Seminar 2

Rezolvarea sistemelor liniare determinate

Acest seminar este dedicat metodelor numerice de rezolvare a sistemelor liniare determinate, i.e. al sistemelor liniare cu numărul de ecuaţii egal cu numărul necunoscutelor. Această problemă apare frecvent în probleme mai complexe iar o rezolvare eficientă şi precisă a sistemelor liniare determinate contribuie esenţial la eficienţa şi stabilitatea unor algoritmi mai complecşi.

2.1 Preliminarii

Fie un sistem liniar de n ecuaţii cu n necunoscute. Un astfel de sistem poate fi scris concis în forma:

$$Ax = b, \qquad A \in \mathbb{R}^{n\times n}, \quad b \in \mathbb{R}^n, \quad x \in \mathbb{R}^n.$$

Problema este de a calcula vectorul x al necunoscutelor atunci când A şi b sunt date. Problema are o soluţie unică doar dacă matricea A este nesingulară, i.e. are o inversă A^{-1}. În acest caz:

$$x = A^{-1}b.$$

Pentru detalii despre teoria sistemelor liniare vezi cursul şi bibliografia recomandată. Aici ne vom concentra asupra unor sfaturi referitoare la metodele numerice pentru rezolvarea problemei de mai sus. De asemenea, vom arăta cum se rezolvă cu ajutorul calculatorului probleme înrudite, cum ar fi calculul determinantului sau al inversei unei matrice.

1. Nu folosiţi "metode binecunoscute" cum ar fi Cramer sau formula $x = A^{-1}b$. Există metode mai bune, i.e. metode numerice mai precise şi mai eficiente.

2. Cea mai bună metodă de rezolvare a unui sistem liniar triunghiular este substituţia numerică. Dacă A este o matrice *inferior* triunghiulară nesingulară, atunci vom folosi metoda *substituţiei înainte*. Dacă A este o matrice *superior* triunghiulară nesingulară, atunci vom folosi metoda *substituţiei înapoi*.

3. Cea mai bună metodă de rezolvare a unui sistem general liniar determinat este metoda "reducerii şi substituţiei", i.e. în prima fază sistemul este redus la unul sau două sisteme triunghiulare iar în a două fază sistemul(le) triunghiular(e) este(sunt) rezolvat(e) prin substituţie.

4. Cea mai populară schemă de reducere este *eliminarea gaussiană cu pivotare parţială* (algoritmul GPP). Cea mai bună este *eliminarea gaussiană cu pivotare completă* (algoritmul GPC). Nu uitaţi că o strategie de pivotare este necesară totdeauna. Fară pivotare este posibil ca eliminarea gaussiană să nu poată reduce o matrice dată la o formă triunghiulară.

5. Eliminarea gaussiană este echivalentă cu metoda factorizării LU. Metoda factorizării LU trebuie să fie utilizată împreună cu o strategie de pivotare adecvată.

6. Aşa numita metodă Gauss-Jordan, care reduce un sistem dat la unul diagonal este cu aproximativ 30% mai puţin eficientă decât eliminarea gaussiană. Aşadar eliminarea gaussiană este mai bună.

7. Cea mai bună metodă de rezolvare a unui sistem liniar simetric pozitiv definit este folosirea factorizării Cholesky.

8. Nu calculaţi determinanţi şi inversele matricelor fară o cerinţă explicită. Asemenea expresii ca $\alpha = c^T A^{-1} b$ pot fi calculate fară calculul lui A^{-1}.

2.2 Probleme rezolvate

2.2.1 Sisteme triunghiulare şi probleme înrudite

Problema 2.1 Fie două matrice nesingulare superior bidiagonalale $B, C \in \mathbf{R}^{n\times n}$ ($b_{ij} = c_{ij} = 0$ pentru $i > j$ sau $i < j - 1$), un vector $d \in \mathbf{R}^n$ şi matricea $A = BC$. Daţi soluţii eficiente pentru:

a. rezolvarea sistemului liniar $Ax = d$;

b. calculul determinantului $\delta = \det A$;

c. calculul inversei $X = A^{-1}$.

Soluţie a. Vom analiza două scheme de calcul. a1) Prima este:

1. Se calculează $A = BC$
2. Se rezolvă $Ax = d$.

Pentru a fi eficientă acestă schemă trebuie să exploateze faptul că matricea A are o structură de matrice superior triunghiulară bandă de laţime 3, i.e. $a_{ij} = 0$ pentu $i > j$ şi $j > i + 2$ (demonstraţi!), în calcularea lui A ca şi în adaptarea metodei substituţiei înapoi pentru rezolvarea sistemului superior triunghiular $Ax = d$. Algoritmul este:

1. **pentru** $i = 1 : n$
    1. **pentru** $j = 1 : n$
        1. $a_{i,j} = 0$

2. $a_{i,i} = b_{i,i}c_{i,i}$
3. **dacă** $i < n$
   1. $a_{i,i+1} = b_{i,i}c_{i,i+1} + b_{i,i+1}c_{i+1,i+1}$
4. **dacă** $i < n-1$
   1. $a_{i,i+2} = b_{i,i+1}c_{i+1,i+2}$

2. $x_n \leftarrow \frac{d_n}{a_{n,n}}$
3. $x_{n-1} \leftarrow \frac{(d_{n-1}-a_{n-1,n}x_n)}{a_{n-1,n-1}}$
4. **pentru** $i = n-2 : -1 : 1$
   1. $x_i \leftarrow \frac{(d_i-a_{i,i+1}x_{i+1}-a_{i,i+2}x_{i+2})}{a_{i,i}}$

Efortul de calcul este de $N_{fl} \approx 10n$ flopi.

a2) Schema a doua evită calcularea explicită a matricei A:

1. Se rezolvă sistemul superior bidiagonal $By = d$
2. Se rezolvă sistemul superior bidiagonal $Cx = y$.

Adaptând metoda substituţiei înapoi pentru rezolvarea sistemelor superior bidiagonale rezultă algoritmul:

1. $y_n \leftarrow \frac{d_n}{b_{n,n}}$
2. **pentru** $i = n-1 : -1 : 1$
   1. $y_i \leftarrow \frac{(d_i-b_{i,i+1}y_{i+1})}{b_{i,i}}$
3. $x_n \leftarrow \frac{y_n}{c_{nn}}$
4. **pentru** $i = n-1 : -1 : 1$
   1. $x_i \leftarrow \frac{(y_i-c_{i,i+1}x_{i+1})}{c_{i,i}}$

Efortul de calclul in acest caz este de $N_{op} \approx 6n$ flopi, aşadar al doilea algoritm este aproape de două ori mai eficient decât primul.

b. Avem $\delta = \det A = \det B * C = \det B * \det C = \Pi_{i=1}^{n} b_{ii}c_{ii}$. Aşadar algoritmul este:

1. $\delta = 1$
2. **pentru** $i = 1 : n$
   1. $\delta = \delta * b_{i,i}c_{i,i}$

c. Vom prezenta doi algoritmi alternativi.

c1. Primul algoritm se bazează pe relaţia $X = A^{-1} = C^{-1}B^{-1}$ şi are următoarea schemă de calcul:

1. Se calculează $Y = B^{-1}$
2. Se calculează $Z = C^{-1}$
3. Se calculează $X = ZY$.

Aşadar, trebuie să stim să calculăm inversa unei matrice superior bidiagonale. Pentru calcularea ei, vom proceda ca şi în cazul superior triunghiular (vezi cursul), i.e. vom rezolva ecuaţia matriceală $BY = I_n$, care, partiţionată pe coloane, devine $By_j = e_j$, $j = 1 : n$, aici $y_j = Y(:, j)$. Bineînţeles, Y este o matrice superior triunghiulară şi, în concluzie,

$y_j(j+1:n) = 0$. Este uşor de văzut ca primele j elemente ale lui y_j pot fi calculate, în ordine inversă, cu ajutorul relaţiilor $y_{jj} = \frac{1}{b_{jj}}$, $y_{ij} = \frac{-b_{i,i+1}y_{i+1,j}}{b_{ii}}$, $i = j-1:-1:1$. Dacă coloanele lui Y sunt calculate în ordine inversă, atunci Y poate suprascrie B în timpul calculului. În concluzie, algoritmul este:

1. **pentru** $j = n:-1:1$
   1. $b_{j,j} \leftarrow y_{j,j} = \frac{1}{b_{j,j}}$
   2. **pentru** $i = j-1:-1:1$
      1. $b_{ij} \leftarrow y_{ij} = \frac{-b_{i,i+1}y_{i+1,j}}{b_{ii}}$

Numărul flopilor necesari pentru a inversa o matrice bidiagonală este: $N_{fl} = \sum_{j=1}^{n}(1 + 2(j-1)) \approx n^2$.

