nessuna operazione, sicché $P_{ij} = P_{ij}^{-1}$; notiamo in particolare che $P_{ii} = I_n$ per ogni $i = 1, ..., n$.

Inoltre

$$M_{ij}(m)M_{ij}(-m) = I_n$$

dato che sommando alla riga i-esima la j-esima moltiplicata per m e poi sommando a questo risultato la j-esima moltiplicata per $-m$ otteniamo di nuovo la i-esima riga. Notiamo ancora che $M_{ii}(0) = I_n$ per ciascun $i = 1, ..., n$.

Ora: il procedimento di Gauss può semplicemente interpretarsi come la moltiplicazione a sinistra di ambo i membri dell'equazione $Ax = b$ per matrici P_{ij} e $M_{ij}(m)$, ottenendo

$$GAx = Gb$$

ove abbiamo posto

$$G = M_{k_{n-1}l_{n-1}}(m_{k_{n-1}l_{n-1}})P_{(n-1)j_{n-1}}...M_{k_2l_2}(m_{k_2l_2})P_{2j_2}M_{k_2l_2}(m_{k_2l_2})P_{1j_1}$$

avendo compiuto $n-1$ passi con opportuni indici i, j, k, l scelti col metodo del pivoting parziale ad esempio (ovviamente se non è necessario alcuno scambio al passo h poniamo ad esempio $i_h = j_h = 1$ in modo che $P_{i_hj_h} = I_n$).

La matrice $T = GA$ è triangolare superiore per costruzione, ed è invertibile in quanto prodotto di matrici invertibili: la fattorizzazione $A = G^{-1}T$ si dice *decomposizione di Gauss*.

Ovviamente per risolvere il sistema non è necessario determinare G ma solo le operazioni elementari cui corrispondono le matrici P_{ij} e $M_{ij}(m)$ che, per moltiplicazione, danno luogo ad essa.

Stabiliamo ora un algoritmo che permetta di determinare le operazioni elementari da effettuare su una matrice A per ridurla in forma triangolare superiore T; non avremo bisogno di sapere quale sia G, ma semplicemente, ad ogni passo, ci basta sapere quale sia P_{ij} e quale sia $M_{ij}(m)$.

Per determinare P_{ij} basta dare una coppia di numeri interi (i, j) e questo può farsi scrivendo un vettore $p = (p_1, ..., p_{n-1})$ con $n - 1$ componenti intere

(tante quanti i passi da compiere), tale che $p_i = j$. Ad esempio, se al passo h-esimo si devono scambiare la riga h e la riga j allora $p(h) = j$.

Per determinare $M_{ij}(m_{ij})$ basta dare il numero m_{ij} e la sua posizione (i, j); poiché queste posizioni non possono coincidere per matrici diverse, possiamo metterli tutti in una matrice triangolare inferiore $(i > j)$ nel modo seguente

$$\begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ m_{21} & 0 & 0 & \dots & 0 \\ m_{31} & m_{32} & 0 & \dots & 0 \\ m_{41} & m_{42} & m_{43} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \dots & 0 \end{pmatrix}$$

Notiamo infine che possiamo calcolare il determinante di A durante il processo di decomposizione di Gauss, dato che

$$\det G = \det \left[M_{k_{n-1}l_{n-1}}(m_{k_{n-1}l_{n-1}}) P_{(n-1)j_{n-1}} \dots M_{k_2l_2}(m_{k_2l_2}) P_{2j_2} M_{k_2l_2}(m_{k_2l_2}) P_{1j_1} \right]$$

Ma $\det M_{ij}(m) = 1$ e $\det P_{ij} = \pm 1$ ove il segno $+$ vale se e solo se $i = j$; inoltre

$$\det G \det A = \det T$$

e, dato che T è triangolare, il suo determinante è il prodotto degli elementi diagonali $t_{11}, \dots, t_{nn}$; morale:

$$\det A = (-1)^s \prod_{i=1}^{n} t_{ii}$$

ove s è il numero complessivo di scambi effettuati in tutto il processo.

Possiamo finalmente dare un algoritmo per la fattorizzazione di Gauss, che economizzerà gli spazi di memorizzazione nel modo seguente: la matrice A, che è data in input, verrà sovrascritta nel seguente modo

$$A \mapsto \begin{pmatrix} t_{11} & t_{12} & t_{13} & \dots & t_{1n} \\ m_{21} & t_{22} & t_{23} & \dots & t_{2n} \\ m_{31} & m_{32} & t_{33} & \dots & t_{3n} \\ m_{41} & m_{42} & m_{43} & \dots & t_{4n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & \dots & t_{nn} \end{pmatrix}$$

ove m_{ij} sono i coefficienti delle matrici $M_{ij}(m_{ij})$ collocati nelle posizioni giuste e t_{ij} sono i coefficienti della matrice triangolare T.

In ANSI C il tutto si esprime come[1]:

```
 1:/* deve essere definita una costante N che denota il numero di equazioni */
 2:/*
 3:   Algoritmo di fattorizzazione di Gauss per una matrice A usando il pivoting
 4:   parziale come scelta per gli scambi di riga: la matrice T ed i dati delle
 5:   matrici M_ij(m) sono memorizzati al posto di A, ed il vettore dei pivot
 6:   contiene gli scambi di riga effettuati nei vari passi; la funzione ritorna
 7:   il determinante di A: se è zero, il risultato in A è da considerarsi
 8:   indeterminato.
 9:*/
10:float factor(
11:   float A[N][N],                  /* matrice da fattorizzare e risultato */
12:   int pivot[N-1]                                    /* vettore dei pivot */
13:   )
14:{
15:   float   a;                                    /* variabile ausiliaria */
16:   float   d = 1;                                        /* determinante */
17:   int     i, j, k;                                   /* indici di ciclo */
18:   int     i0;                                /* indice per gli scambi */
19:   extern float abs_max( float A[N][N], int *i0, int k);
20:          /* abs_max  calcola max_{k<=i<=N} |a_ik| e torna in i0 il più piccolo degli
21:             indici per i quali il massimo è assunto da |a_ik| */

22:   for ( k = 0; k < N-1; ++ k ) {
23:      a = abs_max(A, &i0, k);                   /* i0 passato per indirizzo */
24:      pivot[k] = i0;
25:      if ( a == 0 ) return 0;               /* errore: matrice singolare */
26:      if ( i0 != k ) {  /* scambia la riga k-esima con la riga i0-esima */
27:         for ( i = k; i < N; ++ i ) {
28:            a = A[k][i];
29:            A[k][i] = A[i0][i];
30:            A[i0][i] = a;
31:         }
32:         /* cambia segno al determinante (ha effettuato uno scambio!) */
33:         d = - d;
34:      }
35:      for ( i = k+1 ; i < N; ++ i ) {
36:         /* ora pone a_ik = -a_ik/a_kk(= m_ik) */
37:         A[i][k] /= -A[k][k];                                /* sarebbe m_ik */
38:         /* e pone a_ij = a_ij + m_ik a_kj */
39:         for ( j = k+1; j < N; ++ j )
```

[1] Si rammenti che: le matrici in C partono dall'indice 0 e non 1; è necessario dichiarare almeno la dimensione delle colonne di una matrice quadrata, se questa viene passata come parametro in una procedura; le matrici sono passate come parametri per mezzo del loro indirizzo, quindi se la funzione ne altera il contenuto questo cambiamento è permanente; per alterare il contenuto di un parametro intero o reale è necessario passarlo per indirizzo dichiarandolo come puntatore nell'intestazione della funzione. Scriveremo in `questo font` il testo dei programmi, in **`questo font`** le keyword del C e in *`questo font`* le funzioni di sistema, mentre *i commenti saranno in corsivo*.

```
40:            A[i][j] += A[i][k] * A[k][j];
41:       }
42:       d *= A[k][k];                                  /* d = d · t_kk */
43:    }
44:    return d * A[N-1][N-1];
45:}
```

Resta da scrivere la procedura per il calcolo del massimo valore assoluto degli elementi della colonna k-esima:

```
 1:#include <math.h>  /* importa la definizione di fabs */
 2:/*
 3:   Calcola max_{k≤i≤n} |a_ik| e torna in i0 il più piccolo degli indici per i quali
 4:   il massimo è assunto da |a_ik|
 5:*/
 6:float abs_max(
 7:    float   A[N][N],                                        /* matrice */
 8:    int     *i0,               /* indice da trovare: passato per indirizzo */
 9:    int     k                                     /* riga da cui partire */
10:    )
11:{
12:    float   a = 0;                                 /* conterrà il risultato */
13:    int     i;                                          /* indice di ciclo */

14:    for ( i = k; i < N; ++ i )
15:       if ( a < fabs(A[i][k]) ) {                       /* fabs(x) = |x| */
16:          a = fabs(A[i][k]);                          /* salva il risultato */
17:          *i0 = i;                                      /* annota l'indice */
18:       }
19:    return a;                                  /* ritorna il massimo modulo */
20:}
```

Ovviamente per efficienza sarebbe bene includere questa funzione direttamente dentro `factor`. Si può usare questa routine in un programma del tipo:

```
 1:#define     N    ( /* dimensione della matrice */ )
 2:#include    <stdio.h>  /* importa printf e puts */
 3:int  main() {
 4:    float     A[N][N]= {{ /* riga 1 */ }, ..., { /* riga N */ }}
 5:    int       pivot[N-1];
 6:    int       i, j;
 7:    extern float factor( float A[N][N], int pivot[N-1] );

 8:    printf("Determinante =%g, matrice T=\n", factor(A, p));
```

```
 9:    /* ora stampa la matrice per righe */
10:    for ( i = 0; i < N; ++ i ) {
11:       puts("| ");
12:       for (j = 0; j < N; ++ j)
13:          printf("%g\t", A[i][j]);
14:       puts("\b|\n");
15:    }
16:}
```

ove il file `factor.h` contiene le dichiarazioni delle funzioni `factor` e `abs_max`.

A questo punto, data una matrice A sappiamo trovare il suo fattore triangolare T, le matrici di scambio P (codificate nel vettore p e la matrice G, codificata nella matrice triangolare strettamente inferiore i cui coefficienti sono m_{ij}.

Per risolvere effettivamente il sistema $Ax = b$, dobbiamo considerare $GAx = Gb$ cioè $Tx = Gb$; si tratta quindi di calcolare il vettore Gb e di risolvere un sistema in forma triangolare.

Usando la funzione `factor` possiamo scrivere una funzione `solve` che svolga queste operazioni:

```
 1:/*
 2:   Algoritmo di soluzione di un sistema lineare che utilizza la fattorizzazione
 3:   di Gauss: suppone di avere nella matrice A il risultato di factor() e nel
 4:   vettore p il vettore dei pivot prodotto da factor(); b deve contenere il
 5:   vettore dei termini noti del sistema e, in uscita, conterrà il vettore
 6:   con la soluzione.
 7:*/
 8:void solve(
 9:    float    A[N][N],                  /* matrice fattorizzata del sistema */
10:    int      pivot[N-1],                          /* vettore degli scambi */
11:    float    b[N]                             /* termine noto del sistema */
12:    )
13:{
14:    float    a;                                  /* variabile ausiliaria */
15:    int      i, j, k;                                 /* indici di ciclo */

16:    /* calcola Gb e lo pone in b */
17:    for ( k = 0; k < N-1; ++ k ) {
18:       j = pivot[k];              /* riga con cui va scambiata la k-esima */
19:       if ( j!=k ) {                               /* scambia b[j] con b[k] */
20:          a = b[j]; b[j] = b[k]; b[k] = a;
21:       }
22:       /* ora somma per i multipli degli m_ik */
23:       for ( i = k+1; i < N; ++ i )
24:          b[i] += A[i][k] * b[k];
25:    }
26:    b[N-1] /= A[N-1][N-1];
```

```
27:    /*  ora risolve 'a cascata' le equazioni lineari risultanti partendo dall'ultima
28:        e sostituendoi valori trovati nelle incognite delle precedenti  */
29:    for ( i = N-2; i >= 0; -- i ) {
30:       /* b_i = (b_i - sum_{j=i+1}^n a_ij b_j) / a_ii  */
31:       a = b[i];
32:       for ( j = i+1; j < N; ++ j )
33:          a -= A[i][j] * b[j];
34:       b[i] = a / A[i][i];
35:    }
36:}
```

Una ulteriore osservazione che possiamo fare sulla fattorizzazione di Gauss è che possiamo raggruppare le matrici di scambio P_{ij} e le matrici di somma $M_{ij}(m)$ nel fattore G: precisamente, dato che $P_{ij} = P_{ij}^{-1}$, possiamo scrivere

$$P_{ij}X = (P_{ij}XP_{ij})P_{ij}$$

per ogni matrice X; questo consente di "spostare verso destra le matrici di scambio nella fattorizzazione sostituendo alle $M_{ij}(m)$ le loro coniugate per P_{kl}.

Questo non cambia la natura della fattorizzazione, dato che

$$P_{ij}M_{hk}(m)P_{ij} = P_{ij}(I + mE_{hk})P_{ij} = (P_{ij} + mP_{ij}E_{hk})P_{ij} = I + mP_{ij}E_{hk}P_{ij}$$

Ma la matrice $P_{ij}E_{hk}P_{ij}$ è ancora della forma E_{rs} e quindi coniugare $M_{hk}(m)$ per P_{ij} dà ancora luogo ad una matrice del tipo $M_{rs}(m)$.

