Leveraging Modern C++ in Quantitative Finance

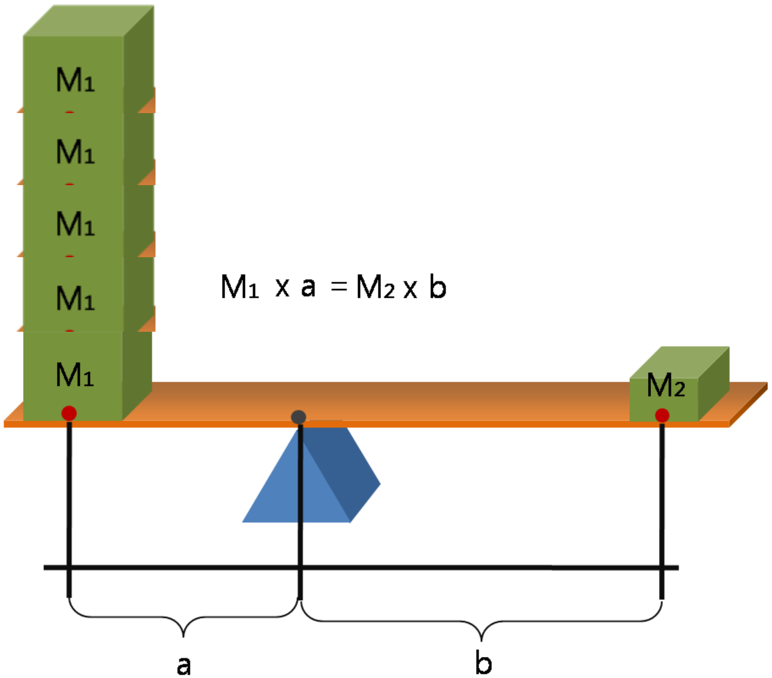
CppCon
September 2019

Daniel Hanson
Lecturer, Computational Finance & Risk Management
Dept of Applied Mathematics
University of Washington

CFRM

What this talk is about

- Focus on end-users
- Easy to use, powerful math-related tools in C++ we can *leverage*



The Start of Something Beautiful

- C++11
  - Math features (coming up)
  - Move semantics => more efficient models code
  - Parallel STL algorithms followed in C++17

- Better availability of quality open source math libraries (late 2000's)
  - Eigen
  - Armadillo
  - More recently:
    - ➢ xtensor – a la NumPy (~2016)
    - ➢ DataFrame – a la R dataframe and Python Pandas (~2018)

- Coming attractions
  - A proper Date class with C++20
  - SG14 linear algebra library proposal

- Mathematical contributions to Boost
  - Some are very intuitive and useful
  - But others could be a lot more user-friendly

