COMPUTATIONAL FINANCE & RISK MANAGEMENT

UNIVERSITY *of* WASHINGTON

Department of Applied Mathematics

Math and C++ Innovations and Improvement

Presented to the Northwest C++ Users Group

May 2019

Daniel Hanson

Lecturer, Computational Finance & Risk Management

Dept of Applied Mathematics

University of Washington

Special Thanks To…

- Daniel Xia (UW Applied Math/Computational Finance) for contributions on parallel STL algorithms (completed comprehensive report as independent study project, Winter Quarter 2019)

- Jared Broad (CEO, QuantConnect – now based in Seattle) for graphics of random equity price scenarios

Overview

- Math in the new C++ Standards (C++11 and after)
  - Selected features useful in quant finance
  - Examples
- Math in the Boost Libraries
  - Boost Math Toolkit
  - Other Math and Math-Related Libraries
  - The Good, the Bad, and the Ugly

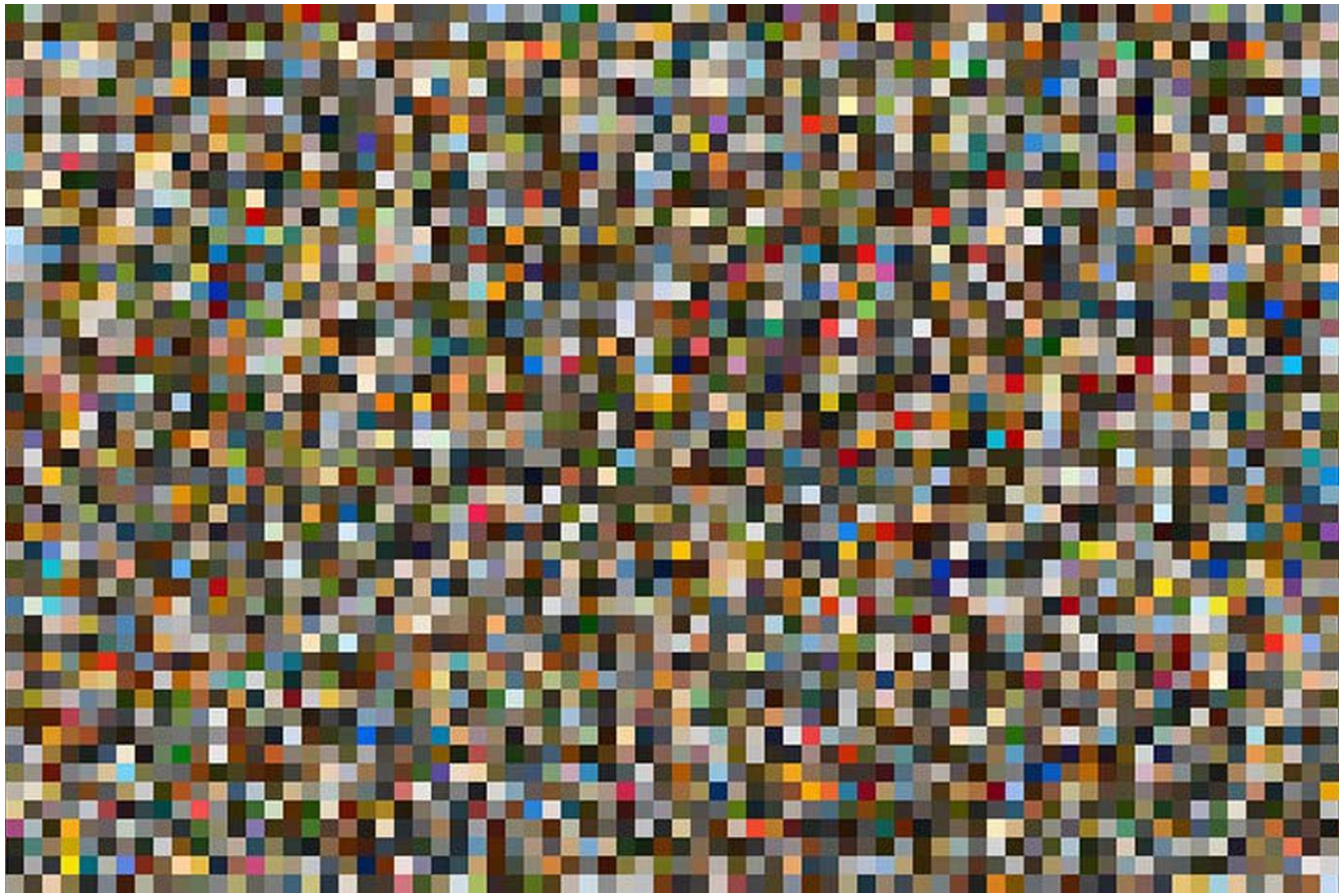
MATH IN THE NEW C++ STANDARD

C++11 and After

Some New Math-Friendly New Features in C++11 and After

- Random number generation from a wide variety of commonly used statistical distributions
- Lambda expressions
- Task-based parallelism
- Parallel STL algorithms

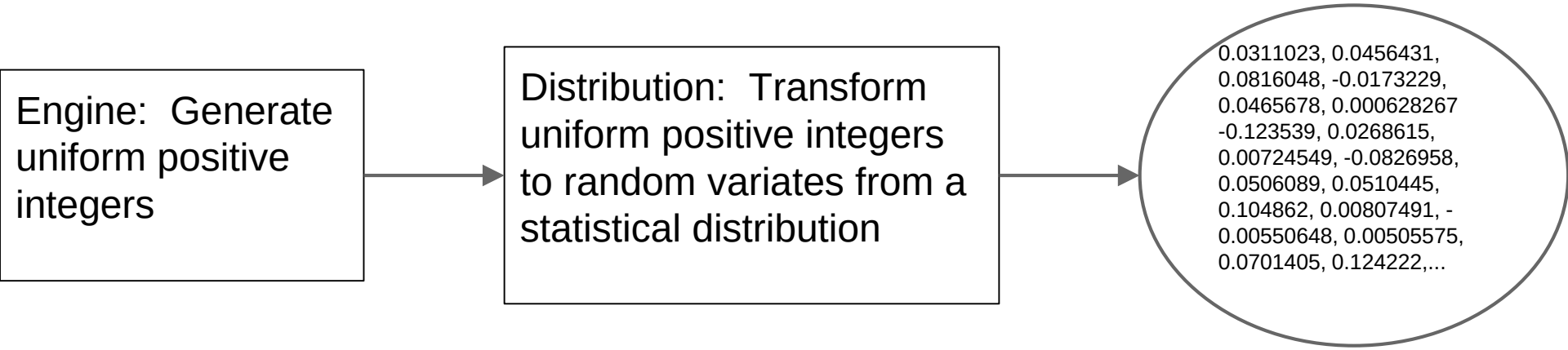
Random Number Generation
Lambda Expressions
Equity Price Simulation Example



