Serie 2a - Soluzione
Sistemi Lineari – Metodi Diretti

N.b.: La function `lugauss` allegata contiene l'implementazione dell'algoritmo di fattorizzazione LU **senza** pivoting (non vengono quindi mai applicate permutatazioni di righe/colonne), così come riportato nel testo del laboratorio. In seguito useremo la funzione built-in di Matlab® `lu`, che effettua il pivoting se necessario o anche solo conveniente dal punto di vista computazionale.

Esercizio 1.1

1. Introduciamo le matrici A, B e C, utilizzando i seguenti comandi

```
>> A = [ 50 1 3; 1 6 0; 3 0 1 ];
>> B = [ 50 1 10; 3 20 1; 10 4 70 ];
>> C = [ 7 8 9; 5 4 3; 1 2 6 ];
```

tramite il comando `det` è possibile verificare che le tre matrici introdotte sono tutte non singolari.
Si osserva che la matrice A è simmetrica. Inoltre si può dimostrare che essa è anche definita positiva, cioè vale la relazione $\mathbf{x}^T A\mathbf{x} > 0\ \forall \mathbf{x} \in \mathbb{R}^n$ con $\mathbf{x} \neq \mathbf{0}$. A tale scopo si utilizza il seguente criterio: condizione necessaria e sufficiente affinchè una matrice $A \in \mathbb{R}^{n\times n}$ sia definita positiva è che le sottomatrici principali A_i di A di ordine $i = 1, \ldots, n$ (cioè quelle ottenute limitando A alle prime i righe e colonne) abbiano determinante positivo (criterio di *Sylvester*). Si verifica con i seguenti comandi che la matrice A in questione soddisfi le ipotesi di questo criterio ed è anche una matrice simmetrica:

```
>> if isequal(A,A')
        disp('A e'' simmetrica')
        na = size(A, 1);
        for i = 1:na
            if (det(A(1:i,1:i))>0)
                if (i == na)
                    disp('A e'' definita positiva')
                end
            else error('A non e'' definita positiva')
            end
        end
    else
        disp('A non e'' simmetrica')
    end

A e' simmetrica
A e' definita positiva
```

Sarà dunque possibile ottenere la fattorizzazione LU senza pivoting.

Consideriamo ora la matrice B; si può osservare che è una matrice a dominanza diagonale stretta per colonne, cioè $|B(j,j)| > \sum_{i=1, i\neq j}^{n} |B(i,j)|$ per ogni colonna j. Con i seguenti comandi verifichiamo che la matrice B soddisfa questa ipotesi per l'esistenza della fattorizzazione LU senza pivoting:

```
>> db = diag(B);
>> nb = size(B,1);
>>  for j = 1:nb
            r(j) = abs(db(j)) - sum(abs(B([1:j-1,j+1:nb], j)));
    end
>>  if (r > 0)
            disp('B e'' a dominanza diagonale stretta per colonne')
    end

B e' a dominanza diagonale per colonne
```

Consideriamo infine la matrice C. Essa non presenta alcuna caratteristica particolare, per cui verifichiamo che la fattorizzazione LU senza pivoting sia applicabile mediante la condizione necessaria e sufficiente. Con i seguenti comandi verifichiamo pertanto che i suoi minori principali di ordine da 1 ad $n-1$ hanno determinante diverso da zero:

```
>> i = 1;
>> dc = 1;
>> nc = size(C,1);
>>  while ( (i < nc) && (dc ~=0 ) )
            dc = det(C(1:i, 1:i));
            i = i+1;
    end
>>  if (dc == 0)
            disp('non posso applicare la decomposizione LU ad C')
    else
            disp('posso applicare la decomposizione LU a C')
    end

posso applicare la decomposizione LU a C
```

2. Calcoliamo la fattorizzazione LU delle precedenti matrici utilizzando il metodo di Gauss.

   Matrice A:

```
>> [LA, UA] = lu(A)
LA =

    1.0000         0         0
    0.0200    1.0000         0
    0.0600   -0.0100    1.0000


UA =

   50.0000    1.0000    3.0000
         0    5.9800   -0.0600
```

```
         0         0    0.8194
```

Verifichiamo infine l'esattezza della fattorizzazione tramite i comandi

```
>> A - LA*UA

ans =

     0     0     0
     0     0     0
     0     0     0
```

Consideriamo ora la matrice B:

```
>> [LB, UB] = lu(B)
LB =

    1.0000         0         0
    0.0600    1.0000         0
    0.2000    0.1906    1.0000


UB =

   50.0000    1.0000   10.0000
         0   19.9400    0.4000
         0         0   67.9238
```

Verifichiamo anche in questo caso l'esattezza della fattorizzazione tramite il comando

```
>> B - LB*UB

ans =

     0     0     0
     0     0     0
     0     0     0
```

Consideriamo Infine la matrice C:

```
 >> [LC, UC] = lu(C)
LC =

    1.0000         0         0
    0.7143    1.0000         0
    0.1429   -0.5000    1.0000


UC =

    7.0000    8.0000    9.0000
         0   -1.7143   -3.4286
         0         0    3.0000
```

e verifichiamo per C l'esattezza della fattorizzazione:

```
>> C - LC*UC

ans =

     0     0     0
     0     0     0
     0     0     0
```

