Fast, High-Quality Pseudo-Random Numbers for Non-Cryptographers

ROTH MICHAELS

Cppcon
The C++ Conference

2022
September 12th-16th

Roth Michaels

Principal Software Engineer,
Soundwide Audio Research

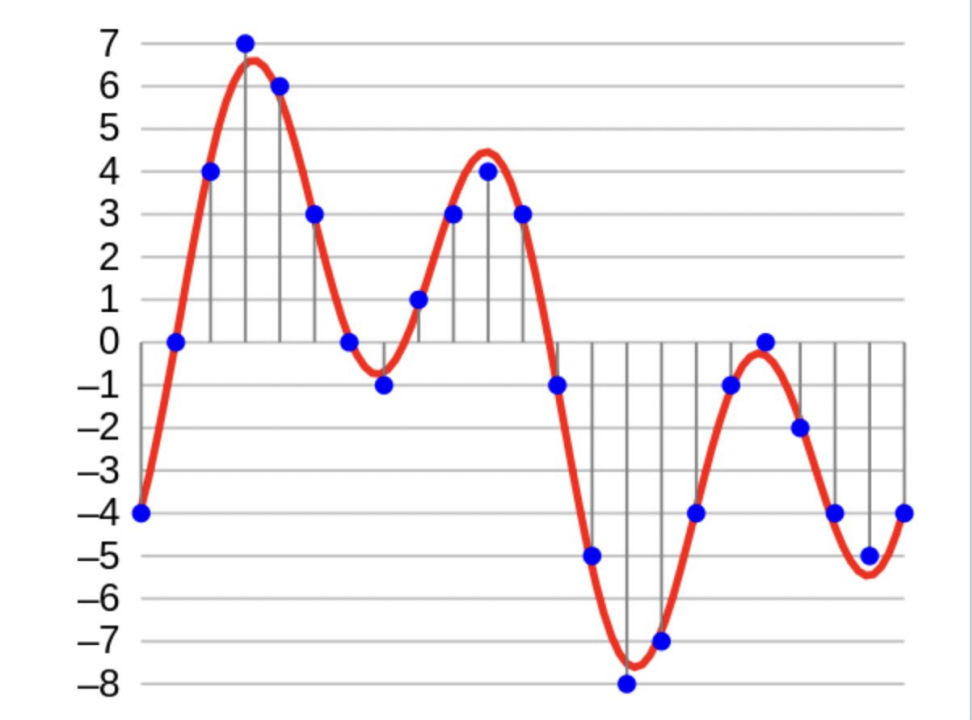
soundwide

soundwide

iZOTOPE NATIVE INSTRUMENTS Plugin Alliance BRAINWORX Sound Stacks


01:00:39:02.62
20 3 1 214
44,1
Varispeed
±0.00%
120,0000
Tempo beibehalten
4/4
/16
Kein In
Kein Out
Bearbeiten
Funktionen
Ansicht
Einrasten: Intelligent
Verschieben: X-Fade
Norweg...d_audio
Inst 1
Vögel
Wind_Audio
Dub chords
Inst 2
Inst 3
Inst 4
Dub ch...alftime
Molek Melodies
Bongos...Merged
Bongos plus 100
Low Ris...ls pings
drums
Big 808
808 Bass
mm wave weaver
norwegi...S JAZZ
norwegi...o spirit
pads
Inst 3
Manuell
Vergleichen
Widerrufen
Wiederholen
Ansicht: Editor
7 SKIES, Magnificence - THE DRILL
FILTER
CUTOFF
RESO
ENV AMT
THE DRILL
FLT ENV
ATTACK
DECAY
SUSTAIN
RELEASE
AMP
AMP ENV
LAYER 1
LAYER 2
Inst 4
gr808
808 Center
Mode
[mono]
808 Sides
Panning
gr808
kick
[attack: fast]
[releaseeeeee]
Reverb: [Cloud Grain]
Size
Amount:
Rev Lo-Cut:
Rev Hi-Cut:
Presets: 1 2 3
Clay and Kelsy
Bouncy FX: [off]
Speed ModW:
Kontakt
Inst 2
MS20 Drums
Cutoff
Saturation
Reverb
Delay
Delay Time
Peter Flint:
MS20 Drums
Kontakt

Digital Audio Basics



Digital Audio Basic

- Sampling Rate: Frequency resolution
  - e.g.: SR=48,000Hz (max freq=24,000Hz)
- Bit-depth: Amplitude resolution
  - 24bit 144dB range (`uint32_t` or `float`)
- Process callback works on buffers
  - `process(span<float> buffer);`
  - `buffer.size() == 512 || // e.g.`