Random Number Generation

- Uniform random number generator + transformation to distribution
- In C++ parlance: Engine + Distribution
- Engine: Generates a random sequence of integers
  - A variety of commonly used algorithms available
  - Includes Mersenne Twister: state of the art for pseudo-random number generation; used extensively in computational finance for Monte Carlo applications
- Distribution:
  - Actually transforms uniform variates to random draws from the desired distribution
  - For the normal distribution, uses the well-known Box-Muller method

Create random numbers by combining an *engine* with a *distribution*:



Examples of Available Distributions

- Uniform: typically on [0, 1] for the real number case – common in statistical applications
- Bernoulli (`bernoulli, binomial` etc)
- Poisson (`poisson, exponential` etc)
- Normal family (See next slide)
- Others

Normal (related) Distributions Category

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

Generate standard normal variates

```
#include <random>
#include <iostream>

using std::mt19937_64;
using std::normal_distribution;

// mtre = Mersenne Twister Random Engine, seed = 100:
mt19937_64 mtre(100);

cout << "Look at 1st few numbers in MT sequence: " << endl;
cout << mtre() << " " << mtre() << " " << mtre() << endl;
```

Output:

```
Look at 1st few numbers in MT sequence:
17952737041105130042 5969071622678286091 17739577640593830518
```

Generate standard normal variates (continued)

```
normal_distribution<> nd;          // see note at bottom of page
cout << "Look at 1st few numbers in standard normal sequence: " << endl;
cout << nd(mtre) << " " << nd(mtre) << " " << nd(mtre) << endl;
```

Output:

```
Look at 1st few numbers in standard normal sequence:
0.444734 0.60953 -0.00525884
```

Remark: Continuous distributions use double as the template parameter default for

```
template< typename RealType = double >
class normal_distribution;
```

Thus, we just put `normal_distribution<> nd;` to accept the default

Equity Price Generator

- A very common task in quant finance, and for options pricing in particular, is to generate random paths of stock prices
- Using what is called the risk neutral measure, we can use the standard lognormal result:

$$S_t = S_{t-1} e^{\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\varepsilon_t\sqrt{\Delta t}}$$

where S_t = underlying (eg equity) price at time t, Δt = time step year fraction, r = risk-free rate, σ = volatility, and $\varepsilon_t \sim \mathrm{N}(0, 1)$ iid.

- Our scenarios are then random realizations of price paths chopped up into time steps each of length Δt .
- With the new **<random>** feature in the Standard Library, this is a snap
- Before this, however, it wasn't!

Equity Price Generator (declaration)

```
#include <vector>


class EquityPriceGenerator
{
public:
    EquityPriceGenerator(double initEquityPrice, unsigned numTimeSteps,
    double timeToMaturity, double drift, double volatility);


    std::vector<double> operator()(int seed) const;


private:
    double yearFraction_;
    const double initEquityPrice_;
    const int numTimeSteps_;
    const double timeToMaturity_;
    const double drift_;
    const double volatility_;
};
```

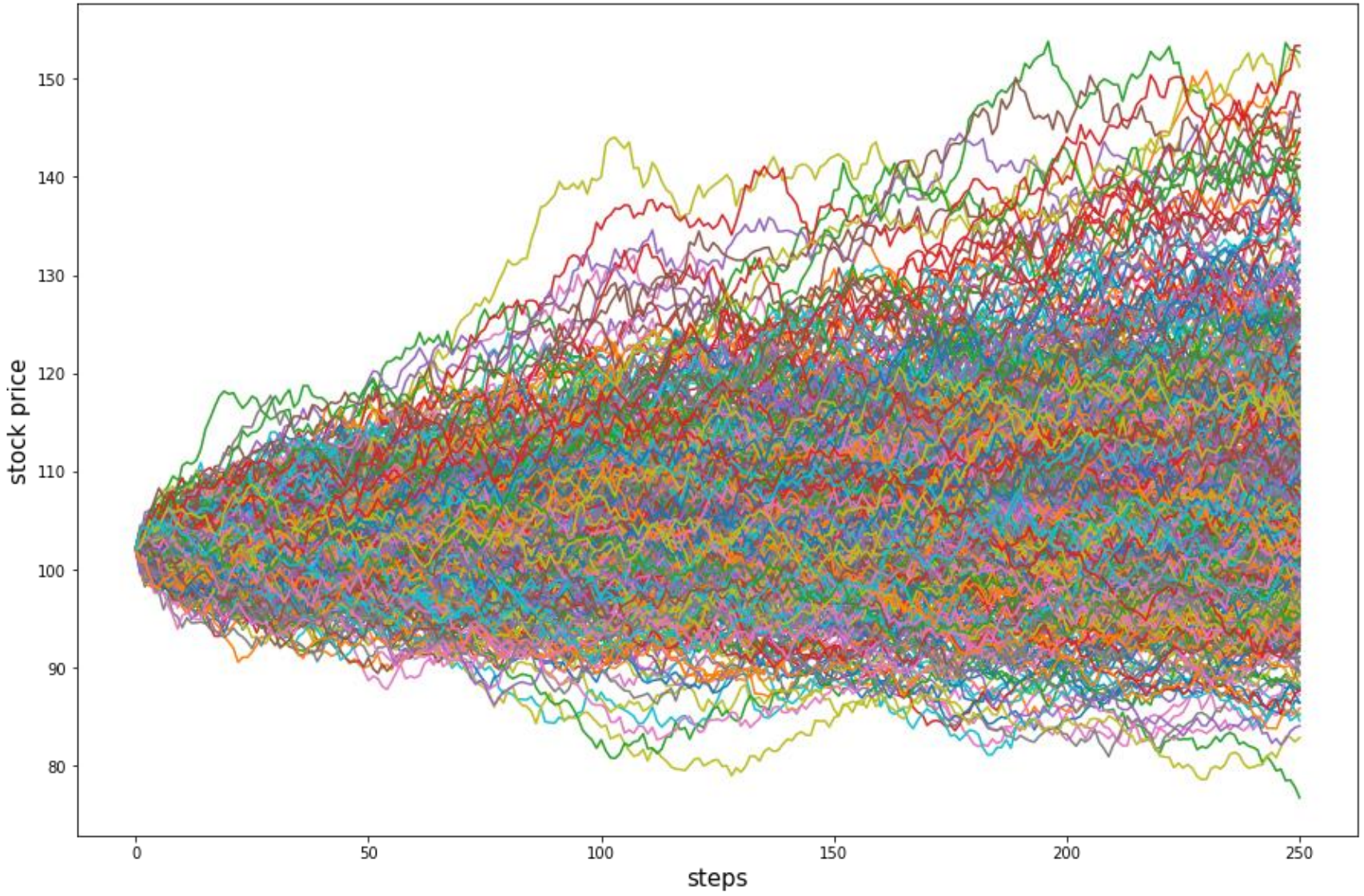
```cpp
#include "EquityPriceGenerator.h"
#include <random>
#include <algorithm>
#include <ctime>
#include <cmath>

using std::vector;
using std::mt19937_64;
using std::normal_distribution;
using std::exp;

EquityPriceGenerator::EquityPriceGenerator(double initEquityPrice, unsigned numTimeSteps, double timeToMaturity,
      double drift, double volatility) : initEquityPrice_(initEquityPrice), numTimeSteps_(numTimeSteps),
      timeToMaturity_(timeToMaturity), drift_(drift), volatility_(volatility),
      yearFraction_(timeToMaturity/numTimeSteps) {}

vector<double> EquityPriceGenerator::operator()(int seed) const
{
      vector<double> v;
      mt19937_64 mtre(seed);
      normal_distribution<> nd;
      auto newPrice = [this](double previousEquityPrice, double norm, double yearFraction) {
            double price = 0.0;
            double expArg1 = (drift_ - ((volatility_ * volatility_) / 2.0)) * yearFraction;
            double expArg2 = volatility_ * norm * sqrt(yearFraction);
            price = previousEquityPrice * exp(expArg1 + expArg2);
            return price;
      };
      v.push_back(initEquityPrice_);    // put initial equity price into the 1st position in the vector
      double equityPrice = initEquityPrice_;
      for (int i = 1; i <= numTimeSteps_; ++i)    // i <= numTimeSteps_ since we need a price at the end of the
      {                                           // final time step.
            equityPrice = newPrice(equityPrice, nd(mtre), yearFraction_);    // norm = nd(mtre)
            v.push_back(equityPrice);
      }
      return v;
}
```

