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, either as plain files, or zipped archives
* calculate indicators, with a growing library of standard indicators
* simulate stock trades
* simulate option trades. Currently this is limited to cash-settled European-style options
* export results to Excel, where you can fully customize reports with 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
* demo algorithms to shorten your learning curve

The following features are planned for the near future:

* automatic download/ update of data files
* customizable optimizer engine

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 in 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 in Logger.cs
* data files for the instruments to simulate. For convenience, some end-of-day quotes are included

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.

# 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);

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.

All techniques presented so far are used in Demo01\_Indicators. It is highly encouraged to run this demo, and try making some simple changes to the code.