On Correctness of Numerical Libraries

Wonyeol Lee

POSTECH

@소프트웨어재난연구센터 워크샵, 02/06/2025

Research Interests

Mathematical Properties of Programs and Computations

Research Interests

Mathematical Properties of Programs and Computations

Correctness

Efficiency

Fundamental Limit

...

Research Interests

Mathematical Properties of Programs and Computations

"Continuous" values

$6,\ 2.8,\ \frac{3}{7},\ \sqrt{5},\ \frac{\pi}{4},\ \ldots$

Operations on them

$6 + 2.8,\ \frac{3}{7} \times \sqrt{5},\ \sin\left(\frac{\pi}{4}\right),\ \ldots$

Research Interests

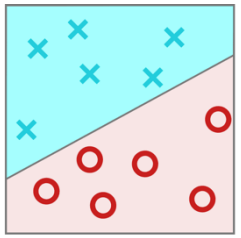
Mathematical Properties of Programs and Computations

"Continuous" values

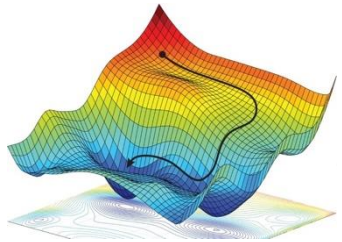
$6,\ 2.8,\ \frac{3}{7},\ \sqrt{5},\ \frac{\pi}{4},\ \ldots$

Operations on them

$6 + 2.8,\ \frac{3}{7} \times \sqrt{5},\ \sin\left(\frac{\pi}{4}\right),\ \ldots$

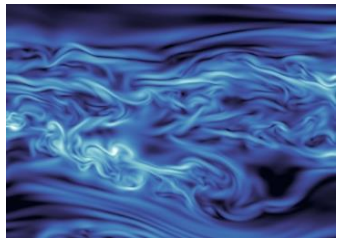


Machine Learning



Optimization

Computer Graphics



Scientific Computing

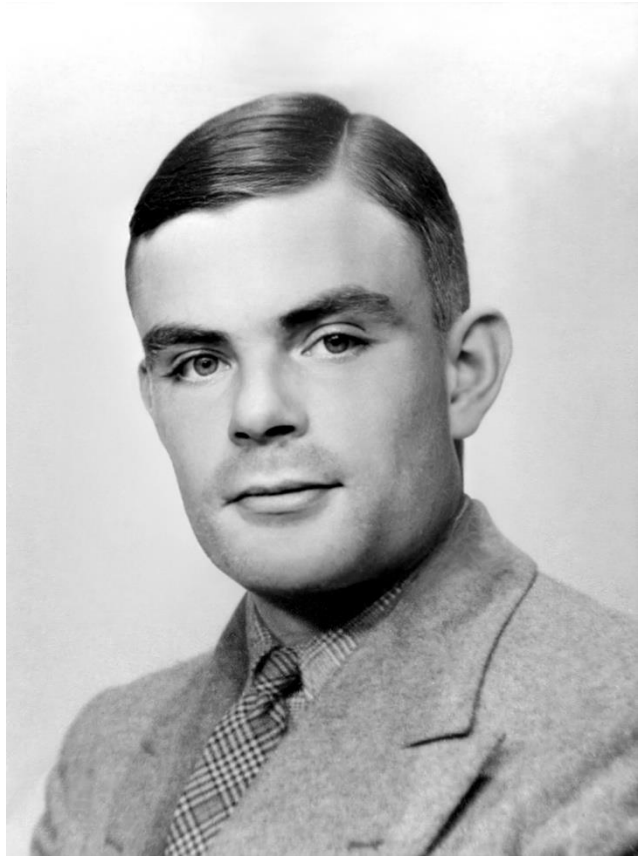
• • •

“Continuous” Computations

Early Works

?

Early Works



Alan Turing

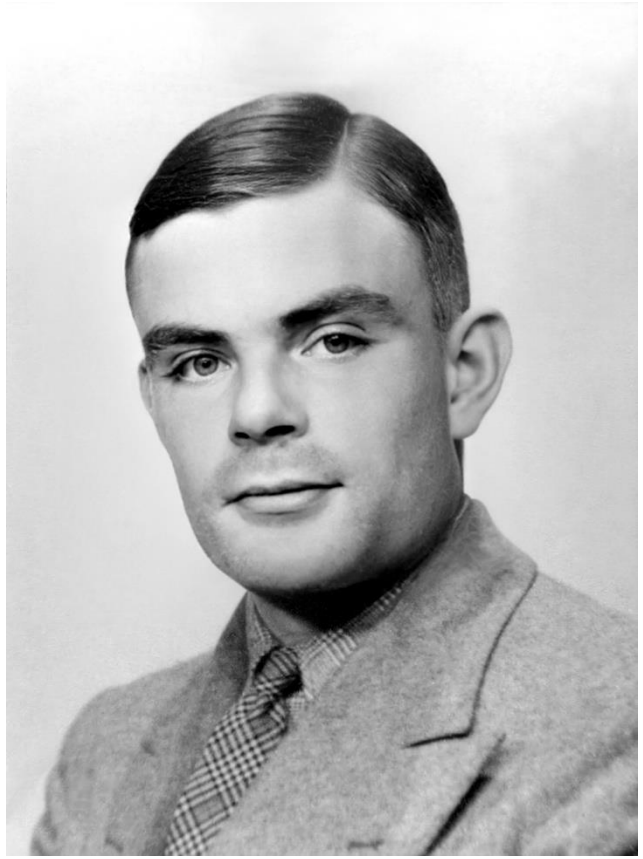
ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

Early Works



Alan Turing

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

(vii) A power series whose coefficients form a computable sequence of computable numbers is computably convergent at all computable points in the interior of its interval of convergence.

(viii) The limit of a computably convergent sequence is computable.

And with the obvious definition of "uniformly computably convergent":

(ix) The limit of a uniformly computably convergent computable sequence of computable functions is a computable function. Hence

(x) The sum of a power series whose coefficients form a computable sequence is a computable function in the interior of its interval of convergence.

From (viii) and $\pi = 4(1-\frac{1}{3}+\frac{1}{5}-\ldots)$ we deduce that π is computable.

From $e = 1+1+\frac{1}{2!}+\frac{1}{3!}+\ldots$ we deduce that e is computable.

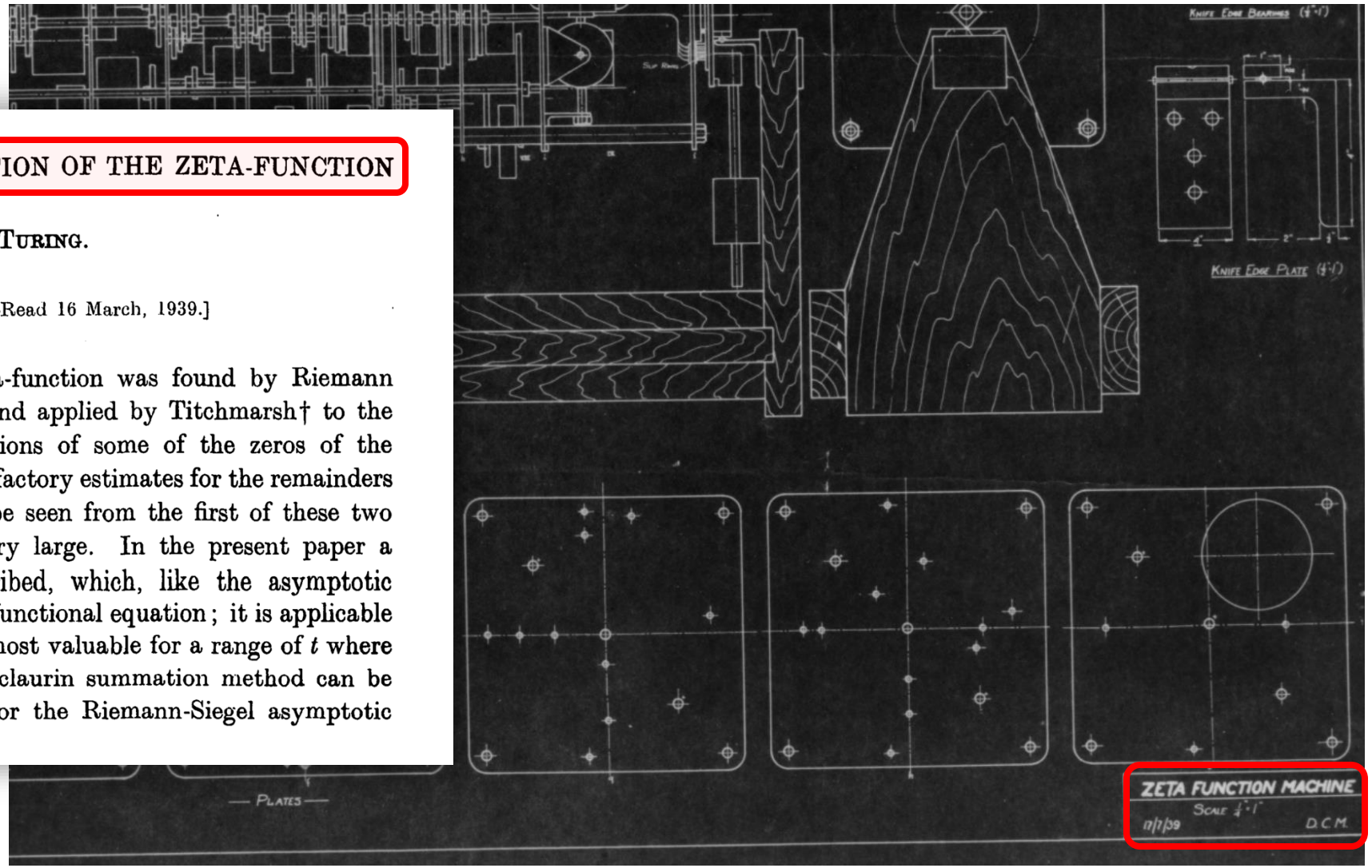
Early Works

A METHOD FOR THE CALCULATION OF THE ZETA-FUNCTION

By A. M. Turing.

[Received 7 March, 1939.—Read 16 March, 1939.]

An asymptotic series for the zeta-function was found by Riemann and has been published by Siegel*, and applied by Titchmarsh† to the calculation of the approximate positions of some of the zeros of the function. It is difficult to obtain satisfactory estimates for the remainders with this asymptotic series, as may be seen from the first of these two papers of Titchmarsh, unless t is very large. In the present paper a method of calculation will be described, which, like the asymptotic formula, is based on the approximate functional equation; it is applicable for all values of s. It is likely to be most valuable for a range of t where t is neither so small that the Euler-Maclaurin summation method can be used (*e.g.* $t > 30$) nor large enough for the Riemann-Siegel asymptotic formula (*e.g.* $t < 1000$).



Early Works

SOME CALCULATIONS OF THE RIEMANN ZETA-FUNCTION

By A. M. TURING

[Received 29 February 1952.—Read 20 March 1952]

Introduction

In June 1950 the Manchester University Mark 1 Electronic Computer was used to do some calculations concerned with the distribution of the zeros of the Riemann zeta-function. It was intended in fact to determine whether there are any zeros not on the critical line in certain particular intervals. The calculations had been planned some time in advance, but had in fact to be carried out in great haste. If it had not been for the fact that the computer remained in serviceable condition for an unusually long period from 3 p.m. one afternoon to 8 a.m. the following morning it is probable that the calculations would never have been done at all. As it was, the interval $2\pi.63^2 < t < 2\pi.64^2$ was investigated during that period, and very little more was accomplished

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Included amongst the methods considered is a generalization of Choleski's method which appears to have advantages over other known methods both as regards accuracy and convenience. This method may also be regarded as a rearrangement of the elimination process.

THIS paper contains descriptions of a number of methods for solving sets of linear simultaneous equations and for inverting matrices, but its main concern is with the theoretical limits of accuracy that may be obtained in the application of these methods, due to rounding-off errors.

Modern Libraries

As a result of ~90 years of substantial efforts:

Modern Libraries

As a result of ~90 years of substantial efforts:

 GNU Math/Scientific Library

intel Intel MKL

 NumPy

 SciPy

 TensorFlow

 PyTorch

 JAX

 Pyro

 Stan

• • •

What are the most **fundamental** computations?

Modern Libraries

Function Evaluation Compute $\sin(x)$.