Where the magic happens

Lambda Expression also convenient!

Equity Price Path Simulations

- Example: 10,000 simulated paths, initial share price = 100, each evolving over 250 time steps
- Graphically, our results might look something like this:



Task-Based Parallelism
Monte Carlo Option Pricing

Task-Based Parallelism

- Another common task in quant finance is to use Monte Carlo simulations, as we saw in the previous discussion, to price option contracts
- The simplest of these cases is a European equity option, which can only be exercised at the expiration date
- Using a set of equity price paths as shown previously, we only need to check the terminal price at time T for each scenario and compare it with the strike price to determine the payoff; viz, for a European call option:

  $\max(S_T - K, 0)$, where S_T = the terminal price, and K = the strike price

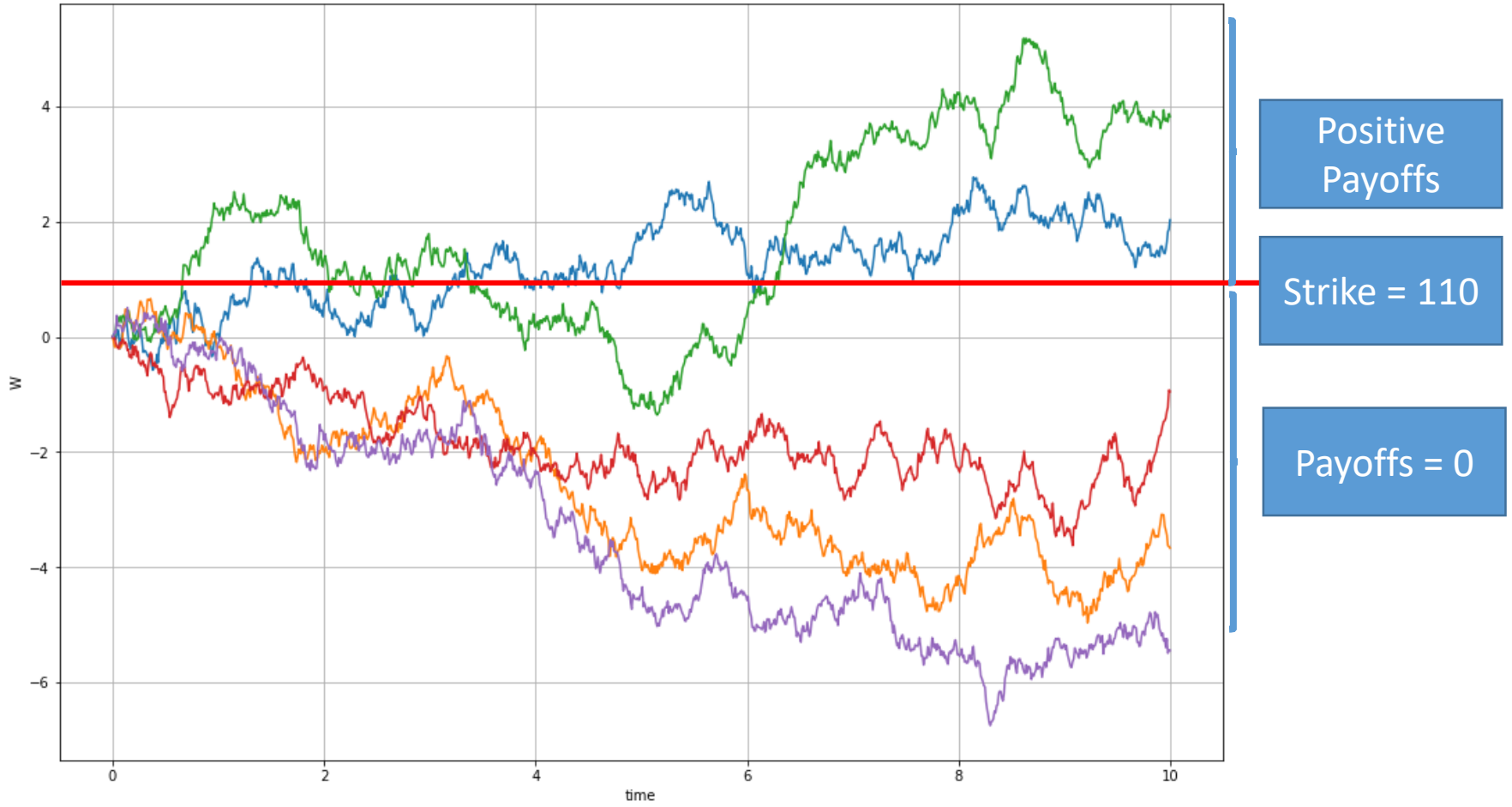
- The option price is then the mean of the payoffs discounted back to the value date ($t = 0$)

