

Computer arithmetic

Victor Eijkhout

Fall 2023



This short session will explain the basics of floating point arithmetic, mostly focusing on round-off and its influence on computations.



- Integers: $\dots, -2, -1, 0, 1, 2, \dots$
- Rational numbers: $1/3, 22/7$: not often encountered
- Real numbers $0, 1, -1.5, 2/3, \sqrt{2}, \log 10, \dots$
- Complex numbers $1 + 2i, \sqrt{3} - \sqrt{5}i, \dots$

Computers use a finite number of bits to represent numbers, so only a finite number of numbers can be represented, and no irrational numbers (even some rational numbers).



First we dig into bits



	boolean	bitwise (C)	bitwise (F)	bitwise (Py)
and	<code>&&</code>	<code>&</code>	<code>iand</code>	<code>&</code>
or	<code> </code>	<code> </code>	<code>ior</code>	<code> </code>
not	<code>!</code>			<code>~</code>
xor		<code>^</code>	<code>ieor</code>	

Bit shift operations in C:

left shift `<<`

right shift `>>`

Fortran: `mvbits`



- Left-shift is multiplication by 2:

`i_times_2 = i<<1;`

- Extract bits:

`i_mod_8 = i & 7`

(How does that last one work?)



Use bit operations to test whether a number is odd or even.
Can you think of more than one way?



Integers



Scientific computation mostly uses real numbers. Integers are mostly used for array indexing.

We look at

1. integers as supported by the hardware;
2. integers as they exist in programming languages;
3. (and not software-defined integers)



C:

- A short int is at least 16 bits;
- An integer is at least 16 bits, but often 32 bits;
- A long integer is at least 32 bits, but often 64;
- A long long integer is at least 64 bits.

Fortran uses kinds, not necessarily equal to number of bytes:

```
integer(2)  :: i2  
integer(4)  :: i4  
integer(8)  :: i8
```

Specify the number of decimal digits with `selected_int_kind(n)`.



Print 2^n for $n = 0, \dots, 31$. There are at least two ways of generating these powers.

Also print the bit pattern. What is unexpected?



Problem:

- How do we represent them?
- How do we do efficient arithmetic on them?

Define

$$\text{rep}: \mathbb{Z} \rightarrow 2^n$$

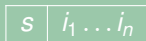
‘representation of the number $N \in \mathbb{Z}$ as bitstring of length n .’

$$\text{int}: 2^n \rightarrow \mathbb{Z}$$

‘interpretation of the bitstring of length n as number $N \in \mathbb{Z}$ ’



Use of sign bit: typically first bit



Simplest solution:

$$\begin{cases} n \geq 0 & \text{rep}(n) = 0, i_1, \dots, i_{31} \\ n < 0 & \text{rep}(-n) = 1, i_1, \dots, i_{31} \end{cases}$$



Interpretation

bitstring	00...0	...	01...1	10...0	...	11...1
as unsigned int	0	...	$2^{31} - 1$	2^{31}	...	$2^{32} - 1$
as naive signed	0	...	$2^{31} - 1$	-0	...	$-2^{31} + 1$



Interpret unsigned number n as $n - B$

bitstring	$00 \dots 0$	\dots	$01 \dots 1$	$10 \dots 0$	\dots	$11 \dots 1$
as unsigned int	0	\dots	$2^{31} - 1$	2^{31}	\dots	$2^{32} - 1$
as shifted int	-2^{31}	\dots	-1	0	\dots	$2^{31} - 1$



Let m be a signed integer, then the 2's complement 'bit pattern' $\text{rep}(m)$ is a non-negative integer defined as follows:

- If $0 \leq m \leq 2^{31} - 1$, the normal bit pattern for m is used, that is

$$0 \leq m \leq 2^{31} - 1 \Rightarrow \text{rep}(m) = m.$$

- For $-2^{31} \leq n \leq -1$, n is represented by the bit pattern for $2^{32} - |n|$:

$$-2^{31} \leq n \leq -1 \Rightarrow \text{rep}(m) = 2^{32} - |n|.$$



bitstring	$00 \dots 0$	\dots	$01 \dots 1$	$10 \dots 0$	\dots	$11 \dots 1$
as unsigned int	0	\dots	$2^{31} - 1$	2^{31}	\dots	$2^{32} - 1$
as 2's comp. integer	0	\dots	$2^{31} - 1$	-2^{31}	\dots	-1



Problem: processor is very good at arithmetic on (unsigned) bit strings.