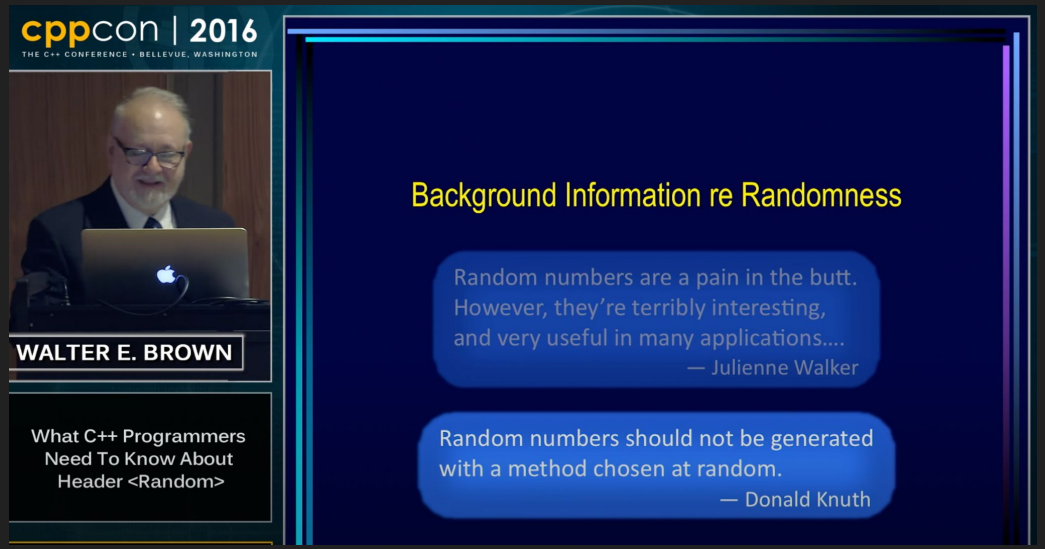
    `buffer.size() == 8192`

Questions working on Neoverb

Questions working on Neoverb

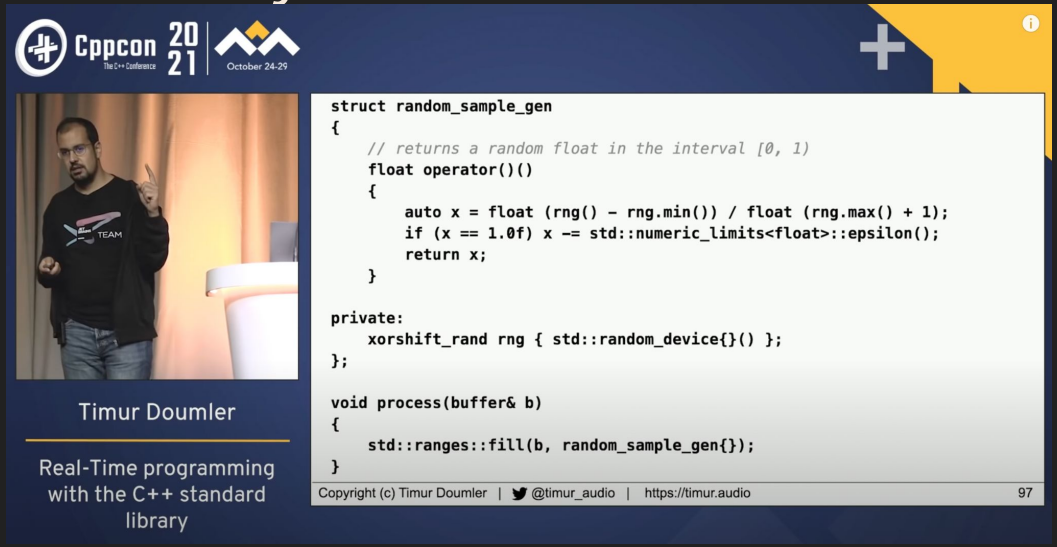
- What's the best way to seed (`std::mt19937`)?
- Should I check out generators outside of `std`?
- What to do about distributions:
  - Timur says we can't trust them for real-time
  - Walter says to "do it right" — we might go to Mars
- Colleague using Xoshiro128+
  - Not a UniformRandomBitGenerator

What C++ Programmers Need to Know about Header <random>

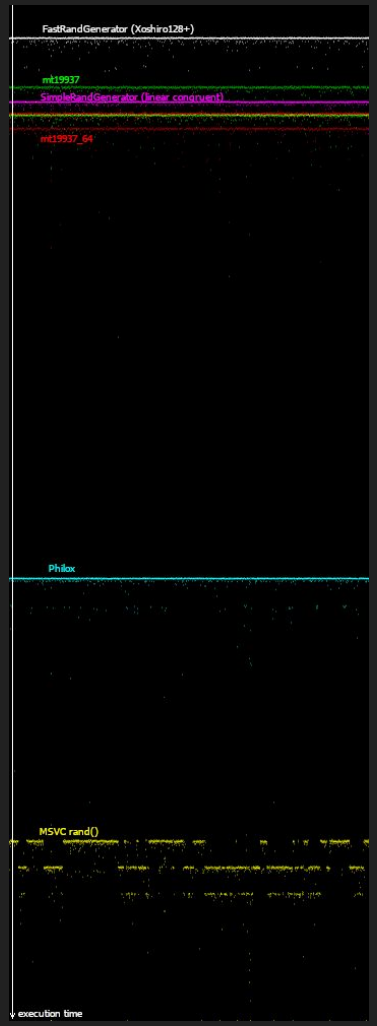


https://www.youtube.com/watch?v=6DPkyvkMkk8

Real-time Programming with the C++ Standard Library



https://www.youtube.com/watch?v=Tof5pRedskI


FastRandGenerator (Xoshiro128+)
mt19937
SimpleRandGenerator (linear congruent)
mt19937_64
Philox
MSVC rand()
execution time

What do I really care about?

Performance? Quality? Does any of it matter?

Part 1 of 2: What matters?

Psychoacoustic analysis:

- numerical analysis of results
- listening tests

H T H H T H T H T T

T H H H T T T T H H

T T T T T T T T T T

T T T H T H H H H T

H T T H H H T T H H

What is random?

