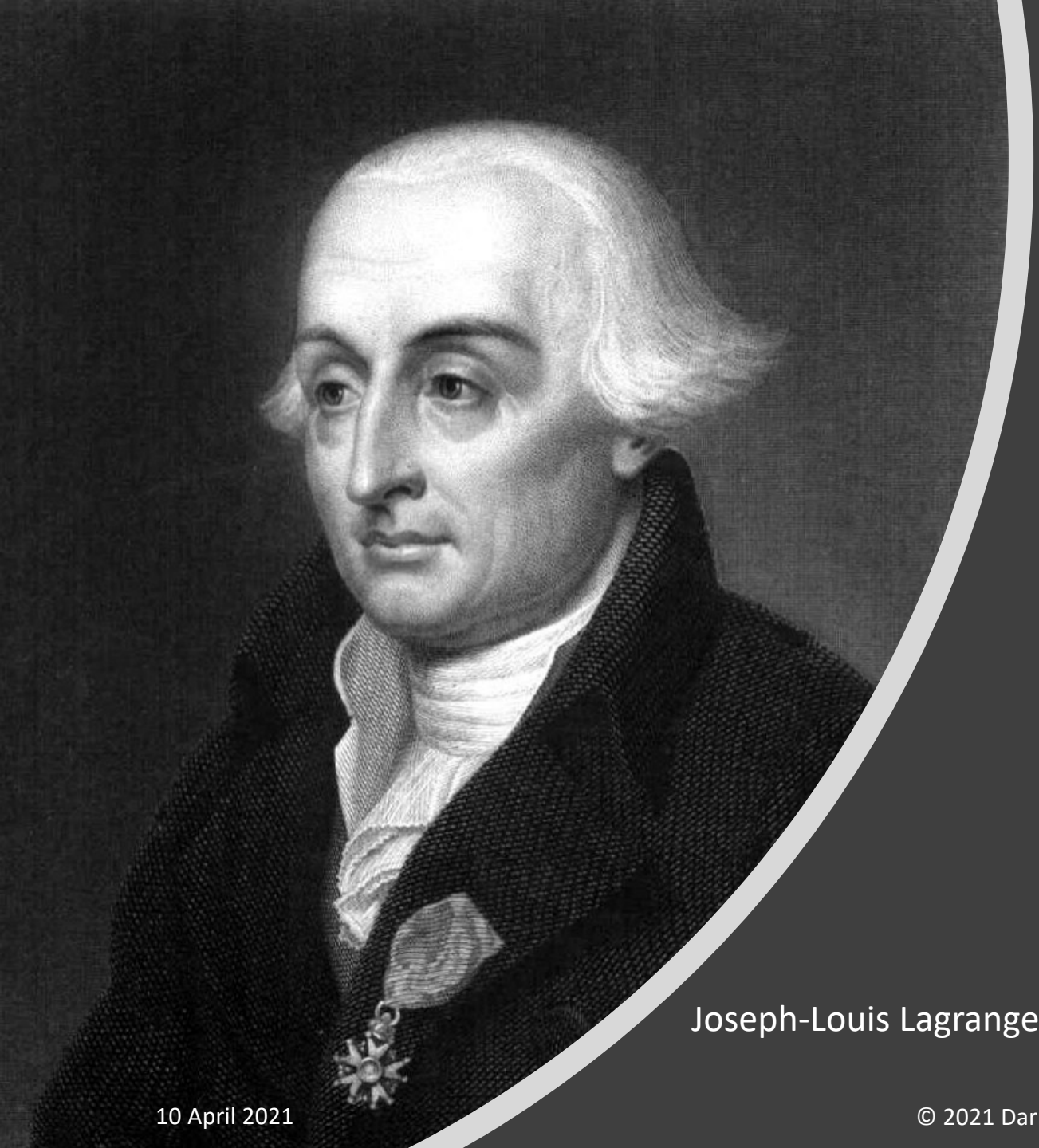


Joseph-Louis Lagrange

Numerical Computation

Darrell Long

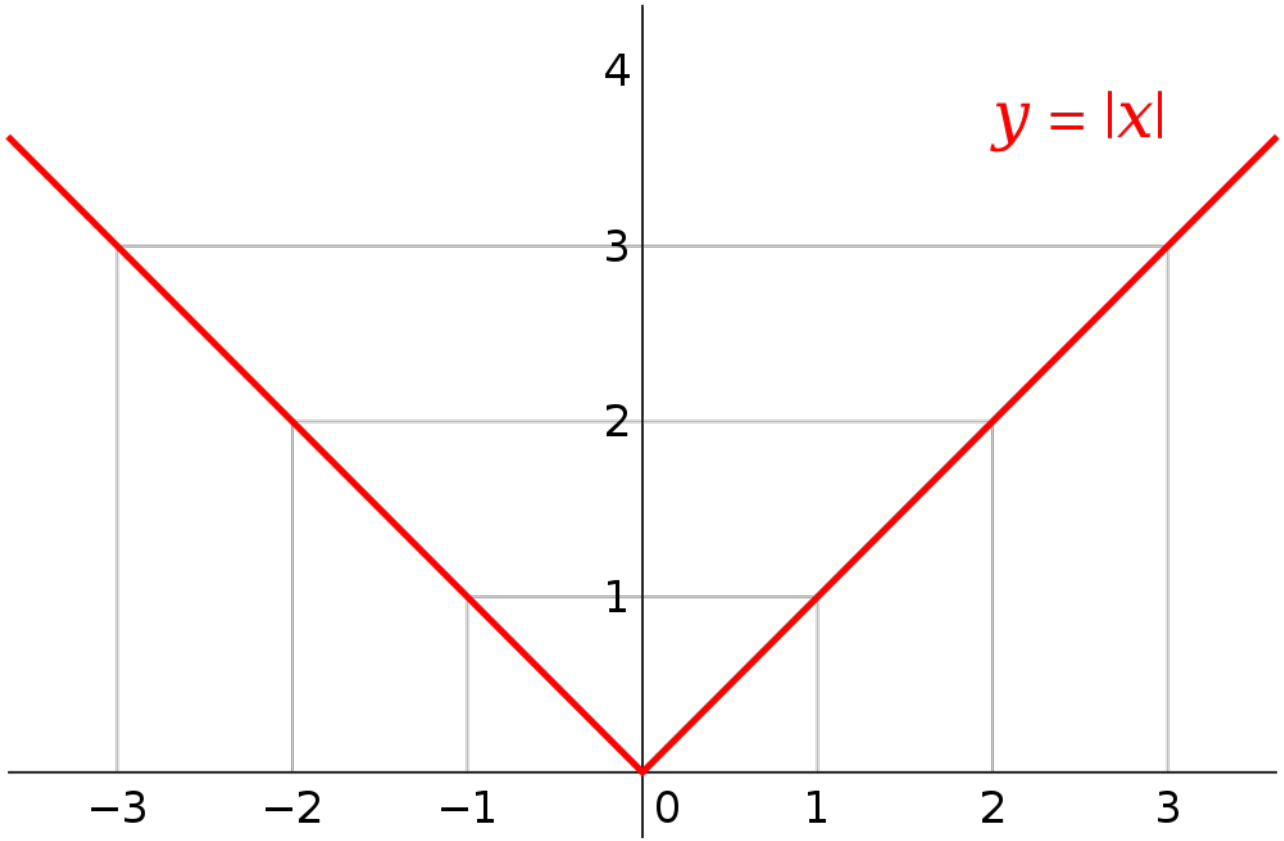
CSE 13S

Basic Arithmetic Operations

- Computers can only do basic operations such as *addition*, *subtraction*, *multiplication* and *division*.
  - Multiplication is *shift* and *addition*,
  - Division is *shift* and *subtraction*,
  - Addition is little more than *exclusive-xor*
- What about trigonometric functions?
  - Some processors do them internally, but they are still using the basic operations.
- What about integrals? What about transforms? What about …?
  - These are in the realm of *numerical methods*.

Absolute Value

- That's an easy one: abs(x) = if $x < 0$ then $-x$ else x
- Should I just write the code?
  - It depends… Most mathematical libraries have families of absolute value functions that depend on the data type.
  - It may be faster to let the compiler or the mathematics library do it, since as in the case of floating point it may be just clearing a bit.

