Seminar 1

Calcul matriceal elementar

Acest seminar este prima prezentare a calculului matriceal. Calculele matriceale elementare nu sunt prezente în curs dar sunt intens utilizate. Fără o înţelegere profundă a acestor noţiuni introductive veţi avea mari probleme la examinare şi nu veţi înţelege calculul ştiintific modern.

1.1 Preliminarii

1.1.1 Notaţii pentru scalari, vectori şi matrice

Structurile de date folosite la cursul de metode numerice şi, în special, în cazul calculului matriceal sunt *matrici de numere reale sau complexe*. O matrice este o structură de date bidimensională (dreptunghiulară) în nodurile căreia se află numere reale sau complexe. Un vector este o structură de date unidimensională (linie sau coloană) în nodurile căreia se află numere reale sau complexe.

Numerele reale sau complexe se numesc *scalari* reali, respectiv complecşi. Un scalar complex este o pereche ordonată de scalari reali. Astfel, calculul cu matrice cu elemente complexe se poate reduce la calcul matriceal cu matrice cu elemente reale. Ca urmare, în continuare ne vom referi numai la scalari, vectori şi matrice reale. Mulţimea numerelor reale va fi notată cu $\mathbb{R}$, mulţimea vectorilor cu n elemente va fi notată cu $\mathbb{R}^n$ iar $\mathbb{R}^{m\times n}$ este mulţimea matricelor cu elemente reale cu m linii şi n coloane. Vom considera un vector ca fiind implicit de tipul *coloană*. Un vector linie va fi văzut ca un vector (coloană) transpus. O matrice de tipul 1×1 este un scalar, o matrice de tipul $1 \times n$ este un vector (linie) şi o matrice te tipul $m \times 1$ este un vector (coloană). Pentru comoditate vom folosi conceptul de matrice vidă, care este o matrice fără elemente, notată $[\,]$.

Vom considera întotdeauna că numerele reale sunt reprezentate în memoria calculatorului într-un format specific, numit format virgulă mobilă (FVM) şi că toate operaţiile aritmetice de bază (cum ar fi adunarea, scăderea, îmmulţirea, împărţirea, extragerea radicalului etc.) pentru numerele în virgulă mobilă sunt implementate în calculator.

Pe parcursul cursului se va utiliza un set de notaţii pe care vă sfătuim să îl folosiţi şi voi. Fiecare scalar, vector sau matrice va fi notat cu un simbol care va fi interpretat ca un

nume de identificare. Pentru fiecare scalar, vector sau matrice reală se alocă în memoria calculatorului o structură de date în virgulă mobilă. Notaţiile standard sunt:

- Litere greceşti mici: α, β, ..., ω pentru scalari (izolaţi);
- Litere latine mici: a, b, c, ..., x, y, z pentru vectori (coloane);
- Litere latine mari: A, B, C, ..., X, Y, Z pentru matrice.

Fie $A \in \mathbb{R}^{m\times n}$ o matrice şi $b \in \mathbb{R}^m$ un vector. Elementul aflat pe linia i şi coloana j a matricei A va fi notat cu $A(i,j)$ sau a_{ij}. Elementul i al vectorului b va fi notat cu $b(i)$ sau b_i.

Vom folosi intens notaţia MATLAB ":" pentru a crea un vector linie. Expresia

$$a = \beta : \sigma : \gamma$$

crează un vector linie a cu primul element β, al doilea element $\beta+\sigma$, al treilea element $\beta+2\sigma$ s.a.m.d., însă ultimul element nu poate fi mai mare decât γ. Valoarea implicită pentru pasul σ este $\sigma = 1$. Aşadar, dacă $n = 5$, atunci $i = 1 : n$ crează vectorul linie $i = [\,1\ 2\ 3\ 4\ 5\,]$ dar $j = n : 1$ crează un vector vid; vectorul $k = n : -1 : 1$ este $[\,5\ 4\ 3\ 2\ 1\,]$.

Pentru a crea o matrice cu elementele unei alte matrice putem indica elementele matricei noi cu coordonatele matricei vechi. Spre exemplu, fie $r = [\,r1\ r2\ \ldots\,]$ si $c = [\,c_1\ c_2\ \ldots\,]$ coordonatele liniilor, respectiv, ale coloanelor matricei vechi. Atunci expresia

$$B = A(r, c)$$

crează matricea

$$B = \begin{bmatrix} A(r_1,c_1) & A(r_1,c_2) & \cdots \\ A(r_2,c_1) & A(r_2,c_2) & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

Utilizând notaţia ":" putem crea o mare varietate de matrice cu elementele unei matrice date. Spre exemplu, fie $A \in \mathbb{R}^{m\times n}$ o matrice data şi $k \le \min(m,n)$. Atunci

- $B = A(1:k, 1:k)$ este colţul din stanga sus de ordin k al lui A, numit *submatrice lider principală* a lui A;
- $R = A(i, 1:n)$ este linia i a lui$A$;
- $C = A(1:m, j)$ este coloana j a lui A.

În acest context, notaţia ":" fără parametri înseamnă "toate". Astfel, linia i şi, respectiv, coloana j a lui A pot fi extrase prin $R = A(i,:)$, respectiv $C = A(:,j)$.

1.1.2 Matrice structurate

Matricele folosite în calculul matriceal pot avea multe elemente nule. Numărul şi poziţia lor fac să existe mai multe tipuri de matrice pe care le definim în cele ce urmează.

În primul rând, o matrice care are un număr mare (să zicem, mai mare de 90%) de elemente nule se numeşte *matrice rară*. O matrice care nu este rară se numeşte *densă*.

În cazul calculului matriceal cu matrice rare, calculatorul foloseşte metode speciale pentru memorare şi procesare (de exemplu, sunt memorate doar elementele nenule împreună cu coordonatele lor). În continuare ne vom referi numai la calcul matriceal cu matrice dense. Astfel, toate matricele folosite (chiar şi matricea zero) vor fi văzute ca matrice dense.

Matricele dense pot avea un număr semnificativ de elemente nule. Matricele în care elementele nule sunt poziţionate într-o formă regulată vor fi numite matrice (dense) *structurate*. Matricele structurate joacă un rol important în calculul matriceal. Cele mai importante matrice structurate sunt definite mai jos.

1. **Matrice triunghiulare.** O matrice $L \in \mathbb{R}^{m\times n}$ se numeşte *inferior triunghiulară* dacă $L(i,j) = 0$, oricare ar fi $i < j$ iar L se numeşte *inferior triunghiulară unitate* dacă în plus $L(i,i) = 1$ oricare ar fi $i \in 1 : \min(m,n)$. O matrice $U \in \mathbb{R}^{m\times n}$ se numeşte *superior triunghiulară* dacă $U(i,j) = 0$, oricare ar fi $j < i$ iar U se numeşte *superior triunghiulară unitate* dacă în plus $U(i,i) = 1$ oricare ar fi $i \in 1 : \min(m,n)$. Remarcaţi că elementele nule ale unei matrice inferior triunghiulare cu $m > n$ şi elementele nule ale unei matrice superior triunghiulare cu $m < n$ formează o structură trapezoidală.

2. **Matrice diagonale.** O matrice $D \in \mathbb{R}^{m\times n}$ se numeşte *diagonală* dacă $D(i,j) = 0$, oricare ar fi $i \neq j$, adică este în acelaşi timp şi inferior triunghiulară şi superior triuunghiulară.

3. **Matrice Hessenberg.** O matrice patratică $G \in \mathbb{R}^{n\times n}$ se numeşte *inferior Hessenberg* dacă $G(i,j) = 0$, oricare ar fi $j > i + 1$. O matrice patratică $H \in \mathbb{R}^{n\times n}$ se numeşte *superior Hessenberg* dacă $H(i,j) = 0$, oricare ar fi $i > j + 1$.

