# Wissenschaftliches Programmieren

**Meik Hellmund**

Meik.Hellmund@math.uni-leipzig.de

https://www.math.uni-leipzig.de/~hellmund/teaching/julia.html

- Montags 11:15 HS 5
- A-Woche Donnerstag 13:15 HS 5

julia LaTeX

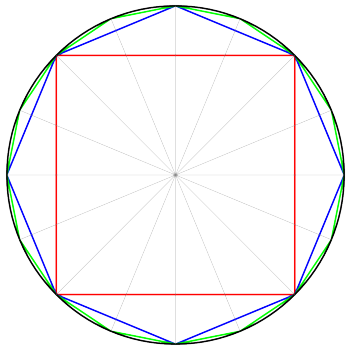
## A case study on floating-point stability

### Calculating $\pi$ according to Archimedes

- A lower bound for $2\pi$ is the sum of the side lengths of a regular $n$-gon inscribed in the unit circle.
- Start with a square, side length $s_4 = \sqrt{2}$.
- Double the number of vertices repeatedly:

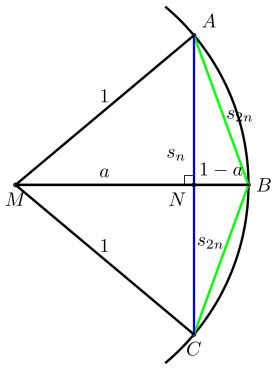
$$4 \to 8 \to 16 \to 32 \to \cdots$$

- We need a formula for $s_{2n}$ in terms of $s_n$.



Example from: Christoph Überhuber, *Computer-Numerik* Vol. 1, Springer 1995, Chap. 2.3

### Doubling the number of vertices




- Let $|\overline{AC}| = s_n$, $|\overline{AB}| = |\overline{BC}| = s_{2n}$, $|\overline{MN}| = a$, $|\overline{NB}| = 1 - a$
- Pythagoras in the triangles $MNA$ and $NBA$ yields

$$a^2 + \left(\frac{s_n}{2}\right)^2 = 1$$

$$(1-a)^2 + \left(\frac{s_n}{2}\right)^2 = s_{2n}^2$$

- Elimination of $a$ gives the recursion

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \qquad \text{with initial value} \qquad s_4 = \sqrt{2}$$

### Two iteration formulas

- The sequence $(n\, s_n)$ converges monotonically from below to $2\pi$. The relative error of the $n$-gon is

$$n\, s_n \to 2\pi, \qquad \epsilon_n = \left|\frac{n\, s_n - 2\pi}{2\pi}\right|$$

- The recursion can be rewritten: multiply numerator and denominator by $\sqrt{2 + \sqrt{4 - s_n^2}}$.

$$s_{2n} = \sqrt{2 - \sqrt{4 - s_n^2}} \qquad \text{(Iteration A)}$$

$$s_{2n} = \frac{s_n}{\sqrt{2 + \sqrt{4 - s_n^2}}} \qquad \text{(Iteration B)}$$

- Mathematically equivalent, numerically quite different.

```julia
using Printf
iterationA(s) =  sqrt(2 - sqrt(4 - s^2))
iterationB(s) = s / sqrt(2 + sqrt(4 - s^2))
s_B = s_A = sqrt(2)     # initial value
ϵ(x) = abs(x - 2π)/2π   # relative error
ϵ_A = Float64[]         # vectors for the plot
ϵ_B = Float64[]
is = Float64[]
@printf """Approx. value of π
          n-gon       Iteration A            Iteration B
"""
for i in 3:35
    push!(is, i)
    s_A = iterationA(s_A)
    s_B = iterationB(s_B)
    doublePi_A = 2^i * s_A
    doublePi_B = 2^i * s_B
    push!(ϵ_A, ϵ(doublePi_A))
    push!(ϵ_B, ϵ(doublePi_B))
    @printf "%14i %20.15f %20.15f\n" 2^i doublePi_A/2 doublePi_B/2
end
```

