University *of Ljubljana*
Faculty *of Mathematics and Physics*

1. naloga: Airyevi funkciji

Ma-Fi Praktikum 20/21

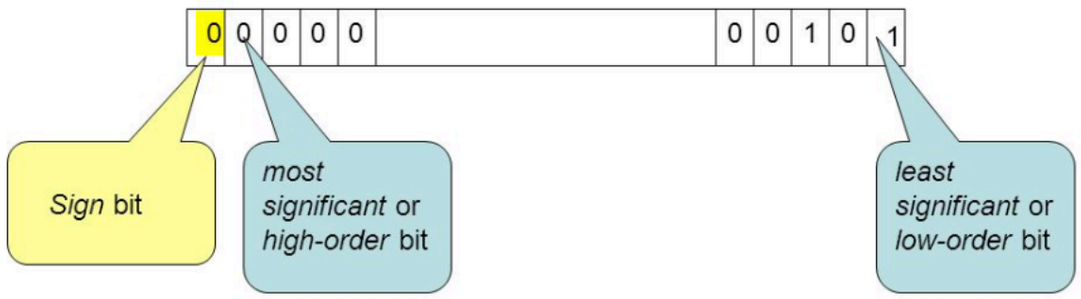
Binarni zapis

- Že znano: Naravna števila…

Binary Number:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Decimal Number:
11001101_2	128	64	32	16	8	4	2	1	205_{10}
	1	1	0	0	1	1	0	1	

- …cela stevila…

- Positive integers:



1*2^0+0*2^1+1*2^2=5

- … zoprni detajli …

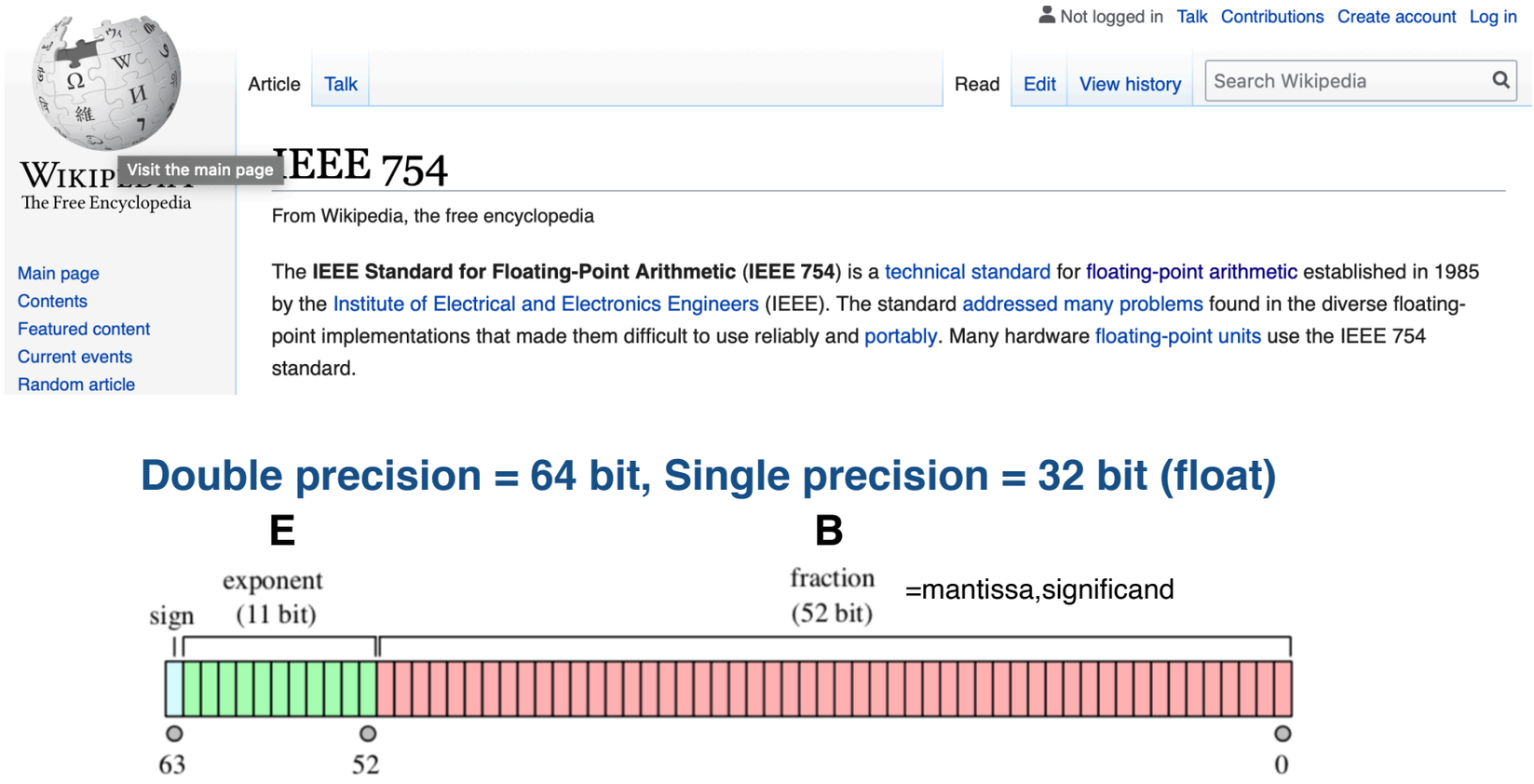
1025 kot 4-Byte (4 x 8 = 32 bit) integer:

```
00000000 00000000 00000100 00000001
```

