Algebra lineare numerica in Matlab

Alvise Sommariva

Università degli Studi di Padova
Dipartimento di Matematica Pura e Applicata

20 maggio 2020

Norma di vettori

Definiamo di seguito alcune norme vettoriali, supponendo $x \in \mathbb{C}^n$.

Posto $p \in [1, +\infty)$, la norma p è definita come

$$\|x\|_p = \left(\sum_{j=1}^{n} |x_j|_{\mathbb{C}}^p \right)^{1/p}$$

Per $p = 1$, si definisce la importante norma 1

$$\|x\|_1 = \sum_{j=1}^{n} |x_j|$$

mentre per $p = 2$, si ottiene la importante norma euclidea

$$\|x\|_2 = \left(\sum_{j=1}^{n} |x_j|^2 \right)^{1/2}.$$

Nel caso $p = \infty$, si definisce la norma del massimo

$$\|x\|_\infty = \max_{j=1,\ldots,n} |x_j|_{\mathbb{C}}.$$

Norma di matrici

Definiamo alcune norme matriciali, supponendo $A \in \mathbb{C}^n$, con la propriet
'a di essere *indotte* da norme di vettori, ovvero tali che

$$\|A\| = \sup_{x \neq 0} \frac{\|Ax\|}{\|x\|}.$$

In questo modo, posto $A = (a_{i,j})$,

- per $p = 1$ si ottiene

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{i,j}|$$

- per $p = 2$,

$$\|A\|_2 = \sqrt{\rho(A^* A)}$$

  dove

$$\rho(B) = \max i = 1, \ldots, n |\lambda_i|$$

  con λ_k autovalore di A

- per $p = \infty$

$$\|A\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{i,j}|$$

Norma di matrici

Tali norme sono implementate in Matlab mediante il comando `norm`. In particolare dall'help di Matlab, si ottiene quanto segue:

```
>> help norm
 norm   Matrix or vector norm.
      norm(X,2) returns the 2-norm of X.
      norm(X) is the same as norm(X,2).
      norm(X,1) returns the 1-norm of X.
      norm(X,Inf) returns the infinity norm of X.
      norm(X,'fro') returns the Frobenius norm of X.
    In addition, for vectors
      norm(V,P) returns the p-norm of V defined as SUM(ABS(V).^P)
           ^(1/P).
      norm(V,Inf) returns the largest element of ABS(V).
 ...
    By convention, NaN is returned if X or V contains NaNs.
    See also cond, rcond, condest, normest, hypot.
...
>>
```

Numero di condizionamento

Una quantità importante é il numero di condizionamento di una matrice invertibile $A \in \mathbb{C}^{n\times n}$, definito da

$$\texttt{cond}(A) := \|A\|\|A^{-1}\|$$

dove $\|\cdot\|$ é una norma matriciale *indotta* da una norma vettoriale.

Quando numero di condizionamento é grande, se invece di risolvere il sistema lineare $Ax = b$ si considera $\hat{A}\hat{x} = \hat{b}$, anche con $\|A - \hat{A}\|$ e $\|b - \hat{b}\|$ *relativamente piccoli*, le rispettive soluzioni x, $\hat{x}$, possono essere tali che $\|x - \hat{x}\|$ possa essere *relativamente grande*.

Quindi a *piccole* perturbazioni sui dati possano corrispondere *grandi* perturbazioni sulle soluzioni.

Numero di condizionamento

Il comando Matlab per poter calcolare tale numero in alcune norme é cond, mentre per aver rapidamente l'ordine di grandezza, che nella maggior parte dei casi é rilevante, si usa condest.

```
>> help cond
 cond   Condition number with respect to inversion.
    cond(X) returns the 2-norm condition number (the ratio of the
    largest singular value of X to the smallest).  Large condition
    numbers indicate a nearly singular matrix.

    cond(X,P) returns the condition number of X in P-norm:

       NORM(X,P) * NORM(INV(X),P).

>>
```

```
>> help condest
 condest 1-norm condition number estimate.
    C = condest(A) computes a lower bound C for the 1-norm condition
    number of a square matrix A.
 ...
>>
```

Numero di condizionamento

La matrice di Hilbert $H = (h_{i,j}) \in \mathbb{R}^{n\times n}$ in cui

$$h_{i,j} = \frac{1}{i+j+1}, \ i,j = 1,\ldots,n$$

è un esempio di matrice malcondizionata.

