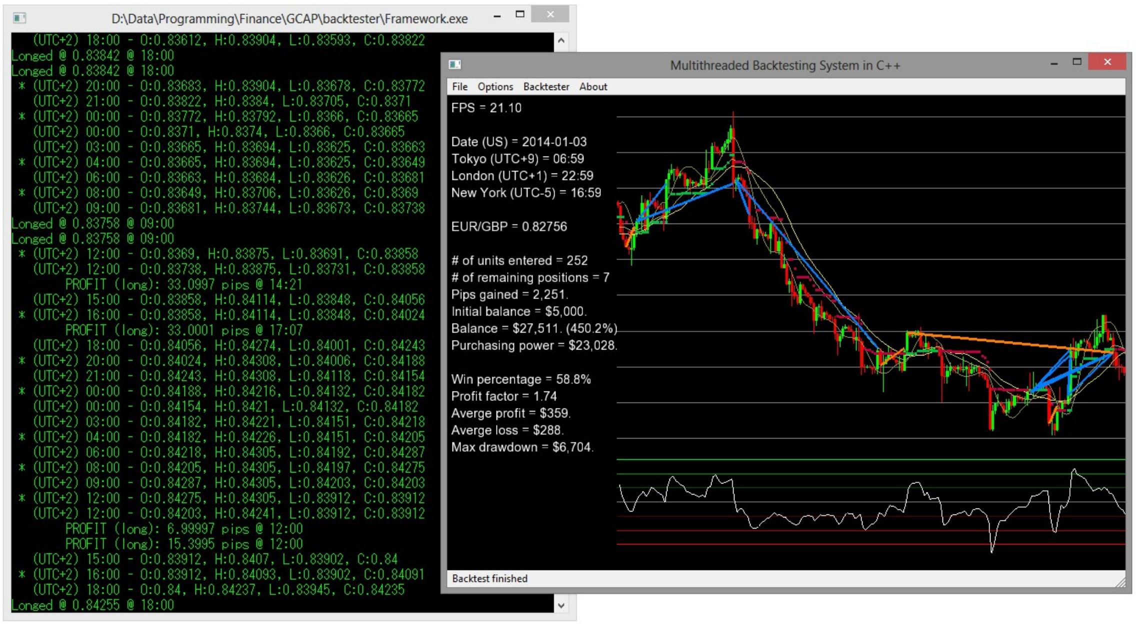
# A Multithreaded FX Algo Backtesting System

Yoshiharu Sato[†]
University of Warsaw / Warsaw School of Economics


D:\Data\Programming\Finance\GCAP\backtester\Framework.exe
Multithreaded Backtesting System in C++
File
Options
Backtester
About
Backtest finished

```
  (UTC+2) 18:00 - O:0.83612, H:0.83904, L:0.83593, C:0.83822
Longed @ 0.83842 @ 18:00
Longed @ 0.83842 @ 18:00
 * (UTC+2) 20:00 - O:0.83683, H:0.83904, L:0.83678, C:0.83772
   (UTC+2) 21:00 - O:0.83822, H:0.8384, L:0.83705, C:0.8371
 * (UTC+2) 00:00 - O:0.83772, H:0.83792, L:0.8366, C:0.83665
   (UTC+2) 00:00 - O:0.8371, H:0.8374, L:0.8366, C:0.83665
   (UTC+2) 03:00 - O:0.83665, H:0.83694, L:0.83625, C:0.83663
 * (UTC+2) 04:00 - O:0.83665, H:0.83694, L:0.83625, C:0.83649
   (UTC+2) 06:00 - O:0.83663, H:0.83684, L:0.83626, C:0.83681
 * (UTC+2) 08:00 - O:0.83649, H:0.83706, L:0.83626, C:0.8369
   (UTC+2) 09:00 - O:0.83681, H:0.83744, L:0.83673, C:0.83738
Longed @ 0.83758 @ 09:00
Longed @ 0.83758 @ 09:00
 * (UTC+2) 12:00 - O:0.8369, H:0.83875, L:0.83691, C:0.83858
   (UTC+2) 12:00 - O:0.83738, H:0.83875, L:0.83731, C:0.83858
        PROFIT (long): 33.0997 pips @ 14:21
   (UTC+2) 15:00 - O:0.83858, H:0.84114, L:0.83848, C:0.84056
 * (UTC+2) 16:00 - O:0.83858, H:0.84114, L:0.83848, C:0.84024
        PROFIT (long): 33.0001 pips @ 17:07
   (UTC+2) 18:00 - O:0.84056, H:0.84274, L:0.84001, C:0.84243
 * (UTC+2) 20:00 - O:0.84024, H:0.84308, L:0.84006, C:0.84188
   (UTC+2) 21:00 - O:0.84243, H:0.84308, L:0.84118, C:0.84154
 * (UTC+2) 00:00 - O:0.84188, H:0.84216, L:0.84132, C:0.84182
   (UTC+2) 00:00 - O:0.84154, H:0.8421, L:0.84132, C:0.84182
   (UTC+2) 03:00 - O:0.84182, H:0.84221, L:0.84151, C:0.84218
 * (UTC+2) 04:00 - O:0.84182, H:0.84226, L:0.84151, C:0.84205
   (UTC+2) 06:00 - O:0.84218, H:0.84305, L:0.84192, C:0.84287
 * (UTC+2) 08:00 - O:0.84205, H:0.84305, L:0.84197, C:0.84275
   (UTC+2) 09:00 - O:0.84287, H:0.84305, L:0.84203, C:0.84203
 * (UTC+2) 12:00 - O:0.84275, H:0.84305, L:0.83912, C:0.83912
   (UTC+2) 12:00 - O:0.84203, H:0.84241, L:0.83912, C:0.83912
        PROFIT (long): 6.99997 pips @ 12:00
        PROFIT (long): 15.3995 pips @ 12:00
   (UTC+2) 15:00 - O:0.83912, H:0.8407, L:0.83902, C:0.84
 * (UTC+2) 16:00 - O:0.83912, H:0.84093, L:0.83902, C:0.84091
   (UTC+2) 18:00 - O:0.84, H:0.84237, L:0.83945, C:0.84235
Longed @ 0.84255 @ 18:00
```

```
FPS = 21.10

Date (US) = 2014-01-03
Tokyo (UTC+9) = 06:59
London (UTC+1) = 22:59
New York (UTC-5) = 16:59

EUR/GBP = 0.82756

# of units entered = 252
# of remaining positions = 7
Pips gained = 2,251.
Initial balance = $5,000.
Balance = $27,511. (450.2%)
Purchasing power = $23,028.

Win percentage = 58.8%
Profit factor = 1.74
Averge profit = $359.
Averge loss = $288.
Max drawdown = $6,704.
```

***Figure 1.*** *Our backtesting system in action*

## Abstract

Over the past decade, algorithmic trading has been increasingly growing its trade volume shares within the foreign exchange market. Thanks to the introduction and subsequent popularization of the MetaTrader 4 electronic trading platform, even individual retail traders are now able to use their own algorithms to trade currencies in the market. In order to devise a profitable FX trading algorithm, it is vital for developers to backtest their algorithms using different sets of parameters, different time frames, different time periods, and different currency pairs. Backtesting therefore is inherently a time-consuming process, thus it is critical for developers to reduce the amount of time taken up by backtesting so as to increase productivity. For this reason, we designed and built a backtesting system for FX trading algorithms completely from scratch in order to maximize the speed, flexibility, and accuracy of backtesting. Using our framework, developers are able to implement their own FX trading algorithms in C++, backtest them in an efficient and accurate manner, and consequently assess the historical profitability and risk of their algorithms.

**Keywords:** Algorithmic Trading, Backtesting, Foreign Exchange, Quantitative Finance, Monte Carlo Simulation, Concurrent Programming, Asynchronous Parallelism

[†]E-mail: yoshi2233@gmail.com

## 1. Introduction

Advancements in information and communications technology (ICT) as well as the full computerization of financial markets in the late 1980s and 1990s gave birth to algorithmic trading – the application of electronic platforms for entering trading orders using computer algorithms which automatically execute pre-programmed trading strategies without human intervention. Today, algorithmic trading is extensively employed by various types of financial institutions (both sell-side and buy-side firms), the most notable ones of which are investment banks and hedge funds.

Since the introduction of algorithmic trading, public attention has been primarily focused on the rate of algorithm adoption in the stock market – computer algorithms today account for more than 50% of all US stock trade volume (Figure 2). One of the typical ways in which buy-side firms use algorithms in the stock market is drip-feeding a large order by dividing it into a number of small orders based on volume- or time-weighted average price (VWAP and TWAP respectively) so as to minimize the adverse effect of market impact. Sell-side firms usually use algorithms to provide liquidity to the market by generating and executing orders automatically. Many stock exchanges around the world nowadays accept trading orders generated by algorithms via direct market access (DMA), through which firms are able to place orders directly onto the order book of an exchange from their algorithmic trading systems without recourse to market makers. [1] A number of stock exchanges also provide co-location facilities wherein firms are able to locate their systems within the exchange's premises, thereby ultimately minimizing latency (i.e., high-frequency trading; HFT).