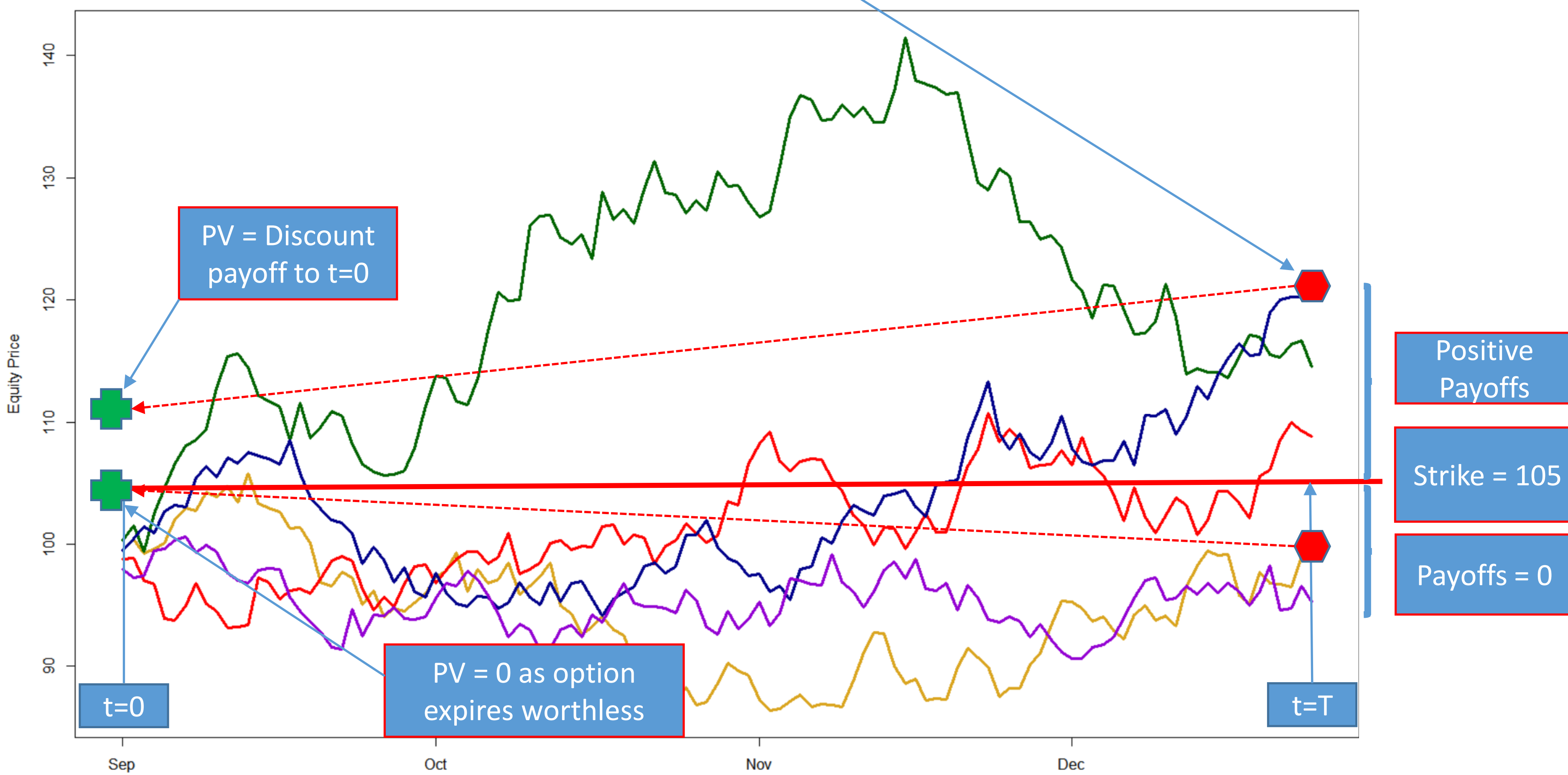
MATH IN THE NEW C++ STANDARD

C++11 and After

Case study: Monte Carlo model for pricing a European option

Monte Carlo Option Pricing

- European Option: A tradeable contract that gives the holder the right to buy or sell a share of stock at a predetermined strike price on its expiration date
- A simple graphical example of a call option is shown below (right to purchase at strike price)
  - Assume the vertical axis represents changes from an underlying asset currently valued at $100/share
  - The red line represents a strike price of $105
  - The positive payoffs at expiration will be the blue (~$15), green (~$10) and red (~$5) scenarios



