Appunti

ESERCITAZIONE 8 - ESERCIZI A

Sistemi Lineari e Fattorizzazione LU

ESERCIZIO A.1

Function lu di MATLAB/Octave - main_linsys.m

OBIETTIVO

Risolvere un sistema lineare Ax = b utilizzando due metodi:

1. Operatore left-division (\ )
2. Fattorizzazione LU con pivoting parziale

CENNO TEORICO

La fattorizzazione LU con pivoting consente di decomporre una matrice A in:

```
P*A = L*U
```

dove:

- P è la matrice di permutazione
- L è triangolare inferiore con diagonale unitaria
- U è triangolare superiore

Per risolvere Ax = b si procede:

1. `PAx = Pb` $\Longrightarrow$ `LUx = Pb`
2. Risolve `Ly = Pb` (forward substitution)
3. Risolve `Ux = y` (backward substitution)

CODICE COMMENTATO

```
%Script main_linsys.m
%Questo script risolve un sistema lineare in più modi

% Definizione della matrice test usando def_mat
% def_mat(14,5) genera una matrice particolare (case 14, dimensione 5x5)
A = def_mat(14,5);

% Determina la dimensione della matrice
[n,m] = size(A);

% Visualizza la matrice A
disp('Matrice A:');
disp(A);

% Definisce il vettore soluzione esatta xs
% Scegliamo xs = [1,1,1,1,1]^T per semplicità
xs = ones(n,1);
disp('Vettore soluzione esatta xs:');
disp(xs);

% Calcola il termine noto b = A*xs
% In questo modo conosciamo già la soluzione esatta!
b = A*xs;

% ========================================================================
% METODO 1: Operatore left-division di MATLAB
% L'operatore \ risolve automaticamente il sistema usando algoritmi ottimali
% (MATLAB decide se usare LU, QR, o altri metodi in base alla struttura di A)
% ========================================================================
x1 = A\b;
disp('Soluzione con left-division x1:');
disp(x1);

% ========================================================================
% METODO 2: Fattorizzazione LU con pivoting parziale
% [L,U,P] = lu(A) calcola:
%   - L: matrice triangolare inferiore con diag(L) = 1
```

```
%   - U: matrice triangolare superiore
%   - P: matrice di permutazione tale che P*A = L*U
% ========================================================================
[L,U,P] = lu(A);

disp('Matrice L (triangolare inferiore):');
disp(L);

disp('Matrice U (triangolare superiore):');
disp(U);

disp('Matrice P (permutazione):');
disp(P);

% Risoluzione sistema triangolare inferiore: Ly = Pb
% lsolve implementa forward substitution
y = lsolve(L, P*b);

% Risoluzione sistema triangolare superiore: Ux = y
% usolve implementa backward substitution
x2 = usolve(U, y);

disp('Soluzione con fattorizzazione LU x2:');
disp(x2);

% VERIFICA: x1 e x2 dovrebbero essere praticamente identici
disp('Differenza tra i due metodi:');
disp(norm(x1 - x2));
```

APPROFONDIMENTO ESERCIZIO A.1

- La fattorizzazione LU con pivoting parziale garantisce stabilità numerica scegliendo ad ogni passo il pivot di valore massimo (in valore assoluto) nella colonna corrente.
- Il costo computazionale della fattorizzazione LU è $O(n^3/3)$, mentre la risoluzione dei due sistemi triangolari costa $O(n^2)$ ciascuno.
- L'operatore `\` di MATLAB è più conveniente ma "nasconde" i dettagli dell'algoritmo. La fattorizzazione LU esplicita è utile quando:
  - Si devono risolvere più sistemi con la stessa matrice A ma b diversi
  - Si vuole studiare le proprietà delle matrici L e U
- Le funzioni `lsolve` e `usolve` implementano rispettivamente:
  - **Forward substitution**: risolve Ly=b procedendo dalla prima riga
  - **Backward substitution**: risolve Ux=y procedendo dall'ultima riga

---

ESERCIZIO A.2

Stabilità della Fattorizzazione LU - main_lufact.m

OBIETTIVO

Verificare sperimentalmente le proprietà di stabilità della fattorizzazione LU con pivoting parziale, controllando che:

- $\max|\ell_{ij}| \leq 1$
- $\max|u_{ij}| \leq 2^{(n-1)} \times \max|a_{ij}|$

CENNO TEORICO

La fattorizzazione LU con pivoting parziale garantisce che tutti gli elementi di L siano limitati in modulo da 1 (grazie alla scelta del pivot massimo). Per U esiste un bound teorico esponenziale 2^(n-1), ma in pratica la crescita è molto più moderata per matrici "tipiche".