Pentru a multiplica cele două matrice superior triunghiulare Z and Y vom folosi formula economică $x_{ij} = \sum_{k=i}^{j} z_{ik}y_{kj}$ şi faptul că produsul este o matrice superior triunghiulară.

Introducând o sintaxă de forma $B \leftarrow Y = \text{UBINV}(B)$ pentru calculul inversei unei maricte superior bidiagonale cu suprascriere, primul algoritm de inversare devine:

1. $B \leftarrow Y = \text{UBINV}(B)$
2. $C \leftarrow Z = \text{UBINV}(C)$
3. **pentru** $i = 1:n$
   1. **pentru** $j = 1:n$
      1. $x_{ij} = 0$
   2. **pentru** $j = i:n$
      1. **pentru** $k = i:j$
      2. $x_{ij} = x_{ij} + c_{ik}b_{kj}$

Efortul de calcul este de $N_{fl}^{(1)} \approx 2n^2 + n^3/6 \approx n^3/6$ flopi.

c2. Al doilea algoritm se bazează pe rezolvarea ecuaţiei matriceale $BCX = I_n$ folosind schema de calcul:

1. Se rezolvă ecuaţia matriceală $BY = I_n$
2. Se rezolvă ecuaţia matriceală $CX = Y$

Prima instrucţiune este echivalentă cu inversarea lui B şi are nevoie de n^2 flopi. A doua instrucţiune va fi efectuată pe coloanele, i.e. vom rezolva sistemele $CX(:,j) = Y(:,j)$, şi ţinând cont că $Y(j+1:n,j) = 0$ vom avea $X(j+1:n,j) = 0$ (i.e. se ştie faptul că X este superior triunghiulară), aşadar trebuie să rezolvăm doar sistemele bidiagonale $C(1:j,1:j)X(1:j,j) = Y(1:j,j)$, $j = 1:n$. Al doilea algoritm este:

1. $B \leftarrow Y = \text{UBINV}(B)$
2. **pentru** $j = n:-1:1$
   1. **pentru** $i = j+1:n$
      1. $x_{ij} = 0$
   2. $x_{jj} = \frac{b_{jj}}{c_{jj}}$
   3. **pentru** $i = j-1:-1:1$
      1. $x_{ij} = \frac{(b_{ij} - c_{i,i+1}x_{i+1,j})}{c_{ii}}$

Al doilea algoritm necesită $N_{fl}^{(2)} \approx n^2 + \frac{3n^2}{2} = \frac{5n^2}{2}$ flopi. În concluzie, al doilea algoritm este net mai bun decât primul: de exemplu, dacă $n = 3000$ atunci $N_{fl}^{(1)} \approx 4.5 \cdot 10^9$ flopi şi $N_{fl}^{(2)} \approx 22.5 \cdot 10^6$ flopi; aşadar, în acest caz, primul algoritm necesită un timp de execuţie de 200 de ori mai mare decât al doilea.

Observaţie. Compararea celor doi algoritmi precedenţi arată că este mai avantajoasă rezolvarea sistemelor liniare decât calcularea inversei unei matrice. Această recomandare este valabilă totdeauna, după cum vom vedea în probleme viitoare.

Problema 2.2 Presupunem că matricea nesingulară $A \in \mathbf{R}^{n\times n}$ are o factorizare LU şi că L, U sunt cunoscute. Scrieţi un algoritm care să calculeze elementul (i, j) al matricei A^{-1} in aproximativ $(n-j)^2 + (n-i)^2$ flopi.

Soluţie. Matricea A fiind nesingulară, aşa vor fi şi matricele L, U şi

$$A^{-1} = (LU)^{-1} = U^{-1}L^{-1}.$$

În concluzie

$$A^{-1}(i, j) = e_i^T A^{-1} e_j = e_i^T U^{-1} L^{-1} e_j = x^T y,$$

aici $x^T = e_i^T U^{-1} = U^{-1}(i, :)$ şi $y = L^{-1}e_j = L^{-1}(: .j)$. Deoarece matricele L^{-1} şi U^{-1} sunt inferior şi, respectiv, superior truiunghiulare, avem $x(1 : i-1) = 0$ şi $y(1 : j-1) = 0$. Pentru a calcula vectorul $\tilde{x} = x(i : n)$ şi $\tilde{y} = y(j : n)$ este suficient să rezolvăm sistemele liniare:

$$\begin{cases} x^T(i : n)U(i : n, i : n) = [\,1\ 0\ \ldots\ 0\,] \\ \\ L(j : n, j : n)y(j : n) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \end{cases}$$

Deci, $A^{-1}(i, j) = x^T y = \sum_{k=\max(i,j)}^{n} x_k y_k$. Primul sistem liniar poate fi scris în următoarea formă:

$$U^T(i : n, i : n)x(i : n) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

aşadar trebuie să rezolvăm două sisteme inferior tiunghiulare de grad $n-i$ şi $n-j$, respectiv un produs scalar a doi vectori. Prin urmare numărul de flopi necesari sunt: $N_{fl} = (n-i)^2 + (n-j)^2 + 2(n - \max(i, j)) \approx (n-i)^2 + (n-j)^2$.

2.2.2 Eliminarea gaussiană şi probleme înrudite

Problema 2.3 Fie $H \in \mathbf{R}^{n\times n}$ o matrice superior Hessenberg nesingulară ($h_{ij} = 0$, pentru $i > j + 1$).

a. Dacă toate submatricele lider principale ale lui H sunt nesingulare, adaptaţi algoritmul de eliminare gaussiană pentru rezolvarea sistemului liniar $Hx = b$, unde $b \in \mathbf{R}^n$ este un vector; calculaţi numărul de operaţii.

b. Adaptaţi algoritmii GPP şi LSS_GPP pentru aceeaşi problemă.

c. Adaptaţi algoritmul Crout pentru calculul factorizării LU al matricei H.

Soluţie. Matricele Hessenberg apar în probleme mai complexe cum ar fi calcularea valorilor proprii şi a vectorilor proprii.

a. Este evident că multiplicatorii μ_{ij} vor fi zero pentru $i > j + 1$. Atunci, eliminarea gaussiană pentru o matrice Hessenberg va avea forma:

> Algoritmul G_HESS. Dată o matrice superioară Hessenberg $H \in \mathbf{R}^{n\times n}$ cu $H(1:k, 1:k)$ nesingulare pentru $k = 1:n-1$, algoritmul efectuează eliminarea gaussiană fară pivotare. Multiplicatorii gaussieni şi matricea superior triunghiulară rezultată vor suprascrie matricea H.

1. **pentru** $k = 1:n-1$
   1. $h_{k+1,k} \leftarrow \mu_{k+1,k} = \frac{h_{k+1,k}}{h_{kk}}$
   2. **pentru** $j = k+1:n$
      1. $h_{k+1,j} \leftarrow h_{k+1,j} - \mu_{k+1,k} h_{kj}$

Acest algoritm necesită $N_{fl} = \sum_{k=1}^{n-1}(1 + 2(n-k)) \approx n^2$ flopi. Vectorul b va fi modificat în funcţie de valorile particulare ale multiplicatorilor:

1. **pentu** $k = 1:n-1$
   1. $b_{k+1} \leftarrow b_{k+1} - h_{k+1,k} b_k$

executând $\approx 2n$ flopi. Atunci, un sistem superior triunghiular trebuie rezolvat (cu un plus de n^2 flopi).

b. Pivotarea parţială nu alternează structura superioar Hessenberg a matricei (dar pivotarea completă da). La fiecare pas k, pentru a determina pivotul trebuie să comparăm doar $|h_{kk}|$ şi $|h_{k+1,k}|$. Algoritmul de triunghiularizare prin eliminare gaussiană cu pivotare parţială a unei matrice superior Hessenberg este:

> Algoritmul GPP_HESS. Dată o matrice superior Hessenberg $H \in \mathbf{R}^{n\times n}$, algoritmul efectuează eliminarea gaussiană cu pivotare parţială. Multiplicatorii gaussieni şi matricea superior triunghiulară rezultată vor suprascrie matricea H. Permutările sunt stocate în vectorul p.

