FUB Trading Simulator

Quick Start Guide

Felix Bertram

2018

# Introduction

The project FUB\_TradingSimulator implements a trading simulator for stocks and options with the following features:

* import data in CSV format, with configurable column-mapping
* automatic data update/ download from web sources
* calculate indicators, with a growing library of standard indicators
* simulate stock trades, and portfolios of stocks. Currently market and stop orders are supported
* simulate option trades. Currently this is limited to cash-settled European-style options
* create fully customized Excel reports with just a few lines of VBA
* export results to R, for further research
* strong focus on easy-to-use time-series APIs, to make coding a breeze
* multi-threaded optimizer engine, able to load all your CPU cores
* demo algorithms to shorten your learning curve

The following features are planned for the near future:

* support for limit orders
* API documentation

The following environment is required for building and running the simulator:

* Microsoft Visual Studio, Community 2015 or better
* Excel, and Microsoft.Office.Interop.Excel for exporting results to Excel. If your environment does not meet these requirements, comment the line #define ENABLE\_EXCEL at the top of Logger.cs
* R, RDotNet, RDotNet.NativeLibrary, and DynamicInterop for exporting results to R. If your environment does not meet these requirements, comment the line #define ENABLE\_R at the top of Logger.cs
* data files for the instruments to simulate. For convenience, some end-of-day quotes are pre-configured

In this quick start guide, we will go through the demo projects and explain the main concepts of the simulation engine, and how to write algorithms based on this engine.

# Demo 01: Calculating Indicators

This demo loads a range of quotes for a common stock, calculates indicators on the quote data, and visualizes the data in a simple plot. In doing so, the demo illustrates the following key concepts:

* loading data
* processing bars and accessing instruments
* processing time series data
* plotting data

## Loading Data

Before we even start simulating algorithms, we need to load data for the relevant instruments into the simulator engine. FUB\_TradingSimulator distinguishes between the following entities:

* Nickname. This is the name, by which we will refer to our instrument. The nickname must be unique for the full database we are using. Therefore, the nickname should be more than just the instrument’s ticker, and include the category and timeframe as well. Examples for nicknames are “AAPL.Stock” and “^XSP.Index”, but could also be “My Favorite Stock”.
* Ticker. This is the sequence of typically 3 to 5 characters used to identify an instrument within a given exchange. The ticker needs to be unique within each algorithm, for us to distinguish the instruments. It is possible for multiple nicknames to point to the same ticker, e.g. when they use different time frames. Examples for tickers are “AAPL”, or “XSP”.
* Symbol. Most of the time, the symbol is identical to the ticker. When it comes to options though, the symbol also includes the expiry date, the option right, and the strike price. Examples for symbols are “[XSP180919C00289000](https://finance.yahoo.com/quote/XSP180919C00289000?p=XSP180919C00289000)”, or “[XSP180919P00284000](https://finance.yahoo.com/quote/XSP180919P00284000?p=XSP180919P00284000)”.
* Data Source. With the nickname in hand, we can create a data source for this instrument. The simulator engine will look up the nickname, gather the specific information about this instrument, and instantiate an appropriate data source for it. Typically, we have little reason to directly interact with data sources while developing our algorithms.
* Instrument. As a data source processes its data, it will instantiate one or more instruments on the fly. An instrument encapsulates everything we know about a tradeable share or contract. Part of the data, e.g. the symbol, or an option’s expiry date are static. Other information, e.g. the open/ high/ low/ close quotes are time series.

To load data into the simulation engine, we need to do the following:

* Point the simulation engine to the correct location for our data base. This is achieved by setting the DataPath property.
* Specify the simulation range. To do so, we need to set the StartTime and EndTime properties.
* Add one or more data sources to our algorithm. We do this, by adding DataSource objects to the DataSources property.

The relevant portion of an algorithm will look like this:

// set simulation time frame

StartTime = DateTime.Parse("01/01/2007");

EndTime = DateTime.Parse("08/01/2018");

// add instruments

DataPath = \_dataPath;

DataSources.Add(DataSource.New("AAPL.Stock"));

## Processing Bars and Accessing Instruments

Now that we have specified our simulation range and instruments, we can start processing data. FUB\_TradingSimulator creates a single event for each bar with the same time stamp. This greatly simplifies the development of algorithms working on instrument portfolios, or with options. Each bar is requested through the enumerable property sim time with a simple loop.

foreach (DateTime simTime in SimTimes)

{

// process bars in this loop

}

Inside that loop, we can access the Instruments dictionary, which will hold all known instruments. There are a number of ways how to find the instrument we are looking for.

If our algorithm only references a single instrument, e.g. a basic stock-trading algorithm, we can simply use the first element of the Instruments dictionary:

Instrument instrument = Instruments.Values.First();

For more complex algorithms referencing multiple instruments, we can look up the instrument via the nickname. Please note that this won’t work, if there is a one-to-many relationship between our nickname and the resulting instruments:

Instrument instrument = FindInstrument(\_instrumentNick);

If there is a one-to-many relationship between our nickname and the resulting instruments, we are probably dealing with options. We’ll talk about that when we discuss the options trading demo.

An instrument holds, among others, the following static fields:

* Nickname. The nickname we used to add the data source
* Name. The descriptive name of the instrument. E.g. “Alphabet Inc Class A”.
* Symbol. The full symbol used to trade this instrument. This may be the ticker, as in “GOOGL”, or the ticker plus expiry, right, and strike as in “[XSP180919P00284000](https://finance.yahoo.com/quote/XSP180919P00284000?p=XSP180919P00284000)”.
* LastTime. This is the timestamp of the last bar received by this instrument. This is important to distinguish active from stale instruments.

