THESIS

IMPLEMENTATION AND EVALUATION OF AN ORDER FLOW IMBALANCE TRADING ALGORITHM

Submitted by

Carl Reed Jessen

In partial fulfillment of the requirements for the Degree of Master of Science in

Predictive Analytics

Northwestern University

Fall 2015

Advisor: Ernest Chan

Acknowledgements

I would like to express my sincere gratitude to my advisor Ernest Chan for guiding and advising me during this long process. His insights were the seed for this work and benefited greatly from his experience.

This would not have been possible without the generous support of Lime Brokerage who granted me access to their powerful simulation algorithms and Strategy Studio software suite. It was a wonderful experience being able to work with the world class software they have developed.

I also must thank Danial Ranjh, my good friend, for being kind enough to assist me in coding many of the advanced features required for true implementation of such a low latency algorithm. He has taught me a great deal regarding proper software engineering methodology as well as about life. I continue to learn from him to this day.

Finally, I would like to offer my sincere gratitude to my wife, Nhung, for graciously putting up with my busy schedule these last few years. I would never have been able to work full-time, study, and produce this work if I lacked a strong partner at home.

Historical Context of High-Frequency, Low Latency Trading

Since the late 1980's, electronic trading has been taking an ever increasing share of the global securities exchange market and providing market participants with ever lower trade latencies. In 1989, the world's first high frequency trading firm, Automated Trade Desk was able execute orders in 1 second, faster than any human trader at the time, through a satellite dish bolted to the roof of the garage the firm was founded in (Philips 2013). A little more than a decade later, in 2000, execution speeds had fallen to 25 milliseconds. By 2010, execution speeds had fallen below 1 millisecond(Cont 2011). Today, electronic traders hold a dominant position in the markets.

An interesting result of the world's evolution from an entirely manual market to electronic based market systems is that data describing supply, demand, and price behavior in securities markets is being increasing recorded. It is now possible to watch price discovery develop in real time, play it back, and analyze it from every angle. The formulaic and mechanical nature of electronic trading makes statistical analysis of price discovery at very short time intervals possible.

A great deal of interest has developed around modeling imbalances in the market. Specifically, analysis of limit order book dynamics at short intervals has become a topic of interest given the availability of ever increasing granularity of data.

Previous academic work by Lee and Ready 1991, and Benedikstdottir 2006, and has shown that changes in order book information can be predictive of future prices

changes. These methodologies however, are difficult or impossible to apply under live trading constraints.

In 2010, Rama Cont and his coauthors proposed a stochastic model of the limit order book which conceptualized it as a queuing system. This "stylized version limit order book" model contemplates a limit order book as a continuous-time Markov process in which limit orders arrive and wait in a queue until removed from the book by either cancellation or matched with a marketable order. The model enables an observer to determine the volume of limit orders at each given price level at any given point in time[1]. Cont's model is motivated by the desire to use information on the current state of the order book to analyze short term price behavior in a given security.

The Cont model has a set of meaningful advantages for those attempting to analyze trading behavior at subsecond and submillisecond timeframes. It can be estimated quickly using high frequency price data, it creates a model which shares the same features as an empirical order book making easy for a human understand, and the analytical mathematics are trivial given standard computational toolsets.

In the original 2010 paper, Cont et al. found that they were able to meaningful predict changes in price midpoint, execution of an order at the best bid before the best ask quote moves, and execution of both a buy and a sell order at the best quotes

[1] (Cont et al. 2010)

before the price moves using a two-sided Laplace transform. Others have also found the Cont model effective and have built upon it. Lee and Kim applied the Cont model to Korean KOSPI 200 futures market with a slight modification and found modest success[2]. Avellaneda et al. determined that this model can be used as a baseline to estimate the amount of hidden liquidity in a given marketplace allowing trading venues to be ranked in terms of their "information content"[3].

Building upon the success of the 2010 model, in 2014 Cont et al. published a follow-on paper titled *The Price Impact of Order Book Events* which found that that Order Flow Imbalance (OFI) derived from the limit order book model has a statistically significant correlation to contemporaneous price movement at very short time frames[4]. The purpose of this work is to build and test a predictive model based on Cont's descriptive work. More specifically, if change in OFI in time interval $[t_{k-1}, t_k]$ is significantly correlated with price change within the same timeframe, does this correlation similarly hold true on a forward looking basis if price change is advanced one period in the future, $[t_k, t_{k+1}]$?

In order to analyze and apply the Cont model it is important to first understand how it attempts to model the continuous process of price discovery by constructing a snapshot of the market state at any given moment in time, *t*. The snapshot framework

[2] (Lee and Kim 2013)
[3] (Avellaneda et al. 2010)
[4] (Cont et al. 2014)

allows computationally easy analysis of the price discovery by comparing the changes in the status of a limit order book from time *t-1* to time *t*, this difference is denoted as k. The snapshot itself is an interplay between two opposing forces, buyers and sellers. Buyers are able to affect the number of best bid limit orders newly listed since the last period, L_k^b, the number of best bid limit orders newly canceled since the last period, C_k^b, as well as the number of market buy orders arriving, M_k^b. Buyers also influence the best bid price, P_k^b. Conversely, sellers influence the number of newly listed best limit ask orders L_k^s, newly canceled best limit ask orders, C_k^s, new market orders, M_k^s, and the best ask price, P_k^s. The volume of all orders beyond the best bid/ask are aggregated into a single variable D.

