Lucrarea 1

EROARI

SCOPUL LUCRĂRII

În prima parte a lucrării se prezintă conceptele fundamentale ale reprezentării numerelor reale, utile în toate problemele de metode numerice. Într-un fel sau altul tehnicile prezentate intervin în etapele de preluare şi prelucrare a datelor, chiar dacă nu în mod vizibil.

În cea de-a doua parte a lucrării sunt aduse în discuţie grafurile de procedură, ca metodă de calcul a marginii erorii relative pentru diferite expresii şi aplicarea acestei metode în câteva situaţii practice concrete.

1. PREZENTARE TEORETICĂ

1.1.ERORI ABSOLUTE ŞI ERORI RELATIVE

Deoarece valoarea măsurată sau calculată a unei mărimi nu reprezintă valoarea adevărată sau exactă a mărimii, o numim *valoare aproximativă*.

Definiţia 1.1. Numim eroare diferenţa dintre valoarea adevărată (exactă) şi valoarea aproximativă. Matematic definiţia poate fi exprimată prin formula:

$$e_x = x - \bar{x}\,, \tag{1.1}$$

unde:

e_x reprezintă eroarea,
x valoarea adevărată,
$\bar{x}$ valoarea aproximativă.

Deoarece unii autori definesc eroarea ca diferenţa:

$$e_x = \bar{x} - x\,, \tag{1.2}$$

pentru a reuni cele două definiţii ale erorii într-una singură s-a trecut la definirea *erorii absolute*, astfel:

$$e_x = \left|x - \bar{x}\right|$$

Definiţia 1.2. Numim *eroare relativă* raportul dintre eroarea absolută şi valoarea aproximativă absolută:

$$\varepsilon_x = |e_x/\bar{x}| \tag{1.3}$$

2. REPREZENTAREA GENERALĂ A NUMERELOR REALE

2.1.FORMA GENERALĂ A UNUI NUMĂR REAL. ENTITĂŢILE CONSTITUTIVE ALE ACESTUIA.

Un număr real se reprezintă sub forma următoare:

$$\texttt{nrReal = mantisa * baza}^{\texttt{exponent}} \tag{1.4}$$

Mantisa se mai numeşte şi *fracţie*, şi este un număr real.
Exponentul este un număr întreg, deci poate fi sau nu interpretat cu semn.

Există şi o reprezentare a numerelor reale de forma:

$$\texttt{nrReal} = \pm \texttt{d}_0\texttt{.d}_1\texttt{d}_2\texttt{d}_3\ldots\texttt{d}_{p-1} \texttt{ * b}^{\texttt{e}} \tag{1.5}$$

unde:

- *mantisa*, constă din poziţiile d_i, în număr de p poziţii;
- *baza* este notată prin b;
- *exponentul* este notat prin e.

Cele două notaţii sunt echivalente. Cea de-a doua prezintă în detaliu construcţia mantisei, pe când prima notaţie este mai directă, şi arată entităţile principale ce alcătuiesc un număr real.

2.2.CONVENŢII DE REPREZENTARE

O cifră în reprezentarea în bază 2 se numeşte *bit*, şi va fi notată în text prin litera **b** minusculă. Entitatea formată din 8 biţi consecutivi se numeşte *octet* (*byte* în limba engleză) şi va fi referită în text prin litera **B**, majusculă.

În reprezentările din memorie ale celor două entităţi (în speţă *mantisa* şi *exponentul*), fiecare poziţie binară va fi reprezentată folosind simbolurile:

- **F**, de la fracţie, pentru fiecare dintre biţii mantisei;
- **E**, de la exponent, pentru fiecare dintre biţii exponentului.
- **S**, de la semn, pentru bitul de semn.

3. DIMENSIUNILE TIPURILOR REALE FUNDAMENTALE

3.1.SPAŢIUL DE MEMORIE

Orice tip de variabilă are sens într-un limbaj de programare dacă există în memorie. Deci dacă are asociat un *spaţiu de reprezentare*. Spaţiul de reprezentare, din punct de vedere *binar*, este format din *biţi*. Care se grupează în *octeţi*.

Spaţiul minim adresabil este *octetul*. Lăţimile în biţi/octeţi pentru fiecare tip de dată fundamental C sunt stabilite prin standard (v. standardul ISO/IEC 9899:1999).

În discuţia de faţă este important cui atribuim / alocăm aceşti biţi, referindu-ne la mantisă, bază şi exponent.

Din punct de vedere hardware baza de reprezentare este 2 (*sistemul de numeraţie binar*). Celelalte baze de numeraţie uzual folosite în domeniul informatic sunt:

- octal (baza 8) ;
- hexazecimal (baza 16).

La ecran (şi deci nu în memorie) o afişare făcută pentru un utilizator obişnuit (utilizatorii diferiţi de cei specializaţi, precum programatori sau depanatori) va fi în sistemul de numeraţie zecimal (baza 10). Pentru a reuşi astfel de afişări trebuie parcurse etape de *conversie*, care aduc numărul binar în echivalentul său dorit (zecimal/octal/hexazecimal).

În concluzie, pentru baza de numeraţie nu se va rezerva spaţiu de memorie: aceasta este 2 şi rămâne neschimbată (nu se poate schimba baza de numeraţie pe parcursul calculelor, în funcţie de situaţii convenabile, pentru că dacă s-ar recurge la aşa ceva, caracterul unitar al prelucrărilor s-ar pierde, generându-se confuzii). În plus, nu există dispozitive hardware în logică multiplă (deci care să folosească diferite baze de numeraţie).

Rămân de analizat mantisa şi exponentul, pentru care evident există un număr de biţi specifici, în funcţie de tipul de dată real folosit.

3.2.ECHIVALENŢA ÎNTRE DENUMIRILE TIPURILOR REALE PREZENTE ÎN STANDARD ŞI CELE ADOPTATE DE LIMBAJELE C/C++

Standardul numeşte trei tipuri de date reale:

- *simplă precizie*;
- *dublă precizie*;
- *precizie extinsă*.

Limbajul C/C++ are un echivalent pentru fiecare. Aceste echivalenţe sunt, respectiv:

- `float`
- `double`
- `long double`

Spaţiul de memorie stabilit prin standardul ANSI C (`ISO/IEC 9899:1999`), diferenţiat pentru fiecare dintre entităţile mantisă şi exponent este următorul:

- `float` ⇒ `4B = 32b` repartizaţi astfel:
  - 23b pentru mantisă;
  - 8b exponentul, care pe poziţia MSb conţine semnul;
  - 1b pentru semnul mantisei, deci semnul numărului real ca atare.
- `double` ⇒ `8B = 64b`, repartizaţi astfel:
  - 52b pentru mantisă;
  - 11b pentru exponent;
  - 1b pentru semnul mantisei, deci semnul numărului real.
- `long double` ⇒ `10B = 80b` (denumit şi *ten bytes*)
  - 64b pentru mantisă;
  - 15b pentru exponent;
  - 1b pentru semnul mantisei, deci semnul numărului real.

Folosind notaţiile simbolice anunţate în secţiunea *Convenţii de reprezentare*, vom detalia aşezarea în memorie a celor 32 de biţi ai tipului `float`.

Astfel, pentru acest tip de dată, şi prin generalizare şi pentru celelalte două, în memorie există următoarea aşezare a biţilor:

$$\overset{31}{S} \mid \overset{30}{E}EEE\ EEE\overset{23}{E} \mid \overset{22}{F}FFFFFFFFFFFFFFFFFFF\overset{0}{F}$$

Am marcat spaţierea dintre fiecare grup de biţi printr-o bară verticală. Deasupra poziţiilor importante am marcat indexul (puterea bazei) asociat acelei poziţii. Plecând din dreapta, indexul biţilor creşte. Primul bit din dreapta are indexul zero, şi este denumit bitul cel mai puţin semnificativ (*Least Significant Bit*), iar primul bit din stânga cum privim aşezarea de ansamblul a biţilor, poartă denumirea de bit cel mai semnificativ (*Most Significant Bit*).

Bitul LSB are utilitate de exemplu în determinarea parităţii numerelor reprezentate în baza 2. Bitul MSB este convenţional asociat semnului, anume:

- 0, pentru numerele pozitive;
- 1, pentru numerele negative.

3.3.INTERPRETAREA GAMEI ASOCIATE FIECĂRUI TIP DE DATĂ REAL

La prima vedere, pentru tipurile reale `float` şi `double` se respectă dubla precizie: tipul `double` are de două ori mai mulţi biţi decât `float`. Privind mai atent însă, acest lucru este îndeplinit şi oarecum depăşit, în sensul următor: la tipul `double`, pentru mantisă, dubla precizie ar fi cerut un număr de 46b, dacă interpretarea se face în sens strict. Însă, fiind vorba de baza 2, se ştie că orice bit luat în considerare în reprezentare, conduce automat la dublarea gamei. Diferenţa de biţi între cele două cantităţi este de:

$52b - 23b = 29b$

Aceasta este considerabilă dacă avem în vedere afirmaţia că fiecare bit dublează gama. Cu 29b spaţiul câştigat în reprezentarea numerelor reale este:

$2^{29} = 2^9 * 2^{20} = 256$ Mega,

deci 256 milioane de combinaţii suplimentare.

Şi pentru exponent situaţia este mulţumitoare, pentru că nu avem strict 1b în plus (care ar duce efectiv la dublarea gamei), ci avem 3b suplimentari, aşadar o operaţie de dublare a gamei repetată de 3 ori.

3.4.SPECIFICATORII DE FORMAT

Pentru fiecare dintre tipurile de date reale, limbajul C rezervă următoarele combinaţii de litere:

- `float: %f`
- `double: %lf`
- `long double: %Lf`

Deşi sunt tipuri de date compatibile, recomandarea în practică este să se folosească specificatorii asociaţi fiecăruia dintre tipuri. În caz contrar, funcţiile de citire/scriere a datelor fiind instruite într-un anumit sens, datele de intrare sosesc în alt sens, şi de aici rezultatele calculelor pot fi neaşteptate, sau mai grav, nedeterminate.

4. VALORI LIMITĂ ÎN REPREZENTAREA NUMERELOR REALE

4.1. REPREZENTĂRILE PSEUDO-NUMERELOR: *zero*, *infinit* şi *NaN*

După cum am anunţat în secţiunea *Convenţii de reprezentare* vom folosi mai jos acele notaţii simbolice, pentru fiecare dintre poziţiile binare ale mantisei şi exponentului, în reprezentarea unor numere speciale pentru mulţimea **R**: zero, infinit şi NaN (*Not a Number*).

Sunt date mai jos reprezentările binare ale acestor situaţii speciale, cu observaţia că s-a presupus tipul real simplă precizie (`float` în C), pentru a oferi expunerii un caracter comprehensiv. În cele din urmă nu spaţiul este esenţial, ci modul de repartiţie al biţilor pentru fiecare entitate componentă a numărului real.

$$0=\begin{cases}\overset{31}{0}\ \ \overset{30}{0}000000\overset{23}{0}\ \ \overset{22}{0}000000000000000000000\overset{0}{0}\\ \overset{31}{1}\ \ \overset{30}{0}000000\overset{23}{0}\ \ \overset{22}{0}000000000000000000000\overset{0}{0}\end{cases}$$

$$\infty=\begin{cases}\overset{31}{0}\ \ \overset{30}{1}111111\overset{23}{1}\ \ \overset{22}{0}000000000000000000000\overset{0}{0}\\ \overset{31}{1}\ \ \overset{30}{1}111111\overset{23}{1}\ \ \overset{22}{0}000000000000000000000\overset{0}{0}\end{cases}$$

$$NaN=\begin{cases}\overset{31}{0}\ \ \overset{30}{1}111111\overset{23}{1}\ \ \overset{22}{0}000000000000000000000\overset{0}{1}\\ \qquad\qquad\qquad\qquad\qquad\ldots\ldots\ldots\ldots\ldots\\ \overset{31}{1}\ \ \overset{30}{1}111111\overset{23}{1}\ \ \overset{22}{1}111111111111111111111\overset{0}{1}\end{cases}$$

Se pot face câteva *observaţii*, pe baza reprezentărilor de mai sus:

- toate cele trei reprezentări sunt obligatoriu *duale* (pozitive şi negative);
- în valoare absolută, `NaN` îl depăşeşte pe $+\infty$ (cum am spune 'la dreapta' lui $+\infty$), iar `-NaN` este ca valoare sub cea a lui $-\infty$ ('la stânga' lui $-\infty$);
- când exponentul are toate poziţiile pe 1 binar, atunci indiferent de valoarea mantisei se obţine o reprezentare rezervată: fie *infinit*, fie ***`NaN`***; aceasta demonstrează că există o gamă întreagă pe care standardul o rezervă acestor reprezentări (datorată valorilor mantisei).

Relativ la ultima observaţie, merită să reprezentăm exponentul pentru cele două situaţii, şi să-l reprezentăm în baza 10. Valorile pe care le vom descoperi dau limitele superioară şi inferioară ale exponentului numerelor reale, şi sunt importante în găsirea valorilor limită inferioară şi superioară.

În paragraful următor detaliem acest lucru.

4.1.1. INTERPRETAREA EXPONENTULUI AVÂND TOATE POZIŢIILE PE VALOAREA 1 BINAR

Reprezentarea pe 8b cu toate valorile 1 binar poate avea o semnificaţie dublă în baza 10, în funcţie de cum considerăm util bitul de semn (MSb-ul reprezentării binare).

Fără semn, avem utile în reprezentare toate poziţiile, deci 8b. Aceasta conduce către valoarea zecimală `255`. Bitul LSb este pe 1, deci numărul binar este impar.

Cu semn, numărul de biţi utili este 7. Semnul este dat de MSb, şi cum el este 1 numărul este negativ. Pentru cele 7 poziţii, cum LSb este 1 rezultă că numărul obţinut trebuie să fie impar. Obţinem: `-127`.

Aceste valori trebuie reţinute, având în vedere discuţia de la reprezentarea normalizată. Acolo *polarizarea* se alege astfel încât să aducă în zero cel mai mic exponent negativ. Vedeţi secţiunea *Reprezentarea normalizată* pentru mai multe detalii.

4.2. GAMA REZERVATĂ PSEUDO-NUMERELOR

Se poate observa că atunci când exponentul are pentru toţi biţii valoarea 1, indiferent de valorile biţilor mantisei, compilatorul va trata acea reprezentare drept o excepţie: sau infinit, sau `NaN`.

Aceasta arată că, pentru cazul în care exponentul are toţi biţii 1, pentru tipul `float`, cu cele 23 de poziţii ale mantisei se obţin 2^{23} combinaţii care nu pot fi utilizate în program. Apare un 'decupaj' din gama totală pentru numerele reale, decupaj care se referă exclusiv la situaţii imposibil de folosit în practică.

Pornind de la această situaţie, gama de reprezentare rămasă nu prezintă dezavantaj în sensul limitării posibilităţilor de reprezentare.

4.3. EXEMPLE DE SITUAŢII PRACTICE ÎN CARE APARE FIECARE DINTRE REPREZENTĂRILE SPECIALE

Zero este un număr uzual. Putem spune că poate apărea în mod normal într-o situaţie practică, fără să constituie o situaţie excepţională. Poate fi folosit fără restricţii. Trebuie însă avute în vedere sunt situaţiile în care acest număr apare la numitorul unei fracţii. Dacă aceste cazuri nu sunt tratate corespunzător, rezultatul calculelor nu mai poate fi prezis cu acurateţe, dat fiind contextul variabil în care o asemenea situaţie poate apărea.

Infinit este deja o reprezentare ce nu poate fi folosită în mod normal într-un caz practic. Dacă am încerca să stabilim manual biţii unei variabile, astfel încât ei să corespundă celor prezentaţi, în momentul în care încercăm să

folosim în mod obişnuit acea variabilă, rezultatul va fi semnalat de către compilator.

NaN este o reprezentare pur convenţională a situaţiilor speciale. Este într-adevăr un pseudo-număr. De exemplu, situaţia unei nedeterminări de tip 0/0 este modelată în standard prin `NaN`. Sau alte situaţii similare: infinit la zero, zero la zero etc.

De exemplu, se poate forţa afişarea la ecran a acestei reprezentări din partea compilatorului, prin simularea unei limite care generează o nedeterminare:

$$\lim_{x\to 0}\left(\frac{\sin(x)}{x}\right) = NaN$$

sau prin calculul unei fracţii ai cărui numitor şi numărător sunt ambii stabiliţi la zero.

Pentru detalii consultaţi desfăşurarea lucrării, secţiunea *Probleme de laborator*, exerciţiul 6.

5. REPREZENTAREA NUMERELOR REALE ÎN *FORMA NORMALIZATĂ*

5.1. DENUMIREA DE NOTAŢIE ŞTIINŢIFICĂ

Numerele reale se pot reprezenta în două variante:

- normalizat;
- nenormalizat.

Utilitatea majoră o are *reprezentarea normalizată*. În această notaţie numerele foarte mari ca şi cele foarte mici nu-şi pierd semnificaţia şi valoarea, şi pot fi interpretate ca ordin de mărime. De exemplu, pentru a face distincţia dintre cele două moduri de reprezentare anunţate, să alegem numărul următor:

`0.0000000001`$_{10}$

După cum se poate uşor observa, acest număr este foarte mic, în speţă foarte apropiat de `0`. Prima cifră diferită de zero a mantisei apare pe poziţia a 10^{-a}. Aceasta înseamnă că, pentru a putea fi folosit ca o entitate diferită de zero, rezoluţia de reprezentare trebuie să fie de cel puţin 10 zecimale. Ca o consecinţă a acestui fapt, ar trebui ca şi aparatele folosite în măsurarea unei astfel de mărimi să permită rezoluţii foarte bune.

Cum în mod obişnuit aplicaţiile folosesc de obicei maximum 6 zecimale exacte, acest număr va fi considerat zero (urmare operaţiei de rotunjire, în care se păstrează doar zecimalele utile).

Aici intervine *reprezentarea normalizată*, în care numărul de mai sus nu-şi va pierde valoarea, putând fi interpretat ca un număr foarte mic, dar nu zero. Scrierea sa echivalentă folosind reprezentarea normalizată este:

$1.2 * 10^{-10}$

dacă considerăm că este reprezentat în *baza 10*. În acest moment, numărul nu mai este zero, ba chiar mai permite încă 5 poziţii zecimale, dacă am considera o reprezentare cu 6 zecimale exacte.

5.2. UTILITATEA REPREZENTĂRII NORMALIZATE

Avantajul reprezentării normalizate este acela de a permite comparaţia numerelor foarte mari sau foarte mici, fără a pierde din vedere poziţiile utile (cele diferite de zero) care pot apărea, pentru numerele foarte mari după multe poziţii de zero în dreapta cifrei utile, iar pentru numerele foarte mici după multe poziţii de zero în stânga cifrei utile.

Denumirea echivalentă a reprezentări normalizate în limbajul matematic obişnuit este *notaţie ştiinţifică*, sau *notaţie în puteri ale lui 10*. Prima denumire este cea uzuală.

Putem considera două exemple sugestive de numere, care fără apel la reprezentarea normalizată (notaţia ştiinţifică) nu ar avea sens în calcule.

Primul exemplu este *sarcina electronului*: $1.60217646 \times 10^{-19}$ coulombs. Este binecunoscută această reprezentare din problemele de fizică, şi nu numai.

Al doilea exemplu este *numărul lui Avogadro*: $NA = 6.0221415 * 10^{23}$. Acest număr este foarte utilizat în problemele de chimie.

După cum se observă, reprezentările au sens şi sunt natural interpretate prin intermediul notaţiei ştiinţifice. În concluzie, pentru o reprezentare nenormalizată, aceste numere ar fi, presupunând un număr de zecimale impus, fie zero (pentru sarcina electronului) fie o valoare limită superioară conform gamei de reprezentare aleasă, pentru numărul lui Avogadro.

5.3. REGULILE DE REPREZENTARE IMPUSE DE FORMA NORMALIZATĂ

Pentru a putea folosi reprezentarea normalizată, trebuie să respectăm următoarele reguli, referitoare la mantisă şi la exponent. Aceste reguli nu ţin de baza de reprezentare.

Regulile prezentate mai jos trebuie respectate în ordinea menţionată, deoarece schimbări ale valorilor mantisei influenţează exponentul, şi nu viceversa.

Aşadar:

1) *Regula pentru mantisă* - cere ca aceasta să fie situată în intervalul:

$[1, baza)$

2) *Regula pentru exponent* - cere ca exponentul să fie reprezentat fără semn **în memorie**. Pentru a putea fi reprezentat fără semn, o valoare de exponent întâlnită în mod obişnuit în calcule va trebui convertită într-una fără semn folosind ceea ce se numeşte *polarizare* (*bias*) în standardul IEEE754.

Polarizarea este un număr special ales pentru fiecare formă de reprezentare reală (simplă, dublă şi precizie extinsă) astfel încât să aducă în 0 valoarea cea mai mică cu semn a exponentului în reprezentarea din acea bază.

Valoarea *polarizării* se calculează cu relaţia:

$$bias = 2^{nrBitiExponent-1} - 1 \quad (1.6)$$

Rezultă următoarele valori, pentru fiecare tip real posibil:

- *simplă precizie* (`float`): $bias = 2^{8-1} - 1 = 127$
- *dublă precizie* (`double`): $bias = 2^{11-1} - 1 = 1023$
- *precizie extinsă* (`long double`): $bias = 2^{15-1} - 1 = 16383$

De exemplu, numărul 10 în baza 10 se scrie în reprezentarea normalizată drept:

$+1.0 * 10^{1}$

Observaţii:

1) Există două reprezentări distincte presupuse de standard, şi respectate şi în limbajul C:
   a. *reprezentarea internă* a unui număr, fiind vorba despre o reprezentare **în memorie**;
   b. *reprezentarea externă* a unui număr real, aici fiind vorba despre o reprezentare **la ecran**.

   Cele două reprezentări sunt echivalente, dar se fac în mod diferit.
2) Polarizarea este un *număr impar*.

Vom da în următoarele două paragrafe două exemple de calcul care sunt foarte utile în înţelegerea mecanismelor care stau la baza funcţionării unei reprezentări în virgulă mobilă:

- conversia unui număr real din baza 10 în baza 2;
- conversia unui număr real din baza 2 în baza 10;

5.3.1. REPREZENTAREA UNUI NUMĂR REAL BINAR ÎN FORMĂ NORMALIZATĂ PLECÂND DIN BAZA 10

Plecăm de la numărul zecimal: -300.65_{10}

Acesta prezintă şi parte întreagă şi zecimală pentru mantisă. Partea zecimală sau cea întreagă putea lipsi.

Trebuie să convertim cele două componente ale mantisei, succesiv.

Începem cu **conversia părţii întregi**.

După cum vă reamintiţi, conversia unui număr întreg din baza 10 în baza 2 se face prin împărţiri succesive la 2. Pentru fiecare pas se reţine restul împărţirii, iar noul deîmpărţit se consideră câtul determinat la pasul curent. Resturile vor fi în final scrise în ordine inversă, astfel încât primul rest găsit devine LSb al reprezentării binare a numărului. Dacă numărul de resturi nu este o putere a lui 2, se vor face completări cu poziţii de 0 înspre MSb pentru atâtea poziţii lipsă până se ajunge la un număr par, putere a lui 2 (4, 8, 16 etc).

Aşadar, pentru $300|_{10}$ avem succesiunea:

```
300      |2
rest 0   150|2
     rest 0   75|2
          rest 1   37|2
               rest 1   18|2
                    rest 0   9|2
                         rest 1   4|2
                              rest 0   2|2
                                   rest 0   1
```

După cum spuneam, primul rest devine LSb, deci este prima poziţie din dreapta, în reprezentarea binară către care ne îndreptăm. Ultimul rest devine primul bit din reprezentarea binară căutată. Avem:

```
1 0010 1100
```

deci o reprezentare pe 9 poziţii binare. În mod normal, la stânga primului bit trebuie să completăm cu 0 până găsim un număr de poziţii putere a lui 2. Pentru situaţia concretă prezentată, următorul număr de poziţii putere a lui 2, ce poate reprezenta dimensiunea unui tip de dată, este 16 (pentru `int`), deci ar trebui să completăm cu poziţii de 0, astfel:

```
0000 0001 0010 1100
```

unde biţii zero scrişi înclinat reprezintă adăugările noastre.

Aceasta este operaţia completă de conversie a unui număr întreg din bază 10 în bază 2.

Pentru situaţia de faţă însă **nu vom face completările cu 0**, pentru că aceasta generează biţi suplimentari, care nu rezultă din conversia efectivă, şi acest lucru are efecte nedorite în 'economia' biţilor pentru reprezentările reale (vă reamintiţi că fiecare număr real are un număr fix, impus, de poziţii binare alocate mantisei). Deci, pentru lucrarea de faţă reprezentarea binară considerată nu va conţine şi adăugările de poziţii `0`. Aşadar, reprezentarea binară finală pentru partea întreagă a mantisei este:

```
1 0010 1100 |2
```

Avem până acum 9 biţi din cei 23b ai mantisei (reamintim că exemplele considerate sunt pentru reprezentări simplă precizie, adică `float` în limbajul C).

Urmează **conversia părţii fracţionare**. Pentru aceasta reamintim că algoritmul de conversie este următorul: numărul fracţionar iniţial se înmulţeşte cu 2. Rezultă un nou număr fracţionar. Se reţine partea întreagă, iar pentru continuarea calculelor se preia partea fracţionară, care se înmulţeşte din nou cu 2. Rezultă un nou număr fracţionar, pentru care se reţine partea întreagă, iar partea fracţionară intră în operaţia de înmulţire cu 2 ş.a.m.d.

Avem, succesiv:

```
0.65 x 2 = 1.3   , reţin 1; continui cu 0.3
0.3 x 2 = 0.6    , reţin 0, continui cu 0.6
0.6 x 2 = 1.2    , reţin 1, continui cu 0.2
0.2 x 2 = 0.4    , reţin 0, continui cu 0.4
0.4 x 2 = 0.8    , reţin 0, continui cu 0.8
0.8 x 2 = 1.6    , reţin 1, continui cu 0.6
ş.a.m.d
```

Continuăm înmulţirile fie până când întâlnim o parte fracţionară zero, fie până numărul de poziţii reţinute ne este suficient în cazul concret de lucru.

Pentru situaţia de faţă, se observă că nu se poate ca în urma unei înmulţiri cu 2 partea fracţionară să ajungă `0`. Astfel, vom continua înmulţirile până contorizăm un număr de biţi dorit (valorile reţinute se observă că sunt doar de 0 şi de 1, deci pot fi considerate biţi). Cum partea întreagă a mantisei a avut nevoie de 9b, vom continua înmulţirile până când 'colecţionăm' 14b pentru partea fracţionară.

Grupând cele două reprezentări de până acum (în total 23b) şi separându-le prin punctul flotant, avem:

```
100101100.10100110011001
```

După cum s-a arătat în secţiunea 6.3, mantisa trebuie să aparţină intervalului `[1,2)` dacă dorim o reprezentare normalizată. Acest lucru este aici posibil doar dacă partea întreagă a mantisei este `1`. Pentru aceasta vom muta punctul mobil la stânga până imediat *în dreapta bitului MSb*. Noua reprezentare este:

```
1.00101100101001100110001
```

Am deplasat punctul mobil cu 8 poziţii la stânga, deci va trebui să înmulţim numărul cu 2^8, pentru ca nimic să nu se schimbe. Deci exponentul este pentru acest caz egal cu 8. Avem:

$1.00101100101001100110001 * 2^8$

Pentru că am aflat care este valoarea exponentului, şi ţinând cont că acum facem o reprezentare în memorie, va trebui *să polarizăm* exponentul. Ţinând cont că reprezentăm pe gama `float`, polarizarea este `127`. Plecând de la aceste observaţii, în memorie va fi stocată pentru exponent valoarea polarizată:

```
8 + 127 = 135
```

Căutăm acum reprezentarea binară a lui `135` şi aceasta va fi cea memorată.

Pentru `135` găsim următoarea reprezentare (pe baza algoritmului folosit anterior):

$$1000\ 0111|_2 = 87|_{16} = (8*161 + 7*160)|_{10} = 128 + 7|10 = 135|_{10}$$

Am marcat pentru fiecare pas baza de numeraţie în care s-au făcut reprezentările de fiecare dată. Exponentul este aici este impar şi se vede că LSb este 1 binar (în acest fel am verificat suplimentar rezultatul obţinut).

Numărul în bază 10 a fost anunţat ca `-300.65`, adică un număr negativ. Pentru aceasta va trebui ca MSb al reprezentării binare din memorie să fie `1`.

Avem toate datele pentru a putea da reprezentarea binară finală a numărului zecimal propus, pentru gama de reprezentare `float`. Aceasta este:

$$\overset{31}{1}\ \overset{30}{1}0000111\ \overset{23}{1}00101100101001100110\overset{0}{0}1$$

Ca şi în prezentarea teoretică am marcat şi indicii poziţiilor binare importante a le numărului.

Observaţii:

1) Fiind vorba despre o discuţie în *sistem binar*, regula pentru mantisă pentru reprezentarea normalizată impune acesteia să aparţină intervalului [1, **2**). Acest lucru este posibil dacă şi numai dacă partea întreagă este $1|_2$. Orice biţi am avea pentru partea fracţionară, chiar şi toţi de $1|_2$, tot nu se reuşeşte atingerea unităţii. Aceasta pentru că în partea dreaptă a punctului zecimal sunt puterile negative ale lui 2, deci o sumă de puteri negative ale lui 2, adică a unor numere *subunitare*, care nu poate atinge valoarea $1|_{10}$.

2) Pentru că se lucrează în formă normalizată, deci mantisa aparţine obligatoriu intervalului `[1, 2)` se poate ca primul bit din stânga al mantisei, cel de index 22, să nu mai fie memorat. Acesta se numeşte ***hidden bit*** şi este presupus implicit. Fără el mantisa nu ar mai aparţine intervalului indicat. Deci se poate câştiga 1b în reprezentarea numărului, ceea ce poate fi important atunci când avem un număr inexact (reprezentat pe un număr infinit de poziţii într-o anumită bază).

3) Ca un corolar al observaţiei 2), pot exista numere care să aibă o reprezentare exactă într-o bază de numeraţie şi inexactă într-o alta. Este şi cazul de faţă. Din această cauză, orice bit câştigat în reprezentarea poate fi important.

5.3.2. REPREZENTAREA UNUI NUMĂR REAL ZECIMAL ÎN FORMĂ NORMALIZATĂ PLECÂND DIN BAZA 2

Presupunem următoarea reprezentare binară ca punct de plecare în conversia reciprocă, din bază 2 în bază 10.

`1 10000111 00101100000000000000000` $|_2$

Dacă presupunem reprezentarea normalizată, înseamnă că avem deja presupus bitul ascuns. Prin urmare, ce există în memorie ca biţi ai mantisei reprezintă *partea fracţionară* a mantisei:

`00101100000000000000000`

ce va trebui prefixată cu valoarea acelui bit ascuns, pentru a genera reprezentarea completă a mantisei.

`1.00101100000000000000000`

5.4. ECHIVALENŢA ÎN LIMBAJUL *C* A REPREZENTĂRII NORMALIZATE

După cum ne-am propus, prezentarea se face în paralel între standard şi limbajul C. Spuneam că limbajul respectă standardul. Astfel, pentru acest moment al prezentării vom explica cum se poate forţa o afişare în notaţie normalizată (format ştiinţific) a valorilor reale folosite în calcule.

Pentru C, funcţiile de citire/afişare de la consolă (tandemul tastatură + ecran) permit şi interpretarea numerelor reale în reprezentare normalizată. Vom denumi această reprezentare *notaţie ştiinţifică*.

Aşadar, în limbaj există posibilitatea afişării datelor la ecran sub forma normalizată folosind unul dintre specificatorii de format **%e** sau **%E**. Diferenţa între aceştia este doar de formă, şi nu de fond. Pentru prima reprezentare, la ecran va apărea litera e minusculă, iar în cea de-a doua litera E majusculă.

Exemplu:

1) Pentru numărul `1234.567`, notaţia ştiinţifică se obţine în C astfel:
   ```
   float varR = 1234.567;
   printf("\n Notatia stiintifica: %e", varR);
   ```
   Rezultatul este: `1.234567e+3`
2) Pentru numărul `123456.789` se poate afişa formatul ştiinţific şi în varianta cu `%E`:
   ```
   float varR = 123456.789;
   printf("\n Notatia stiintifica: %E", varR);
   ```
   Rezultatul este: `1.23456789E+5`

Se poate vedea că rezultatele sunt similare, doar simbolul afişat este diferit.

6. CIFRE SEMNIFICATIVE

6.1. DEFINIŢIA UNEI CIFRE SEMNIFICATIVE

Acea cifră care nu poate lipsi (nu poate fi ignorată) din reprezentarea unui număr real poartă numele de *cifră semnificativă*. Problema este mai delicată mai ales cu cifrele aflate pe ultimele poziţii, privind de la stânga la dreapta. Şi în mod special pentru cifrele de zero situate în aceste poziţii extreme.

6.2. CIFRELE *DE ZERO* SEMNIFICATIVE ÎN REPREZENTARE NORMALIZATĂ ŞI NENORMALIZATĂ

Dacă numărul este în reprezentare ne-normalizată, deci digitul d_0 este zero, atunci se poate spune că acea poziţie poate lipsi, pentru că există o reprezentare în care acea cifră nu mai apare - anume *reprezentarea normalizată* - v. secţiunea *Reprezentarea numerelor în forma normalizată*.

În mod normal, se poate ignora o cifră doar dacă este într-una dintre extremităţile numărului. Şi trebuie ţinut obligatoriu seama de regulile implicate în această decizie.

6.2.1. CIFRA SITUATĂ LA ÎNCEPUTUL NUMĂRULUI

Astfel, în reprezentarea nenormalizată, mantisa începe sigur cu o poziţie zero, în stânga punctului zecimal. Aşa cum am spus în paragraful anterior, în această situaţie acea cifră poate fi ignorată, pentru că există o reprezentare echivalentă ce nu cuprinde şi acea poziţie. Informaţia pe care acea cifră o oferă este asupra modului în care trebuie făcută translaţia punctului zecimal, deci se reflectă în valoarea exponentului. Este jocul care trebuie avut în vedere de fiecare dată: translaţia punctului zecimal influenţează mărimea numărului real, deci are influenţă asupra exponentului.

6.2.2. CIFRA SITUATĂ LA SFÂRŞITUL NUMĂRULUI

O cifră *zero* situată pe prima poziţie din dreapta numărului se poate ignora, pentru că nu are vecin în dreapta sa.

Această operaţie însă nu se face fără efecte. Ignorarea unei cifre în situaţia aceasta are două consecinţe:

- pierderea în reprezentare a unui ordin de mărime (discutăm acest aspect în contextul bazelor de numeraţie *poziţionale*);
- ignorarea eventualelor operaţii anterioare, care au determinat forma curentă a numărului, şi care este posibil să fi presupus deja operaţii de trunchiere/rotunjire. Dacă se alege înlăturarea ultimei cifre se ignoră în

mod direct aceste operaţii. În cele din urmă acele presupuse operaţii se pot repeta, dar de această dată pentru un număr mai redus de zecimale, pentru că prin îndepărtarea cifrei finale, trebuie avute în vedere regulile tandem pentru trunchiere/rotunjire - v. secţiunea *Rotunjiri obişnuite*.

Aşadar, în situaţiile practice în care o astfel de situaţie apare, nu trebuie să înlăturăm cifra situată pe ultima poziţie (în citirea numărului de la stânga la dreapta, ultima poziţie este prima din dreapta).

De exemplu, să considerăm situaţia numărului următor:

`2.560`

pentru care dorim înlăturarea ultimei cifre, acel `0` care nu mai are vecin în dreapta sa.

Care ar fi consecinţele?

Numărul acesta este preluat ca atare de către noi, fără a-i cunoaşte istoria. Acest număr ar fi putut fi în prealabil supus unor operaţii de rotunjire/trunchiere, în urma cărora a căpătat valoarea de faţă. Astfel, altcineva ar fi putut deja aplica rotunjirea la 3 zecimale, pornind însă de la numărul:

$3.5597|_{10}$

Conform regulilor de rotunjire/trunchiere, dacă dorim 3 zecimale exacte pentru numărul `2.5597` obţinem chiar `2.560`. În acest caz, ignorarea din partea noastră a cifrei `0` din final ar da peste cap o operaţie anterioară, pentru care putem presupune că s-a avut în vedere un motiv oarecare.

De asemenea trebuie reţinut din exemplul prezentat că operaţia de trunchiere are rol important în reprezentarea datelor. Vom vedea mai concret acest lucru când discutăm despre rotunjirile obişnuite - v. secţiunea *Rotunjiri obişnuite*.

7. ROTUNJIRI OBIŞNUITE

Se fac prin comparaţia faţă de `5` a cifrei ce urmează să fie înlăturată. În această operaţie apar şi câteva situaţii la limită, în care foarte importantă este cantitatea ce trebuie trunchiată.

Primul pas într-o operaţie de rotunjire este *trunchierea*. Astfel, dacă numărul de zecimale din reprezentarea dată este mai mare decât cel avut în vedere în rotunjire, atunci toate cifrele din acea reprezentare începând cu cifra din dreapta celei care va trebui comparată cu `5` vor fi ignorate. Această cifră ce urmează să fie comparată cu 5 este absolut necesară rotunjirii. Odată comparaţia încheiată această cifră va fi la rândul său ignorată.

Tocmai această trunchiere trebuie avută în vedere în rotunjire. Nu putem ignora pur şi simplu zecimalele suplimentare. Trunchierea are o influenţă concretă şi foarte importantă asupra valorii finale a numărului de rotunjit.

Pentru a aplica rotunjirea vom studia câteva exemple concrete, care acoperă toate situaţiile reale.

Numerele pe care dorim să le rotunjim sunt:

`4.3244`
`4.3236`
`4.3275`
`4.3285`

Ultimele două constituie cazuri limită. Vom vedea imediat despre ce este vorba.

Am marcat în **aldin** cifra ce urmează să fie comparată cu `5`. Orice alte zecimale situate în dreapta sa vor fi trunchiate. Putem avea o parte trunchiată diferită sau egală cu `0`. Pentru al doilea caz rotunjirea ţine cont de o *convenţie*.

Să începem cu primul număr. Trunchiem aşadar toate zecimalele situate în dreapta cifrei marcată în aldin (îngroşat). Iar această cifră o comparăm cu 5. Fiind mai mică, avem de-a face cu o *rotunjire prin lipsă*, aşa cum ne va arăta şi eroarea de rotunjire. In urma acestui tip de rotunjire lăsăm neschimbată valoarea digitului din stânga celui cu care facem comparaţia cu 5. De fapt înlăturăm o anumită valoare astfel încât ultima cifră a numărului să devină `0`. Deoarece dorim evident rotunjirea vom ignora fără probleme acea cifră (vedeţi şi paragraful **6 - *Cifre semnificative*** pentru mai multe detalii privind renunţarea la o cifră `0` situată pe ultima poziţie a unui număr real).

```
4.3244 -> 4.324, err = |nrRotunjit - nrInitial| = |0.0004|-
```

Am pus un semn minus în dreapta rezultatului calculat în valoare absolută pentru a reţine aspectul menţionat: avem de-a face cu o rotunjire *prin lipsă*. Valoarea înlăturată este dată de diferenţa de mai sus.

Pentru al doilea număr. Trunchiem aşadar toate zecimalele situate în dreapta cifrei marcată în aldin. Comparăm cifra rămasă cu 5. În acest caz este mai mare decât 5 şi vom aplica *rotunjirea prin adaos*. Valoarea digitului anterior celui intrat în operaţia de comparare cu 5 va creşte cu o unitate. Aceasta provine din faptul ca de fapt adăugăm o anumită valoare astfel încât ultima cifră a numărului să devină `0`. Cum ceea ce dorim este rotunjirea vom putea ignora fără probleme acea cifră (vedeţi şi paragraful **6 - *Cifre semnificative*** pentru mai multe detalii privind renunţarea la o cifră `0` situată pe ultima poziţie a unui număr real).

```
4.3236 -> 4.324, err = |nrRotunjit - nrInitial| = |0.0004|+
```

Am pus un semn plus în dreapta rezultatului calculat în valoare absolută pentru a reţine aspectul menţionat: avem de-a face cu o rotunjire *prin adaos*. Valoarea adăugată este dată de diferenţa de mai sus.

Pentru ultimele două situaţii există o nuanţă suplimentară ce trebuie avută în vedere: valoarea părţii trunchiate. Anunţam acest lucru mai sus.

Aşadar, sunt două situaţii: valoare diferită de zero SAU valoare nulă.

Pentru *prima situaţie*, indiferent de paritatea cifrei anterioare celei cu care facem comparaţia, rotunjirea se face prin adaos.

În schimb, dacă partea trunchiată este zero, atunci ţinem cont de convenţia care spune că dacă paritatea cifrei anterioare celei care ajută în comparaţia cu 5 este impară rotunjirea se va face prin adaos. Iar în cazul unei parităţi pare rotunjirea se face prin lipsă.

Considerând parte trunchiată diferită de zero rezultatele finale ale rotunjirii sunt:

```
4.3275 -> 4.328, err = |0.0005|+
4.3285 -> 4.329, err=|0.0005|+
```

Această convenţie mai poartă numele şi de *convenţia digitului impar,* marcând prin această denumire situaţia in care se aplică rotunjirea prin adaos.

Observaţii:

1) Aplicarea convenţiei digitului impar are fundamentare în experimentele de laborator. Justificarea sa este legată de eroarea de rotunjire pe ansamblu. Anume, dacă observăm că printre simbolurile reprezentării zecimale avem atâtea cifre pare câte impare aceasta înseamnă că per ansamblu, dacă ţinem cont de această convenţie, eroarea calculelor ce implică rotunjirile se compensează şi tinde la 0. Acest lucru avantajează calculele intense. Pentru acele situaţii nu avem studiată o statistică (sau o distribuţie oarecare) prin care să putem justifica probabilitatea de apariţie a simbolurilor impare în defavoarea celor pare sau reciproc. Doar eventualele erori de calibrare ale aparatelor de măsură folosite pot determina deviaţii de această natură.
2) Pentru primele două exemple de rotunjire se observă că rezultatul final este acelaşi, deşi eroarea este diferită. Aşadar nu putem spune dinainte ce fel de rotunjire s-a aplicat dacă avem un număr dat.
3) Compilatorul de C nu respectă *convenţia digitului impar*. Pentru cazurile limită amintite mai sus acesta va ţine cont de paritatea-imparitatea digitului anterior cifrei comparate cu 5, doar ca tehnica utilizata pare oarecum complementara conventiei prezentate, in sensul ca va rotunji *prin adaos* daca digitul este par si *prin lipsa* pentru digit impar. Se poate verifica acest lucru rulând următorul scurt program:

```
/*Rotunjirea obisnuita - cazuri limita*/
#include<stdio.h>

int main()
{
 float nr1=4.3275, nr2=4.3285;

/*impun reprezentare cu 3 zecimale*/
 printf("Rotunjire la limita -> nr1: %.3f, nr2:%.3f\n",nr1,nr2);

 scanf("%f", &nr1);
 return 0;
}
```

4) Numărul implicit de zecimale folosit în reprezentările numerelor reale în ANSI C este `6`. Puteţi observa acest lucru dacă se renunţă la limitările de reprezentare din programul anterior (construcţiile `%A.Bf`)
5) Exemplele alese anterior pentru situaţiile limită în contextul operaţiei de rotunjire, justifică faptul că *eroarea maximă de rotunjire* este $5*10^{-t}$, unde `t` reprezintă numărul de zecimale pe care-l are un anumit număr *ulterior trunchierii* şi *anterior operaţiei de rotunjire*. Avem aici şi legătura cu paragraful următor, în care sunt studiate *grafurile de procedură*. Acolo, operaţia de majorare în calculul erorilor expresiilor este determinată unilateral chiar de valoarea $5*10^{-t}$.

8. GRAFURI DE PROCEDURĂ

Aceste grafuri sunt *binare*, adică într-un nod intră cel mult două mărimi şi iese o singură mărime. Pentru fiecare operaţie aritmetică marca grafului se calculează diferit. Marca grafului pentru operaţia de adunare este raportul între valoarea din nodul care se însumează şi suma valorilor din nodurile care se însumează, procedeul fiind prezentat prin graful din fig.1.1 - ***Operaţia de adunare***.

Se dau valorile erorilor relative ε_x şi ε_y ale mărimilor care se însumează şi eroarea de rotunjire ε_r .

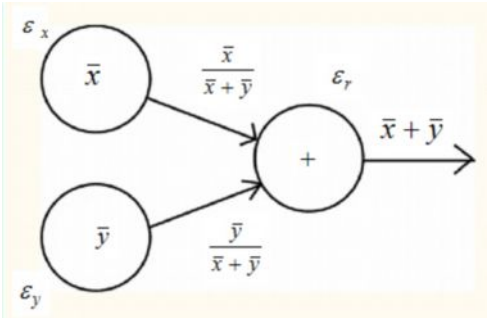

Fig.1.1. - Graful operaţiei de adunare

Eroarea la ieşirea grafului este:

$$\varepsilon_{x+y} = \frac{\bar{x}}{\bar{x}+\bar{y}}\varepsilon_x + \frac{\bar{y}}{\bar{x}+\bar{y}}\varepsilon_y + \varepsilon_r \qquad (1.7)$$

Eroarea relativă propagată prin orice operaţie, se calculează ca suma produselor erorilor relative ale valorilor care intră în operaţie cu marca grafurilor corespunzătoare, la care se mai însumează eroarea de rotunjire

datorată operaţiei de adunare. Acest mod de calcul trebuie efectuat pe întregul graf până la ieşire, obţinând în final eroarea relativă a grafului. Grafurile pot fi mai simple sau mai complexe, funcţie de expresia pe care o implementează. În fiecare nod al grafului se realizează o operaţie aritmetică ce are doi termeni sau facto operaţie aritmetică care are doi termeni sau factori. Eroarea în nodul grafului se calculează după modul expus anterior şi nodul poate deveni la rândul său un termen sau un factor al următoarei operaţii.

În calculul erorii se ţine seama de valoarea obţinută în fiecare nod prin operaţia corespunzătoare expresiei a cărei valoare a erorii relative o calculăm, pentru a determina marca ramurii grafului. Pentru operaţiile de înmulţire, împărţire şi ridicare la putere marca grafului este constantă indiferent de valoarea obţinută în nodul grafului.

Marca grafului pentru descăzut la ***operaţia de scădere*** se calculează ca raportul între valoarea descăzutului şi diferenţa valorilor descăzutului şi scăzătorului, iar pentru scăzător se calculează ca raportul dintre valoarea scăzătorului şi diferenţa valorilor descăzutului şi scăzătorului, luat cu semnul minus. Eroarea la ieşirea grafului este dată în formula (1.8), iar graful este prezentat în fig. 1.2.

$$\varepsilon_{\text{x-y}} = \frac{\bar{x}}{\bar{x}-\bar{y}}\varepsilon_x - \frac{\bar{y}}{\bar{x}-\bar{y}}\varepsilon_y + \varepsilon_r \tag{1.8}$$

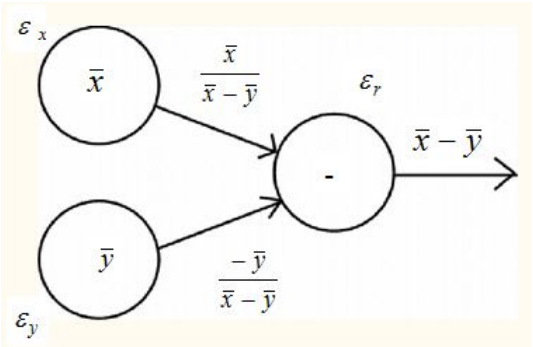

Fig.1.2. - Graful operaţiei de scădere

Marca grafului pentru operaţia de înmulţire este totdeauna constantă egală cu unitatea atât pentru înmulţitor cât şi pentru deînmulţit. Graful pentru această operaţie este prezentat în figura 1.3. Eroarea la ieşirea grafului este dată de expresia următoare:

$$\varepsilon_{\text{xy}} = \varepsilon_{\text{x}} + \varepsilon_{\text{y}} + \varepsilon_{\text{r}} \tag{1.9}$$

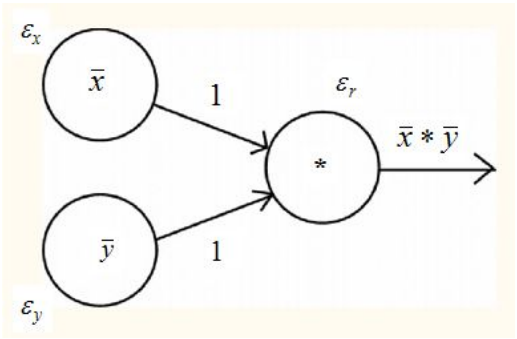

Fig.1.3. - Graful operaţiei de înmulţire

Marca grafului pentru operaţia de ***împărţire*** este totdeauna constantă, având valoarea *1* pentru deîmpărţit şi *-1* pentru împărţitor. În figura 1.4 este prezentat graful pentru operaţia aritmetică de împărţire. Eroarea la ieşirea grafului este:

$$\varepsilon_{x/y} = \varepsilon_x - \varepsilon_y + \varepsilon_r \tag{1.10}$$

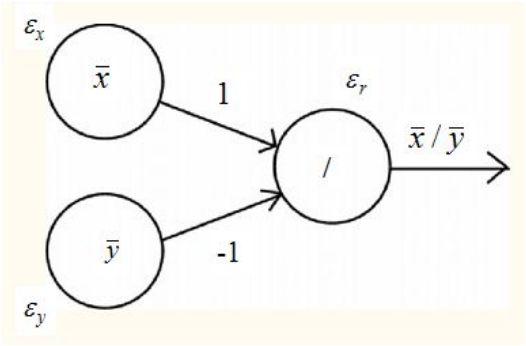

Fig.1.4. - Graful operaţiei de înmulţire

Operaţia de extragere a rădăcinii pătrate are marca grafului constantă egală cu ½ şi este prezentată în figura 1.5.

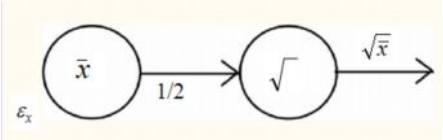

Fig.1.5. - Graful operaţiei de extragere de rădăcină pătrată

Eroarea mărimii la ieşirea grafului este:

$$\varepsilon_{\sqrt{x}} = (1/2)\varepsilon_x + \varepsilon_r \tag{1.11}$$

Pentru uşurinţa calculului mărcilor grafului se ţine seama de modul de calcul prezentat anterior pentru fiecare operaţie aritmetică. Cunoscând erorile relative ale mărimilor care intră într-o expresie în care avem operaţiile de bază ale aritmeticii, putem să determinăm eroarea finală a expresiei cu ajutorul grafurilor de procedură.

9. CONCLUZII

1. Utilizarea în programe a unui număr de zecimale trebuie să fie dictată prioritar de modul implicit de reprezentare din C (`6` zecimale exacte);
2. C nu este un limbaj ştiinţific (ci unul de uz general) şi din acest motiv nu ţine cont de *convenţia digitului impar*. Dar totuşi se comportă diferit în cele două situaţii limită, cea în care digitul din stânga celui cu care facem comparaţia este par sau impar. Tehnica folosită de acesta pare oarecum opusă celei prezentate şi s-ar putea numi *convenţia digitului par* (dacă avem digit par atunci rotunjirea se face prin adaos);
3. Tipul `float` ar trebui să fie suficient pentru toate programele acestui laborator. Tipul `double` este tipul exclusiv al funcţiilor matematice de bibliotecă (biblioteca matematică C este construită exclusiv pe `double`). Orice argument `float` al acestor funcţii va fi automat promovat la `double` de către compilator. In sfârşit, tipul de dată `long double` este utilizat în coprocesoarele matematice. Ştiind dimensiunea acestui tip de date înţelegem performanţa obligatorie pe care coprocesoarele trebuie să o ofere.
4. Există o echivalenţă strânsă între impunerile standardului şi felul în care ANSI C implementează aceste cerinţe;
5. Reprezentările din program trebuie să aibă în vedere situaţia practică din acel moment. Astfel, trebuie folosiţi specificatorii de format tipici fiecărei situaţii;
6. Nu trebuie ignorată o cifră de zero situată ultima într-o reprezentare reală oarecare. Mai general, cifra finală din reprezentarea unui număr nu se va ignora;
7. Utilizarea doar a specificatorilor de format asociaţi fiecărui tip de date (conform cerinţelor de limbaj) şi evitarea interpretării acestor tipuri de către compilator (chiar şi pentru tipurile compatibile, cum sunt cele reale de exemplu, folosirea în tandem a specificatorilor diferiţi generează erori de citire/interpretare/afişare a datelor).

10. DESFĂŞURAREA LUCRĂRII

10.1 . ÎNTREBĂRI

1. Unde sunt utile grafurile de procedură?

2. Cum se modelează o operaţie oarecare prin grafurile de procedură (de exemplu operaţia de adunare)? Dar o operaţie compusă?
3. De ce în grafurile de procedură nu este necesar să cunoaştem ordinul de mărime al erorilor operanzilor?
4. Deduceţi ponderile (valorile asociate arcelor) pentru operaţia de scădere.
5. Care este numărul de biţi rezervaţi exponentului pentru reprezentarea reală în simplă precizie (`float` în C)?
6. Care sunt cele două condiţii impuse reprezentării normalizate?
7. Explicaţi ce semnificaţie are polarizarea în reprezentarea normalizată a numerelor reale.
8. Rotunjiţi obişnuit numerele:
   a. 0.30510
   b. 0.36500

   presupunând că se cer: două şi apoi trei zecimale exacte, pentru fiecare dintre numere.
9. Aduceţi la forma normalizată numerele:
   a. 809.31
   b. 0.05925×10^{2}
   c. 72.090×10^{-2}
   d. 0.6115×10^{100}
10. De unde rezultă şi ce semnificaţie are cantitatea 5×10^{-t} ?

10.2. PROBLEME DE LABORATOR

1. Să se determine o limită superioară a erorii relative pentru expresia:
$$E(x) = (ax^2 + bx + c)/(dx + f)$$
în punctul x_0 ştiind că a, b, c, d, f, x_0 sunt pozitive şi au respectiv erorile relative ε_a , ε_b , ε_c , ε_d , ε_f , ε_{x_0} .

2. Fie a un număr pozitiv rotunjit în mod obişnuit, iar 3 nu are eroare. Se dau expresiile:
$$u=a+a+a \text{ , şi}$$
$$v=3a$$
Să se arate, cu ajutorul grafurilor de procedură, că marginea erorii relative a lui u este mai mare ca marginea erorii relative a lui v.

3. Se consideră expresiile $u = a^2 + 2a + 1$ şi $v = (a+1)^2$ unde a este un număr pozitiv rotunjit în mod obişnuit şi presupunem că 1 şi 2 nu au erori. Să se arate că pentru a foarte mare marginile erorii relative a lui u şi v devin aproximativ egale iar pentru a foarte mic marginea erorii relative a lui u devine aproximativ de trei ori mai mică decât marginea

erorii relative a lui v. Numărul maxim de cifre al mantisei admise de calculator este t. Să se calculeze marginile erorii relative ale expresiilor date şi pentru cazul când a este exact.

4. Fie a,b şi x trei numere pozitive şi exacte. Să se traseze grafurile de procedură şi să se calculeze marginile erorilor relative de rotunjire pentru expresiile:
$$u = ax + bx^2 \text{, şi}$$
$$v = x(a + bx)\,.$$
Să se compare cele două margini ale erorilor, obţinute. Să se calculeze marginile erorilor relative şi pentru cazul când cele trei numere au erori de rotunjire.

5. Se consideră un circuit R, L, C alimentat de la o sursă U. Ştiind că frecvenţa sursei are variaţia ε_f, iar R, L, C au variaţiile $\varepsilon_R, \varepsilon_L, \varepsilon_c$, să se traseze graful impedanţei circuitului şi să se determine o margine a erorii relative a ei.

6. *Exemplificări ale situaţiilor speciale pentru numerele reale*
Ca exemplu s-a construit sursa următoare, care cuprinde exemple practice pentru toate situaţiile în care apar numerele speciale (*pseudo-numere*) la care ne-am referit.

```
// Exemple de situatii in care apar numerele reale speciale
#include<iostream>
#include<math.h>
using namespace std;

int main()
{
   union sitSpeciala {long int nrI; float nrR;} situatie;
   float numerator = 0;
   float denominator = 0;
   float underSqrt = -1;

// zero
   situatie.nrI = 0x00000000;       // zero cu plus
   cout << hex << "\n Calcul eronat... zero 1... " << situatie.nrR;

   situatie.nrI = 0x80000000;       // zero cu minus
   cout << "\n Calcul eronat... zero 2... " << situatie.nrR;

// infinit - cu plus si cu minus
   cout << "\n Calcul eronat... infinit 1... " << 1/denominator;
   cout << "\n Calcul eronat... infinit 2... " << 1/-denominator;

// NaN - calcul eronat - nedeterminari
   cout << "\n Calcul eronat... nedeterminare 1... " <<
```

```
                                numerator/denominator;
    cout << "\n Calcul eronat... nedeterminare 2... " <<
                                sqrt(underSqrt);

// final
    cin >> numerator;
    return 0;
}
```

7. *Folosirea simultană în program a tipurilor reale şi întregi*

Vom alege un scurt program, ce presupune utilizarea ambelor tipuri de date (reale şi întregi) pentru a rezolva problema propusă.

Orice programator va trebui să folosească cu atenţie tipurile mixte, pentru că este vorba despre spaţii de memorie în principiu diferite, dar mai ales de două mulţimi de numere, în speţă **Z** şi **R**, care nu au o echivalenţă imediată între valorile lor.

Va trebui utilizat şi mecanismul *conversiilor explicite*, pentru a garanta păstrarea informaţiei, şi a evita pierderea acesteia - datorată de exemplu micşorării spaţiului de memorie prin trecerea de la real la întreg.

Pentru programul următor, compilatorul avertizează programatorul asupra unor conversii între tipuri de date incompatibile. Aceste avertismente se referă la situaţiile în care nu apare operatorul de conversie explicită. Pentru celelalte situaţii totul este în regulă, pentru că odată ce întâlneşte conversia explicită compilatorul presupune că programatorul ştie ce face. Avertismentele ţin şi de modul în care este configurat compilatorul, având în vedere că se pot impune opţiuni în linia de comandă în momentul lansării sale în execuţie.

```
// Tipuri mixte de date
#include<iostream>
using namespace std;

int main()
{
    int nr ;            // implicit cu semn
    unsigned int nr1;   // fara semn
    float rez = -2.1;

// folosirea mixta a tipurilor de date - fara conversie implicita
// Numarul intreg este cu semn
    nr = rez;
    cout << "\n Fara conversie implicita -> numarul intreg este "
                                                    << nr;

// folosirea mixta a tipurilor de date - cu conversie implicita
// Numarul intreg este cu semn
    nr = (int)rez;
    cout << "\n Cu conversie implicita -> numarul intreg este "
                                                    << nr;
```

```
// =============
// folosirea mixta a tipurilor de date - cu conversie implicita
// Numarul intreg este fara semn
   nr1 = rez;
   cout << "\n Fara conversie implicita -> numarul intreg este "
                                                    << nr1;

// folosirea mixta a tipurilor de date - cu conversie implicita
// Numarul intreg este fara semn
   nr1 = (int)rez;
   cout << "\n Cu conversie implicita -> numarul intreg este "
                                                    << nr1;
// final program
   cin >> nr;
   return 0;
}
```

Compilatorul va avertiza programatorul prin mesajele:

```
Line 14 [Warning] assignment to `int' from `float'
Line 25 [Warning] assignment to `unsigned int' from `float'
```

Este vorba despre liniile 14 şi 25 din program, în care nu apar operatorii de conversie explicită, şi se impune unui număr real să fie salvat într-un număr întreg. Astfel de situaţii pot uşor să fie trecute cu vederea, ceea ce are consecinţe importante în momentul rulării programului.

Pentru situaţia în care numărului real i se impune să fie reţinut în spaţiul asociat unui număr întreg scurt (2B), fără a se face apel la conversia explicită, s-ar putea da următoarea interpretare operaţiei: se pleacă de la bitul 0, se numără 16b (adică 2B) şi ceea ce rezultă este salvat în numărul întreg. Însă nu este vorba despre o trunchiere fizică a biţilor mantisei! Rezultatul oferit de compilator în absenţa conversiei explicite este într-adevăr eronat, dar pentru situaţia în care numărul întreg este fără semn. Pentru celălalt caz, deşi rezultatele sunt similare şi cu şi fără operatorul de conversie explicită, este indicat ca de fiecare dată să se folosească operaţia de conversie, pentru că nu se ştie ce generaţie de compilator este folosit.

10.3. TEME DE CASĂ

1. Presupunem că x şi y sunt două numere pozitive rotunjite simetric. Să se traseze grafurile de procedură ale expresiilor $u = (x + x + x + x) \cdot y$ şi $v = 4xy$ şi să se arate că marginea erorii relative a lui u este mai mare decât marginea erorii relative a lui v.

2. Se consideră expresiile $u = a^3 + 3a^2 + 3a + 1$ şi $v = (a+1)3$ unde a este un număr pozitiv rotunjit în mod obişnuit şi presupunem că 1 şi 2 nu au erori. Să se calculeze marginile erorii relative a lui u şi v pentru a foarte mic şi pentru a foarte mare (infinit). Comparaţi marginile

erorilor relative. Numărul de cifre semnificative a lui *a* este *t*. Toate erorile care intervin în calcul sunt mai mici decât eroarea relativa de rotunjire simetrică.

3. Se consideră circuitul din figura 1.6 in care $R_1 = 4.7k\Omega$ şi toleranţa $\varepsilon_1 = 10\%$, $R_2 = 6.8k\Omega$ şi toleranţa $\varepsilon_2 = 5\%$, $R_3 = 7{,}5k\Omega$ cu toleranţa $\varepsilon_3 = 10\%$ şi R4 = 10k cu toleranţa = 10%. Să se calculeze erorile relative a rezistenţelor. Să se deducă formula de calcul a rezistenţei echivalente a circuitului între punctele A şi B. Scrieţi toate valorile în virgulă mobilă şi considerând că toate erorile sunt mai mici decât eroarea relativă a rotunjirii simetrice determinaţi eroarea relativă maximă a rezistenţei echivalente. Consideraţi numărul de cifre semnificative a valorilor *t* = 2.

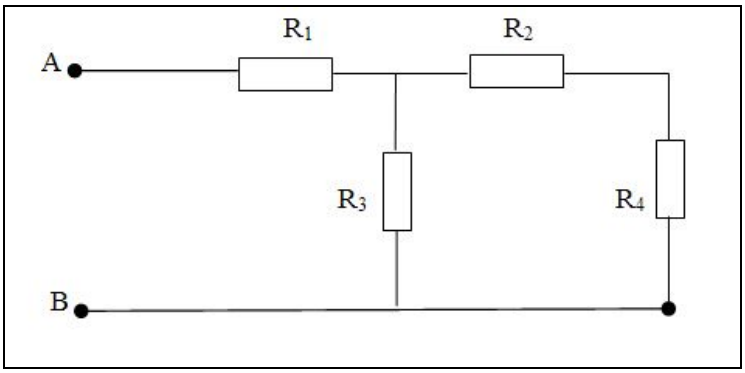

Fig. 1.6 - Circuitul pentru problema 8

4. Se consideră amplificatorul operaţional din figura 1.7. Considerând rezistenţele $R_1 = 100k\Omega$ cu toleranţa de $\varepsilon_1 = 10\%$ şi $R_2 = 10M\Omega$ cu toleranţa de $\varepsilon_2 = 10\%$ să se determine eroarea maximă a amplificării. Se calculează erorile relative a rezistenţelor, se fac calculele în virgulă mobilă si considerăm numărul de cifre semnificative `t = 2`.

$$A = -\frac{R_2}{R_1}$$

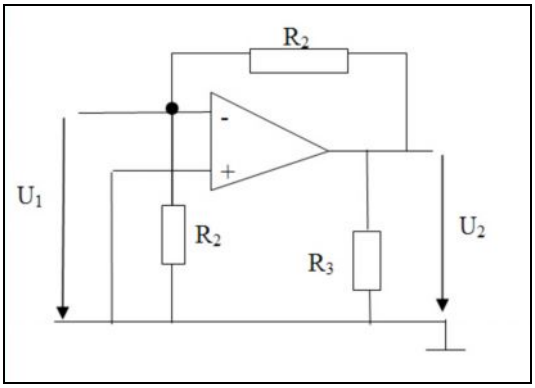

Fig. 1.7 - Circuitul pentru problema 9

BIBLIOGRAFIE

1. David Goldberg - "*What Every Computer Scientist Should Know About Floating-Point Arithmetic*", ACM Computing Surveys, Vol 23, No 1, March 1991, pp 6-48.
2. John R. Hauser - "*Handling Floating-Point Exceptions in Numeric Programs*", ACM Transactions on Programming Languages and Systems, Vol. 18, No. 2, March 1996, pp. 139-174.
3. Ioan Rusu - "*Metode Numerice în electronică - aplicații în limbaj C*", Ed. Tehnică, București, 1997
4. "*IEEE floating point representations*", material disponibil la adresa: http://www.math.frin.edu/~cstone/courses/fundamentals/IEEE-reals.html
5. ****Net* - documentații din diferite surse publice.