Correct Rounding of Mathematical Functions

Vincent LEFÈVRE

Arénaire, INRIA Grenoble – Rhône-Alpes / LIP, ENS-Lyon

SIESTE, 2010-03-23

Outline

- Introduction: Floating-Point Arithmetic
- Relation with Programming Languages
- Solving the Table Maker's Dilemma: Introduction
- Solving the Table Maker's Dilemma: L-Algorithm
- Solving the Table Maker's Dilemma: SLZ Algorithm
- Solving the Table Maker's Dilemma: Periodical Functions with Large Arguments
- Results

Floating-Point Formats

Floating-point representation in radix β (e.g., 2), precision p:

$$x = s \cdot m \cdot \beta^e \qquad (\textit{ machine number })$$

where

- $s = \pm 1$ is the *sign*,
- $m = x_0.x_1x_2 \ldots x_{p-1}$ (with $0 \leq x_i \leq \beta - 1$) is the *significand*,
- the integer e is the *exponent* ($e_{\min} \leq e \leq e_{\max}$).

Normalization: if $e > e_{\min}$, one can require $x_0 \neq 0$ ($x_0 = 1$ if $\beta = 2$).

Special numbers: ± 0, $\pm\infty$, NaN.

IEEE 754 *basic formats*: binary32, binary64, binary128, decimal64, decimal128; binary64 is the most used (most precise supported in hardware in practice), available via the C type `double` (in practice), ECMAScript, XPath, Perl's floating-point type (most often)...

Rounding Direction Attributes (IEEE 754)

The exact result of floating-point function (operation $+$, $-$, $\times$, $\div$, square root, radix conversion, etc.) is not always a machine number; in general, it must be *rounded*.

Rounding direction attributes **(rounding modes):**

- **Rounding to nearest:** $y = \circ(x)$ is the machine number closest to x.
  If x is halfway between two consecutive machine numbers:
  - roundTiesToEven: the one whose least significant digit y_{p-1} is even.
  - roundTiesToAway (new in 2008): the one whose magnitude is larger.
- **Directed rounding:** $\circ(x)$ is the machine number closest to x such that:
  - roundTowardNegative (toward $-\infty$): $\circ(x) \leq x$;
  - roundTowardPositive (toward $+\infty$): $\circ(x) \geq x$;
  - roundTowardZero: $|\circ(x)| \leq |x|$, equivalent to
    - ⋆ roundTowardNegative if $x \geq 0$,
    - ⋆ roundTowardPositive if $x \leq 0$.

Default rounding direction attribute (if dynamic) for binary: roundTiesToEven.

Correct Rounding

The IEEE 754 standard requires the *correct rounding* of various operations.
In the 2008 version:

> *Except where stated otherwise, every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the attributes in this clause.*

Supported operations:

- Already in the 1985 version (well-supported):
  - $+$, $-$, $\times$, $\div$, square root;
  - binary-decimal conversions up to some limits.
- New in the 2008 version (partial support):
  - fused multiply-add (FMA): $\mathrm{fma}(x, y, z) = xy + z$;
  - binary-decimal conversions up to some higher limits (possibly unbounded);
  - some elementary functions recommended.

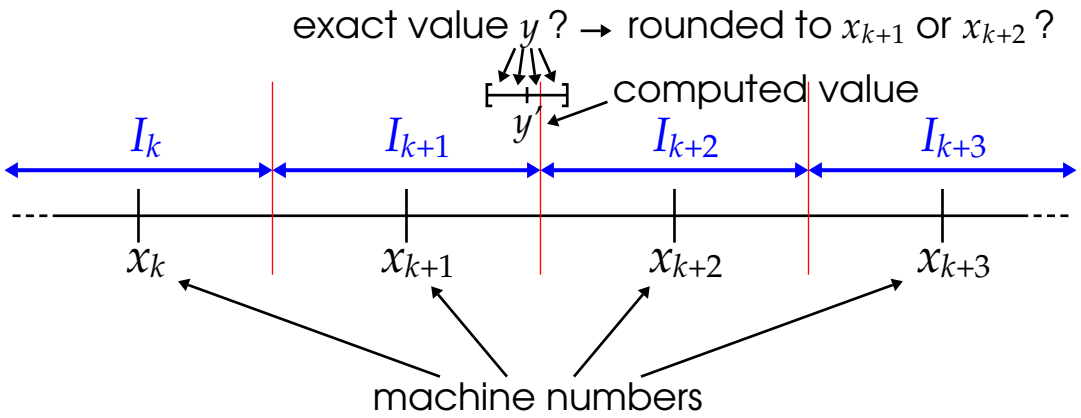
Rounding the Elementary Functions

Elementary functions: correct rounding is difficult, at least to guarantee a correct and/or efficient implementation.

- We want to evaluate and round $y = f(x)$, i.e. to return $\circ(y)$.
- We know how to compute an approximation y' to y with error bound ε.
- We know how to round y', but do we have $\circ(y') = \circ(y)$?

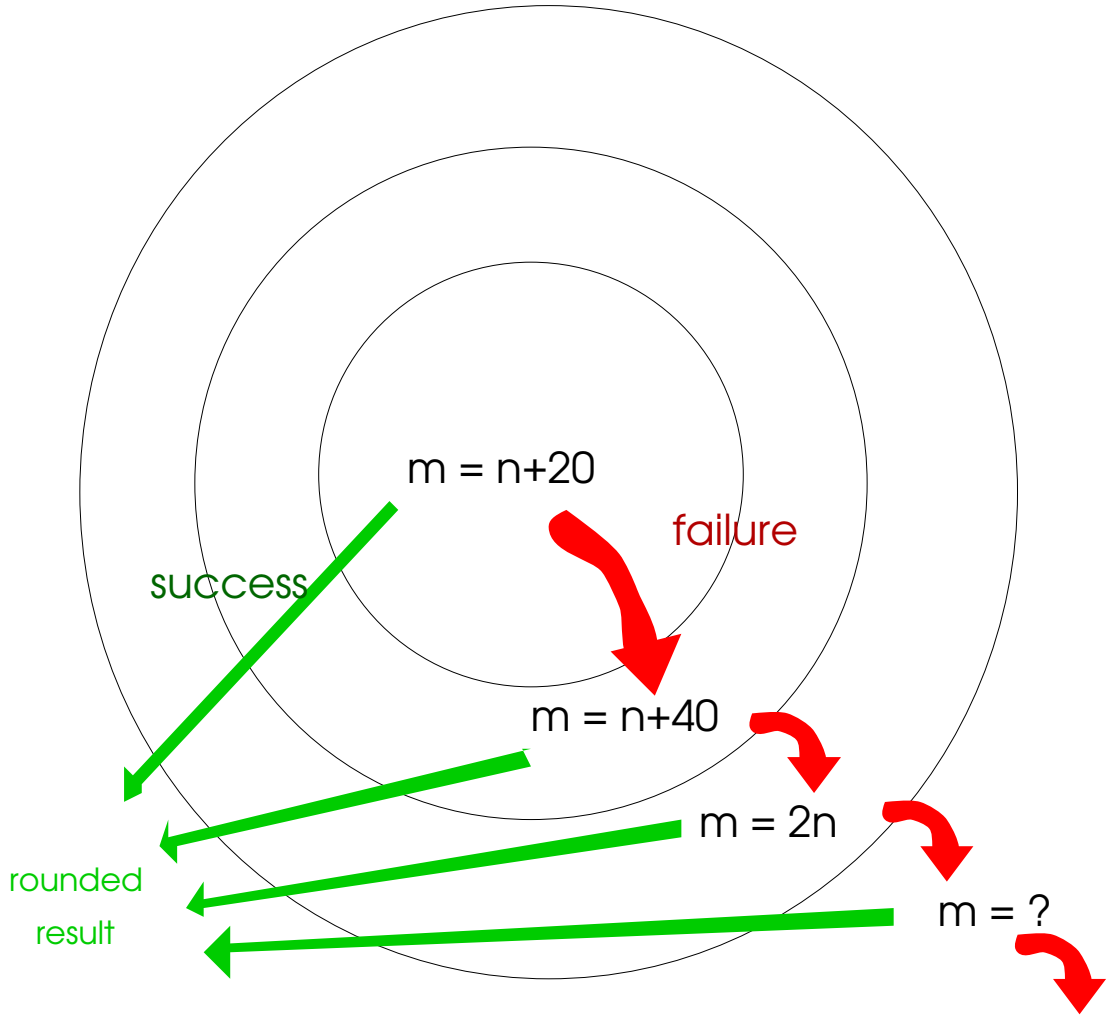
Problem known as the *Table Maker's Dilemma* (TMD).

Example in rounding to nearest:



Minimum ε for which the TMD can occur? Worst case(s)?

Ziv's Strategy



Implementations of Elementary Functions

Implementations of the standard math library in the past few years and today do not provide correct rounding in general.

