LECTURE 13

REAL NUMBERS

How do we represent non-integers?

Keeping in mind:

- If we consider n bits of memory,
  - their values can take 2^n combinations
  - so we can represent 2^n numbers at best with those n bits
- We have a finite amount of memory,
  - so we cannot represent all real numbers
- We (typically) want fast operations,
  - so (ideally) we need hardware to perform them.
  - Hardware has tight limits on the number of logic gates available
  - meaning we use very few bits (say 16, 32 or 64, like for integers)
  - … further restricting how many real numbers we can represent

Practical limitations

- Integer are restricted in one way:
  - their range (e.g. $[\mathtt{INT\_MIN}, \mathtt{INT\_MAX}]$)
- Reals are restricted in two ways:
  - their range (e.g. $[-10^{308}, 10^{308}]$)
  - the number of reals we can represent in that range

    (e.g. $\{\ldots, 0, 10^{-200}, 2 \times 10^{-200}, \ldots\}$)

    i.e. their precision

FIXED-POINT ARITHMETIC

Decimal example

Instead of computing money values in €, we could use ¢:

e.g.   29.99 € = 2999 ¢

then use integer operations.

- This is fixed-point arithmetic
- specifically, with 2 decimal places reserved for the fractional part.

If $+$, $-$, $\times$, / are the elementary integer operations:

- $\texttt{euro\_to\_cent}(e) := e \times 100$
  - euro_to_cent(5 €) = 500 ¢
- $\texttt{cent\_to\_euro}(a) := a/100$
  - cent_to_euro(700 ¢) = 7 €
- $\texttt{cent\_add}(a, b) := a + b$
  - cent_add(700 ¢, 500 ¢) = 1200 ¢
- $\texttt{cent\_sub}(a, b) := a - b$
  - cent_sub(700 ¢, 500 ¢) = 200 ¢
- $\texttt{cent\_mul}(a, b) := (a \times b)/100$
  - 5 € × 7: cent_mul(500 ¢, 700) = 500 × 700 / 100 = 3500 ¢
- $\texttt{cent\_div}(a, b) := (a \times 100)/b$
  - 8 € / 4: cent_div(800 ¢, 400 ¢) = (800 × 100) / 400 = 200 ¢

Binary fixed-point arithmetic

- There is no universally accepted standard for fixed-point arithmetic
- But there is no real need for one:
  - Only two parameters:
    - n: total number of bits
    - p: number of bits after the decimal point
  - All the operations are just integer operations
    - For `mul` and `div`, two integer operations each

Binary example

64-bit integer	
32-bit integer part	32-bit fractional part

- $\texttt{i64\_to\_fix}(e) := e \times 2^{32}$
- $\texttt{fix\_to\_i64}(a) := a/2^{32}$
- $\texttt{fix\_add}(a, b) := a + b$
- $\texttt{fix\_sub}(a, b) := a - b$
- $\texttt{fix\_mul}(a, b) := (a \times b)/2^{32}$
- $\texttt{fix\_div}(a, b) := (a \times 2^{32})/b$

```c
typedef int64_t fix;

static inline fix to_fix(int64_t e)
{
    return e << 32;
}

static inline int64_t from_fix(fix a)
{
    return a >> 32;
}

static inline fix fix_add(fix a, fix b)
{
    return a + b;
}

static inline fix fix_sub(fix a, fix b)
{
    return a - b;
}

static inline fix fix_mul(fix a, fix b)
{
    return ((__int128)a * b) >> 32;
}

static inline fix fix_div(fix a, fix b)
{
    return ((__int128)a << 32) / b;
}
```

Fixed-point arithmetic

Pros:

- fast, no need for extra hardware
- easy to understand and study (predictible):
  - uniform absolute precision (e.g. 2^{-32} over whole range)

Cons:

- limited range (e.g. $[-2147483648.999, \quad 2147483647.999]$)
- limited precision (e.g. $2^{-32} \simeq 0.000000002328$)

Possible improvements:

- larger range
- better absolute precision around zero
- lower absolute precision for big numbers

