“연속적 계산”의 기초 다지기

이원열
POSTECH

SIGPL 여름학교, 08/20/2025

Introduction

- **Education + Employment.**
  - POSTECH: Assistant Professor in CS (2024–Present)
  - CMU: Postdoc in CS (2023–2024)
  - Stanford: PhD in CS (2014–2017, 2020–2023)
    KAIST: Researcher in CS (2017–2020)
  - POSTECH: BS in CS and Math (2010–2014)

- **Research.**
  - PL: POPL (2023, 2020, 2018, 2014), PLDI (2025a, 2025b, 2016), CAV (2025).
  - ML: NeurIPS (2020-Spotlight, 2018), ICML (2025, 2023), ICLR (2024-Spotlight), AAAI (2020).

Research Interests

Mathematical Properties of **Programs and Computations**

Research Interests

Mathematical Properties of **Programs and Computations**

Correctness

Efficiency

Fundamental Limits

…

- Is a practically-used computation "correct" in any formal sense?
- Is there a more "efficient" computation that is correct?
- Is there any "fundamental limit" to achieving the computation?

Research Interests

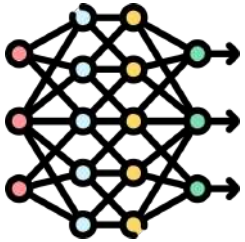
Mathematical Properties of **Programs and Computations**

Continuous Values

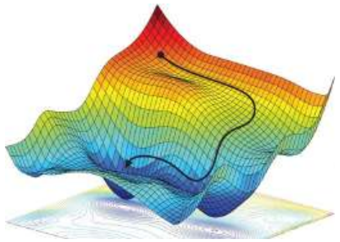
$6,\ 2.8,\ \frac{3}{7},\ \sqrt{5},\ \frac{\pi}{4},\ \ldots$

Operations on Them

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

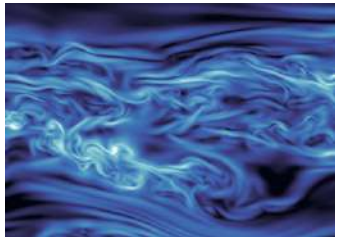


Machine Learning



Optimization

Computer Graphics

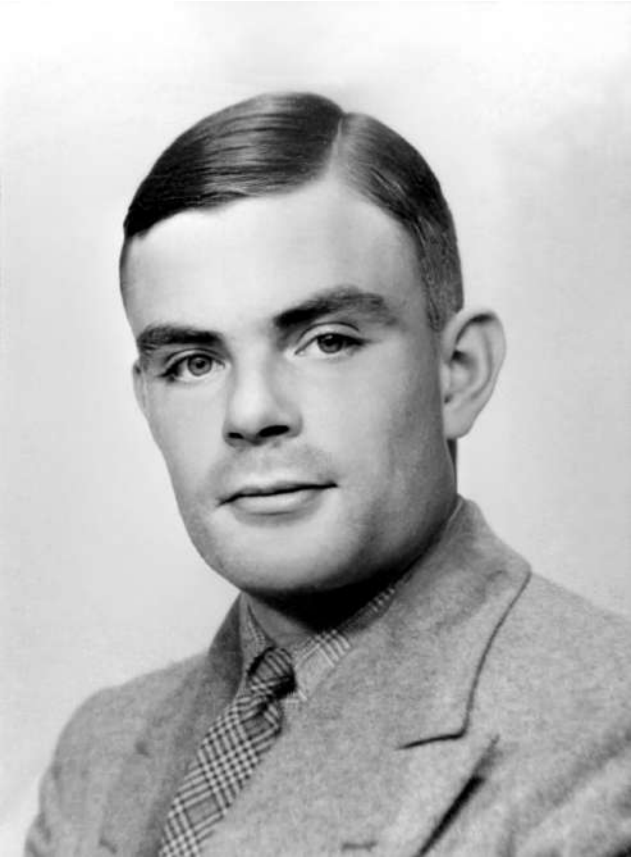


Scientific Computing

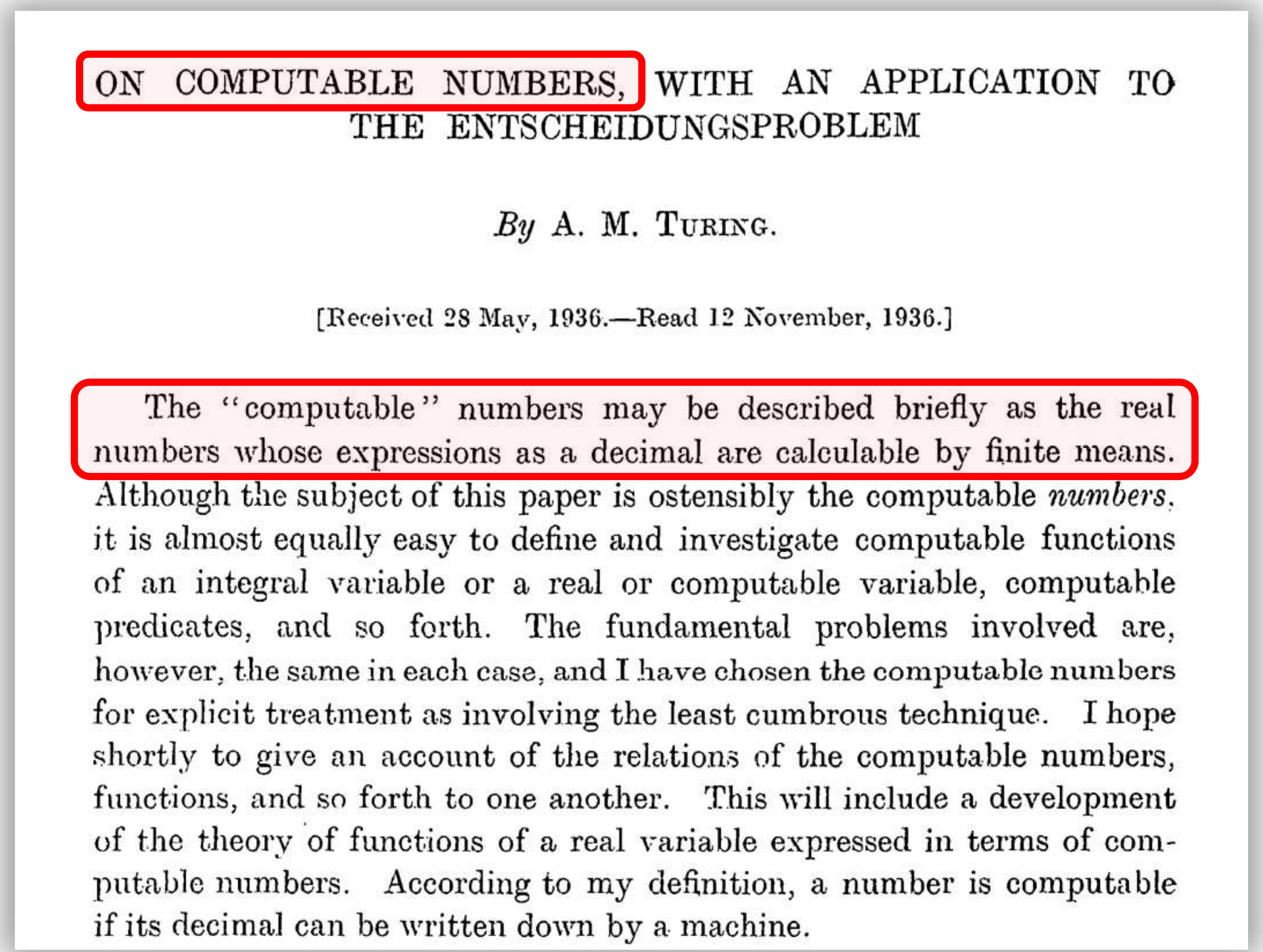
Differential Privacy

...

Early Days



Alan Turing



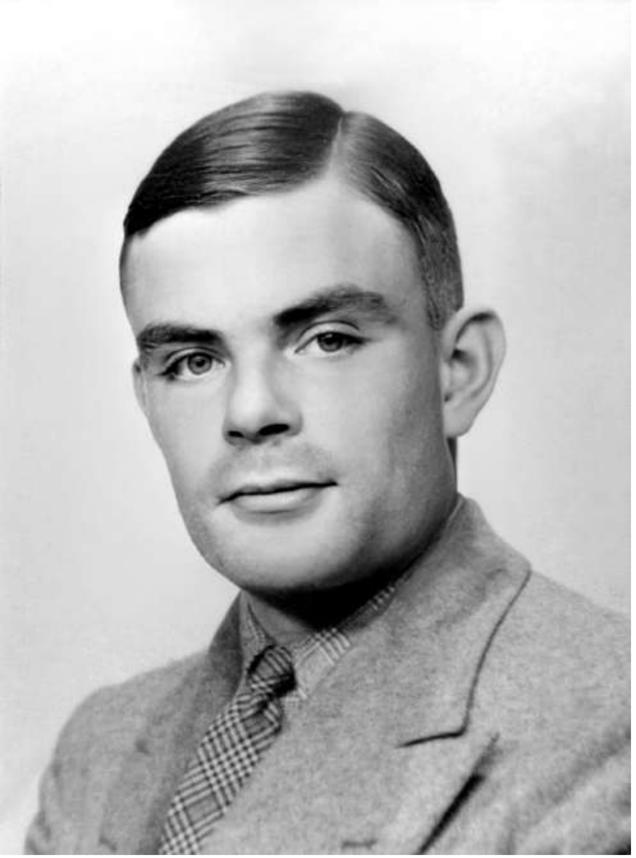
ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

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 Days



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.

