Chapter 8. The Boost Libraries

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *learnmodcppfinance@gmail.com*.

Introduction

The Boost Libraries, as noted on their official website **{1}**, are "free peer-reviewed portable C++ source libraries...that work well with the C++ Standard Library". They are open source libraries, released under the Boost License, which allows their use in other libraries, systems, and applications – including commercial use – "with minimal restrictions". This chapter will present Boost libraries that can be useful in fi-

nancial applications, and other quantitative applications, including the Boost Math Toolkit, along with other libraries—not necessarily mathematical—that lend themselves well to these disciplines.

A fair number of libraries that were originally released in Boost have found their way into the C++ language and Standard Library. In this respect, some of the Boost Libraries could also be considered a proving ground for subsequent ISO C++ enhancements. As also stated in their documentation, Boost authors "aim to establish *existing practice* and provide reference implementations so that Boost libraries are suitable for eventual standardization" (ibid) **{1}**. This is not to say everything in Boost is destined for inclusion in C++, and it should be noted there can be differences in content and implementation with respect to the Boost features that have been incorporated into modern C++.

The introduction on the Boost website (ibid)**{1}** spells out their purpose in more detail:

- Free peer-reviewed portable C++ source libraries
- Libraries that work well with the C++ Standard Library
- Usable across a broad spectrum of applications
- The Boost license that encourages the use of Boost libraries for all users with minimal restrictions
- A valuable source for additions to the Standard C++ Library

This last point is particularly salient, with content rooted in Boost now contained in modern releases of C++. Table 8-1 (below) provides a list of C++ libraries and language features we have discussed in previous chapters that have their origins in Boost.

Table 8-1. New Standard C++ Features with Previous Boost Implementations

Feature	Description/Examples	Release	Chapter
Mathematical special functions	Gamma functions, Bessel functions, Legendre polynomials, etc	C++17	1
Mathematical Constants	π, $\sqrt{x}$, e, e^x, $log(x)$, etc	C++20	1
Lambda Expressions	`auto f = [.](...)`	C++11	1
Smart Pointers	`unique_ptr`, `shared_ptr`	C++11	3
Task-Based Concurrency	`future`, `async`	C++11	5
Distributional Random Number Generation	`normal_distribution`, `student_t_distribution`, `mt19937_64` etc	C++11	5

The Boost libraries to be discussed can be useful in financial programming. We will first look at two features in the Boost Math Toolkit library, namely mathematical constants and statistical distributions. Although mathematical constants have been added to the Standard Library, the Boost version contains two commonly-used constants in particular that were curiously missing from the C++20 version.

This will be followed by the Boost MultiArray library, which can be used for implementing lattice pricing models for options. We will conclude with Boost Accumulators, which can be used for managing trading indicators and signals.

Before proceeding, there is one important point to note regarding the installation of the Boost libraries. A large portion of the source code consists of header-only implementations (template code, similar to Eigen), but it also includes a number of implementation source files requiring compilation. The libraries discussed in this chapter, however, are all header-only. As such, it will suffice to download the com-

pressed file **{2}** suitable for your operating system, extract the header files only (Boost uses the .hpp extension), and specify their location in your compiler settings.

In the examples that follow, as well as those in the Boost documentation, these header files are assumed to be located under a `boost` directory or subdirectory.

Mathematical Constants

C++20 added mathematical constants to the Standard Library, but as we saw, commonly used constants representing the values $\frac{1}{\sqrt{2}}$ and $\frac{1}{\sqrt{2\pi}}$ were curiously missing. Prior to C++20, the Boost Math Toolkit already contained a set of `constexpr` mathematical constants, meaning the values are determined at compile time, which avoids function calls at runtime. Most of the C++20 mathematical constants are also provided in Boost, although there are exceptions, namely the mysterious $\sqrt{3}$ and $\frac{1}{\sqrt{3}}$. Some of the naming conventions are the same, such as the obvious `e` and `pi`, but there are also notable differences. For example, the natural log of 2 in the Standard Library is `ln2`, while in Boost it is `ln_2` . Also, inverses are different, such as $\frac{1}{\pi}$. This is `inv_pi` in the Standard Library, but `one_div_pi` in Boost.

Finally, as for $\frac{1}{\sqrt{2}}$ and $\frac{1}{\sqrt{2\pi}}$, these *are* available in Boost as `one_div_root_two` and `one_div_root_two_pi`, respectively.

Boost mathematical constants are defined in the header file `boost/math/constants/constants.hpp`, and so this file must be included. They are scoped with the namespace `boost::math::double_constants`. Although the standard normal density function is provided in the Boost Statistical Distributions library which follows next, as a demonstration of using Boost mathematical constants, a lambda that implements the mathematical definition of the standard normal density function is shown here:

```
#include <boost/math/constants/constants.hpp>

. . .

auto std_norm_pdf = [](double x) -> double
{
      return boost::math::double_constants::one_div_root_two_pi
            * std::exp(-boost::math::double_constants::half * x * x);
};

double y = std_norm_pdf(0.0);         // Approx: 0.39894
```

Documentation for Boost mathematical constants can be found in **{3}**.

Statistical Distributions

In the previous section, we looked at an example of implementing the standard normal pdf. This served the purpose of demonstrating how to use Boost mathematical constants, but now we will look at a more robust approach, namely using the set of Statistical Distributions and Functions in the Boost Math Toolkit. The beauty of this library is that it is incredibly easy and straightforward to use, while it provides useful functionality for computational finance applications. The available distributions include the common normal, Student's t (aka t-distribution), and uniform distributions, plus others that are sometimes used in finance such as the log normal, Pareto, skew normal, and non-central t distributions. There is also an empirical cumulative distribution function object that takes in a data set and performs goodness of fit tests with known distributions.

A full list of the distributions supported by Boost is available on the Boost website **{4}**.

Table 8-2. Boost Statistical Distributions

Arcsine Distribution	Bernoulli Distribution	Beta Distribution	Binomial Distribution	Cauchy-Lorentz Distribution
Chi Squared Distribution	Exponential Distribution	Extreme Value Distribution	F Distribution	Gamma Distribution
Geometric Distribution	Hyperexponential Distribution	Hypergeometric Distribution	Inverse Chi Squared Distribution	Inverse Gamma Distribution
Inverse Gaussian (or Inverse Normal) Distribution	Kolmogorov-Smirnov 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 full list of the distributions supported by Boost with links to their respective details is available on the Boost website **{4}**.