Dunque possiamo scrivere G nella forma

$$G = M_{n-1}(m_{n-1})...M_2(m_2)M_1(m_1)P_{n-1}...P_2P_1$$

ove $\{m_1, ..., m_{n-1}\} = \{m_{ij}\}_{i>j}$ sono esattamente gli stessi coefficienti di prima, a meno di riordinamento; inoltre le matrici P_i sono esattamente le P_{ij} precedenti nello stesso ordine.

Possiamo quindi scrivere $G = MP$ ed ottenere

$$PA = MT$$

La soluzione del sistema $Ax = b$ equivale quindi a quella dei due sistemi

$$\begin{cases} My = Pb \\ Tx = y \end{cases}$$

ambedue in forma triangolare (il primo inferiore, il secondo superiore).

3 Raffinamento iterativo

Per limitare la propagazione dell'errore dovuta all'instabilità del problema nel metodo di Gauss, quando il problema non è eccessivamente mal condizionato, si può utilizzare l'algoritmo di *raffinamento iterativo*.

Se $Ax = b$ è il sistema che vogliamo risolvere, possiamo codificare questo problema in $\overline{A}\overline{x} = \overline{b}$, ove $\overline{A}$ rappresenta la matrice A scritta usando i numeri disponibili nella macchina; ora supponiamo che la soluzione di $Ax = b$ fornita dall'algoritmo di Gauss sia x', e che la soluzione esatta sia in realtà $x' + y'$: allora

$$A(x' + y') = b$$

e quindi, per $r' = b - Ax'$, il sistema

$$Ay' = r'$$

è soddisfatto; possiamo risolverlo con Gauss (usando la stessa decomposizione $GA = T$ che abbiamo adoperato per risolvere $Ax = b$) e troveremo non la soluzione y' ma una sua approssimazione $\overline{y'}$.

Ora poniamo

$$x'' = x' + \overline{y'}$$

e ripetiamo il procedimento: cioè consideriamo la soluzione esatta $x'' + y''$ e così via.

In presenza di un buon condizionamento del problema, cioè se a una perturbazione dei dati iniziali A e b corrisponde una perturbazione della soluzione trovata *dello stesso ordine di grandezza*, il procedimento converge e consente di trovare approssimazioni migliori alla soluzione cercata.

Poiché il passaggio cruciale del metodo consiste nel calcolo di una differenza $b - Ax^{(i)}$, per limitare la cancellazione numerica valuteremo nell'algoritmo questa differenza in doppia precisione.

```
 1:/*
 2:   Raffinamento iterativo per migliorare l'approssimazione delle soluzioni
 3:   di un sistema lineare risolto con factor/solve ; compie al più kmax
 4:   iterazioni e controlla se dopo queste la precisione della macchina è stata
 5:   raggiunta o meno; se questo non è il caso, torna un valore non nullo.
 6:*/
 7:int refine(
 8:   float    A[N][N],                        /* matrice di partenza */
 9:   float    GT[N][N],            /* matrice dopo la fattorizzazione */
10:   float    b[N],                   /* vettore dei termini costanti */
11:   float    pivot[N-1],                   /* vettore degli scambi */
12:   float    x[N],                   /* soluzione trovata da solve */
```

```
13:   int     kmax                    , /* massimo numero di iterazioni */
14:   float      epsilon                        /* precisione desiderata */
15:   )
16:{
17:   double   d;                 /* usato nel calcolo in doppia precisione */
18:   float    y[n];                        /* soluzione trovata da solve */
19:   float    in, xn, xn1, yn;                     /* norme di vettori */
20:   int      i, j, k;                                /* indici di loop */
21:   extern float norm( float v[N]);   /* calcola la norma di un vettore */
22:   extern void solve( float A[N][N], int pivot[N-1], float b[N] );

23:   /* nella prima iterazione usa x come dato iniziale */
24:   in = norm(x);                           /* calcola la norma iniziale */
25:   for ( i = 0; i < kmax; ++ i ) {
26:      /* calcola y = b − Ax (doppia precisione) */
27:      for ( j = 0; j < n; ++ j ) {
28:         d = b[j];
29:         for ( k = 0; k < n; ++ k )
30:            d -= ( A[j][k] * x[k] );
31:         y[j] = d;                            /* perdita di precisione... */
32:      }
33:      solve(GT, pivot, y);                               /* risolve Ay = r */
34:      yn = norm(y);                   /* calcola la norma della soluzione */
35:      xn = norm(x);                                      /* calcola |x| */
36:      /* somma questa soluzione y a x ottenendo x' */
37:      for ( j = 0; j < N; ++ j )
38:         x[j] += y[j];
39:      xn1 = norm(x);                              /* e ne calcola la norma */
40:      /* ora decide se continuare: se in‖x‖ ≤ ‖y‖ ‖x'‖  termina con un
41:         errore; se ‖y‖ ≤ ε‖x'‖  termina con successo;
42:         altrimenti continua l'iterazione  */
43:      if ( in * xn1 <= yn * xn )  return 1;
44:      if ( yn <= epsilon * xn1 )  return 0;
45:      in = yn;                 /* cambia la norma iniziale in quella di y */
46:   }
47:   /* se arriva qui ha superato il numero di iterazioni consentite */
48:   return 1;
49:}
```

Come norma possiamo ad esempio considerare quella euclidea:

```
1:#include <math.h>  /* importa sqrt */
2:/*
3:   Calcola la norma euclidea di un vettore con N elementi
4:*/
5:float  norm( float v[N] )
6:{
7:   float   x = 0;
8:   int     i;
```

```
 9:   for ( i = 0; i < N; ++ i )
10:      x += v[i] * v[i];
11:   x = sqrt(x);
12:}
```

Ad esempio su una macchina con numeri a 32 bit il sistema

$$\begin{pmatrix} 5 & 7 & 6 & 5 \\ 7 & 10 & 8 & 7 \\ 6 & 8 & 10 & 9 \\ 5 & 7 & 9 & 10 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 23 \\ 32 \\ 33 \\ 31 \end{pmatrix}$$

ha fornito col metodo di Gauss la soluzione

$$\begin{bmatrix} \texttt{0.999993} \\ \texttt{1} \\ \texttt{1} \\ \texttt{0.999999} \end{bmatrix}$$

e, dopo un solo passo del raffinamento (con `epsilon=.000001`), la soluzione esatta

$$\begin{bmatrix} \texttt{1} \\ \texttt{1} \\ \texttt{1} \\ \texttt{1} \end{bmatrix}$$

4 Inverse di matrici

Nota la fattorizzazione di Gauss $GA = T$ di una matrice invertibile A, possiamo calcolare A^{-1} come $A^{-1} = T^{-1}G$.

Questo fornisce una soluzione ovviamente approssimata che possiamo provare a raffinare se il problema non è mal condizionato: l'idea è che se B è una approssimazione di A^{-1}, se definiamo $E = A^{-1} - B$, allora $A(B + E) = I$ e quindi, ponendo

$$R = AE = I - AB$$

otteniamo una migliore approssimazione

$$B' = B + BR$$

Iterando questo procedimento su B' si ottiene una successione $\{B^{(i)}\}$ che converge alla soluzione esatta se

$$\|I - AB\| < 1$$

Anche qui, nel calcolo di R, è dietro l'angolo il pericolo della cancellazione numerica.

5 Metodi iterativi

Se esigenze di spazio non consentono una agevole manipolazione della matrice A, per risolvere il sistema lineare si può usare un metodo di "approssimazioni successive che alteri ad ogni passo solo il vettore soluzione; questo si chiama *metodo iterativo*.

Una idea per applicare questo metodo è decomporre A in somma $A = D + C$; allora il sistema $Ax = b$ diviene

$$Dx = -Cx + b$$

Allora, partendo da un dato iniziale $x^{(0)}$ possiamo risolvere il sistema $Dx^{(1)} = Cx^{(0)} + b$, e poi il sistema $Dx^{(2)} = Cx^{(1)} + b$, e così via fino ad una soluzione $x^{(n)}$ soddisfacente.

Ovviamente la scelta della decomposizione è cruciale: D sarà assunta con le seguenti proprietà:

(1) $\det D \neq 0$.

(2) D è triangolare (o se possibile diagonale).

(3) $\|I - D^{-1}A\| < 1$.

In effetti 3) garantisce la convergenza della successione $\{x^{(i)}\}$ alla soluzione del sistema.

Infatti abbiamo $Dx^{(i+1)} = -Cx^{(i)} + b$, cioè

$$x^{(i+1)} = -D^{-1}Cx^{(i)} + D^{-1}b$$

Poniamo ora $B = -D^{-1}C = I - D^{-1}A$: allora

$$x^{(i+1)} = Bx^{(i)} + D^{-1}b$$

Se $e^{(i+1)}$ è l'errore assoluto relativo alla soluzione approssimata $x^{(i+1)}$, cioè

$$e^{(k+1)} = x - x^{(i+1)} = (Bx + D^{-1}b) - (Bx^{(i)} + D^{-1}b) = B(x - x^{(i)}) = Be^{(i)}$$

allora possiamo determinare la successione degli errori come

$$\begin{aligned}
e^{(0)} &= x - x^{(0)} \\
e^{(1)} &= Be^{(0)} \\
e^{(2)} &= Be^{(1)} = B^2e^{(0)} \\
e^{(3)} &= Be^{(2)} = B^3e^{(0)} \\
&\dots\dots\dots \\
e^{(i+1)} &= Be^{(i)} = B^{i+1}e^{(0)} \\
&\dots\dots\dots
\end{aligned}$$

La convergenza della successione $\{x^{(i)}\}$ alla soluzione esatta equivale all'essere la successione $\{e^{(i)}\}$ infinitesima, cioè che sia infinitesima la successione di matrici $\{B^i\}$: un criterio necessario e sufficiente per questo è che il raggio spettrale $\rho(B)$ sia minore di 1.

Dunque il procedimento iterativo converge se e solo se

$$\rho(B) = \rho(-D^{-1}C) = \rho(I - D^{-1}A) < 1$$

Poiché una norma matriciale (compatibile con una norma vettoriale data) verifica la $\rho(B) < \|B\|$, la 3) precedente implica la convergenza.

Una decomposizione $A = D + C$ molto semplice è quella usata nel *metodo di Jacobi*: si sceglie come D la matrice diagonale i cui elementi diagonali sono quelli di A, e come C ovviamente $A - D$.

Ovviamente dobbiamo avere che $a_{11} \neq 0, a_2 \neq 0, ..., a_{nn} \neq 0$ altrimenti D risulta singolare.

Comunque se qualche elemento diagonale di A dovesse essere nullo, possiamo sempre scambiare due righe in A in modo che l'elemento diagonale in questione risulti diverso da zero (se questo non è possibile allora A possiede una colonna di zeri, e quindi è singolare, il che è escluso per ipotesi).

Quindi, a meno di scambiare le equazioni nel sistema, possiamo sempre supporre che la decomposizione di Jacobi sia possibile.

La convergenza è legata alla relazione 3): usando ad esempio la norma uniforme, questo vuol dire che:

$$\|I - D^{-1}A\| < 1$$

Ma D^{-1} ha come elementi diagonali gli inversi degli elementi diagonali di A, e quindi

$$D^{-1}A = \begin{pmatrix} 1 & a_{11}^{-1}a_{12} & a_{11}^{-1}a_{13} & ... & a_{11}^{-1}a_{1n} \\ a_{22}^{-1}a_{21} & 1 & a_{22}^{-1}a_{23} & ... & a_{22}^{-1}a_{2n} \\ a_{33}^{-1}a_{31} & a_{33}^{-1}a_{32} & 1 & ... & a_{33}^{-1}a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{nn}^{-1}a_{n1} & a_{nn}^{-1}a_{n2} & a_{nn}^{-1}a_{n3} & ... & 1 \end{pmatrix}$$

sicché

$$\|I - D^{-1}A\|_\infty = \max_{1 \leq i \leq n} \sum_{\substack{j=1 \\ j \neq -1}}^{n} \left| \frac{a_{ij}}{a_{ii}} \right|$$

e quindi la 3) si scrive semplicemente come

$$\sum_{\substack{j=1 \\ j \neq -1}}^{n} |a_{ij}| < |a_{ii}|$$

per $i = 1, ..., n$.