Using these simple observed parameters, a liner relationship between order flows and price change by can be developed[5] (δ being an arbitrary tick size):

$$\Delta P_k^b = \delta \frac{L_k^b - C_k^b - M_k^s}{D}$$

$$\Delta P_k^s = \delta \frac{L_k^s - C_k^s - M_k^b}{D}$$

Viewing price change as a function of the additive changes made between two arbitrary points in time enables the use of standard descriptive models to determine the correlation of order book changes and price.

[5] (Cont et al. 2014)

Building on this basic model, Cont et al. 2014 introduces the concept of an order flow imbalance (OFI), sometimes called the net order flow, as a possible predictor of price change. Generally, an OFI is the difference between the volume of orders on either side of the order book and can be defined for any give k as:

$$OFI_k = L_k^b - C_k^b - M_k^s - L_k^s - C_k^s - M_k^b.$$

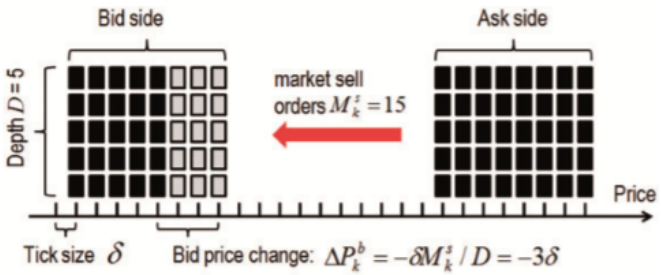

Figure 1 Market sell orders remove M^s shares from the bid (gray squares represent net change in the order book).

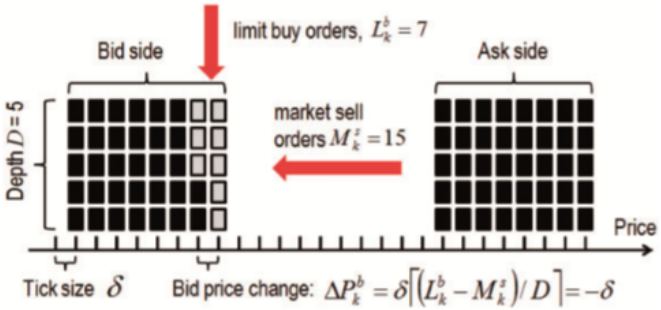

Figure 2 Market sell orders remove M^s shares from the bid, while limit buy orders add L^b shares to the bid.

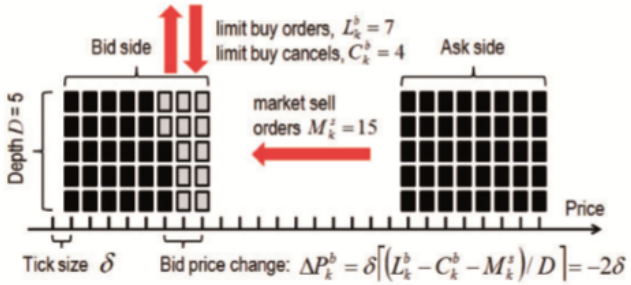

Figure 3 Market sell orders and limit buy cancels remove $M^s + C^b$ shares from the bid, while limit buy orders add L^b shares to the bid.

Figures: Cont et al. 2014

Once one has calculated OFI for any given period of time, it is then trivial to analyze the linear relationship between price change and OFI:

$$\Delta P_{k,i} = \beta_i OFI_{k,i} + \epsilon_{k,i} \text{ where}$$

$$\beta_i = \frac{c}{D_i^{\lambda}} + v_i$$

Cont's et al. estimated this stylized model over 1 month (April 2010) of trading for a panel of equities listed on the S&P500 finding that β_i is statistically significant in 98% of cases with a surprising high average R^2 value of 65%. Results of this significance given such a simple model are profound and warrant further investigation.

The purpose of our work in this paper is to build upon Cont et al.'s results to build predictive model trading model and test it under realistic market conditions. In support of this goal, Lime Brokerage generously provided us access to the advanced back testing tool, Lime Strategy Studio (Lime). Lime is fast enough to back test full depth of book intraday strategies and has an integrated tick-by-tick fill simulation algorithm. This provides a higher degree of verisimilitude than market simulations used by Cont et al. because, in their work, all trades occurred at the midpoint price due to the computational difficulty in estimating orders fills. Using the Lime platform, we are able to more closely approximate whether orders at the best bid and best ask are actually filled at a what price. Additionally, the Lime system allows for the inclusion for transaction fees which are absent from Cont et al.'s work.

Whereas the Cont's estimations pegged the values of c and λ to $\frac{1}{2}$ and 1 respectively, under actual trading conditions, the value of β_i, the price impact coefficient, would likely vary from security to security due to issues like liquidity as well as from time period to time period due to forces like the volatility smile. Lime Strategy Studio allowed for the continuous estimation of c and therefore β_i, thus decreasing the magnitude of the error term $\epsilon_{k,i}$. Additionally, Cont's $\Delta P_{k,i}$ estimations were construed to be the midpoint price based on TAQ consolidated quotes. Lime's fill simulation algorithm allows for a more granular analysis by analyzing actual fills at the best bid/ask and using full level 2 data depth of book data. Finally, Cont's use of consolidated TAQ is unrealistic. Equities in real markets are traded on many different exchanges which are geographically diverse. Knowledge of events occurring on one exchange are not likely to propagate through all exchanges within the time constraints in Cont's model. The use of consolidated data carries the assumption that this market disparity does not exist. While this difference may be trivial for extremely high liquidity securities, live trading algorithms based on Cont's work may result in losses due to slippage in the real market.