In Matlab, è richiamabile mediante il comando `hilb`.

```
>> help hilb

 hilb    Hilbert matrix.
    H = hilb(N) is the N-by-N matrix with elements 1/(i+j-1), which
        is a
    famous example of a badly conditioned matrix. The INVHILB
    function calculates the exact inverse.
    ...
    Example:
    hilb(3) is

           1.0000    0.5000    0.3333
           0.5000    0.3333    0.2500
           0.3333    0.2500    0.2000

    ...
>>
```

Numero di condizionamento

Posta H_n la matrice di Hilbert di ordine n studiato il problema $Hx = b$, dove $b_k = 1$ per $k = 1, \ldots, n$. La soluzione é data da $x^* = H^{-1}b$ ed è calcolata esattamente (l'inversa di H è una matrice a coefficienti interi).

Invece la matrice $\hat{H} = H + \delta H$ implementata sul computer, in virtù degli inevitabili errori dovuti al calcolo di $h_{i,j} = \frac{1}{i+j+1}$, per $i, j = 1, \ldots, n$, consiste in una approssimazione di H, e quindi risolveremo $\hat{H}x = b$, la cui soluzione è $\tilde{x} = x^* + \delta x$, in cui il vettore δx corrisponde all'errore compiuto.

Ci si domanda quanto *pericoloso* sia questo fatto, tenendo conto che dalla teoria sappiamo che

$$\boxed{\frac{\|\delta x\|}{\|x^* + \delta x\|} \leq k(H)\frac{\|\delta H\|}{\|H\|}.}$$

La preoccupazione è che seppure $\frac{\|\delta H\|}{\|H\|}$ è molto piccolo, per le matrici di Hilbert il numero di condizionamento $k(H)$ è molto grande e quindi potenzialmente potrebbe risultare rilevante pure $\frac{\|\delta x\|}{\|x^* + \delta x\|}$.

Numero di condizionamento

Implementiamo l'esperimento nel seguente codice

```
function esempio_hilbert

for n=2:10

    H=hilb(n); % matrice di Hilbert
    Hinv=invhilb(n); % inversa di H
    b=ones(n,1); % termine noto
    solth=Hinv*b; % soluzione
    solnum=H\b; % soluzione numerica

    relerr=norm(solnum-solth,2)/norm(solth,2); % errore relativo
    relerrest=norm(solnum-solth,2)/norm(solnum,2); % errore stima
    fprintf('\n \t Ordine: %3.0f RE: %1.1e RE-stima: %1.1e cond:
        %1.1e',...
        n,relerr,relerrest,condest(H));

end

fprintf('\n');
```

Ricordiamo che la riga `solnum` $= H\backslash b$ calcola numericamente la soluzione `solnum` del sistema lineare $Hx = b$.

Numero di condizionamento

Lanciando il codice, otteniamo:

```
>> esempio_hilbert
   Ordine:    2 RE: 1.4e-16 RE-stima: 1.4e-16 cond: 2.7e+01
   Ordine:    3 RE: 7.4e-16 RE-stima: 7.4e-16 cond: 7.5e+02
   Ordine:    4 RE: 7.1e-14 RE-stima: 7.1e-14 cond: 2.8e+04
   Ordine:    5 RE: 1.2e-12 RE-stima: 1.2e-12 cond: 9.4e+05
   Ordine:    6 RE: 6.6e-11 RE-stima: 6.6e-11 cond: 2.9e+07
   Ordine:    7 RE: 2.7e-09 RE-stima: 2.7e-09 cond: 9.9e+08
   Ordine:    8 RE: 2.0e-08 RE-stima: 2.0e-08 cond: 3.4e+10
   Ordine:    9 RE: 1.4e-06 RE-stima: 1.4e-06 cond: 1.1e+12
   Ordine:   10 RE: 5.9e-05 RE-stima: 5.9e-05 cond: 3.5e+13
>>
```

In particolare:

- `RE` rappresenta $\frac{\|\delta x\|_2}{\|x^*\|_2}$;
- `RE-stima` rappresenta $\frac{\|\delta x\|_2}{\|x^*+\delta x\|_2}$, ovvero il primo membro della stima.