4. **Matrice tridiagonale.** O matrice patratică $T \in \mathbb{R}^{n\times n}$ se numeşte *tridiagonală* dacă $T(i,j) = 0$, oricare ar fi i,j care satisfac $|i - j| > 1$, adică este în acelaşi timp superior şi inferior Hessenberg.

5. **Matrice bandă.** O matrice $B \in \mathbb{R}^{m\times n}$ se numeşte *matrice bandă de laţime (p,q)* dacă $B(i,j) = 0$, oricare ar fi $i < j + p$ şi $j < i + q$. O matrice tridiagonală este o matrice bandă cu $p = q = 1$.

Observaţie: Deşi limbajele de programare de nivel înalt nu permit astfel de structuri de date cum sunt matricele triunghiulare, putem să evităm memorarea elementelor nule şi să obţinem o memorare economică folosind o singură matrice patratică pentru memorarea unei matrice inferior triunghiulare şi a uneia superior tringhiulare. Astfel de strategii de stocare vor fi folosite spre exemplu de algoritmii de factorizare LU şi QR.

Unele matrice pot să nu aibă (multe) elemente nule dar să necesite o procesare specială. Câteva din aceste matrice sunt:

1. **Matrice simetrice.** O matrice patratică $S \in \mathbb{R}^{n\times n}$ se numeşte *simetrică* dacă $S(i,j) = S(j,i)$, oricare ar fi i,j. In mod normal, o matrice simetrică este stocată numai prin partea sa inferior sau superior triunghiulară.

2. **Matrice pozitiv definite.** O matrice patratică simetrică $P \in \mathbb{R}^{n\times n}$ se numeşte *pozitiv definită* dacă toţi scalarii $x^T Px$, unde $x \in \mathbb{R}^n$ este un vector (coloană) nenul arbitrar ales, sunt strict pozitivi. Dacă $x^T Px \geq 0$, $\forall\, x \in \mathbb{R}^n$, atunci P se numeşte *pozitiv semi-definită*. O matrice pozitiv definită are toţi elementii de pe diagonală

pozitivi (demonstraţi) dar elementele care nu se află pe diagonală nu sunt obligatoriu pozitive. O matrice simetrică $N \in \mathbb{R}^{n\times n}$ se numeşte *negativ (semi)-definită* dacă $-N$ este pozitiv (semi)-definită.

3. **Matrice ortogonale.** Doi vectori (coloană) $x, y \in \mathbb{R}^n$ se numesc *ortogonali* dacă $y^T x = 0$ şi *ortonormaţi* dacă, în plus, $\|x\|_2 = \|y\|_2 = 1$. O matrice patratică $Q \in \mathbb{R}^{n\times n}$ se numeşte *ortogonală* dacă toate coloanele ei formează perechi ortonormate, adică $Q^T Q = I_n$, unde I_n este matricea unitate. Matricele ortogonale joacă un rol decisiv în rezolvarea problemei liniare a celor mai mici pătrate.

1.1.3 Probleme de calcul numeric, algoritmi secvenţiali şi performanţele lor

Metodele numerice au rolul să rezolve probleme matematice care folosesc date numerice. O problemă matematică cu date iniţiale numerice şi cu un rezultat numeric trebuie să aibă o soluţie unică. În general, există multe metode de rezolvare a unei probleme de calcul numeric dată. O expunere detaliată a unei metode numerice se numeşte algoritm. Aşadar, un algoritm este o listă de operaţii aritmetice şi logice care duc datele numerice iniţiale la soluţia numerică finală. Aceste rezultate finale sunt interpretate ca *soluţia calculată* a unei probleme date. Când algoritmul este făcut pentru a fi executat pe un calculator cu un singur procesor, adică există o ordine strictă a execuţiei instrucţiunilor (două instrucţiuni nu pot fi executate în acelaşi timp!) atunci algoritmul se numeşte *secvenţial*. Altfel algoritmul se numeşte *paralel*.

Principalele performanţe ale algoritmilor sevenţiali sunt *stabilitatea* şi *eficienţa*. Stabilitatea este un obiectiv esenţial însă foarte greu de măsurat şi de asigurat. Vezi cursul pentru mai multe detalii. Eficienţa algoritmilor secvenţiali poate fi măsurată de doi parametri: numărul de operaţii în virgulă mobilă N (flops) şi volumul de memorie necesar M (exprimat prin numărul de locaţii FVM). Numărul N determină timpul de execuţie al algoritmului.

Toţi algoritmii pe care îi vom studia vor avea o complexitate polinomială, astfel N şi M sunt polinoame de dimensiunea problemei, exprimate în funcţie de numărul iniţial şi cel final de date, spre exemplu în funcţie de ordinul n al matricelor folosite. Dacă

$$N = \alpha_1 n^p + \alpha_2 n^{p-1} + \ldots + \alpha_p n + \alpha_{p+1}, \qquad M = \beta_1 n^q + \beta_2 n^{q-1} + \ldots + \beta_q n + \beta_{q+1}$$

ne vor interesa numai evaluările asimptotice

$$\tilde{N} \approx \alpha_1 n^p, \qquad \tilde{M} \approx \beta_1 n^q$$

(evident $\lim_{n\to\infty} \frac{\tilde{N}}{N} = \lim_{n\to\infty} \frac{\tilde{M}}{M} = 1$).

Pentru a îmbunătăţi eficienţa algoritmilor sunt utile următoarele recomandări:

- nu executaţi operaţii cu rezultatul cunoscut, de exemplu evitaţi adunări şi scăderi cu zero, înmulţiri cu 1 sau 0, împarţiri la 1 etc;
- nu repetaţi aceleaşi calcule; în cazul în care calculatorul funcţionează corect veţi obtine acelaşi rezultat;
- evitaţi execuţia operaţiilor inutile;

- evitaţi memorarea informaţiilor care nu trebuiesc neapărat memorate; de exemplu nu memoraţi niciodată matricea zero sau matricea unitate: le puteţi folosi fară să le aveţi în memoria calculatorului;
- nu memoraţi aceeaşi informaţie de mai multe ori;
- folosiţi metode eficiente de memorare a matricelor structurate, de exemplu matricele diagonale pot fi memorate numai prin memorarea elementelor diagonale etc;
- folosiţi suprascrierea; ex: dacă vreţi să calculaţi vectorul $z = \alpha x + y$ şi nu mai aveţi nevoie de x sau y pentru calculele următoare atunci calculaţi $x \leftarrow \alpha x + y$ sau $y \leftarrow \alpha x + y$.

Observaţie. Nu neglijaţi recomandările de mai sus. B.Francis şi N.Koublanovskaia au devenit celebri descoperind cum să evite calculele excesive dintr-un algoritm. Şi tu poti deveni un cercetător cunoscut reducând valoarea parametrilor p şi/sau q în cazul problemelor numerice standard. Nu se cunoaşte încă valoarea minimă a parametrului p pentru o problemă simplă cum este înmulţirea matricelor!

1.2 Convenţii pentru pseudo-cod

Vom adopta câteva convenţii pentru scrierea algoritmilor. Aceste convenţii ne vor permite să scriem algoritmii de calcul într-o formă concisă, clară, uşor de înţeles şi de implementat într-un limbaj de programare de nivel înalt. Instrucţiunile de bază ale pseudo-cod-ului sunt prezentate mai jos. Ele pot fi completate cu orice altă instrucţiune neambiguă şi uşor de înţeles.

1.2.1 Instrucţiuni de atribuire

Formele generale ale unei instrucţiuni de atribuire sunt:

$$v = \text{expresie} \qquad \text{sau} \qquad v \leftarrow \text{expresie}$$

unde v este numele unei variabile şi *expresie* este orice expresie aritmetică sau logică validă. O expresie aritmetică este validă dacă:

- toate variabilele folosite au o valoare validă (adică au fost iniţializate sau calculate anterior);
- toate operaţiile sunt bine definite şi expresia are o valoare bine definită; astfel nu sunt permise: împărţirea la zero, operaţiile nedeterminate cum ar fi $0 \cdot \infty$, $0/0$, ∞/∞ etc.

