1 Risoluzione di sistemi lineari: metodi diretti - pt. 2

1.1 Fattorizzazione LU, di Cholesky e MEG

Esempio 1.1 (Fattorizzazione LU). *Sia $A \in \mathbb{R}^{n\times n}$ non singolare. Supponiamo che esistano due opportune matrici L ed U, triangolare inferiore e superiore, rispettivamente, tali che*

$$A = LU. \tag{1}$$

In forma estesa

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix} \tag{2}$$

*La (1) è detta **fattorizzazione** (o **decomposizione**) **LU** di A. Siccome abbiamo assunto che A sia non singolare, tali matrici devono essere anch'esse non singolari (in particolare ciò assicura che i loro elementi diagonali siano non nulli). In tal caso, risolvere $A\mathbf{x} = \mathbf{b}$ conduce alla risoluzione dei due seguenti sistemi triangolari*

$$L\boldsymbol{y} = \boldsymbol{b}, \quad U\mathbf{x} = \mathbf{y}. \tag{3}$$

I sistemi triangolari sono a loro volta facilmente risolvibili, tramite i metodi delle sostituzioni in avanti e all'indietro, come visto nella lezione precedente.

Supponiamo che esista una fattorizzazione LU di A. Applicando la legge del prodotto matriciale riga per colonna a (2), le entrate di L ed U soddisfano

$$\sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} = a_{ij} \quad i,j = 1,\dots,n. \tag{4}$$

Il sistema (4) ha n^2 equazioni (tanto quanto gli elementi a_{ij} $i,j = 1,\dots,n$) ed n^2+n incognite l_{ik} ed u_{kj}. Questo è un sistema *sottodeterminato* (i.e. ha più incognite che equazioni), di conseguenza la soluzione **non** è **unica**: possono esistere diverse matrici L ed U che soddisfano (1) (qualora esistessero). La procedura classica per eliminare l'indeterminazione è di imporre che gli elementi diagonali della matrice triangolare inferiore L siano tutti pari a 1: $l_{ii} = 1, \text{per } i = 1,\dots,n$. Abbiamo così tolto n incognite rendendo (4) un sistema quadrato *determinato* di n^2 equazioni in n^2 incognite . Questo può essere risolto con l'**algoritmo di Gauss**.

Esempio 1.2 (Algoritmo di Gauss). *Data $A \in \mathbb{R}^{n\times n}$, posto $A^{(1)} = A$, ovvero $a_{ij}^{(1)} = a_{ij}$ per*

$i, j = 1, \ldots, n$, si calcoli

$$
\begin{aligned}
&\textit{per } k = 1, \ldots, n-1 \\
&\quad \textit{per } i = k+1, \ldots, n \\
&\qquad l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \\
&\qquad \textit{per } j = k+1, \ldots, n, \\
&\qquad\quad a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)} \\
&\qquad \textit{end} \\
&\quad \textit{end} \\
&\textit{end}
\end{aligned}
\tag{5}
$$

*Gli elementi $a_{kk}^{(k)}$ devono essere tutti **diversi** da **zero** e sono detti **elementi pivot**. Si può facilmente dimostrare che il costo computazionale dell'algoritmo di Gauss è dell'ordine $\mathcal{O}(\frac{2}{3}n^3)$ flops.*

Prima di costruire il codice per l'algoritmo, ragioniamo su quale potrebbe essere il modo *migliore* per implementarlo in termini di **risparmio in memoria**.

- ⋆ Innanzitutto, conviene usare **una sola matrice**, inizialmente posta uguale ad A e modificata ad ogni passo $k \geq 2$ con i nuovi elementi $a_{ij}^{(k)}$, per $i, j \geq k+1$ e con i moltiplicatori l_{ik} per $i \leq k+1$.
- ⋆ Infatti, poiché al k-esimo passo gli elementi sottodiagonali della k-esima colonna non influenzano la matrice finale U, essi possono essere rimpiazzati dagli elementi della k-esima colonna di L (i moltiplicatori).
- ⋆ Infine, non è necessario memorizzare gli elementi diagonali l_{ii}, essendo essi uguali a 1.