Di conseguenza, già a gradi bassi, a piccole perturbazioni δH, corrispondono purtroppo grossi errori relativi nel determinare accuratamente la soluzione, come detto dalla stima teorica.

Numero di condizionamento

Ci chiediamo cosa succeda se invece di $b = [1, \ldots, 1]$, cosa succeda nel risolvere il problema $(H + \delta H)(x^* + \delta x) = b + \delta b$.

Per farlo introduciamo una perturbazione δb data da numeri random, dell'ordine di 10^{-8}.

```
function esempio_hilbert_pert
for n=2:10
    H=hilb(n); % matrice di Hilbert
    Hinv=invhilb(n); % inversa di H
    b=ones(n,1); % termine noto
    bpert=b+10^(-12)*rand(size(b)); % term.noto perturbato
    solth=Hinv*b; % soluzione
    solnum=H\bpert; % soluzione numerica
    relerr=norm(solnum-solth,2)/norm(solth,2); % errore relativo
    fprintf('\n \t Ordine: %3.0f RE: %1.1e cond: %1.1e',...
        n,relerr,condest(H));
end
fprintf('\n');
```

La quantitá RE rappresenta $\frac{\|\delta x\|_2}{\|x^*\|_2}$.

Numero di condizionamento

Lanciato il codice, otteniamo

```
>> esempio_hilbert_pert
   Ordine:    2 RE: 1.2e-08 cond: 2.7e+01
   Ordine:    3 RE: 1.1e-09 cond: 7.5e+02
   Ordine:    4 RE: 2.6e-08 cond: 2.8e+04
   Ordine:    5 RE: 6.7e-07 cond: 9.4e+05
   Ordine:    6 RE: 3.5e-06 cond: 2.9e+07
   Ordine:    7 RE: 3.1e-06 cond: 9.9e+08
   Ordine:    8 RE: 9.5e-05 cond: 3.4e+10
   Ordine:    9 RE: 3.6e-04 cond: 1.1e+12
   Ordine:   10 RE: 8.2e-04 cond: 3.5e+13
>>
```

e quindi la perturbazione sul termine noto ha peggiorato ulteriormente la situazione (come facilmente immaginabile).

Soluzione di sistemi lineari con backslash

L'ambiente Matlab utilizza varie strategie per risolvere i sistemi lineari. A tal proposito, digitando "help \" ricaviamo

```
>> help \
 \    Backslash or left matrix divide.
    A\B is the matrix division of A into B, which is roughly the
    same as INV(A)*B , except it is computed in a different way.
    If A is an N-by-N matrix and B is a column vector with N
    components, or a matrix with several such columns, then
    X = A\B is the solution to the equation A*X = B. A warning
    message is printed if A is badly scaled or nearly singular.
    A\EYE(SIZE(A)) produces the inverse of A.
    ...
    See also ldivide, rdivide, mrdivide.

    Reference page for mldivide
    Other functions named mldivide
>>
```

In pratica tale comando serve per poter risolvere sistemi lineari del tipo $Ax = b$.

Soluzione di sistemi lineari con backslash

Vediamo un esempio.

```
>> A=[1 3 5; 2 4 5; 1 1 1];
>> % verifichiamo che det(A) non e' nullo
>> % (ovvero la matrice A e' invertibile)
>> det(A)
ans =
    -2
>> % definiamo il termine noto "b"
>> b=[1 1 1]';
>> % calcoliamo la soluzione di A*x=b;
>> % attenzione che e' "\" e non "/".
>> x=A\b
x =
     2
    -2
     1
>> A*x
ans =
     1
     1
     1
>> % Quindi visto che "b" e' il vettore colonna [1 1 1]'
>> % abbiamo che "x" e' la soluzione richiesta.
```

Fattorizzazione LU

Il comando `lu` calcola la corrispettiva fattorizzazione di una matrice A con n righe e colonne, ottenuta mediante eliminazione di Gauss con pivoting.