From tests done several years ago in *round-to-nearest* on *worst cases* for elementary functions:

- exp, log, exp2, log2, exp10, log10,
- sinh, asinh, cosh, acosh,
- sin, asin, cos, acos, tan, atan,
- $1/x^2$, $1/\sqrt{x}$, x^3, $\sqrt[3]{x}$,

platforms giving 36 different behaviors, including 8 GNU/Linux machines with some apparently correctly-rounded functions, thanks to MathLib...

More details on `http://www.vinc17.net/research/testlibm/`

Implementations of Elementary Functions [2]

Various implementations of correctly rounded functions:

- **Ziv/IBM's libultim/MathLib** (2002).
  Not proved (by Ziv/IBM) and only rounding to nearest.
  Requires the dynamic rounding mode to be round-to-nearest, otherwise results can be completely wrong, with possible crash (glibc bug 3976).
  Included in GNU libc (but not used on all platforms).
- **Sun's libmcr** (2004). Not proved.
- **CRlibm (Arénaire)**, started in 2004.
  Proved and optimized, thanks to the worst cases I obtained!
  Lead to the recommendation in IEEE 754-2008.
- **GNU MPFR (mainly INRIA)**, started in 1999. In arbitrary precision.

Note: my results provide a proof for the method used by Ziv's library, but contrary to CRlibm, whose code is based on them, Ziv's library can be inefficient in some (rare) cases: internal precision of 768 bits, while about 120 would be sufficient.

Relation with Programming Languages

The IEEE 754 standard specifies floating-point arithmetic (though not completely, e.g., correct rounding of elementary functions is only a recommendation), but what about programming languages? Possible specification via *bindings*.

- ECMAScript and XPath: IEEE 754 double precision, round-to-nearest only.
- FORTRAN: no support (forbidden expression transformations).
- Java: needs the `strictfp` modifier.
- ISO C99: optional (Annex F, pragmas).

In practice, difficult:

- Possible bugs due to optimizations. For instance, `pow(x,0.5)` must not be replaced by `sqrt(x)` unconditionally: on -0, the results are $+0$ and -0 respectively. 2010-03-18: GCC PR 43419 (4.3.* and 4.4.3).
- Possible inconsistency between the compiler and the math library.

Elementary functions: math library, but...

Evaluating a Sine: 1st Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test1 (void)
{
  double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1`

Evaluating a Sine: 1st Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test1 (void)
{
  double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1`

Result: 0.24957989804940911016 (correct)

Evaluating a Sine: 2nd Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test2 (void)
{
  volatile double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1` (like `test1`)

Evaluating a Sine: 2nd Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test2 (void)
{
  volatile double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1` (like `test1`)

Result: 0.24957989804940913792

Evaluating a Sine: 2nd Test

For all tests: GCC 4.4.3 and glibc 2.10.2 under Linux/x86_64 (Debian/sid).

```
double test2 (void)
{
  volatile double x = D;
  double x2, y;

  x2 = x;
  y = sin (x2);
  return y;
}
```

compiled with: `-O2 -DD=2.522464e-1` (like `test1`)

Result: 0.24957989804940913792
`test1`: 0.24957989804940911016

Evaluating a Sine: A More Difficult Case

Both tests compiled with: `-O2 -DD=1e22`

Evaluating a Sine: A More Difficult Case

Both tests compiled with: `-O2 -DD=1e22`

Difficulty: 10^{22} is much larger than 2π, and the range reduction must be carried out with enough precision to provide an accurate result.

Evaluating a Sine: A More Difficult Case

Both tests compiled with: `-O2 -DD=1e22`

Difficulty: 10^{22} is much larger than 2π, and the range reduction must be carried out with enough precision to provide an accurate result.

Results:

`test1`: $-0.852200849767188794 99$

Evaluating a Sine: A More Difficult Case

Both tests compiled with: `-O2 -DD=1e22`

Difficulty: 10^{22} is much larger than 2π, and the range reduction must be carried out with enough precision to provide an accurate result.

Results:

`test1`: -0.85220084976718879499
`test2`: -0.85220084976718879499

Evaluating a Sine: 3rd Test

```
double test3 (void)
{
  volatile double x = D, z;
  double x2, y;

  x2 = x;
  y = sin (x2);
  z = cos (x2);
  return y;
}
```

compiled with: `-O2 -DD=1e22`

Results:

`test1`: −0.85220084976718879499

`test2`: −0.85220084976718879499

Evaluating a Sine: 3rd Test

```
double test3 (void)
{
  volatile double x = D, z;
  double x2, y;

  x2 = x;
  y = sin (x2);
  z = cos (x2);
  return y;
}
```

compiled with: `-O2 -DD=1e22`

Results:

test1:	−0.85220084976718879499
test2:	−0.85220084976718879499
test3:	0.46261304076460174617

Evaluating a Sine: The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.

Evaluating a Sine: The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.
- `test2`: Due to the `volatile` qualifier, GCC does not perform the above optimization (assuming possible side effects). The `sin()` function is called. At run time, this function is provided by the glibc math library, based (in 64-bit mode) on IBM's MathLib, which provides correct rounding.

Evaluating a Sine: The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.
- `test2`: Due to the `volatile` qualifier, GCC does not perform the above optimization (assuming possible side effects). The `sin()` function is called. At run time, this function is provided by the glibc math library, based (in 64-bit mode) on IBM's MathLib, which provides correct rounding.
  But there is a bug for $0.25 < |x| < 0.855469$, due to incorrect error analysis (found by Paul Zimmermann, glibc bug 10709).

Evaluating a Sine: The Explanations

- `test1`: The variable `x` has a constant value (and known at compile time), so does `x2`, and GCC can evaluate the expression `sin(x2)`. As of version 4.3.0, GCC uses MPFR, which provides correct rounding.
- `test2`: Due to the `volatile` qualifier, GCC does not perform the above optimization (assuming possible side effects). The `sin()` function is called. At run time, this function is provided by the glibc math library, based (in 64-bit mode) on IBM's MathLib, which provides correct rounding.
  But there is a bug for $0.25 < |x| < 0.855469$, due to incorrect error analysis (found by Paul Zimmermann, glibc bug 10709).
- `test3`: The optimization is still not possible, but GCC notices that both `sin()` and `cos()` are called on the same value `x2` (not volatile), and calls the `sincos()` function, assuming the glibc math library will be used (indeed, `sincos()` is a GNU extension). This function, not provided by MathLib, is implemented by the `fsincos` x87 instruction.

How About Arithmetic Operations?

```
#include <stdio.h>
#include <math.h>

#ifdef FP_CONTRACT
#undef FP_CONTRACT
#define FP_CONTRACT "ON"
#pragma STDC FP_CONTRACT ON
#else
#define FP_CONTRACT "OFF"
#pragma STDC FP_CONTRACT OFF
#endif

static double fct (double a, double b)
{ return a >= b ? sqrt (a * a - b * b) : 0; }

void test (volatile double x)
{ printf ("test(%.20g) = %.20g\n", x, fct (x, x + 0.0)); }
```

Test with:

1. GCC 4.0.1 (Apple) on PowerPC
2. GCC 4.3.2 on `x86_64`
3. GCC 4.1.2 on `ia64`
4. ICC 10.1 on `ia64` with `FP_CONTRACT OFF`
5. ICC 10.1 on `ia64` with `FP_CONTRACT ON`

Note: GCC does not support the `FP_CONTRACT` pragma and assumes it is "on" (this is a bug). And actually, the GCC version doesn't matter in these tests.

Input x	2 and 4	1, 3, and 5
1.0	0	0
1.1	0	2.98023223876953125e-09
1.2	0	nan

Test with:

1. GCC 4.0.1 (Apple) on PowerPC
2. GCC 4.3.2 on `x86_64`
3. GCC 4.1.2 on `ia64`
4. ICC 10.1 on `ia64` with `FP_CONTRACT OFF`
5. ICC 10.1 on `ia64` with `FP_CONTRACT ON`

Note: GCC does not support the `FP_CONTRACT` pragma and assumes it is "on" (this is a bug). And actually, the GCC version doesn't matter in these tests.

Input `x`	2 and 4	1, 3, and 5
`1.0`	0	0
`1.1`	0	`2.9802322387695326562e-09`
`1.2`	0	`nan`

Explanation: `a * a - b * b` replaced by `fma(a, a, - (b * b))`

i.e. $\circ(\circ(a^2) - \circ(b^2))$ replaced by $\circ(a^2 - \circ(b^2))$

Solving the Table Maker's Dilemma: Introduction

Function f, machine number x, rounding $f(x)$?

- In arbitrary precision (GNU MPFR).
- In fixed, small precision: search for the *worst cases*.

Breakpoint numbers: discontinuity points of the rounding functions.

- Directed rounding: the machine numbers.
- Round-to-nearest: the midpoint numbers.

A first problem: the exact cases, i.e., when $f(x)$ is a breakpoint.

- Lindemann, 1882: the exponential of an algebraic complex number $\neq 0$ is not algebraic.
- Machine and midpoint numbers are algebraic.

$\rightarrow$ For the elementary functions exp, log, exp2, log2, exp10, log10, sinh, asinh, cosh, acosh, sin, asin, cos, acos, tan, atan, very few exact cases, which are known. For pow, more difficult. For some special functions, open problem.

