Floating-point arithmetic and error analysis (AFAE)

Elementary Functions

Stef Graillat

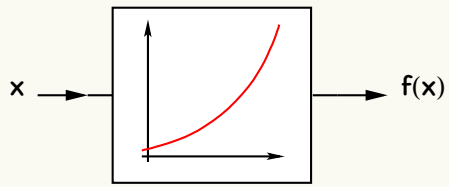
LIP6/PEQUAN - Sorbonne University

Lecture Master 2 CCA

$+, -, \times, /$ and $\sqrt{}$ - it is not all

- Basic operations:
  - Addition, subtraction, multiplication, division, square root
  - For those operations, we know the exact result:
    - it is a rational number or a zero of a polynomial of degree 2
- it is not all: we also need other functions:
  - exp(x) for bacteria or radioactivity, ...
  - sin(x) for wave, ...
  - atan(x) for reflexion of light, ...
  - erf(x) for statistic and finance, ...

Elementary functions must be computed by a computer. It is implemented in `math.h` and in `libm` library.



How can we compute those elementary functions?

- Question 1: give the first decimal digits of $\sqrt{17}$!
  - $4^2 = 16, \quad 5^2 = 25$
  - $4.5^2 = 20.25, \quad 4.25^2 = 18.0625$
  - $4.125^2 = 17.015625$ so 4.12

- Question 2: give the first decimal digits of $\log(17)$!
  - Not simple! How to do that
    - This is the aim of this lecture.

Outline of the lecture

Outline

Functions in libm

In libm mathematical libraries, we often find:

- elementary functions:
  - $\exp, 2^x, 10^x, \log, \log_2, \log_{10}$
  - $\sinh, \cosh, \operatorname{asinh}, \operatorname{acosh}$
  - $\sin, \cos, \tan, \operatorname{asin}, \operatorname{acos}, \operatorname{atan}$
  - x^y
    - They are called elementary from Liouville.
- Some special functions:
  - $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
  - Bessel functions J and Y
  - Γ function
- Some composite functions:
  - $\exp(x) - 1, \log(1 + x)$
  - $\sin(\pi x), \cos(\pi x), \tan(\pi x), \frac{\operatorname{asin}(x)}{\pi} \ldots$
  - $\operatorname{atan}(\frac{y}{x})$

Framework: floating-point arithmetic

- Floating-point arithmetic is a representation of real numbers
- It is a semi-logarithmic system
  - An exponent E gives the order of magnitude.
  - A significand m indicates the digits of the number.
- Examples :

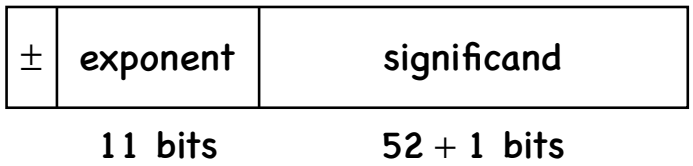
$$\begin{aligned} 105.415 &= 10^3 \cdot 1.05415 &&= 10^E \cdot m \\ 0.000000124565 &= 10^{-7} \cdot 1.24565 &&= 10^E \cdot m \\ 62 &= 2^5 \cdot 1.11110_2 &&= 2^E \cdot m \end{aligned}$$

- The representation is written in radix β; here $\beta = 2$.
- The number of digits is the precision k.
- The set of floating-point numbers with precision k is denoted by $\mathbb{F}k$.

IEEE 754 standard

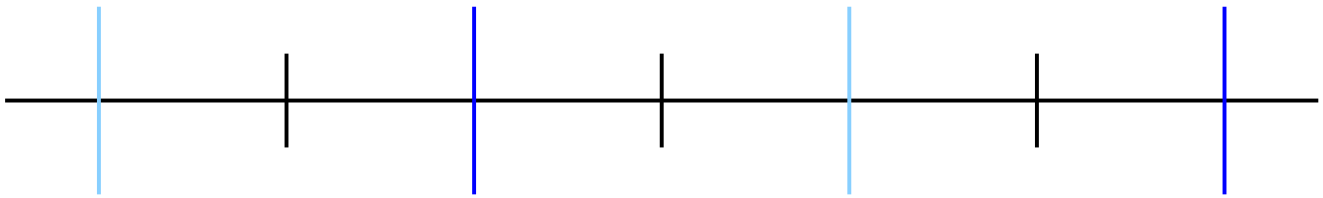
- Computation with floating-point numbers are specified by IEEE 754 standard
- Several formats $\mathbb{F}k$ are specified

Double precision $\mathbb{F}53$

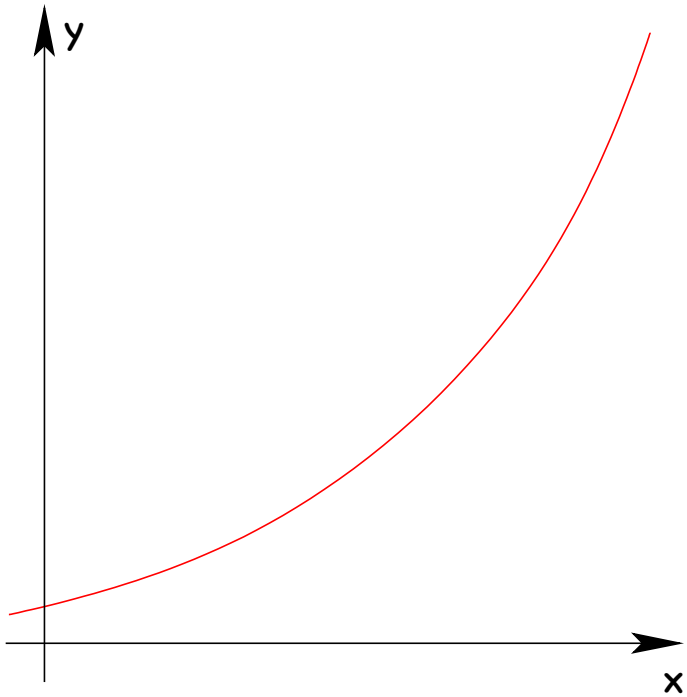


- A rounding in IEEE 754 $\circ_k$ is an application $\mathbb{R} \to \mathbb{F}k$:

$$x \mapsto \circ_k(x)$$



An elementary function f on a computer



$$\forall x \in \mathbb{F}k, \qquad x \xrightarrow{f} f(x) \xrightarrow{\circ_k} \circ_k(f(x))$$

Disclaimer

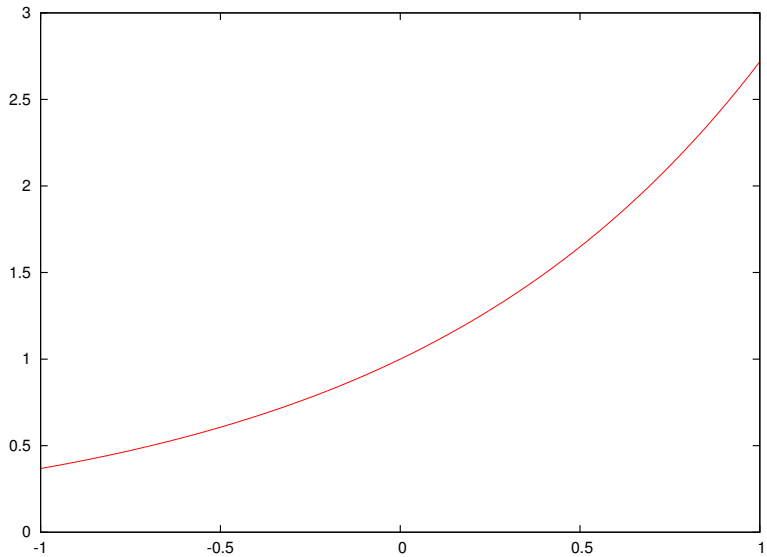
- No multi-precision
  - We know the precision of the input/output format
  - We implement exp for example in binary64 with 53 bits
  - The implementation of exp for k bits in output is more complicated.
    - See MPFR library for that.

- No special techniques developed for hardware
  - The `libm` are often implemented in software
  - Some processors may have some specific units
    - This is less and less used in the general case
    - When it is used, it is Intel/AMD and it is like in software
  - There are special algorithms for hardware (CORDIC)

Outline

Polynomial approximation

- How to implement for example $f = \exp$ ...
- ... on a restricted domain first, for example $I = [-1; 1]$?



