

Metalibm

Olga Kupriianova Christoph Lauter

PEQUAN team
Pierre and Marie CURIE University, Paris, France

GNU Tools Cauldron 2013
July 13, 2013



Outline

- 1 The libm introduction
- 2 Maths background
- 3 Problem statement
- 4 The Metalibm Project

The libm

- gives a set of maths functions (`exp`, `log`, `sin`, `cos`, `pow`, ...)
- supports important precisions (`float`, `double`, `long double`)

The existing implementations

- **glibc libm**
- libmcr by Sun^a
- crlibm by ENS-Lyon
- libultim by IBM
- Intel's proprietary mathimf

^a All other trademarks and copyrights are the property of their respective owners

The glibc libm

Why glibc libm?

The glibc libm

Why glibc libm?

- Math library running on all linux-powered machines



The glibc libm

Why glibc libm?

- Math library running on all linux-powered machines



- Written by different developer teams
- Some codes were written 20 years ago
- Strange naming conventions

```
double __ceil (s_ceil.c)
/* @(#)s_ceil.c 5.1 93/09/24 */
/*
*
* _____
* Copyright (C) 1993 by Sun Microsystems, Inc.
```

e_exp.c file

/*

* **IBM** *Accurate Mathematical Library*

* *written by International Business Machines Corp.*

* *Copyright (C) 2001-2013 Free Software Foundation, Inc.*


```

/*
* IBM Accurate Mathematical Library
* written by International Business Machines Corp.
* Copyright (C) 2001-2013 Free Software Foundation, Inc. */
/*****
*/
/* MODULE NAME: halfulp.c */
/*
*/
/* FUNCTIONS: halfulp */
/* FILES NEEDED: mydefs.h dla.h endian.h */
/* uroot.c */
/*
*/
/* Routine halfulp(double x, double y) computes  $x^y$  where */
/* result does not need rounding. If the result is closer */
/* to 0 than can be represented it returns 0. */
/* In the following cases the function does not compute */
/* anything and returns a negative number: */
/* 1. if the result needs rounding, */
/* 2. if y is outside the interval  $[0, 2^{20}-1]$ , */
/* 3. if x can be represented by  $x=2**n$  for some integer n */
/*****

```

The restrictions of libm

- Limited set of functions: trigonometric, exponentiation and logarithmic, hyperbolic, special functions, pseudo-random numbers
- Limited set of precisions (`float`, `double`, `long double`)
- The only one implementation, hard-coded long time ago
- No choice between implementation accuracies

Speed vs. accuracy compromise

Common wisdom

The more accurate you compute, the more expensive it gets

Speed vs. accuracy compromise

Common wisdom

The more accurate you compute, the more expensive it gets

Observation

The `binary64` format provides about 16 decimal digits, while physical measurements might use only 3!

Speed vs. accuracy compromise

Common wisdom

The more accurate you compute, the more expensive it gets

Observation

The `binary64` format provides about 16 decimal digits, while physical measurements might use only 3!

Compiler flag?

Why can't we have an option like

```
gcc -c code.c -fp-transcendental=15ulps
```

to get less accuracy when we don't need more

Outline

- 1 The libm introduction
- 2 Maths background**
- 3 Problem statement
- 4 The Metalibm Project

Some definitions

Unit in the Last Place - ulp

$\text{ulp}(x)$ is the gap between the two nearest floating-point numbers closest to x , even if x is one of them.

Kahan, 1960

Example

Assume that $x = 12.345$ ($0.12345 \cdot 10^2$),
and we have a FP arithmetic with base $\beta = 10$ and precision $p = 3$,
then $X = 0.123 \cdot 10^2$, so the absolute error is 0.045 , $\text{ulp}(x) = 10^{-1}$.

Flavors of functions

Flavor

A **flavor** is a particular specification of a function

Absolute error in ulps

$$|X - x| \leq \alpha \cdot \text{ulp}(x)$$

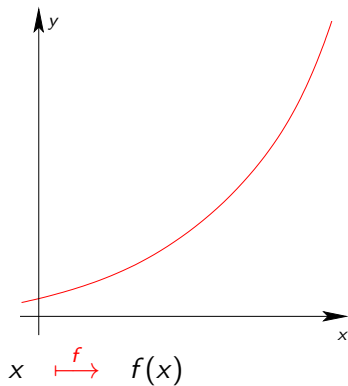
Relative Error

$$\left| \frac{X}{x} - 1 \right| \leq \alpha \cdot \beta^{-p+1}$$

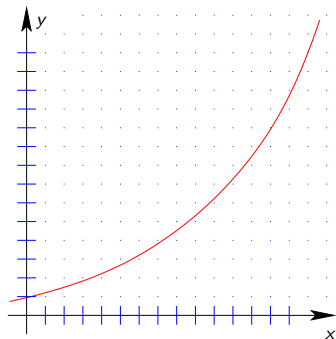
Correct Rounding

$$|X - x| \leq 0.5 \cdot \text{ulp}(x)$$

Correct rounding

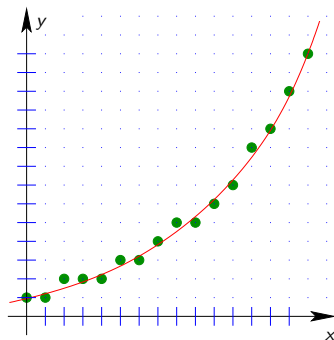


Correct rounding



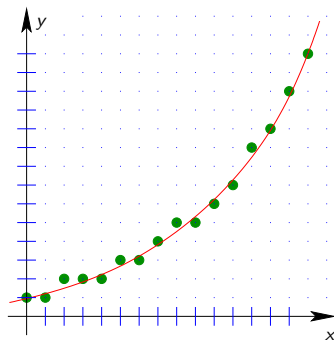
$$\forall x \in \mathbb{F}_p, \quad x \xrightarrow{f} f(x) \xrightarrow{\circ_p}$$

Correct rounding



$$\forall x \in \mathbb{F}_p, \quad x \xrightarrow{f} f(x) \xrightarrow{\circ_p} \circ_p(f(x))$$

Correct rounding



$$\forall x \in \mathbb{F}_p, \quad x \xrightarrow{f} f(x) \xrightarrow{\circ_p} \circ_p(f(x))$$

Definition

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be a function, $\circ_p : \mathbb{R} \rightarrow \mathbb{F}_p$ be rounding.

The function $F : \mathbb{F}_p^n \rightarrow \mathbb{F}_p$ is a correctly rounded version of f iff

$$\forall x \in \mathbb{F}_p^n, \quad F(x) = \circ_p(f(x))$$

Table Maker's Dilemma

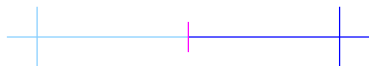
Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

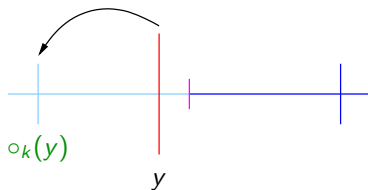


- The value $y = f(x)$ cannot be computed exactly

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

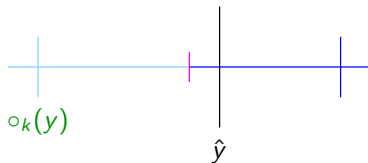


- The value $y = f(x)$ cannot be computed exactly

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

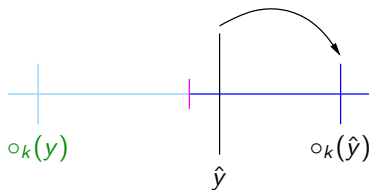


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

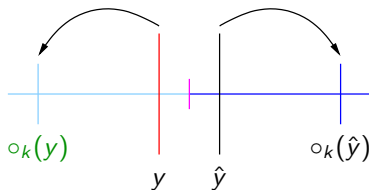


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

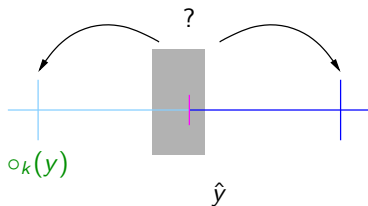


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

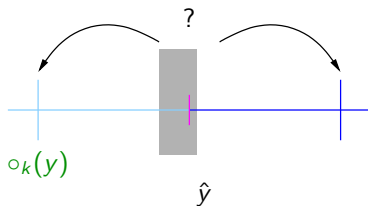


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

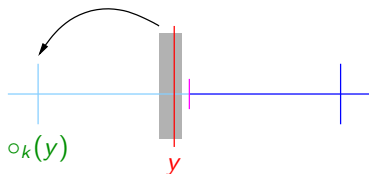


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders



- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Table Maker's Dilemma

Sometimes **huge** accuracy is needed

Runtime of function is not bounded or really huge

Table Maker's Dilemma

Table Maker's Dilemma

Sometimes **huge** accuracy is needed

Runtime of function is not bounded or really huge

Recommendations

- Precompute TMD worst cases
- Avoid correct rounding wherever we do not really need it

Outline

- 1 The libm introduction
- 2 Maths background
- 3 Problem statement**
- 4 The Metalibm Project

Different flavors of the functions

Precision	Ulp Error	Decimal Digits	Needed Accuracy	Verdict
Single	1 ulp	7	2^{-24}	fast
Single (CR)	0.5 ulps	7	2^{-50}	slow
Double	1 ulp	16	$2^{-53.5}$	fast
Double (CR)	0.5 ulps	16	2^{-150}	slow

Different flavors of the functions

Precision	Ulp Error	Decimal Digits	Needed Accuracy	Verdict
Single	1 ulp	7	2^{-24}	fast
Single (CR)	0.5 ulps	7	2^{-50}	slow
Double	1 ulp	16	$2^{-53.5}$	fast
Double (CR)	0.5 ulps	16	2^{-150}	slow

Not fast enough? Still more flavors

Different flavors of the functions

Precision	Ulp Error	Decimal Digits	Needed Accuracy	Verdict
Single	1 ulp	7	2^{-24}	fast
Single (CR)	0.5 ulps	7	2^{-50}	slow
Double	1 ulp	16	$2^{-53.5}$	fast
Double (CR)	0.5 ulps	16	2^{-150}	slow

Not fast enough? Still more flavors

Single	2^5 ulps	5	2^{-17}	faster
Single	2^{11} ulps	3	2^{-11}	the fastest
Double	2^{11} ulps	12	2^{-40}	faster
Double	2^{42} ulps	3	2^{-10}	the fastest

We should get even more flavors!

- set floating-point flags or not **2 possibilities**
- support all rounding modes or not: **4 possibilities**
 - save mode, perform computations, restore mode
 - perform computations to support all the rounding modes
 - perform integer-based computations

Still more flavors

We should get even more flavors!