address	big-endian	little-endian
00	00000000	00000001
01	00000000	00000100
02	00000100	00000000
03	00000001	00000000

IEEE 754

- **Kompaktni zapis in operacije decimalnih števil v binarni bazi,** ki mnogo problemov reši in nekaj povzroči..
  - Naravna števila niso problem! (so le omejena..).



Not logged in Talk Contributions Create account Log in

Article Talk | Read Edit View history | Search Wikipedia

Visit the main page

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article

IEEE 754

From Wikipedia, the free encyclopedia

The **IEEE Standard for Floating-Point Arithmetic** (**IEEE 754**) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units use the IEEE 754 standard.

Double precision = 64 bit, Single precision = 32 bit (float)

$$X = (-1)^s 2^E B$$

$$B = 1 + b_1 2^{-1} + b_2 2^{-2} + \cdots + b_{52} 2^{-52}$$

```
void IterateAllPositiveFloats()
{
    // Start at zero and print that float.
    Float_t allFloats;
    allFloats.f = 0.0f;
    printf("%1.8e\n", allFloats.f);

    // Continue through all of the floats, stopping
    // when we get to positive infinity.
    while (allFloats.RawExponent() < 255)
    {
        // Increment the integer representation
        // to move to the next float.
        allFloats.i += 1;
        printf("%1.8e\n", allFloats.f);
    }
}
```

The (partial) output looks like this:

0.00000000e+000
1.40129846e-045
2.80259693e-045
4.20389539e-045
5.60519386e-045
7.00649232e-045
8.40779079e-045
9.80908925e-045
...
3.40282306e+038
3.40282326e+038
3.40282347e+038
1.#INF0000e+000

IEEE bolj natančno

1. Single Precision

The IEEE single precision floating point standard representation requires a 32 bit word (or 4 bytes), which may be represented as numbered from 31 to 0, left to right. The first bit on the left is the *sign* bit, S, the next eight bits are the *exponent* bits, E, and the final 23 bits are the *fraction* (called also *mantissa*) F:

```
1       8                    23
S   EEEEEEEE   FFFFFFFFFFFFFFFFFFFFFFF
^   ^        ^ ^                      ^
31  30       23 22                     0
```

The value V represented by the word may be determined as follows:

- If $E = 255$ and F is nonzero, then $V =$ NaN ("Not a number")
- If $E = 255$ and F is zero and S is 1, then $V = -\infty$
- If $E = 255$ and F is zero and S is 0, then $V = \infty$
- If $0 < E < 255$ then $V = (-1)^S 2^{E-127} 1.F$ where $1.F$ is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
- If $E = 0$ and F is nonzero, then $V = (-1)^S 2^{-126} 0.F$ These are sub-normal ("unnormalized") values. **tudi 'denormal'**
- If $E = 0$ and F is zero and S is 1, then $V = -0$
- If $E = 0$ and F is zero and S is 0, then $V = 0$

IEEE bolj natančno

In particular,

```
0 00000000 00000000000000000000000 = 0
1 00000000 00000000000000000000000 = -0

0 11111111 00000000000000000000000 = Infinity
1 11111111 00000000000000000000000 = -Infinity

0 11111111 00000100000000000000000 = NaN
1 11111111 00100010001001010101010 = NaN
```

```
0 10000000 00000000000000000000000 = +1 * 2**(128-127) * 1.0 = 2
0 10000001 10100000000000000000000 = +1 * 2**(129-127) * 1.101 = 6.5
1 10000001 10100000000000000000000 = -1 * 2**(129-127) * 1.101 = -6.5
```

$(-1)^S 2^{E-127} 1.F$

Sub-normal (unnormalized) tudi denormal

```
0 00000001 00000000000000000000000 = +1 * 2**(1-127) * 1.0 = 2**(-126)
0 00000000 10000000000000000000000 = +1 * 2**(-126) * 0.1 = 2**(-127)
0 00000000 00000000000000000000001 = +1 * 2**(-126) *
                                     0.00000000000000000000001 =
                                     2**(-149)  (Smallest positive value)
```

$(-1)^S 2^{-126} 0.F$

Natančnost zapisa

- Zapis točnih decimalk v binarni bazi problem:
  - **zato v Python-u paket ‘decimal’ itd…**

What do you mean ‘correct’?