 GNU Math Library intel Intel MKL ⋯

Modern Libraries

Function Evaluation Compute $\sin(x)$.  GNU Math Library intel Intel MKL ⋯

Sample Generation Sample from $\mathcal{N}(\mu, \sigma^2)$. NumPy SciPy ⋯

Modern Libraries

Function Evaluation	Compute $\sin(x)$.	 GNU Math Library intel Intel MKL ⋯
Sample Generation	Sample from $\mathcal{N}(\mu, \sigma^2)$.	NumPy SciPy ⋯
Differentiation	Compute $\nabla f(x)$.	TensorFlow PyTorch ⋯

Modern Libraries

Function Evaluation	Compute $\sin(x)$.	GNU Math Library, Intel MKL, ⋯
Sample Generation	Sample from $\mathcal{N}(\mu, \sigma^2)$.	NumPy, SciPy, ⋯
Differentiation	Compute $\nabla f(x)$.	TensorFlow, PyTorch, ⋯
Integration (≈ Probabilistic Inference)	Compute $\int f(x)\, dx$.	Pyro, Stan, ⋯

Modern Libraries

Function Evaluation	Compute $\sin(x)$.	GNU Math Library, Intel MKL, ⋯
Sample Generation	Sample from $\mathcal{N}(\mu, \sigma^2)$.	NumPy, SciPy, ⋯
Differentiation	Compute $\nabla f(x)$.	TensorFlow, PyTorch, ⋯
Integration **(≈ Probabilistic Inference)**	Compute $\int f(x)\, dx$.	Pyro, Stan, ⋯
Function Approximation	Approx. f using neural nets.	TensorFlow, PyTorch, ⋯

Research Questions

Correct?

Function Evaluation	Compute $\sin(x)$.	 GNU Math Library intel Intel MKL ···
Sample Generation	Sample from $\mathcal{N}(\mu, \sigma^2)$.	NumPy SciPy ···
Differentiation	Compute $\nabla f(x)$.	TensorFlow PyTorch ···
Integration **(≈ Probabilistic Inference)**	Compute $\int f(x)\, dx$.	Pyro Stan ···
Function Approximation	Approx. f using neural nets.	TensorFlow PyTorch ···

Research Questions

Function Evaluation	Compute $\sin(x)$.	GNU Math Library, Intel MKL, ⋯
Sample Generation	Assume **reals.**	NumPy, SciPy, ⋯
Differentiation	Assume **differentiability.**	TensorFlow, PyTorch, ⋯
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	Pyro, Stan, ⋯
Function Approximation		TensorFlow, PyTorch, ⋯

Underlying algorithms

Research Questions

Actual implementations

Function Evaluation	Use **floats** intricately.	GNU Math Library, intel Intel MKL, …
Sample Generation	Assume **reals.**	NumPy, SciPy, …
Differentiation	Assume **differentiability.**	TensorFlow, PyTorch, …
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	Pyro, Stan, …
Function Approximation		TensorFlow, PyTorch, …

Underlying algorithms

Research Questions

Mathematically **Correct**?
Fundamental **Limits**?

Actual implementations

Function Evaluation	Use **floats** intricately.	GNU Math Library, intel Intel MKL, …
Sample Generation	Assume **reals.**	NumPy, SciPy, …
Differentiation	Assume **differentiability.**	TensorFlow, PyTorch, …
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	Pyro, Stan, …
Function Approximation		TensorFlow, PyTorch, …

Underlying algorithms

Research Questions

Mathematically **Correct**?
Fundamental **Limits**?

Function Evaluation	Use **floats** intricately.	[Ongoing], [POPL'18], [PLDI'16].
Sample Generation	Assume **reals.**	[Ongoing], [Submitted].
Differentiation	Assume **differentiability.**	[ICLR'24] (Spotlight), [ICML'23], [NeurIPS'20] (Spotlight).
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	[Submitted], [POPL'23], [POPL'20], [AAAI'20], [NeurIPS'18].
Function Approximation		[Submitted], [Submitted], [Neural Networks'24], [TMLR'23].

Research Questions

Mathematically **Correct**?
Fundamental **Limits**?

Function Evaluation	Use **floats** intricately.	[Ongoing], [POPL'18], [PLDI'16].
Sample Generation	Assume **reals.**	[Ongoing], [Submitted].
Differentia		light), 0] (Spotlight).
Integration (≈ Probabilist		], [POPL'20], PS'18].
Function Approximation		[Submitted], [Submitted], [Neural Networks'24], [TMLR'23].

- What types of **programs** are used?
- What types of **correctness issues** can arise?
- What are key **research directions**?

Function Evaluation

Problem

Given an input $x \in \mathbb{F}$, compute $f(x) \in \mathbb{R}$ accurately and efficiently,

for $f \in$ {exp, ln, sin, asin, sinh, asinh, erf, gamma, ...}.

Problem

Given an input $x \in \mathbb{F}$, compute $f(x) \in \mathbb{R}$ **accurately** and efficiently,

for $f \in$ {exp, ln, sin, asin, sinh, asinh, erf, gamma, ...}.

- Almost all values of $f(x)$ are not floats (and even not rationals).
  - $\exp(x) \notin \mathbb{Q}$ for all $x \neq 0$.
    $\ln(x) \notin \mathbb{Q}$ for all $x \neq 1$.
    $\sin(x) \notin \mathbb{Q}$ for all $x \neq 0$.
    $\text{asin}(x) \notin \mathbb{Q}$ for all $x \neq 0$.
    ...
  - By Lindemann-Weierstrass Theorem (1885) and Siegel–Shidlovsky Theorem (1929).

Problem

Given an input $x \in \mathbb{F}$, compute $f(x) \in \mathbb{R}$ accurately and **efficiently,**

for $f \in$ {exp, ln, sin, asin, sinh, asinh, erf, gamma, ...}.

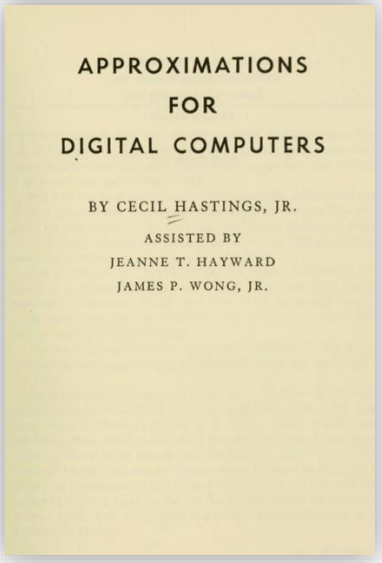
- Compute $y \in \mathbb{F}$ such that $y \approx f(x)$.
- Use floating-point arithmetic (i.e., $+, -, \times, /, \sqrt{\ }$).

Problem

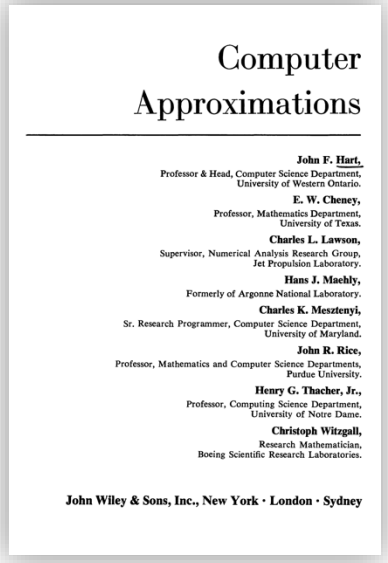
Given an input $x \in \mathbb{F}$, compute $f(x) \in \mathbb{R}$ accurately and efficiently,

for $f \in$ {exp, ln, sin, asin, sinh, asinh, erf, gamma, ...}.

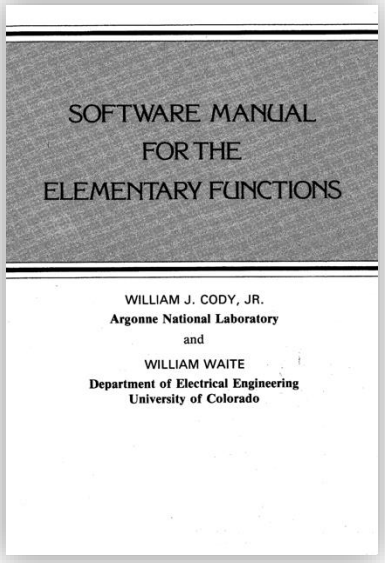
- Well-studied for >70 years.



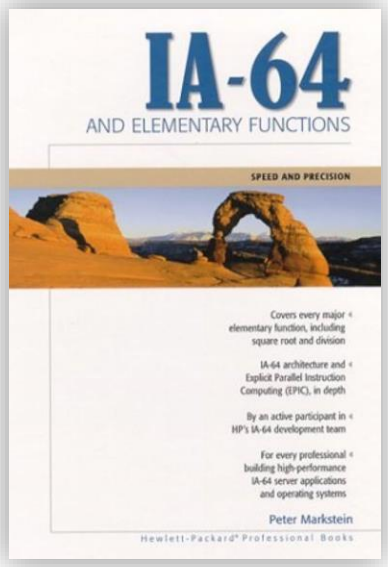
1955



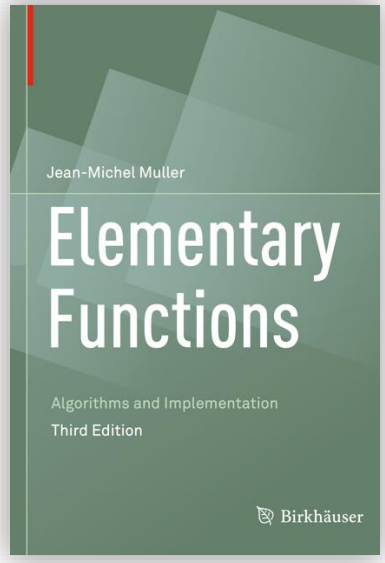
1968



1980



2000



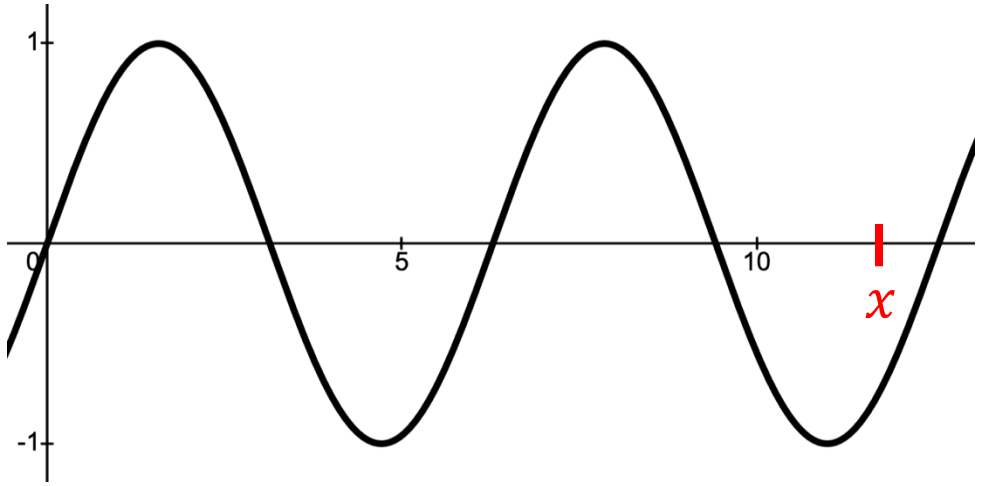
2016

IEEE standard for floats (1985)

Example

Input: $x \in \mathbb{F}$. Then, $x \in [-2^{1024}, 2^{1024}]$ in 64-bit double.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

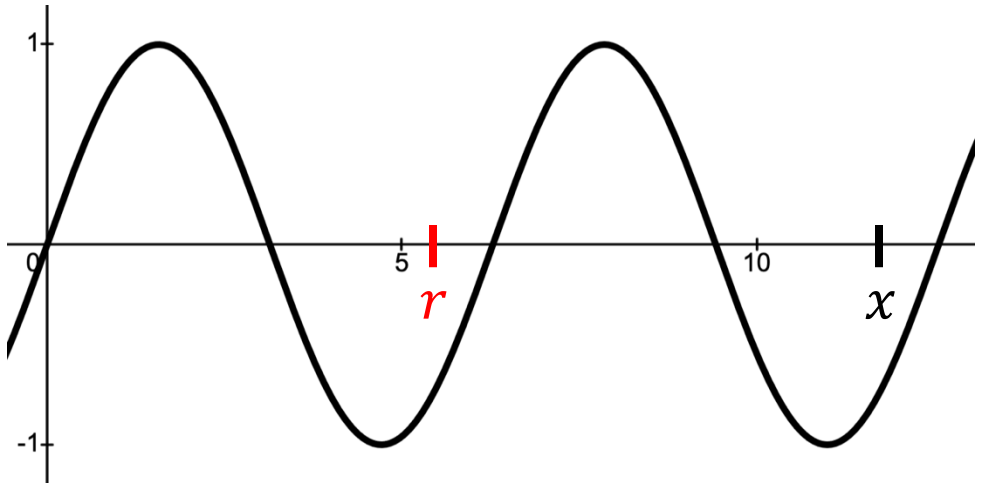


Example (1)

Input: $x \in \mathbb{F}$. Then, $x \in [-2^{1024}, 2^{1024}]$ in 64-bit double.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 1: Use symmetry.
  - $x = 2\pi k + r$ for some $k \in \mathbb{N}$ and $r \in [0, 2\pi)$.
  - Then, $\sin(x) = \sin(2\pi k + r) = \sin(r)$.