FLOATING-POINT ARITHMETIC

Scientific notation

Take the number -2147483648.999:

$$\begin{aligned} & - & 2147483648.999 & & \\ = & - & 2.147483648999 & \times & 10^{9} \end{aligned}$$

`=-2.147483648999e+9`

Similarly, take the number 0.0000000002328:

$$\begin{aligned} & 0.0000000002328 & & \\ = & 2.328 & \times & 10^{-10} \end{aligned}$$

`=2.328e-10`

Scientific notation (definition)

$$-2.147483648999 \times 10^{+9}$$

$$\pm d.\mathtt{mmmmm...} \times 10^{\pm \mathtt{xxx..}}$$

- ± + or -
- d single digit between 1 and 9
- mmmmm... predeterminated number of digits between 0 and 9
- ±xxx.. + or -, predeterminated number of digits between 0 and 9

Binary floating-point numbers

$$\pm\ d.\mathtt{mmmmm...} \times 2^{\pm\mathtt{xxx..}}$$

- $\pm$ sign bit + or -
- d single bit 1 and 1

Binary floating-point numbers

$$\pm 1.\texttt{mmmmm...} \times 2^{\pm\texttt{xxx..}}$$

- $\pm$ sign bit + or -
- `mmmmm...` "mantissa" bits
- `±xxx..` "exponent" bits

Now we just need to all agree on how many bits for each…

Floating-point standard

- In 1985, the Institute of Electrical and Electronics Engineers publishes standard #754 about floating-point arithmetic (IEEE-754)
- Most hardware makers adopt the standard very quickly thereafter (Intel 30387, launched in 1987, is fully compliant)
- x86_64 natively supports binary32 and binary64 formats
- AArch64 natively supports binary16, binary32 and binary64 formats

component	binary16	binary32	binary64
± sign bit	1	1	1
`mmmmm...` mantissa bits	10	23	52
`±xxx..` exponent bits	5	8	11
exponent range	-14..15	-126…127	-1022…1023

Precision

Let $\mathrm{fl}(x)$ be the floating-point representation of the real number $x \in \mathbb{R}$.

- **Absolute precision:** For a given x, the smallest e such that

$$\mathrm{fl}(x + e) \neq \mathrm{fl}(e)$$

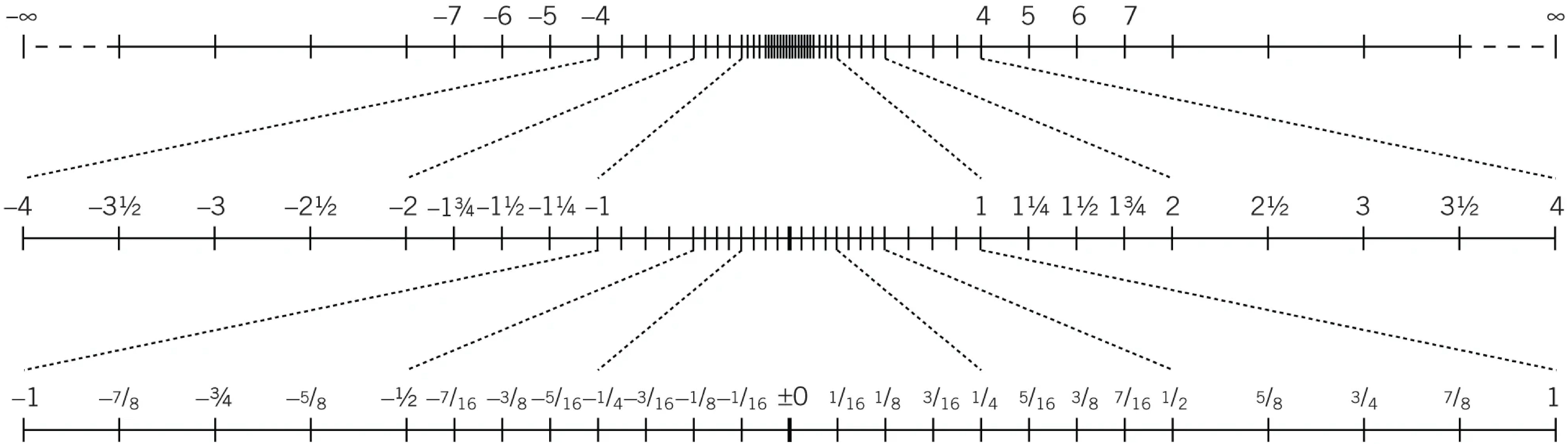
- **Relative precision:** For a given x,