In the past years, the emergence of currencies as a legitimate asset class has resulted in rapid adoption of algorithmic trading in the FX market. Figure 2 shows the trade volume shares of algorithmic trading in the stock, futures, FX, and bond markets in the US, where a clear upward trend of algorithm adoption in the FX market can be observed. According to ICAP, FX algorihtmic trading at their EBS inter-dealer electronic trading platform grew from 28% to 68% over the period 2007–2013, and about 30–35% of total spot trade volume on the platform is HFT-driven. [3]

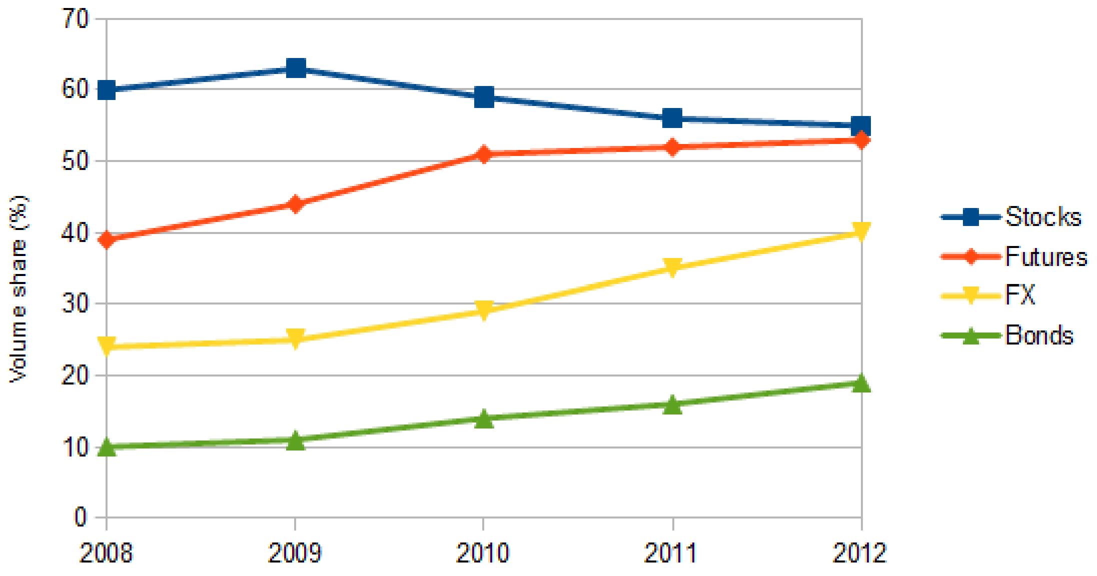


***Figure 2.*** *Trade volume shares of algorithmic trading in US financial markets (FSOC: Financial Stability Oversight Council [2])*

One of the advantages of algorithmic trading in the FX market is that traders themselves do not have to always keep an eye on the market to catch a signal for new entering/exiting orders. In fact, the FX market is open 24 hours a day except weekends (i.e., from Sunday 20:15 GMT until Friday 22:00 GMT), hence only computers can possibly monitor the market throughout the entire period, unless there are multiple traders working collaboratively in different time periods. Being able to catch more signals in the market gives traders more opportunities for making profits, as well as smaller risks to expand losses.

Since the introduction of MetaTrader 4 (MT4) [4] electronic trading platform in 2005, algorithmic trading in the retail FX market has been growing exponetially as MT4 allows users to trade currencies using custom trading algorithms. However, MT4 is not necessarily the best platform to develop trading algorithms for, particularly because the platform's backtester has several serious limitations. It is critically important for developers to use a high-performance, high-fidelity backtester in order to devise truly profitable trading algorithms, since developers are required to backtest their algorithms using different sets of parameters, different time frames, different time periods, and different currency pairs, throughout the development process.

Consequently, as the goal of this research, we designed and built a backtesting system of FX trading algorithms completely from scratch so as to maximize the speed, flexibility, and accuracy of backtesting. Using our framework, developers are able to implement their own FX trading algorithms in the C++ programming language [7], backtest them efficiently and accurately under various kinds of conditions, and assess the historical profitability as well as the risk of their trading algorithms.

# 2. Foreign Exchange Market

## 2-1. Overview

Traders who participate in the FX market usually buy (long) and/or sell (short) currency pairs, such as EURUSD (i.e., the exchange rate of Euro denominated in the US dollar), in which example EUR is called the quote currency and USD the base currency. Theoretically, there could be *C(n, 2)* currency pairs in the market (where *n* is the number of all currencies in the world), however the number of currency pairs that are actually tradable in the market is much smaller since some currencies are either restricted for exchange or too minor and thus there is not much demand for trades.

What should be noted in regards to currency pairs in the FX market is the fact that, even if a pair does not include USD, its exchange transaction always involves buying and selling of USD internally because the US dollar is the world's reserve currency. For instance, when a trader buys EURGBP, in actuality he first buys US dollars with British pounds and subsequently sells the US dollars in exchange for Euros. The currency rate in this case is therefore calculated as: $EURGBP = USDGBP \times EURUSD$.

In the FX market, the amount of profits or losses is often denoted in a unit of change in a currency pair's exchange rate known as “pip” (percentage in point; i.e., 1/100th of a percentage). For major currency pairs except Japanese yen crosses, a pip is one unit of the fourth decimal point. A currency pair is typically traded in lot size of 100,000 units of the base currency (a.k.a. “standard lot”). A trading position of 1 standard lot that experiences a change in the exchange rate of 1 pip thus changes in value by 10 units of the quoted currency.

Prior to the development of retail FX electronic trading platforms in the early 2000s, the FX market was restricted to large financial institutions. It was the commoditization of personal computers, the rapid expansion of the internet around the globe, the advent of electronic trading platforms, and the inflows of many retail FX brokers that started the exponential growth of retail FX trading. Individual traders are today not only able to trade currencies on margin (i.e., they need to put down only a small percentage of their trade size) but also able to trade based on computer algorithms using technical analysis software that has a broker interface and an algorithm execution engine. Notable examples of such software include MetaTrader 4 (MT4), NinjaTrader [5], and TradeStation [6].

## 2-2. Market Structure

The significant difference between the FX market and the stock market is the fact that the former is decentralized, meaning there is no centralized location for order execution. This is attributed to the very nature of currencies (i.e., they are independently issued by central banks around the world). As a result, the FX market is primarily an over-the-counter (OTC) market. Another important aspect of the FX market is that it is cascaded, or divided into levels of access. At the top is the interbank market, which is mostly consisted of large investment banks (e.g., Deutche Bank, Citibank, UBS; a.k.a., "prime banks"), who act as broker-dealers and involve in the determination of actual exchange rates. From the interbank market, exchange rates are disseminated to the lower levels with wider spreads (i.e., the difference between the bid and ask prices, which is the primary source of a broker's profits).

Despite the fact that individual retail traders constitute a growing segment of the FX market, they are usually at the lowest level of access in the market. This is attributable to the size of their trade volume – if a trader can guarantee large numbers of transactions for large units, they can demand a tighter spread. Most individual traders are therefore price takers and participate in the FX market indirectly through retail brokers or banks:

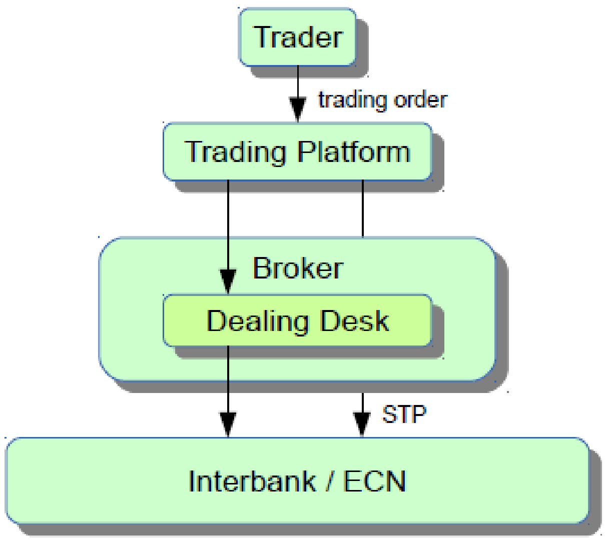


***Figure 3.*** *Flow diagram of trading orders in the retail FX market*

Figure 3 illustrates how a trading order made by a trader is executed by a retail FX broker. First, the trading platform on which the trader made the order sends the order information (i.e., currency pair, lot size, long/short, order type, etc) to the broker's server over the network (typically via the TCP/IP protocol [8][9]). If the broker routes its clients' orders through its dealing desk system and trade against them, it is called a "DD broker" or a market maker, for it literally makes the market for traders – the broker buys from its clients when they want to sell, and sells to them when they want to buy. Losing trades of clients are counter-traded within the dealing desk's system and become the broker's profit. Winning trades of clients, on the other hand, are usually hedged in the interbank market to offset the broker's risks.

If the broker processes its clients' orders without passing through its dealing desk system, it is called an “NDD broker.” This type of broker offers clients direct access to the interbank market or to an electronic communications network (ECN) along with a straight-through processing (STP) bridge [10], providing clients with a transparent and low-latency trading environment. However, some NDD brokers may still have what is known as the “last look” provision, under which they inspect the contents of clients' orders before they send them to the market. By having this functionality, brokers are able to negotiate more favorable terms from their liquidity providers because the likelihood of liquidity providers making losses is significantly reduced.

## 3. FX Algorithmic Trading Systems

### 3-1. Algorithm Development

Developers who try to build their own algoritmic trading systems for the retail FX market are required to create the following software components: (1) a backtester module that allows them to backtest trading algorithms in numerous historical market conditions (i.e., different currency pairs, different time frames, and dfferent time periods); (2) a 2D graphics renderer module that displays price charts of currency pairs along with various types of technical analysis indicators (e.g., moving avergaes, oscillators) on the computer screen; (3) a network module that receives live exchange data feeds from a broker and sends trading orders to the server using the broker's API; and last but not least (4) a trading algorithm thoroughly tested in a scientific manner and most likely to generate profits in live trading.

Developing a truly profitable trading algorithm is highly sophisticated art and science, and requires knowledge and skills not only in computer programming and trading but also in various quantitative analytical methods (e.g., mathematics, statistics, econometrics).

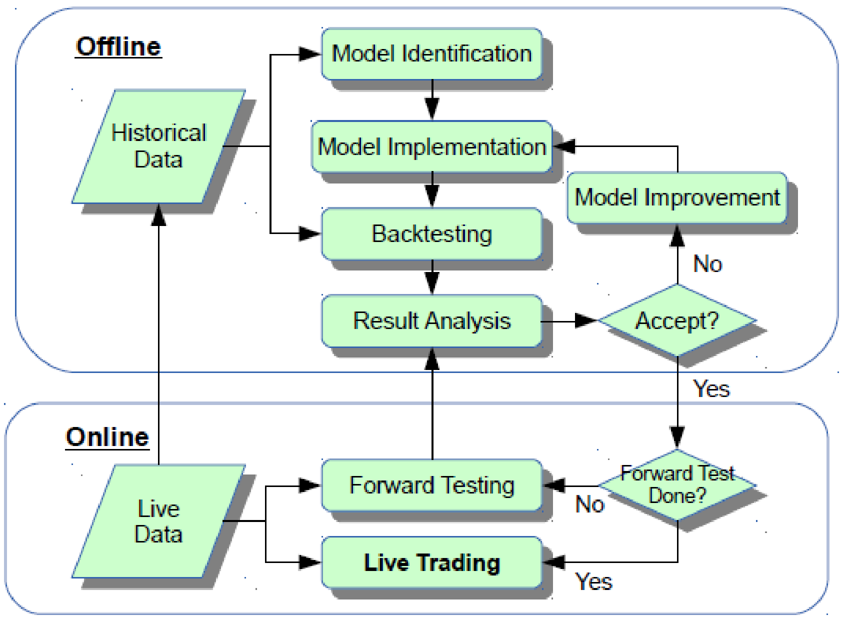


***Figure 4.*** *Flow chart of trading algorithm development*

Consequently, it requires a substantial amount of study and work for developers to build an algorithmic trading system from scratch. For this reason, although there exist some open-source libraries for algorithmic trading, many developers choose to focus solely on creating their own trading algorithms using existing algorithmic trading systems that are available in the marketplace, many of which systems are offered at no extra cost for those who open an account with a broker. The MetaTrader 4 (MT4) is by far the most popular algorithmic trading system among FX trading algorithm developers. We will discuss the MT4 platform in detail in Chapter 3-4.