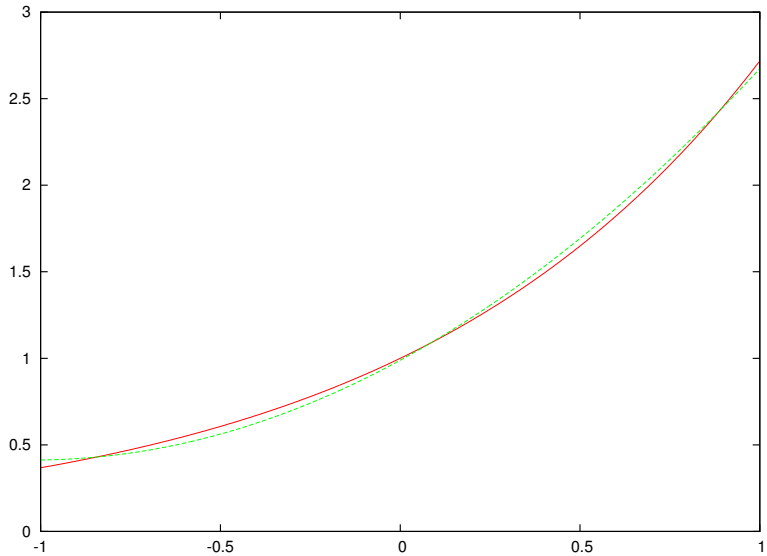
- Solution: replace f by an approximation polynomial p
- Choice for polynomials:
  - Weierstrass theorem tells us it always exists
  - Taylor polynomials
  - Polynomials minimizing the maximal error on the domain

Why choosing polynomials?

- Why choosing polynomials and not other types of approximations?
  - Why not continued fractions (truncated)?
  - Why not approximation with other basis functions?
- An answer (only) purely technological
  - Fast hardware $+$, $\times$ and FMA
  - Less performance for division (19 cycles on Intel Core i5/i7)
  - Possibility to compute the rounding error for $+$ and $\times$ (and /) ... but not for sin and log
- An answer less categorical
  - Some functions are not well approximated by polynomials: asin
  - We talk about software, but on hardware things can be different.
  - The toolchain is not as comprehensive for other approaches.

Need for an argument reduction

- Need to implement f on the whole range of floating-point numbers
- In binary64, $f = \exp$ is defined on $I = [-744.5; 709]$
- The polynomial approximation is not sufficient:



- When the domain is large, approximation error becomes very large for a given degree
- otherwise: we need to increase the degree to become very large

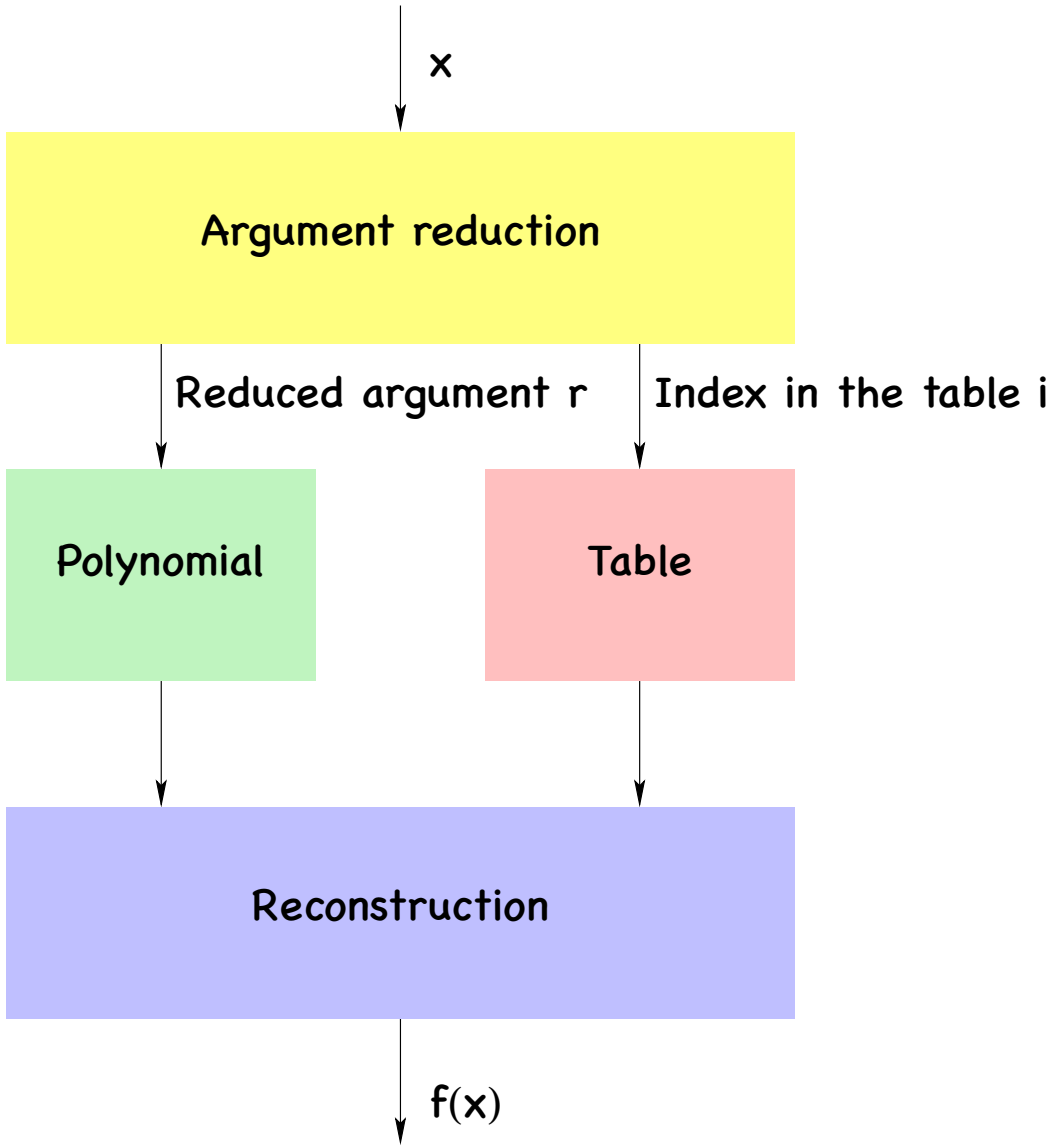
Reduction and tabulation

- Argument reduction brings everything into a small domain
  - Algebraic properties of the function
    - Periodicity: sin
    - Self-similarity: exp
    - Symmetry: asin
  - Semi-logarithmic feature of floating-point format
    - Somehow, convertToInteger and exp are very similar
  - If nothing else: split the domain
- Example for exp:

$$e^x = 2^{\frac{x}{\log 2}} = 2^{\left\lfloor \frac{x}{\log 2} \right\rceil} \cdot 2^{\frac{x}{\log 2} - \left\lfloor \frac{x}{\log 2} \right\rceil} = 2^E \cdot e^{x - E \log 2} = 2^E \cdot e^r$$

- The reduction is often linked to a tabulation
  - Tabulation: pre-computation of a function g at some discrete points
  - Reduction: the polynomial p must be valid only between those discrete points

Scheme for the computation of a function

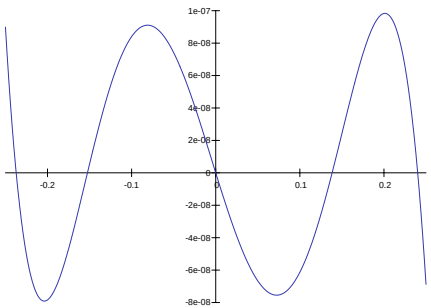


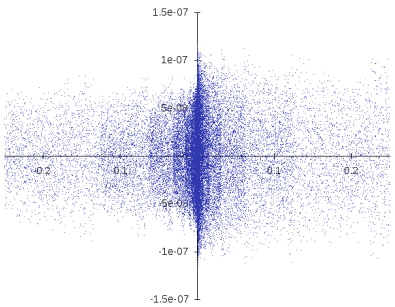
A toy exponential function

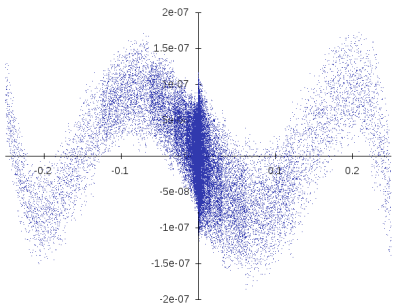
```
// A very crude "toy" implementation of exp(x)
//
// About 45 bits of accuracy. No checks for NaN, Inf whatsoever.
//
double Exp(double x) {
  double z, n, t, r, P, tbl, y;
  uint32_t E, idx, N;
  doubleCaster shiftedN, twoE;

  // Argument reduction
  z = x * TWO_4_RCP_LN_2;          // z = x * 2^4 * 1/ln(2)
  shiftedN.d = z + TWO_52_P_51;    // shiftedN.d = nearestint(z) + 2^52 + 2^51
  n = shiftedN.d □ TWO_52_P_51;    // n = nearestint(z) as double
  N = shiftedN.i[LO];              // N = nearestint(z) as integer
  E = N >> 4;                      // E = floor(2^□4 * N)
  idx = N & 0x0f;                  // idx = N □ E * 2^4
  t = n * TWO_M_4_LN_2;            // t = n * 2^□4 * ln(2)
  r = x □ t;                       // r = x □ t

  // Polynomial approximation        p(r) approximates exp(r)
  P = c0 + r * (c1 + r * (c2 + r * (c3 + r * (c4 + r * c5))));

  // Table access
  tbl = table[idx];                // tbl = 2^(2^□4 * idx)

  // Reconstruction
  twoE.i[HI] = (E + 1023) << 20;
  twoE.i[LO] = 0;                  // twoE.d = 2^E
  y = twoE.d * (tbl * P);          // y = 2^E * tbl * P

  return y;
}
```