CODICE COMMENTATO (Fattorizzazione LU)

```
%main_lufact.m - Analisi stabilità fattorizzazione LU

% Intestazione tabella risultati
fprintf('mat_k  nxn       max|L_ij|    max|u_ij|    2^(n-1)max|a_ij|\n');

% Test su diverse dimensioni: 5x5, 10x10, 50x50
nn = [5, 10, 50];

for n = nn
    % Test su diverse matrici test (case 11-14 di def_mat)
    for k = 11:14
        % Genera matrice test
        A = def_mat(k, n);

```

```
        % Fattorizza la matrice A con pivoting parziale
        % P*A = L*U
        [L, U, P] = lu(A);

        % Calcola il massimo elemento di L in valore assoluto
        % max(max(...)) trova il massimo su tutta la matrice
        maxl = max(max(abs(L)));

        % Calcola il massimo elemento di U in valore assoluto
        maxu = max(max(abs(U)));

        % Calcola il bound teorico per U: 2^(n-1) * max|a_ij|
        bound = 2^(n-1) * max(max(abs(A)));

        % Stampa i risultati in formato tabellare
        fprintf('%5d %2dx%2d  %14.5f  %14.5e  %14.5e\n', ...
                k, n, n, maxl, maxu, bound);
    end
end
```

CODICE COMMENTATO (Fattorizzazione QR)

```
%main_qrfact.m - Analisi stabilità fattorizzazione QR

% La fattorizzazione QR decompone A = Q*R dove:
%   Q: matrice ortogonale (Q^T * Q = I)
%   R: matrice triangolare superiore

fprintf('mat_k  nxn        max|q_ij|     max|r_ij|    sqrt(n)*max|a_ij|\n');

nn = [5, 10, 50];

for n = nn
    for k = 11:14
        A = def_mat(k, n);

        % Fattorizza la matrice A con algoritmo QR
        [Q, R] = qr(A);

        % Massimo elemento di Q in valore assoluto
        % Teoricamente dovrebbe essere ≤ 1 (Q è ortogonale)
        maxq = max(max(abs(Q)));

        % Massimo elemento di R in valore assoluto
        maxr = max(max(abs(R)));

        % Bound teorico per R: sqrt(n) * max|a_ij|
        bound = sqrt(n) * max(max(abs(A)));

        fprintf('%5d %2dx%2d  %14.5f  %14.5e  %14.5e\n', ...
                k, n, n, maxq, maxr, bound);
    end
end
```

APPROFONDIMENTO ESERCIZIO A.2

FATTORIZZAZIONE LU con pivoting

- $\max|\ell_{ij}| \leq 1$ è sempre rispettato grazie alla normalizzazione
- Il bound 2^(n-1) per U è pessimistico: raramente si avvicina a questo valore
- Matrici "patologiche" possono causare crescita esponenziale (growth factor)

FATTORIZZAZIONE QR

- Più stabile numericamente della LU (non richiede pivoting)
- Q ortogonale $\Longrightarrow \max|q_{ij}| \leq 1$
- R ha bound $\sqrt{n}$ più favorevole rispetto al 2^(n-1) della LU
- Costo computazionale più alto: $O(2n^3/3)$ vs $O(n^3/3)$ della LU

Le matrici test (case 11-14)

- **Case 11**: matrice tipo Hilbert (mal condizionata)
- **Case 12**: matrice simmetrica particolare
- **Case 13**: matrice con struttura min(i,j)
- **Case 14**: matrice con -1 sotto la diagonale e 1 nell'ultima colonna

ESERCIZIO A.3

Analisi del Condizionamento - main_ilcond.m

OBIETTIVO

Studiare il condizionamento di una matrice (matrice di Hilbert) e verificare sperimentalmente che il numero di condizionamento fornisce un bound sull'errore relativo nella soluzione.

CENNO TEORICO

Il numero di condizionamento K(A) misura quanto la soluzione x di un sistema Ax=b è sensibile a perturbazioni nel termine noto b o nella matrice A:

```
||δx|| / ||x|| ≤ K(A) × ||δb|| / ||b||
```

dove:

- K(A) = ||A|| × ||A⁻¹|| (numero di condizionamento in norma ρ)
- δb è la perturbazione applicata a b
- δx è l'errore nella soluzione x

Per matrici mal condizionate (K >> 1), piccole perturbazioni nei dati possono causare grandi errori nella soluzione.

