Market Microstructure Simulator: Strategy Definition Language

Anton Kolotaev

Chair of Quantitative Finance, École Centrale Paris

anton.kolotaev@gmail.com

March 11, 2014

Overview

Motivation

Last years financial market microstructure attracts more and more attention due to its potential capability to explain some macroscopic phenomena like liquidity crisis, herd behaviour that were observed during recent crises. Researchers in this field need a simulation tool that would support them in their theoretical studies. This tool would take into account in exhaustive and precise manner market rules:

1. how orders are sent to the market
2. how they are matched
3. how information is propagated

Simulator architecture should allow to model any behaviour of market agents, i.e. to be very flexible in trading strategies definition. It might be used to study impact on market caused by changing tick size, introducing a new order type, adding a new order matching rule.

Design Goals

1. **Flexibility and Extensibility**. A simulation library must have a very modular design in order to provide a high level of flexibility to the user. This requirement comes from the original purpose of a simulation as a test bed for experiments with different models and parameters.

2. **Used-friendliness**. Since a typical simulation model is composed of many hundreds and thousands blocks, it is very important to provide a simple way for user to tell how a behaviour wanted differs from the default one. Simulator API should be friendly to modern IDEs with intellisense support

3. **Performance**. A user should be allowed to choose between small start-up time and high simulation speed.
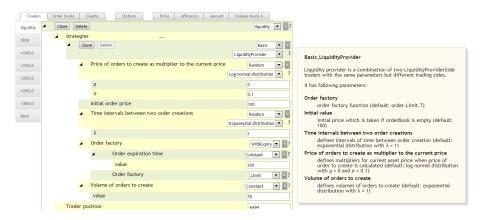
Library Content

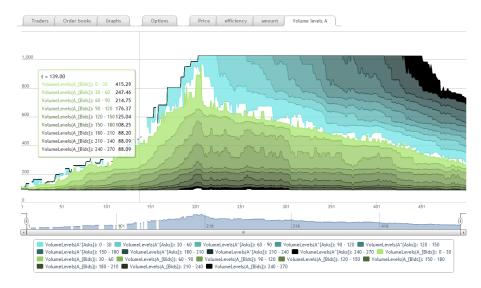Implemented so far:

1. Strategies:
   1. Side based strategies:
      1. Signals: Trend Follower, Crossing Averages
      2. Fundamental Value: Mean Reversion, Pair Trading
   2. Desired position based strategies: RSI, Bollinger Bands
   3. Price based strategies: Liquidity Provider, Market Data, Market Maker
   4. Adaptive strategies: Trade-If-Profitable, Choose-The-Best, MultiarmedBandit
   5. Arbitrage strategy
2. Meta-orders: Iceberg, StopLoss, Peg, FloatingPrice, ImmediateOrCancel, WithExpiry
3. Statistics: Moving/Cumulative/ExponentiallyWeighted Average/Variance, RSI, MACD etc
4. Fast queries: Cumulative volume of orders with price better than given one etc.
5. Local and remote order books

Web interface

Web interface allows to compose a market to simulate from existing objects and set up their parameters

Rendering results

Motivation for a dedicated language

1. **Syntax**. We may design an external DSL so its syntax describes very well domain specific abstractions.

2. **Error checking**. A DSL compiler can detect incorrect parameter values as soon as possible thus shortening simulation development cycle and facilitating refactorings.

3. **Multiple target languages**. A DSL can be compiled into different languages: e.g. into Python to provide fast start-up and into C++ to have highly optimized fast simulations. A new target language introduced, only simple modules need to be re-written into it; compound modules will be ported automatically.

4. **IDE support**. Modern IDEs like Eclipse, Intellij IDEA allow writing plug-ins that would provide smart syntax highlighting, auto-completion and check errors on-the-fly for user-defined DSLs.

5. **High-level optimizations** are easier to implement.