Sources of errors

- Approximations with floating-point numbers, rounding errors







- Five principal sources of errors:
  - Error in the argument reduction
  - Error in the entries of the table
  - Approximation error $\|p/f - 1\|_\infty$
  - Error in the polynomial evaluation $|P(x)/p(x) - 1|$
  - Error in the reconstruction
- Combination of those errors to get the global error

Outline

Problem of polynomial approximation

- We have a function $f : \mathbb{R} \to \mathbb{R}$ ...
- on a small domain $I = [a; b] \subset \mathbb{R}$.
- We are given a targeted error $\bar{\varepsilon}$ we do not want to exceed.
- We are looking for a polynomial p of degree n minimal such that the error $\frac{p}{f} - 1$ is bounded (in absolute value) by $\bar{\varepsilon}$.

Example:

- Function: $f = \exp$
- Domain: $I = \left[-\frac{1}{2}; \frac{1}{2}\right]$
- Targeted error: $\bar{\varepsilon} = 2^{-5}$ (5 bits of accuracy)
- $p(x) = 1 + x + \frac{1}{2}x^2 \quad n = 2$
- $\left\| \frac{p}{f} - 1 \right\|_\infty^I \approx 3.05 \cdot 10^{-2} < 2^{-5}$

Problem of polynomial approximation - 2

- In other words, we have to
  - fix a degree n we hope to be sufficient
  - compute some coefficients c_i of a polynomial

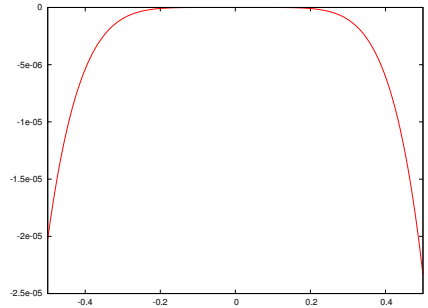
$$p(x) = \sum_{i=0}^{n} c_i x^i$$

  - look at the error $\frac{p}{f} - 1$
  - re-begin with a larger n if the error is not small enough

How can we compute the coefficients c_i of the polynomial p?

Taylor polynomials

- First idea:
  - Let us take $x_0 \in I$
  - $f(x_0)$ approximates $f(x)$ on I
  - $f(x_0) + (x - x_0) \cdot f'(x_0)$ is better
  - $f(x_0) + (x - x_0) \cdot f'(x_0) + \frac{1}{2} \cdot (x - x_0)^2 \cdot f''(x_0)$ is even better...



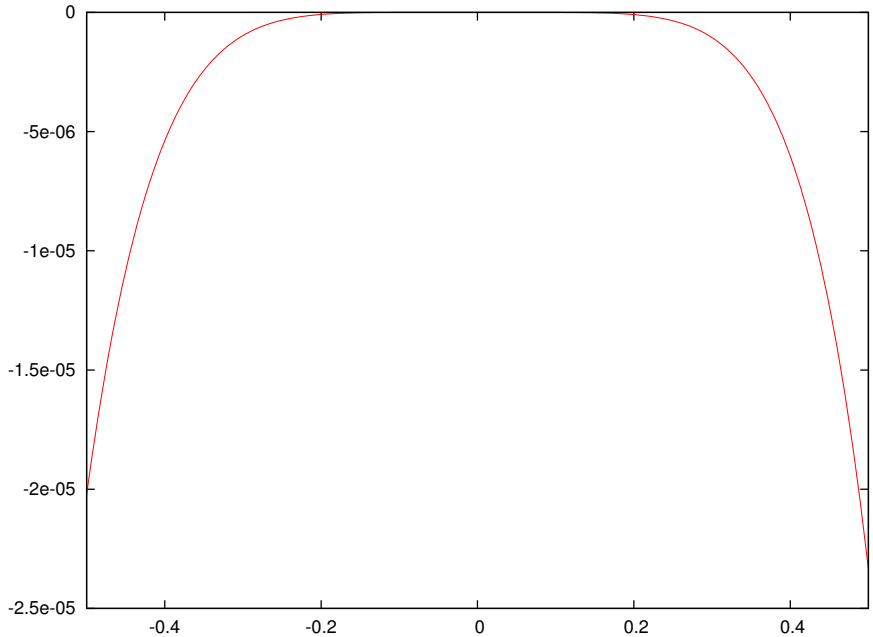
- This is the idea of Taylor polynomials at x_0:

$$f(x) = \underbrace{\sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!} \cdot (x - x_0)^i}_{=:p(x)} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!} \cdot (x - x_0)^{n+1}}_{\text{Lagrange remainder resp. Error}}$$

- Functions are defined by simple differential equations:
  - Taylor polynomials known for the functions in a `libm` library

Problems with Taylor polynomials

Error of Taylor polynomial p of degree 5 with respect to f = exp:



- Taylor polynomial approximates the function the best at the point x_0: indeed the error vanishes at x_0
- The error blow up at the boundary of the domain I
- $\Rightarrow$ it would be necessary for p to approximate f in the whole domain I, or at least the error vanishes several times

Interpolation polynomial

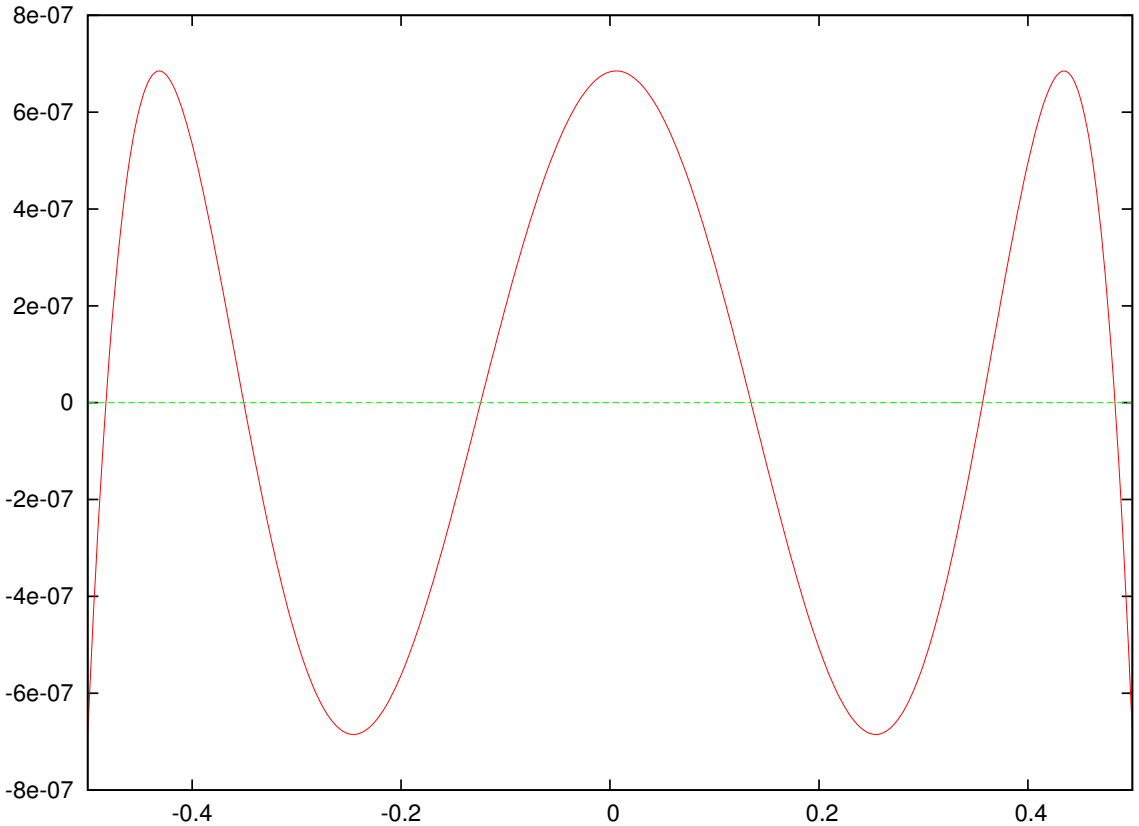
- A polynomial of degree n
  - $n+1$ coefficients
  - $n+1$ degree of freedom
  - defined by $n+1$ points $(x_j, p(x_j))$

- In order for the error $p(x) - f(x)$ to vanish at points x_j, it is sufficient to have $p(x_j) = f(x_j)$.
- The polynomial p interpolates the function f at x_j

- An interpolation polynomial of f is defined by $n+1$ values x_j
  - One value by degree of freedom
  - The ordinates $f(x_j)$ are clear for f and x_j.

