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.

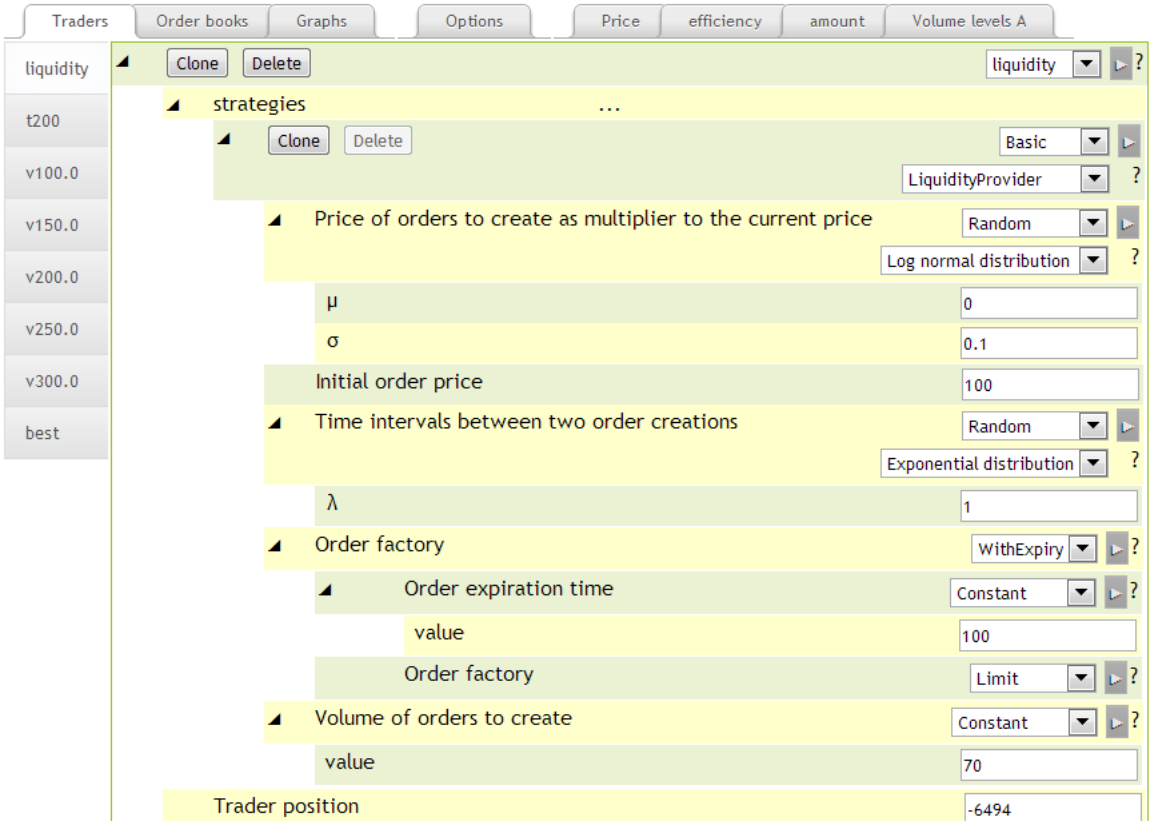
Current State

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



Basic,LiquidityProvider

Liquidity provider is a combination of two LiquidityProviderSide traders with the same parameters but different trading sides.

It has followng parameters:

Order factory
order factory function (default: order.Limit.T)

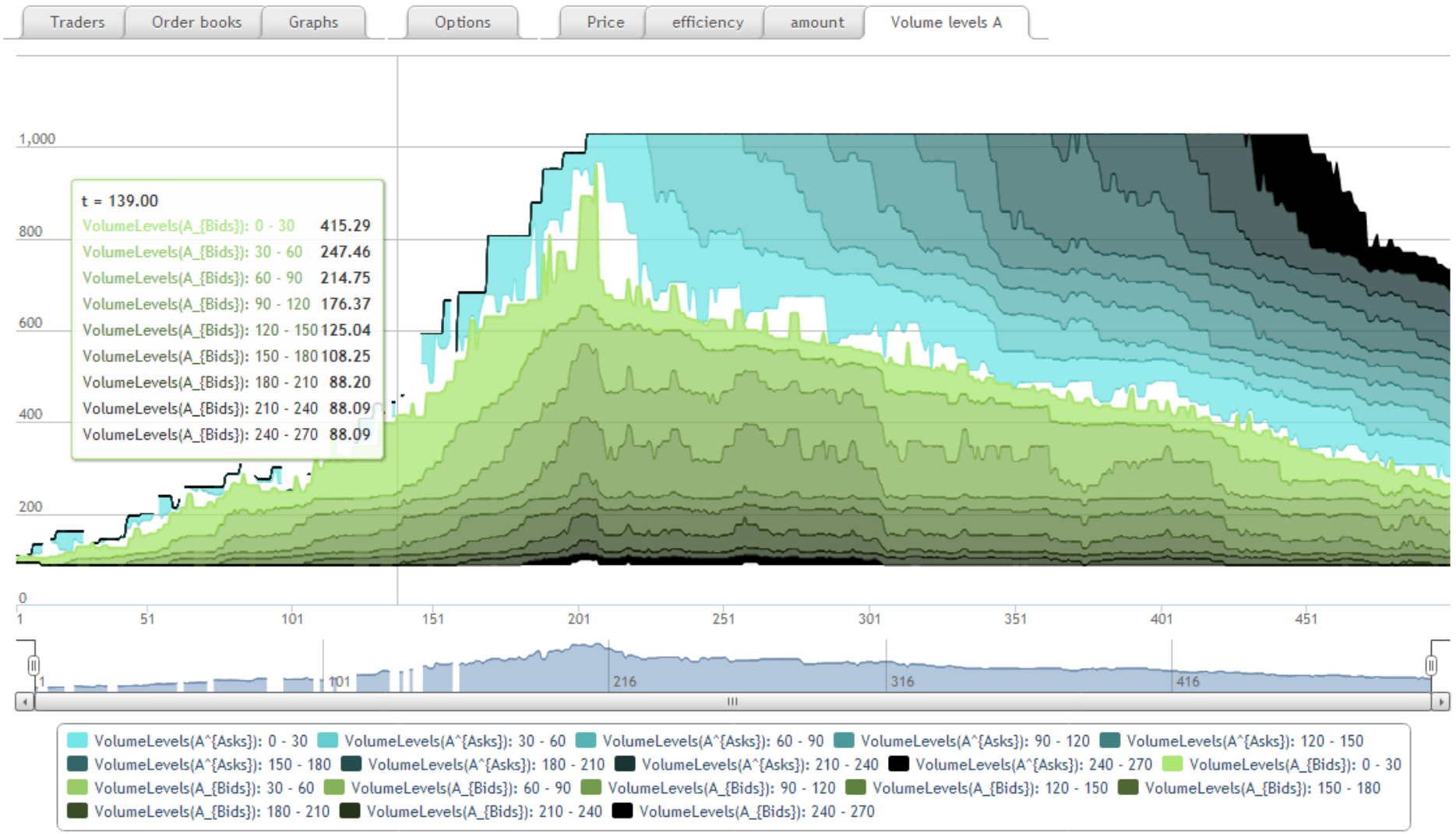
Initial value
initial price which is taken if orderBook is empty (default: 100)

Time intervals between two order creations
defines intervals of time between order creation (default: exponential distribution with $\lambda = 1$)

Price of orders to create as multiplier to the current price
defines multipliers for current asset price when price of order to create is calculated (default: log normal distribution with $\mu = 0$ and $\sigma = 0.1$)

Volume of orders to create
defines volumes of orders to create (default: exponential distribution with $\lambda = 1$)

Rendering results



Motivation for an external DSL

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.

Scala as a language to implement the DSL compiler

1. As a ML-family member, very suitable for sophisticated symbol processing tasks like compilation: an internal DSL to parse texts, algebraic data types, pattern matching, a powerful collections library, mixin inheritance via traits.
2. Good balance between functional and imperative programming features – easier to find software engineers
3. The most mainstream of functional programming languages that results in wide community, excellent library and tool support, mature IDEs.
4. Very portable since runs on JVM.
5. Statically typed
6. Right choice to develop a Web server.

Functions

Conceptually, a "function" declaration defines a term with a constructor provided (like case classes in Scala).
Types for input arguments are inferred automatically from their initializers and for the "return" value - from the function body.
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))
```

All methods can be considered as extension methods of their first argument.
`x~>Lagged(dt)` is equivalent to `.observable.Lagged(x, dt)`.

Intrinsic Functions

Intrinsic functions import simple modules from a target language into the DSL. "Return" types for intrinsic functions must be specified explicitly.

```
/**
 *  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]
```

Type System I

Types correspond to interfaces without methods from mainstream languages. They are used for error checking and for function overloading.

1. Simple types may derive from other types or be aliases

```
type Float
type Int  : Float
type Volume = Int
```

2. Tuple and function types

```
(T1, T2, T3)
(T1, T2, T3) => R
```

3. Top (`Any`) and bottom (`Nothing`) types.
4. Lists (`List[T]`)

Type System II

Types may be generic.
Functions are contravariant in the input type and covariant in the output type: `CanCast(D,B)` $\wedge$ `CanCast(R,T)` $\Rightarrow$ `CanCast(B=>R,D=>T)`.
All other types are covariant.

```
type IEvent
type IFunction[T] = () => T
type IObservable[T] : IFunction[T], IEvent
```

Classes I

Classes are syntax sugar for a type declaration, constructor function and member accessors

```
@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)
}
```

Classes II

Previous definition is de-sugared at typing stage into

```
type macd

@label = "MACD_{%(fast)s}^{%(slow)s}(%(source)s)"
def macd(/** source */       source = .const(1.),
         /** long period */  slow = 26.0,
         /** short period */ fast = 12.0) : macd

def Source(x = macd()) : IObservable[Float]
def Slow(x = macd()) : Float
def Fast(x = macd()) : Float

def Value(x = macd())
    = x~>Source~>EW(2./(x~>Fast+1))~>Avg - x~>Source~>EW(2./(x~>Slow+1))~>Avg

@label = "Signal^{%(timeframe)s}_{%(step)s}(%(x)s)"
def Signal(x = macd(), timeframe = 9.0, step = 1.0)
    = x~>Value~>OnEveryDt(x~>Step)~>EW(2/(x~>Timeframe+1))~>Avg

@label = "Histogram^{%(timeframe)s}_{%(step)s}(%(x)s)"
def Histogram(x = macd(), timeframe = 9.0, step = 1.0)
    = x~>Value - x~>Signal(x~>Timeframe, x~>Step)
```

Class Inheritance

Classes derive fields and methods from base classes.

Methods are treated as "virtual" to stimulate code re-use

```
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
}
```

Packages and Attributes

Packages are used to group functions and types. They can be nested. Attributes are inherited from enclosing package. Anonymous packages are used to assign same attributes to a group of functions without introducing a new name scope.