Data

Our data set comprised two electronically traded funds (ETF) where analyzed, the SPDR S&P 500 ETF Trust (ticker SPY) and PowerShares QQQ Trust (QQQ). These securities were chosen because their high daily trade volume and typically

narrow bid-ask spread were likely to minimized our models error terms. Level 2 full depth of book data from ARCA was acquired from OneTick which was used by the Lime fill simulation engine to simulate the likelihood of any given order being filled. We estimated our model over 3 different full months in 2012, April, June, August as chosen by a random number generator.

Model Specification

Our model heavily leveraged the C++ libraries as provided in the Strategy Studio SDK. $OFI_{k,i}$ for the time period $[t_{k-1}, t_k]$ was updated every millisecond through the additive method illustrated above and D_i^{λ} was similarly updated at ever millisecond tick via simple observation. A forward looking $\Delta P_{k,i}$ for period $[t_k, t_{k+1}]$ was estimated by running a linear regression of the observed OFI values and observed D_i^{λ} over a 30 minutes using the rolling window object provided by Lime. Once the 30-minute rolling window was filled with observations, we compared the predicted ΔP value every millisecond against the known transaction of $.003 fee to take liquidity from the ARCA market[6]. If the predicted price change was larger than this "take fee", a market order of 100 shares was entered depending on the sign of ΔP. If price was predicted to rise, a market buy order was entered whereas a market sell order was sent if the price was predicted to fall. The market impact of our trades was accounted

[6] Brokerage US Execution Rebate and Fee Schedule Exchange and ECNs, Effective May 1st, 2015, Version 150501.1 Published by Lime Brokerage

for by the Lime order fill simulation engine although we do not expect our trading behavior to have a significant impact on market price due to the small size of our trade lots and the high liquidity of our target market. We also attempted to avoid any problems stemming from the well-known intraday volatility seasonality issues of the "volatility smile" by closing out all trading 30 minutes prior to the closing bell. Because commissions fees can vary significantly based on the average monthly volume of a given trader as well as the nature of the relationship between a trade and her brokerage, we have omitted commission fees from our model. If our trading strategy has been shown to be profitable, it would have been prudent to make some assumptions about our hypothetical brokerage relationship and thus add in commission fees. As our model was shown to not be profitable, this additional cost was unnecessary to include. Source code for our model has been provided in appendix A.

Results

Although daily losses were not catastrophic, never exceeding .25%, with limited exception, every trading day of every test produced a loss. The OFI strategy as implemented also failed to beat the QQQ benchmark in monthly returns in all tests and only produced superior daily results in 38.1% of trading days.

	Monthly Results		
	September	June	August
OFI Strategy Returns	-0.8%	-1.5%	-1.2%

QQQ Benchmark Returns	0.8%	5.2%	4.5%	Average
% of Days Strategy > Benchmark	42.1%	38.1%	34.8%	38.1%

September 2012				
Date	Start of Day Equity	Day PnL	Return	QQQ Return
9/4/2012	$100,000.00	-$2.80	0.00%	0.09%
9/5/2012	$99,997.20	-$18.58	-0.02%	-0.03%
9/6/2012	$99,978.62	-$95.67	-0.10%	1.62%
9/7/2012	$99,882.95	-$20.73	-0.02%	0.09%
9/10/2012	$99,862.22	-$76.84	-0.08%	-1.04%
9/11/2012	$99,785.38	-$66.04	-0.07%	-0.16%
9/12/2012	$99,719.34	-$39.36	-0.04%	-0.07%
9/13/2012	$99,679.98	$16.45	0.02%	1.24%
9/14/2012	$99,696.43	-$49.59	-0.05%	0.49%
9/17/2012	$99,646.84	-$19.90	-0.02%	0.01%
9/18/2012	$99,626.94	-$19.34	-0.02%	0.33%
9/19/2012	$99,607.61	-$70.77	-0.07%	0.16%
9/20/2012	$99,536.83	$3.71	0.00%	0.37%
9/21/2012	$99,540.54	-$39.38	-0.04%	-0.45%
9/24/2012	$99,501.16	-$32.63	-0.03%	0.25%
9/25/2012	$99,468.53	-$111.41	-0.11%	-1.66%
9/26/2012	$99,357.12	-$19.05	-0.02%	-0.68%
9/27/2012	$99,338.07	-$102.67	-0.10%	1.16%
9/28/2012	$99,235.40	-$22.67	-0.02%	-0.51%

June 2012				
Date	Start of Day Equity	Day PnL	Return	QQQ Return
6/1/2012	$100,000.00	-$57.51	-0.06%	-0.92%
6/4/2012	$99,942.49	-$50.94	-0.05%	0.50%
6/5/2012	$99,891.56	-$45.26	-0.05%	0.77%
6/6/2012	$99,846.30	-$124.05	-0.12%	1.49%
6/7/2012	$99,722.25	-$135.60	-0.14%	-1.41%
6/8/2012	$99,586.65	-$67.57	-0.07%	1.14%
6/11/2012	$99,519.08	-$186.79	-0.19%	-2.44%
6/12/2012	$99,332.29	-$103.08	-0.10%	0.76%
6/13/2012	$99,229.22	-$28.77	-0.03%	-0.40%
6/14/2012	$99,200.45	-$73.79	-0.07%	0.35%
6/15/2012	$99,126.66	-$52.88	-0.05%	0.99%
6/18/2012	$99,073.78	-$57.80	-0.06%	1.32%
6/19/2012	$99,015.98	-$67.00	-0.07%	0.53%
6/20/2012	$98,948.98	-$54.57	-0.06%	-0.11%
6/21/2012	$98,894.41	-$139.22	-0.14%	-2.37%
6/22/2012	$98,755.19	-$93.87	-0.10%	0.75%
6/25/2012	$98,661.32	-$1.50	0.00%	-1.13%
6/26/2012	$98,659.82	-$33.84	-0.03%	0.30%
6/27/2012	$98,625.98	-$29.20	-0.03%	0.22%
6/28/2012	$98,596.78	-$63.52	-0.06%	-0.35%
6/29/2012	$98,533.26	-$117.59	-0.12%	1.25%