- set floating-point flags or not **2 possibilities**
- support all rounding modes or not: **4 possibilities**
 - save mode, perform computations, restore mode
 - perform computations to support all the rounding modes
 - perform integer-based computations

⇒ Eight more flavors

Rough Computations

- 3 precisions
- ~ 50 functions in libm
- ~ 10 flavors for each precision
- 8 additional flavors

Rough Computations

- 3 precisions
- ~ 50 functions in libm
- ~ 10 flavors for each precision
- 8 additional flavors

I have to implement **12000** functions...

Rough Computations

- 3 precisions
- ~ 50 functions in libm
- ~ 10 flavors for each precision
- 8 additional flavors

I have to implement **12000** functions...

Each function implementation takes ~ 1 man-month

Rough Computations

- 3 precisions
- ~ 50 functions in libm
- ~ 10 flavors for each precision
- 8 additional flavors

I have to implement **12000** functions...

Each function implementation takes ~ 1 man-month

It will take me **1000** years to rewrite the libm

Outline

- 1 The libm introduction
- 2 Maths background
- 3 Problem statement
- 4 The Metalibm Project**

Solution

Metalibm

Write a tool that produces code for math functions

Analogy

assembly → compilers

code → code generators

```
.cfi_startproc
subq $8, %rsp
.cfi_def_cfa_offset 16
movl $52, %r8d
movl $37, %ecx
movl $15, %edx
movl $.LC0, %esi
movl $1, %edi
xorl %eax, %eax
call __printf_chk
xorl %eax, %eax
addq $8, %rsp
.cfi_def_cfa_offset 8
ret
.cfi_endproc
```

```
include <stdio.h>

int main() {
    int a, b, sum;
    a = 15;
    b = 37;
    sum = a + b;
    printf("%d + %d = %d\n", a, b,
           sum);
    return 0;
}
```

Metalibm prototype

Academic prototype

- Produces flexible math functions implementations
- Gives a function on a specified domain
- It's free and already available! Try it on <http://lipforge.ens-lyon.fr/www/metalibm/>

State of the art

- Supports almost all libm functions
- The documentation has to be finished
- We are thinking of the vectorizable implementations
- We are working on support of specific functions (e.g. dickman's, erf)

Input:

$$f(x) = \log(x)$$

$$x \in [0.5; 75]$$

$$\bar{\varepsilon} = 2^{-30}$$

Our work on glibc libm:

- Red Hat Inc. contacts established.
- Regular phone-meetings every 3-4 weeks
- Goals of this cooperation:

Short run. Some functions produced by metalibm.

Medium run. Different flavor functions.

To choose the flavor we'll probably need compiler flags or directives.

Long run. Generate everything for libm.

A carrot and a stick

Things that **complicate** all the task:

- Each math functions family needs a unique processing, error analysis.
- Correct rounding is hard to implement.

A carrot and a stick

Things that **complicate** all the task:

- Each math functions family needs a unique processing, error analysis.
- Correct rounding is hard to implement.

BUT

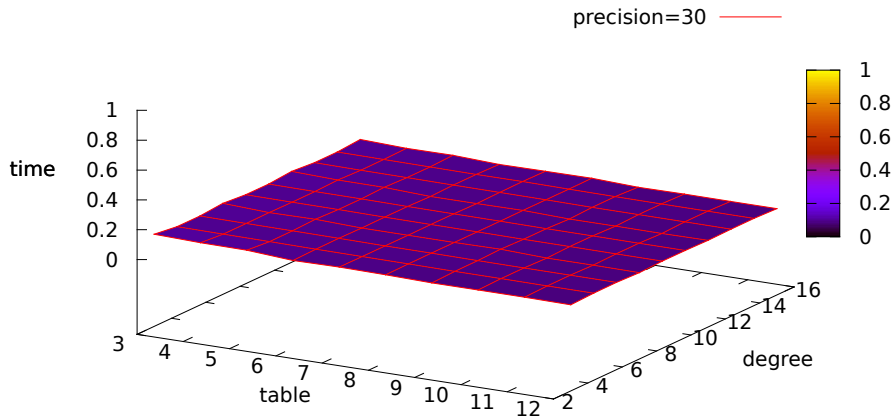
The complete metalibm gives some **bonuses**:

- needed accuracy / precision
- composite functions e.g. $\exp(\sin(x))$ ($\sin(x) \in [-1; 1]$)
- functions that are defined by differential equations
E.g. Dickman's function $\rho(u)$ is a continuous function that satisfies