Programmiamo ora il metodo di Jacobi: è molto semplice, dato che il sistema $Ax = b$ si può scrivere come $Dx = -Cx+b$ e D è facilmente invertibile essendo diagonale: $x = -D^{-1}Cx + D^{-1}b$ equivale a

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

per $i = 1, ..., n$; partendo da una approssimazione $x^{(i)}$ si calcola la $x^{(i+1)}$ usando queste equazioni ove nei membri a destra si pongono le $x^{(i)}$.

```
 1:/*
 2:   Risoluzione del sistema Ax = b col metodo di Jacobi: richiede il numero
 3:   massimo di iterazioni e restituisce nel vettore x una soluzione approssimata
 4:   migliore di quella di partenza
 5:*/
 6:void jacobi(
 7:    float    A[N][N],                            /* matrice del sistema */
 8:    float    x[N],                            /* vettore delle incognite */
 9:    float    b[N],                           /* vettore dei termini noti */
10:    int      kmax                               /* numero di iterazioni */
11:    )
12:{
13:    float    x1[n];                     /* contiene copia dei vecchi dati */
14:    float    a;                                /* variabile ausiliaria */
15:    int      i, j, k;                                 /* indici di ciclo */
16:    extern int arrange( float A[N][N], float b[N] );
17:                 /* funzione che ordina le equazioni in modo che gli
18:                      elementi diagonali di A siano tutti non nulli */
19:    arrange(A, b);
20:    for ( k = 0; k < kmax; ++ k ) {
21:       for ( i = 0; i < N; ++ i )  x1[i] = x[i];     /* copia x in x1 */
22:       for ( i = 0; i < N; ++ i ) {
23:          /* pone x_i = (b_i - sum_{j!=i} a_ij x'_j)/a_ii */
24:          a = b[i];
25:          for ( j = 0; j != i && j < N; ++ j )
26:             a -= A[i][j] * x1[j];
27:          x[i] = a / A[i][i];
28:       }
29:    }
30:}
```

Resta solo da scrivere la funzione di scambio delle righe: si tratta di scambiare le righe della matrice A (e del vettore b) in modo da ottenere elementi diagonali tutti non nulli: una possibile soluzione (non molto elegante visto che utilizza una istruzione `goto` è la seguente funzione

```
 1:/*
 2:   Se non tutti gli elementi diagonali di A sono diversi da zero scambia le righe
 3:   in A e b fino a soddisfare questo requisito; se non è possibile (det A = 0)
 4:   torna un numero diverso da zero
 5:*/
 6:int arrange(
 7:    float    A[N][N],                /* matrice da riordinare per righe */
 8:    float    b[N]                         /* vettore dei termini noti */
 9:    )
10:{
11:    float    a;                                /* variabile ausiliaria */
12:    int      i, j, k;                               /* indici di ciclo */

13:    for ( i = 0; /* ciclo infinito */ ; ++ i ) {
14:        /* cerca una riga il cui termine diagonale sia nullo */
15:        for ( ; i < N; ++ i )
16:            if ( A[i][i] == 0 ) {
17:                /* ora i contiene l'indice della riga da scambiare: cerca una riga
18:                 il cui termine A[i][i] sia non nullo fra quelle seguenti */
19:                for ( j = i+1; j < N; ++ j )
20:                    if ( A[j][i] != 0 )  goto scambia_righe;
21:                /* se arriva qui la riga cercata è fra le precedenti:
22:                    ne cerca una j-esima in modo che A[i][j]!=0 */
23:                for ( j = 1; j < i; ++ j )
24:                    if ( A[j][i] != 0 && A[i][j] != 0 ) goto scambia_righe;
25:                /* se arriva qui esiste una colonna di zeri */
26:                return 1;
27:            }
28:        /* se arriva qui la matrice è a posto */
29:        return 0;
30:scambia_righe:
31:        for ( k = 0; k < N; ++ k ) {    /* scambia la riga i con la riga j */
32:            a = A[j][k]; A[j][k] = A[i][k]; A[i][k] = a;
33:        }
34:        a = b[j]; b[j] = b[i]; b[i] = a;
35:    }                            /* ripete il ciclo infinito: ricomincia la ricerca */
36:}
```

La programmazione del metodo di Jacobi suggerisce un modo di migliorarlo: infatti abbiamo dovuto copiare il vettore x prima di riscriverlo, per poterne usare i valori precedenti; un miglioramento si avrebbe considerando

i nuovi valori delle componenti di x appena calcolati: cioè usare la formula

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

ove x' rappresenta la precedente approssimazione.

Questa modifica del metodo di Jacobi è nota come *metodo di Gauss-Seidel*; nel programmarlo inseriamo nell'algoritmo anche un controllo della precisione raggiunta nei passi stabiliti

```
 1:#include <math.h>   /* importa la definizione di fabs */
 2:/*
 3:   Risoluzione del sistema Ax=b col metodo di Gauss-Seidel: richiede il numero
 4:   massimo di iterazioni e la precisione desiderata nell'approssimazione della
 5:   soluzione; restituisce nel vettore x una soluzione approssimata migliore di
 6:   quella di partenza; se nel numero di passi richiesto è raggiunta la precisione
 7:   desiderata, torna 0, altrimenti torna 1
 8:*/
 9:int gauss_seidel(
10:    float    A[N][N],                              /* matrice del sistema */
11:    float    x[N],                              /* vettore delle incognite */
12:    float    b[N],                             /* vettore dei termini noti */
13:    int     kmax,                                /* numero di iterazioni */
14:    float    prec                               /* precisione desiderata */
15:    )
16:{
17:    float    err;                        /* usato nel calcolo dell'errore */
18:    float    a;                                  /* variabile ausiliaria */
19:    int      i, j, k;                                  /* indici di ciclo */
20:    extern int arrange( float A[N][N], float b[N] );
21:    extern float norm( float v[N] );

22:    arrange(A, b);
23:    for ( k = 0; k < kmax; ++ k ) {
24:       /* calcola a = (b_1 - sum_{j=2}^n a_{1j}x'_j)/a_{11} */
25:       a = b[0];
26:       for ( j = 1 /* parte dal 2° elemento */; j < N; ++ j )
27:          a -= A[0][j] * x[j];
28:       a /= A[0][0];
29:       err = fabs(a - x[0]);
30:       x[0] = a;                                 /* nuovo valore di x[0] */
31:       for ( i = 1; i < N; ++ i) {
32:          /* ora calcola a = (b_i - sum_{j=1}^{i-1} a_{ij}x_j + sum_{j=i+1}^n a_{ij}x'_j)/a_{ii} */
33:          a = b[i];
34:          for ( j = 0; j != i && j < n; ++ j )
35:             a -= A[i][j] * x[j];
36:          a /= A[i][i];
37:          /* sceglie l'errore più grave commesso */
```

```
38:         if ( err < fabs(a - x[i]) )  err = fabs(a - x[i]);
39:         x[i] = a;                              /* nuovo valore di x[i] */
40:      }
41:      if ( err < prec * norm(x) )  return 0;
42:   }
43:   return 1;
44:}
```

Come si vede non c'è più bisogno di memorizzare il vettore x' e quindi si ha anche un risparmio di spazio.

Il metodo di Gauss–Seidel corrisponde ad usare una decomposizione $A = D + C$ ove D è la matrice triangolare inferiore estratta da A (supponendo che i suoi elementi diagonali siano tutti non nulli: altrimenti A andrebbe preventivamente riordinata per righe):

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

Di nuovo la condizione

$$\sum_{\substack{j=1 \\ j\neq -1}}^{n} |a_{ij}| < |a_{ii}|$$

(per $i = 1, ..., n$) garantisce la convergenza dell'algoritmo.

Quindi per le matrici dominanti (per righe o per colonne) la convergenza è assicurata; anche nel caso delle matrici irriducibili abbiamo certamente convergenza dei metodi di Jacobi e Gauss–Seidel.

6 Sovrarilassamento

Il metodo di Gauss–Seidel si riduce sostanzialmente alla formula iterativa

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

ove x' rappresenta la precedente approssimazione; da questa, sottraendo ad ambo i membri il termine x'_i, definiamo

$$r'_i = x_i - x'_i = \frac{b_i - \displaystyle\sum_{j=1}^{i-1} a_{ij}x_j - \sum_{j=i}^{n} a_{ij}x'_j}{a_{ii}}$$

Questo valore r'_i può essere considerato una correzione di x'_i per ottenere x_i, dato che $x_i = x'_i + r'_i$.

Possiamo pensare di migliorare questa correzione riscalandola di un fattore ω:

$$x_i = x'_i + \omega r'_i$$

Chiaramente il problema sta nello scegliere ω in modo da accelerare la convergenza della successione delle approssimanti alla soluzione esatta.

Nell'introduzione di ω consiste il *metodo del rilassamento* (di *sottorilassamento* se $0 < \omega < 1$ e di *sovrarilassamento* se $1 < \omega$); per $\omega = 1$ ritroviamo il metodo di Gauss–Seidel.

Osserviamo che, spezzando A in somma $L + D + U$ ove L è la matrice triangolare strettamente inferiore estratta da A, A la matrice diagonale estratta da A e U la matrice triangolare strettamente superiore estratta da A, troviamo che la formula per il metodo di Gauss–Seidel si scrive in forma vettoriale come

$$x = D^{-1}(b - Lx - Ux')$$

Inoltre

$$r' = D^{-1}(b - Lx - (D + U)x')$$

e

$$(D + \omega L)x = ((1 - \omega)D - \omega U)x' + \omega b$$

Dunque, la convergenza del metodo di rilassamento equivale alla condizione

$$\rho((D + \omega L)^{-1}((1 - \omega)D - \omega U)) < 1$$

Il raggio spettrale misura in questo caso la rapidità della convergenza.

La ricerca del valore di ω ottimale rispetto alla convergenza dipende dal problema: per certe classi di matrici si riesce effettivamente a determinare il valore ottimo per ω.

Per le matrici simmetriche definite positive ogni valore $\omega \in (0, 2)$ il metodo converge; per $\omega \notin (0, 2)$ il raggio spettrale precedente è almeno uno, e quindi il metodo certamente non converge.

La seguente funzione realizza il metodo di Gauss–Seidel tenendo conto di un parametro di rilassamento:

```
1:#include <math.h>   /* importa la definizione di fabs */
2:/*
3:  Risoluzione del sistema Ax=b col metodo di rilassamento: richiede il numero
4:  massimo di iterazioni e la precisione desiderata nell'approssimazione della
5:  soluzione; restituisce nel vettore x una soluzione approssimata migliore di
6:  quella di partenza; se nel numero di passi richiesto è raggiunta la precisione
```

```
 7:   desiderata, torna 0, altrimenti torna 1
 8:*/
 9:int gauss_seidel(
10:   float    A[N][N],                          /* matrice del sistema */
11:   float    x[N],                           /* vettore delle incognite */
12:   float    b[N],                          /* vettore dei termini noti */
13:   float    omega,                       /* parametro di rilassamento */
14:   int      kmax,                            /* numero di iterazioni */
15:   float    prec                            /* precisione desiderata */
16:   )
17:{
18:   float    err;                     /* usato nel calcolo dell'errore */
19:   float    a;                                /* variabile ausiliaria */
20:   int     i, j, k;                               /* indici di ciclo */
21:   extern int arrange( float A[N][N], float b[N] );
22:   extern float norm( float v[N] );

23:   /* per sicurezza controlla omega */
24:   if ( omega <= 0 || omega >= 2 )  omega = 1;
25:   arrange(A, b);
26:   for ( k = 0; k < kmax; ++ k ) {
27:      /* calcola a = (b_1 - sum_{j=2}^{n} a_{1j} x'_j)/a_{11} */
28:      a = b[0];
29:      for ( j = 0 /* parte dal 1° elemento */; j < N; ++ j )
30:         a -= A[0][j] * x[j];
31:      a *= omega / A[0][0];
32:      err = fabs(a - x[0]);
33:      x[0] += a;                             /* nuovo valore di x[0] */
34:      for ( i = 1; i < N; ++ i ) {
35:         /* calcola a = (b_i - sum_{j=1}^{i-1} a_{ij} x_j + sum_{j=i}^{n} a_{ij} x'_j)/a_{ii} */
36:         a = b[i];
37:         for ( j = 0; j < N; ++ j )
38:            a -= A[i][j] * x[j];
39:         a *= omega / A[i][i];
40:         /* sceglie l'errore più grave commesso */
41:         if ( err < fabs(a - x[i]) )  err = fabs(a - x[i]);
42:         x[i] += a;                          /* nuovo valore di x[i] */
43:      }
44:      if ( err < prec * norm(x) )  return 0;
45:   }
46:   return 1;
47:}
```

7 Metodo delle potenze

Supponiamo di avere una matrice reale A e di essere interessati ad i suoi autovalori $\lambda_1, ..., \lambda_n$; supporremo che questi siano ordinabili nel seguente modo:

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$$

Supponiamo inoltre che esista una base di autovettori $\mathcal{B} = (x_1, ..., x_n)$ ove x_i è autovettore di autovalore λ_i.

Ora scegliamo un vettore v_0 che sia tale che la sua decomposizione in coordinate rispetto alla base $\mathcal{B}$

$$v_0 = a_1 x_1 + \cdots a_n x_n$$

abbia la prima coordinata non nulla: $a_1 \neq 0$.

Poiché A è reale, tale è il suo polinomio caratteristico, e le sue radici appaiono quindi o reali o in coppie complesse coniugate: dato che $|\lambda_1|$ è più grande (e quindi diverso) da ogni altro modulo di autovalore, λ_1 è reale, e quindi possiamo supporre che il suo autovettore x_1 abbia componenti reali (rispetto alla base canonica).

