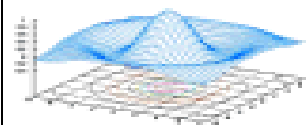


Metode numerice. Fișa de laborator nr. 1: ALGORITMI NUMERICI. CONDIȚIONARE. CALITATEA UNUI ALGORITM. TIPURI DE ERORI



1.1. ALGORITMI NUMERICI

Algoritmul numeric reprezintă un sistem de reguli, care, aplicat la o anumită clasă de probleme de același tip, conduce de la condițiile inițiale \mathbf{I} ale clasei de probleme, la soluția \mathbf{S} , cu ajutorul unor succesiuni finite de operații elementare (adunarea, înmulțirea, scăderea și împărțirea), strict ordonate și unic determinate.

Algoritmii numerici au la bază demonstrațiile matematice constructive care pun la dispoziție algoritmi matematici. În unele cazuri, corectitudinea matematică a unui algoritm trebuie adecvată calcului numeric practic (numărul operațiilor, memoria ocupată de date, stabilitate etc). Pot exista mai mulți algoritmi pentru rezolvarea unei probleme. Alegerea unui algoritm adecvat depinde și de arhitectura calculatorului.

Exemplul 1 \Rightarrow **Algoritmul DOT**: Calculul **produsului scalar** a doi vectori din \mathcal{R}^n . Se dau vectorii \mathbf{x} și \mathbf{y} din \mathcal{R}^n . Să se calculeze $\text{dot}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n x_i \cdot y_i$.

Algoritm:

1. Informația inițială: Citește vectorii \mathbf{x} și \mathbf{y}
2. DACĂ $\text{length}(\mathbf{x}) = \text{length}(\mathbf{y})$, ATUNCI execută instrucțiunea 3, ALTFEL afișează mesaj eroare
3. Inițializează valoarea produsului scalar (notat ps) cu 0: $\text{ps} \leftarrow 0$
4. Pentru $i = 1:n$, execută $\text{ps} \leftarrow \text{ps} + x_i * y_i$
5. Soluția problemei: Afișează valoarea produsului scalar ps

Calcul în GNU Octave: Exemplu pentru vectorii $\mathbf{x} = [1 \ 2 \ 3]$ și $\mathbf{y} = [4 \ 2 \ 9]$ din \mathcal{R}^3 .

octave#1> x=[1 2 3]; y=[-4 2 9]; ps1=dot(x,y), ps2=sum(x.*y), ps3=x*y'

Pentru problemele mai complicate se pot utiliza și **schemele logice** (forma grafică a fluxului de informații al algoritmului), care conțin: **operații de intrare/ieșire, atribuire, calcule elementare, transferuri și cicluri** (vezi exemplul din Problema 1).

1.2. CONDIȚIONARE

Condiționarea unei probleme apare în legătură cu acele probleme la care se cunosc doar aproximații ale datelor inițiale de intrare și caracterizează sensibilitatea soluției în raport cu perturbații mici ale datelor de intrare. De obicei nu cunoaștem soluția exactă, deci nu putem avea certitudinea că soluția aproximativă găsită este suficient de aproape de soluția exactă, decât dacă se controlează foarte bine tipul de condiționare a problemei.

Exemplul 2 \Rightarrow Să se rezolve **sistemul liniar** de mai jos, folosind **algoritmul de pivotare Gauss**:

$$\begin{cases} 4.0000 x + 0.8889 y = 4.0000 \\ 1.0000 x + 0.2222 y = 1.0000 \end{cases}$$

Să se rezolve de asemenea sistemul din Exemplul 2, renunțând la ultima zecimală din toate datele de intrare. Folosind același algoritm de calcul (pivotare Gauss), în primul caz soluția este: $x = 1$, $y = 0$, iar în cel de-al doilea caz, soluția este $x = 0.953030828641245$, $y = 0.211572843958356$. Este un exemplu clasic de **problemă slab condiționată** (dacă o problemă are proprietatea că o mică perturbație în una din date, sau în toate datele, conduce la mici perturbații în soluția numerică, atunci avem de a face cu o **problemă bine condiționată**; în caz contrar, problema este **slab condiționată**).