Example of interpolation

Error of interpolation polynomial p with respect to f = exp:



- The error vanishes at the interpolation points
- By choosing the points, we can constrain the error to oscillate

Computation of interpolation polynomial

How can we compute an interpolation polynomial?

- We have $p(x_j) = \sum_{i=0}^{n} c_i x_j^i = f(x_j)$ for $n+1$ points x_j
- In other words, we have

$$\underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & \cdots & x_1^n \\ 1 & x_2 & \ddots & & & \vdots \\ \vdots & \vdots & & x_j^i & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ 1 & x_{n+1} & \cdots & \cdots & \cdots & x_{n+1}^n \end{pmatrix}}_{=:A} \cdot \underbrace{\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ \vdots \\ c_n \end{pmatrix}}_{=:\vec{p}} = \underbrace{\begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ \vdots \\ f(x_{n+1}) \end{pmatrix}}_{=:\vec{f}}$$

- We know the Vandermonde matrix A and the vector $\vec{f}$.
  - the solution of a linear system gives the coefficients of p
  - Moreover: there are special algorithms for such kind of linear systems

Going around in circles ...

- We know the RHS $\vec{f}$.
- So we can compute $f(x_j)$ for all x_j.
- Strange? we were implementing a function to compute f!
- This is a problem:
  - We first need to write a code to evaluate $f(x_j)$
  - Then, we can construct the interpolation polynomials in order to evaluate f everywhere.
- In practice:
  - We write multi-precision procedures for functions
  - Based on Taylor polynomials which are inefficient
  - Then, we use those codes to develop a `libm`

Which points to interpolate?

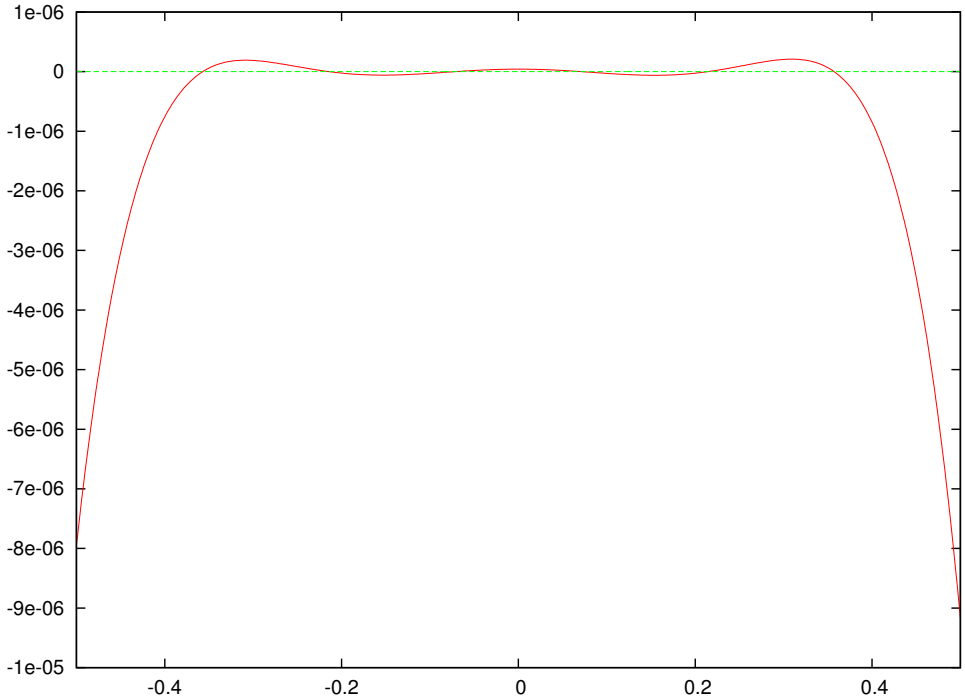
- The choice of points x_j has an influence on the error:
  Let p be an interpolation polynomial of f at points x_j.
  Then we have

$$f(x) = p(x) + \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{j=0}^{n} (x - x_j)$$

- The equidistant points are not optimal
- Tchebychev tried to minimize the error
- Remez finally provided an iterative algorithm
  - The algorithm provides the polynomial with minimal error with maximum norm
  - It only choose the best interpolation points

Equidistant points

Error of interpolation p with $f = \exp$ at equidistant points:

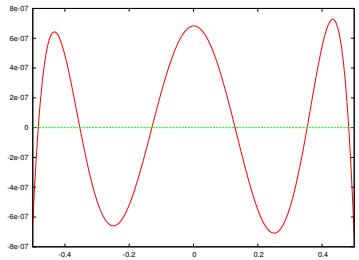


- It is better than Taylor: error 2.5 times lower
- Error oscillates
- Error still blows up at the boundary
  $\Rightarrow$ We want to put more points near the boundary

Tchebychev points

- Interpolation error to minimize with $\|\cdot\|_\infty$ norm:

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{j=0}^{n} (x - x_j)$$



- P. L. Tchebychev: try to minimize

$$\left\| \prod_{j=0}^{n} (x - x_j) \right\|_\infty$$

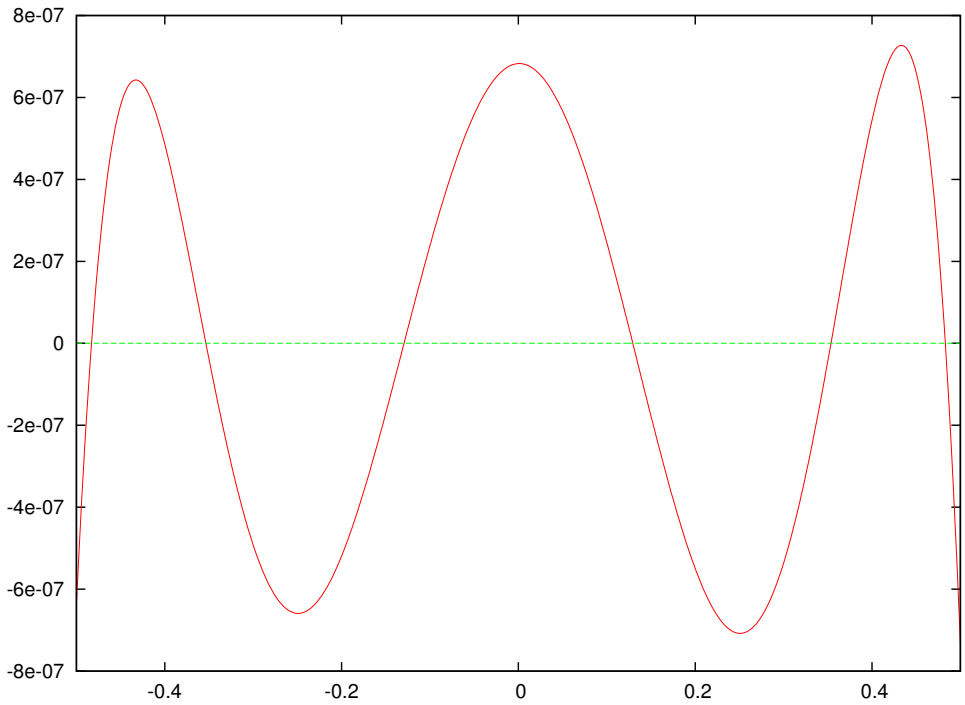
- Points for which it is the case in $I = [a; b]$:

$$x_j = a + \frac{b-a}{2} \cdot \left(\cos\left(\frac{2j-1}{2(n+1)} \right) + 1 \right)$$

- Those are Tchebychev interpolation points.

