L1. Rezolvarea ecuațiilor

1.1. Metoda aproximațiilor succesive

1.1.1. Aspecte teoretice

Una dintre cele mai importante metode numerice pentru rezolvarea ecuațiilor algebrice şi transcendente este *metoda aproximațiilor succesive.* Această metodă poate fi utilizată pentru rafinarea aproximațiilor inițiale ale rădăcinilor furnizate de alte metode (cum ar fi, spre exemplu, metoda înjumătățirii sau metoda poziției false).

Presupunem că se cere rezolvarea ecuației

$$f(x) = 0, \quad (1.1.1)$$

unde *f(x)* este o funcție continuă pe un interval [a,b]. Metoda aproximațiilor succesive implică punerea acestei ecuații sub forma echivalentă:

$$x = \varphi(x), \quad (1.1.2)$$

astfel încât ecuațiile (1.1.1) şi (1.1.2) să aibă aceleaşi rădăcini. Punerea ecuației (1.1.1) sub forma (1.1.2) este întotdeauna posibilă, chiar dacă funcția *f(x)* nu conține în mod explicit termenul x (eventual, se poate aduna acest termen în ambii membri ai ecuației (1.1.1)).

Fie x_0 o aproximație inițială pentru rădăcina ξ a ecuației (1.1.2). Inlocuind această valoare în membrul drept al ecuației, se obține o aproximație îmbunătățită a rădăcinii

$$x_1 = \varphi(x_0).$$

Inlocuind x_1 în membrul drept al ecuației (1.1.2) se obține o nouă aproximație

$$x_2 = \varphi(x_1).$$

Repetând acest procedeu, se obține pornind de la x_0 secvența de numere

$$x_{i+1} = \varphi(x_i), \; i = 0,1,2,... \quad (1.1.3)$$

Dacă acest şir este convergent, atunci există limita $\xi = \lim x_i$. Trecând la limită în (1.1.3) şi admițând că $\varphi(x)$ este continuă, găsim

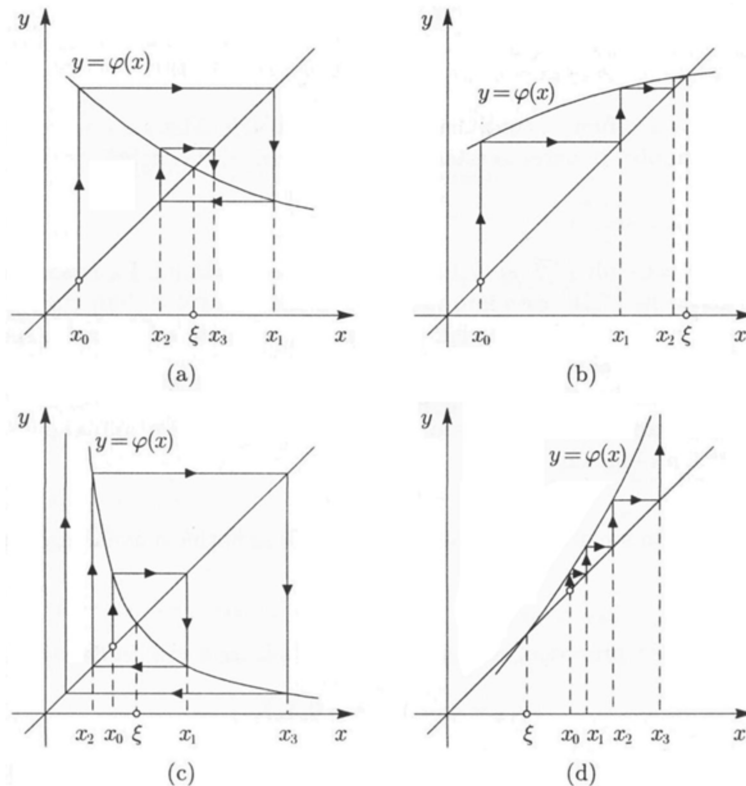
$$\lim_{i\to\infty} x_{i+1} = \varphi\left(\lim_{i\to\infty} x_i\right), \text{ sau } \xi = \varphi(\xi).$$

Prin urmare, ξ este rădăcină a ecuației (1.1.2) şi poate fi calculată pe baza relației de recurență (1.1.3) cu orice precizie.

Din punct de vedere geometric, o rădăcină reală ξ a ecuației $x = \varphi(x)$ este abscisa punctului de intersecție a curbei $y = \varphi(x)$ cu dreapta $y = x$ (prima bisectoare). Relativ la

valoarea derivatei $\varphi'(x)$ în vecinătatea rădăcinii, sunt posibile patru cazuri, iar modul cum rezultă şirul aproximațiilor succesive $x_1, x_2, x_3, ...$ plecând de la aproximația inițială x_0, este ilustrat in fig. 1.1.1. Astfel, în figurile (a) şi (b) procesul iterativ este convergent, iar în figurile (c) şi (d), procesul este divergent. Din studiul acestor figuri se poate trage concluzia că procesul este convergent şi deci metoda este aplicabilă numai în intervalele unde

$$|\varphi'(x)| < 1. \quad (1.1.4)$$



Pentru a putea aplica practic metoda aproximațiilor succesive trebuie îndeplinite condițiile prescrise de următoarea teoremă, care furnizează o condiție suficientă de convergență a şirului de iterare definit de relația (1.1.3).

Teorema 1.1.1 *Fie ecuația*

$$x = \varphi(x), \quad (1.1.5)$$

cu funcția $\varphi(x)$ *definită şi derivabilă pe [a,b]. Dacă este satisfăcută inegalitatea*

$$|\varphi'(x)| \le \lambda < 1 \quad (1.1.6)$$

pentru orice $x \in [a,b]$, *atunci şirul de iterare definit de relația*

$$x_{i+1} = \varphi(x_i), \; i = 0,1,2,3,... \quad (1.1.7)$$

converge către rădăcina (unică dacă există) $\xi \in [a,b]$ *a ecuației, indiferent de valoarea inițială* x_0.

Demonstrație: scădem egalitatea evidentă $\xi = \varphi(\xi)$ din relația (1.1.7)

$$x_{i+1} - \xi = \varphi(x_i) - \varphi(\xi).$$

Din teorema lui Lagrange rezultă că există un punct ζ cuprins intre x_i si ξ astfel încât

$$\varphi(x_i) - \varphi(\xi) = \varphi'(\zeta)(x_i - \xi).$$

Având în vedere inegalitatea (1.1.6), rezultă din cele două relații de mai sus

$$|x_{i+1} - \xi| \le \lambda |x_i - \xi|.$$

Dând pe rând lui i valorile 0,1,2,… obținem succesiv

$$|x_1 - \xi| \le \lambda |x_0 - \xi|$$

$$|x_2 - \xi| \le \lambda |x_1 - \xi| \le \lambda^2 |x_0 - \xi|$$

$$\cdots\cdots\cdots$$

$$|x_{i+1} - \xi| \le \lambda |x_i - \xi| \le \ldots \le \lambda^{i+1} |x_0 - \xi|.$$

Trecând la limită în ultima inegalitate şi având în vedere că $0 \le \lambda \le 1,$ rezultă

$$\lim_{i \to \infty} x_{i+1} = \xi$$

şi deci limita şirului este chiar rădăcina ξ a ecuației.

Pentru a arăta că în condițiile teoremei ξ este unica rădăcină din intervalul [a,b], presupunem contrariul, adică faptul că ar mai exista o rădăcină $\bar{\xi} \in [a,b]$, astfel încât $\bar{\xi} = \varphi\left(\bar{\xi}\right)$. Atunci

$$\xi - \bar{\xi} = \varphi(\xi) - \varphi\left(\bar{\xi}\right) = \varphi'(\zeta)\left(\xi - \bar{\xi}\right),$$

unde ζ se află între ξ şi $\bar{\xi}$. Mai putem scrie

$$\left(\xi - \bar{\xi}\right)[1 - \varphi'(\zeta)] = 0.$$

Cum prin ipoteză $1 - \varphi'(\zeta) \ne 0$, rezulta $\xi = \bar{\xi}$, adică rădăcina ξ este unică.

Din punct de vedere practic, ecuația $f(x) = 0$ poate fi pusă în mai multe moduri sub forma $x = \varphi(x)$. Oricum, pentru ca metoda aproximațiilor succesive să fie aplicabilă, trebuie să fie îndeplinită condiția (1.1.6). Cu cât va fi mai mic numărul λ, cu atât va fi mai rapidă convergența procesului iterativ către rădăcina ξ.

In general, ecuația $f(x) = 0$ poate fi înlocuită prin ecuația echivalentă

$$x = x - qf(x), \quad q > 0 \tag{1.1.8}$$

astfel încât $\varphi(x) = x - qf(x)$. Parametrul q trebuie ales în aşa fel încât să fie satisfăcută condiţia (1.1.6), adică

$$|\varphi'(x)| = |1 - qf'(x)| \leq \lambda < 1.$$

In cazul banal în care se poate alege $q = 1$ sau se include acest parametru în expresia funcţiei $f(x)$, ecuaţia (1.1.8) ia forma $x = x - f(x)$. In aceste condiţii, relaţia de recurenţă (1.1.3) a metodei aproximaţiilor succesive devine

$$x_{i+1} = x_i - f(x_i), \ i = 0,1,2,... \tag{1.1.9}$$

Din punct de vedere practic, se iterează pe baza acestei relaţii până când *diferenţa relativă* pentru două aproximaţii consecutive ale rădăcinii devine mai mică sau egală cu o anumită eroare maximă admisibilă ε,

$$\delta_i \equiv \left| \frac{\Delta x_i}{x_{i+1}} \right| \leq \varepsilon ,$$

unde $\Delta x_i \equiv x_{i+1} - x_i = -f(x_i)$ este corecţia rădăcinii. Rescrierea condiţiei de mai sus sub forma

$$|\Delta x_i| \leq \varepsilon |x_{i+1}| \tag{1.1.10}$$

elimina singularitatea, făcând posibilă tratarea unitară şi a cazului în care $x_{i+1} = 0$. Limitarea numărului de iteraţii la o anumită valoare $i_{\max}$ permite detectarea situaţiilor în care metoda este lent convergentă sau chiar divergentă.

Implementarea metodei aproximaţiilor succesive este ilustrată de funcţia **Iter** din următorul program:

```
#include <stdio.h>
#include <math.h>
double func(double x)
{
	return x - exp(-x);
}
int Iter(double Func(double), double *x)
{
/*
	Determine un zero real al unei functii reale prin metoda aproximatiilor succesive
	Func - functia utilizator
	*x -  aproximatie initiala (la intrare), zeroul gasit (la iesire)
	Returneaza indicele de eroare:		0 - executie normala
						1 - nr. maxim de iteratii depasit
						2 - proces divergent
*/
	const double eps = 1e-6;		/* criteriu relativ de precizie */
	const int itmax = 100;			/* nr. maxim de iteratii */
	double dx, f;
	int it;
```

```
	dx = -Func(*x);				/* initializeaza corectia */
	for (it = 1; it <= itmax; it++)
	{
		f = Func(*x);
		if (fabs(f) > fabs(dx))
			goto divergent;		/* compara noua corectie */
		dx = -f;				/* actualizeaza corectia */
		*x += dx;				/* noua aproximatie */
		if (fabs(dx) <= eps * fabs(*x))
			return 0;			/* testeaza convergenta */
	}
	printf("Iter: nr. maxim de iteratii depasit !\n");
	return 1;
divergent:
	printf("Iter: proces divergent!\n");
	return 2;
}
void main()
{
	double x;
	/* Caz test
	x - exp(-x) = 0;
	x0 = 0;
	Zeroul: 0.567143
	*/
	printf("x0 = ");
	scanf("%lf", &x);
	if (Iter(func, &x) == 0)
		printf("x = %lf \n", x);
}
```

Rutina utilizator func evaluează funcția $f(x)$. La intrarea în funcția **Iter** argumentul x trebuie sa conțină o aproximație inițială pentru rădăcina căutată. In momentul scăderii erorii rădăcinii sub limita admisă *eps* se iese din ciclul iterațiilor şi este returnat codul de eroare 0. Ieşirea normală din ciclu (după efectuarea tuturor celor *itmax* iterații) implică de fapt neatingerea preciziei prescrise datorită convergenței lente a procesului. In acest caz, codul de eroare returnat este 1.

Pentru a detecta timpuriu procesele divergente şi a nu parcurge în mod inutil numărul maxim de iterații *itmax*, imediat după ce este calculată noua valoare a funcției *f*, care (cu semn schimbat) are în metoda aproximațiilor succesive semnificația corecției rădăcinii, aceasta este comparată cu corecția de la pasul anterior, rămasă în variabila *dx*. In cazul în care corecția creşte în valoare absolută *(fabs(f)>fabs(dx))*, procesul este divergent şi se iese din ciclu şi din rutină cu indicele de eroare 2.

Ca exemplu care ilustrează modul crucial în care îndeplinirea condiției $|\varphi'(x)| \equiv |1 - f'(x)| < 1$ influențează convergența algoritmului, considerăm ecuația

$$x - e^{-x} = 0.$$

Alegând $f(x)=x-e^{-x}$, caz în care $|\varphi'(x)|=e^{-x}<1$, procesul iterativ al metodei aproximațiilor succesive converge către rădăcina x=0.567143. In schimb, considerând $f(x)=e^{-x}-x$, procesul iterativ diverge rapid datorită faptului că $|\varphi'(x)|=2+e^{-x}>1$. Prin urmare, deşi cele două alegeri ale funcției $f(x)$ implică ecuații echivalente din punct de vedere matematic, comportarea numerică a metodei aproximațiilor în cele două cazuri este net diferită.

1.1.2. Desfăşurarea laboratorului

1. se scrie algoritmul metodei aproximațiilor succesive şi se verifică pentru ecuația $x-e^{-x}=0$ considerând q=1.
2. se verifică divergența metodei utilizând aceeaşi ecuației de la punctul a) dar având q= -1.
3. să se obțină rădăcina ecuației $x-e^{-x}=0$ cu o precizie de 4 zecimale.
4. să se studieze convergența şi să se obțină rădăcina ecuației $x^3+x-1=0$ pentru q=0,5; q=1 şi q=2.

1.2. Metoda lui Newton (metoda tangentei)

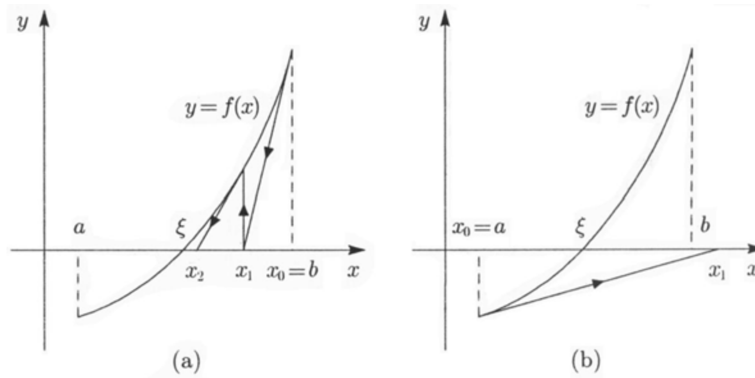
1.2.1. Aspecte teoretice

Metoda lui Newton, cunoscută şi sub numele de *metoda Newton-Raphson,* este probabil cea mai cunoscută dintre metodele unidimensionale pentru determinarea rădăcinilor. Acest lucru se datorează ideii simple care stă la baza ei, preluată sub diferite forme de diversele variante ale metodei, precum şi eficienței deosebite a algoritmilor corespunzători.

Presupunem că ecuația algebrică sau transcendentă

$$f(x)=0 \qquad (1.2.1)$$

are o rădăcină reală ξ izolată în intervalul [a,b] şi că $f'(x)$ şi $f''(x)$ sunt continue şi păstrează semnul pentru $x\in[a,b]$.



Având o aproximație inițială x_0 a rădăcinii ξ, metoda lui Newton permite găsirea unei aproximații îmbunătățite x_1, ca intersecție a tangentei dusă la curba $y = f(x)$ în punctul $(x_0, f(x_0))$ cu axa x, aşa după cum se poate observa din figura 1.2.1(a). Alegând $x_0 = b$, are loc în acest caz, $f'(x_0) = \dfrac{f(x_0)}{x_0 - x_1}$, de unde rezultă $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$.

Ducând tangenta în punctul de coordonate $(x_1, f(x_1))$, se obține o nouă aproximație x_2. Repetând procedeul pe baza relației

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)},\ i = 0,1,2,... \tag{1.2.2}$$

se găseşte un şir de aproximații $x_1, x_2, x_3, ...$ ale soluției ξ a ecuației (1.2.1). Introducând funcția de iterare a metodei lui Newton

$$\varphi(x) = x - \frac{f(x)}{f'(x)}, \tag{1.2.3}$$

relația de recurență (1.2.2) poate fi pusă sub o formă echivalentă metodei aproximațiilor succesive $x_{i+1} = \varphi(x_i),\ i = 0,1,2,...$

Condițiile convergenței acestui şir de aproximații către soluția ξ sunt precizate de teorema 1.1.1, demonstrată pentru metoda aproximațiilor succesive.