Slaba condiționare a sistemului din Exemplul 2 este datorată slabei condiționări a matricei sistemului. Dacă notăm cu \mathbf{A} matricea asociată sistemului, atunci în primul caz rezultă $\text{cond}(\mathbf{A}) = 178395.16$, iar în al doilea caz, $\text{cond}(\mathbf{A}) = 1.558717002732415e + 17$, unde $\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$ este **numărul de condiționare al**

matricei A , iar $\| \cdot \|$ este o **normă matriceală** consistentă (vezi Anexa 2).

1.3. CALITATEA UNUI ALGORITM

1.3.1. Numărul de operații al unui algoritm reprezintă numărul de operații aritmetice elementare (adunare, scădere, înmulțire, împărțire) necesare obținerii rezultatului. Spre exemplu, algoritmul descris în Exemplul 1 necesită $2n$ operații aritmetice elementare. Pentru evaluarea **vitezei calculatoarelor**, se măsoară numărul de operații în virgulă mobilă, pe secundă, denumit **FLOPS** (floating point operations per second).

1.3.2. Memoria ocupată: Pentru execuția unui algoritm este necesară memorarea datelor de intrare, a rezultatelor, precum și a altor valori intermediare. Numărul de elemente în **FVM (format virgulă mobilă** – vezi Anexa 1) necesare în acest scop se numește **memoria ocupată de algoritm**.

1.3.3. Stabilitatea numerică exprimă mărimea erorilor numerice introduse de un algoritm, în ipoteza că datele inițiale sunt exacte. Un algoritm corect trebuie să producă un răspuns unic și corect, determinat pentru date de intrare fixate.

Exemplul 3 \Rightarrow Să se rezolve **sistemul liniar** [Kahan W., *Numerical Methods*, Prentice Hall, 1974]:

$$\begin{cases} 1.2969x + 0.8648y = 0.8642 \\ 0.2161x + 0.1441y = 0.1440 \end{cases}$$

Folosind **algoritmul de pivotare Gauss**, obținem soluția $x = 2$, $y = -2$. Folosind un **algoritm iterativ** cu eroare impusă de 10^{-8} , se poate obține spre exemplu soluția $x = 0.9911$, $y = -0.4870$.

1.3.4. Siguranța în funcționare reprezintă capacitatea de a semnaliza situațiile în care rezultatul poate fi afectat de erori importante, datorate relei condiționări a problemei (vezi Exemplul 2). Decizia de a utiliza sau nu un astfel de rezultat revine utilizatorului, sau unui program expert.

1.3.5. Exactitatea unui algoritm este reprezentată atât prin **acuratețea rezultatelor**, care reflectă măsura în care valoarea calculată se apropie de valoarea adevărată, cât și prin **precizia rezultatelor**, care se referă la numărul de cifre semnificative din FVM.

1.4. TIPURI DE ERORI

1.4.1. Eroarea totală se compune din trei tipuri de erori: ① **erori inerente (inițiale)** – erori de măsurare, de manipulare sau de calcul preliminar a mărimilor ce reprezintă datele de intrare pentru algoritm, respectiv erorile de modelare a fenomenului fizic; ② **eroare de metodă** – metoda aleasă poate să fie neadecvată clasei de probleme matematice (vezi Exemplul 3); ③ **erori de calcul** – erori de reprezentare, erori de trunchiere și erori de rotunjire.

Fie x valoarea adevărată și x^* valoarea aproximată a lui x , obținută prin calcul sau măsurătoare. Când $x^* < x \Rightarrow x^*$ aproximează pe x **prin lipsă**, iar când $x^* > x \Rightarrow x$ este aproximat de către x^* **prin adaos**.