Tchebychev points - example

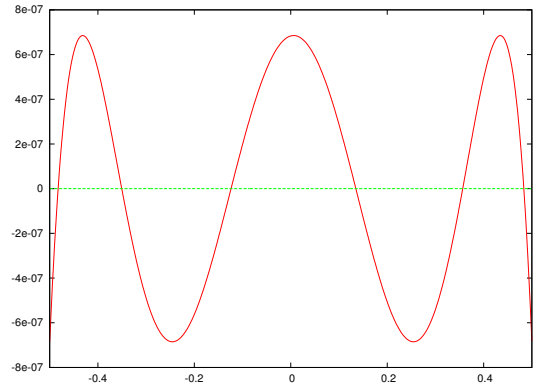
Error of interpolation polynomial p with f = exp at Tchebychev points:



- It is 10 times better than for equidistant points!
- It is not optimal yet: cf. picture
- There are functions for which the gap is very large

Remez polynomial

- Theorem (Tchebychev/La Vallée-Poussin):
  The approximation is optimal iff all the extrema of the error are at the same height.



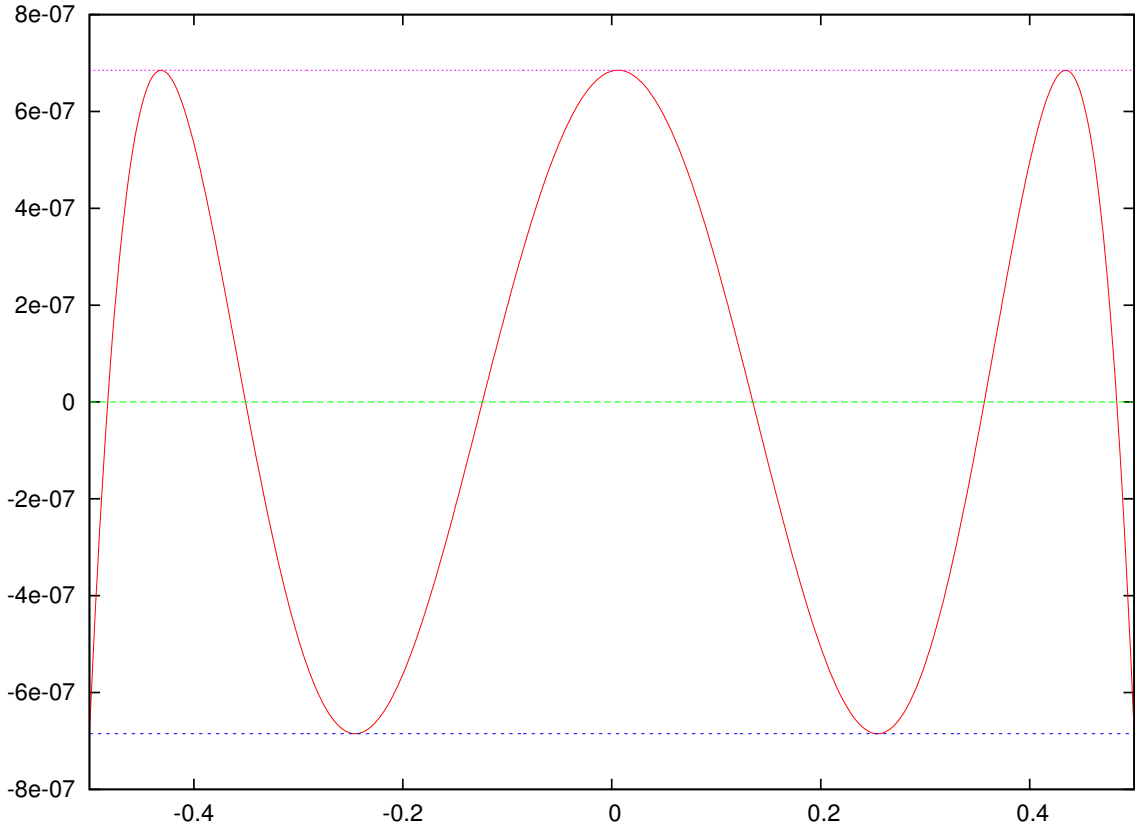
- E. Ya. Remez:
  - Interpolate directly $f(x) + (-1)^j \cdot \varepsilon$
    where ε is the height of the extrema we are looking for
  - While the extrema of the error are not at the same height, exchange interpolation points with points where the extrema are reached.

- This is Remez algorithm
- It converges toward the polynomial with minimal error measured with max norm
- That is why it is also called minimax algorithm

Example: Remez polynomial

Error of the Remez polynomial p of degree 5 with respect to

f = exp:



- The error is a little bit smaller than for Tchebychev
- All the extrema are at the same height

Still one problem $\rightarrow$ fpminimax

- Taylor, Tchebychev, Remez: polynomials with real coefficients
$$p(x) = \sum_{i=0}^{n} c_i x^i, \quad c_i \in \mathbb{R}$$
- On a computer, we only have floating-point numbers to store the c_i.
- If we only round numbers coefficient by coefficient, $\tilde{c}_i = \circ_{c_i}$, we destroy all the work of Remez algorithm.
  - The oscillations disappear.
  - The error blows up.
- It is important to find polynomials with floating-point coefficients
$$p(x) = \sum_{i=0}^{n} c_i x^i, \quad c_i \in \mathbb{F}k$$
- This is a discrete optimization problem
- Since 2006, there is a heuristic method based on lattice reduction.

Outline

Polynomials are not all ...

- The degree of the polynomials blows up when the size of the domain I increases.
- Idea:
  - For a periodic function like sin, it is not necessary to approximate it on $[2\,\pi; 4\,\pi]$ when we already know it on $[0; 2\,\pi]$.
  - A function like 2^x should benefit for the fact that we use a semi-logarithmic representation $2^E \cdot m$.
- Argument reduction: use
  - periodicity,
  - self-similarity,
  - symmetry
  - and the semi-logarithmic property of floating-point numbers

  to make the domain where the function has to be approximated as small as possible.

Increase of the degree - example

To approximate e^x by a polynomial with 24 bits of accuracy, we need

on	a degree n of
$[-\frac{1}{4}; \frac{1}{4}]$	5
$[-\frac{1}{2}; \frac{1}{2}]$	6
$[-1; 1]$	8
$[-2; 2]$	11
$[-16; 16]$	42
$[-32; 32]$	78
$[-256; 256]$	> 93

In double precision, we need to provide 53 bits of accuracy and cover the domain $[-745; 709]$...

Argument reduction

- Difficult to precisely define what argument reduction is.
- We try our best:

Definition (Argument reduction)

Given $f : \mathbb{F} \to \mathbb{F}$, an argument reduction consists in

- a reduction function $r : \mathbb{F} \to \mathbb{F}^n$ which is simple to calculate,
- a reduced function $g : \mathbb{F}^n \to \mathbb{F}^m$ whose projections can be computed easily either by polynomial approximation or by tabulation,
- and a reconstruction function $c : \mathbb{F}^m \to \mathbb{F}$ which can be computed easily.

These functions are such that

$$f(x) = c(g(r(x))) \quad \forall x \in \mathbb{F}.$$

Example 1: exp without table

We want to compute e^x. We have:

$$\begin{aligned} e^x &= 2^{\log_2(e)\cdot x} \\ &= 2^E \cdot 2^{\log_2(e)\cdot x - E} \\ &= 2^E \cdot 2^{\log_2(e)\cdot\left(x - \frac{1}{\log_2(e)}\cdot E\right)} \\ &= 2^E \cdot e^{x - \frac{1}{\log_2(e)}\cdot E} \\ &= 2^E \cdot e^r \end{aligned}$$

with $\underbrace{E = \lfloor \log_2(e)\cdot x \rceil \quad \text{and} \quad r = x - \frac{1}{\log_2(e)}\cdot E}_{\text{reduction function } r}$.

- r is bounded by $|r| \leq \frac{1}{2\cdot\log_2(e)} \approx 0.35$.
- So the function $g(r) = e^r$ is well approximated by a polynomial.
- The reconstruction c is simple, one has to multiply e^r by 2^E.

Tabulation

- Just with multiplications by 2^E and polynomial, it is difficult.
- Instead of computation, we could memorize.

- We try to coupling an argument reduction with the read in precomputed tables.