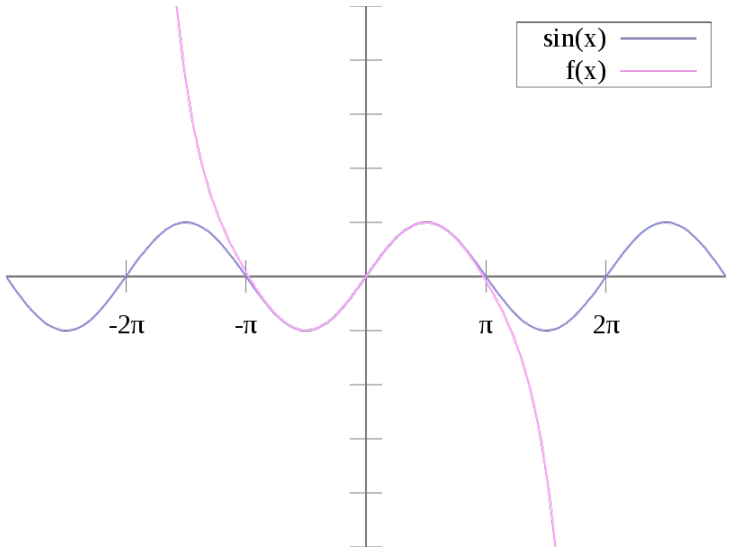


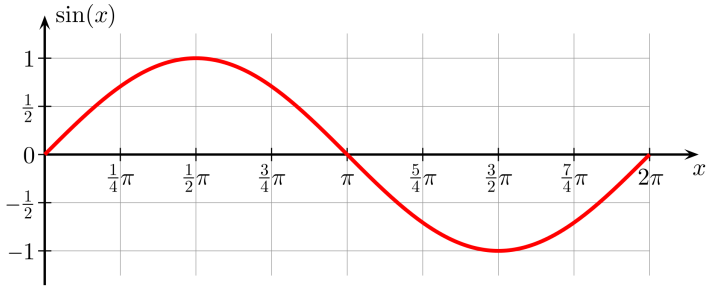
Check the man page!

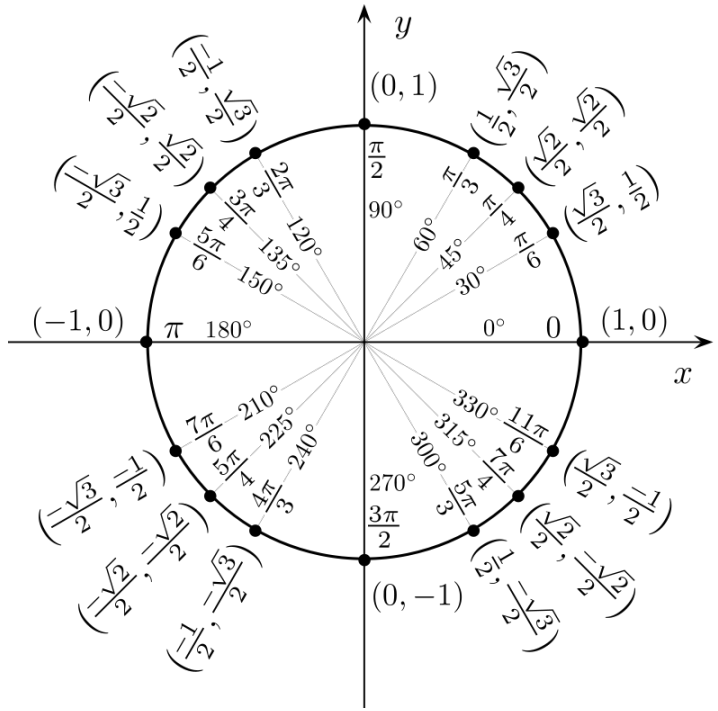
```
FABS(3)                    Linux Programmer's Manual                    FABS(3)



NAME
       fabs, fabsf, fabsl - absolute value of floating-point number

SYNOPSIS
       #include <math.h>

       double fabs(double x);
       float fabsf(float x);
       long double fabsl(long double x);

       Link with -lm.

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       fabsf(), fabsl():
           _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE >= 600 ||
           _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L;
           or cc -std=c99
```

Can I just use the library?

- In general, yes, it is best to use the library.
- The library should be designed to be:
  - Careful with numerical precision,
  - Fast, and portable.
- That being said, it is important for you to have some understanding of what goes into implementing mathematical functions.
- Remember: *There is no magic*!