Probability Functions

Recall from Chapter 5, we introduced statistical distributions in the Standard Library for the express purpose of generating random numbers from among 17 textbook distributions. The Boost Statistical

Distributions library, in contrast, provides the following deterministic functions for an even broader range of distributions, 34 in all:

- Probability Density Function (pdf)
- Cumulative Distribution Function (cdf)
- Quantile Function
- Complement Function

The way it works is you construct a distribution object – eg a normal or Student's t-distribution) – and then use it as an argument in any of the four functions above. Distributions in Boost can be obtained by including the header file `boost/math/distributions.hpp`, and applying the `boost::math` namespace scope. Alternatively, the individual header files can be included separately, for example:

```
#include <boost/math/distributions/normal.hpp>          // Normal distribution
#include <boost/math/distributions/students_t.hpp>      // t-Distribution
```

Using the normal and t-distributions as examples:

- The standard normal distribution has default arguments mean = 0 and standard deviation = 1
- With a non-standard normal distribution, the mean and standard deviation are the constructor parameters
- The t-distribution takes in the degrees of freedom as its constructor argument

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

// Construct a normal distribution with mean 0 and variance 1:
normal std_normal{};
```

```
// Construct a non-standard normal distribution with mean 0.08 and standard deviation 0.25:
normal non_std_normal{ 0.08, 0.25 };

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

The distribution classes have statistical parameter accessor functions that are particular to each distribution. For example, with the `normal` class, it is possible to retrieve the mean and standard deviation, and for `students_t`, the degrees of freedom:

```
double mean = std_normal.mean();
double sd = std_normal.standard_deviation();

mean = non_std_normal.mean();
sd = non_std_normal.standard_deviation();

double dof = stu_t.degrees_of_freedom();
```

Non-member pdf, cdf, and quantile functions will calculate these values for any Boost distribution for a given value. These are taken in as arguments, as shown in the next example:

```
// pdf and cdf values for the standard normal
// and Student's t distributions at x = 0:
double n_pdf = pdf(std_normal, 0.0);        // 0.3989
double n_cdf = cdf(std_normal, 0.0);        // 0.5
double t_pdf = pdf(stu_t, 0.0);             // 0.375
double t_cdf = cdf(stu_t, 0.0);             // 0.5
```

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

Care is advised when calculating upper quantiles approaching the 100%-tile, as the function value is unbounded and can cause overflow errors **{5}**. Instead, the `complement(.)` function applied to lower percentiles is recommended in the case of symmetric distributions. For example, attempting calculation of a very high percentile:

```
students_t stu_t_100{ 100 };

double very_high_quantile = quantile(stu_t_100, 1 - 1e-100);            // Will likely crash program
```

This is nearly the same as attempting a calculation of the 100%-tile:

```
quantile(stu_t_100, 1);
```

which is not a finite value. As a result, attempting to calculate the (1 $-$ 1 $\times$ 10^{-100}) $\times$ 100%-tile could cause an overflow error and possibly crash your program.

Instead, you can apply the complement function as follows:

```
double very_high_quantile = quantile(complement(stu_t_100, 1e-100));
cout << "t-dist((1 - 1e-100) x 100%-tile (using complement) = " << very_high_quantile << "\n\n";
```

This way, a finite value of about 96.3147 is safely returned.

It should be noted, however, the complement function used in this manner at some point will also become a problem. Attempting to compute a higher quantile value using the complement function, such as in the following, will also likely cause problems:

```
double extremely_high_quantile = quantile(complement(stu_t_100, 1e-1000));
```

In general, one therefore needs to exercise caution at either extreme of a distribution.

Random Number Generation with Boost Distributions

As noted above, there are 34 different distributions available in Boost, but only 17 random number distributions in the Standard Library. What if you would like to generate a random sample using a Boost distribution that is not available in the Standard Library? This is possible by applying the Probability Integral Transformation Theorem, where you could generate uniform variates on the interval $(0,1)$ using the Standard Library, and then apply the Boost `percentile(.)` function on the distribution you wish to draw from. The random uniform values are the inputs, and in fact, we can write a class that mimics the provided distributions in the Standard Library.

To address the issue of fat tails and skewness in financial returns data, four-parameter distributions that capture these properties are often fitted to the data. Although there is no option within the Standard Library `<random>` functions, we can build one around the non-central t-distribution available in Boost **{6}**.

Using a form similar to the Standard Library `<random>` classes, we can write a class template, say `rand_non_central_t_dist` with default template parameter `double`. This works by first generating uniform variates on the interval $(0,1)$ using the `std::uniform_real_distribution` class. Each random uniform value is then passed into the Boost `quantile(.)` function, along with a Boost `non_central_t_distribution` object. The functor on `rand_non_central_t_dist` then outputs the next skew-t variate, just like any `<random>`

distribution in the Standard Library. To keep matters simpler for demonstration, we will just use the random engine `std::mt19937_64` 64-bit Mersenne Twister type (introduced in Chapter Five) and a seed value of 100. The class implementation follows, with the constructor argument `nu` representing the degrees of freedom ν, and the non-centrality parameter as `delta` (δ).

```
#include <random>

template<typename T = double>
class rand_non_central_t_dist
{
public:
      // nu = degrees of freedom
   // delta = non-centrality parameter
      rand_non_central_t_dist(double nu, double delta) :nu_{ nu }, delta_{ delta } {}

   // Functor a la <random> in the Standard Library
   T operator()(std::mt19937_64& mt)
      {
            auto unif = ud_(mt);     // Next in pseudorandom unif(0, 1) sequence
            boost::math::non_central_t_distribution nctd(nu_, delta_);
            return quantile(nctd, unif);
      }

private:
      double nu_, delta_;

      std::mt19937_64 mt_;
      std::uniform_real_distribution<T> ud_{ 0.0, 1.0 };
};
```

Now, assuming the ν and δ parameters have been determined externally from fitting the returns data, we can randomly generate return values from the same distribution and place them in a `vector`:

```
std::mt19937_64 mt(100);
rand_non_central_t_dist<> st(3.0, -0.5);

auto skew_gen = [&mt, &st]()
{
      return st(mt);
};

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

NOTE

To keep the code example more readable, there was no bounds checking before applying the Boost `quantile(.)` function inside the body of the `()` operator, but in practice this would be advisable, as discussed previously.

MultiArray

The *Boost Multidimensional Array* Library (aka MultiArray) flagship class is, not surprisingly, the `boost::multi_array` class template. This class provides a clean and efficient implementation of multidimensional arrays. As described in the documentation,

The Boost Multidimensional Array Library provides a class template for multidimensional arrays.... The classes in this library implement a common interface, formalized as a generic programming concept. The interface de-

sign is in line with the precedent set by the C++ Standard Library containers. Boost MultiArray is a more efficient and convenient way to express N-dimensional arrays than existing alternatives (especially the `std::vector<std::vector<. . .>>` *formulation of N-dimensional arrays).* **{7}**

In particular, for a common finance application, it can provide a convenient "out of the box" data structure for implementing binomial lattices for option pricing.

A Simple Two-Dimensional MultiArray

A Boost `MultiArray` in two dimensions defines a matrix-like object that can store an arbitrary, but same-type entry, in each element or node. A simple example is a 2 $\times$ 3 array containing `std::string` objects. Each dimension is defined by an *extent*, similar to that described in Chapter Seven. As we did with `mdspan`, for example, this means that the first extent can be thought of as a row. The second extent then refers to a column element of the first extent (both are zero-indexed).

The 2 $\times$ 3 `MultiArray` is created as follows, with `std::string` as its template parameter type. The Boost library `multi_array.hpp` header needs to be included as shown.