Il comando è

```
[L,U,P]=lu(A)
```

e determina le matrici

- $L = (l_{i,j}) \in \mathbb{R}^{n\times n}$ triangolare inferiore, ovvero tale che $l_{i,j} = 0$ se $j < i$ con $l_{i,i} = 1$,
- $U = (u_{i,j}) \in \mathbb{R}^{n\times n}$ triangolare superiore, ovvero tale che $u_{i,j} = 0$ se $j > i$,
- $P = (p_{i,j}) \in \mathbb{R}^{n\times n}$ di permutazione, ovvero con esclusivamente un valore non nullo e pari a 1 per ogni riga e colonna,

cosicchè $PA = LU$.

Fattorizzazione LU

Vediamo un esempio.

```
>> A=[4 -2 -1 0; -2 4 1 0.5; -1 1 4 1; 0 0.5 1 4];
>> [L,U,P]=lu(A);
>> % L triang. inf. con elementi diagonali uguali a 1
>> L
L =
    1.0000         0         0         0
   -0.5000    1.0000         0         0
   -0.2500    0.1667    1.0000         0
         0    0.1667    0.2500    1.0000
>> % L triangolare sup.
>> U
U =
    4.0000   -2.0000   -1.0000         0
         0    3.0000    0.5000    0.5000
         0         0    3.6667    0.9167
         0         0         0    3.6875
>> % P di permutazione
>> P
P =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
>>
```

Soluzione di sistemi lineari con fattorizzazione LU

Supponiamo sia

- $A \in \mathbb{R}^{n\times n}$, con $\det(A) \neq 0$,
- $b \in \mathbb{R}^n$,

In tali ipotesi, esiste un unico $x^* \in \mathbb{R}^n$ che risolva il sistema lineare $Ax = b$.
Sotto queste ipotesi si può provare che se $PA = LU$, allora necessariamente

- $\det(P) = \pm 1$, $P^{-1} = P^T$,
- $\det(L) = 1$
- $\det(U) \neq 0$.

Quindi, posto $Pb = c$, abbiamo

$$Ax = b \Leftrightarrow PAx = Pb \Leftrightarrow LUx = Pb = c$$

Di conseguenza

- posto $y = Ux$, y è soluzione del sistema triangolare inferiore $Ly = c$;
- calcolato y, la soluzione x del sistema $Ax = b$ è pure soluzione del sistema triangolare superiore $Ux = y$;

I due sistemi lineari

$$Ly = c, \quad Ux = y$$

possono essere convenientemente essere risolti rispettivamente mediante sostituzione in avanti e all'indietro in $O(n^2)$ operazioni moltiplicative.

Soluzione di sistemi lineari con metodo_LU

In generale per risolvere il sistema lineare, utilizzando backslash solo per risolvere convenientemente i sistemi triangolari, possiamo scrivere la seguente funzione fattorizzazione_LU per risolvere i sistemi lineari.

```
function x=metodo_LU(A,b)
[L,U,P]=lu(A); % fattorizzazione PA=LU
c=P*b;
y=L\c; % sistema triangolare inferiore
x=U\y; % sistema triangolare superiore
```

Vogliamo paragonarlo con il comando di backslash su alcune matrici di speciale interesse introdotte nella gallery di Matlab.

```
>> help gallery
  gallery Higham test matrices.
   [out1,out2,...] = gallery(matname, param1, param2, ...)
   takes matname, a string that is the name of a matrix family, and
   the family's input parameters.
     ...
   chebvand    Vandermonde-like matrix for the Chebyshev polynomials.
     ...
   minij       Symmetric positive definite matrix MIN(i,j).
   moler       Moler matrix -- symmetric positive definite.
     ...
   poisson     Block tridiagonal matrix from Poisson's equation ...
     ...
   tridiag     Tridiagonal matrix (sparse).
     ...
>>
```