These Days

As a result of ~90 years of substantial efforts,

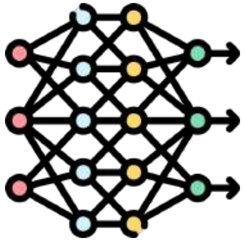
GNU Math/Scientific Library
Intel MKL
NumPy
SciPy
TensorFlow
PyTorch
JAX
Pyro
Stan
…

These Days

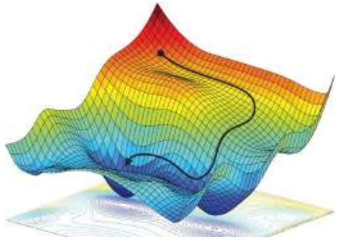
As a result of ~90 years of substantial efforts,

GNU Math/Scientific Library, Intel MKL, NumPy, SciPy

TensorFlow, PyTorch, JAX, Pyro, Stan, …

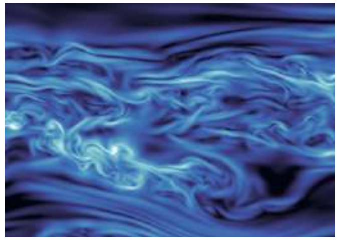


Machine Learning



Optimization

Computer Graphics



Scientific Computing

Differential Privacy

…

Fundamental Computations

Function Evaluation	Compute $\sin(x)$.	GNU Math Library Intel MKL …
Sample Generation	Sample from $\mathcal{N}(\mu, \sigma^2)$.	NumPy SciPy …

Fundamental Computations

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

Mathematically **Correct?**
Can Be More **Efficient?**
Any Fundamental **Limits?**

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

Mathematically **Correct?**
Can Be More **Efficient?**
Any Fundamental **Limits?**

Actual implementations

Function Evaluation	Use **floats** intricately.	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 theory

Our Works

(Dis)Prove **Correctness.**
Improve **Efficiency.**
Prove **Fundamental Limits.**

	Actual implementations ↔ Underlying theory	PL	ML
Function Evaluation	Use **floats** intricately.	[Ongoing 1] [Ongoing 2] [POPL 18] [PLDI 16]	
Sample Generation	Assume **reals.**	[Ongoing 1] [Ongoing 2] [PLDI 25a]	
Differentiation	Assume **differentiability.**		[ICLR 24] (Spotlight) [ICML 23] [NeurIPS 20] (Spotlight)
Integration (≈ **Probabilistic Inference**)	Assume **integrability.**	[Submitted] [PLDI 25b] [POPL 23] [POPL 20]	[AAAI 20] [NeurIPS 18]
Function Approximation		[CAV 25]	[ICML 25] [Neural Networks 24]

Our Works

(Dis)Prove **Correctness.**
Improve **Efficiency.**
Prove **Fundamental Limits.**

	Actual implementations / Underlying theory	PL	ML
Function Evaluation	Use **floats** intricately.	[Ongoing 1] [Ongoing 2] [POPL 18] [PLDI 16]	
Sample Generation	Assume **reals.**	[Ongoing 1] [Ongoing 2] [PLDI 25a]	
Differentiation	Assume **differentiability.**		[ICLR 24] (Spotlight) [ICML 23] [NeurIPS 20] (Spotlight)
Integration (≈ **Probabilistic Inference**)	Assume **integrability.**	[Submitted] [PLDI 25b] [POPL 23] [POPL 20]	[AAAI 20] [NeurIPS 18]
Function Approximation		[CAV 25]	[ICML 25] [Neural Networks 24]

Function Evaluation

Problem

- **Goal.** For $f \in \{\exp, \ln, \sin, \text{asin}, \ldots\}$ and $x \in \mathbb{F}$,

compute $f(x) \in \mathbb{R}$ accurately and efficiently.

Problem

- **Goal.** For $f \in \{\exp, \ln, \sin, \text{asin}, \ldots\}$ and $x \in \mathbb{F}$,

  compute $f(x) \in \mathbb{R}$ accurately and efficiently.

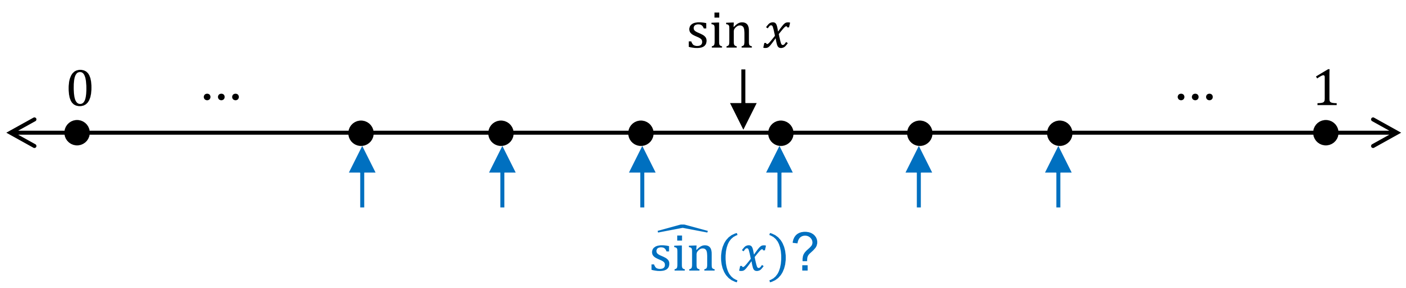
- **Fact.** We cannot exactly compute $f(x)$ for almost all x.
  - $\exp(x) \notin \mathbb{F}$ for all $x \in \mathbb{F} \setminus \{0\}$.
  - $\ln(x) \notin \mathbb{F}$ for all $x \in \mathbb{F} \setminus \{1\}$.
  - $\sin(x) \notin \mathbb{F}$ for all $x \in \mathbb{F} \setminus \{0\}$.
  - $\cdots$
  - These are by Lindemann-Weierstrass and Siegel–Shidlovsky Theorems (1885, 1929).

Problem

- **Goal.** For $f \in \{\exp, \ln, \sin, \text{asin}, \ldots\}$ and $x \in \mathbb{F}$,

  compute $f(x) \in \mathbb{R}$ accurately and efficiently.

- **Question.** How much accuracy do we want?

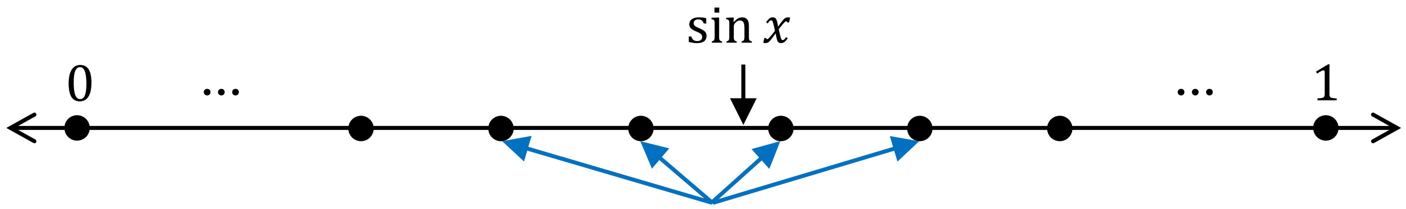


Problem

- **Goal.** For $f \in \{\exp, \ln, \sin, \mathrm{asin}, \ldots\}$ and $x \in \mathbb{F}$,

  compute $f(x) \in \mathbb{R}$ accurately and efficiently.

- **Question.** How much accuracy do we want?



$\widehat{\sin}(x) \iff \widehat{\sin}$ has an error of ≤ 2 ulps.

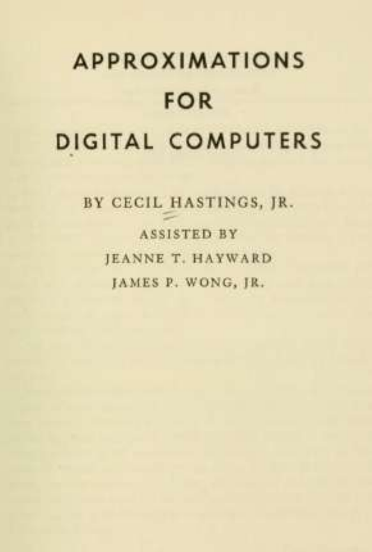
  - ULP error: $\mathrm{err}_{\mathrm{ulp}}(r, \hat{r}) \approx |[r, \hat{r}) \cap \mathbb{F}|$.
  - Best possible accuracy: 0.5 ulps.
  - Typical target accuracy: 1–10 ulps.

Problem

- **Goal.** For $f \in \{\exp, \ln, \sin, \text{asin}, \ldots\}$ and $x \in \mathbb{F}$,

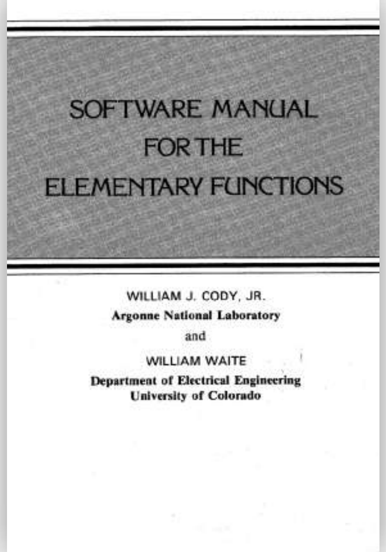
  compute $f(x) \in \mathbb{R}$ accurately and efficiently.

- **Note.** It has been well studied for 70+ years.

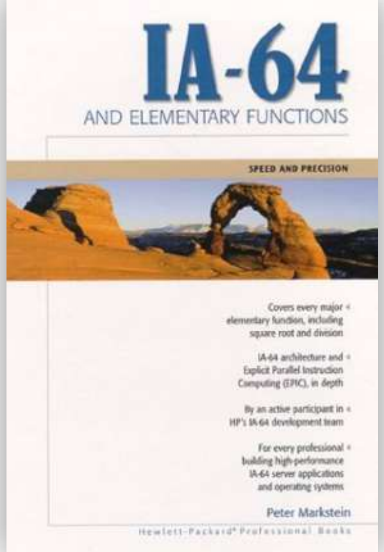


1955

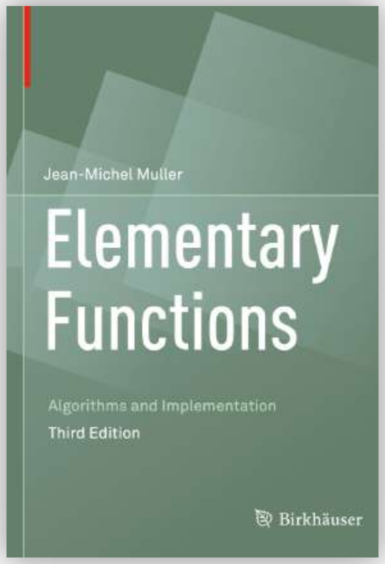
1968



1980



2000



2016

Existing Solution

- **Math Library.** Implements routines for evaluating $f(x)$.