The Form of Bad Cases in Radix 2

We will focus on radix 2. Problems are similar in radix 10.

If n denotes the precision:

- in directed rounding:

$$\overbrace{1.xx\ldots xx}^{n\text{ bits}}\overbrace{0000\ldots00}^{m\text{ bits}}xx\ldots \quad \text{or} \quad \overbrace{1.xx\ldots xx}^{n\text{ bits}}\overbrace{1111\ldots11}^{m\text{ bits}}xx\ldots$$

- in round-to-nearest:

$$\overbrace{1.xx\ldots xx}^{n\text{ bits}}\overbrace{1000\ldots00}^{m\text{ bits}}xx\ldots \quad \text{or} \quad \overbrace{1.xx\ldots xx}^{n\text{ bits}}\overbrace{0111\ldots11}^{m\text{ bits}}xx\ldots$$

In red: *rounding bit*.

Estimating the Minimum Distance d

Minimum distance d between $f(x)$ and a breakpoint number?

Equivalent problem in radix 2 (up to a factor 2): maximum number of identical bits after the rounding bit of $f(x)$? Or the corresponding value of m?

Estimating the Minimum Distance d

Minimum distance d between $f(x)$ and a breakpoint number?

Equivalent problem in radix 2 (up to a factor 2): maximum number of identical bits after the rounding bit of $f(x)$? Or the corresponding value of m?

- Lindemann's theorem: theoretical, no bounds.

Estimating the Minimum Distance d

Minimum distance d between $f(x)$ and a breakpoint number?

Equivalent problem in radix 2 (up to a factor 2): maximum number of identical bits after the rounding bit of $f(x)$? Or the corresponding value of m?

- Lindemann's theorem: theoretical, no bounds.
- Best theorem giving such bounds: Nesterenko and Waldschmidt (1995), for exp, log, and related functions (trigonometric and hyperbolic). Up to several millions or billions of bits for binary64.

Estimating the Minimum Distance d

Minimum distance d between $f(x)$ and a breakpoint number?

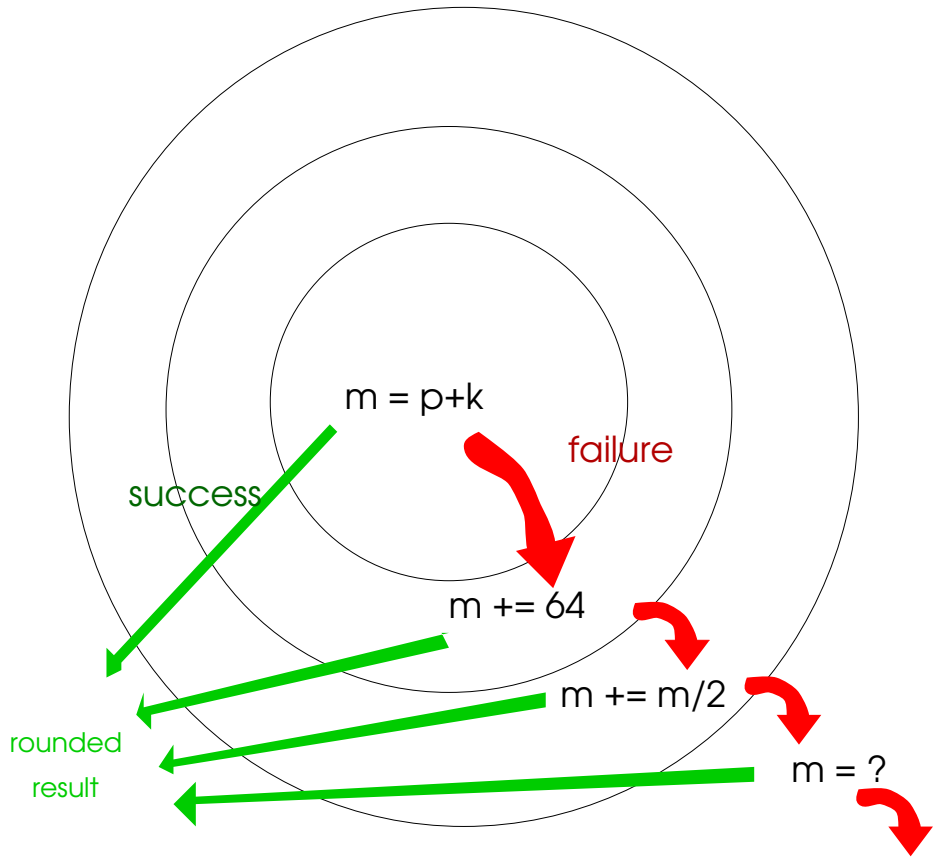
Equivalent problem in radix 2 (up to a factor 2): maximum number of identical bits after the rounding bit of $f(x)$? Or the corresponding value of m?

- Lindemann's theorem: theoretical, no bounds.
- Best theorem giving such bounds: Nesterenko and Waldschmidt (1995), for exp, log, and related functions (trigonometric and hyperbolic). Up to several millions or billions of bits for binary64.
- Probabilistic model: when x is a machine number, $f(x)$ mod ulp(x) is regarded as a random number with uniform distribution.
  $\rightarrow$ For N inputs, d should be of the order of ulp$(x)/N$ and the number of identical bits of the order $\log_2(N)$.
  $\rightarrow$ For one exponent, $N = 2^{n-1}$, so that $m \sim 2n$.
  $\rightarrow$ For all the inputs, $m \sim 2n +$ small constant (for most formats).
  The estimated constant depends on f (e.g., for exp, limited number of exponents: $\exp(x) \simeq 1$ for $|x|$ small, underflow/overflow for $|x|$ large).

Correct Rounding of Math Functions in GNU MPFR

Ziv's strategy in MPFR:

- first evaluate the result with slightly more precision (m) than the target (p);
- if rounding is not possible, then $m \leftarrow m + (32 \text{ or } 64)$, and recompute;
- for the following failures: $m \leftarrow m + \lfloor m/2 \rfloor$.



Detection of the exact cases for the elementary functions.

Open problem for the special functions, but an infinite loop is very unlikely.

Solving the TMD in Fixed, Small Precision

Minimum distance d between $f(x)$ and a breakpoint number?

$\rightarrow$ **Exhaustive search for the worst cases.**

- Single precision ($< 2^{32}$ possible arguments): for each argument x, compute $f(x)$ in higher precision and check.
- Double precision (up to $\simeq 2^{64}$ possible arguments): several hundreds or thousands of years per function would be needed with the same method!
  $\rightarrow$ Specific algorithms designed for these tests.
- Intel's extended precision (up to $\simeq 2^{80}$ possible arguments): still possible, with more time.
- Quadruple precision (up to $\simeq 2^{128}$ possible arguments): out of reach (too much time would be needed).

Search for Worst Cases: History

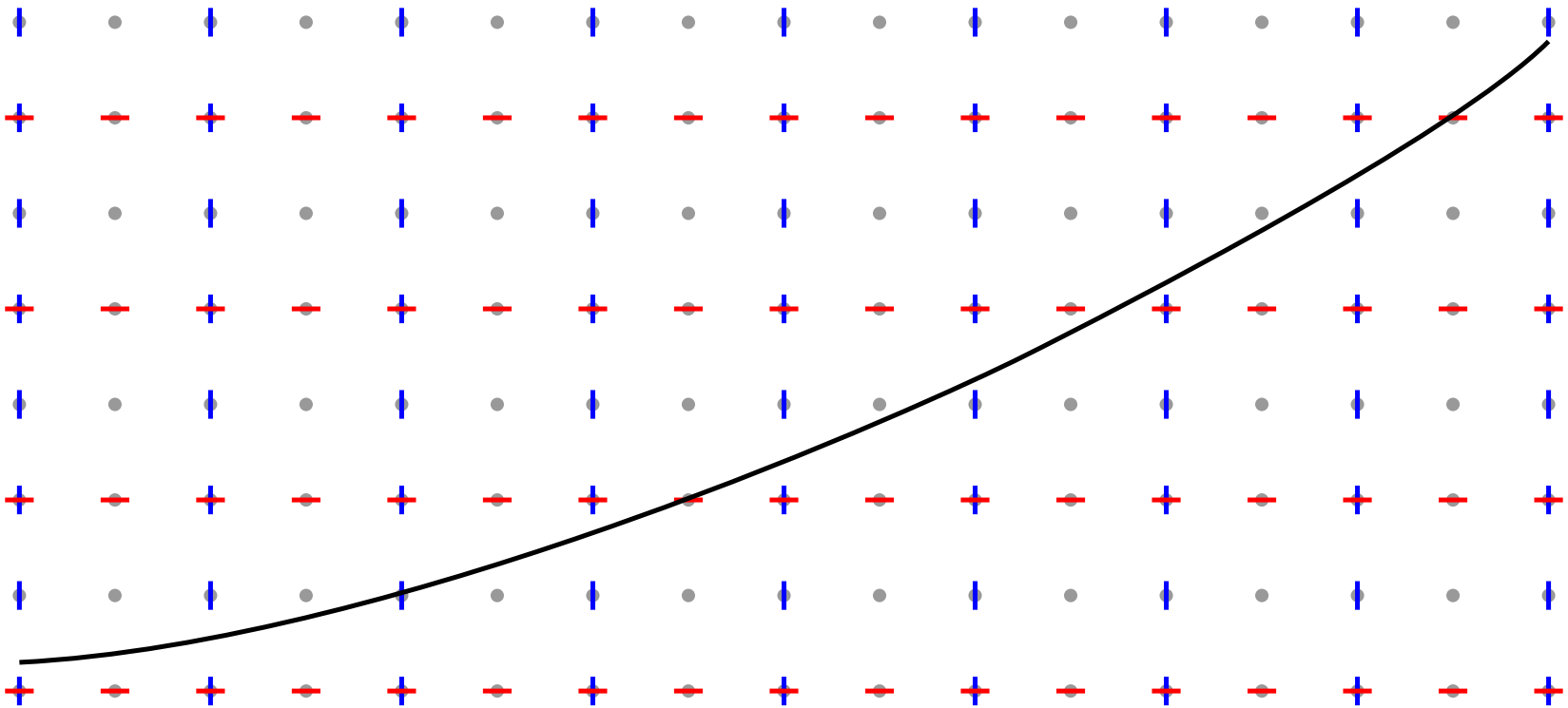
- 1993: first work by M. Schulte and E. E. Swartzlander in single precision.
- Work in double precision (binary64) started in 1996 (DEA training period at LIP). First tests on function exp between 1/2 and 1 using finite differences on degree-2 polynomials (several months on ~ 100 machines).
- October 1996 (beginning of my PhD thesis, with J.-M. Muller): first ideas towards my algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$ (published in June 1997).
- 1998: variant of my algorithm (implemented in 2004).
- October 2002: SLZ algorithm (D. Stehlé – V. Lefèvre – P. Zimmermann), based on lattice reduction (LLL) and Coppersmith's work.
- June 2006: first ideas for the very difficult case of periodical functions sin/cos/tan on large arguments. Work published in January 2007 with G. Hanrot, D. Stehlé and P. Zimmermann.
- May 2009: heuristic bug detections.