How does that translate to arithmetic on integers?

$$\text{int}(\text{rep}(x) * \text{rep}(y)) \stackrel{?}{=} x * y$$



Add $m + n$, where m, n are representable:

$$0 \leq |m|, |n| < 2^{31}.$$

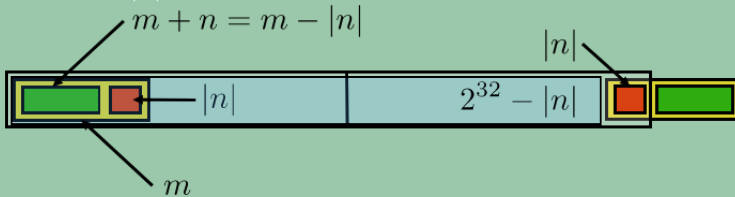
The easy case is $0 < m, n$, as long as there is no overflow.



Case $m > 0$, $n < 0$, and $m + n > 0$. Then $\text{rep}(m) = m$ and $\text{rep}(n) = 2^{32} - |n|$, so the unsigned addition becomes

$$\text{rep}(m) + \text{rep}(n) = m + (2^{32} - |n|) = 2^{32} + m - |n|.$$

Since $m - |n| > 0$, this result is $> 2^{32}$.



However, this is basically $m + n$ with the overflow bit set.



Subtraction $m - n$:

- Case: $m < n$. Observe that $-n$ has the bit pattern of $2^{32} - n$. Also, $m + (2^{32} - n) = 2^{32} - (n - m)$ where $0 < n - m < 2^{31} - 1$, so $2^{32} - (n - m)$ is the 2's complement bit pattern of $m - n$.
- Case: $m > n$. The bit pattern for $-n$ is $2^{32} - n$, so $m + (-n)$ as unsigned is $m + 2^{32} - n = 2^{32} + (m - n)$. Here $m - n > 0$. The 2^{32} is an overflow bit; ignore.



There is a limited number of bits, so numbers that are too large in absolute value can not be represented.

Overflow.

This is not a fatal error: your program continues with the wrong result.



Investigate what happens when you perform an integer calculation that leads to overflow. What does your compiler say if you try to write down a nonrepresentable number explicitly, for instance in a declaration or assignment statement?

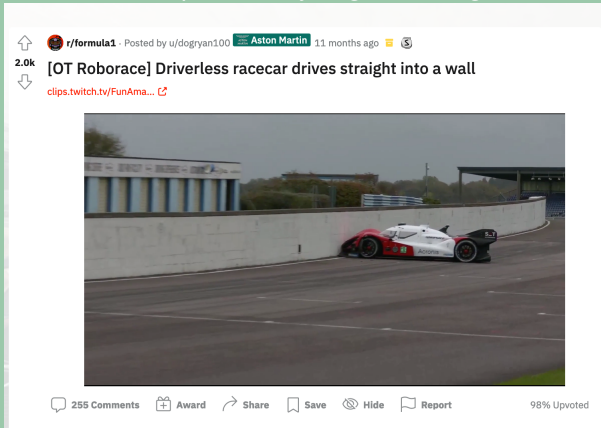
Language lawyer remark: signed integer overflow is Undefined Behavior in C/C++.



Floating point numbers



And the consequences if you get it wrong can be considerable.



Analogous to scientific notation $x = 6.022 \cdot 10^{23}$:

$$x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$$

- sign bit
- β is the base of the number system
- $0 \leq d_i \leq \beta - 1$ the digits of the *mantissa*:
one digit before the *radix point*, so mantissa $< \beta$
- $e \in [L, U]$ exponent, stored with bias: unsigned int where $\text{fl}(L) = 0$



	β	t	L	U
IEEE single (32 bit)	2	23	-126	127
IEEE double (64 bit)	2	53	-1022	1023
Old Cray 64bit	2	48	-16383	16384
IBM mainframe 32 bit	16	6	-64	63
packed decimal	10	50	-999	999

BCD is tricky: 3 decimal digits in 10 bits

(we will often use $\beta = 10$ in the examples, because it's easier to read for humans, but all practical computers use $\beta = 2$)

Internal processing in 80 bit



Overflow: more than $\beta(1 - \beta^{-t+1})\beta^U$ or less than $-\beta(1 - \beta^{-t+1})\beta^U$

Underflow: positive numbers less than β^L

Gradual underflow: $\beta^{-t+1} \cdot \beta^L$

Overflow leads to Inf .