O expresie logică este validă dacă:

- toate variabilele logice folosite au o valoare logică validă ("adevărat" sau "fals");
- sunt folosiţi numai operatori logici binecunoscuţi: ŞI, SAU, NU, NICI etc.

- expresia ia o valoare logică clară ("adevarat" sau "fals").

Execuţia unei instrucţiuni de atribuire implică:

- evaluarea expresiei;
- memorarea valorii expresiei în memoria calculatorului într-o zonă identificată cu numele variabilei v; vechea valoare a lui v se pierde.

Uneori vom folosim o formă specială

$$u \leftrightarrow v$$

pentru interschimbarea valorilor dintre variabilele u şi v. Această instrucţiune este echivalentă cu următoarele trei instrucţiuni de atribuire

1. aux $= u$
2. $u = v$
3. $v =$ aux,

unde aux este o variabila auxiliară.

1.2.2 Instrucţiunea Dacă

Utilizarea condiţionată a unui set de instrucţiuni poate fi realizată folosind instrucţiunea "dacă" a cărei sintaxă este:

1. **dacă** expresie_logică
   1. instrucţiuni 1

   altfel
   1. instrucţiuni 2

Execuţia instrucţiunii "dacă" implică

1. evaluarea expresie_logică; rezultatul are valoareă logică ADEVĂRAT sau FALS;
2. dacă rezultatul are valoarea logică ADEVĂRAT atunci se execută primul grup de instrucţiuni (instrucţiuni 1);
3. dacă rezultatul are valoarea logică FALS atunci se execută al doilea grup de instrucţiuni (instrucţiuni 2).

Partea "altfel" a unei instrucţiuni "dacă" poate lipsi.

Expresia logică poate fi scrisă explicit, dar trebuie să aibă o valoare logică clară la momentul execuţiei. Spre exemplu

1. **dacă** 'este duminică'
   1. stai acasă

   altfel
   1. mergi la scoală.

are sens numai dacă cel care execută algoritmul ştie în ce zi a săptămânii se află.

De asemenea, se recomandă să se evite situaţiile în care expresia logică este o constantă "evidentă" deoarece în acest caz instrucţiunea "dacă" nu mai este necesară. De exemplu

1. **dacă** '$1 > 2$'
   1. $a = 3.14$
   altfel
   1. $a = 2.73$

se poate înlocui imediat cu $a = 2.73$.

1.2.3 Cicluri

Când avem un grup de instrucţiuni care trebuie executate în mod repetat avem la dispoziţie două tipuri de instrucţiuni ciclice.

a) instrucţiunea **pentru** are următoarea sintaxă:
   1. **pentru** $k =$ expresie
      1. instrucţiuni

   Execuţia unei instrucţiuni **"pentru"** implică

   1. evaluarea expresiei; rezultatul trebuie să fie un vector linie;
   2. se atribuie variabilei k primul element al vectorului linie calculat şi se execută instrucţiunile din corpul ciclului;
   3. se atribuie variabilei k cel de-al doilea element al vectorului linie calculat şi se execută instrucţiunile;
   4. şi asa mai departe până la atribuirea lui k a ultimului element al vectorului linie.

   Obsevaţie: Valoarea curentă a lui k poate fi folosită în instrucţiuni dar nu recomandăm modificarea acesteia. În MATLAB rezultatul evaluării expreseiei poate fi o matrice în care caz lui k i se atribuie succesiv, în ordinea naturală, câte o coloană a matricei şi pentru fiecare atribuire se execută instrucţiunile din corpul ciclului.

   Exemplu: fiind dat un scalar real α şi un număr natural $n \geq 1$ următorul ciclu calculează $\beta = \alpha^{2^n}$.
   1. $\beta = \alpha$
   2. **pentru** $k = 1 : n$
      1. $\beta = \beta * \beta$

b) instrucţiunea **cât timp** are următoarea sintaxă:
   1. **cât timp** expresie_logica
      1. instrucţiuni

   Execuţia unei instrucţiuni **"cât timp"** implică:

   1. evaluarea expresie_logică; rezultatul trebuie să fie ADEVĂRAT sau FALS;
   2. dacă rezultatul este ADEVARAT se execută instrucţiunile; altfel ciclul "cât timp" este terminat;
   3. mergi la 1.

Observaţie: Dacă rezultatul primei evaluări este FALS instrucţiunile din corpul ciclului nu sunt executate niciodată. Dacă rezultatul primei evaluări este adevărat şi valoarea expresiei logice nu se schimbă pe parcursul executării instrucţiunilor, atunci ciclul **"cât timp"** nu se termină niciodată; avem un ciclu infinit. Dacă nu aveţi nevoie de un ciclu infinit aveţi grijă să schimbaţi valoarea expresiei pe parcursul executării instrucţiunilor.

Exemplu: fiind dat un scalar real α şi un număr natural $n \geq 1$ următorul ciclu calculează $\beta = \alpha^{2^n}$.

1. $\beta = \alpha$
2. $k = 1$
3. **cât timp** $k \leq n$
    1. $\beta = \beta * \beta$
    2. $k = k + 1$

1.2.4 Structura algoritmilor

Orice algoritm va avea două părţi:

1. prima parte conţine informaţii privitoare la problema rezolvată şi metoda folosită pentru rezolvarea ei:

    - datele de intrare (iniţiale);
    - datele calculate (rezultate);
    - metoda folosită;

    Opţional, prima parte poate conţine

    - numele autorului
    - data elaborării algoritmului
    - versiunea etc.

2. a doua parte este o listă de instrucţiuni numerotate scrise în pseudo-cod care arată cum sunt folosite datele iniţiale pentru a ajunge la rezultatul final.

1.3 BLAS

Cele mai importante operaţii cu matrice tratate în acest seminar sunt implementate într-un pachet software numit "Basic Linear Algebra Soubrutines" (BLAS). Există versiuni speciale a BLAS concepute pentru calculatoare cu organizare ierarhică a memoriei care sunt optimizate pentru a exploata toate atributele calculatorului ţintă. BLAS este organizat pe trei nivele. Lista subnivelelor BLAS este dată în Anexă.

1.3.1 Nivelul 1. Operaţii de bază cu vectori

Nivelul 1 conţine operaţii de complexitate $O(n)$. Câteva dintre acestea sunt:

1. SAXPY (acronim pentru Scalar Alpha X Plus Y)

$$y \leftarrow \alpha x + y$$

unde x şi y sunt vectori daţi de dimensiune n (coloană sau linie) şi α este un scalar dat. Evident dacă y are iniţial toate elementele nule, atunci SAXPY calculează $y = \alpha x$. (Atenţie: dacă vreţi să calculaţi $y = \alpha x$ folosind SAXPY, nu uitaţi să iniţializaţi y cu 0).

2. DOT calculează produsul scalar a doi vectori (coloană) $x, y \in \mathbb{R}^n$ care este, prin definiţie,

$$\alpha = y^T x = \sum_{i=1}^{n} x_i y_i$$

3. Norme vectoriale. Ne mărginim să prezentăm aici norma vectorială euclidiană în $\mathbb{R}^n$ definită de

$$\nu = \sqrt{x^T x} = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

1.3.2 Nivelul 2. Operaţii de bază matrice-vector

Nivelul 2 al BLAS conţine operaţii de complexitate $O(n^2)$. Câteva dintre acestea sunt:

1. GAXPY (acronim pentru General A X Plus Y)

$$y \leftarrow Ax + y$$

unde x şi y sunt vectori (coloană) de dimensiune n şi A este o matrice $n \times n$ dată. Evident, dacă iniţial toate elementele lui y sunt nule, atunci GAXPY calculează produsul matrice-vector $y = Ax$. (Atenţie: dacă vreţi să calculaţi $y = Ax$ folosind GAXPY, nu uitaţi să iniţializaţi y cu zero). În acest context vom folosi şi operaţia cu vectori linie $y^T \leftarrow x^T A + y^T$.