- Random numbers
- Pseudo-random numbers
- Quasi-random numbers

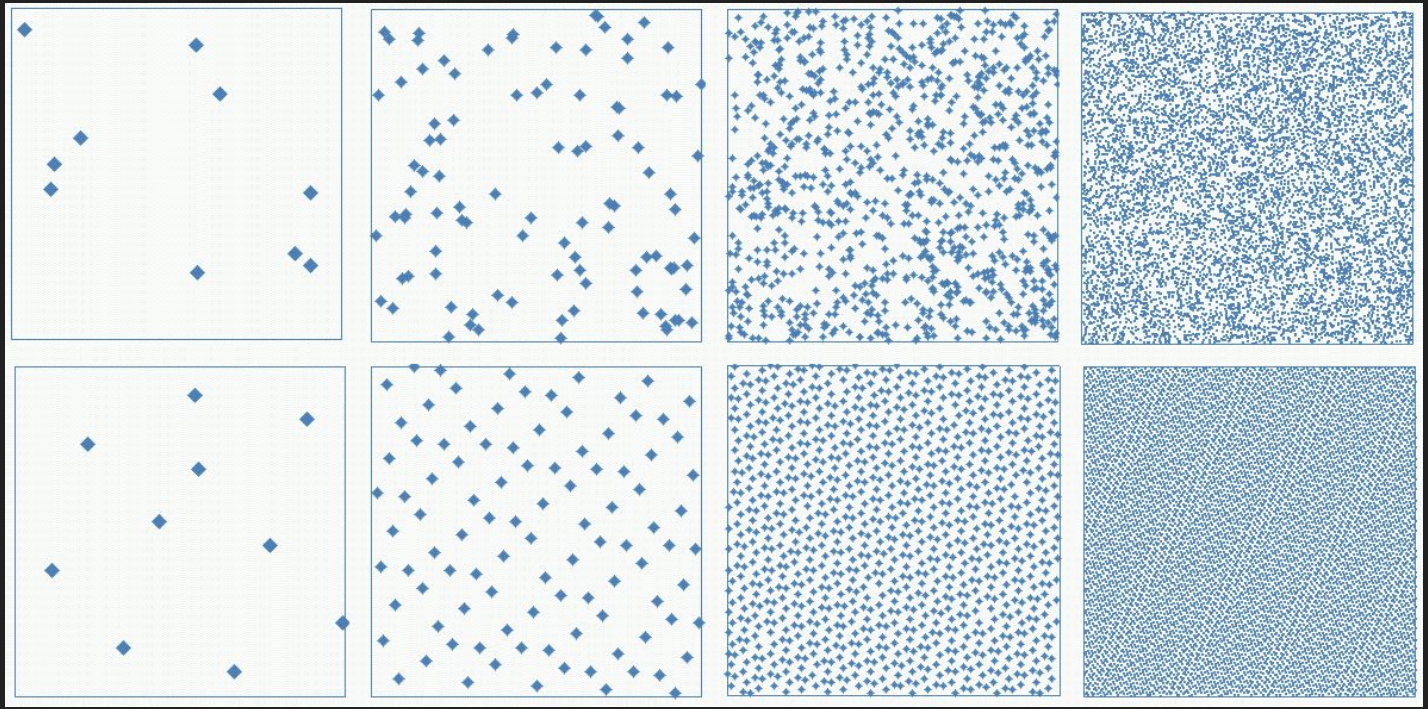
Random numbes

- Coin toss
- Fair die roll
- /dev/random /dev/urandom
- Atmospheric noise
- Radioactive decay
- Lava lamps

Silicon Graphics: Lavarand

Quasi-random numbers

- a.k.a. Low-discrepancy sequence



Pseudo-random number generator

Deterministic algorithm that approximates randomness.

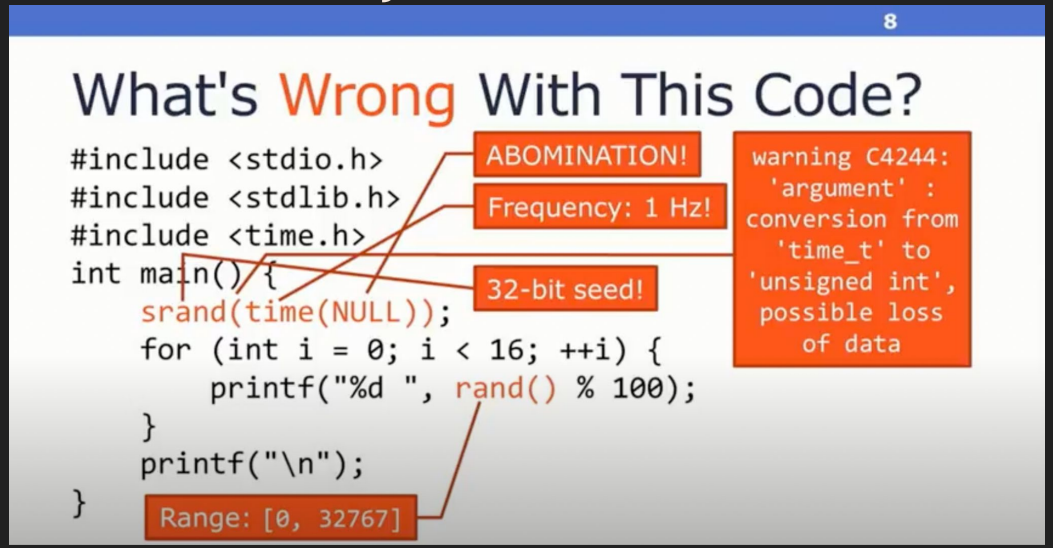
Use-cases

- Cryptography
- Slot machines
- Monte carlo and other simulations
- ML (initial weights, input to generative networks)
- Videos games
- Pinball
- VFX/Graphics
- Audio

Roll a 6-sided die

```
uint8_t d6() {
   return std::rand() % 6u + 1u;
}
```

rand Considered Harmful
- Stephan T. Lavajev



https://www.youtube.com/watch?v=Gb-1grkVGSg

Roll a 6-sided die

```
uint8_t d6() {
   return std::rand() % 6u + 1u;
}
```

Roll a 6-sided die

```
uint8_t d6() {
   return std::rand() % 6u + 1u;
}
```

How to roll [1, 5] with a 6-sided die

How to roll [1, 7] with a 6-sided die

Fair 6-sided die

```
uint8_t d6() {
  static std::mt19937 g{std::random_device{}()};
  static std::uniform_int_distribution<uint8_t> d(1, 6);
  return d(g);
}
```

Fair 6-sided die

```
uint8_t d6() {
  static std::mt19937 g{std::random_device{}()};
  static std::uniform_int_distribution<uint8_t> d(1, 6);
  return d(g);
}
```