For real numbers x, y , the quantity $g = \sqrt{(x^2 + y^2)/2}$ satisfies

$$g \leq \max\{|x|, |y|\}$$

so it is representable if x and y are. What can go wrong if you compute g using the above formula? Can you think of a better way?



Overflow: Inf

$\text{Inf} - \text{Inf} \rightarrow \text{NaN}$

also $0/0$ or $\sqrt{-1}$

This does not stop your program in general
sometimes possible



Do we allow

$$1.100 \cdot 10^0, \quad 0.110 \cdot 10^1, \quad 0.011 \cdot 10^2?$$

This makes testing for equality hard.

Solution: normalized numbers have one nonzero before the radix point.



Require first digit in the mantissa to be nonzero.

Equivalent: mantissa part $1 \leq x_m < \beta$

Unique representation for each number,

also: in binary this makes the first digit 1, so we don't need to store that.

(do you see a problem?)

With normalized numbers, underflow threshold is $1 \cdot \beta^L$;

'gradual underflow' possible, but usually not efficient.



sign	exponent	mantissa
p	$e = e_1 \cdots e_8$	$s = s_1 \cdots s_{23}$
31	$30 \cdots 23$	$22 \cdots 0$
\pm	2^{e-127} (except $e = 0, 255$)	$s_1 \cdot 2^{-1} + \cdots + s_{23} \cdot 2^{-23}$



$(e_1 \dots e_8)$	numerical value	range
$(0 \dots 0) = 0$	$\pm 0.s_1 \dots s_{23} \times 2^{-126}$	$s = 0 \dots 01 \Rightarrow 2^{-23} \cdot 2^{-126} = 2^{-149} \approx 10^{-45}$ $s = 1 \dots 11 \Rightarrow (1 - 2^{-23}) \cdot 2^{-126}$
$(0 \dots 01) = 1$	$\pm 1.s_1 \dots s_{23} \times 2^{-126}$	$s = 0 \dots 01 \Rightarrow 1 \cdot 2^{-126} \approx 10^{-37}$
$(0 \dots 010) = 2$	$\pm 1.s_1 \dots s_{23} \times 2^{-125}$	
...		
$(01111111) = 127$	$\pm 1.s_1 \dots s_{23} \times 2^0$	$s = 0 \dots 00 \Rightarrow 1 \cdot 2^0 = 1$ $s = 0 \dots 01 \Rightarrow 1 + 2^{-23} \cdot 2^0 = 1 + \epsilon$ $s = 1 \dots 11 \Rightarrow (2 - 2^{-23}) \cdot 2^0 = 2 - \epsilon$
$(10000000) = 128$	$\pm 1.s_1 \dots s_{23} \times 2^1$	$s = 0 \dots 00 \Rightarrow 1 \cdot 2^1 = 2$
...		et cetera
$(11111110) = 254$	$\pm 1.s_1 \dots s_{23} \times 2^{127}$	
$(11111111) = 255$	$s_1 \dots s_{23} = 0 \Rightarrow \pm \infty$ $s_1 \dots s_{23} \neq 0 \Rightarrow \text{NaN}$	



Note that the exponent doesn't come at the end. This has an interesting consequence.

What is the interpretation of

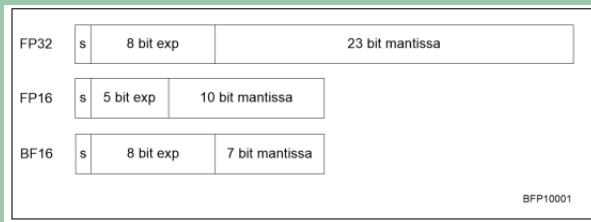
$0 \dots 0111$

as int? What as float?

What is the largest integer that is representible as float?