```
#include <boost/multi_array.hpp>
. . .
boost::multi_array<std::string, 2> ma(boost::extents[2][3]);
//Each element can be individually set, with each index placed in the square bracket operator:
ma[0][0] = "Watch ";
ma[0][1] = "out ";
ma[0][2] = "where ";
ma[1][0] = "the ";
ma[1][1] = "huskies ";
ma[1][2] = "go...";

// The result can be displayed to the screen using a nested loop:
```

```
for (unsigned i = 0; i < 2; ++i)
{
      for (unsigned j = 0; j < 3; ++j)
      {
            cout << ma[i][j] << "\t";
      }
      cout << "\n";
}
```

With some additional formatting, the result displays the message in two rows and three columns:

```
Watch         out         where
 the       huskies         go...
```

To use a `MultiArray` to price options, the `std::string` type is replaced with a struct or class that holds the projected underlying equity price and payoff at each node, as discussed next.

Binomial Lattice Option Pricing

As a class template, `boost::multi_array` is generic, so this makes it convenient for storing generated price and payoff data in node objects. For example, consider a binomial lattice for pricing a European or American option. At each node in the `MultiArray`, we could place a simple struct, say `Node`, that will store the generated underlying prices traversing the lattice out to expiration, and then the calculated interim payoffs at each node when traversing back toward the valuation date.

```
struct Node
{
      double underlying;
```

```
        double payoff;
    };
```

Before proceeding with the code example, let us review the binomial lattice pricing method. The time to expiration, say from t = 0 to t = T (in units of years or a year fraction), is divided into smaller discrete time steps of equal length Δt. Starting with the spot price of the equity S_0 _ at time t = 0, a recombining tree, or lattice, will result from assumed constant up and down price movements. In other words, for any underlying equity price at one node, it can only assume one of these two "states of the world" over a time step of Δt, by multiplying the price by fixed up and down factors u and d. As the lattice expands over time, a set of possible equity price scenarios is generated, terminating at expiration of the option at a time t = T. At expiration, the terminal payoffs are calculated based on an exercise price K.

A probability is assigned to an up move and is denoted by p; hence, the probability of a down move is $1-p$. The up and down multipliers, u and d, are dependent on the particular discretization chosen. One of the most popular methods is the Jarrow-Rudd discretization, which we will use for the examples that follow. The value of p then falls out from the calculations of u and d.

In a Jarrow-Rudd discretization, the resulting up and down multipliers u and d are as follows **{8}**:

$u = e^{\sigma\sqrt{\Delta t}}$

$d = \frac{1}{u} = e^{-\sigma\sqrt{\Delta t}}$

where σ represents a constant annual volatility rate. The annual risk-free interest rate and dividend rate are also assumed to be constant continuous rates r and q.

The probability p is then:

$p = \frac{e^{(r-q)\Delta t} - e^{-\sigma\sqrt{\Delta t}}}{e^{\sigma\sqrt{\Delta t}} - e^{-\sigma\sqrt{\Delta t}}}$

To see how we could use a Boost `MultiArray` to implement a binomial lattice, let us first look at a simple example of an at-the-money European call option with the following data: **{9}**:

S_0 = 36 _ (underlying stock price at value date)

K = 36 (exercise price)

r = 6% (risk-free interest rate)

q = 0 (dividend rate - assume none)

σ = 20% (volatility)

T = 1 (time to maturity: one year)

n = 4 (number of time steps, starting from t = 0)

Δt = $\frac{T}{n}$ (time step length = $\frac{1}{4}$ = 0.25)

From this, we can determine that

u = 1.10517

d = 0.90484

p = 0.55000

Prospectively multiplying each underlying price by u and d, the generated price paths result in the following lattice (put diagram here):

The payoff at each of the five terminal nodes is just the call payoff $max(S^{(i)}_T - K, 0)$ for $i = 0,\dots,4$ (zero-indexed in C++)

The option value is determined by working backwards in time along the lattice, calculating the discounted expected payoffs based on the subsequent up and down underlying values, the probabilities of each (p and $1-p$), and the continuous discount factor over each time step. This discount factor, from an arbitrary time point t to $t - \Delta t$, is

$e^{(r-0)\Delta t} = 0.98511$ (dividend is zero)

Now, for example, the payoff of the uppermost node at time $T - \Delta t$ (see figure below—to be added later) would be

$e^{-r\Delta t}53.71 - 36)p + (43.97 - 36)(1-p = e^{-0.06(0.25)}(17.7057(0.55) + 7.9705(0.45)) = 13.1265$

Repeating this process for every node gives us the lattice as shown, with both projected underlying equity prices and discounted payoffs. The value of the option is $3.77, the discounted payoff at the starting node.

((put figure here - textbook triangular lattice))

When setting this up in a two-dimensional `MultiArray` in code, it will more convenient to configure it with an up movement from each element in a given column except for the last, which is a down movement. This works because we have a recombining tree, so an up movement for any of a column's internal

elements—other than the first and last element—also corresponds to a down movement from the element above.

This way, computing the projected prices becomes a reasonably simple nested iteration. The outer loop will move horizontally over time, and then for each time point the inner loop will apply the up multiplier to every element besides the last which is a down movement.

Moving backward in time from maturity back to option valuation at t = 0, the respective expected discounted payoffs can be computed. Note that because we are working in C++, zero-indexing will now apply to both column and row indices. The result takes takes the form of an upper triangular matrix contained in a `MultiArray`, where each element is a (*price*, *payoff*) pair set on a `Node` struct. In the following diagram, the projected underlying values are shown in each node with the respective payoff in the same node indicated by square brackets.

Table 8-3. Projected Underlying Equity Prices with Discounted Expected Payoffs (ATM European Call Option)

Index	0	1	2	3	4
0	36.00 [3.77]	39.79 [5.93]	43.97 [9.03]	48.59 [13.13]	53.71 [17.71]
1		32.57 [1.27]	36.00 [2.34]	39.79 [4.32]	43.97 [7.97]
2			29.47 [0]	32.57 [0]	36.00 [0]
3				26.67 [0]	29.47 [0]
4					24.131[0]

American Options

Applying a binomial lattice to calculate the price of a European option is not all that interesting, as it is just an approximation of the closed-form Black-Scholes solution. Where a lattice does become useful is in

cases allowing early exercise, where it becomes necessary to check whether it is optimal to exercise the option at each node, but no closed-form solution exists. The most common example is an American option, which allows for exercise at any time prior to expiration.

As an example, consider the following example, where the underlying spot price, risk-free rate, volatility, and time to expiration are the same as in the previous example, except for the following variations:

- The option type is a deep in-the-money put with strike price = 40
- The payoff type is American

The equity price projection phase yields exactly the same results as before. On the return trip, however, we need to compare the actual payoff $max(K - S^{(i)}_t))$ against the projected equity price at each node. If the actual payoff is greater than the expected payoff, then the former replaces the latter, as it would then be optimal to exercise the option at that point.

This can be seen, for example, in row index 2 and column index 3 of the lattice result below. The discounted expected payoff computed in the usual way is 6.83, but if we look at the payoff based on the projected equity price at that point, subtracting 32.57 from the strike price of 40.00 gives us 7.43. As this is greater than the expected payoff, we replace it (denoted by strikethrough) with the updated value of 7.43. The same is true in the last node in column 3, where the expected payoff is replaced with the actual payoff of 13.33.

These values are then used for the discounted expected payoff calculation in the last node in the previous time step (t = 2, i = 2), which comes out to 9.35, but again this is replaced by the actual payoff of 10.53. The actual payoff in the top node in time step (t = 2, i = 0), however, is zero, so we keep the expected payoff of 43.79. Continuing this process back to $t = $0 node gives us the estimated option price of 4.54.

Table 8-4. Projected Underlying Equity Prices with Discounted Expected Payoffs (ITM American Put Option)

Index	0	1	2	3	4
0	36.00 [4.54]	39.79 ~~{2.19}~~ [2.31]	43.97 [0.79]	48.59 [0]	53.71 [0]
1		32.57 ~~{6.30}~~ [7.42]	36.00 ~~{3.99}~~ [4.25]	39.7962 [1.77]	43.97 [0]
2			29.47 ~~{9.35}~~ [10.53]	32.57 ~~{6.83}~~ [7.43]	36.00 [4.00]
3				26.67 ~~{12.74}~~ [13.33]	29.47 [10.53]
4					24.13 [15.87]

Implementation with Boost MultiArray

A way to implement this in C++ is to define a class, `BinomialLatticePricer` that will hold a `MultiArray<Node>` member called `grid_`, and where the `Node` struct is as defined previously. A series of member functions will then project the underlying prices out in time, and then will reverse direction to calculate the payoffs at each node, arriving at the option price at t = 0. In the process, the underlying `price` on each `Node` element is updated moving forward in time, and then the discounted expected `payoff` member is calculated and set over the return trip.

It should be noted that this is not the only way to implement lattice models, and there may be methods out there (some proprietary) that yield better performance. The point here is a Boost MultiArray gives you a generalized and proven underlying structure, so you don't have to implement it from scratch yourself, and its perfomance may very well be within the range required .

Returning to the demonstration, the option types will be assumed to be limited to European and American, defined in an enum class:

```
enum class OptType
{
	Euro,
	American
};
```

The lattice pricing class can then be summarized with its declaration:

```
class BinomialLatticePricer
{
public:
	BinomialLatticePricer(VanillaOption opt,
		double vol, double int_rate, int time_steps,
		double div_rate = 0.0);

	double calc_price(double spot, OptType opt_type);

private:
	VanillaOption opt_;
	int time_points_;		// = time_steps + 1
	double div_rate_;

	// Assigned:
	double u_, d_, p_;		// up and down factors and probability of up move
	double disc_fctr_;		// Discount factor

	boost::multi_array<Node, 2> grid_;
	double spot_{ 0.0 }, opt_price_{ 0.0 };
```

```
        void project_prices_();
        void calc_payoffs_(OptType opt_type);
};
```

Note that `BinomialLatticePricer` takes in a `VanillaOption` at construction that is stored as a data member. This class stores and provides accessors to the time to maturity and the payoff type (as before in Ch 3, call or put), as shown in its declaration:

```
#include "Payoffs.h"
#include <memory>

class VanillaOption
{
public:
        VanillaOption(std::unique_ptr<Payoff> payoff, double time_to_exp);
        double option_payoff(double spot) const;
        double time_to_expiration() const;

private:
        std::unique_ptr<Payoff> payoff_ptr_;
        double time_to_exp_;
};
```

Along with the `VanillaOption` argument, the `BinomialLatticePricer` constructor takes in the market data necessary to compute the values necessary for the Jarrow-Rudd discretization. The length of a time step, the u and d factors, and the probability p of an upward move in the underlying equity price are then calculated and set in the body of the constructor.

```
#include "BinomialLatticePricer.h"

#include <algorithm>
#include <limits>
#include <utility>        // std::move

BinomialLatticePricer::BinomialLatticePricer(VanillaOption&& opt,
      double vol, double int_rate, int time_steps, double div_rate) :
      opt_{ std::move(opt) }, time_points_{ time_steps + 1 }, div_rate_{ div_rate }
{
      double dt{ opt_.time_to_expiration() / (time_steps) };
      u_ = std::exp(vol * std::sqrt(dt));
      d_ = 1.0 / u_;
      p_ = 0.5 * (1.0 + (int_rate - div_rate - 0.5 * vol * vol) * std::sqrt(dt) / vol);
      disc_fctr_ = std::exp(-(int_rate)*dt);
}
```

Note, that because a `VanillaOption` object holds a unique pointer to its payoff, we need to explicitly move it in the constructor initializations.

The public member function `calc_price(.)` serves as an interface that sets the pricing computation in motion. It takes in the spot price of the underlying equity and the option type – call or put – as inputs.

The labor is then divided between two private member functions that project the underlying prices from the initial node out to expiration (`project_prices_()`) and traverse the lattice back to calculate the payoffs at each node (`calc_payoffs_(.)`).

```
double BinomialLatticePricer::calc_price(double spot, OptType opt_type)
{
      spot_ = spot;
```

```
        project_prices_();
        calc_payoffs_(opt_type);

        return opt_price_;
    }
```

The `MultiArray` now comes into play as the lattice (`grid_`) is first traversed forward in time by column, with each row element updated with an up move, except for the last, which is a down move. This results in the upper diagonal form as shown in Figures 11-2 and 11-3.