## Result

```
Approx. value of π
          n-gon       Iteration A            Iteration B
             8    3.061467458920719    3.061467458920718
            16    3.121445152258053    3.121445152258052
            32    3.136548490545941    3.136548490545939
            64    3.140331156954739    3.140331156954753
           128    3.141277250932757    3.141277250932773
           256    3.141513801144145    3.141513801144301
           512    3.141572940367883    3.141572940367092
          1024    3.141587725279961    3.141587725277160
          2048    3.141591421504635    3.141591421511200
          4096    3.141592345611077    3.141592345570118
          8192    3.141592576545004    3.141592576584873
         16384    3.141592633463248    3.141592634338564
         32768    3.141592654807589    3.141592648776986
         65536    3.141592645321215    3.141592652386592
        131072    3.141592607375720    3.141592653288993
        262144    3.141592910939673    3.141592653514594
        524288    3.141594125195191    3.141592653570994
```

- Digits that agree with $\pi = 3.14159265358979\ldots$ are **green**.
- Up to $n = 2^{13}$ both iterations are equally accurate.
- Then rounding errors show up in iteration A: its best value is at $n = 2^{15}$, afterwards it gets worse.

## Result

```
          n-gon       Iteration A            Iteration B
        1048576    3.141596553704820    3.141592653585094
        2097152    3.141596553704820    3.141592653588619
        4194304    3.141674265021758    3.141592653589501
        8388608    3.141829681889202    3.141592653589721
       16777216    3.142451272494134    3.141592653589776
       33554432    3.142451272494134    3.141592653589790
       67108864    3.162277660168380    3.141592653589794
      134217728    3.162277660168380    3.141592653589794
      268435456    3.464101615137754    3.141592653589795
      536870912    4.000000000000000    3.141592653589795
     1073741824    0.000000000000000    3.141592653589795
     2147483648    0.000000000000000    3.141592653589795
     4294967296    0.000000000000000    3.141592653589795
     8589934592    0.000000000000000    3.141592653589795
    17179869184    0.000000000000000    3.141592653589795
    34359738368    0.000000000000000    3.141592653589795
```

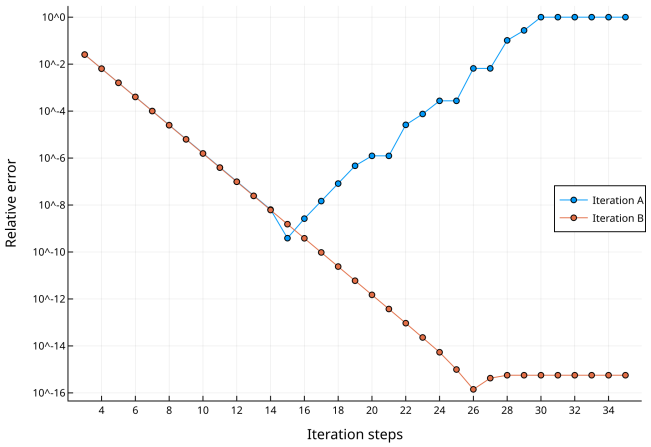
- Iteration B stabilizes at a value accurate within machine precision.
- Iteration A is ill-conditioned and numerically unstable.

## Plot of the relative errors

```julia
using Plots
plotlyjs()

plot(is, [ϵ_A ϵ_B], label=["Iteration A" "Iteration B"],
     marker=:circle, markersize=3, legend=:right,
     xlabel="Iteration steps", ylabel="Relative error",
     xticks=2:2:36, yscale=:log10, yticks=10.0 .^ (-16:2:0))
```

## Plot of the relative errors




- Iteration B: the error decreases by a factor of 4 per step, down to the order of machine epsilon.
- Iteration A: already at $i = 16$ the error begins to grow again.

## Iteration B is stable

- At $i = 26$, iteration B reaches a relative error on the order of machine epsilon.
  (The vectors start at $i = 3$, so step $i$ is stored at index $i - 2$.)

```
julia> ϵ_B[22:28]
7-element Vector{Float64}:
 5.3716034620272725e-15
 9.895059008997609e-16
 1.4135798584282297e-16
 4.240739575284689e-16
 5.654319433712919e-16
 5.654319433712919e-16
 5.654319433712919e-16
```

- Further iterations do not improve the result; it stabilizes at about 2.5 machine epsilons:

```
julia> ϵ_B[end]/eps(Float64)
2.5464790894703255
```

## Iteration A: cancellation

- The cause is a typical numerical cancellation effect.
- The side lengths $s_n$ become very small very quickly. Thus

$$a_n = \sqrt{4 - s_n^2}$$

  is only slightly smaller than 2, and computing

$$s_{2n} = \sqrt{2 - a_n}$$

  leads to a catastrophic cancellation: subtracting two nearly equal numbers leaves only the few trailing digits of $a_n$.
- In iteration B, the same $a_n$ is only *added*: $2 + a_n$ is harmless.