Monte Carlo Option Pricing

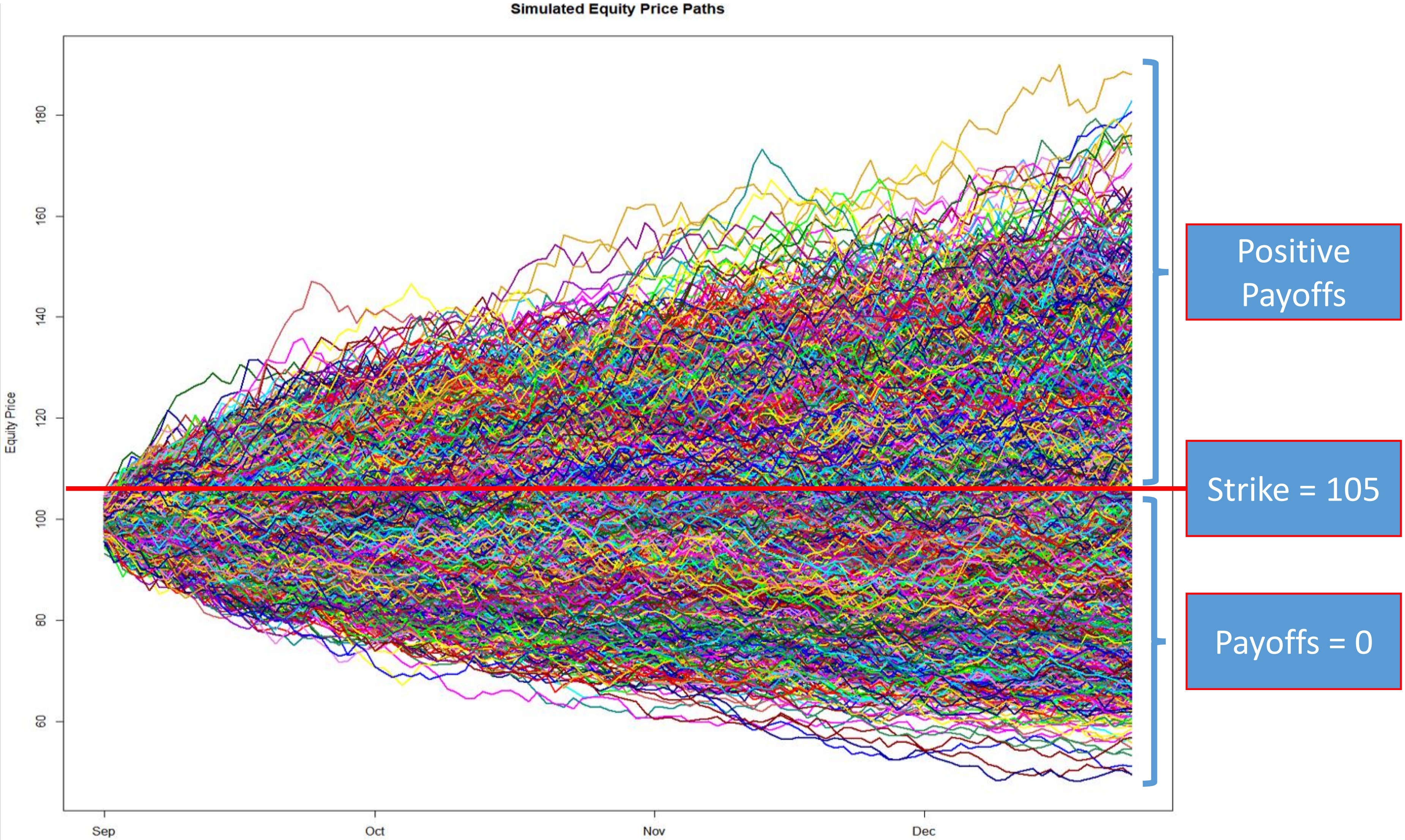
- In our five-scenario example, to determine the option price:

  - If the risk-free interest rate is, say 1.2%,
  - and the time to expiration is four months (1/3 of a year),
  - the option value would be

$$\frac{\underbrace{e^{-(0.012)(0.333)}}_{\text{Discount factor}}\{\overbrace{\boxed{(120-105)+(115-105)+(110-105)+0+0}}^{\text{Payoffs at expiration}}\}}{\underbrace{5}_{\text{Calculate average to get option price}}} = 5.98$$

Monte Carlo Option Pricing

- In reality, the number of scenarios required can be on the order of 10,000 – 100,000
- This can lead to computationally intensive operations
- First, however, we need to generate *a single equity price path* (next slide)



