Supporting materials for CppCon 2023 Submission

Daniel Hanson

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 8.25 and 47.44.

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, a window of 20 hours is 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*.

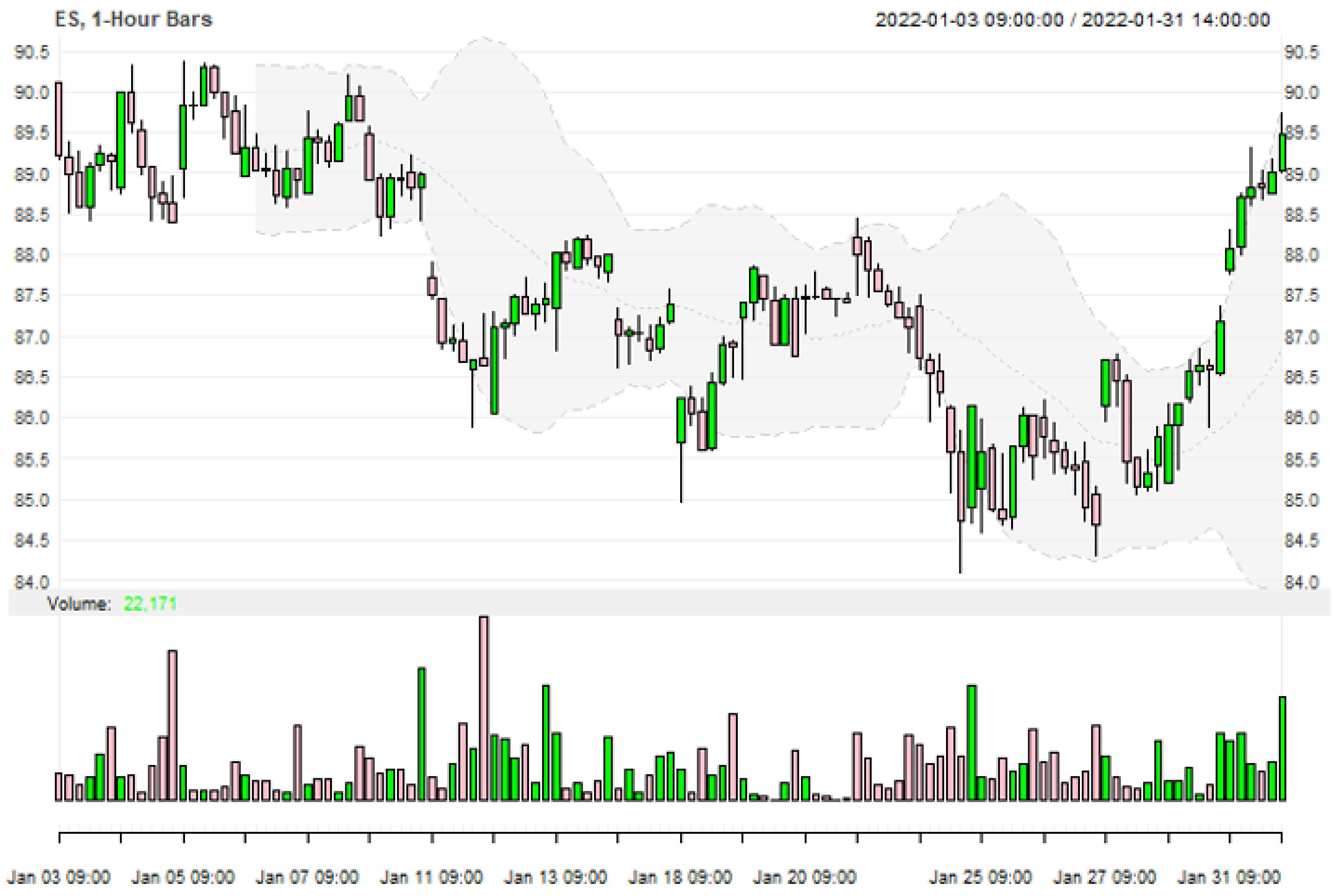

Figure 11.1: Bollinger Band Indicator, 20-hour window, two standard deviations (TTR R package).