Before we can continue I need to make clear the difference between 0.1, float(0.1), and double(0.1). In C/C++ the numbers 0.1 and double(0.1) are the same thing, but when I say “0.1” in text I mean the exact base-10 number, whereas float(0.1) and double(0.1) are rounded versions of 0.1. And, to be clear, float(0.1) and double(0.1) don’t have the same value, because float(0.1) has fewer binary digits, and therefore has more error. Here are the exact values for 0.1, float(0.1), and double(0.1):

Number	Value
0.1	0.1 (of course)
float(.1)	0.100000001490116119384765625
double(.1)	0.1000000000000000055511151231257827021181583404541015625

`f` becomes 0.100000001490116119384765625, the closest 32-bit float value to 0.1.

Floating-point Formats

Several different representations of real numbers have been proposed, but by far the most widely used is the floating-point representation.[1] Floating-point representations have a base β (which is always assumed to be even) and a precision p. If $\beta = 10$ and $p = 3$, then the number 0.1 is represented as 1.00×10^{-1}. If $\beta = 2$ and $p = 24$, then the decimal number 0.1 cannot be represented exactly, but is approximately $1.10011001100110011001101 \times 2^{-4}$.

Natančnost zapisa

- Nekatera števila seveda lahko zapišemo točno.
  - ‘Razdalje’ med njimi so poučne!

```
#include <iostream>
#include <iomanip>
#include <cmath>

void show(float f) {
    std::cout << std::nextafterf(f, 0.0) << "\n"
              << f << "\n"
              << std::nextafterf(f, f*2) << "\n";
    putchar('\n');
}

int main(void) {
    std::cout << std::setprecision(24);

    show(1);
    show(1<<23);
    show(1<<24);
    show(1<<30);
}
```

Smer iskanja

produces this output:

```
0.99999994039535522460937 5
1
1.00000011920928955078125

8388607.5
8388608
8388609

16777215
16777216
16777218

1073741760
1073741824
1073741952
```

It shows the immediate predecessor and successor, in type `float`, of the numbers 1, 2^{23}, 2^{24}, and 2^{30}. As you can see, the gaps get bigger for larger numbers, with the gap doubling in size at each power of 2.

- Razdalja: premik za najmanjši bit v mantisi (*integer representation*)

$$X = (-1)^s 2^E B$$

$$B = 1 + b_1 2^{-1} + b_2 2^{-2} + \cdots + b_{52} 2^{-52}$$

- funkcije **std::nextafter v C++** in **numpy.nextafter v Python-u**

Hexadecimalni zapis bitov za kompaktnost:

Value	Bits	Exponent	Mantissa bits
1.99999988	0x3FFFFFFF	127	0x7FFFFF
2.0	0x40000000	128	0x000000

Natančnost zapisa

- Ne samo, da imamo končno natančnost, ta se spreminja z vrednostjo decimalnega števila!!

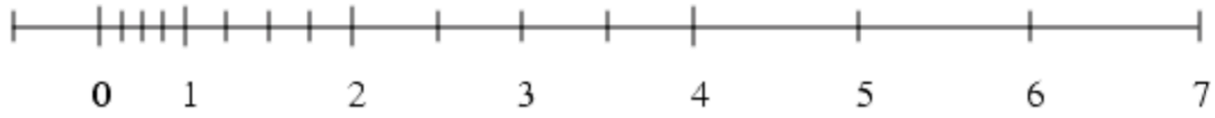
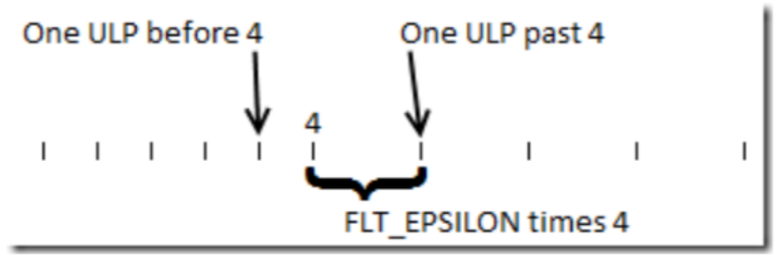


FIGURE D-1 Normalized numbers when $\beta = 2$, $p = 3$, $e_{min} = -1$, $e_{max} = 2$



ULP, he said nervously

ULP - v bistvu število bitov, če upoštevaš se eksp. zapis…

We already know that adjacent floats have integer representations that are adjacent. This means that if we subtract the integer representations of two numbers then the difference tells us how far apart the numbers are in float space. That brings us to:

Dawson's obvious-in-hindsight theorem:

> *If the integer representations of two same-sign floats are subtracted then the absolute value of the result is equal to one plus the number of representable floats between them.*

In other words, if you subtract the integer representations and get one, then the two floats are as close as they can be without being equal. If you get two then they are still really close, with just one float between them. The difference between the integer representations tells us how many Units in the Last Place the numbers differ by. This is usually shortened to ULP, as in "these two floats differ by two ULPs."