```
@X = "X"
@Y = "Y"
package A.B {
    @X = "Xa"
    def f() => Float

    @X = "Xb"
    package {
        def g() => Float
        def h() => Float
    }
}
```

In this sample `.A.B.f` will have attributes `X == "Xa", Y == "Y"` and `.A.B.g` and `.A.B.h` will have attributes `X == "Xb", Y == "Y"`

Example: Relative Strength Index

```
@label = "Ups_{%(timeframe)s}(%(source)s)"
def UpMovements(source = const (1.), timeframe = 10.0)
    = Max(0.0, source - source~>Lagged(timeframe))

@label = "Downs_{%(timeframe)s}(%(source)s)"
def DownMovements(source = const (1.), timeframe = 10.0)
    = Max(0.0, source~>Lagged(timeframe) - source)

@label = "RSI_{%(timeframe)s}^{%(alpha)s}(%(source)s)"
type RSI(/** observable data source */   source      = .const (1.),
         /** lag size */                 timeframe   = 10.0,
         /** alpha parameter for EWMA */ alpha       = 0.015)
{
    def Raw =
        source~>UpMovements   (timeframe)~>EW(alpha)~>Avg /
        source~>DownMovements(timeframe)~>EW(alpha)~>Avg

    def Value = 100.0 - 100.0 / (1.0 + Raw)
}
```

Relative Strength Index Strategy

```
abstract type DesiredPositionStrategy
{
    def Position
        = DesiredPosition - trader~>Position - trader~>PendingVolume

    def Strategy(orderFactory = order.signedVolume.MarketSigned())
        = (orderFactory(Position))~>Strategy
}

type RSI_linear(
        alpha     = 1./14.,
        k         = .const(-0.04),
        timeframe = 1.,
        trader    = .trader.SingleProxy()) : DesiredPositionStrategy
{
    def DesiredPosition
        = (50. - trader~>Orderbook~>MidPrice
                                    ~>RSI(timeframe, alpha)~>Value
                                    ~>OnEveryDt(1.0)) * k
}
```

Bollinger Bands Strategy

```
/**
 *  Strategy believing that trader position should be proportional
 *  to the relative standard deviation of its price
 */
type Bollinger_linear(
            /** alpha parameter for exponentially weighted
              * moving everage and variance */
            alpha    = 0.15,
            /** observable scaling function that maps
              * relative deviation to desired position */
            k        = .const(0.5),
            /** trader in question */
            trader   = .trader.SingleProxy()) : DesiredPositionStrategy
{
    def DesiredPosition = trader~>Orderbook~>MidPrice
                                        ~>EW(alpha)~>RelStdDev
                                        ~>OnEveryDt(1.0) * k
}
```

Side Strategies

```
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)
}

type Noise(side_distribution = math.random.uniform(0., 1.)) : SideStrategy
{
    def Side = if side_distribution > 0.5 then side.Sell() else side.Buy()
}
```

Signal Strategy I

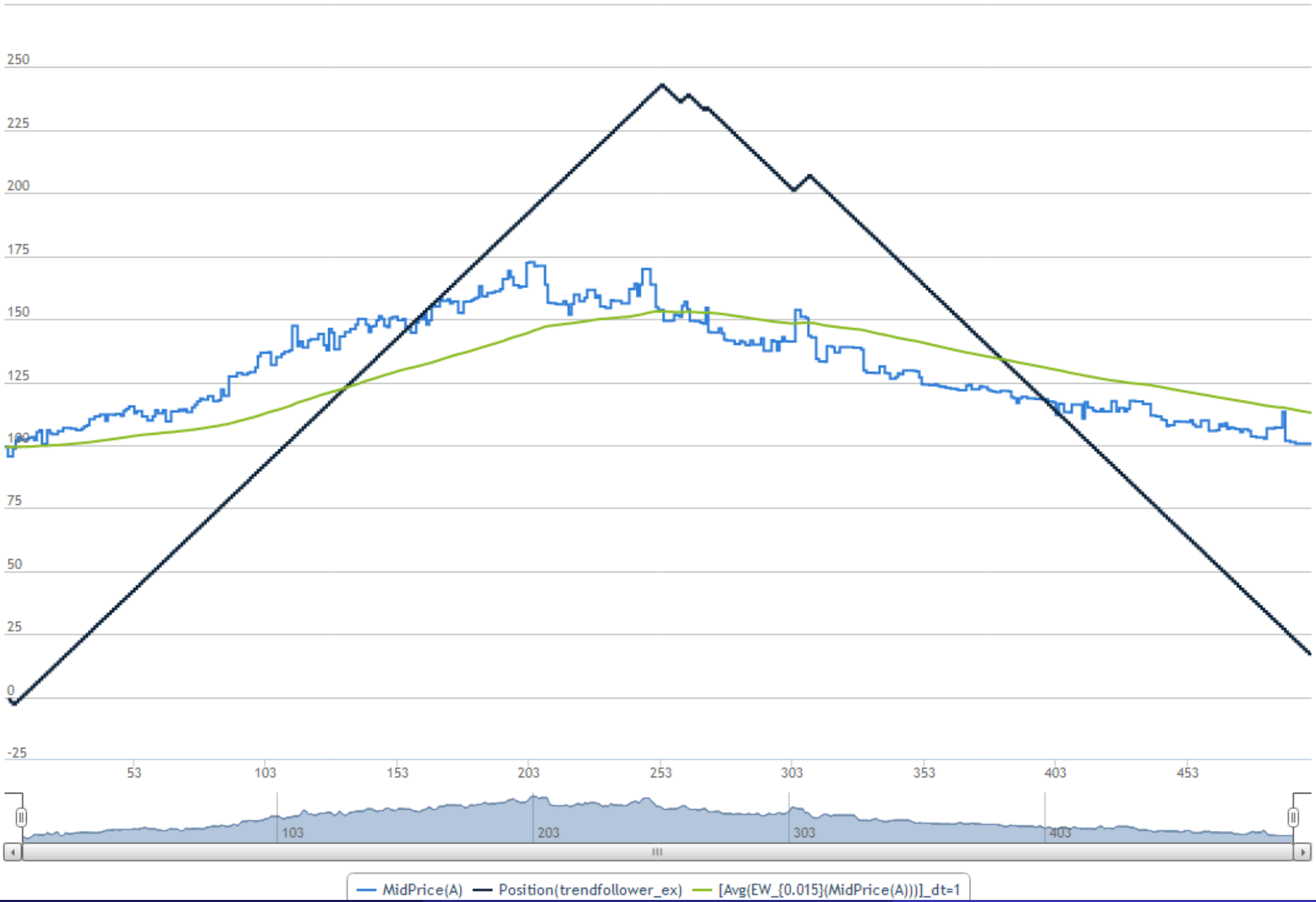
```
/**
 * Signal strategy listens to some discrete signal
 * and when the signal becomes more than some threshold the strategy starts to buy.
 * When the signal gets lower than -threshold the strategy starts to sell.
 */
type Signal(/** signal to be listened to */
            source       = .constant(0.),
            /** threshold when the trader starts to act */
            threshold    = 0.7) : SideStrategy
{
    def S_Side =
        if source >   threshold then side.Buy()  else
        if source < 0-threshold then side.Sell() else
                                     side.Nothing()

    def Side = S_Side
}
```

Signal Strategy II

Trend Follower Strategy I

```
/**
 * Trend follower can be considered as a sort of a signal strategy
 * where the *signal* is a trend of the asset.
 * Under trend we understand
 * the first derivative of some moving average of asset prices.
 * If the derivative is positive, the trader buys; if negative - it sells.
 * Since moving average is a continuously changing signal, we check its
 * derivative at moments of time given by *eventGen*.
 */
type TrendFollower(
        /** parameter |alpha| for exponentially weighted moving average */
        alpha    = 0.15,
        /** threshold when the trader starts to act */
        threshold    = 0.,
        /** asset in question */
        book = .orderbook.OfTrader()) : SideStrategy
{
    def Side = (book~>MidPrice
                    ~>EW(alpha)~>Avg
                    ~>Derivative)
                    ~>Signal(threshold)~>S_Side
}
```

Trend Follower Strategy II



Crossing Averages Strategy I

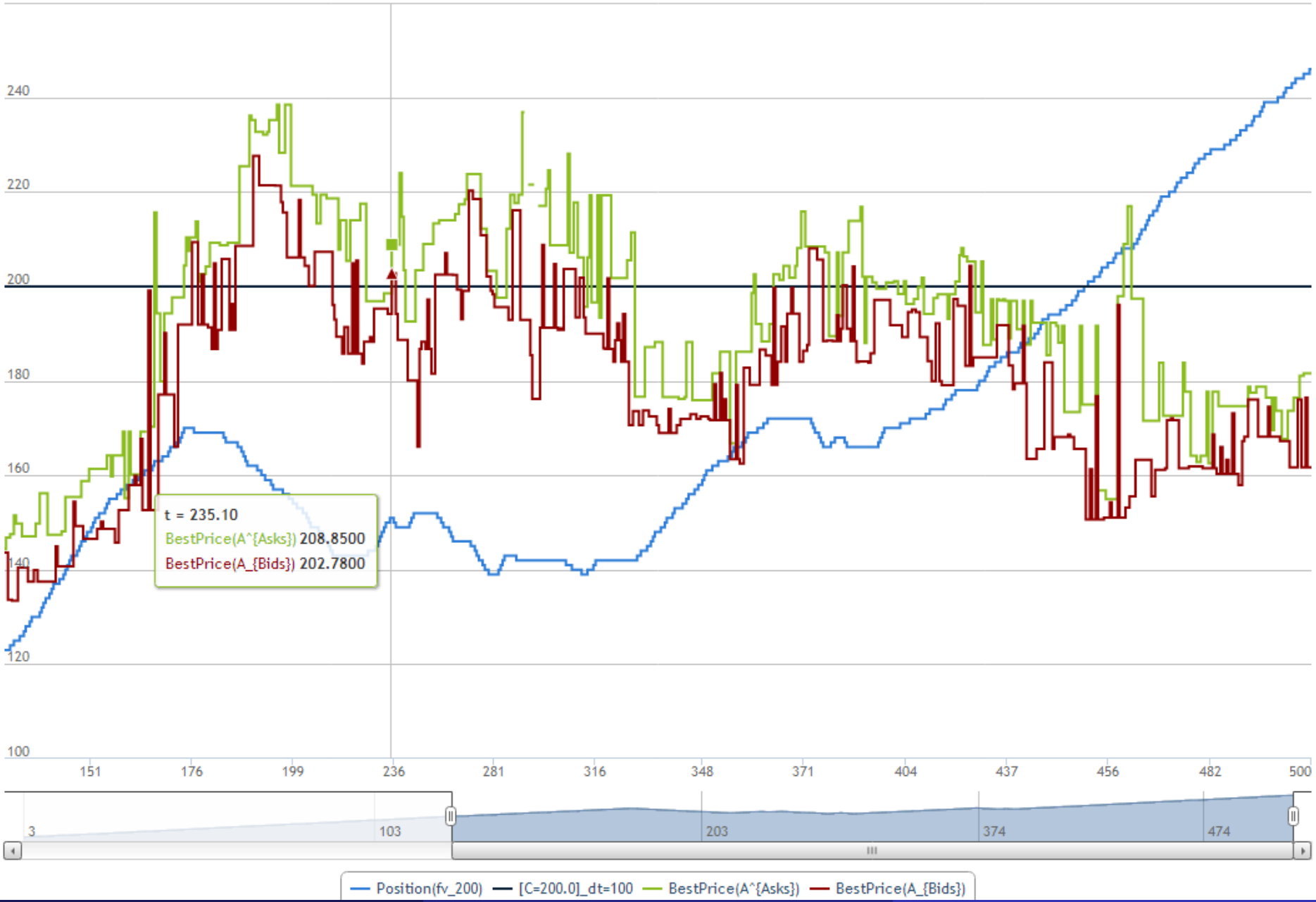
```
/**
  * Two averages strategy compares two averages of price of the same asset but
  * with different parameters ('slow' and 'fast' averages) and when
  * the first is greater than the second one it buys,
  * when the first is lower than the second one it sells
  */
type CrossingAverages(
        /** 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 when the trader starts to act */
        threshold    = 0.,
        /** asset in question */
        book = .orderbook.OfTrader()) : SideStrategy
{
    def Side = (book~>MidPrice~>EW(alpha_1)~>Avg -
                book~>MidPrice~>EW(alpha_2)~>Avg)
                    ~>Signal(threshold)~>S_Side
}
```

Crossing Averages Strategy II

Fundamental Value Strategy I