August 2012				
Date	Start of Day Equity	Day PnL	Return	QQQ Return
8/1/2012	$100,000.00	$2.72	0.00%	-0.009808429
8/2/2012	$100,002.72	-$79.50	-0.08%	0.004368175
8/3/2012	$99,923.23	-$240.31	-0.24%	0.003364943
8/6/2012	$99,682.92	-$37.75	-0.04%	0.002426448
8/7/2012	$99,645.17	-$76.00	-0.08%	0.00406749
8/8/2012	$99,569.17	-$5.97	-0.01%	0.002105897
8/9/2012	$99,563.21	-$16.38	-0.02%	0.003002101
8/10/2012	$99,546.82	-$63.61	-0.06%	0.004054663
8/13/2012	$99,483.21	-$52.10	-0.05%	0.002692998
8/14/2012	$99,431.11	-$89.16	-0.09%	-0.003418549
8/15/2012	$99,341.95	-$11.09	-0.01%	0.003283582
8/16/2012	$99,330.86	-$72.52	-0.07%	0.008151771
8/17/2012	$99,258.34	-$40.00	-0.04%	0.002200381
8/20/2012	$99,218.35	-$73.44	-0.07%	0.002197158
8/21/2012	$99,144.91	-$85.45	-0.09%	-0.006994026
8/22/2012	$99,059.46	-$39.55	-0.04%	0.00647154
8/23/2012	$99,019.91	-$25.21	-0.03%	-0.00396243
8/24/2012	$98,994.70	-$81.36	-0.08%	0.008417011
8/27/2012	$98,913.34	-$55.17	-0.06%	-0.002915452
8/28/2012	$98,858.17	-$2.25	0.00%	0.000877963
8/29/2012	$98,855.91	$6.45	0.01%	-0.000146177
8/30/2012	$98,862.36	-$19.10	-0.02%	-0.005872853
8/31/2012	$98,843.26	-$32.50	-0.03%	-0.00058651

Conclusions

This specific application of Cont's OFI model as a predictive indicator has not been shown to produce profitable intraday trading results. That being said, the daily returns from a strategy of buying QQQ on open and selling on and the implemented OFI strategy are not statistically significantly different with a two-tailed P value of 0.1957. This strategy shows promise and further development may be warranted.

Possible Future Research

The results produced in this work may be improved in a variety of ways, each potentially warranting further research. For example, increased estimation accuracy as well as increased computational efficiency may be found by estimation c with a Kalman Filter rather than using the mean of a 30 minute rolling window. There are 2 key advantages to using Kalman Filter's in low latency trading algorithms which this algorithm may benefit from. Because a Kalman filter is only estimated from one previous time step, less time is spent reading and writing to memory. This increased efficiency may be beneficial in a live trading situation. Similarly, under living trading conditions, data points can be lost or not arrive at all. In these instances, a Kalman Filter may be preferable because losing data is does not significantly impact a Kalman filter estimation.

Another modification to this work may be to analyze a set of securities in which all trading is accomplished on a single market, for example the CME e-Mini contracts. By studying a security with a single centralized market, the effects of latency between geographically disparate exchanges is eliminated. Similarly, it may be interesting to analyze securities other than equities. The effects of supply/demand imbalances may be felt differently in commodity or bond markets.

Hidden liquidity may also significantly impact the profitability of a OFI-based trading algorithm. Avellaneda, Reed, and Stoikov 2010 proposed a method of estimating the amount of hidden liquidity percent in a given exchange. This

methodology could perhaps be adapted to target only the exchanges in which the algorithm is likely to be profitable.

Another interesting avenue of research was suggested to us by Rama Cont himself during an email exchange with the authors. D is used in calculating the price impact coefficient but "in principle, one can have an asymmetric price impact, modeled as a piecewise linear function with different slopes for positive and negative imbalances. In that case you could use D_+= depth at the best bid for positive imbalances D_-=depth at the best ask for negative imbalance instead of averaging." In his work however, the sign of the D changes "randomly with higher frequency" as so defining D as the average depth appears to be sufficient.

The fees paid to take liquidity from the ARCA market may have a significant impact on the profitability of this algorithm. Further research on the application of this strategy markets where rebates is paid to take liquidity such as the NASDAQ BSX, Direct Edge "A", or BATS "Y"[7] may prove increase profitability.

Finally, generally volatility in a given market may have an impact on the profitability of this algorithm. Further research focused on tracking the relationship between VIX prices or other indicators of expected volatility may prove fruitful.

[7] Ibid.