Trigonometric Functions

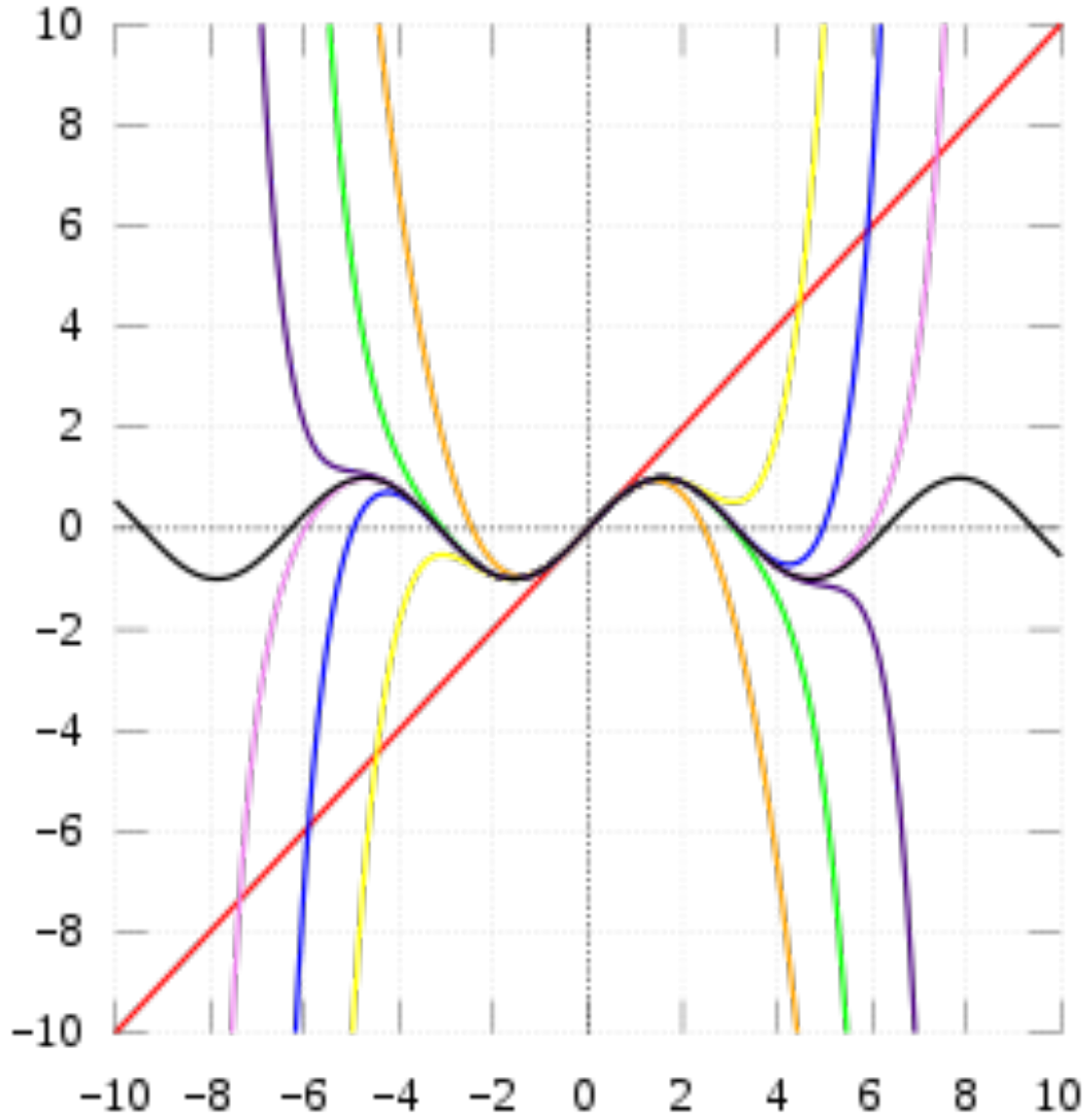
Taylor Series

- If $f(x)$ is a real or complex valued function, and it is infinitely differentiable then:
  - $f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n$
- For example, consider e^x centered at $a = 0$:
  - $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \ -\infty < x < \infty$

Brook Taylor

Increasingly Better Approximations

Each derivative adds a term to the approximating polynomial.



Computing e^x

- $k! > x^k$ for all x, eventually (when k gets large enough), so:
  - $\frac{x^k}{k!}$ gets smaller with each step (again, eventually)
  - We can stop when it gets small enough that it is "good enough" for our computation.

```
#define EPSILON 0.00001

long double Exp(long double x) {
  long double term = 1.0;
  long double sum  = term;
  for (long double k = 1.0; fabsl(term) > EPSILON; k += 1.0)
  {
    term = x / k * term;
    sum += term;
  }
  return sum;
}
```

Computing $\sin(x)$

What do we know about $\sin(x)$?

- It is:
  - Periodic on $[-2\pi, 2\pi]$, meaning that $\sin(9\pi) = \sin(\pi)$
  - Infinitely differentiable, making a good candidate for a Taylor series.

- Where shall we center it? How about right in the middle, at *zero*:
  - $\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \frac{x^{11}}{39916800} + O(x^{12})$

- Is that good enough?
  - It depends on how close x is to zero.
  - In general, you will need more terms.

Computing the Terms

- We skip even numbered terms, and
- The sign of the terms *alternate*.
- Before computing $\sin(x)$ do not forget to remove extra factors of 2π.
- How many terms do we need?

```
double sum = x;
double numerator = x;
double denominator = 1;
for (double k = 3.0; k < N; k += 2) {
  numerator *= -x * x;
  denominator *= k * (k - 1.0);
  sum += numerator / denominator;
}
```

Henri Padé

Padé Approximants

A *Padé Approximant* is the ratio of two polynomials that correspond to a series but is easier to compute and fit better around the center.