1. **pentru** $k = 1:n-1$
   1. $p(k) = k$
   2. **dacă** $|h_{k+1,k}| > |h_{kk}|$
      1. $p(k) = k+1$
      2. **pentru** $j = k:n$
         1. $h_{kj} \leftrightarrow h_{k+1,j}$
   3. $h_{k+1,k} \leftarrow \mu_{k+1,k} = \frac{h_{k+1,k}}{h_{kk}}$
   4. **pentru** $j = k+1:n$
      1. $h_{k+1,j} \leftarrow h_{k+1,j} - \mu_{k+1,k} h_{kj}$

Introducând sintaxa $[H, p] \leftarrow GPP\_HESS(H)$ pentru apelul algoritmului de mai sus csi folosind funcţiile MATLAB $TRIU$ pentru a extrage partea superior triunghiulară dintr-o matrice, algoritmul pentru rezolvarea unui sistem liniar Hessenberg devine:

Algoritmul LSS_GPP_HESS. Fiind dată o matrice superior Hessenberg nesingulară $H \in \mathbf{R}^{n\times n}$ şi un vector $b \in \mathbf{R}^n$, algoritmul calculează soluţia $x \in \mathbf{R}^n$ a sistemului $Hx = b$ folosind eliminarea gaussiană cu pivotare parţială.

1. $[H, p] \leftarrow GPP\_HESS(H)$
2. **pentru** $k = 1 : n-1$
   1. **dacă** $p(k) = k+1$
      1. $b_k \leftrightarrow b_{k+1}$
   2. $b_{k+1} \leftarrow b_{k+1} - h_{k+1,k}b_k$
3. $x = UTRIS(TRIU(H), b)$

Acest algoritm necesită $N_{fl} \approx 2n^2$ flopi (în comparaţie cu $\approx \frac{2n^3}{3}$ flopi necesari algoritmului fundamental LSS_GPP).

c. Presupunem că toate submatricele lider principale ale lui H sunt nesingulare. Algoritmul G_HESS calculează factorizarea LU Doolittle a matricei H. Observăm că L este inferior bidiagonală. Desigur, în factorizarea Crout LU matricea L va fi deasemeni inferior bidiagonală. Ţinând cont de acest lucru şi de faptul că termenii diagonali ai lui U sunt egali cu 1, din identitatea $H = LU$ va rezulta:

pasul 1. $l_{11} = h_{11}$, $l_{21} = h_{21}$ şi din $h_{1j} = l_{11}u_{1j}$ avem $u_{1j} = \dfrac{h_{1j}}{l_{11}}$, $j = 2 : n$.

pasul k. Presupunând ca am calculat primele $k-1$ coloane ale lui L şi primele $k-1$ linii ale lui U, din $h_{kk} = l_{k,k-1}u_{k-1,k} + l_{kk}$ şi $h_{k+1,k} = l_{k+1,k}$ avem $l_{kk} = h_{k,k} - l_{k,k-1}u_{k-1,k}$ şi $l_{k+1,k} = h_{k+1,k}$. Deasemeni, din egalitatea $h_{kj} = l_{k,k-1}u_{k-1,j} + l_{kk}u_{kj}$ vom obţine $u_{kj} = \dfrac{h_{kj} - l_{k,k-1}u_{k-1,j}}{l_{kk}}$, $j = k+1 : n$.

În concluzie algoritmul Crout pentru o matrice superior Hessenberg este:

Algoritmul $CROUT\_HESS$. Dată o matrice superior Hessenberg $H \in \mathbf{R}^{n\times n}$ cu $H(1:k, 1:k)$ nesingulare pentru $k = 1 : n-1$, algoritmul calculează o matrice inferior bidiagonală L şi o matrice superior triunghiulară unitate U astfel încât $H = LU$. Matricea L suprascrie triunghiul inferioar a lui H iar partea strict superioară a lui U suprascrie triunghiul strict superioar a lui H.

1. **pentru** $j = 2 : n-1$
   1. $h_{1j} \leftarrow u_{1j} = \dfrac{h_{1j}}{h_{11}}$
2. **pentru** $k = 2 : n$
   1. $h_{kk} \leftarrow l_{kk} = h_{k,k} - h_{k,k-1}h_{k-1,k}$
   2. **dacă** $k = n$
      1. RETURN
   3. **pentru** $j = k+1 : n$
      1. $h_{kj} \leftarrow u_{kj} = \dfrac{h_{kj} - h_{k,k-1}h_{k-1,j}}{h_{kk}}$

Problema 2.4 Elaboraţi un algoritm pentru rezolvarea ecuaţiei matriceale $AX = B$, unde $A \in \mathbf{R}^{n\times n}$ este nesingulară şi $B \in \mathbf{R}^{n\times p}$.

Soluţie. Ecuaţia matriceală $AX = B$ este formată din p sisteme liniare: $Ax_j = b_j$, pentru $j = 1 : p$, unde x_j şi b_j sunt notaţii pentru coloana j din X respectiv B, i.e. $x_j = X(:, j)$, $b_j = B(:, j)$. Folosirea algoritmului:

1. **pentru** $j = 1 : p$
   1. Se rezolvă $Ax_j = b_j$ folosind *LSS_GPP*

nu este o idee bună, întrucât numărul de operaţii ar fi $\frac{2pn^3}{3}$. În schimb, GPP poate fi folosit doar o dată pentru a triunghiulariza A, ca în următorul algoritm:

1. $[M, U, p] = GPP(A)$
2. **pentru** $j = 1 : p$
   1. **pentru** $k = 1 : n - 1$
      1. $b_{kj} \leftrightarrow b_{p(k),j}$
      2. **pentru** $i = k + 1 : n$
         1. $b_{ij} \leftarrow b_{ij} - u_{ik} b_{kj}$
   2. $x_j = UTRIS(U, b_j)$

Numărul operaţiilor este $\frac{2n^3}{3} + O(pn^2)$ flopi.

Problema 2.5 Propuneţi un algoritm care să rezolve sistemul liniar $Ax = f$, unde $A \in \mathbf{C}^{n\times n}$ este nesingulară şi $f \in \mathbf{C}^n$, folosind numai aritmetica reală.

Soluţie. Dacă notăm $A = B + iC$, $f = g + ih$, $x = y + iz$, aici $B, C \in \mathbf{R}^{n\times n}$ şi $g, h, y, z \in \mathbf{R}^n$, sistemul $Ax = f$ poate fi scris sub forma:

$$\begin{cases} By - Cz = g, \\ Cy + Bz = h, \end{cases}$$

sau

$$Du = e, \quad \text{cu} \quad D = \begin{bmatrix} B & -C \\ C & B \end{bmatrix} \in \mathbf{R}^{2n\times 2n}, \quad e = \begin{bmatrix} g \\ h \end{bmatrix} \in \mathbf{R}^{2n}, \quad u = \begin{bmatrix} y \\ z \end{bmatrix} \in \mathbf{R}^{2n}.$$

În concluzie trebuie să rezolvăm un sistem liniar de $2n$ ecuaţii de $2n$ necunoscute, e.g. folosind algoritmul GPP care necesită $\approx \frac{(2n)^3}{3} = \frac{8n^3}{3}$ flopi.

Problema 2.6 Descrieţi o variantă de eliminare gaussiană care introduce zerouri deasupra diagonalei principale, în ordinea $n : -1 : 2$, şi care produce factorizarea $A = UL$, unde U este superior triunghiulară unitate şi L este inferior triunghiulară.

Soluţie. Definim o matrice superior triunghiulară elementară (STE) de ordin n şi indice k astfel:

$$N_k = I_n - n_k e_k^T, \quad n_k = \begin{bmatrix} \nu_{1k} \\ \vdots \\ \nu_{k-1,k} \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Această tip de a matrice poate fi folosit pentru a introduce zerouri pe primele $k-1$ poziţii ale unui vector dat x cu $x_k \neq 0$. Întradevăr, $(N_k x)(i) = x(i) - \nu_{ik} x_k$, $i = 1 : k-1$ şi, deci, dacă $\nu_{ik} = x_i/x_k$, atunci $(N_k x)(i) = 0$, $i = 1 : k-1$. Bineînteles $(N_k x)(i) = x_i$, $i = k : n$. Dacă $x_k = 0$, atunci $N_k x = x$ pentru orice matrice STE N_k.