Figure 4 illustrates how an FX trading algorithm is typically developed (this flow chart is universally applicable to the algorithm development of other types of asset classes). There are broadly two groups of stages in the development process: offline stages (backtesting process) and online stages (forward testing process). The explanation of each stage is given below:

**Model Identificatin**

The very first thing that trading algorithm developers need to do is to analyze historical market data, make a hypothesis, and build a model. Although this process often involves quantitative analysis of historical data, developers are not necessarily required to come up with a quantitative model for time-series forecasting. What they are required to do is to devise a trading strategy whose rules of market entry and exit are clearly defined and detailed enough to actually be coded as an algorithm. Trading strategies are in many cases devised using indicators used for technical analysis of price charts.

**Model Implementation**

The second stage is to implement the model from the first stage as an algorithm which computers are able to execute. This means for developers to write program code of the trading algorithm in a programming language or a scripting language employed by their algorithmic trading system. Most proprietary scripting languages, such as the MetaQuotes Language 4 (MQL4) [11] employed by MT4, come with built-in functions and other features that are useful for algorithmic trading, but they have limitations in many ways, particularly in execution efficiency and flexibility. Many developers thus still prefer to implement trading algorithms in general-purpose programming languages (e.g., C++) if their algorithmic trading system allows the integration thereof.

**Backtesting**

The third stage is to backtest the trading algorithm using historical market data. This stage is especially important as it is indispensable for developers to backtest their trading algorithm so as not only to estimate the profitability and risk thereof but also to validate the algorithm implementation. Unexpected behaviors and operations of the algorithm during backtests must be scrutinized, since it is highly likely that the algorithm is not implemented properly in such cases. We will disscuss backtesting in detail later in Chapter 3-2.

**Result Analysis**

The fourth stage is to analyze the results obtained from backtests of the trading algorithm and to assess the reliability and suitability thereof. To this end, various types of statistics and measures are calculated and recorded in backtesting. Developers can subsequently estimate the profitability as well as the risk of their trading algorithm based on those measures (cf. Chapter 3-3).

**Forward Testing**

If the backtest results are acceptable, developers should proceed to forward testing, in which they execute their trading algorithm by feeding live market data in real time so as to see if it still generates profits as it did in backtesting. Naturally, forward testing takes much longer than backtesting. Some developers therefore prefer to substitute forward tests with out-of-sample backtests (cf. Chapter 3-2).

Developers should finally use their trading algorithm in live trading if and only if it produced an acceptable result in forward testing. Furthermore, it is recommended to start running the algorithm in live trading only with a limited amount of money in order to minimize the initial risk associated with the unpredictable nature of live trading. This is also because of the fact that, if the algorithm is truly profitable in a constant and stable manner, developers are able to increase their money exponentially with compound interests. Developers hence do not have to take an excessive risk from the very beginning of their live trading.

## 3-2. Backtesting

Backtesting (a.k.a. historical testing or historical simulation) is an indispensable process for developers to analyze, verify, and improve their trading algorithms. Backtesting provides developers with a simulation result of the performance that is likely to have been achieved had the algorithm been actually used in live trading on prior time periods. In other words, developers are able to estimate the profitability and risk of their trading algorithms by doing a simulation against historical market data, and thus they are able to determine the historical performance of their algorithm and reject models that are potentially unprofitable or even harmful in future time periods. Backtesting is useful since it provides information that cannot be attained when trading algorithms are tested against synthetic market data (e.g., Geometric Brownian motion). It also provides developers with a sense of security, which is a significant psychological factor when they are running their algorithms in live trading and risking real money.

In backtesting, the initial part of historical data on which a trading algorithm is tested and optimized is specifically referred to as the "in-sample (IS) data," while it is conventional among developers to reserve a portion of the historical data as the "out-of-sample (OOS) data" for validating the algorithm performance obtained from the previous IS data (Figure 5). By having this setup of historical data, developers can substitute forward testing with OOS backtesting because the trading algorithm has never been tested against the OOS data, yet the OOS data is composed of real market data, thereby effectively acting as forward testing. This allows developers to avoid spending a long time on forward testing (which usually takes weeks or months). In order for OOS backtesting to be valid and meaningful, it is critically important for developers to strictly use only the IS data for optimizing their trading algorithm. Otherwise the OOS data becomes in-sample, and thus OOS backtesting loses its validity.

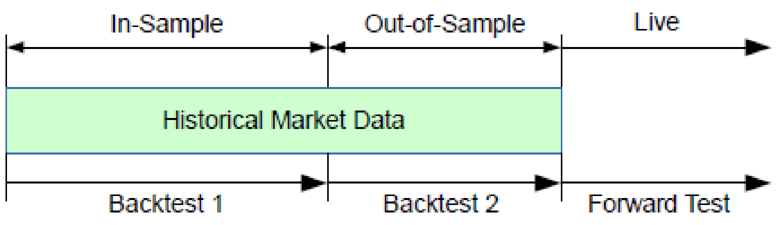


***Figure 5.*** *Division into in-sample (IS) data and out-of-sample (OOS) data*

What trading algorithm developers need to understand fundamentally is that backtesting does not guarantee by any means that historically profitable algorithms will maintain their profitability in the future. In other words, positive backtested results do not provide sufficient information in helping developers know how their trading algorithms will perform when traded live. We discuss major limitations to backtesting below:

- It is not necessarily easy for developers to obtain high quality historical market data with which they can sufficiently reconstruct past market conditions (this is one of the primary reasons why backtesting had historically been performed mostly by large financial institutions).
- Due to the static nature of historical data, in backtesting it is very difficult to take into account the market impact of a trading algorithm on price and liquidity. While in many cases this is not an issue for most retail traders with small trade volumes, it is not negligible for large players such as hedge funds.
- It is also difficult in backtesting to fully take into account implementation shortfall or slippage, which is the difference between the initial price at which a trading order was entered and the final price at which the order was actually executed (including broker fees, taxes, etc). Implementation shortfall is an execution cost, thus it can be problematic when the algorithm is high-frequency.
- Backtesting all too often results in overfitting. It is not technically difficult for developers to make a trading algorithm perform well in a backtest by overly optimizing it to a specific historical dataset. While the trading algorithm would have worked well in the time period in which it was optimized, the result achieved is highly dependent on the price movements during that period, hence there is no guarantee that it will work well in the future.

The long, iterative selection process of analyzing historical market data and optimizing a trading algorithm through a series of backtests is data mining, thus trading algorithm developers should be prudent about the data mining bias [12]. Data mining bias in algorithmic trading arises when the data miner mistakenly uses the best backtested result to estimate the algorithm's expected performance in the future. This is attributable to the fact that the observed performance of the algorithm, among a large set of parameter combinations, is a positively biased estimate of the algorithm's expected performance, since it incorporates a substantial degree of good luck (naturally, the probability of a good result caused by chance increases with the number of parameter combinations tested).

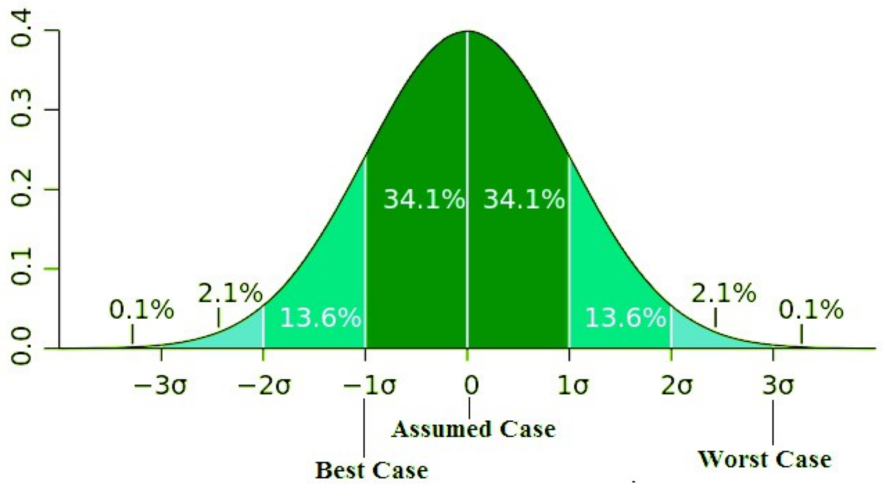


***Figure 6.*** *Imaginary distribution of all possible trading results of an algorithm*

To better understand data mining bias, let us assume the ultimate population of a trading algorithm's lifetime results, i.e., a statistical population consisting of the algorithm's rates of return from every possible parameter combination in every possible constant-length time period in the past and the future. Figure 6 illustrates the distribution of this imaginary population. Here, a normal distribution is assumed based on the Central limit theorem (CLT), in which normal distribution 99.7% of the values lie within three standard deviations of the mean. In reality, the population is unlikely to be normally distributed, but still it is guaranteed by Chebyshev's inequality that 89% of the values lie within three standard deviations. [13]

The question that developers want to ask themselves then is, where does the backtested result (sample) fall in this distribution? In the best case the sample is below average, meaning the algorithm has a higher potential profitability. Typically, developers think that the sample is about the mean and thus the algorithm could do either better or worse equally. However, it is almost assured that the sample is near/at the worst case if the backtested result was obtained through intensive optimization (i.e., overfitting). In that case, the live-trading performance of the algorithm will most likely be worse than the backtested result.

Aronson [14] lists five major factors that contribute to the data mining bias:

1. The more trading rules and parameter combinations backtested, the larger the data mining bias (because we have more opportunities to get lucky).
2. The smaller the number of trades the algorithm made in backtesting, the larger the data mining bias (due to the small sample size).
3. The lower the correlation among trading rules used for the algorithm, the larger the data mining bias (100% correlation means the trading rules have the same behavior, we are therefore testing only one rule in effect).
4. The larger the effect of positive outliers on the backtested performance, the larger the data mining bias (e.g., returns generated from black swan events).
5. The lower the variation in expected returns among the trading rules, the larger the data mining bias.