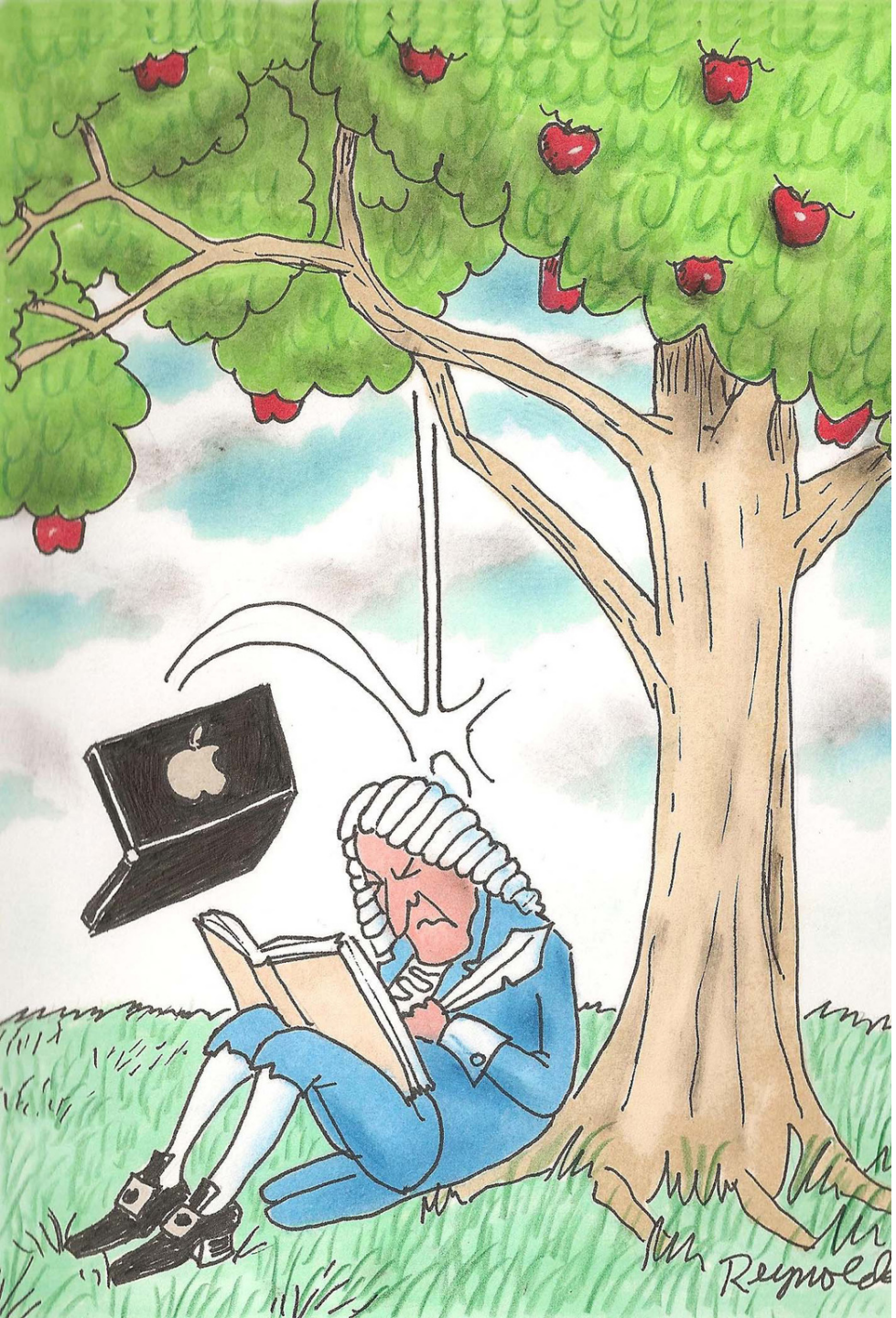
A 15 term Taylor series results in the Padé approximant:

- $p(x) = -72x\ (62077121x^6 - 7479608290x^4 + 242757835320x^2 - 1727021696400)$
- $q(x) = 75x^4 + 5460x^2 + 166320$
- $\sin(x) \approx \frac{p(x)}{q(x)}$

- How do you find them?
  - Take a course on numerical methods.
  - Look them up in old books.

What happens if the series converges too slowly?