$$\varepsilon := \frac{e}{x}$$

binary64 vs. fixed-point 32+32

	fixed-point 32+32		floating-point binary64	
	absolute	**relative**	**absolute**	**relative**
precision at 10^{-9}	2.33×10^{-10}	0.0233	2.07×10^{-25}	2.22×10^{-16}
precision at 10^{-6}	2.33×10^{-10}	2.33×10^{-5}	2.12×10^{-22}	2.22×10^{-16}
precision at 10^{-3}	2.33×10^{-10}	2.33×10^{-8}	2.17×10^{-19}	2.22×10^{-16}
precision at 1	2.33×10^{-10}	2.33×10^{-11}	2.22×10^{-16}	2.22×10^{-16}
precision at 10^{+3}	2.33×10^{-10}	2.33×10^{-14}	1.14×10^{-13}	2.22×10^{-16}
precision at 10^{+6}	2.33×10^{-10}	2.33×10^{-17}	1.16×10^{-10}	2.22×10^{-16}
precision at 10^{+9}	2.33×10^{-10}	2.33×10^{-20}	1.19×10^{-7}	2.22×10^{-16}
precision at 10^{+16}	❌		2.00	2.22×10^{-16}
range	$\lvert x\rvert \leq 2.15 \times 10^{9}$		$\lvert x\rvert \leq 1.80 \times 10^{308}$	

The floating-point number line



Languages that mandate IEEE-754 for floating-point

language	since	binary32	binary64
C	C99	`float`	`double`
C++	C++03	`float`	`double`
Fortran	Fortran 2003	`real`	`double`
Rust		`f32`	`f64`
Python			✔
JavaScript			✔

Inaccuracy

In base 10,

- $1/3 \simeq 0.3333$
- $2/3 \simeq 0.6666$
- $1/3 + 2/3 \simeq 0.9999$

In base 2,

```
>>> a = 0.1
>>> f'{a:.50f}'
'0.10000000000000000555111512312578270211815834045410'
```

Numerical instability

Consider the following approximation of the derivative of f:

$$\frac{d}{dx}f(x) \simeq \frac{f(x+\delta) - f(x)}{\delta}$$

Let us consider the function f:

$$f(x) = x \quad \text{so} \quad \frac{d}{dx}f(x) = 1$$

and compute its derivative with $\delta = 10^{-6}$.

- at $x = 10^{+5}$,

```
>>> ((1e+5 + 1e-6) - 1e+5) / 1e-6
0.9999930625781417
```

- at $x = 10^{+8}$,

```
>>> ((1e+8 + 1e-6) - 1e+8) / 1e-6
0.998377799987793
```

- at $x = 10^{+10}$,

```
>>> ((1e+10 + 1e-6) - 1e+10) / 1e-6
1.9073486328125
```

What is happening?

```
>>> ((1e+10 + 1e-6) - 1e+10) / 1e-6
1.9073486328125
```

- At $x = 10^{+10}$, we first compute `(1e+10 + 1e-6)`
  - which is a big number, close to `1e+10`
  - floating-point numbers have a good *relative* accuracy everywhere, $\simeq 2.22 \times 10^{-16}$
  - but at 10^{+10}, the *absolute* accuracy is not great, $\simeq 1.91 \times 10^{-6}$
  - so the result of `(1e+10 + 1e-6)` may be off by roughly 1.91×10^{-6}
- We then subtract `1e+10`.
  - If we were using exact arithmetic, we would get `1e-6` exactly,
  - but we are using floating-point arithmetic,
  - so we get something close to `1e-6`…
  - … but potentially off by roughly 1.91×10^{-6}
- We divide by `1e-6`,
  - and get a number in $[1 - 1.91, \quad 1 + 1.91]$

Therefore,

- floating-point accuracy is often great
- but some algorithms are unstable
- we need to be extremely careful before trusting floating-point results

Never do exact comparisons

```
>>> 0.1 + 0.2 == 0.3
False
```

```
>>> 1.0 + 1e-16 <= 1.0
True
```

So how do we do comparisons?