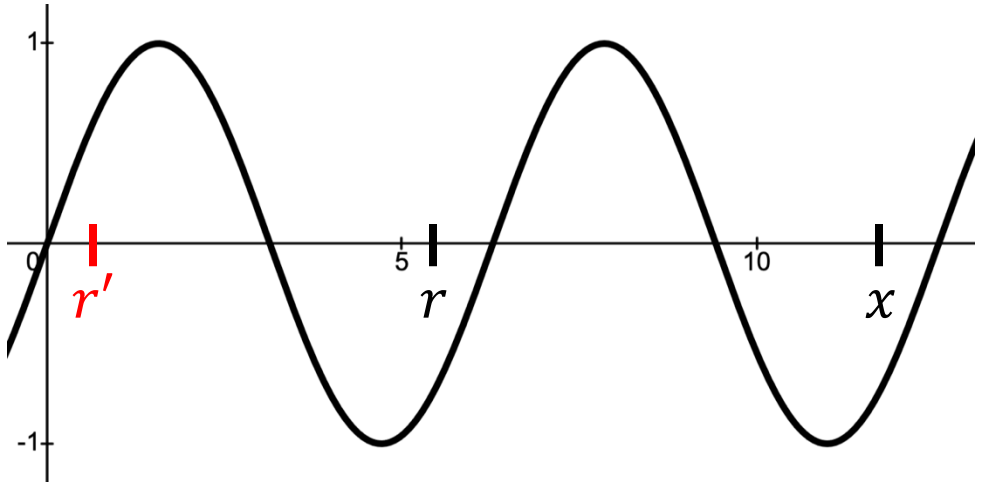


Example (1)

Input: $x \in \mathbb{F}$. Then, $x \in [-2^{1024}, 2^{1024}]$ in 64-bit double.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 1: Use symmetry.
  - $x = 2\pi k + r$ for some $k \in \mathbb{N}$ and $r \in [0, 2\pi)$.
  - Then, $\sin(x) = \sin(2\pi k + r) = \sin(r)$.
  - $r' \in \{|r|, |\pi - r|, |2\pi - r|\}$ s.t. $r' \in \left[0, \frac{\pi}{2}\right)$.
  - $\sin(r) = (-1)^s \sin(r')$ for some $s \in \{0,1\}$.

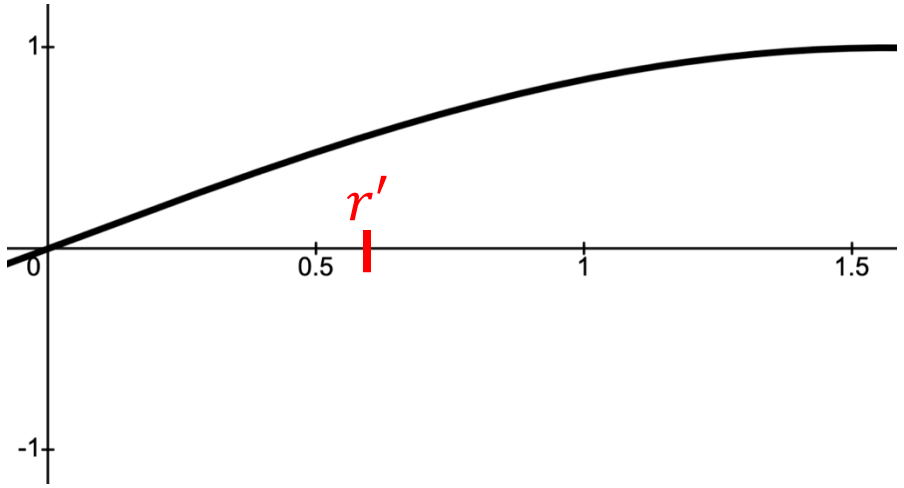


Example (1)

Input: $x \in \mathbb{F}$. Then, $x \in [-2^{1024}, 2^{1024}]$ in 64-bit double.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 1: Use symmetry.
  - $x = 2\pi k + r$ for some $k \in \mathbb{N}$ and $r \in [0, 2\pi)$.
  - Then, $\sin(x) = \sin(2\pi k + r) = \sin(r)$.
  - $r' \in \{|r|, |\pi - r|, |2\pi - r|\}$ s.t. $r' \in \left[0, \frac{\pi}{2}\right)$.
  - $\sin(r) = (-1)^s \sin(r')$ for some $s \in \{0,1\}$.

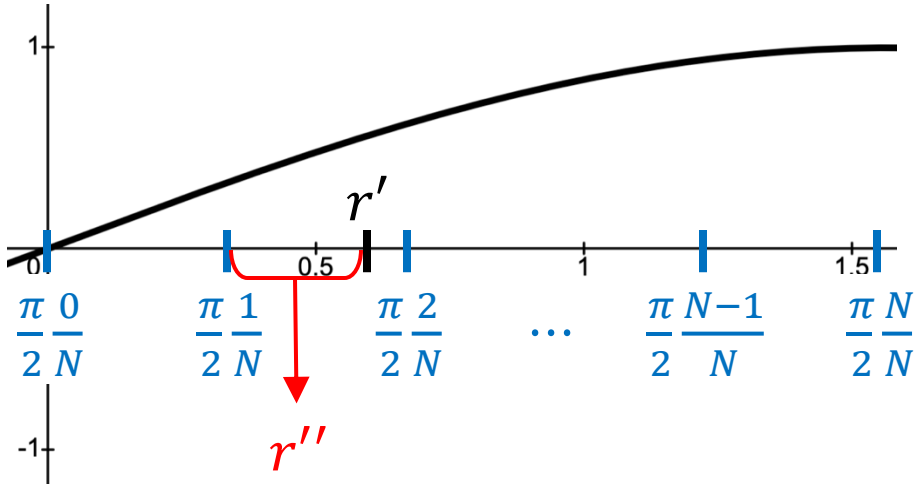


Example (2)

Input: $x \in \mathbb{F}$.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 2: Use tables.
  - $r' = \frac{\pi}{2}\frac{i}{N} + r''$ for some $i \in \mathbb{N}$ and $r'' \in \left[0, \frac{\pi}{2N}\right)$.

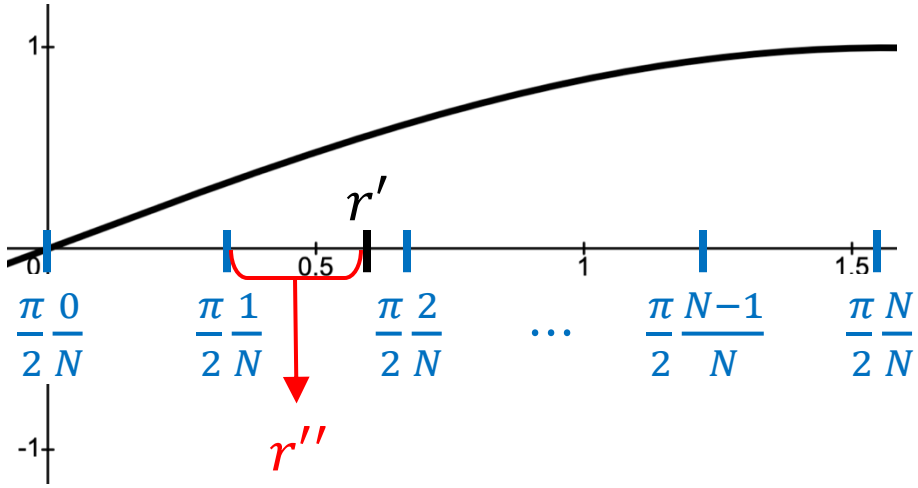


Example (2)

Input: $x \in \mathbb{F}$.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 2: Use tables.
  - $r' = \frac{\pi}{2}\frac{i}{N} + r''$ for some $i \in \mathbb{N}$ and $r'' \in \left[0, \frac{\pi}{2N}\right)$.
  - Then, $\sin(r') = \sin\left(\frac{\pi}{2N} i + r''\right)$
    $= \sin\left(\frac{\pi}{2N} i\right)\cos(r'') + \cos\left(\frac{\pi}{2N} i\right)\sin(r'')$.



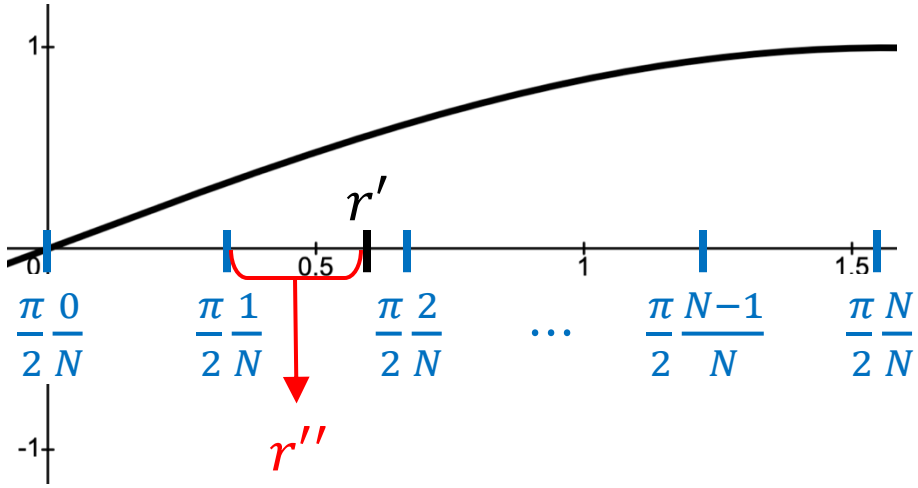
Example (2)

Input: $x \in \mathbb{F}$.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 2: Use tables.
  - $r' = \frac{\pi}{2}\frac{i}{N} + r''$ for some $i \in \mathbb{N}$ and $r'' \in \left[0, \frac{\pi}{2N}\right)$.
  - Then, $\sin(r') = \sin\left(\frac{\pi}{2N} i + r''\right)$

$$= \underbrace{\sin\left(\frac{\pi}{2N} i\right)}_{T_{sin}[i]} \cos(r'') + \underbrace{\cos\left(\frac{\pi}{2N} i\right)}_{T_{cos}[i]} \sin(r'').$$



for $0 \le i < N$

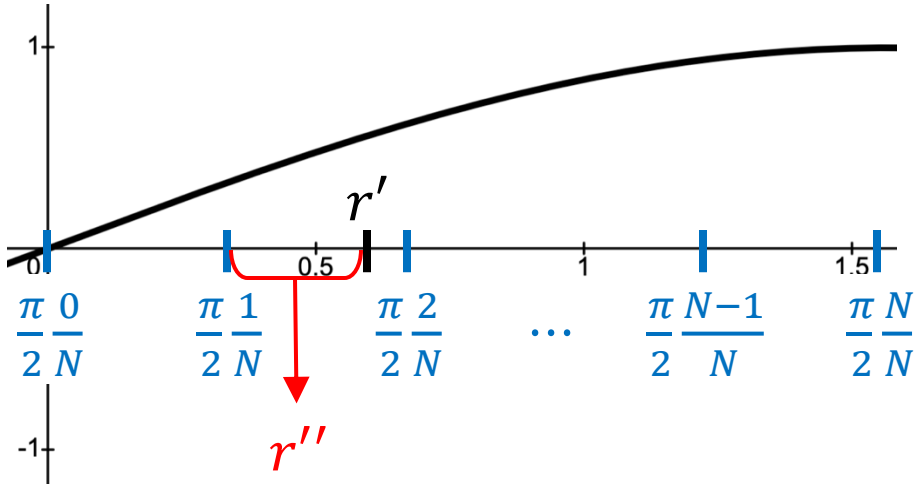
Example (2)

Input: $x \in \mathbb{F}$.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 2: Use tables.
  - $r' = \frac{\pi}{2}\frac{i}{N} + r''$ for some $i \in \mathbb{N}$ and $r'' \in \left[0, \frac{\pi}{2N}\right)$.
  - Then, $\sin(r') = \sin\left(\frac{\pi}{2N} i + r''\right)$

$$= \underbrace{\sin\left(\frac{\pi}{2N} i\right)}_{T_{sin}[i]} \cos(r'') + \underbrace{\cos\left(\frac{\pi}{2N} i\right)}_{T_{cos}[i]} \sin(r'').$$



for $0 \le i < N$

Example (3)

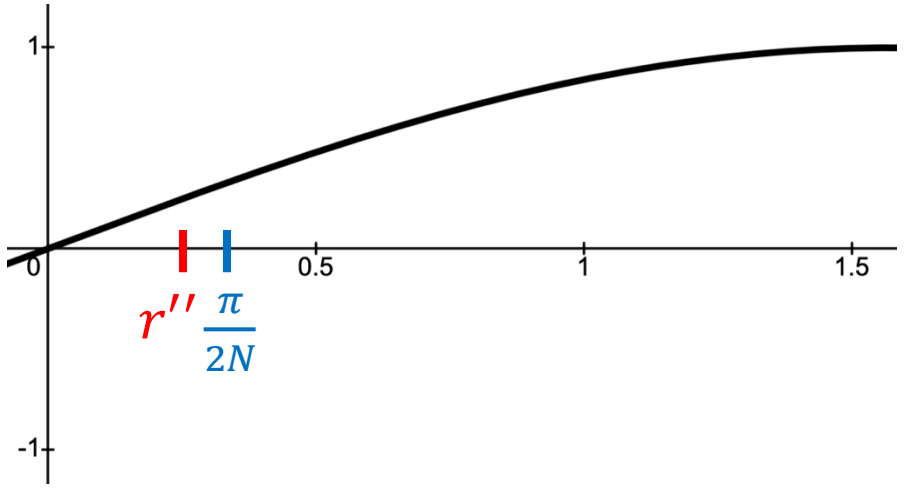
Input: $x \in \mathbb{F}$.