$$e^{-rT} \frac{1}{N} \sum_{j=1}^{N} \max(S_T^j - K, 0)$$

  where N = number of simulated equity price paths, and r is the risk-free interest rate

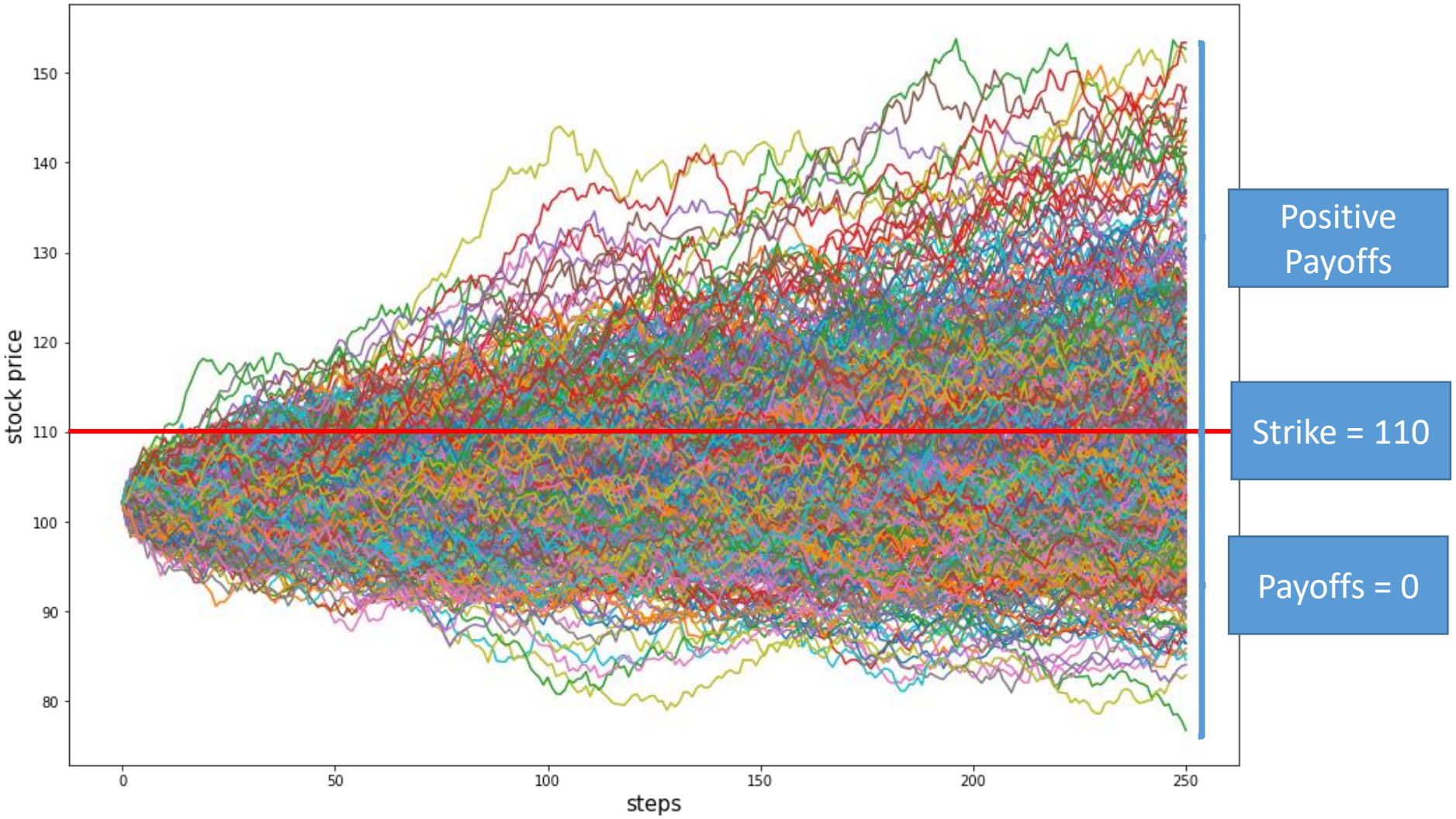
Monte Carlo Option Pricing

- We will price a European call option using the Monte Carlo method explained mathematically in the previous slide.
- A simple graphical example is shown below
  - Assume the vertical axis represents changes from an underlying asset currently valued at $100/share
  - The red line then represents a strike price or $110
  - The only positive payoffs at expiration will be the blue ($10) and green ($30) scenarios



Monte Carlo Option Pricing

- In reality, however, the number of scenarios run can be on the order of 10,000 – 100,000
- In our C++ example, we will treat each scenario as a threaded task, using **`std::future`** and **`std::async`**



Task-Based Parallelism

- As none of the paths "care" about any of the others, this is an *embarrassingly parallelizable* problem

- We can therefore generate each path in parallel and have no concern whatsoever for shared memory issues

- Task-based parallelism using **`std::future`** and **`std::async`** is a no-nonsense and fairly straightforward solution in C++

- Why mess with threading on one's own, when task-based parallelism encapsulates everything for us?
  - Quicker to implement
  - Less error-prone
  - Can actually be more efficient (Meyers: Effective Modern C++)

Option Calculations (Declaration)

