

Algoritmi legați de Aritmetica în virgulă flotantă

Radu Trîmbițaș

28 februarie 2021

1 Evitarea anulării

1.1 Calculul lui π

Exemplul este din [1]. O problemă celebră a antichității 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 și compasul.

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

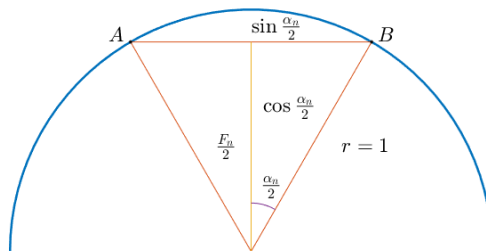


Figura 1: Cuadratura cercului

Pentru a calcula o astfel de aproximație 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

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

iar aria poligonului regulat corespunzător cu n laturi devine

$$A_n = nF_n = \frac{n}{2} \left(2 \cos \frac{\alpha_n}{2} \sin \frac{\alpha_n}{2} \right) = \frac{n}{2} \sin \alpha_n = \frac{n}{2} \sin \frac{2\pi}{n}.$$

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

$$\sin \frac{\alpha_n}{2} = \sqrt{\frac{1 - \cos \alpha_n}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}}. \quad (1)$$

Am obținut astfel o relație de recurență pentru $\sin(\alpha_n/2)$ din $\sin \alpha_n$. Pentru a inițializa recurența vom calcula aria A_6 a hexagonului regulat. Lungimea laturii fiecăruia din cele șase 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 $F_6 = \frac{\sqrt{3}}{4} \sin \alpha_n \sin \alpha_n$, iar $A_6 = \frac{\sqrt{3}}{2}$. Am obținut un program pentru calculul șirului de aproximații 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, afișate în tabela 1, nu sunt cele așteptate: inițial, observăm că A_n se apropie de π , dar pentru $n > 49152$, eroarea începe să crească și la final se obține $A_n = 0$!? Deși teoria și programul sunt ambele corecte, rezultatele sunt incorecte. Vom explica de ce se întâmplă așa.

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

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

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

Deoarece $\sin \alpha_n \rightarrow 0$, numărătorul din membrul drept este $1 - \sqrt{1 - \varepsilon^2}$, cu $\varepsilon = \sin \alpha_n$ mic și suferă de anulare severă. Acesta este motivul pentru care algoritmul lucrează așa de prost, chiar dacă teoria și programul sunt ambele corecte.

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

$$\begin{aligned} \sin \frac{\alpha_n}{2} &= \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2}} = \sqrt{\frac{1 - \sqrt{1 - \sin^2 \alpha_n}}{2} \frac{1 + \sqrt{1 - \sin^2 \alpha_n}}{1 + \sqrt{1 - \sin^2 \alpha_n}}} \\ &= \sqrt{\frac{1 - (1 - \sin^2 \alpha_n)}{2(1 + \sqrt{1 - \sin^2 \alpha_n})}} = \frac{\sin \alpha_n}{\sqrt{2(1 + \sqrt{1 - \sin^2 \alpha_n})}}. \end{aligned}$$

Ultima expresie nu mai suferă de anulare și obținem 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 criteriul de oprire elegant: deoarece suprafața următorului poligon crește, teoretic avem

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

Totuși, în aritmetica cu precizie finită, acest lucru nu poate avea loc la nesfârșit, deoarece avem o mulțime finită de numere mașină. Astfel, la un moment dat, trebuie să apară situația $A_n \geq A_{2n}$ și aceasta va fi condiția de oprire a iterațiilor. De notat că această condiție este independentă de mașină, în sensul că iterația se va termina întotdeauna, atât timp cât vom avea aritmetica de precizie finită și atunci când se termină, ne va da cea mai bună aproximație pentru prezizia mașinii. Mai multe exemple de algoritmi independenți de mașină se vor da în secțiunea 3.