CODICE COMMENTATO

```
%main_ilcond.m - Studio del condizionamento

% ========================================================================
% PARTE 1: Calcolo numero di condizionamento della matrice di Hilbert
% ========================================================================

% Test su diverse dimensioni n=2,3,...,10
for n = 2:10
    % Genera matrice di Hilbert: H(i,j) = 1/(i+j-1)
    H = hilb(n);

    % Calcola numero di condizionamento in norma infinito
    % cond(H,Inf) = ||H||∞ * ||H⁻¹||∞
    cH = cond(H, Inf);

    % Stampa dimensione e condizionamento
    fprintf('n= %3d  Cond(H_n)= %15.7e\n', n, cH);
end

% OSSERVAZIONE: il condizionamento cresce esponenzialmente con n!
% Per n=10, K ≈ 10^13, rendendo il sistema quasi insolubile con precisione standard

disp(' ');

% ========================================================================
% PARTE 2: Verifica sperimentale del bound teorico
% Fissiamo n=5 e perturbiamo b con intensità crescente
% ========================================================================
n = 5;
H = hilb(n);
cH = cond(H, Inf);

% Soluzione esatta (per costruzione)
x = ones(n, 1);

% Calcola termine noto b = H*x
b = H * x;

fprintf('Condizionamento della matrice di Hilbert 5x5: K(H)= %15.7e\n', cH);
fprintf('ρ   Kρ sperimentale      K teorico\n');

% Esperimento: aggiunge perturbazioni di diversa entità a b
% e misura il condizionamento effettivo (sperimentale)
% ========================================================================
for ρ = 1:5
    % Genera perturbazione δb = 10^(-ρ) * vettore_casuale
    % ρ=1 => perturbazione O(10^-1)
    % ρ=5 => perturbazione O(10^-5)
    db = 10^(-ρ) * rand(n,1);

    % Risolve il sistema perturbato H*x̃ = b + δb
```

```
    dy = H\(b + db);

    % Calcola il numero di condizione sperimentale usando la formula:
    % Kp = (||x̃ - x|| / ||x||) / (||δb|| / ||b||)
    % Uso della norma 1 (norma-infinito darebbe risultati simili)
    cond_sper = norm(dy - x, 1) / norm(x, 1) * norm(b, 1) / norm(db, 1);

    % Stampa confronto tra Kp sperimentale e K teorico
    fprintf('p=%2d  Kp= %15.7e K= %15.7e\n', p, cond_sper, cH);
end

disp(' ');
disp('NOTA: Kp ≤ cond(H) per ogni p (come previsto dalla teoria)');
```

APPROFONDIMENTO ESERCIZIO A.3

LA MATRICE DI HILBERT

- H(i,j) = 1/(i+j-1) per i,j = 1,...,n
- È simmetrica e definita positiva
- Ha un condizionamento che cresce esponenzialmente: $K(H_n) \approx \exp(3.5n)$
- Esempio: $K(H_5) \approx 10^5$, $K(H_{10}) \approx 10^{13}$, $K(H_{15}) \approx 10^{18}$

INTERPRETAZIONE DEI RISULTATI

- K_p dovrebbe essere sempre ≤ cond(H) (disuguaglianza teorica)
- K_p dipende dalla particolare perturbazione casuale scelta
- All'aumentare di p (perturbazioni più piccole), K_p può oscillare ma resta inferiore al bound teorico
- In pratica, per matrici mal condizionate come Hilbert, anche piccole perturbazioni (rumore numerico) possono causare grandi errori nella soluzione

CONSEGUENZE PRATICHE

- Sistemi con K >> 1 sono difficili da risolvere con precisione
- Se K ≈ 10^k, si perdono circa k cifre significative
- La fattorizzazione LU o QR non può "migliorare" il condizionamento
- Per matrici mal condizionate servono tecniche di regolarizzazione

---

FUNZIONI DI SUPPORTO

FUNZIONE: lsolve.m (Forward Substitution)

```
function [y] = lsolve(L, b)
%lsolve - Risolve sistema triangolare inferiore Ly = b
%
%Input:
%  L : matrice triangolare inferiore n×n
%  b : termine noto n×1
%Output:
%  y : soluzione del sistema n×1
%
%Algoritmo: Forward substitution
%  y(1) = b(1) / L(1,1)
%  y(i) = (b(i) - Σ_{j=1}^{i-1} L(i,j)*y(j)) / L(i,i)  per i=2,...,n
%
%Costo computazionale: O(n²) operazioni

y = b;

% Risolve la prima equazione: L(1,1)*y(1) = b(1)
y(1) = y(1) / L(1,1);

n = length(y);

% Procede dalla seconda riga in avanti (forward)
for i = 2:n
    % Sottrae il contributo delle variabili già calcolate y(1),...,y(i-1)
    for j = 1:i-1
        y(i) = y(i) - L(i,j) * y(j);
    end
    % Divide per l'elemento diagonale L(i,i)
    y(i) = y(i) / L(i,i);
end

end
```