```
/**
 * Fundamental value strategy believes that an asset should have
 * some specific price (*fundamental value*) and if the current
 * asset price is lower than the fundamental value it starts to buy
 * the asset and if the price is higher it starts to sell the asset.
 */
type FundamentalValue(
    /** observable fundamental value */
    fv      = .constant(200.),
    /** asset in question */
    book    = .orderbook.OfTrader()) : SideStrategy
{
    /**
     * Side function for fundamental value strategy
     */
    def FV_Side
        =   if book~>Bids~>BestPrice > fv then side.Sell() else
            if book~>Asks~>BestPrice < fv then side.Buy()  else
                                               side.Nothing()

    def Side = FV_Side
}
```

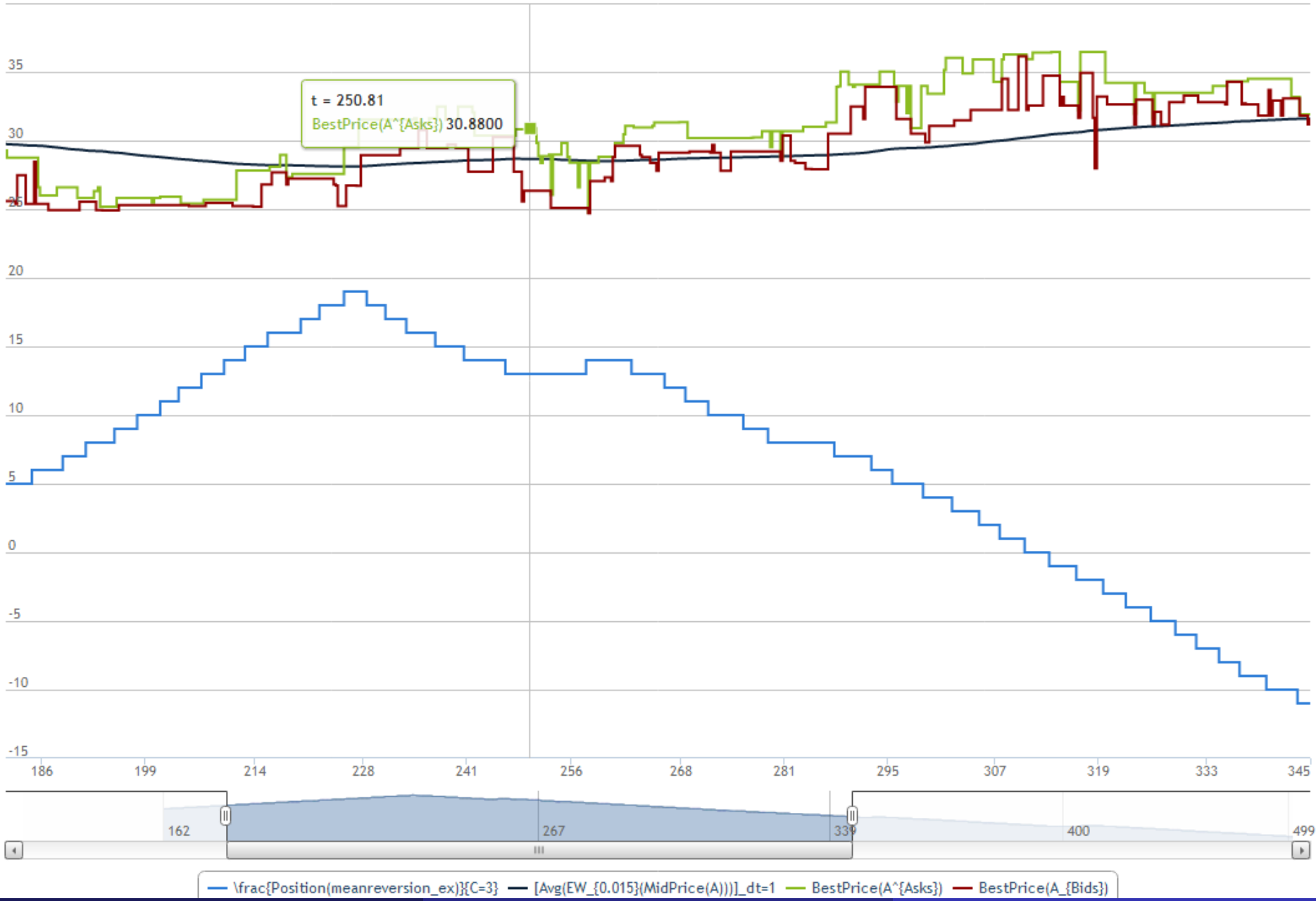
Fundamental Value Strategy II



Mean Reversion Strategy I

```
/**
  * Mean reversion strategy believes that
  * asset price should return to its average value.
  * It estimates this average using some functional and
  * if the current asset price is lower than the average
  * it buys the asset and if the price is higher it sells the asset.
  */
type MeanReversion(
    /** parameter |alpha| for exponentially weighted moving average */
    alpha = 0.015,
    /** asset in question */
    book = orderbook.OfTrader()) : SideStrategy
{
    def Side =  book~>MidPrice
                    ~>EW(alpha)~>Avg
                    ~>FundamentalValue(book)~>FV_Side
}
```

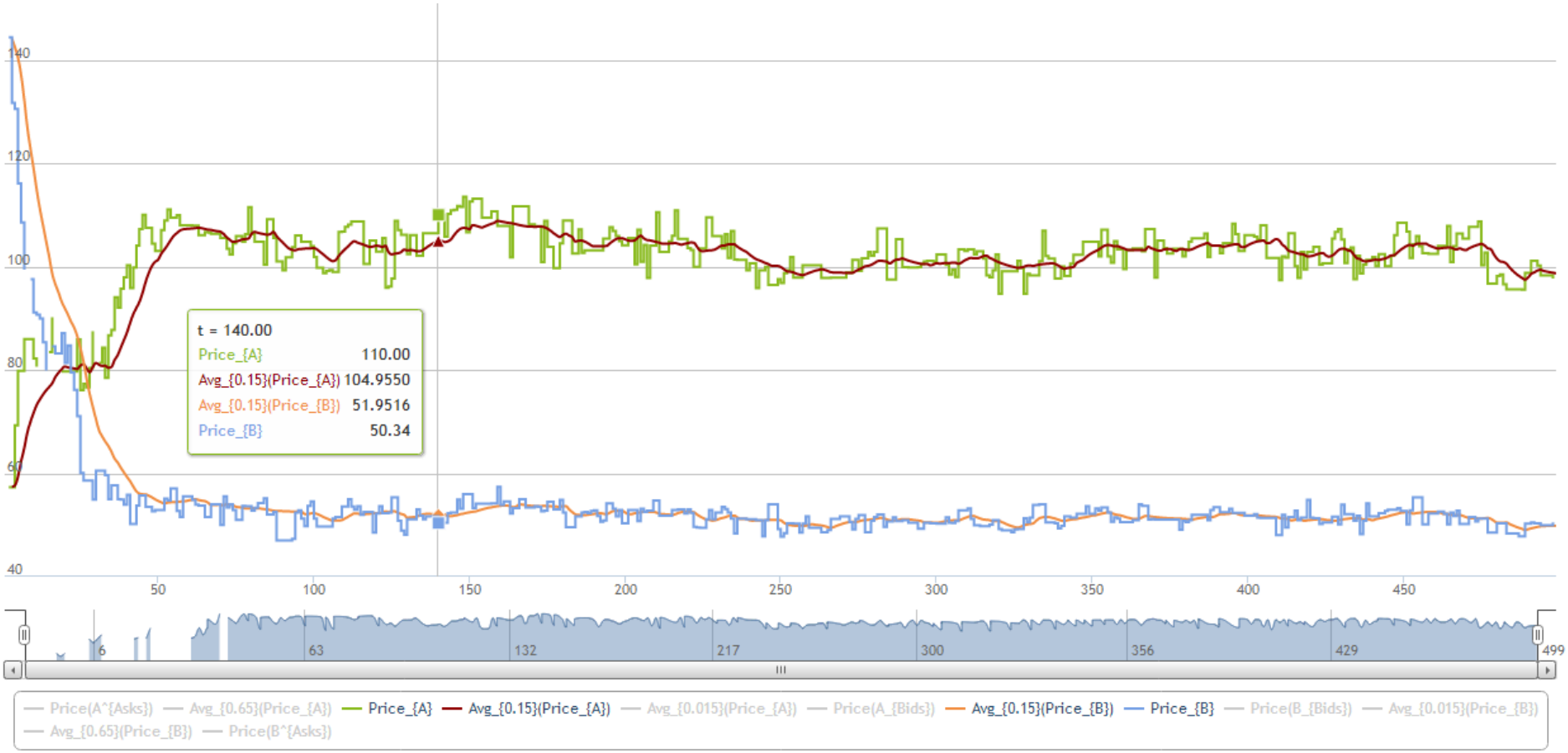
Mean Reversion Strategy II



Pair Trading Strategy I

```
/**
  * Dependent price strategy believes that the fair price of an asset *A*
  * is completely correlated with price of another asset *B* and
  * the following relation should be held: *PriceA* = *kPriceB*,
  * where *k* is some factor. It may be considered as a variety of
  * a fundamental value strategy with the exception that it is invoked
  * every the time price of another asset *B* changes.
  */
type PairTrading(
    /** reference to order book for another asset
      * used to evaluate fair price of our asset */
    bookToDependOn  = .orderbook.OfTrader(),
    /** multiplier to obtain fair asset price from the reference asset price */
    factor          = 1.0,
    /** asset in question */
    book = orderbook.OfTrader()) : SideStrategy
{
    def Side = (bookToDependOn~>MidPrice * factor)
                    ~>FundamentalValue(book)~>FV_Side
}
```

Pair Trading Strategy II



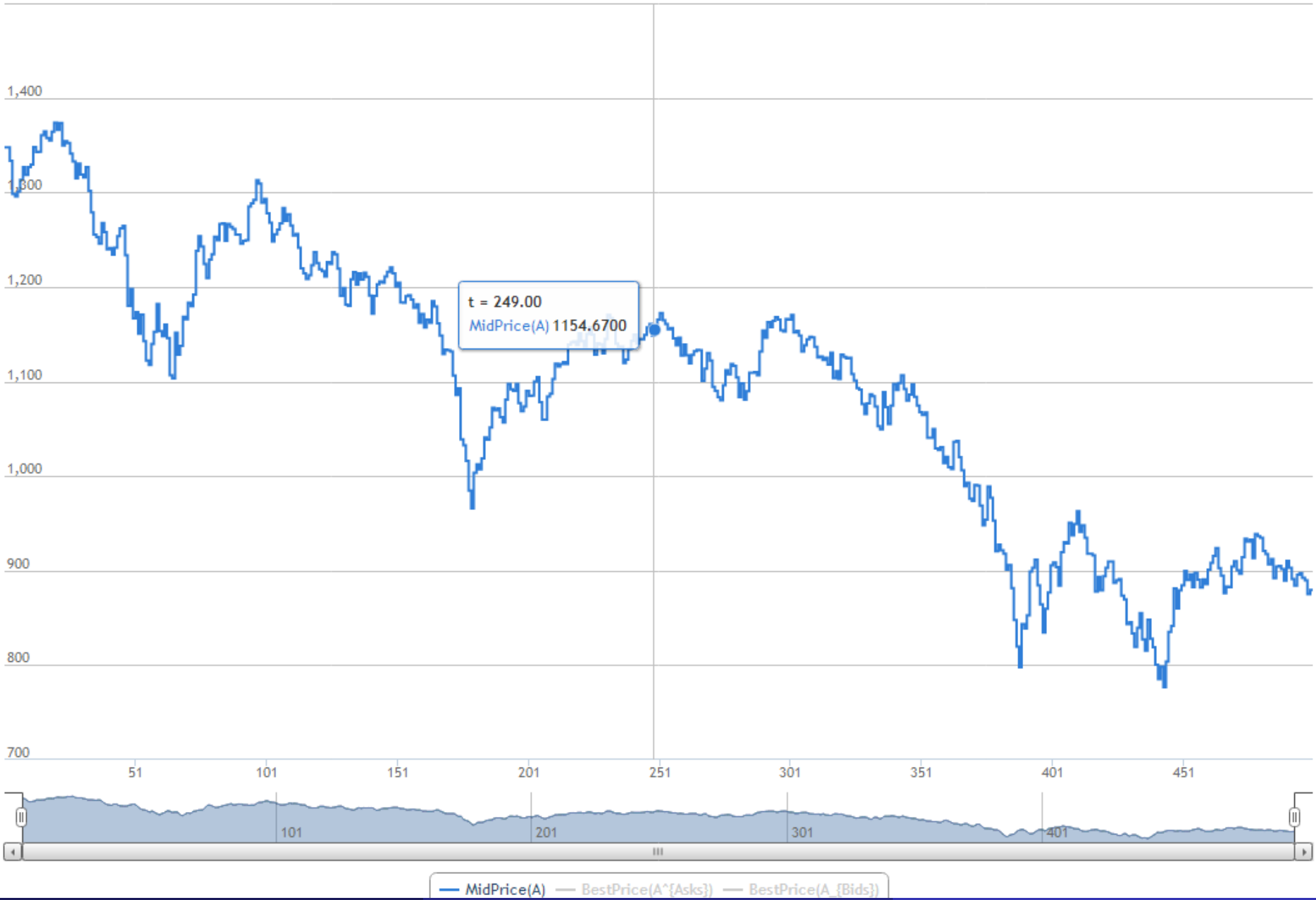
Liquidity Provider Strategy