Fair 6-sided die

```
uint8_t d6() {
  static std::mt19937 g{std::random_device{}()};
  static std::uniform_int_distribution<uint8_t> d(1, 6);
  return d(g);
}
```

Fair 6-sided die

```
uint8_t d6() {
  static std::mt19937 g{std::random_device{}()};
  static std::uniform_int_distribution<uint8_t> d(1, 6);
  return d(g);
}
```

Fair 6-sided die

```
class D6 {
public:
  D6() : m_gen{std::random_device{}()} {}

  uint8_t Roll() {
    return m_dist(m_gen);
  }
private:
  std::mt19937 m_gen;
  std::uniform_int_distribution<uint8_t> m_dist{1u, 6u};
};
```

Fair 6-sided die

```
class D6 {
public:
  D6() : m_gen{std::random_device{}()} {}

  uint8_t Roll() {
    return m_dist(m_gen);
  }
private:
  std::mt19937 m_gen;
  std::uniform_int_distribution<uint8_t> m_dist{1u, 6u};
};
```

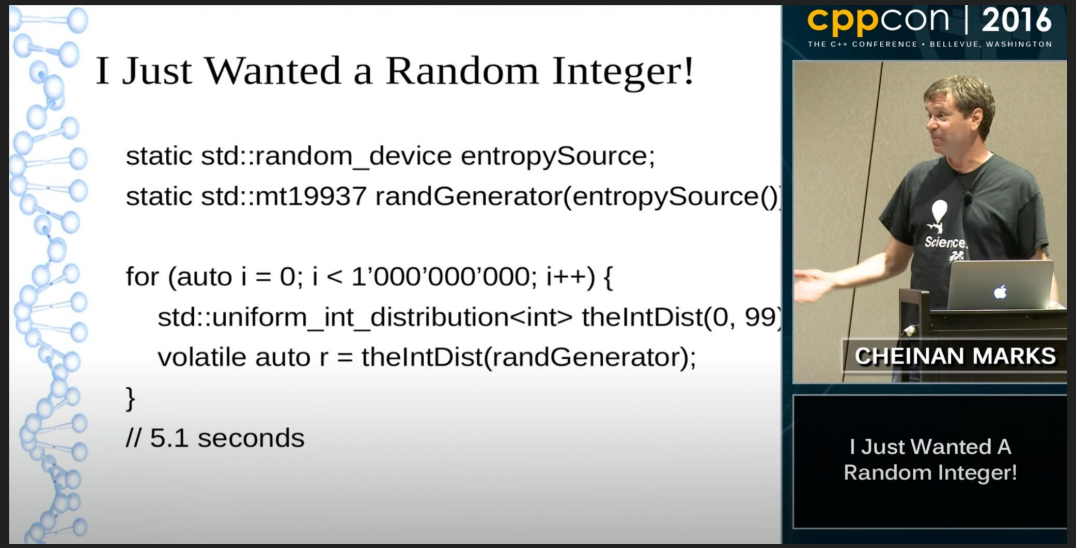
Fair 6-sided die

```
class D6 {
public:
  D6() : m_gen{std::random_device{}()} {}

  uint8_t Roll() {
    static std::uniform_int_distribution<uint8_t> d(1, 6);
    return m_dist(m_gen);
  }
private:
  std::mt19937 m_gen;
};
```

Fair 6-sided die

```
class D6 {
public:
  D6() : m_gen{std::random_device{}()} {}

  uint8_t Roll() {
    std::uniform_int_distribution<uint8_t> d(1, 6);
    return m_dist(m_gen);
  }
crivate:
  std::mt19937 m_gen;
};
```

I Just Wanted a Random Integer!



https://www.youtube.com/watch?v=4_QO1nm7uJs

Fair 6-sided die

```
class D6 {
public:
  D6() : m_gen{std::random_device{}()} {}

  uint8_t Roll() {
    return m_dist(m_gen);
  }
private:
  std::mt19937 m_gen;
  std::uniform_int_distribution<uint8_t> m_dist{1u, 6u};
};
```

Fair 6-sided die

```
class D6 {
public:
  D6() {
    const auto seed = std::random_device{}();
    m_gen.seed(seed);
    std::print("D6 Seed: {}", seed);
  }
  // ...
private:
  std::mt19937 m_gen{};
  std::uniform_int_distribution<uint8_t> m_dist{1u, 6u};
};
```

Fair 6-sided die

```
class D6 {
public:
  D6() {
    const auto seed = std::random_device{}();
    m_gen.seed(seed);
    std::print("D6 Seed: {}", seed);
  }
  D6(std::mt19937::result_type seed) : m_gen{seed} {}
private:
  std::mt19937 m_gen{};
  std::uniform_int_distribution<uint8_t> m_dist{1u, 6u};
};
```

How do I see mt19937 well?

Found Xoshiro and PCG blogs

Twitter Problem

```
std::random_device rd{};
std::mt19937 my_rng(rd());
if (my_rng() == 7) {
  // lucky seven!  send report
  send_detailed_tracking_info_secretly();
}
```

Twitter Problem

```
std::random_device rd{};
std::mt19937 my_rng(rd());
if (my_rng() == 13) {
  // unlucky thirteen!  send report
  send_detailed_tracking_info_secretly();
}
```

Twitter Problem

```
std::random_device rd{};
std::seed_seq seed{rd(), rd(), rd(), rd()};
std::mt19937 my_rng(seed);
if (my_rng() == 13) {
  // unlucky thirteen!  send report
  send_detailed_tracking_info_secretly();
}
```

mt19937 Pros / Cons

- Pros
  - Large period: $2^{19937} - 1$
  - High quality?
  - Mersenne primes seem cool
  - Mersenne's Laws: music theory/strings
- Cons
  - Hard to seed
  - Slow?
  - Fails some statistical tests

Seedings strategy

- Random device
- std::seed_seq
- Time
- Sequential
- Jumps

Seeding Multiple Instances (Reload)