Este uşor de văzut că dacă $A(k : n, k : n)$ este nesingulară pentru toţi $k = n : -1 : 1$, atunci există matricele superior triunghiulare elementare N_k astfel încât matricea

$$L = N_2 N_3 \cdots N_{n-1} N_n A$$

este inferior triunghiulară. Schema de calcul este,

1. **pentru** $k = n : -1 : 2$
   1. Se calculează matricea STE N_k astfel încât $(N_k A)(1 : k-1, k) = 0$
   2. $A \leftarrow N_k A$

Ordinea inversă de aplicare a transformărilor elementare este esenţială pentru a nu modifica zerourile create la paşii anteriori. Algoritmul este:

Algoritmul G'. (O versiune a eliminării gaussiene). Dată o matrice $A \in \mathbf{R}^{n\times n}$ cu $A(k : n, k : n)$ nesingulare pentru $k = n : -1 : 2$, algoritmul calculează matricea inferior triunghiulară $L = N_2 N_3 \cdots N_{n-1} N_n A$. Matricea L suprascrie partea inferioar triunghiulară a lui A şi multiplicatorii gaussieni ν_{ik} suprascriu partea strict superior triunghiulară a matricei A.

1. **pentru** $k = n : -1 : 2$
   1. **pentru** $i = 1 : k-1$
      1. $a_{ik} \leftarrow \nu_{ik} = \frac{a_{ik}}{a_{kk}}$
   2. **pentru** $j = 1 : k-1$
      1. **pentru** $i = 1 : k-1$
         1. $a_{ij} \leftarrow a_{ij} - \nu_{ik} a_{kj}$

Algoritmul G' oferă o factorizare UL, i.e. furnizează o matrice unitate superior triunghiulară U şi una inferior triunghiulară L astfel încât $A = U * L$. Întradevar, din $L = N_2 N_3 \cdots N_{n-1} N_n A$ rezultă $A = N_n^{-1} N_{n-1}^{-1} \cdots N_2^{-1} L = U * L$, unde $U = N_n^{-1} N_{n-1}^{-1} \cdots N_2^{-1}$ este superior triunghiulară ca un produs de matrice superior triunghiulare. Mai mult, $N_k^{-1} = I_n + n_k e_k^T$ şi

$$U = N_n^{-1} N_{n-1}^{-1} \cdots N_2^{-1} = I_n + \sum_{k=2}^{n} n_k e_k^T = \begin{bmatrix} 1 & \nu_{12} & \cdots & \nu_{1,n-1} & \nu_{1n} \\ 0 & 1 & \cdots & \nu_{2,n-1} & \nu_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \nu_{n-1,n} \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

(demonstraţi). În concluzie, algoritmul G' calculează factorizarea UL. Efortul de calcul este $N_{fl} \approx \frac{2n^3}{3}$ flops.

Problema 2.7 Fie $u, v \in \mathbf{R}^n$ doi vectori nenuli şi $A = I_n + uv^T$.

a. Dacă A este nesingulară şi $b \in \mathbf{R}^n$, scrieţi un algoritm eficient pentru rezolvarea sistemului liniar $Ax = b$.

b. Dacă A este nesingulară, scrieţi un algoritm eficient care calculează $X = A^{-1}$.

Soluţie. Desigur, putem calcula A aplicând metode cunoscute precum eliminarea gaussiană sau factorizarea LU. Dar probleme speciale necesită deseori soluţii speciale. În acest caz obţinem algoritmi mai eficienţi dacă procedăm astfel:

a. Determinantul lui $A = I_n + uv^T$ is $1 + v^T u$ (vezi seminarul 1). Să multiplicăm $Ax = b$ la stânga cu v^T. Vom avea $v^T x + v^T uv^T x = v^T b$. Deoarece A este nesingulară, i.e. $1 + v^T u \neq 0$ rezultă $\alpha = v^T x = \dfrac{v^T b}{1 + v^T u}$. În concluzie, $x = b - \alpha u$. Un algoritm foarte eficient care necesită doar $N_{fl} \approx 4n$ flopi, este:

1. $\alpha = \dfrac{v^T b}{1 + v^T u}$
2. **pentru** $i = 1 : n$
   1. $x_i = b_i - \alpha * u_i$

b. Dacă A este nesingulară, un algoritm eficient pentru a calcula $X = A^{-1}$ este următoarea schemă:

1. **pentru** $j = 1 : n$
   1. Rezolvă $AX(:, j) = e_j$ folosind algoritmul de la punctul **a**.

Ţinând seama de faptul că $v^T e_j$ şi notând $\beta = 1 + v^T u = 1 + \sum_{i=1}^{n} u_i v_i$ forma sa detaliată este:

1. $\beta = 1$
2. pentru i=1:n
   1. $\beta = \beta + u_i v_i$
3. **pentru** $j = 1 : n$
   1. $\alpha = \dfrac{v_j}{\beta}$
   2. **pentru** $i = 1 : n$
      1. $X(i, j) = -\alpha u_i$
   3. $X(j, j) = 1 - X(j, j)$

Algoritmul necesită doar $N_{fl} \approx n^2$ flopi.

2.2.3 Sisteme simetrice pozitiv definite. Factorizarea Cholesky

Problema 2.8 Fie $T \in \mathbf{R}^{n \times n}$ o matrice simetrică, tridiagonală, pozitiv definită. Scrieţi un algoritm eficient care calculează factorizarea Cholesky $T = LL^T$ a lui T;

Soluţie. Vom proceda ca în cazul general (vezi cursul).

Pasul 1. Din egalitatea $T = LL^T$ avem: $t_{11} = l_{11}^2$, $t_{21} = l_{21}l_{11}$ şi $0 = l_{i1}l_{11}$, $i = 3 : n$; în concluzie

$$l_{11} = \sqrt{t_{11}}, \quad l_{21} = \frac{t_{21}}{l_{11}}, \quad l_{i1} = 0,\ i = 3 : n,$$

i.e. matricea L este bidiagonală în prima sa coloană.

Pasul k. Presupunem că am calculat primele $k-1$ coloane ale lui L şi că L este bidiagonală în primele $k-1$ coloane. Din egalitatea $T = LL^T$ avem: $t_{kk} = l_{k,k-1}^2 + l_{kk}^2$, $t_{k+1,k} = l_{k+1,k}l_{kk}$ şi $0 = l_{ik}l_{kk}$, $i = k+2 : n$; în concluzie

$$l_{kk} = \sqrt{t_{kk} - l_{k,k-1}^2}, \quad l_{k+1,k} = \frac{t_{k+1,k}}{l_{kk}}, \quad l_{ik} = 0,\ i = k+1 : n,$$

i.e. matricea L este bidiagonală şi în coloana k; prin inducţie, L este o matrice inferior bidiagonală.

Algoritmul este:

> Algoritm $CHOL\_TRID$. Dată o matrice simetrică, tridiagonală, pozitiv definită $T \in \mathbf{R}^{n\times n}$, acest algoritm suprascrie partea inferior triunghiulară cu matricea L din factorizarea Cholesky $T = LL^T$.
>
> 1. $t_{11} \leftarrow l_{11} = \sqrt{t_{11}}$
> 2. $t_{21} \leftarrow l_{21} = \frac{t_{21}}{l_{11}}$
> 4. **pentru** $k = 2 : n$
>   1. $t_{kk} \leftarrow l_{kk} = \sqrt{t_{kk} - l_{k,k-1}^2}$
>   2. **dacă** $k = n$ **atunci stop**
>   3. $t_{k+1,k} \leftarrow l_{k+1,k} = \frac{t_{k+1,k}}{l_{kk}}$

Problema 2.9 Dacă $A \in \mathbf{R}^{n\times n}$ este simetrică şi pozitiv definită, propuneţi un algoritm pentru calculu factorizării $A = UU^T$, unde U este superior triunghiulară şi are toate elementele diagonale pozitive.