- If exact comparisons are important, do not use floating-point arithmetic.
- If we care about speed and can tolerate some errors…

```
>>> 0.1 + 0.2 == 0.3
False
```

becomes

```
>>> tolerance = 1e-10
>>> abs(  (0.1 + 0.2) - 0.3   ) <= tolerance
True
```

```
>>> x >= 0.0
```

becomes

```
>>> x >= -tolerance
```

FLOATING-POINT ROUNDING

Given a floating point number a, we want to compute $x = a/3$.

Q: If $a/3$ cannot be represented exactly by a floating-point number, what value do we give x?

A: We "round" x to the floating-point number "closest" to the real value $a/3$.

Rounding modes

- Round to nearest, ties to even (default)
    - nearest value
    - in case of ties, set last mantissa bit to zero
- Round to nearest, ties away from zero
    - nearest value
    - in case of ties, set last mantissa bit to one
- Round toward zero
    - if between two numbers, choose the one nearest to zero
    - even if it is not the nearest to the real value
- Round toward $+\infty$: always round up
- Round toward $-\infty$: always round down

Determinism

- Floating-point arithmetic is sometimes inaccurate
- but it is deterministic:
  - the result of most operations is precisely defined
  - we can predict the result of such operations bit-for-bit

Let us denote by $\mathrm{fl}(x)$ the floating-point representation of the real number $x \in \mathbb{R}$.

The IEEE-754 standard mandates correct rounding
as specified by the currently-selected rounding mode
for:

- addition, negation, subtraction: `x + y` gives $\mathrm{fl}(x + y)$
- multiplication, division: `x / y` gives $\mathrm{fl}(x/y)$
- square root: $\mathtt{sqrt}(x) = \mathrm{fl}\left(\sqrt{x}\right)$
- fused multiply-add: $\mathtt{fma}(x, y, z) = \mathrm{fl}(x \times y + z)$

Division example

When executing

```
z = x / y
```

- we first take the floating-point numbers x and y, and consider them as if the were (exact, infinite-precision) real numbers
- we compute the (exact, infinite-precision) real quotient x/y.
- we round the result according to the current rounding mode: $\mathrm{fl}(\mathrm{x}\ /\ \mathrm{y})$
- we store the rounded floating-point value into z

Expression example

```
(y * (x + 4.0)) / (z - 3.0)
```

gives:

$$\mathrm{fl}\left(\ \mathrm{fl}(y \times \mathrm{fl}(x + 4)) \ / \ \mathrm{fl}(z - 3)) \ \right)$$

About fused multiply-add

Beware:

$$\texttt{fma(x, y, z)} \neq \texttt{x * y + z}$$

Indeed:

- $\texttt{fma}(x, y, z) = \mathrm{fl}(x \times y + z)$
- but `x * y + z` gives $\mathrm{fl}(\ \mathrm{fl}(x \times y) + z\ )$

More floating-point non-identities

- associativity does not hold: `x + (y + z)` $\neq$ `(x + y) + z`
- distributivity does not hold: `x * (y + z)` $\neq$ `x * y + x * z`

The IEEE-754 standard mandates correct rounding for:

`+, -, ×, /, sqrt(), fma()`

The IEEE-754 standard does not mandate correct rounding for most other functions, in particular:

- `sin, cos, tan`
- `asin, acos, atan`
- `sinh, cosh, tanh`
- `pow, log, log2, log10, exp, exp2, exp10`

Floating-point and compilers

- C99 and C++03 mandate IEEE-754
- which in turn mandates correct rounding for `+, -, ×, /, sqrt(), fma()`.
- However, if we do not specify a C or C++ standard (e.g. `-std=c17` or `-std=c++20`), `gcc` and `clang` do not follow IEEE-754
    - they will happily exploit associativity and distributivity
    - they will replace `x * y + z` by `fma(x, y, z)`

Why does correct rounding matter?