- **Example.** GNU libc includes an implementation of math.h.

```
double exp (double x)                          [Function]
float expf (float x)                           [Function]
long double expl (long double x)               [Function]
_FloatN expfN (_FloatN x)                      [Function]
_FloatNx expfNx (_FloatNx x)                   [Function]

double sin (double x)                          [Function]
float sinf (float x)                           [Function]
long double sinl (long double x)               [Function]
_FloatN sinfN (_FloatN x)                      [Function]
_FloatNx sinfNx (_FloatNx x)                   [Function]

double erf (double x)                          [Function]
float erff (float x)                           [Function]
long double erfl (long double x)               [Function]
_FloatN erffN (_FloatN x)                      [Function]
_FloatNx erffNx (_FloatNx x)                   [Function]
```

Existing Solution

- **Math Library.** Implements routines for evaluating $f(x)$.
  - Different implementations.

- **Example.**
  - GNU libc
  - LLVM libc
  - CORE-MATH
  - Intel Math Library →
  - AMD Math Library
  - Apple Math Library
  - CUDA Math Library
  - …

NumPy
SciPy
TensorFlow
PyTorch
Pyro
Stan
…

Existing Solution

- **Math Library.** Implements routines for evaluating $f(x)$.
  - Different implementations. Different claims.

- **Example.**
  - GNU libc — Claim: ≤ 10 ulps.
  - LLVM libc — Claim: ≤ 0.5–1 ulps.
  - CORE-MATH — Claim: ≤ 0.5 ulps.
  - Intel Math Library — Claim: ≤ 0.6 ulps.
  - AMD Math Library — [Undocumented]
  - Apple Math Library — [Undocumented]
  - CUDA Math Library — Claim: ≤ 1–8 ulps.
  - …

Table 3: Double precision: Largest known error.

library version	GNU libc 2.41	IML 2025.0.0	AMD 5.0	Newlib 4.5.0	OpenLibm 0.8.5	Musl 1.2.5	Apple 15.1.1
acos	**0.523**	0.531	1.36	0.930	0.930	0.930	1.06
acosh	2.25	**0.509**	1.32	2.25	2.25	2.25	2.25
asin	**0.516**	0.531	1.06	0.981	0.981	0.981	0.709
asinh	1.92	**0.507**	1.65	1.92	1.92	1.92	1.58
atan	**0.523**	0.528	0.863	0.861	0.861	0.861	0.876
atanh	1.78	**0.507**	1.04	1.81	1.81	1.80	2.01
cbrt	3.67	0.523	1.53e22	0.670	0.668	0.668	0.729
cos	**0.516**	0.518	0.919	0.887	0.834	0.834	0.948
cosh	1.93	**0.516**	1.85	2.67	1.47	1.04	0.523
erf	1.43	**0.773**	1.00	1.02	1.02	1.02	6.41
erfc	5.19	**0.827**		4.08	4.08	3.72	10.7
exp	0.511	0.530	1.01	0.949	0.949	0.511	0.521

Existing Solution

- **Math Library.** Implements routines for evaluating $f(x)$.
  - Different implementations. Different claims.

- **Example.**
  - GNU libc
  - LLVM libc
  - CORE-M
  - Intel Math
  - AMD Math Library [Undocumented]
  - Apple Math Library [Undocumented]
  - CUDA Math Library Claim: ≤ 1–8 ulps.
  - …

m error.

					Libm	Musl	Apple
					.5	1.2.5	15.1.1
					30	0.930	1.06
					25	2.25	2.25
					81	0.981	0.709
					92	1.92	1.58
atan	**0.523**	0.523	0.863	0.861	0.861	0.861	0.876
atanh	1.78	**0.507**	1.04	1.81	1.81	1.80	2.01
cbrt	3.67	0.523	1.53e22	0.670	0.668	0.668	0.729
cos	**0.516**	0.518	0.919	0.887	0.834	0.834	0.948
cosh	1.93	**0.516**	1.85	2.67	1.47	1.04	0.523
erf	1.43	**0.773**	1.00	1.02	1.02	1.02	6.41
erfc	5.19	**0.827**		4.08	4.08	3.72	10.7
exp	0.511	0.530	1.01	0.949	0.949	0.511	0.521

Are existing math libraries **correct?**

Can we make them more **efficient?**

Issues

- **GNU libc.** Aims to have ≤ 10 ulp error.
  - glibc 2.18: `cos(4.83...e+9) = -0.396131987972...`
  - glibc 2.19: `cos(4.83...e+9) = +0.396131987972...`

Issues

- **GNU libc.** Aims to have ≤ 10 ulp error.
  - glibc 2.18: `cos(4.83...e+9) = -0.396131987972...` (correct)
  - glibc 2.19: `cos(4.83...e+9) = +0.396131987972...` (error $> 10^{18}$ ulps)

Math libraries keep **evolving.** Some updates introduce **new errors!**

Issues

- **GNU libc.** Aims to have ≤ 10 ulp error.
  - glibc 2.18: `cos(4.83...e+9) = -0.396131987972...` (correct)
  - glibc 2.19: `cos(4.83...e+9) = +0.396131987972...` (error > 10^{18} ulps)

Math libraries keep **evolving.** Some updates introduce **new errors!**

Joseph Myers **2014-02-21 22:30:12 UTC** Comment 1

```
Siddhesh, you've been doing the most with these functions lately....
```

Rich Felker **2014-02-22 02:52:15 UTC** Comment 2

```
For the record, the old behavior with the negative sign is correct. Tested against
other libm implementations that handle large trig arguments and Wolfram Alpha.
```

Siddhesh Poyarekar **2014-02-24 02:34:25 UTC** Comment 3

```
Must be due to some of the code consolidation I did recently.  I'll take a look.

   In 84ba214c, I removed some redundant sign computations and in the
   process, I incorrectly got rid of a temporary variable, thus passing
   the absolute value of the input to bsloww1.  This caused #16623.
```

Issues

- **GNU libc.** Aims to have ≤ 10 ulp error.
  - glibc 2.18: `cos(4.83...e+9)  = -0.396131987972...` (correct)
  - glibc 2.19: `cos(4.83...e+9)  = +0.396131987972...` (error > 10^{18} ulps)

Math libraries keep **evolving.** Some updates introduce **new errors!**

  - glibc 2.27: `sin(2.41...e+23) = 2.3881763752596...e-17` (correct)
  - glibc 2.28: `sin(2.41...e+23) = 2.3881763752648...e-17` (error > 10^{4} ulps)
  - glibc 2.40: Same as glibc 2.28.

```
2018-04-03  Wilco Dijkstra  <wdijkstr@arm.com>

        * sysdeps/ieee754/dbl-64/s_sin.c (reduce_sincos_1): Rename to
        reduce_sincos, improve accuracy to 136 bits.
        (do_sincos_1): Rename to do_sincos, remove fallbacks to slow functions.
        (__sin): Use improved reduction and simplified do_sincos calculation.
```

Issues

- **CORE-MATH.** Claims to have ≤ 0.5 ulp error.

2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)

The CORE-MATH Project

Alexei Sibidanov
University of Victoria
British Columbia, Canada V8W 3P6
sibid@uvic.ca

Paul Zimmermann
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
paul.zimmermann@inria.fr

Stéphane Glondu
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
stephane.glondu@inria.fr

***Abstract*—The CORE-MATH project aims at providing open-source mathematical functions with correct rounding that can be integrated into current mathematical libraries. This article demonstrates the CORE-MATH methodology on two functions:**

libc 2.27 `benchtests` mechanism reports 440,000 cycles for the binary64 pow function in the "768-bit" path.

Another correctly rounded library is CR-LIBM [6], also

Best Paper @ ARITH 22

Paul Zimmermann
(INRIA, France)

MPFR

Issues

- **CORE-MATH.** Claims to have ≤ 0.5 ulp error.

  - Correct: `acos(+7.49...e-01) = +0.72...`
  - CORE-MATH: `acos(+7.49...e-01) = +1.49...` (error > 10^{17} ulps)

  - Correct: `erf(+1.48...e+306) = +1.00...e+00`
  - CORE-MATH: `erf(+1.48...e+306) = +1.48...e+306` (error > 10^{18} ulps)

Even libraries developed by **world experts** have serious errors!

Issues

- **CORE-MATH.** Claims to have ≤ 0.5 ulp error.

  - Correct: `acos(+7.499999...e-01) = +7.227342...e-01`
  - CORE-MATH: `acos(+7.499999...e-01) = +1.494609...e+00` (error > 10^{17} ulps)

  - Correct:
  - CORE-MA … ^{18} ulps)

Why do such correctness issues arise?

Even libraries developed by **world experts** have serious errors!

Intricate Implementations

- **Why.** These implementations are extremely **sophisticated** and **error-prone.**
- **Example.** CORE-MATH implementation of sin.

```c
/* Table containing 128-bit approximations of sin2pi(i/2^11) for 0 <= i <
   (to nearest).
   Each entry is to be interpreted as (hi/2^64+lo/2^128)*2^ex*(-1)*sgn.
   Generated with computeS() from sin.sage. */
static const dint64_t S[256] = {
  {.hi = 0x0, .lo = 0x0, .ex = 128, .sgn=0},
  {.hi = 0xc90fc5f66525d257, .lo = 0x480f7956b6470765, .ex = -8, .sgn=0},
  {.hi = 0xc90f87f3380388d5, .lo = 0xcb3ff35bd4d81baa, .ex = -7, .sgn=0},
  {.hi = 0x96cb587284b81770, .lo = 0xb767005691b9d9d1, .ex = -6, .sgn=0},
  {.hi = 0xc90e8fe6f63c2330, .lo = 0xf1d7d06db39ea9fc, .ex = -6, .sgn=0},
  {.hi = 0xfb514b55ccbe541a, .lo = 0xd784e031f9af76d6, .ex = -6, .sgn=0},
  {.hi = 0x96c9b5df1877e9b5, .lo = 0xf91ee371d6467dca, .ex = -5, .sgn=0},
  {.hi = 0xafea690fd5912ef3, .lo = 0xf56e3c87ae3c56df, .ex = -5, .sgn=0},
  {.hi = 0xc90aafbd1b33efc9, .lo = 0xc539edcbfda0cf2c, .ex = -5, .sgn=0},
  {.hi = 0xe22a7a6729d8e453, .lo = 0x850021e392744a4f, .ex = -5, .sgn=0},
  {.hi = 0xfb49b98e8e7807f6, .lo = 0xb21ccebc9caac3, .ex = -5, .sgn=0},
  {.hi = 0x8a342eda160bf5ae, .lo = 0xde5b1068d174be9c, .ex = -4, .sgn=0},
  {.hi = 0x96c32baca2ae68b4, .lo = 0x37b2dd49d5fca3c0, .ex = -4, .sgn=0},
  {.hi = 0xa351cb7fc30bc889, .lo = 0xb56007d16d4ad5a3, .ex = -4, .sgn=0},
  {.hi = 0xafe00694866a1b44, .lo = 0xcd34d2751c2e1da7, .ex = -4, .sgn=0},
  {.hi = 0xbc6dd52c3a342eb5, .lo = 0xf10bfca3d6464012, .ex = -4, .sgn=0},
  {.hi = 0xc8fb2f886ec09f37, .lo = 0x6a17954b2b7c5171, .ex = -4, .sgn=0},
```