```
void BinomialLatticePricer::project_prices_()
{
    grid_.resize(boost::extents[time_points_][time_points_]);

    grid_[0][0].underlying = spot_;        // Terminal node

    // j: columns, i: rows.
    // Traverse by columns, then set node in each row.
    for (unsigned j = 1; j < time_points_; ++j)
    {
        for (unsigned i = 0; i <= j; ++i)
        {
            if (i < j)
            {
                grid_[i][j].underlying = u_ * grid_[i][j - 1].underlying;
            }
            else // (i == j)
            {
                grid_[i][j].underlying = d_ * grid_[i - 1][j - 1].underlying;
            }
        }
```

```
    }
}
```

Backward iteration over the lattice starts at expiration in step (1) as noted in the code comments in the `calc_payoffs_(.)` function shown next. This occurs in the final column of the `MultiArray` member object `grid_`, where the raw payoffs are trivially computed and set on the `payoff` value of each `Node` struct in the column.

```
void BinomialLatticePricer::calc_payoffs_(OptType opt_type)
{
    for (int j = time_points_ - 1; j >= 0; --j)
    {
        for (int i = 0; i <= j; ++i)
        {
            if (j == time_points_ - 1)      // Payoffs at expiration (1)
            {
                grid_[i][j].payoff = opt_.option_payoff(grid_[i][j].underlying);
            }
            else // (2)
            {
                double expct_val = p_ * grid_[i][j + 1].payoff;
                expct_val += (1.0 - p_) * grid_[i + 1][j + 1].payoff;
                expct_val *= disc_fctr_;
                if (opt_type == OptType::American)        // (3)
                    grid_[i][j].payoff = std::max(expct_val,
                        opt_.option_payoff(grid_[i][j].underlying));
                else // OptType::Euro
                    grid_[i][j].payoff = expct_val;
            }
        }
    }
```

```
        opt_price_ = grid_[0][0].payoff;        // (4)
    }
```

Iterating back through each time step prior to expiration (step (2)), the discounted expected values are computed by applying the up and down probabilities `p` and `1 - p` to the prospective payoffs and adding the results, and then multiplying this sum, stored in `expct_val`, by the `disc_fctr_` value. In the case of an American option (step (3)), the discounted expected payoff is compared to what the raw payoff of the option would be at that point, using the `option_payoff(.)` function on the `VanillaOption opt_` member object itself. If this value is greater than the incumbent, then the `payoff` amount is reset to the higher value. This process continues back to the origin node, which determines the option value. This is set on the `opt_price_` member (step (4)) and returned by the public `calc_price(.)` member function.

Applying the same data as in the American option example, the option value is that at t = 0 in Figure 11-3 above, \$4.54. For comparison, a European option with all else being the same, the value comes out to \$3.98, so the additional premium for early exercise is evident.

```
double strike =  40.0, rf_rate = 0.06, mkt_vol = 0.2, time_to_exp = 1.0;
int time_steps = 4;
double spot = 36.0;

// ITM American put option
auto pp = make_unique<PutPayoff>(strike);           // pp = "put pointer"
VanillaOption put(std::move(pp), time_to_exp);
BinomialLatticePricer put_pricer{ std::move(put), mkt_vol, rf_rate, time_steps };     // div_rate = 0 (default)

double opt_price = put_pricer.calc_price(spot, OptType::American);          // 4.54
```

```
// Compare with European case:
opt_price = put_pricer.calc_price(spot, OptType::Euro);                    // 3.98
```

Convergence

The previous examples purposely used only a handful of timepoints in order to demonstrate how binomial pricing can be implemented using a Boost `MultiArray`. In practice, more nodes are necessary to ensure reasonable convergence. Using the American put option example, if we were to instead run a series of lattices out to 50 timepoints, we would see convergence to 4.49, meaning the result in the previous four-timestep example of 4.54 was off by about 1.1%.

As a side note, convergence of lattice valuations will usually exhibit an oscillating pattern. Staying with the current American put example, looking at the results where the number of timepoints is incremented from 45 to 50 (44 to 49 time steps), the results—out to four decimal points—are as follows.

Table 8-5. Projected Underlying Equity Prices with Discounted Expected Payoffs (ITM American Put Option with Dividend)

Number of Timepoints	Option Valuation
45	4.4869
46	4.4901
47	4.4862
48	4.4910
49	4.4855
50	4.4915

Because of this effect, it is often preferred to take the average of the last two iterations as the option value, giving us 4.4885, but still resulting in 4.49 rounded to the nearest cent. For a more in-depth illustration, the James *Option Theory* book (op cit) **{10}** is again recommended.

Extensions

A two-dimensional Boost MultiArray can be extended to accommodate a trinomial lattice, often used in the presence of stochastic interest rates or volatilities. A MultiArray can also be expanded into higher dimensions for valuing options on more than one asset. However, in this latter case, perfomance can be degraded with additional dimensions, so Monte Carlo-based methods might be a better alternative for valuing derivatives such as basket options.

Accumulators

The *Boost.Accumulators* library provides efficient incremental descriptive statistical computations on dynamic sets of data. For example, a set of financial data -- eg prices or returns -- might be sequentially updated with new data values. Then, each time a new value is appended, a set of descriptive statistical values – such as mean, variance, maximum, and minimum – are updated as well.

The headers `boost/accumulators/accumulators.hpp` and `boost/accumulators/statistics/stats.hpp` need to be included, plus individual header files for each descriptive statistic desired, as will be shown in the examples that follow.

Max and Min Example

As a first example, let us apply the `max` and `min` accumulators to a set of real numbers that is updated with new data over time. Note that individual header files need to be included for both `max` and `min`.

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics/stats.hpp>
#include <boost/accumulators/statistics/min.hpp>
#include <boost/accumulators/statistics/max.hpp>
```

Accumulators, *accumulator sets*, and *extractors* -- for accessing each statistical metric - are scoped with the `boost::accumulators` namespace, but to save ourselves some typing, we can assign it to an alias:

```
namespace bacc = boost::accumulators;
```

An *accumulator set* refers to a collection of accumulators. So now in this case, we create an accumulator set containing the `max` and `min` accumulators:

```
bacc::accumulator_set<double, bacc::stats<bacc::tag::min, bacc::tag::max>> acc{};
```

Note that an `accumulator_set` is a class template, first taking in as template parameters the type (`double`), followed by the set of statistical measures, scoped with the `boost::accumulators::tag` namespace.

Using `acc(.)` as a function object, data can be appended incrementally:

```
acc(5.8);
acc(-1.7);
acc(2.9);
```

To access the sample statistics, use the extractors for `max` and `min`.

```
cout << bacc::extract::min(acc) << ", " << bacc::extract::max(acc) << "\n";
```

The minimum and maximum values at this stage are -1.7 and 5.8. Next, append a new value to the data, and check the results on the updated set. This shows how the maximum and minimum values are automatically updated as each new value is appended to the accumulator:

```
acc(524.0);
```

The minimum value remains unchanged, but the new maximum is 524.