```
type LiquidityProvider(
        /** initial price which is taken if orderBook is empty */
        initialValue = 100.0,
        /** defines multipliers for current asset price when price of
         order to create is calculated*/
        priceDistr   = .math.random.lognormvariate(0., .1),
        /** asset in question */
        book = .orderbook.OfTrader())
{
    def Price(side = .side.Sell())
        =   book~>Queue(side)~>SafeSidePrice(initialValue) * priceDistr

    def OneSideStrategy(eventGen     = event.Every(math.random.expovariate(1.)),
                        orderFactory = order.side_price.Limit(),
                        side = .side.Sell())
        =   (orderFactory(side, Price(side)))~>Strategy(eventGen)

    def Strategy(eventGen     = event.Every(math.random.expovariate(1.)),
                 orderFactory = order.side_price.Limit())
        =   Combine(OneSideStrategy(eventGen, orderFactory, side.Sell()),
                    OneSideStrategy(eventGen, orderFactory, side.Buy()))
```

Market Data Strategy I

```
type MarketData(/** Ticker of the asset */
                ticker = "^GSPC",
                /** Start date in DD-MM-YYYY format */
                start = "2001-1-1",
                /** End date in DD-MM-YYYY format */
                end = "2010-1-1",
                /** Price difference between orders placed and underlying quotes */
                delta = 1.,
                /** Volume of Buy/Sell orders.
                  * Should be large compared to the volumes of other traders. */
                volume = 1000.)
{
    def OneSide(side = side.Sell(), sign = 1.)

        = order.price.Limit(side, volume*1000)
            ~>FloatingPrice((ticker~>Quote(start, end) + delta*sign)
                                ~>BreaksAtChanges)
            ~>Iceberg (volume)
            ~>Strategy(event.After(0.))

    def TwoSides = Combine(OneSide(side.Sell(), 1.), OneSide(side.Buy(), -1.))
}
```

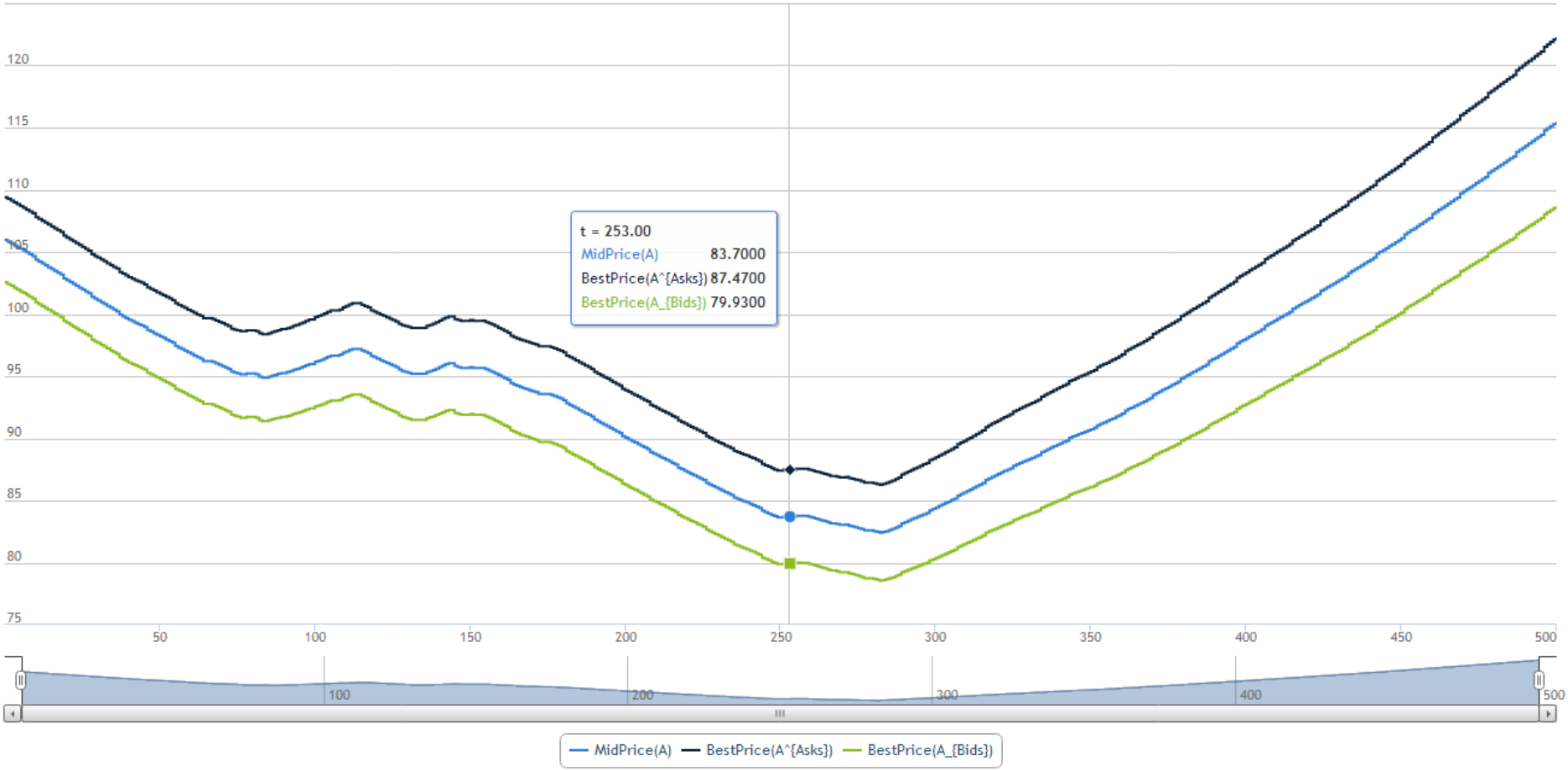
Market Data Strategy II



Market Maker Strategy I

```
type MarketMaker(delta = 1., volume = 20.)
{
    def OneSide(side = side.Sell(), sign = 1.) =

        order.price.Limit(side, volume*1000)
            ~>FloatingPrice(
                (orderbook.OfTrader()
                        ~>Queue(side)
                        ~>SafeSidePrice(100 + delta*sign)
                        / (trader.Position()~>Atan / 1000)~>Exp)
                    ~>OnEveryDt(0.9)~>BreaksAtChanges)
            ~>Iceberg(volume)
            ~>Strategy(event.After(0.))