Output: $y \in \mathbb{F}$ such that $y \approx \sin(x)$.

- Idea 3: Use polynomial approximations.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots,$

    $\cos(r'') = 1 - \frac{(r'')^2}{2!} + \frac{(r'')^4}{4!} - \cdots,$

    where $r'' \in \left[0, \frac{\pi}{2N}\right)$.



Example

- Step 1: Argument Reduction.
  - $r := x - 2\pi k,$ where $k := \left\lfloor \frac{x}{2\pi} \right\rfloor.$
  - $r' \in \{|r|, |\pi - r|, |2\pi - r|\}.$
  - $r'' := r' - \frac{\pi}{2N} i,$ where $i := \left\lfloor \frac{r'}{\pi/2N} \right\rfloor.$
- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots.$
  - $\cos(r'') = 1 - \frac{(r'')^2}{2!} + \frac{(r'')^4}{4!} - \cdots.$
- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\sin\left(\frac{\pi}{2N} i\right) \cos(r'') + \cos\left(\frac{\pi}{2N} i\right) \sin(r'') \right).$

Example

- Step 1: Argument Reduction.
  - $r \coloneqq x - 2\pi k,$ where $k \coloneqq \left\lfloor \frac{x}{2\pi} \right\rceil.$
  - $r' \in \{|r|, |\pi - r|, |2\pi - r|\}.$
  - $r'' \coloneqq r' - \frac{\pi}{2N} i,$ where $i \coloneqq \left\lfloor \frac{r'}{\pi/2N} \right\rceil.$
- Step 2
  - si
  - $\cos(r'') = 1 - \frac{(r'')^2}{2!} + \frac{(r'')^4}{4!} - \cdots.$
- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\sin\left(\frac{\pi}{2N} i\right) \cos(r'') + \cos\left(\frac{\pi}{2N} i\right) \sin(r'') \right).$

How to implement these using **floating point**?

Actual Implementation

- Step 1: Argument Reduction.
  - $r := x - 2\pi k,$ where $k := \left\lfloor \frac{x}{2\pi} \right\rceil.$
  - $r' \in \{|r|, |\pi - r|, |2\pi - r|\}.$
  - $r'' := r' - \frac{\pi}{2N} i,$ where $i := \left\lfloor \frac{r'}{\pi/2N} \right\rceil.$
- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots.$
  - $\cos(r'') = 1 - \frac{(r'')^2}{2!} + \frac{(r'')^4}{4!} - \cdots.$
- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\sin\left(\frac{\pi}{2N} i\right) \cos(r'') + \cos\left(\frac{\pi}{2N} i\right) \si$

```
2033 double
2034 cr_sin (double x)
2035 {
2036   b64u64_u t = {.f = x};
2037   int e = (t.u >> 52) & 0x7ff;
2038
2039   if (__builtin_expect (e == 0x7ff, 0)) /* NaN, +Inf and -Inf. */
2040     {
2041       t.u = ~0ull;
2042       return t.f;
2043     }
2044
2045   /* now x is a regular number */
2046
2047   /* For |x| <= 0x1.7137449123ef6p-26, sin(x) rounds to x (to nearest):
2048      we can assume x >= 0 without loss of generality since sin(-x) = -sin(x),
2049      we have x - x^3/6 < sin(x) < x for say 0 < x <= 1 thus
2050      |sin(x) - x| < x^3/6.
2051      Write x = c*2^e with 1/2 <= c < 1.
2052      Then ulp(x)/2 = 2^(e-54), and x^3/6 = c^3/6*2^(3e), thus
2053      x^3/6 < ulp(x)/2 rewrites as c^3/6*2^(3e) < 2^(e-54),
2054      or c^3*2^(2e+53) < 3 (1).
2055      For e <= -26, since c^3 < 1, we have c^3*2^(2e+53) < 2 < 3.
2056      For e=-25, (1) rewrites 8*c^3 < 3 which yields c <= 0x1.7137449123ef6p-1.
2057   */
2058   uint64_t ux = t.u & 0x7fffffffffffffff;
2059   // 0x3e57137449123ef6 = 0x1.7137449123ef6p-26
2060   if (ux <= 0x3e57137449123ef6)
2061     // Taylor expansion of sin(x) is x - x^3/6 around zero
2062     // for x=-0, fma (x, -0x1p-54, x) returns +0
2063     return (x == 0) ? x : __builtin_fma (x, -0x1p-54, x);
2064
2065   double h, l, err;
2066   err = sin_fast (&h, &l, x);
2067   double left  = h + (l - err), right = h + (l + err);
2068   /* With SC[] from ./buildSC 15 we get 1100 failures out of 50000000
2069      random tests, i.e., about 0.002%. */
2070   if (__builtin_expect (left == right, 1))
2071     return left;
2072
2073   return sin_accurate (x);
2074 }
```

Actual Implementation (1)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k,$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rceil.$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}.$
  - $\widehat{r''} := \widehat{r'} - \frac{\hat{\pi}}{2N} i,$ where $i := \left\lfloor \frac{\hat{r'}}{\hat{\pi}/2N} \right\rceil.$

- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots.$
  - $\cos(r'') = 1 - \frac{(r'')^2}{2!} + \frac{(r'')^4}{4!} - \cdots.$

- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\sin\left(\frac{\pi}{2N} i\right) \cos(r'') + \cos\left(\frac{\pi}{2N} i\right) \sin(r'') \right).$

- $\pi = 3.141592653589793 \ldots \notin \mathbb{F}$

  $\hat{\pi} = 3.14159265358931 \ldots \in \mathbb{F}$

Actual Implementation (1)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k,$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rceil.$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}.$
  - $\widehat{r''} := \widehat{r'} - \frac{\hat{\pi}}{2N} i,$ where $i := \left\lfloor \frac{\widehat{r'}}{\hat{\pi}/2N} \right\rceil.$
- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots.$
  - $\cos(r'') = 1 - \frac{(r'')^2}{2!} + \frac{(r'')^4}{4!} - \cdots.$
- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\sin\left(\frac{\pi}{2N} i\right) \cos(r'') + \cos\left(\frac{\pi}{2N} i\right) \sin(r'') \right).$

- $\pi = 3.141592653589793 \ldots \notin \mathbb{F}$

  $\hat{\pi} = 3.14159265358931 \ldots \in \mathbb{F}$

- For $x = 10^{18} \in \mathbb{F},$

  $r = 4.83 \ldots \notin \mathbb{F}.$

  $\hat{r} = 128 \quad \in \mathbb{F}.$

Actual Implementation (1)

```
1755  static double
1756  sin_fast (double *h, double *l, double x)
1757  {
1758    int neg = x < 0, is_sin = 1;
1759    double absx = neg ? -x : x;
1760
1761    /* now x > 0x1.7137449123ef6p-26 */
1762    double err1;
1763    int i = reduce_fast (h, l, absx, &err1);
1764    /* err1 is an absolute bound for | 1/2^11 + h + l - frac(x/(2pi)) |:
1765       | i/2^11 + h + l - frac(x/(2pi)) | <= err1 */
```

```
1622  /* Assuming 0x1.7137449123ef6p-26 < x < +Inf,
1623     return i and set h,l such that i/2^11+h+l approximates frac(x/(2pi)).
1624     If x <= 0x1.921fb54442d18p+2:
1625     | i/2^11 + h + l - frac(x/(2pi)) | < 2^-104.116 * |i/2^11 + h + l|
1626     with |h| < 2^-11 and |l| < 2^-52.36.
1627
1628     Otherwise only the absolute error is bounded:
1629     | i/2^11 + h + l - frac(x/(2pi)) | < 2^-75.998
1630     with 0 <= h < 2^-11 and |l| < 2^-53.
1631
1632     In both cases we have |l| < 2^-51.64*|i/2^11 + h|.
1633
1634     Put in err1 a bound for the absolute error:
1635     | i/2^11 + h + l - frac(x/(2pi)) |.
1636  */
1637  static int
1638  reduce_fast (double *h, double *l, double x, double *err1)
1639  {
```

```
1708          int i = (e - 1138 + 63) / 64; // i = ceil((e-1138)/64), 0 <= i <= 15
1709          /* m*T[i] contributes to f = 1139 + 64*i - e bits to frac(x/(2pi))
1710             with 1 <= f <= 64
1711             m*T[i+1] contributes a multiple of 2^(-f-64),
1712                      and at most to 2^(53-f)
1713             m*T[i+2] contributes a multiple of 2^(-f-128),
1714                      and at most to 2^(-11-f)
1715             m*T[i+3] contributes a multiple of 2^(-f-192),
1716                      and at most to 2^(-75-f) <= 2^-76
1717          */
1718          u = (u128) m * (u128) T[i+2];
1719          c[0] = u;
1720          c[1] = u >> 64;
1721          u = (u128) m * (u128) T[i+1];
1722          c[1] += u;
1723          c[2] = (u >> 64) + (c[1] < (uint64_t) u);
1724          u = (u128) m * (u128) T[i];
1725          c[2] += u;
1726          e = 1139 + (i<<6) - e; // 1 <= e <= 64
1727          // e is the number of low bits of C[2] contributing to frac(x/(2pi))
```

```
423  /* This table approximates 1/(2pi) downwards with precision 1280:
424     1/(2*pi) ~ T[0]/2^64 + T[1]/2^128 + ... + T[i]/2^((i+1)*64) + ...
425     Computed with computeT() from sin.sage. */
426  static const uint64_t T[20] = {
427    0x28be60db9391054a, // i=0
428    0x7f09d5f47d4d3770,
429    0x36d8a5664f10e410,
430    0x7f9458eaf7aef158,
431    0x6dc91b8e909374b8,
432    0x1924bba82746487, // i=5
433    0x3f877ac72c4a69cf,
434    0xba208d7d4baed121,
435    0x3a671c09ad17df90,
436    0x4e64758e60d4ce7d,
437    0x272117e2ef7e4a0e, // i=10
```

- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots$.
  - $\cos(r'') = 1$
- Step 3: Table L
  - $\sin(x) = (-$

Actual Implementation (1)

```
1755  static double
1756  sin_fast (double *h, double *l, double x)
1757  {
1758    int neg = x < 0, is_sin = 1;
1759    double absx = neg ? -x : x;
1760
1761    /* now x > 0x1.7137449123ef6p-26 */
1762    double err1;
1763    int i = reduce_fast (h, l, absx, &err1);
1764    /* err1 is an absolute bound for | i/2^11 + h + l - frac(x/(2pi)) |:
1765       | i/2^11 + h + l - frac(x/(2pi)) | <= err1 */
```

```
1622  /* Assuming 0x1.7137449123ef6p-26 < x < +Inf,
1623     return i and set h,l such that i/2^11+h+l approximates frac(x/(2pi)).
1624     If x <= 0x1.921fb54442d18p+2:
1625     | i/2^11 + h + l - frac(x/(2pi)) | < 2^-104.116 * |i/2^11 + h + l|
1626     with |h| < 2^-11 and |l| < 2^-52.36.
1627
1628     Otherwise only the absolute error is bounded:
1629     | i/2^11 + h + l - frac(x/(2pi)) | < 2^-75.998
1630     with 0 <= h < 2^-11 and |l| < 2^-53.
1631
1632     In both cases we have |l| < 2^-51.64*|i/2^11 + h|.
1633
1634     Put in err1 a bound for the absolute error:
1635     | i/2^11 + h + l - frac(x/(2pi)) |.
1636  */
1637  static in
1638  reduce_fa
1639  {
```

Payne-Hanek algorithm

- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots$.
  - $\cos(r'') = 1$
- Step 3: Table L
  - $\sin(x) = (-$

```
423  /* This table approximates 1/(2pi) downwards with precision 1280:
424     1/(2*pi) ~ T[0]/2^64 + T[1]/2^128 + ... + T[i]/2^((i+1)*64) + ...
425     Computed with computeT() from sin.sage. */
426  static const uint64_t T[20] = {
427    0x28be60db9391054a, // i=0
428     0x7f09d5f47d4d3770,
429     0x36d8a5664f10e410,
430     0x7f9458eaf7aef158,
431     0x6dc91b8e909374b8,
432     0x1924bba82746487, // i=5
433     0x3f877ac72c4a69cf,
434     0xba208d7d4baed121,
435     0x3a671c09ad17df90,
436     0x4e64758e60d4ce7d,
437     0x272117e2ef7e4a0e, // i=10
```

```
1708
1709
1710
1711
1712
```

```
int i = (e - 1138 + 63) / 64; // i = ceil((e-1138)/64), 0 <= i <= 15
/* m*T[i] contributes to f = 1139 + 64*i - e bits to frac(x/(2pi))
   with 1 <= f <= 64
   m*T[i+1] contributes a multiple of 2^(-f-64),
            and at most to 2^(53-f)
   m*T[i+2] contributes a multiple of 2^(-f-128),
            and at most to 2^(-11-f)
   m*T[i+3] contributes a multiple of 2^(-f-192),
            and at most to 2^(-75-f) <= 2^-76
*/
u = (u128) m * (u128) T[i+2];
c[0] = u;
c[1] = u >> 64;
u = (u128) m * (u128) T[i+1];
c[1] += u;
c[2] = (u >> 64) + (c[1] < (uint64_t) u);
u = (u128) m * (u128) T[i];
c[2] += u;
e = 1139 + (i<<6) - e; // 1 <= e <= 64
// e is the number of low bits of C[2] contributing to frac(x/(2pi))
```