- First look at the function declarations
- Also note the use of an enum class to indicate whether the option is a call or a put

```
#include "EquityPriceGenerator.h"
enum class OptionType
{
    CALL,
    PUT
};

class MCEuroOptPricer
{
public:
    MCEuroOptPricer(double strike, double spot, double riskFreeRate, double volatility,
        double tau, OptionType optionType, int numTimeSteps, int numScenarios,
        bool runParallel, int initSeed, double quantity);

    double operator()() const;
    double time() const;        // Time required to run calcutions (for comparison using concurrency)
private:
    void calculate_();          // Start calculation of option price

    // Private helper functions:
    void computePrice_();
    void generateSeeds_();
    double payoff_(double terminalPrice);

    // Compare results:  non-parallel vs in-parallel with async and futures
    void computePriceNoParallel_();
    void computePriceAsync_();
```

```
void MCEuroOptPricer::computePriceAsync_()
{
    EquityPriceGenerator epg(spot_, numTimeSteps_, tau_, riskFreeRate_, volatility_);
    generateSeeds_();

    using realVector = std::vector<double>;
    std::vector<std::future<realVector> > futures;
    futures.reserve(numScenarios_);

    for (auto& seed : seeds_)
    {
        futures.push_back(std::async(epg, seed));
    }

    std::vector<double> discountedPayoffs;
    discountedPayoffs.reserve(numScenarios_);

    for (auto& future : futures)
    {
        double terminalPrice = future.get().back();
        double payoff = payoff_(terminalPrice);
        discountedPayoffs.push_back(discFactor_ * payoff);
    }

    double numScens = static_cast<double>(numScenarios_);
    price_ = quantity_ * (1.0 / numScens) *
        std::accumulate(discountedPayoffs.begin(), discountedPayoffs.end(), 0.0);
}
```

Option Calculations (Summary and Results)

Expiry (years)	Time Steps	Scenarios	Time – Serial (sec)	Time - Parallel (Task)	Pct Drop in RT
1	12	10000	0.039	0.039	0%
1	120	50000	0.677	0.423	37.5%
5	120	50000	0.869	0.492	43.4%
5	600	50000	2.755	1.494	45.8%
5	600	100000	5.488	2.865	47.8%
10	600	100000	5.703	2.904	49.1%
10	1200	100000	10.818	5.925	45.2%

Option Calculations (Summary and Results)

- But wait, a 10-year option?!
  - True, not going to find an exchange traded 10-year equity option
  - However, 10+ year option products are found in practice in various forms, and Monte Carlo pricing models are commonly used:
    - Interest rate and FX swaptions
    - Hybrid exotic interest rate/FX structured products
    - Guaranteed investment products
      - Capital guarantees
      - Variable annuities
      - Fixed index annuities

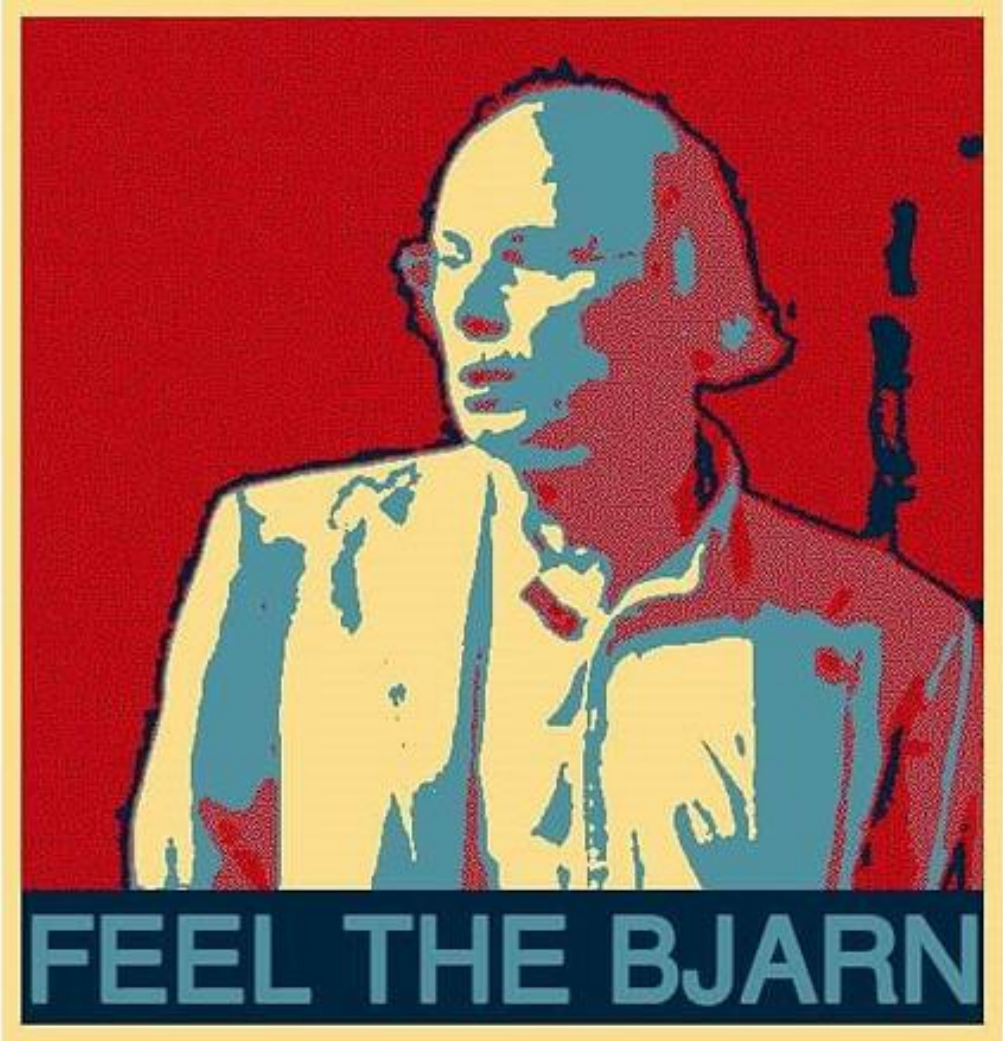
- The above examples are, in fact, where the power, speed, and scalability of C++ can become even more advantageous vs other programming languages

Option Calculations (Summary and Results)

- On a simple Surface Pro 4 running Windows 10, run time was cut on the order of 43-49%
- Would expect better on higher-performance machines (additional investigation in the works – stay tuned)
- Note that the spawning and management of threads is all contained in the task-based approach using **`std::future`**(s) and **`std::async`**

- *Open question*: Why do textbooks and reference books on C++, and classes offered at the undergraduate and graduate level, remain stuck in the past by ignoring this very powerful new tool?

