2016 IEEE 55th Conference on Decision and Control (CDC)
ARIA Resort & Casino
December 12-14, 2016, Las Vegas, USA

MatTrader: an automated trading and financial data analysis framework for Matlab and Java

Giuseppe C. Calafiore, Luca Poletti, Alessandro Preziosi

Abstract— **This paper introduces MatTrader, a framework for real-time trading, historical data analysis and simulation, aimed at giving researchers and investors an instrument for developing and testing quantitative investment strategies and paradigms. MatTrader is built to grant maximum flexibility and control, providing data structures for representing financial assets and data, methods for trading, managing orders and retrieving historical data, and a set of events for implementing real time investment strategies. It also includes Matlab-specific classes for automatic validation of trading strategies and for feedback-driven trading. The framework is fully documented, open source, and it contains several ready-to-use examples. It is freely available for use by the community, and the Java library allows to easily port the framework to other programming languages.**

I. INTRODUCTION

Trading strategies and automated technical analysis have gained notable interest in recent years, in particular in the field of control theory, where a significant amount of theoretical and algorithmic results have appeared in the literature (references [1] to [6]).

In order to facilitate research in this area, we developed an open source framework, in collaboration with Directa SIMpA (`http://www.directa.it/`), one of the first Italian brokerage companies allowing private investors to trade online on financial markets since 1996.

MatTrader has been developed to provide a flexible tool for researchers and investors to help in the process of development and validation of real-time trading strategies. There are few software packages that allow users to implement real-time strategies and backtest them, but most of them are not open source, and rarely come with a fully programmable interface. Examples of other providers of programmable trading systems are, e.g., `http://www.metaquotes.net`, and `https://www.interactivebrokers.com`. Usually, they use their own programming language and do not provide APIs. MatTrader is fully open source, available at `https://github.com/kalup/MatTrader`, and it has a Java API which allows researchers to extend the framework to cover their needs, e.g., by developing statistical analysis packages in R, collecting historical data in Excel, or performing data mining with RapidMiner. In particular, we provide a Matlab toolbox, relying on this Java core, to give a ready-to-use implementation in a standard scientific computing platform. In the following sections we will refer to the Matlab implementation in the description of the package features and in the examples.

Three main "categories" can be identified in the framework, that aim to cover all the basic needs: a *Data Feed*, a *Trading* interface and an *Historical Feed*. Every method, event or data structure related to the current state of a security falls under the first category; the second one offers categorized methods used for placing an order and the structures necessary to control its current state, while the third one contains the set of methods used for retrieving historical data. The core data structure in MatTrader is an object that models a security, and most of the data structures provided by the framework are derived from this object. The Matlab toolbox provides additional classes, specifically designed to be used in feedback driven strategies and to easily build an automated trading system or a backtesting machine.

In the following chapters we describe the basic functioning of the toolbox, and we show implementation of some simple trading strategies, such as the moving-average crossing strategy, or the Barmish-Iwarere feedback strategy.

II. THE MATTRADER FRAMEWORK

MatTrader provides a modular design approach where multiple clients, with their own scope, can be spawned at the same time. Clients can be though of as distinct portfolios, or different trading machines, implementing their own strategies. The user can manage as many clients as he wishes, identifying them by their user-defined name.

A. Initialization

The framework communicates with the market using a generic Java interface that can be extended to provide access to different brokers. During development we created a custom class, implementing that interface, to communicate with Directa Darwin API[1].

The central manager and core of the framework is the class `MTManager`, which provides methods to handle the lifecycle of a client. In the next snippet of code it is shown how to get a reference to a client identified by the name "myClient" using information provided by Directa.

```
>> manager = MTManager(DirectaAPI);
>> client = manager.getClient('myClient');
```

A client can manage many securities concurrently. In the following, we will use as an example Saipem SpA, a

G.C. Calafiore is with Faculty of Control and Computer Engineering Department, Politecnico di Torino, Italy. Email: giuseppe.calafiore@polito.it, luca.poletti.89@gmail.com, lsnpreziosi@gmail.com.

The authors sincerely thank all the people at Directa SIMpA for their guidance in accessing their information infrastructures, and the continued assistance in interfacing with their financial APIs.