Actual Implementation (1)

$$|r(x) - \hat{r}(x)|$$

```
1755  static double
1756  sin_fast (double *h, double *l, double x)
1757  {
1758    int neg = x < 0, is_sin = 1;
1759    double absx = neg ? -x : x;
1760
1761    /* now x > 0x1.7137449123ef6p-26 */
1762    double err1;
1763    int i = reduce_fast (h, l, absx, &err1);
1764    /* err1 is an absolute bound for | i/2^11 + h + l - frac(x/(2pi)) |:
1765       | i/2^11 + h + l - frac(x/(2pi)) | <= err1 */
```

```
1622  /* Assuming 0x1.7137449123ef6p-26 < x < +Inf,
1623     return i and set h,l such that i/2^11+h+l approximates frac(x/(2pi)).
1624     If x <= 0x1.921fb54442d18p+2:
1625     | i/2^11 + h + l - frac(x/(2pi)) | < 2^-104.116 * |i/2^11 + h + l|
1626     with |h| < 2^-11 and |l| < 2^-52.36.
1627
1628     Otherwise only the absolute error is bounded:
1629     | i/2^11 + h + l - frac(x/(2pi)) | < 2^-75.998
1630     with 0 <= h < 2^-11 and |l| < 2^-53.
1631
1632     In both cases we have |l| < 2^-51.64*|i/2^11 + h|.
1633
1634     Put in err1 a bound for the absolute error:
1635     | i/2^11 + h + l - frac(x/(2pi)) |.
1636  */
1637  static int
1638  reduce_fast (double *h, double *l, double x, double *err1)
1639  {
```

```
1708        int i = (e - 1138 + 63) / 64; // i = ceil((e-1138)/64), 0 <= i <= 15
1709        /* m*T[i] contributes to f = 1139 + 64*i - e bits to frac(x/(2pi))
1710           with 1 <= f <= 64
1711           m*T[i+1] contributes a multiple of 2^(-f-64),
1712                    and at most to 2^(53-f)
1713           m*T[i+2] contributes a multiple of 2^(-f-128),
1714                    and at most to 2^(-11-f)
1715           m*T[i+3] contributes a multiple of 2^(-f-192),
1716                    and at most to 2^(-75-f) <= 2^-76
1717        */
1718        u = (u128) m * (u128) T[i+2];
1719        c[0] = u;
1720        c[1] = u >> 64;
1721        u = (u128) m * (u128) T[i+1];
1722        c[1] += u;
1723        c[2] = (u >> 64) + (c[1] < (uint64_t) u);
1724        u = (u128) m * (u128) T[i];
1725        c[2] += u;
1726        e = 1139 + (i<<6) - e; // 1 <= e <= 64
1727        // e is the number of low bits of C[2] contributing to frac(x/(2pi))
```

```
423  /* This table approximates 1/(2pi) downwards with precision 1280:
424     1/(2*pi) ~ T[0]/2^64 + T[1]/2^128 + ... + T[i]/2^((i+1)*64) + ...
425     Computed with computeT() from sin.sage. */
426  static const uint64_t T[20] = {
427    0x28be60db9391054a, // i=0
428     0x7f09d5f47d4d3770,
429     0x36d8a5664f10e410,
430     0x7f9458eaf7aef158,
431     0x6dc91b8e909374b8,
432     0x1924bba82746487, // i=5
433     0x3f877ac72c4a69cf,
434     0xba208d7d4baed121,
435     0x3a671c09ad17df90,
436     0x4e64758e60d4ce7d,
437     0x272117e2ef7e4a0e, // i=10
```

- Step 2: Polynomial Approximation.
  - $\sin(r'') = r'' - \frac{(r'')^3}{3!} + \frac{(r'')^5}{5!} - \cdots$.
  - $\cos(r'') = 1$
- Step 3: Table L
  - $\sin(x) = (-$

Actual Implementation (2)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k, \quad$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rfloor$.
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}$.
  - $\widehat{r''} := \widehat{r'} - \frac{\hat{\pi}}{2N} i, \quad$ where $i := \left\lfloor \frac{\hat{r'}}{\hat{\pi}/2N} \right\rfloor$.

- Step 2: Polynomial Approximation.
  - $\sin(\widehat{r''}) = \widehat{r''} - \left(\widehat{\frac{1}{3!}}\right)(\widehat{r''})^3 + \cdots + \left(\widehat{\frac{(-1)^{2D+1}}{(2D+1)!}}\right)(\widehat{r''})^{2D+1}$.
  - $\cos(\widehat{r''}) = 1 - \left(\widehat{\frac{1}{2!}}\right)(\widehat{r''})^2 + \cdots + \left(\widehat{\frac{(-1)^{2D}}{(2D)!}}\right)(\widehat{r''})^{2D}$.

- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\sin\left(\frac{i\pi}{2N}\right)\cos(r'') + \cos\left(\frac{i\pi}{2N}\right)\sin(r'')\right)$.

Actual Implementation (2)

```
1803  /* Now 0 <= i < 256 and 0 <= h+l < 2^-11
1804     with | i/2^11 + h + l - frac(x/(2pi)) | cmod 1/4 < err1
1805     If is_sin=1, sin |x| = sin2pi (R + err1);
1806     if is_sin=0, sin |x| = cos2pi (R + err1).
1807     In both cases R = i/2^11 + h + l, 0 <= R < 1/4.
1808  */
1809  double sh, sl, ch, cl;
1810  /* since the SC[] table evaluates at i/2^11 + SC[i][0] and not at i/2^11,
1811     we must subtract SC[i][0] from h+l */
1812  /* Here h = k*2^-55 with 0 <= k < 2^44, and SC[i][0] is an integer
1813     multiple of 2^-62, with |SC[i][0]| < 2^-24, thus SC[i][0] = m*2^-62
1814     with |m| < 2^38. It follows h-SC[i][0] = (k*2^7 + m)*2^-62 with
1815     2^51 - 2^38 < k*2^7 + m < 2^51 + 2^38, thus h-SC[i][0] is exact.
1816     Now |h| < 2^-11 + 2^-24. */
1817  *h -= SC[i][0];
1818  // now -2^-24 < h < 2^-11+2^-24
1819  // from reduce_fast() we have |l| < 2^-52.36
1820  double uh, ul;
1821  a_mul (&uh, &ul, *h, *h);
1822  ul = __builtin_fma (*h + *h, *l, ul);
1823  // uh+ul approximates (h+l)^2
1824  evalPSfast (&sh, &sl, *h, *l, uh, ul);
1825  /* the absolute error of evalPSfast() is less than 2^-77.09 from
1826     routine evalPSfast() in sin.sage:
1827     | sh + sh - sin2pi(h+l) | < 2^-77.09 */
1828  evalPCfast (&ch, &cl, uh, ul);
1829  /* the relative error of evalPCfast() is less than 2^-69.96 from
1830     routine evalPCfast(rel=true) in sin.sage:
1831     | ch + cl - cos2pi(h+l) | < 2^-69.96 * |ch + cl| */
```

```
1320  /* Put in h+l an approximation of sin2pi(xh+xl),
1321     for 2^-24 <= xh+xl < 2^-11 + 2^-24,
1322     and |xl| < 2^-52.36, with absolute error < 2^-77.09
1323     (see evalPSfast() in sin.sage).
1324     Assume uh + ul approximates (xh+xl)^2. */
1325  static void
1326  evalPSfast (double *h, double *l, double xh, double xl, double uh, double ul)
1327  {
1328    double t;
1329    *h = PSfast[4]; // degree 7
1330    *h = __builtin_fma (*h, uh, PSfast[3]); // degree 5
1331    *h = __builtin_fma (*h, uh, PSfast[2]); // degree 3
1332    s_mul (h, l, *h, uh, ul);
1333    fast_two_sum (h, &t, PSfast[0], *h);
1334    *l += PSfast[1] + t;
1335    // multiply by xh+xl
1336    d_mul (h, l, *h, *l, xh, xl);
1337  }
```

```
975  /* The following is a degree-7 polynomial with odd coefficients
976     approximating sin2pi(x) for -2^-24 < x < 2^-11+2^-24
977     with relative error 2^-77.306.
978     Generated with sin_fast.sollya. */
979  static const  double PSfast[] = {
980    0x1.921fb54442d18p+2, 0x1.1a62645446203p-52, // degree 1 (h+l)
981    -0x1.4abbce625be53p5,                        // degree 3
982    0x1.466bc678d8d63p6,                         // degree 5
983    -0x1.331554ca19669p6,                        // degree 7
984  };
```

Actual Implementation (2)

$|p(x) - \hat{p}(x)|$

```
1803  /* Now 0 <= i < 256 and 0 <= h+l < 2^-11
1804     with | i/2^11 + h + l - frac(x/(2pi)) | cmod 1/4 < err1
1805     If is_sin=1, sin |x| = sin2pi (R + err1);
1806     if is_sin=0, sin |x| = cos2pi (R + err1).
1807     In both cases R = i/2^11 + h + l, 0 <= R < 1/4.
1808  */
1809  double sh, sl, ch, cl;
1810  /* since the SC[] table evaluates at i/2^11 + SC[i][0] and not at i/2^11,
1811     we must subtract SC[i][0] from h+l */
1812  /* Here h = k*2^-55 with 0 <= k < 2^44, and SC[i][0] is an integer
1813     multiple of 2^-62, with |SC[i][0]| < 2^-24, thus SC[i][0] = m*2^-62
1814     with |m| < 2^38. It follows h-SC[i][0] = (k*2^7 + m)*2^-62 with
1815     2^51 - 2^38 < k*2^7 + m < 2^51 + 2^38, thus h-SC[i][0] is exact.
1816     Now |h| < 2^-11 + 2^-24. */
1817  *h -= SC[i][0];
1818  // now -2^-24 < h < 2^-11+2^-24
1819  // from reduce_fast() we have |l| < 2^-52.36
1820  double uh, ul;
1821  a_mul (&uh, &ul, *h, *h);
1822  ul = __builtin_fma (*h + *h, *l, ul);
1823  // uh+ul approximates (h+l)^2
1824  evalPSfast (&sh, &sl, *h, *l, uh, ul);
1825  /* the absolute error of evalPSfast() is less than 2^-77.09 from
1826     routine evalPSfast() in sin.sage:
1827     | sh + sh - sin2pi(h+l) | < 2^-77.09 */
1828  evalPCfast (&ch, &cl, uh, ul);
1829  /* the relative error of evalPCfast() is less than 2^-69.96 from
1830     routine evalPCfast(rel=true) in sin.sage:
1831     | ch + cl - cos2pi(h+l) | < 2^-69.96 * |ch + cl| */
```

```
1320  /* Put in h+l an approximation of sin2pi(xh+xl),
1321     for 2^-24 <= xh+xl < 2^-11 + 2^-24,
1322     and |xl| < 2^-52.36, with absolute error < 2^-77.09
1323     (see evalPSfast() in sin.sage).
1324     Assume uh + ul approximates (xh+xl)^2. */
1325  static void
1326  evalPSfast (double *h, double *l, double xh, double xl, double uh, double ul)
1327  {
1328    double t;
1329    *h = PSfast[4]; // degree 7
1330    *h = __builtin_fma (*h, uh, PSfast[3]); // degree 5
1331    *h = __builtin_fma (*h, uh, PSfast[2]); // degree 3
1332    s_mul (h, l, *h, uh, ul);
1333    fast_two_sum (h, &t, PSfast[0], *h);
1334    *l += PSfast[1] + t;
1335    // multiply by xh+xl
1336    d_mul (h, l, *h, *l, xh, xl);
1337  }
```

$|\sin(x) - p(x)|$

```
975  /* The following is a degree-7 polynomial with odd coefficients
976     approximating sin2pi(x) for -2^-24 < x < 2^-11+2^-24
977     with relative error 2^-77.306.
978     Generated with sin_fast.sollya. */
979  static const double PSfast[5] = {
980    0x1.921fb54442d18p2,                    // degree 1
981    -0x1.4abbce625be53p2,                   // degree 3
982    0x1.466bc6775ab16p1,                    // degree 5
983    -0x1.32d2cce62dc33p0,                   // degree 7
984  };
```