Suppose 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

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 column 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.

```
                 Upper  Lower
 Price    MA      Band   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:

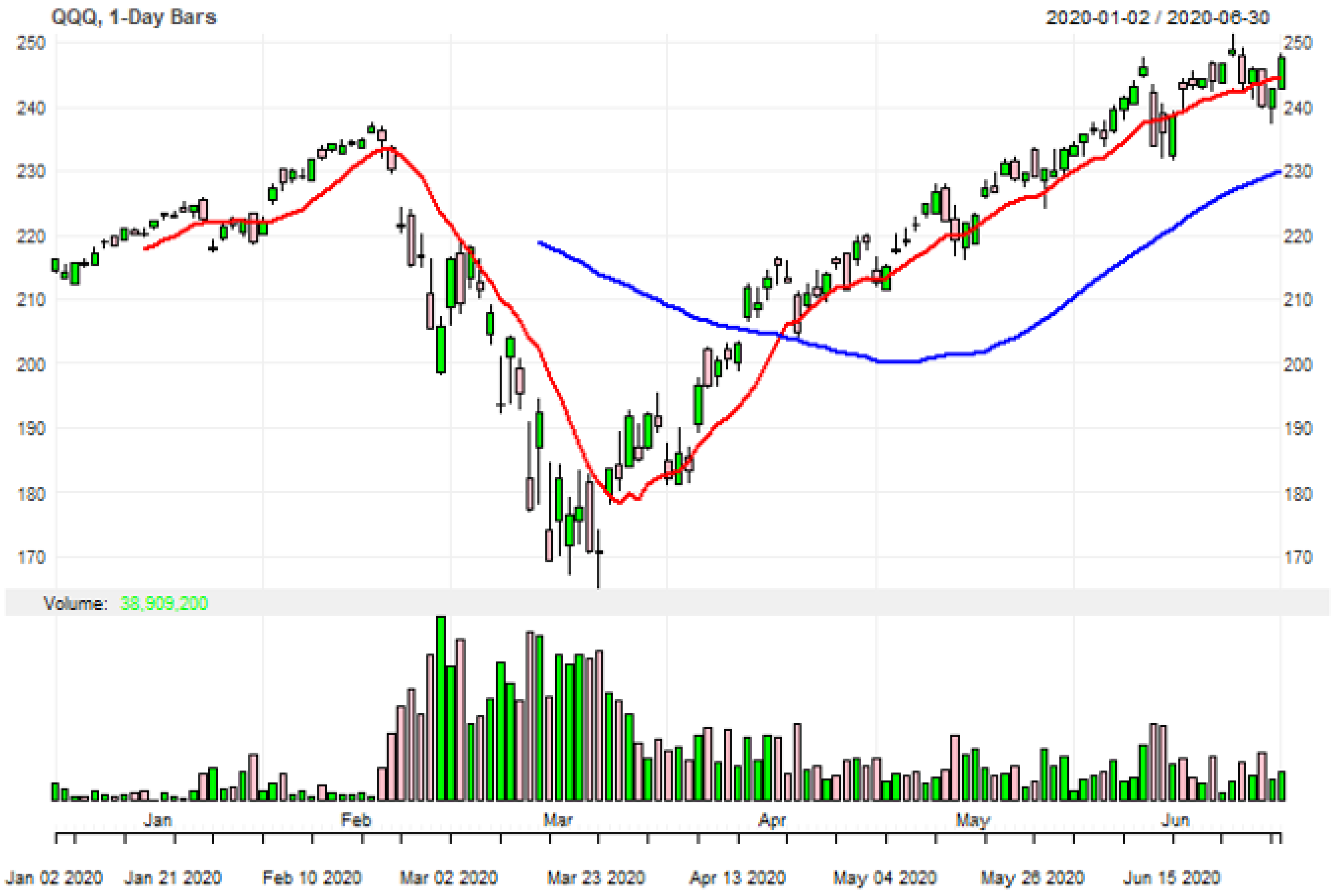

Figure 11.2: Dual Moving Average Cross Indicators (TTR R package)

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

Last updated 2023-06-25 17:25:26 -0700