1.4.2. Eroarea absolută $|\varepsilon_x|$ redă doar mărimea aproximației făcute, ignorând semnul ei:

$$|\varepsilon_x| = |x - x^*|. \quad (1.1)$$

1.4.3. Eroarea relativă ε_{rx} este raportul dintre eroarea absolută $|\varepsilon_x|$ și valoarea absolută a lui x . Când nu se cunoaște x , la numitor se poate utiliza și x^* . Pentru evidențierea semnului erorii relative, se utilizează relația

$\varepsilon_{rx} = \frac{x - x^*}{x}$, care se poate scrie și în procente:

$$\varepsilon_{rx} = \frac{x - x^*}{x} 100 \text{ [%]}. \quad (1.2)$$

Dacă nu se cunoaște valoarea adevărată x , iar rezultatul final x^* este obținut în urma unui calcul iterativ, atunci între iterația k și $(k+1)$, eroarea absolută $|\varepsilon_a|$ și eroarea relativă ε_{ra} se calculează cu relațiile:

$$|\varepsilon_a| = |x_{k+1}^* - x_k^*| \quad \text{și} \quad \varepsilon_{ra} = \frac{x_{k+1}^* - x_k^*}{x_{k+1}^*} 100 \text{ [%]}. \quad (1.3)$$

Calculul iterativ se repetă până când: $|\varepsilon_{ra}| < \varepsilon_{ad}$. Pentru **eroarea admisibilă** ε_{ad} se recomandă formula:

$$\varepsilon_{ad} = 0.5 \cdot 10^{2-m} [\%], \quad (1.4)$$

în care m reprezintă **numărul de cifre semnificative**, de exemplu, numărul de zecimale exacte (identice) obținute pentru valoarea aproximată de la o iterație la alta. Dacă $m = 3$, calculele iterative vor fi efectuate până când $|\varepsilon_{ra}| < 0.05 \%$.

Exemplul 4 ⇒ **Erori de reprezentare și rotunjire**: Pe orice calculator obișnuit, spre exemplu în **GNU Octave**, calculul $(0.1 + 0.2 - 0.3)$ returnează rezultatul $\text{ans} = 5.55111512312578e-17$ și nu 0. Pericolul apare în cazul acumulării unor astfel de mici erori, acumulare care poate duce la degradarea uneori fatală a rezultatului. La fel, spre exemplu, calculul $(0.001 + 1 - 1)$ returnează rezultatul $\text{ans} = 0.000999999999999999$ în loc de 0.001.

Exemplul 5 ⇒ **Erori de reprezentare și rotunjire**: În \mathcal{R}^2 se dau vectorii $\mathbf{x} = [0.1 \ -0.2]$ și $\mathbf{y} = [-0.2 \ 0.1]$. Se observă că vectorii sunt ortogonali. În **GNU Octave**, secvența care folosește algoritmul **DOT** de la Exemplul 1, anume:

octave#1> format long; x=[0.1 -0.2]; y=[-0.2 0.1]; dot(x,y)

returnează rezultatul $\text{ans} = 0.040000000000000000$ și nu 0; aceasta se datorează atât erorilor numerice apărute în execuția algoritmului **DOT**, cât și erorilor de reprezentare asociate elementelor celor doi vectori.

Exemplul 6 ⇒ **Erori de trunchiere și rotunjire**: Să se calculeze valoarea lui e^x , cu $x = \{1; 2; 4\}$, folosind trunchierea cu primii n termeni ai **dezvoltării în serie Mac-Laurent** a funcției e^x , pentru $n = \{4; 10; 20\}$. Să se calculeze în fiecare caz în parte eroarea absolută $|\varepsilon_x|$ și eroarea relativă ε_{rx} în procente, față de valoarea «exactă». Să se scrie algoritmul și să se editeze o **funcție GNU Octave** (numită spre exemplu **fexp.m**) care să permită calculul direct a lui e^x pentru x și n date, oarecare.