One ULPs difference good (adjacent floats).

One million ULPs difference bad (kinda different).

Even a ridiculously small number, like 1e-30 has an 'large' integer representation of 228,737,632. So, while 1e-30 is 'close' to zero and 'close' to negative 1e-30 it is a gargantuan distance from those numbers in ULPs.

Natančnost zapisa

- Pogosto nas zanima primerjava števil:
  - Preprosta logika iz analitičnih računov: ‘zahtevana natančnost = epsilon’

Epsilon comparisons

If comparing floats for equality is a bad idea then how about checking whether their difference is within some error bounds or epsilon value, like this:

bool isEqual = fabs(f1 – f2) <= epsilon;

With this calculation we can express the concept of two floats being close enough that we want to consider them to be equal. But what value should we use for epsilon?

Given our experimentation above we might be tempted to use the error in our sum, which was about 1.19e-7f. In fact, there’s even a define in float.h with that exact value, and it’s *called* FLT_EPSILON.

Clearly that’s it. The header file gods have spoken and FLT_EPSILON is the one true epsilon!

Except that that is rubbish. For numbers between 1.0 and 2.0 FLT_EPSILON represents the difference between adjacent floats. For numbers smaller than 1.0 an epsilon of FLT_EPSILON quickly becomes too large, and with small enough numbers FLT_EPSILON may be bigger than the numbers you are comparing (a variant of this led to a flaky Chromium test)!

Res natančni…

- Postane kompliciran:
  - dodatno: rel. ali abs. napaka?

```
// Absolute tolerance comparison of x and y
if (Abs(x - y) <= EPSILON) …
```

and

```
// Relative tolerance comparison of x and y
if (Abs(x - y) <= EPSILON * Max(Abs(x), Abs(y)) …
```

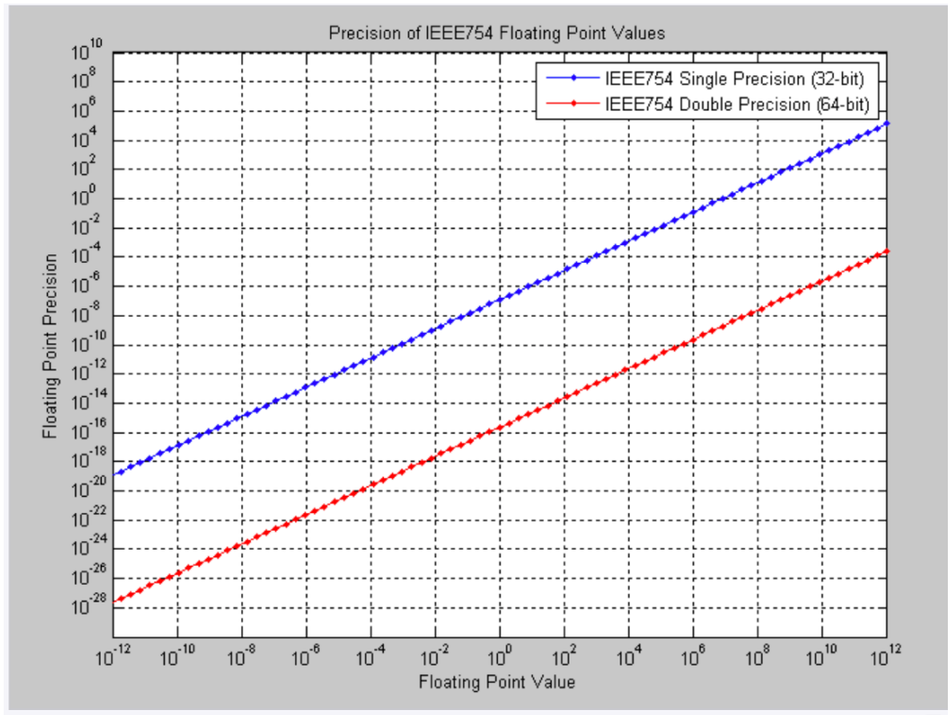
The absolute tolerance test fails when `x` and `y` become large, and the relative tolerance test fails when they become small. It is therefore desired to combine these two tests together in a single test. Over the years at GDC, as well as in my book, I've suggested the following combined tolerance test:

```
if (Abs(x - y) <= EPSILON * Max(1.0f, Abs(x), Abs(y)) …
```

```
bool nearlyEqual(float a, float b,
        float fixedEpsilon, int ulpsEpsilon)
{
    // Handle the near-zero case.
    const float difference = fabs(a - b);
    if (difference <= fixedEpsilon) return true;

    return ulpsDistance(a, b) <= ulpsEpsilon;
}
```

- `FLT_EPSILON`… isn't float epsilon, except in the ranges [-2, -1] and [1, 2]. The distance between adjacent values depends on the values in question.
- When comparing to some known value—especially zero or values near it—use a fixed ϵ that makes sense for your calculations.
- When comparing non-zero values, some ULPs-based comparison is probably the best choice.
- When values could be anywhere on the number line, some hybrid of the two is needed. Choose epsilons carefully based on expected outputs.