Soluzione di sistemi lineari con metodo_LU

Implementiamo il seguente confronto tra metodo_LU e backslash.

```
function test_metodo_LU

warning off; % non scrive "warnings"
for n=5:5:20
    A=gallery('chebvand',n);b=rand(n,1); condA=condest(A);
    % risoluzione con "metodo_LU"
    tic; x_LU=metodo_LU(A,b);  t_metodo_LU=toc; % tempo impiegato
    % risoluzione con "backslash"
    tic; x_backslash=A\b; t_backslash=toc; % tempo impiegato
    % errore norma 2 soluzione
    err=norm(x_LU-x_backslash)/norm(x_backslash);
    % errore relativo soluzione backslash
    residuo_backslash=norm(b-A*x_backslash)/norm(b);
    % errore fattorizzazione LU
    [L,U,P]=lu(A); errLU=norm(P*A-L*U);
    fprintf('\n n: %3.0f cond: %1.3e err LU: %1.3e',n,condA,errLU);
    fprintf('\n err. LU vs backs.: %1.3e res. relativo: %1.3e',...
        err,residuo_backslash);
    fprintf('\n  tempo impiegato: LU: %1.3e backslash: %1.3e \n',...
        t_metodo_LU,t_backslash);
end
fprintf('\n')
```

Soluzione di sistemi lineari con `metodo_LU`

Il codice

- mediante la gallery di Matlab, definisce una particolare matrice di Vandermonde di ordine `n` e la assegna ad `A` e di seguito un vettore `b` random, calcolando infine un approssimazione del numero di condizionamento `condA` della matrice `A`;
- calcola un approssimazione `x_LU` soluzione del sistema $Ax = b$ con la routine `metodo_LU` e "stima" il tempo impiegato per determinare l'approssimazione;
- calcola un approssimazione `x_backslash` soluzione del sistema $Ax = b$ con il comando `backslash` e "stima" il tempo impiegato per determinare l'approssimazione;
- valuta l'errore relativo `err` in norma 2, tra le due approssimazioni;
- valuta il *residuo relativo*

$$\texttt{residuo\_backslash} = \frac{\|b - A * \texttt{x\_backslash}\|_2}{\|b\|_2}$$

  che è un indicatore di quanto `x_backslash` è soluzione del sistema lineare;
- calcola quanto accurata è la fattorizzazione LU mediante `errLU=norm(P*A-L*U);`
- stampa alcuni risultati di rilievo.

Soluzione di sistemi lineari con metodo_LU

Otteniamo quali risultati

```
>> test_metodo_LU

 n:    5 cond: 1.493e+03 err LU: 3.165e-16
 err. LU vs backs.: 0.000e+00 res. relativo: 4.509e-15
  tempo impiegato: LU: 1.709e-04 backslash: 7.191e-05

 n:   10 cond: 4.846e+07 err LU: 7.295e-16
 err. LU vs backs.: 0.000e+00 res. relativo: 1.886e-10
  tempo impiegato: LU: 1.999e-04 backslash: 7.516e-05

 n:   15 cond: 1.586e+12 err LU: 1.303e-15
 err. LU vs backs.: 0.000e+00 res. relativo: 5.375e-08
  tempo impiegato: LU: 1.512e-04 backslash: 6.478e-05

 n:   20 cond: 5.301e+16 err LU: 1.590e-15
 err. LU vs backs.: 0.000e+00 res. relativo: 1.486e-02
  tempo impiegato: LU: 2.225e-04 backslash: 2.802e-04

>>
```

che mostrano che

- i tempi di calcolo per risolvere problemi di piccola dimensione siano meno di millesimi di secondo;
- la routine `metodo_LU` e `backslash` calcolano la stessa approssimazione(!);
- purtroppo il condizionamento delle matrici non permette di calcolare accuratamente la soluzione (nonostante lo sia la fattorizzazione `PA=LU`.

Soluzione di sistemi lineari con metodo_LU

Partendo dalla demo precedente definiamo la routine test_metodo_LU2 per cui:

- utilizziamo il ciclo for: `for n=200:200:1000` invece di `for n=5:5:20`;
- sostituiamo `A=gallery('minij',n);` a `A=gallery('chebvand',n);`