- Here is the series for $\log(x+1)$:
  - $x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \frac{x^6}{6} + \frac{x^7}{7} - \frac{x^8}{8} + \frac{x^9}{9} - \frac{x^{10}}{10} + O(x^{11})$
- It converges *very* slowly unless x is close to 0.
  - Just look at the denominator!
- We can do a little algebra, and remember that:
  - $\log(x) = -\log\left(\frac{1}{x}\right)$, for $x > 0$.
- But still, it will converge far too slowly. What can we do?
- We recall that $x = \log(e^x) = e^{\log(x)}$
- And so, we will *invert* the exponential function!

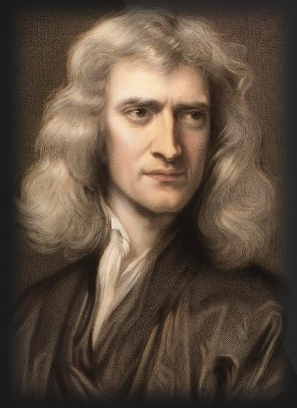


Inverting a Function

- One of the simplest functions that give us trouble is square root ($\sqrt{x}$).
- We can't really expand a Taylor series for $x^{\frac{1}{k}} = \sqrt[k]{x}$ since it only has one term.
- So, what do we do?
- We could search for it… using binary search.
  - Why can we do that? The solutions are *monotonic*.
- Can we do better?
  - Yes, we can. But first you need an apple to fall on your head.
- What is the inverse? Recall that $\sqrt{x^2} = x$.

$\sqrt{n}$ using a Newton Iterate

- Sir Isaac Newton comes to the rescue again.
- We use this iterate:
  - $x_{k+1} = \frac{1}{2}(x_k + \frac{n}{x_k})$
- As before, we iterate until the approximation is *good enough*.



```
#define EPSILON 0.00001

long double Sqrt(long double x) {
  long double y = 1.0;
  long double old = 0.0;
  while (fabsl(y - old) > EPSILON) {
    old = y;
    y = 0.5 * (y + x / y);
  }
  return y;
}
```

$\log(x)$ using Newton's Method

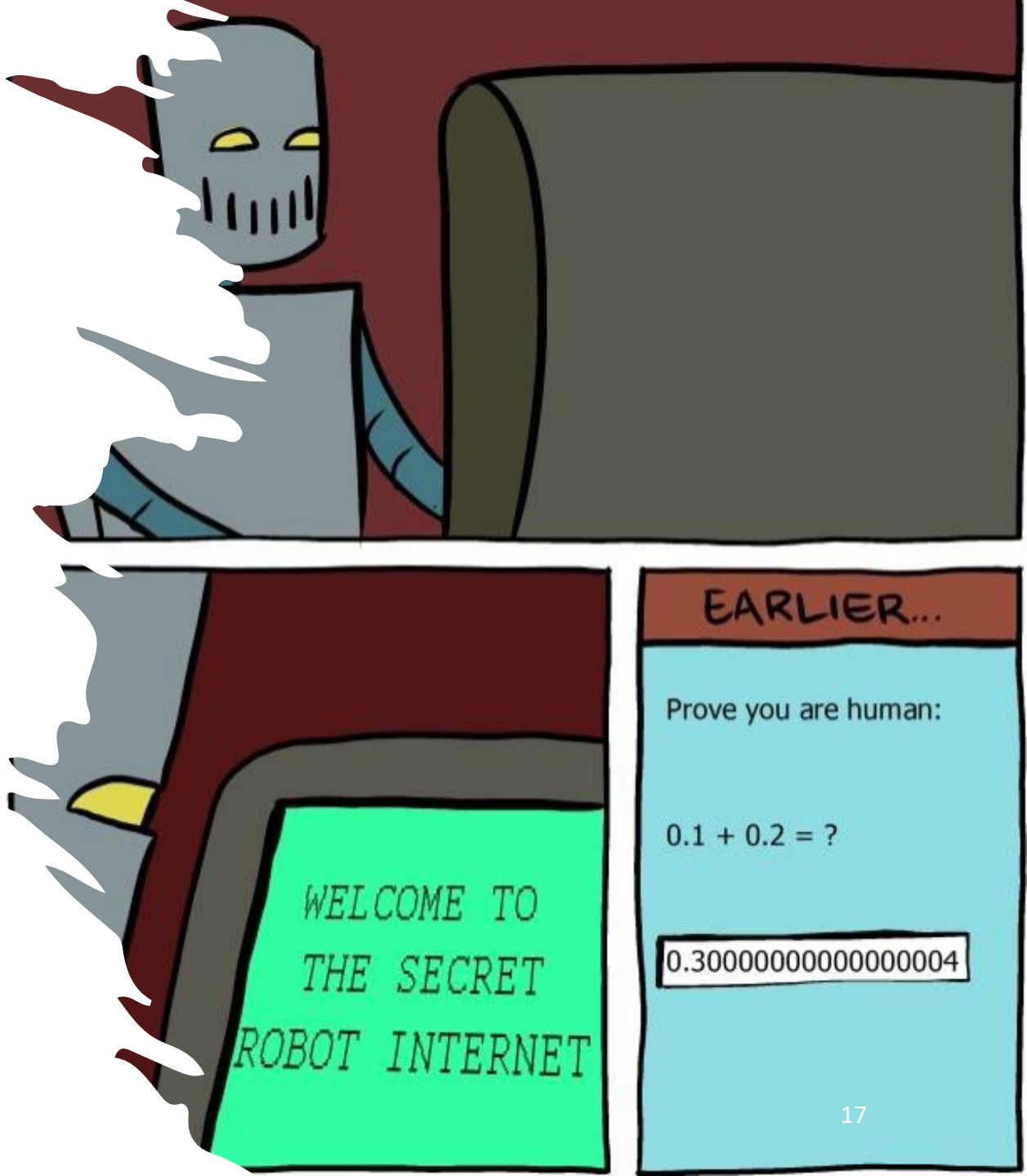
```
double Log(double x) {
    double y = 1.0;
    double p = Exp(y); // First guess
    while (Abs(p - x) > EPSILON) {
        y = y + (x - p) / p;
        p = Exp(y);
    }
    return y;
}
```

