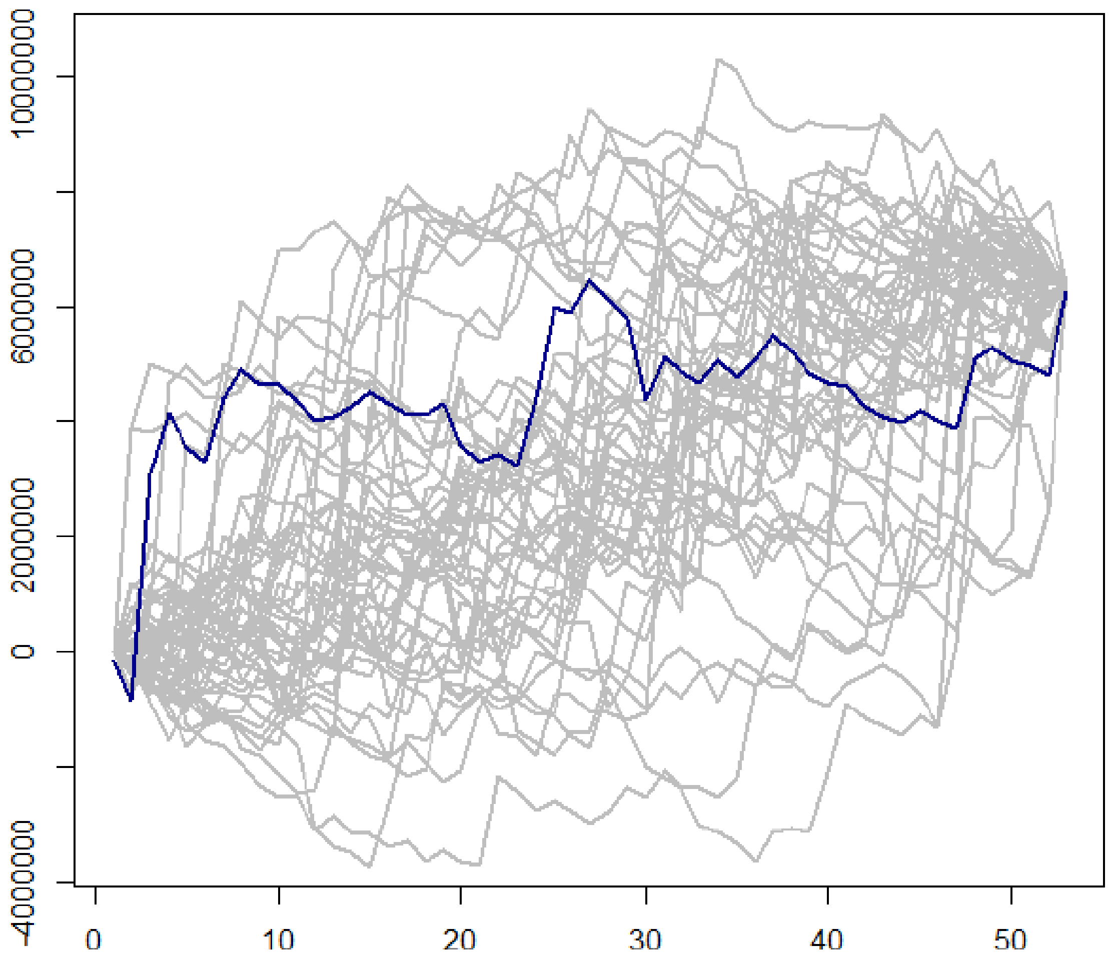
Probability and Statistics in C++

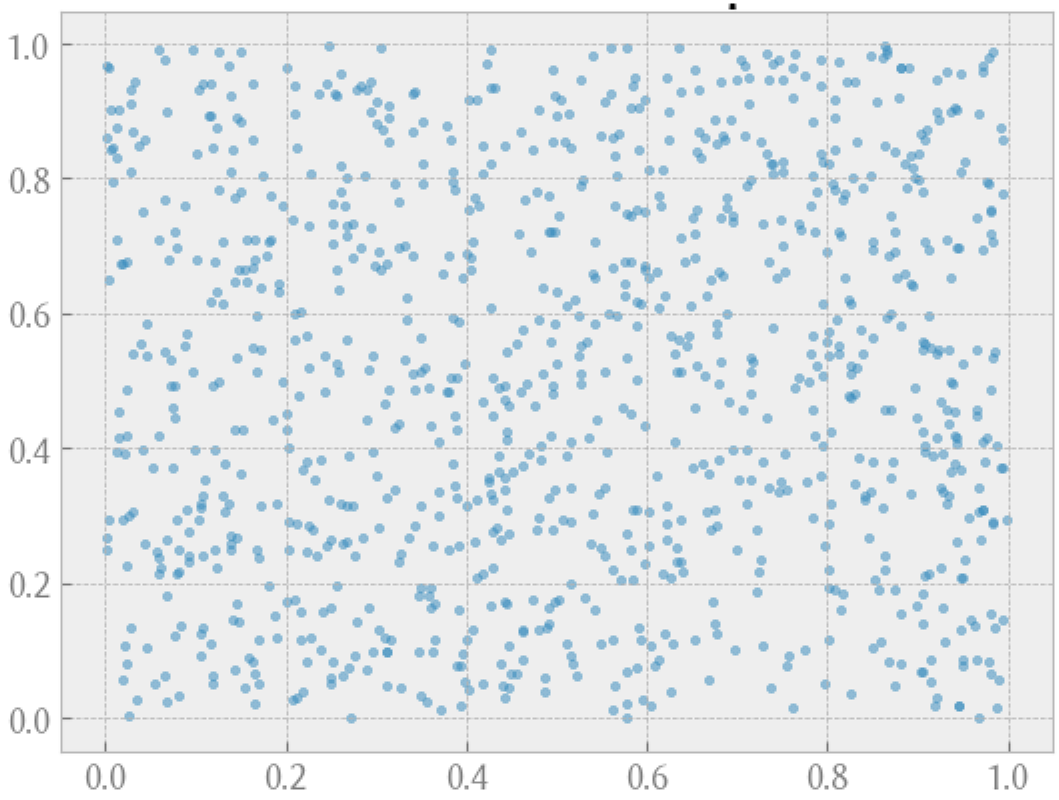


Daniel Hanson, NWCPP 19 March 2026

Overview: Outline

- Random number generators in the Standard Library
- Boost Statistical Distributions and Functions
- Some newer and lighter weight alternatives to Standard Library random number generators (PCG and Xoshiro)
- Boost Accumulators: Provide incremental statistical computations
- Proposal for the C++29 Standard (P1708): *Basic Statistics*