Formularea matematică este:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!} + \dots \quad (1.5)$$

Algoritm:

1. Citește x și n
2. Generează vectorul $\mathbf{k} = [1 \ 2 \ 3 \ \dots \ n]$, adică: $\mathbf{k} \leftarrow 1:n$
3. Calculează suma S din membrul drept al (1.5), cu ajutorul produsului scalar între $1/k!$ și x^k :
 $S \leftarrow 1 + \text{dot}(1/k!, x^k)$
4. Calculează eroarea absolută: $\text{eroare} = \text{abs}(e^x - S)$
5. Calculează eroarea relativă: $\text{eroarer} = 100 * (e^x - S) / e^x$.

Funcție GNU Octave:

```
octave#1>function S=fexp(x,n)
>format long
>k=1:n;
>S=1+dot(1./cumprod(k),x.^k);
>eroare=abs(exp(x)-S)
>eroarer=100*(exp(x)-S)./exp(x)
>format
>endfunction
```

Rezultate: Pentru $x = \{1; 2; 4\}$ și $n = \{4; 10; 20\}$, valorile calculate cu ajutorul funcției **fexp(x,n)** sunt incluse în tabelul 1.

Tabelul 1

		$x = 1$	$x = 2$	$x = 4$
$n = 4$	valoare exactă e^x	2.71828182845905	7.38905609893065	54.5981500331442
	valoare S calculată cu (1.5)	2.70833333333333	7	34.3333333333333
	eroare absolută	0.00994849512571161	0.389056098930650	20.2648166998109
	eroare relativă [%]	0.365984682734360	5.26530173437112	37.1163064820127

n = 10	valoare S calculată cu (1.5)	2.71828180114638	7.38899470899471	54.4431040564374
	eroare absolută	2.73126601335605e-08	6.13899359427350e-05	0.155045976706845
	eroare relativă [%]	1.00477661468398e-06	0.000830822436868755	0.283976612051366
n = 20	valoare S calculată cu (1.5)	2.71828182845905	7.38905609893060	54.5981499281488
	eroare absolută	4.44089209850063e-16	4.61852778244065e-14	1.04995422134380e-07
	eroare relativă [%]	-1.63371290349908e-14	6.25049765572770e-13	1.92305823678351e-07

APLICAȚII DE LABORATOR

Problema 1 ⇒ Să se calculeze valoarea lui e^x pentru un x dat, cu o eroare absolută dată. Să se scrie algoritmul și să se editeze o funcție GNU Octave (numită spre exemplu **fexp1.m**) care să permită calculul direct a lui e^x pentru x și eroare absolută date, oarecare.

Formulare matematică: Considerăm dezvoltarea în serie Mac-Laurent a funcției e^x definită în (1.5). Fie șirul sumelor parțiale ale seriei e^x , anume $(S_n)_{n \in \mathbb{N}}$ astfel:

$$S_0 = 1, S_1 = S_0 + \frac{x}{1!}, \dots, S_n = S_{n-1} + \frac{x^n}{n!}, \dots$$

Calculul valorii lui e^x cu o precizie dată, impune calculul sumelor parțiale consecutive, până când $(S_n - S_{n-1}) < \text{eroare}$, cu afișarea lui S_n drept rezultat final.

Algoritm:

1. Citește x și eroare
2. Inițializări ciclu: sveche $\leftarrow 1$; $k \leftarrow 1$; diferența $\leftarrow 1$
3. PÂNĂ CÂND $\text{abs}(\text{diferența}) > \text{eroare}$, execută instrucțiunea 4
4. snoua = sveche + $x^k/k!$
 $k = k+1$
diferența = sveche - snoua
sveche = snoua
5. Afișează rezultate.