3. Si vedano le funzioni `fwsub.m` e `bksub.m`

4. Costruiamo il termine noto **b**:

```
>> b = A * ones(na,1);
```

   Per risolvere il sistema lineare $A\mathbf{x} = \mathbf{b}$ usiamo le funzioni `fwsub.m` e `bksub.m` con i seguenti comandi:

```
>> sol = ones(3,1);
>> b = A * sol;
>> y = fwsub(LA, b);
>> x = bksub(UA, y)
x =

    1.0000
    1.0000
    1.0000
```

5. La norma 2 dell'errore relativo residuo viene calcolata con il seguente comando

```
 >> err_rel = norm(sol - x)/norm(x)
err_rel =

   1.4333e-16
```

   e la norma 2 del residuo relativo si puo ottenere con il comando

```
>> res_nor = norm(b - A*x)/norm(b)
res_nor =

   1.6267e-17
```

Esercizio 1.2

Costruiamo la matrice A con i comandi:

```
>>  A = [ 50 1 3; 1 6 0; 3 0 1 ];
>>  n = size(A,1);
```

Vogliamo calcolare la matrice inversa di A, utilizzando le funzioni `lu`, `bksub.m` e `fwsub.m` implementate nel precedente esercizio. Per risolvere l'esercizio utilizziamo i seguenti comandi:

```
>> [L,U] = lu(A);
>> I = eye(n);
>> Ainv = [];
>>  for i = 1:n
            yn = fwsub( L, I(:,i));
            xn = bksub( U, yn );
            Ainv = [Ainv, xn];
    end
>> Ainv
Ainv =

    0.0245   -0.0041   -0.0735
   -0.0041    0.1673    0.0122
   -0.0735    0.0122    1.2204
```

Controlliamo che il risultato ottenuto sia corretto:

```
 >> Id =A*Ainv

Id =

    1.0000         0   -0.0000
    0.0000    1.0000   -0.0000
    0.0000         0    1.0000

>> eye(3) - Id

ans =

   1.0e-15 *

         0         0    0.4441
   -0.0069    0.2220    0.0139
   -0.0139         0         0
```

Si osservi che l'errore di arrotondamento (roudoff) compiuto nel calcolo dell'inversa è dello stesso ordine di grandezza dell'epsilon macchina della matrice identità (gli elementid della diagonale sono l'epsilon macchina di 1):

```
 >> eps(eye(3))

ans =

   1.0e-15 *

    0.2220    0.0000    0.0000
    0.0000    0.2220    0.0000
    0.0000    0.0000    0.2220
```

Questo significa che l'errore `eye(3)-Id` rappresenta una perturbazione della matrice identità che può essere amplificato in eventuali calcoli successivi. Notiamo, infine, che la funzione

Matlab® `inv` implementa un metodo analogo a quello da noi utilizzato se pur in modo più efficiente e stabile, a meno di possibili errori di roundoff.

```
 >> inv(A)

ans =

    0.0245   -0.0041   -0.0735
   -0.0041    0.1673    0.0122
   -0.0735    0.0122    1.2204
```

Esercizio 1.3

1. Introduciamo la matrice A utilizzando i seguenti comandi:

   ```
   >> n = 20;
   >> I0 = 2;
   >> R = ones(n,1);
   >> A = -diag(R) + diag(R(1:n-1),-1);
   >> A(1,:) = 1;
   >> b = zeros(n,1);
   >> b(1) = I0;
   ```

2. La fattorizzazione LU della matrice A si può calcolare mediante il comando Matlab® `lu`:

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

   Le tre variabili restituite sono, rispettivamente, le matrici L ed U della fattorizzazione e la matrice di permutazione P. Per controllare se è stato effettuato il pivoting confrontiamo la matrice P con la matrice identità:

   ```
   >> if (P == eye(n))
        disp('non e'' stato effettuato pivoting')
    else
        disp('e'' stato effettuato pivoting')
    end
   ```

   In questo caso l'output è:

   ```
   non e' stato effettuato pivoting
   ```

   Non è stato quindi effettuato alcun pivoting per righe sulla matrice A.

3. Risolviamo numericamente i due sistemi triangolari $L\mathbf{y} = \mathbf{b}$ e $U\mathbf{i} = \mathbf{y}$, dato che in questo caso $P \equiv I$, utilizzando le funzioni `bksub` e `fwsub`:

   ```
   >> y = fwsub(L, b);
   >> i = bksub(U, y);
   ```

4. Calcoliamo l'errore relativo, il residuo normalizzato in norma 2 e il numero di condizionamento:

```
>> i_ex = b(1)/n * ones(n,1);
>> err_rel = norm(i - i_ex)/norm(i_ex)
err_rel =

   5.2479e-16


>> res_nor = norm(A*i -b)/norm(b)
res_nor =

   2.7773e-16


>> Cond_A = cond(A)
Cond_A =

   28.4998
```

Osserviamo che, essendo il numero di condizionamento "piccolo", l'errore è dello stesso ordine di grandezza del residuo. In tal caso, piccoli valori del residuo indicano una buona accuratezza della soluzione. Ciò non è in generale vero se il numero di condizionamento è grande.