2. OUT produsul extern a doi vectori (coloană) $x, y \in \mathbb{R}^n$ este definit ca o matrice $n \times n$

$$A = xy^T, \quad \text{i.e.} \quad A(i,j) = x_i y_j, \quad i,j = 1:n$$

Nivelul 2 conţine totodată algoritmi de rezolvare a sistemelor triunghiulare liniare (vezi seminarul 2).

1.3.3 Nivelul 3. Operaţii de bază matrice-matrice

Nivelul 3 al BLAS conţine operaţii de complexitate $O(n^3)$. Cele mai importante dintre acestea sunt:

1. Înmulţirea matricelor: fiind daţi scalarii reali α şi β şi matricele reale $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, $C \in \mathbb{R}^{m \times p}$, procedura de înmulţire a matricelor BLAS calculează

$$C \leftarrow \alpha * A * B + \beta * C.$$

Luând $\alpha = \beta = 1$ şi matricea iniţiala C zero, procedura de mai sus calculează produsul $C = A * B$. BLAS acoperă toate cazurile particulare ale înmulţirii matrice-matrice. Pentru cazul în care măcar una dintre matrice este simetrică există o procedură specială. De asemenea, există proceduri speciale pentru cazul în care matricele implicate sunt triunghiulare.

2. Actualizarea de rang k a unei matrice simetrice: fiind daţi scalarii reali α şi β , o matrice simetrică $C \in \mathbb{R}^{m\times n}$ şi o matrice $A\mathbb{R}^{n\times k}$, se calculeaza matricea

$$C \leftarrow \alpha * A * A^T + \beta * C.$$

Dacă $k < n$ şi A are coloanele liniar independente, atunci rang$AA^T = k$. Acest lucru dă numele procedurii.

3. Actualizarea de rang $2k$ a unei matrici simetrice: fiind daţi scalari reali α şi β, o matrice simetrică $C \in \mathbb{R}^{m\times n}$ şi două matrice $n \times k$ A şi B, procedura calculează:

$$C \leftarrow \alpha * A * B^T + \alpha * B * A^T + \beta * C.$$

Nivelul 3 conţine şi algoritmi de rezolvare a sistemelor liniare triunghiulare cu membrul drept multiplu (vezi seminar 2) care au o complexitate $O(n^3)$.

1.4 Probleme rezolvate

Vom începe cu probleme simple, cum sunt cele rezolvate de BLAS, dar care sunt des întâlnite în probleme mai complexe. Programele MATLAB de implementare a algoritmilor elaboraţi sunt prezentate în Anexa B.

Problema 1. Fiind date numerele reale $\alpha_1, \alpha_2, \ldots, \alpha_n$, scrieţi un algoritm care să calculeze suma lor $\sigma = \sum_{i=1}^{n} \alpha_i$ şi un alt algoritm care să calculeze produsul lor $\pi = \prod_{i=1}^{n} \alpha_i$.

Soluţie. Amintiţi-vă că o sumă este întotdeauna iniţializată cu 0 şi un produs cu 1. Algoritmi posibili pentru a calcula σ şi π sunt:

Algoritmul 1.1 (Date α_i, $i = 1 : n$, algoritmul calculează $\sigma = \sum_{i=1}^{n} \alpha_i$)

1. $\sigma = 0$
2. **pentru** $i = 1 : n$
    1. $\sigma = \sigma + \alpha_i$

Algoritmul 1.2 (Date α_i, $i = 1 : n$, algoritmul calculeaza $\sigma = \sum_{i=1}^{n} \alpha_i$)

1. $\sigma = 0$
2. **pentru** $i = n : -1 : 1$
    1. $\sigma = \sigma + \alpha_i$

Observaţi că cei doi algoritmi pentru calcularea sumei sunt *diferiţi*. Datorită erorilor de rotunjire, care apar la efectuarea fiecărei adunări, rezultatele executării celor doi algoritmi pot fi diferite. Adunarea în virgulă mobilă nu este o operaţie asociativă. Este foarte greu de găsit ordinea optimă de însumare astfel încât eroarea să fie minimă.

Algoritmul 1.3 (Date α_i, $i = 1 : n$, algoritmul calculează $\pi = \prod_{i=1}^{n} \alpha_i$)

1. $\pi = 1$
2. **pentru** $i = 1 : n$
    1. $\pi = \pi * \alpha_i$

Evident, observaţiile de mai sus rămân valabile.

Problema 2. Fiind dată o matrice superior triunghiulară $U \in \mathbb{R}^{n\times n}$, scrieţi un algoritm eficient care să calculeze

1. urma matricei U, i.e. $\tau = \operatorname{tr} U$;
2. determinantul $\delta = \det U$
3. normele Frobenius, 1 şi infinit ale matricei U definite în felul următor:

$$\nu_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} u_{ij}^2} \qquad \text{(normaFrobenius)},$$

$$\nu_1 = \max_{j\in 1:n} \sum_{i=1}^{n} |u_{ij}| \qquad \text{(norma1)}$$

şi

$$\nu_\infty = \max_{i\in 1:n} \sum_{j=1}^{n} |u_{ij}| \qquad \text{(norma } \infty\text{)}.$$

Soluţie. Prin definiţie $\tau = \sum_{i=1}^{n} U(i,i)$. De asemenea, este uşor de demonstrat că $\delta = \prod_{i=1}^{n} U(i,i)$. Astfel puteţi folosi algoritmii de calcul a sumei, respectiv a produsului a n numere reale de la problemele precedente. Pentru a putea calcula eficient norma matricei vom exploata structura superior triunghiulară a matricei U evitând efectuarea unor operaţii inutile. Astfel avem

$$\nu_F = \sqrt{\sum_{i=1}^{n}\sum_{j=i}^{n} u_{ij}^2}, \qquad \nu_1 = \max_{j\in 1:n} \sum_{i=1}^{j} |u_{ij}|, \qquad \nu_\infty = \max_{i\in 1:n} \sum_{j=i}^{n} |u_{ij}|.$$

Prin urmare, algoritmul este:

Algoritmul 1.4

1. $\nu_1 = 0$
2. **pentru** $j = 1 : n$
    1. $\sigma = 0$
    2. **pentru** $i = 1 : j$
        1. $\sigma = \sigma + |u_{ij}|$
    3. **dacă** $\nu_1 < \sigma$
        1. $\nu_1 = \sigma$
3. $\nu_\infty = 0$
4. **pentru** $i = 1 : n$
    1. $\sigma = 0$

1. **pentru** $j = i : n$
2. $\sigma = \sigma + |u_{ij}|$

2. **dacă** $\nu_\infty < \sigma$
1. $\nu_\infty = \sigma$

5. $\nu_F = 0$
6. **pentru** $i = 1 : n$
1. **pentru** $j = i : n$
1. $\nu_F = \nu_F + u_{ij}^2$

7. $\nu_F = \sqrt{\nu_F}$

Problema 3. a) Fie doi vectori necoliniari daţi $b_1, b_2 \in \mathbb{R}^n$. Calculaţi un vector $q \in \mathbb{R}^n$ aflat în subspaţiul generat de b_1 şi b_2, ortogonal la b_1. b) Fie p vectori liniar independenţi $b_j \in \mathbb{R}^n,\ j = 1 : p$. Calculaţi un set de p vectori $q_j,\ j = 1 : p$ din subspaţiul generat de vectorii b_j, ortogonali doi câte doi, i.e. astfel încât $q_i^T q_j = 0, \forall i \neq j,\ q_j \in \mathbb{R}^n,\ j = 1 : p$.