Parallel STL Algorithms
Mathematical Examples



Parallel STL Algorithms

- Introduced with C++17
- Visual Studio is only major C++ compiler with (nearly) full implementation
- Many STL algorithms can be parallelized by including an execution policy argument
  - **`std::execution::seq`**: the execution may not be parallelized, only executed sequentially
  - **`std::parallel::par`**: the execution may be parallelized, but executions within the same thread are indeterminately sequenced with respect to each other
  - **`std::parallel::par_unseq`**: the execution may be parallelized and vectorized, executions within the same thread can be executed unsequenced with respect to each other
- For some STL algorithms, however, this is not possible; eg, **`std::set_intersection`**, **`std::set_union`**, **`std::accumulate`**
- New parallelizable algorithms have also been added, such as **`std::reduce`**, **`std::transform_reduce`**

Parallel STL Algorithms

- Additional information:

https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/

- std::transform example (next slides)
  - 1st: Same as existing **`std::transform`** by default (sequential)
  - Then: Using additional execution policy parameter, set to **`std::par`**
  - Compare timing of each

std::transform

- Example problem
  - Have large random sample from a standard normal distribution
  - For each value in the sample, compute the approximation of the exponential function using the power series

$$e^x \cong \sum_{k=0}^{n} \frac{x^k}{k!}$$

for n "sufficiently large"

  - Replace each element (in a **std::vector**) with its transformed value
  - After stopping the timer, calculate the average to verify identical results

- Compare results using **std::transform** in serial (default) and parallel (**std::parallel::par**)

std::transform

```
// Exponential power series approximation
// as lambda expression expSeries.  The terms
// parameter in the capture is taken as input.
// Start from k = 2 for simplicity and efficiency.
auto expSeries = [terms](double x) {
    double num = x;
    double den = 1.0;
    double res = 1.0 + x;
    for (int k = 2; k < terms; ++k)
    {
        num *= x;
        den *= static_cast<double>(k);
        res += num / den;
    }
    return res;
};
```

std::transform

```
// u and v are identical vectors of standard
// random normal random variates

// Use std::transform to run exponential power series
// on each element in v:
std::clock_t begin = std::clock();// begin time with threads
std::transform(u.begin(), u.end(), u.begin(), expSeries);
std::clock_t end = std::clock(); // end transform time with no par

// std::reduce is a parallelized alternative for std::accumulate(.)
auto mean = (1.0 / u.size())*std::reduce(u.cbegin(), u.cend(), 0.0);
auto time = (end - begin) / CLOCKS_PER_SEC;


// Use std::transform with std::par execution policy
// to run exponential power series on each element in v:
begin = std::clock();// begin time with threads
std:: transform(std::execution::par,v.begin(),v.end(),v.begin(),expSeries);
end = std::clock();                  // end transform time with par

auto meanPar = (1.0 / v.size())*std::reduce(v.cbegin(), v.cend(), 0.0);
time = (end - begin) / CLOCKS_PER_SEC;
```

std::transform

- Some results (Microsoft Surface Pro)
  - Two identical pairs of normal variates for each elems/terms combination
  - 50% cut or better in run time for parallel vs sequential

Num Elements	Num Terms in Sum	Seq or Par	Time Elapsed (Seconds)	Mean of Transformed Elem's	Percent Drop in Run Time
5,000,000	400	Seq	9	1.64821	
5,000,000	400	Par	4	1.64821	55.6%
5,000,000	400	Seq	9	1.64967	
5,000,000	400	Par	4	1.64967	55.6%
50,000,000	200	Seq	58	1.64878	
50,000,000	200	Par	28	1.64878	51.7%
50,000,000	200	Seq	60	1.64894	
50,000,000	200	Par	29	1.64894	51.7%
100,000,000	200	Seq	153	1.64865	
100,000,000	200	Par	66	1.64865	56.9%
100,000,000	200	Seq	144	1.6488	
100,000,000	200	Par	66	1.6488	54.2%

MATH IN THE BOOST LIBRARIES
The Good, the Bad, and the Ugly

Math in Boost: An Overview

- Boost Math Toolkit (2.9.0): https://www.boost.org/doc/libs/1_70_0/libs/math/doc/html/index.html
  - Statistical Distributions
  - Mathematical Constants
  - Numerical Integration
  - Others…

- Additional Math-Related Boost Libraries (not in Math Toolkit)
  - Circular Buffers
  - MultiArray
  - Accumulators
  - Odeint (ODE Solver)
  - uBLAS (Matrices and very basic linear algebra)

- Some features have been migrated to recent versions of the C++ Standard Library
  - Random number generation from parameterized statistical distributions (as we saw previously)
  - Special math functions: Bessel Functions, Laguerre Polynomials, others
  - GCD and LCD functions

Math in Boost – an Opinion

- The Good: There are some very useful libraries and contents in Boost for quantitative models development and general mathematical research

- The Bad: There are also math libraries and functions that are so poorly designed, or simply dated or neglected, as to be practically useless

- The Ugly: Overdesigned to the point that it takes way too much time to figure out how to use them; quants and researchers aren't interested in, nor do they have time for, gratuitously complex and esoteric design patterns

Math in Boost – Brief Tour

- Not enough time to present in detail (might require a half-day)

- Will focus primarily on “The Good”, and how math in Boost could be “even better”

Boost Math Toolkit: Selected Features

- In Boost:
  - Mathematical Constants (π, etc; far preferable to C macros!)
  - Statistical Distributions and Functions (pdf, cdf, quantile)
  - Quadrature (Integration) and Differentiation

- Migrated from Boost Math Toolkit to recent C++ Standard Library releases:
  - Integer Utilities (GCD and LCD, etc, now available in C++17)
  - Special Functions (C++14) – Bessel Functions, Laguerre Polynomials, etc

Mathematical Constants

- Examples: π, 2π, e, e^{π}, $\sqrt{e}$, $e^{-\frac{1}{2}}$, etc
  - Consistent definitions
  - Avoids C-style macro definitions
  - Clear to programmers what they mean (no guessing about fixed numbers)
  - Constants determined at compile time; one doesn't have to call a function each time to get a particular value.

```
#include <boost/math/constants/constants.hpp>
using boost::math::double_constants::pi;
using boost::math::double_constants::two_pi;
```