Funcție GNU Octave:

```

octave#1> function sveche=fexp1(x,eroare)
># Calculeaza si afiseaza (in format long si format short)
># e^x cu precizia eroare
># Afiseaza numarul de termeni ai seriei folositi
># Afiseaza e^x in format long si format short
>sveche=1;
>k=1;
>diferenta=1;
>while abs(diferenta)>eroare
> snoua=sveche+x^k/factorial(k);
> k=k+1;
> diferenta=sveche-snoua;
> sveche=snoua;
>endwhile
>format long
>disp('Numarul de termeni ai seriei luati in calcul k='),disp(k)
>disp('Valoarea calculata ='),disp(sveche)
>disp('Verificare: eroare='),disp(sveche-e^x)
>format
>endfunction

```

Problema 2 ⇒ Să se scrie un program de calcul în GNU Octave, pentru **algoritmul de sortare directă**.
Aplicație: Să se ordoneze crescător secvența de $n = 10$ numere reale: $\mathbf{x} = [7; 21; 83; 2; 15; 54; 37; 12; 43; 68]$.

Algoritm:

Se prezintă un **algoritm de sortare directă**, care necesită circa $n^{3/2}$ operații. Acest algoritm de ordonare crescătoare a unor numere include următoarele *etape*:

⇒ În secvența de n numere reale $\{x_1, x_2, \dots, x_n\}$, se determină cel mai mic element, denumit u :

$$u = \min\{x_1, x_2, \dots, x_n\} = x_r$$

⇒ Se permută u cu x_1 , adică se plasează u în locul lui x_1 iar x_1 trece pe poziția lui x_r , rezultând:

$$\{x'_1, x'_2, \dots, x'_r, \dots, x'_n\}, \text{ unde } x'_1 = u \text{ și } x'_k = x_k \text{ pentru } k \neq r \text{ și } 2 \leq k \leq n, \text{ iar } x'_r = x_1$$

⇒ Se repetă pașii de mai sus pentru secvența de numere $\{x'_2, x'_3, \dots, x'_n\}$, rezultând:

$$u = \min\{x'_2, x'_3, \dots, x'_n\} = x'_r \text{ și noua succesiune } \{x''_2, x''_3, \dots, x''_r, \dots, x''_n\}, \text{ cu } x''_2 = u \text{ și } x''_k = x'_k \text{ pentru } k \neq r, 3 \leq k \leq n, \text{ iar } x''_r = x'_2$$

⇒ Se continuă raționamentul și operațiile, până când secvența de elemente ce se compară mai conține doar două elemente, după care se realizează obiectivul urmărit de problemă.

Schema logică: Este prezentată în figura 1.

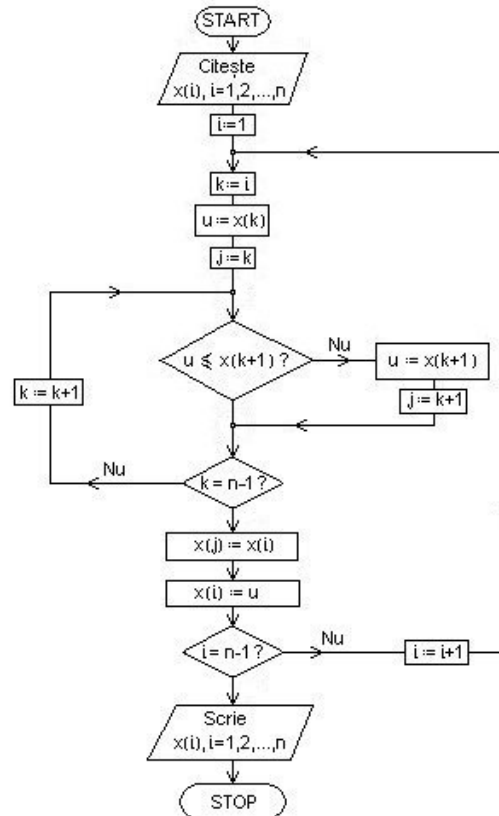


Figura 1 – Schema logică a algoritmului de sortare directă

Program GNU Octave:

```
octave#1> function sd=sortare(x)
>n=length(x)
>for i=1:n-1
>    j=i
>    u=x(i)
>    for k=i+1:n
>        if ( u > x(k) )
>            u=x(k)
>            j=k
>        endif
>    endfor
>    x(j)=x(i)
>    x(i)=u
>endfor
>endfunction
```