Note: 1996-2000 at LIP (Arénaire), 2000-2006 at LORIA (SPACES), 2006-? at LIP (Arénaire).

Search for Worst Cases: the Problem

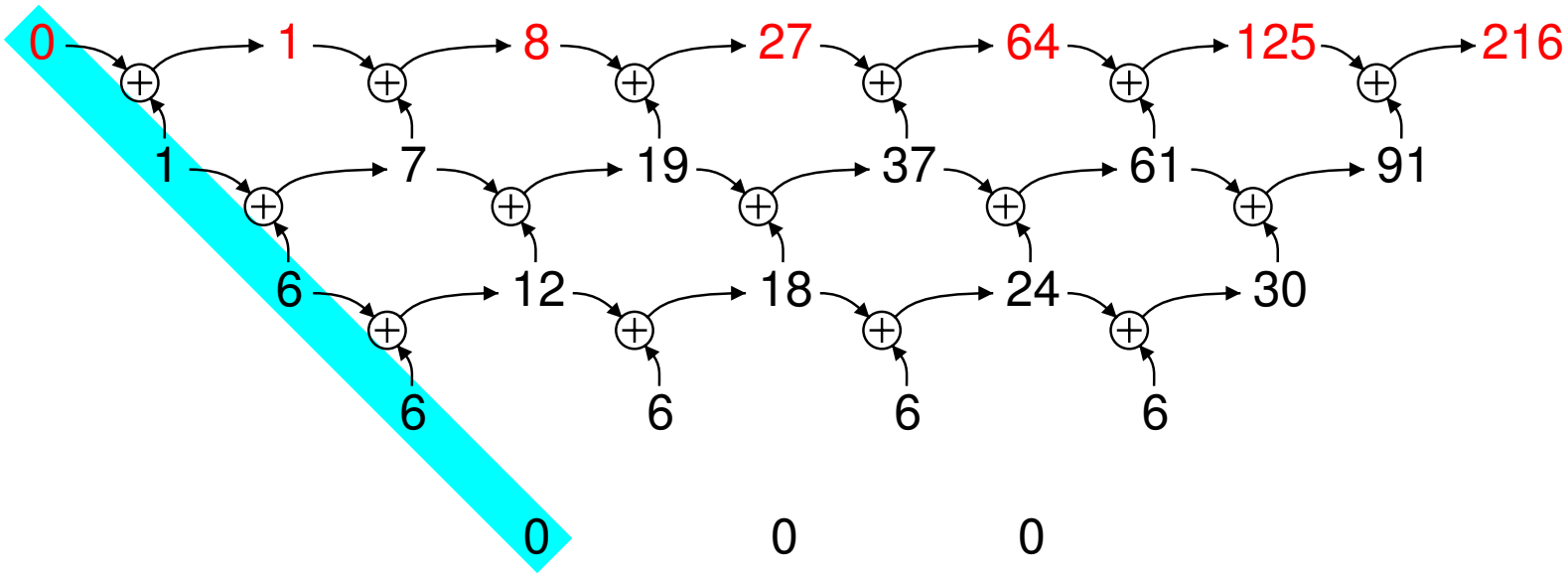
Goal: find all the breakpoint numbers x such that $f(x)$ is very close to a breakpoint number.

Worst cases for f and the inverse function f^{-1}.



Computing the Successive Values of a Polynomial

Example: $P(X) = X^3$. Difference table:



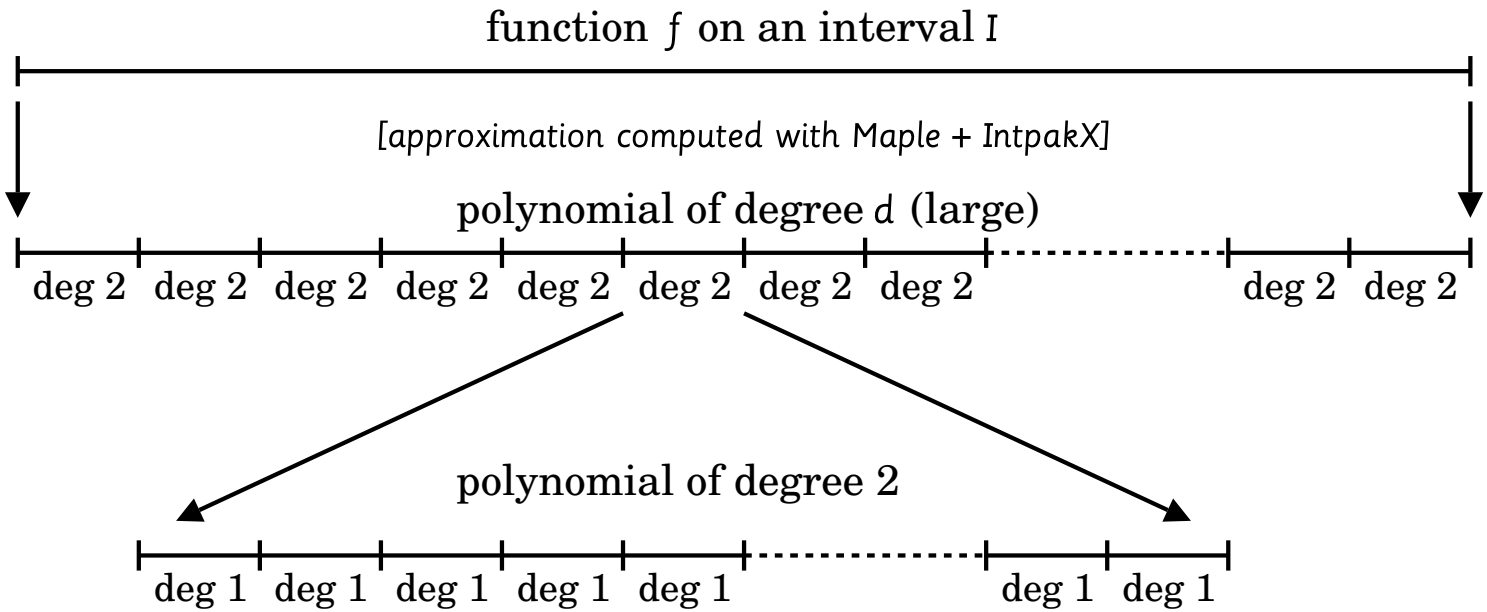
On the left: coefficients in the basis $\left\{1, X, \frac{X(X-1)}{2}, \frac{X(X-1)(X-2)}{3!}, \ldots\right\}$.

Can be done modulo some constant (very useful here).

Can be used to solve the TMD once we have approximated f by a polynomial (valid on a small interval), but also for...

Hierarchical Approximations by Polynomials

Current implementation (but one could have more than 3 levels):



Finding approximations must be very fast: from the previous one, based on regularly-spaced intervals. 2 methods:

- Take into account the computations that haven't been done (add $\rightarrow$ mul).
- Use the fact that each (initial) degree-i coefficient of P_k on the interval J_k can be seen as the value of a polynomial $a_i(k)$.