- The comprehensive list may be found here:

  https://www.boost.org/doc/libs/1_70_0/libs/math/doc/html/math_toolkit/constants.html

Mathematical Constants (selected)

Rational fractions	
half	1/2
third	1/3
two_thirds	2/3
three_quarters	3/4
two and related	
root_two	√2
root_three	√3
half_root_two	√2 /2
ln_two	ln(2)
ln_ten	ln(10)
ln_ln_two	ln(ln(2))
root_ln_four	√ln(4)
one_div_root_two	1/√2

π and related	
pi	π
half_pi	π/2
third_pi	π/3
sixth_pi	π/6
two_pi	2π
two_thirds_pi	2/3 π
three_quarters_pi	3/4 π
four_thirds_pi	4/3 π
one_div_two_pi	1/(2π)
root_pi	√π
root_half_pi	√ π/2
root_two_pi	√ π*2

Euler's e and related	
e	e
exp_minus_half	$e^{-1/2}$
e_pow_pi	e^{π}
root_e	√ e
log10_e	log10(e)
one_div_log10_e	1/log10(e)
Trigonometric	
degree	radians = π / 180
radian	degrees = 180 / π
sin_one	sin(1)
cos_one	cos(1)
sinh_one	sinh(1)
cosh_one	cosh(1)

Boost Statistical Distributions

- Distributions are Objects
- Each kind of distribution in this library is a class type - an object.
- Examples: Construct objects of Student's t and Standard Normal distribution types:

```
// Avoid potential  ambiguity
// Safer to declare specific functions with using statement(s):
#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 d1(4);

// Construct a normal distribution with mean 0 and variance 1:
normal std_normal(0.0, 1.0);

// See http://www.boost.org/doc/libs/1_70_0/libs/math/doc/html/math_toolkit/stat_tut/overview/objects.html
```

Boost Statistical Distributions

- We are then able to use the *cumulative distribution functions, probability density functions,* and *quantiles* etc for these distributions.
- Generic operations common to all distributions are non-member functions
- Want to calculate the PDF (Probability Density Function) of a distribution? No problem, just use:
  - `pdf(my_dist, x);` `// Returns PDF (density) at point x`
    `// of distribution my_dist.`
- Or how about the CDF (Cumulative Distribution Function):
  - `cdf(my_dist, x);` `// Returns CDF (integral from -`
    `// infinity to point x)`
    `// of distribution my_dist.`
- And quantiles are just the same:
  - `quantile(my_dist, p);` `// Returns the value of the`
    `// random variable x such that`
    `// cdf(my_dist, x) == p.`

Statistical Distributions in Boost

- 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

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

Boost Statistical Distributions

- For mathematical and statistical modeling, we typically need the following for any given distribution:
  - pdf
  - cdf
  - quantile
  - random sampling
- Boost gives us the *first three*
- The *last one* is in **`<random>`** in C++11
- Might we also see the first three make it into the Standard Library?

Integration and Differentiation

- Example: Trapezoid Rule to evaluate integral that computes π:

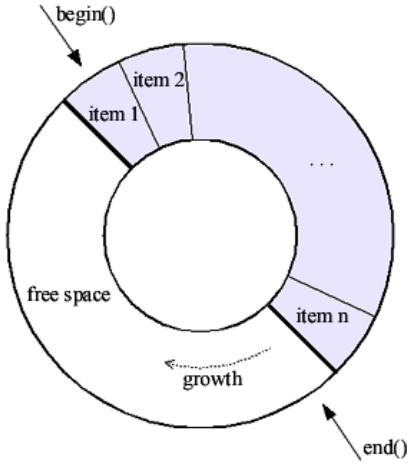
```
auto f = [](double x)
{
    return 4.0 / (1.0 + x * x);
};

double appPi = trapezoidal(f, 0.0, 1.0);     // Boost function (default)
// Boost function (user supplied parameters):
double tol = 1e-6;
int max_refinements = 20;
double appPi2 = trapezoidal(f, 0.0, 1.0, tol, max_refinements);
cout << appPi << ", " << appPi2 << endl;
```

```
3.14159, 3.14159
```

- Trapezoid Rule: Will also accept a function object (very convenient, and uses modern C++)
- Differentiation using first finite difference, and Monte Carlo integration also available
- These will also accept a lambda or a function object to represent the mathematical function
- This is an example of a very well designed math library!