- There is a 64-bit format, with 53 bits mantissa.
- IEEE envisioned a sliding scale of precisions: see Intel 80-bit registers
- Half precision, and recent invention `bfloat16`



Floating point math



Error between number x and representation \tilde{x} :

absolute $x - \tilde{x}$ or $|x - \tilde{x}|$

relative $\frac{x - \tilde{x}}{x}$ or $\left| \frac{x - \tilde{x}}{x} \right|$

Equivalent: $\tilde{x} = x \pm \epsilon \Leftrightarrow |x - \tilde{x}| \leq \epsilon \Leftrightarrow \tilde{x} \in [x - \epsilon, x + \epsilon]$.

Also: $\tilde{x} = x(1 + \epsilon)$ often shorthand for $\left| \frac{\tilde{x} - x}{x} \right| \leq \epsilon$



Decimal, $t = 3$ digit mantissa: let $x = 1.256$, $\tilde{x}_{\text{round}} = 1.26$,
 $\tilde{x}_{\text{truncate}} = 1.25$

Error in the 4th digit.

Different story for decimal vs binary.

How would this story change with a non-zero exponent,
for instance $1.256 \cdot 10^{12}$?



The number $e \approx 2.72$, the base for the natural logarithm, has various definitions. One of them is

$$e = \lim_{n \rightarrow \infty} (1 + 1/n)^n. \quad (1)$$

Write a single precision program that tries to compute e in this manner. (Do not use the `pow` function: code the power explicitly.) Evaluate the expression for an upper bound $n = 10^k$ for some k . (How far do you let k range?) Explain the output for large n . Comment on the behavior of the error.



Any real number can be represented to a certain precision:

$\tilde{x} = x(1 + \varepsilon)$ where

truncation: $\varepsilon = \beta^{-t+1}$

rounding: $\varepsilon = \frac{1}{2}\beta^{-t+1}$

This is called *machine precision*: maximum relative error.

32-bit single precision: $mp \approx 10^{-7}$

64-bit double precision: $mp \approx 10^{-16}$

Maximum attainable accuracy.

Another definition of machine precision: smallest number ε such that $1 + \varepsilon > 1$.



Write a small program that computes the machine epsilon for both single and double precision. Does it make any difference if you set the compiler optimization levels low or high?
(For C++ programmers: can you write a templated program that works for single and double precision?)



1. align exponents
2. add mantissas
3. adjust exponent to normalize

Example: $1.00 + 2.00 \times 10^{-2} = 1.00 + .02 = 1.02$. This is exact, but what happens with $1.00 + 2.55 \times 10^{-2}$?

Example: $5.00 \times 10^1 + 5.04 = (5.00 + 0.504) \times 10^1 \rightarrow 5.50 \times 10^1$

Any error comes from limiting the mantissa: if x is the true sum and \tilde{x} the computed sum, then $\tilde{x} = x(1 + \varepsilon)$ with $|\varepsilon| < 10^{-2}$



Assumption (enforced by IEEE 754):

The numerical result of an operation is the rounding of the exactly computed result.

$$\text{fl}(x_1 \odot x_2) = (x_1 \odot x_2)(1 + \varepsilon)$$

where $\odot = +, -, *, /$

Note: this holds only for a single operation!



Correctly rounding is not trivial, especially for subtraction.

Example: $t = 2, \beta = 10$: $1.0 - 9.5 \times 10^{-1}$, exact result $0.05 = 5.0 \times 10^{-2}$.

- Simple approach:

$$1.0 - 9.5 \times 10^{-1} = 1.0 - 0.9 = 0.1 = 1.0 \times 10^{-1}$$

- Using 'guard digit':

$$1.0 - 9.5 \times 10^{-1} = 1.0 - 0.95 = 0.05 = 5.0 \times 10^{-2}, \text{ exact.}$$

In general 3 extra bits needed.



(also ‘fused multiply-accumulate’)

$$c \leftarrow a * b + c$$

- Addition plus multiplication, but not independent
- Processors can have dedicated hardware for FMA (also IEEE 754-2008)
- Internally evaluated in higher precision: 80-bit.
- Very useful for certain linear algebra (which?) Not for other operations (examples?)



Compute $4 + 6 + 7$ in one significant digit.