Appendix A: Source Code

PriceImpactStrategy.cpp
PriceImpactStrategy.h

```
#ifdef _WIN32
    #include "stdafx.h"
#endif

#include "PriceImpactStrategy.h"

#include "FillInfo.h"
#include "AllEventMsg.h"
#include "ExecutionTypes.h"
#include <Utilities/Cast.h>
#include <Utilities/utils.h>

#include <math.h>
#include <iostream>
#include <cassert>
#include <cstdlib>

using namespace LimeBrokerage::StrategyStudio;
using namespace LimeBrokerage::StrategyStudio::MarketModels;
using namespace LimeBrokerage::StrategyStudio::Utilities;

using namespace std;

PriceImpact::PriceImpact(StrategyID strategyID, const std::string&
strategyName, const std::string& groupName):
    Strategy(strategyID, strategyName, groupName),
    m_instrument_order_id_map(),
    m_aggressiveness(0),
    m_position_size(100),
    m_debug_on(false),
    m_rolling_window_double(180.0),
    m_rollingWindow(180),
    m_time_before_close(30),
    m_nearing_close_time(false),
    m_debug_event_count(0)
{
    //this->set_enabled_pre_open_data_flag(true);
    //this->set_enabled_pre_open_trade_flag(true);
    //this->set_enabled_post_close_data_flag(true);
    //this->set_enabled_post_close_trade_flag(true);
}

PriceImpact::~PriceImpact()
{
}

void PriceImpact::OnResetStrategyState()
{
    m_rollingWindow.clear();
    m_nearing_close_time = false;
}
```

```
void PriceImpact::DefineStrategyParams()
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Defining Strategy Params");

    CreateStrategyParamArgs arg1("aggressiveness",
STRATEGY_PARAM_TYPE_RUNTIME, VALUE_TYPE_DOUBLE, m_aggressiveness);
    params().CreateParam(arg1);

    CreateStrategyParamArgs arg2("position_size",
STRATEGY_PARAM_TYPE_RUNTIME, VALUE_TYPE_INT, m_position_size);
    params().CreateParam(arg2);

    CreateStrategyParamArgs arg3("rolling_window_size",
STRATEGY_PARAM_TYPE_STARTUP, VALUE_TYPE_DOUBLE, m_rolling_window_double);
    params().CreateParam(arg3);

    CreateStrategyParamArgs arg4("time_before_close",
STRATEGY_PARAM_TYPE_STARTUP, VALUE_TYPE_INT, m_time_before_close);
    params().CreateParam(arg4);

    CreateStrategyParamArgs arg5("debug", STRATEGY_PARAM_TYPE_RUNTIME,
VALUE_TYPE_BOOL, m_debug_on);
    params().CreateParam(arg5);
}

void PriceImpact::DefineStrategyCommands()
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Defining Strategy Commands");

    StrategyCommand command1(1, "Reprice Existing Orders");
    commands().AddCommand(command1);

    StrategyCommand command2(2, "Cancel All Orders");
    commands().AddCommand(command2);
}

void PriceImpact::RegisterForStrategyEvents(StrategyEventRegister*
eventRegister, DateType currDate)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Registering for events");

    for (SymbolSetConstIter it = symbols_begin(); it != symbols_end(); ++it)
{
        eventRegister->RegisterForBars(*it, BAR_TYPE_TIME, 10);
        eventRegister->RegisterForMarketData(*it);
    }

    eventRegister->RegisterForSingleScheduledEvent("NearingClose",
USEquityCloseUTCTime(currDate) -
boost::posix_time::minutes(m_time_before_close), true);

    m_nearing_close_time = false;
}

void PriceImpact::OnBar(const BarEventMsg& msg)
{
    if (m_debug_on) {
```

```
        ostringstream str;
        str << msg.instrument().symbol() << ": " << msg.bar();
        logger().LogToClient(LOGLEVEL_DEBUG, str.str().c_str());
    }

    if(msg.bar().close() < .01) return;

    const MarketModels::IAggrOrderBook* order_book;
    const SymbolTag strSymbol = msg.instrument().symbol();
    const MarketModels::Instrument* m_instrument = &msg.instrument();
    double c = 0;
    double c_avg = 0;
    double beta_i = 0;
    double delta_p = 0;

    order_book = &msg.instrument().aggregate_order_book();

    OFI ofi = OFI(order_book);

    LogBarEvent(strSymbol, ofi, portfolio().position(m_instrument),
msg.bar().volume());

    if (msg.bar().volume() == 0 || ofi.Result() == 0)
        return;

    c = msg.bar().volume() / ofi.Result();
    m_rollingWindow.push_back(c);

    // only process when we have a complete rolling window
    if (!m_rollingWindow.full())
        return;

    logger().LogToClient(LOGLEVEL_DEBUG, "Rolling Window Fully Initialized");

    c_avg = m_rollingWindow.Mean();

    beta_i = c_avg / msg.bar().volume();

    delta_p = beta_i * ofi.Result();

    ostringstream str7;
    str7 << strSymbol << " Delta P: " << delta_p;
    logger().LogToClient(LOGLEVEL_DEBUG, str7.str().c_str());



    /*if (side == BUY && portfolio().position(m_instrument) <
m_position_size)
        SendBuyOrder(m_instrument, m_position_size);
    else if (side == SELL && portfolio().position(m_instrument) >=
m_position_size)
        SendSellOrder(m_instrument, m_position_size);*/

    if (m_nearing_close_time) {
        logger().LogToClient(LOGLEVEL_DEBUG, "Nearing close time: closing
positions if exist");
        if (portfolio().position(m_instrument) >= m_position_size)
```

```
            SendSellOrder(m_instrument, m_position_size);
        return;
    }

    if (delta_p > 0)
        SendBuyOrder(m_instrument, m_position_size);
    else if (delta_p < 0)
        SendSellOrder(m_instrument, m_position_size);

    //AdjustPortfolio(&msg.instrument(), m_position_size * side);

}

void PriceImpact::OnDepth(const MarketDepthEventMsg& msg)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "On Depth order book message
arrived");

    ostringstream str;
    str << msg.instrument().symbol() << ": " << msg.name();
    logger().LogToClient(LOGLEVEL_DEBUG, str.str().c_str());
}

void PriceImpact::OnOrderUpdate(const OrderUpdateEventMsg& msg)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "On order update received");
    logger().Log(LOGLEVEL_INFO, "On order update received");

    if(msg.completes_order())
                m_instrument_order_id_map[msg.order().instrument()] = 0;
}

void PriceImpact::OnScheduledEvent(const ScheduledEventMsg& msg)
{
    if (msg.scheduled_event_name() == "NearingClose") {
        m_nearing_close_time = true;
    }
}

void PriceImpact::AdjustPortfolio(const Instrument* instrument, int
desired_position)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Adjusting Portfolio");

    int trade_size = desired_position - portfolio().position(instrument);

    if (trade_size != 0) {
        OrderID order_id = m_instrument_order_id_map[instrument];
        //if we're not working an order for the instrument already, place a
new order
        if (order_id == 0) {