In base a queste osservazioni, al k-esimo passo del processo gli elementi memorizzati al posto dei coefficienti originali della matrice A saranno

$$
\begin{bmatrix}
a_{11}^{(1)} & a_{12}^{(1)} & \ldots & \ldots\ldots & a_{1n}^{(1)} \\
l_{21} & a_{22}^{(2)} & \ldots & \ldots\ldots & a_{2n}^{(2)} \\
\vdots & \ddots & \ddots & & \vdots \\
l_{k1} & \ldots & l_{k,k-1} & a_{kk}^{(k)} \ldots & a_{kn}^{(k)} \\
\vdots & \vdots & & \vdots & \vdots \\
l_{n1} & \ldots & l_{n,k-1} & a_{nk}^{(k)} \ldots & a_{nn}^{(k)}
\end{bmatrix}
\tag{6}
$$

```
function A=lugauss(A)
 % A=lugauss(A)
 % Calcola la fattorizzazione LU della matrice A
 % senza pivoting, memorizzando nella
 % parte triangolare inferiore stretta di A la
```

```
% matrice L (gli elementi diagonali di L sono tutti
% uguali a 1 e non serve memorizzarli) ed in
% quella superiore il fattore U

%verifichiamo se la matrice A e' quadrata
[n,m]=size(A);
if n ~= m;
 error('A non e'' una matrice quadrata');
else
 for k= 1:n-1
     %verifichiamo se ci sono pivot che si annullano
     if A(k,k) == 0;
        error('Un elemento pivot si e'' annullato');
     end

     %calcoliamo i moltiplicatori
     for i = k+1:n
     A(i,k) = A(i,k)/A(k,k);

     %aggiorniamo il resto della matrice
     j = [k+1:n];
     A(i,j) = A(i,j) - A(i,k)*A(k,j);
   end
 end
end
```

Dopo l'esecuzione del programma, viene restituita la matrice A tale che

1. la matrice L (privata della diagonale che sappiamo essere costituita da elementi tutti uguali a 1) è memorizzata nella parte triangolare inferiore di A,
2. la matrice U (inclusa la diagonale) è memorizzata nella parte superiore di A.

I due fattori L ed U possono essere ricostruiti tramite i comandi

```
L = eye(n) + tril(A,-1)
U = triu(A)
```

MATLAB calcola la fattorizzazione LU di una matrice tramite il comando `lu`, secondo la sintassi `[L U] = lu(A)`.

Esercizio 1.1. *Per verificare la correttezza dell'algoritmo che abbiamo costruito tramite la funzione* `lugauss`*, si calcoli la fattorizzazione LU della matrice* `A=[1 0 1; 1 3 2; 1 -3 -8]` *prima implementando la funzione* `lugauss`*, successivamente richiamando la funzione built-in di MATLAB* `lu`*.*

Soluzione:

```
A=[1 0 1; 1 3 2; 1 -3 -8]
Anuova=lugauss(A)
L = eye(3) + tril(Anuova,-1)
U = triu(Anuova)
[Lmatlab, Umatlab]=lu(A)
isequal(L, Lmatlab)
isequal(U, Umatlab)
```

Esempio 1.3. *Per risolvere un sistema lineare con A e $\mathbf{b}$ dati tramite la fattorizzazione LU si può procedere secondo*

```
[L U] = lu(A)
y = L\b;
x = U\y
```

Fare alcune prove e confrontare i risultati con il richiamo diretto del comando backshlash.

La fattorizzazione di Gauss potrebbe non esistere anche se la matrice A è non singolare. In generale, la fattorizzazione di Gauss di una matrice $A \in \mathbb{R}^{n\times n}$ esiste ed è unica se e solo se le sottomatrici principali A_i di A di ordine $i = 1, \ldots, n-1$ (cioè quelle ottenute limitando A alle sole prime i righe e colonne) sono non singolari. Questo si verifica in particolare

1. per le matrici **a dominanza diagonale stretta per righe**

$$a_{ii} > \sum_{j=1 j\neq i}^{n} |a_{ij}|, \quad i = 1, \ldots, n, \tag{7}$$

   oppure **per colonne**

$$a_{ii} > \sum_{j=1 j\neq i}^{n} |a_{ji}|, \quad i = 1, \ldots, n; \tag{8}$$

2. per le matrici **reali simmetriche e definite positive**, i.e. $\mathbf{x}^T A\mathbf{x} > 0$, per ogni $\mathbf{x} \in \mathbb{R}^n, \mathbf{x} \neq \mathbf{0}$;

3. per le matrici complesse $A \in \mathbb{C}^{n\times s}$ e **definite positive**, i.e. $\mathbf{x}^H A\mathbf{x} > 0$, per ogni $\mathbf{x} \in \mathbb{C}^n, \mathbf{x} \neq \mathbf{0}$.