Standard Library Random Number Generators



Source: Risk Engineering

https://risk-engineering.org/notebook/monte-carlo-LHS.html

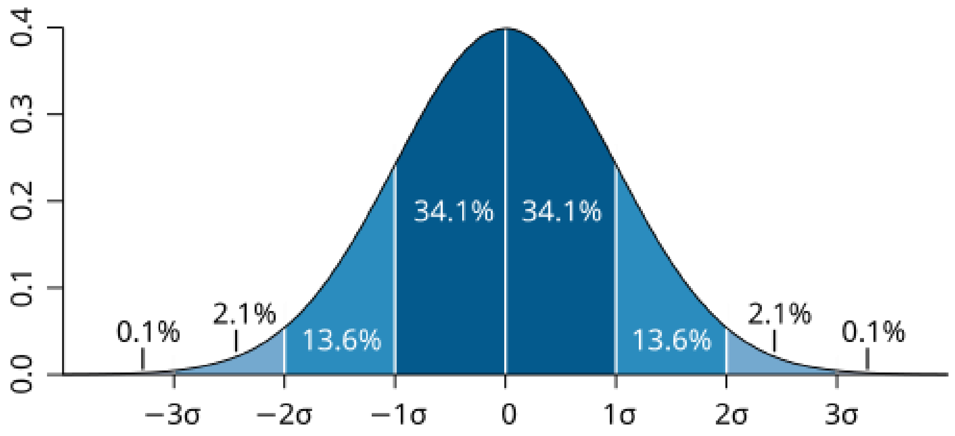
Random Number Generators

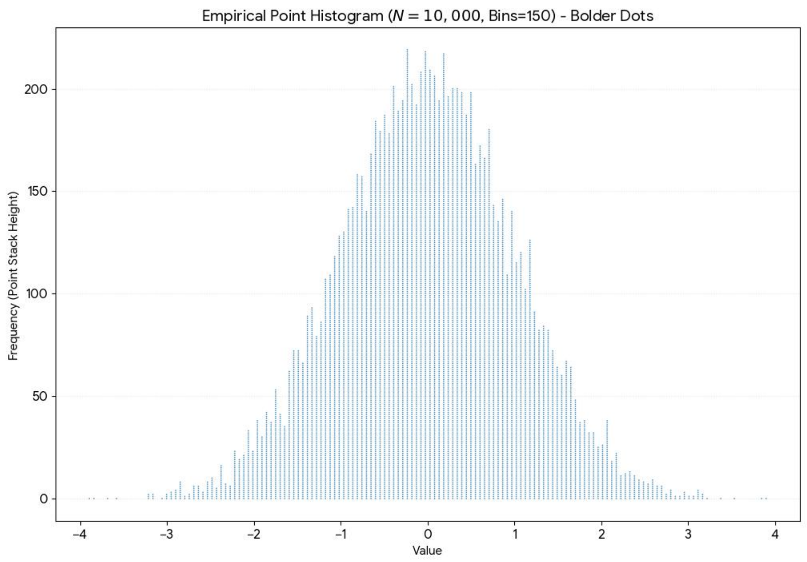
- Random number generation is widely used for stochastic simulation

- Applications in areas such as:
  - Finance (risk management, option pricing)
  - Branching processes in cancer growth and treatment modeling
  - Reliability testing in aviation and aerospace engineering
  - Simulations to "test" building integrity vs earthquakes
  - Particle physics
  - Computer graphics for video games

Random Number Generators

- For a given seed (or more generally, state initialization)
- A deterministic series of numbers that "look random"
- With frequencies that follow a particular probability distribution
- Example: Normal distribution (centered at mean = 0, take σ = 1)





Graph 1: https://en.wikipedia.org/wiki/Normal_distribution

Graph 2: Generated by Google Gemini

Distributional Random Number Generation

- Added to the C++ Standard Library in C++11
- Adapted from the Boost Random Number Generator Library
- The purpose is to generate random numbers from a particular distribution (e.g. normal, chi-squared, Poisson etc)
  - Deterministic sequence of “random-looking” values
  - Based on a seed value
  - Advantage: Reproducible for a known seed (suitable for simulations)
  - Disadvantage: Reproducible for a known seed (not suitable for cryptography)

Engines and Distributions

- **Engines** serve as a *stateful source of randomness*. They are function objects that generate random unsigned integer *values* that are uniformly distributed.

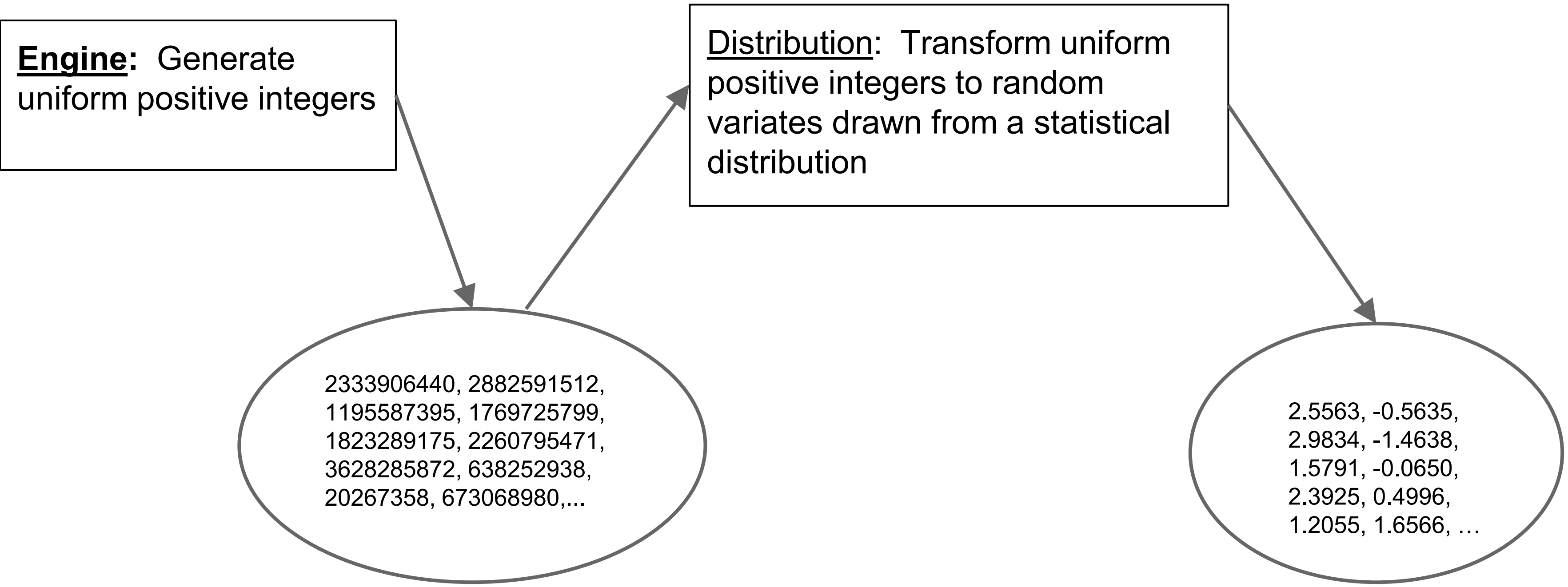
- **Distributions** apply transformations to these random integer values into random *numbers* that are distributed according to a user-supplied probability distribution

---

Nicolai Josuttis, The C++ Standard Library (2nd edition), Addison Wesley, 2012 (paraphrased)

§ 17.1: Random Numbers and Distributions (excellent reference)

Visually, combining an *engine* with a *distribution*:



Standard Library Engine Examples

- Selected engine examples…

Engine	Description	Period	Relative Performance
`std::default_random_engine`	Implementation-defined (linear congruential on MSVC, gcc)	$2^{31} - 2$ (MSVC, gcc)	Fast
`std::minstd_rand`	Linear Congruential	$2^{31} - 2$	Fast
`std::ranlux48`	Discard Block	$\sim 10^{170}$	Slow
`std::mt19937_64`	Mersenne Twister (64-bit)	$2^{19337} - 1$	Fast
`std::philox4x64`	Counter-based, C++26. "Stateless" in that a random number not dependent on its predecessor, but the engine requires a key value (technically state).	2^{256}	Fast (SIMD), std::async, etc But not as fast as `mt19937_64` on single CPU

Examples of Standard Library Distributions

- Uniform
  - `std::uniform_int_distribution`
  - `std::uniform_real_distribution`
- Bernoulli Family
  - `std::bernoulli_distribution`
  - `std::binomial_distribution`
- Poisson Family
  - `std::poisson_distribution`
  - `std::exponential_distribution`
- Normal Family (See next slide)
- Empirical non-parametric distributions

Normal (related) Distributions Category

- `std::normal_distribution`
- `std::lognormal_distribution`
- `std::chi_squared_distribution`
- `std::cauchy_distribution`
- `std::student_t_distribution`
- `std::fisher_f_distribution`

Example 1: `std::default_random_engine` (no probability distribution yet)

```
// An engine requires an integer seed value, e.g. 100:
std::default_random_engine def{100};

for (int i = 0; i < 10; ++i)
{
    // Random integers generated directly from the engine,
    // accessed as a functor (implemented on each std lib engine):
    cout << def() << " ";    // Reproducible deterministic sequence
                             // for each seed value
}

// MSVC: 2333906440 2882591512 1195587395 1769725799 1823289175
//       2260795471 3628285872 638252938 20267358 673068980

// gcc:  1680700 330237489 1203733775 1857601685 594259709
//       1923970613 1512819812 1903683451 1996387951 1007791729
```