Precision of binary32 and binary64 in the range 10^{-12} to 10^{12}.

Računske operacije

- Računske operacije in zapis funkcij postanejo kompliciran**i!**
  - **Hard to get it right!**

Guard Digits

and then rounded. Take another example: 10.1 - 9.93. This becomes

$x = 1.01 \times 10^1$
$y = 0.99 \times 10^1$
$x - y = .02 \times 10^1$

The correct answer is .17, so the computed difference is off by 30 ulps and wrong in every digit! How bad can the error be?

digits. With a guard digit, the previous example becomes

$x = 1.010 \times 10^1$
$y = 0.993 \times 10^1$
$x - y = .017 \times 10^1$

and the answer is exact. With a single guard digit, the relative error of the

Calculating Elementary Functions is not easy

TABLE V. $\sin(10^{22})$ AND $\cos(10^{22})$ ON VARIOUS SYSTEMS

Computer	$\sin(10^{22})$	$\cos(10^{22})$
Maple 8 (15 digs)	$-0.852200849\ldots$	$+0.523214785\ldots$
Maxima 5.9 (bfloat)	$-0.852200849\ldots$	$+0.523214785\ldots$
Matlab 6.5 (15 digs)	$-0.852200849\ldots$	$+0.523214785\ldots$
O-Matrix 5.5(e format)	$+0.226946577\ldots$	$-0.973907232\ldots$
O-Matrix 5.5(d format)	$+0.412143367\ldots$	$-0.911119007\ldots$
Scilab 3.0 (15 digs)	$+10^{22}$	$+10^{22}$
DVF 5.0 D (sp)	$+0.2269466\ldots$	$-0.9739072\ldots$
DVF 5.0 D (dp)	$+0.412143367\ldots$	$-0.911119007\ldots$
Intel Fortran 8 (sp)	$+9.9999998 \times 10^{21}$	$+9.9999998 \times 10^{21}$
Intel Fortran 8 (dp)	$+10^{22}$	$+10^{22}$
Intel Fortran 8 (ep)	$-0.852200849\ldots$	$+0.523214785\ldots$
Watfor 11.2 (sp)	$+0.2812271\ldots$	$-0.9596413\ldots$
Watfor 11.2 (dp)	$+0.4626130\ldots$	$-0.8865603\ldots$
TMT Pascal(all precs)	$+0.0$	$+0.0$
FranzLisp	$+0.2269465\ldots$	$-0.9739072\ldots$
Sharp EL-531VH	error2	error2
MS Windows Calc (32 digs)	$-0.852200849\ldots$	$+0.523214785\ldots$
PariGP 2.2.7 (28 digs)	$-0.852200849\ldots$	$+0.523214785\ldots$
Correct Answer	$-0.852200849\ldots$	$+0.523214785\ldots$

so can be safely ignored.[19] As noted by Kahan, the unhandled trap consecutive to a floating-point to 16-bit integer conversion overflow that caused the loss of an Ariane 5 rocket would not have happened under the default IEEE 754 floating-point policy.[18]

Lekcije

- Za fizike dovolj, če se zavedamo, da problem obstaja…
  - Vodilo:
    - Single precision, 32 bit (float): 7 mest!
    - Double precision, 64 bit: 16 mest!
    - Pazi, ko uporabljaš zahteve za enakost, epsilon natančnost ipd.
- Za nalogo preišči natančnost Airy-evih funkcij….