## 3-3. Metrics for Algorithm Performance and Risk

It is indispensable for developers to collect and analyze objective information regarding the performance of their trading algorithm in order to assess its profitability and risk. There are various types of performance metrics for trading algorithms, the most typical ones of which include: rate of return, rate of winning, average profit, average loss, profit factor, payoff ratio, expected payoff, Sharpe ratio [15], and maximum drawdown.

The *average profit* per winning trade is equal to the sum of all profits (i.e., *gross profit*) divided by the number of winning trades. It is useful for getting an idea of how much we could expect to gain from a winning trade.

$$\overline{W} = \frac{1}{N_W} \sum_{i=1}^{N_W} W_i \tag{1}$$

Similarly, the *average loss* per losing trade is equal to the sum of all losses (i.e., *gross loss*) divided by the number of losing trades. It is useful for getting an idea of how much we could expect to lose from a losing trade.

$$\overline{L} = \frac{1}{N_L} \sum_{i=1}^{N_L} L_i \tag{2}$$

The *profit factor* is defined as the ratio of the gross profit to the gross loss:

$$P_f = \frac{\sum_{i=1}^{N_W} W_i}{\sum_{j=1}^{N_L} L_j} = \frac{N_W}{N_L} \frac{\overline{W}}{\overline{L}} \tag{3}$$

If the profit factor is greater than 1, it indicates that the backtested algorithm has more wins than losses in the historical market data, hence it would have been profitable in the tested time period, otherwise it would have incurred losses.

Profit factor is a simple yet useful measure of how efficient the profitability of a trading algorithm is. For example, let us assume that we have an algorithm denoted by A which has made \$10,000 of gross profit and \$5,000 of gross loss (therefore has a profit factor of 2), and another algorithm B which has made \$5,000 of gross profit and \$2,000 of gross loss (i.e., profit factor of 2.5). While the algorithm A has a net profit of \$5,000 and the algorithm B has a net profit of \$3,000, B is actually superior in terms of profitability because if we had doubled the leverage ratio for our margin account we would have made \$10,000 of gross profit and \$4,000 of gross loss using B, therefore obtaining a net profit of \$6,000.

It is always crucial in backtesting for developers to make their algorithm achieve a profit factor at the very least greater than 1, since any algorithm that is profitable in live trading is also profitable in backtesting (i.e., backtest profitability is a necessary condition for live-trading profitability). Having said that, higher values of profit factor do not necessarily guarantee the algorithm's prospect in live trading as it may have been overfitted to the historical data used.

The ratio of the average profit (1) to the average loss (2) is referred to as the *payoff ratio*:

$$P_r = \frac{\overline{W}}{\overline{L}} \tag{4}$$

This alone indicates how many losses on average can be compensated with one winning trade. For instance, a trading algorithm that has a payoff ratio of 2 can averagely compensate for 2 losses with 1 winning trade.

The payoff ratio is not a direct indication of the algorithm's profitability; instead, it is highly related to the *rate of winning* (a.k.a. *winning percentage*):

$$R_W = \frac{N_W}{N} \tag{5}$$

where $N$ is the total number of trades that the algorithm has made during the backtested time period. The profit factor (3) can subsequently be derived from the payoff ratio (4) and the winning percentage (5):

$$P_f = \frac{N_W}{N_L} \frac{\overline{W}}{\overline{L}} = \frac{N_w}{N - N_W} P_r = \frac{R_W}{1 - R_W} P_r \tag{6}$$

If we solve the equation above for *Rw*, we get:

$$R_W = \frac{1}{1+\frac{P_r}{P_f}} = \frac{P_f}{P_f + P_r} \tag{7}$$

The equation (7) is known as the Profitability rule [16].

Even if the payoff ratio is below 1, the algorithm can be profitable as long as the winning percentage is sufficiently high so that the winning trades compensate for the losing trades. In the same manner, even if the winning percentage is low, the algorithm can be profitable if the payoff ratio is sufficiently high. Table 1 shows this relationship between payoff ratio and winning percentage. The required winning percentages are computed using the Profitability Rule with a fixed profit factor equal to 2 as an example.

| **Payoff ratio** | **Required $R_W$ (%)** |
|---|---|
| 10 | 16.67 |
| 5 | 28.57 |
| 2 | 50.00 |
| 1 | 66.67 |
| 0.5 | 80.00 |
| 0.2 | 90.91 |
| 0.1 | 95.24 |

***Table 1.*** *Required Rw as a function of payoff ratio (profit factor = 2)*

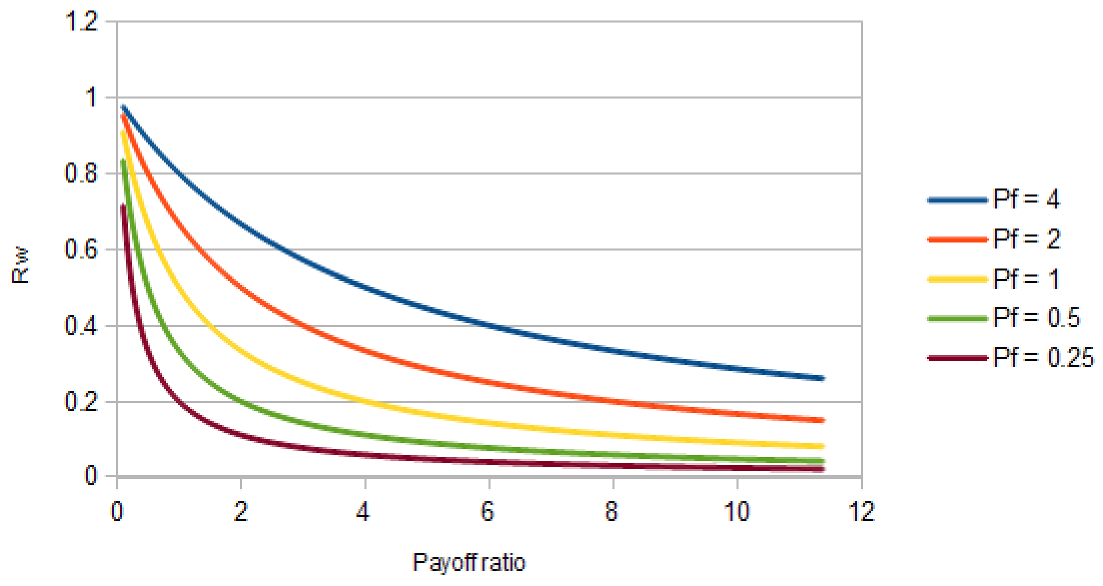


***Figure 7.*** *Required Rw as a function of payoff ratio (with various profit factors)*

The *expected payoff* is the amount of payoff that we can expect from a single trade. It simply boils down to the *average net profit* per trade:

$$E(payoff) = R_W \overline{W} - (1 - R_W)\overline{L} = \frac{net\ profit}{N} \tag{8}$$

Now, if we denote by $R_T$ the cumulative daily return of a trading algorithm up to period $T$, then the *Sharpe ratio* is given by:

$$S_r = \frac{mean(R_T) - R_f}{std(R_T)} \tag{9}$$

where $mean(R_T)$ and $std(R_T)$ are respectively the average and the standard deviation of the cumulative daily return. $R_f$ is a risk-free rate, which is usually assumed to be zero in FX algorithmic trading. Here, $mean(R_T)$ represents the expected return and $std(R_T)$ the risk. Concordantly, the Sharpe ratio represents the risk premium per unit of risk of the trading algorithm. It helps us determine if an increase in risk is appropriate for a higher return (high returns may often come at a cost of excess risk). A Sharpe ratio of 1.0 or greater is considered sufficient, 2.0 or greater is ideal, and 3.0 or greater is excellent.

The *maximum drawdown* is the largest percentage change between a local maximum (peak) and the subsequent local minimum (valley) on a historical equity curve of a trading algorithm (Figure 8), and it represents the relative downside risk thereof. Since it is calculated as a percentage change, the drawdown which has the maximum difference between the peak and the valley is not always the maximum drawdown.

An ideal trading algorithm would have no drawdowns (i.e., its equity curve is monotonically increasing; thus completely "painless"), but in reality most algorithms incurr drawdowns during backtesting. For mitigating downside risk in terms of capital preservation, it is important for developers to minimize both the values and the frequency of their algorithm's drawdowns.

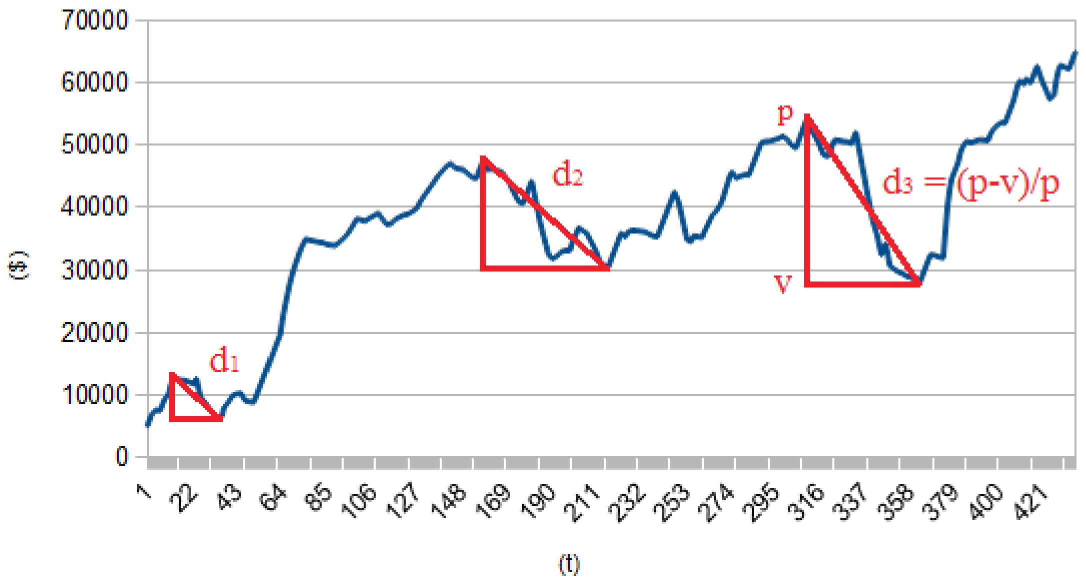


***Figure 8.*** *Drawdowns on an equity curve and the maximum drawdown ($d_3$)*

Recall now that an equity curve is essentially a cumulative sum of trades performed by a trading algorithm in a chronological order. What this suggests is that if we reshuffle the order of their occurrence, we can construct a new equity curve with exactly the same starting and ending equities as the original's, since no trade is added nor removed.