5. Poniamo ora $R_1 = 10^3$ e calcoliamo la nuova distribuzione delle correnti.

```
>> A_mod = A;
>> A_mod(2,1) = 10^3;
>> [L_mod,U_mod,P_mod] = lu(A_mod);
>> if (P_mod == eye(n))
     disp('non e'' stato effettuato pivoting')
 else
     disp('e'' stato effettuato pivoting')
 end
>> y = fwsub(L_mod, P_mod*b);
>> i_mod = bksub(U_mod, y);
```

È importante notare che, a differenza del punto 2, in questo caso il pivoting (per righe) è stato effettivamente effettuato. I due sistemi triangolari da risolvere in questo caso sono dunque $L\mathbf{y} = P\mathbf{b}$ e $U\mathbf{i} = \mathbf{y}$. Notiamo inoltre che il pivoting non era strettamente necessario per generare la fattorizzazione LU di `A_mod`. Infatti, la matrice `A_mod` soddisfa la condizione necessaria e sufficiente di esistenza della fattorizzazione LU senza pivoting, in quanto le sottomatrici principali di ordine $i = 1 \dots n-1$ sono tutte non singolari:

```
>> i = 1;
>> dc = 1;
>> nc = size(A_mod,1);
>> while ( (i < nc) && (dc ~= 0) )
     dc = det(A_mod(1:i, 1:i));
     i = i+1;
 end
```

```
>> if (dc == 0)
     disp('non posso applicare la decomposizione LU ad A_mod')
 else
     disp('posso applicare la decomposizione LU a A_mod')
 end
posso applicare la decomposizione LU a A_mod
```

Tuttavia, l'utilizzo nell'algoritmo di fattorizzazione LU di moltiplicatori m_{ik} grandi, generati da elementi pivotali $a_{kk}^{(k)}$ piccoli, può considerevolmente amplificare gli errori macchina di arrotondamento, che vengono propagati durante la fattorizzazione. Per questo Matlab® (che implementa un algoritmo estremamente ottimizzato) ad ogni passo k della fattorizzazione LU tramite MEG sceglie come pivot l'elemento di modulo massimo nella colonna `A(k:n, k)`. In generale pertanto la permutazione per righe viene effettuata anche quando le condizioni di esistenza e unicità della fattorizzazione LU sono comunque verificate e i pivot sono quindi tutti non nulli.

Calcoliamo infine il numero di condizionamento di A:

```
>> Cond_A_mod = cond(A_mod)
Cond_A_mod =

   6.0548e+03
```

Il numero di condizionamento della nuova matrice è "grande"; questo è dovuto al fatto che ora le componenti della matrice hanno ordini di grandezza diversi.

Osservazione. Dalle soluzioni `I` e `Imod` si nota che:

- nel caso in cui le resistenze siano tutte uguali in ciascun ramo del circuito passa la medesima corrente, pari a `I0/n`.
- nel caso in cui una delle resistenze sia molto maggiore delle altre, la corrente che attraversa tale ramo risulta molto più piccola rispetto alle altre e quasi tutta la corrente `I0` si ripartisce negli altri $n-1$ rami.

Osservazione. Il comando Matlab® `lu` si comporta in maniera differente a seconda del numero di variabili in output richieste:

- `>> [L,U,P] = lu(A)` restituisce una matrice triangolare inferiore L, una matrice triangolare superiore U e una matrice di permutazione (per righe) P tali che $LU = PA$;
- `>> [M,U] = lu(A)` restituisce una matrice *permutata* M e una matrice triangolare superiore U tali che $MU = A$. Attenzione: in generale M *non* è triangolare inferiore (nonostante nell'help di Matlab® sia genericamente indicata come L). Infatti, Matlab® esegue la tecnica del pivoting per righe anche se non strettamente richiesto per generare la fattorizzazione LU di A tramite MEG; inoltre, la matrice di permutazione P, pur non coicidendo con l'identità in caso di permutazioni di righe, non viene restituita dal comando Matlab® `[M,U] = lu(A)`. La matrice M e la matrice triangolare inferiore L restituite in output utilizzando la sintassi precedente sono legate dalla relazione $M = P^T L$.

Pertanto, se utilizziamo la prima sintassi `[L,U,P] = lu(A)` (con 3 variabili in output), è possibile risolvere il sistema lineare $A\mathbf{x} = \mathbf{b}$ con i seguenti passaggi (da confrontare con lo script al punto 3):

```
>> y = fwsub(L, P*b);
>> i = bksub(U, y);
```

Se utilizziamo invece la seconda sintassi `[M,U] = lu(A)` (con 2 variabili in output), *non* è possibile in generale risolvere il sistema utilizzando le funzioni `fwsub` e `bksub`, siccome la matrice M potrebbe non essere triangolare inferiore in presenza di permutazioni di righe.