            SendOrder(instrument, trade_size);
        } else {
                    //otherwise find the order and cancel it if we're now
trying to trade in the other direction
            const Order* order = orders().find_working(order_id);
```

```
            if(order && ((IsBuySide(order->order_side()) && trade_size < 0)
||
                                    ((IsSellSide(order->order_side()) &&
trade_size > 0)))) {
                trade_actions()->SendCancelOrder(order_id);
                //we're avoiding sending out a new order for the other side
immediately to simplify the logic to the case where we're only tracking one
order per instrument at any given time
            }
        }
    }
    else {
        logger().LogToClient(LOGLEVEL_DEBUG, "Trade Size Zero: No trade
made");
    }
}

void PriceImpact::SendOrder(const Instrument* instrument, int trade_size)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Sending order");

    if(instrument->top_quote().ask()<.01 ||
        instrument->top_quote().bid()<.01 ||
        !instrument->top_quote().ask_side().IsValid() ||
        !instrument->top_quote().ask_side().IsValid()) {
        std::stringstream ss;

        ss  << "Sending buy order for " << instrument->symbol() << " at price
" << instrument->top_quote().ask()
            << " and quantity " << trade_size <<" with missing quote data";

        logger().LogToClient(LOGLEVEL_DEBUG, ss.str());
        return;
     }

    double price = trade_size > 0 ? instrument->top_quote().bid() +
m_aggressiveness : instrument->top_quote().ask() - m_aggressiveness;

    OrderParams params(*instrument,
        abs(trade_size),
        price,
        (instrument->type() == INSTRUMENT_TYPE_EQUITY) ?
MARKET_CENTER_ID_NASDAQ : ((instrument->type() == INSTRUMENT_TYPE_OPTION) ?
MARKET_CENTER_ID_CBOE_OPTIONS : MARKET_CENTER_ID_CME_GLOBEX),
        (trade_size>0) ? ORDER_SIDE_BUY : ORDER_SIDE_SELL,
        ORDER_TIF_DAY,
        ORDER_TYPE_LIMIT);

    if (trade_actions()->SendNewOrder(params) ==
TRADE_ACTION_RESULT_SUCCESSFUL) {
        m_instrument_order_id_map[instrument] = params.order_id;
    }
}

void PriceImpact::SendBuyOrder(const Instrument* instrument, int unitsNeeded)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Sending Buy order");
```

```
    OrderParams params(*instrument,
        unitsNeeded,
        (instrument->top_quote().ask() != 0) ? instrument->top_quote().ask()
: instrument->last_trade().price(),
        (instrument->type() == INSTRUMENT_TYPE_EQUITY) ?
MARKET_CENTER_ID_NASDAQ : ((instrument->type() == INSTRUMENT_TYPE_OPTION) ?
MARKET_CENTER_ID_CBOE_OPTIONS : MARKET_CENTER_ID_CME_GLOBEX),
        ORDER_SIDE_BUY,
        ORDER_TIF_DAY,
        ORDER_TYPE_MARKET);

    trade_actions()->SendNewOrder(params);
}

void PriceImpact::SendSellOrder(const Instrument* instrument, int
unitsNeeded)
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Sending sell order");
    OrderParams params(*instrument,
        unitsNeeded,
        (instrument->top_quote().bid() != 0) ? instrument->top_quote().bid()
: instrument->last_trade().price(),
        (instrument->type() == INSTRUMENT_TYPE_EQUITY) ?
MARKET_CENTER_ID_NASDAQ : ((instrument->type() == INSTRUMENT_TYPE_OPTION) ?
MARKET_CENTER_ID_CBOE_OPTIONS : MARKET_CENTER_ID_CME_GLOBEX),
        ORDER_SIDE_SELL,
        ORDER_TIF_DAY,
        ORDER_TYPE_MARKET);

    trade_actions()->SendNewOrder(params);
}

void PriceImpact::RepriceAll()
{
    logger().LogToClient(LOGLEVEL_DEBUG, "Repricing All Command");

    for (IOrderTracker::WorkingOrdersConstIter ordit =
orders().working_orders_begin(); ordit != orders().working_orders_end();
++ordit) {
        Reprice(*ordit);
    }
}

void PriceImpact::Reprice(Order* order)
{
    OrderParams params = order->params();
    params.price = (order->order_side() == ORDER_SIDE_BUY) ? order-
>instrument()->top_quote().bid() + m_aggressiveness : order->instrument()-
>top_quote().ask() - m_aggressiveness;
    trade_actions()->SendCancelReplaceOrder(order->order_id(), params);
}

void PriceImpact::OnStrategyCommand(const StrategyCommandEventMsg& msg)
{
    switch (msg.command_id()) {
        case 1:
            RepriceAll();
```

```
            break;
        case 2:
            trade_actions()->SendCancelAll();
            break;
        default:
            logger().LogToClient(LOGLEVEL_DEBUG, "Unknown strategy command
received");
            break;
    }
}

void PriceImpact::OnParamChanged(StrategyParam& param)
{
    if (param.param_name() == "aggressiveness") {
        if (!param.Get(&m_aggressiveness))
            throw StrategyStudioException("Could not get m_aggressiveness");
    } else if (param.param_name() == "position_size") {
        if (!param.Get(&m_position_size))
            throw StrategyStudioException("Could not get position size");
    } else if (param.param_name() == "m_rollingWindow") {
        if (!param.Get(&m_rolling_window_double)) {
            throw StrategyStudioException("Could not get
rolling_window_size");
        } else
            m_rollingWindow = Analytics::ScalarRollingWindow<double>
(m_rolling_window_double);
    } else if (param.param_name() == "time_before_close") {
        if (!param.Get(&m_time_before_close))
            throw StrategyStudioException("Could not get time_before_close");
    } else if (param.param_name() == "debug") {
        if (!param.Get(&m_debug_on))
            throw StrategyStudioException("Could not get trade size");
    }
}

void PriceImpact::LogBarEvent(const SymbolTag symbol, const OFI ofi, const
int position, const int volume)
{
    ostringstream str1;
    str1 << symbol << " Bid Size: " << ofi.BidSize();
    logger().LogToClient(LOGLEVEL_DEBUG, str1.str().c_str());

    ostringstream str2;
    str2 << symbol << " Ask Size: " << ofi.AskSize();
    logger().LogToClient(LOGLEVEL_DEBUG, str2.str().c_str());

    ostringstream str3;
    str3 << symbol << " OFI: " << ofi.Result();
    logger().LogToClient(LOGLEVEL_DEBUG, str3.str().c_str());

    ostringstream str4;
    str4 << symbol << " Portfolio: " << position;
    logger().LogToClient(LOGLEVEL_DEBUG, str4.str().c_str());

    ostringstream str5;
    str5 << symbol << " Trade Side: " << ofi.TradeSide();
    logger().LogToClient(LOGLEVEL_DEBUG, str5.str().c_str());
```

```
    ostringstream str6;
    str6 << symbol << " Volume: " << volume;
    logger().LogToClient(LOGLEVEL_DEBUG, str6.str().c_str());
}
```

