Cross-Platform Floating-Point Determinism Out of the Box

SHERRY IGNATCHENKO ET AL

Authors_

All the authors are from Six Impossible Things Before Breakfast Limited

Sherry Ignatchenko

🇦🇹 si@6it.dev

Overall idea, and Insisting that it is doable, in spite of setbacks

Victor Istomin

🇺🇦 vi@6it.dev

Wrestling compilers, implementing and testing various sixit::dmath IEEE floats: shared-lib, static-lib, soft-float, and inline-asm

Serhii Iliukhin

🇺🇦 sil@6it.dev

Templatizing math and geometry libs and tests

Guy Davidson

🇬🇧 gd@6it.dev

Advising on C++ standard with regards to floating-point, idea about intrinsics, reviews

Dmytro Ivanchykhin

🇺🇦 di@6it.dev

Advising on math aspects, code organization and refactoring, reviews

Mykhailo Borovyk

🇺🇦 mbo@6it.dev

Implementing fixed-point math with floating-point fallback

Vladyslav Merais

🇺🇦 vmer@6it.dev

Implementing support for RISC-V, including inline-asm

sixit:: × / − +

Talk Outline_

1. Task Definition. Why Determinism?
2. Pre-Existing Work. What Went Wrong?
3. Our First Attempt. Failing and Learning
4. Working Versions. 6 of Them
5. Ongoing Battle For Performance
6. Conclusions and C++ Standard Proposal

We are Here

10 min

15 min

20 min

30 min

40 min

45 min

T. A. L. K T. I. M E

What Are We Speaking About?_

1. Same executable behaves exactly the same given exactly the same inputs

(Sort of) ✅

2. Executables built from the same source but on different platforms behave exactly the same given exactly the same inputs

❌

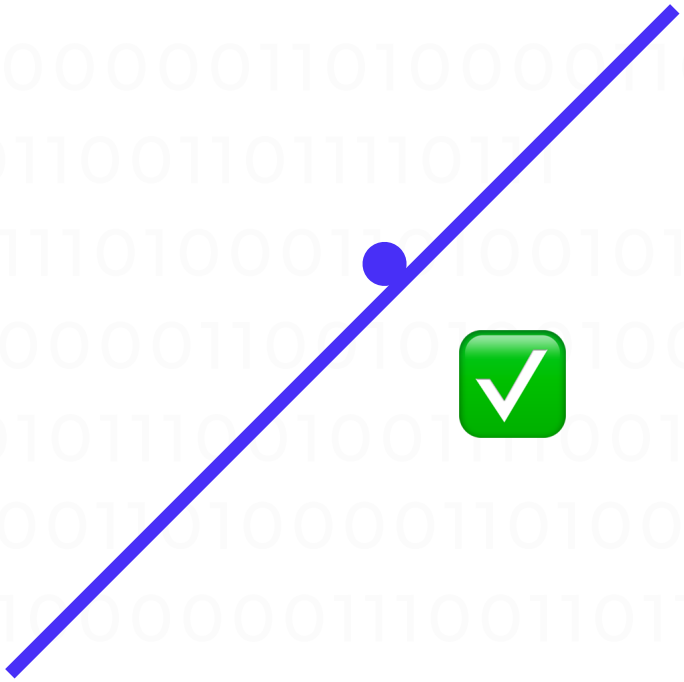
3. The same function behaves exactly the same given exactly the same inputs, in different contexts of the same executable