## Processing Time Series Data

An instrument also holds, among others, the following time series:

* Open, High, Low, Close, Volume. The market quote data
* Bid, Ask, BidVolume, AskVolume. The bid/ask quote data

We can access time series data like an array. The index 0 points to the most current data, index 1 points to the data from the previous bar, and so forth. Typically, we can access up to 256 bars in the past:

double currentClose = instrument.Close[0];

double previousClose = instrument.Close[1];

Indicators can be calculated on any time series data. The result of an indicator is typically also a time series:

ITimeSeries<double> ema26Series = instrument.Close.EMA(26);

double ema26Value = ema26Series[0];

With the output of indicators being time series, we can calculate indicators on indicators. The following code shows the calculation of the MACD step by step:

ITimeSeries<double> ema26 = instrument.Close.EMA(26);

ITimeSeries<double> ema12 = instrument.Close.EMA(12);

ITimeSeries<double> macd = ema12.Subtract(ema26);

ITimeSeries<double> signal = macd.EMA(9);

## Plotting data

We can also plot our data in Excel and R. To do so, we need to first create a Logger object:

private Logger \_plotter = new Logger();

A Logger object can hold multiple plots. Therefore, we need to select the plot we are working on first, and specify the name of the horizontal axis:

\_plotter.SelectPlot("indicator vs time", "date");

Now, for every bar we process, we need to set the value on the horizontal axis:

\_plotter.SetX(simTime);

One we have done that, we can add one or more values on the vertical axis:

\_plotter.Log(instrument.Symbol, instrument.Close[0]);

\_plotter.Log("ema26", ema26[0]);

\_plotter.Log("ema12", ema12[0]);

\_plotter.Log("macd", macd[0]);

\_plotter.Log("signal", signal[0]);

This will result in internal CSV file being generated with the following format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| date | ^GSPC | ema26 | ema12 | macd | signal |
| 1/6/2015 | 202.61 | 216.4622222 | 212.0276923 | -4.434529915 | -0.886905983 |
| 1/7/2015 | 225.9 | 216.8562963 | 212.8461538 | -4.01014245 | -1.688934473 |
| 1/8/2015 | 262.14 | 220.2398354 | 220.5557396 | 0.315904254 | -1.448372426 |

In case we are interested in this CSV file, we can save it to a file:

\_plotter.SaveAsCsv(filePath);

Typically, a graphical representation of our data is more appealing. We can create a plot in Excel like this:

\_plotter.OpenWithExcel(\_excelPath);

The parameter passed into the OpenWithExcel function points to an Excel template including a VBA macro that will receive the CSV, and create the desired output. The project includes a simple template, which will create plots like this:

By modifying the Excel templates, there are virtually no limits to the customized plots and reports that can be created.

# Demo 02: Trading Stocks

This demo loads a range of quote data for a common stock, calculates indicators on the quote data, and places trades based on the indicator values. The demo illustrates the following key concepts:

* Setting initial capital
* Determining net asset value
* Placing orders
* Keeping track of open positions

The method implemented here is a simple moving-average crossover, as this is easy to understand and widely referenced. Please don’t use this for actual trading.

## Setting Initial Capital

Before we start trading, we need to set simulator’s initial capital. As all simulations start with all flat positions, this capital will be all in cash, which is why we set the simulator’s cash property to do so.

Cash = \_initialCash;

We can use the Cash property at any point in the simulation, to determine the amount of cash at hand. We can also modify the Cash property, in order to simulate deposits or withdrawals at any point during the simulation.

## Determining Net Asset Value

When we are ready to place orders, we need to determine the net asset value of our portfolio, which is the sum of our cash at hand, plus the liquidation value of all positions we hold. The simulator calculates the net asset value on every bar, and provides it as a time series.

The net asset value, divided by the current price of the instrument, should give us the number of shares we can afford:

int targetPosition = (int)Math.Floor(NetAssetValue[0] / instrument.Close[0]);

Note that typically we can’t place an order to be filled on the close of the current bar. Therefore, and in case the instrument’s price moves significantly before the next bar’s open, this calculation might not be fully correct.

With the net asset value being a time series, we have easy access to the recent performance of our trading system. For example, we could determine the highest high of the last month like this:

double highestHigh = NetAssetValue.Highest(21)[0];

This is very convenient for implementing money-management schemes.

## Placing Orders

To place an order, we use the Trade() method. It takes at least one parameter: the quantity of shares/ contracts to trade. A positive value will buy the instrument, while a negative value will sell the instrument. The second parameter specifies the order type and is optional. Typically, our orders will be submitted on the next bar. Here, we place a market order to be executed on the next bar’s open:

instrument.Trade(targetPosition - currentPosition, OrderType.openNextBar);

There are other order types available, most importantly stop orders, which allow us to place orders contingent on the price touching the specified value.

## Keeping Track of Open Positions

The simulator keeps track of all open positions via the Positions property. Flat instruments will be removed from the property. Therefore, checking the current position is a two-step process: First, we need to check if we have a position, and then we can check the position size:

int currentPosition = Positions.ContainsKey(instrument)

? Positions[instrument]

: 0;

The integer returned for the position size will be positive for long positions, and negative for short positions. It is good practice to calculate target positions and current positions separately, and then trade the difference between the two. That way, we should not get confused with long and short positions, and the sign of the quantity to trade.

At the end of the demo, we can see how the algorithm attempts to follow bullish trends, and exit positions in bearish times:

# Demo 03: Trading Stock Portfolios

# Demo 04: Trading Options

# Demo 05: Running the Optimizer

# Demo 06: Order Types