This particular permutation method is called "selection without replacement," and we can use this method to execute Monte Carlo simulations for downside risk analysis. [17] The reshuffling of the trade orders results in the effect that there are huge variations among the new equity curves with different drawdowns occurring at different times (Figure 9).

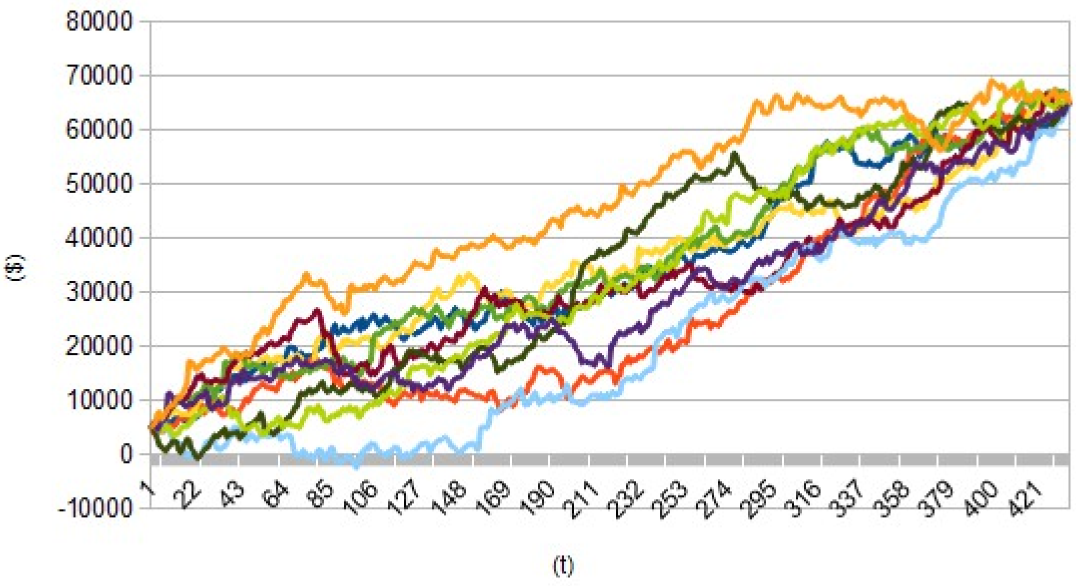


***Figure 9.*** *New equity curves generated by randomly reshuffling the trade orders*

We define the *expected maximum drawdown* as follows:

$$E(\boldsymbol{MDD}) = \exp\left[\frac{1}{N_R}\sum_{i=1}^{N_R}\ln(MDD_i)\right] = \left[\prod_{i=1}^{N_R} MDD_i\right]^{\frac{1}{N_R}} \tag{10}$$

where $N_R$ is the number of reshuffled equity curves. It is simply the geometric mean of the maximum drawdowns from the reshuffled curves.

Last but not least, we define what we call the *blow-up probability* as below:

$$P(Y<Z) \sim \frac{1}{N_R}\sum_{i=1}^{N_R} IF[min(\boldsymbol{X}_i)<Z, 1, 0] \tag{11}$$

where $Y$ is the future equity, $Z$ is some lower-bound threshold at which the algorithm should stop trading, $\boldsymbol{X}_i$ is the i-th reshuffled equity curve (a sequence with length $N$), and $IF$ is an integer predicate. The value $P(Y<Z)$ therefore represents the Monte-Carlo-estimated probability of the future equity falling below the threshold level.

The advantage of using the selection-without-replacement method for Monte Carlo simulations is that it exactly duplicates the probability distribution of the input sequence (i.e., original equity curve). The downside, on the other hand, is that the randomly sampled trade sequences are limited to the number of trades $N$, meaning in order for the Monte Carlo simulations to be accurate $N$ must be sufficiently large.

## 3-4. MetaTrader 4

The MetaTrader 4 (MT4) is an electronic trading platform specifically designed for retail FX margin trading. Since its release in 2005, the platform has been licensed to a large number of retail FX brokers worldwide due to its popularity among individual traders. MT4 has become the de facto trading platform for the retail FX market, mainly because it allows users to write their own scripts for custom indicators, trading algorithms (which are referred to as "Expert Advisors" or "EAs" on the platform) and other types of add-ins.

MT4's proprietary scripting language for EAs is called MQL4, which is similar to the C programming language. It comes with various built-in functions that are useful for creating custom indicators and algorithms. MT4 also supports custom DLLs (dynamic-link libraries), allowing developers to write their own trading algorithms in C++ or other languages that support DLL compilation, thereby providing developers with more flexibilities and control.

Although MT4 has a built-in trading algorithm backtesting system (which is referred to as "Strategy Tester" on the platform), it has some severe limitations and thus it has never been an ideal backtesting system for FX trading algorithm developers, at least in our opinion, because:

- Historical market data used by MT4 only contains bid prices. The platform therefore executes backtests only with a fixed spread (it uses the current live spread at the time a backtest is launched). Obviously such situation never exists in the real market, hence this is a serious issue since developers cannot take into account all the dynamic changes in spread that are inherent in the real market. Not only does it create overly optimistic results for spread-sensitive algorithms (e.g., scalpers) when the live spread is below average at the time the backtest is launched, but also it lacks reproducibility because test results can vary significantly depending on the spread size.
- Historical market data used by MT4 is not consisted of raw ticks but instead optimized for candlestick charts (the lowest timeframe the platform recognizes is one-minute). The platform generates tick data for backtests from its historical data by interpolating between the open, high, low and close prices. [18] Clearly, these ticks are synthetic, and thus this is an issue for developers who want to reconstruct the market as much as possible for spread-sensitive or high-frequency trading algorithms on a tick-by-tick basis.
- Backtesting on MT4 could take a long time especially when the trading algorithm is complex and entirely written in the MQL4 scripting language, because MQL4 is not native code (about 17 times slower than C++ [19]) and the platform does not support multithreading that utilizes multicore CPUs, which most of today's computers are equipped with.
- MT4 lacks transparency in what formulas are actually used to calculate its indicators since they are closed-source, hence developers cannot know how they are calculated internally. It is known that some indicators that are native to MT4 give different results from the industry-standard implementations, which poses an issue in reliability as well as reproducibility. It is still possible for developers to implement those indicators manually in MQL4 as a custom script, but obviously this could take a lot of time.

Our backtesting system, on the other hand, solves all the limitations of the MT4's backtesting system that have been mentioned above. Namely, (1) the system uses full tick-by-tick historical data with dynamically changing spreads, (2) it is multithreaded in

C++ native code so as to take the best advantage of modern multicore CPUs for maximum execution efficiency, and (3) it uses TA-Lib [20] (an open-source technical analysis library) to increase transparency in indicator implementation.

## 4. System Design & Architecture

### 4-1. C++ Programming Language

Execution efficiency is the most critical aspect of a backtesting system inasmuch as algorithm backtesting is a time-consuming process. Minimizing the execution time taken up by backtesting therefore results in a significant increase in the productivity of trading algorithm developers, since it is indispensable for them to backtest their algorithms repeatedly throughout their development process in order to make their algorithms truly profitable.

For this reason, we (quite naturally) selected the C++ programming language to develop our backtsting system, because C++'s execution efficiency is unmatched with other programming languages due to its native code generation (Figure 10). Nowadays, most performance-critical software is written in C++ (e.g., operating systems, 3D games, and other real-time systems). In fact, many financial institutions use C++ to build their proprietary algorithmic trading systems.

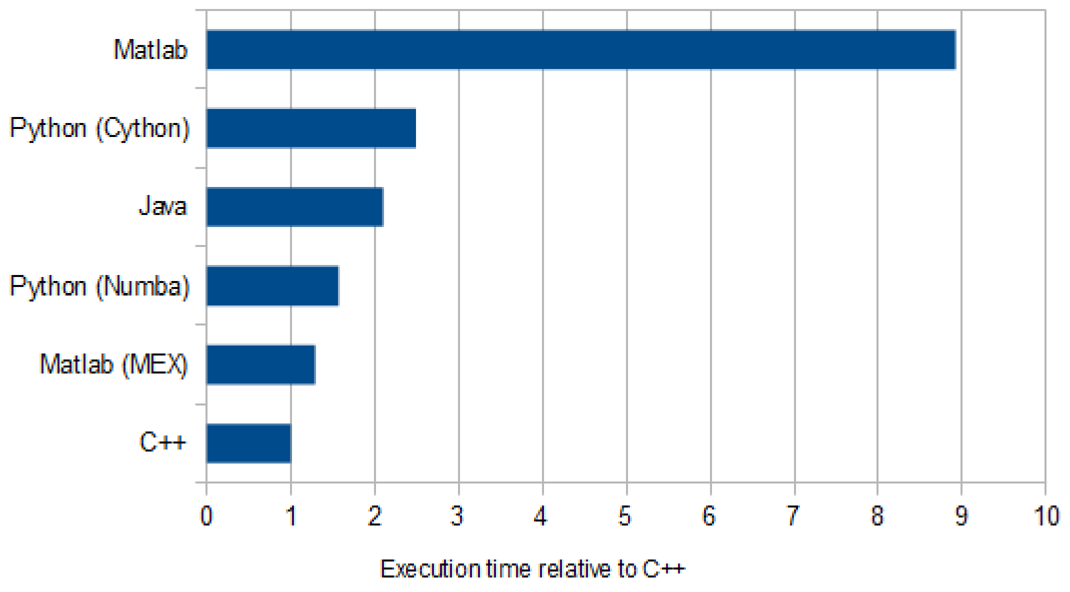


***Figure 10.*** *Comparison of execution time in programming languages [21]*

C++ is sufficiently high-level to write complex, large-scale programs, while simultaneously it is sufficiently low-level to write efficient code (e.g., manual memory management, bitwise operations, SIMD intrinsic functions, or even inline assembly). Consequently, it is one of the most flexible, versatile, and efficient programming languages, and is also highly efficient on hardware with limited computing resources (e.g., embedded systems, mobile devices) because efficiency is not just running faster or running bigger programs but also running with less resources – these are the two sides of the same coin.