Constant Tables

```c
      {
        int i = (e - 1138 + 63) / 64; // i = ceil((e-1138)/64), 0 <= i
        /* m*T[i] contributes to f = 1139 + 64*i - e bits to frac(x/(2
           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/
      }
```

Bit-Level Operations

Intricate Implementations

- **Why.** These implementations are extremely **sophisticated** and **error-prone.**
- **Example.** CORE-MATH implementation of sin.

```c
static void
evalPSfast (double *h, double *l, double xh, double xl, double uh, do
{
  double t;
  *h = PSfast[4]; // degree 7
  *h = __builtin_fma (*h, uh, PSfast[3]); // degree 5
  *h = __builtin_fma (*h, uh, PSfast[2]); // degree 3
  s_mul (h, l, *h, uh, ul);
  fast_two_sum (h, &t, PSfast[0], *h);
  *l += PSfast[1] + t;
  // multiply by xh+xl
  d_mul (h, l, *h, *l, xh, xl);
}
```

```c
static inline void s_mul (double *hi, double *lo, double a, double bh,
                          double bl) {
  a_mul (hi, lo, a, bh); /* exact */
  *lo = __builtin_fma (a, bl, *lo);
  /* the error is bounded by ulp(lo), where |lo| < |a*bl| + ulp(hi) */
}
```

```c
static inline void a_mul(double *hi, double *lo, double a, double b) {
  *hi = a * b;
  *lo = __builtin_fma (a, b, -*hi);
}
```

Floating-Point Operations

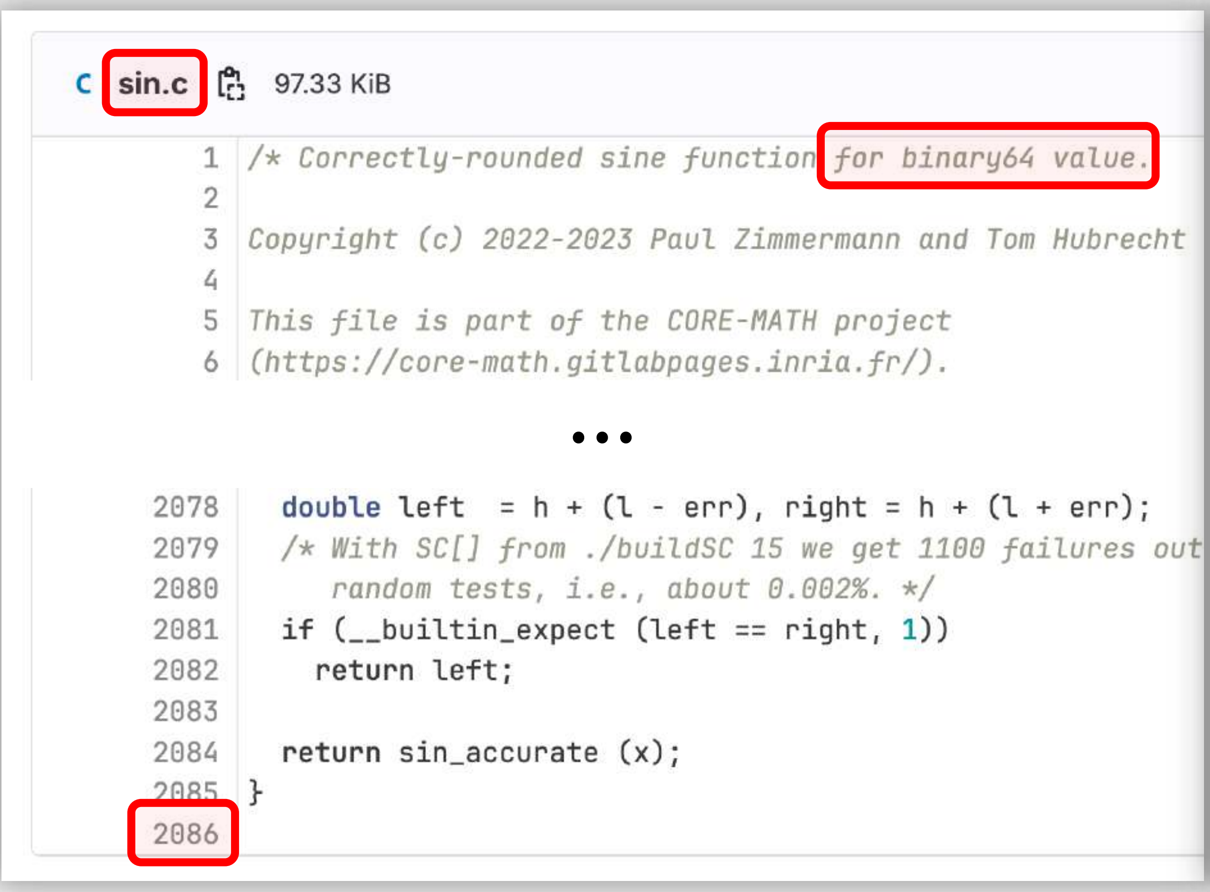
```c
/* Assuming 0x1.7137449123ef6p-26 < x < +Inf,
   return i and set h,l such that i/2^11+h+l approximates frac(x/(2pi)).
   If x <= 0x1.921fb54442d18p+2:
   | i/2^11 + h + l - frac(x/(2pi)) | < 2^-104.116 * |i/2^11 + h + l|
   with |h| < 2^-11 and |l| < 2^-52.36.

   Otherwise only the absolute error is bounded:
   | i/2^11 + h + l - frac(x/(2pi)) | < 2^-75.998
   with 0 <= h < 2^-11 and |l| < 2^-53.

   In both cases we have |l| < 2^-51.64*|i/2^11 + h|.

   Put in err1 a bound for the absolute error:
   | i/2^11 + h + l - frac(x/(2pi)) |.
*/
static int
reduce_fast (double *h, double *l, double x, double *err1)
```

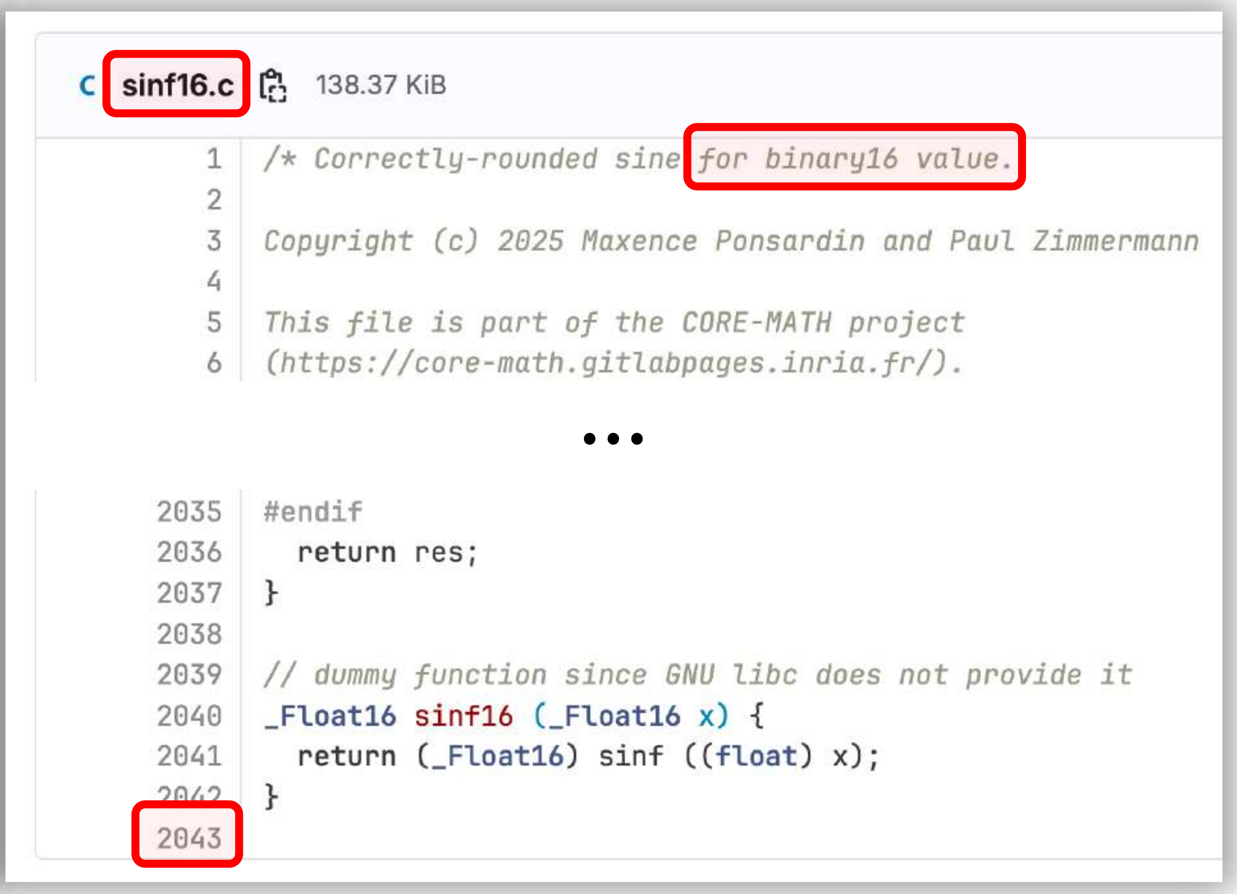
```c
/* Put in h+l an approximation of sin2pi(xh+xl),
   for 2^-24 <= xh+xl < 2^-11 + 2^-24,
   and |xl| < 2^-52.36, with absolute error < 2^-77.09
   (see evalPSfast() in sin.sage).
   Assume uh + ul approximates (xh+xl)^2. */
static void
evalPSfast (double *h, double *l, double xh, double xl, double uh, double
```

Error Analyses

Intricate Implementations

- **Why.** These implementations are extremely **sophisticated** and **error-prone.**
- **Example.** CORE-MATH implementation of sin.