Example 1: `std::default_random_engine` + `std::uniform_real_distribution`

```cpp
// Use the default_random_engine with continuous
// uniform distribution on [0, 1):

std::uniform_real_distribution<double> unif_rand_dist{0.0, 1.0};

std::vector<double> unifs(10);

// Generate 10 uniform variates from [0.0, 1.0) and set
// as elements in the vector 'unifs':
for (double& x : unifs)
{
    x = unif_rand_dist(def);
}
// MSVC:  0.186467 0.210108 0.45274 0.870143 0.063681
//        0.624312 0.523348 0.562296 0.0058172 0.307423
// gcc:   0.941637 0.457211 0.970019 0.769819 0.684224
//        0.677271 0.0436495 0.692878 0.391896 0.119059
```

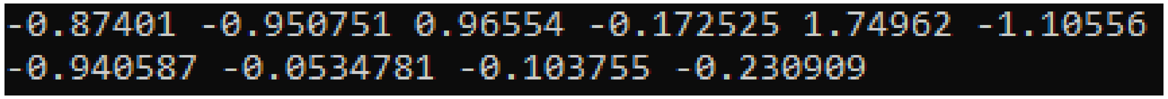
Example 2: `std::mt19937_64` (non-deterministic seed) & Standard Normal Dist

- Example: Standard Normal Distribution (mean = 0, standard deviation = 1)
- **`std::random_device`**: Seed is no longer deterministic

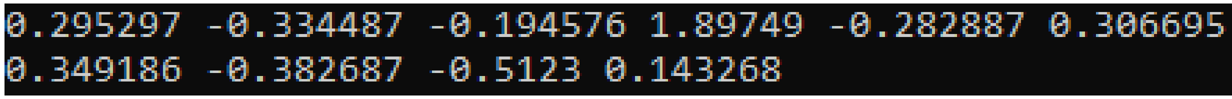
```
// Non-deterministic seed (kernel-provided entropy source*)
std::random_device rd;
std::mt19937_64 mt{rd()};


//
std
```

```
// Can also use default form, mean = 0, std dev = 1, T = double:
std::normal_distribution<> st_norm_rand_dist{};
```

```
std::vector<double> norms(10);
// Generate 10 standard normal variates and set as
// elements in the vector 'norms':
for (double& x : norms)
{
    x = st_norm_rand_dist(mt);
}
for (double x : norms)
{
    cout << x << " ";
}
```



```
-0.87401 -0.950751 0.96554 -0.172525 1.74962 -1.10556
-0.940587 -0.0534781 -0.103755 -0.230909
```



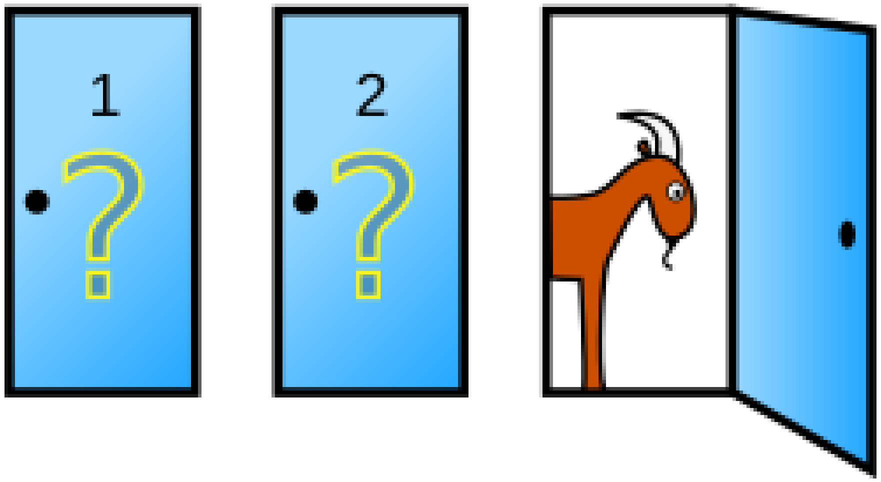
```
0.295297 -0.334487 -0.194576 1.89749 -0.282887 0.306695
0.349186 -0.382687 -0.5123 0.143268
```

Monty Hall Paradox

- Here is a fun little problem: Recall the show “Let’s Make a Deal”, hosted by Monty Hall, 1963-76

Monty Hall Paradox

- Contest: Three doors marked number 1, 2, and 3
  - Behind one of the doors would be the prize, such as a new car
  - While behind the other two doors, there were "zonk" prizes, such as a goat (i.e., you lose)
- When the contestant would make a choice, say Door #1, Monty would have one of the other doors opened, revealing a goat.



- Monty would then ask the contestant whether s/he would like to switch the choice to the remaining door.
- This gave rise to what became a somewhat famous math problem called "The Monty Hall Paradox": If you were giving the chance to switch, should you?
- It turns out, the answer is yes, as the probability of your winning the car, rather than the goat, would double from 1/3 to 2/3, a rather non-intuitive result

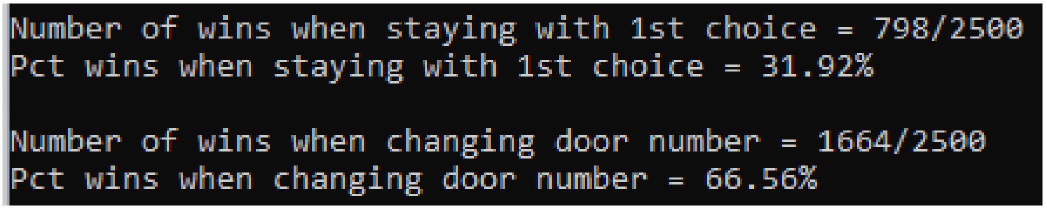
---

Image: https://en.wikipedia.org/wiki/Monty_Hall_problem

Monty Hall Paradox

- This can be solved in C++; define:
  - Two distinct Mersenne Twister 64-bit random engines (different seeds)
  - One *discrete* uniform distribution on [1, 2, 3] $\left(P(X = x) = \frac{1}{3}, x = 1, 2, 3\right)$

```cpp
mt19937_64 mt_car{seed_car_position};                      // Door where car is located
mt19937_64 mt_choice{seed_choice_position};                // Initial door choice
uniform_int_distribution<> ud{1, 3};

int wins = 0;       // Number of wins (to be incremented)

// Case where contestant does not change choice
for (unsigned i = 1; i <= n_sim; ++i)  // n_sim = Number of simulations from input (exmpl: n_sim = 2500)
{
    if (ud(mt_choice) == ud(mt_car))
        ++wins;                        // wins/n ~= 33%
}

wins = 0;       // Reset counter

// Case where contestant *does* change choice
for (unsigned i = 1; i <= n_sim; ++i)
{
    if (ud(mt_choice) != ud(mt_car))   // => Contestant changes choice to door with the car
        ++wins;                         // wins/n ~= 67%
}
```



```
Number of wins when staying with 1st choice = 798/2500
Pct wins when staying with 1st choice = 31.92%

Number of wins when changing door number = 1664/2500
Pct wins when changing door number = 66.56%
```

Boost Statistical Distributions

$$F(x) = P(a \leq x \leq b) = \int_a^b f(x)dx \geq 0$$

Boost Statistical Distributions

- Not random, but provide deterministic calculation of:
  - the *probability density function (PDF)*
  - the *cumulative distribution function (CDF)*
  - and the *quantile function*
  - for a wide variety of probability distributions

- These are provided as generic non-member functions common to all the distribution types in Boost

- Demonstrated by example next

- Straightforward, intuitive, and easy to use…

Boost Statistical Distributions

- Examples:

```
#include <boost/math/distributions/students_t.hpp>
#include <boost/math/distributions/normal.hpp>
using boost::math::students_t;
using boost::math::normal;

// Construct a students_t distribution with 4 degrees of freedom:
students_t t_dist{4};

// Construct a normal distribution with mean 0 and variance 1:
normal std_normal{0.0, 1.0};           // Default: std_normal{};

double n_pdf = pdf(std_normal, 0.0);          // 0.3989
double n_cdf = cdf(std_normal, 0.0);          // 0.5
double t_pdf = pdf(t_dist, 0.0);              // 0.375
double t_cdf = cdf(t_dist, 0.0);              // 0.5

// The fifth percentile of the standard normal distribution:
double norm_five_pctle = quantile(std_normal, 0.05);    // -1.64485

// The 95th percentile of a t distribution with 4 dof:
double t_95_pctle = quantile(t_dist, 0.95);             // 2.1318
```