Soluţie. a) Orice vector din subspaţiul generat de vectorii b_1 şi b_2 este o combinaţie liniară a acestor doi vectori. Fie q o astfel de combinaţie liniară a lui b_1 şi b_2, i.e. $q = \alpha b_1 + b_2$. Pentru a determina α avem condiţia de ortogonalitate $b_1^T q = \alpha b_1^T b_1 + b_1^T b_2 = 0$. Vectorul b_1 fiind nenul, rezultă $\alpha = -\dfrac{b_1^T b_2}{b_1^T b_1}$. Algoritmul este:

1. $\alpha = 0$
2. $\beta = 0$
3. **pentru** $i = 1 : n$
1. $\alpha = \alpha + b_1(i) * b_2(i)$
2. $\beta = \beta + b_1(i) * b_1(i)$

4. $\alpha = -\alpha/\beta$
5. **pentru** $i = 1 : n$
1. $q(i) = \alpha * b_1(i) + b_2(i)$

b) Binecunoscuta procedură de ortogonalizare Gram-Schmidt utilizează şi generalizează rezultatul de la punctul a); vectorii ortogonali q_j generează acelaşi subspaţiu al lui $\mathbb{R}^n$ ca şi setul de vectori b_j. Ideea algoritmului Gram-Schmidt constă în a impune $q_1 = b_1$ şi a exprima un vector q_{j+1} ca o combinaţie liniară a vectorilor q_k deja calculaţi şi b_{j+1}, conform schemei de calcul la nivel vectorial:

1. $q_1 = b_1$
2. **pentru** $j = 1 : p - 1$
1. $q_{j+1} = \sum_{i=1}^{j} \alpha_{ij} q_i + b_{j+1}$

unde scalarii $\alpha_{ij},\ i = 1 : j$ trebuie să asigure ortogonalitatea vectorului q_{j+1} la $q_k,\ k = 1 : j$. La fel ca mai sus, $\alpha_{11} = -\dfrac{b_1^T b_2}{b_1^T b_1}$. Mai general, având în vedere că $q_k^T q_{j+1} = \sum_{i=1}^{j} \alpha_{ij} q_k^T q_i + q_k^T b_{j+1} = 0$ şi că $q_k^T q_i = 0$ pentru $k \neq i$ avem

$$\alpha_{ij} = -\frac{q_i^T b_{j+1}}{q_i^T q_i}, \qquad i = 1 : j$$

Astfel algoritmul Gram-Schmidt detaliat este:

Algoritmul 1.5 (Gram-Schmidt) Fiind daţi vectorii liniar independenţi $b_j,\ j = 1 : p$, algoritmul calculează un set de p vectori ortogonali $q_j,\ j = 1 : p$ care formează o bază ortogonală a subspaţiului generat de vectorii $b_j,\ j = 1 : p$.

1. **pentru** $k = 1 : n$
   1. $q_1(k) = b_1(k)$
2. **pentru** $j = 1 : p - 1$
   1. $\beta_j = 0$
   2. **pentru** $k = 1 : n$
      1. $\beta_j = \beta_j + q_j(k) * q_j(k)$
      2. $q_{j+1}(k) = b_{j+1}(k)$
   3. **pentru** $i = 1 : j$
      1. $\alpha_{ij} = 0$
      2. **pentru** $k = 1 : n$
         1. $\alpha_{ij} = \alpha_{ij} + q_i(k) * b_{j+1}(k)$
      3. $\alpha_{ij} = -\alpha_{ij}/\beta_i$
      4. **pentru** $k = 1 : n$
         1. $q_{j+1}(k) = q_{j+1}(k) + \alpha_{ij} q_i(k)$

Observaţi că algoritmul Gram-Schmidt conţine multe operaţii vector-vector cum sunt produsele SAXPY şi DOT, implementate profesional la nivelul 1 BLAS.

La curs vom prezenta un algortitm mai bun pentru rezovarea aceleiaşi probleme.

Problema 4. Fie o matrice $A \in \mathbb{R}^{m\times n}$, un vector $b \in \mathbb{R}^n$ şi un vector $c \in \mathbb{R}^m$, toate date. Scrieţi un algoritm eficient care să calculeze $c \leftarrow c + A * b$. (Daca iniţial $c = 0$, atunci aceasta este o problemă de înmultire matrice-vector $c = A * b$).

Soluţie. Vom da doi algoritmi: primul bazat pe algoritmul DOT de calcul al produsului scalar a doi vectori (un algortim orientat pe linii) şi un al doilea bazat pe SAXPY (un algoritm orientat pe coloane). Pentru primul algoritm vom partiţiona matricea A pe linii şi vom folosi formula

$$c(i) \leftarrow c(i) + A(i,:) * b = c(i) + \sum_{j=1}^{n} a_{ij} b_j, \qquad i = 1 : m.$$

Algoritmul, numit versiunea (i, j) a lui GAXPY, este:

Algoritmul 1.6 GAXPY pe linii: Fiind dată o matrice $A \in \mathbb{R}^{m\times n}$ şi vectorii $b \in \mathbb{R}^n$ şi $c \in \mathbb{R}^m$, algoritmul calculează $c \leftarrow c + A * b$

1. **pentru** $i = 1 : m$
   1. **pentru** $j = 1 : n$
      1. $c_i = c_i + a_{ij} * b_j$

Pentru al doilea algoritm vom partiţiona matricea A pe coloane şi vom folosi formula

$$c \leftarrow c + \sum_{j=1}^{n} A_{:,j} b_j.$$

Algoritmul, numit versiunea (j, i) a lui GAXPY, este:

Algoritmul 1.7 GAXPY pe coloane: Fiind dată o matrice $A \in \mathbb{R}^{m\times n}$ şi vectorii $b \in \mathbb{R}^n$ şi $c \in \mathbb{R}^m$, algoritmul calculează $c \leftarrow c + A * b$

1. **pentru** $j = 1 : n$
    1. **pentru** $i = 1 : m$
        1. $c_i = c_i + a_{ij} * b_j$

Complexitatea celor doi algoritmi este aceeaşi $N_{op} \approx 2mn$ dar există diferenţe între ei: primul este recomandat când matrica A este memorată pe linii şi al doilea când matricea A este memorată pe coloane.

Problema 5. Fie trei matrice $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$ şi $C \in \mathbb{R}^{m\times p}$ date, scrieţi un algoritm eficient care să calculeze $C \leftarrow C + A * B$. (Dacă iniţial $C = 0$, atunci aceasta este o problemă de înmulţire matrice-matrice $C = A * B$).

Soluţia. Soluţia standard se bazează pe formula binecunoscută

$$c_{ij} \leftarrow c_{ij} + \sum_{k=1}^{n} a_{ik} * b_{kj}, \qquad i = 1 : m, \quad j = 1 : p.$$

Având trei indici (i, j, k) putem scrie $3! = 6$ algoritmi esenţial diferiţi. Complexitatea acestor algoritmi este aceeaşi $N_{op} \approx 2mnp$. Prezentăm trei dintre aceştia cu scurte comentarii.

a) Versiunea (i, j, k) se bazează pe partiţionarea lui A pe linii şi a lui B pe coloane. Ultimul ciclu intern calculează un produs DOT şi ultimele două cicluri calculează o operaţie GAXPY-linie, i.e. $C(i, j) \leftarrow C(i, j) + A(i, :) \cdot B(:, j)$ respectiv $C(i, :) \leftarrow C(i, :) + A(i, :) \cdot B$.

Algoritmul 1.8 Versiunea (i, j, k) a înmulţirii matrice-matrice. Date $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$ şi $C \in \mathbb{R}^{m\times p}$ algoritmul calculează $C \leftarrow C + A * B$.

1. **pentru** $i = 1 : m$
    1. **pentru** $j = 1 : p$
        1. **pentru** $k = 1 : n$
            1. $c_{ij} = c_{ij} + a_{ik} * b_{kj}$

b) Versiunea (j, k, i) se bazează pe partiţionarea lui A şi a lui B pe coloane. Ultimul ciclu intern calculează o SAXPY-coloană şi ultimele două cicluri calculează o GAXPY-coloană, i.e. $C(:, j) \leftarrow C(:, j) + A(:, k) \cdot B(k, j)$, respectiv $C(:, j) \leftarrow C(:, j) + A \cdot B(:, j)$.