42

43

44

Seeding Multiple Instances (Reload)

44

42

43

Seeding Multiple Instances (Reload)

Seeding Multiple Instances (Reload)

42

43

44

Seeding Multiple Instances (Reload)

42

43

44

45

```
template <class PRNG>
class SeedPod {
public:
      using SeedType = typename PRNG::result_type;

      SeedPod(SeedType seed) : m_nextSeed{seed} {}

      SeedType NewSeed() {
            auto newSeed = m_nextSeed.fetch_add(1);
            if (newSeed == 0) {
                  ClaimSeed(1);
                  return 1u;
            }
            return newSeed;
      }
      void ClaimSeed(SeedType seed) {
            auto nextSeed = m_nextSeed.load();
            while (seed >= nextSeed) {
                  if (m_nextSeed.compare_exchange_weak(nextSeed, seed + 1u)) {
                        return;
                  }
            }
      }
private:
      std::atomic<SeedType> m_nextSeed;
};
```

```
template <class PRNG>
class SeedPod {
public:
    // ...

    SeedType NewSeed() {
        auto newSeed = m_nextSeed.fetch_add(1);
        if (newSeed == 0) {
            ClaimSeed(1);
            return 1u;
        }
        return newSeed;
    }

    // ...
};
```

```
template <class PRNG>
class SeedPod {
public:
    // ...

    void ClaimSeed(SeedType seed) {
        auto nextSeed = m_nextSeed.load();
        while (seed >= nextSeed) {
            if (m_nextSeed.compare_exchange_weak(nextSeed, seed + 1u)) {
                return;
            }
        }
    }

    // ...
};
```

Generators outside the standard

- Xoshiro/Xoroshiro
  - https://prng.di.unimi.it/
  - https://github.com/Reputeless/Xoshiro-cpp
- PCG
  - https://www.pcg-random.org/
  - https://github.com/imneme/pcg-cpp
- Random123 / Philox
  - https://github.com/DEShawResearch/random123
  - P2075

Xoshiro/Xoroshiro

- Xoshiro256**
- Xoshiro256++
- Xoshiro256+
- Xoshiro128**
- Xoshiro128++
- Xoshiro128+
- Xoroshiro128**
- Xoroshiro128++
- Xoroshiro128+

Xoshiro/Xoroshiro Seeding

- Use **`result_type`** to seed **`SplitMix64`**
- Fill state with **`SplitMix64`**

PCG

- pcg32
- pcg64

PCG Seeding

- Provides improved **`seed_seq`**
- Can scramble **`result_type`** seed

Random123 (proposed for standard)

- `philox2x32`
- `philox4x32`
- `philox2x64`
- `philox4x64`

Random123 seeding

- `result_type` seed
- `result_type` counter

Demo Time!

Benchmarks

Benchmark notes

- Apple M1 Max
- Xcode 13.2.1
- Sampling rate 48000Hz
- Buffer size 512 or 8192

Uniform noise microbenchmark

XoshiroCpp::Xoshiro128Plus	0.963ns
XoshiroCpp::Xoshiro256Plus	0.964ns
pcg32	1.06ns
XoshiroCpp::Xoshiro128PlusPlus	1.12ns
XoshiroCpp::Xoshiro128StarStar	1.30ns
prng_experiments::Xorshift32	1.86ns
std::mt19937	3.49ns
r123::Engine<r123::Philox4x32>	3.77ns
r123::Engine<r123::Philox2x32>	3.92ns
std::minstd_rand	4.04ns
std::knuth_b	5.30ns

Uniform white noise

```
template <class PRNG, template <class> class Dist = std::uniform_real_distribution>
class UniformWhiteNoiseProcessor {
public:
	UniformWhiteNoiseProcessor(typename PRNG::result_type seed) : m_gen{seed}, m_seed{seed}
{}

	void ProcessInPlace(DSP::MultichannelBufferRef<float> buffer) {
		Expects(buffer.GetNumChannels() == 1u); // only 1 channel supported for now
		auto channel = buffer[0u];
		std::generate(channel.begin(), channel.end(), [this] {
			return 1.224616f * m_gain * m_dist(m_gen);
		});
	}

	void SetGain(float gain) { m_gain = gain; }

private:
	PRNG m_gen;
	Dist<float> m_dist{-1.f, std::nextafter(1.f, std::numeric_limits<float>::max())};
	float m_gain{.1f};
	typename PRNG::result_type m_seed;
}
```

Gaussian white noise

```
template <class PRNG, template <class> class Dist = std::uniform_real_distribution>
class GaussianWhiteNoiseProcessor {
public:
      GaussianWhiteNoiseProcessor(typename PRNG::result_type seed) : m_gen{seed},
                                                                      m_seed{seed} {}

      void ProcessInPlace(DSP::MultichannelBufferRef<float> buffer) {
            Expects(buffer.GetNumChannels() == 1u); // only 1 channel supported for now
            auto channel = buffer[0u];
            std::generate(channel.begin(), channel.end(), [this] {
                  return 3.5359f * m_gain * m_dist(m_gen);
            });
      }

      void SetGain(float gain) { m_gain = gain; }

private:
      PRNG m_gen;
      Dist<float> m_dist{0.f, 0.2f};
      float m_gain{.1f};
      typename PRNG::result_type m_seed;
};
```

DustGenerator

```
struct RandomDust {
      float delayFactor; //!< 0 - 1
      unsigned clipIndex;
};
template <class PRNG>
class RandomDustGenerator {
public:
      static constexpr unsigned NumberOfDustClips = 13;
      RandomDustGenerator(typename PRNG::result_type seed) : m_gen{seed}, m_seed{seed} {}

      RandomDust operator()() {
            const auto delay = m_delayDist(m_gen);
            const auto clip = m_clipDist(m_gen);
            return {delay, clip};
      }

private:
      PRNG m_gen;
      std::uniform_real_distribution<float> m_delayDist{0.f, 1.f};
      std::uniform_int_distribution<unsigned> m_clipDist{0u, NumberOfDustClips - 1u};
      typename PRNG::result_type m_seed;
}
```