- Those tables are modelized by discrete reduced functions with hundred to thousand possible entries (otherwise, it is too expensive).

Example 2: exp with table

We still want to compute e^x. We have:

$$\begin{aligned} e^x &= 2^E \cdot 2^{k\cdot 2^{-w}-E} \cdot e^{x-\frac{1}{\log_2(e)}\cdot k\cdot 2^{-w}} \\ &= 2^E \cdot 2^i \cdot e^r \end{aligned}$$

with

$$\begin{aligned} k &= \lfloor \log_2(e) \cdot x \cdot 2^w \rceil \\ E &= \lfloor k \cdot 2^{-w} \rfloor \\ i &= k \cdot 2^{-w} - E \\ r &= x - \frac{1}{\log_2(e)} \cdot k \cdot 2^{-w} \end{aligned}$$

where w is the number of bits used for the index of the table.

- r is bounded by $|r| \leq 2^{-w} \cdot \frac{1}{2\cdot\log_2(e)}$.
- We read 2^i in a precomputed table.

How to find a good argument reduction?

- An argument reduction
  - needs the use of the properties of the function f
  - and of floating-point arithmetic
  - needs several functions which are easily computable or that can be put into tables
- Consequences: find a good argument reduction is difficult
- For that, we can use:
  - The Handbook of Mathematical Functions written in the 1950s by Abramowitz and Stegun
  - experiences and intuitions.

Relations Between Circular Functions

4.3.10 $\sin^2 z+\cos^2 z=1$

4.3.11 $\sec^2 z-\tan^2 z=1$

4.3.12 $\csc^2 z-\cot^2 z=1$

Negative Angle Formulas

4.3.13 $\sin(-z)=-\sin z$

4.3.14 $\cos(-z)=\cos z$

4.3.15 $\tan(-z)=-\tan z$

Addition Formulas

4.3.16 $\sin(z_1+z_2)=\sin z_1\cos z_2+\cos z_1\sin z_2$

4.3.29 $\sin 4z=8\cos^3 z\sin z-4\cos z\sin z$

4.3.30 $\cos 4z=8\cos^4 z-8\cos^2 z+1$

Products of Sines and Cosines

4.3.31 $2\sin z_1\sin z_2=\cos(z_1-z_2)-\cos(z_1+z_2)$

4.3.32 $2\cos z_1\cos z_2=\cos(z_1-z_2)+\cos(z_1+z_2)$

4.3.33 $2\sin z_1\cos z_2=\sin(z_1-z_2)+\sin(z_1+z_2)$

Addition and Subtraction of Two Circular Functions

4.3.34

$$\sin z_1+\sin z_2=2\sin\left(\frac{z_1+z_2}{2}\right)\cos\left(\frac{z_1-z_2}{2}\right)$$

Outline

Performances

- On typical Intel/AMD or IBM processors
  - a floating-point addition or multiplication takes 4 cycles
  - a division 20 cycles
  - a function call 15 cycles
  - a function like `exp` takes 40 cycles
- Within 40 cycles, one has to do lots of things:
  - an argument reduction
  - one or two read in a table
  - one polynomial evaluation
  - a reconstruction
  - dealing with NaN, $\pm\infty$, etc.
- Consequences:
  - We need to highly parallelize the code
  - The functions `libm` cannot call other functions
  - A clever manipulation of floating-point numbers

Example 1: computation of $\lfloor x \rceil$

- We want to compute $\lfloor x \rceil$:
  - that is to say the nearest integer to the floating-point number x.
  - We could use `rint(x)` or `floor(x + 0.5)`
  - It would just cost 30 cycles for the call
- Or we can notice that:
  - $\lfloor x \rceil$ is the rounding to nearest of a quantity for which ulp = 1
  - For x sufficiently small, $2^{52} + 2^{51} + x$ is such a quantity (for double precision)
  - It is just then needed to remove the shift $2^{52} + 2^{51}$.
- So we do:

```
double x, shifter, tmp, nearest;
x = ....                                  // computation that gives x
shifter = 6755399441055744.0;             // shifter = 2^52 + 2^51
tmp = x + shifter;                        // tmp = round(2^52 + 2^51 + x)
nearest = tmp □ shifter;                  // nearest = nearestint(x) □ Sterbe
```

Even better

- For the computation $\lfloor x \rceil$ in the previous slide
  - the input `x` is a `double` floating-point number
  - the output `nearest` is a `double` floating-point number
- Often, we want the output in a variable `n` of type `int`
  - We could do `n = (int) nearest;` in C
  - The compiler would insert a conversion instruction or a function call
  - $\Rightarrow$ it would take plenty of cycles
- But in fact, with IEEE 754-2008 standard,
  - the unit on the last place is stored at the end in memory
  - before, there is the sign, the exponent and the other bits of the significand
- So we can do:

```
double x, shifter, tmp;
int n;
x = ....
shifter = 6755399441055744.0;
tmp = x + shifter;
n = (*((unsigned long long int *) &tmp)
     & 0xffffffff);

// computation that gives x
// shifter = 2^52 + 2^51
// tmp = round(2^52 + 2^51 + x)

// n = nearestint(x)
```

Example 2: multiplication by 2^E

- We often need to
  - decompose a floating-point number x into exponent E and significand m
  - reconstruct it knowing E and m
  - or multiply by 2^E

- The construction of 2^E on a variable in `double` with a variable `int E` can be done as follows;

```
int E;
double x, twoE;
unsigned long long int tmp;
tmp = E + 1023;                 // add double prec. bias to signed expon
tmp <<= 52;                     // shift last exponent bit to the right
twoE = *((double *) &tmp); // make double variable out of it
x = x * twoE;                   // e.g. multiply x by 2^E
```

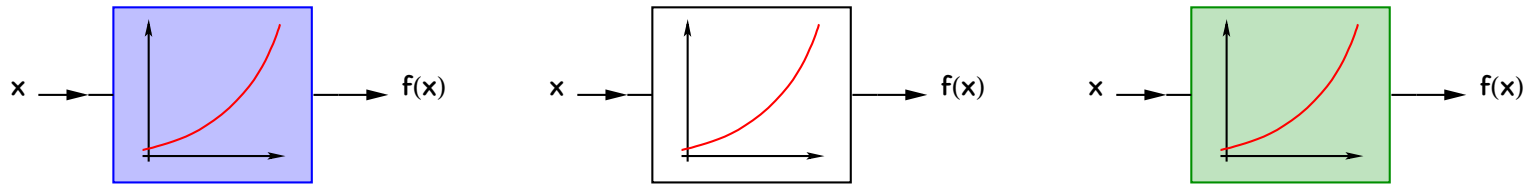
Our toy exponential

```
// A very crude "toy" implementation of exp(x)
//
// About 45 bits of accuracy. No checks for NaN, Inf whatsoever.
//
double Exp(double x) {
  double z, n, t, r, P, tbl, y;
  uint32_t E, idx, N;
  doubleCaster shiftedN, twoE;

  // Argument reduction
  z = x * TWO_4_RCP_LN_2;          // z = x * 2^4 * 1/ln(2)
  shiftedN.d = z + TWO_52_P_51;    // shiftedN.d = nearestint(z) + 2^52 + 2^51
  n = shiftedN.d □ TWO_52_P_51;    // n = nearestint(z) as double
  N = shiftedN.i[LO];              // N = nearestint(z) as integer
  E = N >> 4;                      // E = floor(2^□4 * N)
  idx = N & 0x0f;                  // idx = N □ E * 2^4
  t = n * TWO_M_4_LN_2;            // t = n * 2^□4 * ln(2)
  r = x □ t;                       // r = x □ t

  // Polynomial approximation        p(r) approximates exp(r)
  P = c0 + r * (c1 + r * (c2 + r * (c3 + r * (c4 + r * c5))));

  // Table access
  tbl = table[idx];                // tbl = 2^(2^□4 * idx)

  // Reconstruction
  twoE.i[HI] = (E + 1023) << 20;
  twoE.i[LO] = 0;                  // twoE.d = 2^E
  y = twoE.d * (tbl * P);          // y = 2^E * tbl * P

  return y;
}
```

Outline

Mathematical functions on a computer

- Developing mathematical functions on computers
  - Programs computing the value f(x) for a given x