Problema 3 ⇒ Să se întocmească algoritmul și programul de calcul în GNU Octave, pentru **calculul sumei seriei Riemann**

$$S_4 = \frac{1}{1^4} + \frac{1}{2^4} + \frac{1}{3^4} + \dots + \frac{1}{n^4} + \dots$$

cu o eroare absolută de 0.00001. Să se efectueze verificarea, știind că seria este convergentă și că suma sa este $\pi^2/90$.

Problema 4 ⇒ Să se întocmească algoritmul și programul de calcul în GNU Octave, pentru calculul lui $\sin x$, pentru un x în radiani, dat și oarecare, cu o eroare absolută de 0.001. Se reamintește că dezvoltarea în serie Mac-Laurent a funcției $\sin x$ este:

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \dots$$

Problema 5 ⇒ Să se calculeze valoarea numerică aproximativă a lui $\cos x$, folosind trunchierea cu primii n termeni ai dezvoltării în serie Mac-Laurent a funcției $\cos x$, pentru $n = \{3; 9; 25\}$. Să se calculeze în fiecare caz în parte eroarea absolută $|\varepsilon_x|$ și eroarea relativă ε_{rx} , față de valoarea «exactă». Să se scrie algoritmul și să se editeze în linia de comandă o funcție GNU Octave (numită spre exemplu **fcos.m**) care să permită calculul direct a lui $\cos x$ pentru x și n date, oarecare.

Problema 6 ⇒ Să se demonstreze (folosind un contraexemplu) că adunarea în FVM nu este asociativă.

Exemplu în GNU Octave:

octave#1> (0.1+0.2)-0.3

ans = 5.5511e-17

octave#2>0.1+(0.2-0.3)

ans = 2.7756e-17

ANEXA 1: Despre reprezentarea numerelor în FVM (format virgulă mobilă)

Calculatorul poate opera cu numere întregi, fracționare și iraționale. Dacă operațiile se fac cu numere întregi, reprezentarea internă se va face în **FVF (format virgulă fixă)** - în acest caz nu se pune problema erorilor de rotunjire. Problema erorilor de rotunjire apare în cazul calculului efectuate cu numere fracționare și iraționale. În acest caz, reprezentarea internă a numerelor se face în **FVM**. Un număr N scris în **FVM** este compus dintr-o fracție M cu semn (**mantisă**) și un întreg e cu semn (**exponent**) astfel:

$$N = M \cdot b^e$$

unde b este **baza de numerație** și $M = \pm 0.f_1 f_2 \dots f_t$, cu t cifre; f_1 este diferit de zero; f_1, f_2, \dots, f_t sunt **simboluri** din baza în care se lucrează, iar $e = \pm e_1 e_2 \dots e_p$ are p cifre.

Exemplu ⇒ În baza 10, numărul 64.35 va fi reprezentat sub forma $0.6435 \cdot 10^2$, iar numărul -0.00156 sub forma $-0.156 \cdot 10^{-2}$.

Numerele în FVM sunt normalizate, adică prima cifra (după virgulă) din mantisă este diferită de zero.

ANEXA 2: Câteva tipuri de norme consistente

Norma Frobenius a matricei \mathbf{A} este $\|\mathbf{A}\|_F = (\mathbf{A}, \mathbf{A})^{1/2}$, iar $\|\mathbf{A}\|_F^2 = \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2$.

$\|\mathbf{A}\|_p = \max_{\|x\|_p=1} \|\mathbf{A}x\|_p$ reprezintă **p-norme** induse. În calcule se utilizează în special norma

$$\|\mathbf{A}\|_1 = \max_{j=1:n} \sum_{i=1:m} |a_{ij}| \text{ și norma } \|\mathbf{A}\|_\infty = \max_{i=1:m} \sum_{j=1:n} |a_{ij}|.$$