❌

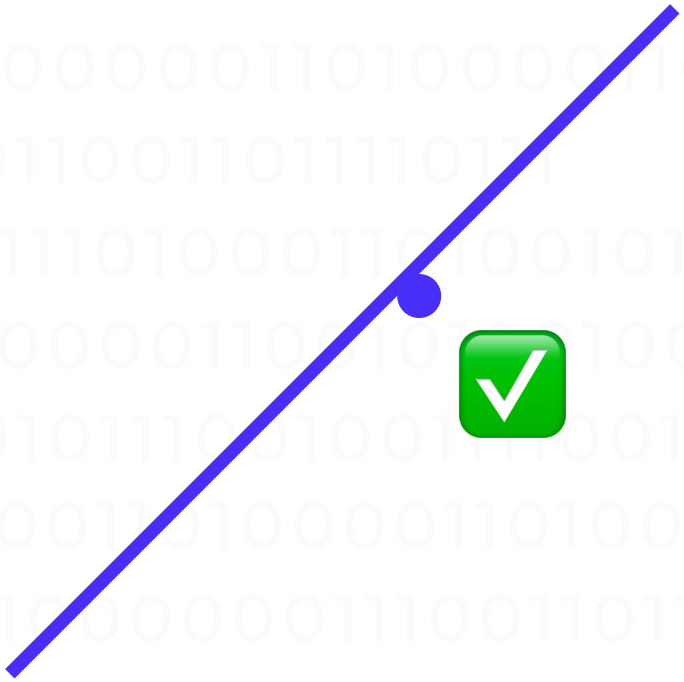
Why FP Determinism is Important?_

1. Same Executable
   - General Sanity, Testability
   - (Sort of) ✅
2. Cross Platform
   - Distributed Simulations, Games, especially RTS
   - ❌
3. Different Contexts
   - Algorithm correctness. Avoiding catastrophic cancellations. And so on
   - ❌

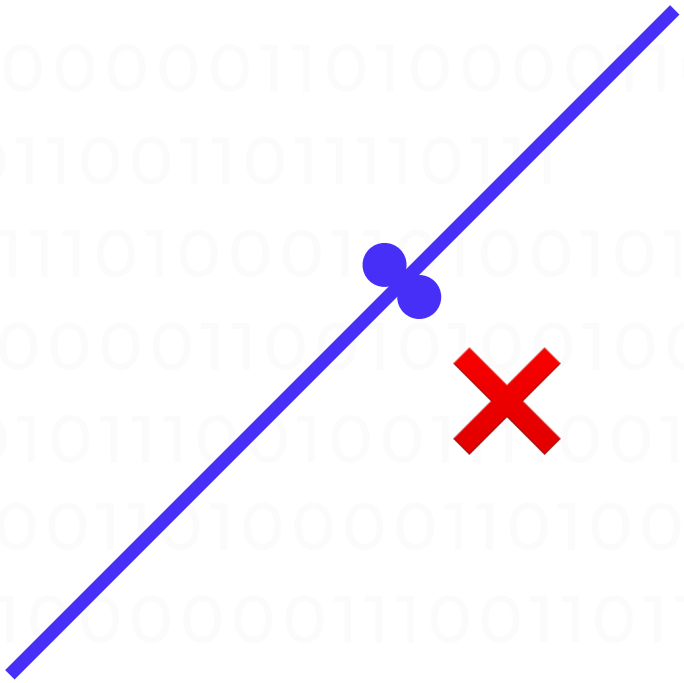
On Algorithm Correctness_



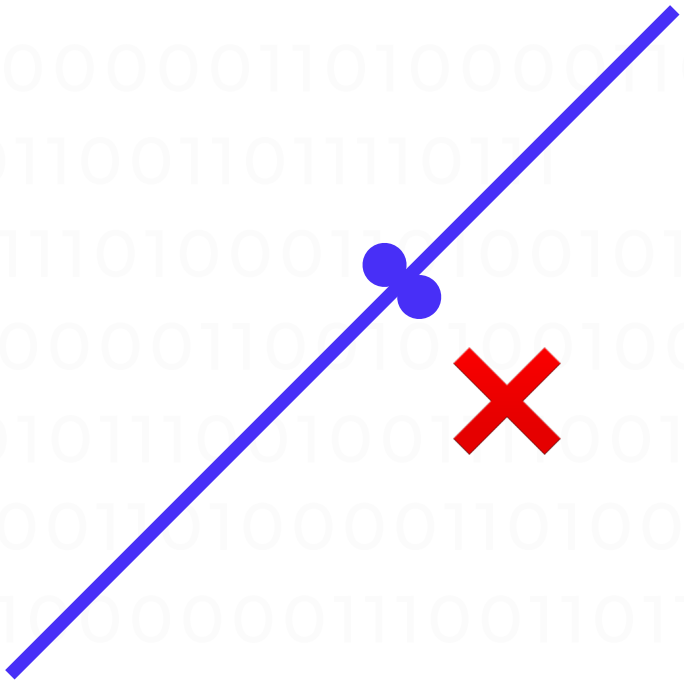
On Algorithm Correctness_



On Algorithm Correctness_



On Algorithm Correctness_



There are algorithms, which are:

1. Mathematically ✅
2. Fixed-Point ✅
3. Floating-Point ❌

Can be usually made solid - but only within deterministic fp

Earlier Attempts: FAILED (1/2)_

"It is incredibly naive to write arbitrary floating point code in C or C++ and expect it to give exactly the same result across different compilers or architectures, or even the same results across debug and release builds" - Glenn Fiedler

"If you store replays as controller inputs, they cannot be played back on machines with different CPU architectures, compilers, or optimization settings. In MotoGP, this meant we could not share saved replays between Xbox and PC." - Shawn Hargreaves, MSDN Blog