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 Exponențiala

Considerăm un al doilea exemplu, legat de calculul funcției exponențiale utilizând dezvoltarea Taylor:

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

Se știe 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):

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
```

Pentru x pozitiv, dar și pentru x negativ mic în modul, codul funcționează bine:

```
>> ExpUnstable(20,1e-8)
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 $x = -20$ și $x = -50$, obținem

```
>> 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

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

devin mari și semnele alternează. Cei mai mari termeni sunt

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

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

Nu ajută nici însumarea separată a termenilor pozitivi și a celor negativi, deoarece atunci când sumele sunt scăzute la sfârșit, 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 parțială intermediară și standardul IEEE prevede doar o precizie de 16 cifre zecimale, nu ne așteptăm să obținem nici măcar o cifră corectă!

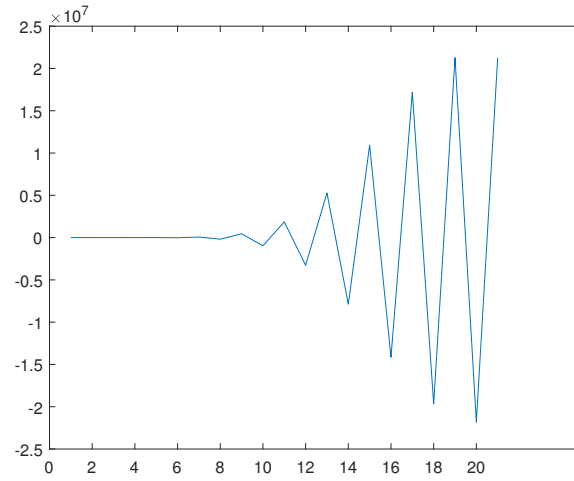


Figura 2: Suma parțială a dezvoltării Taylor a lui e^{-20}

#termeni	suma parțială
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 parțiale ale lui e^{-20} calculate numeric

3 Algoritmi independenți de mașină

Considerăm din nou exemplul cu calculul funcției exponențiale utilizând serii Taylor. Am văzut că obținem 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

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

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

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

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 înțelegem independența de mașină a algoritmului; el nu funcționează în aritmetica exactă, deoarece nu s-ar termina niciodată.

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

Sursa MATLAB 4 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 obținem rezultate foarte bune și 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 $t_k = t_{k-1} \frac{x}{k}$ și nu explicit $t_k = \frac{x^k}{k!}$ pentru a evita posibilele depășiri la numărător sau la numitor și a reduce numărul de flops.

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

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

Aplicând iterația Newton, obținem

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

și iterația convergentă pătratic (cunoscută și sub numele de *formula lui Heron*.)

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

Când se termină iterația? Putem desigur să testăm dacă două iterații diferă până la o anumită toleranță 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 iterația cu $\sqrt{a} < x_0$, atunci șirul $\{x_k\}$ este monoton crescător către $s = \sqrt{a}$. Această monotonie nu poate avea loc la nesfârșit pe o mașină cu aritmetică de precizie finită. Ea se pierde când atingem precizia mașinii.

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

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

Deci, alegem $x_0 = \frac{a+1}{2}$. Obținem algoritmul 5.

Observați eleganța algoritmului 5: nu este nevoie de nici o toleranță pentru criteriul de terminare. Algoritmul calculează rădăcina pătrată pe orice calculator fără a ști precizia mașinii, pur și simplu utilizând faptul că există doar un număr finit de numere mașină. Acest algoritm nu va lucra pe o mașină cu aritmetică exactă — el se bazează pe aritmetica de precizie finită. Adesea aceștia sunt cei mai buni algoritmi care se pot proiecta.

Un alt algoritm care se justifică prin sine și este independent de mașină este algoritmul biseției sau înjumătățirii pentru determinarea unei rădăcini

Sursa MATLAB 5 Calculul lui \sqrt{a} independent de mașină

```
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;
```

simple. El se poate implementa astfel ca să facă uz de finitudinea mulțimii numerelor mașină. Înjumătățirea continuă atât timp cât există un număr mașină în intervalul (a, b) . Când intervalul constă numai din capete, iterația se termină într-un mod independent de mașină. Vezi algoritmul 6 pentru detalii.

Algoritmii independenți de mașină nu sunt ușor de găsit.

Sursa MATLAB 6 Înjumătățirea 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 nargin==2, y=f(x); end;
```

Bibliografie

- [1] Walter Gander, Martin J. Gander, and Felix Kwok, *Scientific Computing. An Introduction using Maple and MATLAB*, Springer, Heidelberg, New York,

Dordrecht, London, 2014.