Esempio 1.4 (Metodo di Eliminazione di Gauss (MEG))**.** *L'algoritmo* (5) *può essere esteso al termine noto* $\mathbf{b}$ *del sistema lineare* $A\mathbf{x} = \mathbf{b}$. *L'algoritmo derivante viene chiamato* ***metodo di eliminazione di Gauss (MEG)****. Le istruzioni sono: data* $A \in \mathbb{R}^{n\times n}$ *e* $\mathbf{b} \in \mathbb{R}^n$, *posto*

$A^{(1)} = A$, ovvero $a_{ij}^{(1)} = a_{ij}$ per $i, j = 1, \ldots, n$; posto $\mathbf{b}^{(1)} = \mathbf{b}$, ovvero $b_i^{(1)} = b_i$ per $i = \ldots, n$, si calcoli

$$
\begin{array}{l}
\textit{per } k = 1, \ldots, n-1 \\
\quad \textit{per } i = k+1, \ldots, n \\
\qquad l_{ik} = \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \\
\qquad \textit{per } j = k+1, \ldots, n, \\
\qquad\quad a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)} \\
\qquad \textit{end} \\
\qquad b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)} \\
\quad \textit{end} \\
\textit{end}
\end{array}
\tag{9}
$$

Alla fine del processo, il sistema triangolare superiore ottenuto $A^{(n)}\mathbf{x} = \mathbf{b}^{(n)}$ è equivalente a quello originale (ovvero ha la stessa soluzione). Questo può essere poi facilmente risolto chiamando l'algoritmo di sostituzioni all'indetro che abbiamo definito in precedenza con la funzione `backward`.

Esempio 1.5 (Fattorizzazione di Cholesky)**.** *Se $A \in \mathbb{R}^{n\times n}$ è **simmetrica** e **definita positiva**, esiste ed è unica la sua **fattorizzazione di Cholesky***

$$
A = R^T R \tag{10}
$$

*laddove R è una matrice **triangolare superiore** con elementi positivi sulla diagonale. Gli elementi di R possono essere calcolati: posto $r_{11} = \sqrt{a_{11}}$, per $j = 2, \ldots, n$*

$$
\begin{aligned}
r_{ij} &= \frac{1}{r_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki} r_{kj} \right) \quad i = 1, \ldots, j-1 \\
r_{jj} &= \sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{kj}^2}.
\end{aligned}
\tag{11}
$$

Si può facilmente dimostrare che il calcolo di R richiede circa $n^3/3$ operazioni (la metà di quelle richieste per calcolare le due matrici della fattorizzazione LU). Nel caso in cui sia $A \in \mathbb{C}^{n\times n}$ sia definita positiva in $\mathbb{C}^n$, la formula (10) diventa $A = R^H R$, dove R^H è la trasposta coniugata di R.

Il comando MATLAB per calcolare la fattorizzazione di Cholesky è chol.

Sebbene ci siano diverse condizioni equivalenti che stabiliscono quando una matrice sia definita positiva, il modo migliore per controllarlo in MATLAB è tramite la funzione `chol`. Se la funzione viene chiamata con un solo argomento `R=chol(A)`, restituisce R in caso affermativo, altrimenti dà l'errore `Error using chol. Matrix must be positive definite.`

Alternativamente, la funzione può essere chiamata con due argomeneti `[R,p]=chol(A)`, laddove p è un flag che vale 0 se A è definita positiva, ed un intero positivo altrimenti. Provare per le seguenti matrici particolari

```
M = magic(n)
H = hilb(n)
P = pascal(n)
I = eye(n,n)
R = randn(n,n)
R = randn(n,n); A = R'*R
R = randn(n,n); A = R'+R
R = randn(n,n); I = eye(n,n); A = R'+R+n*I
```

In seguito, richiamiamo le principali funzioni MATLAB per la fattorizzazione di matrici