"Experiments are usually reproducible only on the same machine with the same system library and the same compiler using the same options." - STREFLOP Library

Earlier Attempts:FAILED (2/2)_

"A deterministic game will only be deterministic when using the identically compiled files and run on systems that adhere to the IEEE standards. Cross platform synchronized network simulations or replays will not possible [sic]." - Most upvoted answer on StackOverflow (by AttackingHobo)

"If we are talking practicabilities [sic], then things are very different, and expecting repeatable results in real programs is crying for the moon" - Nick Maclaren on IEEE 754 mailing list

MOST OPTIMISTIC:

"...with a good deal of work you may be able to coax exactly the same floating point results out of different compilers or different machine architectures" - Glenn Fiedler

What are We Trying to Achieve_

1. Out of the Box Solution. It should Just Work
2. Try To Be Consistent Across All Modern CPUs. No Discontinued or x86/x87 though
3. Support At Least The 3 Major C++ Compilers
4. Mix&Match: Combining Deterministic and !Deterministic in One Executable
5. Different Performance-vs-Guarantees Options
6. Extensive Testing

Sources of FP non-determinism_

1. CPU
2. Compiler
3. <math.h>

Handling FP non-determinism_

1. CPU
   - Modern CPUs and even MCUs do provide IEEE754-compliant instructions at least for + - * / .
2. Compiler
   - Tries to use non-compliant instructions, but there are several ways to handle it. Some are even compiler-flags-neutral
3. <math.h>
   - Just need to have our own deterministic math lib, using only + - * / . Anything less will fail sooner rather than later ☹

Our Take 1: Premise_

1. Adapting Existing Lib to become Deterministic
2. Assumption: ‘;’ guarantees sequencing
3. Idea: if we place enough ‘;’, the code will become deterministic

   Like rewriting ambiguous `x = a+b+c;` into unambiguous `auto tmp=a+b; x=tmp+c;`
4. Then we could write static analyzer to find potential non-determinism
5. Bingo - or maybe not?!

Our Take 1: FAILED_

Assumption is the mother of all screw-ups
— proverb

2 Assumption: ‘;’ prohibits reordering around it

↓

Reality: FMA over ‘;’ under 2 out of 3 major compilers

↓

Analysis revealed that even within WG21 there is no obvious consensus about the standard in this regard 😱

Our Take 1: Takeaways_

1. Adapting Existing Lib to become Deterministic ✅
2. Our Take 1: CPU FP Determinism: ✅
   Lib FP Determinism: ✅ Compiler FP Determinism ❌
3. BOTH reordering AND FMAs are Big Problems 😱
4. Bug Chasing is a Bad Idea™ ➡ Need to Separate Determinism from Math
5. Performance Hit is Significant

Hence, We Need to Allow Mix&Match
Deterministic and Non-Deterministic Code

Templates to the Rescue!_

…one thing in common too
It's a, it's a, it's a, it's a sin()!
— almost Pet Shop Boys

```
template<typename fp>
fp sixit::dmath::sin(fp x) {
  //adapted from MUSL sinf()
  using tr = sixit::dmath::fp_traits<fp>;

  uint32_t ix = tr::to_ieee_uint(x);
  bool sign = ix >> 31;
  ix &= 0x7fffffff;

  if (ix <= 0x3f490fda) {  /* |x| ~<= pi/4 */
      if (ix < 0x39800000) {  /* |x| < 2**-12 */
 //and so on...
```

The beauty of this code is that it means EXACTLY the same thing for ALL fp's with 23 bits of fraction and 8 bits of exp

While we're at it: sixit::geometry..._

We also have our own geometry lib sixit::geometry, also templatized on fp (and working on rigid body physics lib too)

```
template<typename fp>
fp sixit::geometry::line_segment2::length() const {
  auto dx = p2.x-p1.x;
  auto dy = p2.y-p1.y;
  return sixit::dmath::sqrt(dx*dx+dy*dy);
}
```

Now:6 Deterministic IEEE fp's_

1. sixit::dmath::ieee_float_soft (based on Berkeley SoftFloat, GUARANTEED to work)
2. sixit::dmath::ieee_float_shared_lib (GUARANTEED to work - as long as CPUs handle +-*/ identically)
3. sixit::dmath::ieee_float_static_lib (SHOULD work if no LTO)
4. sixit::dmath::ieee_float_inline_asm (SHOULD work, but for MSVC we have to use intrinsics ➡ more shaky)
5. sixit::dmath::ieee_float_if_strict_fp (requires specific compiler flags)
6. sixit::dmath::ieee_float_if_semicolon_prohibits_reordering (requires less restrictive compiler flags)