Like every programming language, however, C++ has its own disadvantages, which include: (1) C++ does not offer developers a higher code productivity because it is an extremely complex language and thus learning and fully understanding the language require a tremendous effort; (2) C++ by default does not have any automatic memory management mechanisms such as garbage collector, hence programmers must be prudent about memory leaks as well as memory protection errors; (3) the C++ standard library and the standard template library (STL) is somewhat limited compared to other languages; in many cases programmers therefore need to adapt an external library such as Boost [22] for advanced functionalities; (4) C++ is generally not suited for developing safety-critical software because of its language complexity and compiler-dependent nature (that said, it is possible to build safety-critical systems in C++ securely by imposing strict coding standards [23]); and finally (5) C++ code generally takes a lot of time to compile, and every time a change is made in the code, it needs to be re-compiled since new native code must be generated accordingly.

With all of that being said, the new C++ standard introduced in 2011 (a.k.a. C++11 [24]) has greatly enhanced the language with a lot of new features and new standardized libraries – most notably smart pointers and multithreading facilities. We, however, did not use C++11 for our backtesting system mainly because of its immature compiler support at the time of the development of our system. We instead used the C++03 [7] along with STL and Boost.

We also placed great importance on portability so as to make our system executable on multiple platforms (i.e., Windows and Linux). To this end, we wrote our C++ code in accordance with the language standards as much as possible and also used crossplatform libraries; namely Qt [25] for the GUI, OpenGL [26] for the 2D graphics, and TA-Lib [20] for the technical analysis indicators. We consider that Linux support is particularly important as many financial institutions develop their proprietary algorithmic trading systems on UNIX-like operating systems, mostly Linux.

## 4-2. Concurrent Programming

As Sutter [27] precisely foresaw in 2005, concurrent programming has today become a dominant programming paradigm as single-threaded programs no longer get any significant increase in performance because the CPU's clock speed has hit a wall due to physical limitations (i.e., heat, power consumption, and current leakage). Yet, the CPU's transistor density is still growing, and as a result we are having more processor cores instead of faster clock speeds. Consequently, it has become essential for programmers to adapt concurrent programming in order to fully exploit the continuing gains in processor throughput.

Naturally, our backtesting system is designed to take advantage of multicore CPUs via multithreading. However, the biggest problem in parallelizing a backtester is the fact that backtesting of trading algorithms is inherently a serial process because historical market data used in backtesting is strictly sequential (i.e., time series), it therefore needs to be evaluated serially, making it impossible for developers to exploit data parallelism. We therefore turned to function parallelism instead (cf. Chapter 4-3).

It is known that the speedup of a multithreaded program in parallel computing is limited by the sequential fraction of the program. The theoretical speedup that can be achieved by executing a computer program on a processor capable of executing *n* threads concurrently is quantified as *S(n)* in the following formula:

$$S(n)=\frac{T(1)}{T(n)}=\frac{T(1)}{T(1)\left[B+\frac{1}{n}(1-B)\right]}=\frac{1}{B+\frac{1}{n}(1-B)} \tag{12}$$

where *T(n)* is the time the program takes to finish when executed on *n* threads in parallel, *B* is the fraction of the program that is strictly serial.

If we denote *P* = *(1 – B)* the fraction of the program that is made parallel, then the speedup tends to the reciprocal of *B* as the number of threads tends to infinity in the limit:

$$\lim_{n\to\infty} \frac{1}{(1-P)+\frac{P}{n}} = \frac{1}{1-P} = \frac{1}{B} \tag{13}$$

This equation is commonly known as Amdahl's law. [28] What this implies is that parallelization is only useful for either small numbers of threads when *B* is large, or so-called “embarrassingly parallel” tasks with very high values of *P*. As we already discussed, backtesting of trading algorithms is predominantly a serial process (hence large *B*); consequently, only a few threads are utilized in parallelizing a single backtesting task.

## 4-3. Asynchronous Function Parallel Model

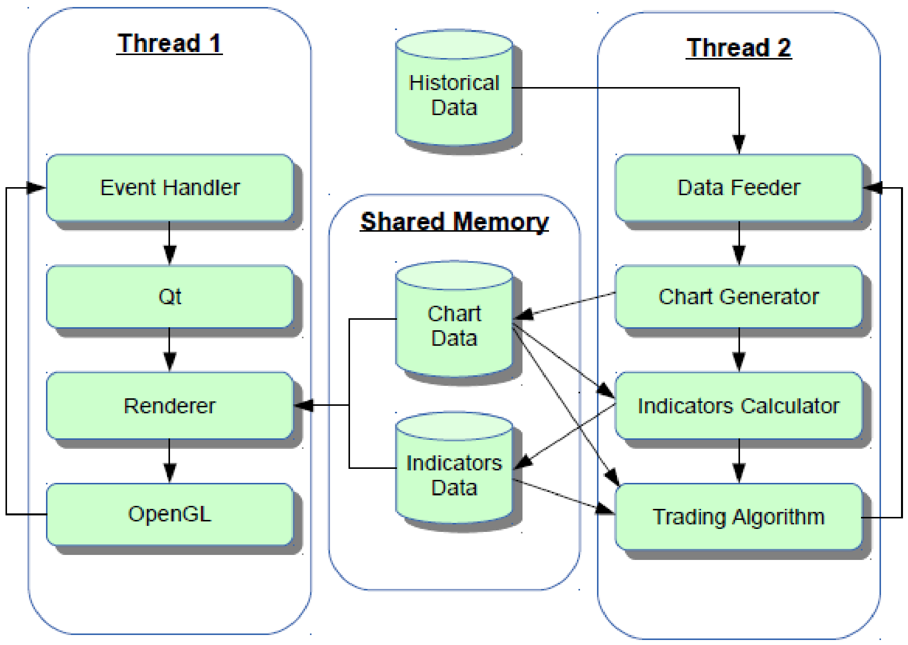


***Figure 11.*** *Asynchronous function parallel model [29] of our backtesting system*

Figure 11 illustrates the architecture of our backtesting system. Here, the first thread (main thread) is dedicated to the GUI loop of the system, and the second thread (child thread) is dedicated to the backtester loop of the system. Both of the two threads run in parallel independently without any synchronization, thereby minimizing the interactions between them. This is an ideal design of multithreading because frequent interactions between threads can often be problematic due to increased complexity (e.g., deadlock).

For each iteration in the backtester loop, the “Data Feeder” at the beginning of the loop reads historical market data one line at a time, retrieving the latest tick in the data, and subsequently the “Chart Generator” (CG) and the “Indicators Calculator” (IC) construct a price chart and technical indicators respectively according to the tick data. The CG and the IC must be executed in that serial order because the IC requires the latest chart data to calculate its indicators correctly. The data produced by both the CG and the IC are stored in shared memory, from which the “Renderer” in the GUI loop asynchronously reads the latest data and draws a chart and indicators on the screen. Each read/write operation associated with the shared memory is done by locking a mutex so that only one thread at a time may access the data in the memory so as to avoid race conditions and data inconsistencies, where nested locks of multiple mutexes are also carefully avoided in order to prevent deadlocks. Finally, the “Trading Algorithm” is executed along with the latest tick, chart and indicators data. All of these modules in the backtester loop as a whole constitute a sequential cohesion and data coupling (i.e., the output from one module is the input to another).

Some trading algorithms use multiple time frame (MTF) charts and indicators for detecting entry/exit points in the market in a more precise manner. Backtesting of such algorithms can be accelerated by having additional threads concurrently generating the charts and indicators of different time frames.

While some other algorithms use combinations of a number of indicators whose calculations are computationally expensive. Backtesting of such algorithms can be accelerated by assigning each additional thread a group of indicators that can be calculated independently in parallel. Note that some indicators are composites of other indicators, i.e., they depend on two or more other indicators for their computation; thus they need to be calculated serially according to their dependencies. It is also important to note that, for the additional threading to be beneficial in this case, the granularity of indicators calculations assigned to each thread must be coarse, i.e., the amount of work per thread must be larger than the parallel overhead associated with the thread.

In both types of trading algorithms above, all the additional threads are synchronized before the Trading Algorithm gets executed (Figure 12):

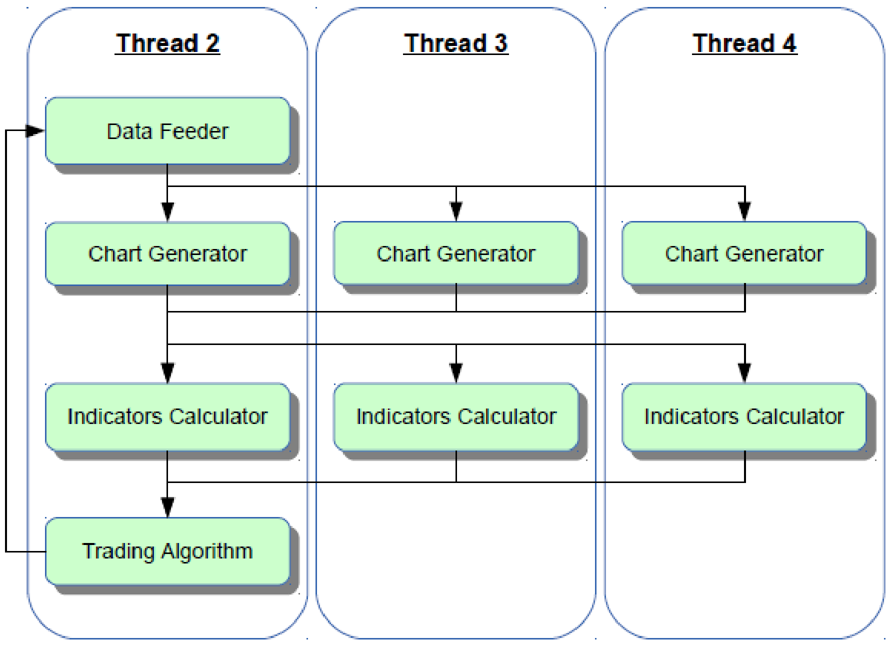
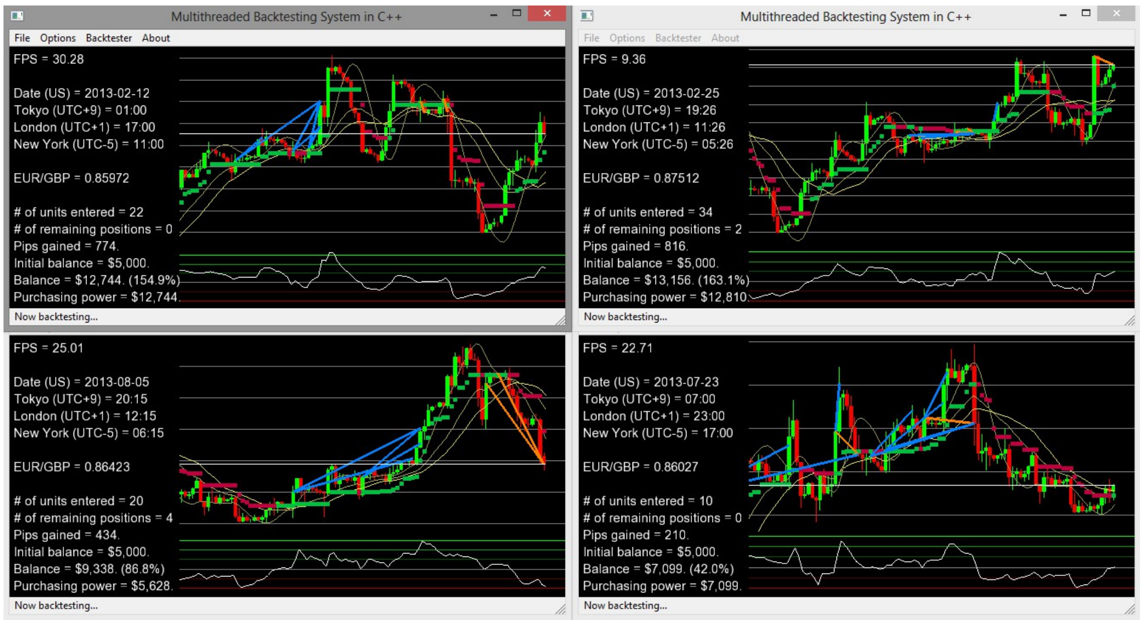