- Newton's formula provides a linear approximation of the function.
- Solve for x_{n+1}
  - $x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)}$
- Find the *root* of $f(y) = x - e^y$
- We iterate this equation until our answer is *close enough*:
  - $y_{n+1} = y_n + \frac{x - e^{y_n}}{e^{y_n}}$
- For details, ask your calculus professor!
  - You can get very deep very quickly!

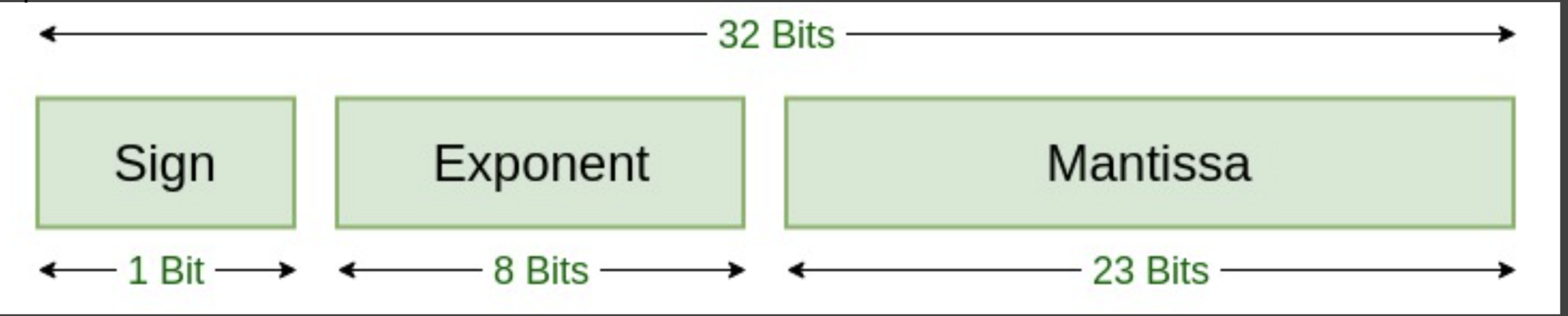
Floating Point Arithmetic

- Normal mathematics:
  - Real numbers are exact
- Computer arithmetic:
  - numbers are approximated (finite)

- Approximations cause round-off errors. Why?
- Because the result is also an approximation!