`lu`	fattorizzazione `PA=LU`
`chol`	fattorizzazione $\texttt{A=R}^T\texttt{R}$
`qr`	fattorizzazione `A=QR`
`qz`	fattorizzazione `QZ`
`eig`	decomposizione spettrale
`svd`	decomposizione in valori singolari $\texttt{A=U}\Sigma\texttt{V}^T$
`schur`	decomposizione di Schur $\texttt{A=UTU}^H$

1.2 Tecnica del pivoting

Al fine di portare a compimento il processo di fattorizzazione LU per una qualunque matrice A non singolare si introduce la tecnica del pivoting. Essa consiste nel **permutare** (ossia scambiare) *opportunamente* le righe della matrice A di partenza, in modo che non ci siano elementi pivot che si annullino. Più precisamente, la tecnica si chiama *pivoting per righe*.

Ricordiamo che una **matrice di permutazione** è una matrice identità con le righe e le colonna scambiate: ogni riga e ogni colonna contiene solamente un uno, le altre entrate sono tutte nulle. Ad esempio,

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}. \tag{12}$$

Lo scambio di righe di P rispetto alla matrice identità può essere rappresentato dal *vettore di permutazione* `p = [4 3 1 2]`, mentre quello di colonne dal vettore di permutazione `q = [3 4 2 1]`. Data una qualsiasi matrice A, i seguenti comandi equivalenti permutano rispettivamente le righe e le colonne di A secondo la sintassi

```
%comandi che producono la stessa permutazione per righe
P*A    %permutazione delle righe di A secondo P
A(p,:) %permutazione delle righe di A secondo p
```

```
%comandi che producono la stessa permutazione per colonne
A*P    %permutazione delle colonne di A secondo P
A(:,q) %permutazione delle colonne di A secondo q
```

Tra i due modi, la seconda notazione (`A(p,:)` e `A(:,q)`) è più veloce e richiede meno memoria.

Enfatizziamo che il pivoting è possibile solo se A è non singolare, se e solo se $\det(A) \neq 0$. Purtroppo non è possibile stabilire a priori quali siano le righe che dovranno essere tra loro scambiate, ma si può decidere ad ogni passo k per cui (5) genera elementi $a_{kk}^{(k)}$ nulli. La fattorizzazione con pivoting restituisce la matrice A di partenza a meno di una permutazione fra le righe, per cui si avrà

$$PA = LU, \tag{13}$$

laddove P è un'opportuna matrice di permutazione. I sistemi triangolari corrispondenti da risolvere sono

$$L\mathbf{y} = P\mathbf{b}, \quad U\mathbf{x} = \mathbf{y}. \tag{14}$$

Osservando le equazioni in (5) è evidente che bisogna evitare elementi $a_{kk}^{(k)}$ piccoli, che stanno al denominatore dei moltiplicatori e possono comportare gravi perdite di accuratezza nel risultato finale, in quanto gli eventuali errori di arrotondamento presenti nei coefficienti $a_{kj}^{(k)}$ che vengono premoltiplicati con essi potrebbero risultare fortemente amplificati. Si sceglie allora, tra tutti i pivot disponibili $a_{ik}^{(k)}$ con $i = k, \ldots, n$, quello di modulo massimo.

Esempio 1.6 (Algoritmo di Gauss con pivoting per righe)**.** *Data $A \in \mathbb{R}^{n\times n}$, posto $A^{(1)} = A$,*

ovvero $a_{ij}^{(1)} = a_{ij}$ *per* $i, j = 1, \ldots, n$, *si calcoli*

$$
\begin{aligned}
&\textit{per } k = 1, \ldots, n-1 \\
&\quad \textit{trovare } \bar{r} \textit{ tale che } |a_{\bar{r}k}^{(k)}| = \max_{r=k,\ldots,n} |a_{rk}^{(k)}|, \\
&\quad \textit{scambiare la riga } k \textit{ con la riga } \bar{r} \\
&\quad \textit{sia in } A \textit{ che in } P \\
&\quad \textit{per } i = k+1, \ldots, n \\
&\qquad l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \\
&\qquad \textit{per } j = k+1, \ldots, n, \\
&\qquad\quad a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)} \\
&\qquad \textit{end} \\
&\quad \textit{end} \\
&\textit{end}
\end{aligned}
\tag{15}
$$