Soluţie. Este uşor de văzut că trebuie început cu calculul elementului (n, n) al matricei superior triunghiulare U şi că putem calcula U pe coloane în ordine inversă. Vom numerota paşii de calcul cu numărul coloanei calculate a matricei U. *Pasul n.* Din egalitatea $A = UU^T$ avem: $a_{nn} = u_{nn}^2$ şi $a_{in} = u_{in}u_{nn}$, $i = 1 : n-1$; în concluzie

$$u_{nn} = \sqrt{a_{nn}}, \quad u_{in} = \frac{a_{in}}{u_{nn}},\ i = 1 : n-1.$$

Pasul k. Presupunem că am calculat ultimele $k+1 : n$ coloane ale lui U. Din egalitatea $A = UU^T$ avem: $a_{kk} = u_{kk}^2 + \sum_{j=k+1}^n u_{kj}^2$ şi $a_{ik} = u_{ik}u_{kk} + \sum_{j=k+1}^n u_{ij}u_{kj}$, $i = 1 : k-1$; în concluzie

$$u_{kk} = \sqrt{a_{kk} - \sum_{j=k+1}^{n} u_{kj}^2}, \quad u_{ik} = \frac{a_{ik} - \sum_{j=k+1}^n u_{ij}u_{kj}}{u_{kk}},\ i = 1 : k.$$

Matricea A fiind pozitiv definită, argumentele rădăcinilor pătrate sunt pozitive (demonstraţi). Algoritmul este:

Algoritmul UUT. Dată o matrice simetrică $A \in \mathbf{R}^{n\times n}$, acest algoritm stabileşte dacă matricea este pozitiv definită, în caz afirmativ, suprascrie partea superioară cu matricea U din factorizarea $A = UU^T$.)

1. **dacă** $a_{nn} \leq 0$ **atunci**
   1. Imprimă ('A nu este pozitiv definită')
   2. **stop**
2. $a_{nn} \leftarrow u_{nn} = \sqrt{a_{nn}}$
3. **pentru** $i = 1 : n-1$
   1. $a_{in} \leftarrow u_{in} = \frac{a_{in}}{u_{nn}}$
4. **pentru** $k = n-1 : -1 : 1$
   1. $\alpha \leftarrow a_{kk} - \sum_{j=k+1}^{n} u_{kj}^2$
   2. **dacă** $\alpha \leq 0$ **atunci**
      1. Imprimă ('A este pozitiv definită')
      2. **stop**
   3. $a_{kk} \leftarrow u_{kk} = \sqrt{\alpha}$
   4. **dacă** $k = 1$ **atunci stop**
   5. **pentru** $i = 1 : k-1$
      1. $a_{ik} \leftarrow u_{ik} = \frac{\left(a_{ik} - \sum_{j=k+1}^{n} u_{ij}u_{kj}\right)}{u_{kk}}$

Algoritmul UUT necesită aproximativ $N_{fl} = \frac{n^3}{3}$ flopi şi în plus n extrageri de rădăcini pătrate. Memoria necesară este de $M_{fl} = \frac{n^2}{2}$ (o matrice simetrică este de obicei stocată în una din parţile sale triunghiulare).

2.3 Probleme propuse

Problema 2.10 Considerăm date matricele $H \in \mathbf{R}^{n\times n}$, nesingulară superior triunghiulară şi $R \in \mathbf{R}^{n\times n}$, superior triunghiulară unitate. Propuneţi algoritmi eficienţi pentru:

a. Rezolvarea sistemului liniar $HRx = b$, cu $b \in \mathbf{R}^n$.

b. Când toate submatricele lider principale ale lui H sunt nesingulare, factorizarea Crout a lui $A = HR$ poate fi obţinută folosind următoarele două scheme:

Schema 1.
1. Se calculează $A = HR$.
2. Se calculează factorizarea Crout a lui A: $A = LU$.

Schema 2.
1. Se calculează factorizarea Crout a lui H: $H = L\bar{U}$.
2. Se calculează $U = \bar{U}R$.

Care este mai eficientă?

Problema 2.11 Propuneţi un algoritm eficient pentru a rezolva sistemul liniar $A^k x = b$, unde $A \in \mathbf{R}^{n\times n}$ este nesingulară, $b \in \mathbf{R}^n$ şi $k \in \mathbf{N}$, $k > 1$.

Problema 2.12 Fie $B = \begin{bmatrix} A & 0 \\ R & A \end{bmatrix}$, cu $A, R \in \mathbf{R}^{n\times n}$, nesingulare, R superior triunghiulară. Factorizarea LU a lui A există şi e cunoscută ca ($A = LU$).

a. Propuneţi un algoritm pentru calcularea factorizării LU a lui B, $B = \tilde{L}\tilde{U}$.

b. Propuneţi un algoritm pentru a rezolva sistemul liniar $Bx = d$, unde $d \in \mathbf{R}^{2n}$ este un vector dat.

Calculaţi numărul de operaţii pentru amândoi algoritmi.

Problema 2.13 Fie $A \in \mathbf{R}^{2n\times 2n}$ o matrice nesingulară cu toate submatricele lider principale nesingulare, $A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}$, cu $A_1, A_2, A_3, A_4 \in \mathbf{R}^{n\times n}$ and A_3 este superior triunghiulară.

a. Scrieţi un algoritm pentru rezolvarea sistemului liniar $Ax = b$, unde $b \in \mathbf{R}^{2n}$ este un vector dat.

b. Acelaşi lucru, păstrând numai ipoteza că A este nesingulară.

Problema 2.14 Fie $A \in \mathbf{R}^{n\times n}$ o matrice nesingulară tridiagonală ($a_{ij} = 0$, pentru $i > j+1$ or $i < j-1$).

a. Adaptaţi algoritmul de eliminare gaussiană pentru acest tip de matrice.

b. Propuneţi algoritmul care rezolvă $Ax = b$, cu $b \in \mathbf{R}^n$.

Problema 2.15 Dacă $A, B \in \mathbf{R}^{n\times n}$ este o matrice nesingulară, propuneţi un algoritm care rezolvă sistemul liniar $(AB)^k x = c$, unde $c \in \mathbf{R}^n$.

Problema 2.16 Propuneţi o variantă a algoritmului LSS_GPP pentru rezolvarea sistemului liniar $Ax = b$ folosind eliminarea gaussiană cu pivotare parţială, *fară* a reţine multiplicatorii.

Problema 2.17 Dacă $A \in \mathbf{R}^{n\times n}$ este simetrică, propuneţi un algoritm pentru a testa dacă matricea A este pozitiv definită sau nu.

Problema 2.18 Fie $T \in \mathbf{R}^{n\times n}$ o matrice simetrica, tridiagonală, pozitiv definită şi matricea $T = AA^T$. Propuneţi un algoitm eficient pentru:

a. a rezolva sistemul liniar $Tx = b$, unde $b \in \mathbf{R}^n$;

b. calcularea $\det(T)$;

c. calcularea inversei lui T.

2.4 Bibliografie

1. B. Jora, B. Dumitrescu, C. Oară, Numerical Methods, UPB, Bucharest, 1995
2. G.W. Stewart, Introduction to Matrix Computations, Academic Press, 1973.
3. G. Golub, Ch. Van Loan, Matrix Computations, 3-rd edition, John Hopkins University Press, 1998.
4. G.W. Stewart, Matrix Algorithms, vol.1: Basic Decompositions, SIAM, 2000.
5. G.W. Stewart, Matrix Algorithms, vol.2: Eigensystems, SIAM, 2002.
6. B. Dumitrescu, C. Popeea, B. Jora, Metode de calcul numeric matriceal. Algoritmi fundamentali, ALL, Bucureşti, 1998.