FUNZIONE: usolve.m (Backward Substitution)

```
function [x] = usolve(U, y)
%usolve - Risolve sistema triangolare superiore Ux = y
%
%Input:
%  U : matrice triangolare superiore n×n
%  y : termine noto n×1
%Output:
%  x : soluzione del sistema n×1
%
%Algoritmo: Backward substitution
%  x(n) = y(n) / U(n,n)
%  x(i) = (y(i) - Σ_{j=i+1}^{n} U(i,j)*x(j)) / U(i,i)  per i=n-1,...,1
%
%Costo computazionale: O(n²) operazioni

x = y;
n = length(x);

% Procede dall'ultima riga verso la prima (backward)
for i = n:-1:2
    % Divide per l'elemento diagonale U(i,i)
    x(i) = x(i) / U(i,i);

    % Aggiorna le righe precedenti sottraendo U(j,i)*x(i)
    % (eliminazione di x(i) dalle equazioni j=1,...,i-1)
    for j = 1:i-1
        x(j) = x(j) - U(j,i) * x(i);
    end
end

% Risolve la prima equazione: U(1,1)*x(1) = y(1) - (termini già sottratti)
x(1) = x(1) / U(1,1);

end
```

FUNZIONE: def_mat.m (Matrici Test)

```
function A = def_mat(imat, n)
%def_mat - Genera matrici test per sistemi lineari
%
%Input:
%  imat : indice che identifica la matrice test (1-14)
%  n    : dimensione della matrice (per case 11-14), default n=10
%Output:
%  A    : matrice test
%
%Matrici disponibili:
%  Case 1-10: matrici predefinite di piccole dimensioni (per test manuali)
%  Case 11: tipo Hilbert modificata (mal condizionata)
%  Case 12: matrice simmetrica con A(i,j) = n-1-max(i,j)
%  Case 13: matrice con A(i,j) = min(i,j)
%  Case 14: matrice particolare con -1 sottodiagonale e 1 ultima colonna

if (nargin < 2)
    n = 10;
end

switch imat
    % ... (case 1-10 omessi per brevità) ...

    case 11
        % Matrice tipo Hilbert: A(i,j) = 1/(i+j+1)
        % Matrice mal condizionata, simile alla matrice di Hilbert
        for j = 1:n
            A(1,j) = 1.0;
        end
        for i = 1:n
            for j = 1:n
                A(i,j) = 1.0 / (i + j + 1.0);
            end
        end

    case 12
        % Matrice simmetrica con pattern particolare
        for i = 1:n
            for j = 1:i
```

```
            A(i,j) = n - 1 - i;
            A(j,i) = A(i,j);  % simmetria
        end
    end

    case 13
        % Matrice con A(i,j) = min(i,j)
        % Matrice simmetrica, definita positiva
        for i = 1:n
            for j = 1:n
                if (i >= j)
                    A(i,j) = i;
                else
                    A(i,j) = j;
                end
            end
        end

    case 14
        % Matrice con struttura particolare:
        % Diagonale = 1, sottodiagonale = -1, ultima colonna = 1
        A = eye(n);
        A(1,n) = 1;
        for i = 2:n
            A(i,n) = 1;
            for j = 1:i-1
                A(i,j) = -1;
            end
        end
end
end
```

NOTE FINALI

- Gli esercizi A sono focalizzati sulla comprensione pratica della risoluzione di sistemi lineari tramite fattorizzazione LU e sull'analisi della stabilità numerica e del condizionamento.
- Le fattorizzazioni LU e QR sono alla base di molti algoritmi numerici:
  - Risoluzione sistemi lineari
  - Calcolo determinante (det(A) = det(L)×det(U) = det(U))
  - Calcolo inversa (risoluzione di n sistemi)
  - Metodo dei minimi quadrati (QR)
- Il pivoting parziale è essenziale per la stabilità della fattorizzazione LU, mentre la QR è intrinsecamente stabile anche senza pivoting.
- Il condizionamento è una proprietà intrinseca della matrice A e non può essere migliorato dagli algoritmi numerici: dipende solo da A e dalla norma scelta.

FINE DOCUMENTO