Se remarcă faptul că în cazul ilustrat în figura 1.2.1(a) are loc $f(x_0)f''(x_0) > 0$. Dacă se consideră ca aproximație inițială cealaltă extremitate a intervalului de căutare $(x_0 = a)$, ca în figura 1.2.1(b), are loc $f(x_0)f''(x_0) < 0$ şi se obține aproximația de ordinul întâi x_1 situată în afara intervalului [a, b]. Prin urmare, aproximația inițială $x_0 = a$ este în acest caz mai puțin avantajoasă, în sensul că sunt necesare mai multe iterații pentru ca şirul aproximațiilor rădăcinii să conveargă cu precizia dorită. Se poate demonstra ca o regulă

generală că, în vecinătatea rădăcinii, aproximația inițială cea mai avantajoasă satisface condiția $f(x_0)f''(x_0) > 0$.

In metoda lui Newton, ca şi în metoda aproximațiilor succesive, se iterează până când eroarea relativă corespunzătoare unor aproximații consecutive x_i şi x_{i+1} devine mai mică sau egală cu o eroare maximă admisibila ε, adică $\delta_i \equiv \left|\frac{\Delta x_i}{x_{i+1}}\right| \le \varepsilon$.

Expresia corecției rădăcinii Δx_i rezultă din relația de recurență (1.2.2)

$$\Delta x_i \equiv x_{i+1} - x_i = -\frac{f(x_i)}{f'(x_i)}. \tag{1.2.4}$$

Tratarea unitară a cazului în care $x_{i+i} = 0$ implică rescrierea condiției de convergență sub forma

$$|\Delta x_i| \le \varepsilon |x_{i+1}|. \tag{1.2.5}$$

Limitând numărul de iterații la o anumită valoare $i_{\max}$, pot fi detectate situațiile în care algoritmul este lent convergent sau divergent.

Cu deosebiri care vizează exclusiv modul de calcul al corecției rădăcinii, algoritmul metodei lui Newton este identic cu cel al metodei aproximațiilor succesive şi implementarea sa se regăseşte în funcția următoare:

```
int Newton(double Func(double,double *), double *x)
/*      Determine un zero real al unei functii reale prin metoda Newton-Raphson utilizand derivata analitica
        Func - functia utilizator (returneaza si derivata)
        *x - aproximatie initiala (la intrare), zeroul gasit (la iesire)
        Returneaza indicele de eroare:    0 - executie normala
                                          1 - nr. maxim de iteratii depasit
*/
{
        const double eps = 1e-6;                            /* criteriu relativ de precizie */
        const int itmax = 100;                              /* nr. maxim de iteratii */
        double df, dx, f;
        int it;
        for (it = 1; it <= itmax; it++)
        {
                f = func(*x, &df);                          /* functia si derivata */
                dx = (fabs(df) > eps) ? -f / df : -f;       /* corectia radacinii */
                *x += dx;                                   /* noua aproximatie */
                if (fabs(dx) <= eps * fabs(*x))             /* test convergenta */
                        return 0;
```

```
	}
	printf("Newton: nr. maxim de iteratii depasit !\n");
	return 1;
}
```

Funcția utilizator, recunoscută sub numele Func de rutina Newton, trebuie să evalueze atât funcția $f(x)$ pentru care se caută zeroul, cât şi derivata corespunzătoare, returnată prin cel de-al doilea argument, df.

Parametrul x al funcției Newton trebuie să conțină la intrare o aproximație inițială şi returnează valoarea găsită de rutină pentru zero. Ieşirea din ciclul aproximațiilor se realizează fie prin scăderea erorii rădăcinii sub limita admisă *eps*, caz în care rutina returnează indicele de eroare 0, fie prin depăşirea numărului maxim de iterații datorită convergenței lente sau divergenței procesului iterativ. In cel de-al doilea caz este returnat indicele de eroare 1.

Dacă pe parcursul procesului iterativ se anulează derivata funcției (ceea ce numeric revine la *fabs (df)* < *eps*), se execută o iterație de "salvare" în care corecția rădăcinii se calculează ca în cazul metodei aproximațiilor succesive, adică fără a o împărți la derivata funcției. Această abordare are avantajul că rezolvă şi situațiile zerourilor de ordin superior, în care funcția şi prima ei derivată au zerouri comune.

Ca exemplu de construire a funcției utilizator asociată rutinei Newton, considerăm cazul particular al ecuației $x - e^{-x} = 0$:

```
float func(float x, float *df)
{
	*df = 1 + exp(-x);
	return x – exp(-x);
}
```

In cazul în care exprimarea derivatei funcției $f(x)$ este dificilă sau nu este posibilă, se poate recurge la evaluarea ei numerică. Funcția *NewtonNumDeriv*, listată mai jos, aproximează derivata pe baza valorilor funcției pentru două argumente separate printr-o valoare dx. Exceptând modul de calcul al derivatei şi faptul că funcția utilizator nu mai returnează derivata funcției $f(x)$, această rutină este identică cu cea anterioară.

```
int NewtonNumDeriv(double Func(double), double *x)
/*	Determine un zero real al unei functii reale prin metoda Newton-Raphson utilizand derivarea numerica
	Func - functia utilizator
	*x - aproximatie initiala (la intrare), zeroul gasit (la iesire)
	Returneaza indicele de eroare:	0 - executie normala
					1 - nr. maxim de iteratii depasit
*/
{
```

```
        const double eps = 1e-6;                        /* criteriu relativ de precizie */
        const int itmax = 100;                          /* nr. maxim de iteratii */
        double df, dx, f;
        int it;
        for (it = 1; it <= itmax; it++)
        {
                f = Func(*x);
                dx = *x ? 1e-4 * fabs(*x) : 1e-4;       /* pasul de derivare */
                df = (Func(*x + dx) - f) / dx;          /* derivata numerics. */
                dx = (fabs(df) > eps) ? –f / df : -f;   /* corectia radacinii */
                *x += dx;                               /* none aproximatie */
                if (fabs(dx) <= eps * fabs(*x))         /* test convergenta */
                        return 0;
        }
        printf("NewtonNumDeriv: nr. maxim de iteratii depasit !\n");
        return 1;
}
```

Convergența metodei lui Newton nu este dependentă de modul de scriere al funcției $f(x)$ în aceeaşi măsură ca şi în cazul metodei aproximațiilor succesive, întrucât împărțirea la derivata funcției în termenul de corecție, $\frac{f(x_i)}{f'(x_i)}$, ajustează valoarea funcției de iterare în vecinătatea rădăcinilor în sensul satisfacerii teoremei 1.1.1. Referindu-ne, astfel, la exemplul ecuației $x - e^{-x} = 0$,
se constată că metoda lui Newton converge către rădăcina $x = 0{,}567143$, spre deosebire de metoda aproximațiilor succesive, indiferent dacă se consideră $f(x) = x - e^{-x}$ sau $f(x) = e^{-x} - x$. Intr-adevăr, în ambele cazuri funcția de iterare are aceeaşi expresie, iar derivata ei $\varphi'(x) = e^{-x}\frac{x - e^{-x}}{\left(1 + e^{-x}\right)^2}$ este subunitară în valoare absolută (cum cere teorema) pentru orice *x* pozitiv.

Metoda lui Newton este in general mai rapid convergenta decât metoda aproximațiilor succesive. Astfel, în cazul ecuației $x - \sin x - 0{,}25 = 0$
pornind de la aproximația inițială $x_0 = 0$, metodei lui Newton îi sunt necesare 10 iterații pentru a obține soluția $x = 1{,}17123$, în timp ce metodei aproximațiilor succesive îi sunt necesare 28 de iterații.

1.2.2. Desfăşurarea laboratorului

a) se scrie algoritmul metodei Newton şi se verifică pentru ecuaţia $x - e^{-x} = 0$ considerând q=1.
b) să se studieze convergenţa şi să se obţină rădăcina ecuaţiei $x - \sin x - 0{,}25 = 0$.
c) să se obţină rădăcina ecuaţiei $x^2 - \sin x - 0{,}7 = 0$ cu o precizie de 6 şi 4 zecimale.
d) să se studieze convergenţa şi să se obţină rădăcina ecuaţiei $x^3 + x - 1 = 0$ pentru q=0,5; q=1 şi q=2.

L2. Rezolvarea sistemelor de ecuații liniare

2.1. Metoda Gauss-Jordan

2.1.1. Aspecte teoretice

Metoda Gauss-Jordan reprezintă o variantă a metodei lui Gauss. Spre deosebire de metoda Gauss, în care matricea sistemului este adusă prin transformări elementare la formă superior triunghiulară, în metoda Gauss-Jordan matricea sistemului este transformată în matricea unitate. Prin aceasta, deşi faza eliminării este mai laborioasă, faza substituției inverse este eliminată. In plus, printr-o codificare eficientă, simultan cu rezolvarea unei ecuații matriciale, metoda Gauss-Jordan permite şi calculul inversei matricei sistemului.

In urma pasului k de eliminare este eliminată necunoscuta x_k din toate ecuațiile sistemului, cu excepția ecuației pivot k şi sistemul este adus la forma:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & a_{1k+1}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & 1 & \cdots & 0 & a_{2k+1}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 & a_{kk+1}^{(k)} & \cdots & a_{kn}^{(k)} \\ 0 & 0 & \cdots & 0 & a_{k+1k+1}^{(k)} & \cdots & a_{k+1n}^{(k)} \\ \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{nk+1}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_k^{(k)} \\ b_{k+1}^{(k)} \\ \vdots \\ b_n^{(k)} \end{bmatrix} \tag{2.1.1}$$

Noile elemente ale liniei pivot k sunt:

$$\begin{cases} a_{kk}^{(k)} = 1 \\ a_{kj}^{(k)} = a_{kj}^{(k-1)} \big/ a_{kk}^{(k-1)}, \qquad j = k+1,\ldots,n \\ b_k^{(k)} = b_k^{(k-1)} \big/ a_{kk}^{(k-1)} \end{cases} \tag{2.1.2}$$

Noile elemente ale liniilor nepivot sunt:

$$\begin{cases} a_{ik}^{(k)} = 0 \\ a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k)}, \qquad j = k+1,\ldots,n, \ \ i = 1,\ldots,n, \quad i \neq k \\ b_i^{(k)} = b_i^{(k-1)} - a_{ik}^{(k-1)} b_k^{(k)} \end{cases} \tag{2.1.3}$$

Se observă că la pasul k se modifică numai elementele matricii sistemului situate în dreapta coloanei k, care devine identică cu coloana corespunzătoare a matricii unitate. La fiecare pas se modifică, în schimb, toate elementele matricii termenilor liberi.

In final, după pasul $k = n$, sistemul are forma $E_n \cdot x = b^{(n)}$, unde E_n reprezintă matricea unitate de ordinul *n*. Este evident că nu este necesară, ca în metoda Gauss, o fază a substituției inverse, iar soluția sistemului este:

$$x_k = b_k^{(n)}, \qquad k = 1,\ldots,n \tag{2.1.4}$$

Ca şi în cazul metodei Gauss, determinantul matricii $A^{(n)} = E$ este egal cu 1. Având însă în vedere că în obținerea matricii $A^{(n)}$ liniile matricii inițiale au fost împărțite pe rând la elementele pivot, avem

$$\det A^{(n)} = \frac{\det A}{a_{11}a_{22}^{(1)}\ldots a_{nn}^{(n-1)}} = 1,$$

de unde

$$\det A = a_{11}a_{22}^{(1)}\ldots a_{nn}^{(n-1)} \tag{2.1.5}$$

Relațiile de calcul ale metodei Gauss-Jordan, generalizate pentru rezolvarea unei ecuații matriciale (prin adăugarea unui indice de coloană suplimentar termenilor liberi b_{kj} şi necunoscutelor x_{kj}) pot fi scrise compact sub forma:

$$\begin{cases} a_{kj}^{(k)} = a_{kj}^{(k-1)} \big/ a_{kk}^{(k-1)}, & j = k+1,\ldots,n \\ b_{kj}^{(k)} = b_{kj}^{(k-1)} \big/ a_{kk}^{(k-1)}, & j = 1,\ldots,m \\ a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)}a_{kj}^{(k)}, & j = k+1,\ldots,n, \quad i = 1,\ldots,n \\ b_{ij}^{(k)} = b_{ij}^{(k-1)} - a_{ik}^{(k-1)}b_{kj}^{(k)}, & j = 1,\ldots,m \end{cases} \tag{2.1.6}$$

unde $k = 1,\ldots,n$ iar

$$x_{kj} = b_{kj}^{(n)}, \qquad j = 1,\ldots,m, \quad k = 1,\ldots,n \tag{2.1.7}$$

Intre metodele Gauss şi Gauss-Jordan apar deosebiri vizând deopotrivă faza eliminării şi faza substituției inverse.

Mai întâi, indicele *i* al liniilor nepivot, asupra cărora se operează reducerea, variază în cazul metodei Gauss între $k+1$ şi *n* (se elimină necunoscuta x_k numai din liniile de sub linia pivot *k*), iar în cazul metodei Gauss-Jordan indicele *i* variază de la 1 la *n* (sunt reduse toate liniile nepivot). In plus, deoarece în metoda Gauss-Jordan informația relevantă din matricea $A^{(k)}$, necesară ulterior pasului *k* de eliminare, se găseşte exclusiv în coloanele $j = k+1,\ldots,n$, sunt efectiv calculate şi trebuie stocate numai elementele matricii $A^{(k)}$ de pe aceste coloane.

Cea de a doua deosebire se referă la faptul că metoda Gauss-Jordan nu prezintă o fază a substituției inverse deoarece atribuirile prescrise de relația (2.1.7) nu trebuie codificate

efectiv, putându-se utiliza acelaşi tablou pentru matricea termenilor liberi $B = \lfloor b_{kj} \rfloor$ şi pentru matricea soluţiei $X = \lfloor x_{kj} \rfloor$.

Funcţia GaussJordan prezentată în continuare constituie o variantă de implementare a metodei Gauss-Jordan utilizând pivotarea parţială pe coloane.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define N 3
#define M 1
void GaussJordan(double *det)
/*
Rezolva ecuatia matriciala ax = b prin metoda Gauss-Jordan utilizand pivotarea partiala pe coloane.
a - matricea (n x n) a sistemului
b - matricea (n x m) a termenilor liberi / solutia x la iesire
n - ordinul sistemului
m - numarul de coloane al matricei termenilor liberi (este 1 pentru sisteme liniare)
*det - determinantul matricii sistemului (iesire)
*/
{
double a[N][N] = {{2, 1, 2},
                  {1, 3, 1},
                  {2, 1, 1}};
double b[N][M] = {{0}, {0}, {1}};
double amax,t;
int i,imax,j,k;
int n=N, m=M;

*det=1.0;
for (k=0;k<n;k++){
        amax=0.0;                       //determina linia pivot avand
        for (i=k;i<n;i++)               //elementul maxim pe coloana k
                if (amax<fabs(a[i][k])) {amax=fabs(a[i][k]);imax=i;}
        if (amax==0.0) {
                printf ("GaussJordan: matrice singulara !\n");
                return;
        }
                                                //interschimba liniile imax si k
        if (imax!=k) {
                *det=-(*det);
```

```
			for (j=k;j<n;j++) {t=a[imax][j]; a[imax][j]=a[k][j]; a[k][j]=t;}
			for (j=0;j<m;j++) {t=b[imax][j]; b[imax][j]=b[k][j]; b[k][j]=t;}
		}
		*det*=a[k][k];			//inmulteste determinantul cu pivotul
		t=1.0/a[k][k];			//imparte linia pivot
		for (j=k+1;j<n;j++) a[k][j]*=t;
		for (j=0;j<m;j++) b[k][j]*=t;
		for (i=0;i<n;i++)  //reduce liniile nepivot
			if (i!=k) {
				t=a[i][k];
				for (j=0;j<n;j++) a[i][j]-=a[k][j]*t;
				for (j=0;j<m;j++) b[i][j]-=b[k][j]*t;
			}
	}
	printf ("Solutia sistemului:\nx=[");
		for (i=0;i<n;i++){
			for (j=0;j<m;j++) printf("%lf,",b[i][j]);
			printf("\n");
		}
	printf("]\n");
	}

	void main()
	{double det;

	GaussJordan(&det);
	}
```

Datele de intrare ale rutinei sunt tabloul *a*, corespunzător matricii sistemului şi tabloul *b* al termenilor liberi care va conține în finalul algoritmului tabloul *x* al soluțiilor sistemului.

Rutina este scrisă pentru cazul general de rezolvare a ecuațiilor matriciale. Dacă matricea termenilor liberi are o singură coloană (cazul sistemelor liniare) atunci parametrul M ia valoarea 1.

2.1.2. Desfăşurarea laboratorului

1. se scrie algoritmul metodei Gauss-Jordan şi se verifică pentru sistemul

$$\begin{cases} 2x_1 + x_2 + 2x_3 = 0 \\ x_1 + 3x_2 + x_3 = 0 \\ 2x_1 + x_2 + x_3 = 1 \end{cases} .$$

2. să se aplice metoda Gauss-Jordan pentru sistemul $\begin{cases} 4x_1 - x_2 + 2x_3 = 10 \\ -x_1 + 5x_2 + x_3 = 10 \\ 2x_1 - x_2 + 5x_3 = 10 \end{cases}$.

2.2. Metoda iterativă Jacobi

2.2.1. Aspecte teoretice

Metodele directe (metoda eliminării gaussiene, metoda Gauss-Jordan etc.) permit determinarea soluției unui sistem de ecuații liniare de ordinul n într-un număr finit de paşi, prin efectuarea unui număr de operații elementare de ordinul n^3. Odată cu creşterea ordinului sistemului, erorile de rotunjire se pot acumula dramatic, ducând la erori relative apreciabile ale soluției. Pentru minimizarea acestor erori se impune, în general, pivotarea, adică reordonarea ecuațiilor după fiecare etapă de calcul pentru a avea elemente maximale pe diagonala principală a matricii sistemului. Această tehnică implică un efort de calcul suplimentar, care poate fi semnificativ în cazul sistemelor mari.

Metodele iterative permit, în principiu, determinarea soluției unui sistem de ecuații liniare pe baza unui proces iterativ, pornind de la o aproximație inițială a soluției. Dacă sistemul este bine condiționat numeric (matricea lui satisface anumite condiții), procedeul iterativ converge către soluția exactă. Practic, procesul este întrerupt după un număr finit de paşi, furnizând soluția cu o anumită precizie, afectată de erori de rotunjire (mai mici decât în cazul metodelor exacte) şi de erori de trunchiere. Unul dintre avantajele importante ale metodelor iterative constă însă în faptul că erorile de rotunjire şi chiar cele de trunchiere pot fi practic eliminate. De altfel, unele metode iterative pot fi utilizate pentru îmbunătățirea soluției obținute prin alte metode. Dacă se cunoaşte o aproximație inițială apropiată de soluția exactă a sistemului, convergența metodelor iterative este rapidă, necesitând pentru aceeaşi precizie în soluție un număr de operații mai mic decât metodele directe. In plus, metodele iterative se disting prin marea simplitate a programării lor.

Una dintre cele mai vechi şi mai cunoscute metode iterative pentru rezolvarea sistemelor de ecuații liniare este metoda Jacobi. Fie sistemul

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}, \tag{2.2.1}$$

sau

$$A \cdot x = b, \tag{2.2.2}$$

cu $A = \lfloor a_{ij} \rfloor_{nn}$, $b = \lfloor b_i \rfloor_n$ şi $x = \lfloor x_i \rfloor_n$. Presupunând că elementele diagonale ale matricii A sunt nenule, rezolvăm prima ecuaţie în raport cu x_1, cea de-a doua în raport cu x_2 etc. Obţinem sistemul echivalent:

$$\begin{cases} x_1 = t_1 + s_{12}x_2 + \ldots + s_{1n}x_n \\ x_2 = s_{21}x_1 + t_2 + \ldots + s_{2n}x_n \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n = s_{n1}x_1 + s_{n2}x_2 + \ldots + t_n \end{cases}, \tag{2.2.3}$$

unde
$$\begin{cases} s_{ii} = 0, i = 1,2,\ldots,n \\ s_{ij} = -\dfrac{a_{ij}}{a_{ii}}, j = 1,2,..,n, j \neq i \\ t_i = \dfrac{b_i}{a_{ii}} \end{cases}.$$

Introducând matricea $S = [sij]_{nn}$ şi vectorul coloană $t = \lfloor t_i \rfloor_n$, putem scrie sistemul (2.1.3) sub formă matricială

$$x = S \cdot x + t. \tag{2.2.4}$$

Rezolvăm acest sistem, numit şi *sistem redus*, prin *metoda aproximaţiilor succesive.* Ca aproximaţie de ordin zero considerăm coloana termenilor liberi

$$x^{(0)} = t \tag{2.2.5}$$

şi construim aproximaţia de ordinul *k* pe baza aproximaţiei de ordinul (*k*-1), utilizând formula de recurenţă

$$x^{(k)} = S \cdot x^{(k-1)} + t \tag{2.2.6}$$

pentru *k*=1,2,…. Dacă şirul $x^{(0)}, x^{(1)}, \ldots, x^{(k)}, \ldots$ are o limită $x = \lim_{k \to \infty} x^{(k)}$, atunci aceasta este soluţia sistemului (2.2.4). Intr-adevăr, trecând la limită în (2.2.6), rezultă ecuaţia (2.2.4), adică vectorul limită x este soluţia sistemului (2.2.4) şi, in consecinţă, şi a sistemului iniţial (2.2.2).

Relaţiile procesului iterativ pot fi scrise explicit:

$$x_i^{(k)} = \sum_{j \neq i}^{n} s_{ij} x_j^{(k-1)} + t_i, \ i = 1,2,\ldots,n. \tag{2.2.7}$$

Introducând corecțiile componentelor soluției

$$\Delta_i^{(k)} = x_i^{(k)} - x_i^{(k-1)},\ i = 1,2,...,n. \tag{2.2.8}$$

care pot constitui totodată şi estimări ale erorilor absolute corespunzătoare şi redefinind elementele diagonale ale matricii *S* ca fiind $s_{ii} = -1$, se pot rescrie relațiile metodei Jacobi sub forma

$$\begin{cases} \Delta_i^{(k)} = \sum_{j=1}^{n} s_{ij} x_j^{(k-1)} + t_i \\ x_i^{(k)} = x_i^{(k-1)} + \Delta_i^{(k)}, i = 1,2,...,n \end{cases}, \tag{2.2.9}$$

unde

$$\begin{cases} s_{ij} = -\dfrac{a_{ij}}{a_{ii}}, i, j = 1,2,...,n \\ t_i = \dfrac{b_i}{a_{ii}} \end{cases}. \tag{2.2.10}$$

Procedeul iterativ descris de aceste relații trebuie continuat în principiu până când eroarea relativă maximă a componentelor soluției devine mai mică decât o eroare maximă admisibilă, ε, adică $\max_i \left| \dfrac{\Delta_i^{(k)}}{x_i^{(k)}} \right| \le \varepsilon$. (2.2.11)

Dăm fără demonstrație o condiție suficientă pentru convergența procedeului iterativ (2.2.6) sub forma următoarei teoreme:

Teorema 2.2.1 *Dacă pentru sistemul redus (2.2.4) este valabilă cel puțin una dintre următoarele două condiții:*

$$\begin{aligned} \sum_{j=1}^{n} \left| s_{ij} \right| < 1, i = 1,2,...,n \\ \sum_{i=1}^{n} \left| s_{ij} \right| < 1, j = 1,2,...,n \end{aligned}, \tag{2.2.12}$$

atunci procesul iterativ (2.2.6) converge către soluția unică a sistemului, indiferent de alegerea aproximației inițiale.

Corolar 2.2.1 *Pentru sistemul (2.2.2) procesul iterativ (2.2.6) este convergent dacă au loc inegalitățile*

$$\left| a_{ii} \right| > \sum_{j \ne i} \left| a_{ij} \right|,\ i = 1,2,...,n, \tag{2.2.13}$$

adică dacă pe fiecare linie a sistemului valoarea absolută a coeficientului diagonal este mai mare decât suma valorilor absolute ale coeficienților extradiagonali.

In practică, pentru ca metoda Jacobi să fie convergentă, nu este neapărat necesară îndeplinirea condiţiilor (2.2.12) sau (2.2.13), ci în multe situaţii este suficient ca matricea sistemului să fie *dominant diagonală,* adică

$$|a_{ii}| > \max_{j \neq i} |a_{ij}|, \quad i = 1,2,...,n. \tag{2.2.14}$$

Din relaţiile (2.2.9) se observă că metoda Jacobi necesită în implementări rezervarea a două tablouri, corespunzătoare soluţiilor consecutive $x^{(k-1)}$ şi $x^{(k)}$. Datorită faptului că metoda Gauss-Seidel, care va fi discutată în secţiunea următoare, prezintă importante avantaje, implementarea algoritmului Jacobi nu este de interes practic, păstrând o importanţă pur teoretică.

2.3. Metoda Gauss-Seidel

2.3.1. Aspecte teoretice

Metoda Gauss-Seidel reprezintă o variantă superioară a metodei Jacobi, caracterizată prin viteza de convergenţă sporită şi necesar de memorie redus. Ideea de bază a metodei Gauss-Seidel constă în utilizarea în procesul iterativ Jacobi (2.2.9) a celor mai recente componente ale soluţiei sistemului, pe măsura determinării lor, nu a celor de la iteraţia anterioară.

Presupunând că sistemul de ecuaţii liniare $A \cdot x = b$ a fost pus sub forma redusă

$$x = S \cdot x + t, \tag{2.3.1}$$

relaţiile de recurenţă care stau la baza metodei Gauss-Seidel sunt

$$\begin{cases} \Delta_i^{(k)} = \sum_{j=1}^{i-1} s_{ij} x_j^{(k)} + \sum_{j=i}^{n} s_{ij} x_j^{(k-1)} + t_i \\ x_i^{(k)} = x_i^{(k-1)} + \Delta_i^{(k)}, i = 1,2,...,n \end{cases}. \tag{2.3.2}$$

unde, ca si în cazul metodei Jacobi, elementele matricilor *S* şi *t* au expresiile

$$\begin{cases} s_{ij} = -\dfrac{a_{ij}}{a_{ii}}, i, j = 1,2,...,n \\ t_i = \dfrac{b_i}{a_{ii}} \end{cases}. \tag{2.3.3}$$

După cum se poate remarca, în calculul componentei $x_i^{(k)}$ a soluţiei intervin componentele $x_1^{(k)}, x_2^{(k)}, ..., x_{i-1}^{(k)}$, deja calculate la iteraţia *k*, în locul componentelor corespunzătoare de la iteraţia anterioară.

Şi în acest caz se poate considera ca aproximaţie iniţială coloana termenilor liberi, adică $x^{(0)} = t$. O utilizare alternativă a metodei o constituie însă rafinarea unei aproximaţii iniţiale a soluţiei, rezultată eventual prin aplicarea altei metode de rezolvare.

Criteriul de convergenţă în metoda Gauss-Seidel poate fi exprimat, ca şi în metoda Jacobi, cerând ca eroarea relativă maximă a componentelor soluţiei să devină mai mică decât o toleranţă prescrisă, $\varepsilon > 0$, adică

$$\max_i \left| \frac{\Delta_i^{(k)}}{x_i^{(k)}} \right| \le \varepsilon . \qquad (2.3.4)$$

In cazul în care există componente $x_i^{(k)}$ nule, în locul erorii relative $\frac{\Delta_i^{(k)}}{x_i^{(k)}}$ trebuie utilizată eroarea absolută $\Delta_i^{(k)}$.

Teorema de convergenţă 2.2.1 dată în secţiunea anterioară pentru metoda Jacobi îşi păstrează valabilitatea şi în cazul metodei Gauss-Seidel.

In general, metoda Gauss-Seidel converge mai rapid decât metoda Jacobi şi, mai mult, poate converge chiar dacă algoritmul Jacobi diverge. Cu toate acestea, sunt posibile şi cazuri în care procesul Gauss-Seidel sub forma prezentată nu converge deloc. Realizând însa, aşa după cum se va arăta în continuare, anumite transformări asupra sistemului liniar iniţial, convergenţa algoritmului este garantată.

Definitia 2.3.1 *Un sistem liniar* $A \cdot x = b$ *se numeşte* normal *dacă matricea coeficienţilor* $A = \lfloor a_{ij} \rfloor_{nn}$ *este* simetrică, *adică* $a_{ij} = a_{ji}$, *şi* pozitiv definită, *adică dacă are loc* $x \cdot A \cdot x > 0$ *pentru orice vector x din spaţiul în care este definită matricea A.*

Teorema 2.3.1 *Dacă ambii membri ai sistemului liniar* $A \cdot x = b$, *având matricea A nesingulară, sunt înmulţiţi la stânga cu transpusa* A^T, *atunci sistemul rezultant,* $A^T \cdot A \cdot x = A^T \cdot b$ *,este* normal.

Teorema 2.3.2 *Procesul iterativ Gauss-Seidel converge întotdeauna pentru un sistem normal, indiferent de alegerea aproximaţiei iniţiale.*

Din analiza relaţiilor (2.3.2)-(2.3.3) rezultă că în implementarea algoritmului Gauss-Seidel se poate utiliza pentru stocarea soluţiei de la paşii *(k - 1)* şi *k* acelaşi tablou, ale cărui componente sunt actualizate pe măsura calculului noilor componente $x_i^{(k)}$. Suplimentar economiei de memorie, această constatare conduce şi la o codificare mai compactă a algoritmului, aşa cum se vede în următoarea implementare.

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <math.h>
#define N 3
void GaussSeidel(double *err)
/* Rezolva sistemul liniar ax = b prin metoda Gauss-Seidel. Pentru asigurarea convergentei, sistemul este
inmultit la stanga cu a-trans.
        a - matricea (n x n) a sistemului
        b[] - vectorul termenilor liberi
        x[] - aproximatie initiala la intrare / solutia x la iesire
        n - ordinul sistemului
        *err - eroarea relativa maxima a componentelor solutiei (iesire)
        init - optiune de initializare:
                0 - rafineaza aproximatia initiala
                1 - initializeaza solutia
*/
{
        double a[N][N] = {{2, 1, 2},
                          {1, 3, 1},
                          {2, 1, 1}};
        double b[N] = {0, 0, 1};
        double x[N] = {1, 1, 1};

        const double eps = 1e-6;                    /* criteriu relativ de precizie */
        const int itmax = 500;                      /* nr. Maxim de iteratii */
        double del, f;
        int i, j, k;
        int n = N;
        int init = 1;
        double s[N][N];
        double t[N];

        for (i = 0; i < n; i++)                     /* matricile sistemului normal */
        {
                for (j = 0; j <= i; j++)            /* inmultind a si b cu a-trans */
                {
                        s[i][j] = 0.0;              /* depune rezultatul in s si t */
                        for (k = 0; k < n; k++)
                                s[i][j] += a[k][i] * a[k][j];
                        s[j][i] = s[i][j];
                }
                t[i] = 0.0;
```

```
		for (j = 0; j < n; j++)
			t[i] += a[j][i] * b[j];
	}

	for (i = 0; i < n; i++)				/* matricile s si t */
	{
		f = -1.0/s[i][i];
		t[i] /= s[i][i];				/* ale sist. redus */
		for (j = 0; j < n; j++)
			s[i][j] *= f;
	}
	if (init)
		for (i = 0; i < n; i++)
			x[i] = t[i];				/* initializeaza solutia */
	*err = eps + 1;
	k = 0;
	while ((k < itmax) && (*err > eps))		/* ciclul iteratiilor */
	{
		*err = 0.0;
		k++;
		for (i = 0; i < n; i++)
		{
			del = t[i];				/* corectia */
			for (j = 0; j < n; j++)
				del += s[i][j] * x[j];
			x[i] += del;				/* noua aproximatie */
			if (x[i])
				del /= x[i];			/* eroarea relativa */
			if (fabs(del) > *err)
				*err = fabs(del);		/* eroarea maxima */
		}
	}
	if (k == itmax)
		printf("GaussSeidel: nr. Maxim de iteratii depasit!\n");
	else
		{
		printf("x=[");
		for (i = 0; i < n-1; i++)
			printf("%lf ",x[i]);
		printf("%lf]\n",x[n-1]);
```

```
		}
}

void main()
{
	double err;
	GaussSeidel(&err);
}
```

Datele de intrare ale rutinei sunt tabloul *a*, corespunzător matricii sistemului, tabloul *b* al termenilor liberi, tabloul *x*, prin care se introduce (eventual) aproximaţia iniţială a soluţiei, ordinul *n* al sistemului şi opţiunea de iniţializare *init*. Dacă parametrul *init* este nul, rutina este utilizată pentru rafinarea aproximaţiei iniţiale transmise prin tabloul *x*. Daca, dimpotrivă, *init* este nenul, este initializată soluţia utilizând termenii liberi ai sistemului redus. La ieşirea din procedură, tabloul *x* returnează soluţia, iar parametrul *err*, eroarea relativă maximă a componentelor acesteia.

Convergenţa algoritmului este asigurată prin înmulţirea sistemului cu transpusa matricii sistemului. Tabloul *s* este folosit mai întâi pentru stocarea matricii $A^T \cdot A$ a sistemului normal şi apoi a matricii *S* a sistemului redus. Analog, în tabloul *t* sunt mai întâi depuse componentele vectorului $A^T \cdot b$, şi apoi componentele vectorului *t.*

2.3.2. Desfăşurarea laboratorului

1. se scrie algoritmul metodei Gauss-Seidel şi se verifică pentru sistemul $\begin{cases} 2x_1 + x_2 + 2x_3 = 0 \\ x_1 + 3x_2 + x_3 = 0 \\ 2x_1 + x_2 + x_3 = 1 \end{cases}$. Parametrul *init* este nenul.
2. să se aplice metoda Gauss-Seidel pentru sistemul $\begin{cases} 8x_1 + x_2 - x_3 = 10 \\ -x_1 + 7x_2 - 2x_3 = 4 \\ 2x_1 + x_2 + 9x_3 = 12 \end{cases}$. Calculele vor avea o precizie de 4 zecimale.
3. să se aplice metoda Gauss-Seidel pentru sistemul $\begin{cases} 4x_1 - x_2 + 2x_3 = 10 \\ -x_1 + 5x_2 + x_3 = 10 \\ 2x_1 - x_2 + 5x_3 = 10 \end{cases}$. Să se afişeze numărul de iteraţii pentru parametrul *init* nenul şi nul, caz în care se utilizează ca aproximaţie iniţială soluţia obţinută prin aplicarea metodei Gauss-Jordan. Precizia va fi mărită la 8 zecimale.

L3. Interpolarea

3.1. Polinomul de interpolare Lagrange

3.1.1. Aspecte teoretice

Una dintre cele mai vechi şi generale formule de interpolare este cea datorată lui Lagrange. Mai mult decât în utilitatea practică directă, importanţa ei constă în consecinţele teoretice. O întreagă clasă de formule de integrare numerică are la bază aproximarea funcţiilor prin polinomul de interpolare al lui Lagrange. Pe de altă parte, plecând de la acest interpolant pot fi construite scheme de derivare numerică cu diferite ordine de precizie.

Sa presupunem că, pentru funcţia $f(x)$ care trebuie aproximată, sunt cunoscute n valori corespunzătoare argumentelor $x_1, x_2, \ldots, x_n$ din intervalul $[\alpha, \beta]$:

$$f(x_i) = y_i, \quad i = 1, 2, \ldots, n.$$

Ne propunem construirea unui polinom $P_m(x)$ care să ia în punctele de tabelare x_i aceleaşi valori ca şi funcţia $f(x)$,

$$P_m(x_i) = y_i, \quad i=-1,2,\ldots,n. \tag{3.1.1}$$

In acest scop să construim mai întâi o familie de polinoame $p_i(x)$, pentru care are loc

$$p_i(x_j) = \delta_{ij} = \begin{cases} 1, & j = i \\ 0, & j \neq i \end{cases} \tag{3.1.2}$$

unde δ_{ij} este simbolul lui Kronecker. Deoarece $p_i(x)$ trebuie să se anuleze în toate cele n puncte de tabelare cu excepţia punctului x_i, el poate fi scris sub forma unui produs de factori de forma $(x - x_j)$, din care, în particular, lipseşte factorul $(x - x_i)$:

$$p_i(x) = C_i \prod_{j \neq i} (x - x_j).$$

Astfel definit, $p_i(x)$ este un polinom de ordinul $(n-1)$. Luând $x = x_i$ şi având în vedere că $p_i(x_i) = 1$, se găseşte coeficientul constant $C_i = \dfrac{1}{\prod_{j \neq i} (x_i - x_j)}$. Cu acestea, pentru polinomul $p_i(x)$ rezultă expresia

$$p_i(x) = \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)}. \tag{3.1.3}$$

Revenind acum la problema iniţială, polinomul $P_m(x)$ care satisface condiţiile de interpolare (3.1.1) poate fi scris ca o combinaţie liniară a polinoamelor $p_i(x)$, pornind de la faptul

că fiecare dintre acestea din urmă se anulează în toate punctele de tabelare cu excepția câte unuia singur - x_i. In plus, deoarece polinoamele $p_i(x)$ sunt de ordinul $(n - 1)$, $P_m(x)$ este el însuşi de ordinul $(n - 1)$ şi avem

$$P_{n-1}(x) = \sum_{i=1}^{n} p_i(x) y_i \ . \tag{3.1.4}$$

Utilizând proprietățile (3.1.2), se poate arăta uşor că sunt satisfăcute condițiile de interpolare (3.1.1):

$$P_{n-1}(x_j) = \sum_{i=1}^{n} p_i(x_j) y_i = \sum_{i=1}^{n} \delta_{ij} y_i = y_j \ , \quad j = 1,2,...,n.$$

Inlocuind expresia (3.1.3) a polinoamelor $p_i(x)$ in (3.1.4) rezultă polinomul de interpolare Lagrange:

$$P_{n-1}(x) = \sum_{i=1}^{n} \frac{\prod_{j \neq i}^{n} (x - x_j)}{\prod_{j \neq i}^{n} (x_i - x_j)} y_j \ . \tag{3.1.5}$$

In general, numărul de puncte utilizate într-o schemă de interpolare minus unu este numit ordinul interpolării şi în cazul de față acesta este egal cu ordinul polinomului de interpolare.

Pentru a demonstra unicitatea polinomului de interpolare Lagrange vom presupune contrariul, adică faptul că există un polinom $\overline{P}_{n-1}(x)$ diferit de $P_{n-1}(x)$, care satisface de asemenea condițiile de interpolare

$$\overline{P}_{n-1}(x_i) = y_i, \quad i = 1,2,...,n.$$

In consecință, polinomul

$$Q_{n-1}(x) = \overline{P}_{n-1}(x) - P_{n-1}(x)$$

se anulează în toate cele n puncte de interpolare x_i. Fiind însă de ordinul $(n - 1)$, nu are decât $(n - 1)$ zerouri, şi deci rezultă a fi identic nul. Prin urmare, $\overline{P}_{n-1}(x) \equiv P_{n-1}(x)$.

In cazul particular $n = 2$ formula lui Lagrange se reduce la

$$P_1(x) = \frac{x - x_2}{x_1 - x_2} y_1 + \frac{x - x_1}{x_2 - x_1} y_2 \tag{3.1.6}$$

şi descrie dreapta care trece prin cele două puncte de tabelare. Pentru $n = 3$ interpolantul Lagrange corespunde parabolei definite de cele trei puncte de tabelare:

$$P_2(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} y_1 + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} y_2 + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} y_3. \tag{3.1.7}$$

Următoarea funcție implementează formula (3.1.5) a polinomului de interpolare Lagrange.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

double Lagrange(double xi[], double yi[], int ni, double x)
/*---------------------------------------------------------------------
Evalueaza polinomul de interpolare Lagrange
xi[] - abscisele punctelor interpolate
yi[] - ordonatele punctelor interpolate
ni - numarul punctelor interpolate
x - argumentul polinomului
---------------------------------------------------------------------_*/
{
	double p, y;
	int i, j;
	y = 0.0;
	for (i = 0; i < ni; i++)
	{
		p = 1.0;
		for (j = 0; j < ni; j++)
			if (j != i)
				p *= (x - xi[j]) / (xi[i] - xi[j]);
		y += p * yi[i];
	}
	return y;
}
void main()
{
	double x[8] = {0.15, 0.2, 0.3, 0.5, 0.8, 1.1, 1.4, 1.7};
	double y[8];
	for (int i = 0; i < 8; i++)
		y[i] = 1 / x[i];
	printf("Valoarea interpolantului in punctul 1.3 este %lf\n",Lagrange(x, y, 8, 1.3));
}
```

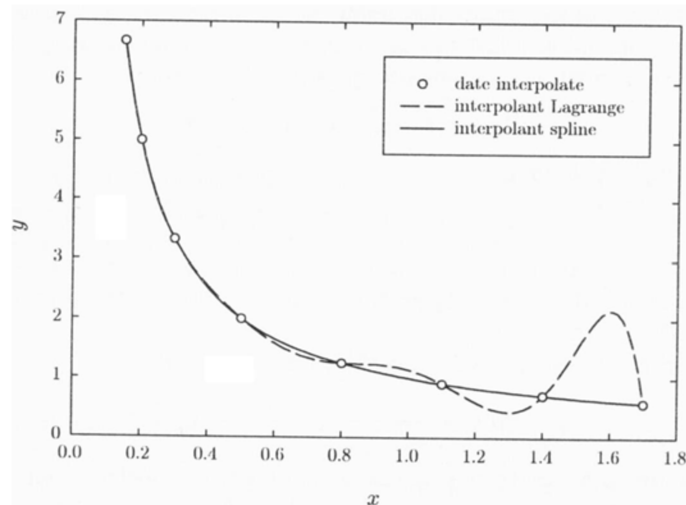


FIGURA 3.1.1

Exemplu în care interpolarea cu ajutorul polinomului lui Lagrange conduce la oscilații. Datele interpolate provin din eşantionarea funcției *f(x)=1/x*

Relatia (3.1.5) evidențiază faptul că aproximarea realizată cu ajutorul polinomului de interpolare Lagrange este esențialmente *nelocală,* în sensul că la evaluarea interpolantului contribuie informații din *toate* punctele tabelate, nu numai din cele din vecinătatea argumentului considerat. Nelocalitatea garantează pe de o parte "netezimea" aproximării prin continuitatea derivatelor, însă, pe de alte parte, poate induce oscilații complet străine de comportarea reala a funcției aproximate.

Una dintre cauzele apariției oscilațiilor polinomului Lagrange o constituie dispunerea neadecvată a punctelor de tabelare. Pentru ilustrarea acestui fenomen a fost considerată funcția *f(x) = 1/x*, pentru care, aşa cum se vede din figura 3.1.1, au fost generate puncte de interpolare corespunzătoare argumentelor 0.15, 0.2, 0.3, 0.5, 0.8, 1.1, 1.4 şi 1.7. Distanța relativ mare dintre abscisele ultimelor puncte face ca interpolantul Lagrange să nu poată reproduce funcția modelată în acest domeniu şi să prezinte oscilații din ce in ce mai ample. Distribuind însă în mod echidistant abscisele, oscilațiile se reduc până aproape de dispariție. Mai mult, adăugând încă un singur punct de tabelare, tot in condițiile dispunerii echidistante a absciselor, oscilațiile pot fi eliminate complet în acest caz. Trebuie avut în vedere totuşi că, neînsoțite de o dispunere judicioasă a absciselor, creşterea numărului punctelor de tabelare nu îmbunătățeşte întotdeauna precizia interpolării, deoarece, în general, prin creşterea corespunzătoare a ordinului polinomului de interpolare apar zerouri suplimentare şi deci oscilații suplimentare ale interpolantului.

Prin constrast cu comportamentul interpolantului Lagrange în exemplul considerat mai sus, din figura 3.1.1 se poate constata că aproximarea realizată cu ajutorul funcțiilor spline cubice, nu prezintă oscilații. Calitativ, acest comportament diferit poate fi explicat prin faptul ca restricțiile interpolantului spline între punctele de tabelare sunt polinoame de ordin scăzut, cu număr mic de zerouri şi deci tendință redusă de oscilație.

Pe baza formulei de interpolare Lagrange se poate rezolva şi *problema interpolării inverse,* adică găsirea argumentului pentru care polinomul Lagrange are o anumită valoare. Pentru aceasta este suficient sa se considere y ca variabilă independentă şi să se scrie, prin analogie cu relația (3.1.5), o formulă exprimând x ca funcție de y:

$$x = \sum_{i=1}^{n} \frac{\prod_{j \neq i} (y - y_i)}{\prod_{j \neq i} (y_i - y_j)} x_i \ . \qquad (3.1.8)$$

Cu ajutorul acestei formule se poate găsi o rădăcină aproximativă a ecuației $f(x) = 0$. In acest scop se calculează un tablou de valori y_i pentru n argumente x_i apropiate de rădăcină. Punând apoi $y = 0$ în relația (3.1.8), se găseşte rădăcina respectivă. Dacă $f(x) = P_{n-1}(x)$ este un polinom de ordinul n - 1, rădăcina determinată prin această metodă este exactă.

3.1.2. Desfăşurarea laboratorului

1. să se scrie algoritmul de calcul al polinomului de interpolare Lagrange pentru funcția f(x) = 1/x şi să se afişeze valoarea interpolantului în trei puncte la alegere. Corectitudinea valorilor obținute se va aprecia prin comparare cu valoarea calculată a funcției în punctele respective. Se vor considera 8 puncte de interpolare distribuite aleator în intervalul (0,2].
2. să se scrie algoritmul de calcul al polinomului de interpolare Lagrange pentru funcția f(x) = (1-x)/5 şi să se afişeze valoarea interpolantului în trei puncte la alegere. Corectitudinea valorilor obținute se va aprecia prin comparare cu valoarea calculată a funcției în punctele respective. Se vor considera 10 puncte de interpolare distribuite aleator în intervalul (0,2].
3. să se scrie algoritmul de calcul al polinomului de interpolare Lagrange pentru funcția f(x) = 1/(x-3) şi să se afişeze valoarea interpolantului în trei puncte la alegere. Se vor considera 8 apoi 15 puncte de interpolare distribuite uniform în intervalul (0,2] şi se va face comparație între valorile obținute în cele două cazuri.

3.2. *Metoda Neville*

3.2.1. Aspecte teoretice

Metoda Neville nu constituie o abordare distinctă a problemei interpolării, ci reprezintă mai degrabă un algoritm optim de construire a polinomului de interpolare unic printr-un număr dat de puncte, care este polinomul Lagrange. Un dezavantaj al utilizării directe a formulei (3.1.5) a polinomului Lagrange este acela că nu oferă o estimare a erorii, decât prin compararea valorii interpolanților pentru seturi de puncte de tabelare care se subîntind. Acest mod de lucru nu permit însă refolosirea rezultatelor obținute pentru un anumit interpolant în evaluarea interpolanților de ordin superior.

Metoda Neville implică în esență calculul iterativ al interpolanților de ordine consecutive, construiți relativ la subseturi crescânde de puncte de interpolare, cu refolosirea în cel mai înalt grad a rezultatelor anterioare şi furnizând în mod natural o estimare a erorii aproximării.

Fie funcția $f(x)$ specificată prin cele n valori corespunzătoare şirului de argumente $x_1, x_2, \ldots, x_n$:

$$f(x_i) = y_i, \quad i = 1,2,\ldots,n .$$

Pentru a construi eficient polinomul Lagrange care interpolează toate cele n puncte de tabelare (x_i, y_i), şi modelează funcția $f(x)$, definim mai întâi o familie de polinoame de interpolare, astfel ca $P_{i,j}(x)$ să fie polinomul unic care interpolează succesiunea de puncte cu abscisele cuprinse intre x_i şi x_j, adică:

$$P_{i,j}(x_k) = y_k, \quad i \le k \le j \qquad (3.2.1)$$

Desigur, interpolând $j-i+1$ puncte, polinomul $P_{i,j}(x)$ este de ordinul $j-i$. In particular, $P_{i,i}(x)$ este polinomul de ordin zero care interpolează numai punctul (x_i,y_i), adică funcția constantă care ia pentru orice argument valoarea y_i. Pentru ansamblul polinoamelor $P_{i,i}(x)$ avem atunci:

$$P_{i,i}(x) = y_i, \quad i = 1,2,\ldots,n . \qquad (3.2.2)$$

Evident, polinomul Lagrange căutat poate fi identificat cu $P_{1,n}(x)$, care este polinomul de interpolare unic în raport cu întregul set de date:

$$P_{n-1}(x) = P_{1,n}(x). \qquad (3.2.3)$$

Evaluarea polinomului $P_{1,n}(x)$ pentru un anumit argument x poate fi realizată în mod optim aranjând valorile $P_{i,j}(x)$ potrivit următorului tablou cu "predecesori" şi "descendenți", pe fiecare coloană aflându-se polinoamele de acelaşi ordin m (egal cu diferența $j-i$ dintre cei doi indici):

	$m=0$	1	2	$\dots$	$n-2$	$n-1$
x_1	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$\dots$	$P_{1,n-1}$	$P_{1,n}$
x_2	$P_{2,2}$	$P_{2,3}$	$\dots$	$P_{2,n-1}$	$P_{2,n}$	
x_3	$P_{3,3}$	$\dots$	$\dots$	$\dots$		
$\dots$	$\dots$	$P_{n-2,n-1}$	$P_{n-2,n}$			
x_{n-1}	$P_{n-1,n-1}$	$P_{n-1,n}$				
x_n	$P_{n,n}$					

Metoda Neville este propriu-zis un algoritm de completare recursivă a coloanelor tabloului - de la stânga la dreapta, pornind de la polinoamele de ordin zero $P_{i,i}(x)$ şi conducând pe ultima coloană la un singur descendent, care este valoarea căutată $P_{1,n}(x)$. Propagarea valorilor se poate efectua pe baza relației dintre un anumit descendent $P_{i,j}(x)$ şi cei doi predecesori ai săi de ordin inferior, $P_{i,j-1}(x)$ şi $P_{i+1,j}$ (caracterizați prin aceeaşi diferență $j-i-1$ între indici), situați unul sub altul pe coloana anterioară:

$$P_{i,j}(x)=\frac{(x-x_j)P_{i,j-1}(x)+(x_i-x)P_{i+1,j}(x)}{x_i-x_j}. \tag{3.2.4}$$

Intr-adevăr, pentru orice x_k cu $i<k<j$ sunt satisfăcute condițiile de interpolare, deoarece predecesorii $P_{i,j-1}(x)$ şi $P_{i+1,i}(x)$ satisfac ei înşişi aceste condiții pentru subsetul comun de puncte de interpolare:

$$P_{i,j}(x_k)=\frac{(x_k-x_j)y_k+(x_i-x_k)y_k}{x_i-x_j}=y_k.$$

In particular, pentru punctele extreme x_i şi x_j are loc

$$P_{i,j}(x_i)=\frac{(x_i-x_j)y_i+(x_i-x_i)P_{i+1,j}(x_i)}{x_i-x_j}=y_i,$$

$$P_{i,j}(x_j)=\frac{(x_j-x_j)P_{i,j-1}(x_j)+(x_i-x_j)y_j}{x_i-x_j}=y_j.$$

Prin urmare, polinomul $P_{i,j}(x)$ definit de relația de recurență (3.2.4) interpolează toate cele $j-i+1$ puncte $(x_i,y_i),\dots,(x_j,y_i)$ şi, fiind un polinom de ordinul $j-i$, reprezintă într-adevăr polinomul unic de interpolare, adică polinomul Lagrange.

Având în vedere că metoda Neville implică parcurgerea *pe coloane* a tabloului de valori $P_{i,j}(x)$ şi că indicii polinoamelor de pe o anumită coloană m sunt legați prin relația $j=m+i$, formula de recurență (3.2.4) poate fi rescrisă sub o formă mai adecvată pentru implementări prin introducerea notațiilor $P_i^{(m)}(x)=P_{i,j}(x)$ şi $x_{m+i}=x_j$, care pun în evidență indicele de coloană şi, implicit, ordinul polinoamelor:

$$P_i^{(m)}(x) = \frac{(x - x_{m+i})P_i^{m-1}(x) + (x_i - x)P_{i+1}^{(m-1)}(x)}{x_i - x_{m+i}}, \tag{3.2.5}$$

$m=1,2,\dots,n-1,\ i=1,2,\dots,n-m.$

Indicele de coloană *m* poate fi parcurs în mod independent, iar pentru fiecare coloană indicele de linie *i* ia *n-m* valori, cu una mai puțin decât pentru coloana anterioară. In implementări concrete este suficientă utilizarea unui tablou unidimensional, în ale cărui componente sunt memorate valorile polinoamelor $P_i^{(m)}(x)$ de pe coloana curentă.

O estimare utilă pentru eroarea aproximării funcției tabelate prin polinomul Lagrange $P_{n-1}(x) = P_1^{(n-1)}(x)$ este furnizată de diferența ultimilor doi predecesori:

$$\Delta = \left| P_1^{(n-2)}(x) - P_2^{(n-2)}(x) \right|.$$

Semnificația argumentelor următoarei implementări a metodei Neville este în esență aceeaşi cu cea din cazul rutinei Lagrange, prezentată în secțiunea anterioară: x_i şi y_i - tablouri care conțin perechile de coordonate ale punctelor de interpolare, *ni* - numărul punctelor de interpolare şi *x* - argumentul pentru care se evaluează polinomul de interpolare. Parametrul **err* returnează, în plus, eroarea aproximării.

```
#include "stdio.h"
#include "stdlib.h"
#include "math.h"

double Neville(double xi[], double yi[], int ni, double x, double *err)
/*-------------------------------------------------------------------------
Evalueaza polinomul de interpolare Lagrange prin metoda Neville si returneaza o estimare a erorii absolute
xi[] - abscisele punctelor interpolate
yi[] - ordonatele punctelor interpolate
ni - numarul punctelor interpolate
x - argumentul polinomului
*err - estimarea erorii absolute a valorii polinomului (iesire)
-------------------------------------------------------------------------*/
{
	double *p, y;
	int i, m;
	p = (double*)malloc(ni * sizeof(double));

	for (i = 0; i < ni; i++)		/* initializeaza tabloul cu */
		p[i] = yi[i];		/* polinoamele de ordin 0    */
	for (m = 1; m < ni - 1; m++)		/* parcurge coloanele tabloului */
```

```
		for (i = 0; i < ni - m; i++)
			p[i] = ((x - xi[m + i]) * p[i] + (xi[i] - x) * p[i + 1]) / (xi[i] - xi[m + i]);
	y = p[1];				/* valoarea polinomului */
	*err = fabs(p[1] - p[2]);		/* estimarea erorii */
	free(p);
	return y;
}

void main()
{
	double x[8] = {0.15, 0.2, 0.3, 0.5, 0.8, 1.1, 1.4, 1.7};
	double y[8];
	double err=0;

	for (int i = 0; i < 8; i++)
		y[i] = 1 / x[i];
	printf("Valoarea interpolantului in punctul 1.55 este %lf,", Neville(x, y, 8, 1.55, &err));
	printf(" cu o valoare a erorii de %.5lf\n", err);
}
```

3.2.2. Desfăşurarea laboratorului

1. să se scrie algoritmul metodei Neville pentru funcția f(x) = 1/x şi să se afişeze valoarea interpolantului şi a erorii în trei puncte la alegere. Se vor considera 8 puncte de interpolare distribuite aleator în intervalul (0,2].
2. să se scrie algoritmul metodei Neville pentru funcția f(x) = (7-x)/23 şi să se afişeze valoarea interpolantului şi a erorii în trei puncte la alegere. Se vor considera 10 puncte de interpolare distribuite aleator în intervalul (0,2].
3. să se scrie algoritmul metodei Neville pentru funcția f(x) = 1/(x-3) şi să se afişeze valoarea interpolantului şi a erorii în trei puncte la alegere. Se vor considera 15 apoi 20 puncte de interpolare distribuite uniform în intervalul (0,2] şi se vor compara valorile erorilor obținute în cele două cazuri.

L4. Calculul integralelor unidimensionale

4.1. Formulele de cuadratură Newton-Cotes

4.1.1. Aspecte teoretice

Să presupunem că pentru o funcție y = f(x) se cere calculul integralei definite

$$I = \int_a^b f(x)dx . \qquad (4.1.1)$$

Impărțim intervalul [a,b] în (n-1) subintervale egale, de lungime

$$h = (b-a)/(n-1) \qquad (4.1.2)$$

prin punctele

$$x_i = a + (i-1)h, i = 1,2,...,n \qquad (4.1.3)$$

şi presupunând că sunt cunoscute valorile $f_i \equiv f(x_i)$ în nodurile x_i, vom aproxima funcția $f(x)$ prin polinomul de interpolare Lagrange

$$P_{n-1}(x) = \sum_{i=1}^{n} \frac{\prod_{j\neq i}^{n}(x-x_j)}{\prod_{j\neq i}^{n}(x_i-x_j)} f_i .$$

Introducând variabila adimensională

$$q = (x-a)/h, \;\; q \in [0, n-1],$$

argumentele funcțiilor pot fi exprimate sub forma $x = a + qh$ şi produsele din expresia polinomului de interpolare Lagrange devin

$$\prod_{j\neq i}^{n}(x-x_j) = h^{n-1}\prod_{j\neq i}^{n}[q-(j-1)]$$

$$\prod_{j\neq i}^{n}(x_i-x_j) = h^{n-1}\prod_{j\neq i}^{n}(i-j) = (-1)^{n-1}h^{n-1}\prod_{j=1}^{i-1}(i-j)\prod_{j=i+1}^{n}(j-i) = (-1)^{n-i}h^{n-1}(i-1)!(n-i)!$$

Cu acestea, polinomul Lagrange se scrie

$$P_{n-1}(x) = \sum_{i=1}^{n} \frac{\prod_{j\neq i}^{n}[q-(j-1)]}{(-1)^{n-i}(i-1)!(n-i)!} f_i \qquad (4.1.4)$$

şi obținem următoarea aproximație a integralei căutate

$$\int_a^b f(x)dx \approx \int_a^b P_{n-1}(x)dx = \sum_{i=1}^{n} A_i f_i \qquad (4.1.5)$$

cu coeficienți A_i dați de

$$A_i = \int_a^b \frac{\prod_{j\neq i}^{n}[q-(j-1)]dx}{(-1)^{n-i}(i-1)!(n-i)!} = \frac{h\int_0^{n-1}\prod_{j\neq i}^{n}[q-(j-1)]}{(-1)^{n-i}(i-1)!(n-i)!} \qquad (4.1.6)$$

Punând aceşti coeficienți sub forma $A_i = (b-a)H_i$, rezultă din (4.1.5) *formula de cuadratură Newton-Cotes,*

$$\int_a^b f(x)dx \approx (b-a)\sum_{i=1}^{n} H_i f_i \tag{4.1.7}$$

în care coeficienții H_i, numiți *coeficienți Cotes,* au expresiile

$$H_i = \frac{\int_0^{n-1}\prod_{j\neq i}^{n}\left[q-(j-1)\right]dq}{(-1)^{n-1}(i-1)!(n-i)!(n-1)}, \quad i = 1,2,...,n. \tag{4.1.8}$$

Se remarcă faptul că, potrivit definiției de mai sus, coeficienții Cotes sunt independenți atât de funcția integrată cât şi de intervalul de integrare. Trebuie menționat, de asemenea, că au loc proprietățile:

$$\sum_{i=1}^{n} H_i = 1, \quad H_i = H_{n-i+1.} \tag{4.1.9}$$

Prima dintre aceste relații poate fi demonstrată imediat punând în formula de cuadratură (4.1.7) $f(x) \equiv 1$, iar cea de a doua proprietate rezultă chiar din expresia (4.1.8) a coeficienților.

4.2. Formula trapezelor

4.2.1. Aspecte teoretice

Particularizând relațiile (4.1.8) pentru cazul n=2, obținem coeficienții Cotes

$$H_1 = -\int_0^1 (q-1)dq = \frac{1}{2} \tag{4.2.1}$$

$$H_2 = \int_0^1 q dq = \frac{1}{2}, \tag{4.2.2}$$

şi înlocuindu-i în formula de cuadratură (4.1.7), rezultă *formula trapezului:*

$$\int_{x_1}^{x_2} f(x)dx \approx \frac{h}{2}(f_1 + f_2). \tag{4.2.3}$$

Numele formulei se datorează faptului că, aşa cum se vede în figura de mai jos, ea poate fi obținută prin înlocuirea funcției $f(x)$ cu segmentul care uneşte punctele de coordonate (x_1, f_1) şi (x_1, f_2), valoarea integralei fiind aproximată de aria trapezului determinat prin proiectarea acestui segment pe axa absciselor.

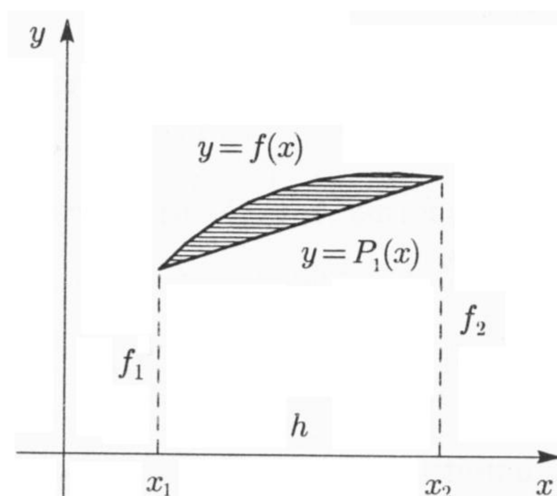


Fig. 4.2.1 Formula trapezului aproximează funcția prin dreapta care trece prin punctele *(x_1,f_1)* şi *(x_2,f_2)*.

Restul (eroarea) formulei de cuadratură (4.2.3), figurat haşurat în figura 4.2.1, poate fi dedus presupunând că $f(x)$ este continuă împreună cu primele două derivate pe $[a,b]$ $\left(f(x)dx \in C^{(2)}[a,b]\right)$. Astfel, exprimând R ca funcție de h avem:

$$R(h) = \int_{x_1}^{x_1+h} f(x)dx - \frac{h}{2}\left[f(x_1) + f(x_1+h)\right]$$

Derivând această relație de două ori în raport cu h, obținem:

$$R'(h) = \frac{1}{2}\left[f(x_1+h) - f(x_1)\right] - \frac{h}{2} f'(x_1+h)$$

$$R''(h) = -\frac{h}{2} f''(x_1+h).$$

Integrăm acum $R''(h)$ de două ori în raport cu h, observând că R(0) = R'(0) = 0 şi utilizând *teorema mediei.* Rezultă succesiv:

$$R'(h) = R'(0) + \int_0^h R''(t)dt = -\frac{1}{2} f''(\xi_1)\int_0^h t dt = -\frac{h^2}{4} f''(\xi_1)$$

$$R(h) = R(0) + \int_0^h R'(t)dt = -\frac{1}{4} f''(\xi)\int_0^h t^2 dt = -\frac{h^3}{12} f''(\xi),$$

unde $\xi, \xi_1 \in (x_1, x_1+1)$. Astfel, avem pentru restul formulei de cuadratură a trapezului

$$R = -\frac{h^3}{12} f''(\xi), \quad \xi \in (x_1, x_2). \qquad (4.2.4)$$

Se constată că restul are semn opus derivatei secunde din intervalul (x_1, x_2) şi, prin urmare, formula trapezului supraestimează valoarea integralei dacă $f'' > 0$ şi o subestimează în caz contrar.

Formula trapezului (4.2.3) nu prezintă interes ca atare, deoarece, implicând doar două valori ale integrandului, oferă precizie satisfăcătoare numai pentru integrarea funcțiilor liniare. Utilizând însă proprietatea de *aditivitate* a integralelor față de intervalul de integrare, vom generaliza acest rezultat în cele ce urmează pentru a obține o formulă de cuadratură de interes practic.

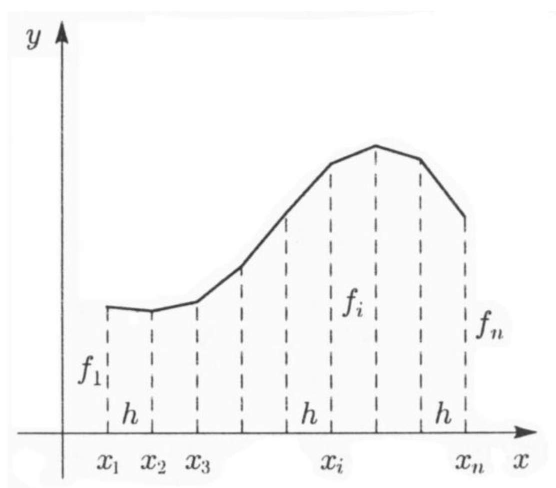


Figura 4.2.2. Formula trapezelor aproximează integrând cu linia poligonală definită de abscisele şi valorile corespunzătoare ale funcției

In acest scop împărțim, aşa cum se vede în figura 4.2.2, intervalul [*a,b*] în (*n*-1) subintevalele egale de lungime

$$h = (b-a)/(n-1) \qquad (4.2.5)$$

prin intermediul punctelor echidistante

$$x_i = a + (i-1)h, \quad i = 1,2,\ldots,n \qquad (4.2.6)$$

Desemnând prin $f_i \equiv f(x_i)$ valorile integrandului în nodurile rețelei, aplicăm formula trapezului (4.2.3) fiecărui subinterval $[x_1, x_2], \ldots, [x_{n-1}, x_n]$,

$$\int_a^b f(x)dx \approx \frac{h}{2}(f_1 + f_2) + \frac{h}{2}(f_2 + f_3) + \ldots + \frac{h}{2}(f_{n-1} + f_n)$$

şi, grupând termenii, obținem *formula trapezelor:*

$$\int_a^b f(x)dx \approx h\left[\frac{f_1}{2} + \sum_{i=2}^{n-1} f_i + \frac{f_n}{2}\right]. \qquad (4.2.7)$$

Geometric, această aproximație implică înlocuirea graficului funcției $y = f(x)$ cu linia poligonală care uneşte punctele $(x_1, f_1), (x_2, f_2), \ldots, (x_n, f_n)$.

Restul formulei trapezelor se compune din resturile corespunzătoare celor (n-1) subintervale de integrare,

$$R = -\frac{(n-1)h^3}{12} f''(\xi) = h\left[\frac{f_1}{2} + \sum_{i=2}^{n-1} f_i + \frac{f_n}{2}\right] \qquad (4.2.8)$$

şi prezintă o dependență tipică de h^2. Această dependență are ca şi consecință faptul că, spre exemplu, prin reducerea lungimii subintervalelor la valoarea h/2 sau, echivalent, prin dublarea numărului de subintervale considerând (2n-1) puncte de integrare, eroarea de integrare scade de 4 ori.

Funcția *Trapez*, prezentată în continuare, calculează integrale definite prin metoda trapezelor. Datele de intrare ale rutinei sunt limitele domeniului de integrare, a şi b, şi numărul punctelor de integrare, n. Numele funcției care trebuie integrată este transmis ca parametru corespunzător parametrului formal *Func*. Valoarea integralei este returnată prin numele funcției.

```
double Trapez (double a, double b, int n)
/* Calculeaza integrala functiei Func pe intervalul [a,b]
utilizand formula trapezelor cu n puncte de integrare */
{
     double h, s;
     int i;

     h = (b - a) / (n - 1);
     s = 0.5 * (Func(a) + Func(b));
     for (i = 1; i <= (n - 2); i++)
          s += Func(a + i * h);

     return h * s;
}
```

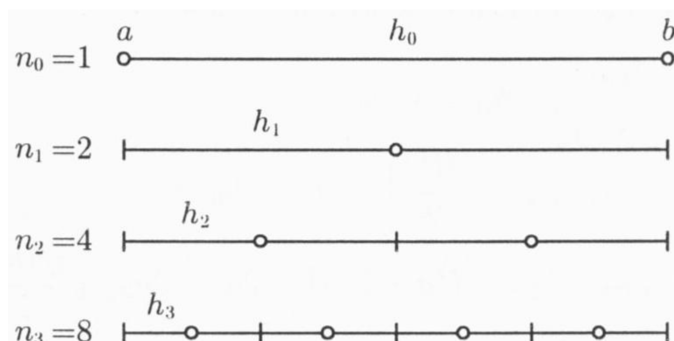


Figura 4.2.3. Schema de înjumătățire a intervalelor în metoda trapezelor cu control automat al pasului. Prin cerințe sunt figurate valorile care trebuie calculate în fiecare etapă

Pentru generarea absciselor de integrare, adunarea repetată a lungimii pasului, $x_{i+1} = x_i + h$, poate provoca acumularea de erori în cazul numerelor mari de puncte de integrare, fiind deci preferabilă utilizarea relației

$$x_{x+1} = a + ih\,, \quad i = 0,1,...,n\text{-}1.$$

In plus, valori generate cu ajutorul acesteia din urmă pot fi comunicate direct ca argumente în apelul funcției *Func.*

Controlul pasului şi implicit al preciziei de integrare se realizează în practică adesea prin proces de „exprimare" numerică. Acest proces implică de obicei calculul integralei pentru o valoare oarecare *h* a pasului, corespunzând unui număr *n* de puncte de integrare, şi compararea valorii astfel obținute cu valoarea rezultată pentru pasul *h*/2, corespunzând unui număr de (2*n*-1) puncte de integrare. Injumătățirea pasului trebuie continuată, în principiu, până când eroarea relativă pentru două aproximații consecutive ale integralei scade sub o valoare admisibilă ε.

Calculul integralei cu o precizie prestabilită prin metoda înjumătățirii pasului poate fi semnificativ optimizat observând că, potrivit figurii 4.2.3, care reprezintă rețelele punctelor de integrare pentru primele patru etape ale procesului de înjumătățire, nu este necesară evaluarea funcției în toate punctele de integrare, ci doar în cele (figurate prin cercuri) prin care au fost înjumătățite subintervalele rețelei utilizate în etapa anterioară. Restul valorilor funcției, deja calculate la paşii anteriori, sunt folosite implicit stabilind o relație de recurență între două aproximații consecutive ale integralei.

Primele aproximații furnizate de formula trapezelor pot fi scrise:

$$T_0 = h_0\left[\frac{f(a)}{2} + \frac{f(b)}{2}\right], \quad h_1 = b - a$$

$$T_1 = h_1\left[\frac{f(a)}{2} + f(a + h_1) + \frac{f(b)}{2}\right], \quad h_1 = h_0 / 2 \qquad (4.2.9)$$

$$T_2 = h_2\left[\frac{f(a)}{2} + f(a + h_2) + f(a + 2h_2) + f(a + 3h_2) + \frac{f(b)}{2}\right], \quad h_2 = h_1 / 2 \quad (4.2.10)$$

....

Se poate verifica fără dificultate că acestă succesiune de aproximații poate fi descrisă cu ajutorul procesului iterativ

$$T_0 = \frac{h}{2}\left[f(a) + f(b)\right], \quad h_0 = b - a, n_0 = 1 \qquad (4.2.11)$$

$$T_k = \frac{1}{2}\left[T_{k-1} + h_{k-1}\sum_{i=1}^{n_{k-1}} f\left(a + (i - 1/2)h_{k-1}\right)\right], \qquad (4.2.12)$$

$$h_k = h_{k-1} / 2, n_k = 2n_{k-1}, k = 1,2,... \qquad (4.2.13)$$

unde $h_k = (b-a)/n_k$ este lungimea subintervalelor după etapa *k*, iar $n_k = 2^k$ reprezintă numărul corespunzător de subintervale sau, echivalent, numărul punctelor noi de integrare şi de înjumătățire a subintervalelor pentru etapa următoare.

Procesul recurent (4.2.11) - (4.2.13) trebuie continuat până când diferența relativă dintre două aproximații succesive ale integralei devine mai mică sau egală cu o toleranță prestabilită ε. Pentru a evita împărțirea cu 0 implicată de evaluarea erorii relative în cazul $T_k = 0$, exprimăm criteriul de convergență sub forma

$$|T_k - T_{k-1}| \le \varepsilon |T_k|. \qquad (4.2.14)$$

Algoritmul (4.2.11) – (4.2.14) al metodei trapezului cu control automat al pasului de integrare se regăseşte sub forma funcției *TrapezControl* care urmează:

```
#include <stdio.h>
#include <math.h>
#define Pi 3.1415926535897932384626433832795

double Func(double x)
{
     return atan(x);
}

double TrapezControl(double a, double b)
/* Calculeaza integrala functiei Func pe intervalul [a,b]
utilizand formula trapezelor cu control automat al pasului de
integrare */
{
     const double eps = 1e-6;
     const int kmax = 30;
     double h, sum, t, t0;
     long i, n;
     int k;

     h = b - a;
     n = 1;
     t0 = 0.5 * h * (Func(a) + Func(b));     //aprox. initiala

     for (k = 1; k <= kmax; k++)
     {
          sum = 0.0;
          for (i = 1; i <= n; i++)
               sum += Func(a + (i - 0.5) * h);
          t = 0.5 * (t0 + h * sum);
          if (fabs(t - t0) <= eps * fabs(t))
               break;
          h *= 0.5;
          n *= 2;
          t0 = t;
     }

     if (k >= kmax)
```

```
        printf("TrapezControl:   nr.   maxim   de   iteratii
depasit!\n");

    return t;
}

void main()
{
    printf("integrala functiei atan(x) pe intervalul [-pi/2
3pi/2] are valoarea: %.8f\n",TrapezControl(-Pi/2, 3*Pi/2));
}
```

Parametri formali ai acestei rutine au aceeaşi semnificație cu cei ai funcției *Trapez*. Pentru ca numărul punctelor noi de integrare n, ca putere a lui 2, să nu depăşească valoarea maximă de tip *long* reprezentabilă (pentru majoritatea implementărilor limbajului C egală cu $2^{31}-1$), numărul de înjumătățiri ale subintervalelor este limitat prin variabila *kmax* la 30. Dacă, datorită convergenței slabe a procesului, se epuizează cele *kmax* iterații posibile fără a se fi atins precizia dorită *eps* în calculul integralei, se iese din ciclul înjumătățirilor cu variabila de control *k* incrementată la valoarea *kmax*+1 şi se emite un mesaj de eroare.

4.3. Formula lui Simpson

4.3.1. Aspecte teoretice

O formulă de cuadratură cu un grad de precizie mai ridicat decât formula trapezului se obține particularizând formulele Newton – Cotes pentru n = 3. Se obțin coeficienții Cotes:

$$H_1 = \frac{1}{4}\int_0^2 (q-1)(q-2)dq = \frac{1}{6} \tag{4.3.1}$$

$$H_2 = -\frac{1}{2}\int_0^2 q(q-2)dq = \frac{2}{3} \tag{4.3.2}$$

$$H_3 = \frac{1}{4}\int_0^2 q(q-1)dq = \frac{1}{6} \tag{4.3.3}$$

şi, având în vedere că $b-a \equiv x_3 - x_1 = 2h$, rezultă *formula lui Simpson*:

$$\int_{x_1}^{x_3} f(x)dx \approx \frac{h}{3}(f_1 + 4f_2 f_3). \tag{4.3.4}$$

Din punct de vedere geometric, această formulă implică înlocuirea curbei $y=f(x)$ cu parabola $y = P_2(x)$ definită de punctele $(x_1, f_1), (x_2, f_2)$ şi (x_3, f_3).

Admițând că $f(x)$ aparține clasei funcțiilor continue împreună cu primele patru derivate pe $[a,b](f(x) \in C^4[a,b])$, se poate obține o estimare a restului formulei lui Simpson printr-o tehnică similară celei utilizate în cazul formulei trapezului,

$$R = -\frac{h^5}{90} f^{(4)}(\xi), \xi \in [a,b] \tag{4.3.5}$$

Este important de observat că restul formulei lui Simpson depinde de h^5, în timp ce restul formulei trapezului depinde doar de h^3. Se constată că formula Simpson este exactă nu numai pentru polinoame de ordinul doi, ci şi pentru cele de ordinul trei.

Pentru a stabili o formulă de interes practic, care să asigure o precizie satisfăcătoare pentru un integrand arbitrar, se generalizează relația (4.3.4), similar formulei trapezelor,

făcând uz de *aditivitatea* integralei față de intervalul de integrare. Astfel, divizând intervalul [*a,b*] printr-un număr *impar*, $n=2m+1$, de puncte echidistante,

$$x_i = a + (i-1)h, \quad i = 1,2,...,n,$$

caracterizate prin echidistanța

$$h = \frac{b-a}{n-1} = \frac{b-a}{2m},$$

se poate aplica formula lui Simpson (4.3.4) pentru fiecare din cele m subintervale duble de lungime $2h$ determinate de cele trei puncte de rețea consecutive: $[x_1, x_3], [x_3, x_5], ..., [x_{n-2}, x_n]$. Geometric, această abordare revine la aproximarea integrandului prin arce de parabolă pe fiecare pereche de subintervale consecutive. Este evident că pentru un număr par de puncte de rețea, formula lui Simpson nu poate fi aplicată de un număr întreg de ori. Cu toate acestea, integrala pe întregul interval [*a,b*] poate fi scrisă:

$$\int_a^b f(x)dx \approx \frac{h}{3}(f_1 + 4f_2 + f_3) + \frac{h}{3}(f_3 + 4f_4 + f_5) + ... + \frac{h}{4}(f_{n-2} + 4f_{n-1} + f_n).$$

Regrupând termenii rezultă de aici *formula lui Simpson generalizată*,

$$\int_a^b f(x)dx \approx \frac{h}{3}(f_1 + 4\sigma_2 + 2\sigma_1 + f_n), \tag{4.3.6}$$

unde

$$\sigma_1 = \sum_{i=1}^{(n-3)/2} f_{2i+1}, \quad \sigma_2 \sum_{i=1}^{(n-1)/2} f_{2i} \tag{4.3.7}$$

Suma σ_1 se compune din termenii de indice impar, în timp ce σ_2 însumează termenii cu indice par.

Admițând că $f(x) \in C^{(4)}[a,b]$, restul formulei lui Simpson generalizate se obține însumând resturile de tipul (4.3.5) pentru fiecare din cele m subintervale de lungime $2h$ ale intervalului [*a,b*]:

$$R = -\frac{mh^5}{90} f^{(4)}(\xi) = -\frac{(b-a)h^4}{180} f^{(4)}(\xi), \quad \xi \in [a,b]. \tag{4.3.8}$$

Funcția *Simpson0*, listată în continuare, codifică întocmai formulele (4.3.6) – (4.3.7). Semnificația parametrilor este aceeaşi cu cea a parametrilor funcției *Trapez*, descrisă anterior, şi numele funcției utilizator este transmis şi aici prin lista de argumente.

```
float Simpson0 (float Func(float), float a, float b, int n)
/* Calculeaza integrala functiei Func pe intervalul [a,b]
utilizand formula lui Simpson cu n puncte de integrare */
{
     float h, s1, s2;
     int i, par = 0;

     if ((n/2)*2 == n) n++;     //incrementeaza n daca este par

     h = (b - a) / (n - 1);
     s1 = s2 = 0.0;
     for (i = 1; i <= (n - 2); i++) {
          if (par) s1 += Func(a + i * h);
            else s2 += Func(a + i * h);
          par = ~par;
     }
```

```
	return (h/3) * (Func(a) + 4*s2 + 2*s1 + Func(b));
}
```

In situațiile în care în mod eronat numărul punctelor de integrare *n*, comunicat rutinei, este par, el este incrementat. Adunarea alternativă a termenilor la o sumă sau alta este controlată de valorile variabilei întregi *par*, care, prin acțiunea operatorului „~” de incrementare bit-cu-bit, ia alternativ valori nule şi nenule.

Se poate realiza o codificare mai eficientă rescriind relațiile (4.3.7) sub forma

$$\sigma_1 = \sum_{i=1}^{(n-3)/2} f(x_{2i+1}), x_{2i+1} = a + i(2h) \tag{4.3.9}$$

$$\sigma_2 = f(x_2) + \sum_{i=1}^{(n-3)/2} f(x_{2(i+1)}), x_{2(i+1)} = x_{2i+1} + h, \tag{4.3.10}$$

astfel încât numărul termenilor din cele două sume să fie egal, indicele *i* parcurgând acelaşi şir de valori. In implementarea dată mai jos nu mai este necesară structura de selecție *if*, deoarece la fiecare iterație este adunat câte un termen fiecărei sume.

```
float Simpson (float Func(float), float a, float b, int n)
/* Calculeaza integrala functiei Func pe intervalul [a,b]
utilizand formula lui Simpson cu n puncte de integrare */
{
	float h, h2, s1, s2, x;
	int i;

	if ((n/2)*2 == n) n++;    //incrementeaza n daca este par

	h = (b - a) / (n - 1); h2 = 2*h;
	s1 = 0.0; s2 = Func(a+h);
	for (i = 1; i <= (n - 3)/2; i++) {
		x = a + i*h2;
		s1 += Func(x); s2 += Func(x+h);
	}

	return (h/3) * (Func(a) + 4*s2 + 2*s1 + Func(b));
}
```

Ca şi în cazul metodei trapezelor, se poate concepe un algoritm având la bază formula lui Simpson, care să realizeze în mod automat adaptarea pasului de integrare pentru efectuarea cuadraturii cu o anumită precizie prestabilită ε. In esență, trebuie comparate aproximațiile calculate recent ale integralei, corespunzătoare valorilor *h* şi *h/2* ale pasului de integrare, înjumătățind pasul până când eroarea relativă pentru două aproximații consecutive scade sub valoarea ε.

Având în vedere aproximațiile date de formula trapezelor pentru valori ale pasului de integrare înjumătățite succesiv, se poate verifica fără dificultate că aproximațiile formulei lui Simpson pot fi exprimate în raport cu cele dintâi:

$$S_1 = \frac{h_1}{3}[f(a) + 4f(a+h_1) + f(b)] = \frac{4T_1 - T_0}{3}$$

$$S_2 = \frac{h_2}{3}[f(a) + 4f(a+h_2) + 2f(a+h_2) + 4f(a+3h_2) + f(b)] = \frac{4T_2 - T_1}{3}$$

.... (4.3.11)

Generalizând aceste rezultate, avem

$$S_k = \frac{4T_k - T_{k-1}}{3} \tag{4.3.12}$$

Cu acestea, algoritmul pentru calculul recursiv al aproximațiilor formulei lui Simpson poate fi descris cu ajutorul relațiilor:

$$T_0 = \frac{h_0}{2}[f(a) + f(b)], \quad h_0 = b - a, \quad n_0 = 1 \tag{4.3.13}$$

$$T_k = \frac{1}{2}\left[T_{k-1} + h_{k-1}\sum_{i=1}^{n_{k-1}} f(a + (i - 1/2)h_{k-1})\right], \quad k = 1,2,... \tag{4.3.14}$$

$$S_k = \frac{4T_k - T_{k-1}}{3}, \quad h_k = h_{k-1}/2, \quad n_k = 2n_{k-1}. \tag{4.3.15}$$

Acest proces este continuat până când eroarea relativă în calculul integralei devine mai mică sau egală cu toleranța prestabilită ε. Ca şi în cazul metodei trapezelor, pentru a trata unitar şi situațiile în care $S_k = 0$, scriem acest criteriu sub forma:

$$|S_k - S_{k-1}| \le \varepsilon |S_k|. \tag{4.3.16}$$

Argumentele, variabilele şi constantele locale ale funcției *SimpsonControl*, scrisă pe baza acestui algoritm, păstrează semnificația pe care o au în cazul funcției *TrapezControl.*

```
#include <stdio.h>
#include <math.h>
#define Pi 3.14159265358979323846264338327950

float func(float x)
{
     return atan(x);
}

float SimpsonControl(float Func(float), float a, float b)
/* Calculeaza integrala functiei Func pe intervalul [a,b]
utilizand formula lui Simpson cu control automat al pasului de
integrare */
{
     const float eps = 1e-6;  //precizia relativa a integralei
     const int kmax = 30;   //numar maxim de injumatatiri
     float h, s, s0, sum, t, t0;
     long i, n;
     int k;

     h = b - a;
     n = 1;
     s0 = t0 = 0.5 * h *(Func(a) + Func(b)); //aprox. initiala

     for (k = 1; k <= kmax; k++)
     {
          sum = 0.0;
          for (i = 1; i <= n; i++)
               sum += Func(a + (i - 0.5) * h);
          t = 0.5 * (t0 + h*sum);
```

```
            s = (4*t - t0)/3;                   //noua aproximatie
            if (fabs(s - s0) <= eps * fabs(s))
                break;                          //testeaza convergenta
            h *= 0.5;
            n *= 2; s0 = s;
            t0 = t;
      }

      if (k >= kmax)
            printf("SimpsonControl:   nr.   maxim   de   iteratii
depasit!\n");

      return s;
}

void main()
{
      printf("integrala functiei atan(x) pe intervalul [-pi/2
3pi/2] are valoarea: %.8f\n",SimpsonControl(func,-Pi/2,
3*Pi/2));
}
```

4.3.2. Desfăşurarea laboratorului

1. Se va aplica algoritmul metodei trapezelor şi algoritmul metodei lui Simpson, ambele cu control automat al pasului de integrare, pentru a calcula valoarea integralelor de la punctele a)...e). Precizia relativă va fi 6 zecimale iar rezultatele se vor afişa cu 8 zecimale. Se va alcătui un tabel cu diferenţele între valoarea integralei calculată direct şi cea rezultată în urma aplicării metodei trapezelor respectiv Simpson pentru fiecare din cele 5 integrale.

a) $\int_2^7 \frac{1}{x} dx = \ln|x| \Big|_2^7$

b) $\int_{-1}^1 x^{10} dx = \left.\frac{x^{11}}{11}\right|_{-1}^1$

c) $\int_{-5}^0 e^x dx = e^x \Big|_{-5}^0$

d) $\int_0^\pi \sin x dx = -\cos x \Big|_0^\pi$

e) $\int_{-\frac{\pi}{2}}^{\frac{3\pi}{2}} \arctan x dx = x \arctan x - \left.\frac{1}{2\ln(1+x^2)}\right|_{-\frac{\pi}{2}}^{\frac{3\pi}{2}}$

L5. Rezolvarea numerică a ecuațiilor diferențiale ordinare

5.1. Metoda Euler

5.1.1. Aspecte teoretice

Problemele modelate cu ecuații diferențiale ordinare se caracterizează printr-o singură variabilă independentă şi una sau mai multe variabile dependente. În aplicațiile științifice şi tehnice, variabila dependentă este timpul (t) sau spațiul (x). Indiferent de ordinul ecuației diferențiale, care rezultă din modelarea matematică a problemei, rezolvarea numerică se poate reduce întotdeauna la o ecuație diferențială de ordinul I.

De exemplu, o ecuație diferențială de ordinul II:

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

este echivalentă cu un sistem de două ecuații diferențiale de ordinul I:

$$\frac{dy}{dx} = z(x)$$

$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

În general, o ecuație diferențială nu are soluție unică. În aplicațiile concrete, se poate obține soluție unică dacă se cunosc valorile inițiale ale funcției necunoscute (*problemă Cauchy*) sau valorile pe frontiera domeniului de definiție (*problemă Dirichlet*).

Formularea completă a unei probleme ce constă în rezolvarea unei ecuații diferențiale de ordinul I (*problemă Cauchy*) este:

$$y' = f(x, y)$$
$$y(x_0) = y_0 \quad ,$$

unde x_0 este valoarea inițială a variabilei independente.

Metoda Euler este foarte importantă, nu atât din punct de vedere practic (se acumulează erori mari de aproximare) cât mai ales conceptual, pentru înțelegerea metodelor numerice evoluate, de mare eficiență.

Fie problema Cauchy:

$$y' = f(x, y)$$
$$y(x_0) = y_0, \tag{5.1.1}$$

Dezvoltăm funcția *y(x)* în serie Taylor, într-o vecinătate a punctului x_0:

$$y(x) = y(x_0) + y'(x_0)(x - x_0) + \frac{(x - x_0)^2}{2!} y''(x_0) + \dots ,$$

Introducem pasul de discretizare $h = x - x_0$ şi rescriem relația de mai sus neglijând termenii de ordin superior lui 2:

$$y(x) = y(x_0) + h \cdot y'(x_0) \; .$$

Termenii din partea dreaptă sunt cunoscuți din relațiile (5.1.1) iar valoarea lui $y(x)$ se evaluează în $x = x_0 + h$:

$y(x_0 + h) = y(x_0) + h \cdot f(x_0, y_0)$.

Notăm $y(x_0) = y_0$, $y(x_0 + k \cdot h) = y_k$, $k = 1, 2, \ldots, n$, şi obţinem:

Pasul 1: $y_1 = y_0 + h \cdot f(x_0, y_0)$,

Pasul 2: $y_2 = y_1 + h \cdot f(x_1, y_1)$

Pasul 3: $y_3 = y_2 + h \cdot f(x_2, y_2)$

. . . .

Pasul n+1: $y_{n+1} = y_n + h \cdot f(x_n, y_n)$, unde $x_n = x_0 + n \cdot h$.

Funcţia necunoscută $y(x)$ se determină astfel prin valorile sale în puncte echidistante: $y_1,\ y_2,\ y_3,\ \ldots\ y_n$.

Programul în C++ care determină valorile funcţiei *y(x)* este listat mai jos.

```
#include <stdio.h>
 float f(float x, float y) //funcţia f(x,y)
    {
    return y /(y-x);
    }

 int main(void)
 {
   float x=0.0,  y=1.0, h=0.5; // valorile iniţiale x0, y0, h
   int n=0;
   printf("n=%3d x=%8.4f y=%8.4f \n",n,x,y);

   for(n=1;n<=10;n++) {
     y=y+h*f(x,y);            // relaţia y_{n+1} = y_n + h·f(x_n, y_n)
     x=x+h;
     printf("n=%3d x=%8.4f y=%8.4f \n",n,x,y);
     }
     return 0;
}
```

5.2. *Metoda Euler modificată*

5.2.1. Aspecte teoretice

Dezavantajul principal al algoritmului Euler îl constituie eroarea de calcul cumulată (determinată de neglijarea termenilor de ordin superior lui 2). Pentru a reduce această eroare se impune folosirea unui pas de discretizare *h*, foarte mic, ceea ce duce la creşterea timpului de execuţie. Agoritmul lui Euler poate fi mai eficient dacă se utilizează în calcul, valoarea funcţiei *f(x,y)* la ambele capete ale intervalui de discretizare *h*. De fapt, la fiecare pas, valoarea y_{n+1} se calculează în două etape. Prima etapă este cea din algoritmul Euler iar în a doua etapă se efectuează o corecţie a valorii obţinute.

Pas 1: $y_1 = y_0 + h \cdot f(x_0, y_0)$,

$y_1' = f(x_1, y_1)$;

Pas 1 – corecție: $y_1 = y_0 + h \cdot (y_0' + y_1')/2$,

$y_1' = f(x_1, y_1)$;

Pas 2: $y_2 = y_1 + h \cdot f(x_1, y_1)$,

$y_2' = f(x_2, y_2)$;

Pas 2 – corecție: $y_2 = y_1 + h \cdot (y_1' + y_2')/2$,

$y_2' = f(x_2, y_2)$;

. . . .

Pas n+1: $y_{n+1} = y_n + h \cdot f(x_n, y_n)$,

$y_{n+1}' = f(x_n, y_n)$;

Pas n+1 – corecție: $y_{n+1} = y_n + h \cdot (y_n' + y_{n+1}')/2$,

$y_{n+1}' = f(x_{n+1}, y_{n+1})$;

unde $x_n = x_0 + n \cdot h$.

În etapa a II-a, se corectează valoarea lui y_{n+1} considerând media derivatelor în punctele x_n şi x_{n+1}. Pentru creşterea preciziei, secvența de corecție se poate relua încă o dată. Cerința de a folosi un pas de discretizare h, cât mai mic se menține şi în cazul metodei Euler modificate.

Programul în C++ care determină valorile funcției *y(x)* este listat în continuare. Problema Cauchy din program este:

$y' = \dfrac{y}{y - x}$; $x \in [0, \infty)$

$y_0 = y(0) = 1$; (Soluția analitică, exactă, este $y(x) = x + \sqrt{x^2 + 1}$).

```
// Algoritmul Euler -modificat
#include <stdio.h>
#include <math.h>
float f(float x, float y)
      {
      return y/(y-x);
      }

int main(void)
{
float x=0.0,  y=1.0, h=0.05, y1, y2;
int n=0;

printf("n=%3d x=%8.4f y=%8.4f \n",n,x,y);
   for(n=1;n<=50;n++) {
      y1=f(x,y);
```

```
        y=y+h*f(x,y);
        x=x+h;y2=f(x,y);
        y=y+h*(y1+y2)/2;
        printf("n=%3d x=%8.4f y=%8.4f  val_ex%8.4f \n",
                                   n,x-h,y,x+sqrt(x*x+1));
        }
return 0;}
```

5.3. Metode Runge – Kutta

5.3.1. Aspecte teoretice

Se bazează pe evaluări ale funcţiei *f(x,y)* (cunoscută ca expresie analitică) şi derivatelor sale în raport cu variabila independentă *x*, în mai multe puncte ale intervalului de integrare. Se asigură astfel o precizie mai bună, ca urmare a faptului că funcţia *f(x,y)* este dată iar derivatele de ordin superior ale funcţiei necunoscute *y(x)* sunt exprimate prin derivate parţiale de ordin superior ale lui *f(x,y(x))*.

Fie problema Cauchy:

$$y'(x) = f(x, y);$$

$$y(x_0) = y_0;$$

Se evaluează derivatele de ordin superior ale funcţiei *y(x)*:

$$y'(x) = f(x, y);$$

$$y''(x) = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\cdot\frac{dy}{dx} = f_x(x, y) + f_y(x, y)\cdot f(x, y) = f_x + f_y f \qquad (5.3.1)$$

$$y'''(x) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial x\partial y}\frac{dy}{dx} + \frac{\partial^2 f}{\partial x\partial y}\frac{dy}{dx} + \frac{\partial^2 f}{\partial y^2}\left(\frac{dy}{dx}\right)^2 + \frac{\partial f}{\partial y}\frac{d^2 y}{dx^2} =$$

$$= f_{xx} + f_{xy}y' + f_{xy}y' + f_{yy}(y')^2 + f_y y'' = f_{xx} + 2ff_{xy} + f_{yy}f^2 + f_y(f_x + ff_y) = \qquad (5.3.2)$$

$$= f_{xx} + 2ff_{xy} + f_{yy}f^2 + f_x f_y + ff_y{}^2.$$

Algoritmul Runge – Kutta de ordinul 2, se bazează pe dezvoltarea funcţiei *y(x)* în serie Taylor luând în consideraţie termenii până la ordinul 2 de derivare. Formulele de bază, iterative, sunt:

$$y_{n+1} = y_n + ak_1 + bk_2 \quad unde \qquad (5.3.3)$$

$$k_1 = hf(x_n, y_n),$$

$$k_2 = hf(x_n + \alpha\cdot h, y_n + \beta\cdot k_1).$$

Constantele a, b, α, β, urmează a fi determinate din condiţia ca (5.3.3) să reprezinte o dezvoltare în serie Taylor a funcţiei *y(x)* în jurul punctului x_n, pentru $x = x_{n+1}$:

$$y(x_{n+1}) = y(x_n) + hy'(x_n) + \frac{h^2}{2}y''(x_n) + \frac{h^3}{6}y'''(x_n) + \cdots\cdots =$$

$$= y(x_n) + hf(x_n, y_n) + \frac{h^2}{2}(f_x + ff_y)_n + \frac{h^3}{6}(f_{xx} + 2ff_{xy} + f_{yy}f^2 + f_x f_y + ff_y{}^2)_n + \cdots \qquad (5.3.4)$$

Cu indicele *n* la paranteze, am indicat că evaluarea funcţiei se face în (x_n, y_n).

Pe de altă parte, folosind dezvoltarea în serie Taylor pentru funcții de două variabile, în jurul punctului (x_n, y_n), obținem:

$$f(x_n + \alpha \cdot h,\ y_n + \beta \cdot k_1) = f(x_n, y_n) + \alpha \cdot hf_x + \beta \cdot k_1 f_y + \frac{\alpha^2 h^2}{2} f_{xx} + \alpha \cdot h \cdot \beta \cdot k_1 f_{xy} + \frac{\beta^2 k_1^2}{2} f_{yy} + \cdots \quad (5.3.5)$$

unde toate derivatele au fost evaluate în punctul (x_n, y_n).

În expresia (5.3.3), înlocuim parametrii k_1, k_2 folosind expresia (5.3.5):

$$y_{n+1} = y_n + (a+b)h \cdot f + b \cdot h^2(\alpha \cdot f_x + \beta \cdot ff_y) + bh^3\left(\frac{\alpha^2}{2} f_{xx} + \alpha\beta f_{xy} + \frac{\beta^2}{2} f^2 f_{yy}\right) + \cdots \quad (5.3.6)$$

Comparând expresiile (5.3.4) cu (5.3.6) și identificând coeficienții pentru h și h^2 obținem relațiile:

$$a + b = 1$$
$$b \cdot \alpha = b \cdot \beta = \frac{1}{2} \quad (5.3.7)$$

Sistemul (5.3.7) are mai multe soluții, fiind format din trei ecuații cu patru necunoscute. O soluție simplă și echilibrată este

$$a = b = \frac{1}{2}, \qquad \alpha = \beta \quad (5.3.8)$$

Algoritmul Runge – Kutta de ordinul 2:

Pentru problema Cauchy:

$y'(x) = f(x, y);$

$y(x_0) = y_0;$

Se alege pasul de integrare *h*, $x_n = x_0 + n \cdot h$ și se calculează pentru *n=1, 2, 3, . . . N,* valorile funcției $y(x_n)$:

$k_1 = hf(x_n, y_n),$

$k_2 = hf(x_n + h,\quad y_n + k_1).$

$y_{n+1} = y_n + \frac{1}{2}k_1 + \frac{1}{2}k_2.$

Algoritmul este de ”ordinul 2” pentru că în seriile Taylor se iau în considerație termenii ce conțin derivate până la ordinul 2, inclusiv.

Algoritmul Runge – Kutta de ordinul 4 este similar dar se iau în considerație termenii ce conțin derivate până la ordinul 4, inclusiv; în final, din identificarea termenilor ce conțin h, h^2, h^3, h^4, se obțin 11 ecuații cu 13 necunoscute, deci două necunoscute se aleg arbitrar.

Algoritmul Runge – Kutta de ordinul 4:

Pentru problema Cauchy:

$y'(x) = f(x, y);$

$y(x_0) = y_0;$

Se alege pasul de integrare *h*, $x_n = x_0 + n \cdot h$ și se calculează pentru *n=1, 2, 3, . . . N,* valorile funcției $y(x_n)$:

$$k_1 = hf(x_n, y_n),$$

$$k_2 = hf\left(x_n + \frac{1}{2}h, \quad y_n + \frac{1}{2}k_1\right),$$

$$k_3 = hf\left(x_n + \frac{1}{2}h, \quad y_n + \frac{1}{2}k_2\right),$$

$$k_4 = hf(x_n + h, \quad y_n + k_3),$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

Algoritmul Runge – Kuta devine şi mai performant prin controlul adaptiv al pasului de iterație, h; când viteza de variație a funcției y(x) creşte, trebuie micşorat pasul iar când funcția are o variație lentă, pasul poate fi mărit. Viteza de variație a unei funcții este dată de valoarea derivatei sale.