Boost Math Toolkit: Statistical Distributions

A wide selection of univariate statistical distributions and functions that operate on them

https://www.boost.org/doc/libs/latest/libs/math/doc/html/dist.html

- Arcsine Distribution
- Bernoulli Distribution
- Beta Distribution
- Binomial Distribution
- Cauchy-Lorentz Distribution
- Chi Squared Distribution
- Exponential Distribution
- Extreme Value Distribution
- F Distribution
- Gamma (and Erlang) Distribution
- Geometric Distribution
- Hyperexponential Distribution
- Hypergeometric Distribution
- Inverse Chi Squared Distribution
- Inverse Gamma Distribution
- Inverse Gaussian (or Inverse Normal) Distribution
- Laplace Distribution
- Logistic Distribution
- Log Normal Distribution
- Negative Binomial Distribution
- Noncentral Beta Distribution
- Noncentral Chi-Squared Distribution
- Noncentral F Distribution
- Noncentral T Distribution
- Normal (Gaussian) Distribution
- Pareto Distribution
- Poisson Distribution
- Rayleigh Distribution
- Skew Normal Distribution
- Students t Distribution
- Triangular Distribution
- Uniform Distribution
- Weibull Distribution

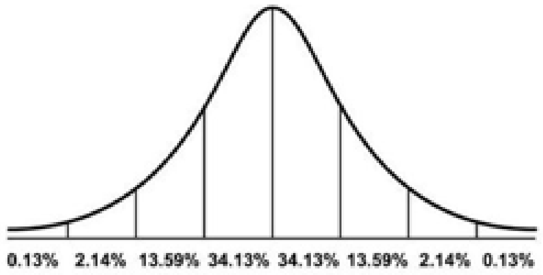
Using a Boost Distribution for Random Number Generation

- There are 17 closed-form distributions in the Standard Library from which to generate random numbers
- There are 34 distributions in Boost
- What if we need random number generation from a distribution in Boost but not in the Standard Library?
- Typical problem: Need random numbers from a distribution that can fit skewness and kurtosis (fat tails)

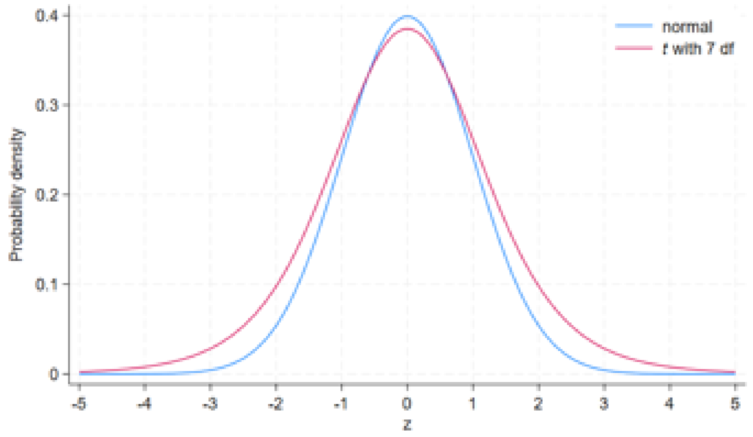
Using a Boost Distribution for Random Number Generation

- Skewness and Kurtosis (first)

  - Kurtosis measures the likelihood of extreme occurrences
    - "Normal" kurtosis



    - High kurtosis



Using a Boost Distribution for Random Number Generation

- Example: Non-Central t distribution
    - Can have skewness
    - Can have (high) kurtosis
    - Example: Common in financial returns
    - $\nu = 10$ degrees of freedom; $\delta =$ non-centrality parameter

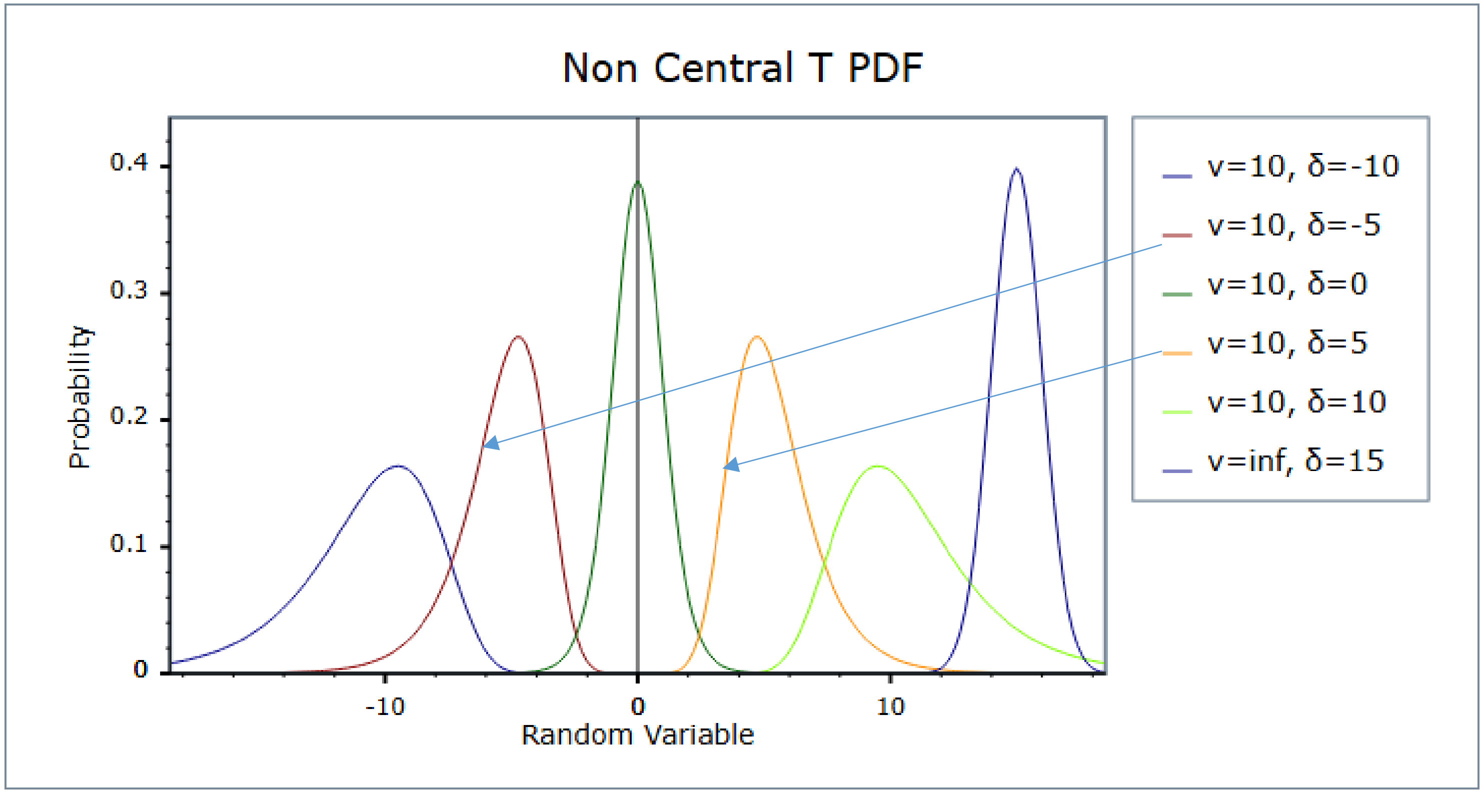


Image: Boost Documentation of the non-central t distribution

Using a Boost Distribution for Random Number Generation

- Create a class template **`std_non_central_t_dist`**
- This will act like a Standard Library distribution function
- The way it works:
  - Generate random draws from a Std Library uniform (0, 1) (real) distribution
  - Apply the Probability Integral Transformation Theorem
    - ➢ Feed random uniform variates into the non-central t inverse CDF
    - ➢ The inverse CDF is the quantile function in Boost
    - ➢ Results will be random numbers drawn from a non-central t distribution