ScratchGenerator

```
struct RandomScratch {
      unsigned clipIndex;
      float relativeTiming; //!< 0 - 1
};
template <class PRNG>
class RandomScratchGenerator {
public:
      static constexpr unsigned NumberOfScratchClips = 3;
      RandomScratchGenerator(typename PRNG::result_type seed) : m_gen{seed}, m_seed{seed} {}

      RandomScratch operator()() {
            const auto clip = m_clipDist(m_gen);
            const auto delay = m_delayDist(m_gen);
            return {clip, delay};
      }

private:
      PRNG m_gen;
      std::uniform_real_distribution<float> m_delayDist{0.f, 1.f};
      std::uniform_int_distribution<unsigned> m_clipDist{0u, NumberOfScratchClips - 1u};
      typename PRNG::result_type m_seed;
}
```

Naive quality investigation

Amortized cost distributions

What can we really get away with in real-time audio?

DustGenerator

```
struct RandomDust {
      float delayFactor; //!< 0 - 1
      unsigned clipIndex;
};
template <class PRNG>
class RandomDustGenerator {
public:
      static constexpr unsigned NumberOfDustClips = 13;
      RandomDustGenerator(typename PRNG::result_type seed) : m_gen{seed}, m_seed{seed} {}

      RandomDust operator()() {
            const auto delay = m_delayDist(m_gen);
            const auto clip = m_clipDist(m_gen);
            return {delay, clip};
      }

private:
      PRNG m_gen;
      std::uniform_real_distribution<float> m_delayDist{0.f, 1.f};
      std::uniform_int_distribution<unsigned> m_clipDist{0u, NumberOfDustClips - 1u};
      typename PRNG::result_type m_seed;
}
```

Uniform real probably fine

Check your implementation

std::generate_canonical

Defined in header <random>

```
template< class RealType, std::size_t Bits, class Generator >
RealType generate_canonical( Generator& g );
```

(since C++11)

Generates a random floating point number in range $[0, 1)$.

To generate enough entropy, `generate_canonical()` will call `g()` exactly k times, where $k = \max(1, \lceil \frac{b}{\log_2 R} \rceil)$ and

- `b = std::min(Bits, std::size_t{std::numeric_limits<RealType>::digits})`,
- `R = g.max() - g.min() + 1`.

Simple distribution benchmarks (Dust)

```
mt19937,uniform_int_distribution,uniform_real_distribution                14.3ns
mt19937,SimpleUniformIntDistribution,uniform_real_distribution            7.27ns
mt19937,SimpleUniformIntDistribution,SimpleUniformRealDistribution        7.29ns

Xoshiro128Plus,uniform_int_distribution,uniform_real_distribution         3.15ns
Xoshiro128Plus,SimpleUniformIntDistribution,uniform_real_distribution     2.35ns
Xoshiro128Plus,SimpleUniformIntDistribution,SimpleUniformRealDistribution 2.41ns

pcg32,uniform_int_distribution,uniform_real_distribution                  6.13ns
pcg32,SimpleUniformIntDistribution,uniform_real_distribution              2.23ns
pcg32,SimpleUniformIntDistribution,SimpleUniformRealDistribution          2.27ns
```

Let's benchmark a real plug-in (Neoveb)

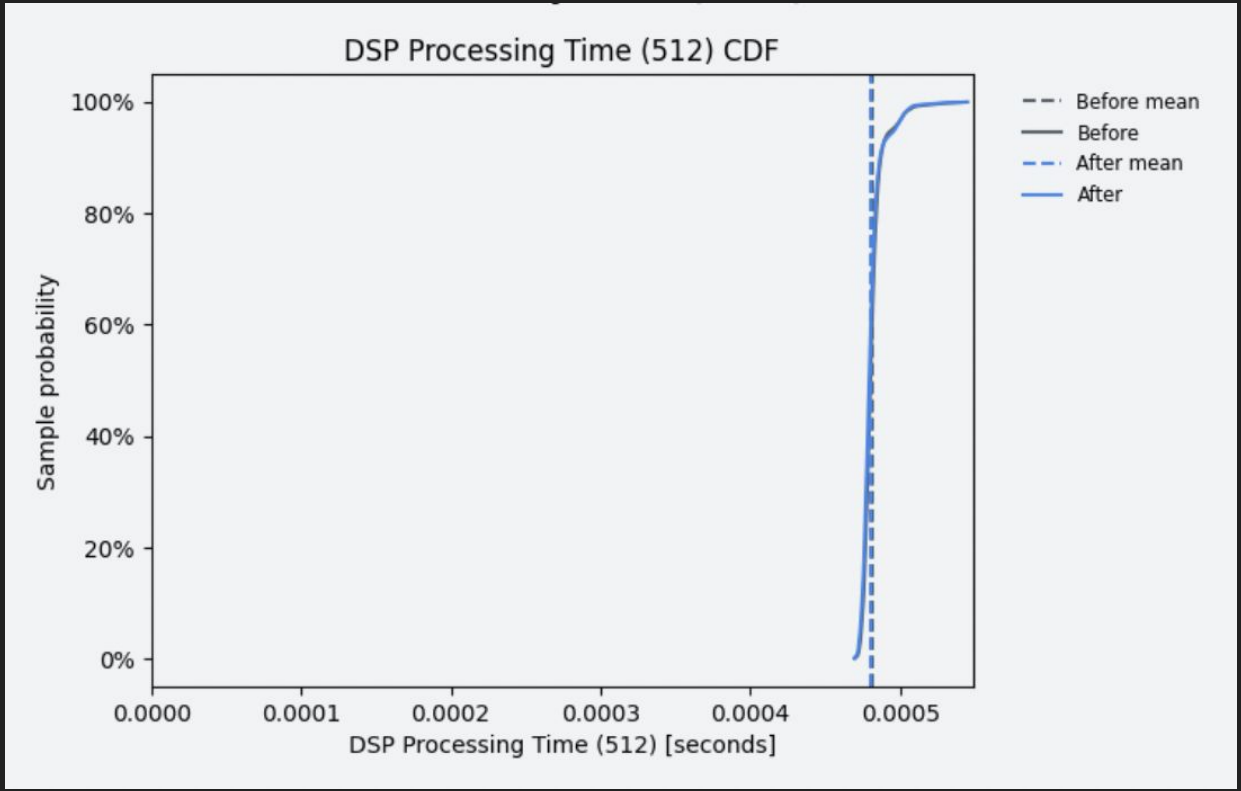
Xoshiro128+ vs mt19937 (Neoverb)