PriceImpactStrategy.h

```
#pragma once

#ifndef _STRATEGY_STUDIO_LIB_EXAMPLES_SIMPLE_MOMENTUM_STRATEGY_H_
#define _STRATEGY_STUDIO_LIB_EXAMPLES_SIMPLE_MOMENTUM_STRATEGY_H_

#ifdef _WIN32
    #define _STRATEGY_EXPORTS __declspec(dllexport)
#else
    #ifndef _STRATEGY_EXPORTS
    #define _STRATEGY_EXPORTS
    #endif
#endif

#include <Strategy.h>
#include <Analytics/ScalarRollingWindow.h>
#include <Analytics/InhomogeneousOperators.h>
#include <Analytics/IncrementalEstimation.h>
#include <MarketModels/Instrument.h>
#include <Utilities/ParseConfig.h>

#include <vector>
#include <map>
#include <iostream>

using namespace LimeBrokerage::StrategyStudio;

enum DesiredPosition {
    BUY=1,
    SELL=-1,
    UNKNOWN=0
};

class BetaI {
public:

    BetaI();
};

class OFI {
public:

    OFI(const MarketModels::IAggrOrderBook* order_book) :
m_bid_size(order_book->TotalBidSize()), m_ask_size(order_book-
>TotalAskSize()) {}
    int BidSize() const {return m_bid_size;}
    int AskSize() const {return m_ask_size;}
    int Result() const {return (m_bid_size - m_ask_size)/2;}

    DesiredPosition TradeSide() const
    {
        if(m_bid_size > m_ask_size)
            return BUY;
        else
            return SELL;
```

```
    }

private:
    int m_bid_size;
    int m_ask_size;

};

class PriceImpact : public Strategy {
public:
    PriceImpact(StrategyID strategyID, const std::string& strategyName, const
std::string& groupName);
    ~PriceImpact();

public: /* from IEventCallback */

    /**
     * This event triggers whenever trade message arrives from a market data
source.
     */
    virtual void OnTrade(const TradeDataEventMsg& msg){}

    /**
     * This event triggers whenever aggregate volume at best price changes,
based
     * on the best available source of liquidity information for the
instrument.
     *
     * If the quote datasource only provides ticks that change the NBBO, top
quote will be set to NBBO
     */
    virtual void OnTopQuote(const QuoteEventMsg& msg){}

    /**
     * This event triggers whenever a new quote for a market center arrives
from a consolidate or direct quote feed,
     * or when the market center's best price from a depth of book feed
changes.
     *
     * User can check if quote is from consolidated or direct, or derived
from a depth feed. This will not fire if
     * the data source only provides quotes that affect the official NBBO, as
this is not enough information to accurately
     * mantain the state of each market center's quote.
     */
    virtual void OnQuote(const QuoteEventMsg& msg){}

    /**
     * This event triggers whenever a order book message arrives. This will
be the first thing that
     * triggers if an order book entry impacts the exchange's DirectQuote or
Strategy Studio's TopQuote calculation.
     */
    //virtual void OnDepth(const MarketDepthEventMsg& msg){}
    virtual void OnDepth(const MarketDepthEventMsg& msg);

    /**
```

```
     * This event contains timed events requested by the strategy
     */
    virtual void OnScheduledEvent(const ScheduledEventMsg& msg);

    /**
     * This event triggers whenever a Bar interval completes for an
instrument
     */
    virtual void OnBar(const BarEventMsg& msg);

    /**
     * This event contains alerts about the state of the market
     */
    virtual void OnMarketState(const MarketStateEventMsg& msg){};

    /**
     * This event triggers whenever new information arrives about a
strategy's orders
     */
    virtual void OnOrderUpdate(const OrderUpdateEventMsg& msg);

    /**
     * This event contains strategy control commands arriving from the
Strategy Studio client application (eg Strategy Manager)
     */
    virtual void OnStrategyControl(const StrategyStateControlEventMsg& msg){}

    /**
     *  Perform additional reset for strategy state
     */
    void OnResetStrategyState();

    /**
     * This event contains alerts about the status of a market data source
     */
    void OnDataSubscription(const DataSubscriptionEventMsg& msg){}

    /**
     * This event triggers whenever a custom strategy command is sent from
the client
     */
    void OnStrategyCommand(const StrategyCommandEventMsg& msg);

    /**
     * Notifies strategy for every succesfull change in the value of a
strategy parameter.
     *
     * Will be called any time a new parameter value passes validation,
including during strategy initialization when default parameter values
     * are set in the call to CreateParam and when any persisted values are
loaded. Will also trigger after OnResetStrategyState
     * to remind the strategy of the current parameter values.
     */
    void OnParamChanged(StrategyParam& param);

private: // Helper functions specific to this strategy
    void AdjustPortfolio(const Instrument* instrument, int desired_position);
```

```
    void SendOrder(const Instrument* instrument, int trade_size);
    void RepriceAll();
    void Reprice(Order* order);

    void SendBuyOrder(const Instrument* instrument, int unitsNeeded);
    void SendSellOrder(const Instrument* instrument, int unitsNeeded);

    void LogBarEvent(const SymbolTag symbol, const OFI ofi, const int
position, const int volume);

private: /* from Strategy */

    virtual void RegisterForStrategyEvents(StrategyEventRegister*
eventRegister, DateType currDate);

    /**
     * Define any params for use by the strategy
     */
    virtual void DefineStrategyParams();

    /**
     * Define any strategy commands for use by the strategy
     */
    virtual void DefineStrategyCommands();

private:
    boost::unordered_map<const Instrument*, OrderID>
m_instrument_order_id_map;
    double m_max_notional;
    double m_aggressiveness;
    Analytics::ScalarRollingWindow<double> m_rollingWindow;
    double m_rolling_window_double;
    int m_position_size;
    int m_time_before_close;
    bool m_nearing_close_time;
    bool m_debug_on;
    int m_debug_event_count;

};

extern "C" {

    _STRATEGY_EXPORTS const char* GetType()
    {
        return "PriceImpact";
    }

    _STRATEGY_EXPORTS IStrategy* CreateStrategy(const char* strategyType,
                                  unsigned strategyID,
                                  const char* strategyName,
                                  const char* groupName)
    {
        if (strcmp(strategyType,GetType()) == 0) {
            return *(new PriceImpact(strategyID, strategyName, groupName));
        } else {
            return NULL;
        }
```

```
    }

     // must match an existing user within the system
    _STRATEGY_EXPORTS const char* GetAuthor()
    {
        return "rjessen";
    }

    // must match an existing trading group within the system
    _STRATEGY_EXPORTS const char* GetAuthorGroup()
    {
        return "QTS";
    }

    // used to ensure the strategy was built against a version of the SDK
compatible with the server version
    _STRATEGY_EXPORTS const char* GetReleaseVersion()
    {
        return Strategy::release_version();
    }
}

#endif
```