2.5 Programe MATLAB

În această secţiune sunt date programele MATLAB pentru implementarea algoritmilor prezentaţi în acest seminar. Programele au fost testate şi pot fi obţinute de la autorul lor menţionat în comentariile ataşate.

```
function [delta]=determinant(B,C)
%-------------------------------------------------------------
% Problema calculeaza determinanul unei matrice superior tridiagonala A(nxn).
% (A fiind rezultatul unei inmultiri a doua matrice superior bidiagonale
% B(nxn) si C(nxn)).Ca date de intrare avem cele doua matrice B respectiv C
% care inmultite dau A, iar ca date de iesire avem determinanul notat cu delta.
% Apelul: delta=determinant(B,C)
%
% Dumitru Iulia, aprilie 2006.
%-------------------------------------------------------------

[n,m]=size(B);

%analizam daca matricea e patratica
if n~=m
    error('Matricea nu e patratica');
end

%analizam daca matricele B si C sunt superior bidiagonale
for i=1:n
    for j=i+2:n
        if B(i,j)~=0
            error('Matricea B nu e superior bidiagonala');
        end
        if C(i,j)~=0
            error('Matricea C nu e superior bidiagonala');
        end
    end
    for j=1:i-1
        if B(i,j)~=0
            error('Matricea B nu e superior bidiagonala');
        end
        if C(i,j)~=0
            error('Matricea C nu e superior bidiagonala');
        end
    end
end

delta =1;
for i=1:n
```

```
41:         delta = delta *B(i,i)*C(i,i);
42:     end
```

```
1:  function [B]=UBINV(B);
2:  %-----------------------------------------
3:  % Algoritmul calculeaza inversa unei matrice patratica
4:  % B(nxn) superior bidiagonala. Inversa suprascrie
5:  % triunghiul superior al matricei B.
6:  % Apelul: B=UBINV(B)
7:  %
8:  %
9:  % Dumitru Iulia, aprilie 2006.
10: %-----------------------------------------
11: [n,m]=size(B);
12:
13: %analizam daca matricea e patratica
14: if n~=m
15:     error('Matricea nu e patratica');
16: end
17:
18: for i=1:n
19:     for j=i+2:n
20:         if B(i,j)~=0
21:             error('Matricea B nu e superior bidiagonala');
22:         end
23:     end
24:     for j=1:i-1
25:         if B(i,j)~=0
26:             error('Matricea B nu e superior bidiagonala');
27:         end
28:     end
29: end
30:
31: for j=n:-1:1
32:     B(j,j)=1/B(j,j);
33:     for i=j-1:-1:1
34:         B(i,j)=-B(i,i+1)*Y(i+1,j)/ B(i,i);
35:     end
36: end
```

```
1:  function [x]=UBSM1(B,C,d)
2:  %-----------------------------------------------------
3:  % Programul calculeaza solutia sistemului liniar Ax=d, unde A=B*C. Ca date
4:  % de intrare avem vectorul d si matricele nesingulare superior bidiagonale
```

```
5:  % B(nxn) respectiv C(nxn) care inmultite dau matricea A(nxn). Pasi de
6:  % rezolvare: intai se calculeaza matricea A superior tridiagonala cu formula
7:  % A=B*C apoi se rezolva sistemul liniar Ax=d.
8:  % Nota: folosim numele de UBSM pentru upper bidiagonal matrix solver
9:  % Apelul: x=UBSM1(B,C,d)
10: %
11: % Dumitru Iulia, aprilie 2006.
12: %-------------------------------------------------------------
13:
14: [n,m]=size(B);
15:
16: %analizam daca matricea e patratica
17: if n~=m
18:     error('Matricea nu e patratica');
19: end
20:
21: %analizam daca matricele B si C sunt superior bidiagonale
22: for i=1:n
23:     for j=i+2:n
24:         if B(i,j)~=0
25:             error('Matricea B nu e superior bidiagonala');
26:         end
27:         if C(i,j)~=0
28:             error('Matricea C nu e superior bidiagonala');
29:         end
30:     end
31:     for j=1:i-1
32:         if B(i,j)~=0
33:             error('Matricea B nu e superior bidiagonala');
34:         end
35:         if C(i,j)~=0
36:             error('Matricea C nu e superior bidiagonala');
37:         end
38:     end
39: end
40:
41: for i=1:n
42:     for j=1:m
43:         A(i,j)=0;
44:     end
45:     A(i,i)=B(i,i)*C(i,i);
46:     if i<n
47:         A(i,i+1)=B(i,i)*C(i,i+1)+B(i,i+1)*C(i+1,i+1);
48:     end
49:     if i<n-1
50:         A(i,i+2)=B(i,i+1)*C(i+1,i+2);
51:     end
```

```
52:   end
53:   x(n)=d(n)/A(n,m);
54:   x(n-1)=(d(n-1)-A(n-1,m)*x(n)) / A(n-1,m-1);
55:   for i=n-2:-1:1
56:       x(i)=(d(i) - A(i,i+1)*x(i+1)- A(i,i+2)*x(i+2))/ A(i,i);
57:   end
```

```
1:    function [x]=UBSM2(B,C,d)
2:    %-------------------------------------------------------------
3:    % Problema calculeaza sistemul liniar Ax=d. Ca date de intrare avem
4:    % vectorul d si matricele nesingulare superior bidiagonale B(nxn)
5:    % respectiv C(nxn) care inmultite dau matricea A(nxn). Pasi de rezolvare:
6:    % se inlocuieste A cu B*C in sistem, rezultand B*C*x=d. Notand y=C*x avem
7:    % B*y=d. Algoritmul urmator calculeaza intai y apoi x.
8:    % Apelul: x=UBSM2(B,C,d)
9:    %
10:   % Dumitru Iulia, aprilie 2006.
11:   %-------------------------------------------------------------
12:
13:   [n,m]=size(B);
14:
15:   %analizam daca matricea e patratica
16:   if n~=m
17:       error('Matricea nu e patratica');
18:   end
19:
20:   %analizam daca matricele B si C sunt superior bidiagonale
21:   for i=1:n
22:       for j=i+2:n
23:           if B(i,j)~=0
24:               error('Matricea B nu e superior bidiagonala');
25:           end
26:           if C(i,j)~=0
27:               error('Matricea C nu e superior bidiagonala');
28:           end
29:       end
30:       for j=1:i-1
31:           if B(i,j)~=0
32:               error('Matricea B nu e superior bidiagonala');
33:           end
34:           if C(i,j)~=0
35:               error('Matricea C nu e superior bidiagonala');
36:           end
37:       end
38:   end
39:
```

```
40:  y(n)=d(n)/B(n,m);
41:  for i=n-1:-1:1
42:       y(i)=(d(i)-B(i,i+1)*y(i+1))/B(i,i);
43:       if B(i,i)==0
44:            error('Matricea nu este nesingulara')
45:       end
46:  end
47:  x(n)=y(n)/C(n,m);
48:  for i=n-1:-1:1
49:       x(i)=(y(i)-C(i,i+1)*x(i+1))/C(i,i);
50:       if C(i,i)==0
51:            error('Matricea nu este nesingulara')
52:       end
53:  end
```

```
 1:  function [X]=UTRIINVa(B,C)
 2:  %-------------------------------------------------------------
 3:  % Algoritmul calculeaza inversa unei matrice superior tridiagonale
 4:  % patratice A(nxn).Ca date de intrare avem doar matricele superior
 5:  % bidiagonale B(nxn) si C(nxn). Matricea A este rezultatul inmultirii
 6:  % celor doua matrice. Pasii de calculare ai inversei sunt urmatorii:
 7:  % se calculeaza inversa matricei B care se suprascrie in B si inversa
 8:  % matricei C care se suprascrie in C. Inversa matricei A este rezultatul
 9:  % inmultirii celor doua matrice.
10:  % Apelul: X=UTRIINVa(B,C)
11:  %
12:  % Dumitru Iulia, aprilie 2006.
13:  %-------------------------------------------------------------
14:
15:  [n,m]=size(B);
16:
17:  %analizam daca matricea e patratica
18:  if n~=m
19:       error('Matricea nu e patratica');
20:  end
21:
22:  %analizam daca matricele B si C sunt superior bidiagonale
23:  for i=1:n
24:       for j=i+2:n
25:            if B(i,j)~=0
26:                 error('Matricea B nu e superior bidiagonala');
27:            end
28:            if C(i,j)~=0
29:                 error('Matricea C nu e superior bidiagonala');
30:            end
31:       end
```

```
32:         for j=1:i-1
33:             if B(i,j)~=0
34:                 error('Matricea B nu e superior bidiagonala');
35:             end
36:             if C(i,j)~=0
37:                 error('Matricea C nu e superior bidiagonala');
38:             end
39:         end
40:     end
41:
42:     Y=UBINV(B); %calculeaza inversa matricei B
43:     B=Y;
44:     Z=UBINV(C); %calculeaza inversa matricei C
45:     C=Z;
46:     for i=1:n %efectueaza inmultirea noilor matrice B si C
47:         for j=1:n
48:             X(i,j)=0;
49:             end
50:         for j=i:n
51:         for k=i:j
52:             X(i,j)=X(i,j)+C(i,k)*B(k,j);
53:             end
54:             end
55:         end
```

```
1:  function [X]=UTINVb(C,B)
2:  %-----------------------------------------------------------
3:  % Algoritmul calculeaza inversa unei matice superior tridiagonala
4:  % patratica A(nxn).Ca date de intrare avem doar matricele superior
5:  % bidiagonale B(nxn) si C(nxn). Matricea A este rezultatul inmultirii
6:  % celor doua matrice. Pasii de calculare ai inversei sunt urmatorii:
7:  % se rezolva ecuatia maticeala BY=In (se calculeaza practic inversa
8:  % matricei B) apoi ecuatia CX=Y (X reprezinta inversa maticei A).
9:  % In algoritm se foloseste o problema rezolvata anterior: inversa unei
10: % matrice superior bidiagonala patratica, cu suprascriere (algoritmul UBINV)
11: % Apelul: X=UTRIINVb(B,C)
12: %
13: % Dumitru Iulia, aprilie 2006.
14: %-----------------------------------------------------------
15:
16: [n,m]=size(B);
17:
18: %analizam daca matricea e patratica
19: if n~=m
20:     error('Matricea nu e patratica');
21: end
```

```
22:
23:   %analizam daca matricele B si C sunt superior bidiagonale
24:   for i=1:n
25:       for j=i+2:n
26:           if B(i,j)~=0
27:               error('Matricea B nu e superior bidiagonala');
28:           end
29:           if C(i,j)~=0
30:               error('Matricea C nu e superior bidiagonala');
31:           end
32:       end
33:       for j=1:i-1
34:           if B(i,j)~=0
35:               error('Matricea B nu e superior bidiagonala');
36:           end
37:           if C(i,j)~=0
38:               error('Matricea C nu e superior bidiagonala');
39:           end
40:       end
41:   end
42:
43:   Y=UBINV(B);
44:   B=Y;
45:   for j=n:-1:1
46:       for i=j+1:n
47:           X(i,j)=0;
48:       end
49:       X(j,j)=B(j,j)/C(j,j);
50:       if C(j,j)==0
51:           error('Matricea nu este nesingulara')
52:       end
53:       for i=j-1:-1:1
54:           X(i,j)=(B(i,j)-C(i,i+1)*X(i+1,j))/C(i,i);
55:           if C(i,i)==0
56:               error('Matricea nu este nesingulara')
57:           end
58:       end
59:   end
60:
61:
62:
63:
64:
65:
66:
```

```
1:  function [x]=UTRIS(U,b)
2:  %-------------------------------------------------------
3:  % Fiind data o matrice superior triunghiulara patratica A (nxn) si un
4:  % vector b, algoritmul calculeaza sistemul liniar Ax=b.
5:  % UTRIS=upper triunghiular solver.
6:  % Apelul: x=UTRIS(U,b)
7:  %
8:  % Dumitru Iulia, aprilie 2006.
9:  %-------------------------------------------------------
10:
11: [m,n] = size(U);
12:
13: if m~=n
14:     % in caz ca nu este o matrice patratica
15:     x = [ ];%x devine o matrice nula
16:     return;
17: end
18:
19: if m~=length(b)
20:     error('Nu exista solutie pentru');
21: end
22:
23: for i=n:-1:1
24:     s = b(i);
25:     if i<n
26:         for k=i+1:n
27:             s = s - U(i,k)*x(k);
28:         end
29:     end
30:     x(i) = s/U(i,i);
31:     if U(i,i)==0
32:         error('Matricea nu este nesingulara')
33:     end
34: end
```

```
1:  function [H,b,miu]=G_HESS(H,b)
2:  %-------------------------------------------------------
3:  % Fiind data o matrice superior Hessenberg nesingulara, patratica (nxnx)
4:  % cu toate submaticele lider principale nesingulare, se cere rezolvarea
5:  % sistemului liniar Hx=b. Algoritmul reprezinta o adaptare a algoritmului
6:  % de eliminare gaussiana fara pivotare. Algoritmul G_HESS calculeaza
7:  % factorizarea LU Doolittle a matricei H.
8:  % Apelul: H,b,miu=G_HESS(H,b)
9:  %
10: % Dumitru Iulia, aprilie 2006.
11: %-------------------------------------------------------
```

```
12:
13:   [n,m]=size(H);
14:
15:   %analizam daca matricea e patratica
16:   if n~=m
17:       error('Matricea nu e patratica');
18:   end
19:
20:   %analizam daca matricea este superior Hessenberg
21:   for i=3:n
22:   for j=1:i-2
23:             if H(i,j)~=0
24:                 error('Matricea H nu e superior Hessenberg')
25:             end
26:       end
27:   end
28:
29:   for k=1:n-1 %Multiplicatorii gaussieni si matricea
30:       miu(k+1,k)=H(k+1,k)/H(k,k); %superior triunghiulara rezultata vor
31:       if H(k,k)==0 %suprascrie matricea H.
32:           error('Matricea nu este nesingulara')
33:       end
34:       H(k+1,k)=miu(k+1,k);
35:       for j=k+1:n
36:           H(k+1,j)=H(k+1,j)-miu(k+1,k)*H(k,j);
37:       end
38:       b(k+1)=b(k+1)-H(k+1,k)*b(k);
39:   end
```

```
1:   function [H]=CROUT_HESS(H)
2:   %-----------------------------------------------------------------
3:   % Data o matrice superior Hessenberg H nesingulara patratica (nxn)
4:   % algoritmul calculeaza o matrice inferior bidiagonala L si o matrice
5:   % superior unitate triunghiulara U astfel incat H=L*U. Matricea L este
6:   % suprascrisa partii inferior triunghiulare a matricei A(inclusiv diagonala)
7:   % iar matricea U este suprascrisa partii superior triunghiulare a
8:   % matricei A(fara diagonala).
9:   % Apelul: H=CROUT_HESS(H)
10:  %
11:  % Dumitru Iulia, aprilie 2006.
12:  %-----------------------------------------------------------------
13:
14:  [n,m]=size(H);
15:
16:  %analizam daca matricea e patratica
17:  if n~=m
```

```
18:         error('Matricea nu e patratica');
19:     end
20:
21:     %analizam daca matricea este superior Hessenberg
22:     for i=3:n
23:     for j=1:i-2
24:             if H(i,j)~=0
25:                 error('Matricea H nu e superior Hessenberg')
26:             end
27:         end
28:     end
29:
30:     for i=1:n
31:     U(i,i)=1;
32:     end
33:
34:     for j=2:n-1
35:         U(1,j)=H(1,j)/H(1,1);
36:         H(1,j)=U(1,j);
37:     end
38:
39:     for k=2:n
40:         L(k,k)=H(k,k)-H(k,k-1)*H(k-1,k);
41:         H(k,k)=L(k,k);
42:         if k~=n
43:             for j=k+1:n
44:                 U(k,j)=(H(k,j)-H(k,k-1)*H(k-1,j))/H(k,k);
45:                 if H(k,k)==0
46:                     error('Matricea nu este nesingulara')
47:                 end
48:                 H(k,j)=U(k,j)
49:             end
50:         end
51:     end
```

```
1:  function [H,p]=GPP_HESS(H)
2:  %-------------------------------------------------------------
3:  % Data o matrice superior Hassenberg H patratica(de ordin n), algoritmul
4:  % efectueaza eliminarea gaussiana cu pivotare partiala. Multiplicatorii
5:  % gaussieni si matricea superior triunghiulara rezultata vor suprascrie
6:  % matricea H.
7:  % Apelul: H,p=GPP_HESS(H)
8:  %
9:  % Dumitru Iulia, aprilie 2006.
10: %-------------------------------------------------------------
11:
```

```
12:  [n,m]=size(H);
13:
14:  %analizam daca matricea e patratica
15:  if n~=m
16:      error('Matricea nu e patratica');
17:  end
18:
19:  %analizam daca matricea este superior Hessenberg
20:  for i=3:n
21:  for j=1:i-2
22:          if H(i,j)~=0
23:              error('Matricea H nu e superior Hessenberg')
24:          end
25:      end
26:  end
27:
28:  for k=1:n-1
29:      p(k)=k;
30:      max = abs( H(k,k) );
31:      if abs( H(k+1,k) ) >max
32:          max = abs( H(k+1,k) );
33:      end
34:      p(k)=k+1;
35:      for j=k:n
36:          temp = H(k,j);
37:          H(k,j) = H(k+1,j);
38:          H(k+1,j) = temp;
39:      end
40:      miu(k+1,k) = H(k+1,k)/H(k,k);
41:      if H(k,k)==0
42:          error('Matricea nu este nesingulara')
43:      end
44:      H(k+1,k)=miu(k+1,k);
45:      for j = k+1:n
46:          H(k+1,j)=H(k+1,j)-miu(k+1,k)*H(k,j);
47:      end
48:  end
49:
```

```
1:  function [x]=LSS_GPP_HESS(H,b)
2:  %-------------------------------------------------------------------
3:  % Fiind data o matrice superior Hassenberg nesingulara H patratica (de
4:  % ordin n) si un vector b, algoritmul calculeaza solutia x a sistemului
5:  % Hx=b folosind eliminarea gaussiana cu pivotare partiala.
6:  % Apelul: x=LSS_GPP_HESS(H,b)
7:  %
```

```
% Dumitru Iulia, aprilie 2006.
%-----------------------------------------------------------------------

[n,m]=size(H);

%analizam daca matricea e patratica
if n~=m
    error('Matricea nu e patratica');
end

%analizam daca matricea este superior Hessenberg
for i=3:n
for j=1:i-2
        if H(i,j)~=0
            error('Matricea H nu e superior Hessenberg')
        end
    end
end

%conditionam ca numarul liniilor vectorului b sa fie egal cu numarul coloanelor
%matricei H
if m~=length(b)
    error('Nu exista solutie pentru');
end

[H,p]=GPP_HESS(H);
for k=1:n-1
    if p(k)==k+1
    temp=b(k);
    b(k)=b(k+1);
    b(k+1)=temp;
    end
    b(k+1)=b(k+1)-H(k+1,k)*b(k);
end

x=UTRIS(triu(H),b)
```

```
function [A,miu,p]=GPP(A)
%-----------------------------------------------------------------------
% Data o matrice patratica A(nxn), algoritmul suprascrie triunghiul
% superior al lui A cu matricea superior triunchiulara U. Triunghiul
% strict inferior al lui A este suprascris de multiplicatorii gaussieni
% miu. In vectorul p se memoreaza intregii i(k), care definesc permutarile
% de linii.
% Apelul: A,miu,p=GPP(A)
```

```
%
% Dumitru Iulia, aprilie 2006.
%-------------------------------------------------------------


[n,m] = size(A);

%analizam daca matricea e patratica
if n~=m
    error('Matricea nu e patratica');
end

for k=1:n-1
    ik = k;
    max = abs( A(k,k) );

    for i=k:n
        if abs( A(i,k) ) >max
            ik = i;
            max = abs( A(i,k) );
        end
    end

    p(k)=ik;

    for j=k:n
        temp = A(k,j);
        A(k,j) = A(ik,j);
        A(ik,j) = temp;
    end

    for i=k+1:n
        miu(i,k) = A(i,k)/A(k,k);
        if A(k,k)==0
            error('Matricea nu este nesingulara')
        end
        A(i,k)=miu(i,k);

    end

    for i=k+1:n
        for j = k+1:n
            A(i,j)=A(i,j)-miu(i,k)*A(k,j);
        end
    end
end