La gallery relativa a minij definisce la matrice $A = (a_{i,j})$ simmetrica e definita positiva t.c. $a_{i,j} = \min(i, j)$. Lanciando tale routine, ricaviamo:

```
>> test_metodo_LU2

 n: 200 cond: 8.040e+04 err LU: 0.000e+00
 err. LU vs backs.: 5.819e-16 res. relativo: 3.890e-14
  tempo impiegato: LU: 3.922e-03 backslash: 1.944e-03

 n: 400 cond: 3.208e+05 err LU: 0.000e+00
 err. LU vs backs.: 3.851e-16 res. relativo: 1.313e-13
  tempo impiegato: LU: 7.169e-03 backslash: 1.860e-03

 n: 600 cond: 7.212e+05 err LU: 0.000e+00
 err. LU vs backs.: 5.199e-16 res. relativo: 3.332e-13
  tempo impiegato: LU: 1.275e-02 backslash: 4.595e-03

 n: 800 cond: 1.282e+06 err LU: 0.000e+00
 err. LU vs backs.: 5.058e-16 res. relativo: 5.925e-13
  tempo impiegato: LU: 2.992e-02 backslash: 1.200e-02

 n: 1000 cond: 2.002e+06 err LU: 0.000e+00
 err. LU vs backs.: 7.805e-16 res. relativo: 7.416e-13
  tempo impiegato: LU: 5.314e-02 backslash: 2.064e-02

>>
```

Soluzione di sistemi lineari con `metodo_LU`

Dall'esperimento numerico su `minij` si vede che:

- Le matrici questa volta non sono *piccole*, ma comunque il tempo di calcolo e' ancora nell'ordine dei centesimi/millesimi di secondo;
- I condizionamenti delle matrici non sono piccoli, ma molto inferiori a quelli visti nell'esempio precedente e i risultati ottenuti sono relativamente accurati.
- Le fattorizzazioni LU sono molto accurate.
- Il metodo LU e il `backslash` di Matlab non forniscono gli stessi risultati ma sono comunque entrambi *simili*, peró questa volta il `backslash` sembra più rapido.
- Nonostante le matrici abbiano comunque condizionamenti rilevanti, le soluzioni sono calcolate *relativamente bene*.

Esercizi. La routine `fattorizzazione_LU`

Esercizio (1)

Il seguente pseudocodice

function [L,A]=fattorizzazione_LU(A)

for *k=1,...,n-1* **do**
  $l_{k,k} = 1$ **for** *i=k+1,...,n* **do**
    $l_{i,k} = \frac{a_{i,k}}{a_{k,k}}$
    for *j=k,...,n* **do**
      $a_{i,j} = a_{i,j} - l_{i,k} a_{k,j}$
    end
  end
end
$l_{n,n} = 1$

- *calcola la fattorizzazione LU di una matrice* $A = (a_{i,j})$ *data in input,*
- *in output offre le matrici triangolari* $L = (l_{i,j})$ *e* $U = (u_{i,j})$ *che corrisponde alla matrice A alla fine del processo.*

Lo si implementi in Matlab mediante la function `fattorizzazione_LU` *ricordando che "n" è il numero di righe e colonne di A.*

Esercizio (2)

Si scriva uno script `demo_eliminazione_gaussiana` *che definita la matrice*

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

e il termine noto

$$b = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix}$$

calcoli la soluzione del sistema lineare $Ax = b$, risolvendo i due sistemi triangolari citati in precedenza (con P uguale alla matrice identica), mediante il comando $\backslash$ di Matlab.
La soluzione corretta è

$$x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Metodi iterativi

Si supponga di dover risolvere il sistema lineare $Ax = b$ con $A \in \mathbb{R}^{n\times n}$, $\det(A) \neq 0$, $b \in \mathbb{R}^n$.

Per una matrice A in generale il metodo di eliminazione gaussiana richiede $O(n^3/3)$ operazioni, calcolando la soluzione esatta.

Qualora

- si sia interessati a una approssimazione della soluzione esatta, ad esempio con un certo numero di cifre decimali esatte, e magari n sia molto grande,
- e/o A abbia molte componenti nulle,

tipicamente si utilizzano metodi iterativi, che spesso raggiungono questo risultato con complessità dell'ordine $O(n^2)$.

Metodi iterativi: Jacobi

Un primo esempio di metodo iterativo è quello di Jacobi, che calcola una sequenza di vettori $x^{(k)}$ che in certe ipotesi si dimostra convergono alla soluzione x^*.

In dettaglio, se $A = (a_{i,j})$ e $a_{k,k} \neq 0$ per $k = 1, \ldots, n$,

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)})/a_{ii}, \qquad (1)$$

con $i = 1, \ldots, n$.