Ora consideriamo $v_1 = Av_0$:

$$\begin{aligned} v_1 =& a_1 A x_1 + a_2 A x_2 + \cdots + a_n A x_n = a_1 \lambda_1 x_1 + a_2 \lambda_2 x_2 + \cdots + a_n \lambda_n x_n \\ &= \lambda_1 \left(a_1 x_1 + \frac{\lambda_2}{\lambda_1} a_2 x_2 + \cdots + \frac{\lambda_n}{\lambda_1} a_n x_n \right) \end{aligned}$$

Iteriamo questa operazione, considerando $v_2 = Av_1 = A^2 v_0$,..., $v_k = A^k v_0$:

$$\begin{aligned} v_k &= a_1 A^k x_1 + a_2 A^k x_2 + \cdots + a_n A^k x_n = a_1 \lambda_1^k x_1 + a_2 \lambda_2^k x_2 + \cdots + a_n \lambda_n^k x_n \\ (*) \quad &= \lambda_1^k \left(a_1 x_1 + \left(\frac{\lambda_2}{\lambda_1} \right)^k a_2 x_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1} \right)^k a_n x_n \right) \end{aligned}$$

Dato che $\left| \frac{\lambda_i}{\lambda_1} \right| < 1$ per $i = 1, ..., n$, abbiamo che

$$(**) \qquad \lim_{k \to \infty} \frac{1}{\lambda_1^k} v_k = a_1 x_1$$

Ora ragioniamo sulle singole componenti di $(v_k)_i$ e $(x_1)_i$ rispetto alla base canonica; il limite appena scritto consente di calcolare, per $i \in \{1, ..., n\}$ tale

che $(x_1)_i \neq 0$:

$$\begin{aligned}
\lim_{k\to\infty} \frac{(v_{k+1})_i}{(v_k)_i} &= \lim_{k\to\infty} \frac{(v_{k+1})_i}{(v_k)_i} \\
&= \lim_{k\to\infty} \frac{\lambda_1^{k+1}}{(v_k)_i} \left(a_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^{k+1} a_2 x_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^{k+1} a_n x_n \right)_i \\
&= \lim_{k\to\infty} \frac{\lambda_1}{(a_1 x_1)_i} \left(a_1 x_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^{k+1} a_2 x_2 + \cdots + \left(\frac{\lambda_n}{\lambda_1}\right)^{k+1} a_n x_n \right)_i \\
&= \frac{\lambda_1}{(a_1 x_1)_i} (a_1 x_1)_i \\
&= \lambda_1
\end{aligned}$$

(prima abbiamo usato la (*) e poi la (**)).

Dunque la successione $(v_{k+1})_i/(v_k)_i$ converge a λ_1 e la velocità di questa convergenza è misurata dalle potenze $|\lambda_i/\lambda_1|^k$; naturalmente possiamo scegliere di normalizzare i vettori v_m rispetto a una certa norma, dato che ci interessa solo il rapporto delle loro compomenti; inoltre moltiplicare per un fattore non nullo non altera l'essere un vettore x_k un autovettore di autovalore λ_k.

Questo procedimento suggerisce un algoritmo iterativo per calcolare λ_1 partendo da un valore iniziale arbitrario v_0.

```
 1:/*
 2:   Calcola un autovalore lambda1 di una matrice A (nelle ipotesi di
 3:   crescenza degli autovalori) usando una successione di vettori normalizzati
 4:*/
 5:float  lambda1(
 6:   float    A[N][N],                                      /* matrice */
 7:   int      kmax                               /* numero di iterazioni */
 8:   )
 9:{
10:   float    v[N];                          /* vettore usato nel calcolo */
11:   float    w[N];                          /* vettore usato nel calcolo */
12:   float    a;                                          /* autovalore */
13:   float    f;                                /* variabile ausiliaria */
14:   int      i, j, k;                                 /* indici di ciclo */
15:   extern float norm( float v[N] );

16:   for ( i = 1; i < N; ++ i )
17:      v[i] = 0;
18:   v[0] = 1;         /* inizializza il vettore ad uno di norma (euclidea) 1 */
19:   for ( k = 0; k < kmax; ++ k ) {
20:      /* ora pone w = Av */
```

```
21:       for ( i = 0; i < N; ++ i ) {
22:          w[i] = 0;
23:          for ( j = 0; j < N; ++ j )
24:             w[i] += A[i][j] * v[j];
25:       }
26:       /* sceglie un indice per cui v[i]!=0 */
27:       for ( i = 0; v[i] == 0 && i < N; ++ i )
28:          /* enunciato vuoto */ ;
29:       /* calcola l'autovalore */
30:       a = w[i] / v[i];
31:       /* ora assegna a v il normalizzato di w */
32:       f = norm(w);
33:       for ( i = 0; i < N; ++ i )
34:          v[i] = w[i] / f;
35:    }
36:    return a;
37:}
```

Osserviamo che questa funzione calcola anche una approssimazione di un autovettore (di norma 1) relativo all'autovalore cercato; comunque la convergenza di questo algoritmo non segue esattamente dal ragionamento svolto in precedenza, perché la scelta dell'indice rispetto al quale considerare la componente del vettore v non è necessariamente la stessa ad ogni iterazione; in ogni caso la convergenza si può dimostrare anche in questo caso più generale. Inoltre questo metodo non funziona per autovalori complessi, perché abbiamo usato in modo essenziale la realtà dell'autovettore x_1.

Se la matrice A è simmetrica si può considerare un metodo alternativo che consiste nell'usare il quoziente di Rayleigh: ricordiamo che se v è un autovettore di una matrice, il suo autovalore è dato dal quoziente di Rayleigh ($\langle , \rangle$ denota il prodotto scalare canonico)

$$\lambda = \frac{\langle Av, v\rangle}{\langle v, v\rangle}$$

(infatti $Av = \lambda v$ e $v \neq 0$, sicché $\langle Av, v\rangle = \lambda\langle v, v\rangle$).

Ora, se A è simmetrica, possiede una base ortonormale di autovettori $\{x_1, ..., x_n\}$ e

$$v_k = \frac{1}{c_k}\left(a_1\lambda_1^k x_1 + ... + a_n\lambda_n^k x_n\right)$$

ove la costante c_k è il prodotto $\|w_1\| \cdot \|w_n\|$ delle norme delle approssimazioni degli autovettori (non normalizzati) trovati al passo precedente dell'iterazione; si pone allora

$$v_{k+1} = \frac{w_{k+1}}{\|w_{k+1}\|}$$

e si trova

$$\langle w_{k+1}, v_k \rangle = \frac{\langle Av_k, v_k \rangle}{\langle v_k, v_k \rangle} = \frac{\sum_{i=1}^{n} a_i^2 \lambda_i^{2k+1}}{\sum_{i=1}^{n} a_i^2 \lambda_i^{2k}} = \lambda_1 \frac{a_1 + \sum_{i=2}^{n} a_i^2 \left(\frac{\lambda_i}{\lambda_1}\right)^{2k+1}}{a_1 + \sum_{i=2}^{n} a_i^2 \left(\frac{\lambda_i}{\lambda_1}\right)^{2k}}$$

Allora, passando al limite per $k \to \infty$, troviamo

$$\lim_{k \to \infty} \langle w_{k+1}, v_k \rangle = \lambda_1$$

con velocità di convergenza pari a quella con la quale $|\lambda_2/\lambda_1|^{2m}$ tende a zero.

Le opportune modifiche alla funzione precedente sono:

```
 1:/*
 2:   Calcola un autovalore lambda1 di una matrice simmetrica A (nelle ipotesi di
 3:   crescenza degli autovalori) usando una successione di vettori normalizzati
 4:*/
 5:float  eigen_simm(
 6:   float    A[N][N],                                         /* matrice */
 7:   int      kmax                                 /* numero di iterazioni */
 8:   )
 9:{
10:   float    v[N];                            /* vettore usato nel calcolo */
11:   float    w[N];                            /* vettore usato nel calcolo */
12:   float    a;                                             /* autovalore */
13:   float    f;                                   /* variabile ausiliaria */
14:   int      i, j, k;                                   /* indici di ciclo */
15:   extern float norm( float v[N] );

16:   for ( i = 1; i < N; ++ i )
17:      v[i] = 0;
18:   v[0] = 1;         /* inizializza il vettore ad uno di norma (euclidea) 1 */
19:   for ( k = 0; k < kmax; ++ k ) {
20:      /* pone w = Av */
21:      for ( i = 0; i < N; ++ i ) {
22:         w[i] = 0;
23:         for ( j = 0; j < N; ++ j )
24:            w[i] += A[i][j] * v[j];
25:      }
26:      /* calcola l'autovalore: a = <v, w> = v1w1 + ... + vnwn */
27:      a = w[0] * v[0];
28:      for ( i = 1; i < N; ++ i )
29:         a += w[i] * v[i];
30:      /* ora assegna a v il normalizzato di w */
31:      f = norm(w);
32:      for ( i = 0; i < N; ++ i )
33:         v[i] = w[i] / f;
```

```
34:   }
35:   return a;
36:}
```

Ad esempio sulla matrice

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 0 \\ 3 & 4 & 1 & 2 \\ 4 & 0 & 2 & 3 \end{pmatrix}$$

questo metodo, su una macchina a 32 bit, ha prodotto la soluzione `9.5208` in 9 passi:

```
1.80562E-035
1
7.46667
9.31746
9.50276
9.51909
9.52062
9.52078
9.52079
9.5208
```

mentre il metodo delle potenze ne ha richiesti 16.

Supponiamo ora che gli autovalori siano ordinati come

$$|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_n|$$

e di aver già calcolato λ_1 ed un suo autovettore v_1; se vogliamo determinare λ_2 ed un suo autovalore possiamo ancora ricorrere al metodo delle potenze, con la seguente modifica: supponiamo di poter determinare una matrice ortogonale U tale che

$$Uv_1 = -\sigma e_1$$

ove e_1 è il primo vettore della base canonica e $|\sigma| = \|v_1\|_2$; allora

$$(UAU^T)Uv_1 = \lambda_1 Uv_1 \implies UAU^T e_1 = \lambda_1 e_1$$

Ma A e UAU^T hanno gli stessi autovalori (U è ortogonale, quindi se λ è un autovalore di A e v un suo autovettore: $UAU^TUv = UAv = \lambda Uv$); inoltre

$$UAU^T = \begin{pmatrix} \lambda_1 & a^T \\ 0 & A_1 \end{pmatrix}$$

ove A_1 è una matrice $(n-1)\times(n-1)$ con gli stessi autovalori di A (eccetto al più λ) e $a \in \mathbb{R}^{n-1}$. Questo si vede così: certamente

$$UAU^T = \begin{pmatrix} \lambda_1 & a^T \\ b & A_1 \end{pmatrix}$$

con $b \in \mathbb{R}^{n-1}$; ma $UAU^T e_1 = \lambda_1 U v_1$ da cui

$$\begin{pmatrix} \lambda_1 \\ 0 \end{pmatrix} = \begin{pmatrix} \lambda_1 & a^T \\ b & A_1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \lambda_1 \\ b \end{pmatrix}$$

Ora: poiché i rimanenti autovalori di A sono esattamente quelli di A_1 possiamo applicare a A_1 il metodo delle potenze per determinare il suo autovalore di modulo massimo, cioè λ_2; inoltre in questo modo abbiamo un autovettore w_1 di A_1 relativo a quell'autovalore.

Per trovare un autovettore di A relativo allo stesso autovalore basta trovare un $x \in \mathbb{R}$ tale che

$$\begin{pmatrix} \lambda_1 & a^T \\ 0 & A_1 \end{pmatrix} \begin{pmatrix} x \\ w_1 \end{pmatrix} = \lambda_2 \begin{pmatrix} x \\ w_1 \end{pmatrix}$$

cioè $\lambda_1 x + a^T w_1 = \lambda_2 x$ da cui

$$x = \frac{a^T w_1}{\lambda_2 - \lambda_1}$$

Allora il vettore $U^T \begin{pmatrix} x \\ w_1 \end{pmatrix}$ è autovettore di A relativo all'autovalore λ_2.

Iterando questa tecnica in teoria potrebbero trovarsi tutti gli autovalori di A e per ciascuno di essi un autovettore non nullo; comunque la propagazione degli errori a ciascun passo è drammatica, perché si utilizzano soluzioni sempre più approssimate.

Una ulteriore modifica al metodo delle potenze potrebbe essere sfruttare la conoscenza di un valore approssimato per λ_1 e cercare di migliorare questa approssimazione: questo è realizzato dal *metodo delle potenze inverse*.

Supponiamo che l sia una approssimazione di un autovalore λ di A: allora, se v è un autovettore, $(A - lI)v = (\lambda - l)v$ e quindi è corretto affermare che $(\lambda - l)$ è autovalore di $(A - lI)$, sicché $(\lambda - l)^{-1}$ è autovalore di $(A - lI)^{-1}$; se l'approssimazione l di λ è buona allora $\mu = (\lambda - l)^{-1}$ è l'autovalore di modulo massimo di $(A - lI)^{-1}$.