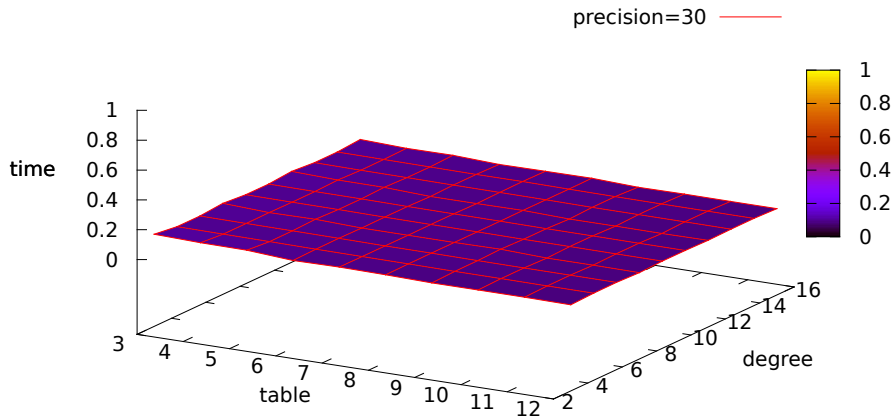
$$u\rho'(u) + \rho(u - 1) = 0$$

- automatization means more optimization

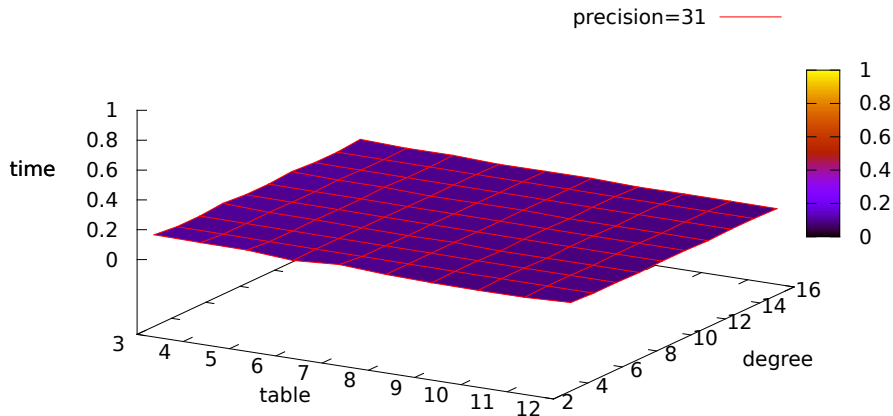
Timings for exponential function



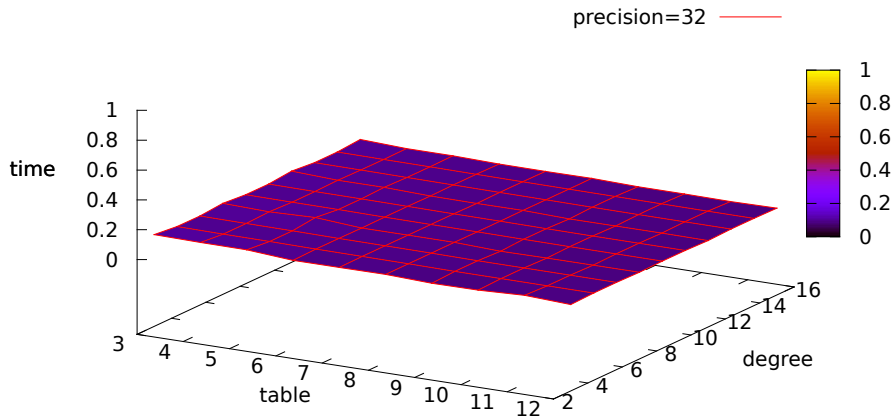
Timings for exponential function



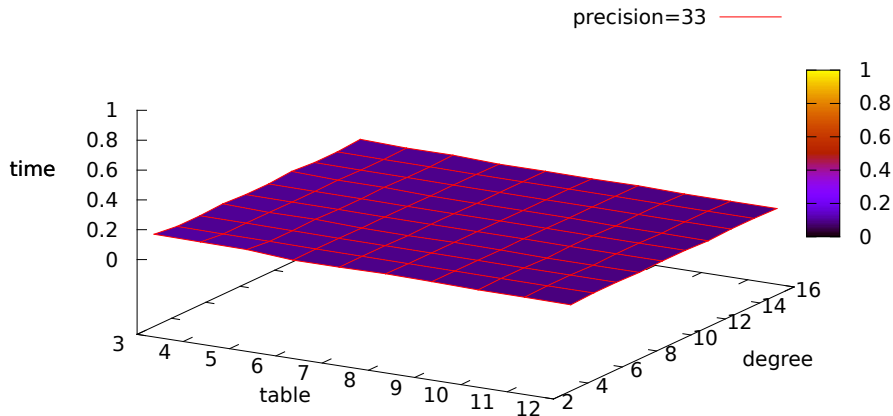
Timings for exponential function



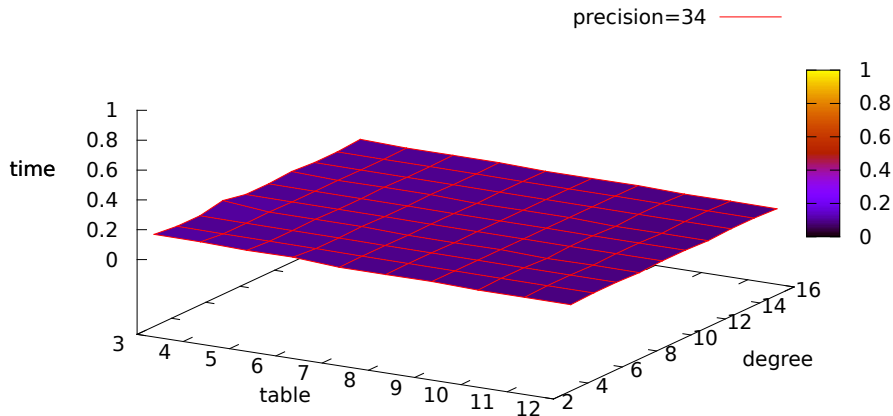
Timings for exponential function



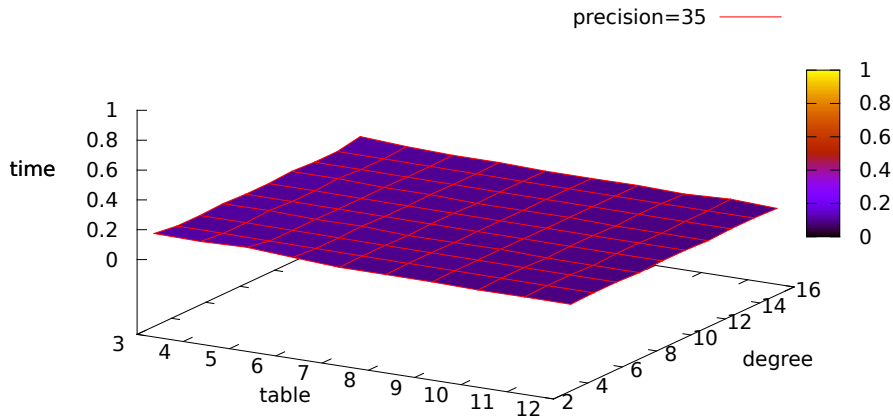
Timings for exponential function



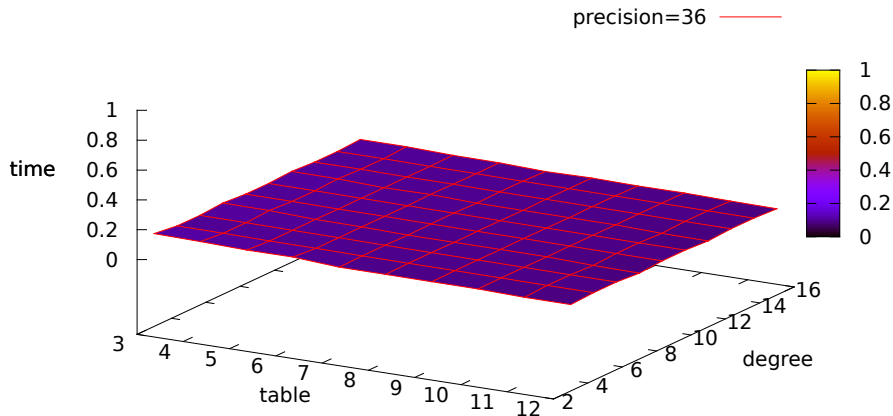
Timings for exponential function



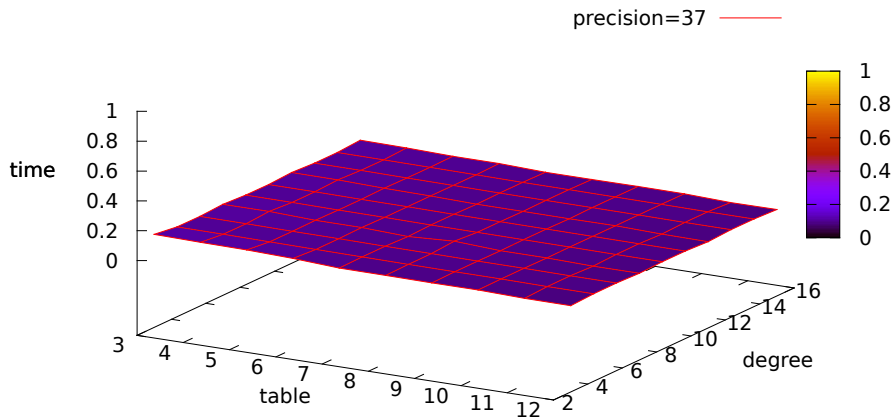
Timings for exponential function



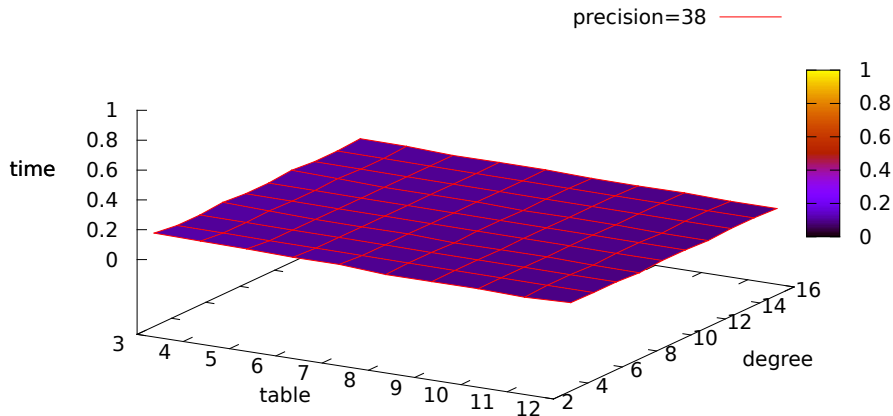
Timings for exponential function



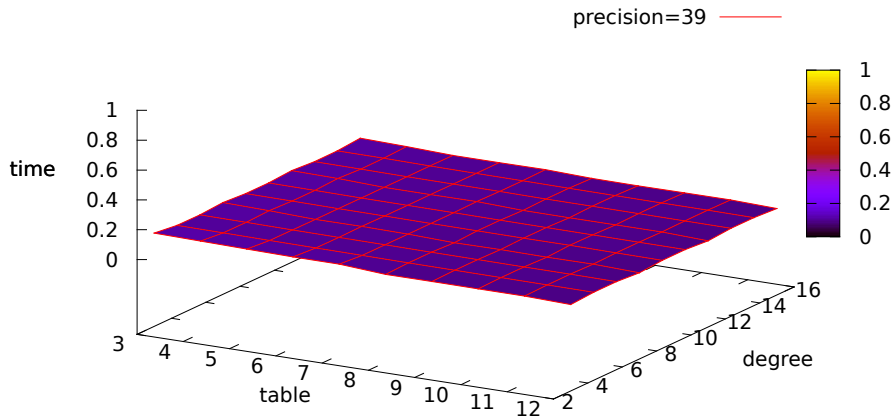
Timings for exponential function



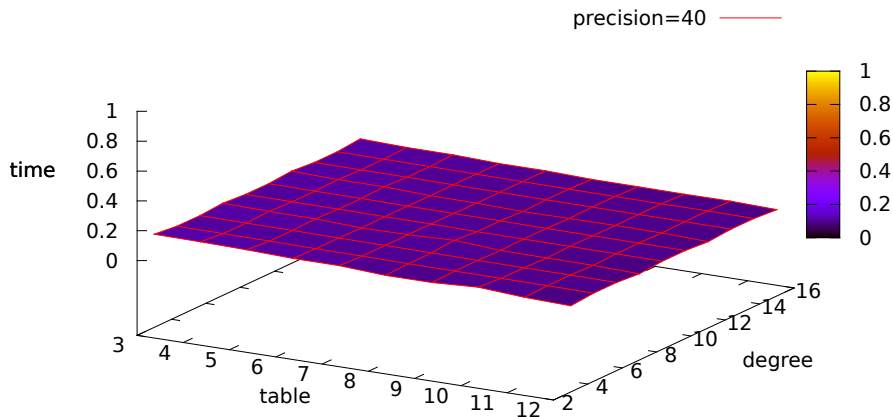
Timings for exponential function



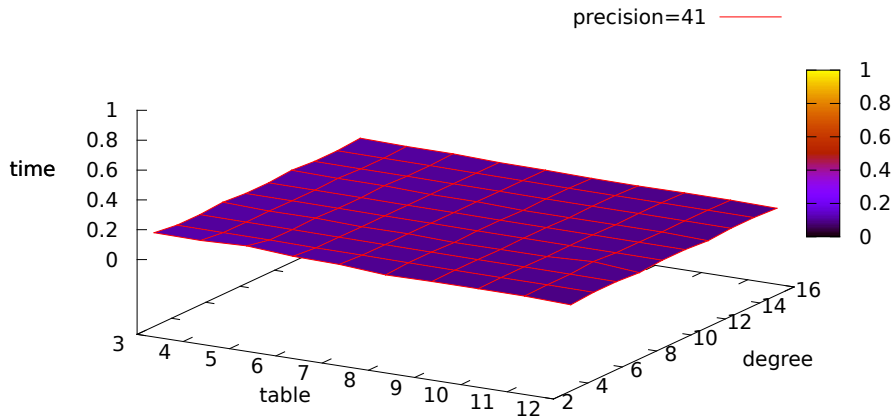