***Figure 12.*** *Additional threading for the backtester loop with synchronization*

There are also some algorithms that simultaneously refer to charts and indicators of multiple currency pairs in order to calculate entry/exit signals. For such algorithms, the “Data Feeder” is assigned to each additional thread with historical data of different currency pairs, and subsequently charts and indicators are computed in parallel. In this model it is vital to synchronize the backtesting process of each thread based on the time recorded within all the historical data, because the execution speed of each thread is different.

As for backtesting multiple trading algorithms in parallel, we decided to run multiple instances of our system instead of having multiple threads executing multiple algorithms within the system. This allowed us to maintain the code complexity of our system as minimal as possible since the operating system instead handles the creation and management of additional threads for the multiple algorithms. This approach is also useful when backtesting the same algorithm recurrently with different parameters, different time frames, different time periods, etc.


Multithreaded Backtesting System in C++
File Options Backtester About
Multithreaded Backtesting System in C++
File Options Backtester About

```
FPS = 30.28

Date (US) = 2013-02-12
Tokyo (UTC+9) = 01:00
London (UTC+1) = 17:00
New York (UTC-5) = 11:00

EUR/GBP = 0.85972

# of units entered = 22
# of remaining positions = 0
Pips gained = 774.
Initial balance = $5,000.
Balance = $12,744. (154.9%)
Purchasing power = $12,744.
Now backtesting...
```

```
FPS = 9.36

Date (US) = 2013-02-25
Tokyo (UTC+9) = 19:26
London (UTC+1) = 11:26
New York (UTC-5) = 05:26

EUR/GBP = 0.87512

# of units entered = 34
# of remaining positions = 2
Pips gained = 816.
Initial balance = $5,000.
Balance = $13,156. (163.1%)
Purchasing power = $12,810.
Now backtesting...
```

```
FPS = 25.01

Date (US) = 2013-08-05
Tokyo (UTC+9) = 20:15
London (UTC+1) = 12:15
New York (UTC-5) = 06:15

EUR/GBP = 0.86423

# of units entered = 20
# of remaining positions = 4
Pips gained = 434.
Initial balance = $5,000.
Balance = $9,338. (86.8%)
Purchasing power = $5,628.
Now backtesting...
```

```
FPS = 22.71

Date (US) = 2013-07-23
Tokyo (UTC+9) = 07:00
London (UTC+1) = 23:00
New York (UTC-5) = 17:00

EUR/GBP = 0.86027

# of units entered = 10
# of remaining positions = 0
Pips gained = 210.
Initial balance = $5,000.
Balance = $7,099. (42.0%)
Purchasing power = $7,099.
Now backtesting...
```

***Figure 13.*** *Running multiple instances of our system in parallel*

## 5. Trading Algorithm Optimization

### 5-1. Walk-Forward Analysis

As already discussed in Chapter 3-2, trading algorithm developers usually split their historical market data in two sets: the in-sample data for optimization and the out-of-sample data for validation. However, this simple binary separation of data is not always the best, especially for the testing of low-frequency algorithms (e.g., the trend-following), because the out-of-sample data period is usually too short for such algorithms to make a sufficient amount of trades with which a statistically significant test can be conducted.

Pardo [30] devised a better approach called walk-forward analysis, in which developers still optimize their algorithm on the in-sample data and validate it on the out-of-sample data, but they repeat the process by forwarding the data period, i.e., after each out-of-sample test the in-sample data period is shifted forward by the same amount of data period covered by the out-of-sample test.

Let us illustrate this method with an example. Suppose we have 12 months of historical data (from January to December, 2013) and our algorithm requires at least 3 months of data for sufficient testing and optimization, and 1 month of data for proper validation. We start the development and optimization of our algorithm using only the first 3 months of data as the in-sample data (January–March), and then we validate the algorithm on the 1-month out-of-sample data (April), after which we record the optimum parameters of the algorithm in the data period. Subsequently, we shift the data period forward by 1 month. As a result we now have the new in-sample data (February-April) and out-of-sample data (May). We proceed to optimize the algorithm on this new in-sample data using the optimal parameters that we found in the previous step, validate it on the new out-of-sample data, find and record new optimal parameters, and shift the data period forward again. We repeat this process until we reach the end of our historical data (December). When we reach the end, we revert to the first month and test the algorithm for the entire period recurrently by switching between parameters that were found optimal during all the earlier steps.

Apropos, backtesting systems are of no use without trading algorithms to test with. Therefore, not only did we develop the backtsting system but we also simultaneously developed our own trading algorithm and integrated it in the backtesting system as a black-box. Our trading algorithm is intended to trade the EUR/GBP currency pair, although it is possible to trade other currency pairs. Table 2 shows the setup of the EUR/GBP historical data for the walk-forward analysis of our trading algorithm.

| Walk # | In-sample data period | Out-of-sample data period |
|---|---|---|
| 1 | 06/01/'13 – 28/06/'13 (In1) | 30/06/'13 – 30/08/'13 (Out1) |
| 2 | 03/03/'13 – 30/08/'13 (In2) | 01/09/'13 – 01/11/'13 (Out2) |
| 3 | 05/05/'13 – 01/11/'13 (In3) | 03/11/'13 – 03/01/'14 (Out3) |
| 4 | 06/01/'13 – 03/01/'14 (In4) | N/A |

***Table 2.*** *Setup of the historical data for the walk-forward analysis*

In Walk #1, we initially optimize our algorithm by strictly using the 6-month in-sample data (In1) and subsequently validate it on the 2-month out-of-sample data (Out1), after which the in-sample data period is shifted forward by the period covered by the out-of-sample test (i.e., 2 months). we repeat this process up to Walk #3 and finally optimize the algorithm on the 12 months of data in Walk #4 using the information obtained from the previous steps.

## 5-2. Results

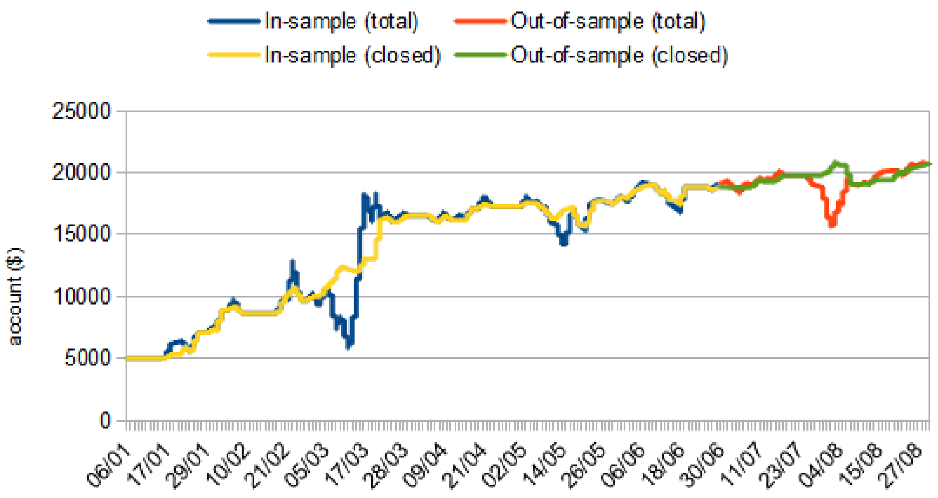


***Figure 14.*** *Equity curves of our trading algorithm in Walk #1*

Figure 14 shows the equity curve generated in Walk #1 by backtesting our trading algorithm which we initially optimized for the in-sample data (In1)*. Although the algorithm experienced a drawdown during the out-of-sample backtest (Out1), it was still profitable. we proceeded to optimize the algorithm to the in-sample data in Walk #2 (In2), which gave us an astonishing result as shown in Figure 15 below.