Equity Price Generator (implementation) –Initialization

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

$S_{t-1} = S_0$ = **initEquityPrice_**

r = **rfRate_**

σ = **volatility_**

Δt = **dt_**

ε_t = N(0, 1) random variates to be generated in the code (see next slide)

```
#include "EquityPriceGenerator.h"
#include <random>
using std::mt19937_64;
using std::normal_distribution;
```

Equity Price Generator (implementation)

```
vector<double> EquityPriceGenerator::operator()(int seed) const
{
    vector<double> v;
    mt19937_64 mtEngine(seed);
    normal_distribution<> nd;
    auto newPrice = [this](double previousEquityPrice, double norm)
    {
        double price = 0.0;
        double expArg1 = (rfRate_ - ((volatility_ * volatility_) / 2.0)) * dt_;
        double expArg2 = volatility_ * norm * sqrt(dt_);
        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_;
    // i <= numTimeSteps_ since we need a price at the end of the
    // final time step.
    for (int i = 1; i <= numTimeSteps_; ++i)
    {
        equityPrice = newPrice(equityPrice, nd(mtEngine));    // norm = nd(mtre)
        v.push_back(equityPrice);
    }
    return v;
}
```

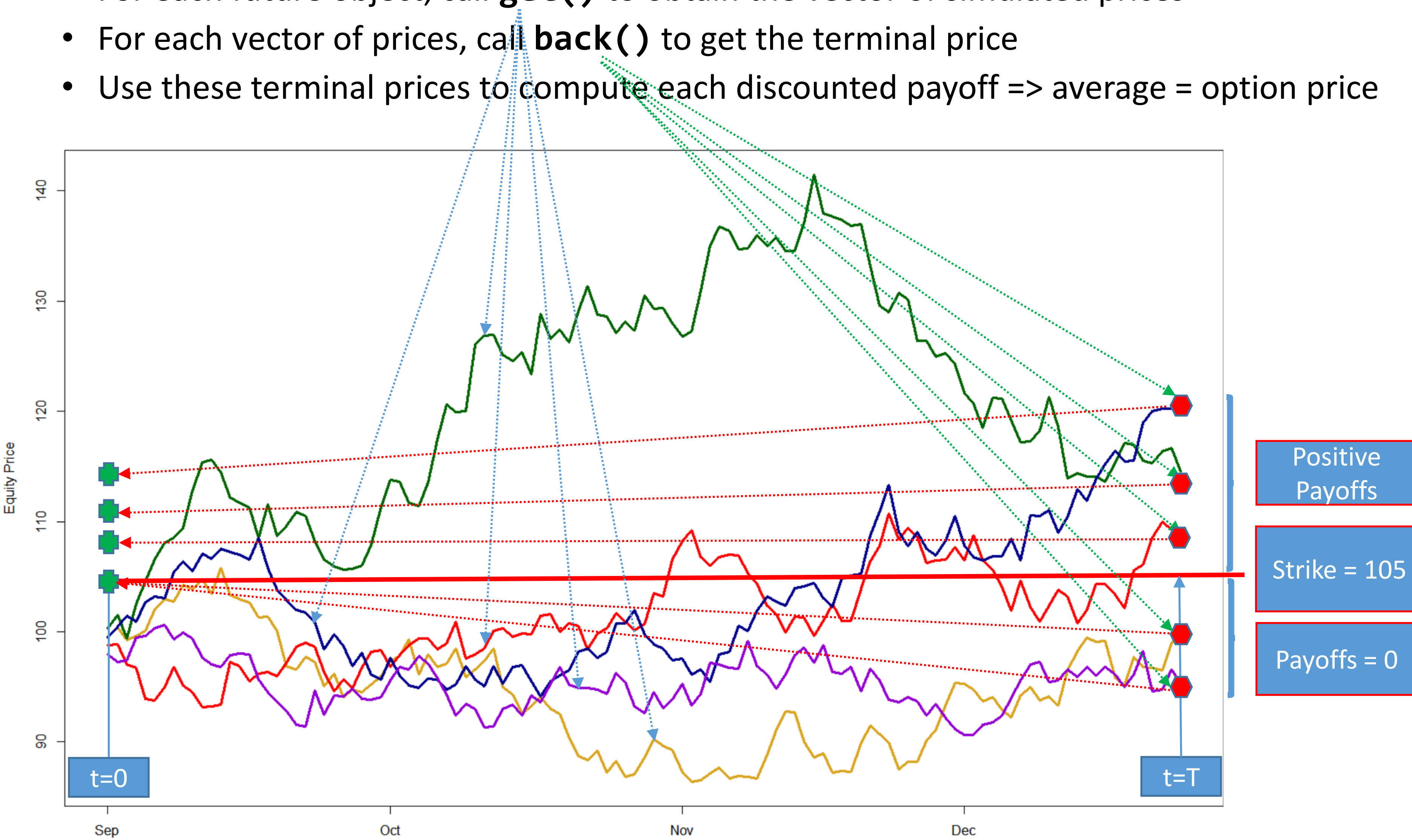
$$S_t = S_{t-1} e^{\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma \varepsilon_t \sqrt{\Delta t}}$$