Il comando di MATLAB corrispondente alla fattorizzazione LU con pivoting per righe è `[L,U,P]=lu(A)`. Si può anche implementare la **pivotazione totale**, che estende la ricerca a tutti i pivot disponibili $a_{ik}^{(k)}$ con $i, j = k, \ldots, n$. Essa coinvolge sia le righe che le colonne del sistema e conduce alla costruzione di due matrici di permutazione P e Q, una sulle righe, l'altra sulle colonne tali

$$PAQ = LU. \tag{16}$$

In questo caso, la soluzione di $A\mathbf{x} = \mathbf{b}$ riconduce alla soluzione di due sistemi triangolari e uno di permutazione

$$L\mathbf{y} = P\mathbf{b}, \quad U\mathbf{x}^\star = \mathbf{y}, \quad x = Q\mathbf{x}^\star. \tag{17}$$

Il comando di MATLAB corrispondente è `[L,U,P,Q]=lu(A)`.

Parlando di costi computazionali, la pivotazione totale ha un costo superiore rispetto a quella parziale in quanto ad ogni passo della devono essere svolti molti più confronti. D'altra parte essa apporta dei vantaggi in termini di risparmio di memoria e di stabilità.

1.3 Matrici sparse, a banda e fenomeno fill-in

La fattorizzazione LU di una matrice A è la procedura che sta alla base di alcuni comandi principali di MATLAB (cf. sezione successiva). Prima di vedere questo, in questa sezione, discutiamo brevemente quando si presenta il fenomeno **fill-in** e come evitarlo.

Molto spesso nelle applicazioni si incontrano matrici con molti elementi nulli. È questo il caso delle

- ⋆ **matrici sparse**: matrici quadrate di dimensione n (con n^2 elementi) aventi un numero di elementi non nulli dell'ordine di n (e non di n^2);

⋆ **matrici a banda**: matrice $A \in \mathbb{R}^{n\times n}$ (o in $\mathbb{C}^{n\times n}$) *a banda inferiore* p se $a_{ij} = 0$ per $i > j + p$ e *banda superiore* q se $a_{ij} = 0$ per $j > i + q$; il massimo fra p e q viene detto larghezza di banda della matrice.

Il **pattern** di una matrice sparsa è l'insieme dei suoi elementi non nulli. Esso è ottenuto in MATLAB con l'istruzione `spy(A)`.

Per evitare sprechi di memoria, è naturale e conveniente memorizzare i soli elementi non nulli di matrici sparse, a banda di grandi dimensioni. Questo si può realizzare tramite i comandi

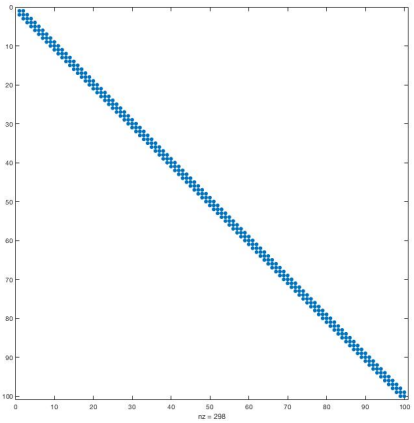
⋆ `A=sparse(m,n)`: inizializza a zero uno *sparse-array* (tipo di variabile) di m righe ed n colonne;

⋆ `A=spdiags(B,d,m,n)`: crea una matrice sparsa m per n nel formato *sparse-array* a partire dalle colonne di B e le posizione lungo le diagonali specificate da d.

Un noto esempio di matrice sparsa costruita tramite il comando `spdiags` è la matrice sparsa tridiagonale che rappresenta l'**operatore differenziale secondo** su n punti generata tramite i comandi

```
e = ones(n,1);
A = spdiags([e -2*e e], -1:1, n, n)
```

Il risultato di questo esempio mostra come certi elementi di B, che corrispondo alle posizioni "fuori" da A non vengono effettivamente usati. Ad esempio, per $n = 100$, la matrice A di sopra, ha questo pattern prodotto dal commando `spy(A)`



Un effetto non desiderato che può capitare è che il processo di fattorizzazione LU di queste matrici con molti elementi nulli riempia le matrici L ed U con elementi non nulli generando il cosiddetto fenomeno del **fill-in** (o **riempimento**). Il verificarsi di questo fenomeno dipende

fortemente sia dalla struttura della matrice A sia dal valore dei singoli elementi non nulli. Ad ogni caso, per questioni di risparmio di memoria, è un efffetto non desiderato.