Implementation

For a given function f, sign $+/-$ and exponent, the binade is split into intervals I (typically, 2^{13} intervals of size 2^{40}, with 54-bit significands to obtain results for the inverse function).

A set of Perl scripts:

- First step: generate, compile and run code to test an interval I and return potential worst cases. Parameters chosen to obtain around 256 potential worst cases (under the probabilistic hypotheses).
- Second step: check the results of the first step with a naive computation. Heuristic bug detections.
  - The number of results must not be too low (e.g. not < 150).
  - Each result must be close enough to a breakpoint (either real or additional, when the exponent of $f(x)$ is variable on the interval).

  If OK, results stored on disk.
- Script to read the results, filter them, and so on. Uses MPFR.
- Client-server system for the first step (slowest part, thus parallelized).

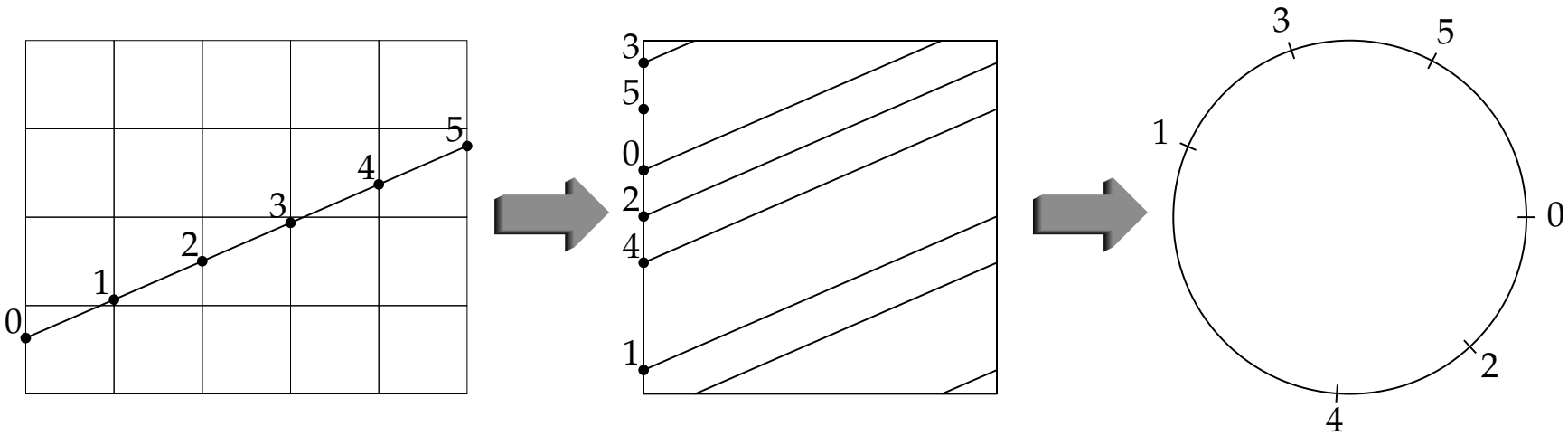
L-Algorithm: the Problem With a Degree-1 Polynomial

In each interval:

- f is approximated by a polynomial of degree $1 \rightarrow$ segment $y = b - ax$.
- Multiplication of the coordinates by powers of $2 \rightarrow$ grid $= \mathbb{Z}^2$.

One searches for the values n such that $\{b - n.a\} < d_0$, where a, b and d_0 are real numbers and $n \in [\![0, N-1]\!]$.

$\{x\}$ denotes the positive fractional part of x.



L-Algorithm: the Problem With a Degree-1 Polynomial [2]

- We chose a positive fractional part instead of centered.

  $\rightarrow$ An upward shift is taken into account in b and d_0.

- If a is rational, then the sequence $0.a$, $1.a$, $2.a$, $3.a$, ... (modulo 1) is periodical.

  $\rightarrow$ This makes the theoretical analysis more difficult.

  $\rightarrow$ In the proof, one assumes that a is irrational, or equivalently, that a is a rational number + an arbitrary small irrational number.

  But in the implementation, a is rational.

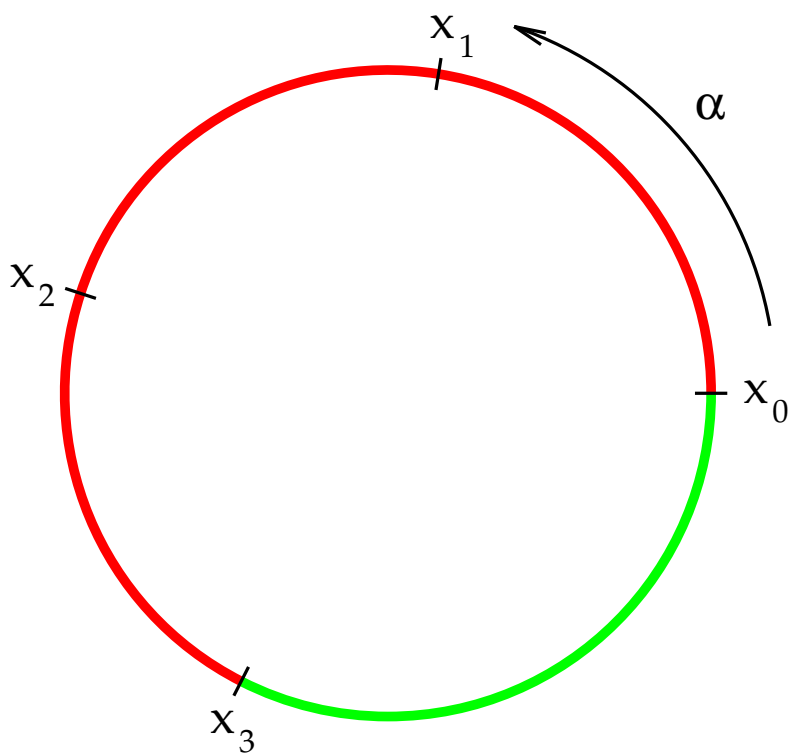
  $\rightarrow$ Extension to rational numbers by continuity.

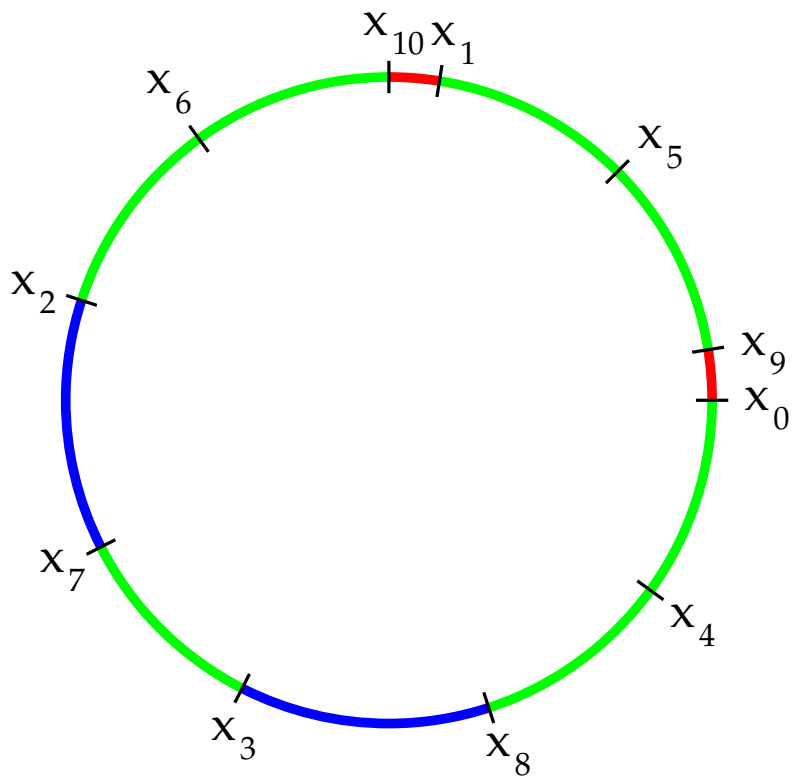
  $\rightarrow$ Care has to be taken with the inequality tests since

  - they are not continuous functions;
  - problems can occur when the period has been reached: endless loops...

The Three-Distance Theorem

Note: related to the three-distance theorem.