Suggerimento. Quando si effettua la fattorizzazione LU di una matrice A tramite la funzione `lu` di Matlab®, si raccomanda di utilizzare sempre il comando:

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

ovvero, si assuma che la fattorizzazione LU venga sempre eseguita dalla funzione `lu` di Matlab® con la tecnica del pivoting (permutazione) per righe. Infine, dopo aver ottenuto la matrice triangolare inferiore L, la matrice triangolare superiore U e la matrice di permutazione per righe P si risolva il sistema lineare $A\mathbf{x} = \mathbf{b}$ come:

```
>> y = fwsub(L, P*b);
>> i = bksub(U, y);
```

Esercizio 2.1

I comandi Matlab® da usare sono:

```
>> n=1000;
>> A=hilb(n);
>> x_ex=ones(n,1);
>> b=A*x_ex;

>> B=rand(n);
>> y_ex=ones(n,1);
>> c=B*y_ex;

>> x=A\b;
>> y=B\c;
```

Notiamo che A è estremamente mal condizionata, al contrario di B:

```
>> cond(A)
ans =
   5.5647e+20
>> cond(B)
ans =
   8.0236e+04
```

Come è noto, risolvere un sistema lineare con matrice mal condizionata dà risultati poco accurati. Questo è evidente se confrontiamo gli errori relativi per le soluzioni dei due sistemi:

```
>> norm(x-x_ex)/norm(x_ex)
ans =
   4.5304e+03
>> norm(y-y_ex)/norm(y_ex)
ans =
   9.6576e-13
```

Esercizio 3.1

La funzione che implementa l'algoritmo di Thomas è riportata nel file `thomas.m` allegato. Il suo utilizzo è il seguente:

```
>> K=100;        L=20;    N=20;
>> n=N-1;
>> %costruisco la matrice
>> extradiag=ones(n-1,1);
>> maindiag=-2*ones(n,1);
>> A=diag(maindiag,0)+diag(extradiag,1)+diag(extradiag,-1);
>> A=K*A;
>> %costruisco il termine noto
>> t_noto=zeros(n,1);
>> t_noto(end)=t_noto(end)-K*L;

>> [LL,UU,x_thom]=thomas(A,t_noto);
```

La soluzione è riportata in figura 1. Il grafico di sinistra rappresenta la posizione di ciascun anello, mentre quello di destra riporta gli allungamenti di ciascuna molla (calcolati come distanza fra due anelli consecutivi). Si noti che in questo caso ciascuna molla è sottoposta allo stesso allungamento, dal momento che su nessun anello agiscono forze esterne e tutte le molle hanno la stessa costante elastica.

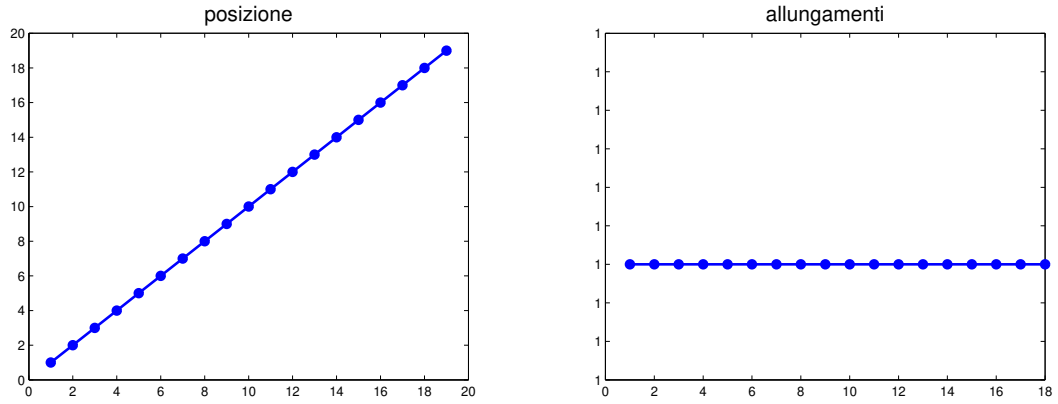

Figura 1: soluzione del sistema lineare in assenza di forzanti esterne

Se a ciascun anello viene invece applicata una forza esterna $\boldsymbol{F}_{i,ext}$, si deve modificare il termine noto del sistema lineare: il bilancio delle forze all'i-esimo anello non è più

$$\boldsymbol{F}_i + \boldsymbol{F}_{i+1} = \boldsymbol{0}$$

ma diventa

$$\boldsymbol{F}_i + \boldsymbol{F}_{i+1} + \boldsymbol{F}_{i,ext} = \boldsymbol{0}$$

e quindi l'i-esima equazione diventa (supponendo che $\boldsymbol{F}_{i,ext}$ sia diretta come $\boldsymbol{F}_i$)

$$-K(x_i - x_{i-1}) + K(x_{i+1} - x_i) - |\boldsymbol{F}_{i,ext}| = 0 \Rightarrow Kx_{i-1} - 2Kx_i + Kx_{i+1} = |\boldsymbol{F}_{i,ext}|.$$

Ad esempio possiamo imporre su ciascun anello una forzante esterno di intensità casuale tramite i comandi:

```
>> t_noto=16*rand(n,1);
>> t_noto(end)=t_noto(end)-K*L;
```

In questo modo si ottengono degli allungamenti non uniformi, come si puó osservare in figura 2.