    def TwoSides = Combine(OneSide(side.Sell(), 1.), OneSide(side.Buy(), -1.))
}
```

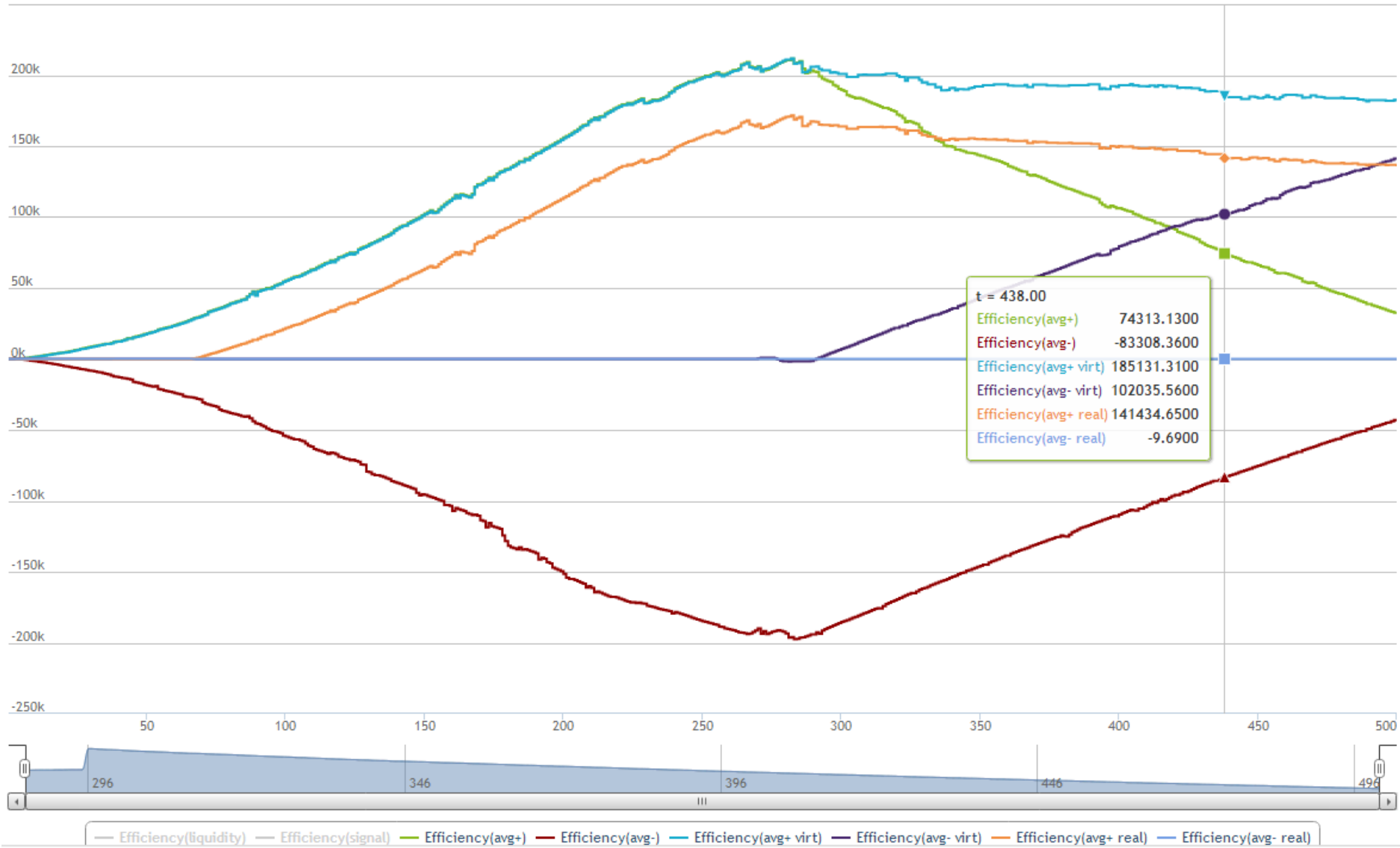
Market Maker Strategy II



Trade-If-Profitable Strategy I

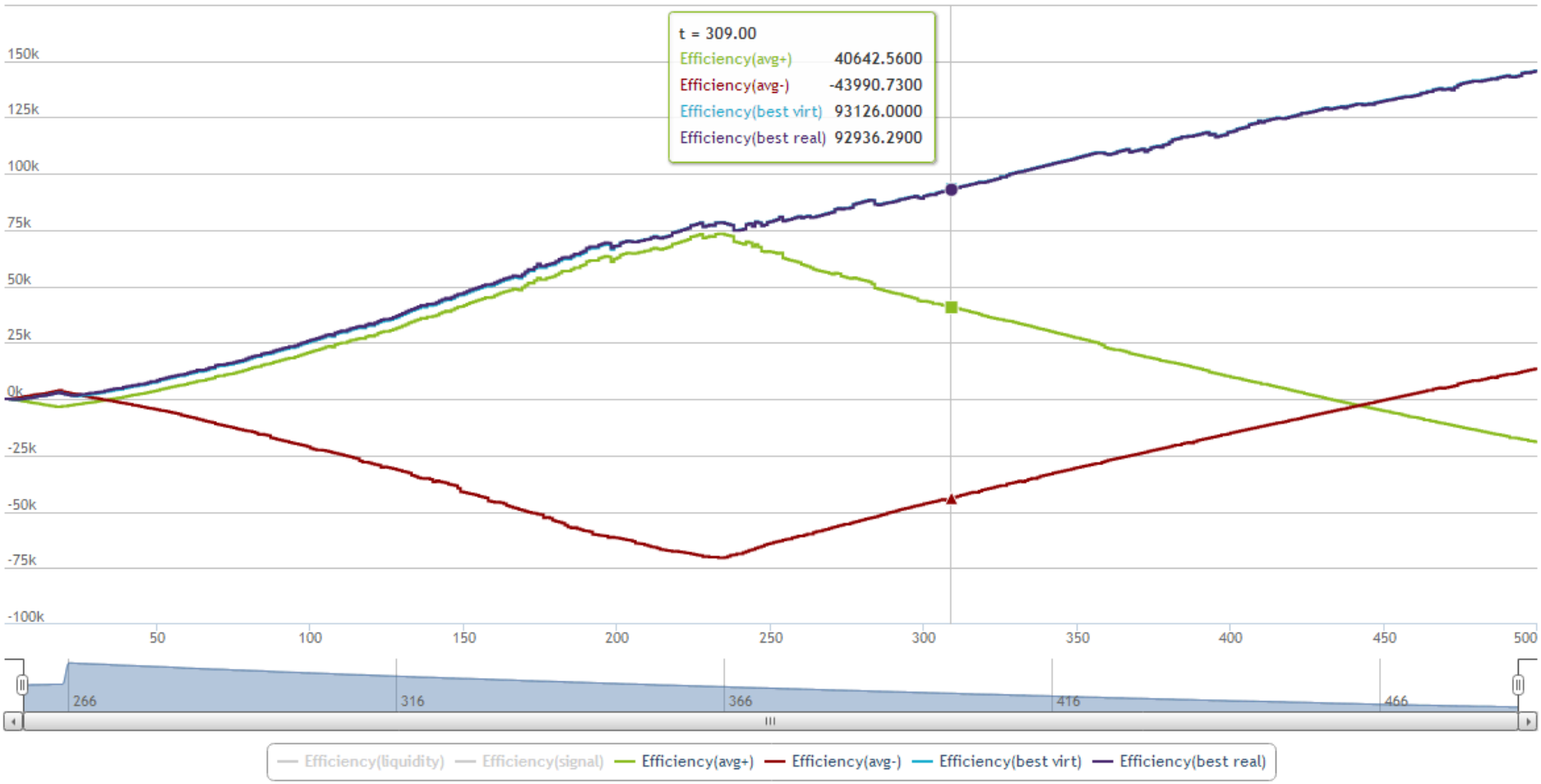
```
/**
 *  Adaptive strategy that evaluates *inner* strategy efficiency
 *  and if it is considered as good, sends orders
 */
def TradeIfProfitable(
        /** wrapped strategy */
        inner       = Empty(),
        /** defines how strategy trades are booked:
          * actually traded amount or virtual market orders are
          * used in order to estimate how the strategy would have traded
          * if all its orders appeared at market */
        account     = account.virtualMarket(),
        /** given a trading account tells
          * should it be considered as effective or not */
        performance = weight.efficiencyTrend())

    =   inner~>Suspendable(performance(account(inner)) >= 0)
```

Trade-If-Profitable Strategy II



Choose-the-best strategy

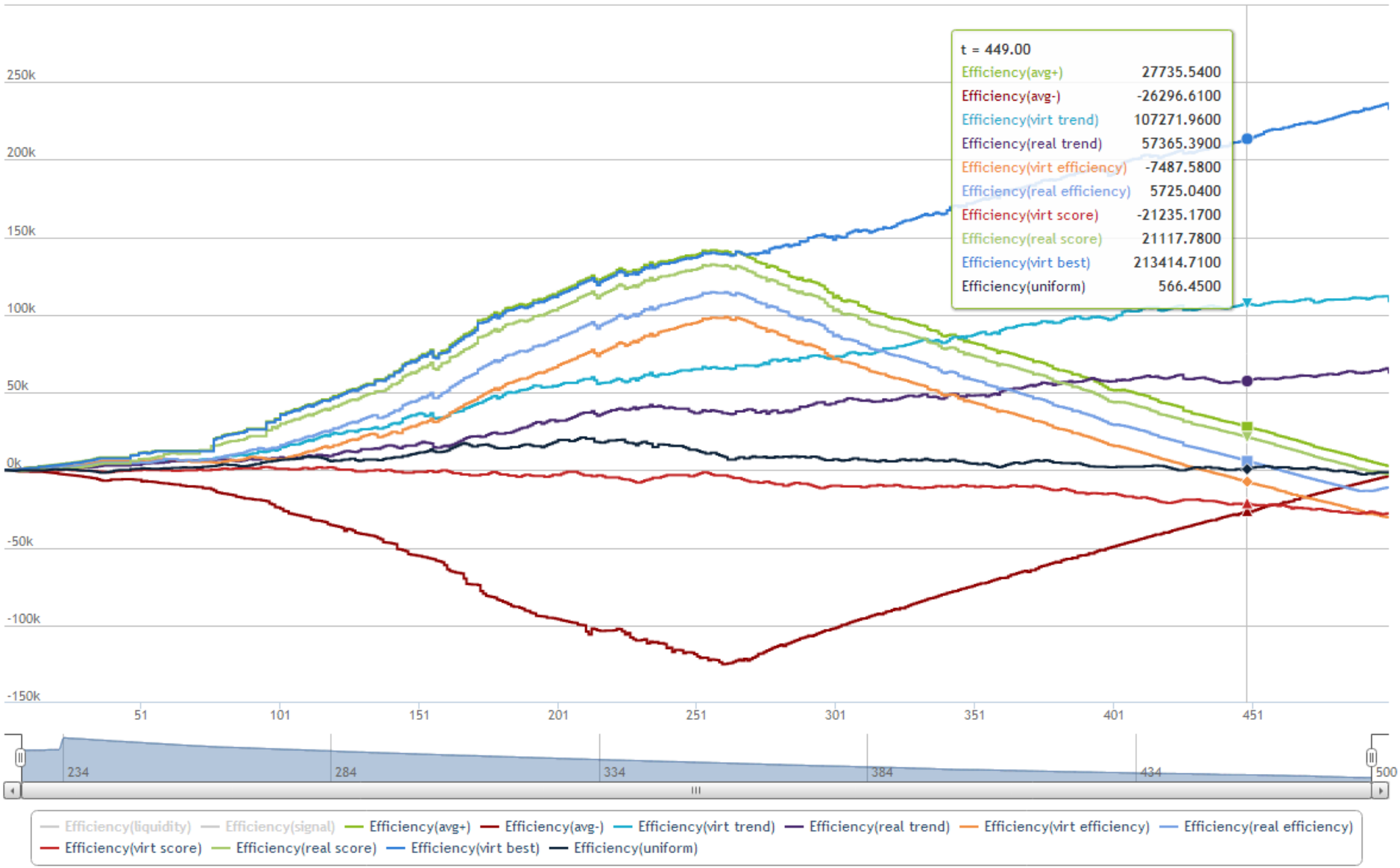
Backtests aggregated strategies and allows to run only to that one who has the best performance. By default, first derivative of a moving average of 'cleared' trader's balance is used to evaluate the efficiency.



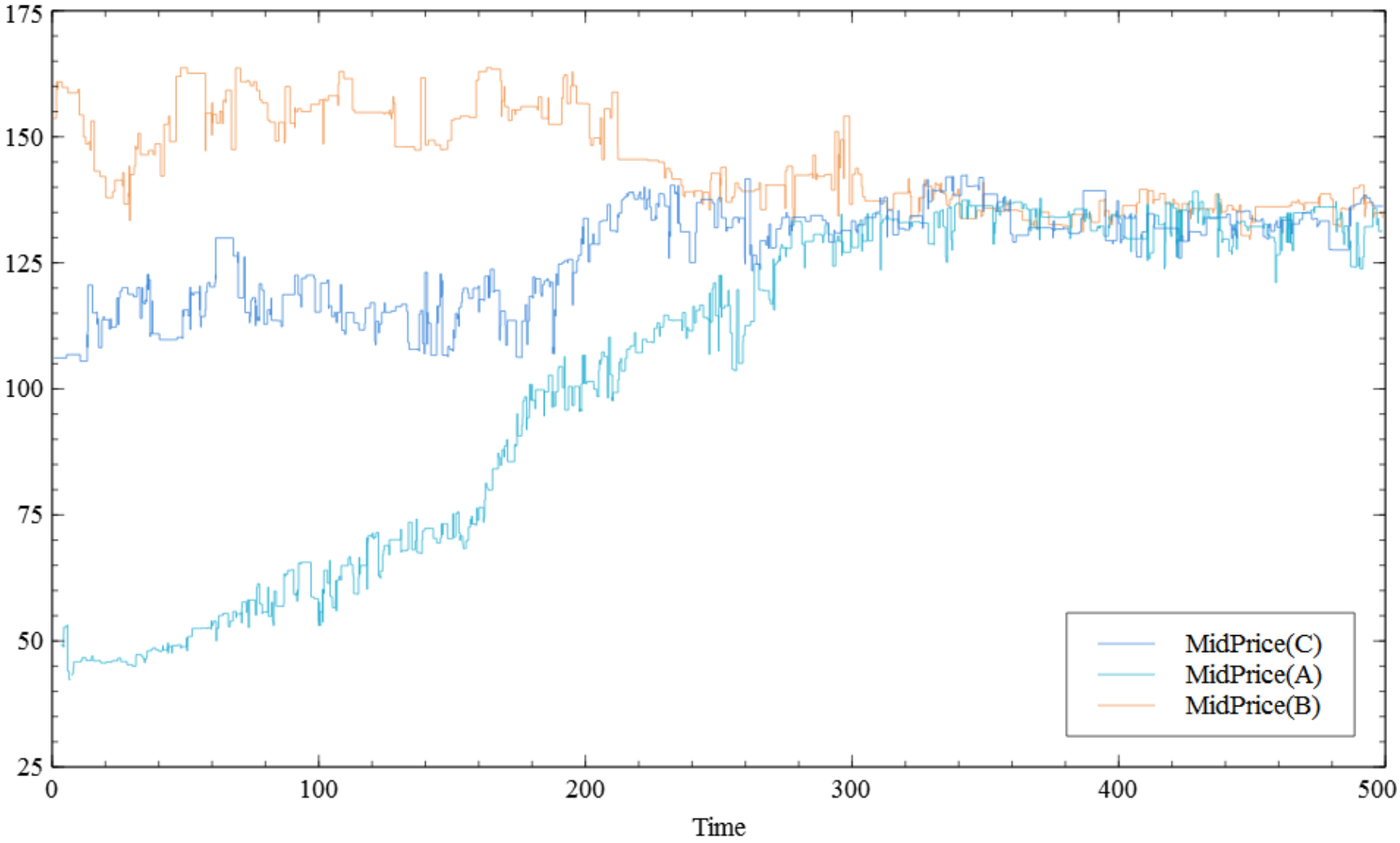
Multiarmed Bandit Strategy I