VST3 Benchmark Suite

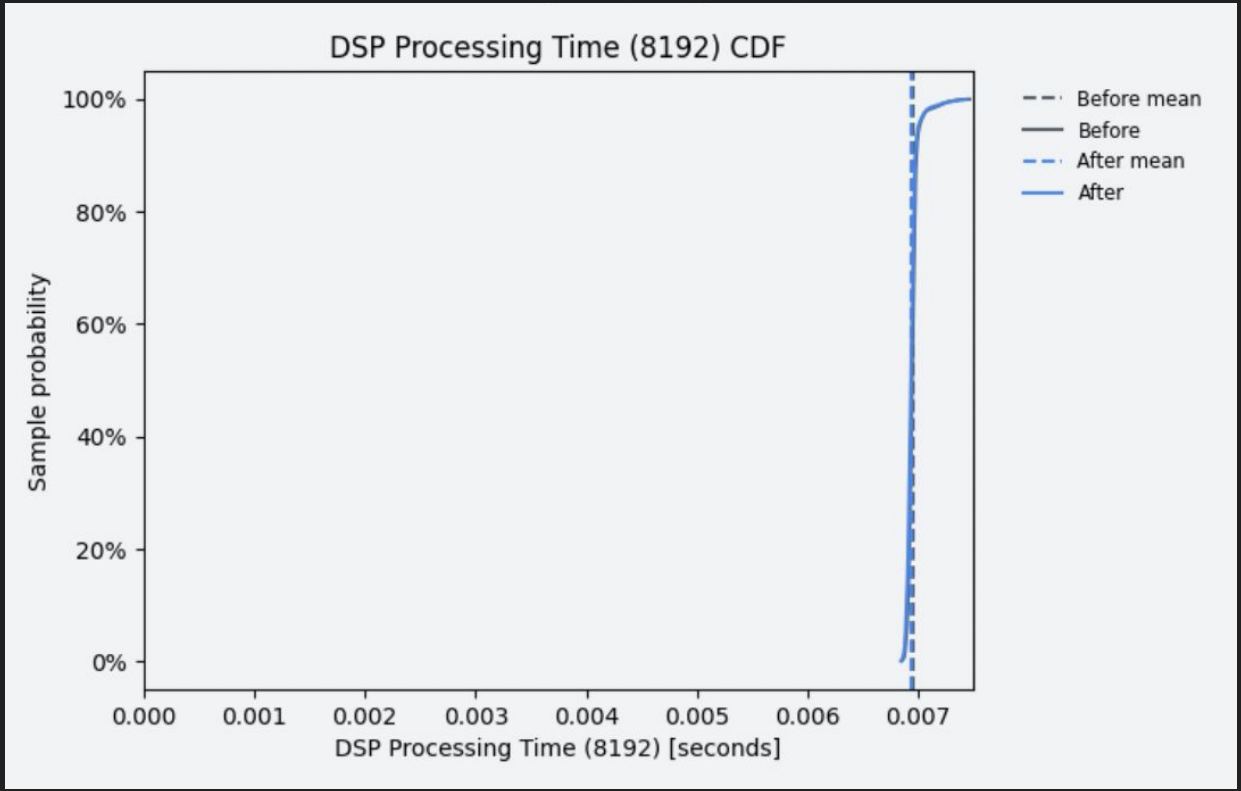
Comparing **Before (before)** with **After (after)**
significance level of this report: 0.05

- Start-up Time (No change detected)
- Real-time DSP (No change detected)
- Offline DSP (No change detected)

Xoshiro128+ vs mt19937 (Realtime 512)



Xoshiro128+ vs mt19937 (Offline 8192)