Algoritmul 1.9 Versiunea (j, k, i) a înmulţirii matrice-matrice.

1. **pentru** $j = 1 : p$
    1. **pentru** $k = 1 : n$
        1. **pentru** $i = 1 : m$
            1. $c_{ij} = c_{ij} + a_{ik} * b_{kj}$

c) Versiunea (k, i, j) se bazează pe partiţionarea lui A pe coloane şi a lui B pe linii. Ultimul ciclu calculează un produs SAXPY-linie şi ultimele două cicluri calculează produsul OUT a doi vectori, concret $C(i, :) \leftarrow C(i, :)+A(i, k)\cdot B(k, :)$, respectiv $C \leftarrow C+A(:, k)\cdot B(k, :)$.

Algoritmul 1.10 Versiunea (k, i, j) a înmulţirii matrice-matrice.

1. **pentru** $k = 1 : m$
    1. **pentru** $i = 1 : p$
        1. **pentru** $j = 1 : n$
            1. $c_{ij} = c_{ij} + a_{ik} * b_{kj}$

Problema 6. Fie $L \in \mathbb{R}^{n\times n}$ o matrice inferior triunghiulară şi $U \in \mathbb{R}^{n\times n}$ o matrice superior triunghiulară, ambele date. Scrieţi un algoritm eficient care să calculeze $A = L*U$. (Problema inversă: fiind dată A, calculaţi L şi U, este cunoscută ca problema factorizării LU şi va fi tratată la curs).

Soluţia. Pentru a obţine un algoritm eficient trebuie să evităm calculele inutile cum sunt înmulţirile şi adunările cu zero. Astfel, avem

$$A(i,j) = \sum_{k=1}^{n} L(i,k) * U(k,j) = \sum_{k=1}^{\min(i,j)} L(i,k) * U(k,j),$$

şi ca să evitam comparaţia dintre i şi j vom împarţi calculul în calcularea părţii inferior triunghiulare şi a celei superior triunghiulare a lui A. Algoritmul este:

Algoritmul 1.11

1. **pentru** $i = 1 : n$
    1. **pentru** $j = 1 : i$
        1. $a_{ij} = 0$
        2. **pentru** $k = 1 : j$
            1. $a_{ij} = a_{ij} + l_{ik} * u_{kj}$
    2. **pentru** $j = i + 1 : n$
        1. $a_{ij} = 0$
        2. **pentru** $k = 1 : i$
            1. $a_{ij} = a_{ij} + l_{ik} * u_{kj}$

Numărul de operaţii necesare pentru algoritmul de mai sus este

$$N_{op} = \sum_{i=1}^{n}(\sum_{j=1}^{i} 2j + \sum_{j=i+1}^{n} 2i) \approx \frac{2}{3}n^3,$$

de trei ori mai mic decât înmulţirea matrice-matrice standard.

Problema 7. a. Fie matricele $U, V \in \mathbb{R}^{n\times n}$ date. Propuneţi un algoritm eficient pentru a calcula matricea:

$$X = (I_n + U(:,1) * V(1,:)) * (I_n + U(:,2) * V(2,:)) * \cdots * (I_n + U(:,n) * V(n,:)).$$

b Scrieţi un algoritm eficient care să calculeze determinantul matricei X calculate la punctul a).

Soluţia a) Scriem mai întâi un algoritm eficient pentru calcularea produsului $X \leftarrow X * (I_n + uv^T)$, unde X este o matrice dată iar $u, v \in \mathbb{R}^n$ sunt vectori coloană daţi. Evident $Y = X * (I_n + uv^T) = X + X * u * v^T$, astfel că $Y(i,j) = X(i,j) + X(i,:) * u * v(j) = X(i,j) + \sigma * v(j)$ unde scalarul $\sigma = X(i,:) * u = \sum_{k=1}^{n} X(i,k)u(k)$ depinde de i dar nu depinde de j. Avem astfel următorul algoritm eficient:

Algoritmul 1.12. Fiind dată matricea $X \in \mathbb{R}^{n\times n}$ şi vectorii coloană $u, v \in \mathbb{R}^n$, acest algoritm calculează $X \leftarrow X * (I_n + uv^T)$.

1. **pentru** $i = 1 : n$
   1. $\sigma = 0$
   2. **pentru** $k = 1 : n$
      1. $\sigma = \sigma + x_{ik} * u_k$
   3. **pentru** $j = 1 : n$
      1. $x_{ij} = x_{ij} + \sigma * v_j$

Acum, introducând numele **Xuv** pentru procedura de mai sus şi apelând secvenţa de instrucţiuni de mai sus prin

$$X = \mathbf{Xuv}(X, u, v)$$

putem rezolva problema folosind urmatoarea procedură:

Algoritmul 1.13

1. $X = I_n$
2. **pentru** $k = 1 : n$
   1. $X = \mathbf{Xuv}(X, U(:,k), V(k,:))$

Observaţi că algoritmul 1.12 necesită $O(n^2)$ operaţii, deci am rezolvat problema înmulţirii a n matrice cu o complexitate de numai $O(n^3)$ flopi.

b) Metoda eficientă de a calcula determinantul se bazează pe faptul că dacă $u, v \in \mathbb{R}^n$ sunt vectori coloană, atunci $\det(I_n + uv^T) = 1 + v^T u$ (demonstraţi, spre exemplu folosind inducţia matematică). Aşadar,

$$\det X = \prod_{k=1}^{n} (1 + V(k,:) * U(:,k)) = \prod_{k=1}^{n} (1 + \sum_{i=1}^{n} V(k,i) * U(i,k))$$

şi algoritmul este

Algoritmul 1.14

1. $\delta = 1$
2. **pentru** $k = 1 : n$
   1. $\nu = 1$
   2. **pentru** $k = 1 : n$
      1. $\nu = \nu + v_{ki} * u_{ik}$
   3. $\delta = \delta * \nu$

1.5 Probleme propuse

Problema 1.1 Fie două matrice $A, B \in \mathbf{R}^{mn\times n}$. Produsul scalar dintre A şi B este definit de $\sigma =< A, B >= \text{trace}(B^T A)$. Demonstraţi că $\sigma = \sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}b_{ij}$ şi scrieţi un algoritm care să calculeze σ.

Problema 1.2 Fie $u, v \in \mathbf{R}^n$ doi vectori nenuli şi fie $A = I_n + uv^T$. Găsiţi un algoritm eficient care să calculeze norma Frobenius a lui A.

Problema 1.3 Fie $A, B, C \in \mathbb{R}^{n\times n}$ trei matrice date şi $d \in \mathbb{R}^n$ un vector dat. Scrieţi un algoritm eficient care să calculeze vectorul $x = A * B * C * d$.

Problema 1.4 Fie $A, B \in \mathbb{R}^{n\times n}$ două matrice date. Scrieţi algoritmi eficienţi pentru calcularea $A \leftarrow A * B$, şi a $B \leftarrow A * B$ folosind cât mai puţin spaţiu de memorie. Repetaţi problema când A şi B sunt amândouă inferior triunghiulare (superior triunghiulare).

Problema 1.5 Se consideră date matricele: $H \in \mathbb{R}^{n\times n}$ superior Hessenberg, $R \in \mathbf{R}^{n\times n}$ superior triunghiulara şi vectorul $b \in \mathbb{R}^n$.

a. Demonstraţi că matricele $X = H * R$ şi $Y = R * H$ sunt ambele superior Hessenberg.

b. Scrieţi algoritmi eficienţi pentru calculul matricelor X si Y; studiaţi posibilitatea de a suprascrie H cu X (cu Y).

c. Scrieţi algoritmi eficienţi pentru a calcula vectorii $x = HRb$ şi $y = RHb$.

Problema 1.6 Fie $C \in \mathbb{R}^{n\times n}$, o matrice simetrică dată şi $A \in \mathbf{R}^{n\times p}$ o matrice oarecare. Scrieţi un algoritm eficient care să calculeze actualizarea de rang p a lui C, adică $C \leftarrow C + A * A^T$. Demonstraţi că dacă iniţial C este zero, atunci matricea C calculată este simetrică şi pozitiv semi-definită.