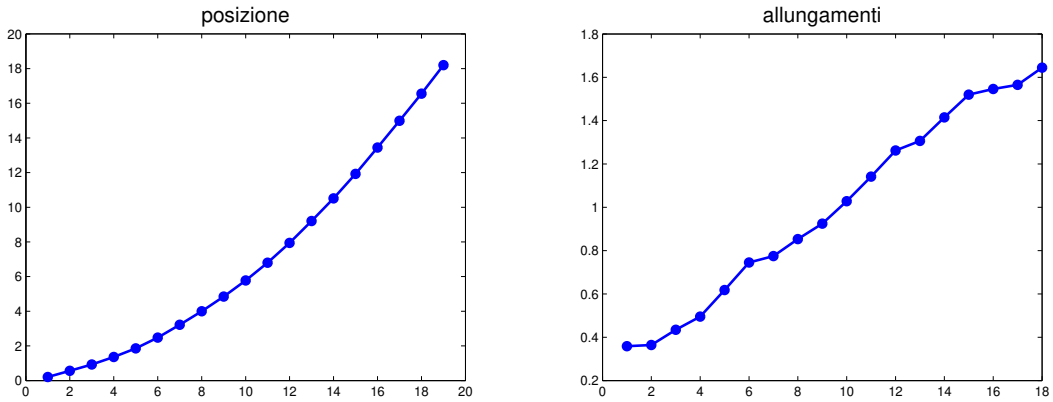

Figura 2: soluzione del sistema lineare in presenza di forzanti esterne

Esercizio 4.1

Tramite i comandi

```
clc
clear

% parametri
K = 100;
l = 20;

% vettore dei tempi di calcolo
iter = 10;
T = zeros (3, iter);

for i = 1:iter
    % stampo il numero di iterazione
    disp (strcat ('iter ', num2str(i)))

    %aumento la dimensione
    n = 500*i;

    %costruisco la matrice
    extradiag = ones(n-1, 1);
    maindiag = -2*ones(n, 1);
    A = diag(maindiag, 0) + diag (extradiag, 1) + diag (extradiag, -1);
    A = K*A;
    %costruisco il termine noto
    t_noto = zeros (n, 1);
    %aggiungo la forzante esterna
    t_noto(end) = t_noto(end) - K*l;


    %risolvo con LU, thomas, Choleski e memorizzo nella matrice T

    t = tic;
    [L, U, P] = lu (A);
    x_lu = U \ (L \ (P*t_noto));
    T (1, i) = toc (t);

    t = tic;
    [L, U, x_thom] = thomas (A, t_noto);
    T (2, i) = toc (t);

    A_chol = -A;
    t_chol = -t_noto;
    t = tic;
    H = chol (A_chol);
    x_chol = H \ (H' \ t_chol);
    T (3, i) = toc (t);
end

figure;
hold on;
dim = 500 * [1:iter];
plot(dim, T(1,:),'-ob', 'linewidth', 2);
plot(dim, T(2,:),'-or', 'linewidth', 2);
```

```
plot(dim, T(3,:),'-ok', 'linewidth', 2);

legend('lu','thomas','chol', 'Location', 'NorthWest')
```

si memorizzano i tempi di calcolo nella matrice `T`, e si ottiene un grafico in cui in ascissa è riportata la dimensione del sistema lineare, e in ordinata il tempo di calcolo impiegato dai vari metodi (vedi figura 3).

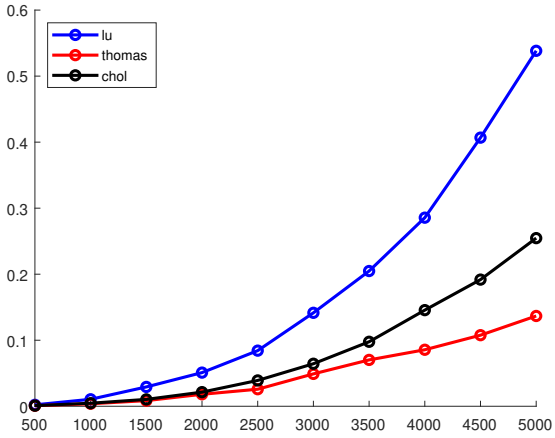

Figura 3: confronto fra i tempi di calcolo

Il guadagno in termini di tempo di esecuzione comincia a farsi sensibile (rispetto al metodo di Thomas) intorno a $N = 1500$ per il metodo LU, e a $N = 2500$ per Cholesky. Si vede che l'algoritmo di Thomas, come ci si aspettava, è quello con le prestazioni migliori dal momento che è l'unico algoritmo fra quelli utilizzati ad essere disegnato *ad hoc* per sistemi tridiagonali.

N.b.: Il comando \ di Matlab® utilizza la fattorizzazione di Cholesky o il metodo LU, a seconda che la matrice sia simmetrica e definita positiva o meno. Non utilizza invece mai il metodo di Thomas, a meno che la matrice usata non sia in formato sparso. Il controllo sulla tridiagonalità di una matrice piena avrebbe infatti costo computazionale dell'ordine di n^2, cosa che renderebbe inutile l'utilizzo di un metodo di soluzione lineare nel tempo di calcolo.