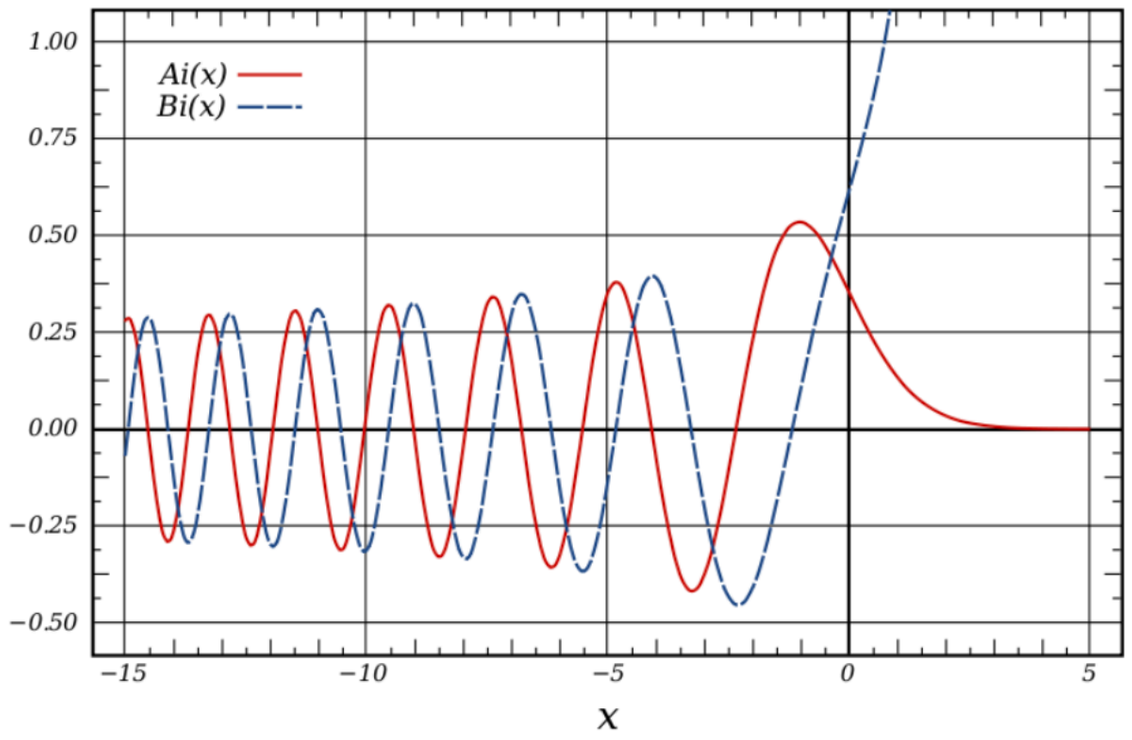
Naloga: Airijevi funkciji

- Cilj naloge je čim bolj natančno sprogramirati t.i. Airijevi funkciji:
  - LN rešitvi nehomogene DE:

$$\frac{d^2y}{dx^2} - xy = 0,$$

Airyjeva ali Stokesova DE

  - Npr. Stac. Schrödingerjeva enačba v **trikotnem** potencialu.
  - Rešitvi označimo z Ai(x) in Bi(x)



- **Očitno sta funkciji monotoni pri x>0 in oscilirata za x < 0, pri x→∞ gre Ai(x) proti 0 in Bi(x) divergira.**
- Rešitvi sta zapisani v integralski obliki:

$$\mathrm{Ai}(x) = \frac{1}{\pi}\int_0^\infty \cos(t^3/3 + xt)\,\mathrm{d}t\,, \quad \mathrm{Bi}(x) = \frac{1}{\pi}\int_0^\infty \left[\mathrm{e}^{-t^3/3+xt} + \sin(t^3/3 + xt)\right]\,\mathrm{d}t$$

- Dandanes tudi že znamo izračunati tovrstni integral z uporabo metod numerične integracije : **opcionalno:** uporabi ustrezno integracijsko formulo in poglej, kako natančen rezultat lahko dosežeš v razumnem času..

Naloga: Airijevi funkciji

- **Obstajajo pa tudi druge metode…**

- Že dobro znani pristop je uporaba potenčne (Taylorjeve) vrste, vendar je to dobro samo za **majhne vrednosti parametra x!**
  - Maclaurinova vrsta (Taylor okrog ničle…):

$$\mathrm{Ai}(x) = \alpha f(x) - \beta g(x)\,, \qquad \mathrm{Bi}(x) = \sqrt{3}\left[\alpha f(x) + \beta g(x)\right],$$

$$\alpha = \mathrm{Ai}(0) = \mathrm{Bi}(0)/\sqrt{3} \approx 0.355028053887817239,$$
$$\beta = -\mathrm{Ai}'(0) = \mathrm{Bi}'(0)/\sqrt{3} \approx 0.258819403792806798.$$

$$f(x) = \sum_{k=0}^{\infty} \left(\frac{1}{3}\right)_k \frac{3^k x^{3k}}{(3k)!}\,, \qquad g(x) = \sum_{k=0}^{\infty} \left(\frac{2}{3}\right)_k \frac{3^k x^{3k+1}}{(3k+1)!}\,,$$

$$(z)_n = \Gamma(z+n)/\Gamma(z)\,, \qquad (z)_0 = 1\,.$$

- … v zgornjih izrazih **razmisli,** kako boš zapisal člene!
  - Optimalno izrazi člen s prejšnjim (rekurzivno) … (poskusi sam!)