Using a Boost Distribution for Random Number Generation

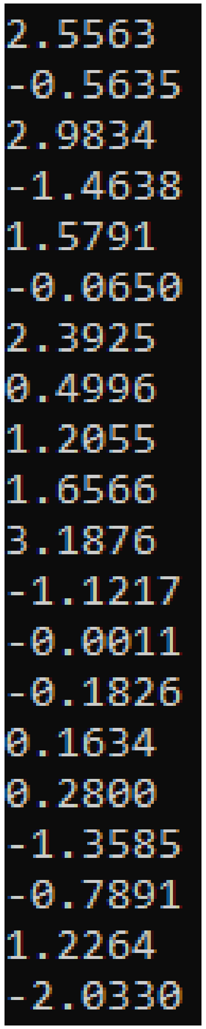
- We can write a distribution class template similar to Standard Library form:

```
  template<typename T = double>
class std_non_central_t_distribution requires std::floating_point<T>
  {
  public:
      // nu = degrees of freedom, delta = non-centrality parameter
      std_non_central_t_distribution(T nu, T delta) :nu_{nu}, delta_{delta}, nctd_{nu_, delta_} {}
      // Functor similar to <random> in the Standard Library
      // (for simplicity just use mt19937_64 engine):
      T operator()(std::mt19937_64& mt)
      {
          auto unif = ud_(mt);    // Next in random unif[0, 1) sequence
          return boost::math::quantile(nctd_, unif);
      }
  private:
      T nu_, delta_;
      boost::math::non_central_t_distribution<T> nctd_;
      std::uniform_real_distribution<T> ud_{0.0, 1.0};
  };
```

Using a Boost Distribution for Random Number Generation

- And then, we can use the distribution to generate random numbers, just like we did with Standard Library distributions:

```cpp
// Similar look and feel as functions in <random>:
std::mt19937_64 mt{100};
std_non_central_t_distribution<> st{3.0, -0.05};

// Wrap in lambda for use in the generate algo:
auto nc_t_gen = [&mt, &st]()
{
    return st(mt);
};

std::vector<double> nc_t_vals(20);  // nc = non-central
std::ranges::generate(nc_t_vals, nc_t_gen);

cout << std::fixed << std::setprecision(4);

// View results:
for (auto s : nc_t_vals)
{
    cout << s << " " << "\n";
}
```



```
2.5563
-0.5635
2.9834
-1.4638
1.5791
-0.0650
2.3925
0.4996
1.2055
1.6566
3.1876
-1.1217
-0.0011
-0.1826
0.1634
0.2800
-1.3585
-0.7891
1.2264
-2.0330
```

Some Alternatives to the `mt19937_64` Engine

Alternatives to the `mt19937_64` Engine

- The Mersenne Twister algorithm and the std::mt19937_64 engine is generally considered the best option in the Standard Library
  - Massively long period, avoiding repetition ($2^{19937} - 1$)
  - Better performance than other options with sufficiently long periods

- However, MT object/state sizes are large
  - Cannot entirely fit into CPU registers
  - Requires more cycles to move from RAM -> cache -> registers
  - Similar issues with the Mersenne Twister 32-bit version

- Two new open-source libraries offer (potential) improvements:
  - PCG (Permuted Congruential Generator)
  - Xoshiro (XOR/shift/rotate)

Alternatives to the `mt19937_64` Engine

- A comparison table of challengers to std::mt19937_64:

Generator	Period	State Size (bits)	Register Fit
PCG-XSH-RS 64	2^{128}	128	Yes (2)
xoshiro256++	$2^{256} - 1$	256	Yes (4)
std::mt19937_64	$2^{19937} - 1$	20032	No

Source: A PRNG Shootout, ©2026 Sebastiano Vigna, https://prng.di.unimi.it/

- A period of even 2^{128}, while dwarfed by $2^{19937} - 1$ for the Mersenne Twister, is still
  - An astronomically large number: $\sim 3.4 \times 10^{38}$
  - Still effectively infinite for just about any practical computational task

PCG and Xoshiro

- Using PCG and Xoshiro
  - Create a PCG or Xoshiro engine object
  - Use as arguments for Standard Library distributions as before
- There are various constructor overloads
- Introductory examples: use the respective single seed argument overload, similar to Standard Library random engines

```
// PCG:
pcg64_fast pcg_rng{42};                          // Seed = 42
std::chi_squared_distribution<> cs{15}; // 15 dof
double next_pcg = cs(pcg_rng);

// Xoshiro:
XoshiroCpp::Xoshiro256PlusPlus xpp_rng{42};
std::binomial_distribution<> bd{20, 0.45};
double next_xsh = bd(xpp_rng);
```

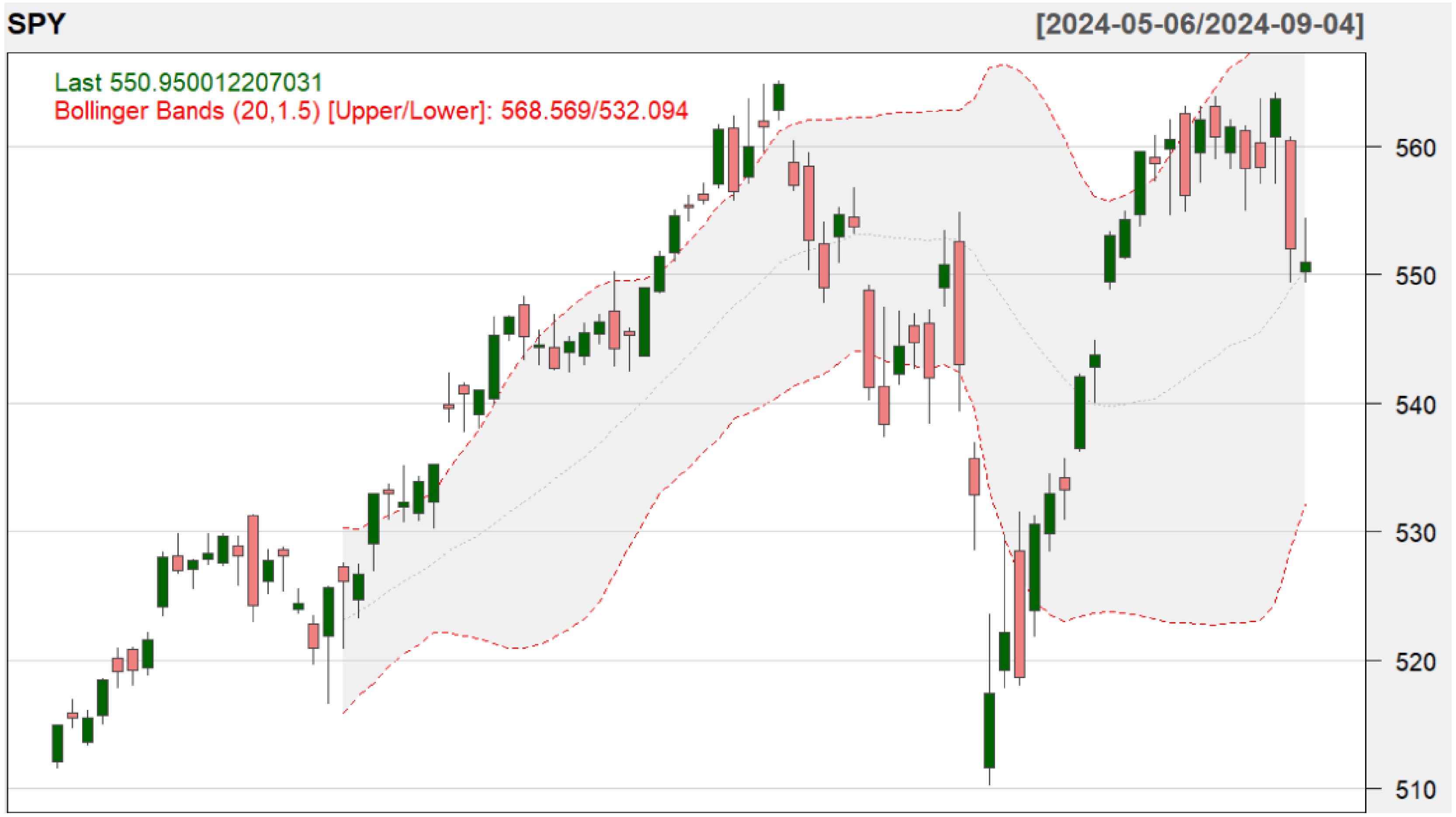
- Full examples in sample code

`std::mt19937_64` vs PCG and Xoshiro: Performance

- Random numbers drawn over 10-years
- Annual, monthly, and weekly time steps
- 50,000 random paths
- Mixed results – your mileage may vary