Non-trivial code: the all-important semicolon

```
ieee_float_if_semicolon operator-(ieee_float_if_semicolon
                                  other) const {
    float ret = data - other.data;
    // !!! temporary variable and the semicolon is
    // substantial here.
    // Do not even think about merging in a single line !!!
    return ret;
}
```

Non-trivial code: inline_asm

```
#ifdef SIXIT_CPU_X64
inline asm_float_t ieee_subtract_float(asm_float_t a,
                                       asm_float_t b) {
    asm("subss %[rhs], %[a]" : [a] "+x"(a) : [rhs] "xm"(b));
    return a;
}
#endif
```

Correctness: dmath&geometry tests

	float, max flags	float, no-fast-math flags	float, strict-fp flags	All ieee_float_* classes, sum across 3 different sets of optimizations flags
GCC/x64	**17**/163	**16**/163	**14**/163	**0**/489
MSVC/x64	**17**/163	**17**/163	**14**/163	**0**/489
Apple Clang/ARM64	**23**/163	**23**/163	**14**/163	**0**/489
GCC/RISC-V	**25**/163	**24**/163	**14**/163	**0**/489

Tests: determinism failed / all tests

Performance: dmath&geometry tests

	ieee_float _soft	ieee_float _shared_lib	ieee_float _static_lib	ieee_float _inline_asm	ieee_float _if_strict_fp	ieee_float _if_semi…
GCC/x64	6.03x	3.67x	2.62x	**1.19x**	1.44x	1.40x
MSVC/x64	4.83x	1.89x	1.87x	1.33x	2.20x	**1.31x**
Apple-Clang/ ARM64	5.54x	2.43x	3.09x	**1.32x**	2.47x	2.39x
GCC/RISC-V	3.97x	3.07x	1.55x	1.10x	**1.09x**	1.10x
Process-wide Deoptimizations	—	—	No LTO	—	No fast-math, no reordering, no FMA	No fast-math, no fp-contract=fast

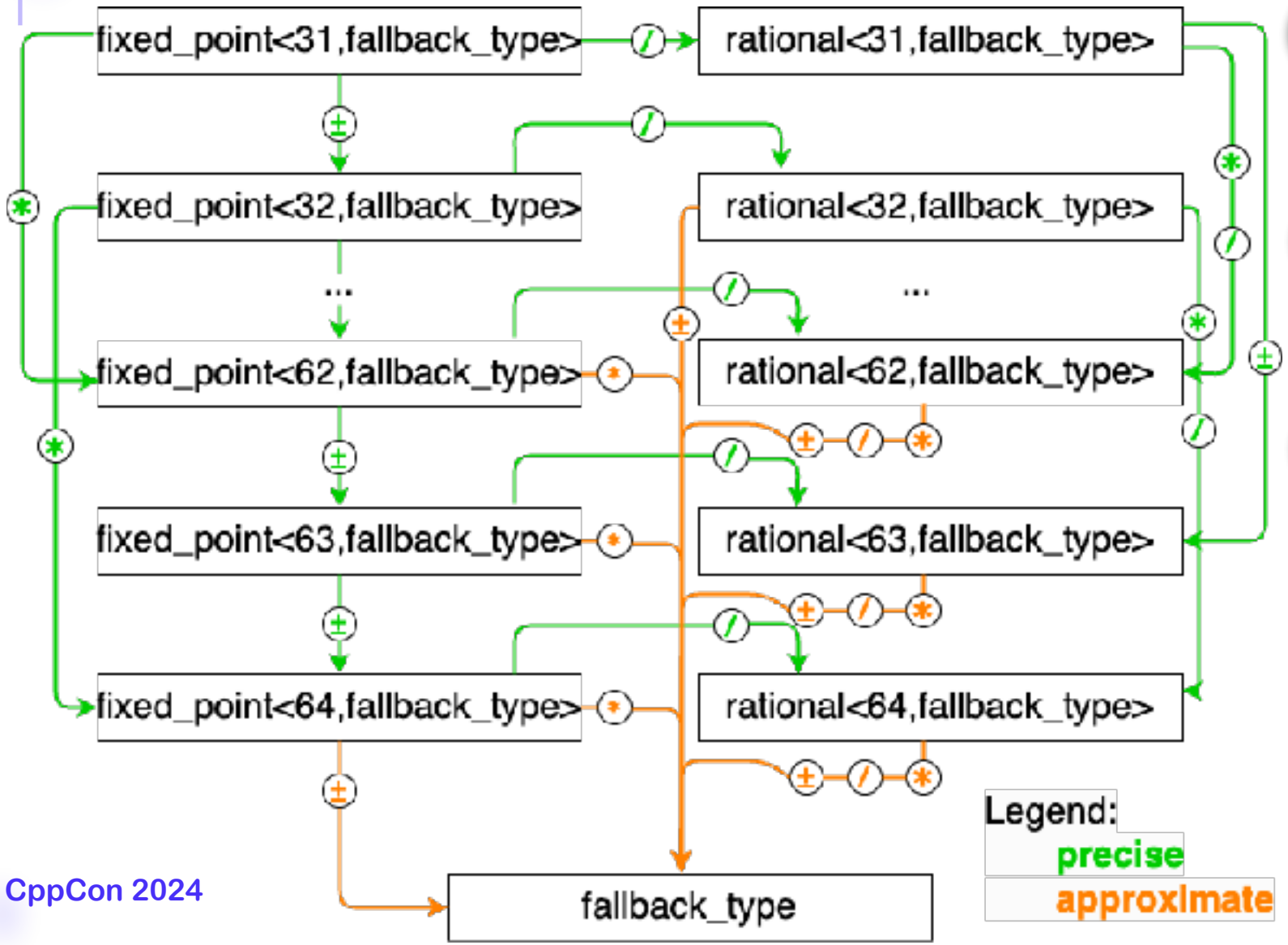
Results are execution times, normalized to performance of built-in float with max flags, averaged using geometric mean over 163 tests.