Actual Implementation (3)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k,$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rfloor.$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}.$
  - $\widehat{r''} := \widehat{r'} - \frac{\hat{\pi}}{2N} i,$ where $i := \left\lfloor \frac{\hat{r'}}{\hat{\pi}/2N} \right\rfloor.$
- Step 2: Polynomial Approximation.
  - $\sin(\widehat{r''}) = \widehat{r''} - \left(\widehat{\frac{1}{3!}}\right)(\widehat{r''})^3 + \cdots + \left(\widehat{\frac{(-1)^{2D+1}}{(2D+1)!}}\right)(\widehat{r''})^{2D+1}.$
  - $\cos(\widehat{r''}) = 1 - \left(\widehat{\frac{1}{2!}}\right)(\widehat{r''})^2 + \cdots + \left(\widehat{\frac{(-1)^{2D}}{(2D)!}}\right)(\widehat{r''})^{2D}.$
- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\widehat{\sin\left(\frac{\pi}{2N} i\right)} \cos(r'') + \widehat{\cos\left(\frac{\pi}{2N} i\right)} \sin(r'') \right).$

Actual Implementation (3)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k, \quad$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rceil$.
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}$.

```
1832  double err;
1833  static const double sgn[2] = {1.0, -1.0};
1834  if (is_sin)
1835    {
1836      s_mul (&sh, &sl, sgn[neg] * SC[i][2], sh, sl);
1837      s_mul (&ch, &cl, sgn[neg] * SC[i][1], ch, cl);
1838      fast_two_sum (h, l, ch, sh);
1839      *l += sl + cl;
1840      /* absolute error bounded by 2^-68.588
1841         from global_error(is_sin=true,rel=false) in sin.sage:
1842         | h + l - sin2pi (R) | < 2^-68.588
1843         thus:
1844         | h + l - sin |x| | < 2^-68.588 + | sin2pi (R) - sin |x| |
1845                            < 2^-68.588 + err1 */
1846      err = 0x1.55p-69; // 2^-66.588 < 0x1.55p-69
1847    }
```

```
1021  /* Table generated with ./buildSC 15 using accompanying buildSC.c.
1022     For each i, 0 <= i < 256, xi=i/2^11+SC[i][0], with
1023     SC[i][1] and SC[i][2] approximating sin2pi(xi) and cos2pi(xi)
1024     respectively, both with 53+15 bits of accuracy. */
1025  static const double SC[256][3] = {
1026    {0x0p+0, 0x0p+0, 0x1p+0}, /* 0 */
1027    {-0x1.c0f6cp-35, 0x1.921f892b900fep-9, 0x1.ffff621623fap-1}, /* 1 */
1028    {-0x1.9c7935ep-35, 0x1.921f0ea27ce01p-8, 0x1.fffd8858eca2ep-1}, /* 2 */
1029    {-0x1.d14d1acp-34, 0x1.2d96af779b0bbp-7, 0x1.fffa72c986392p-1}, /* 3 */
1030    {-0x1.dba8f6a8p-33, 0x1.921d1ce2d0a1cp-7, 0x1.fff62169dddaap-1}, /* 4 */
1031    {0x1.a6b7cdfp-32, 0x1.f6a29bdb7377p-7, 0x1.fff0943c02419p-1}, /* 5 */
1032    {0x1.b49618dp-33, 0x1.2d936d1506f3dp-6, 0x1.ffe9cb44829cp-1}, /* 6 */
1033    {-0x1.398d6fcp-35, 0x1.5fd4d1e21de6dp-6, 0x1.ffe1c687174b1p-1}, /* 7 */
1034    {-0x1.e9e9a8c8p-31, 0x1.9215597791e0ap-6, 0x1.ffd886097afcfp-1}, /* 8 */
1035    {-0x1.34e844cp-32, 0x1.c454f2e9480c7p-6, 0x1.ffce09ce95933p-1}, /* 9 */
1036    {-0x1.989a8a4p-32, 0x1.f693709b94f92p-6, 0x1.ffc251dfbac0cp-1}, /* 10 */
1037    {0x1.04a9b99p-30, 0x1.146860e69a571p-5, 0x1.ffb55e40a5c43p-1}, /* 11 */
1038    {-0x1.56947cp-36, 0x1.2d865748774adp-5, 0x1.ffa72efff95d1p-1}, /* 12 */
1039    {-0x1.c348768p-35, 0x1.46a396d34121ap-5, 0x1.ff97c420a8451p-1}, /* 13 */
1040    {0x1.9e80552p-32, 0x1.5fc00e6e4c65cp-5, 0x1.ff871dacd8761p-1}, /* 14 */
1041    {0x1.3f11d74p-34, 0x1.78dbaa97099ebp-5, 0x1.ff753bb18af95p-1}, /* 15 */
1042    {0x1.c039af4p-33, 0x1.91f65fc0abc0ap-5, 0x1.ff621e370ca7ap-1}, /* 16 */
1043    {0x1.53e1f8p-35, 0x1.ab101bf74ac2ep-5, 0x1.ff4dc54b00181p-1}, /* 17 */
1044    {0x1.114a649p-29, 0x1.c428d7de920e9p-5, 0x1.ff3830f2e9043p-1}, /* 18 */
1045    {0x1.adf0ef4p-31, 0x1.dd40723a3cdfbp-5, 0x1.ff21614b9d9adp-1}, /* 19 */
```

```
1310  static inline void
1311  fast_two_sum(double *hi, double *lo, double a, double b)
1312  {
1313    double e;
1314
1315    *hi = a + b;
1316    e = *hi - a; /* exact */
1317    *lo = b - e; /* exact */
1318  }
```

- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\overbrace{\sin\left(\frac{\pi}{2N} i\right)}\cos(r'') + \overbrace{\cos\left(\frac{\pi}{2N} i\right)}\right)$

Actual Implementation (3)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k,$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rceil.$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}.$

```
1832  double err;
1833  static const double sgn[2] = {1.0, -1.0};
1834  if (is_sin)
1835    {
1836      s_mul (&sh, &sl, sgn[neg] * SC[i][2], sh, sl);
1837      s_mul (&ch, &cl, sgn[neg] * SC[i][1], ch, cl);
1838      fast_two_sum (h, l, ch, sh);
1839      *l += sl + cl;
1840      /* absolute error bounded by 2^-68.588
1841         from global_error(is_sin=true,rel=false) in sin.sage:
1842         | h + l - sin2pi (R) | < 2^-68.588
1843         thus:
1844         | h + l - sin |x| | < 2^-68.588 + | sin2pi (R) - sin |x| |
1845                             < 2^-68.588 + err1 */
1846      err = 0x1.55p-69; // 2^-66.588 < 0x1.55p-69
1847    }
```

```
1021  /* Table generated with ./buildSC 15 using accompanying buildSC.c.
1022     For each i, 0 <= i < 256, xi=i/2^11+SC[i][0], with
1023     SC[i][1] and SC[i][2] approximating sin2pi(xi) and cos2pi(xi)
1024     respectively, both with 53+15 bits of accuracy. */
1025  static const double SC[256][3] = {
1026     {0x0p+0, 0x0p+0, 0x1p+0}, /* 0 */
1027     {-0x1.c0f6cp-35, 0x1.921f892b900fep-9, 0x1.ffff621623fap-1}, /* 1 */
1028     {-0x1.9c7935ep-35, 0x1.921f0ea27ce01p-8, 0x1.fffd8858eca2ep-1}, /* 2 */
1029     {-0x1.d14d1acp-34, 0x1.2d96af779b0bbp-7, 0x1.fffa72c986392p-1}, /* 3 */
1030     {-0x1.dba8f6a8p-33, 0x1.921d1ce2d0a1cp-7, 0x1.fff62169dddaap-1}, /* 4 */
1031     {0x1.a6b7cdfp-32, 0x1.f6a29bdb7377p-7, 0x1.fff0943c02419p-1}, /* 5 */
1032     {0x1.b49618dp-33, 0x1.2d936d1506f3dp-6, 0x1.ffe9cb44829cp-1}, /* 6 */
1033     {-0x1.398d6fcp-35, 0x1.5fd4d1e21de6dp-6, 0x1.ffe1c687174b1p-1}, /* 7 */
1034     {-0x1.e9e9a8c8p-31, 0x1.9215597791e0ap-6, 0x1.ffd886097afcfp-1}, /* 8 */
1035     {-0x1.34e844cp-32, 0x1.c454f2e9480c7p-6, 0x1.ffce09ce95933p-1}, /* 9 */
1036     {-0x1.989a8a4p-32, 0x1.f693709b94f92p-6, 0x1.ffc251dfbac0cp-1}, /* 10 */
1037     {0x1.04a9b99p-30, 0x1.146860e69a571p-5, 0x1.ffb55e40a5c43p-1}, /* 11 */
1038     {-0x1.56947cp-36, 0x1.2d865748774adp-5, 0x1.ffa72efff95d1p-1}, /* 12 */
1039     {-0x1.c348768p-35, 0x1.46a396d34121ap-5, 0x1.ff97c420a8451p-1}, /* 13 */
1040     {0x1.9e80552p-32, 0x1.5fc00e6e4c65cp-5, 0x1.ff871dacd8761p-1}, /* 14 */
1041     {0x1.3f11d74p-34, 0x1.78dbaa97099ebp-5, 0x1.ff753bb18af95p-1}, /* 15 */
1042     {0x1.c039af4p-33, 0x1.91f65fc0abc0ap-5, 0x1.ff621e370ca7ap-1}, /* 16 */
1043     {0x1.53e1f8p-35, 0x1.ab101bf74ac2ep-5, 0x1.ff4dc54b00181p-1}, /* 17 */
1044     {0x1.114a649p-29, 0x1.c428d7de920e9p-5, 0x1.ff3830f2e9043p-1}, /* 18 */
1045     {0x1.adf0ef4p-31, 0x1.dd40723a3cdfbp-5, 0x1.ff21614b9d9adp-1}, /* 19 */
```

```
1310  static inline void
1311  fast_two_sum(double *hi, double *lo, double a, double b)
1312  {
1313    double e;
1314
1315    *hi = a + b;
1316    e = *hi - a; /* exact */
1317    *lo = b - e; /* exact */
1318  }
```

- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\overbrace{\sin\left(\frac{\pi}{2N} i\right)}\cos(r'') + \overbrace{\cos\left(\frac{\pi}{2N} i\right)}\right)$

Actual Implementation (3)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k,$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rceil.$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}.$

```
1832  double err;
1833  static const double sgn[2] = {1.0, -1.0};
1834  if (is_sin)
1835    {
1836      s_mul (&sh, &sl, sgn[neg] * SC[i][2], sh, sl);
1837      s_mul (&ch, &cl, sgn[neg] * SC[i][1], ch, cl);
1838      fast_two_sum (h, l, ch, sh);
1839      *l += sl + cl;
1840      /* absolute error bounded by 2^-68.588
1841         from global_error(is_sin=true,rel=false) in sin.sage:
1842         | h + l - sin2pi (R) | < 2^-68.588
1843         thus:
1844         | h + l - sin |x| | < 2^-68.588 + | sin2pi (R) - sin |x| |
1845                            < 2^-68.588 + err1 */
1846      err = 0x1.55p-69; // 2^-66.588 < 0x1.55p-69
1847    }
```

```
1021  /* Table generated with ./buildSC 15 using accompanying buildSC.c.
1022     For each i, 0 <= i < 256, xi=i/2^11+SC[i][0], with
1023     SC[i][1] and SC[i][2] approximating sin2pi(xi) and cos2pi(xi)
1024     respectively, both with 53+15 bits of accuracy. */
1025  static const double SC[256][3] = {
1026     {0x0p+0, 0x0p+0, 0x1p+0}, /* 0 */
1027     {-0x1.c0f6cp-35, 0x1.921f892b900fep-9, 0x1.ffff621623fap-1}, /* 1 */
1028     {-0x1.9c7935ep-35, 0x1.921f0ea27ce01p-8, 0x1.fffd8858eca2ep-1}, /* 2 */
1029     {-0x1.d14d1acp-34, 0x1.2d96af779b0bbp-7, 0x1.fffa72c986392p-1}, /* 3 */
1030     {-0x1.dba8f6a8p-33, 0x1.921d1ce2d0a1cp-7, 0x1.fff62169dddaap-1}, /* 4 */
1031     {0x1.a6b7cdfp-32, 0x1.f6a29bdb7377p-7, 0x1.fff0943c02419p-1}, /* 5 */
1032     {0x1.b49618dp-33, 0x1.2d936d1506f3dp-6, 0x1.ffe9cb44829cp-1}, /* 6 */
1033     {-0x1.398d6fcp-35, 0x1.5fd4d1e21de6dp-6, 0x1.ffe1c687174b1p-1}, /* 7 */
1034     {-0x1.e9e9a8c8p-31, 0x1.9215597791e0ap-6, 0x1.ffd886097afcfp-1}, /* 8 */
1035     {-0x1.34e844cp-32, 0x1.c454f2e9480c7p-6, 0x1.ffce09ce95933p-1}, /* 9 */
1036     {-0x1.989a8a4p-32, 0x1.f693709b94f92p-6, 0x1.ffc251dfbac0cp-1}, /* 10 */
1037     {0x1.04a9b99p-30, 0x1.146860e69a571p-5, 0x1.ffb55e40a5c43p-1}, /* 11 */
1038     {-0x1.56947cp-36, 0x1.2d865748774adp-5, 0x1.ffa72efff95d1p-1}, /* 12 */
1039     {-0x1.c348768p-35, 0x1.46a396d34121ap-5, 0x1.ff97c420a8451p-1}, /* 13 */
1040     {0x1.9e80552p-32, 0x1.5fc00e6e4c65cp-5, 0x1.ff871dacd8761p-1}, /* 14 */
1041     {0x1.3f11d74p-34, 0x1.78dbaa97099ebp-5, 0x1.ff753bb18af95p-1}, /* 15 */
1042     {0x1.c039af4p-33, 0x1.91f65fc0abc0ap-5, 0x1.ff621e370ca7ap-1}, /* 16 */
1043     {0x1.53e1f8p-35, 0x1.ab101bf74ac2ep-5, 0x1.ff4dc54b00181p-1}, /* 17 */
1044     {0x1.114a649p-29, 0x1.c428d7de920e9p-5, 0x1.ff3830f2e9043p-1}, /* 18 */
1045     {0x1.adf0ef4p-31, 0x1.dd40723a3cdfbp-5, 0x1.ff21614b9d9adp-1}, /* 19 */
```

```
1310  static inline void
1311  fast_two_sum(double *hi, double *lo, double a, double b)
1312  {
1313    double e;
1314
1315    *hi = a + b;
1316    e = *hi - a;
1317    *lo = b - e;
1318  }
```

$$(a + b) - (a \oplus b) = b \ominus ((a \oplus b) \ominus a)$$

- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\overbrace{\sin\left(\frac{\pi}{2N} i\right)}^{} \cos(r'') + \overbrace{\cos\left(\frac{\pi}{2N} i\right)}^{}\right)$