	Time Steps	Serial – time elapsed (ms)	std::async – time elapsed(ms)
std::mt19937_64	10	52.3	96.5
	120	179.7	98.0
	520	710.6	158.5
PCG: pcg64_fast	10	17.8	75.6
	120	176.5	107.1
	520	588.0	151.6
Xoshiro: Xoshiro256PlusPlus	10	23.6	92.2
	120	128.7	102.9
	520	580.3	136.7

Boost Accumulators



Accumulators

- The Boost Accumulators library is a framework for incremental statistical computations.
- The library deals primarily with the concept of an *accumulator*, which is a primitive computational entity that accepts data one sample at a time and maintains some internal state, e.g.,
  - Max
  - Min
  - Mean
  - Variance
  - etc
- Accumulators are grouped within an *accumulator set*

Accumulators

- The Accumulators framework uses the following pattern:
    - User builds **`accumulator_set<...>`** by selecting computations in which s/he is interested (mean, variance, etc)
    - User provides data into the **`accumulator_set<...>`** one data element at a time
    - The **`accumulator_set<...>`** computes the requested quantities while possibly caching intermediate results
- The template parameters are as follows:
    - **`<typename Sample, typename Features, typename Weight>`**
    - *Sample*: The type of data that will be accumulated (eg **`double`**)
    - *Features*: A sequence of features to be calculated (min, variance, etc)
    - *Weight*: The type of the (optional) weight parameter (e.g. weighted mean)
- The following line declares an **`accumulator_set<...>`** that will accept a sequence of doubles one at a time and calculate the mean and variance:

```
accumulator_set<double , features<tag::mean, tag::variance>> acc ;
```

- Once an **`accumulator_set<...>`** is defined, sample data is fed into the accumulator and the specified statistics are computed

Accumulators

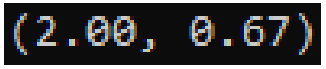
- Example: Mean and Variance of a data set

```cpp
#include <boost/accumulators/statistics/mean.hpp>
#include <boost/accumulators/statistics/variance.hpp>
```

```cpp
namespace bacc = boost::accumulators;
bacc::accumulator_set<double, features<tag::mean, tag::variance>> acc;

// Again, push some data into the accumulator . . .
acc(1.0);
acc(2.0);
acc(3.0);

// Display the results:
cout << '(' << extract::mean(acc) << ", " << extract::variance(acc) << ")\n";
```

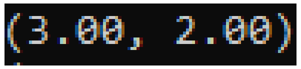
Note: The variance in this case is the population variance (max likelihood estimator): $\frac{1}{n} = \frac{1}{3}$, not $\frac{1}{n-1} = \frac{1}{2}$, which is used for the sample variance (e.g. in hypothesis tests).

$$\text{Var} = \frac{1}{3}[(1-2)^2 + = (1-1)^2 + (2-1)^2] = 1$$



```
(2.00, 0.67)
```

```cpp
acc(4.0);
acc(5.0);

cout << '(' << extract::mean(acc) << ", " << extract::variance(acc) << ")\n";
```



```
(3.00, 2.00)
```

Accumulators

- Rolling Windows

- Example: Moving average and moving variance of a series of 25 daily closing share prices
- Use accumulators to compute the running 5-day moving average and moving variance
- Use a vector here (and pretend it came in as input data, not hard-coded values):

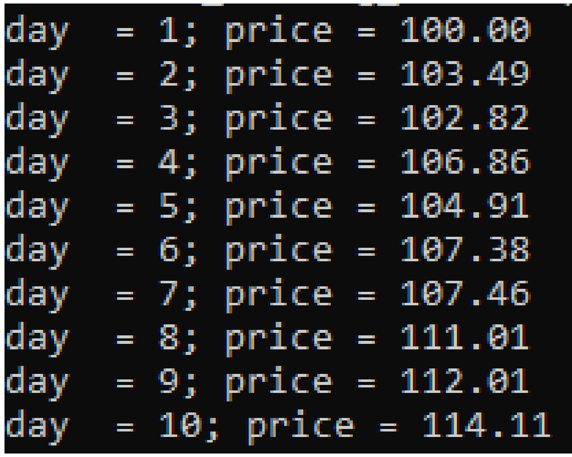
```cpp
#include <boost/accumulators/statistics/rolling_mean.hpp>
#include <boost/accumulators/statistics/rolling_variance.hpp>
// Series of 25 closing daily prices:
std::vector<double> prices
{
    100.00, 103.49, 102.82, 106.86, 104.91, 107.38, 107.46, 111.01, 112.01, 114.11,
    116.91, 121.74, 120.04, 120.24, 120.12, 120.61, 121.31, 119.25, 118.11, 120.36,
    117.36, 119.12, 119.36, 123.54, 123.42
};

// Take moving average and moving variance of most recent 5 prices => rolling_window::window_size = 5
    bacc::accumulator_set<double, bacc::stats<bacc::tag::rolling_mean, bacc::tag::rolling_variance>>
        prices_acc{bacc::tag::rolling_window::window_size = 5};
```

Accumulators

- Rolling Windows

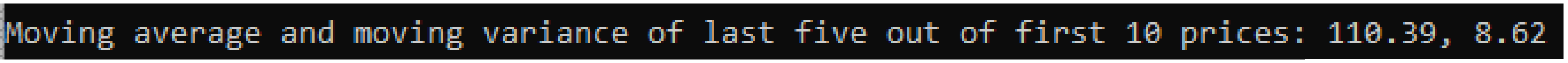
- Example: 5-day Moving average and moving variance over a series of 25 daily closing share prices
- Load the first 10 prices into the accumulator and get results over the last rolling 5-day period

```cpp
auto iter = prices.cbegin();
auto end = iter + 10;
int day = 0;
for (; iter != end; ++iter)
{
    ++day;
    cout << "day  = " << day << "; price = " << *iter << "\n";
    prices_acc(*iter);
}
```



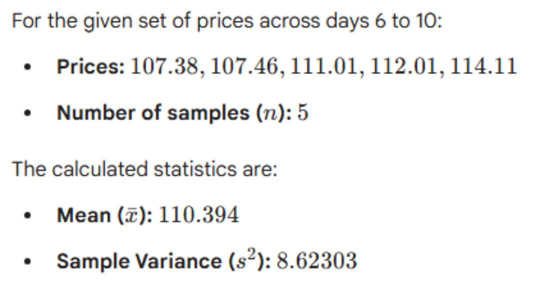
```
day  = 1; price = 100.00
day  = 2; price = 103.49
day  = 3; price = 102.82
day  = 4; price = 106.86
day  = 5; price = 104.91
day  = 6; price = 107.38
day  = 7; price = 107.46
day  = 8; price = 111.01
day  = 9; price = 112.01
day  = 10; price = 114.11
```

```cpp
cout << "Moving average and moving variance of last five out of first 10 prices: "
    << bacc::extract::rolling_mean(prices_acc) << ", "
    << bacc::extract::rolling_variance(prices_acc) << "\n\n";
```



```
Moving average and moving variance of last five out of first 10 prices: 110.39, 8.62
```

Remark: The rolling variance in Boost accumulators is the *sample variance*
So now, the sum of squared differences from the mean is divided by $n - 1$, not n



For the given set of prices across days 6 to 10:

- **Prices:** $107.38, 107.46, 111.01, 112.01, 114.11$
- **Number of samples** (n): 5