Timings for exponential function



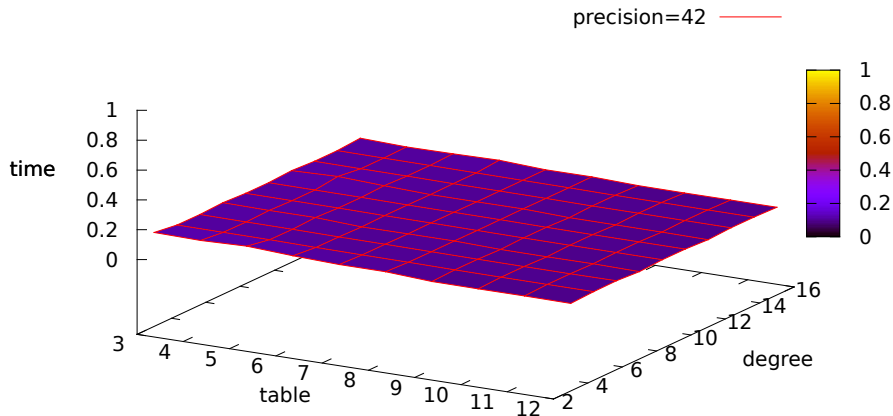
Timings for exponential function



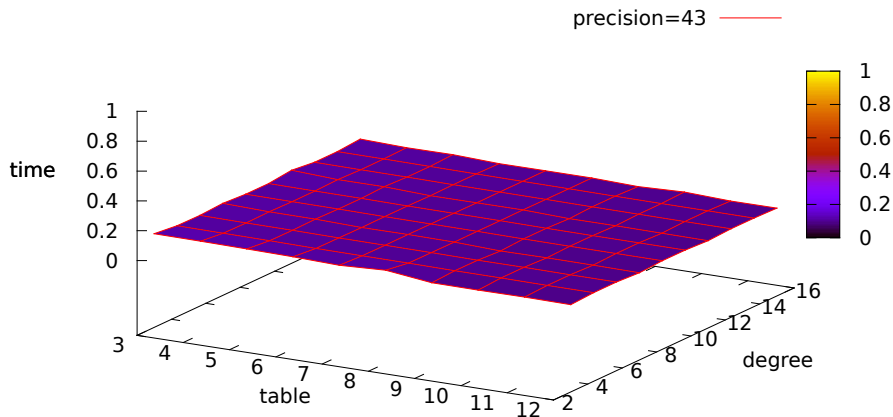
Timings for exponential function



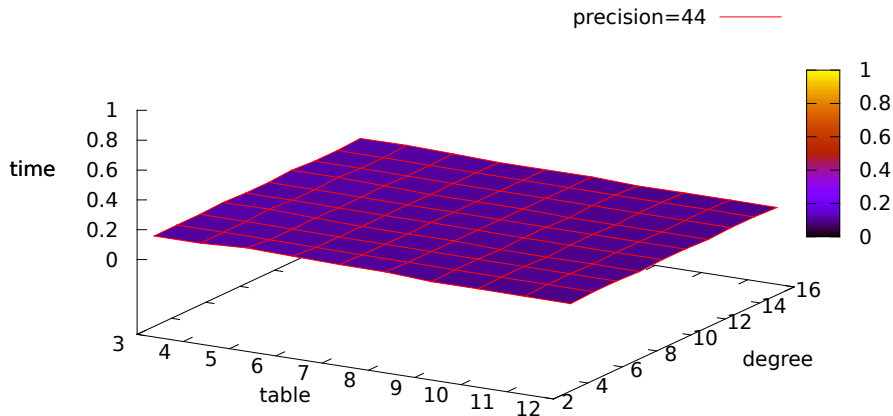
Timings for exponential function



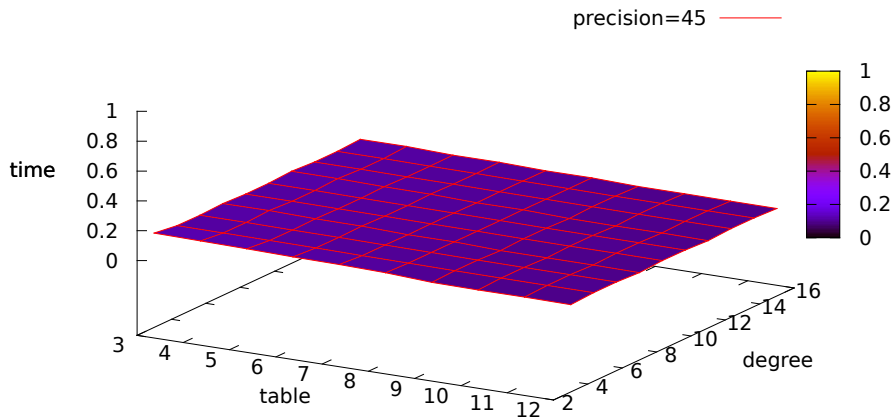
Timings for exponential function



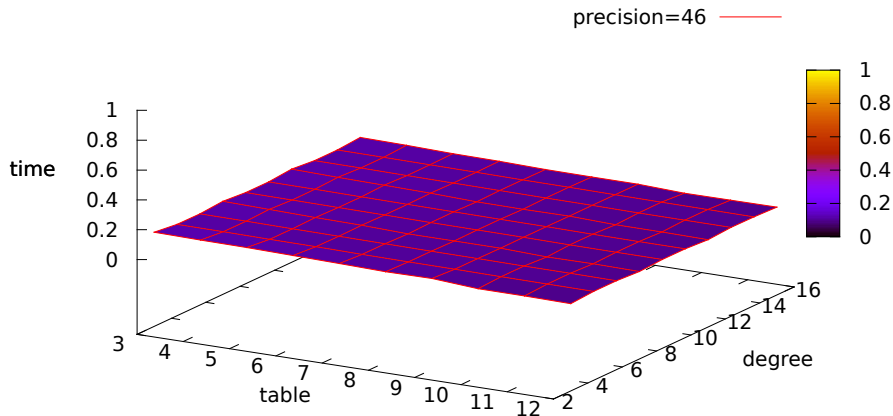
Timings for exponential function



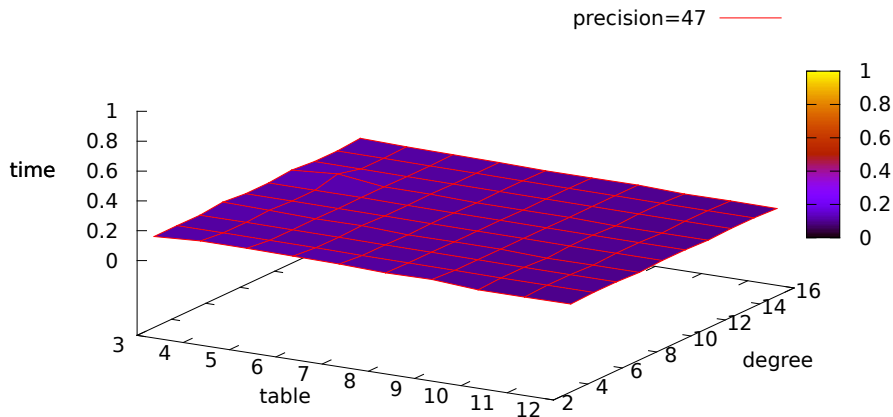
Timings for exponential function



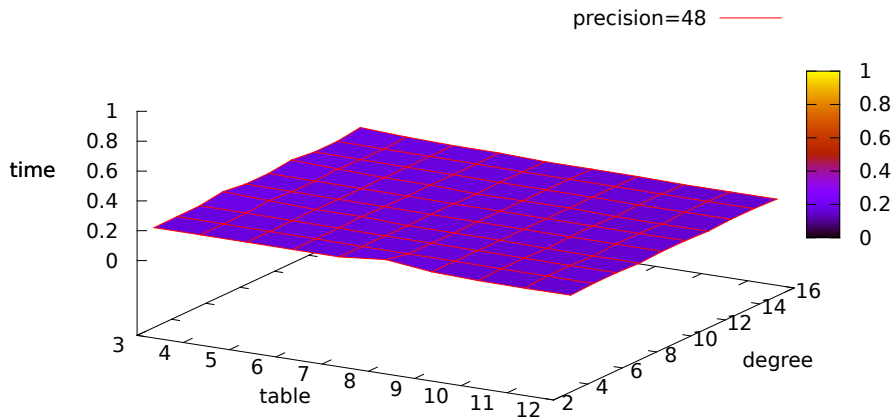
Timings for exponential function



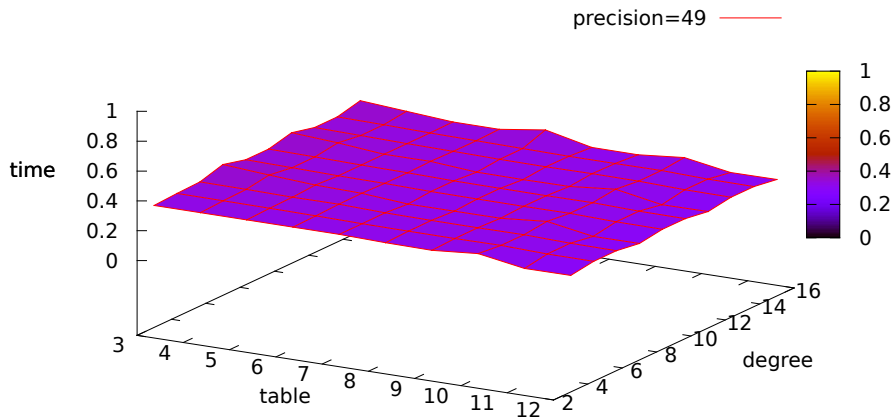
Timings for exponential function



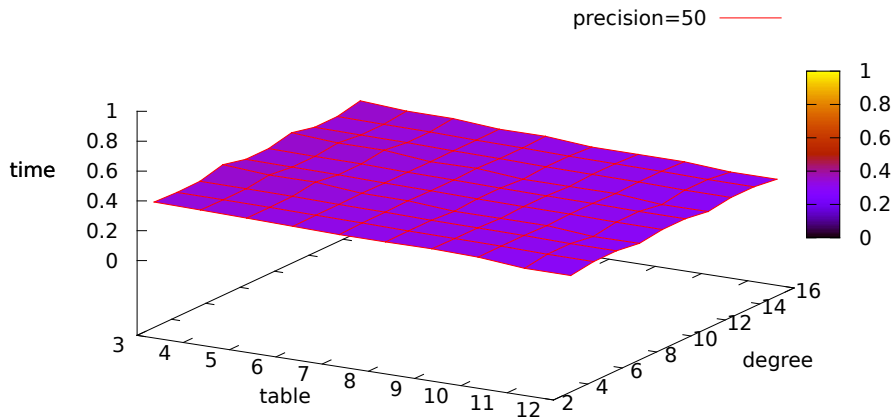
Timings for exponential function



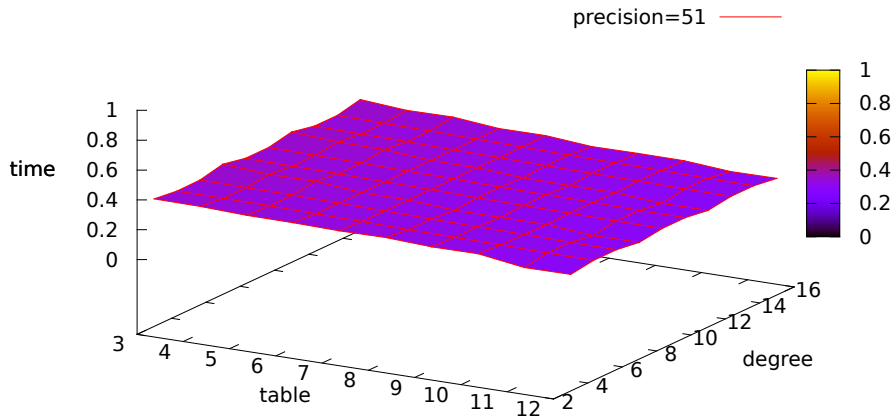
Timings for exponential function



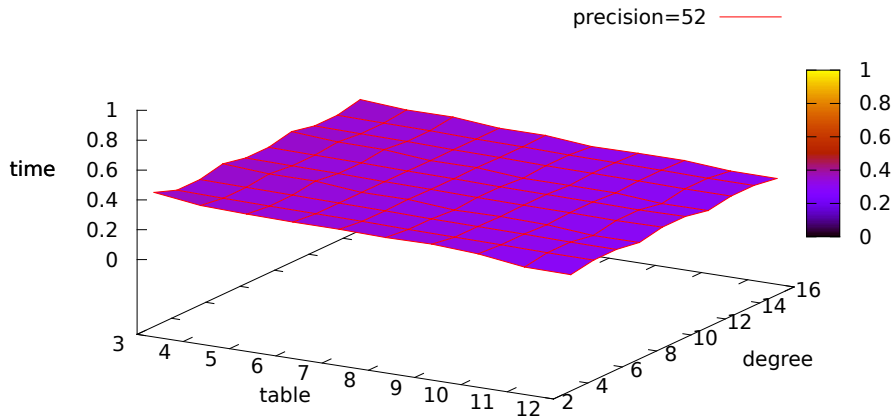
Timings for exponential function



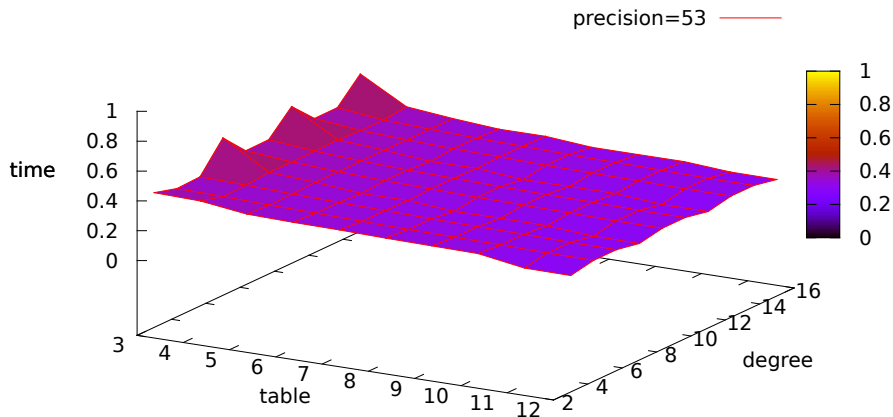