Dunque si può calcolare μ (e quindi $\lambda = l + 1/\mu$) usando il metodo delle potenze sulla matrice $(A - lI)^{-1}$.

```
 1:#include <math.h>  /* importa la definizione di fabs */
 2:/*
 3:   Metodo delle potenze inverse: data la matrice A ed una approssimazione l
 4:   dell'autovalore di modulo massimo torna una migliore approssimazione di l
 5:   e calcola anche una approssimazione dell'autovettore relativo a l; richiede
 6:   anche il massimo numero di iterazioni da effettuare ed un valore di precisione
 7:   oltre il quale l'approssimazione è ritenuta soddisfacente; torna 0 se è possibile
 8:   calcolare l con la precisione richiesta, torna 1 se la matrice non è invertibile,
 9:   torna 2 se non riesce a calcolare l con la precisione desiderata
10:*/
11:int inv_pow(
12:    float    A[N][N],                                            /* matrice */
13:    float    x[N],                       /* autovettore: usato solo in output */
14:    float    *l,                        /* approssimazione iniziale e finale */
15:    float    prec,                                  /* precisione desiderata */
16:    int      kmax                           /* massimo numero di iterazioni */
17:    )
18:{
19:    float    pivot[N];                    /* usato nella chiamata a factor() */
20:    float    w[size];                /* approssimazione successiva di x[] */
21:    float    lambda;                   /* approssimazione successiva di l */
22:    float    a;                                     /* variabile ausiliaria */
23:    int      i, j, k, k0;                                  /* indici di ciclo */
24:    extern float factor( float A[N][N], int pivot[N-1] );

25:    /* per prima cosa cambia A in A − lI */
26:    for ( i = 0; i < N; ++ i )  A[i][i] -= *l;
27:    if ( factor(A, pivot) == 0 )  return 1;              /* se det A = 0... */
28:    /* inizializza x ad (1,1,...,1) */
29:    for ( i = 0; i < N; ++ i )  x[i] = 1;
30:    lambda = *l;
31:    for ( k = 0; k < kmax; ++ k ) {
32:       /* calcola w = Gx (G sta nella parte inferiore di A),
33:             cioè w_i = x_i + sum_{j=1}^{i-1} a_ij x_j  */
34:       for ( i = 0; i < N; ++ i ) {
35:          w[i] = x[i];
36:          for ( j = 0; j < i; ++ j )
37:             w[i] += A[i][j] * x[j];
38:       }
39:       /* calcola a = ||w||_∞ */
40:       a = fabs(w[0]);
41:       for ( i = 1; i < N; ++ i )
42:          if ( fabs(w[i]) > a)  a = fabs(w[i]);
43:       /* e trova l'indice k0 per il quale w[k0] assume il valore massimo */
44:       for ( k0 = 0 ; ; ++ k0 )
45:          if ( fabs(w[k0]) == a )  break;
46:       /* ora mette in x il normalizzato di w */
47:       for ( i = 0; i < N; ++ i )
48:          x[i] = w[i] / a;
49:       /* e calcola una migliore approssimazione di l */
50:       a = lambda;                                            /* salva lambda */
```

```
51:       lambda = *l + x[k0] / w[k0];
52:       /* controlla se ha raggiunto la precisione voluta */
53:       if ( fabs(a - lambda) <= prec * fabs(lambda) )  return 0;
54:    }
55:    *l = lambda;
56:    return 2;
57:}
```

8 Trasformazioni di Householder

Partiamo da una matrice invertibile S: una *similitudine* è la trasformazione lineare nello spazio delle matrici data da

$$A \mapsto SAS^{-1}$$

Se A e B sono tali che esiste S invertibile tale che $B = SAS^{-1}$ allora A e B si dicono *simili*: la similitudine è iuna relazione di equivalenza.

Inoltre, se $Av = \lambda v$ sono un autovalore ed un suo autovettore di A allora

$$SAS^{-1}Sv = S\lambda v = \lambda Sv$$

sicché A e SAS^{-1} hanno gli stessi autovalori ed i relativi autovettori sono i trasformati di quelli di A per tramite dell'isomorfismo S.

Vale il

Teorema 8.1 *Se A è una matrice reale di ordine n allora esiste una matrice ortogonale U tale che $H = UAU^T$ è una matrice di Hessenberg, tridiagonale se A è simmetrica.*

Chiaramente questo è un caso particolare del teorema spettrale: il vantaggio è che U può costruirsi con un numero finito di passi.

Definizione 8.2 *Un riflettore elementare è una matrice del tipo*

$$U = I - 2uu^T$$

ove u è un vettore di norma 1.

Chiaramente un riflettore elementare determina un isomorfismo che si chiama *trasformazione di Householder*.

Lemma 8.3 *U è simmetrica, ortogonale ed idempotente.*

(Infatti $U^T = (I - 2uu^T)^T)I - 2u^{TT}u^T = U$ e $U^TU = U^2 = (I - 2uu^T)(I - 2uu^T) = I - 2uu^T - 2uu^T + 4uu^Tuu^T = I - 4uu^T + 4uu^T = I$.)

Geometricamente, Ux è il vettore riflesso di x rispetto all'asse perpendicolare a u e passante per l'origine: infatti

$$Ux = x - 2(uu^T)x = x - 2u(u^Tx) = x - 2 < u, x > u$$

dato che $< u, x > u$ è la proiezione di x su u.

I riflettori possono essere utilizzati per azzerare opportune coordinate di vettori:

Teorema 8.4 *Se x è un vettore (non nullo), il riflettore*

$$U = I - \frac{1}{p}uu^T$$

ove $u = x + \sigma e_1$, essendo $\sigma = \pm\|x\|$ e $p = \|u\|^2/2$, è tale che

$$Ux = -\sigma e_1$$

Notiamo intanto che

$$\begin{aligned} p =& \frac{1}{2} < u, u >= \frac{1}{2} < x + \sigma e_1, x + \sigma e_1 >= \frac{1}{2}(\|x\|^2 + 2\sigma x_1 + \sigma^2) \\ =& \frac{1}{2}(\|x\|^2 + 2\sigma x_1 + \|x\|^2) = \|x\|^2 + \sigma x_1 \end{aligned}$$

ove x_1 è la prima coordinata del vettore x nella base canonica.

Ora

$$\begin{aligned} Ux &= x - \frac{1}{p}(uu^T)x = x - \frac{1}{p}(x + \sigma e_1)(x + \sigma e_1)^T x \\ &= x - \frac{1}{p}(x + \sigma e_1)(x^T + \sigma e_1^T)x \\ &= x - \frac{1}{p}(xx^T + \sigma x e_1^T + \sigma e_1 x^T + \sigma^2 e_1 e_1^T)x \\ &= x - \frac{1}{p}(xx^T x + \sigma x e_1^T x + \sigma e_1 x^T x + \sigma^2 e_1 e_1^T x) \\ &= x - \frac{1}{p}(\|x\|^2 x + \sigma x_1 x + \sigma\|x\|^2 e_1 + \sigma^2 x_1 e_1) \\ &= x - \frac{\|x\|^2 + \sigma x_1}{p}x - \frac{\sigma\|x\|^2 + \sigma^2 x_1}{p}e_1 \\ &= -\sigma e_1 \end{aligned}$$

(nell'ultimo passaggio abbiamo sfruttato la precedente $p = \|x\|^2 + \sigma x_1$).

Per determinare completamente un riflettore elementare di questo tipo, basta quindi il numero p e il vettore u: costruiamo direttamente questi a partire da x:

```
 1:#include <math.h> /* importa le definizioni di fabs, pow e sqrt */
 2:/*
 3:   Costruzione di un riflettore elementare in termini di un vettore non nullo x:
 4:   restituisce in x stesso il vettore di riflessione e in come valore il termine
 5:   p = 1/2 ||u||^2
 6:*/
 7:float  elem_refl(
 8:    float x[n],                    /* vettore del quale costruire il riflettore */
 9:    float epsilon                                    /* precisione desiderata */
10:)
11:{
12:    float   sigma;                              /* coefficiente del riflettore */
13:    float   a;                                  /* contiene la norma di x */
14:    int     i;                                           /* indice di ciclo */

15:    /* calcola a = ||x||_∞ */
16:    a = fabs(x[0]);
17:    for ( i = 1; i < n; ++ i )
18:       if ( fabs(x[i]) > a )
19:          a = x[i];
20:    /* ora inizia il ciclo principale */
21:    sigma = 0;
22:    for ( i = 0; i < n; ++ i )
23:       /* se |x_i| > a√ε pone σ = σ + (x_i/a)^2 (trascura quantità inutili) */
24:       if ( fabs(x[i]) > a * sqrt(epsilon) )
25:          sigma += pow(x[i]/a, 2);
26:       /* Il segno di σ è arbitrario, quindi possiamo supporlo > 0 o < 0
27:          a piacimento ed usare il segno che non dà luogo a
28:          cancellazione numerica: σ = sgn x_1 √σ a */
29:    sigma = sqrt(sigma) * a;
30:    return sigma * ( x[0] < 0 ? (x[0] -= sigma) : (x[0] += sigma) );
31:}
```

Conoscendo p ed il vettore u possiamo calcolare $z = Uy$ come segue: per prima cosa determiniamo

$$a = \frac{1}{p} \sum_{i=1}^{n} u_i y_i$$

quindi si calcola il risultato come

$$z = y - au$$

9 Fattorizzazione QR

Generalizziamo la costruzione precedente dei riflettori elementari al seguente caso: abbiamo visto come, dato un vettore $x \neq 0$, produrre un riflettore elementare che lo trasformi in un vettore con la sola prima coordinata non nulla ($U = I - \frac{1}{p}uu^T$); chiediamoci se non sia possibile trovare un riflettore che trasformi x in un vettore con le sole prime k coordinate non nulle.

L'idea è di considerare $U^{(n-k+1)} = I - \frac{1}{p}uu^T$ (sempre con $u = x \pm \|x\|e_1$ e $p = \|u\|^2/2$) nello spazio $\mathbb{R}^{n-k+1}$ e di usarlo per formare un operatore nello spazio $\mathbb{R}^n$ definito come

$$U^{(n)} = I_{k-1} \oplus U^{(n-k+1)} = I - \frac{1}{p_k}u_k u_k^T$$

ove

$$u_k = 0_{k-1} \oplus u_{n-k+1}$$

e

$$p_k = p_{n-k+1} = \frac{1}{2}\|u_{n-k+1}\|^2$$

$U^{(n-k+1)}$ è il riflettore elementare in $\mathbb{R}^{n-k+1}$ tale che

$$U^{(n-k+1)}(x_k e_k + ... + x_n e_n) = -\sigma_k e_k$$

Avendo a disposizione per ciascun $k = 1, ..., n$ riflettori $U_1, ..., U_n$ di questo tipo potremmo, data una matrice A, ridurla in forma triangolare superiore per mezzo delle moltiplicazioni

$$U_{n-1}...U_2U_1A = R$$

in modo che, se $Q = (U_{n-1}...U_2U_1)^T = U_1^T U_2^T ... U_{n-1}^T$ allora

$$Q^T A = R \implies A = QR$$

Si noti che QQ è ortogonale essendo prodotto di matrici ortogonali.

Ciascun fattore U_i è determinato da un numero p_i e da un vettore $u_i \in \mathbb{R}^{n-i+1}$.

Applichiamo ora questa costruzione alla riduzione di una matrice *simmetrica* A in una matrice simile (quindi con gli stessi autovalori) che sia tridiagonale: l'idea è di azzerare successivamente gli elementi sotto la diagonale usando i riflettori U_i.

Si comincia con la prima prima colonna: consideriamo il vettore $(a_{11}, a_{21}, ..., a_{n1})^T$ ed il riflettore U_2 che azzeri gli elementi $a_{31}, ..., a_{n1}$: la matrice (i riflettori sono idempotenti ed ortogonali: $U = U^T = U^{-1}$)

$$A_2 = U_2AU_2$$

è simile ad A; ora scriviamo A a blocchi come

$$\begin{pmatrix} a_{11} & v_1^T \\ v_1 & A_{11} \end{pmatrix}$$

e calcoliamo

$$\begin{aligned} U_2 A U_2 &= \begin{pmatrix} 1 & 0 \\ 0 & U^{(n-1)} \end{pmatrix} \begin{pmatrix} a_{11} & v_1^T \\ v_1 & A_{11} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & U^{(n-1)} \end{pmatrix} \\ &= \begin{pmatrix} a_{11} & v_1^T \\ U^{(n-1)} v_1 & U^{(n-1)} A_{11} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & U^{(n-1)} \end{pmatrix} \\ &= \begin{pmatrix} a_{11} & v_1^T U^{(n-1)} \\ U^{(n-1)} v_1 & U^{(n-1)} A_{11} U^{(n-1)} \end{pmatrix} \end{aligned}$$

Ora osserviamo che $U^{(n-1)}$ applicato ad un vettore di $\mathbb{R}^{n-1}$ ne azzera tutte le componenti tranne la prima, sicché $U^{(n-1)} v_1 = (*, 0, ..., 0)^T$ (con $*$ indichiamo un valore non nullo); inoltre $v_1^T U^{(n-1)} = ((U^{(n-1)}) T v_1)^T = (U^{(n-1)} v_1)^T = (*, 0, ..., 0)$. Dunque A_2 sarà una matrice (simmetrica) della forma

$$A_2 = \begin{pmatrix} * & * & 0 & ... & 0 \\ * & * & * & ... & * \\ 0 & * & * & ... & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & * & * & ... & * \end{pmatrix}$$

Ora iteriamo il procedimento considerando un riflettore U_3 che applicato alla seconda colonna di questa matrice A_2 ne azzeri gli ultimi $n-3$ elementi, in modo che $U_3 A_2 U_3$ sia una matrice

$$A_2 = \begin{pmatrix} * & * & 0 & 0 & ... & 0 \\ * & * & * & 0 & ... & 0 \\ 0 & * & * & * & ... & * \\ 0 & 0 & * & * & ... & * \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & * & * & ... & * \end{pmatrix}$$

Proseguiamo in questo modo producendoa alla fine del processo una matrice simmetrica

$$A_{n-1} = U_{n-1} ... U_3 U_2 A U_2 U_3 ... U_{n-1}$$

tridiagonale, simile ad A.