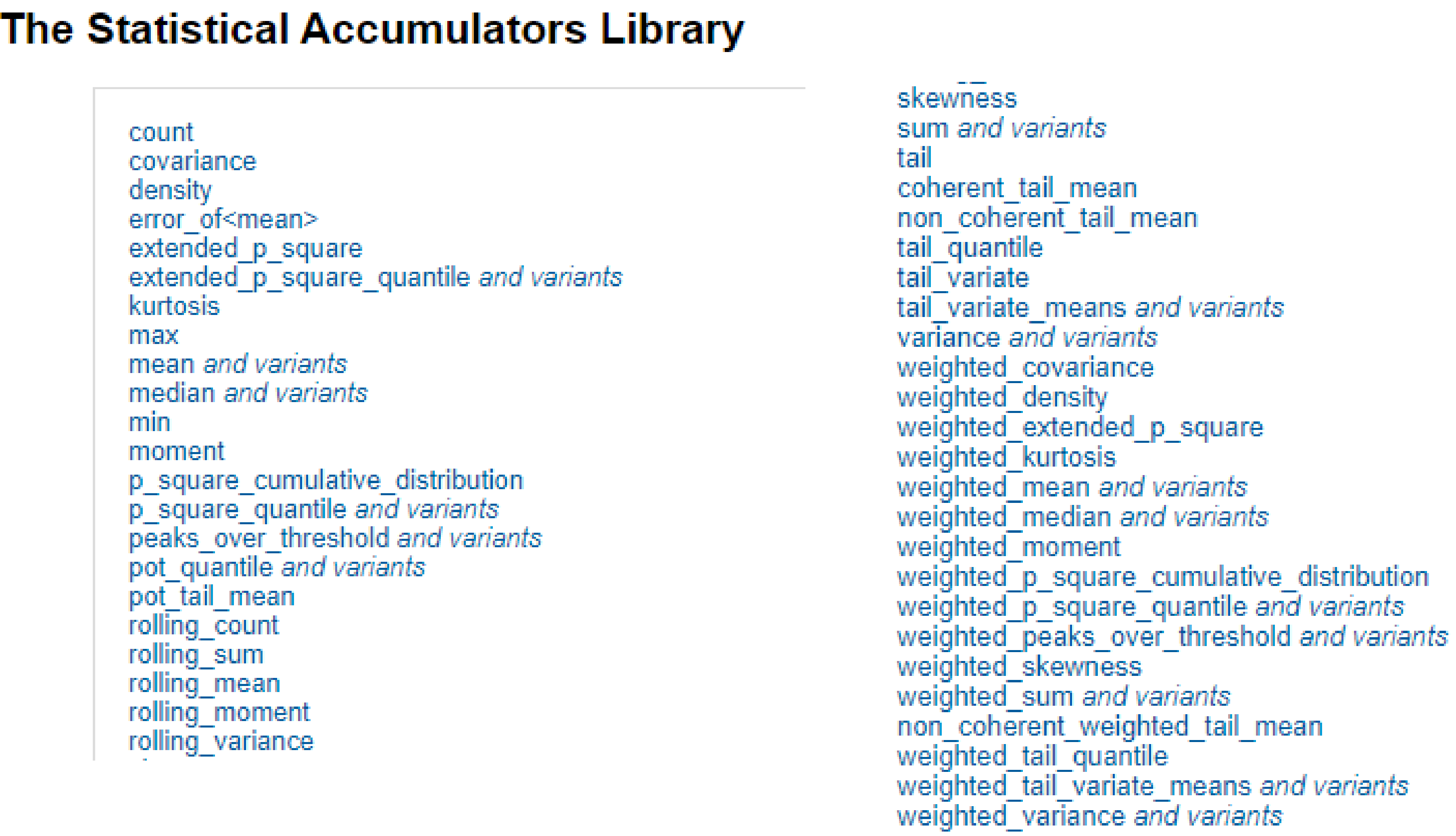
The calculated statistics are:

- **Mean** ($\bar{x}$): 110.394
- **Sample Variance** (s^2): 8.62303

Accumulators

- The complete list of statistical operators for Boost Accumulators is available here:

  https://www.boost.org/doc/libs/latest/doc/html/accumulators/user_s_guide.html



The Statistical Accumulators Library

count
covariance
density
error_of<mean>
extended_p_square
extended_p_square_quantile *and variants*
kurtosis
max
mean *and variants*
median *and variants*
min
moment
p_square_cumulative_distribution
p_square_quantile *and variants*
peaks_over_threshold *and variants*
pot_quantile *and variants*
pot_tail_mean
rolling_count
rolling_sum
rolling_mean
rolling_moment
rolling_variance

skewness
sum *and variants*
tail
coherent_tail_mean
non_coherent_tail_mean
tail_quantile
tail_variate
tail_variate_means *and variants*
variance *and variants*
weighted_covariance
weighted_density
weighted_extended_p_square
weighted_kurtosis
weighted_mean *and variants*
weighted_median *and variants*
weighted_moment
weighted_p_square_cumulative_distribution
weighted_p_square_quantile *and variants*
weighted_peaks_over_threshold *and variants*
weighted_skewness
weighted_sum *and variants*
non_coherent_weighted_tail_mean
weighted_tail_quantile
weighted_tail_variate_means *and variants*
weighted_variance *and variants*

P1708: Basic Statistics
A Proposal for C++29

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\bar{x}_{geom} = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}}$$

$$q_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n - 1}$$

$$= \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n - 1}}$$

Basic Probability C++29 Proposal

- Compute basic statistics for values in a range (P1708)
  - Frequently arise in scientific, industrial, and general applications
  - Are available in Python, R, Julia, MATLAB, et al
- Statistics that are defined in this proposal
  - Mean
    - ➢ Arithmetic mean and weighted arithmetic mean (sum divided by n)
    - ➢ Geometric mean and weighted geometric mean (n^{th} root of the product)
    - ➢ Harmonic mean and weighted harmonic mean (inverse of arithmetic mean of reciprocals)
  - Variance
    - ➢ Population variance (divide b $\bar{x}_{geom} = \sqrt[n]{\prod_{i=1}^{n} x_i}$ $\bar{x}_{harm} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$
    - ➢ Sample variance (divide by $n - 1$)
  - Standard Deviation: square roots of population and sample variances
  - Skewness (population, sample)
  - Kurtosis (population, sample)
  - Covariance (population, sample)

Basic Probability C++29 Proposal

- Not included in this proposal
  - Median
  - Mode
  - Quantile
  - “These more involved statistics are deferred to a future proposal”
- Accumulators (à la Boost) were also originally proposed but have been deferred as well

Basic Probability C++29 Proposal

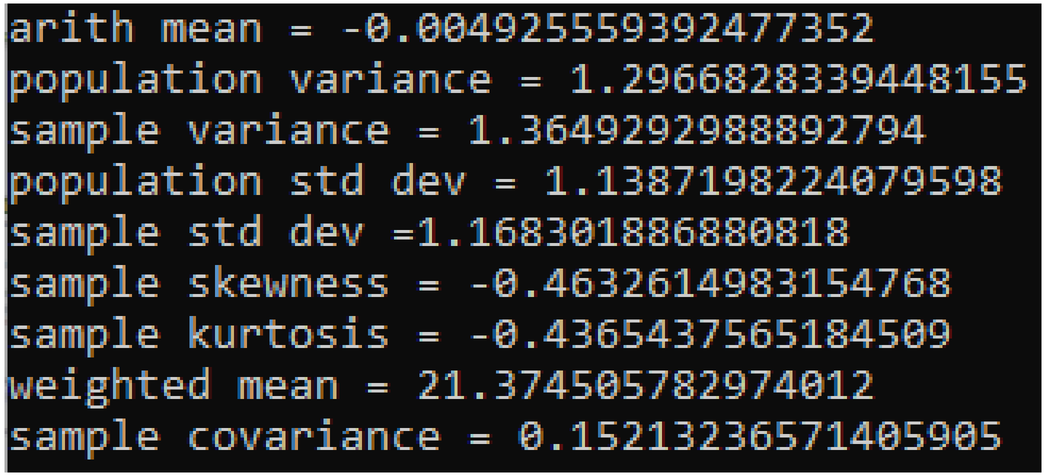
- Examples (can choose sample or population statistics)

```
std::vector<double> v;
v.reserve(20);

std::mt19937_64 mt{42};
std::normal_distribution<> nd{};        // N(0, 1)
std::ranges::generate_n(std::back_inserter(v), v.capacity(), [&]
    {
        return nd(mt);
    }
);


double mean_val = std::mean(v);
double samp_variance = std::variance(v, true);      // default: sample = true
double pop_variance = std::variance(v, false); // sample == false => population variance
double samp_std_dev = std::standard_deviation(v);   // default: sample = true
double samp_skewness = std::skewness(v);            // default: sample = true
double samp_kurtosis = std::kurtosis(v, {true, true});  // sample = true
                                                        // excess kurtosis = true
samp_kurtosis = std::kurtosis(v);   // default: sample = true, excess kurtosis = true
```