```
/**
 * A composite strategy initialized with an array of strategies.
 * In some moments of time the efficiency of the strategies is evaluated
 * These efficiencies are mapped into weights using *weight* and *normilizer*
 * functions per every strategy and *corrector* for the whole collection of weights
 * These weights are used to choose randomly a strategy to run for the next quant of time.
 * All other strategies are suspended
 */
@python.intrinsic("strategy.multiarmed_bandit._MultiarmedBandit2_Impl")
def MultiArmedBandit(
        /** original strategies that can be suspended */
        strategies = [Empty()],
        /** function creating a virtual account used
          * to estimate efficiency of the strategy itself */
        account    = account.virtualMarket(),
        /** function estimating is the strategy efficient or not */
        weight     = weight.efficiencyTrend(),
        /** function that maps trader efficiency to its weight
          * that will be used for random choice */
        normalizer = weight.atanPow(),
        /** given array of strategy weights corrects them.
          * for example it may set to 0 all weights except the maximal one */
        corrector  = weight.identityL()) : ISingleAssetStrategy
```

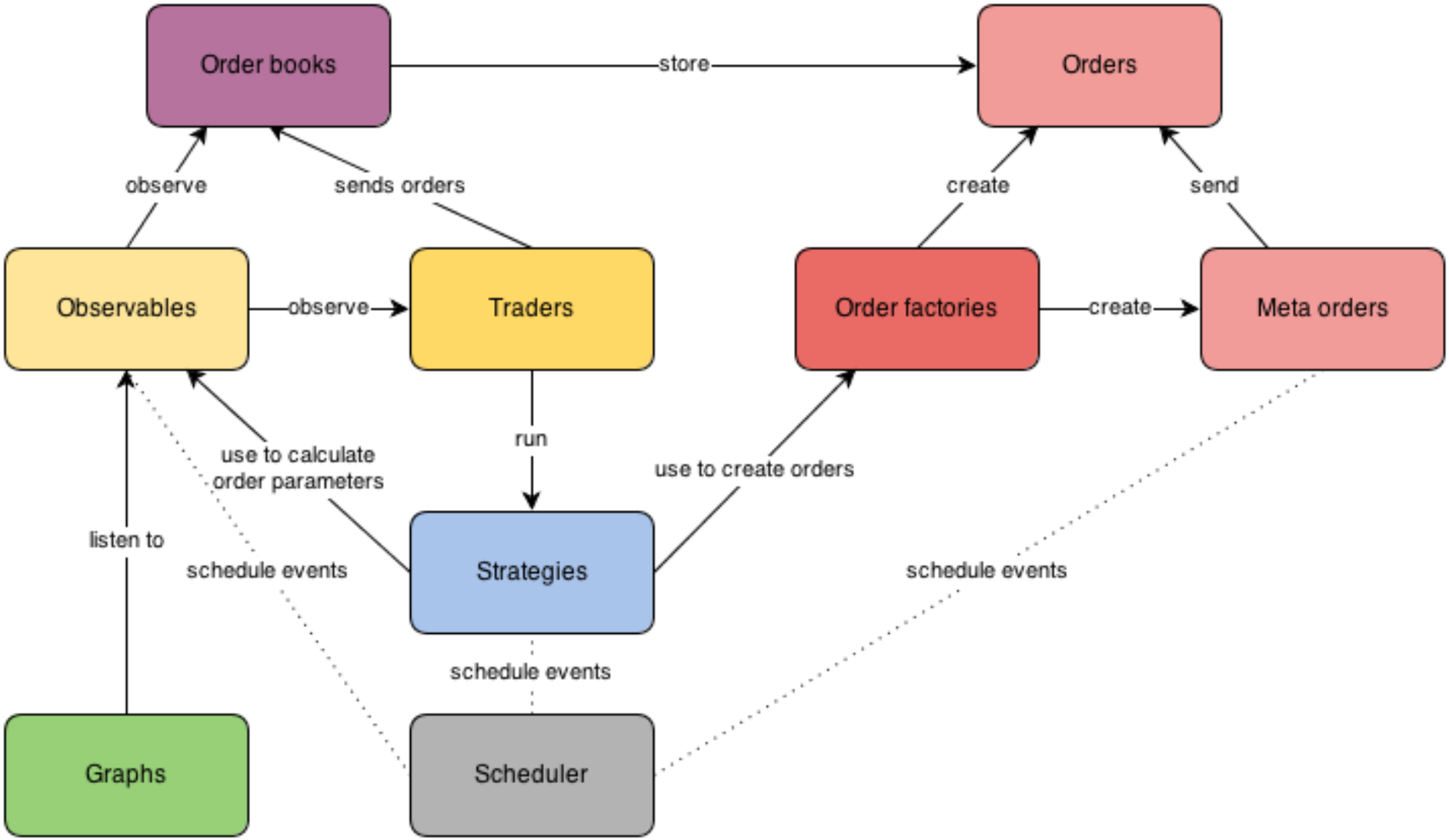
Multiarmed Bandit Strategy II



Arbitrage Strategy



Simulator components



Scheduler

- Main class for every discrete event simulation system.
- Maintains a set of actions to fulfill in future and launches them according their action times: from older ones to newer.

Interface:

- Event scheduling:
  - `schedule(actionTime, handler)`
  - `scheduleAfter(dt, handler)`
- Simulation control:
  - `workTill(limitTime)`
  - `advance(dt)`
  - `reset()`

Events

- `event.Event`. Base class for multicast events.
- `event.Conditional`. Multicast event class with conditional events support. Allows effective notification mechanism for collections of event handler of form `event.GreaterThan(x, listener)` and `event.LessThan(x, listener)`. Internally it keeps two dequeues of event handlers sorted by their trigger values and notifies only those handlers whose trigger conditions are fullfilled.
- `event.GreaterThan(x, listener)`. Fires `listener` only if the value observed is greater than `x`.
- `event.LessThan(x, listener)`. Fires `listener` only if the value observed is less than `x`.

Order book

- Represents a single asset traded in some market (Same asset traded in different markets would be represented by different order books)
- Matches incoming orders
- Stores unfulfilled limit orders in two order queues (`Asks` for sell orders and `Bids` for buy orders)
- Corrects limit order price with respect to tick size
- Imposes order processing fee
- Supports queries about order book structure
- Notifies listeners about trades and price changes

Order book for a remote trader

- Models a trader connected to a market by a communication channel with non-negligible latency
- Introduces delay in information propagation from a trader to an order book and vice versa (so a trader has outdated information about market and orders are sent to the market with a certain delay)
- Assures correct order of messages: older messages always come earlier than newer ones

Basic orders

Orders supported internally by an order book:

- `Market(side, volume)`
- `Limit(side, price, volume)`
- `Cancel(limitOrder)`

Limit and market orders notifies their listeners about all trades they take part in. Factory functions are usually used in order to create orders.

Meta orders I

Follow order interface from trader's perspective (so they can be used instead of basic orders) but behave like a sequence of base orders from an order book point of view.

- `Iceberg(volumeLimit, underlyingFactory)` splits orders created by `underlyingFactory` to pieces with volume less than `volumeLimit` and sends them one by one to an order book ensuring that only one order at time is processed there
- `FloatingPrice(price, underlyingFactory)` listens to an observable `price` and when it changes, cancels its order on the markets and resends it with the new price.
- `Peg(underlyingFactory)` creates a limit-like order with given volume and the most attractive price, sends it to an order book and if the order book best price changes, cancels it and resends with a better price. Implemented via `FloatingPrice`.

Meta orders II

- `WithExpiry(lifetime, underlyingFactory)` sends a limit-like order and after `lifetime` cancels it
- `ImmediateOrCancel(underlyingFactory)` is like `WithExpiry` but with `lifetime` equal to 0 making a limit order to act as a conditional market order
- `StopLoss(lossFactor, underlyingFactory)` order is initialised by an underlying order and a maximal acceptable loss factor. It keeps track of position and balance change induced by trades of the underlying order and if losses from keeping the position exceed certain limit (given by maximum `lossFactor`), the meta order clears its position.

Traders

Single asset traders

- send orders to order books
- bookkeep their position and balance
- run a number of trading strategies
- notify listeners about trades done and orders sent

Single asset traders operate on a single or multiple markets. Multiple asset traders are about to be added.

Observable

Traders and order books provide basic accessors to their current state but don't collect any statistics. In order to do it in an interoperable way a notion of observable value was introduced: it allows to read its current value and notifies listeners about its change. Examples of observables are:

- on traders: position, balance, market value of the portfolio, 'cleared' balance etc.
- on order books: ask/mid/bid price, last trade price, price at volume, volume of orders with price better than given one etc.
- `OnEveryDt(dt, dataSource)` evaluates `dataSource` every `dt` moments of time.

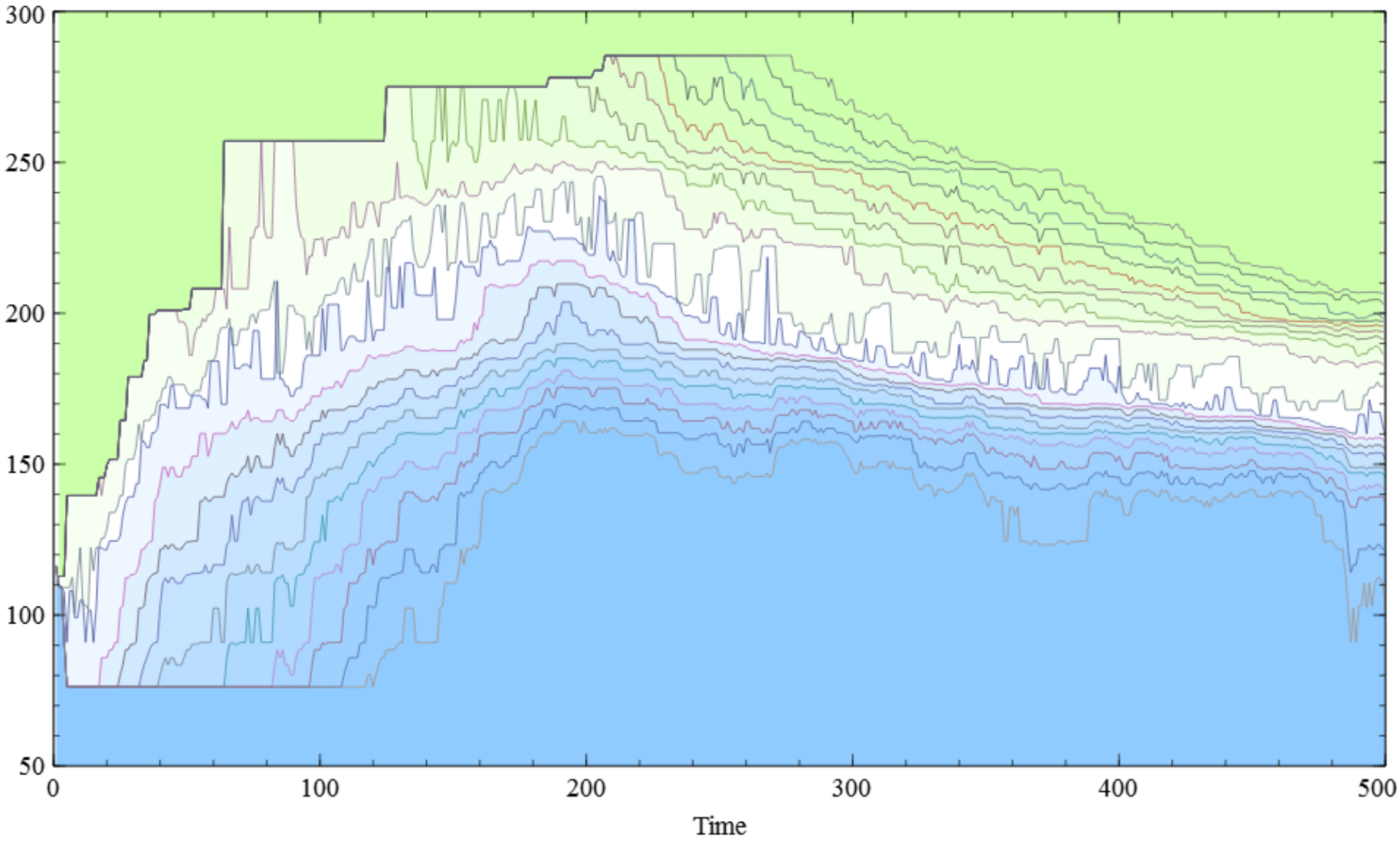
History of an observable can be stored in a `TimeSerie` and rendered later on a graph.

Using Veusz

When developing a new strategy it is reasonable to test it using scripts and visualize results by Veusz

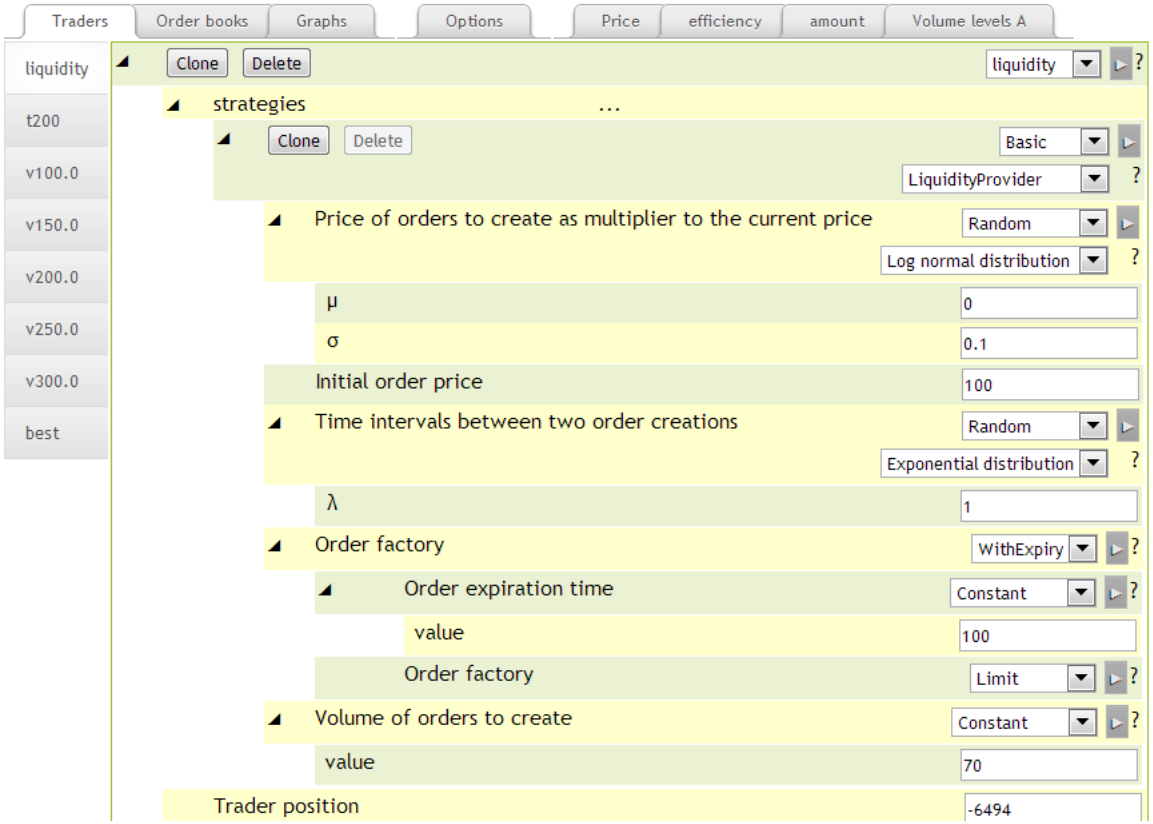
```
@expose("Signal", __name__)
def Signal(ctx):