WARNING

As both `max(.)` and `min(.)` functions are also included in the Standard Library, the code in the previous example can show why namespaces are important. If `std` and `boost::accumulators` were imported into the global namespace with

```
using namespace std;
using namespace boost::accumulators;
```

the compiler would complain if either `max` or `min` were used without its namespace context.

Mean and Variance

We can also create `mean` and `variance` accumulator sets. Again, individual header files for mean and variance need to be included. The following example creates an accumulator set containing `mean` and

`variance` accumulators, and then extracts the corresponding values as new data is added to the data set. Note once again, the respective header files must be included.

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

. . .

bacc::accumulator_set<double, bacc::stats<bacc::tag::mean, bacc::tag::variance>> mv_acc{};

// push in some data . . .
mv_acc(1.0);
mv_acc(2.0);
mv_acc(3.0);


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

This results in a mean of 2, and a *population variance*—the sum of squares divided by the entire sample size—equal to approximately 0.666667.

Next, append two additional values:

```
mv_acc(4.0);
mv_acc(5.0);

cout << bacc::extract::mean(mv_acc) << ", " << bacc::extract::variance(mv_acc) << "\n";
```

The mean and variance values have been updated and are now 3 and 2, respectively.

Append three more values to bring the total data set size to eight, so that the accumulators recalculate the mean and variance values again:

```
mv_acc(16.0);
mv_acc(17.0);
mv_acc(18.0);

cout << bacc::extract::mean(mv_acc) << ", " << bacc::extract::variance(mv_acc) << "\n\n";
```

This now yields the values 8.25 and 47.4375.

Rolling Mean and Variance

In the previous section, we looked at accumulators that returned cumulative statistical values. For example, if there are three data values, then the mean is equivalent to the sum of these values divided by 3. If two new values are loaded into the accumulator, then the mean is updated by summing all five elements and dividing by 5.

Typical metrics used in trading indicators and signals, however, rely on rolling values, such as a moving average over a fixed number of observations. In Boost, these values can be obtained by using *rolling window* accumulators. Before proceeding, however, it is important to note that the rolling variance accumulator computes the *sample* variance, where for a sample size of n, the sum of squares is divided by $n-1$ rather n. The price volatility is then the square root of the variance, as there is no standard deviation accumulator in Boost.

Let us first look at a simple example to illustrate how rolling mean and rolling variance accumulators work. To demonstrate, we will define an accumulator set, `ma_acc`, consisting of `rolling_mean` and

`rolling_variance` accumulators. The rolling period is five observations, reflected in the `rolling_window::window_size` parameter, as shown here:

```
// Include header files for rolling mean and rolling variance:
#include <boost/accumulators/statistics/rolling_mean.hpp>
#include <boost/accumulators/statistics/rolling_variance.hpp>

bacc::accumulator_set<double, bacc::stats<bacc::tag::rolling_mean,
      bacc::tag::rolling_variance>> roll_acc{ bacc::tag::rolling_window::window_size = 5 };
```

Next, load the first three observations (same as before):

```
roll_acc(1.0);
roll_acc(2.0);
roll_acc(3.0);
```

One more thing to note at this point is if the rolling mean and rolling (sample) variance are extracted, their values based on these three observations alone will be computed, as five observations are not yet available.

```
      cout << bacc::extract::rolling_mean(roll_acc) << ", "
            << bacc::extract::rolling_variance(roll_acc) << "\n\n";
```

The code above will display 2 and 1 on the screen (mean and variance over three observations).

Next, load two more observations, and extract the values again.

```
roll_acc(4.0);
roll_acc(5.0);
```

The mean and variance are now based on all five elements, yielding 3 and 2 respectively.

Finally, append three more values:

```
roll_acc(16.0);
roll_acc(17.0);
roll_acc(18.0);
```

The mean and variance are then based on the five last elements, resulting in 12.0 and 47.5.

Trading Indicator Examples

In a *Bollinger Band* trading strategy, signals are based on an indicator consisting of a rolling average of security prices (the middle band), and two outer bands defined by $\pm$ a multiple of the rolling average of the price volatility (standard deviation).

The frequency used for rolling averages of prices in trading indicators can be in terms of an arbitrary unit, ranging from high-frequency fractions of a second, to mid-frequency hourly or daily observations, and out to lower frequencies monthly, or quarterly. In the figure below **{11}**, hourly observations with a window of 20 hours are used, with bands of $\pm$ 2 standard deviations from the moving average.

NOTE

Boost uses the term “rolling average”, but in trading parlance this is usually referred to as a *moving average*.