Actual Implementation (3)

- Step 1: Argument Reduction.
  - $\hat{r} := x - 2\hat{\pi}k, \quad$ where $k := \left\lfloor \frac{x}{2\hat{\pi}} \right\rceil$.
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi} - \hat{r}|\}$.

```
1832  double err;
1833  static const double sgn[2] = {1.0, -1.0};
1834  if (is_sin)
1835    {
1836      s_mul (&sh, &sl, sgn[neg] * SC[i][2], sh, sl)
1837      s_mul (&ch, &cl, sgn[neg] * SC[i][1], ch, cl)
1838      fast_two_sum (h, l, ch, sh);
1839      *l += sl + cl;
1840
1841
1842
1843
1844
1845
1846
1847
```

Represent $v \notin \mathbb{F}$ by $v := v_h + v_l$,
where $v_h, v_l \in \mathbb{F}$ and $|v_h| \gg |v_l|$.

```
1021  /* Table generated with ./buildSC 15 using accompanying buildSC.c.
1022     For each i, 0 <= i < 256, xi=i/2^11+SC[i][0], with
1023     SC[i][1] and SC[i][2] approximating sin2pi(xi) and cos2pi(xi)
1024     respectively, both with 53+15 bits of accuracy. */
1025  static const double SC[256][3] = {
1026     {0x0p+0, 0x0p+0, 0x1p+0}, /* 0 */
1027     {-0x1.c0f6cp-35, 0x1.921f892b900fep-9, 0x1.ffff621623fap-1}, /* 1 */
1028     {-0x1.9c7935ep-35, 0x1.921f0ea27ce01p-8, 0x1.fffd8858eca2ep-1}, /* 2 */
1029     {-0x1.d14d1acp-34, 0x1.2d96af779b0bbp-7, 0x1.fffa72c986392p-1}, /* 3 */
1030     {-0x1.dba8f6a8p-33, 0x1.921d1ce2d0a1cp-7, 0x1.fff62169dddaap-1}, /* 4 */
1031     {0x1.a6b7cdfp-32, 0x1.f6a29bdb7377p-7, 0x1.fff0943c02419p-1}, /* 5 */
1032     {0x1.b49618dp-33, 0x1.2d936d1506f3dp-6, 0x1.ffe9cb44829cp-1}, /* 6 */
1033     {-0x1.398d6fcp-35, 0x1.5fd4d1e21de6dp-6, 0x1.ffe1c687174b1p-1}, /* 7 */
1034     {-0x1.e9e9a8c8p-31, 0x1.9215597791e0ap-6, 0x1.ffd886097afcfp-1}, /* 8 */
1035     {-0x1.34e844cp-32, 0x1.c454f2e9480c7p-6, 0x1.ffce09ce95933p-1}, /* 9 */
1036     {-0x1.989a8a4p-32, 0x1.f693709b94f92p-6, 0x1.ffc251dfbac0cp-1}, /* 10 */
1037     {0x1.04a9b99p-30, 0x1.146860e69a571p-5, 0x1.ffb55e40a5c43p-1}, /* 11 */
1038     {-0x1.56947cp-36, 0x1.2d865748774adp-5, 0x1.ffa72efff95d1p-1}, /* 12 */
1039     {-0x1.c348768p-35, 0x1.46a396d34121ap-5, 0x1.ff97c420a8451p-1}, /* 13 */
1040     {0x1.9e80552p-32, 0x1.5fc00e6e4c65cp-5, 0x1.ff871dacd8761p-1}, /* 14 */
1041     {0x1.3f11d74p-34, 0x1.78dbaa97099ebp-5, 0x1.ff753bb18af95p-1}, /* 15 */
1042     {0x1.c039af4p-33, 0x1.91f65fc0abc0ap-5, 0x1.ff621e370ca7ap-1}, /* 16 */
1043     {0x1.53e1f8p-35, 0x1.ab101bf74ac2ep-5, 0x1.ff4dc54b00181p-1}, /* 17 */
1044     {0x1.114a649p-29, 0x1.c428d7de920e9p-5, 0x1.ff3830f2e9043p-1}, /* 18 */
1045     {0x1.adf0ef4p-31, 0x1.dd40723a3cdfbp-5, 0x1.ff21614b9d9adp-1}, /* 19 */
```

- Step 3: Table Lookup + Reconstruction.
  - $\sin(x) = (-1)^s \left(\overbrace{\sin\left(\frac{\pi}{2N} i\right)}^{} \cos(r'') + \overbrace{\cos\left(\frac{\pi}{2N} i\right)}^{}\right)$

```
1310  static inline void
1311  fast_two_sum(double *hi, double *lo, double a, double b)
1312  {
1313    double e;
1314
1315    *hi = a + b;
1316    e = *hi - a; /* exact */
1317    *lo = b - e; /* exact */
1318  }
```

Actual Implementation (4)

- Step 1: Argument Redu
  - $\hat{r} := x - 2\hat{\pi}k,$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi}$
  - $\widehat{r''} := \widehat{r'} - \frac{\hat{\pi}}{2N} i,$
- Step 2: Polynomial App
  - $\sin(\widehat{r''}) = \widehat{r''} - \left(\frac{1}{3!}\right)$
  - $\cos(\widehat{r''}) = 1 - \left(\frac{1}{2!}\right)$
- Step 3: Table Lookup +
  - $\sin(x) = (-1)^s \left(\sin$

```
2033  double
2034  cr_sin (double x)
2035  {
2036    b64u64_u t = {.f = x};
2037    int e = (t.u >> 52) & 0x7ff;
2038
2039    if (__builtin_expect (e == 0x7ff, 0)) /* NaN, +Inf and -Inf. */
2040      {
2041        t.u = ~0ull;
2042        return t.f;
2043      }
2044
2045    /* now x is a regular number */
2046
2047    /* For |x| <= 0x1.7137449123ef6p-26, sin(x) rounds to x (to nearest):
2048       we can assume x >= 0 without loss of generality since sin(-x) = -sin(x),
2049       we have x - x^3/6 < sin(x) < x for say 0 < x <= 1 thus
2050       |sin(x) - x| < x^3/6.
2051       Write x = c*2^e with 1/2 <= c < 1.
2052       Then ulp(x)/2 = 2^(e-54), and x^3/6 = c^3/6*2^(3e), thus
2053       x^3/6 < ulp(x)/2 rewrites as c^3/6*2^(3e) < 2^(e-54),
2054       or c^3*2^(2e+53) < 3 (1).
2055       For e <= -26, since c^3 < 1, we have c^3*2^(2e+53) < 2 < 3.
2056       For e=-25, (1) rewrites 8*c^3 < 3 which yields c <= 0x1.7137449123ef6p-1.
2057    */
2058    uint64_t ux = t.u & 0x7fffffffffffffff;
2059    // 0x3e57137449123ef6 = 0x1.7137449123ef6p-26
2060    if (ux <= 0x3e57137449123ef6)
2061      // Taylor expansion of sin(x) is x - x^3/6 around zero
2062      // for x=-0, fma (x, -0x1p-54, x) returns +0
2063      return (x == 0) ? x : __builtin_fma (x, -0x1p-54, x);
2064
2065    double h, l, err;
2066    err = sin_fast (&h, &l, x);
2067    double left  = h + (l - err), right = h + (l + err);
2068    /* With SC[] from ./buildSC 15 we get 1100 failures out of 50000000
2069       random tests, i.e., about 0.002%. */
2070    if (__builtin_expect (left == right, 1))
2071      return left;
2072
2073    return sin_accurate (x);
2074  }
```

Actual Implementation (4)

- Step 1: Argument Redu
  - $\hat{r} := x - 2\hat{\pi}k,$
  - $\widehat{r'} \in \{|\hat{r}|, |\hat{\pi} - \hat{r}|, |2\hat{\pi}$
  - $\widehat{r''} := \widehat{r'} - \frac{\hat{\pi}}{2N} i,$
- Step 2: Polyn
  - $\sin(\widehat{r''}) =$
  - $\cos(\widehat{r''}) = 1 - \left(\frac{1}{2!}\right)($
- Step 3: Table Lookup +
  - $\sin(x) = (-1)^s \left(\sin$

```
2033  double
2034  cr_sin (double x)
2035  {
2036    b64u64_u t = {.f = x};
2037    int e = (t.u >> 52) & 0x7ff;
2038
2039    if (__builtin_expect (e == 0x7ff, 0)) /* NaN, +Inf and -Inf. */
2040      {
2041        t.u = ~0ull;
2042        return t.f;
2043      }
2044
2045    /* now x is a regular number */
2046
2047    /* For |x| <= 0x1.7137449123ef6p-26, sin(x) rounds to x (to nearest):
2048       we can assume x >= 0 without loss of generality since sin(-x) = -sin(x),
2049       we have x - x^3/6 < sin(x) < x for say 0 < x <= 1 thus
2058    uint64_t ux = t.u & 0x7fffffffffffffff;
2059    // 0x3e57137449123ef6 = 0x1.7137449123ef6p-26
2060    if (ux <= 0x3e57137449123ef6)
2061      // Taylor expansion of sin(x) is x - x^3/6 around zero
2062      // for x=-0, fma (x, -0x1p-54, x) returns +0
2063      return (x == 0) ? x : __builtin_fma (x, -0x1p-54, x);
2064
2065    double h, l, err;
2066    err = sin_fast (&h, &l, x);
2067    double left  = h + (l - err), right = h + (l + err);
2068    /* With SC[] from ./buildSC 15 we get 1100 failures out of 50000000
2069       random tests, i.e., about 0.002%. */
2070    if (__builtin_expect (left == right, 1))
2071      return left;
2072
2073    return sin_accurate (x);
2074  }
```

Is this implementation **correct**?

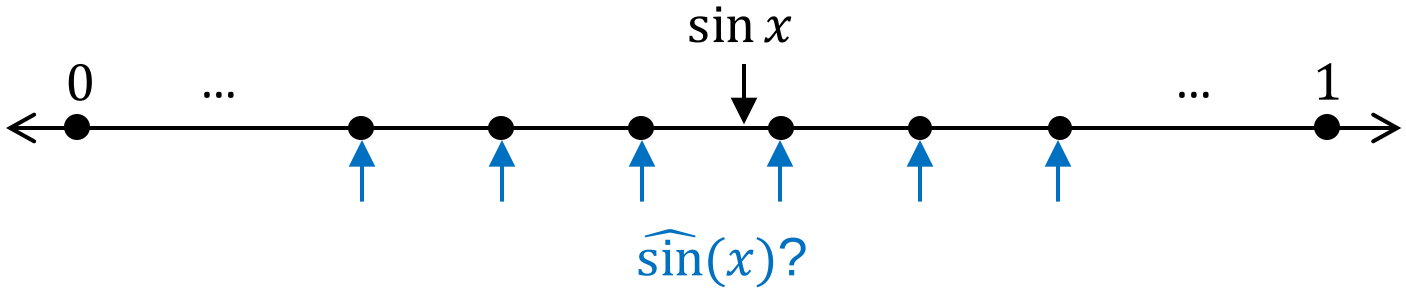
Correctness

Are existing implementations of $\sin$ correct in any formal sense?