    const = constant
    linear_signal = math.RandomWalk(initialValue=20,
                                    deltaDistr=const(-.1),
                                    name="20-0.1t")

    return [
        ctx.makeTrader_A(
            strategy.price.LiquidityProvider()
                          .Strategy(event.Every(constant(1.)),
                                    order.side_price.Limit(volume=const(5))),
            "liquidity"),

        ctx.makeTrader_A(strategy.side.Signal(linear_signal)
                                      .Strategy(event.Every(constant(1.)),
                                                order.side.Market(const(1))),
                         "signal"),
```

Rendering graphs by Veusz



Web interface

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



Basic,LiquidityProvider

Liquidity provider is a combination of two LiquidityProviderSide traders with the same parameters but different trading sides.

It has followng parameters:

Order factory

order factory function (default: order.Limit.T)

Initial value

initial price which is taken if orderBook is empty (default: 100)

Time intervals between two order creations

defines intervals of time between order creation (default: exponential distribution with $\lambda = 1$)

Price of orders to create as multiplier to the current price

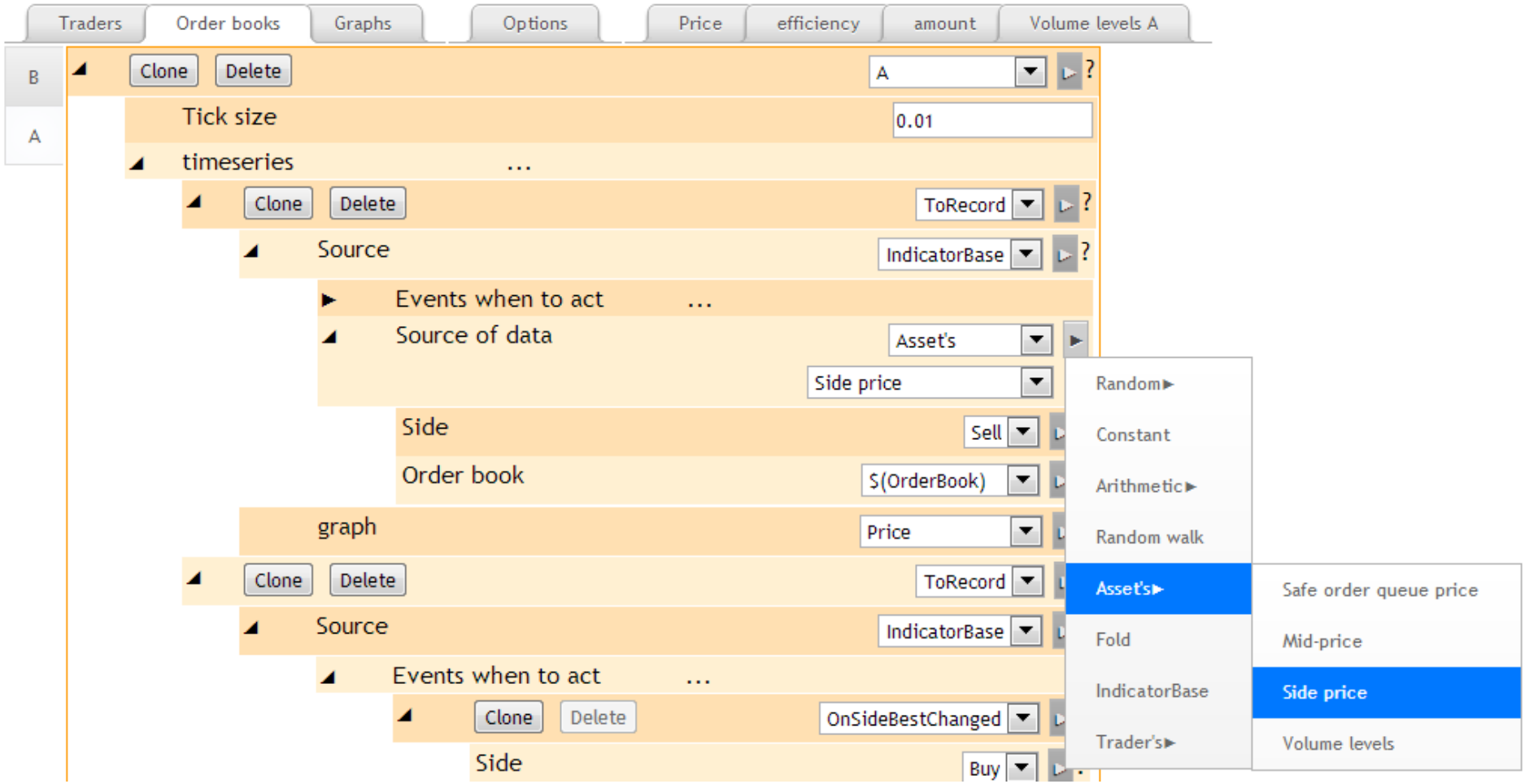
defines multipliers for current asset price when price of order to create is calculated (default: log normal distribution with $\mu = 0$ and $\sigma = 0.1$)