Osserviamo inoltre che, se A non è simmetrica questo procedimento fornisce una matrice di Hessenberg simile ad A.

Diamo ora un algoritmo per questo procedimento: come al solito non sarà necessario costruire esplicitamente le matrici U_i, ma solo determinarle per mezzo delle coppie (p_i, u_i).

```
 1:#include <math.h> /* importa le definizioni di fabs, pow e sqrt */
 2:/*
 3:  Data una matrice simmetrica A costruisce una tridiagonale a lei simile usando
 4:   i riflettori elementari; poiché il risultato è determinato semplicemente dalla
 5:   diagonale e dalla diagonale immediatamente superiore (quella inferiore è
 6:   uguale a questa dato che A è simmetrica), il risultato consisterà in questa
 7:   coppia di diagonali: la diagonale principale nel vettore d1 e la diagonale
 8:   secondaria nel vettore d2; inoltre le coppie (p,u) che determinano i riflettori
 9:   elementari sono poste nella parte triangolare strettamente inferiore di A
10:   come colonne il cui elemento sulla diagonale è p (o zero se la trasformazione
11:   in quel passo è l'identità) con sotto il vettore colonna u; la parte triangolare
12:   superiore di A è lasciata intatta.
13:*/
14:void qr_factor(
15:    float    A[N][N],                      /* matrice da trasformare */
16:    float    d1[N],              /* conterrà la diagonale principale */
17:    float    d2[N],                      /* conterrà la codiagonale */
18:    float      epsilon                     /* precisione desiderata */
19:    )
20:{
21:    float a;                                /* variabile ausiliaria */
22:    float sigma;                       /* coefficiente del riflettore */
23:    int   i, j, k;                                /* indici di ciclo */

24:    for ( k = 0; k < n-2; ++ k ) {
25:        /* decompone a blocchi la matrice */
26:        d1[k] = A[k][k];
27:        /* determina il riflettore U_{k+1}: prima calcola a = max_{i=k+1}^n |a_{ik}| */
28:        a = fabs(A[k+1][k]);
29:        for ( i = k+2; i < N; ++ i )
30:            if ( fabs(A[i][k]) > a )  a = A[i][k];
31:        /* se la colonna è di tutti zeri pone U_{k+1} = I */
32:        if ( a == 0 ) {
33:            A[k][k] = 0;                                   /* p_{k+1} = 0 */
34:            d2[k] = 0;
35:            continue;                        /* ripete il ciclo for (k...) */
36:        }
37:        /* ora calcola p_{k+1} */
38:        sigma = 0;
39:        for ( i = k+1; i < N; ++ i )
40:            /* se |a_{ik}| > a√ε pone σ = σ + (a_{ik}/a)^2 (trascura qtà inutili) */
41:            if ( fabs(A[i][k]) > a * sqrt(epsilon) )
42:                sigma += pow(A[i][k]/a, 2);
43:        /* Il segno di σ è arbitrario, quindi possiamo supporlo > 0 o < 0
44:            a piacimento ed usare il segno che non dà luogo a cancellazione
45:            numerica: σ = sgn a_{k+1,k} √σ a */
46:        sigma = sqrt(sigma) * a;
```

```
47:       if ( A[k+1][k] < 0 )
48:          A[k+1][k] -=sigma;
49:       else
50:          A[k+1][k] +=sigma;
51:       A[k][k] = sigma * A[k+1][k];                          /* p_{k+1} */
52:       d2[k] = -sigma;                             /* U_{k+1}v_k = −σe_k */
53:       /* calcola la matrice B = U_{k+1}A_k */
54:       for ( j = k+1; j < N; ++ j ) {
55:          /* a = 1/a_{kk} ∑_{i=k+1}^{n} a_{ik}a_{ij} */
56:          a = A[k+1][k] * A[k+1][j];
57:          for ( i = k+2; i < N; ++ i )
58:             a += A[i][k] * A[i][j];
59:          a /= A[k][k];
60:          /* a_{ij} = a_{ij} − aa_{jk} */
61:          for ( i = k+1; i < N; ++ i )
62:             A[i][j] -= a * A[i][k];
63:       }
64:       /* calcola la matrice A_{k+1} = BU_{k+1} */
65:       for ( i = k+1; i < N; ++ i ) {
66:          /* a = 1/a_{kk} ∑_{j=k+1}^{n} a_{ij}a_{jk} */
67:          a = A[i][k+1] * A[k+1][k];
68:          for ( j = k+2; j < N; ++ j )
69:             a += A[i][j] * A[j][k];
70:          a /= A[k][k];
71:          /* a_{ij} = a_{ij} − aa_{jk} */
72:          for ( j = k+1; j < N; ++ j )
73:             A[i][j] -= a * A[j][k];
74:       }
75:    }                                                /* ciclo for (k...) */
76:    d1[n-2] = A[n-2][n-2];
77:    d1[n-1] = A[n-1][n-1];
78:    d2[n-2] = A[n][n-1];
79:}
```

Una applicazione di questo metodo è alla determinazione degli autovalori di una matrice simmetrica: per prima cosa la si riduce in forma tridiagonale

$$B = \begin{pmatrix} a_1 & b_1 & 0 & \dots & 0 & 0 \\ b_1 & a_2 & b_2 & \dots & 0 & 0 \\ 0 & b_2 & a_3 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{n-1} & b_{n-1} \\ 0 & 0 & 0 & \dots & b_{n-1} & a_n \end{pmatrix}$$

Evidentemente possiamo supporre che ciascun b_i sia diverso dallo zero, altrimenti la matrice si decompone in somma diretta di matrici tridiagonali

con questa proprietà, e gli autovalori della matrice sono dati da quelli delle singole matrici addendi diretti.

Ora scriviamo $\lambda I - B$ il cui determinante è il polinomio caratteristico:

$$\lambda I - B = \begin{pmatrix} \lambda - a_1 & -b_1 & 0 & ... & 0 & 0 \\ -b_1 & \lambda - a_2 & -b_2 & ... & 0 & 0 \\ 0 & -b_2 & \lambda - a_3 & ... & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & ... & \lambda - a_{n-1} & -b_{n-1} \\ 0 & 0 & 0 & ... & -b_{n-1} & \lambda - a_n \end{pmatrix}$$

e consideriamo la successione di polinomi

$$\begin{cases} p_0(\lambda) = 1 \\ p_1(\lambda) = \lambda - a_1 \\ ... \\ p_i(\lambda) = (\lambda - a_1)p_{i-1}(\lambda) - b_{i-1}^2 p_{i-2}(\lambda) \\ ... \end{cases}$$

Si tratta dei determinanti delle sottomatrici principali di ordine i estratte dalla matrice $\lambda I - B$; in particolare $p_n(\lambda)$ è il polinomio caratteristico di B.

Si possono dimostrare le seguenti proprietà dei polinomi p_i:

(1) Le radici di p_i sono reali e distinte (se $i > 1$);

(2) Per ogni $x \in \mathbb{R}$ tale che se $p_n(x) \neq 0$, il numero di radici (cioè di autovalori) maggiori di x è dato dal numero di variazioni del segno nella successione di numeri reali

$$1,\ p_1(x),\ p_2(x),\ ...,1, p_n(x)$$

dalla quale siano stati eliminati eventuali valori nulli.

Per localizzare le radici del polinomio caratteristico, cioè gli autovalori della matrice è anche utile il teorema di Gerschgorin:

Teorema 9.1 *Se A è una matrice $n \times n$, poniamo*

$$r_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \qquad e \qquad c_j = \sum_{\substack{i=1 \\ i \neq j}}^{n} |a_{ij}|$$

Allora

(1) *Ogni autovalore di A appartiene all'insieme*

$$R = \bigcup_{i=1}^{n} R_i \qquad \textit{dove} \qquad R_i = \{z \mid |z - a_{ii}| < r_i\}$$

(2) *Ogni autovalore di A appartiene all'insieme*

$$C = \bigcup_{i=j}^{n} C_j \qquad \textit{dove} \qquad C_j = \{z \mid |z - a_{jj}| < c_j\}$$

(3) *Ogni componente connessa di R o C contiene tanti autovalori contati con le loro molteplicità quanti sono i cerchi della componente stessa.*

Ora: supponiamo di conoscere un intervallo $[a, b]$ che contenga tutti gli autovalori di B (ad esempio considerando il raggio spettrale della matrice o utilizzando il teorema di Gerschgorin); allora possiamo, considerato $c = (b-a)/2$, dire quanti autovalori si trovano nell'intervallo $[c, b)$ e quanti nell'intervallo (a, c) usando la 2) con $c = x$; a loro volta suddividiamo questi intervalli a metà, considerando cioè $c_1 = (c - a)/2$ e $c_2 = (b - c)/2$ e di nuovo determiniamo il numero di autovalori che cadono in questi intervalli usando la 2).

Iterando queste *bisezioni* possiamo (dato che gli autovalori sono in numero finito, non pi di n) con un numero finito di bisezioni determinare una suddivisione di (a, b) ion intervalli finiti di eguale ampiezza tali che ciascuno di essi non contenga più di un autovalore; allora, se λ è contenuto nell'intervallo (c_i, c_j) possiamo considerare come valore iniziale per l'approssimazione di λ il punto medio $(c_j - c_i)/2$; a questo punto, magari col metodo delle potenze inverse, possiamo provare a determinare migliori approssimazioni degli autovalori stessi.

10 Approssimazione di funzioni

Il problema dell'approssimazione consiste nel sostituire una funzione f con un'altra funzione che abbia un comportamento simile ma che consenta in pratica di effettuare i calcoli richiesti: ad esempio si potrebbe voler calcolare un integrale, o simili.

Le classi di funzioni alle quali "si attinge" per trovare candidati validi per queste sostituzioni sono:

(1) Polinomi di grado n: $\mathbb{P}_n$

$$\sum_{i=0}^{n} a_n x^n$$

ad esempio una funzione continua in un compatto si approssima sempre, con precisione desiderata $\varepsilon > 0$, con un poilinomio P_ε;

(2) Polinomi trigonometrici di grado n e periodo ω: $\mathbb{T}_n(\omega)$

$$a_0 + \sum_{i=1}^{n} (a_i \cos i\omega x + b_i \sin i\omega x)$$

ad esempio una funzione continua e periodica si approssima in un compatto con polinomi trigonometrici;

(3) Funzioni razionali quozienti di polinomi di gradi n e d: $\mathbb{R}_{n,d}$

$$\frac{P_n(x)}{Q_d(x)}$$

ad esempio funzioni che hanno poli o singolarità nell'intorno di un certo punto si lasciano approssimare in $\mathbb{R}_{n,d}$.

(4) Somme esponenziali di ordine n: $\mathbb{E}_n$

$$\sum_{i=1}^{n} a_i \exp(-b_i x)$$

ad esempio funzioni esponenziali possono approssimarsi in $\mathbb{E}_n$.

(5) Funzioni spline di ordine n: S_n; si tratta di funzioni C^n che a tratti sono polinomi di grado n.

Infine si possono moltiplicare fra loro elementi appartenenti a diverse fra le classi precedenti.

Supponiamo di voler approssimare una funzione $f(x)$ della quale conosciamo il valore in un certo insieme finito $\{x_0, ..., x_m\}$ di punti; lo faremo scegliendo l'approssimante $f_n(x)$ in una classe fra le precedenti.

Questa scelta può avvenire ad esempio in base ad uno dei seguenti criteri:

(a) Interpolazione: si sceglie f_n in modo che per ogni $i = 0, ..., m$ si abbia $f_n(x_i) = f(x_i)$; ovviamente la conoscenza degli $f(x_i)$ deve essere sufficientemente precisa.

(b) Minimi quadrati: si sceglie una f_n nella classe scelta "disti il meno possibile" da f nei punti nei quali questa è nota: cioè si sceglie la f_n che rende minimo il numero

$$\sum_{i=0}^{m}(f_n(x_i) - f(x_i))^2$$

Talvolta è opportuno "pesare" la distanza come

$$\sum_{i=0}^{m}(f_n(x_i) - f(x_i))^2 p_i$$

ove $p_i > 0$ sono appunti dei pesi che sono tanto più grandi quanto più stimiamo essere accurata la conoscenza del valore della funzione $f(x_i)$ la cui distanza stiamo pesando on p_i.

(c) MiniMax: analogo al precedente ma si usa una distanza uniforme e non euclidea; cioè f_n è scelta in modo da minimizzare il valore

$$\max_{0 \le i \le m} |f_n(x_i) - f(x_i)|$$

Poiché le funzioni $f(x)$ che andremo ad approssimare appartengono sempre a qualche spazio vettoriale di funzioni, cercheremo di approssimarle con funzioni f_n che stanno in qualche sotto-spazio di dimensione finita: in questo modo, fissata una base di questo sotto-spazio, possiamo limitare il numero di funzioni approssimanti in considerazione; la scelta di questa base influenzerà in modo essenziale le prestazioni numeriche dell'approssimazione.

11 Interpolazione di Lagrange

Proviamo ad approssimare funzioni definite in un intervallo $[a, b]$ con polinomi usando il criterio di interpolazione.