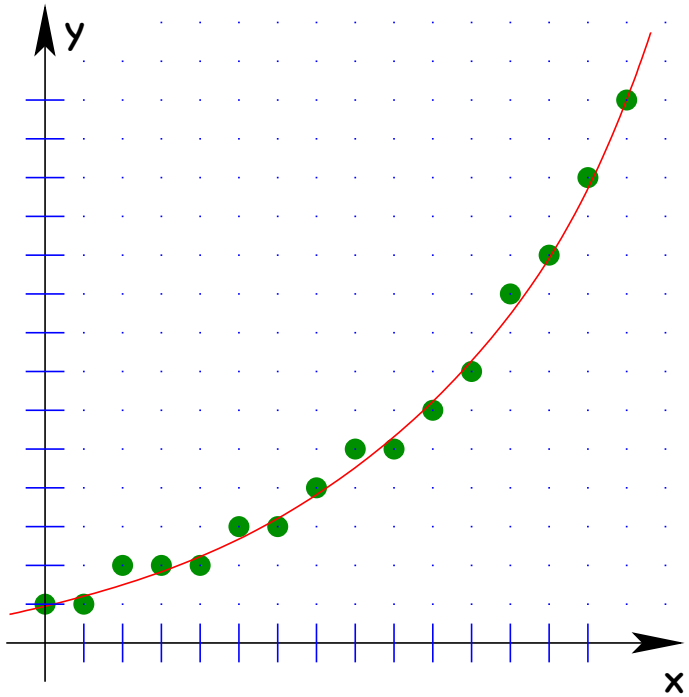


  - Different algorithms can be used to compute f

- The result of the computation must be the same everywhere.
  - deterministic results

- $\Rightarrow$ Correctly rounded mathematical functions

Correctly rounded result of a function f



$\forall x \in \mathbb{F}k, \quad x \xrightarrow{f} f(x) \xrightarrow{\circ_k} \circ_k(f(x))$

Definition

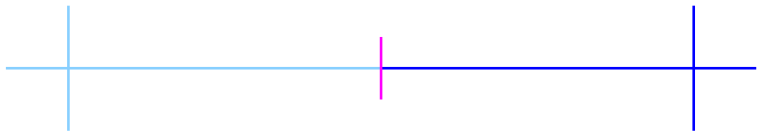
Let $f : \mathbb{R}^n \to \mathbb{R}$ be a function and $\circ_k : \mathbb{R} \to \mathbb{F}k$ be a rounding. The function $F : \mathbb{F}k^n \to \mathbb{F}k$ is a correctly rounding of f iff

$$\forall x \in \mathbb{F}k^n, \qquad F(x) = \circ_k(f(x))$$

Hard cases to round

Correctly rounding $\circ_k(f(x))$:

- Combination of two phenomena
  - The value of f(x), transcendental, cannot be well approximated.
  - The rounding $\circ_k$ changes at the rounding boundaries.



- The value $y = f(x)$ cannot be computed exactly.
- An approximation $\hat{y} = f(x) \cdot (1 + \varepsilon)$ can be computed.
- The precision $\overline{\varepsilon}$ necessary for the worst case is unknown.
- This is the Table Maker's Dilemma.

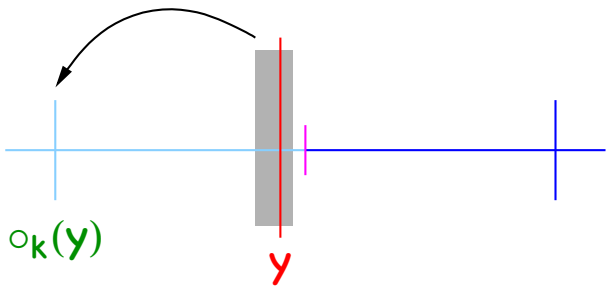
Accuracy in the worst cases

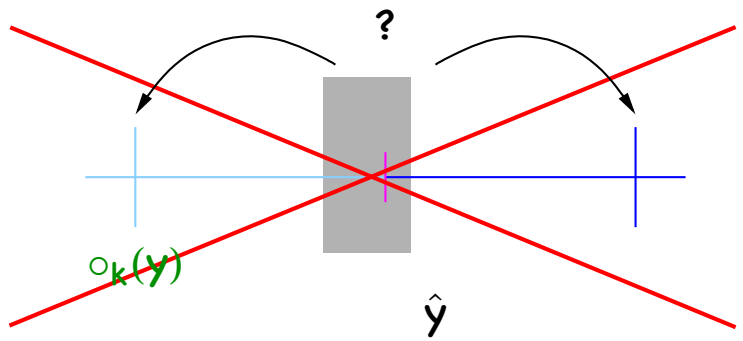
- Cannot directly compute correct rounding $F(x) = \circ(f(x))$
- Computation of an approximation $f(x) \cdot (1+\varepsilon)$ more precise than the relative distance to the worst case
- Use of the following lemma:

Lemma 1

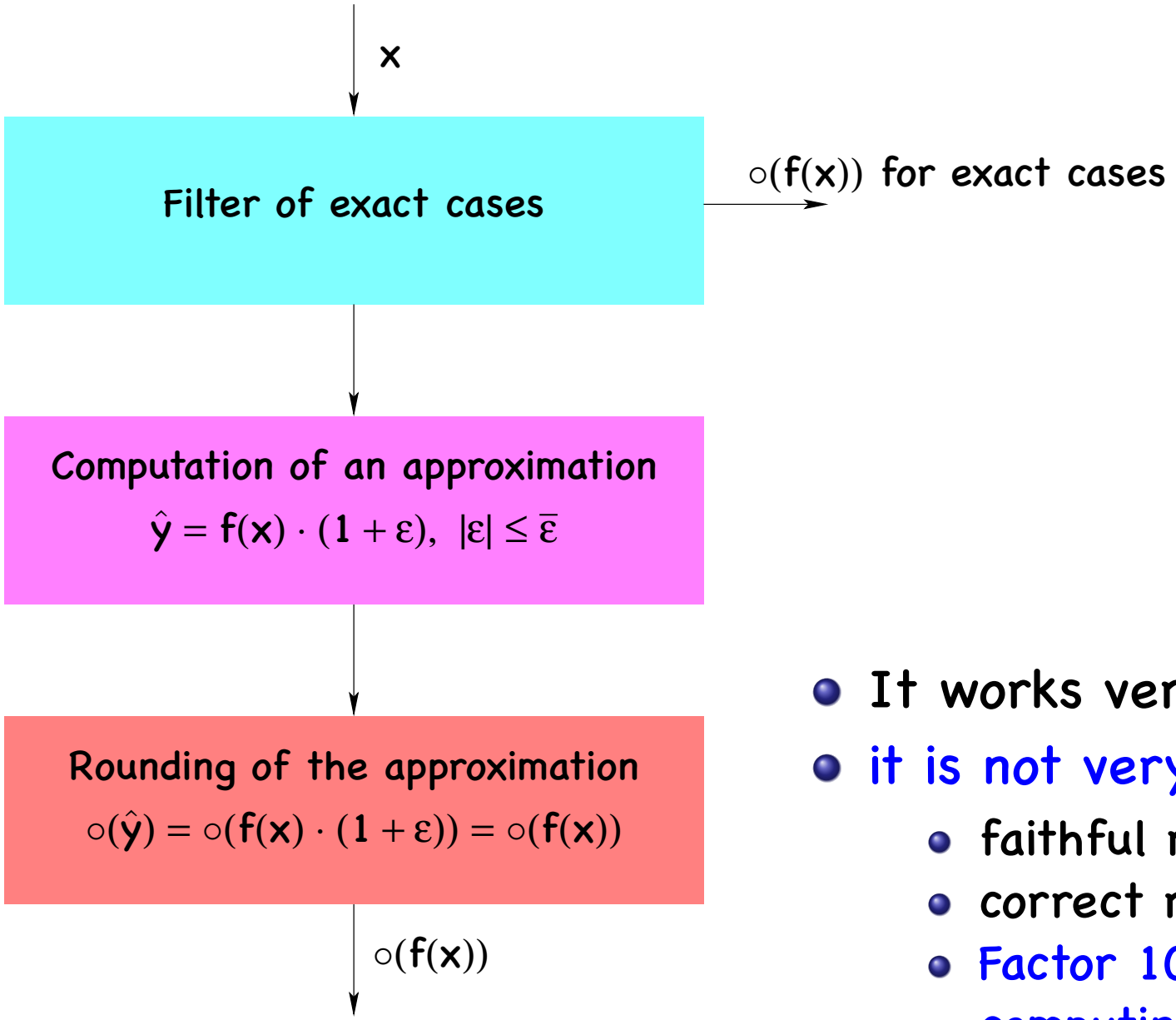
Let $f = \exp.$ and $\circ_{53} : \mathbb{R} \to \mathbb{F}53$ a rounding in double precision. Let $\mathbb{D} = \mathbb{F}53 \cap [-744.5; 709]$ be the domain of definition (in double precision) of f.
Then for all $x \in \mathbb{D}\backslash\{0\}$ and all $|\varepsilon| \leq 2^{-159}$,