Metodi iterativi: Gauss-Seidel

Un altro metodo iterativo, è quello di Gauss-Seidel, che calcola una sequenza di vettori $x^{(k)}$ che in certe ipotesi si dimostra convergono alla soluzione x^*.

In dettaglio, se $A = (a_{i,j})$ e $a_{k,k} \neq 0$ per $k = 1, \ldots, n$,

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}, \qquad (2)$$

con $i = 1, \ldots, n$.

Metodi iterativi

Risulta possibile riscrivere questi metodi in forma matriciale.
Sia

- $A \in \mathbb{R}^{n\times n}$ una matrice quadrata,
- $A = P - N$ un cosidetto splitting della matrice A, con $\det(P) \neq 0$

Allora consideriamo metodi le cui iterazioni siano fornite dalle iterazioni successive

$$x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b. \tag{3}$$

Sia $A = D - E - F$ con

1. D matrice diagonale,
2. E triangolare inferiore,
3. F triangolare superiore.

Allora

- il metodo di Jacobi corrisponde a scegliere, se D è invertibile,

$$P = D, \qquad N = E + F;$$

- il metodo di Gauss-Seidel corrisponde a scegliere, se D è invertibile,

$$P = D - E, \qquad N = F.$$

Implementazione di alcuni metodi iterativi

Implementiamo in Matlab tali routines, con la caratterizzazione matriciale.

```
function [x,errs,iter,flag]=metodo_jacobi(A,x,b,maxit,tol)

% Oggetto:
% risoluzione del sistema lineare Ax=b con il metodo di Jacobi
%
% Input:
% A: matrice quadrata non singolare
% x: approssimazione iniziale della soluzione (vettore colonna)
% b: termine noto (vettore colonna)
% maxit: numero massimo di iterazioni
% tol: tolleranza del metodo di Jacobi
%
% Output:
% x: approssimazione della soluzione, fornita dal metodo di Jacobi.
% errs: norme dell'errore
%                   norm(x_new-x_old)/norm(x_new)
%     al variare delle iterazioni, ovvero lo step relativo.
% iter: numero di iterazioni del metodo
% flag: 0: si e' raggiunta l'approssimazione della soluzione
%       1: non si e' raggiunta l'approssimazione della soluzione
```

Implementazione di alcuni metodi iterativi

```
% inizializzazioni
flag=0;
% --- le matrici P, N per il metodo di Jacobi ---
P=diag(diag(A));
% se "det(P)=0", allora il metodo di Jacobi non e' applicabile in
% quanto "P" non invertibile
if det(P) == 0
    errs=[]; iter=0; flag=1;  return;
end
N=diag(diag(A))-A;
% --- iterazioni del metodo di Jacobi ---
for iter=1:maxit
    x_old=x;
    x=P\(N*x_old+b);   % nuova iterazione
    % calcolo step relativo
    errs(iter)=norm(x-x_old)/norm(x);
    % se error e' suff. piccolo si esce dalla routine
    if (errs(iter)<=tol), return, end
end
% se abbiamo raggiunto questo punto abbiamo fatto
%  troppe iterazioni senza successo e quindi
% poniamo "flag=1".
flag=1;
```

Commento a `metodo_jacobi`

Cominciamo discutendo il codice `metodo_jacobi`.

- Dapprima inizializziamo le variabili. In particolare se si esce correttamente `flag=0`, come da inizializzazione, e si assegnano le altre variabili di output.
- Di seguito, come suggerito dalla versione matriciale, definiamo M, N, facendo attenzione al caso in cui M abbia qualche qualche componente diagonale nulla, ovvero non sia singolare. In tal caso assegniamo `flag=1` e usciamo dalla routine per `return`.
- Entriamo nel ciclo for che fa iterazioni fino a quando il loro indice `iter` non supera `maxit`.
- Assegna a `x_old` il valore della vecchia iterazione e calcola la nuova iterazione mediante il comando Matlab "\". Si osservi che la matrice M è diagonale e quindi, una volta calcolato `N*x_old+b`, Matlab risolve il sistema in $O(n)$ divisioni (pensarci su).