Intricate Implementations

- **Why.** These implementations are extremely **sophisticated** and
- **Example.** CORE-MATH implementation of sin.
    - Implements 43 functions.
    - Supports 5 floating-point formats.

function	float16	binary32	binary64	binary80	binary128
acos	code	code	code glibc patch		
acosh	code	code	code glibc patch		
acospi	code	code glibc patch	code glibc patch		
asin	code	code	code glibc patch		
asinh	code	code	code glibc patch		
asinpi	code	code glibc patch	code glibc patch		
atan	code	code	code glibc patch		
atan2		code	code glibc patch		
atan2pi		code glibc patch	code glibc patch		
atanh	code	code	code glibc patch		
atanpi	code	code glibc patch	code glibc patch		
cbrt	code	code	code (proof) glibc patch	code glibc patch	code
compound		code			
cos	code	code glibc patch	code glibc patch		
cosh	code	code	code glibc patch		
cospi	code	code	code glibc patch		
erf		code	code glibc patch		
erfc		code	code glibc patch		
exp	code	code glibc patch	code glibc patch	code	code
exp10	code	code glibc patch	code glibc patch		
exp10m1		code	code glibc patch		
exp2	code	code glibc patch	code glibc patch	code	
exp2m1		code	code glibc patch		
expm1		code	code glibc patch		
hypot	code	code glibc patch	code glibc patch	code glibc patch	code
lgamma		code	code glibc patch		
log	code	code glibc patch	code (with Gappa proof) glibc patch		
log10	code	code	code glibc patch		
log10p1		code glibc patch	code glibc patch		
log1p		code	code glibc patch		
log2	code	code glibc patch	code glibc patch	code	
log2p1		code	code glibc patch		
pow	code	code	code glibc patch	code	
rsqrt	code	code glibc patch	code glibc patch	code glibc patch	code
sin	code	code glibc patch	code glibc patch		
sincos		code glibc patch	code glibc patch		
sinh	code	code	code glibc patch		
sinpi	code	code glibc patch	code glibc patch		
sqrt	code				code
tan	code	code	code glibc patch		
tanh	code	code	code glibc patch		
tanpi	code	code glibc patch	code glibc patch		
tgamma		code	code glibc patch		

Intricate Implementations

- **Why.** These implementations are extremely **sophisticated** and
- **Example.** CORE-MATH implementation of sin.
  - Implements 43 functions.
  - Supports 5 floating-point formats.

function	float16	binary32	binary64	binary80	binary128
acos	code	code	code glibc patch		
acosh	code	code	code glibc patch		
acospi	code	code glibc patch	code glibc patch		
asin	code	code	code glibc patch		
asinh	code	code	code glibc patch		
asinpi	code	code glibc patch	code glibc patch		
atan	code	code	code glibc patch		
atan2		code	code glibc patch		
atan2pi		code glibc patch	code glibc patch		
atanh	code	code	code glibc patch		
atanpi	code	code glibc patch	code glibc patch		
cbrt	code	code	code (proof) glibc patch	code glibc patch	code
compound		code			
cos	code	code glibc patch	code glibc patch		
cosh	code	code	code glibc patch		
cospi	code	code	code glibc patch		
erf		code	code glibc patch		
lgamma		code	code glibc patch		
log	code	code glibc patch	code (with Gappa proof) glibc patch		
log10	code	code	code glibc patch		
log10p1		code glibc patch	code glibc patch		
log1p		code	code glibc patch		
log2	code	code glibc patch	code glibc patch	code	
log2p1		code	code glibc patch		
pow	code	code	code glibc patch	code	
rsqrt	code	code glibc patch	code glibc patch	code glibc patch	code
sin	code	code glibc patch	code glibc patch		
sincos		code glibc patch	code glibc patch		
sinh	code	code	code glibc patch		
sinpi	code	code glibc patch	code glibc patch		
sqrt	code				code
tan	code	code	code glibc patch		
tanh	code	code	code glibc patch		
tanpi	code	code glibc patch	code glibc patch		
tgamma		code	code glibc patch		

How to **ensure the correctness** of existing libraries?

Research Directions

- **Case 1.** Input is ≤ 32 bits (and univariate).
    - **Exhaustive testing.** Compute $\max_{x \in X} \mathrm{err}_{\mathrm{ulp}}\big(f(x), P(x)\big)$.

Research Directions

- **Case 1.** Input is ≤ 32 bits (and univariate).
  - **Exhaustive testing.** Compute $\max_{x \in X} \mathrm{err}_{\mathrm{ulp}}\big(f(x), P(x)\big)$.
  - **MPFR library.** Used to compute $f(x)$.

High Performance Correctly Rounded Math Libraries for 32-bit Floating Point Representations

Jay P. Lim
Department of Computer Science
Rutgers University
United States

Santosh Nagarakatte
Department of Computer Science
Rutgers University
United States

RLibm
[POPL 21/22, PLDI 21/22/24/25, Dist. Paper x2]

The CORE-MATH Project

Alexei Sibidanov
University of Victoria
British Columbia, Canada V8W 3P6
sibid@uvic.ca

Paul Zimmermann
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
paul.zimmermann@inria.fr

Stéphane Glondu
Université de Lorraine
CNRS, Inria, LORIA
F-54000 Nancy, France
stephane.glondu@inria.fr

CORE-MATH
[ARITH 22/23/25, Best Paper]

. . .

Research Directions

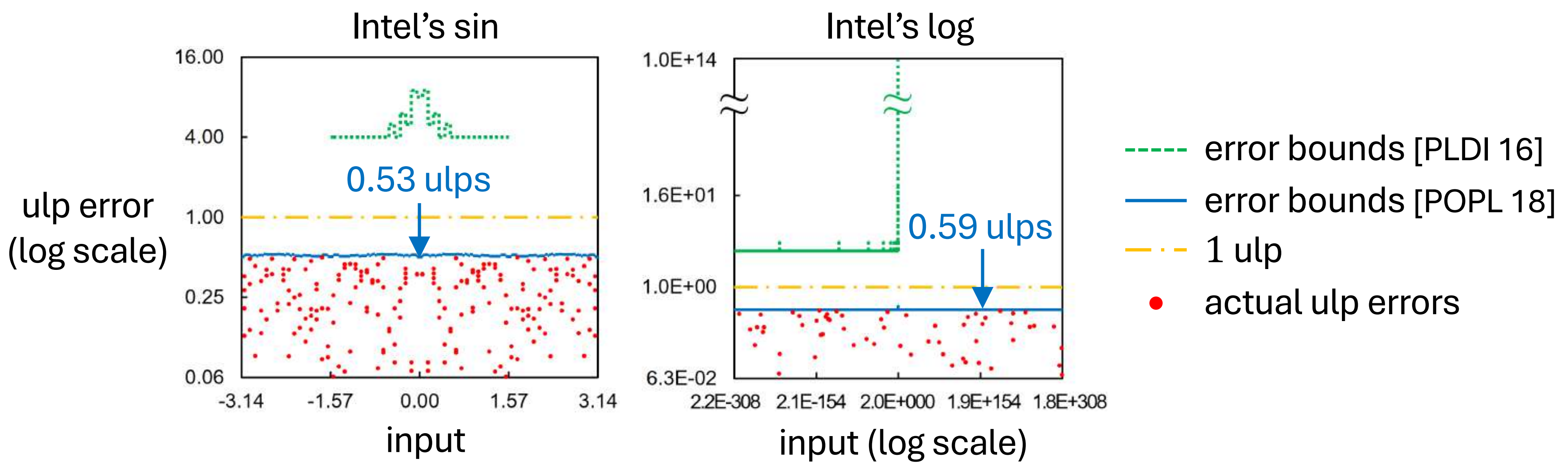
- **Case 1.** Input is ≤ 32 bits (and univariate).
  - **Exhaustive testing.** Compute $\max_{x \in X} \mathrm{err}_{\mathrm{ulp}}\big(f(x), P(x)\big)$.
  - **MPFR library.** Used to compute $f(x)$.
  - **Limitations.** Cannot trust MPFR. Cannot apply to new functions.
  - **Why.** Complicated implementation. Implements only basic functions.

Fixed bugs, with patches:

5. With some mparam.h files, the mpfr_div function can return an incorrect result. This is fixed by the divhigh-basecase patch, which also provides a testcase. Note that this bug is new in MPFR 3.1 and cannot be triggered with the mparam.h files distributed in the tarball. Thus most users should not be affected. However this bug may be visible after a "make tune" (which generates a new mparam.h file). More details in the discussion in the MPFR list.
   Corresponding changeset in the 3.1 branch: da9ac8ba (r9711).
6. The Bessel functions (mpfr_j0, mpfr_j1, mpfr_jn, mpfr_y0, mpfr_y1, mpfr_yn) can return an incorrect result. This is fixed by the jn patch, which also provides a testcase. Bug report by Fredrik Johansson.
   Corresponding changeset in the 3.1 branch: d1617da2 (r9845).
7. The Riemann Zeta function mpfr_zeta can return an incorrect result when the argument is near an even negative integer. This is fixed by the zeta patch, which also provides a testcase. Bug report by Fredrik Johansson.

Research Directions

- **Case 2.** Input is ≥ 64 bits (or multivariate).
  - **Exhaustive testing.** Infeasible (since 2^{64} is too large).
  - **Program analysis.** My previous work [POPL 18, PLDI 16].



Research Directions

- **Case 2.** Input is ≥ 64 bits (or multivariate).
  - **Exhaustive testing.** Infeasible (since 2^{64} is too large).
  - **Program analysis.** My previous work [POPL 18, PLDI 16].