Timings for exponential function



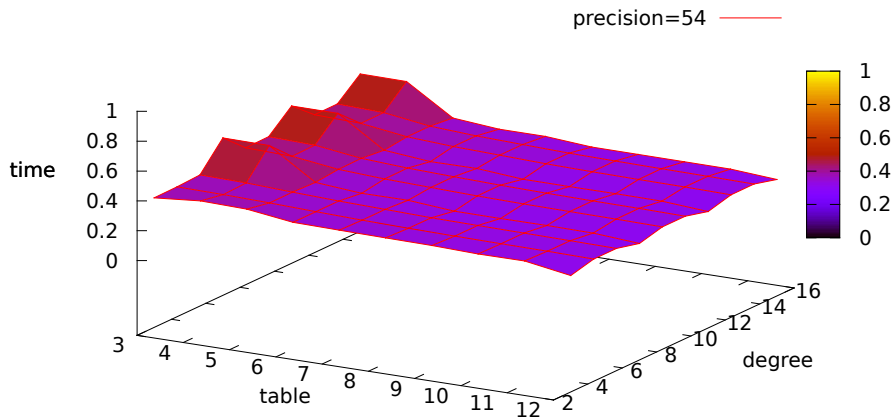
Timings for exponential function



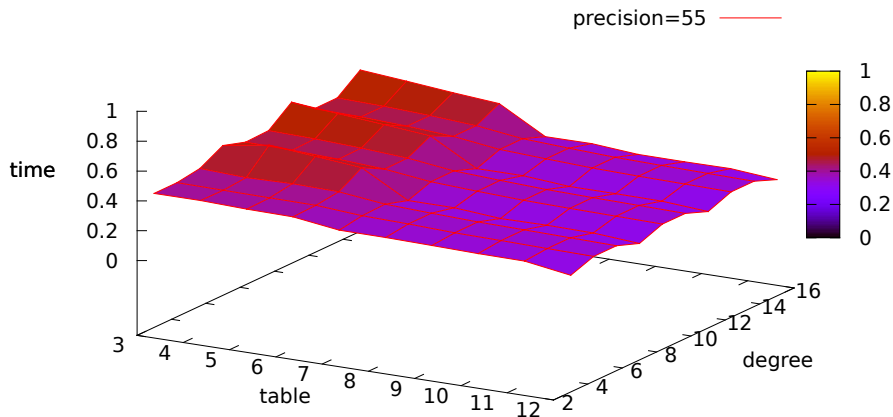
Timings for exponential function



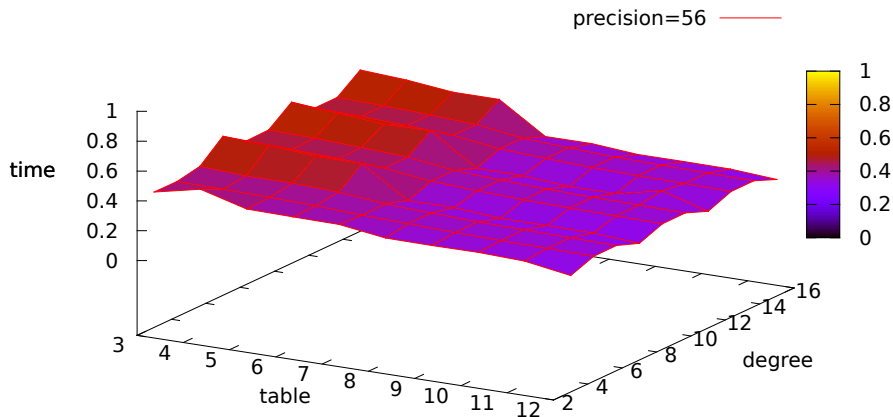
Timings for exponential function



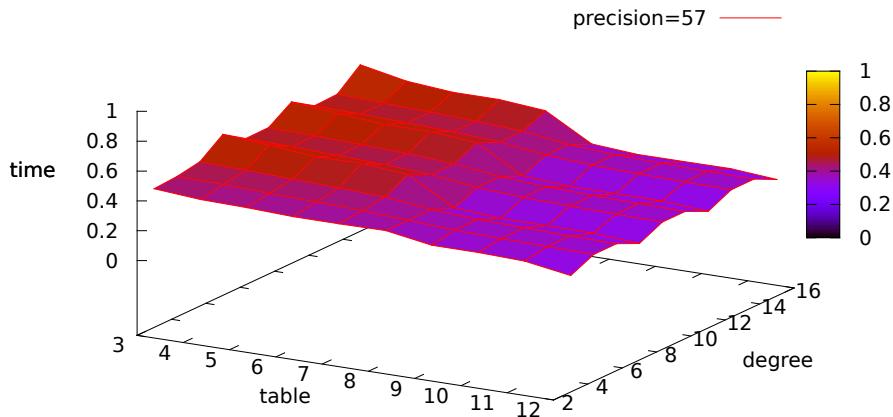
Timings for exponential function



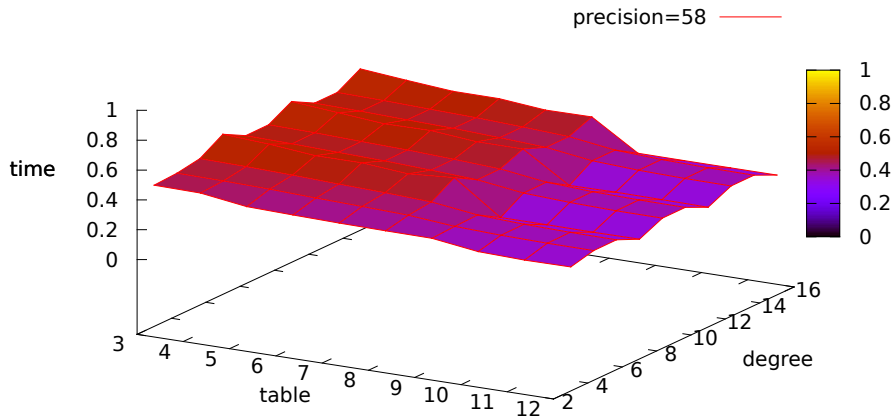
Timings for exponential function



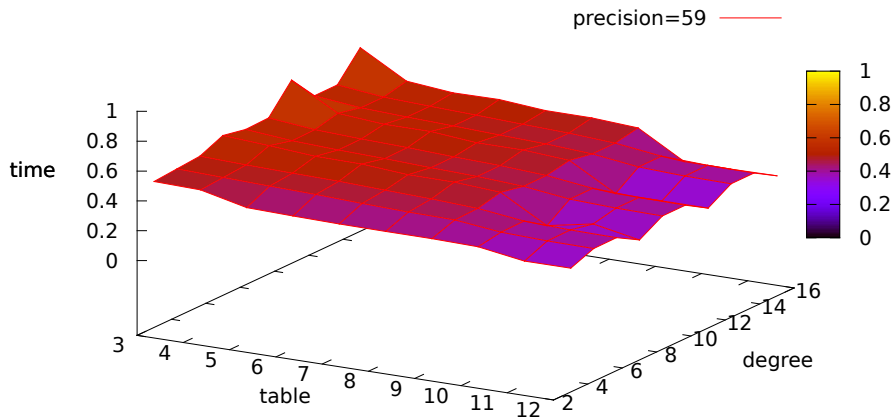
Timings for exponential function



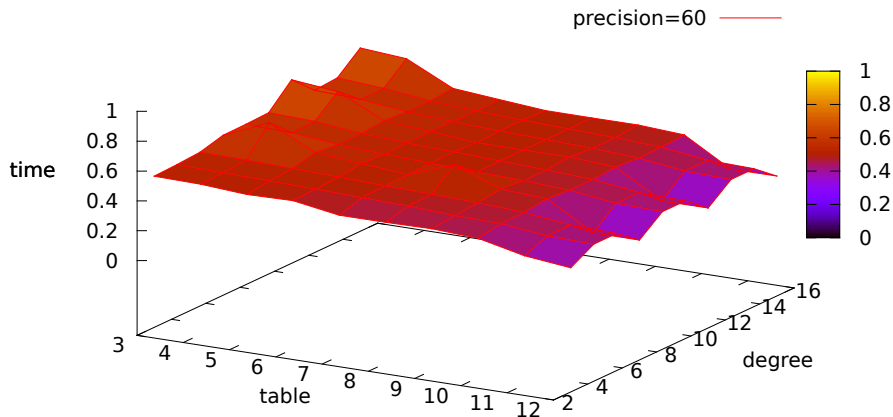
Timings for exponential function



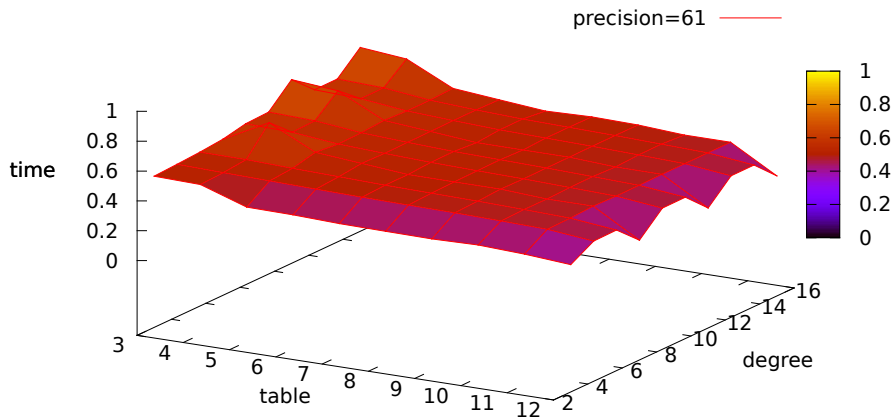
Timings for exponential function



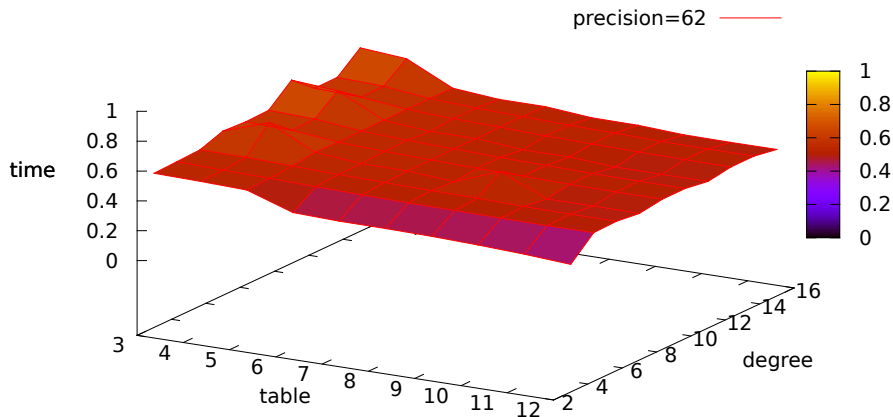
Timings for exponential function



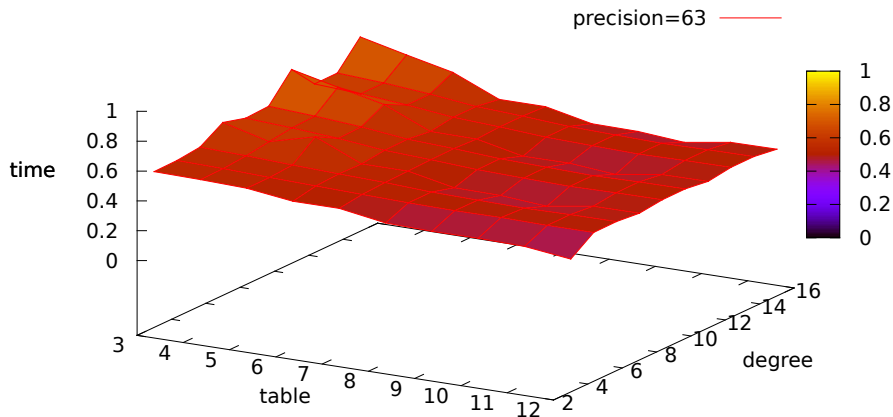
Timings for exponential function



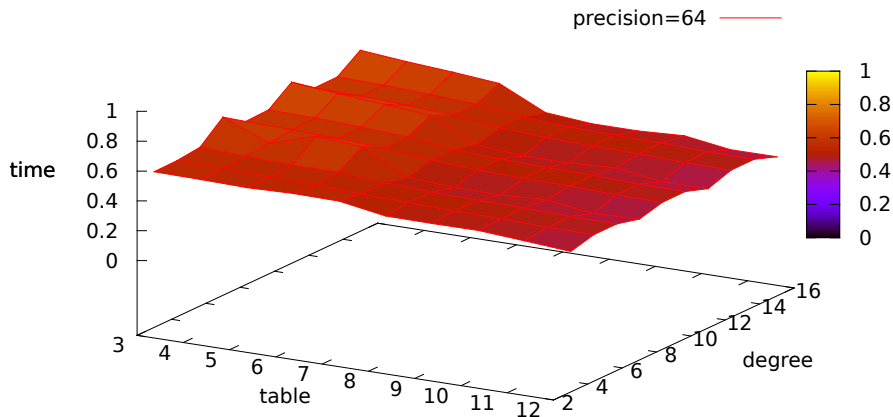
Timings for exponential function



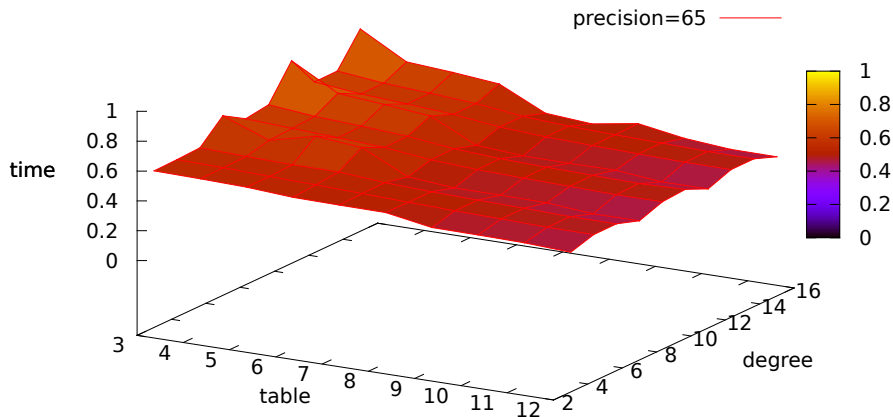