Notations / Properties of $k.a \bmod 1$ $(0 \leq k < n)$

Properties of the two-length configurations $C_n = \{k.a \in \mathbb{R}/\mathbb{Z} : k \in \mathbb{N},\ k < n\}$, to be proved by induction:

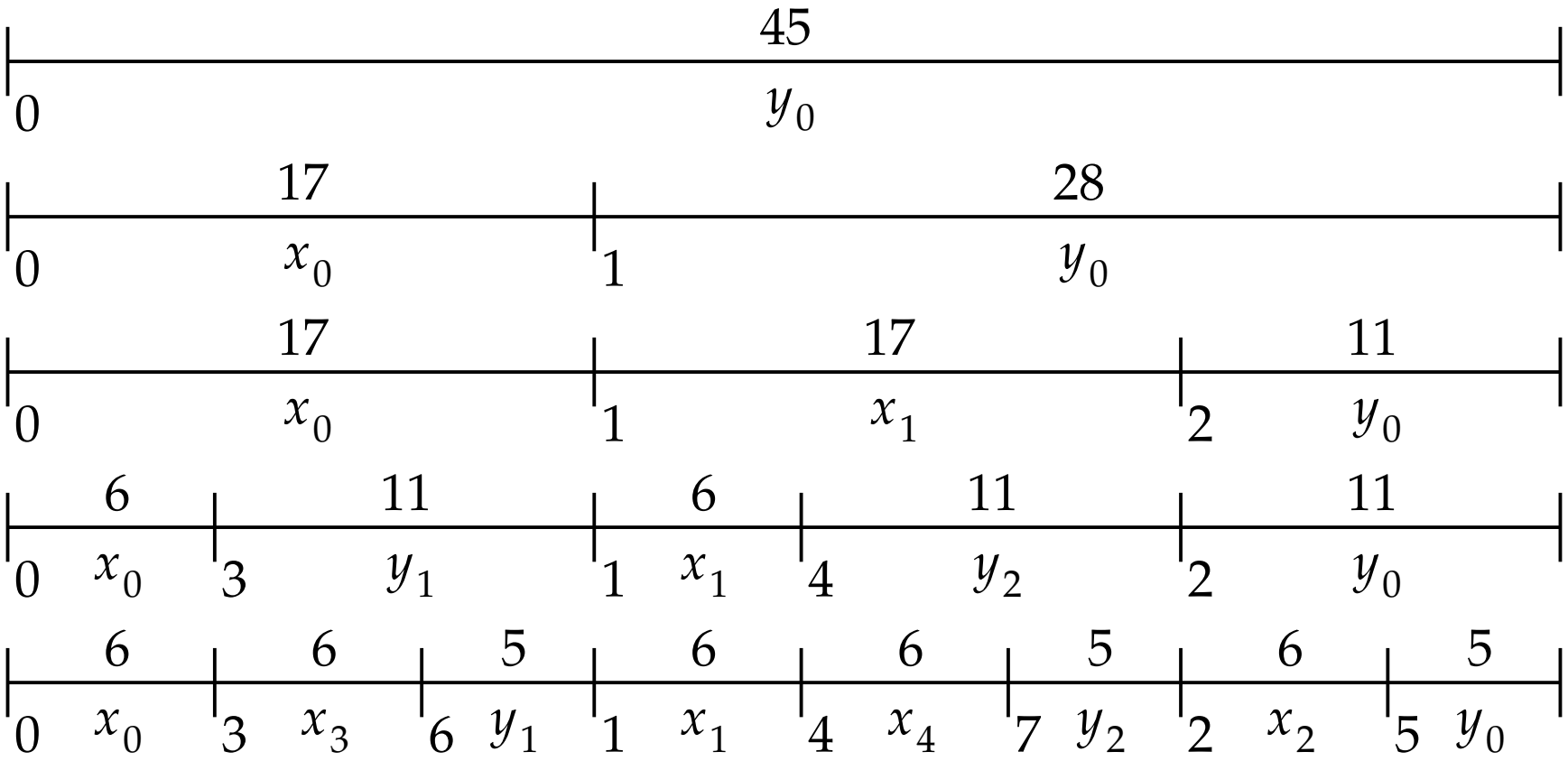
- Intervals x_0, x_1, ..., x_{u-1} of length x, where x_0 is the left-most interval and $x_r = x_0 + r.a$ (translation by $r.a$ modulo 1).
- Intervals y_0, y_1, ..., y_{v-1} of length y, where y_0 is the right-most interval and $y_r = y_0 + r.a$ (translation by $r.a$ modulo 1).
- Total number of points (or intervals): $n = u + v$ (determined by induction).

In short: **2 primary intervals x_0 (left) and y_0 (right) + images.**

Initial configuration: $n = 2$, $u = v = 1$.

Example: The First Configurations

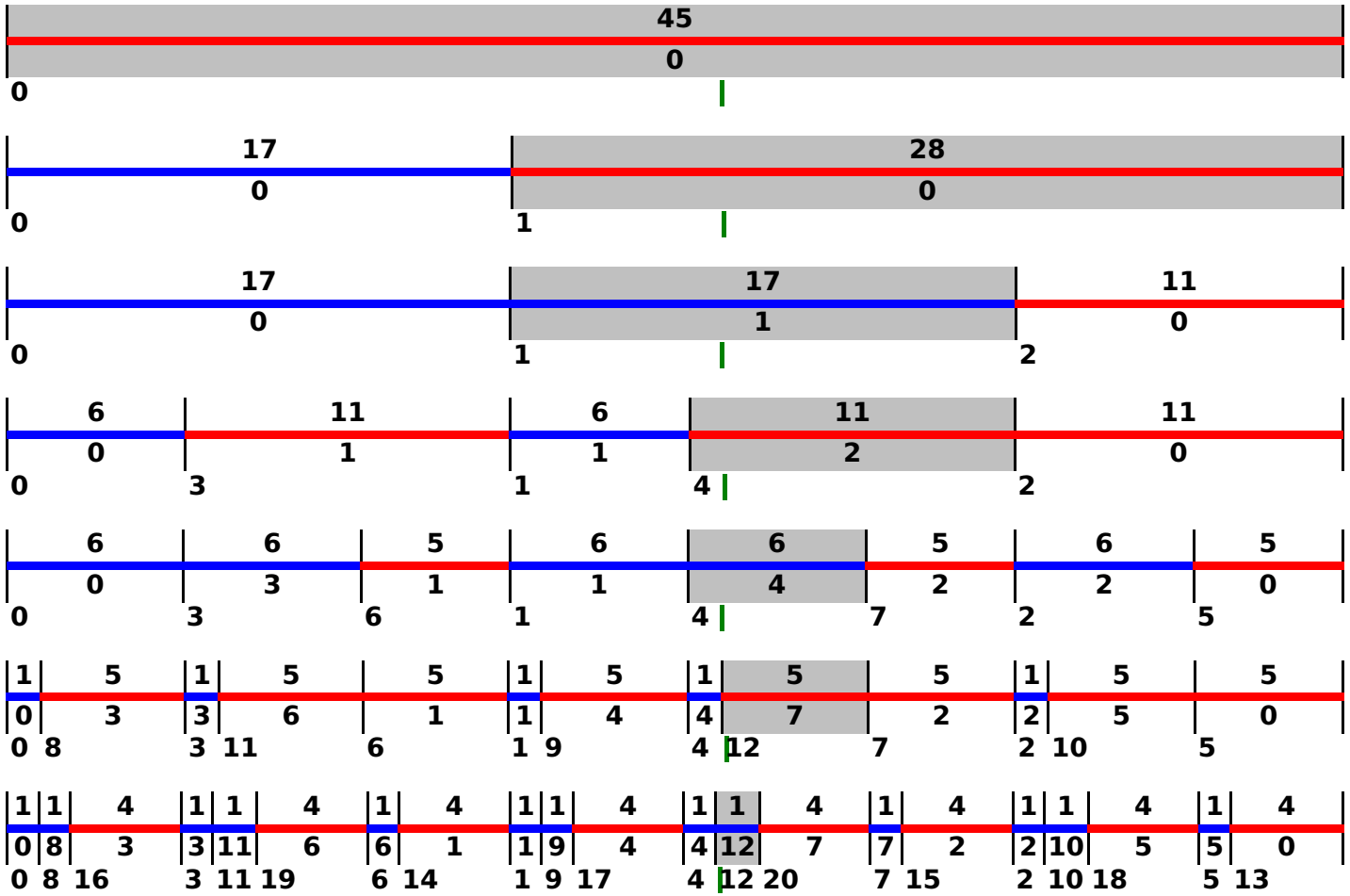
with $a = 17/45$.



Note: scaling by 45 on the figure.

Example: The First Configurations [2]

with $a = 17/45$ and $b = 24/45$.



From a Configuration to the Next One

The main idea: when adding new points, one of the primary intervals (no inverse image) is affected first, then all its images are affected in the same way.
For instance, see both intervals of length 17 on the figure.

- Since a is irrational, $n.a$ is strictly between two points of smaller indices, one of which, denoted r is non zero.
- Therefore the points of indices $r-1$ and $n-1$ (obtained by a translation) are adjacent, and their distance ℓ is either x or y.
  $\rightarrow$ Same distance ℓ between the points of indices r and n.
- Thus the new point n splits an interval of length $h = \max(x, y)$ into two intervals of respective lengths $\ell = \min(x, y)$ and $h - \ell$.
- The length $h - \ell$ is new, therefore the corresponding interval does not have an inverse image (i.e. by adding $-a$).
- Therefore this interval has as a boundary point of index 0.

$\rightarrow$ As a consequence, the point of index n is completely determined.

From a Configuration to the Next One [2]

The other intervals of length h will be split in the same way, one after the other with increasing indices (translations by a).