Evaluation left-to-right gives:

$$\begin{aligned}(4 \cdot 10^0 + 6 \cdot 10^0) + 7 \cdot 10^0 &\Rightarrow 10 \cdot 10^0 + 7 \cdot 10^0 && \text{addition} \\ &\Rightarrow 1 \cdot 10^1 + 7 \cdot 10^0 && \text{rounding} \\ &\Rightarrow 1.0 \cdot 10^1 + 0.7 \cdot 10^1 && \text{using guard digit} \\ &\Rightarrow 1.7 \cdot 10^1 \\ &\Rightarrow 2 \cdot 10^1 && \text{rounding}\end{aligned}$$

On the other hand, evaluation right-to-left gives:

$$\begin{aligned}4 \cdot 10^0 + (6 \cdot 10^0 + 7 \cdot 10^0) &\Rightarrow 4 \cdot 10^0 + 13 \cdot 10^0 && \text{addition} \\ &\Rightarrow 4 \cdot 10^0 + 1 \cdot 10^1 && \text{rounding} \\ &\Rightarrow 0.4 \cdot 10^1 + 1.0 \cdot 10^1 && \text{using guard digit} \\ &\Rightarrow 1.4 \cdot 10^1 \\ &\Rightarrow 1 \cdot 10^1 && \text{rounding}\end{aligned}$$



Let $s = x_1 + x_2$, and $x = \tilde{s} = \tilde{x}_1 + \tilde{x}_2$ with $\tilde{x}_i = x_i(1 + \varepsilon_i)$

$$\begin{aligned}\tilde{x} &= \tilde{s}(1 + \varepsilon_3) \\ &= x_1(1 + \varepsilon_1)(1 + \varepsilon_3) + x_2(1 + \varepsilon_2)(1 + \varepsilon_3) \\ &= x_1 + x_2 + x_1(\varepsilon_1 + \varepsilon_3) + x_2(\varepsilon_2 + \varepsilon_3) \\ \Rightarrow \tilde{x} &= s(1 + 2\varepsilon)\end{aligned}$$

\Rightarrow errors are added

Assumptions: all ε_i approximately equal size and small;

$x_i > 0$



1. add exponents
2. multiply mantissas
3. adjust exponent

Example:

$$.123 \times .567 \times 10^1 = .069741 \times 10^1 \rightarrow .69741 \times 10^0 \rightarrow .697 \times 10^0.$$

What happens with relative errors?



Examples



Correct rounding only applies to a single operation.

Example: $1.24 - 1.23 = 0.01 \rightarrow 1. \times 10^{-2}$:

result is exact, but only one significant digit.

What if $1.24 = \text{fl}(1.244)$ and $1.23 = \text{fl}(1.225)$? Correct result 1.9×10^{-2} ; almost 100% error.

- *Cancellation* leads to loss of precision
- subsequent operations with this result are inaccurate
- this can not be fixed with guard digits and such
- \Rightarrow avoid subtracting numbers that are likely close.



Example: $ax^2 + bx + c = 0 \rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

suppose $b > 0$ and $b^2 \gg 4ac$ then the '+' solution will be inaccurate

Better: compute $x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ and use $x_+ \cdot x_- = -c/a$.



Equation

$$f(x) = \varepsilon x^2 - (1 + \varepsilon^2)x + \varepsilon,$$

Roots for ε small:

$$x_+ \approx \varepsilon^{-1}, \quad x_- \approx \varepsilon.$$

Textbook solution for small ε

$$x_- \approx 0, \quad f(x_-) \approx \varepsilon$$

Accurately:

$$f(x_-) \approx \varepsilon^3$$



ε	<i>textbook</i>		<i>accurate</i>	
	x_-	$f(x_-)$	x_-	$f(x_-)$
10^{-3}	$1.000 \cdot 10^{-03}$	$-2.876 \cdot 10^{-14}$	$1.000 \cdot 10^{-03}$	$-2.168 \cdot 10^{-19}$
10^{-4}	$1.000 \cdot 10^{-04}$	$5.264 \cdot 10^{-14}$	$1.000 \cdot 10^{-04}$	0.000
10^{-5}	$1.000 \cdot 10^{-05}$	$-8.274 \cdot 10^{-13}$	$1.000 \cdot 10^{-05}$	$-1.694 \cdot 10^{-21}$
10^{-6}	$1.000 \cdot 10^{-06}$	$-3.339 \cdot 10^{-11}$	$1.000 \cdot 10^{-06}$	$-2.118 \cdot 10^{-22}$
10^{-7}	$9.992 \cdot 10^{-08}$	$7.993 \cdot 10^{-11}$	$1.000 \cdot 10^{-07}$	$1.323 \cdot 10^{-23}$
10^{-8}	$1.110 \cdot 10^{-08}$	$-1.102 \cdot 10^{-09}$	$1.000 \cdot 10^{-08}$	0.000
10^{-9}	0.000	$1.000 \cdot 10^{-09}$	$1.000 \cdot 10^{-09}$	$-2.068 \cdot 10^{-25}$
10^{-10}	0.000	$1.000 \cdot 10^{-10}$	$1.000 \cdot 10^{-10}$	0.000