Where the magic starts

Lambda Expression implements discretized SDE

S_0

- Going back to our five-scenario illustration:
  - Each generated path is a parallel task: **`std::future<std::vector<double>>`**
  - For each future object, call **`get()`** to obtain the vector of simulated prices
  - For each vector of prices, call **`back()`** to get the terminal price
  - Use these terminal prices to compute each discounted payoff => average = option price



Option Calculations (Declaration)

- First look at the function declarations

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

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

    double operator()() const;  // Returns the option price

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

    // ...
};
```

Use our previous result

enum class represents the option type: Call or Put

Additional option model inputs

computePriceAsync() will generate each scenario path in parallel

```
void MCEuroOptPricer::computePriceAsync_()
{
    EquityPriceGenerator epg(spot_, numTimeSteps_, timeToExpiry_, 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));
    }

    realVector 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);
}
```

#include<future>

Create **vector** of **future** objects

async instructs the program to generate a **vector** of simulated stock prices on a separate thread; each threaded operation is encapsulated in a **future** object

The **get()** member function on the **future** object retrieves the asynchronously generated **vector**. We only need the last simulated equity price, so we just use **back()**.

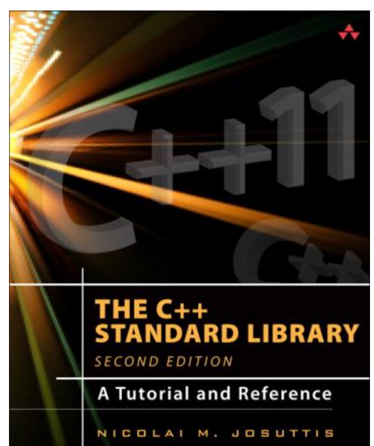
Option Calculations (Summary and Results)

- Hyper-V 20-core virtual machine
- Windows 2019 Server
- Visual Studio 2017 Compiler

Expiry (years)	Time Steps	Scenarios	Time – Serial (sec)	Time - Parallel (Task)	Pct Drop in RT
1	12	10000	0.027	0.027	0.0%
1	120	50000	0.608	0.179	70.6%
5	120	50000	0.604	0.18	70.2%
5	600	50000	2.643	0.261	90.1%
5	600	100000	5.224	0.563	89.2%
10	600	100000	5.17	0.576	88.9%
10	1200	100000	10.201	0.933	90.9%

Option Calculations (Summary and Results)

- This example used the simple case of a European equity option
- Much more complex and computationally intensive option contracts exist
- Remark: There are different engine algorithms and distributions available in <random>
  - Mersenne Twister 64 is the most robust engine
  - Total of 17 well-known “textbook” distributions
  - See Nicolai Josuttis: The C++ Standard Library (2E), §17.1.4: Distributions



MATH IN THE BOOST LIBRARIES

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

- Additional Math-Related Boost Libraries (not in Math Toolkit)
  - Circular Buffers
  - Accumulators
  - MultiArray

Boost Math Toolkit: Statistical Distributions

- Each probability distribution in this library is a class type
- Examples: Construct objects of Student's t and Standard Normal distribution types:

```
#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 std dev 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 Math Toolkit: Statistical Distributions

- We can then apply the *cumulative distribution function, probability density function,* and *quantile function* for any distribution.
- Generic operations 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 similar:
  - ```
    quantile(my_dist, p);// Returns the value of the
                         // random variable x such that
                         // cdf(my_dist, x) == p.
    ```
    ```

- 34 probability distributions available in BSD > 17 in Standard Lib

- Will we see a union of all these distributions and functions in the Standard Library eventually?

Boost Math Toolkit: 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)
```

The function `f` can also be represented as a functor

```
3.14159
```

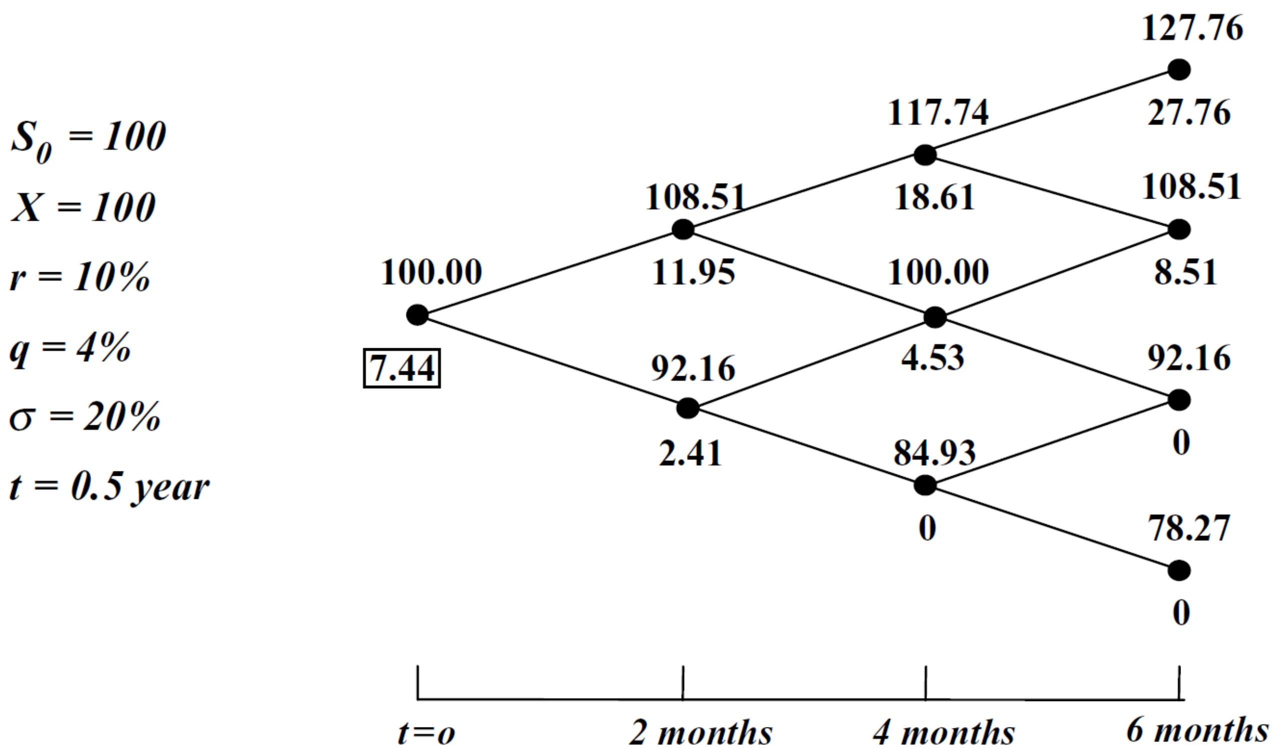
- All numerical differentiation and integration methods will accept the function input as a *lambda or a functor*
- Could also be applied to root finding algorithms; examples in sample code:
  - Bisection Method
  - Steffenson's Method