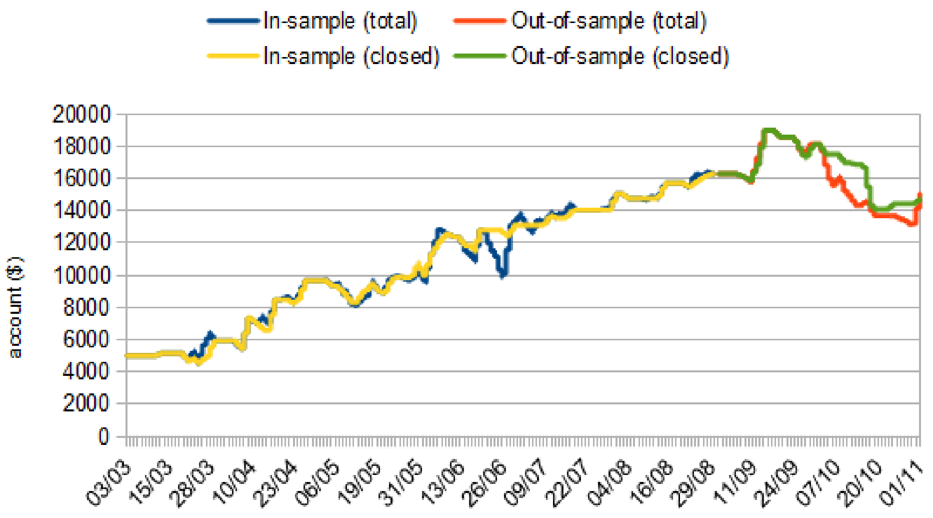


***Figure 15.*** *Equity curves of our trading algorithm in Walk #2*

However, the subsequent out-of-sample backtest (Out2) suggested that the algorithm might be overfitted to the in-sample data (In2), since, as it can clearly be seen in Figure 10, the constant profitability of the algorithm during the in-sample testing ceased to continue on the out-of-sample data (but still the algorithm made a new high during the out-of-sample test).

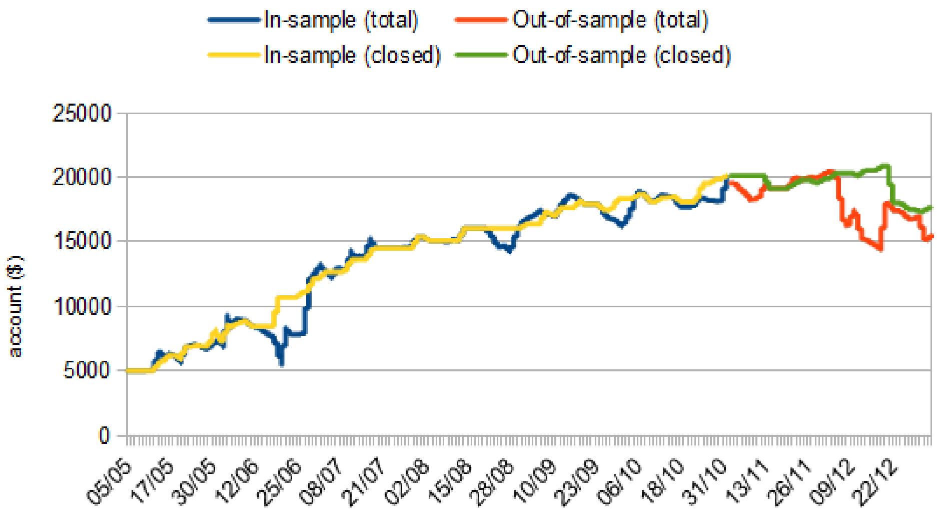


***Figure 16.*** *Equity curves of our trading algorithm in Walk #3*

We further optimized the algorithm to the new in-sample data (In3); however, the new out-of-sample testing (Out3) also showed a sign of over-optimization with a decline in the account value.

*Video available at: https://www.youtube.com/watch?v=VtPTsXnxzT4

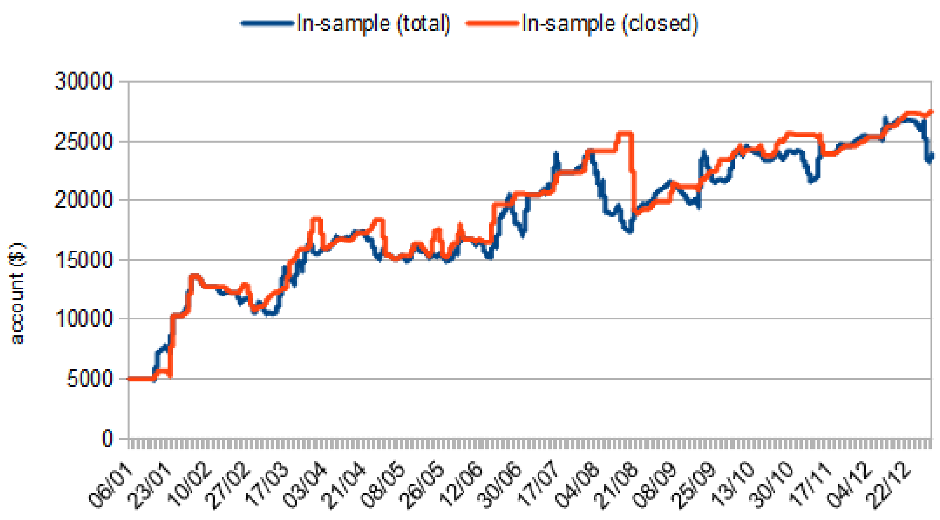


***Figure 17.*** *Equity curves of our trading algorithm in Walk #4*

After analyzing the results of all the previous steps, we finally achieved 450% of return per annum in Walk #4 (Figure 17), with the winning percentage = 58.8%, profit facor = 1.74, average profit = $359, average loss = $288, maximum drawdown = $6,704, and Sharpe Ratio = 2.45.

Now that we have finished the walk-forward analysis of our trading algorithm, we can start porting it over to the MetaTrader 4 or other electronic trading platform and forward test it on a demo account with a broker. Once we are satisfied with the result, we may go live with it and risk our own real money.

## 6. Conclusion

The main goal of this research was to design and build a fast, flexible, and accurate backtesting system of trading algorithms for the foreign exchange market. Great importance was placed upon execution efficiency since backtesting is a time-consuming process as it is necessary for trading algorithm developers to backtest their algorithm recurrently with different conditions in order to make it profitable. For this reason we wrote the system in the C++ programming language and exploited function parallelism via multithreading so as to take the best advantage of modern multicore CPUs.

We also designed and implemented a trading algorithm for the EUR/GBP currency pair on the system, and analyzed its historical performance via walk-forwad analysis, according to which the algorithm is highly profitable at least in the data period used.

## 7. Future Work

As previously shown in Chapter 4-3, running our backtesting system in multiple instances allows users to backtest a trading algorithm with various conditions in parallel, helping them find an optimum set of parameters heuristically. This heuristic process of finding optimum parameters can be efficiently automated with genetic algorithms (GAs).

In order to reduce the time taken up by this finding process using GAs, it will be crucial to increase the degree of parallelism – that is, to increase the number of instances of the system (i.e., individuals) running in parallel. To this end, we can modify the system and make it run on a computer cluster or alternatively on the GPU by using a GPGPU (general-purpose computing on GPUs) API, such as OpenCL [31].

Another possible modification for the backtesting system is to allow fast prototyping of trading algorithms using Runtime-Compiled C++ [32. As already discussed in Chapter 4-1, it takes a long time to compile C++ code, and every time a change is made in the code, it needs to be re-compiled for the new native code generation. Using the Runtime-Compiled C++ allows us to run the modified code without re-compillation, thereby significantly increase the productivity of trading algorithm developers.

## References

[1] Barry Johnson, *“Algorithmic Trading and DMA: An Introduction to Direct Access Trading Strategies,”* 4Myeloma Press, 2010.

[2] FSOC, *“2012 Annual Report,”* US Department of the Treasury, 2012. http://www.treasury.gov/initiatives/fsoc/studies-reports/Pages/2012-Annual-Report.aspx

[3] Dagfinn Rime and Andreas Schrimpf, *“The anatomy of the global FX market through the lens of the 2013 Triennial Survey,”* Bank for International Settlements, 2013.

[4] MetaTrader 4. http://www.metatrader4.com/

[5] NinjaTrader. http://www.ninjatrader.com/

[6] TradeStation. http://www.tradestation.com/

[7] ISO/IEC 14882, *“Programming Languages – C++ 2nd Edition,”* 2003.

[8] Jon Postel, *“Internet Protocol (RFC 760),”* ISI/USC, 1980. http://www.rfc-editor.org/rfc/rfc760.txt

[9] Jon Postel, *“Transmission Control Protocol (RFC 768),”* ISI/USC, 1980. http://www.rfc-editor.org/rfc/rfc761.txt

[10] Jarrad Hee, Yining Chen, and Wayne Huang, *“Straight Through Processing Technology in Global Financial Market: Readiness Assessment and Implementation,”* Journal of Global Information Management, 11(2), 2003.

[11] MetaQuotes Language 4. http://www.mql4.com/

[12] Dave Walton, *“Data Mining Bias,”* Van Tharp Institute, 2013. http://www.vantharp.com/Tharps-Thoughts/655_nov_13_2013.html

[13] Michael Chernick, *“The Essentials of Biostatistics for Physicians, Nurses, and Clinicians,”* pp.49–50, John Wiley & Sons, 2011.

[14] Dave Aroson, *“Evidence Based Technical Analysis,”* John Wiley & Sons, 2006.

[15] William Sharpe, *“The Sharpe Ratio,”* The Journal of Portfolio Management, 21(1), 1994.

[16] Michael Harris, *"The Profitability Rule,"* Stocks & Commodities, September Issue, 2002.

[17] Emilio Tomasini and Urban Jaekle, *"Trading Systems: A New Approach to System Development and Portfolio Optimisation,"* Harriman House, 2009.

[18] MetaQuotes Software, "The algorithm of ticks' generation within the strategy tester of the MetaTrader 5 terminal." https://www.mql5.com/en/articles/75

[19] MetaQuotes Software, *"Automated Trading with MetaTrader 4."* http://www.metaquotes.net/en/metatrader4/automated_trading

[20] TA-Lib. http://ta-lib.org/

[21] Jesus Fernandez-Villaverde and S. Boragan Aruoba, *"A Comparison of Programming Languages in Economics,"* 2014.

[22] Boost. http://www.boost.org/

[23] Gunter Obiltschnig, *"C++ for Safety-Critical Systems,"* 2009.

[24] ISO/IEC 14882:2011, *"Programming Languages – C++,"* 2011.

[25] Qt. https://qt-project.org/

[26] OpenGL. http://www.opengl.org/

[27] Herb Sutter, *"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software,"* Dr. Dobb's Journal, 30(3), 2005.

[28] David Rodgers, "Improvements in multiprocessor system design," p.226, ACM SIGARCH Computer Architecture News, 13(3), 1985.

[29] Ville Mönkkönen, *"Multithreaded Game Engine Architectures,"* 2006.

[30] Robert Pardo, *"Design, Testing, and Optimization of Trading Systems,"* Wiley Trader's Exchange, 1992.

[31] OpenCL. http://www.khronos.org/opencl/

[32] Runtime-Compiled C++. http://runtimecompiledcplusplus.blogspot.com/