## Iteration A: cancellation

- Let's look at $a_n$ more closely, using `BigFloat` as a reference:

```
setprecision(80)   # precision for BigFloat: 80 bits of mantissa

s = sqrt(BigFloat(2))

@printf "     a = √(4-s^2) as BigFloat       and as Float64\n\n"

for i = 3:44
    s = iterationA(s)
    x = sqrt(4-s^2)
    if i > 20
        @printf "%i %30.26f %20.16f \n" i  x  Float64(x)
    end
end
```

## Iteration A: cancellation

| i | a as BigFloat | as Float64 |
|---|---|---|
| 21 | 1.99999999999775591177215422 | 1.9999999999977560 |
| 22 | 1.99999999999943897794303856 | 1.9999999999994389 |
| 23 | 1.99999999999985974448576005 | 1.9999999999998597 |
| 24 | 1.99999999999996493612143919 | 1.9999999999999649 |
| 25 | 1.99999999999999123403035980 | 1.9999999999999913 |
| 26 | 1.99999999999999780850758995 | 1.9999999999999978 |
| 27 | 1.99999999999999945212689707 | 1.9999999999999996 |
| 28 | 1.99999999999999986303172344 | 1.9999999999999998 |
| 29 | 1.99999999999999996575793045 | 2.0000000000000000 |
| 30 | 1.99999999999999999143948303 | 2.0000000000000000 |
| 31 | 1.99999999999999999785987034 | 2.0000000000000000 |
| 32 | 1.99999999999999999946496800 | 2.0000000000000000 |

| i | a as BigFloat | as Float64 |
|---|---|---|
| 33 | 1.99999999999999999986624159 | 2.0000000000000000 |
| 34 | 1.99999999999999999996656040 | 2.0000000000000000 |
| 35 | 1.99999999999999999999163886 | 2.0000000000000000 |
| 36 | 1.99999999999999999999790889 | 2.0000000000000000 |
| 37 | 1.99999999999999999999947722 | 2.0000000000000000 |
| 38 | 1.99999999999999999999986931 | 2.0000000000000000 |
| 39 | 1.99999999999999999999996691 | 2.0000000000000000 |
| 40 | 1.99999999999999999999999173 | 2.0000000000000000 |
| 41 | 1.99999999999999999999999835 | 2.0000000000000000 |
| 42 | 1.99999999999999999999999835 | 2.0000000000000000 |
| 43 | 1.99999999999999999999999835 | 2.0000000000000000 |
| 44 | 1.99999999999999999999999835 | 2.0000000000000000 |

- Only the red digits survive in $2 - a_n$: the number of significant digits shrinks with each step.
- In `Float64`, $a_n$ is rounded to exactly 2 from $i = 29$ on. Then $s_{2n} = 0$.
- With 80 bits this happens only slightly later: from $i = 41$ on, $a_n$ stays at the same value.
  Since $2 - a_n \approx s_n^2/4 \approx \pi^2/4^i$, each step costs 2 bits of the mantissa.

## Conclusion

- Two formulas that are mathematically equivalent can behave completely differently in floating-point arithmetic.
- `BigFloat` with 80 bits of precision only slightly delays the onset of the cancellation effect:
  - Iteration A, `Float64`: best relative error $\approx 4 \cdot 10^{-10}$ (at $i = 15$)
  - Iteration A, 80-bit `BigFloat`: best relative error $\approx 1.4 \cdot 10^{-13}$ (at $i = 22$)
  - Iteration B, `Float64`: relative error $\approx 6 \cdot 10^{-16}$

**Countermeasures against unstable algorithms typically require better, stable algorithms, not more precise machine numbers.**