Problema 1.7 Fie $C \in \mathbb{R}^{n\times n}$ o matrice simetrică şi matricele $A, B \in \mathbf{R}^{n\times p}$. Scrieţi un algoritm eficient care să calculeze actualizarea de rang $2p$ a lui C, adică $C \leftarrow C + A * B^T + B * A^T$.

Problema 1.8 Fie$\in \mathbf{R}^{n\times n}$ o matrice inferior triunghiulară nesingulară. Scrieţi un algoritm eficient care să calculeze $T = LL^T$ şi demonstraţi că T este o matrice simetrică pozitiv-definită.

1.6 Bibliografie

1. B. Jora, B. Dumitrescu, C. Oară, Numerical Methods, UPB, Bucureşti, 1995.
2. G.W. Stewart, Introduction to matrix computation, Academic Press, 1973.
3. G. Golub, Ch. Van Loan, Matrix Computations, a treia editie, John Hopkins University Press, 1998.
4. G.W. Stewart, Matrix Algorithms, vol.1: Basic Decompositions, SIAM, 1999.
5. G.W. Stewart, Matrix Algorithms, vol.2: Eigensystems, SIAM, 2001.
6. B. Dumitrescu, C. Popeea, B. Jora, Metode de calcul numeric matriceal. Algoritmi fundamentali, ALL, Bucureşti, 1998.