Consideriamo dunque gli $n+1$ punti $\{x_0, ..., x_n\}$ nei quali delle approssimazioni $y_i = f(x_i)$ del valore della funzione sono note (supporremo ovviamente distinti tali punti); consideriamo il polinomio di grado n

$$p_n(x) = a_0 + a_1 x + ... + a_n x^n$$

ed imponiamo i vincoli $p_n(x_i) = y_i$:

$$\begin{cases} a_0 + a_1 x_0 + ... + a_n x_0^n = y_0 \\ \text{............................} \\ a_0 + a_1 x_n + ... + a_n x_n^n = y_n \end{cases}$$

Poiché vogliamo determinare p_n, le nostre incognite sono i numeri a_i, e questi soddisfano un sistema lineare la cui matrice dei coefficienti è quella di Vandermonde V_n ed il cui vettore dei termini noti è il vettore degli y_i:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Come noto il determinante di Vandermonde è

$$\det V_n = \prod_{i>j} (x_i - x_j)$$

ed è diverso da zero nell'ipotesi $i \neq j \implies x_i \neq x_j$: dunque il sistema lineare possiede un'unica soluzione, cioè esiste un unico polinomio p_n di grado n che soddisfa ai vincoli $p_n(x_i) = y_i$ con $i = 0, 1, ..., n$.

Purtroppo la soluzione di un sistema lineare la cui matrice dei coefficienti è una di Vandermonde è mal condizionato, nel senso che a piccole variazioni dei dati corrispondono variazioni della soluzione approssimata non necessariamente dello stesso ordine di grandezza.

Dunque questo ragionamento vale solo teoricamente per accertare l'esistenza ed unicità del polinomio p_n, non per il suo calcolo effettivo.

Il metodo di Lagrange consiste in questo: fissiamo $j \in \{0, 1, ..., n\}$ e consideriamo la funzione che nei punti x_i vale δ_{ij}; sia l_j il suo polinomio di interpolazione: siccome le x_i con $i \neq j$ (che sono n numeri distinti) debbono essere radici di l_j (che ha grado n), per il teorema fondamentale dell'algebra queste sono tutte e sole le radici di $l_j(x)$, e dunque abbiamo che

$$l_j(x) = a_n \prod_{\substack{i=0 \\ i \neq j}}^{n} (x - x_i)$$

ove a_n è il coefficiente del termine di grado n in l_j: questo si determina facilmente, dato che $l_j(x_j) = 1$ sicché

$$1 = a_n \prod_{\substack{i=0 \\ i \neq j}}^{n} (x_j - x_i) \implies a_n = \frac{1}{\displaystyle\prod_{\substack{i=0 \\ i \neq j}}^{n} (x_j - x_i)}$$

Dunque

$$l_j(x) = \frac{\prod_{\substack{i=0\\ i\neq j}}^{n}(x - x_i)}{\prod_{\substack{i=0\\ i\neq j}}^{n}(x_j - x_i)} = \frac{\omega_{n+1}(x)}{(x - x_j)\omega'_{n+1}(x_j)}$$

ove

$$\omega_{n+1}(x) = \prod_{i=0}^{n}(x - x_i) \qquad \text{e} \qquad \omega'_{n+1}(x_j) = \frac{d}{dx}\Big|_{x=x_j}\omega_{n+1}(x) = \prod_{\substack{i=0\\ i\neq j}}^{n}(x_j - x_i)$$

A questo punto il polinomio di interpolazione di $f(x)$ è combinazione lineare degli $l_j(x)$ come

$$P_n(x) = \sum_{j=0}^{n} y_j l_j(x)$$

Infatti

$$P_n(x_i) = \sum_{j=0}^{n} y_j l_j(x_i) = y_i$$

I polinomi $l_j(x)$ si dicono *polinomi fondamentali di Lagrange*.

Stimiamo ora l'errore commesso nel sostituire a $f(x)$ il suo polinomio di interpolazione nel caso in cui $f(x)$ presenti un comportamento regolare, cioè appartenga a qualche spazio $C^k(a, b)$ con $k > 0$.

Precisamente supponiamo che $f \in C^{n+1}(a, b)$ e stimiamo l'errore

$$E_n(x) = f(x) - P_n(x)$$

Poiché $E(x_i) = 0$ per $i = 0, 1, ..., n$, è naturale scrivere

$$E_n(x) = \prod_{i=0}^{n}(x - x_i)R_n(x) = \omega_{n+1}(x)R_n(x)$$

per qualche funzione $R_n(x)$ che deve essere in $C^{n+1}(a, b)$; dunque

$$f(x) = P_n(x) + \omega_{n+1}(x)R_n(x)$$

Introduciamo ancora una funzione

$$G(t) = f(t) - P_n(t) - \omega_{n+1}(t)R_n(x)$$

(con $x \neq x_i$) che ovviamente si annulla in $x, x_0, x_1, ..., x_n$.

Poiché si tratta di una funzione derivabile $n+1$ volte possiamo applicare il teorema di Rolle in ciascuno degli intervalli di estremi $x, x_0, x_1, ..., x_n$, ottenendo dei punti $\xi_1^{(1)}, \xi_2^{(1)}, ...\xi_{n+1}^{(1)}$ nei quali la derivata di G si annulla; ripetiamo lo stesso ragionamento sulle derivate successive di G fino all'ordine $n+1$, troviamo senz'altro un punto $\xi \in (a,b)$ tale che $G^{(n+1)}(\xi) = 0$, sicché

$$G^{(n+1)}(\xi) = f^{(n+1)}(\xi) - (n+1)!R_n(x) = 0$$

Questo ξ dipenderà da x.

Comunque abbiamo determinato $R_n(x)$ e quindi in qualche misura f:

$$R_n(x) = \frac{1}{(n+1)!}f^{(n+1)}(\xi(x)) \implies f(x) = \sum_{j=0}^{n} f(x_j)l_j(x) + \frac{\omega_{n+1}(x)}{(n+1)!}f^{(n+1)}(\xi)$$

Ad esempio, per $f \equiv 1$, troviamo

$$\sum_{j=0}^{n} l_j(x) = 1$$

Il condizionamento dell'interpolazione di Lagrange può studiarsi come segue: supponiamo di perturbare i valori $f(x_0), ..., f(x_n)$ come

$$\overline{y_i} = f(x_i) + \varepsilon_i$$

Allora possiamo considerare il polinomio

$$\overline{P_n}(x) = \sum_{j=0}^{n} \overline{y_j} l_j(x)$$

e maggiorare l'errore

$$\begin{aligned}|P_n(x) - \overline{P_n}(x)| &= \left| \sum_{j=0}^{n} l_j(x)(f(x_j) - \overline{y_j}) \right| \leq \left(\sum_{j=0}^{n} |l_j(x)| \right) \max_{0 \leq j \leq n} |f(x_j) - \overline{y_j}| \\ &= \left(\sum_{j=0}^{n} |l_j(x)| \right) \max_{0 \leq j \leq n} |\varepsilon_j| \leq \left\| \sum_{j=0}^{n} |l_j(x)| \right\|_\infty \|\varepsilon\|_\infty\end{aligned}$$

ove $\varepsilon = (\varepsilon_0, ..., \varepsilon_n)^T$); da questo, dato che $\|P_n\|_\infty \geq \max_j |P_n(x_j)| = \max_j |f(x_j)|$, troviamo, per $y = (f(x_0), ..., f(x_n))^T$:

$$\frac{\|P_n(x) - \overline{P_n}(x)\|_\infty}{\|P_n(x)\|_\infty} \leq \Lambda_n \frac{\|\varepsilon_j\|_\infty}{\|f_\infty\|}$$

avendo definito il *numero di Lebesgue* come

$$\Lambda_n = \left\| \sum_{j=0}^{n} |l_j(x)| \right\|_\infty$$

Teorema 11.1 *Se $f \in C[a,b]$ e P_n è il polinomio di interpolazione di Lagrange relativo ai valori $x_0, ..., x_n \in [a,b]$ allora*

$$\|f - P_n\|_\infty \leq (1 + \Lambda_n)E_n(f)$$

ove

$$E_n(f) = \min_{Q \in \mathbb{P}_n} \|f - Q\|_\infty$$

(ove $\mathbb{P}_n$ è lo spazio dei polinomi di grado non superiore a n).

L'idea è di considerare un polinomio $Q \in \mathbb{P}_n$ che renda minima la $\|f - Q\|_\infty$: allora $q(x) = \sum_j l_j(x)q(x_j)$ sicché

$$\begin{aligned} f(x) - P_n(x) =& (f(x) - Q_n(x)) - (P_n(x) - Q_n(x)) \\ =& (f(x) - Q_n(x)) - \sum_{j=0}^{n} l_j(x)(f(x_j) - Q(x_j)) \end{aligned}$$

Fissato n, scelti $x_0, ..., x_n$ in modo da minimizzare Λ_n, denotando con $\overline{\Lambda_n}$ questo minimo si può dimostrare che

$$\lim_{n\to\infty} \frac{\overline{\Lambda_n}}{\log n} = \frac{2}{\pi}$$

12 Formule di Newton

Consideriamo $x_0, ..., x_n$ assegnati e distinti; la *differenza divisa* di ordine 1 di f è

$$f[x_0, x_1] = f[x_1, x_0] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

La differenza divisa di ordine 2 è

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

ed in generale, quella di ordine n è

$$f[x_0, ..., x_n] = \frac{f[x_1, ..., x_n] - f[x_0, ..., x_{n-1}]}{x_n - x_0}$$

Per induzione si ha che

$$f[x_0, ..., x_n] = \sum_{i=0}^{n} \frac{f(x_i)}{\prod_{\substack{j=0 \\ j\neq i}}^{n} (x_i - x_j)}$$

da cui si vede immediatamente che $f[x_0, ..., x_n]$ è una funzione simmetrica (nel caso 1 lo è per definizione).

Vale il

Teorema 12.1 *Se* $f \in C^k[a, b]$ *e* $x0, ..., x_n \in [a, b]$ *(non necessariamente distinti) allora esiste* $\xi \in (a, b)$ *con* $\min x_i \leq \xi \leq \max x_i$ *tale che*

$$f[x_0, ..., x_k] = \frac{f^{(k)}(\xi)}{k!}$$

In particolare $f[x, x, ..., x] = \dfrac{f^{(k)}(x)}{k!}$.

Ora consideriamo

$$f[x, x_0] = \frac{f(x_0) - f(x)}{x_0 - x} \implies f(x) = f(x_0) + (x - x_0)f[x, x_0]$$

Questo definisce un polinomio interpolante di grado zero $P_0(x) = f(x_0)$ e $(x-x_0)f[x, x_0]$ è l'errore $f(x)-P_0(x)$; se aggiungiamo un valore x_1 possiamo considerare

$$f[x, x_0, x_1] = \frac{f[x_0, x_1] - f[x, x_0]}{x1 - x} \implies f[x, x_0] = f[x_0, x_1]+(x-x_1)f[x, x_0, x_1]$$

cioè

$$f(x) = f(x_0) + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x, x_0, x_1]$$

Abbiamo dunque un nuovo polinomio interpolante $P_1(x) = f(x_0) + (x - x_0)f[x_0, x_1]$; proseguendo in questo modo troviamo una successione di polinomi di interpolazione per quanti sono i valori x_i a disposizione:

$$f(x) = P_n(x) + E_n(x)$$

ove

$$P_n(x) = f(x_0) + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] + ... \\ + (x - x_0)...(x - x_{n-1})f[x_0, ..., x_n]$$

e

$$E_n(x) = (x - x_0)...(x - x_n)f[x, x_0, ..., x_n]$$

Questa è la *formula di interpolazione di Newton*.

Scriviamo un algoritmo che dati $x_0, ..., x_n$ e $f(x_0), ..., f(x_n)$ costruisca il valore del polinomio di interpolazione di Newton in un punto p.

```
 1:/* deve essere definita una costante N contenente il numero di punti x_i */
 2:/*
 3:   Dati i vettori x e y contenenti i punti ed i valori della funzione in questi
 4:   punti, e dato un punto t restituisce il valore del polinomio di Newton in t
 5:*/
 6:float  newton(
 7:   float    x[N+1],                                         /* x_0, ..., x_n */
 8:   float    y[N+1],                                   /* f(x_0), ..., f(x_n) */
 9:   float    t                       /* valore nel quale calcolare il polinomio */
10:   )
11:{
12:   float    p;                                           /* conterrà il risultato */
13:   int      i, j;                                             /* indici di ciclo */

14:   /* Calcola le colonne della tabella delle differenze divise e
15:       lascia in y[] la diagonale di questa tabella */
16:   for ( i = 1; i <= N; ++ i )
17:      for ( j = N; j > i; -- j )
18:         y[j] = (y[j] - y[j-1]) / (x[j] - x[j-i]);
19:   /* valuta il polinomio di Newton in t */
20:   p = y[n];
21:   for ( i = N-1; i >= 0; -- i )
22:      p = y[i] + (t - x[i]) * p;
23:   return p;
24:}
```

13 Funzioni polinomiali a tratti

Consideriamo la sequenza di punti

$$a = x_0 < x_1 < ... < x_{n-1} < x_n = b$$

Più n è grande e più la funzione polinomiale approssimante tende ad oscillare: può essere quindi conveniente considerare come approssimante non un singolo polinomio di grado n ma una sequenza di polinomi di grado d al più tre.