WIP: fixed-point for geometry_

1. Floating-point is not the best choice for geometry coordinates (which have uniform distribution)
2. Associativity Guarantees of fixed-point make Writing Correct Algorithms Simpler
3. Normalize to Bounding Box
4. fixed-point class, usable as fp for dmath and geometry

   ➡ need for “fallback-type”
5. Will improve geometry performance for ALL ieee_float_* classes used as fallback_type

fixed_point<N,fallback_type>_



1. We're trying to keep precise operations as long as possible
2. For geometry, fixed_point<31> is normalized to (-1,1)
3. "fallback_type" means "whenever we cannot do precise, we'll have to resort to 'real' floating-point-like type"

sixit::×/−+

Performance: geometry tests with fixed_point<>_

fallback_type ➡	ieee_float _soft	ieee_float _shared_lib	ieee_float _static_lib	ieee_float _inline	f… …t_fp	ieee_float _if_semi…
Clang/x64						
MSVC/x64						
GCC/x64						
Clang/ARM64						
GCC/RISC-V						
Process-wide Deoptimizatic		—	No LTO	—	No fast-math, no reordering, no FMA	No fast-math

WORK IN PROGRESS

…ults are normalized to performance of built-in float, and averaged using geometric mean.

What We Did Achieve_

1. Out of the Box Solution. It should Just Work
   ➡ Yes, but if you want performance, flags may be needed
2. Try To Be Consistent Across All Modern CPUs
   ➡ WIP (so far x64+ARM+RISC-V, WIP WASM)
3. Support At Least The 3 Major C++ Compilers
   ➡ Yes
4. Mix&Match: Combining Deterministic and !Deterministic in One Executable ➡ Yes
5. Different Performance-vs-Guarantees Options ➡ Yes
6. Extensive Testing ➡ Eternal WIP

Limitations_

1. It is not possible to make existing code magically deterministic

   Templatizing is always possible, but may take significant (albeit mostly mechanistic) efforts

2. Making 3rd-party libs deterministic is a MAJOR effort

3. Performance Is Inherently Worse
   Currently Performance within 75-90% of float, WIP

   Partially Mitigated by Allowing Mix&Match

4. Fighting for Performance may require messing with compiler flags

Our C++ Standard/WG21 Proposal_

1. P3375: Reproducible floating-point results
   Spearheaded by Guy Davidson
2. Proposed: std::strict_float<float>, etc.
3. Reproducible as per Clause 11 of ISO/IEC 60559:2020
   (a.k.a. IEEE 754-2019)
4. Allows to avoid catastrophic cancellations:

   Convert to strict_float, subtract, convert back if desirable
5. Expected to perform better than any of our variants

~~x87~~

github.com/sixitbb/sixit-dmath

assert(a+b+c==b+c+a);

FMA, OMG…

Q&A…

I hope I said enough so you can start asking questions

{ unsigned sign : 1; unsigned exp : 8; unsigned fraction: 23; }

sixit::dmath::sin()

CppCon 2024

sixit::

7 Authors from Six Impossible Things Before Breakfast Limited