  - **Limitations.** High computational cost (19 days for `log`), etc.
  - **Why.** Lack of proper abstraction in existing implementations.

```
// if i >= 2^10: 1/2 <= frac(x/(2pi)) < 1 thus pi <= x <= 2pi
// we use sin(pi+x) = -sin(x)
neg = neg ^ (i >> 10);
i = i & 0x3ff;
// | i/2^11 + h + l - frac(x/(2pi)) | mod 1/2 < err1

// now i < 2^10
// if i >= 2^9: 1/4 <= frac(x/(2pi)) < 1/2 thus pi/2 <= x <= pi
// we use sin(pi/2+x) = cos(x)
is_sin = is_sin ^ (i >> 9);
i = i & 0x1ff;
// | i/2^11 + h + l - frac(x/(2pi)) | mod 1/4 < err1
```

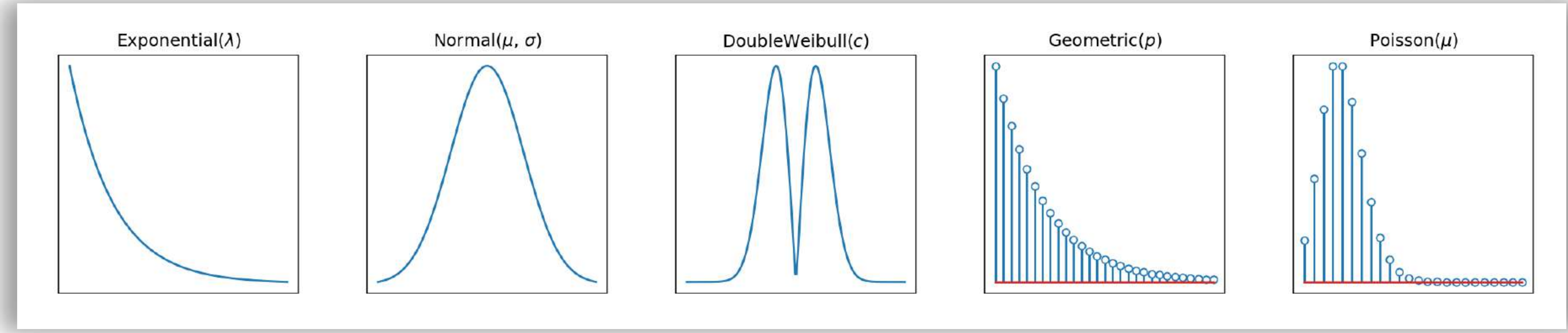
```
/* Table generated with ./buildSC 15 using accompanying buildSC.c.
   For each i, 0 <= i < 256, xi=i/2^11+SC[i][0], with
   SC[i][1] and SC[i][2] approximating sin2pi(xi) and cos2pi(xi)
   respectively, both with 53+15 bits of accuracy. */
static const double SC[256][3] = {
  {0x0p+0, 0x0p+0, 0x1p+0}, /* 0 */
  {-0x1.c0f6cp-35, 0x1.921f892b900fep-9, 0x1.ffff621623fap-1}, /* 1 */
  {-0x1.9c7935ep-35, 0x1.921f0ea27ce01p-8, 0x1.fffd8858eca2ep-1}, /* 2 */
  {-0x1.d14d1acp-34, 0x1.2d96af779b0bbp-7, 0x1.fffa72c986392p-1}, /* 3 */
  {-0x1.dba8f6a8p-33, 0x1.921d1ce2d0a1cp-7, 0x1.fff62169dddaap-1}, /* 4 */
  {0x1.a6b7cdfp-32, 0x1.f6a29bdb7377p-7, 0x1.fff0943c02419p-1}, /* 5 */
  {0x1.b49618dp-33, 0x1.2d936d1506f3dp-6, 0x1.ffe9cb44829cp-1}, /* 6 */
```

These are essentially assembly code. They need civilization!

Sample Generation

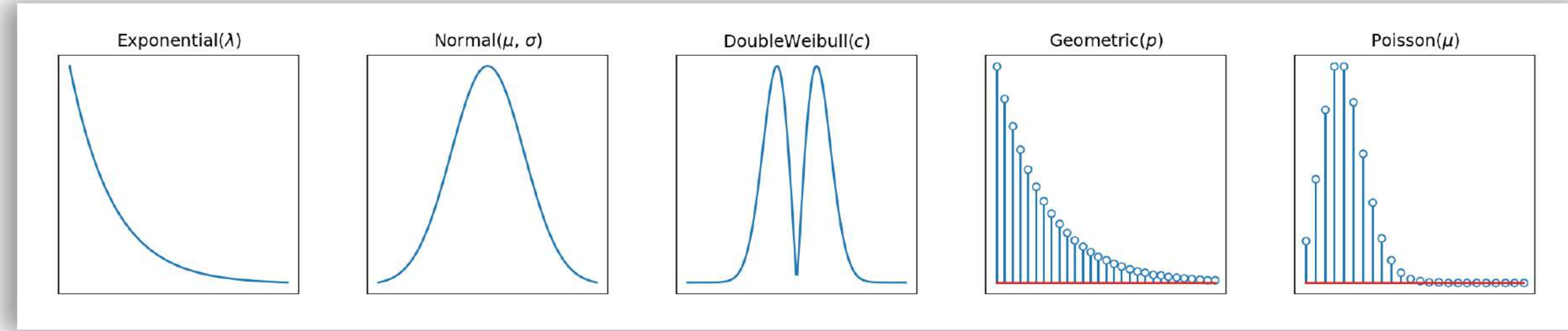
Problem

- **Goal.** Let $\mathcal{D} \in \{\text{Exponential}(\mu), \text{Normal}(\mu, \sigma), \ldots\}$ be a probability distribution.
  - Generate random variates $X \sim \mathcal{D}$.



Problem

- **Goal.** Let $\mathcal{D} \in \{\text{Exponential}(\mu), \text{Normal}(\mu, \sigma), \ldots\}$ be a probability distribution.
  - Generate random variates $X \sim \mathcal{D}$.
  - Compute cumulative probabilities $F(x) \coloneqq \Pr(X \leq x)$ for $x \in \mathbb{R}$.
  - Compute quantiles $Q(u) \coloneqq \inf\{x \mid u \leq F(x)\}$ for $u \in [0,1]$.
  - Compute probabilit densities (if exist) $f(x) \coloneqq d\mathcal{D}/d\lambda$ for $x \in \mathbb{R}$.
  - …



Existing Solution

- **Libraries for Probability Distributions.**
  - C GNU Scientific Library, …
  - C++ Standard Library, Boost, …
  - Python NumPy, SciPy, PyTorch, …
  - Julia Distributions.jl, …

Existing Solution

- **Libraries for Probability Distributions.**
  - C — GNU Scientific Library, …
  - C++ — Standard Library, Boost, …
  - Python — NumPy, SciPy, PyTorch, …
  - Julia — Distributions.jl, …

The Exponential Distribution

$X \sim \mathcal{D}$

`double gsl_ran_exponential(const gsl_rng *r, double mu)`

This function returns a random variate from the exponential distribution with mean `mu`. The distribution is,

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx$$

for $x \geq 0$.

$F(x)$

`double gsl_cdf_exponential_P(double x, double mu)`

`double gsl_cdf_exponential_Q(double x, double mu)`

$Q(u)$

`double gsl_cdf_exponential_Pinv(double P, double mu)`

`double gsl_cdf_exponential_Qinv(double Q, double mu)`

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the exponential distribution with mean `mu`.

Issues

- **Issue 1.** These functions **cannot be exact** due to "double".
  - Even worse, their properties are barely known.
  - E.g., support, approximation error, … are unknown.

The Exponential Distribution

$\hat{X} \sim \hat{\mathcal{D}}$ — **double gsl_ran_exponential(const gsl_rng *r, double mu)**

This function returns a random variate from the exponential distribution with mean `mu`. The distribution is,

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx$$

for $x \geq 0$.

$\hat{F}(x)$ — **double gsl_cdf_exponential_P(double x, double mu)**

double gsl_cdf_exponential_Q(double x, double mu)

$\hat{Q}(u)$ — **double gsl_cdf_exponential_Pinv(double P, double mu)**

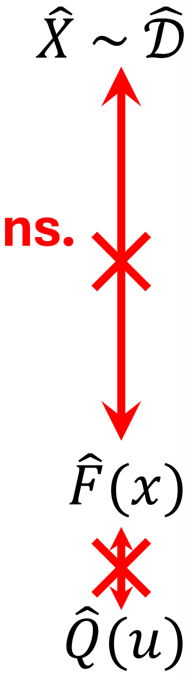
double gsl_cdf_exponential_Qinv(double Q, double mu)

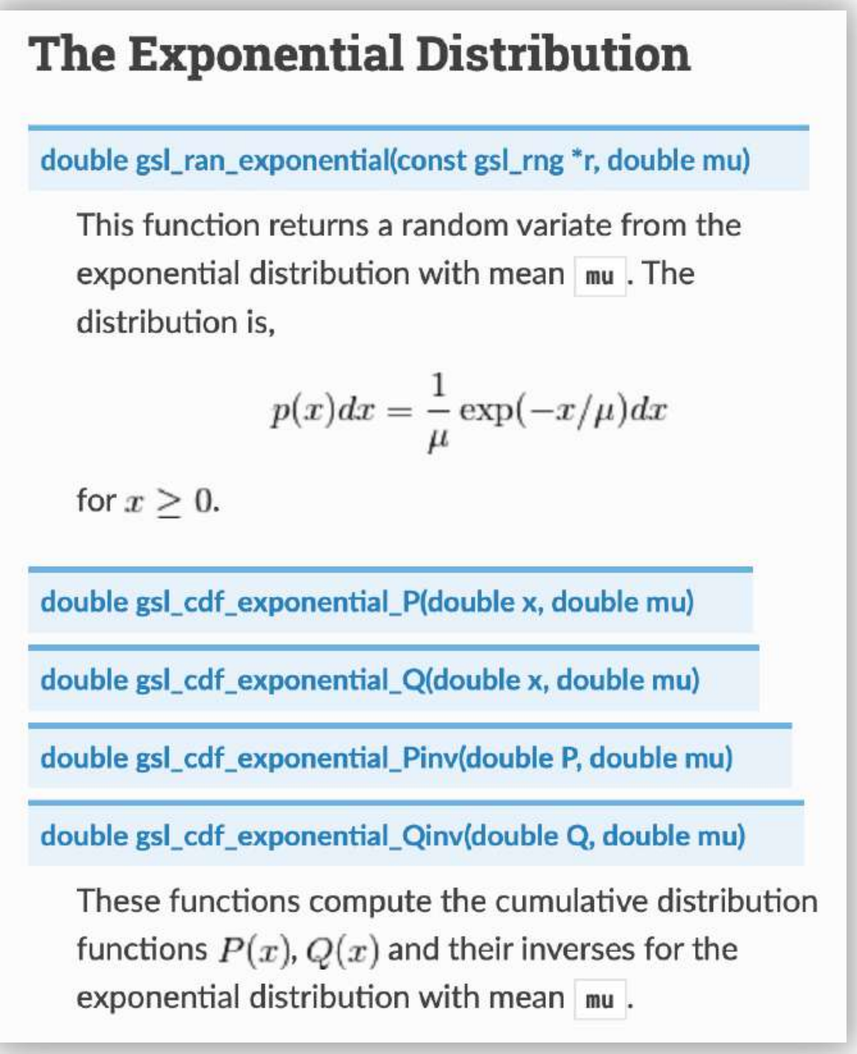
These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the exponential distribution with mean `mu`.