Ad esempio nel caso $d = 1$ questo si riduce ad approssimare il grafico della funzione con la spezzata che congiunge i punti del piano $(x_i, f(x_i))$, e che ha equazione

$$x \in [x_i, x_{i+1}] \implies f_n(x) = \frac{(x_{i+1} - x)y_i + (x - x_i)y_{i+1}}{x_{i+1} - x_i}$$

Tuttavia questa funzione, per quanto continua, non è sempre derivabile.

Definizione 13.1 *Sia $d \geq 1$: $S_d(x)$ è la funzione spline di ordine d associata ai dati $x_0, ..., x_n$ se*

(1) *$S_d(x)$ è un polinomio di grado d in ciascun intervallo $[x_i, x_{i+1}]$.*

(2) *$S_d(x) \in C^{d-1}(a, b)$.*

(3) *$S_d(x_i) = f(x_i)$.*

Ovviamente la derivata di una spline di ordine k è una spline di ordine $k - 1$ e la primitiva di una spline di ordine k è una spline di ordine $k + 1$.

Consideriamo ad esempio le spline cubiche: dati $x_0, ..., x_n$ ed i valori assunti da f in questi punti: $y_0, ..., y_n$ dobbiamo avere che, in ciascun tratto $[x_i, x_{i+1}]$

$$S_3(x) = a_i + b_i x + c_i x^2 + d_i x^3$$

e che per ogni $i = 0, ..., n$:

$$\lim_{x \to x_i^-} \frac{S_3^{(k)}(x) - S_3^{(k)}(x)}{x - x_i} = \lim_{x \to x_i^+} \frac{S_3^{(k)}(x) - S_3^{(k)}(x)}{x - x_i}$$

ove $k < 3 - 1 = 2$. Ovviamente anche le $S_3(x_i) = y_i$ vanno imposte.

Siano

$$M_i = S_3''(x_i)$$

In ciascun intervallo $[x_i, x_{i+1}]$ S_3'' è una funzione lineare, precisamente

$$S_3''(x) = \frac{(x_{i+1} - x)M_i - (x - x_i)M_{i+1}}{x_{i+1} - x_i}$$

Infatti $S_3''(x)$ in $[x_i, x_{i+1}]$ rappresenta la retta per i punti (x_i, M_i) e (x_{i+1}, M_{i+1}) cioè la retta di equazione

$$\frac{y - M_i}{M_{i+1} - M_i} = \frac{x - x_i}{x_{i+1} - x_i}$$

Quindi

$$y = \frac{(x - x_i)(M_{i+1} - M_i) + M_i(x_{i+1} - x_i)}{x_{i+1} - x_i} = \frac{(x - x_i)M_{i+1} + M_i(x_{i+1} - x)}{x_{i+1} - x_i}$$

cioè l'equazione precedente, dato che $M_i = S_3''(x_i)$ e $y = S_3''(x)$.

A questo punto identifichiamo questa y con $S_3''(x) = 2c_i + 6d_i x$ ottenendo

$$c_i = \frac{1}{2}\frac{M_i x_{i+1} - M_{i+1} x_i}{x_{i+1} - x_i}$$

e

$$d_i = \frac{1}{6}\frac{M_{i+1} - M_i}{x_{i+1} - x_i}$$

da cui

$$S_3(x) = a_i + b_i x + \frac{1}{2}\frac{M_i x_{i+1} - M_{i+1} x_i}{x_{i+1} - x_i} x^2 + \frac{1}{6}\frac{M_{i+1} - M_i}{x_{i+1} - x_i} x^3$$

Ora imponiamo che $S_3(x_i) = y_i$ e $S_3(x_{i+1}) = y_{i+1}$ ottenendo il sistema lineare

$$\begin{cases} y_i = a_i + b_i x_i + \frac{1}{2}\frac{M_i x_{i+1} - M_{i+1} x_i}{x_{i+1} - x_i} x_i^2 + \frac{1}{6}\frac{M_{i+1} - M_i}{x_{i+1} - x_i} x_i^3 \\ y_{i+1} = a_i + b_i x_{i+1} + \frac{1}{2}\frac{M_i x_{i+1} - M_{i+1} x_i}{x_{i+1} - x_i} x_{i+1}^2 + \frac{1}{6}\frac{M_{i+1} - M_i}{x_{i+1} - x_i} x_{i+1}^3 \end{cases}$$

che consente di determinare a_i e b_i.

Infine si debbono trovare gli M_i, e questo si fa imponendo la continuità delle derivate prime delle $S_3(x)$, cioè delle

$$S_3'(x) = b_i + \frac{M_i x_{i+1} - M_{i+1} x_i}{x_{i+1} - x_i} x + \frac{1}{2}\frac{M_{i+1} - M_i}{x_{i+1} - x_i} x^2$$

Si deve imporre che

$$\lim_{x \to x_i^-} \frac{S_3'(x) - S_3'(x)}{x - x_i} = \lim_{x \to x_i^+} \frac{S_3'(x) - S_3'(x)}{x - x_i}$$

cioè che

$$\lim_{x \to x_i^-} \left(b_{i-1} + \frac{M_{i-1} x_i - M_i x_{i-1}}{x_i - x_{i-1}} x + \frac{1}{2}\frac{M_i - M_{i-1}}{x_i - x_{i-1}} x^2 \right) =$$
$$= \lim_{x \to x_i^+} \left(b_i + \frac{M_i x_{i+1} - M_{i+1} x_i}{x_{i+1} - x_i} x + \frac{1}{2}\frac{M_{i+1} - M_i}{x_{i+1} - x_i} x^2 \right)$$

Infine resteranno due parametri incogniti: M_0 e M_n; per determinarli ci sono vari metodi.

1. Si può ad esempio imporre la coincidenza delle derivate prime agli estremi dell'intervallo:

$$S_3'(x_0) = f'(x_0) \qquad \text{e} \qquad S_3'(x_n) = f'(x_n)$$

A conti fatti si ottiene il sistema lineare tridiagonale simmetrico

$$\begin{pmatrix} 2(x_1-x_0) & x_1-x_0 & 0 & \dots & 0 & 0 & 0 \\ x_1-x_0 & 2(x_2-x_0) & x_2-x_1 & \dots & 0 & 0 & 0 \\ 0 & x_2-x_1 & 2(x_3-x_2) & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & x_{n-1}-x_{n-2} & 2(x_{n-1}-x_{n-3}) & x_n-x_{n-1} \\ 0 & 0 & 0 & \dots & 0 & x_n-x_{n-1} & 2(x_n-x_{n-1}) \end{pmatrix} \times$$

$$\times \begin{pmatrix} M_0 \\ M_1 \\ M_2 \\ \vdots \\ M_{n-1} \\ M_n \end{pmatrix} = \begin{pmatrix} \frac{y_1-y_0}{x_1-x_0} - f'(x_0) \\ \frac{y_2-y_1}{x_2-x_1} - \frac{y_1-y_0}{x_1-x_0} \\ \frac{y_3-y_2}{x_3-x_2} - \frac{y_2-y_1}{x_2-x_1} \\ \vdots \\ \frac{y_n-y_{n-1}}{x_n-x_{n-1}} - \frac{y_{n-1}-y_{n-2}}{x_{n-1}-x_{n-2}} \\ f'(x_n) - \frac{y_n-y_{n-1}}{x_n-x_{n-1}} \end{pmatrix}$$

Questo è un sistema non singolare a diagonale dominante che si può risolvere con Gauss.

2. Analogamente possiamo imporre la coincidenza delle derivate seconde:

$$M_0 = S_3''(x_0) = f''(x_0) \qquad \text{e} \qquad M_n = S_3''(x_n) = f''(x_n)$$

Se si impone che

$$M_0 = M_n = 0$$

le splines così ottenute si dicono *naturali*.

3. Se il fenomeno è periodico, cioè $y_0 = y_n$ si impone di solito

$$S_3'(x_0) = S_3'(x_n) \qquad \text{e} \qquad S_3''(x_0) = S_3''(x_n)$$

Fra tutte le funzioni $f \in C^2(a, b)$ tali che

(1) $f(x_i) = y_i$ per $i = 0, ..., n$;

(2) $f'(x_0) = y_0'$ e $f'(x_n) = y_n'$, oppure $f''(x_0) = f''(x_n) = 0$ oppure $f'(x_0) = f'(x_n)$ e $f''(x_0) = f''(x_n)$

le splines cubiche sono quelle che minimizzano il funzionale

$$\int_{x_0}^{x_n} (f''(x))^2 dx$$

Le splines cubiche naturali minimizzano questo funzionale nella classe delle funzioni $C^2(a,b)$ tali che $f(x_i) = y_i$.

14 Minimi quadrati

Supponiamo di avere una quantità di $m+1$ dati $\{x_i\}$ (non necessariamente distinti) ed i corrispondenti valori $\{y_i = f(x_i)\}$, e di voler approssimare f attingendo da un insieme di $n+1$ funzioni $\{\varphi_i\}$ con $n \ll m$, per combinazione lineare

$$f_n(x) = \sum_{i=0}^{n} c_i \varphi_i(x)$$

a coefficienti costanti determinati col metodo dei minimi quadrati, cioè in modo che

$$\varepsilon^2 = \sum_{i=0}^{m} (y_i - f_n(x_i))^2$$

sia minimo.

Se $r_i = y_i - f_n(x_i)$ sono gli errori, i coefficienti c_i sono determinati come soluzioni del sistema

$$\begin{pmatrix} \varphi_0(x_0) & \varphi_1(x_0) & \dots & \varphi_n(x_0) \\ \varphi_0(x_1) & \varphi_1(x_1) & \dots & \varphi_n(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_0(x_m) & \varphi_1(x_m) & \dots & \varphi_n(x_m) \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \end{pmatrix} + \begin{pmatrix} r_0 \\ r_1 \\ \vdots \\ r_m \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}$$

in modo che la funzione $\sqrt{\sum_{i=0}^{m} r_i^2}$ sia minimizzata.

Si noti che $m > n$ quindi ci sono più soluzioni che incognite e non si può sperare di trovare una soluzione con $r = 0$.

Abbiamo dunque una matrice A $(n+1) \times (n+1)$, un vettore b in $\mathbb{R}^{n+1}$ e vogliamo determinare un vettore $x \in \mathbb{R}^{n+1}$ tale che

$$\varepsilon^2 = \|b - Ax\|_2^2$$

sia minimo.

Calcoliamo allora i punti estremali di ε^2 risolvendo il sistema

$$\frac{\partial \varepsilon^2}{\partial c_i} = 0$$

cioè

$$\begin{cases} \sum_{j=0}^{n}\sum_{k=0}^{m}\varphi_0(x_k)\varphi_j(x_k)c_j = \sum_{k=0}^{m} y_k\varphi_0(x_k) \\ \sum_{j=0}^{n}\sum_{k=0}^{m}\varphi_1(x_k)\varphi_j(x_k)c_j = \sum_{k=0}^{m} y_k\varphi_1(x_k) \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ \sum_{j=0}^{n}\sum_{k=0}^{m}\varphi_n(x_k)\varphi_j(x_k)c_j = \sum_{k=0}^{m} y_k\varphi_n(x_k) \end{cases}$$

Notiamo che questo sistema ha come matrice dei coefficienti $B = A^T A$ c come termine noto $d = A^T b$.

Dunque la matrice del sistema è simmetrica, e definita positiva se A ha le colonne linearmente indipendenti: in questo caso la soluzione è ovviamente

$$x = (A^T A)^{-1} A^T b$$

ed il residuo $r = b - Ax_0$ di un'approssimazione ottima x_0 è ortogonale allo spazio vettoriale generato dalle colonne di A:

$$A^T r = A^T b - A^T A x_0 = A^T b - A^T A(A^T A)^{-1} A^T b = 0$$

Vale il

Teorema 14.1 *Il problema dei minimi quadrati ha sempre soluzione, e questa è unica se le colonne di A sono linearmente indipendenti.*

Il problema è quindi trovare effettivamente una tale soluzione: ad esempio si potrebbe usare la decomposizione QR di A; supponiamo che le colonne di A siano linearmente indipendenti: al più con $n+1$ trasformazioni di Householder possiamo trasformare A in una matrice

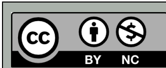
$$A' = Q^T A = \begin{pmatrix} r_{11} & r_{12} & r_{13} & \dots & r_{1n+1} \\ 0 & r_{22} & r_{23} & \dots & r_{2n+1} \\ 0 & 0 & r_{33} & \dots & r_{3+n1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & r_{n+1,n+1} \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

ove la matrice triangolare superiore è non singolare se e solo se A ha le colonne linearmente indipendenti.

Dato che Q è ortogonale, abbiamo

$$\|b - Ax\| = \|Q^T(b - Ax)\| = \|b' - A'x\|$$

che è minimizzata se $Rx = c'$ ove $b' - A'x = \begin{pmatrix} c' - Rx \\ d' \end{pmatrix}$ (essendo R la matrice triangolare superiore di cui sopra) ed assume in tal caso valore minimo $\|d'\|$.


This work is licensed under a *Creative Commons Attribution-Non Commercial 3.0 Unported License.*