Functions

Functions represent compound modules of a simulation model.

```
@label = "LogReturns_{%(timeframe)s}{%(x)s}"
def LogReturns(/** observable data source */   x = const(1.),
               /** lag size */                 timeframe  = 10.0)

     =  Log(x / x~>Lagged(timeframe))
```

Intrinsic functions represent simple modules and import classes from the target language into the DSL.

```
/**
 *  Observable that adds a lag to an observable data source
 *  so Lagged(x, dt)(t0+dt) == x(t0)
 */
@python.intrinsic("observable.lagged.Lagged_Impl")
@label = "Lagged_{%(timeframe)s}{%(source)s}"
def Lagged (/** observable data source */   source   = const (1.),
            /** lag size */                 timeframe = 10.0) : IObservable[Float]
```

Types for parameters and return values are inferred automatically

Classes

Classes group functions and share fields among them

```
@label = "MACD_{%(fast)s}^{%(slow)s}(%(source)s)"
type macd(/** source */          source = .const(1.),
          /** long period */     slow = 26.0,
          /** short period */    fast = 12.0)
{
    def Value = source~>EW(2./(fast+1))~>Avg - source~>EW(2./(slow+1))~>Avg

    @label = "Signal^{%(timeframe)s}_{%(step)s}(%(x)s)"
    def Signal(/** signal period */          timeframe = 9.0,
               /** discretization step */    step = 1.0)

        = Value~>OnEveryDt(step)~>EW(2/(timeframe+1))~>Avg

    @label = "Histogram^{%(timeframe)s}_{%(step)s}(%(x)s)"
    def Histogram(  /** signal period */          timeframe = 9.0,
                    /** discretization step */    step = 1.0)

        = Value - Signal(timeframe, step)
}
```

Class Inheritance

To stimulate code re-use classes may derive from other classes

```
abstract type IStatDomain(source = .const(0.))
{
    def StdDev    = Var~>Sqrt
    def RelStdDev = (source - Avg) / StdDev
}


@label = "EW_{%(alpha)s}(%(source)s)"
type EW(alpha = 0.015) : IStatDomain
{
    @python.intrinsic("moments.ewma.EWMA_Impl") def Avg : IDifferentiable
    @python.intrinsic("moments.ewmv.EWMV_Impl") def Var => Float
}


@label = "Moving_{%(timeframe)s}(%(source)s)"
type Moving(timeframe = 100.) : IStatDomain
{
    @python.intrinsic("moments.ma.MA_Impl") def Avg : IDifferentiable
    @python.intrinsic("moments.mv.MV_Impl") def Var => Float
}
```

Example: Signal Side Strategies

```
type TrendFollower(
        /** parameter |alpha| for exponentially weighted moving average */
        alpha         = 0.15,
        threshold     = 0.0,
        book          = .orderbook.OfTrader()) : SignalStrategy
{
    def Signal_Value = book~>MidPrice ~>EW(alpha)~>Avg
                                      ~>Derivative
}


type CrossingAverages(
        /** parameter |alpha| for exponentially weighted moving average 1 */
        alpha_1       = 0.15,
        /** parameter |alpha| for exponentially weighted moving average 2 */
        alpha_2       = 0.015,
        threshold     = 0.0,
        book          = .orderbook.OfTrader()) : SignalStrategy
{
    def Signal_Value =  book~>MidPrice~>EW(alpha_1)~>Avg -
                        book~>MidPrice~>EW(alpha_2)~>Avg
}
```

```scala
abstract type SideStrategy
{
    def Strategy(/** Event source making the strategy to wake up*/
                eventGen        = event.Every(math.random.expovariate(1.)),
                /** order factory function*/
                orderFactory    = order.side.Market())

        = Generic(orderFactory(Side), eventGen)
}

abstract type SignalStrategy : SideStrategy
{
    def Side =
        if Signal_Value >   threshold then side.Buy()  else
        if Signal_Value < 0-threshold then side.Sell() else
                                      side.Nothing()
}
```

- Developed at the Chair of Quantitative Finance at École Centrale Paris
- Source code and documentation can be found at https://github.com/fiquant/marketsimulator
- OS supported: Linux, Mac OS X, Windows
- Browsers supported: Chrome, Firefox, Safari, Opera
- Requirements: Scala 10.2, Python 2.7 (Flask, NumPy, Pandas, Blist, Veusz).
- Translator from the DSL into optimized C++ code is to be developed.

Thank you for your attention!