Issues

- **Issue 1.** These functions **cannot be exact** due to “double”.
  - Even worse, their properties are barely known.
  - E.g., support, approximation error, … are unknown.

- **Issue 2.** These functions represent **different distributions.**
  - RV: $\hat{X}$ can be at most $\approx 22.2.$
  - CDF: $\hat{F}$ becomes 1 at $\approx 17.3.$
  - QF: $\hat{Q}$ takes $\hat{Q}(1) \approx 16.6.$





The Exponential Distribution

double gsl_ran_exponential(const gsl_rng *r, double mu)

This function returns a random variate from the exponential distribution with mean `mu`. The distribution is,

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx$$

for $x \geq 0$.

double gsl_cdf_exponential_P(double x, double mu)

double gsl_cdf_exponential_Q(double x, double mu)

double gsl_cdf_exponential_Pinv(double P, double mu)

double gsl_cdf_exponential_Qinv(double Q, double mu)

These functions compute the cumulative distribution functions $P(x)$, $Q(x)$ and their inverses for the exponential distribution with mean `mu`.

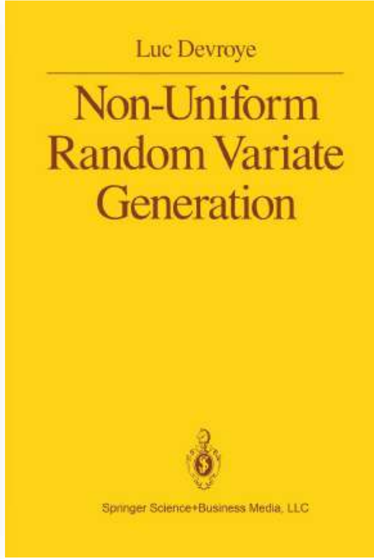
Main Culprit

- **Theory.** Underlying algorithms assume the **Real-RAM model.**
- **Practice.** Actual implementations simply use **floating point.**

Assumption 1. Our computer can store and manipulate real numbers.

Assumption 2. There exists a perfect uniform [0,1] random variate generator, i.e. a generator capable of producing a sequence $U_1, U_2, \ldots$ of independent random variables with a uniform distribution on [0,1].

Assumption 3. The fundamental operations in our computer include addition, multiplication, division, compare, truncate, move, generate a uniform random variate, exp, log, square root, arc tan, sin and cos. (This implies that each of these operations takes one unit of time regardless of the size of the operand(s). Also, the outcomes of the operations are real numbers.)



1986

Luc Devroye
(McGill U, Canada)

Main Culprit

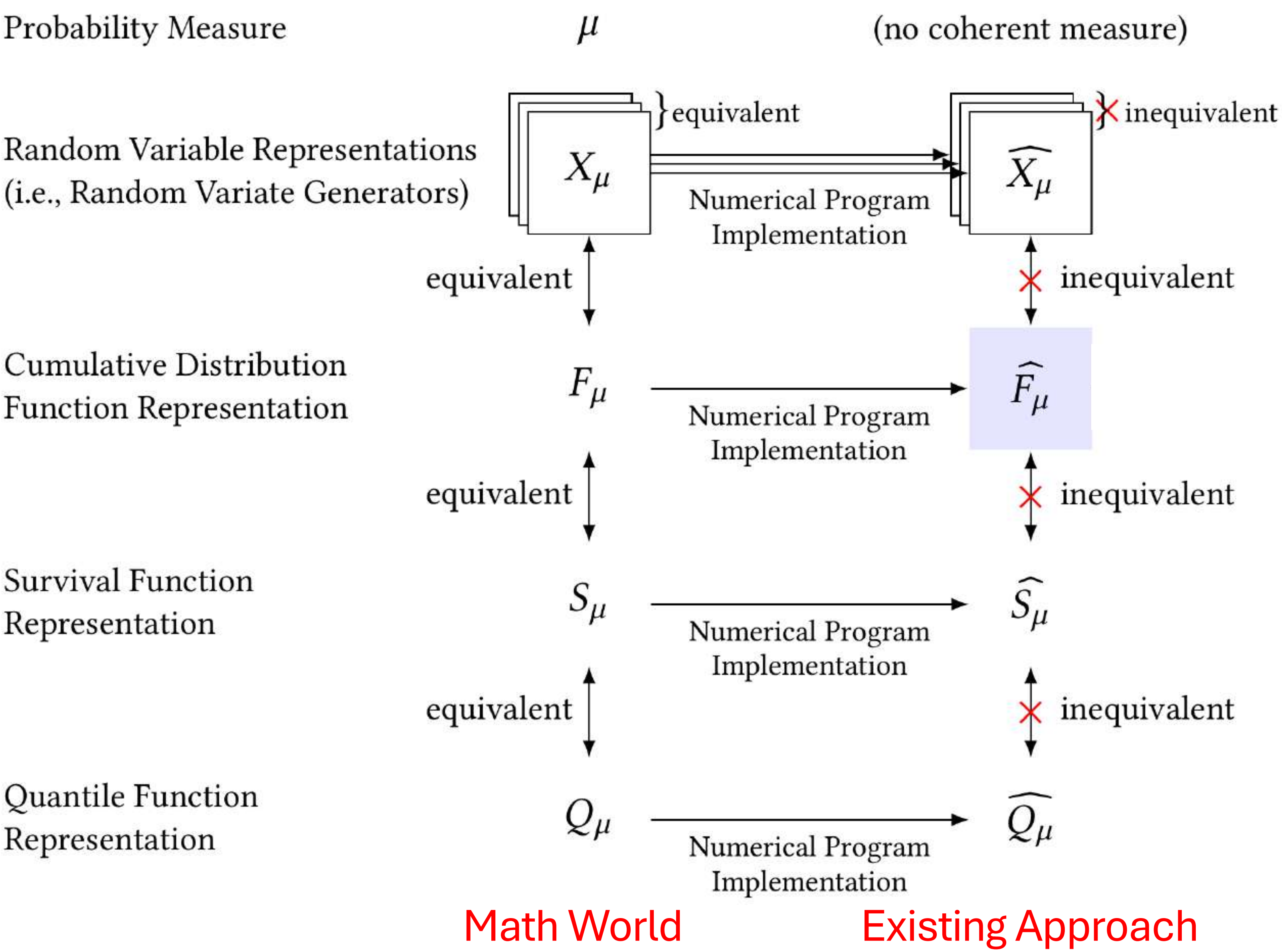
- **Theory.** Underlying algorithms assume the **Real-RAM model.**
- **Practice.** Actual implementations simply use **floating point.**

NumPy	BUG: random: Problems with hypergeometric with ridiculously large arguments	https://github.com/numpy/numpy/issues/11443
NumPy	Possible bug in random.laplace	https://github.com/numpy/numpy/issues/13361
NumPy	Bias of random.integers() with int8 dtype	https://github.com/numpy/numpy/issues/14774
NumPy	Geometric, negative binomial and poisson fail for extreme arguments	https://github.com/numpy/numpy/issues/1494
NumPy	numpy.random.hypergeometric: error for some cases	https://github.com/numpy/numpy/issues/1519
NumPy	numpy.random.logseries - incorrect convergence for k=1, k=2	https://github.com/numpy/numpy/issues/1521
NumPy	Von Mises draws not between -pi and pi [patch]	https://github.com/numpy/numpy/issues/1584
NumPy	Negative binomial sampling bug when p=0	https://github.com/numpy/numpy/issues/15913
NumPy	default_rng.integers(2**32) always return 0	https://github.com/numpy/numpy/issues/16066
NumPy	Beta random number generator can produce values outside its domain	https://github.com/numpy/numpy/issues/16230
NumPy	OverflowError for np.random.RandomState()	https://github.com/numpy/numpy/issues/16695
NumPy	binomial can return unitialized integers when size is passed with array values for a or p	https://github.com/numpy/numpy/issues/16833
NumPy	np.random.geometric(10**-20) returns negative values	https://github.com/numpy/numpy/issues/17007
NumPy	numpy.random.vonmises() fails for kappa > 10^8	https://github.com/numpy/numpy/issues/17275

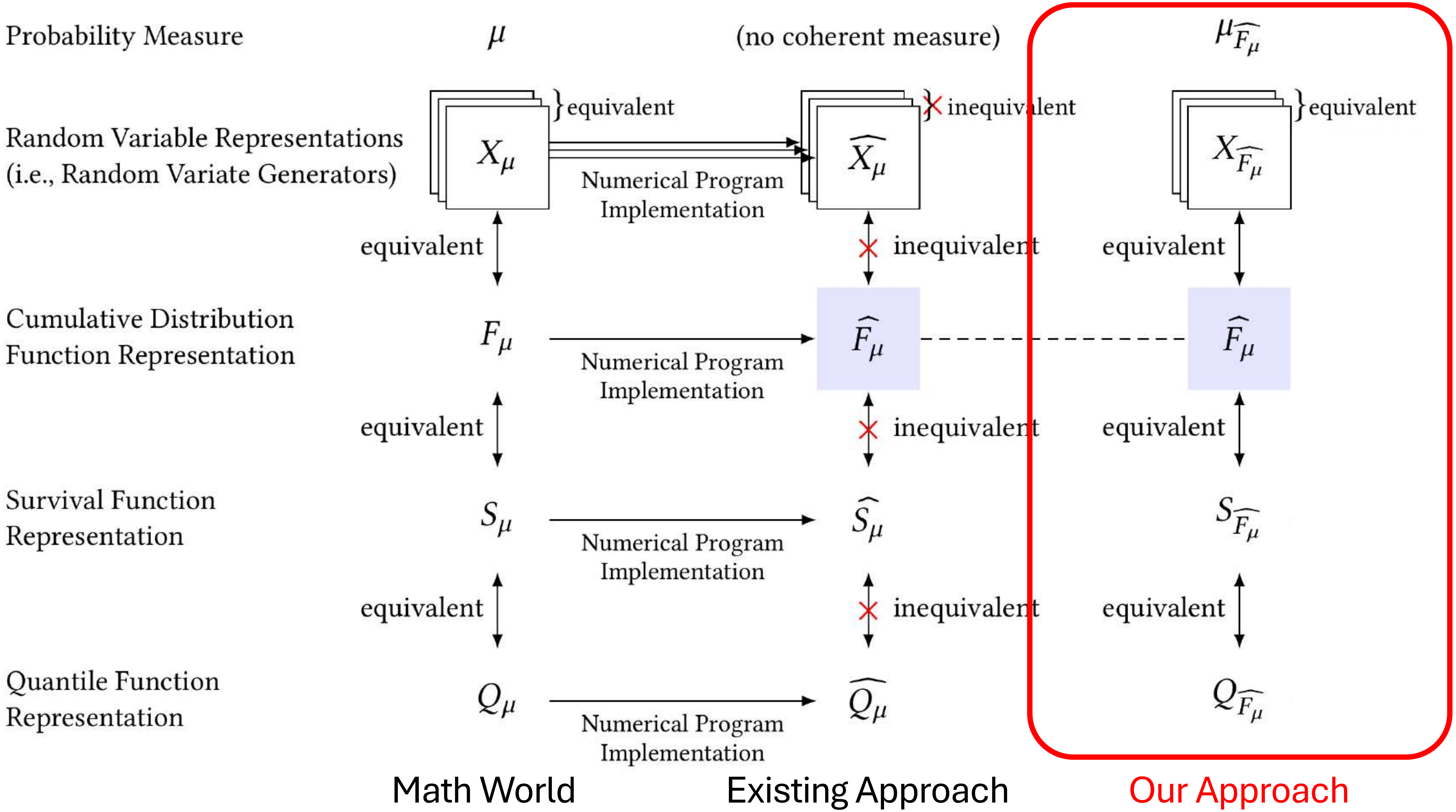
NumPy	np.random.geometric(10**-20) returns negative values
PyTorch	CPU torch.exponential_ function may generate 0 which can cause downstream NaN
PyTorch	Hang: sampling VonMises distribution gets stuck in rejection sampling for small kappa

NumPy	BUG: numpy.random.Generator.dirichlet should accept zeros.	https://github.com/numpy/numpy/issues/22547
NumPy	numpy.random.randint(-2147483648, 2147483647) raises ValueError: low >= high	https://github.com/numpy/numpy/issues/2286
NumPy	BUG: random: beta (and therefore dirichlet) hangs when the parameters are very small	https://github.com/numpy/numpy/issues/24203
NumPy	BUG: random: dirichlet(alpha) can return nans in some cases	https://github.com/numpy/numpy/issues/24210
NumPy	BUG: random: beta can generate nan when the parameters are extremely small	https://github.com/numpy/numpy/issues/24266
NumPy	BUG: Inaccurate left tail of random.Generator.dirichlet at small alpha	https://github.com/numpy/numpy/issues/24475
NumPy	Cannot generate random variates from noncentral chi-square distribution with dof = 1	https://github.com/numpy/numpy/issues/5766
NumPy	Bug in np.random.dirichlet for small alpha parameters	https://github.com/numpy/numpy/issues/5851
NumPy	numpy.random.poisson(0) should return 0	https://github.com/numpy/numpy/issues/827
NumPy	Could random.hypergeometric() be made to match behavior of random.binomial() when sample or n = 0?	https://github.com/numpy/numpy/issues/9237
NumPy	BUG: np.random.zipf hangs the interpreter on pathological input	https://github.com/numpy/numpy/issues/9829
PyTorch	torch.distributions.categorical.Categorical samples indices with zero probability	https://github.com/pytorch/pytorch/issues/100884
PyTorch	Torch randperm with device mps does not sample exactly uniformly from all possible permutations	https://github.com/pytorch/pytorch/issues/104315
PyTorch	torch.distributions.Pareto.sample sometimes gives inf	https://github.com/pytorch/pytorch/issues/107821
PyTorch	torch.multinomial - Unexpected (incorrect) results when replacement=True in version 2.1.1+cpu	https://github.com/pytorch/pytorch/issues/114945

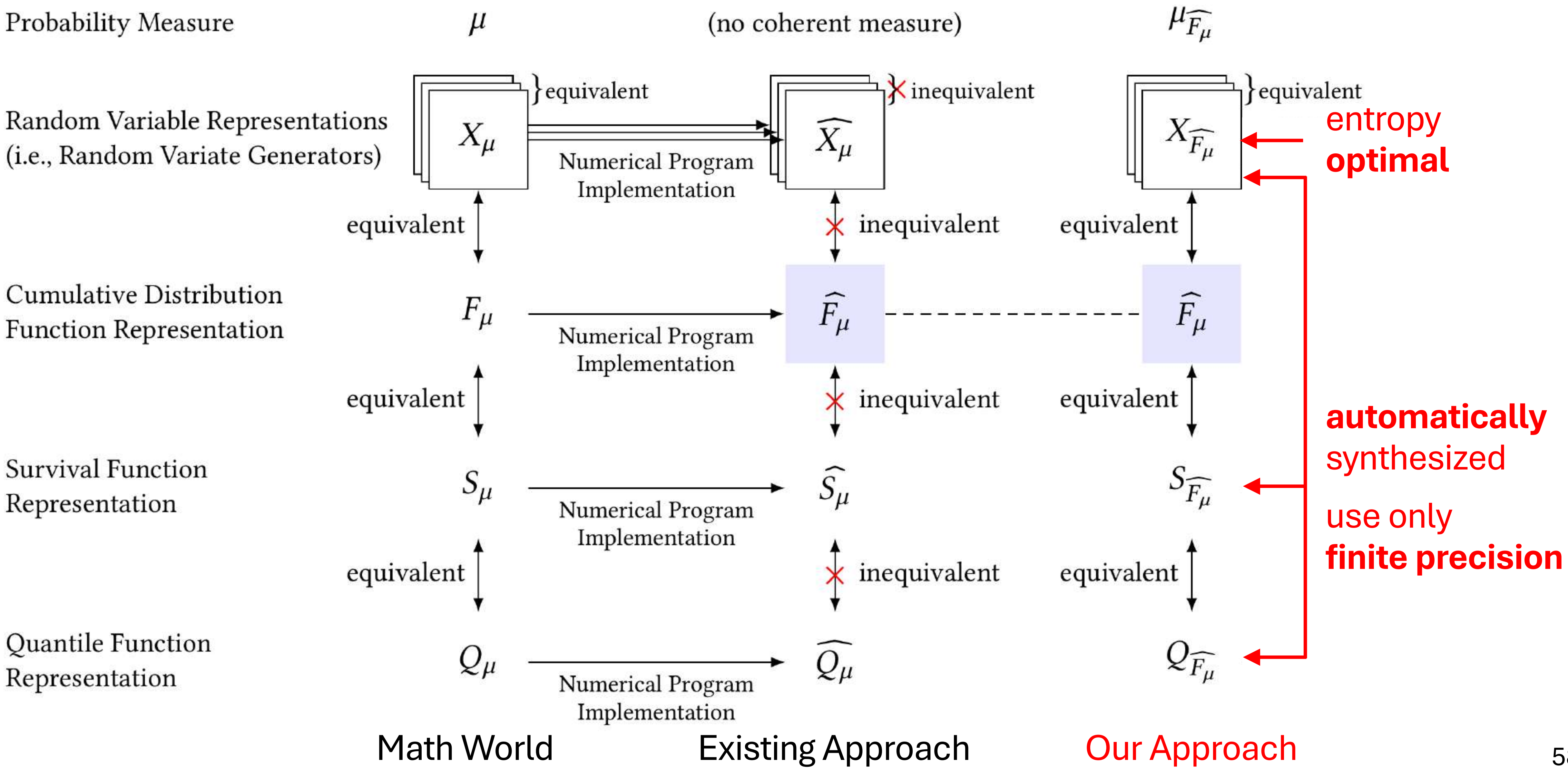
Our Work (PLDI 25)



Our Work (PLDI 25)



Our Work (PLDI 25)



Conclusion

Our Works: Correctness, Efficiency, Fundamental Limits

Function Evaluation	Use **floats** intricately.	[Ongoing 1] [Ongoing 2] [POPL 18] [PLDI 16]	
Sample Generation	Assume **reals.**	[Ongoing 1] [Ongoing 2] [PLDI 25a]	
Differentiation	Assume **differentiability.**		[ICLR 24] (Spotlight) [ICML 23] [NeurIPS 20] (Spotlight)
Integration **(≈ Probabilistic Inference)**	Assume **integrability.**	[Submitted] [PLDI 25b] [POPL 23] [POPL 20]	[AAAI 20] [NeurIPS 18]
Function Approximation		[CAV 25]	[ICML 25] [Neural Networks 24]

Our Works: Correctness, Efficiency, Fundamental Limits

Function Evaluation	Use **floats** intricately.	[Ongoing 1] [Ongoing 2] [POPL 18] [PLDI 16]	
Sample Generation	Assume **reals.**	[Ongoing 1] [Ongoing 2] [PLDI 25a]	
Differentiation	Derivatives of functions that are **non-smooth?**		[ICLR 24] (Spotlight) [ICML 23] [NeurIPS 20] (Spotlight)
Integration **(≈ Probabilistic Inference)**	Integrals of functions that are **diverging?**	[Submitted] [PLDI 25b] [POPL 23] [POPL 20]	[AAAI 20] [NeurIPS 18]
Function Approximation	Universal approximation theorem **over floats?**	[CAV 25]	[ICML 25] [Neural Networks 24]

High-Level Messages

- Continuous computations have been actively studied for nearly a century.
- Despite these efforts, many such computations still lack rigorous foundations.
- PL approaches would be crucial in establishing solid foundations of practical computations.
- If you are interested, please feel free to contact me!