- (generally) not because of accuracy
- but because for any real number x, there is exactly one correct rounding
- as a result, there is no ambiguity:
  - given a set of floating-point numbers
  - given any expression involving those numbers and `+, -, ×, /, sqrt(), fma()`
  - there is exactly one correct answer
  - which is precisely specified by IEEE-754, down to its bit representation

What happens without correct rounding?

Results can change when:

- we change architecture
- we change compiler
- we change the standard C library
- we change the version of the compiler
- we change the version of the standard C library
- we change our code (even a completely unrelated part)

Note: If we use `sin, cos, log, exp, ...`, which are not correctly rounded, then we are exposed to result changes whenever we change the version of the standard C library (which could be dynamically linked!)

BEYOND FLOATING-POINT ARITHMETIC

Interval arithmetic

We represent every real number $x \in \mathbb{R}$
by a pair of floating-point number (l, u)
with $x \in [l, u]$.

We exploit the Round toward +∞ and Round toward -∞ modes
to compute the appropriate interval for every operation.

Pros

- fast
- we always know how accurate a result is

Cons

- the interval $[l, u]$ often becomes large very quickly

  (the bounds are usually too pessimistic)

Unum

- introduced in 2015, latest revision 2017
- For a given fixed bit width, claims better allocation of available precision
- optional interval arithmetic
- very limited adoption (no hardware support on any mainstream platforms)

The GNU multi-precision library

GMP is a C library that provides support for:

- variable-width (a.k.a. arbitrary-size) integers
- arbitrary-size rational numbers (i.e. fractions):

$$\text{fraction} = \frac{\text{numerator}}{\text{denominator}},$$

where $\gcd(\text{numerator}, \text{denominator}) = 1$

> gmplib.org

The GNU MPFR library

MPFR builds on top of GMP to add arbitrary-size floating-point numbers

```
double x = 22.0 / 7.0;

printf(".20f\n", x);
```

```
mpfr_t x;

mpfr_init2(x, 512);                 // initialize x with 512-bit mantissa

mpfr_set_ui(x, 22, MPFR_RNDN);      // set x to value 22, round-to-nearest
mpfr_div_ui(x, 7, MPFR_RNDN);       // divide x to 7, round-to-nearest
mpfr_printf("%.200Rf\n", x);        // print x

mpfr_clear(x);                      // free memory
```

> mpfr.org

Python fractions

Python integers are already variable-width by default:

```
>>> -2 ** 65
-36893488147419103232     # <-- correct result, no overflow
```

Python `fractions` add support for (variable-width) rationals in top of them:

```
import fractions

a = fractions.Fraction(numerator, denominator)
```

Why don’t we always use exact rational numbers?

- convenience (unfortunately)
  - need to use GMP in C
  - need “`import fractions`” in Python
- memory
  - the size of the numerator and denominator can explode in iterative algorithms (despite gcd reductions)
- speed
  - since arbitrary-sized integers don’t come with native hardware support, operations are much slower (typically 10× for small numbers, then it grows with size)

Should we use exact rational numbers more often?

(in particular when exactness matters)

(or when and speed does not matter)

YES

Symbolic computations

In a symbolic algebra system:

- $\sqrt{2}$ is never evaluated to $\simeq 1.4142$:

```
sage: sqrt(8)
2*sqrt(2)
```

- We can also carry variables that have no specific value:

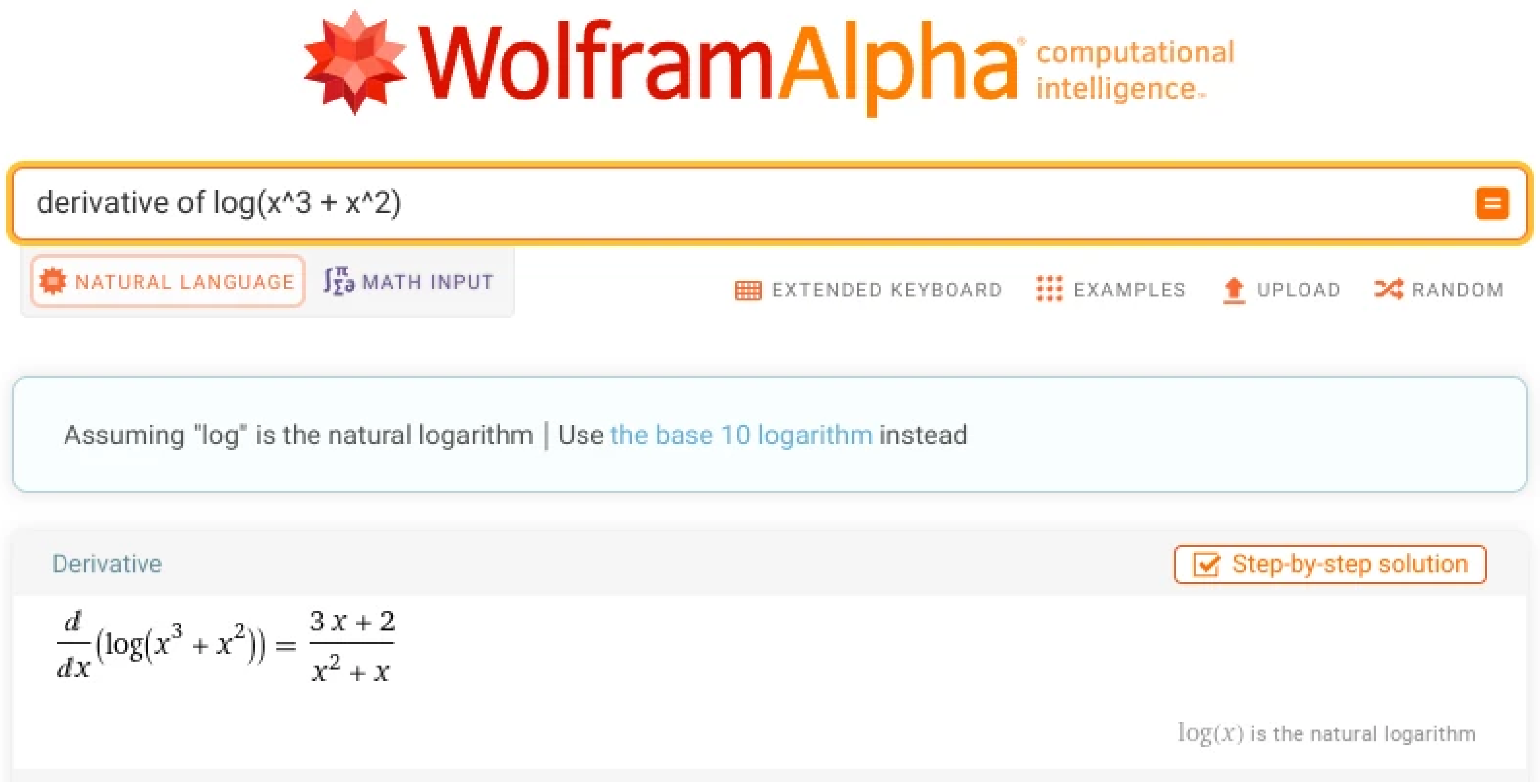
```
sage: x, y, z = var('x y z')
sage: sqrt(8) * x
2*sqrt(2)*x
```

- This allows us to solve problems symbolically:

```
sage: x, b, c = var('x b c')
sage: solve([x^2 + b*x + c == 0], x)
[x == -1/2*b - 1/2*sqrt(b^2 - 4*c), x == -1/2*b + 1/2*sqrt(b^2 - 4*c)]
```

Symbolic algebra systems

- SageMath (free software, syntax similar to Python)
- Maple
- Wolfram Mathematica



> wolframalpha.com