```

```
1:  function [X]=MMS(A,B)
2:  %-------------------------------------------------------------------
3:  % Fiind date doua matrice: A (nxn) si B (nxp) se cere sa se rezolve ecuatia
4:  % matriceala A*X=B.
5:  % Apelul: X=MMS(A,B)
6:  %
7:  % Dumitru Iulia, aprilie 2003.
8:  %-------------------------------------------------------------------
9:
10: [n,m]=size(A);
11:
12: %analizam daca matricea e patratica
13: if n~=m
14:     error('Matricea A nu e patratica');
15: end
16:
17: [n,p]=size(B);
18:
19: [M,U,p]=GPP(A);
20: for j=1:p
21:     for k=1:n-1
22:         B(k,j)=B(p(k),j);
23:         for i=k+1:n
24:             B(i,j)=B(i,j)-U(i,k)*B(k,j);
25:         end
26:         X(j)=UTRIS(U,B(j));
27:     end
28: end
```

```
1:  function [A]=G_prim(A)
2:  %-------------------------------------------------------------
3:  % Data fiind matricea patratica A(nxn) algoritmul calculeaza matricea
4:  % L pe care o suprascrie partii inferioare triunghiulare a lui A si
5:  % multiplicatorii miu pe care ii suprascriu partii superior triunghiulare
6:  % a lui A. Practic alogritmul ofera o factorizare UL (furnizeaza o
7:  % matrice unitate superior triunghiulara U si una inferior triunghiulara
8:  % L astefl incat A=U*L)
9:  % Apelul: A=G_prim(A)
10: %
11: % Dumitru Iulia, aprilie 2006.
12: %-------------------------------------------------------------
13:
14: [n,m]=size(A);
```

```
15:
16:   %analizam daca matricea e patratica
17:   if n~=m
18:       error('Matricea nu e patratica');
19:   end
20:
21:   for k=n:-1:2
22:       for i=1:k-1
23:           miu(i,k)=A(i,k)/A(k,k);
24:           if H(k,k)==0
25:               error('Matricea nu este nesingulara')
26:           end
27:           A(i,k)=miu(i,k);
28:       end
29:       for j=1:k-1
30:           for i=1:k-1
31:               A(i,j)=A(i,j)-miu(i,k)*A(k,j);
32:           end
33:       end
34:   end
```

```
1:    function [T]=CHOL_TRID(T)
2:    %-------------------------------------------------------------
3:    % Fiind data o matrice T patratica (de ordin n), tridiagonala, simetrica
4:    % pozitiv definita, nesingulara care satisface relatia: T=L*L_transpus
5:    % algoritmul suprascrie partea inferior triunghiulara a matricei T cu
6:    % matricea L. Obs: matricea L rezulta prin identificare a fi inferior
7:    % bidiagonala.
8:    % Apelul: T=CHOL_TRID(T)
9:    %
10:   % Dumitru Iulia, aprilie 2006.
11:   %-------------------------------------------------------------
12:
13:   [n,m]=size(T);
14:
15:   %analizam daca matricea e patratica
16:   if n~=m
17:       error('Matricea nu e patratica');
18:   end
19:
20:   for i=1:n-2
21:       for j=i+2:n
22:           if T(i,j)~=0
23:               error('Matricea nu e tridiagonala')
24:           end
25:       end
```

```
26:   end
27:
28:   for i=3:n
29:       for j=1:i-2
30:           if T(i,j)~=0
31:               error('Matricea nu e tridiagonala')
32:           end
33:       end
34:   end
35:
36:   for i=1:n-1
37:       if T(i,i+1)~=T(i+1,i)
38:           error('Matricea nu este simetrica')
39:       end
40:   end
41:
42:   L(1,1)=sqrt(T(1,1));
43:   T(1,1)=L(1,1);
44:   L(2,1)=T(2,1)/L(1,1);
45:   T(2,1)=L(2,1);
46:   for k=2:n
47:       L(k,k)=sqrt(T(k,k)-L(k,k-1)*L(k,k-1));
48:       T(k,k)=L(k,k);
49:       if k~=n
50:       L(k+1,k)=T(k+1,k)/L(k,k);
51:       if L(k,k)==0
52:           error('Matricea nu este nesingulara')
53:       end
54:       T(k+1,k)=L(k+1,k);
55:       end
56:   end
```

```
1:   function [A]=UUT(A)
2:   %-------------------------------------------------------------------
3:   % Fiind data o matrice simetrica si pozitiv definita, propuneti un algoritm
4:   % pentru factorizarea A=U*U transpus. Aici U este superior triunghiulara si
5:   % are elemente pozitive pe diagonala. Practic, algoritmul stabileste daca
6:   % matricea A este pozitiv definita, in caz afirmativ, suprascrie partea
7:   % superioara cu matricea U.
8:   % Apelul: A=UUT(A)
9:   %
10:  % Dumitru Iulia, aprilie 2006.
11:  %-------------------------------------------------------------------
12:
13:  [n,m]=size(A);
14:
```

```
%analizam daca matricea e patratica
if n~=m
    error('Matricea nu e patratica');
end

for i=1:n
    for j=n:-1:i+1
        if A(i,j)~=A(j,i)
            error('Matricea nu este simetrica')
        end
    end
end

if A(n,n)<=0,
    error('A nu este pozitiv definita');
end
U(n,n)=sqrt(A(n,n));
A(n,n)=U(n,n);
for i=1:n-1
    U(i,n)=A(i,n)/U(n,n);
    A(i,n)=U(i,n);
end
for k=n-1:-1:1
    s=0;
    for j=k+1:n
        s=s+U(k,j)*U(k,j);
    end
    alpha=A(k,k)-s;
    if alpha <= 0
        error('A nu este pozitiv definita');
    end
    U(k,k)=sqrt(alpha);
    A(k,k)=U(k,k);
    if k~1
        for i=1:k-1
            suma=0;
            for j=k+1:n
                suma=suma+U(i,j)*U(k,j);
            end
            U(i,k)=(A(i,k)-suma)/U(k,k);
            if U(k,k)==0
                error('Matricea nu este nesingulara')
            end
            A(i,k)=U(i,k);
        end
    end
end
```

62:
63: