Algoritmi legaţi de Aritmetica în virgulă flotantă

Radu Trîmbiţaş

13 martie 2024

1 Introducere

O sursă importantă legată de aritmetica în virgulă flotantă este articolul lui David Goldberg din 1991 [3]. Recomandăm şi cartea lui Overton [6]. O sursă exhaustivă este [5].

2 Evitarea anulării

2.1 Calculul lui π

Exemplul este din [2]. O problemă celebră a antichităţii este să se construiască un pătrat care are aria egală cu cea a cercului unitate. Problema determinării unei metode de transformare a cercului în acest mod (cuadratura cercului) a rămas nerezolvată până în secolul al 19-lea, când s-a demonstrat cu ajutorul teoriei Galois că nu poate fi rezolvată cu rigla şi compasul.

Ştim că aria cercului este $A = \pi r^2$, unde r este raza cercului. O aproximare se obţine înscriind poligoane regulate în cerc şi calculând ariile lor. Aproximarea se îmbunătăţeşte crescând numărul de laturi. Arhimede a reuşit să producă un poligon cu 96 de laturi şi a încadrat π în intervalul $\left(3\frac{10}{71}, 3\frac{1}{7}\right)$. Lungimea intervalului este $1/497 = 0.00201207243$ — o aproximare bună pentru aplicaţiile din acel timp.

Pentru a calcula o astfel de aproximaţie poligonală a lui π, să considerăm figura 1. Fără a restrânge generalitatea, putem presupune că $r = 1$. Atunci aria F_n a triunghiului isocel ABC cu unghiul la centru $\alpha_n := \frac{2\pi}{n}$ este

$$F_n = \frac{\sin \alpha_n}{2},$$

iar aria poligonului regulat corespunzător cu n laturi devine

$$A_n = nF_n = \frac{n}{2} \sin \alpha_n = \frac{n}{2} \sin \frac{2\pi}{n}.$$

Evident, calculul aproximantei A_n utilizând π poate părea vicioasă. Din fericire, A_{2n} poate fi dedusă din A_n printr-o formulă simplă, exprimând $\sin(\alpha_n/2)$ în

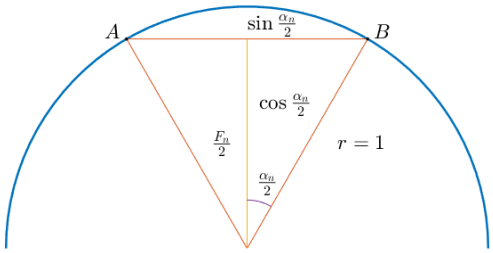

Figura 1: Cuadratura cercului

funcţie de $\sin\alpha_n$ cu formula trigonometrică:

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}. \tag{1}$$

Am obţinut astfel o relaţie de recurenţă pentru $\sin(\alpha_n/2)$ din $\sin\alpha_n$. Pentru a iniţializa recurenţa vom calcula aria A_6 a hexagonului regulat. Lungimea laturii fiecăruia din cele şase triunghiuri echilaterale este 1 iar unghiul la centru este $\alpha_6 = \frac{\pi}{3}$, de unde $\sin\alpha_6 = \frac{\sqrt{3}}{2}$. Deci, aria triunghiului este $F_6 = \frac{\sqrt{3}}{4}\sin\alpha_n$, iar $A_6 = \frac{\sqrt{3}}{2}$. Am obţinut un program pentru calculul şirului de aproximaţii A_n (vezi algoritmul 1).

Sursa MATLAB 1 Calculul lui π, versiunea naivă

```
s=sqrt(3)/2; A=3*s; n=6; % initialization
z=[A-pi n A s]; % store the results
while s>1e-10 % termination if s=sin(alpha) small
   s=sqrt((1-sqrt(1-s*s))/2); % new sin(alpha/2) value
   n=2*n; A=n/2*s; % A=new polygon area
   z=[z; A-pi n A s];
end
m=length(z);
for i=1:m
   fprintf('%10d %20.15f %20.15f %20.15f\n',...
        z(i,2),z(i,3),z(i,1),z(i,4))
end
```

Rezultatele, afişate în tabela 1, nu sunt cele aşteptate: iniţial, observăm că A_n se apropie de π, dar pentru $n > 49152$, eroarea începe să crească şi la final se obţine $A_n = 0$?! Deşi teoria şi programul sunt ambele corecte, rezultatele sunt incorecte. Vom explica de ce se întâmplă aşa.

n	A_n	$A_n - \pi$	$\sin \alpha_n$
6	2.598076211353316	−0.543516442236477	0.866025403784439
12	3.000000000000000	−0.141592653589794	0.500000000000000
24	3.105828541230250	−0.035764112359543	0.258819045102521
48	3.132628613281237	−0.008964040308556	0.130526192220052
96	3.139350203046872	−0.002242450542921	0.065403129230143
192	3.141031950890530	−0.000560702699263	0.032719082821776
384	3.141452472285344	−0.000140181304449	0.016361731626486
768	3.141557607911622	−0.000035045678171	0.008181139603937
1536	3.141583892148936	−0.000008761440857	0.004090604026236
3072	3.141590463236762	−0.000002190353031	0.002045306291170
6144	3.141592106043048	−0.000000547546745	0.001022653680353
12288	3.141592516588155	−0.000000137001638	0.000511326906997
24576	3.141592618640789	−0.000000034949004	0.000255663461803
49152	3.141592645321216	−0.000000008268577	0.000127831731987
98304	3.141592645321216	−0.000000008268577	0.000063915865994
196608	3.141592645321216	−0.000000008268577	0.000031957932997
393216	3.141592645321216	−0.000000008268577	0.000015978966498
786432	3.141593669849427	0.000001016259634	0.000007989485855
1572864	3.141592303811738	−0.000000349778055	0.000003994741190
3145728	3.141608696224804	0.000016042635011	0.000001997381017
6291456	3.141586839655041	−0.000005813934752	0.000000998683561
12582912	3.141674265021758	0.000081611431964	0.000000499355676
25165824	3.141674265021758	0.000081611431964	0.000000249677838
50331648	3.143072740170040	0.001480086580246	0.000000124894489
100663296	3.159806164941135	0.018213511351342	0.000000062779708
201326592	3.181980515339464	0.040387861749671	0.000000031610136
402653184	3.354101966249685	0.212509312659892	0.000000016660005
805306368	4.242640687119286	1.101048033529493	0.000000010536712
1610612736	6.000000000000000	2.858407346410207	0.000000007450581
3221225472	0.000000000000000	−3.141592653589793	0.000000000000000

Tabela 1: Calcul instabil al lui π

Pentru a calcula $\sin(\alpha/2)$ din $\sin\alpha$, am folosit recurenţa (1):

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}.$$

Deoarece $\sin\alpha_n \to 0$, numărătorul din membrul drept este $1-\sqrt{1-\varepsilon^2}$, cu $\varepsilon = \sin\alpha_n$ mic şi suferă de anulare severă. Acesta este motivul pentru care

algoritmul lucrează aşa de prost, chiar dacă teoria şi programul sunt ambele corecte.

Este posibil să rearanjăm în acest caz calculele şi să evităm anularea:

$$\sin\frac{\alpha_n}{2}=\sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}=\sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}\frac{1+\sqrt{1-\sin^2\alpha_n}}{1+\sqrt{1-\sin^2\alpha_n}}}$$
$$=\sqrt{\frac{1-\left(1-\sin^2\alpha_n\right)}{2\left(1+\sqrt{1-\sin^2\alpha_n}\right)}}=\frac{\sin\alpha_n}{\sqrt{2\left(1+\sqrt{1-\sin^2\alpha_n}\right)}}.$$

Ultima expresie nu mai suferă de anulare şi obţinem un nou program (vezi susa MATLAB 2)

Sursa MATLAB 2 Calculul lui π, versiune stabilă

```
oldA=0;s=sqrt(3)/2; newA=3*s; n=6; % initialization
z=[newA-pi n newA s]; % store the results
%The stopping criterion
while newA>oldA % quit if area does not increase
   oldA=newA;
   s=s/sqrt(2*(1+sqrt((1+s)*(1-s)))); % new sine value
   n=2*n; newA=n/2*s;
   z=[z; newA-pi n newA s];
end
m=length(z);
for i=1:m
   fprintf('%10d %20.15f %20.15f\n',z(i,2),z(i,3),z(i,1))
end
```

De data aceasta converge către valoarea corectă a lui π (vezi tabela 2). De remarcat criterul de oprire elegant: deoarece suprafaţa următorului poligon creşte, teoretic avem

$$A_6<\cdots<A_n<A_{2n}<\pi.$$

Totuşi, în aritmetica cu precizie finită, acest lucru nu poate avea loc la nesfârşit, deoarece avem o mulţime finită de numere maşină. Astfel, la un moment dat, trebuie să apară situaţia $A_n\geq A_{2n}$ şi aceasta va fi condiţia de oprire a iteraţiilor. De notat că această condiţie este independentă de maşină, în sensul că iteraţia se va termina întotdeauna, atât timp cât vom avea aritmetică de precizie finită şi atunci când se termină, ne va da cea mai bună aproximaţie pentru prezizia maşinii. Mai multe exemple de algoritmi independenţi de maşină se vor da în secţiunea 5.

n	A_n	$A_n - \pi$
6	2.598076211353316	−0.543516442236477
12	3.000000000000000	−0.141592653589793
24	3.105828541230249	−0.035764112359544
48	3.132628613281238	−0.008964040308555
96	3.139350203046867	−0.002242450542926
192	3.141031950890509	−0.000560702699284
384	3.141452472285462	−0.000140181304332
768	3.141557607911858	−0.000035045677936
1536	3.141583892148318	−0.000008761441475
3072	3.141590463228050	−0.000002190361744
6144	3.141592105999271	−0.000000547590522
12288	3.141592516692156	−0.000000136897637
24576	3.141592619365383	−0.000000034224410
49152	3.141592645033690	−0.000000008556103
98304	3.141592651450766	−0.000000002139027
196608	3.141592653055036	−0.000000000534757
393216	3.141592653456104	−0.000000000133690
786432	3.141592653556371	−0.000000000033422
1572864	3.141592653581438	−0.000000000008355
3145728	3.141592653587705	−0.000000000002089
6291456	3.141592653589271	−0.000000000000522
12582912	3.141592653589663	−0.000000000000130
25165824	3.141592653589761	−0.000000000000032
50331648	3.141592653589786	−0.000000000000008
100663296	3.141592653589791	−0.000000000000002
201326592	3.141592653589794	0.000000000000000
402653184	3.141592653589794	0.000000000000001
805306368	3.141592653589794	0.000000000000001

Tabela 2: Calcul stabil al lui π

2.2 Ecuația de gradul al doilea

Pentru elaborarea acestei secțiuni s-au folosit materialele [4, 1]. La calculul rădăcinilor ecuației $ax^2+bx+c=0$ în AVF pot să apară următoarele probleme:

1. Anulare dacă $b^2 \gg 4ac$; în acest caz una dintre rădăcini este prost conditionată, depinzând de semnul lui b. Rădăcina bine condiționată se calculează cu formula
$$x_1 = \frac{-b + \text{sign}(b)\sqrt{\Delta}}{2a},$$
iar pentru cea prost condiționată se utilizează relațiile lui Viete,
$$x_2 = \frac{\frac{c}{a}}{x_1}.$$

2. Anulare dacă $b^2 \approx 4ac$; în acest caz este necesar calculul lui Δ cu precizie mai mare.

3. Posibilă depăşire la calculul lui $\Delta = b^2 - 4ac$; dacă $\sqrt{\Delta}$ este reprezentabil, el trebuie calculat.

`QUADEQU` - rezolvitor pentru ecuația de gradul al doilea
Apel: `[x1,x2] = quadequ(a,b,c)`
Parametrii:
`a,b,c` - coeficienții ecuației $ax^2 + bx + c = 0$ Utilizează formula clasică

$$x_1 = \frac{-b + \sqrt{\Delta}}{2a}$$
$$x_2 = \frac{-b - \sqrt{\Delta}}{2a},$$

unde $\Delta = b^2 - 4ac$.

```
function [x1,x2] = quadequ(a,b,c)
b=b/2;
delta=Kahandiscr(a,b,c);
if ~isinf(delta) && ~isnan(delta)
    d=sqrt(delta);
    x1 = (-b - sign(b)*d) / a;
    x2 = c/a/x1;
else %overflow
    m=max(abs([a,b,c]));
    delta=Kahandiscr(a/m,b/m,c/m);
    d=m*sqrt(delta);
    x1 = (-b-sign(b)*d)/a;
    x2 = c/a/x1; %-(2*c)/w;
```

```
end
end
```

Subfuncția `Kahandiscr` calculează Δ.

```
function d=Kahandiscr(a,b,c)
p=b*b; q=a*c; %normal
if p+q<=3*abs(p-q)
    d=p-q;
else %discriminant affected by cancellation
    dp=exactmult(b,b,p);
    dq=exactmult(a,c,q);
    d=(p-q)-(dp-dq);
end
end
```

Subfuncția `exactmult` simulează calculul discriminantului în precizie cvadruplă.

```
function rez=exactmult(x,y,xy)
C=pow2(27)+1; %2^27+1
px=x*C;
hx=(x-px)+px;
tx=x-hx;
py=y*C;
hy=(y-py)+py;
ty=y-hy;
rez=-((((xy-hx*hy)-hx*ty)-hy*tx)-tx*ty);
end
```

3 Exponenţiala

Considerăm un al alt exemplu, legat de calculul funcţiei exponenţiale utilizând dezvoltarea Taylor:

$$e^x = \sum_{j=0}^{\infty} \frac{x^j}{j!} == 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \frac{1}{24}x^4 + \frac{1}{120}x^5 + \cdots$$

Se ştie că seria converge pentru orice x. O abordare naivă este (în pregătirea unei versiuni ulterioare mai bune, am scris calculele din ciclu într-o formă particulară):

Pentru x pozitiv, dar şi pentru x negativ mic în modul, codul funcţionează bine:

```
>> ExpUnstable(20,1e-8)
```

Sursa MATLAB 3 Computation of e^x, Naive Version function

```
function s=ExpUnstable(x,tol)
% EXPUNSTABLE computation of the exponential function
% s=ExpUnstable(x,tol); computes an approximation s of exp(x)
% up to a given tolerance tol.
% WARNING: cancellation for large negative x.

if nargin<2, tol=eps; end
s=1; term=1; k=1;
while abs(term)>tol*abs(s)
   so=s; term=term*x/k;
   s=so+term; k=k+1;
end
```

```
ans =
     4.851651930670549e+08
>> exp(20)
ans =
     4.851651954097903e+08
>> ExpUnstable(1)
ans =
     2.718281828459046e+00
>> exp(1)
ans =
     2.718281828459046e+00
>> ExpUnstable(-1, 1e-8)
ans =
     3.678794413212817e-01
>> exp(-1)
ans =
     3.678794411714423e-01
>>
```

Dar pentru x negativ, mare în modul, e.g. pentru $x = -20$ şi $x = -50$, obţinem

```
>> ExpUnstable(-20, 1e-8)
ans =
     5.621884467407823e-09
>> exp(-20)
ans =
     2.061153622438558e-09
>> ExpUnstable(-50)
ans =
     1.107293338289197e+04
```

```
>> exp(-50)
ans =
     1.928749847963918e-22
>>
```

care sunt complet incorecte. Motivul este că pentru $x = -20$, termenii seriei

$$e^{-20} = 1 - 20 + \frac{20^2}{2} - \frac{20^3}{3!} + \cdots + \frac{20^{20}}{20!} - \frac{-20^{21}}{21!} + \cdots$$

devin mari şi semnele alternează. Cei mai mari termeni sunt

$$\frac{20^{19}}{19!} = \frac{20^{20}}{20!} \approx 4.31e7.$$

Şirul sumelor parţiale converge către $e^{-20} \approx 2.06e - 9$. Dar, datorită creşterii termenilor, sumele parţiale cresc şi oscilează, aşa cum se arată în figura 2. Tabela 3 ne arată că cele mai mari sume parţiale au cam aceeaşi mărime ca cel mai mare termen. Deoarece sumele parţiale mari trebuie *diminuate prin adunare/scădere de termeni*, aceasta nu se poate întâmpla fără anulare.

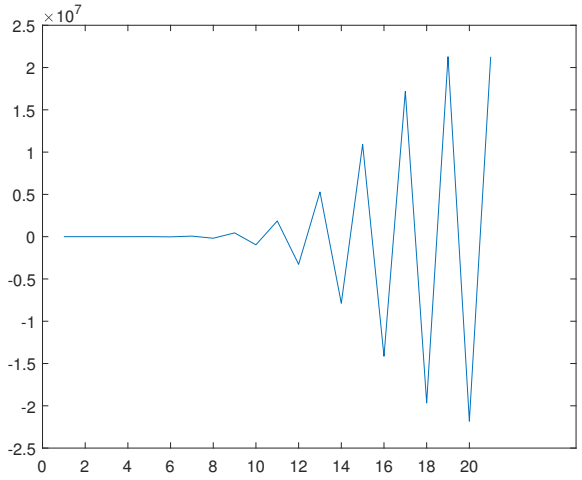

Figura 2: Suma parţială a dezvoltării Taylor a lui e^{-20}

Nu ajută nici însumarea separată a termenilor pozitivi şi a celor negativi, deoarece atunci când sumele sunt scăzute la sfârşit, rezultatul va suferi din nou de anulare catastrofală. Într-adevăr, deoarece rezultatul

$$e^{-20} \approx 5 \cdot 10^{-17} \frac{20^{20}}{20!}$$

este cu aproximativ 17 ordine de mărime mai mic decât cea mai mare sumă parţială intermediară şi standardul IEEE prevede doar o precizie de 16 cifre zecimale, nu ne aşteptăm să obţinem nici măcar o cifră corectă!

#termeni	suma parţială
20	-2.182259377927747e+07
40	-9.033771892138389e+03
60	-1.042349683933131e-04
80	+5.621883118107117e-09
100	+5.621884472130418e-09
120	+5.621884472130418e-09
valoare exactă	2.061153622438558e-09

Tabela 3: Sume parţiale ale lui e^{-20} calculate numeric

4 Sumarea compensată a lui Kahan

Goldberg [3] recomandă dublarea preciziei pentru însumarea unui număr mare de termeni. Kahan a dat un algoritm pentru însumare fără dublarea preciziei.

Teorema 1 (Sumare Kahan) *Presupunem că* $S = \sum_{j=1}^{N} x_j$ *se calculează cu algoritmul din sursa MATLAB 4.*

Sursa MATLAB 4 Sumare compensată Kahan

```
function s = KahanSummation(x)
s=x(1);        % partial sum
c=0;           % carry
N=length(x);
for j=2:N
   y=x(j)-c;
   t=s+y;     %next partial sum, with roundoff
   c=(t-s)-y; %recapture error and store as carry
   s=t;
end
end
```

Atunci suma calculată S *este egală cu* $\sum x_j(1+\delta_j)+O(N\text{eps}^2)\sum|x_j|$*, unde* $|\delta_j| \leq 2\text{eps}$.

Utilizând formula naivă $\sum x_j$, suma calculată este egală cu $\sum x_j(1+\delta_j)$unde $|\delta_j| \leq (n-j)\epsilon$. Comparând aceasta cu eroarea din teoremă, se observă o îmbunătățire dramatică. Fiecare sumand este perturbat cu doar 2ϵ în loc de $n\epsilon$ în formula naivă. Pentru demonstrație vezi [3].

Cea mai simplă abordare de îmbunătățire a preciziei acurateții este dublarea peciziei. Pentru a obtine o estimare grosieră a cât de mult se îmbunătățește acuratețea sumei prin dublarea preciziei, fie $s_1 = x_1, s_2 = s_1 \oplus x_2, \ldots, s_i =$

$s_{i-1} \oplus x_i$. Then, $s_i = (1+\delta_i)(s_{i-1}+x_i)$, where $|\delta_i| \leq$ eps si ignorând termenii de ordinul al doilea în δ_i avem

$$\begin{aligned} s_n &= \sum_{j=1}^{N} x_j \left(1 + \sum_{k=j}^{N} \delta_k\right) \\ &= \sum_{j=1}^{N} x_j + \sum_{j=1}^{N} x_j \left(\sum_{k=j}^{N} \delta_k\right) \end{aligned} \tag{2}$$

Prima egalitate din (2) ne arată că valoarea calculată a lui $\sum x_j$ este la fel ca sumarea exactă a valorilor perturbate ale lui x_j. Primul termen x_l este perturbat cu $n$eps, ultimul x_n doar cu eps. A doua egalitate din (2) ne arată că termenul de eroare este mărginit de $n\text{eps}\sum|x_j|$. Dublarea preciziei are ca efect ridicarea la pătrat a lui eps. Dacă însumarea se face în dublă precizie după standardul IEEE, 1/eps $=\approx 10^{16}$, astfel că $n\text{eps} \ll 1$ pentru orice valoare rezonabilă a lui n. Astfel, dublarea preciziei schimbă perturbația maximă din $n\epsilon$ în $n\text{eps}^2 \ll$ eps. Astfel, marginea 2eps din sumarea Kahan (teorema 1) nu este la fel de bună ca cea de la dubla precizie, dar este mult mai buna decât cea din simplă precizie.

Am testat funcția `KahanSummation` la calculul sumei armonice $\sum_{k=1}^{N} \frac{1}{k}$ pentru $N = 10^6$.

```
N=1e6;
x=1./(1:N);
sKahan=KahanSummation(x)
```

```
sKahan =
1.439272672286572e+01
```

Pentru o explicație intuitivă a modului în care lucrează sumarea Kahan, vezi diagrama procedurii din figura 3.

5 Algoritmi independenţi de maşină

Considerăm din nou exemplul cu calculul funcţiei exponenţiale utilizând serii Taylor. Am văzut că obţinem rezultate bune pentru $x > 0$. Utilizând formula lui Stirling $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, observăm că pentru un x dat, al n-lea termen satisface

$$t_n = \frac{x^n}{n!} \sim \frac{1}{\sqrt{2\pi n}}\left(\frac{xe}{n}\right)^n \to 0, \qquad n \to \infty.$$

Cel mai mare termen al dezvoltării este deci în jurul lui $n \approx x$, aşa cum se poate vedea prin derivare. Pentru n mai mare, termenii descresc şi converg către zero. Numeric, termenul t_n devine atât de mic în aritmetica de precizie finită, încât

$$s_n + t_n = s_n, \quad \text{unde } s_n = \sum_{i=1}^{n} \frac{x^i}{i!}$$

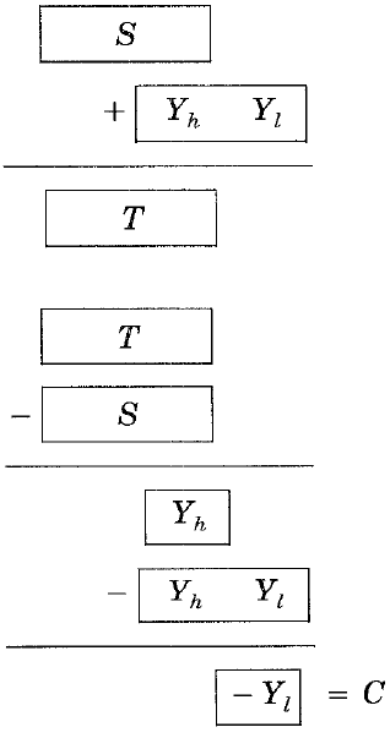

Figura 3: Sumarea Kahan

Acesta este un criteriu elegant de terminare, care nu depinde de detaliile aritmeticii în virgulă flotantă, dar face uz de numărul finit de cifre din semnificant. În acest sens înţelegem independenţa de maşină a algoritmului; el nu funcţionează în aritmetica exactă, deoarece nu s-ar termina niciodată.

Pentru a evita anularea când $x < 0$, utilizăm o proprietate a funcţiei exponenţiale şi anume $e^x = 1/e^{-x}$: calculăm întâi $e^{|x|}$ şi apoi $e^x = 1/e^{|x|}$. Obţinem în acest mod un algoritm stabil pentru calculul funcţiei exponenţiale pentru orice x:

Sursa MATLAB 5 Calculul stabil al lui e^x

```
function s=ExpStable(x)
% EXPSTABLE stable computation of the exponential function
% s=Exp(x); computes an approximation s of exp(x) up to machine
% precision.
if x<0, v=-1; x=abs(x); else v=1; end
so=0; s=1; term=1; k=1;
while s~=so
   so=s; term=term*x/k;
   s=so+term; k=k+1;
end
if v<0, s=1/s; end;
end
```

Acum obţinem rezultate foarte bune şi pentru valori negative mari în modul ale lui x:

```
>> format long
[ExpStable(-20), exp(-20)]
ans =
   1.0e-08 *
   0.206115362243856   0.206115362243856
>> [ExpStable(-50), exp(-50)]
ans =
   1.0e-21 *
   0.192874984796392   0.192874984796392
>> [ExpStable(-100), exp(-100)]
ans =
   1.0e-43 *
   0.372007597602084   0.372007597602084
>>
```

De notat că am calculat termenii recursiv $t_k = t_{k-1}\frac{x}{k}$ şi nu explicit $t_k = \frac{x^k}{k!}$ pentru a evita posibilele depăşiri la numărător sau la numitor şi a reduce numărul de flops.

Un al doilea exemplu este legat de proiectarea unui algoritm pentru calculul rădăcinii pătrate. Fiid dat $a > 0$, dorim să calculăm

$$x = \sqrt{a} \Longleftrightarrow f(x) = x^2 - a = 0.$$

Aplicând iteraţia Newton, obţinem

$$x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x} = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

şi iteraţia convergentă pătratic (cunoscută şi sub numele de *formula lui Heron.*)

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right). \tag{3}$$

Când se termină iteraţia? Putem desigur să testăm dacă două iteraţii diferă până la o anumită toleranţă relativă. Dar, în acest caz putem concepe un criteriu de oprire mai elegant. Interpretarea geometrică a metodei lui Newton ne arată că dacă $a < x_k$, atunci $\sqrt{a} < x_{k+1} < x_k$. Astfel, dacă pornim iteraţia cu $\sqrt{a} < x_0$, atunci şirul $\{x_k\}$ este monoton crescător către $s = \sqrt{a}$. Această monotonie nu poate avea loc la nesfârşit pe o maşină cu aritmetică de precizie finită. Ea se pierde când atingem precizia maşinii.

Pentru a utiliza acest criteriu, trebuie să ne asigurăm că $\sqrt{a} < x_0$. Aceasta se realizează uşor, aplicând inegalitatea mediilor

$$\sqrt{a} = \sqrt{a \cdot 1} \leq \frac{a+1}{2}.$$

Deci, alegem $x_0 = \frac{a+1}{2}$. Obţinem algoritmul 6.

Sursa MATLAB 6 Calculul lui $\sqrt{a}$ independent de maşină

```
function y=Sqrt(a)
% SQRT computes the square-root of a positive number
% a using Newton's method, up to machine precision.
xo=(1+a)/2; xn=(xo+a/xo)/2;
% stopping criterion breaking the monotony
while xn<xo
   xo=xn; xn=(xo+a/xo)/2;
end
y=(xo+xn)/2;
```

Observaţi eleganţa algoritmului 6: nu este nevoie de nici o toleranţă pentru criteriul de terminare. Algoritmul calculează rădăcina pătrată pe orice calculator fără a şti precizia maşinii, pur şi simplu utilizând faptul că există doar un număr finit de numere maşină. Acest algoritm nu va lucra pe o maşină cu aritmetică exactă — el se bazează pe aritmetica de precizie finită. Adesea aceştia sunt cei mai buni algoritmi care se pot proiecta.

Un alt algoritm care se justifică prin sine şi este independent de maşină este algoritmul bisecţiei sau înjumătăţirii pentru determinarea unei rădăcini simple. El se poate implementa astfel ca să facă uz de finitudinea mulţimii numerelor maşină. Înjumătăţirea continuă atât timp cât există un număr maşină

în intervalul (a, b). Când intervalul constă numai din capete, iteraţia se termină într-un mod independent de maşină. Vezi algoritmul 7 pentru detalii.

Algoritmii independenţi de maşină nu sunt uşor de găsit.

Sursa MATLAB 7 Înjumătăţirea intervalului

```
function [x,y]=Bisection(f,a,b,tol)
% BISECTION computes a root of a scalar equation
% [x,y]=Bisection(f,a,b,tol) finds a root x of the scalar function
% f in the interval [a,b] up to a tolerance tol. y is the
% function value at the solution
fa=f(a); v=1;
if fa>0, v=-1; end
if fa*f(b)>0
   error('f(a) and f(b) have the same sign')
end
if (nargin<4), tol=0; end;
x=(a+b)/2;
while (b-a>tol) && ((a < x) && (x<b))
   if v*f(x)>0, b=x; else a=x; end;
   x=(a+b)/2;
end
if nargout==2, y=f(x); end;
```

Bibliografie

[1] S. Boldo, *Kahan's Algorithm for a Correct Discriminant Computation at Last Formally Proven*, IEEE Transactions on Computers **58** (2009), no. 2, 220–225.

[2] Walter Gander, Martin J. Gander, and Felix Kwok, *Scientific Computing. An Introduction using Maple and MATLAB*, Springer, Heidelberg, New York, Dordrecht, London, 2014.

[3] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, Computing Surveys **23** (1991), no. 1, 5–48.

[4] W. Kahan, *On the Cost of Floating-Point Computation Without Extra-Precise Arithmetic*, https://people.eecs.berkeley.edu/~wkahan/Qdrtcs.pdf, 2004, Accessed: 2024-03-06.

[5] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres, *Handbook of floating-point arithmetic*, 2nd ed.,

Birkhäuser Boston, 2018, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[6] M.L. Overton, *Numerical computing with IEEE floating point arithmetic*, SIAM, Philadelphia, PA, 2001.