```
      // Runge-Kutta ordin 2:
#include <stdio.h>
#include <math.h>
float f(float x, float y)
      {
      return y/(y-x);
      }

int main(void)
{
float x=0.0,  y=1.0, h=0.05, k1, k2 ;
 int n=0;
 printf("n=%3d x=%8.4f y=%8.4f \n",n,x,y);

   for(n=1;n<=50;n++) {
      k1=h*f(x,y);
      k2=h*f(x+h,y+k1);
      y=y+(k1+k2)/2;
      x=x+h;
      printf("n=%3d x=%8.4f y=%8.4f 
val_ex:%8.4f\n",n,x,y,x+sqrt(x*x+1));
      }
      return 0;
}
```

```
 // Runge-Kutta de ordin 4

#include <stdio.h>
#include <math.h>
float f(float x, float y)
      {
      return y/(y-x);
      }
```

```
int main(void)
{
float x=0.0,  y=1.0, h=0.05, k1, k2, k3, k4 ;
 int n=0;
 printf("n=%3d x=%8.4f y=%8.4f \n",n,x,y);

   for(n=1;n<=50;n++) {
      k1=h*f(x,y);
      k2=h*f(x+h/2.0,y+k1/2.0);
      k3=h*f(x+h/2.0,y+k2/2.0);
      k4=h*f(x+h,y+k3);
      y=y+(k1+2*k2+2*k3+k4)/6.0;
      x=x+h;
      printf("n=%3d x=%8.4f y=%8.4f
val_ex:%8.4f\n",n,x,y,x+sqrt(x*x+1));
      }
      return 0;
     }
```

5.3.2. Desfăşurarea laboratorului

1. Se va aplica algoritmul metodei Euler modificată şi algoritmul metodei Runge – Kutta pentru a rezolva ecuațiile diferențiale de la punctele a)...c). La fiecare ecuație, pentru soluțiile ultimilor 4 paşi, se determină eroarea procentuală față de valoarea exactă după formula:

$$procent = \frac{|exact - aprox|}{exact} \times 100 .$$

a) $y' = \frac{3-4y}{2x}, \quad y(1) = -4$; soluția analitică: $y(x) = \frac{3}{4} - \frac{19}{4x^2}$

b) $y' = \frac{x}{y}, \quad y(2) = -1$; soluția analitică: $y(x) = -\sqrt{x^2 - 3}$

c) $y' = 2 - e^{-4x} - 2y, \quad y(0) = 1$; soluția analitică: $y(x) = 1 + \frac{1}{2}e^{-4x} - \frac{1}{2}e^{-2x}$

L6. Utilizarea mediului MATLAB

6.1. Introducere

6.1.1. Aspecte generale

Mediul MATLAB este un software dedicat conceput pentru rezolvarea numerică a unei game de probleme matematice. MATLAB este format din două elemente principale: nucleul programului şi Simulink.

Simulink este de fapt o extensie a MATLAB care permite simularea unor sisteme şi fenomene reale sub formă grafică prin utilizarea de blocuri elementare interconectate.

Nucleul MATLAB se bazează pe un limbaj de programare foarte asemănător cu limbajul C (din care este derivat) şi un set puternic de funcţii predefinite pentru calcul matematic, procesări numerice şi grafice în două sau trei dimensiuni.

6.1.2. Limbajul de programare MATLAB

Cea mai importantantă particularitate a limbajului de programare este faptul că lucrează nativ cu vectori şi matrici. Astfel, alocarea de spaţiu în memorie se face în mod dinamic iar operaţiile matematice elementare (adunare, scădere, înmulţire, împărţire) se pot aplica direct matricilor sau între matrici şi scalari (aceştia sunt consideraţi tot matrici cu o singură dimensiune). Trebuie reţinut că indexarea matricilor se face pornind de la 1 şi nu de la 0 aşa cum era cazul în C.

De asemenea limbajul MATLAB poate lucra direct şi foarte intuitiv cu numere complexe. Astfel, declararea unei variabile complexe se face utilizând notaţia matematică *a+jb* sau *a+ib*.

In MATLAB variabilele se pot utiliza direct, fără a fi necesară o declarare prealabilă a acestora. Se face diferenţierea între litere mici şi litere mari.

Programele în MATLAB se scriu sub formă de fişiere cu extensia .m la care prima linie trebuie să fie de forma *function nume(p1,p2,...,pn)*. Nume trebuie să fie identic cu cel al fişierului iar între paranteze se vor trece argumentele pe care le acceptă funcţia (nu trebuie precizat tipul lor). Aceste fişiere pot fi apelate ulterior din alte fişiere MATLAB, la fel cum se face apelul unor funcţii în limbajul C.

Fiecare linie dintr-un fişier .m trebuie să se termine prin caracterul „ ; ”. Atunci când este necesară utilizarea unui bloc de instrucţiuni ele nu trebuie încadrate între caracterele acolade aşa cum se proceda în limbajul C.

Structuri decizionale

In MATLAB se folosesc două structuri decizionale: *if* şi *switch*.

Sintaxa structurii *if* este:

```
if condiţie
        <instrucţiuni>
    elseif condiţie
        <instrucţiuni>
    else
        <instrucţiuni>
    end
```

Utilizarea ramurilor *elseif* şi *else* este opţională.

Sintaxa structurii *switch* este:

```
switch variabilă
case valoare1
```

<instrucțiuni>
case {*valoare2, valoare3*}
<instrucțiuni>
case *valoare4*
<instrucțiuni>
otherwise
<instrucțiuni>
end

Utilizarea ramurii *otherwise* este opțională. Trebuie remarcat că, spre deosebire de limbajul C, structura *switch* se opreşte după executarea instrucțiunilor corespunzătoare cazului identificat.

Structura *for*

Sintaxa structurii *for* este următoarea:
for *contor*=*valoare inițială* : *pas* : *valoare finală*
<instrucțiuni>
end

Structura *while*

Structura *while* are următoarea sintaxă:
while *condiție*
<instrucțiuni>
end

Utilizarea expresiilor simbolice

MATLAB permite operarea cu expresii simbolice şi obținerea de rezultate exprimate în mod analitic. Declararea unei variabile simbolice se face utilizând cuvântul cheie *syms*:

syms x y;
y=x^2+3*x+4;

In exemplul de mai sus variabila *y* nu va fi evaluată numeric ci va reține doar expresia analitică prin care a fost definită. Pentru a evalua numeric o variabilă simbolică se utilizează funcția *vpa(s,d)*. Variabila *s* reprezintă o expresie simbolică iar *d* reprezintă numărul de zecimale dorit pentru reprezentare.

6.2. Rezolvarea ecuațiilor

6.2.1. Aspecte teoretice

In continuare este prezentat modul în care pot fi rezolvate ecuațiile algebrice în MATLAB utilizând reprezentarea simbolică. Funcția folosită în acest scop este:

solve('ec', 'var')

unde *ec* este o expresie simbolică a ecuației de rezolvat iar *var* reprezintă variabila în raport cu care se găseşte soluția. Dacă funcția *solve* nu poate identifica o expresie analitică a soluției atunci va întoarce soluția numerică.

Algoritmul de mai jos rezolvă ecuația $x = e^{-x}$ şi afişează rezultatul în formă analitică şi numerică.

```
function laborator6();
     % fisierul trebuie sa se numeasca laborator6.m
syms x y;
     % se declara variabilele simbolice x si y
y=solve('x=exp(-x)', 'x');
     % se rezolva ecuatia in raport cu x
```

```
disp('Solutia analitica a ecuatiei x=exp(-x):');
disp(y);
disp('Solutia numerica a ecuatiei x=exp(-x):');
disp(vpa(y,6));
```

6.2.2. Desfăşurarea laboratorului

1. Se verifică algoritmul prezentat.

6.3. Rezolvarea sistemelor de ecuaţii

6.3.1. Aspecte teoretice

Pentru a rezolva sisteme de ecuaţii liniare vom utiliza funcţiile implicite de operare cu matrici de care dispune MATLAB.

Fie sistemul:

$$\begin{cases} 2x_1 + x_2 + 2x_3 = 0 \\ x_1 + 3x_2 + x_3 = 0 \\ 2x_1 + x_2 + x_3 = 1 \end{cases} \tag{6.1}$$

Acesta poate fi scris matricial sub forma:

$$A \cdot x = b \tag{6.2}$$

Ecuaţia (6.2) poate fi rezolvată împărţind la stânga cu matricea A. In MATLAB există un operator special pentru acest scop reprezentat prin simbolul „\". Pentru a efectua această operaţie MATLAB utilizează algoritmi de tip Gauss.

Programul prezentat mai jos rezolvă sistemul (6.1) şi afişează soluţia acestuia.

```
function laborator6();
     % fisierul trebuie sa se numeasca laborator6.m
A=[2,1,2; 1,3,1; 2,1,1];
b=[0; 0; 1];
     % se declara si se initializeaza matricile A si b
x=A\b;
     % se rezolva sistemul de ecuatii
disp('Solutia sistemului:');
disp(x);
```

6.3.2. Desfăşurarea laboratorului

1. Se verifică algoritmul prezentat.

6.4. Interpolarea

6.4.1. Aspecte teoretice

Mediul MATLAB dispune de o serie de funcţii care permit operaţii de interpolare atât unidimensionale cât şi multidimensionale.

Una din cele mai uzuale funcţii pentru interpolare unidimensională este *interp1*. Această funcţie utilizează algoritmi polinomiali şi are următoarea sintaxă:

yi=interp1(x,y,xi,'method') unde:

yi – vector care conţine valorile iniţiale şi cele interpolate

y=f(x) – funcţia ale cărei valori se interpolează

xi – vector care defineşte punctele în care se realizează interpolarea

method – poate fi: *nearest* (punctul interpolat ia valoarea celui mai apropiat punct); *linear* (punctul interpolat este identificat pe dreapta care uneşte punctele vecine); *spline* (se utilizează funcții cubice spline); *pchip* (se utilizează polinoame Hermite).

O funcție similară cu *interp1* este *interp2*. Aceasta realizează interpolare bidimensională şi are următoarea sintaxă:

zi=interp2(x,y,z,xi,yi,'method')

unde:

zi – matrice care conține valorile inițiale şi cele interpolate

z=f(x,y) – funcția bidemnsională ale cărei valori se interpolează

xi, yi – matrici care definesc punctele în care se realizează interpolarea

method – poate fi: *nearest* (punctul interpolat ia valoarea celui mai apropiat punct); *bilinear* (punctul interpolat este determinat de o combinație între cele mai apropiate 4 puncte vecine); *bicubic* (punctul interpolat este determinat de o combinație între cele mai apropiate 16 puncte vecine).

Programul prezentat mai jos interpolează funcția $\frac{7-x^2}{x-3}$ pe intervalul [0 2] şi reprezintă grafic diferențele între diversele metode de interpolare oferite de funcția *interp1*.

```
function laborator6();
     % fisierul trebuie sa se numeasca laborator6.m
x=0:.1:2;
     % simuleaza punctele in care se cunoaste functia
y=(7-x.^2)./(x-3);
     % genereaza functia in punctele simulate
xi=0:.05:2;
yi=interp1(x,y,xi,'nearest');
     % realizeaza interpolarea
subplot 221;
plot(x,y,xi,yi,'--red');
     % reprezinta grafic functia originala si cea
     % interpolata
title('Interpolare nearest');
grid;
yi=interp1(x,y,xi,'linear');
subplot 222;
plot(x,y,xi,yi,'--red');
title('Interpolare linear');
grid;
yi=interp1(x,y,xi,'spline');
subplot 223;
plot(x,y,xi,yi,'--red');
title('Interpolare spline');
```

```
grid;
yi=interp1(x,y,xi,'pchip');
subplot 224;
plot(x,y,xi,yi,'--red');
title('Interpolare pchip');
grid;
```

6.4.2. Desfăşurarea laboratorului

1. Se verifică funcționarea algoritmului prezentat pentru interpolarea funcției $\frac{7-x^2}{x-3}$ pe intervalul [0 2].

2. Se modifică algoritmul prezentat pentru a interpola funcția $e^{-\frac{5}{x}}$ în intervalul [0 3.5].

3. Se verifică funcționarea programului de mai jos care realizează o interpolare bidimensională pentru funcția $f(x,y) = z = xe^{-(x^2+y^2)}$.

```
function laborator6();
     % fisierul trebuie sa se numeasca laborator6.m
[x,y] = meshgrid(-3:1:3);
     % se simuleaza punctele initiale ale functiei
z = x.*exp(-x.^2-y.^2);
     % se genereaza functia
subplot 221;
surf(x,y,z);
     % se reprezinta grafic functia
title('Neinterpolat');
[xi,yi] = meshgrid(-3:0.25:3);
     % se genereaza punctele de interpolare
zi1 = interp2(x,y,z,xi,yi,'nearest');
     % se realizeaza interpolarea
subplot 222;
surf(xi,yi,zi1);
     % se reprezinta grafic functia interpolata
title('Interpolare nearest');
zi1 = interp2(x,y,z,xi,yi,'bilinear');
subplot 223;
surf(xi,yi,zi1);
title('Interpolare bilinear');
zi1 = interp2(x,y,z,xi,yi,'bicubic');
subplot 224;
surf(xi,yi,zi1);
title('Interpolare bicubic');
```

4. Se verifică funcționarea programului de mai jos care realizează o interpolare bidimensională pentru funcția $\sin c(x,y) = z = \frac{\sin\left(\sqrt{x^2+y^2}\right)}{\sqrt{x^2+y^2}}$. Deoarece în domeniul de interpolare ales, [-8 8], apărea împărțirea prin zero, în program s-a adăugat constanta de valoare foarte mică *eps* la expresia funcției.

```
function laborator6();
     % fisierul trebuie sa se numeasca laborator6.m
[x,y] = meshgrid(-8:1.2:8);
```

```
     % se simuleaza punctele initiale ale functiei
R = sqrt(x.^2 + y.^2) + eps;
z = sin(R)./R;
     % se genereaza functia
subplot 221;
surf(x,y,z);
     % se reprezinta grafic functia
title('Neinterpolat');
[xi,yi] = meshgrid(-8:0.5:8);
     % se genereaza punctele de interpolare
zi1 = interp2(x,y,z,xi,yi,'nearest');
     % se realizeaza interpolarea
subplot 222;
surf(xi,yi,zi1);
     % se reprezinta grafic functia interpolata
title('Interpolare nearest');
zi1 = interp2(x,y,z,xi,yi,'bilinear');
subplot 223;
surf(xi,yi,zi1);
title('Interpolare bilinear');
zi1 = interp2(x,y,z,xi,yi,'bicubic');
subplot 224;
surf(xi,yi,zi1);
title('Interpolare bicubic');
```

6.5. Calcularea integralelor unidimensionale

6.5.1. Aspecte teoretice

MATLAB dispune de două funcţii care implementează algoritmul metodei trapezului cu control automat al pasului şi respectiv algoritmul metodei Simpson cu control automat al pasului.

Metoda trapezului este implementată prin funcţia *trapz* care are următoarea sintaxă:

z=trapz(x,y) unde:
- z – reprezintă valoarea integralei
- y=f(x) – reprezintă funcţia care trebuie integrată. Intervalul de integrare este determinat de valorile din vectorul x.

Metoda Simpson se implementează cu funcţia *quad*:

z=quad(fun,a,b,tol) unde:
- z – reprezintă valoarea integralei
- fun – este un pointer către o funcţie
- a,b – reprezintă limitele intervalului de integrare
- tol – reprezintă valoarea erorii absolute

Programul de mai jos calculează şi afişează integrala $\int_{-\frac{\pi}{2}}^{\frac{3\pi}{2}} \arctan x dx$ utilizând cele două metode prezentate.

```
function laborator6();
     % fisierul trebuie sa se numeasca laborator6.m
x = linspace(-pi/2,3*pi/2,300);
     % creaza punctele in care se calculeaza functia
y = atan(x);
```

```
    z = trapz(x,y);
    disp(sprintf('Integrala calculata cu metoda trapezului: %.8f',z));
         % se afiseaza valoarea integralei
    f = inline('atan(x)');
         % creaza pointer catre functie
    z = quad(f,-pi/2,3*pi/2,1e-6);
    disp(sprintf('Integrala calculata cu metoda Simpson: %.8f',z));
         % se afiseaza valoarea integralei
```

6.5.2. Desfăşurarea laboratorului

1. Se verifică algoritmul prezentat.
2. Se modifică algoritmul pentru a calcula integrala $\int_{-1}^{1} x^{10} dx$.

6.6. Rezolvarea ecuațiilor diferențiale ordinare

6.6.1. Aspecte teoretice

In MATLAB sunt implementate mai multe metode de rezolvare a unei ecuații diferențiale ordinare, atât numeric cât şi analitic. Dintre acestea vor fi prezentate in continuare funcția *ode45* care implementează un algoritm numeric Runge – Kutta de ordin 4 şi funcția *dsolve* care determină soluția analitică.

Sintaxa funcției *ode45* este:

[T,Y]=ode45(fun, [t0 tf], [y0]) unde:

fun – reprezintă un pointer către funcția care defineşte ecuația diferențială. Aceasta este de forma $y'(t) = \frac{dy}{dt} = fun(y,t)$.

[t0 tf] – reprezintă limitele intervalului de integrare. Pasul de integrare este stabilit automat de algoritm.

[y0] – reprezintă valoarea inițială la timpul t0 adică $y(t0) = y0$.

[T Y] – fiecare element din vectorul Y reprezintă valoarea derivatei la timpul corespunzător din vectorul T ale cărui elemente sunt cuprinse în intervalul [t0 tf].

O ecuație diferențială poate fi rezolvată analitic în MATLAB utilizând funcția *dsolve*. Dacă aceasta nu reuşeşte să identifice o soluție analitică va returna un mesaj de eroare. Sintaxa funcției este:

A=dsolve('Ds=expresie', 's(x0)=s0','x')

unde variabila independentă este *x* iar ecuația de rezolvat are forma $s'(x) = \frac{ds}{dx} = \text{expresie}(s,x)$ având valoarea inițială s0 în punctul x0.

Programul prezentat mai jos rezolvă ecuația $y' = 2 - e^{-4x} - 2y,\ \ y(0) = 1$ în intervalul [0 2.5] prin metoda numerică şi analitică iar apoi afişează grafic cele două soluții obținute.

```
    function laborator6();
         % fisierul trebuie sa se numeasca laborator6.m
    A=dsolve('Ds=2-exp(-4*x)-2*s','s(0)=1','x');
```

```
     % se rezolva analitic
f=inline('2-exp(-4*t)-2*y');
     % se defineste pointer-ul catre functie
[T,Y]=ode45(f,[0 2.5],[1]);
     % se rezolva numeric
plot(T,Y,'Color','red');
     % se reprezinta grafic solutia numerica
set(gca,'NextPlot','add','LineStyle','--');
     % sunt setate proprietati ale graficului
ezplot(A,[0 2.5]);
     % se reprezinta grafic solutia analitica
grid;
     % se activeaza caroiajul graficului
```

6.6.2. Desfăşurarea laboratorului

1. Se verifică algoritmul prezentat.

2. Se va modifica algoritmul pentru a rezolva ecuația $y' = \frac{3-4y}{2x}$, $y(1) = -4$ în intervalul [1 3.5].