Timings for exponential function



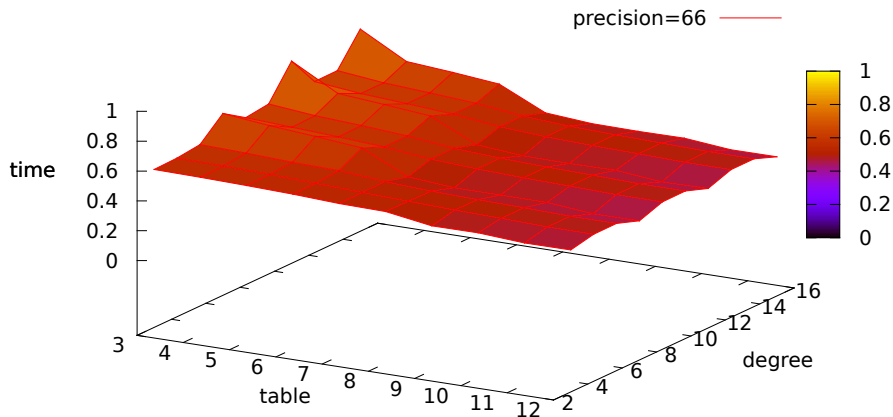
Timings for exponential function



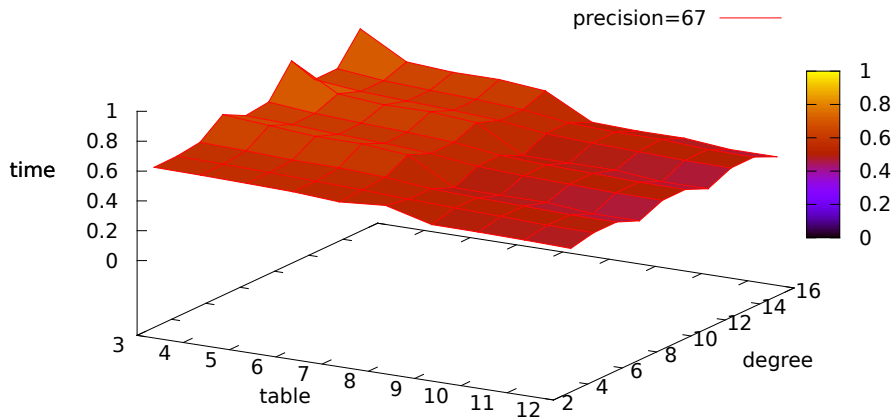
Timings for exponential function



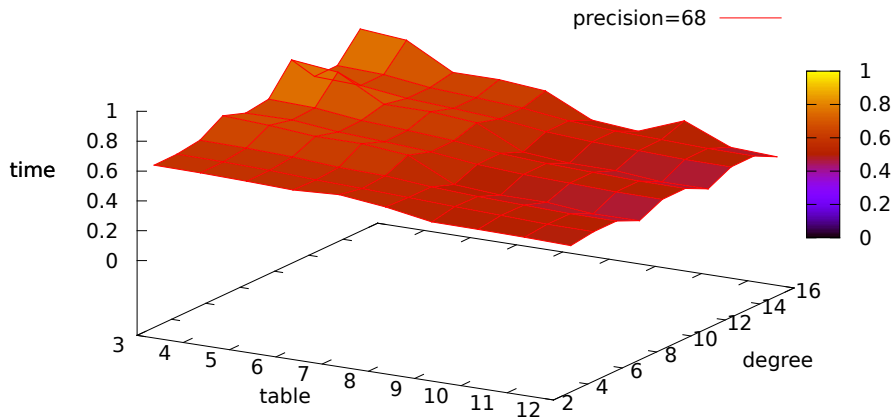
Timings for exponential function



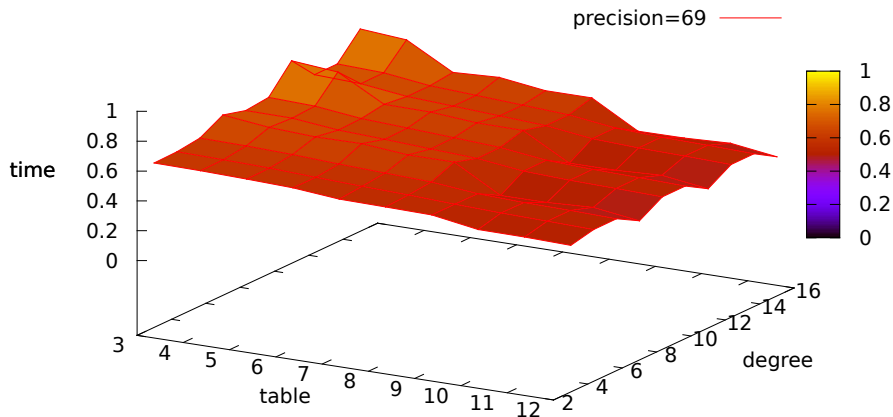
Timings for exponential function



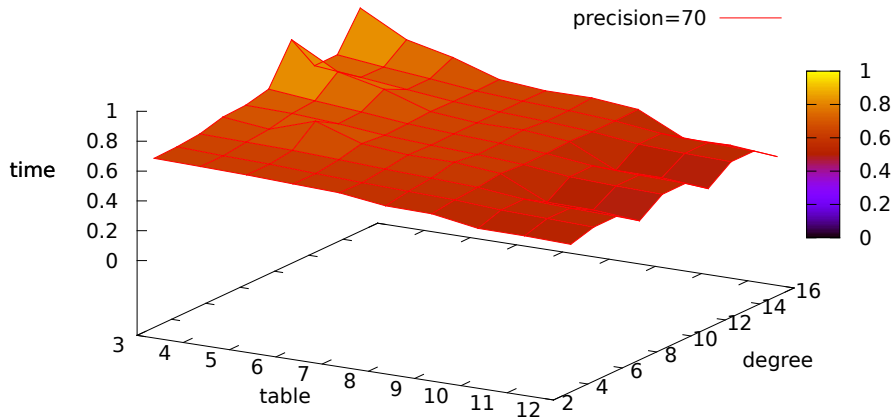
Timings for exponential function



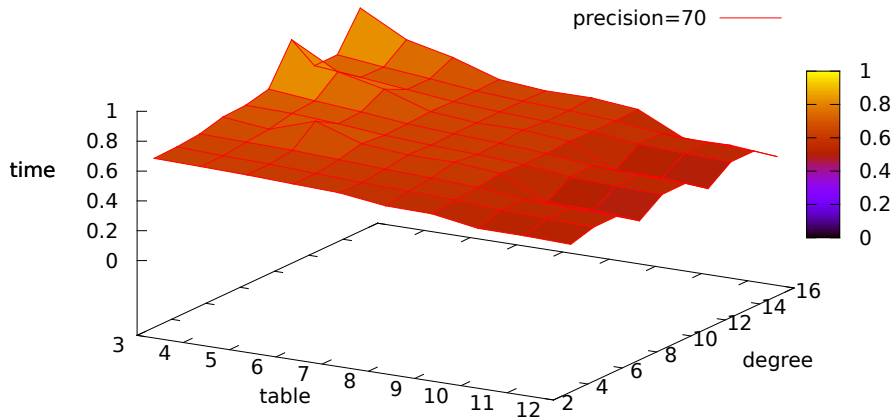
Timings for exponential function



Timings for exponential function



Timings for exponential function



Making this movie meant implementing 6150 functions

Conclusions

- the libm restrictions
- problems in solving them
- writing code \rightarrow writing code generator
- get functions for the specified domains with the specified accuracy
- generate math function code in several minutes

Thank you for your attention!
Questions?