- Different libraries (and even different versions of them) have distinct implementations.
  - GNU libc
  - LLVM libc
  - Intel Math Library
  - AMD Math Library
  - Apple Math Library
  - Microsoft Math Library
  - CUDA Math Library
  - …

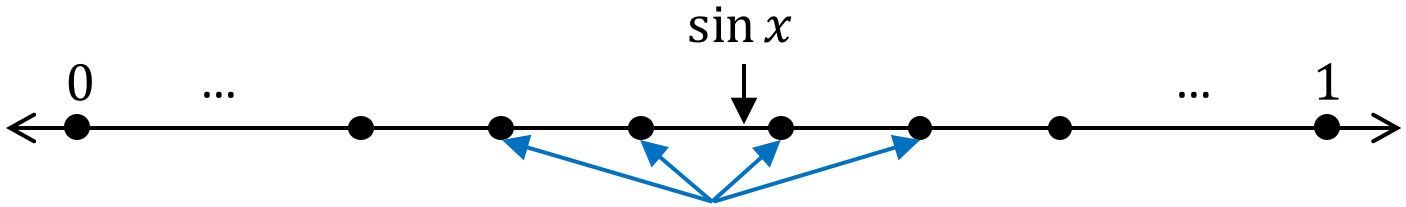
Correctness

Are existing implementations of sin correct in any formal sense?



Correctness

Are existing implementations of $\sin$ correct in any formal sense?



$\widehat{\sin}(x)$ $\Leftrightarrow$ It has an error of < 2 ulp.

- ULP error: standard metric for numerical libraries.
- Worst-case error: $\max_{x \in \mathbb{F}} err_{ulp}\left(\sin(x), \widehat{\sin}(x)\right)$.

Correctness Issues

- Intel Processors: `fsin`, `fcos`, … were claimed to have < 1 ulp error.

Pentium Processor User's Manual (1993)

Table 6-14. Transcendental Instructions

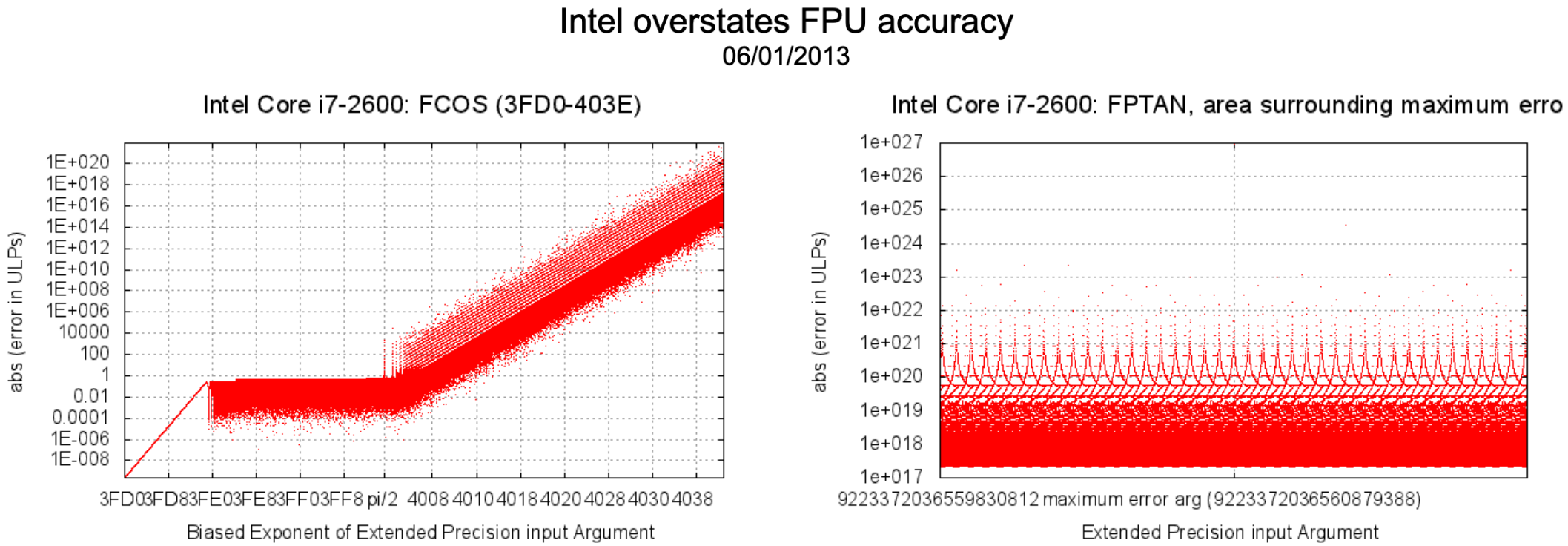
Mnemonic	Operation
FSIN*	Sine
FCOS*	Cosine
FSINCOS*	Sine and Cosine

…

On the Pentium processor, the worst case error on functions is less than 1 ulp when rounding to the nearest-even and less than 1.5 ulps when rounding in other modes. The functions are

Correctness Issues

- Intel Processors: `fsin`, `fcos`, … were claimed to have < 1 ulp error.



These errors still persist to support backward compatibility…

Correctness Issues

- GNU libc: `sin`, `cos`, … are aimed to have < 10 ulp error.
  - glibc 2.18:
    ```
    sin(4831921577.4601803) = -0.918193578775631
    cos(4831921577.4601803) = -0.396131987972694
    ```
  - glibc 2.19:
    ```
    sin(4831921577.4601803) = -0.918193578775631
    cos(4831921577.4601803) = +0.396131987972694
    ```

Correctness Issues

- GNU libc: `sin`, `cos`, … are aimed to have < 10 ulp error.
  - glibc 2.18:
    ```
    sin(4831921577.4601803) = -0.918193578775631
    cos(4831921577.4601803) = -0.396131987972694
    ```
    ← This is correct!
  - glibc 2.19:
    ```
    sin(4831921577.4601803) = -0.918193578775631
    cos(4831921577.4601803) = +0.396131987972694
    ```

Math libraries keep evolving, and can have huge errors…

Correctness Issues

- CORE-MATH (since 2022): `sin`, `cos`, … are claimed to have ≤ 0.5 ulp error.

  - `acos(+7.4999999999999889e-01) = +1.4946091309285376`**`4e+00`**
    `acos(+7.4999999999999889e-01) = +7.2273424781341577`**`5e-01`** `(correct)`

  - `erf(+1.4812534985742959`**`4e+306`**`) = +1.4812534985742959`**`4e+306`**
    `erf(+1.4812534985742959`**`4e+306`**`) = +1.0000000000000000e+00 (correct)`

Correctness Issues

- CORE-MATH (since 2022): `sin`, `cos`, … are claimed to have ≤ 0.5 ulp error.

  - `acos(+7.4999999999999889e-01) = +1.4946091309285376 4e+00`
    `acos(+7.4999999999999889e-01) = +7.2273424781341577 5e-01 (correct)`

  - `erf(+1.4812534985742959 4e+306) = +1.4812534985742959 4e+306`
    `erf(+1.4812534985742959 4e+306) = +1.0000000000000000 0e+00 (correct)`

Even extremely carefully developed libraries can have huge errors…

Correctness Issues

- CORE-MATH (since 2022): `sin`, `cos`, ... are claimed to have ≤ 0.5 ulp error.
  - `acos(+7.4999999999999889e-01) = +1.4946091309285376 4e+00`
    `acos(+7.4999999999999889e-01) = +7.2273424781341577 5e-01 (correct)`
  - `er`
    `er`

Can we develop **correct** and **efficient** libraries?

Even extremely carefully developed libraries can have huge errors...

Research Directions

"How to develop **correct** and **efficient** math libraries?"

Approach 1. Establish the correctness of existing libraries.

Approach 2. Develop new libraries with correctness in mind.

Research Directions

"How to develop **correct** and **efficient** math libraries?"

Approach 1. Establish the correctness of existing libraries.

- **Testing.** Exhaustive testing is infeasible. Random testing is commonly used.

library	GNU libc	IML	AMD	Newlib	OpenLibm	Musl	Apple	LLVM	MSVC	FreeBSD	ArmPL	CUDA	ROCm
version	2.40	2024.0.2	4.2	4.4.0	0.8.3	1.2.5	14.5	18.1.8	2022	14.1	24.04	12.2.1	5.7.0
acos	**0.523**	0.531	1.36	0.930	0.930	0.930	1.06		0.934	0.930	1.52	1.53	0.772
acosh	2.25	**0.509**	1.32	2.25	2.25	2.25	2.25		3.22	2.25	2.66	2.52	0.661
asin	**0.516**	0.531	1.06	0.981	0.981	0.981	0.709		1.05	0.981	2.69	1.99	0.710
asinh	1.92	**0.507**	1.65	1.92	1.92	1.92	1.58		2.05	1.92	2.04	2.57	0.661
atan	**0.523**	0.528	0.863	0.861	0.861	0.861	0.876		0.863	0.861	2.24	1.77	1.73
atanh	1.78	**0.507**	1.04	1.81	1.81	1.80	2.01		2.50	1.81	3.00	2.50	0.664
cbrt	3.67	0.523	1.53e22	0.670	0.668	0.668	0.729		1.86	0.668	1.79	**0.501**	**0.501**
cos	**0.516**	0.518	0.919	0.887	0.834	0.834	0.948	Inf	0.897	0.834		1.52	0.797
cosh	1.93	**0.516**	1.85	2.67	1.47	1.04	0.523		1.91	1.47	1.93	1.40	0.563
erf	1.43	**0.773**	1.00	1.02	1.02	1.02	6.41		4.62	1.02	2.29	1.50	1.12
erfc	5.19	**0.826**		4.08	4.08	3.72	10.7		8.46	4.08	1.71	4.51	4.08

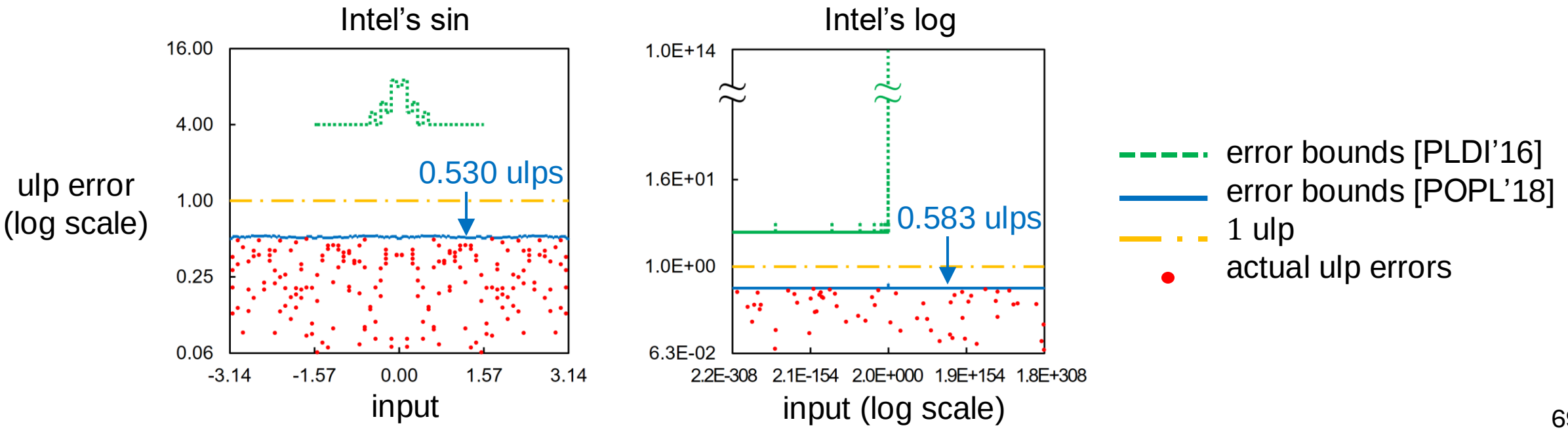
Largest known ulp errors (2024)

Research Directions

"How to develop **correct** and **efficient** math libraries?"

Approach 1. Establish the correctness of existing libraries.

- **Verification.** Some program analyses were developed (e.g., [POPL'18], [PLDI'16]).



Wrap-Up

Correctness / Fundamental Limits / Efficiency

Function Evaluation	Use **floats** intricately.	[Ongoing], [POPL'18], [PLDI'16].
Sample Generation	Assume **reals.**	[Ongoing], [Submitted].
Differentiation	Assume **differentiability.**	[ICLR'24] (Spotlight), [ICML'23], [NeurIPS'20] (Spotlight).
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	[Submitted], [POPL'23], [POPL'20], [AAAI'20], [NeurIPS'18].
Function Approximation		[Submitted], [Submitted], [Neural Networks'24], [TMLR'23].

Correctness / Fundamental Limits / Efficiency

Function Evaluation	Use **floats** intricately.	[Ongoing], [POPL'18], [PLDI'16].
Sample Generation	Assume **reals.**	[Ongoing], [Submitted].
Differentiation	Assume **differentiability.**	[ICLR'24] (Spotlight), [ICML'23], [NeurIPS'20] (Spotlight).
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	[Submitted], [POPL'23], [POPL'20], [AAAI'20], [NeurIPS'18].
Function Approximation		[Submitted], [Submitted], [Neural Networks'24], [TMLR'23].

High-Level Messages

- Numerical libraries have been studied and developed for nearly a century.
- Despite such efforts, these libraries often still lack rigorous correctness guarantees.
- PL techniques would be helpful in ensuring the correctness of practical libraries.