1.7 Anexa B

```
1:  function [sigma] = suma1(alfa)
2:
3:  %------------------------------------------------------------------------
4:  % Algoritmul 1.1
5:  % Functia calculeaza suma a n numere reale stocate in vectorul alfa
6:  % numerele sunt insumate in ordine normala de la alfa(1) pana la alfa(n)
7:  % Apelul:  [sigma] = suma1(alfa)
8:  %
9:  % Buta Valentin, aprilie, 2006
10: %------------------------------------------------------------------------
11:
12: n=length(alfa);
13: m=min(size(alfa));
14: if m~=1
15:     error('Datele de intrare nu sunt corecte')
16: end
17: sigma=0;
18: for i=1:n
19:     sigma=sigma+alfa(i);
20: end
```

```
1:  function [sigma] = suma2(alfa)
2:
3:  %------------------------------------------------------------------------
4:  % Algoritmul 1.2
5:  % Functia calculeaza suma a n numere reale stocate in vectorul alfa
6:  % numerele sunt insumate in ordine inversa de la alfa(n) pana la alfa(1)
7:  % Apelul:  [sigma] = suma2(alfa)
8:  %
9:  % Buta Valentin, aprilie, 2006
10: %------------------------------------------------------------------------
11:
12: n=length(alfa);
13: m=min(size(alfa));
14: if m~=1
15:     error('Datele de intrare nu sunt corecte')
16: end
17: sigma=0;
18: for i=n:-1:1
19:     sigma=sigma+alfa(i);
20: end
```

```
1:  function [p] = produs(alfa)
2:
```

```
%-------------------------------------------------------------------------
% Algoritmul 1.3
% Functia calculeaza produsul a n numere reale stocate in vectorul alfa
% numerele sunt inmultite in ordine normala de la alfa(1) pana la alfa(n)
% Apelul:  [p] = produs(alfa)
%
% Buta Valentin, aprilie, 2006
%-------------------------------------------------------------------------

n=length(alfa);
m=min(size(alfa));
if m~=1
    error('Datele de intrare nu sunt corecte')
end
p=1;
for i=1:n
    p=p*alfa(i);
end
```

```
function [v1,vi,vF]=norme(U)
%-------------------------------------------------------------------------
% Algoritmul 1.4
% Functia calculeaza norma unu, norma infinit si norma Frobenius a unei
% matrice superior triunghiulara U. Norma unu este notata cu v1, norma
% infinit cu vi iar norma Frobenius cu vF.
% Apelul:  [v1,vi,vF] = norme(U)
%
% Buta Valentin, aprilie, 2006
%-------------------------------------------------------------------------
[n,m]=size(U);
if n~=m
    error('Matricea nu este patratica!');
end
for i=2:n-1
    for j=1:i-1
        if U(i,j)~=0
            error ('Matricea nu este superior triunghiulara');
        end
    end
end
v1=0;
for j=1:n
    sigma=0;
    for i=1:j
        sigma =sigma + abs(U(i,j));
    end
    if v1<sigma
        v1=sigma;
    end
end
```

```
32:   vi=0;
33:   for i=1:n
34:       sigma=0;
35:       for j=i:n
36:           sigma =sigma + abs(U(i,j));
37:       end
38:       if vi < sigma
39:           vi = sigma;
40:       end
41:   end
42:   vF=0;
43:   for i=1:n
44:       for j=i:n
45:           vF=vF + U(i,j)*U(i,j);
46:       end
47:   end
48:   vF=sqrt(vF);
```

```
 1:   function [Q]=gramschmidt(B)
 2:
 3:   %------------------------------------------------------------------------
 4:   % Algoritmul 1.5
 5:   % Functia calculeaza p vectori ortogonali care genereaza acelasi subspatiu
 6:   % ca p vectori liniar independenti dati.
 7:   % Vectorii dati sunt memorati in coloanele matricei B, iar vectorii
 8:   % calculati sunt memorati in coloanele matricei Q.
 9:   % Algoritmul este cunoscut sub numele de Algoritmul Gram-Schmidt
10:   % Apelul:  [Q]=gramschmidt(B)
11:   %
12:   % Buta Valentin, aprilie, 2006
13:   %------------------------------------------------------------------------
14:
15:   [n,p]=size(B);
16:   for k=1:n
17:       Q(k,1)=B(k,1);
18:   end
19:   for j=1:p-1
20:       beta(j)=0;
21:       for k=1:n
22:           beta(j)=beta(j)+Q(k,j)*Q(k,j);
23:           Q(k,j+1)=B(k,j+1);
24:       end
25:       for i=1:j
26:           alfa(i,j)=0;
27:           for k=1:n
28:               alfa(i,j)=alfa(i,j)+Q(k,i)*B(k,j+1);
29:           end
30:           alfa(i,j)=-alfa(i,j)/beta(i);
31:           for k=1:n
32:               Q(k,j+1)=Q(k,j+1)+alfa(i,j)*Q(k,i);
```

```
33:           end
34:       end
35:   end
```

```
1:    function [c]=GAXPY1(A,b,c)
2:
3:    %-------------------------------------------------------------------------
4:    % Algoritmul 1.6
5:    % Functia actualizeaza vectorul (coloana) c cu produsul unei matrice
6:    % A cu un vector (coloana) b.  Algoritmul este cunoscut sub numele de GAXPY
7:    % pe linii.
8:    % Apelul:  [c]=GAXPY1(A,b)
9:    %
10:   % Buta Valentin, aprilie, 2006
11:   %-------------------------------------------------------------------------
12:
13:   [m,n]=size(A);
14:   [mb,nc]=size(b);
15:   if mb~=1
16:       error('b nu este vector coloana');
17:   end
18:   if nc~=n
19:       error('A si b nu pot fi inmultite');
20:   end
21:   [mc,nc]=size(c);
22:   if mc~=1
23:       error('c nu este vector coloana');
24:   end
25:   for i=1:m
26:       for j=1:n
27:           c(i)=c(i)+A(i,j)*b(j);
28:       end
29:   end
```

```
1:    function [c]=GAXPY2(A,b,c)
2:
3:    %-------------------------------------------------------------------------
4:    % Algoritmul 1.7
5:    % Functia actualizeaza vectorul (coloana) c cu produsul unei matrice
6:    % A cu un vector (coloana) b.  Algoritmul este cunoscut sub numele de GAXPY
7:    % pe coloane.
8:    % Apelul:  [c]=GAXPY2(A,b)
9:    %
10:   % Buta Valentin, aprilie, 2006
11:   %-------------------------------------------------------------------------
12:
13:   [m,n]=size(A);
14:   [mb,nc]=size(b);
```

```
15:   if mb~=1
16:       error('b nu este vector coloana');
17:   end
18:   if nc~=n
19:       error('A si b nu pot fi inmultite');
20:   end
21:   [mc,nc]=size(c);
22:   if mc~=1
23:       error('c nu este vector coloana');
24:   end
25:   for j=1:n
26:       for i=1:m
27:           c(i)=c(i)+A(i,j)*b(j);
28:       end
29:   end
30:   end
```

```
 1:   function [C]=MxMijk(A,B)
 2:
 3:   %-------------------------------------------------------------------------
 4:   % Algoritmul 1.8
 5:   % Functia calculeaza matricea C definita ca produsul a doua
 6:   % matrice A si B. Algoritmul se bazeaza pe partitionarea lui A pe linii,
 7:   % si a lui B pe coloane.
 8:   % Apelul:  [C]=MxMijk(A,B)
 9:   %
10:   % Buta Valentin, aprilie, 2006
11:   %-------------------------------------------------------------------------
12:
13:   [m,n]=size(A);
14:   [x,p]=size(B);
15:   if n~=x
16:   error('Matricele nu pot fi inmultite!')
17:   end
18:   for i=1:m
19:       for j=1:p
20:           C(i,j)=0;
21:       end
22:   end
23:   for i=1:m
24:       for j=1:p
25:           for k=1:n
26:               C(i,j)=C(i,j)+A(i,k)*B(k,j);
27:           end
28:       end
29:   end
30:   end
```

```
1:  function [C]=MxMjki(A,B)
2:
3:  %-------------------------------------------------------------------------
4:  % Algoritmul 1.9
5:  % Functia calculeaza matricea C definita ca produsul a doua
6:  % matrice A si B. Algoritmul se bazeaza pe partitionarea lui A si B pe
7:  % coloane.
8:  % Apelul:  [C]=MxMjki(A,B)
9:  %
10: % Buta Valentin, aprilie, 2006
11: %-------------------------------------------------------------------------
12:
13: [m,n]=size(A);
14: [x,p]=size(B);
15: if n~=x
16: error('Matricele nu pot fi inmultite!')
17: end
18: for i=1:m
19:     for j=1:p
20:         C(i,j)=0;
21:     end
22: end
23: for j=1:p
24:     for k=1:n
25:         for i=1:m
26:             C(i,j)=C(i,j)+A(i,k)*B(k,j);
27:         end
28:     end
29: end
```

```
1:  function [C]=MxMkij(A,B)
2:
3:  %-------------------------------------------------------------------------
4:  % Algoritmul 1.10
5:  % Functia calculeaza matricea C definita ca produsul a doua
6:  % matrice A si B. Algoritmul se bazeaza pe partitionarea lui A pe coloane
7:  % si a lui B pe linii.
8:  % Apelul:  [C]=MxMkij(A,B)
9:  %
10: % Buta Valentin, aprilie, 2006
11: %-------------------------------------------------------------------------
12:
13: [m,n]=size(A);
14: [x,p]=size(B);
15: if n~=x
16: error('Matricele nu pot fi inmultite!')
17: end
18: for i=1:m
19:     for j=1:p
20:         C(i,j)=0;
```

```
21:       end
22:   end
23:   for k=1:n
24:       for i=1:m
25:           for j=1:p
26:               C(i,j)=C(i,j)+A(i,k)*B(k,j);
27:           end
28:       end
29:   end
```

```
 1:   function [A]=inmultire_LU(L,U)
 2:
 3:   %------------------------------------------------------------------------
 4:   % Algoritmul 1.11
 5:   % Functia calculeaza eficient produsul a doua matrice patratice L si U,
 6:   % prima fiind inferior triunghiulara in timp ce a doua este superior
 7:   % triunghiulara.  Rezultatul este memorat in matricea A.
 8:   % Apelul:  [A]=inmultire_LU(L,U)
 9:   %
10:   % Buta Valentin, aprilie, 2006
11:   %------------------------------------------------------------------------
12:
13:   [n,x]=size(L);
14:   [m,y]=size(U);
15:   if n~=x
16:       error('Matricea L nu este patratica');
17:   end
18:   if m~=y
19:       error('Matricea U nu este patratica');
20:   end
21:   if n~=m
22:   error('Matricele nu pot fi inmultite!')
23:   end
24:   for i=1:n
25:       for j=1:i
26:           A(i,j)=0;
27:           for k=1:j
28:               A(i,j)=A(i,j)+L(i,k)*U(k,j);
29:           end
30:       end
31:       for j=i+1:n
32:           A(i,j)=0;
33:           for k=1:i
34:               A(i,j)=A(i,j)+L(i,k)*U(k,j);
35:           end
36:       end
37:   end
```

```
1:  function [X]=Xuv(X,u,v)
2:
3:  %-------------------------------------------------------------------------
4:  % Algoritmul 1.12
5:  % Algoritmul calculeaza eficient produsul a doua matrice patratice una
6:  % oarecate (X) in timp ce cea de a doua are o forma particulara (In+u*v)
7:  % unde u si v sunt doi vectori (primul de tip coloana iar al doilea de tip
8:  % linie).  Algoritmul suprascrie rezultatul in matricea X.
9:  % Apelul:  [X]=Xuv(X,u,v)
10: %
11: % Buta Valentin, aprilie, 2006
12: %-------------------------------------------------------------------------
13:
14: [n,m]=size(X);
15: if n~=m
16:     error('Matricea X nu este patratica');
17: end
18: for i=1:n
19:     sigma=0;
20:     for k=1:n
21:         sigma=sigma+X(i,k)*u(k);
22:     end
23:     for j=1:n
24:         X(i,j)=X(i,j)+sigma*v(j);
25:     end
26: end
```

```
1:  function [X]=prodUV(U,V)
2:
3:  %-------------------------------------------------------------------------
4:  % Algoritmul 1.13
5:  % Algoritmul calculeaza eficient produsul a n matrice de forma
6:  % (In+U(:,i)*V(i,:)), unde i=1:n, folosind procedura XUV
7:  % Apelul:  [X]=doi(U,V)
8:  %
9:  % Buta Valentin, aprilie, 2006
10: %-------------------------------------------------------------------------
11:
12: [n,x]=size(U);
13: if n~=x
14:     error('Matricea U nu este patratica');
15: end
16: [m,y]=size(V);
17: if m~=y
18:     error('Matricea V nu este patratica');
19: end
20: for i=1:n
21:     for j=1:n
22:         X(i,j)=0
23:     end
```

```
24:       X(i,i)=1;
25:   end
26:   for k=1:n
27:       X=Xuv(X,U(:,k),V(k,:));
28:   end
```

```
 1:   function [detX]=determinant(U,V)
 2:
 3:   %------------------------------------------------------------------------
 4:   % Algoritmul 1.14
 5:   % Functia calculeaza determinantul matricei X egala cu produsul a n elementi
 6:   % de tipul In+U(:,k)V(k,:).  Algoritmul se bazeaza pe faptul ca datorita formei
 7:   % matricei X determinantul putea fi calculat ca produsul a n factori de
 8:   % tipul 1+V(k,:)*U(:,k) unde k=1:n.%
 9:   % Apelul:  [detX]=determinant(V,U)
10:   %
11:   % Buta Valentin, aprilie, 2006
12:   %------------------------------------------------------------------------
13:
14:   [n,x]=size(U);
15:   [m,y]=size(V);
16:   if n~=x
17:       error('Matricea U nu este patratica');
18:   end
19:   if m~=y
20:       error('Matricea V nu este patratica');
21:   end
22:   detX=1;
23:   for k=1:n
24:       nu=1;
25:       for i=1:n
26:           nu=nu+V(k,i)*U(i,k);
27:       end
28:       detX=detX*nu;
29:   end
```