Naloga: Airijevi funkciji

- Na tem mestu se lahko spomnimo še ene metode, in sicer **asimptotske vrste:**
  - metoda je dobila to ime, ker velja kot dober približek **samo za velike vrednosti argumenta z!**

- Splošen zapis asimptotske vrste (ta posebni enačaj označuje asimptotsko vedenje):

$$f(z) \asymp \sum_{n=0}^{N} F_n(z)$$

- Členi *F* v vsoti predstavljajo popravke k funkciji. Posebnost asimptotske vrste, v primerjavi s potenčno vrsto pa je, da:
  - moramo vedno vzeti **končno število teh členov!**
  - Število členov, ki jih lahko vzamemo, je odvisno od velikosti argumenta z: **večji ko je argument, več členov lahko vzamemo in dosežemo boljšo natančnost…**
  - V praksi to vidimo tako, da **začnejo po nekem N členi znova naraščati** (in tako prekinemo prištevanje le-teh): Členi so tipično oblike:

$$\frac{n!}{z^n}$$

Primer asimptotike: Gaussov integral

- Poglejmo, kako pridemo do asimptotske vrste v praksi, za naš zgled vzemimo erf(z):

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \exp(-t^2)\, dt.$$

  - Uporabimo erfc(z) =1-erf(z), zamenjamo spremenljivko:

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty \exp(-t^2)\, dt = \frac{2}{\sqrt{\pi}} \frac{1}{2} \int_{z^2}^\infty \exp(-u)\, u^{-\frac{1}{2}}\, du = \frac{1}{\sqrt{\pi}} \int_{z^2}^\infty \exp(-u)\, u^{-\frac{1}{2}}\, du$$

- …in računamo *per partes*:

$$\frac{1}{\sqrt{\pi}} \int_{z^2}^\infty \exp(-u)\, u^{-\frac{1}{2}}\, du = \frac{1}{\sqrt{\pi}} \left[(-\exp(-u))\, u^{-\frac{1}{2}} \Big|_{z^2}^\infty - \int_{z^2}^\infty (-\exp(-u))\, (-\frac{1}{2} u^{-\frac{3}{2}})\, du \right]$$

$$= \frac{1}{\sqrt{\pi}} \left[\frac{\exp(-z^2)}{z} - \frac{1}{2} \int_{z^2}^\infty \exp(-u)\, u^{-\frac{3}{2}}\, du \right]$$

- Hitro vidimo, da se postopek lahko ponavlja in tako dobimo:

$$erfc(z) \asymp \frac{1}{\sqrt{\pi}} \frac{\exp(-z^2)}{z} \left[1 - \frac{1}{2z^2} + \frac{1 \cdot 3}{2^2 z^4} - \frac{1 \cdot 3 \cdot 5}{2^3 z^6} + \ldots + \frac{(2N-1)!!}{(-2)^N z^{2N}} \right]$$

Primer asimptotike: Gaussov integral

- Očitno je, da členi pri nekem N začnejo znova naraščati!

$$erfc(z) \asymp \frac{1}{\sqrt{\pi}} \frac{\exp(-z^2)}{z} \left[1 - \frac{1}{2z^2} + \frac{1 \cdot 3}{2^2 z^4} - \frac{1 \cdot 3 \cdot 5}{2^3 z^6} + \cdots + \frac{(2N-1)!!}{(-2)^N z^{2N}} \right]$$

- Pogosta je malo preurejena formula (ki pa je očitno enaka):

$$z\sqrt{\pi}\, \exp(z^2)\, \mathrm{erfc}(z) \longrightarrow 1 + \sum_{m=1}^{\infty} \frac{(2m-1)!!}{(-2z^2)^m}$$

Asimptotika za Airijevi funkciji

- Tu je asimptotski zapis malce bolj kompliciran, je pa lepo ‘modularen’:
  - Zapišimo tri generične asimptotske vrste:

$$L(z) \sim \sum_{s=0}^{\infty} \frac{u_s}{z^s}\,, \qquad P(z) \sim \sum_{s=0}^{\infty} (-1)^s \frac{u_{2s}}{z^{2s}}\,, \qquad Q(z) \sim \sum_{s=0}^{\infty} (-1)^s \frac{u_{2s+1}}{z^{2s+1}}\,,$$

$$u_s = \frac{\Gamma(3s+\frac{1}{2})}{54^s s!\,\Gamma(s+\frac{1}{2})}$$

Zopet razmisli, kako bi pisal zaporedne člene!