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

- Circular Buffers (CircularBuffers.cpp in sample code)
  - STL compliant
  - 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 on at back

- Accumulators (Accumulators.cpp in sample code)
  - STL compliant
  - Ideal for managing data columns
  - Functions for descriptive statistics (min, max, mean, median, variance, etc) come implemented – very convenient

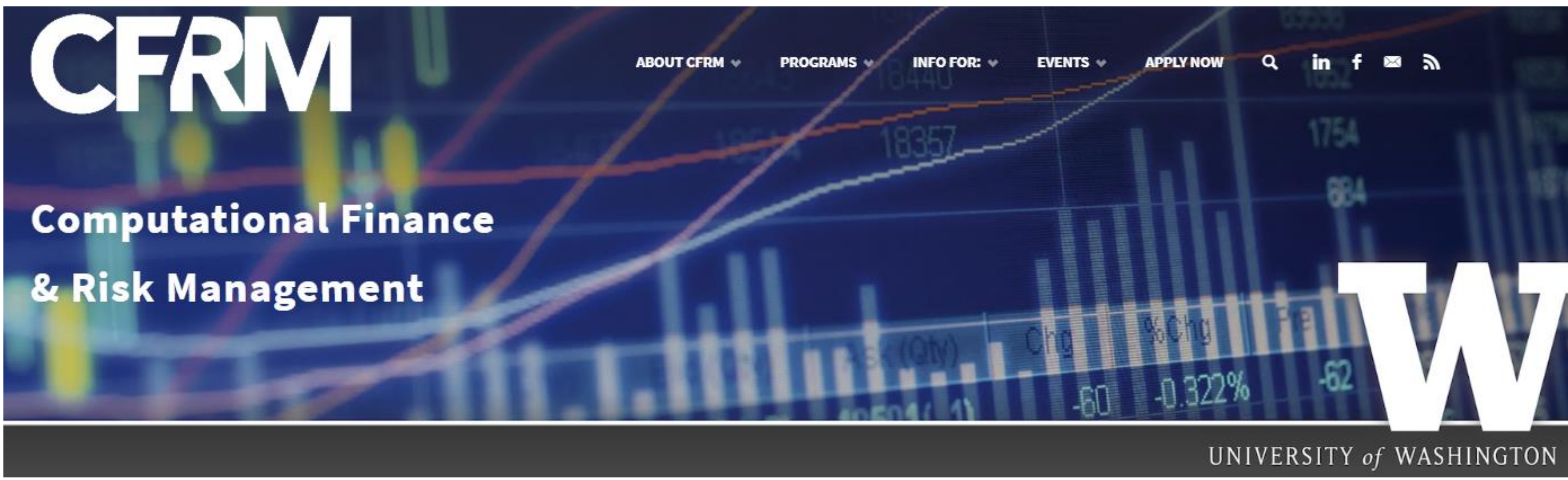
- MultiArray (MultiArray.cpp in sample code)
  - STL compliant
  - Templated multidimensional array
  - Useful for lattice models for pricing options
  - Binomial lattice for option on single asset



Peter James: Option Theory, Figure 7.4 (left), Figure 12.1 (right)

References

- Jon Kalb and Gasper Azman, C++ Today: The Beast is Back, O'Reilly
- Nicolai Josuttis, The C++ Standard Library (2E), Addison Wesley
- Scott Meyers, Effective Modern C++, O'Reilly
- Peter James, Option Theory, Wiley
- Numerical Library in C++ project (2019): Root finding algorithms implemented by Tania Luo, MSc student, Dept of Applied Mathematics, Univ of Washington
- Information about the CFRM program at the University of Washington: http://cfrm.uw.edu



End

- Accompanying sample code available on GitHub: https://github.com/QuantDevHacks/CppCon2019Backup
- Contact:
  - email: hansondj (at) uw.edu
  - LinkedIn: https://www.linkedin.com/in/danielhanson/
- Leverage a beverage

- Thanks for attending!