Per ridurre questo fenomeno si possono usare **algoritmi di riordinamento** della matrice di partenza che consistono nel permutare le sue righe e colonne prima di realizzare la fattorizzazione. In molti casi, la pivotazione totale permette di raggiungere lo stesso obiettivo.

2 Cos'è nascosto dietro ai comandi MATLAB `det`, `inv` e backslash \ ?

Esempio 2.1 (Commando `det` per il calcolo del determinante)**.** *Osservando che*

$$det(A) = det(L)det(U) = \prod_{i=1}^{n} l_{ii} \prod_{i=1}^{n} u_{ii}, = \prod_{i=1}^{n} u_{ii} \tag{18}$$

MATLAB calcola il determinante, basandosi sulla fattorizzazione LU di A, con un costo computazione di $\mathcal{O}(n^3)$ operazioni.

Esempio 2.2 (Commando `inv` per il calcolo della matrice inversa)**.** *MATLAB utilizza una decomposizione LU della matrice di input, poi usa i risultati per formare un sistema lineare la cui soluzione è la matrice inversa.*

Esempio 2.3 (Commando `backslash` \ per la risoluzione di sistemi lineari)**.** *Questa funzione di MATLAB richiama diversi algoritmi specifici a seconda delle caratteristiche della matrice A. Più precisamente, MATLAB prima controlla se la matrice è full oppure spare*

- ⋆ *se A è full:*
  1. *se A è triangolare, superiore o inferiore (anche a meno di permutazioni), richiama l'algoritmo delle sotituzioni all'indietro o in avanti;*
  2. *se A è simmetrica con elementi positivi sulla diagonale principale, si richiama l'algoritmo di Cholesky;*
  3. *se A non è simmetrica, si richiama la fattorizzazione LU con pivoting per righe*
  4. *se A non è quadrata, si calcola una soluzione nel senso dei minimi quadrati mediante la fattorizzazione QR;*
- ⋆ *se A ha formato sparse, si eseguono i controlli 1.-4. precedenti e si controlla in più infine se la matrice è a banda; e richiamano varianti ottimizzate degli algoritmi precedenti.*

3 Problema complementare

Esercizio 3.1. *Si costruisca un M-file che rappresenti una versione semplificata della funzione built-in di MATLAB dell'operatore backslash. Più precisamente, si costruisca solamente un codice che controlli se la matrice sia*

1. *triangolare inferiore*
2. *triangolare superiore*
3. *simmetrica definita positiva*

e risolva per ciascun caso. Questa funzione deve richiamare al suo interno le funzioni forward.m *e* backward.m *definite nella lezione precedente, così come la funzione bult-in* `chol` *di MATLAB. Si facciano diverse prove con esempi vari.*

Soluzione:

```
function x = bslash(A,b)
% x = bslash(A,b) risolve A*x = b per alcuni casi particolari

[n,n] = size(A);
if isequal(triu(A,1),zeros(n,n))
    %se la matrice e' triangolare inferiore
     x = forward(A,b);
     return
elseif isequal(tril(A,-1),zeros(n,n))
    %se la matrice e' triangolare superiore
    x = backward(A,b);
    return
elseif isequal(A,A')
    [R,fail] = chol(A);
    if ~fail
      %la matrice e' definita positiva
       y = forward(R',b);
       x = backward(R,y);
      return
    end
end
```

Ultima versione aggiornata: 20 Maggio 2020

Link alle lezioni precedenti:
Lezioni 9-10
Lezioni 7-8

Lezioni 5-6
Lezioni 3-4
Lezioni 1-2

Referenze

1. MATLAB® The language of technical computing: computation, visualization, programming, The MathWorks Inc
2. MATLAB: An introduction with applications, A. Gilat, Wiley
3. Scientific Computing with MATLAB and Octave, A. Quarteroni, , F. Saleri, P. Gervasio, Springer
4. MATLAB Guide1 D. J. Higham and N. J. Higham, SIAM
5. Numerical Computing with MATLAB, C. Moler, SIAM