- Vpeljemo še novo spremenljivko: $\xi = \frac{2}{3}|x|^{3/2}$, pa dobimo:
- Za (velike) pozitivne vrednosti x:

$$\mathrm{Ai}(x) \sim \frac{\mathrm{e}^{-\xi}}{2\sqrt{\pi}x^{1/4}}\,L(-\xi)\,, \qquad \mathrm{Bi}(x) \sim \frac{\mathrm{e}^{\xi}}{\sqrt{\pi}x^{1/4}}\,L(\xi)\,,$$

- Za negativne (in absolutno velike) vrednosti x:

$$\mathrm{Ai}(x) \sim \frac{1}{\sqrt{\pi}(-x)^{1/4}}\Big[\ \ \sin(\xi-\pi/4)\,Q(\xi) + \cos(\xi-\pi/4)\,P(\xi)\Big]\,,$$
$$\mathrm{Bi}(x) \sim \frac{1}{\sqrt{\pi}(-x)^{1/4}}\Big[-\sin(\xi-\pi/4)\,P(\xi) + \cos(\xi-\pi/4)\,Q(\xi)\Big]\,.$$

Cilj naloge

- Cilj naloge je čim bolje zlepiti oba približka (in še kaj, če hočete…), z vizijo:
  - Da je **relativna napaka** najmanjša (kaj je dosegljivo? 10^{-10}) …
  - Da je **absolutna napaka** najmanjša (kaj je dosegljivo? 10^{-10}) …
- Pomagajte si z orodji, ki znajo računati s poljubno natančnostjo (Python paketi decimal, mpmath, Mathematica, če jo prepričate …).
- Toplo vabljeni tudi, da si pogledate dodatno nalogo, kjer poiščete še prvih sto ničel Ai(x) in Bi(x) in primerjate s približnimi formulami…

Dodatna naloga: Ničle funkcije Ai pogosto srečamo v matematični analizi pri določitvi intervalov ničel specialnih funkcij in ortogonalnih polinomov [2] ter v fiziki pri računu energijskih spektrov kvantno-mehanskih sistemov [3]. Poišči prvih sto ničel $\{a_s\}_{s=1}^{100}$ Airyjeve funkcije Ai in prvih sto ničel $\{b_s\}_{s=1}^{100}$ funkcije Bi pri $x < 0$ ter dobljene vrednosti primerjaj s formulama

$$a_s = -f\left(\frac{3\pi(4s-1)}{8}\right), \qquad b_s = -f\left(\frac{3\pi(4s-3)}{8}\right), \qquad s = 1, 2, \dots,$$

kjer ima funkcija f asimptotski razvoj [4]

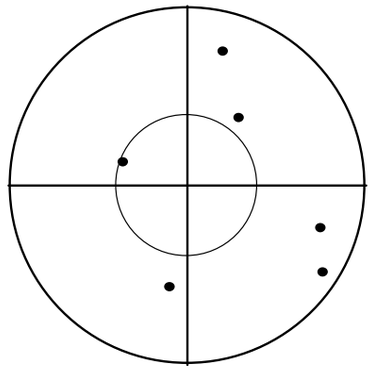
$$f(z) \sim z^{2/3}\left(1 + \frac{5}{48}\,z^{-2} - \frac{5}{36}\,z^{-4} + \frac{77125}{82944}\,z^{-6} - \frac{108056875}{6967296}\,z^{-8} + \dots\right).$$

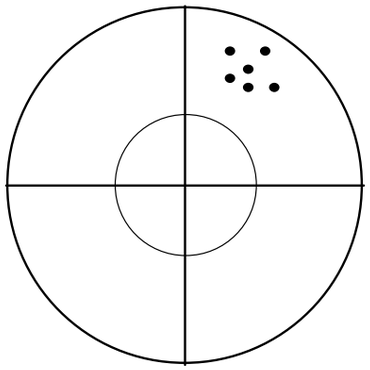
Pomembna razlika!

Accuracy vs Precision

In numerical algorithms a very important distinction is made.

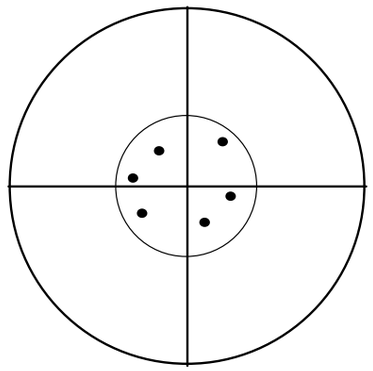
Low Accuracy -- Low Precision

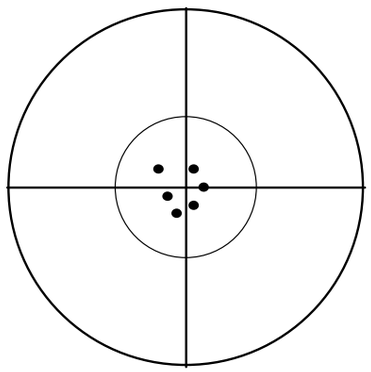




Low Accuracy -- High Precision

High Accuracy -- Low Precision





High Accuracy -- High Precision

[the electricity companies] calculate that it would cost them a further £100m, a sum which is hardly more than a rounding error in the mathematics of power supply.

—*Daily Telegraph*, Feb. 16 2005.