Esercizio 5.1

La matrice A si definisce attraverso i seguenti comandi

```
>> n = 20;
>> A = 4*diag(ones(n,1)) - diag(ones(n-1,1),-1) - diag(ones(n-1,1),1);
>> A(1,:) = ones(1,n);
>> A(:,1) = ones(n,1);
```

e la posizione dei suoi elementi non nulli si visualizza ad esempio come segue (Figura 4)

```
>> figure; spy(A); title('A');
```

Calcoliamo ora la fattorizzazione LU di A e visualizziamo le matrici L e U

```
>> [L,U,P] = lu(A);
>> figure; spy(L); title('L');
>> figure; spy(U); title('U');
```

Otteniamo così la Figura 4. Si osserva che le matrici L e U non sono sparse, poiché il loro numero di elementi non nulli è circa $n^2/2$. Il fenomeno per cui una matrice sparsa dà luogo a fattori (LU o Cholesky) generalmente non sparsi è chiamato *fill-in* ("riempimento" delle matrici con elementi non nulli). Dunque, in generale l'algoritmo di fattorizzazione sostituisce ad elementi nulli in A degli elementi non nulli in L e U.

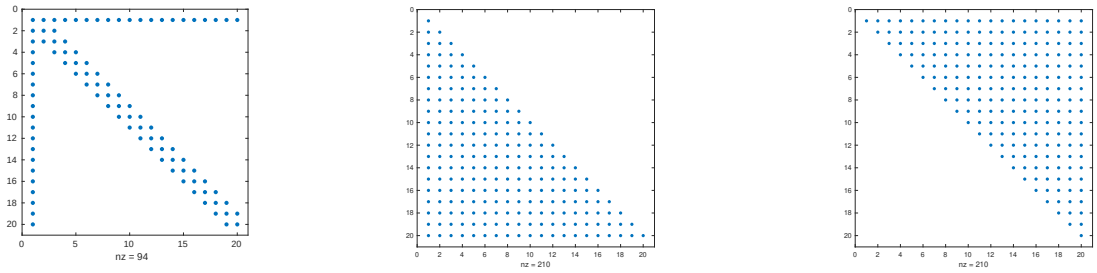

Figura 4: Rappresentazione degli elementi non nulli di A, L e U

Il *fill-in* è svantaggioso in termini di gestione della memoria. Nelle applicazioni al calcolo scientifico, ovvero con n grande, si ha quasi sempre a che fare con matrici sparse per le quali viene riservata memoria solo per gli elementi non nulli, che tipicamente sono $\mathcal{O}(n) \ll \mathcal{O}(n^2)$. Ad esempio, definendo le matrici del presente laboratorio per $n = 100$, si osserva che l'occupazione di memoria nel formato sparso di Matlab consente un notevole risparmio per la matrice A, ma non per i fattori LU:

```
>> n = 100;
>> A = 4*diag(ones(n,1)) - diag(ones(n-1,1),-1) - diag(ones(n-1,1),1);
>> A(1,:) = ones(1,n); A(:,1) = ones(n,1);
>> [L,U,P] = lu(A);
>> As = sparse(A); Ls = sparse(L); Us = sparse(U);
>> % Esaminiamo il Workspace e i Bytes di memoria allocati
whos

  Name        Size              Bytes  Class     Attributes

  A         100x100             80000  double
  As        100x100              8712  double    sparse
  L         100x100             80000  double
  Ls        100x100             81608  double    sparse
  P         100x100             80000  double
  U         100x100             80000  double
  Us        100x100             81608  double    sparse
  n           1x1                   8  double
```

Si noti come la memoria allocata per `As` è meno di un decimo di quella allocata per `A`; lo stesso guadagno non si ottiene invece per `Ls` e `Us`, in quanto la loro struttura non è sparsa.

Quando la fattorizzazione LU comporta una generazione di elementi non nulli dell'ordine di $\mathcal{O}(n^2)$ (*fill-in*), l'allocazione ulteriore di memoria è proibitiva per n grande e rende impos-

sibile l'uso della fattorizzazione stessa. Per questo, possibili rimedi consistono in un riordino opportuno degli elementi di A in maniera tale da minimizzare il *fill-in* dei fattori L e U.