$$\circ_{53}(f(x) \cdot (1+\varepsilon)) = \circ_{53}(f(x)).$$



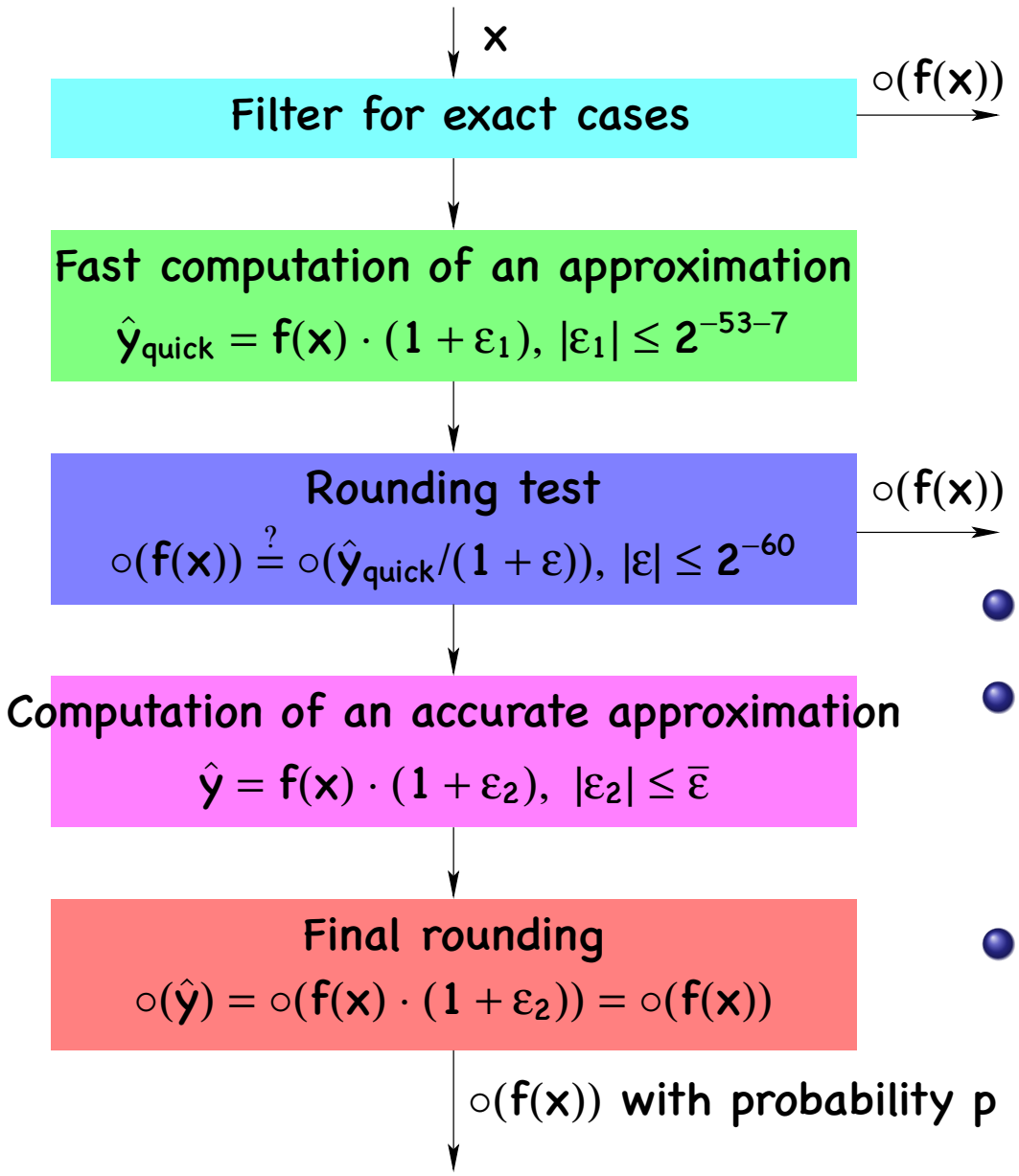


A simple scheme for correct rounding



- It works very well, but...
- it is not very efficient
  - faithful rounding: $53+5$ bit
  - correct rounding: 159 bits
  - Factor 10 in terms of computing time

Strategy in two steps



- Now, it is efficient
- Time average

$$T_{average} = T_{quick} + p \cdot T_{accurate}$$

- Typical values
  - $T_{quick} = 1.01 \cdot T_{faithful}$
  - $T_{accurate} = 10 \cdot T_{quick}$
  - $p = 2^{-7}$

Outline

Cost of a libm

- Cost of a function in terms of development
  - Beginning of CRLibm: 3-18 person months by function
  - Mean cost CRLibm: 1 person months
  - In industry: 3 person weeks
- Number of functions in a libm
  - IEEE 754-2008: 17 functions
  - Standard libm: around 70 functions
- Maintenance at a high cost
  - Several codes suitable for different architectures
  - Optimization for new architectures
  - Some codes are proved: synchronize proof and code
- Problems with correctly rounded libm
  - Computation of worst cases necessary for each function
  - Codes precisely analyzed and proved:
    - A work for specialists

The problem with proof

- Correct rounding implies a large precision.
- The cases where the maximal precision is needed are very rare.

example : exp, $\geq 2^{58}$ valid inputs

bits needed	number of cases
159	1
150	6
140	36
130	121
⋮	⋮

$\Rightarrow$ a detection of bug by sampling is not sufficient

Toward an automatic generation of libm

- Assembly language is tough $\Rightarrow$ compilers
- A libm is difficult to write $\Rightarrow$ automatic generation!
- MetaLibm [1] – a prototype generator:
  - Input:
    - A function $f : \mathbb{R} \to \mathbb{R}$
    - A small domain $dom = [a; b]$, after argument reduction
    - A targeted accuracy $\varepsilon \in \mathbb{R}^+$
  - Output:
    - A code F such that

$$\forall x \in dom, \quad \left| \frac{F(x) - f(x)}{f(x)} \right| \leq \varepsilon$$

    - A formal proof for that property
  - Ingredients:
    - Polynomial approximation with floating-point coefficients
    - Evaluation in multi-precision arithmetic (multi-double)

Generation of argument reduction

- Generate a code for exp in $\left[-2^{-5}; 2^{-5}\right]$ is not sufficient
- $\Rightarrow$ we need to generate codes for argument reduction
- Argument reduction – a major challenge:
  - A multitude of heterogeneous techniques
    - All the remarkable identities have to be coded?
    - How to choose the good one?
  - Codes very irregular
    - Mix of floating-point and fixed-point numbers
    - Tricks for manipulating floating-point numbers
  - Proof by hand difficult to understand
- But: many possibilities then
  - Research for compromise tabulation / approximation
  - Codes generated for any precisions

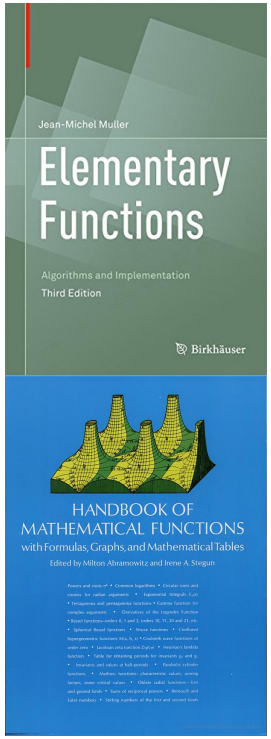
Outline

Conclusion

- The `libm` functions are based on simple concept:
  - An argument reduction to work on a small domain
  - A polynomial approximation
    - computed by Remez algorithm
  - Some lookup tables
  - A reconstruction
- At first, it seems to be simple
  - because it relies on old result (Taylor, Tchebychev and Remez)
  - But: it is needed to known floating-point arithmetic very well in order to provide efficient code to compute
    - `exp` in around 40 cycles
- Correctly rounding is possible but difficult to obtain
- An active research area is to write generators of `libm`s.

Some references

Muller,
Elementary Functions, Algorithms and Implementation,
Birkhäuser, 2016.



Abramowitz and Stegun,
Handbook of Mathematical Functions,
National Bureau of Standards.

And also

- P.-T. Tang, Table-driven implementation of the exponential function in IEEE floating-point arithmetic, TOMS, 15(2), 1989.
- E. W. Cheney, Introduction to Approximation Theory, New York, 2nd edition, 1982.