Basic Probability C++29 Proposal

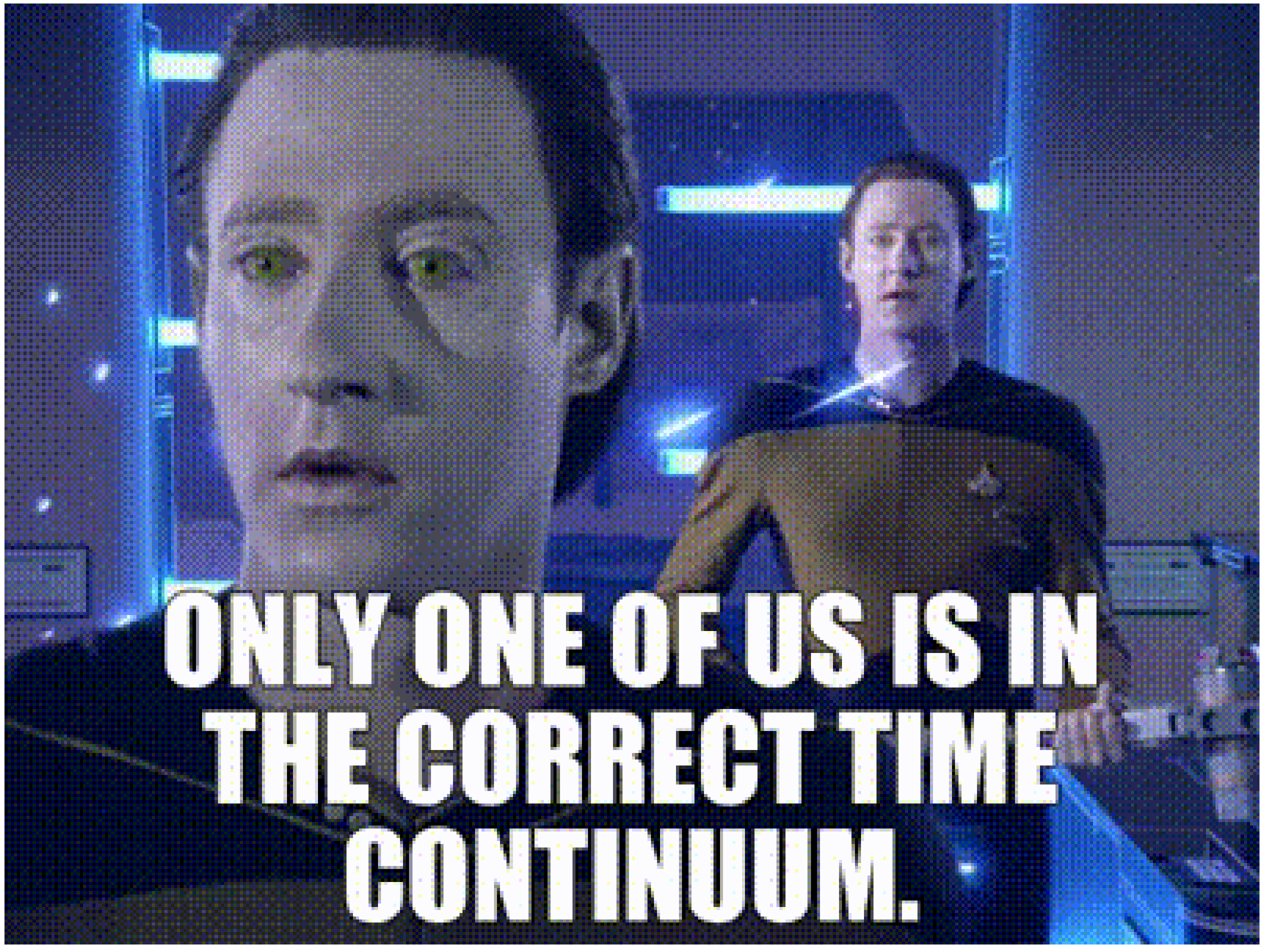
- Example: Covariance

```
// Another vector u, so we can compute covariance(v, u):
std::vector<double> u;
u.reserve(20);

std::ranges::generate_n(std::back_inserter(u), u.capacity(), [&]
    {
        return nd(mt);
    }
);

// Parallel execution policy is also an option:
double samp_covariance = std::covariance(std::execution::par, v, u);
```



```
arith mean = -0.004925559392477352
population variance = 1.2966828339448155
sample variance = 1.3649292988892794
population std dev = 1.1387198224079598
sample std dev =1.168301886880818
sample skewness = -0.4632614983154768
sample kurtosis = -0.4365437565184509
weighted mean = 21.374505782974012
sample covariance = 0.15213236571405905
```

Summary/Closing

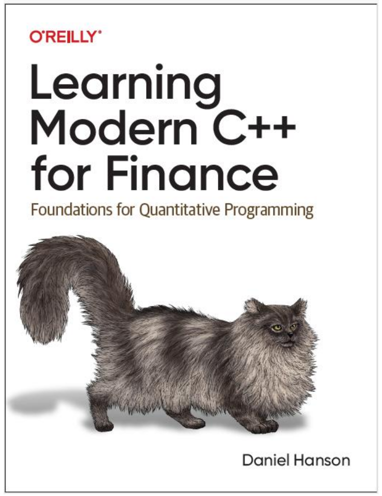


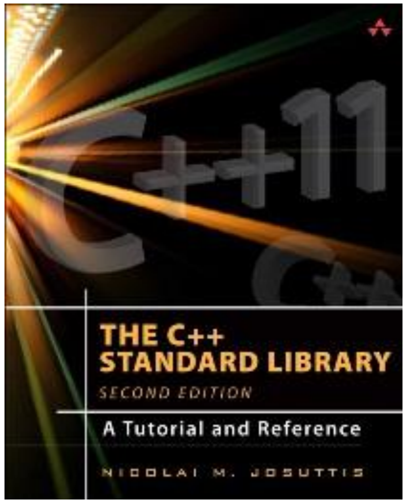
Some Closing Thoughts

- Between the Standard Library, Boost, and PCG/Xoshiro, C++ has a much-improved variety of options for probability and statistics
- C++ will probably never be as feature-rich mathematically as Python or R
- But it wouldn't hurt to add more
- Some probability-related ideas:
  - Incorporate all the Boost distributions in **`<random>`**
  - Add more four-parameter distributions
    - Generalized Lambda
    - Generalized Hyperbolic
    - Skew-t
  - Incorporate Boost(-like) Accumulators

References/Further Reading

- Hanson, *Learning Modern C++ for Finance* (O'Reilly, November 2024), Ch's 6 and 9 (shameless plug)
- Nicolai Josuttis, The C++ Standard Library (2E), Ch 17


O'REILLY
Learning Modern C++ for Finance
Foundations for Quantitative Programming
Daniel Hanson



- Geant4 Particle Physics Simulator Toolkit (CERN)
  https://geant4.web.cern.ch/
- Application of C++ in Computational Cancer Modeling (RNG, **`std::async`**)
  Ruibo Zhang, CppCon 2024: https://www.youtube.com/watch?v=_SDySGM_gJ8
- P1708 Basic Statistics proposal for C++29: https://wg21.link/P1708
- C++日本語リファレンス (Japanese Language C++ Reference):
  https://cpprefjp.github.io/reference/random/minstd_rand.html
  https://cpprefjp.github.io/reference/random/ranlux48_base.html
- John K Salmon: Parallel Random Numbers: As Easy as 1, 2, 3 (Philox64 generator):
  https://www.thesalmons.org/john/random123/papers/random123sc11.pdf

Wrap-up/ Contact

- Contact:
  - `daniel (at) cppcon.org` (Student Program Coordinator, CppCon)
  - https://www.linkedin.com/in/danielhanson/
- Thank You!

Questions?

- Questions?