La tecnica di pivotazione totale può essere convenientemente applicata per prevenire e/o contenere il fenomeno del fill-in. La pivotazione totale applicata alla matrice $A \in \mathbb{R}^{n\times n}$ consiste nell'introdurre, oltre alla matrice di permutazione per righe $P \in \mathbb{R}^{n\times n}$ da pre-moltiplicare ad A, una matrice di permutazione per colonne $Q \in \mathbb{R}^{n\times n}$ da post-moltiplicare ad A come AQ. La matrice di permutazione per colonne Q è ortogonale, cioè $Q^T = Q^{-1}$ (per cui $Q^TQ = I$); se $Q = I$, allora non vi sono permutazioni di colonne della matrice A. Applicando la tecnica di pivotazione totale per la determinazione della fattorizzazione LU della matrice non singolare A, si ha

$$PAQ = \mathrm{LU}.$$

Osserviamo che la permutazione totale comporta costi computazionali aggiuntivi rispetto alla pivotazione per righe; infatti, per ogni $k = 1, \ldots, n-1$, la ricerca dell'elemento pivotale *migliore* avviene su in insieme di $(n+1-k)^2$ elementi – della sottomatrice $A^{(k)} \in \mathbb{R}^{(n+1-k)\times(n+1-k)}$ – che è molto più grande di quello corrispondente alla riga k della sottomatrice $A^{(k)}$, ovvero gli $(n+1-k)$ elementi $\{(A^{(k)})_{ik}\}_{i=k}^{n}$. Per tale motivo, in Matlab la pivotazione totale viene operata solo su matrici sparse.

Nel caso in esame, con i comandi

```
As = sparse(A); Ls = sparse(L); Us = sparse(U);
[Ls,Us,Ps,Qs] = lu(As)
```

è possibile ottenere fattori L e U sparsi. Nella figura 5 sono riportati i pattern di Ls e Us (ovvero dei fattori L e U in formato sparso), oltre che della matrice PAQ. Risulta evidente l'effetto del pivoting totale sulla sparsità dei fattori L e U ottenuti in presenza di pivoting totale.

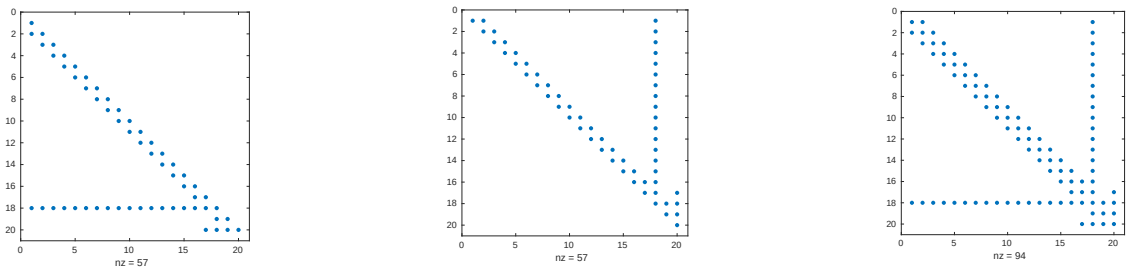

Figura 5: Rappresentazione degli elementi non nulli di L, U e di PAQ.

Concludiamo osservando un'importante proprietà della fattorizzazione LU. Una matrice $A \in \mathbb{R}^{n\times n}$ è detta *a banda*, con *ampiezza di banda* p, se $a_{ij} = 0$ per $|i-j| \geq p$. La fattorizzazione LU di A conserva la struttura a banda, ossia, posto $\mathrm{L} = (l_{ij})$ e $\mathrm{U} = (u_{ij})$)

$$a_{ij} = 0 \quad \text{per} \quad |i-j| \geq p \implies \begin{cases} l_{ij} = 0 & \text{per} \quad i-j \geq p, \\ u_{ij} = 0 & \text{per} \quad j-i \geq p. \end{cases}$$

Per questo motivo, l'applicazione del metodo di eliminazione di Gauss alla matrice A (che ha banda di ampiezza n) comporta il riempimento della banda di molti elementi diversi da zero (fill-in della banda); è dunque coerente trovare dei fattori L e U avendo entrambi banda di ampiezza n.

Esercizio 6.1

1. Il numero di condizionamento spettrale è definito come il rapporto tra il modulo massimo e minimo degli autovalori. In questo caso:

$$K(A) = \frac{|\lambda_1(A)|}{|\lambda_2(A)|} = \frac{2 + 2\cos\left(\frac{\pi}{n+1}\right)}{2 + 2\cos\left(\frac{\pi n}{n+1}\right)} = \frac{1 + \cos\left(\frac{\pi}{n+1}\right)}{1 - \cos\left(\frac{\pi n}{n+1} - \pi\right)} = \frac{1 + \cos\left(\frac{\pi}{n+1}\right)}{1 - \cos\left(\frac{\pi}{n+1}\right)}$$

   Nel rapporto precedente, il numeratore tende a 2 per $n \to \infty$. Il denominatore tende a zero, in modo asintotico a $\frac{1}{2}\left(\pi/(n+1)\right)^2$. Per $n \to \infty$, avremo quindi:

$$K(A) \sim \frac{2}{\frac{1}{2}\left(\frac{\pi}{n+1}\right)^2} = \frac{4(n+1)^2}{\pi^2} \sim \frac{4n^2}{\pi^2} \, .$$

2. Possiamo costruire la matrice A e calcolarne il numero di condizionamento spettrale, al variare di n, con i seguenti comandi:

```
nn = 10 * (1:10);
K  = zeros(size(nn));

for i = 1:length(nn)
      n = nn(i);

      A        = 2 * eye(n) - diag(ones(n-1, 1), 1) - diag(ones(n-1, 1), -1);
      abs_eigA = abs(eig(A));
      K(i)     = max(abs_eigA) / min(abs_eigA);
end

K_stima = 4 * nn.^2 / (pi^2);

semilogy(nn, K, 'bo-', 'LineWidth', 2);
hold on;
semilogy(nn, K_stima, 'r-', 'LineWidth', 2);
xlabel("n");
ylabel("K(A)");
grid on;
legend("K(A)", "K(A) stimato");
```

   I comandi precedenti producono il grafico di Figura 6. Osserviamo come la stima ottenuta al punto 1 descriva accuratamente l'andamento di $K(A)$ per valori di n crescenti.