Bibliography

Avellaneda, Marco, Josh Reed, and Sasha Stoikov. 2010. "Forecasting Prices in the Presence of Hidden Liquidity." *Preprint*. https://www.math.nyu.edu/faculty/avellane/hiddenliquidity3.pdf.

Benediktsdottir, Sigridur. 2006. "An Empirical Analysis of Specialist Trading Behavior at the New York Stock Exchange." http://www.federalreserve.gov/pubs/ifdp/2006/876/IFDP876.pdf.

Cont, R. 2011. "Statistical Modeling of High-Frequency Financial Data." *Signal Processing Magazine, IEEE* 28 (5): 16–25.

Cont, Rama, Arseniy Kukanov, and Sasha Stoikov. 2014. "The Price Impact of Order Book Events." *Journal of Financial Econometrics* 12 (1): 47–88.

Cont, Rama, Sasha Stoikov, and Rishi Talreja. 2010. "A Stochastic Model for Order Book Dynamics." *Operations Research* 58 (3): 549–63.

Lee, Charles M. C., and Mark J. Ready. 1991. "Inferring Trade Direction from Intraday Data." *The Journal of Finance* 46 (2): 733–46.

Lee, Yongjae, and Woo Chang Kim. 2013. "A Stochastic Model for Order Book Dynamics: An Application to Korean Stock Index Futures." *Management Science and Financial Engineering* 19 (1): 37–41.

Brokerage US Execution Rebate and Fee Schedule Exchange and ECNs, Effective May 1st, 2015, Version 150501.1 Published by Lime Brokerage

Philips, Matthew. 2013. "How the Robots Lost: High-Frequency Trading's Rise and Fall." *Bloomberg Businessweek*. http://www.bloomberg.com/bw/articles/2013-06-06/how-the-robots-lost-high-frequency-tradings-rise-and-fall.