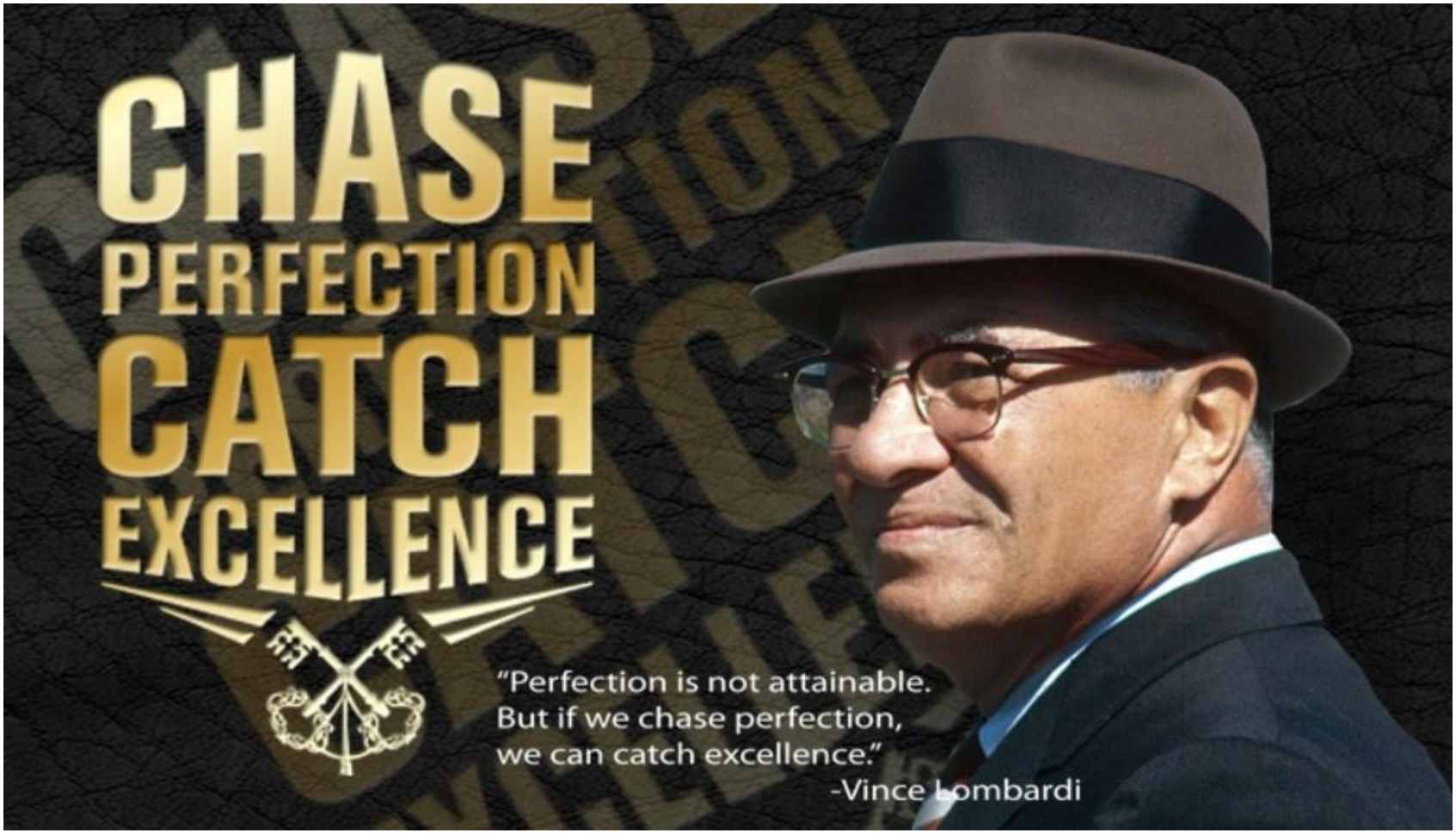
Some Additional Math-Related Boost Libraries (outside Math Toolkit)

- Circular Buffers (Good)
  - Similar to **`std::deque`**, but with fixed capacity
  - Very useful for managing dynamic time series data
  - Old data popped off at capacity as new data pushed back
  - STL compliant



- MultiArray (Good)
  - Templated multidimensional array
  - Very useful for lattice models for pricing options
  - Not as robust as TensorFlow etc for hard core machine learning models
- Accumulators (Good)
  - Ideal for managing data columns
  - Functions for descriptive statistics (mean, median, etc) included
- Odeint Ordinary Differential Equations Solver (Good reputation)
  - Next step to investigate
  - Might we also get a PDE solver sometime (very big for finance)?
- uBLAS (Ugly)
  - Matrices and very basic linear algebra
  - Very outdated and overdesigned
  - More modern, highly performant, and robust open source libraries are available
    - Eigen
    - Armadillo

SUMMARY AND CONCLUSION



The Way We Were

- From about 1992-2006, C++ was the go-to language for model implementation in quant finance
- Java and C# moved in
  - “Fast enough”
  - More rapid development
  - Easier (and fast) database connections
  - Less (perceived) risk of memory leaks and program crashes
- Open source math libraries became available
  - Apache and NIST libraries for Java
  - Math.NET and Iridium for C# and .NET (have since merged).
  - Python: NumPy, SciPy, Pandas, Scikit-Learn are all de facto standard libraries
  - C++ libraries were often incomplete, difficult to integrate, overdesigned, and lacked sufficient documentation (Eigen, Armadillo, CNTK etc came later)
- .NET platform grew in popularity
  - Some functional programming features, and other math/finance friendly capabilities added to C# over time
  - Full functional programming available in F#, with easy integration with C#
  - C# even had an easy to use Excel-based Date class, which most quants used anyway when prototyping or calling external libraries in production
  - Very fast database connections and much less time consuming to integrate
  - ML.NET machine learning library now published as open source by Microsoft

The Way We Are Now

- C++11: The Beast is Back
  - **`<random>`**
  - Task-based parallelism
  - **`std::unique_ptr`**
  - Lambda expressions
  - Move semantics – all of these were great additions for quant work
- C++14/17/20
  - Math special functions
  - Parallel STL algorithms
  - Proper date class coming in C++20 (finally!)
  - GPU-enabled parallel STL algorithms also planned for C++20
- Greater availability of quality math libraries
  - Eigen and Armadillo for linear algebra
  - CNTK (C++, Microsoft), TensorFlow (C++/Python, Google)
- Boost: Good, Bad, and Ugly

The Way We Are Now

- However:
  - Very rare to have C++ specifically for quant dev skills requested in job descriptions for our students
    - C# & Java more common as platform languages
    - Python also big for general purpose + quant development
  - Common gripes from quants and hiring managers
    - Too many different ways to do the same thing in C++
    - C# or Java can be sufficiently optimized; developing in C++ first would be a premature optimization
    - Still afraid of the dangers of pointers and memory leaks; don't want to deal with the bugs
- Many data science and machine learning platforms are dominated by Java/JVM (eg Spark/Hadoop); Microsoft also is ascendant and C#/.NET-focused
- C++ is dominant in trading/order/execution systems, but this is less about math and more about raw speed and communication
- Very little emphasis on quantitative programming at C++ conferences, books on modern C++ or in the blogosphere
- "C++ has a public relations problem" -Bjarne Stroustrup

How to Get our Mojo Back – Some Ideas

- Modernize and expand math/statistics libraries in Boost
  - Eliminate old and highly unused components of the Boost Math Library
  - Replace with a comprehensive numerical library with integrated features and functionality, rather than being separate and fragmented as now
  - Look to de facto Python standard math libraries as a good example (NumPy, SciPy, Pandas, Scikit-Learn) and support similar C++ implementations
- Rumblings of adding matrix algebra to a future C++ release: *make it so*
- Establish a subgroup of the Committee that is specifically focused on mathematical and scientific computing
- More emphasis during conferences on language and library features that will help solve problems (rather than sophisticated solutions in search of problems)
- Promote and distribute integration with scientific languages such as R and Python for visualization of results and user-friendly function calls (de facto standard interfaces – robust and maintained – would be a big plus for C++; could also put in Boost)
- Centralized repository for peer-reviewed and standardized open source C++ math libraries, a la CRAN for R packages (including cutting edge research)

End