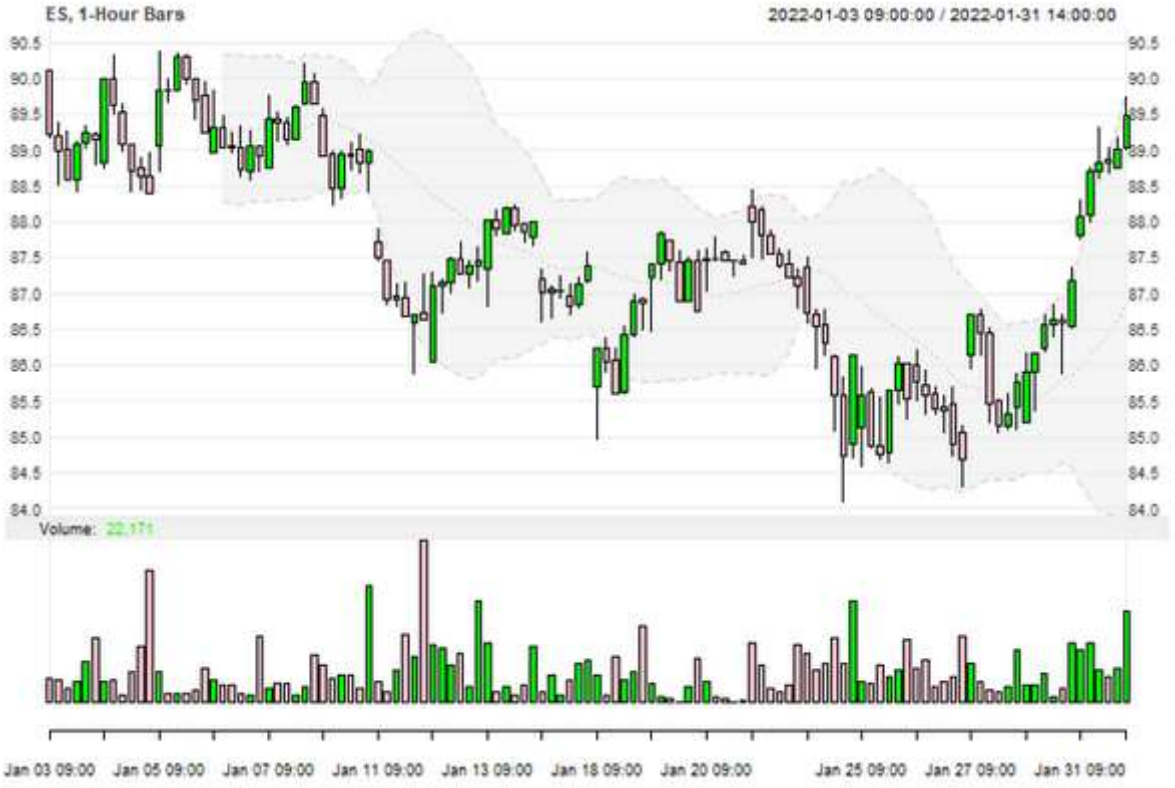

Figure 11.1: Bollinger Band Indicator, 20-hour window, two standard deviations (TTR R package).

Suppose now for a programming example we have a `vector` called `prices` that contains a set of daily stock price data from an external source over a certain period of time. What we will do now is append each observed price to the `prices_acc` accumulator set, and extract the rolling mean and variance with the length set in `win_size`. As these calculations are updated with each new observation added to the accumulator, they will be extracted at each step, where the share price, moving average value, and upper and lower band values will be stored in an Eigen matrix.

```
vector<double> prices{100.0, . . .};

// win_size = length of the inner band moving average.
// Assume win_size = 20 is chosen.
```

```
bacc::accumulator_set<double, bacc::stats<bacc::tag::rolling_mean, bacc::tag::rolling_variance>>
      prices_acc(bacc::tag::rolling_window::window_size = win_size);
MatrixXd indicators{ prices.size(), 4 };
unsigned rec{ 0 };

for (double price : prices)
{
      // Columns of matrix:  price, ma, ma + n*sig, ma - n*sig
      indicators(rec, 0) = price;
      prices_acc(price);
      if (rec >= win_size - 1)
      {
            indicators(rec, 1) = bacc::extract::rolling_mean(prices_acc);
            double dev = n * std::sqrt(bacc::extract::rolling_variance(prices_acc));
            indicators(rec, 2) = indicators(rec, 1) + dev;
            indicators(rec, 3) = indicators(rec, 1) - dev;
      }
      else
      {
            indicators(rec, 1) = 0.0;
            indicators(rec, 2) = 0.0;
            indicators(rec, 3) = 0.0;
      }

      ++rec;
}
```

The values stored in the `MatrixXd` object can then be used as Bollinger Bands indicators in a backtest.

For a small-scale example, suppose we have a sample of 25 daily price observations, and set the inner band moving average to five days, and the volatility multiplier to 1.5, we could put each band into a col-

umn so that the backtest can check if/where either band had been crossed. Zeros have been placed in the accumulator value columns until five observations—the length of the moving average—has been attained.

Price	MA	Upper Band	Lower Band
100.00	0.00	0.00	0.00
103.49	0.00	0.00	0.00
102.82	0.00	0.00	0.00
106.86	0.00	0.00	0.00
104.91	103.61	107.43	99.80
107.38	105.09	108.10	102.08
107.46	105.88	108.89	102.88
111.01	107.52	110.83	104.21
112.01	108.55	112.92	104.19
114.11	110.39	114.80	105.99
116.91	112.30	117.59	107.01
121.74	115.16	121.64	108.68
120.04	116.96	123.01	110.92
120.24	118.61	123.21	114.01
120.12	119.81	122.46	117.16
120.61	120.55	121.60	119.50
121.31	120.47	121.25	119.68
119.25	120.31	121.43	119.18
118.11	119.88	121.75	118.02
120.36	119.93	121.82	118.04
117.36	119.28	121.69	116.87
119.12	118.84	120.56	117.12
119.36	118.86	120.60	117.12
123.54	119.95	123.37	116.53
123.42	120.56	124.72	116.40

As a second example, two separate moving average accumulators can be used to represent the indicators in a fast(shorter length)/slow (longer length) moving average crossing strategy. In the figure below, a fast moving average over 10 days and a slow moving average of 50 days is used (ibid)**{11}**:

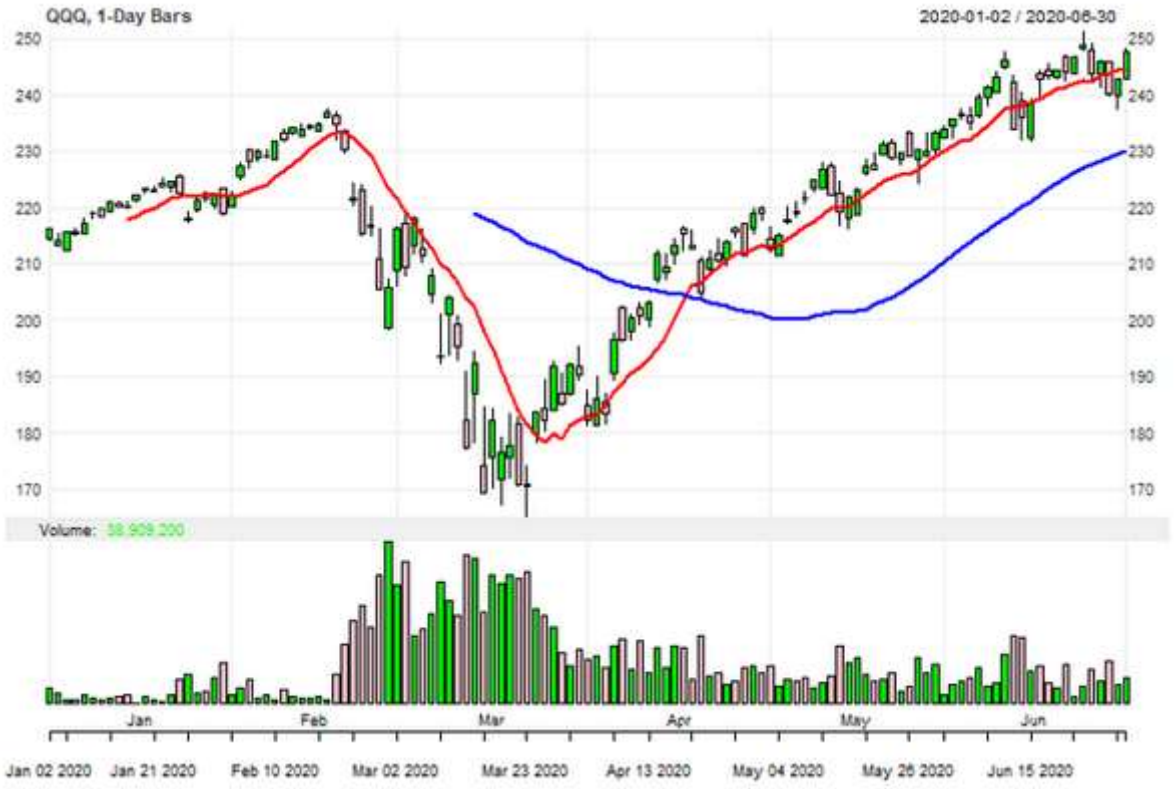

Figure 11.2: Dual Moving Average Cross Indicators (TTR R package)