(2)



Evaluate $\sum_{n=1}^{10000} \frac{1}{n^2} = 1.644834$

in 6 digits: machine precision is 10^{-6} in single precision

First term is 1, so partial sums are ≥ 1 , so $1/n^2 < 10^{-6}$ gets ignored, \Rightarrow last 7000 terms (or more) are ignored, \Rightarrow sum is 1.644725: 4 correct digits

Solution: sum in reverse order; exact result in single precision

Why? Consider ratio of two terms:

$$\frac{n^2}{(n-1)^2} = \frac{n^2}{n^2 - 2n + 1} = \frac{1}{1 - 2/n + 1/n^2} \approx 1 + \frac{2}{n}$$

with aligned exponents:

$$\begin{array}{rcl} n-1: & .00 \cdots 0 & 10 \cdots 00 \\ n: & .00 \cdots 0 & 10 \cdots 01 \quad 0 \cdots 0 \end{array}$$

$k = \log(n/2)$ positions

The last digit in the smaller number is not lost if $n < 2/\epsilon$



Previous example was due to finite representation; this example is more due to algorithm itself.

Consider $y_n = \int_0^1 \frac{x^n}{x-5} dx = \frac{1}{n} - 5y_{n-1}$ (monotonically decreasing)

$y_0 = \ln 6 - \ln 5$.

In 3 decimal digits:

computation		correct result
$y_0 = \ln 6 - \ln 5 = .182 322 \times 10^1 \dots$		1.82
$y_1 = .900 \times 10^{-1}$.884
$y_2 = .500 \times 10^{-1}$.0580
$y_3 = .830 \times 10^{-1}$	going up?	.0431
$y_4 = -.165$	negative?	.0343

Reason? Define error as $\tilde{y}_n = y_n + \epsilon_n$, then

$$\tilde{y}_n = 1/n - 5\tilde{y}_{n-1} = 1/n + 5\epsilon_{n-1} = y_n + 5\epsilon_{n-1}$$

so $\epsilon_n \geq 5\epsilon_{n-1}$: exponential growth.



Problem: solve $Ax = b$, where b inexact.

$$A(x + \Delta x) = b + \Delta b.$$

Since $Ax = b$, we get $A\Delta x = \Delta b$. From this,

$$\left\{ \begin{array}{l} Ax = b \\ \Delta x = A^{-1} \Delta b \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \|A\| \|x\| \geq \|b\| \\ \|\Delta x\| \leq \|A^{-1}\| \|\Delta b\| \end{array} \right.$$

$$\Rightarrow \frac{\|\Delta x\|}{\|x\|} \leq \|A\| \|A^{-1}\| \frac{\|\Delta b\|}{\|b\|}$$

‘Condition number’. Attainable accuracy depends on matrix properties



Multiplication and addition are not associative:
problems for parallel computations.

compute $a + b + c + d$
sequential parallel

$((a + b) + c) + d$	$(a+b)+(c+d)$
---------------------	---------------

Operations with “same” outcomes are not equally stable:
matrix inversion is unstable, elimination is stable



Consider the iteration

$$x_{n+1} = f(x_n) = \begin{cases} 2x_n & \text{if } 2x_n < 1 \\ 2x_n - 1 & \text{if } 2x_n \geq 1 \end{cases}$$

Does this function have a fixed point, $x_0 \equiv f(x_0)$, or is there a cycle $x_1 = f(x_0)$, $x_0 \equiv x_2 = f(x_1)$ et cetera?

Now code this function and see what happens with various starting points x_0 . Can you explain this?



More



Two real numbers: real and imaginary part.

Storage:

- Store real/imaginary adjacent: easy to pass address of one number
- Store array of real, then array of imaginary. Better for stride 1 access if only real parts are needed. Other considerations.



Some compilers support higher precisions.

Arbitrary precision: GMPlib

Interval arithmetic

Half precision bfloat16