IEEE 754 Floating Point Standard



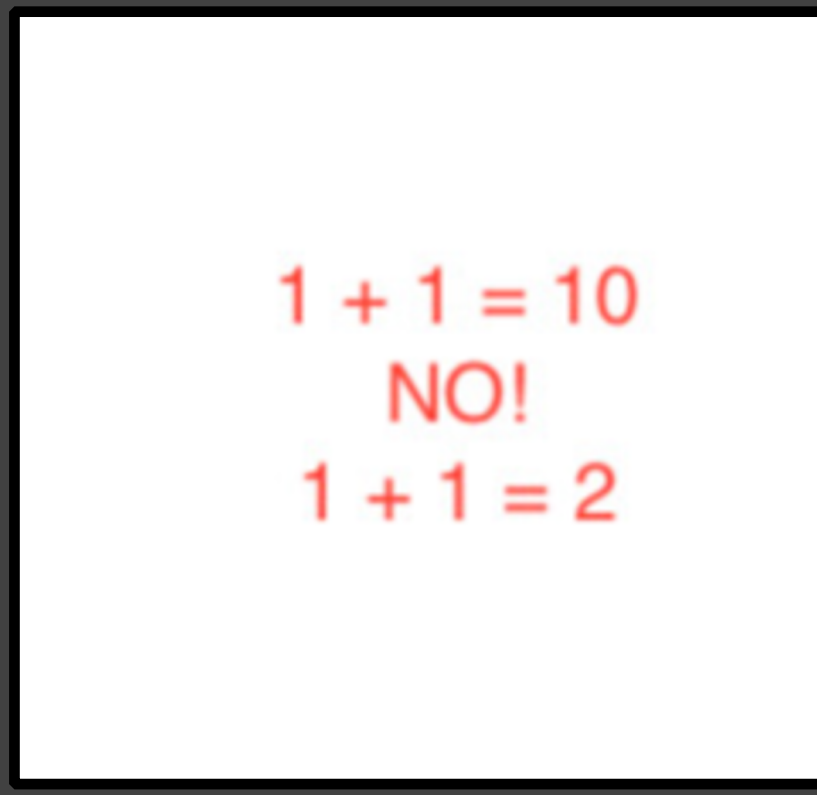
- The floating point standard is used to represent numbers in binary
- Single precision: 32 bit
  - Sign bit (s) - indicates sign
  - Characteristic (c) - 8 bit base-2 exponent with a range of -126 to 127
  - Mantissa (f) - 23 bit binary fraction which amounts to 6 digit decimal precision
- Double precision: 64 bit
  - 1 bit sign, 11 bit characteristic and 52 bit mantissa

Decimal Representation

- Decimal representation is easier for us to understand conceptually
- Float point form is obtained by chopping or rounding the mantissa.
- For ex. $\pi = 3.14159265$,
- On five-digit chopping:

  $\pi = 3.1415$
- On five-digit rounding:

  $\pi = 3.1416$



Absolute error and relative error

- If p* is an approximation of p, absolute error is given by:

$$|p - p^*|$$

- Relative error is given by:

$$\frac{|p - p^*|}{|p|}$$

Relative error is more meaningful

- Because it takes the size of the value into consideration.
- For ex., p = 3.0, p* = 3.1
  - Absolute error = 0.1
  - Relative error = 0.03333
- p = 0.003, p* = 0.0031
  - Absolute error = 0.0001
  - Relative error = 0.03333

Finite Digit Arithmetic

- Calculation in actuality uses a lot of logical and shifting operations as opposed to math operations.
- After every calculation the result is once again approximated to fit it into a finite number of digits.
- This results in a loss of accuracy as the number of calculations increases.

For ex.,

If we were subtracting two nearly equal numbers, there is a large loss in the number of significant digits

$p = 0.54617$, $q = 0.54601$

Exact: $p - q = 0.00016$

Chopping approximation (4 digits):
$p^* - q^* = 0.5461 - 0.5460 = 0.0001$

Rounding approximation (4 digits):
$p^* - q^* = 0.5462 - 0.5460 = 0.0002$

Nested Arithmetic

- Loss of accuracy can be mitigated by re-arranging calculations
- For ex.,
  - Let's evaluate

f(x) = x^3 - 6.1x^2 + 3.2x + 1.5 at x = 4.71

Method	Value
Exact	-14.263899
Chopping	-13.5
Rounding	-13.4

Relative error on chopping :

$$\frac{|-14.263899+13.5|}{|-14.263899|} \approx 0.05$$

Relative error on rounding :

$$\frac{|-14.263899+13.4|}{|-14.263899|} \approx 0.06$$

- On nesting the calculation:

$$f(x) = ((x-6.1)x + 3.2)x + 1.5$$

Method	Value
Exact	-14.263899
Chopping	-14.2
Rounding	-14.3

Relative error on chopping :

$$\frac{|-14.263899 + 14.2|}{|-14.263899|} \approx 0.0045$$

Relative error on rounding :

$$\frac{|-14.263899 + 14.3|}{|-14.263899|} \approx 0.0025$$

What do we know?

- Floating point numbers are *not* real numbers.
- We need to take care when working with floating point numbers due to limited precision.
  - For example, subtracting two numbers of very different magnitudes can lose precision.
- A Taylor series can approximate an infinitely differentiable function to arbitrary precision, but:
  - Often the series converges too slowly to be useful.

What should we do?

- Use a good library when you can.
- Be cognizant of the difference between floating point and real numbers.
  - *Real* numbers are *all* of the numbers, *rational* numbers are a subset of the reals, and *floating point* numbers are a subset of the rationals.
- If you forget that they are different, you *will* get incorrect answers.
- Understand that all of the special functions we have discussed are *approximations*.