- Indices of the intervals of length $h - \ell$: these are the indices of the corresponding intervals of length h.
- Indices of the intervals of length ℓ: assume that $\ell = x$ (same reasoning for $\ell = y$); the first interval of length x is obtained by a translation of an old interval of length x (as shown in previous slide), necessarily x_{u-1} (the last one) since the image of x_{i-1} is x_i for all $i < u$. Thus this interval is x_u and we have $x_u = x_0 + u.a$. The next intervals: x_{u+1}, x_{u+2}, etc.

For the algorithm(s):

- We only need to focus on what occurs in the primary intervals.
- At the same time, we track the position of the point b:
  - whether it is in an interval x_k or in an interval y_k;
  - its distance to the left endpoint of the interval.

The Algorithms

Basic algorithm (1997): returns a lower bound d on $\{b - n.a\}$ for $n \in [\![0, N-1]\!]$ (in fact, d is the *exact* distance for $n \in [\![0, N'-1]\!]$, where $N \leq N' < 2N$).

Here: parameters chosen so that $d \geq d_0$ in most intervals, allowing to immediately conclude that there are no worst cases in the interval.

New algorithm (mentioned in 1998): returns the index $n < N$ of the first point such that $\{b - n.a\} < d_0$, otherwise any value $\geq N$ if there are no such points.

Gives the information we need, but uses an additional variable, so that it is slower.

Good replacement for the naive algorithm.

Another improvement: test with a shift (fast!) if it is interesting to replace a sequence of iterations by a single one with a division.

The Algorithms [2]

The necessary data:

- the lengths x and y, and the numbers u and v of these intervals;
- a binary value saying whether the point b is in an interval of length x or y;
- the index r of this interval (new algorithm only);
- the distance d between b and the left endpoint of this interval.

Immediate consequence of the properties:

- the left endpoint of an interval x_r has index r;
- the left endpoint of an interval y_r has index $u + r$.

Subtractive Version of the Algorithms

In red: additional statements for the new algorithm.

Initialization: $x = \{a\}\,;\ y = 1 - \{a\}\,;\ d = \{b\}\,;\ u = v = 1\,;\ r = 0\,;$

if $(d < d_0)$ **return** 0

Unconditional loop:

if $(d < x)$

- **while** $(x < y)$
  - **if** $(u + v \geq N)$ **return** N
  - $y = y - x;\ u = u + v;$
- **if** $(u + v \geq N)$ **return** N
- $x = x - y;$
- **if** $(d \geq x)$ $r = r + v;$
- $v = v + u;$

else

- $d = d - x;$
- **if** $(d < d_0)$ **return** $r + u$
- **while** $(y < x)$
  - **if** $(u + v \geq N)$ **return** N
  - $x = x - y;\ v = v + u;$
- **if** $(u + v \geq N)$ **return** N
- $y = y - x;$
- **if** $(d < x)$ $r = r + u;$
- $u = u + v;$

Divisions Can Be Introduced

But for most iterations (small partial quotient), a succession of subtractions is faster. → LOGMS parameter (the value 3 seemed to be the best choice).

Excerpt of generated code

```
if (LOGMS == 0 || (LOGMS < 64 && (y >> LOGMS) > x))
  {
    uint64_t q = y / x;
    TESTEND(q >= N);  /* avoid overflow below */
    y -= (unsigned int) q * x;
    u += (unsigned int) q * v;
  }
else
  while (x < y)
    {
      TESTEND(u + v >= N);
      y -= x;
      u += v;
    }
```

Divisions Can Be Introduced [2]

Notations for the following timings:

- Option c=k: a succession of subtractions are replaced by a single division when one needs to do at least 2^k subtractions without modifying the value d ($-$: subtractive algorithm only).
- Algorithm selection:

	$-$	l=3	w	old w
default	basic	basic	basic	new
if failed	naive	8-split	new	
if failed		naive		

8-split: the interval is split into $2^3 = 8$ subintervals and the basic algorithm is tried again.

Divisions Can Be Introduced [3]

Tests on a 2 GHz AMD Opteron (2005).

	$\exp x$, exponent 0				2^x, exponent 0			
c	–	l=3	w	old w	–	l=3	w	old w
0	42.30	35.46	35.26	(39.22)	37.83	32.95	32.82	(49.24)
1	26.32	19.27	19.09	(18.40)	23.83	18.72	18.67	(20.45)
3	24.09	16.82	16.85	(16.67)	22.21	16.96	17.04	(18.79)
5	24.47	17.29	17.29	(16.76)	23.23	18.03	18.08	(19.04)
–	21.54	14.23	14.26	(15.38)	21.68	16.42	16.52	(18.36)

	$\sin x$, exponent 0				$\cos x$, exponent 0			
c	–	l=3	w	old w	–	l=3	w	old w
0	40.24	31.72	31.67	(42.88)	39.08	33.52	33.51	(36.04)
1	28.28	19.52	19.49	(19.58)	25.87	20.10	20.18	(19.61)
3	26.41	17.54	17.55	(17.72)	22.76	16.93	17.08	(17.11)
5	27.15	18.36	18.32	(17.55)	23.15	17.29	17.47	(17.24)
–	23.71	14.74	14.85	(16.11)	19.99	14.12	14.30	(15.20)

Divisions Can Be Introduced [4]

Tests on a 2 GHz AMD Opteron (2005).

	$\exp x$, exponent -6				2^x, exponent -6			
c	–	l=3	w	old w	–	l=3	w	old w
0	18.29	18.15	18.09	(59.08)	21.42	21.31	21.27	(81.95)
1	12.54	12.52	12.51	(18.05)	13.27	13.18	13.16	(22.15)
3	12.10	11.95	11.86	(17.07)	12.84	12.91	12.68	(21.26)
5	14.41	14.31	14.16	(17.65)	14.67	14.56	14.54	(22.34)
–	22.13	21.94	21.97	(26.25)	17.62	17.40	17.44	(21.31)

	$\sin x$, exponent -6				$\cos x$, exponent -6			
c	–	l=3	w	old w	–	l=3	w	old w
0	15.74	15.56	15.59	(16.21)	15.61	15.43	15.44	(19.10)
1	10.22	10.06	10.10	(9.79)	10.72	10.57	10.58	(10.74)
3	9.45	9.25	9.26	(9.33)	10.12	9.99	10.04	(10.58)
5	9.34	9.16	9.20	(9.30)	10.50	10.30	10.33	(10.72)
–	314.8	314.3	314.6	(369.9)	161.3	161.1	161.1	(188.6)

Example of Domain Splitting

Input interval $[1, 2[$ decomposed into $2^{13} = 8192$ sub-intervals I.

For each sub-interval I of size 2^{40}:

- Function f is approximated by a degree-d polynomial.
- Code (C with the mpn layer of GMP) is generated: my algorithm is applied on sub-intervals J of $2^{15} = 32768$ points (64-bit integer arithmetic), and in case of failure, $2^{12} = 4096$ (or $2^{11} = 2048$) points, and if this still fails, the naive method (difference table). Note: this can probably be improved, e.g. larger intervals J (with 128-bit arithmetic?), variant instead of the naive method...
- If GCC is used, the code is compiled using `-fprofile-generate` and tested on the first $2^8 = 256$ sub-intervals (for up to 22% speed-up on Opteron).
- The code is recompiled using `-fprofile-use` and run.

The accuracy (chosen for efficiency) is not sufficient to determine the worst cases. A second filter step is necessary: conventional algorithm (much slower but run on much fewer inputs) on each potential worst case.

Polynomial Degree and Coefficient Size

Examples with a 54-bit significand and splitting into intervals of size 2^{40}.

For some functions and left endpoints of the interval, the table gives the degree of the polynomial and the size (in bits) of the coefficient of highest degree.

function	x_0	degree	size
$\exp x$	1	6	320
$\exp x$	8	7	352
$\exp x$	64	9	416
$\log x$	2	6	320
$\log x$	2^{1000}	6	320
x^4	1	4	224
x^{17}	1	8	384
x^{345}	1	12	544
x^{2065}	1	18	736
x^{2065}	$2-\varepsilon$	15	640

Timings (Example)

Note: timings can differ from an interval to the other and parameters may not be optimal, but the following timings give orders of magnitude. . .

Timings in seconds on a recent machine (Intel Xeon E5520 at 2.27 GHz, only one core used) for function exp:

Exponent	Interval	
	0	8191
0	11.8	11.8
1	12.4	12.2
2	16.2	15.8
3	22.0	20.0
4	34.7	37.3
5	51.6	58.4
6	84.1	92.1
7	209	177

SLZ Algorithm: the Problem

My algorithm: limited to degree-1 polynomials.

Algorithm working with degree-d polynomials and asymptotically faster?

SLZ Algorithm: the Problem

My algorithm: limited to degree-1 polynomials.

Algorithm working with degree-d polynomials and asymptotically faster?