Commento a `metodo_jacobi`

- Nella riga successiva, alla k-sima iterazione valuta la quantità

$$\frac{\|x - x_{old}\|_2}{\|x\|_2}$$

dove al solito se $u = (u_1, \ldots, u_n)$ allora

$$\|u\|_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$$

e assegna tale valore a `errs(iter)`.

- Se la quantità `errs(iter)` è minore o uguale della tolleranza `tol` esce dalla routine, altrimenti prosegue a iterare il ciclo for.
- Se esce dal ciclo for dopo `maxit` iterazioni, pone `flag=1` perchè nel numero di iterazioni richieste non ha trovato la soluzione.

La routine demo_jacobi

Per testare il metodo salviamo la seguente demo_jacobi.

```
function demo_jacobi

% demo del metodo di Jacobi, per la risolvere sistemi lineari Ax=b.

% problema A*x=b
A=gallery('poisson',10); % matrice
sol=ones(size(A,1),1); % soluzione
b=A*sol; % termine noto
% preferenze
maxit=100000; % numero massimo di iterazioni
tol=10^(-6); % tolleranza
x0=zeros(size(b)); % vettore iniziale
% soluzione col metodo di Jacobi
[x,errs,iter,flag]=metodo_jacobi(A,x0,b,maxit,tol);
% norma 2 residuo relativo
err=norm(b-A*x,2)/norm(b);
% statistiche
fprintf('\n \t dimensione matrice: %6.0f', size(A,1));
fprintf('\n \t numero iterazioni : %6.0f', iter);
fprintf('\n \t flag              : %6.0f', flag);
fprintf('\n \t residuo relativo  : %1.2e \n \n', err);
```

Commento a demo_jacobi

La demo non è troppo complicata, ma ha alcuni aspetti interessanti da discutere.

- La prima riga è

  ```
  A=gallery('poisson',10);
  ```

  calcola la matrice di Poisson P_{10} che è una matrice con 100 righe e colonne. Digitando `help gallery` nella command-window, si capisce che tale gallery permette di definire molte matrici di vario interesse matematico.
- Si impone che la soluzione `sol` sia un vettore colonna che abbia lo stesso numero di righe dell'ordine della matrice, e tutte le componenti uguali a 1.
- Se `sol` è la soluzione del sistema lineare $Ax = b$, necessariamente $b = A * \text{ttsol}$.
- Quale vettore iniziale, si utilizza il vettore colonna `x`, che abbia la stessa dimensione di `b`, ma con componenti tutte nulle.
- Il metodo di Jacobi fornisce l'approssimazione `x` della soluzione `sol`.
- Di seguito si calcola la norma 2 del residuo dovuto a `x` (relativo rispetto al termine noto `b`), ovvero

$$\frac{\|b - A * \texttt{sol}\|_2}{\|b\|_2}.$$

- Alla fine si salvano alcune statistiche.

Commento a demo_jacobi

Alcuni teoremi asseriscono che il metodo di Jacobi, genera una sequenza di vettori $\{x^{(k)}\}$ che converge alla soluzione esatta, nel nostro esempio `sol`.
In effetti, da command-window otteniamo

```
>> demo_jacobi

   dimensione matrice:     100
   numero iterazioni :     255
   flag              :       0
   residuo relativo  : 5.42e-06

>>
```

Nota.

Si noti che la statistica della demo è il residuo relativo e non lo step relativo proprio delle iterazioni di

`metodo_jacobi`

Per questo, pure essendo lo step relativo inferiore alla tolleranza, nel nostro caso 10^{-6}*, il residuo relativo è maggiore (ovvero* $5.42 \cdot 10^{-6}$*).*

Le routines `metodo_GS` e `demo_GS`

Esercizio (Metodo di Gauss-Seidel)

- *Utilizzando i comandi Matlab* `tril` *e* `triu`*, si possono calcolare le matrici P ed N del metodo di Gauss-Seidel.*
- *Modificare di conseguenza le routines* `metodo_jacobi` *e* `demo_jacobi`*, così da definire* `metodo_GS` *e* `demo_GS`*.*
- *"Lanciare" l'esperimento visto col metodo di Jacobi, nel caso del metodo di Gauss-Seidel. Converge? In quante iterazioni? Sono pari al doppio o circa la meta'?*