Volume of orders to create

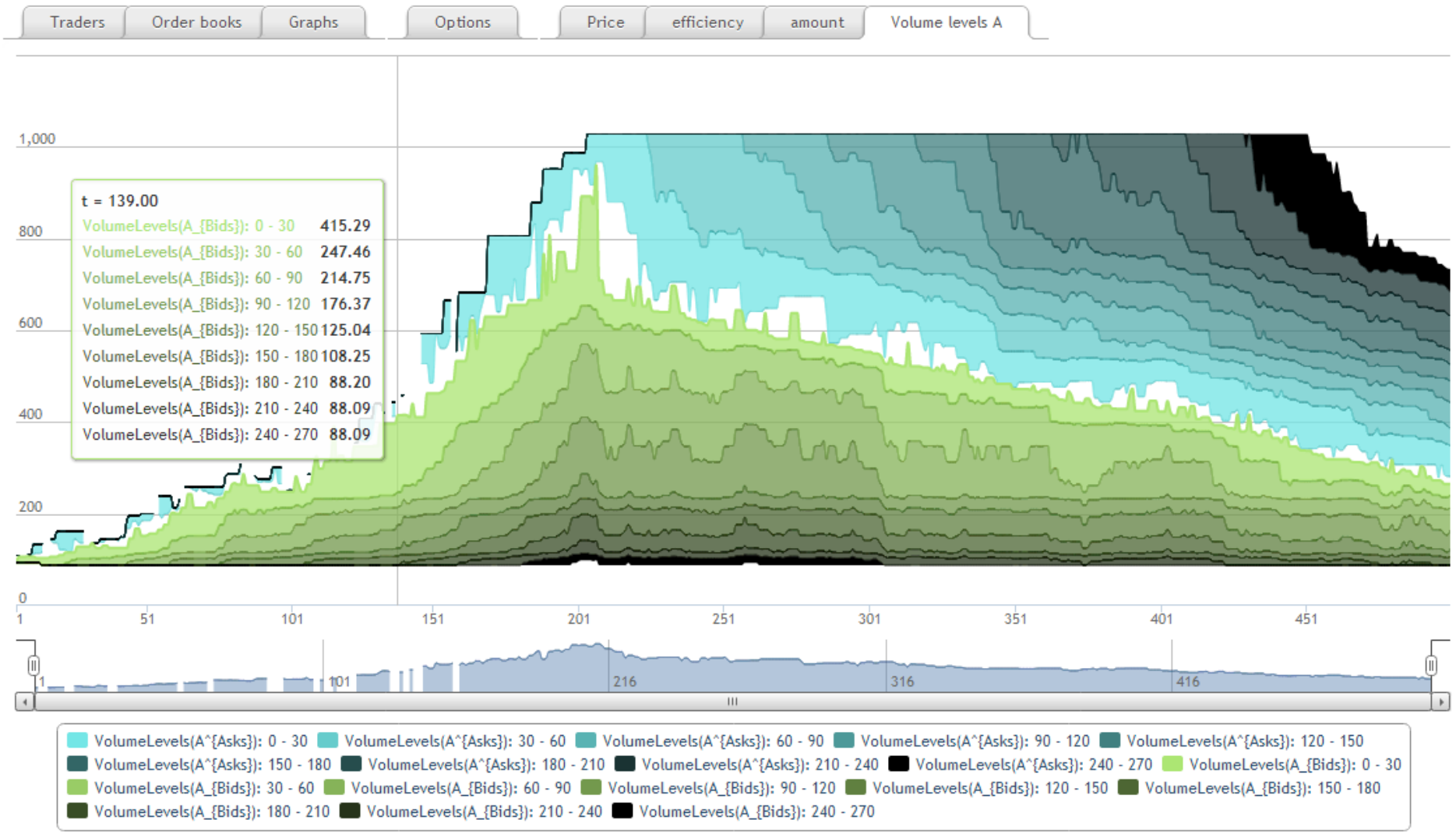
defines volumes of orders to create (default: exponential distribution with $\lambda = 1$)

Time series

Timeseries field of a trader or an order book instructs what data should be collected and rendered on graphs

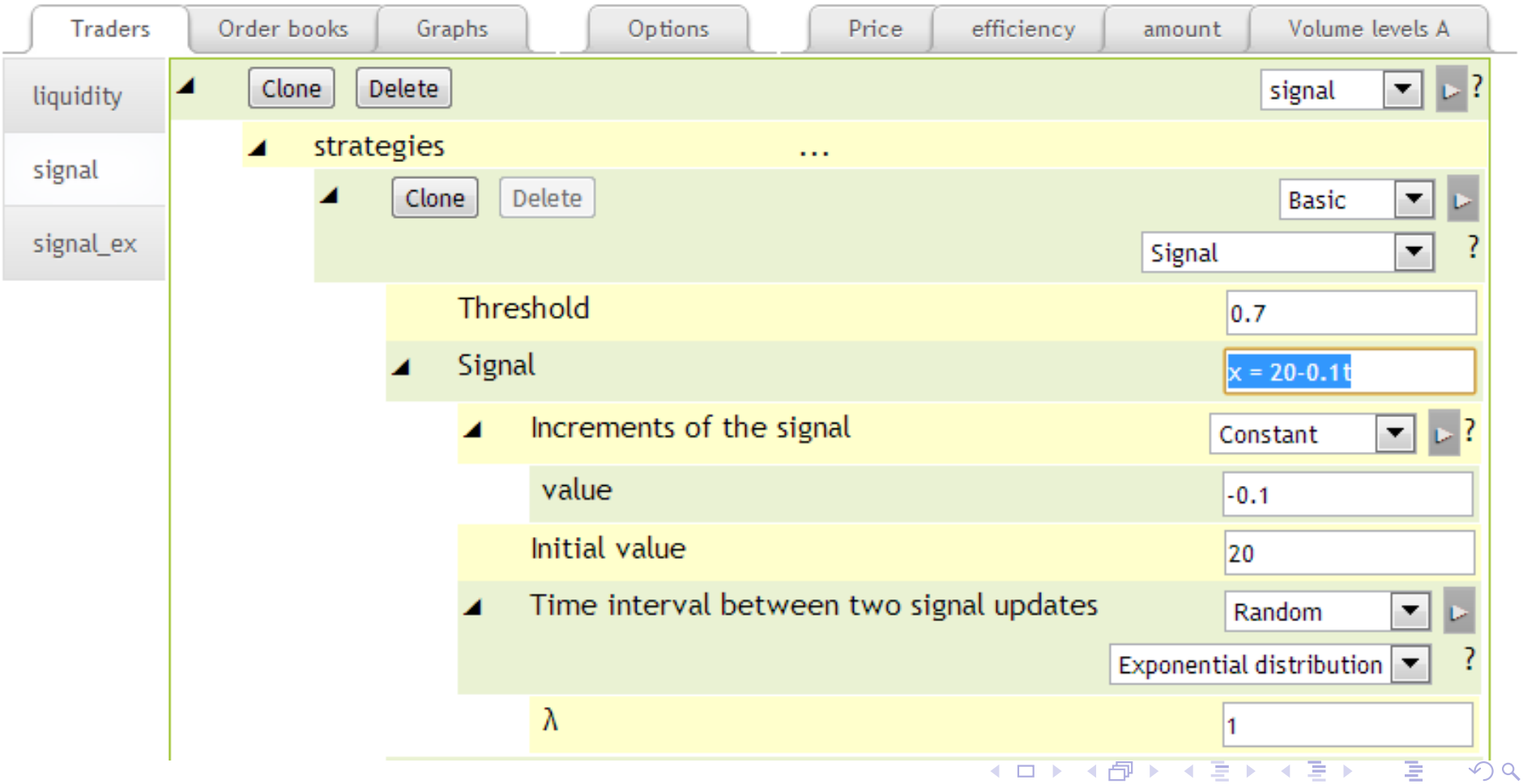


Rendering results



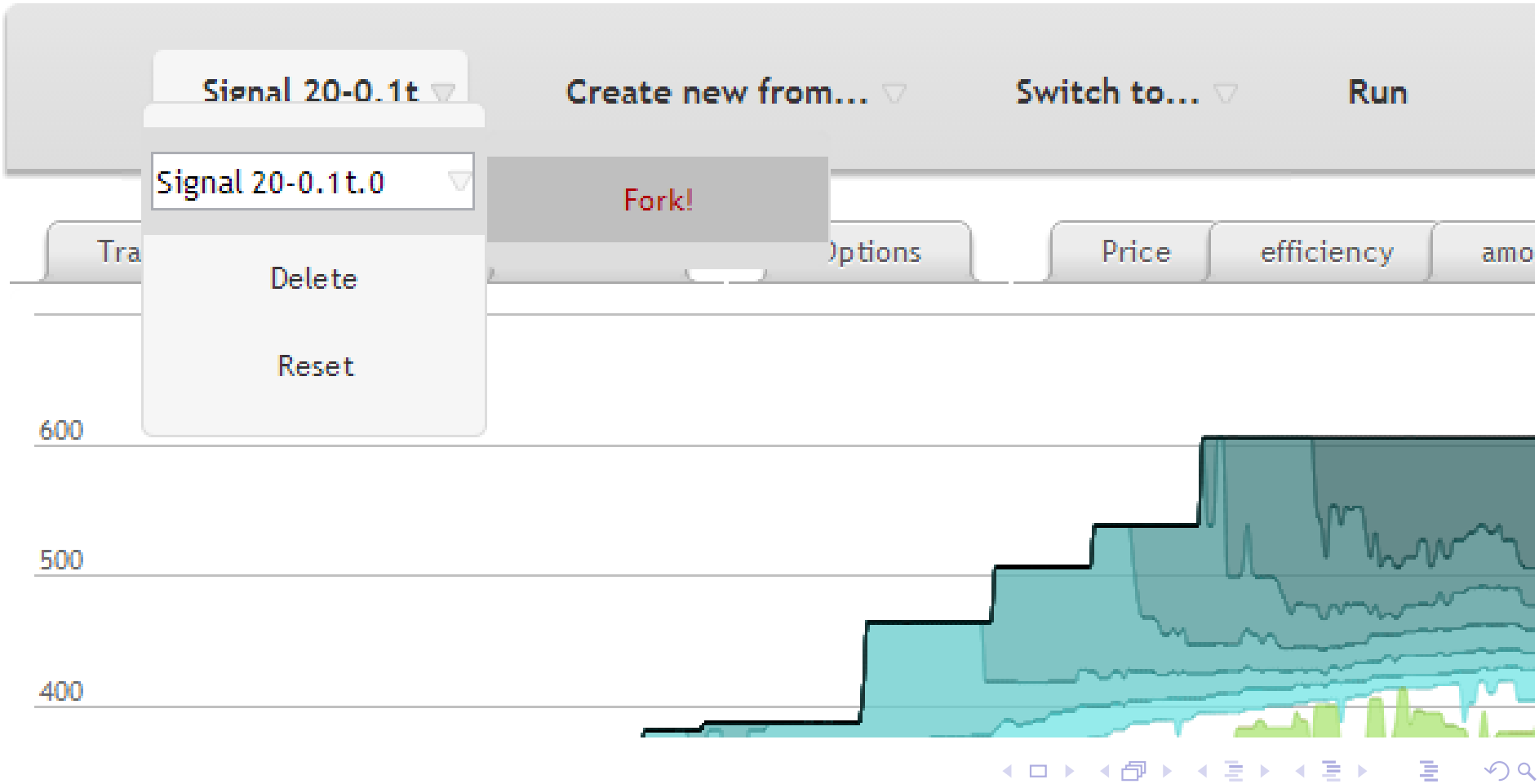
Node aliases

Object tree nodes can be assigned aliases that can be used later to refer to the sub-tree (explicit by-value or by-reference cloning semantics is to be implemented)



Workspaces

Every user (identified by browser cookies) may switch between multiple workspaces. Workspaces can be forked, removed or created from a set of predefined ones.



Installation

- OS supported: Linux, Mac OS X, Windows
- Browsers supported: Chrome, Firefox, Safari, Opera
- Scala 10.2 can be installed using SBT
- Python 2.7
- Python packages can be installed using `pip` or `easyinstall`:
  - Veusz (for graph plotting)
  - Flask and Docutils (to run a Web-server)
  - Blist (sorted collections used by ArbitrageTrader)
  - Pandas (only needed for `observable.Quotes` at the moment)
  - Numpy (only needed for `strategy.MultiArmedBandit` at the moment)
- Source code downloadable from GitHub
- Latest public version of the simulator can be downloaded from here.
- Online documentation can be found here.

Thank you!