In code, the moving averages can be applied using two Boost accumulators: one for the fast MA (`fast_ma_acc`), the other for the slow MA (`slow_ma_acc`):

```
bacc::accumulator_set<double, bacc::stats<bacc::tag::rolling_mean>>
      fast_ma_acc(bacc::tag::rolling_window::window_size = fast_ma_win);

bacc::accumulator_set<double, bacc::stats<bacc::tag::rolling_mean>>
```

```
        slow_ma_acc(bacc::tag::rolling_window::window_size = slow_ma_win);

    MatrixXd indicators{ prices.size(), 3 }; // Three columns: Price, Fast(Short) MA, Slow(Long) MA
    unsigned rec{ 0 };

    for (double price : prices)
    {
        // Columns of matrix:  price, ma, ma + n*sig, ma - n*sig
        indicators(rec, 0) = price;
        fast_ma_acc(price);
        slow_ma_acc(price);
        if (rec >= fast_ma_win - 1 && rec >= slow_ma_win - 1)
        {
            indicators(rec, 1) = bacc::extract::rolling_mean(fast_ma_acc);
            indicators(rec, 2) = bacc::extract::rolling_mean(slow_ma_acc);
        }
        else if (rec >= fast_ma_win - 1 && rec < slow_ma_win - 1)
        {
            indicators(rec, 1) = bacc::extract::rolling_mean(fast_ma_acc);
            indicators(rec, 2) = 0.0;
        }
        else
        {
            indicators(rec, 1) = 0.0;
            indicators(rec, 2) = 0.0;
        }

        ++rec;
    }
```

Again, because the moving averages are updated with each new share price, their values need to be captured and stored in a separate container. As in the previous example, we can use a matrix as shown, over

which a backtest will determine if and where one moving average line crosses over the other.

Taking the same set of prices, a fast moving average over five days and a slower one over 10, the results would be as follows. Note again the moving average values are set to zero until the moving average length has been reached in each of the fast and slow cases.

```
Price  Short MA     Long MA
100.00   0.00   0.00
103.49   0.00   0.00
102.82   0.00   0.00
106.86   0.00   0.00
104.91 103.61   0.00
107.38 105.09   0.00
107.46 105.88   0.00
111.01 107.52   0.00
112.01 108.55   0.00
114.11 110.39 107.00
116.91 112.30 108.69
121.74 115.16 110.52
120.04 116.96 112.24
120.24 118.61 113.58
120.12 119.81 115.10
120.61 120.55 116.43
121.31 120.47 117.81
119.25 120.31 118.63
118.11 119.88 119.24
120.36 119.93 119.87
117.36 119.28 119.91
119.12 118.84 119.65
119.36 118.86 119.58
```

```
123.54 119.95 119.91
123.42 120.56 120.24
```

Further information on Boost Accumulators can be found in the online documentation **{12}**.

Conclusion

Although the history of Boost library releases go back to 1999, they tie in with modern C++ in that some of the new features beginning in C++11—such as smart pointers, distributional random number generation, and task-based concurrency—saw their start in Boost years before being adopted by the by the ISO C++ Committee.

There are also Boost libraries that remain separate from the current C++ specification but complement it, including those presented here that can be used in financial programming. There are also cases where Boost can extend features in standard C++, such as providing the commonly used constants $\frac{1}{\sqrt{2}}$ and $\frac{1}{\sqrt{2\pi}}$, and random number generation from a wider range of distributions.

Finally, it should be noted that the `mdarray` class, the owning analog of `mdspan`, might eventually provide in the Standard Library similar functionality **{13}** as Boost MultiArray. This should become clearer once the proposal (P1684) **{14}** is accepted by the ISO Committee, as should its target release, which at present is looking like C++26.

References

{1} Boost official website *https://www.boost.org*

{2} Documentation for Boost mathematical constants:
https://www.boost.org/doc/libs/1_82_0/libs/math/doc/html/math_toolkit/constants_intro.html

{3} Boost source code downloads: https://www.boost.org/users/download/

{4} A full list of the distributions supported by Boost:
https://www.boost.org/doc/libs/1_82_0/libs/math/doc/html/math_toolkit/dist_ref/dists.html

{5} Complements for quantile calculations, Boost documentation:
https://www.boost.org/doc/libs/1_82_0/libs/math/doc/html/math_toolkit/stat_tut/overview/complements.html

{6} Boost Non-Central t-Distribution documentation:
https://www.boost.org/doc/libs/1_82_0/libs/math/doc/html/math_toolkit/dist_ref/dists/nc_t_dist.html

{7} Boost MultiArray documentation: https://www.boost.org/doc/libs/1_82_0/libs/multi_array/doc/user.html

{8} Peter James, *Option Theory* (Wiley 2003), Section 7.3 p 80 (Jarrow-Rudd discretization, multipliers u and d)

{9} Hilpisch, *Python for Finance* (O'Reilly 2018) (1st binomial tree example)
https://learning.oreilly.com/library/view/python-for-finance/9781492024323/

{10} op cit (James) Section 7.3 part (v), pp 84-85 (convergence of lattice model)

{11} TTR R Package (Joshua Ulrich) https://cran.r-project.org/web/packages/TTR/index.html

{12} Boost Accumulators documentation
https://www.boost.org/doc/libs/1_82_0/doc/html/accumulators.html

{13} Hanson, *Pricing Equity Options with `mdarray`*, presentation slides from the Northwest C++ Users Group, May 2023, *https://nwcpp.org/talks/2023/NWCPP_2023_05_MDArray_Final.pdf*

{14} ISO C++ Proposal 1684: *`mdarray`: An Owning Multidimensional Array Analog of `mdspan`* *https://wg21.link/p1684*

For more information about trading indicators such as Bollinger Bands and Moving Averages, the book by Jaekle and Tomasini, *Trading Systems (2E)* is recommended.