3. Data la struttura tridiagonale della matrice, il metodo diretto più efficace è il metodo di Thomas, che impiega $8n-7$ operazioni per calcolare la fattorizzazione LU della matrice A e risolvere il sistema lineare tramite sostituzioni in avanti e all'indietro. Il metodo è più

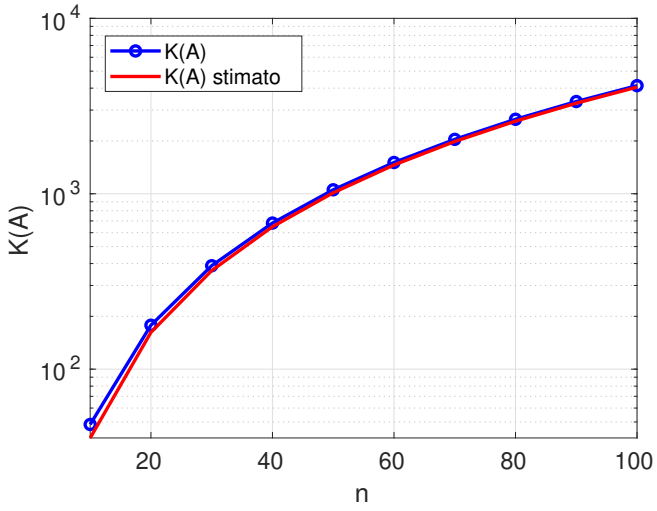

Figura 6: Condizionamento spettrale $K(A)$ e sua stima asintotica al variare di n.

conveniente della semplice fattorizzazione LU, dal momento che sfrutta la conoscenza della struttura della matrice per svolgere meno operazioni sia durante la fattorizzazione sia durante la risoluzione dei sistemi $L\mathbf{y} = \mathbf{b}$ e $U\mathbf{x} = \mathbf{y}$ rispettivamente tramite i metodi delle sostituzioni in avanti e all'indietro. Il costo computazionale complessivo del metodo della fattorizzazione LU è $O\left(\frac{2}{3}n^3\right)$. Di conseguenza, il rapporto tra i numeri di operazioni della fattorizzazione LU e dell'algoritmo di Thomas è $O\left(\frac{n^2}{12}\right)$.

4. Sfruttiamo la proprietà che il determinante del prodotto di due matrici è il prodotto dei loro determinanti. Questo si applica in particolare alla fattorizzazione LU di una matrice: se $A = LU$, allora $\det(A) = \det(L)\det(U)$. Infine, per matrici triangolari (superiori o inferiori), il determinante è il prodotto degli elementi sulla diagonale. Gli elementi diagonali di L sono tutti pari ad 1, pertanto $\det(L) = 1$ e $\det(A) = \det(U) = U_{11}\, U_{22}\, \ldots\, U_{nn}$. Un algoritmo efficiente per calcolare $\det(A)$ è dunque il seguente:

   (a) utilizzare il metodo di Thomas per calcolare la fattorizzazione LU di A (in $3(n-1)$ operazioni);

   (b) calcolare il prodotto $\det(A) = \det(U) = U_{11}\, U_{22}\, \ldots\, U_{nn}$ ($n-1$ operazioni).

   Il numero totale di operazioni richieste per il calcolo del determinante è dunque pari a $4(n-1)$. Si veda il file `det_tridiag.m` per una possibile implementazione dell'algoritmo.

5. Vale la seguente stima:

$$\frac{\|\delta\mathbf{x}\|_2}{\|\mathbf{x}\|_2} \leq K_2(A)\frac{\|\delta\mathbf{b}\|_2}{\|\mathbf{b}\|_2}\,. \tag{1}$$

   I comandi seguenti valutano l'errore stimandolo sia usando la disuguaglianza precedente, sia risolvendo il sistema lineare con un metodo diretto tramite il comando \

```
n = 100;
A = 2 * eye(n) - diag(ones(n-1, 1), 1) - diag(ones(n-1, 1), -1);
```

```
x = ones(n, 1);
b = A * x;

c = rand(size(b));
c = c / norm(c);

delta_b = 1e-6 * c;
delta_x = A \ (b + delta_b) - x;

err       = norm(delta_x) / norm(x)           % = 8.0608e-05
err_stima = cond(A) * norm(delta_b) / norm(b) % = 0.0029
```

La stima dell'errore (1) risulta dunque verificata, ovvero l'errore commesso è effettivamente inferiore all'errore stimato. Si noti che il valore dell'errore effettivo, salvato nella variabile **`err`**, può variare a causa della definizione casuale del vettore **c**.