Real Small Value Problem (Real SValP)

Given positive integers M and T, and a polynomial $P \in \mathbb{R}[X]$, find all integers $|t| < T$ such that:

$$|P(t) \bmod 1| < \frac{1}{M}$$

SLZ Algorithm: the Problem

My algorithm: limited to degree-1 polynomials.

Algorithm working with degree-d polynomials and asymptotically faster?

Real Small Value Problem (Real SVaIP)

Given positive integers M and T, and a polynomial $P \in \mathbb{R}[X]$, find all integers $|t| < T$ such that:

$$|P(t) \bmod 1| < \frac{1}{M}$$

Integer Small Value Problem

Given $P \in \mathbb{Z}[X]$ of degree d, find on which small integer entries it has small values modulo a large integer N, i.e. find the small integer roots of

$$Q(x, y) = P(x) + y \pmod{N}$$

SLZ Algorithm: Lattice Reduction Theory / LLL

Lattice: a discrete subgroup of $\mathbb{R}^n$.

$$L = \left\{ \sum_{i=1}^{\ell} n_i \mathbf{b}_i \mid n_i \in \mathbb{Z} \right\}$$

where the $\mathbf{b}_i$'s are linearly independent vectors.

Let $\{\mathbf{b}_1, \ldots, \mathbf{b}_\ell\}$ be a basis of a lattice $L \subset \mathbb{Z}^n$. The well-known LLL algorithm computes, in time polynomial in the bit length of the input, a basis $\{\mathbf{v}_1, \ldots, \mathbf{v}_\ell\}$ satisfying:

- $||\mathbf{v}_1|| \leq 2^{\frac{\ell}{2}} \det(L)^{\frac{1}{\ell}}$;
- $||\mathbf{v}_2|| \leq 2^{\frac{\ell}{2}} \det(L)^{\frac{1}{\ell-1}}$.

SLZ Algorithm: Coppersmith's Technique

Without details...

Let α be a positive integer. One considers the family of polynomials

$$Q_{i,j}(x,y) = x^i\, Q^j(x,y)\, N^{\alpha-j}$$

with $0 \leq i + dj \leq d\alpha$.

If (x_0, y_0) is a root of Q modulo N, then it is a root of each $Q_{i,j}$ modulo N^α.

LLL $\rightarrow$ two polynomials $v_1(x,y)$ and $v_2(x,y)$ linear combinations of the $Q_{i,j}$'s, which take small values ($< N^\alpha$) for small x and y, i.e. $|x| \leq X$ and $|y| \leq Y$.

Thus, if (x_0, y_0) is a root of v_1 and v_2 modulo N^α, then it is also a root on $\mathbb{Z}$.

$\rightarrow$ Search for x_0 by finding the integer roots of the resultant $\mathrm{Res}_y(v_1, v_2) \in \mathbb{Z}[x]$, assuming $\mathrm{Res}_y(v_1, v_2) \neq 0$.

Note about X and Y: one can show that $X^d Y = O(N)$.

SLZ Algorithm

Input: function f, positive integers N, T, M, d, α.

1. $P(t)$: Taylor expansion of $N f(t/N)$ up to order d; $n = \frac{(\alpha+1)(d\alpha+2)}{2}$.
2. Compute ε such that $|P(t) - N f(t/N)| < \varepsilon$ for $|t| \leq T$.
3. $M' = \left\lfloor \frac{1/2}{1/M + \varepsilon} \right\rfloor$, $C = (d+1)M'$ and $P'(x) = \frac{1}{C} \lfloor CP(Tx) \rceil$.
4. $\{e_1, \ldots, e_n\} \leftarrow \{x^i y^j\}$ for $0 \leq i + dj \leq d\alpha$.
5. $\{g_1, \ldots, g_n\} \leftarrow \{C^\alpha (Tx)^i (P'(x) + \frac{y}{M'})^j\}$ for $0 \leq i + dj \leq d\alpha$.
6. Form the integral matrix L where $L_{k,\ell}$ is the coefficient of e_k in g_ℓ.
7. $V \leftarrow C^{-\alpha}$LatticeReduce(L).
8. v_1 and v_2: the two smallest vectors from V;
   p_1 and p_2: corresponding polynomials.
9. If $\exists\, x, y \in [-1, 1]$ with $|p_1(x, y)| \geq 1$ or $|p_2(x, y)| \geq 1$, then return `FAIL`.
10. $p(t) \leftarrow \mathrm{Res}_y(p_1(t/T, y), p_2(t/T, y))$.
    If $p = 0$, then return `FAIL` (should never occur).
11. For each root $t_0 \in [-T, T]$ of p, check whether it is a bad case.

Periodical Functions with Large Arguments

The described methods work well only if the function can be approximated by a small-degree polynomial. Not the case with large arguments for:

- exponentials, but one gets an *overflow* before the problem becomes too hard;
- periodical functions: deductions if the period is a rational, otherwise...

Periodical Functions with Large Arguments

The described methods work well only if the function can be approximated by a small-degree polynomial. Not the case with large arguments for:

- exponentials, but one gets an *overflow* before the problem becomes too hard;
- periodical functions: deductions if the period is a rational, otherwise...

The problem: consecutive arguments are not close to each other modulo 2π.

Periodical Functions with Large Arguments

The described methods work well only if the function can be approximated by a small-degree polynomial. Not the case with large arguments for:

- exponentials, but one gets an *overflow* before the problem becomes too hard;
- periodical functions: deductions if the period is a rational, otherwise...

The problem: consecutive arguments are not close to each other modulo 2π.

But consider a whole binade $[2^e, 2^{e+1}[$, and all the reduced arguments modulo 2π: the average distance between two consecutive values is similar to the one for basic functions. The arguments are just in the wrong order!

Periodical Functions with Large Arguments

The described methods work well only if the function can be approximated by a small-degree polynomial. Not the case with large arguments for:

- exponentials, but one gets an *overflow* before the problem becomes too hard;
- periodical functions: deductions if the period is a rational, otherwise...

The problem: consecutive arguments are not close to each other modulo 2π.

But consider a whole binade $[2^e, 2^{e+1}[$, and all the reduced arguments modulo 2π: the average distance between two consecutive values is similar to the one for basic functions. The arguments are just in the wrong order!

Moreover $2^e + k.\mathrm{ulp}(2^e) \bmod 2\pi$ is just like $k.a \bmod 1$.

Periodical Functions with Large Arguments

The described methods work well only if the function can be approximated by a small-degree polynomial. Not the case with large arguments for:

- exponentials, but one gets an *overflow* before the problem becomes too hard;
- periodical functions: deductions if the period is a rational, otherwise...

The problem: consecutive arguments are not close to each other modulo 2π.

But consider a whole binade $[2^e, 2^{e+1}[$, and all the reduced arguments modulo 2π: the average distance between two consecutive values is similar to the one for basic functions. The arguments are just in the wrong order!

Moreover $2^e + k.\mathrm{ulp}(2^e) \bmod 2\pi$ is just like $k.a \bmod 1$.

An idea: extract subsequences in arithmetic progression.

Periodical Functions with Large Arguments

The described methods work well only if the function can be approximated by a small-degree polynomial. Not the case with large arguments for:

- exponentials, but one gets an *overflow* before the problem becomes too hard;
- periodical functions: deductions if the period is a rational, otherwise...

The problem: consecutive arguments are not close to each other modulo 2π.

But consider a whole binade $[2^e, 2^{e+1}[$, and all the reduced arguments modulo 2π: the average distance between two consecutive values is similar to the one for basic functions. The arguments are just in the wrong order!

Moreover $2^e + k.\mathrm{ulp}(2^e) \bmod 2\pi$ is just like $k.a \bmod 1$.

An idea: extract subsequences in arithmetic progression.

Or any hope to use more properties of $k.a \bmod 1$?

Results

Worst cases for the double-precision functions:

- e^x, 2^x, 10^x, sinh, cosh, $\sin(2\pi x)$, $\cos(2\pi x)$;
- x^n for $3 \leq n \leq 2381$ and $-180 \leq n \leq -2$;
- sin, cos, tan between $-\pi/2$ and $\pi/2$;
- the corresponding inverse functions.

Results

Worst cases for the double-precision functions:

- e^x, 2^x, 10^x, sinh, cosh, $\sin(2\pi x)$, $\cos(2\pi x)$;
- x^n for $3 \leq n \leq 2381$ and $-180 \leq n \leq -2$;
- sin, cos, tan between $-\pi/2$ and $\pi/2$;
- the corresponding inverse functions.

A few bad cases:

- $x = 1.11100001001011010110011001110100010011111111110000001 \times 2^{429}$:

$$\begin{aligned} \log_{10} x &= 10000001.01101010011110101001100 \\ &\quad 11100101001111001000100\ 1\ \underbrace{000\ldots000}_{68\text{ bits}}\ 10\ldots \end{aligned}$$

- $x = 1.11011100101110100000110001001000101100111111100101001 \times 2^{253}$:

$$\begin{aligned} x^{1/1039} &= 1.00101111010000000010011110110 \\ &= 0100101101011001101111\ 0\ \underbrace{111\ldots111}_{73\text{ bits}}\ 01\ldots \end{aligned}$$