Table Maker's Dilemma

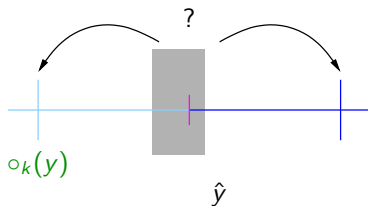
Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

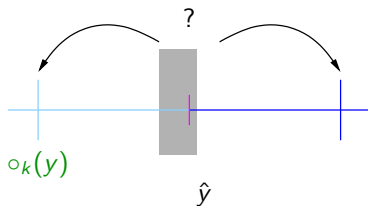


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

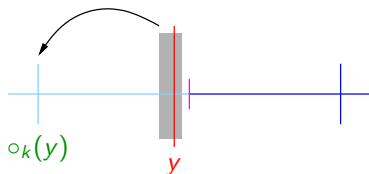


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Table Maker's Dilemma

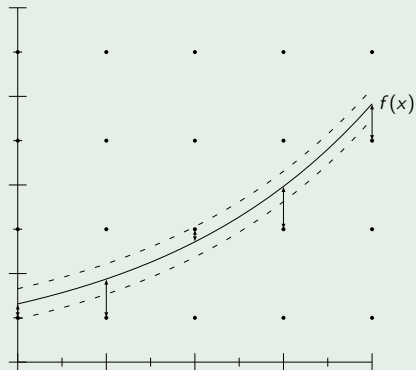
Correct rounding $\circ_p(f(x))$:

- The main phenomena:
 - The value of transcendental $f(x)$ can be only approximated
 - The rounding \circ_p changes abruptly on the rounding borders

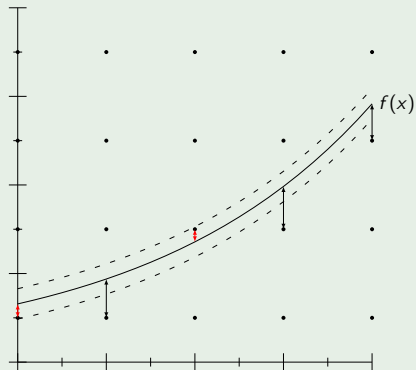


- The value $y = f(x)$ cannot be computed exactly
- We can compute only an approximation $\hat{y} = f(x)(1 + \varepsilon)$

Hard to Round Cases



Hard to Round Cases



HR-case existence test

Problem

Given $P \in \mathbb{R}[x]$, is there any $x \in \mathbb{Z}$ such that

$$P(x) \bmod 1 < \varepsilon.$$

Solutions (with p the floating-point precision)

- Exhaustive search: $O(2^p)$;
- Lefèvre when $\deg P = 1$: $O(2^{2p/3})$ intervals in $O(p^2)$;
- SLZ when $\deg P > 1$: $O(2^{p/2})$ intervals in $O(\text{poly}(p, \deg P, \alpha))$.

Example of computation times

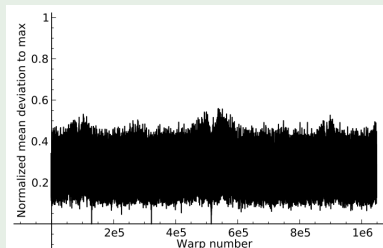
- e^x in full domain and $p = 53$ with Lefèvre: 5 years of CPU time;
- 2^x in $[1/2, 1[$ and $p = 64$ with SLZ: 8 years of CPU time.

Idle Time Because of Branches

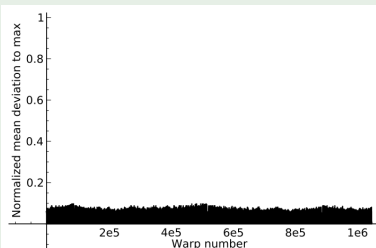
Lefèvre algorithm: irregular control flow

⇒ Gouicem: provide a more regular control flow

Lefèvre Algorithm



Gouicem Algorithm



A regular control flow

	Seq.	MPI	CPU-GPU
Lefèvre	80116.91	10544.2	3235.11
Gouicem	77340.75	9968.5	1500.76
Lef. /Reg.	1.04	1.06	2.16

Table: Performance result on e^x in $[1, 2[$ for binary64 (Intel Xeon X5650 hexa-core, Nvidia C2070).

Lefèvre on CPU / Gouicem on MPI $\Rightarrow 8.04$

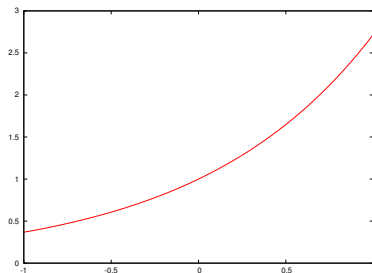
Lefèvre on CPU / Gouicem on GPU $\Rightarrow 53.38$

Lefèvre on MPI / Gouicem on GPU $\Rightarrow 7.03$

\Rightarrow Computation of few years now takes few weeks (estimated time for full range e^x in binary64 : < 1 week on one GPU)

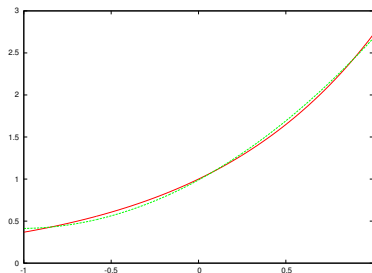
Polynomial approximation

- How to implement $f = \exp \dots$
- ... on a small domain $I = [-1; 1]$, to start with?



Polynomial approximation

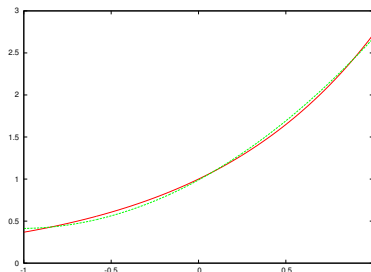
- How to implement $f = \exp \dots$
- ... on a small domain $I = [-1; 1]$, to start with?



- Solution: replace f by an approximation polynomial p

Polynomial approximation

- How to implement $f = \exp \dots$
- ... on a small domain $I = [-1; 1]$, to start with?



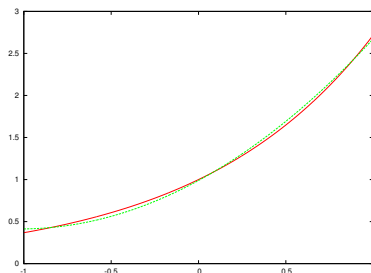
- Solution: replace f by an **approximation polynomial p**
- Classes of polynomials:
 - **Weierstraß** tells us that the techniques always work
 - Taylor expansion as a first idea
 - Polynomials that **minimize the maximum error** on the domain.

Reasons why argument reduction is needed

- Functions f need to be implemented on **the whole FP domain**
- For binary64, $f = \exp$ is defined on $I = [-744.5; 709]$

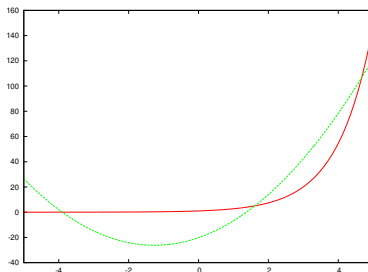
Reasons why argument reduction is needed

- Functions f need to be implemented on the whole FP domain
- For binary64, $f = \exp$ is defined on $I = [-744.5; 709]$
- Polynomial approximation alone is not enough:



Reasons why argument reduction is needed

- Functions f need to be implemented on the whole FP domain
- For binary64, $f = \exp$ is defined on $I = [-744.5; 709]$
- Polynomial approximation alone is not enough:



- When the domain is large, the error explodes for a given degree
- or: a huge degree is needed to compensate.

Argument reduction

- **Argument reduction** reduces the range of the function
 - Use of **algebraic properties** of the function
 - Periodicity: \sin
 - Self-similarity: \exp
 - Symmetry: asin
 - Use of the **semi-logarithmic character** of the FP formats
 - From space, `convertToInteger` and `exp` are kind of alike.
 - Fallback: **splitting of the domain** into subdomains.

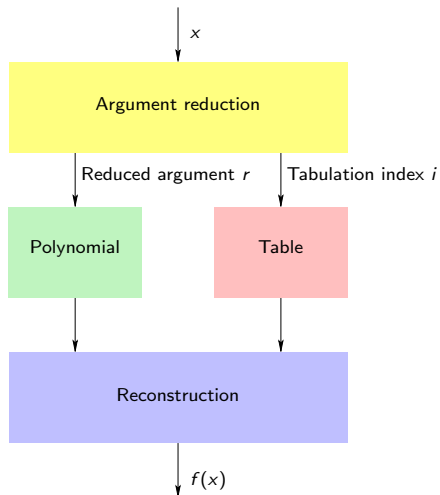
Argument reduction

- Argument reduction reduces the range of the function
 - Use of **algebraic properties** of the function
 - Periodicity: \sin
 - Self-similarity: \exp
 - Symmetry: \arcsin
 - Use of the **semi-logarithmic character** of the FP formats
 - From space, `convertToInteger` and `exp` are kind of alike.
 - Fallback: **splitting of the domain** into subdomains.
- Example for \exp :

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

- The reduction is used often in a couple with a tabulation
 - Tabulation: precomputation of a function g on a discrete set of points
 - Reduction: the polynomial p only has to fill in between these discrete points

Algorithm scheme for a function's implementation



A toy exponential

```
// A very crude "toy" implementation of exp(x)
//
// About 42 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;
}
```


Accuracy of the HR-cases

- No *direct* computation of the correct rounding $F(x) = \circ(f(x))$
- Computation of an *approximation* $f(x) \cdot (1 + \varepsilon)$ *more accurate* than ...
... the relative distance of the *HR-case*

Accuracy of the HR-cases

- No *direct* computation of the correct rounding $F(x) = \circ(f(x))$
- Computation of an *approximation* $f(x) \cdot (1 + \varepsilon)$ *more accurate* than ...
... the relative distance of the *HR-case*
- Utilisation of the lemmas like the following:

Lemma

Let $f = \exp$. Let $\circ_{53} : \mathbb{R} \rightarrow \mathbb{F}_{53}$ be a double precision rounding.