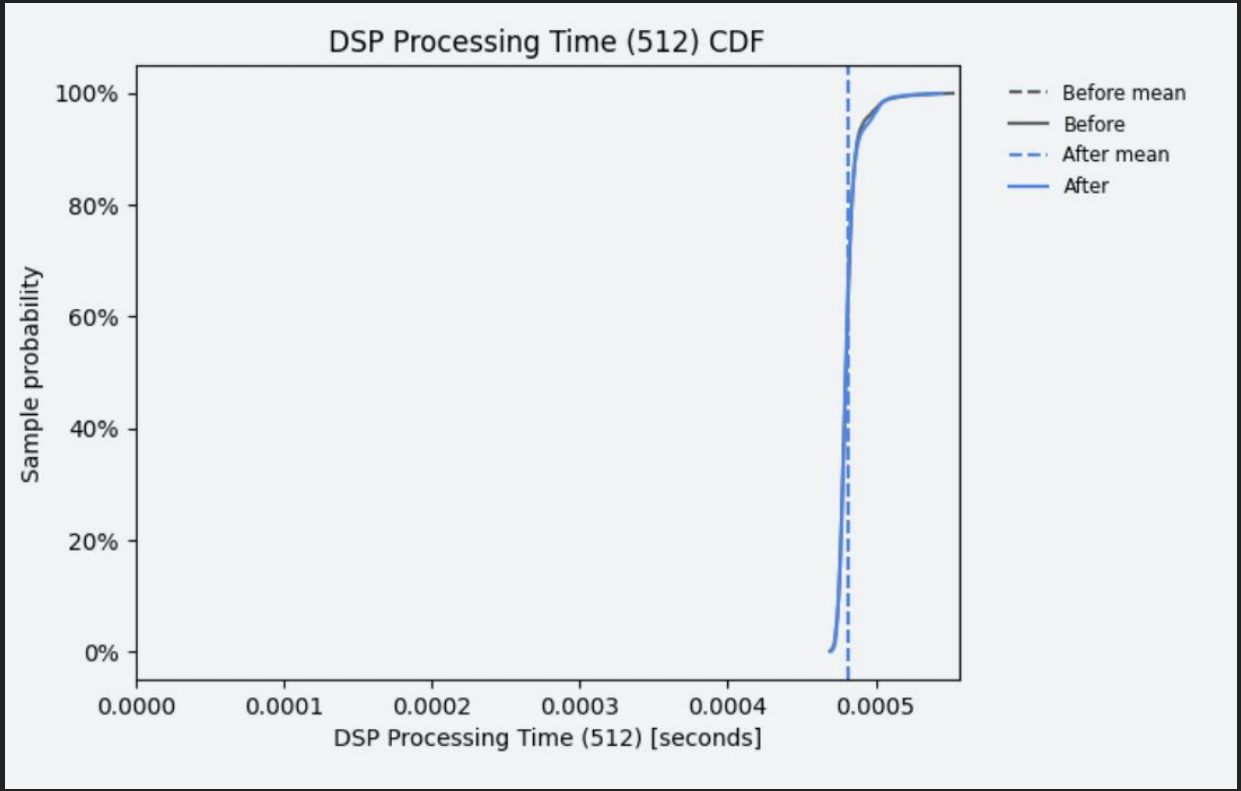
Xoshiro128+ vs pcg32 (Neoverb)

VST3 Benchmark Suite

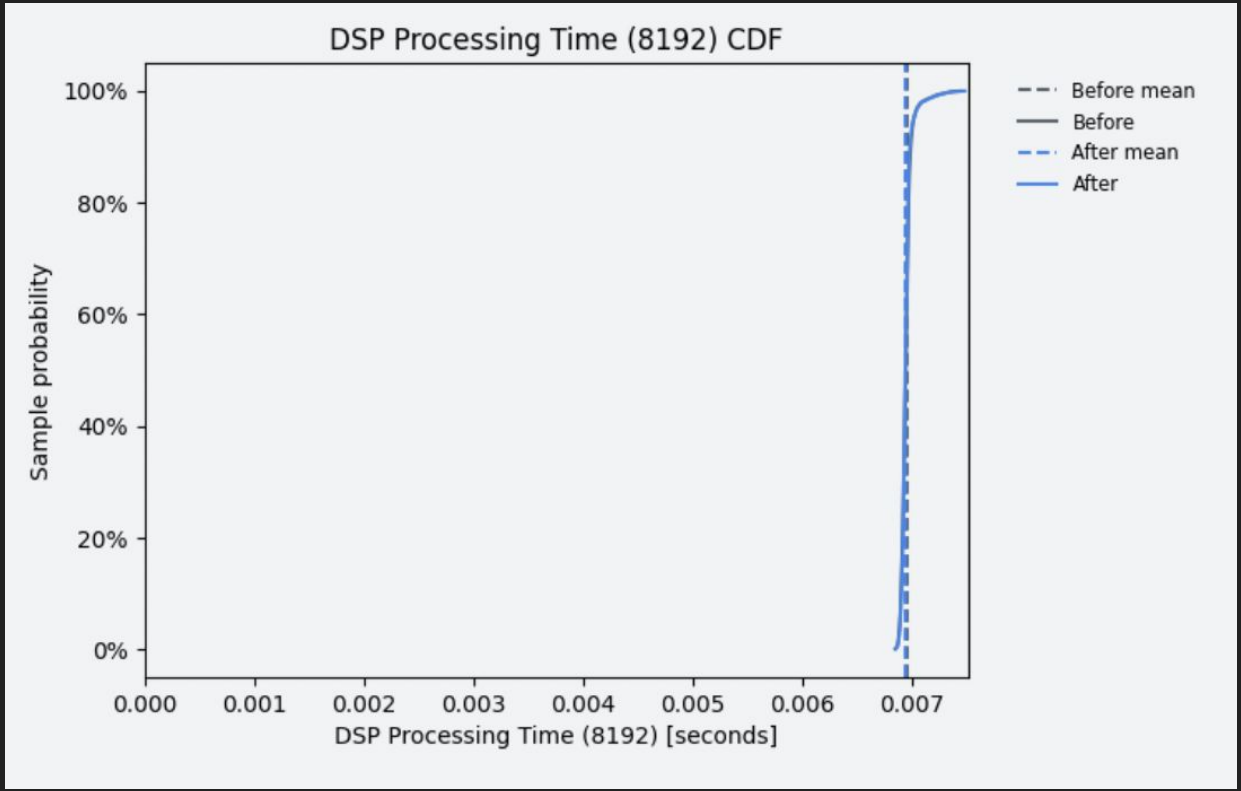
Comparing **Before (before)** with **After (after)**
significance level of this report: 0.05

- ▶ Start-up Time (No change detected)
- ▶ Real-time DSP (No change detected)
- ▶ Offline DSP (No change detected)

Xoshiro128+ vs pcg32 (Realtime 512)



Xoshiro128+ vs pcg32 (Offline 8192)



Conclusions

- Maybe none of this matters?
- I like convenience seeding Xoshiro128+
- Perhaps Xoshiro128+ is good enough for ints
- Stick with standard library distributions
  - use boost or one open-source standard library implementation if you need portability

Thank you!

Roth Michaels — Principal Software Engineer
rmichaels@izotope.com
@thevibesman

soundwide