Let $\mathbb{D} = \mathbb{F}_{53} \cap [-744.5; 709]$ a domain of a function f .

Then for every $x \in \mathbb{D} \setminus \{0\}$ and every $|\varepsilon| \leq 2^{-159}$,

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

Accuracy of the HR-cases

- No *direct* computation of the correct rounding $F(x) = \circ(f(x))$
- Computation of an *approximation* $f(x) \cdot (1 + \varepsilon)$ *more accurate* than ...
... the relative distance of the *HR-case*
- Utilisation of the lemmas like the following:

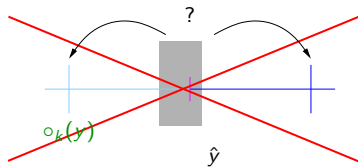
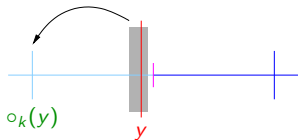
Lemma

Let $f = \exp$. Let $\circ_{53} : \mathbb{R} \rightarrow \mathbb{F}_{53}$ be a double precision rounding.

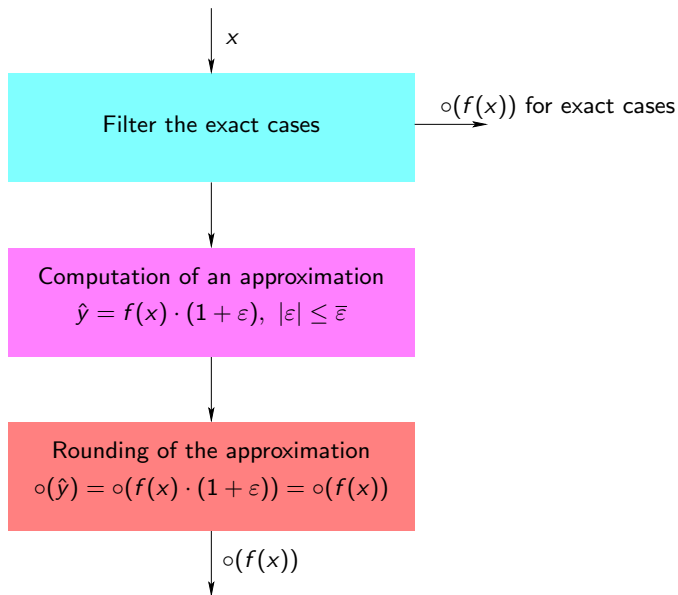
Let $\mathbb{D} = \mathbb{F}_{53} \cap [-744.5; 709]$ a domain of a function f .

Then for every $x \in \mathbb{D} \setminus \{0\}$ and every $|\varepsilon| \leq 2^{-159}$,

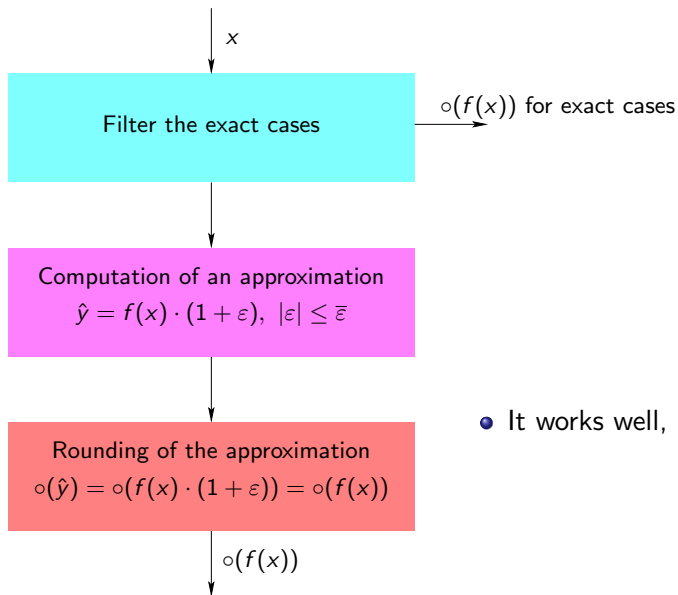
$$\circ_{53}(f(x) \cdot (1 + \varepsilon)) = \circ_{53}(f(x)).$$



A simple diagram for the correct rounding

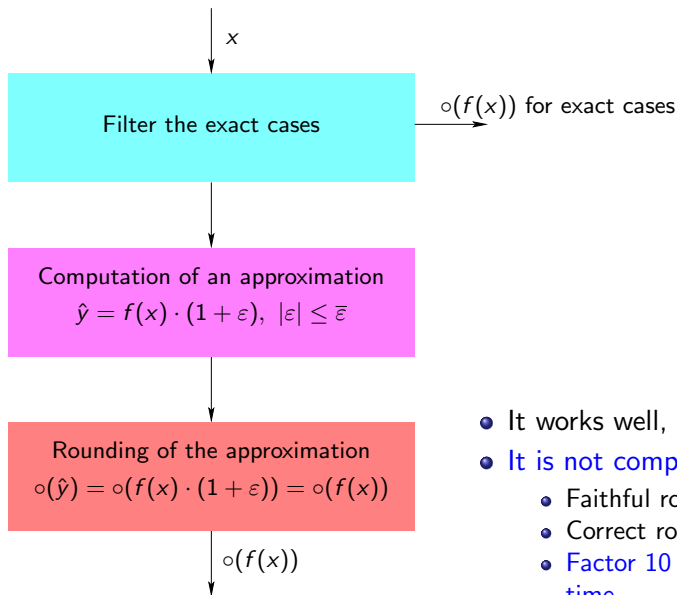


A simple diagram for the correct rounding



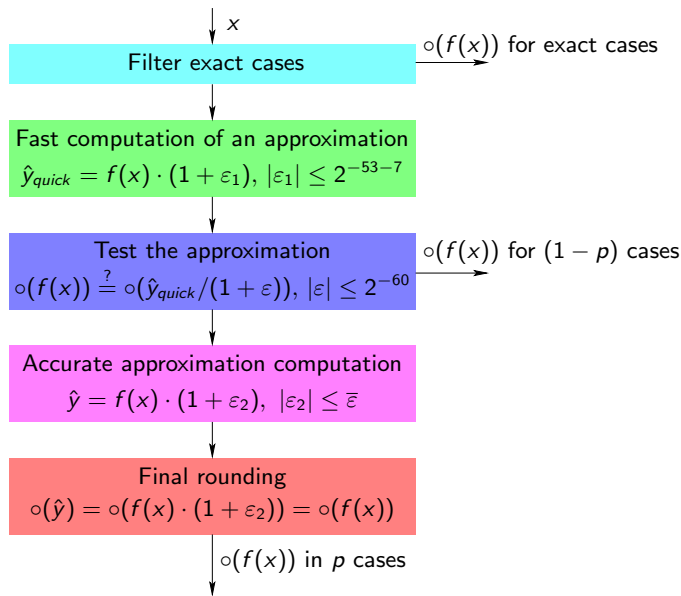
- It works well,

A simple diagram for the correct rounding

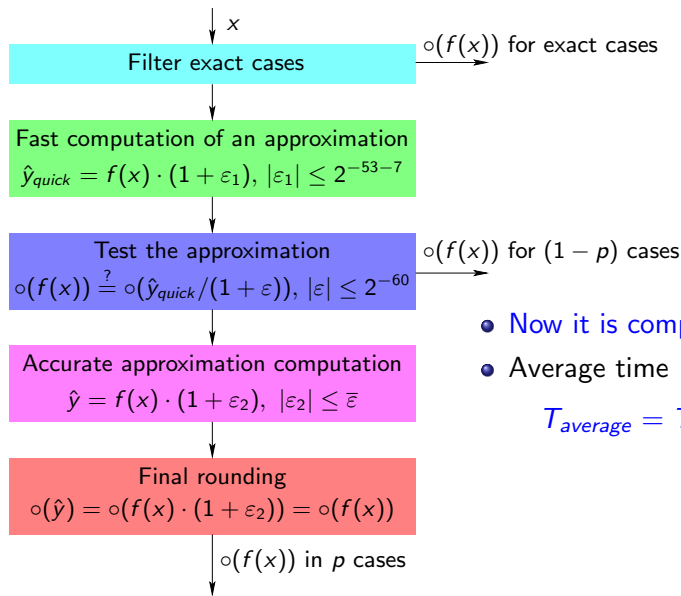


- It works well, but...
- **It is not competitive at all**
 - Faithful rounding: 53 + 5 bits
 - Correct rounding: 159 bits
 - **Factor 10 for the computation time**

Approximation in 2 steps



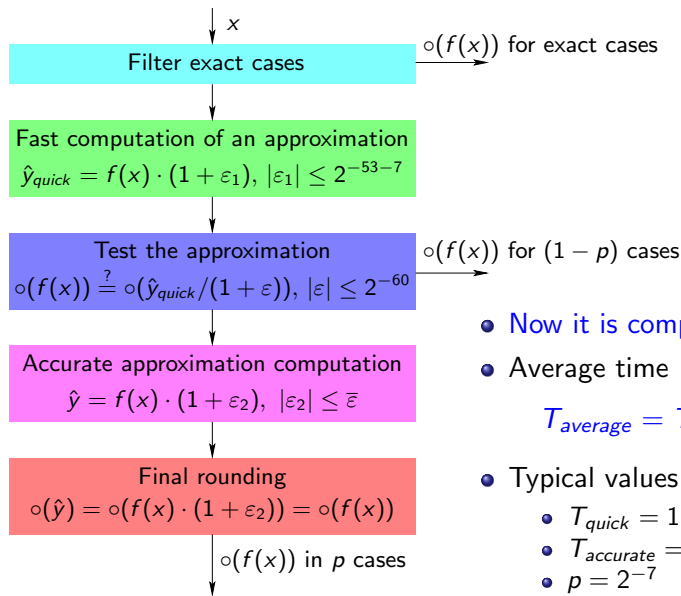
Approximation in 2 steps



- Now it is competitive
- Average time

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

Approximation in 2 steps



- Now it is competitive
- Average time

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

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