[1]Information about this service are available at `http://www.directa.it/pub2/it/darwin/api.html`

component of the Italian FTSE MIB index, identified by the ticker 'SPM'. At first an object `Ticker` is requested to the client, all data relative to this security is stored within this object and can be accessed using dot-notation. The following snippet shows how to retrieve the International Securities Identification Number (ISIN) and a human-readable description.

```
>> spm = client.getTicker('SPM');
>> spm.ISIN
IT0000068525
>> spm.description
SAIPEM
```

B. Data Feed

Real time information is automatically collected from Directa servers - by means of Darwin API - and stored within the `Ticker` object. In this way, using simple dot-notation, users can access any data related to the security.

```
>> spm.dailyMin
0.3607
>> spm.openPrice
0.3680
>> spm.price
0.3640
>> spm.volume
22546
>> spm.book.bid
3.6490e-01   9.1850e+03
3.6480e-01   5.0450e+04
3.6470e-01   5.0000e+03
3.6460e-01   7.5450e+03
3.6450e-01   1.7045e+04
```

Information is encapsulated within different type of objects to provide maximum flexibility, an example is the `spm.book` call previously shown. This returns a static snapshot of the order book in the time instant in which the call is made; the same call at different times leads to two comparable objects, referencing distinct snapshot of the market.

```
>> bookTime1 = spm.book;
>> pause(5)
>> bookTime2 = spm.book;
>> bookTime1.price
3.6560e-01   1.2000e+04
>> bookTime2.price
3.6550e-01   6.0418e+04
>> % Compare variance of volumes of ask
>> volumeAsk1 = bookTime1.ask;
>> volumeAsk2 = bookTime2.ask;
>> std(volumeAsk1(:,2)) - ...
    std(volumeAsk2(:,2))
2.3989e+04
```

C. Trading

Trading orders are issued directly to the `Ticker` object. Buy, sell and stop-loss orders can be placed by the mean of specific methods. Any call to those methods returns an `Order` object allowing users to monitor their orders programmatically. The following snippet shows an example on how to insert an order to buy 5 shares at a price of 0.3646, place a stop loss order triggered at 0.3595 with sell price of 0.3590, revoking it and finally sell 5 shares.

```
>> spm.buy(0.3646,5);
>> order_1 = spm.stopLoss(...
   0.3595,5,0.3590);
>> spm.revoke(order_1);
>> spm.sell(0.3639,5);
```

D. Historical data

It is possible to access historical data of securities to perform price evolution analysis and back-testing. This framework provides different data structures to access historical data, in particular an `HistoricalTable` object containing the most relevant data. In the following snippet, data from the past 5 days is requested, with 1 second sampling interval, then a check on the availability of data is made, the average of minimum and maximum price and the coefficient of variation of volumes are computed, and finally data are written to a .csv file.

```
>> h_data = spm.hist(5,'s');
>> h_data.isReady
1
>> mean(h_data.minPrice)
0.3933
>> mean(h_data.maxPrice)
0.3937
>> std(h_data.volume) / ...
    mean(h_data.volume)
2.3761
>> csvTable('spm_5.csv',h_data.table);
```

E. Event sample

In a real-time execution environment trading events are essential. In our framework, this is obtained using callbacks and two main objects: `EventManagerCom` and `EventHandler`. The first object is a generic event manager implemented in Java, the second one is an object specific to the Matlab environment that acts as middleware. The following snippet shows how to register a simple callback that shows prices as soon as they are received.

```
>> priceEventManager = ...
        spm.onPriceEventManager;
>> onPriceReceived = EventHandler( ...
        priceEventManager);
>> onPriceReceived.setCallback( ...
        @(~,event) disp(event.price));
>> pause(5);
0.3643
0.3643
0.3641
0.3644
>> onPriceReceived.setNullCallback;
```

F. Examples

Together with the framework, some tools and examples are provided as guidelines for developers. Two examples are shown next. In Figure 1 the prices and volumes of the A2A SpA security are plotted, together with their moving averages at 40 and 80 intervals; prices are sampled at one second intervals, while volumes have been aggregated over 300 seconds. This plot is the result of the example function `meanPlotDaily`.

```
>> meanPlotDaily('A2A',40,80,1,300);
```

Another example is given by the function `volatility(ticker, days)` provided with the toolbox, which returns a measure of the variation of prices over a given period and an index I_{acc} showing how much this measure is consistent. Historical volatility is a statistical

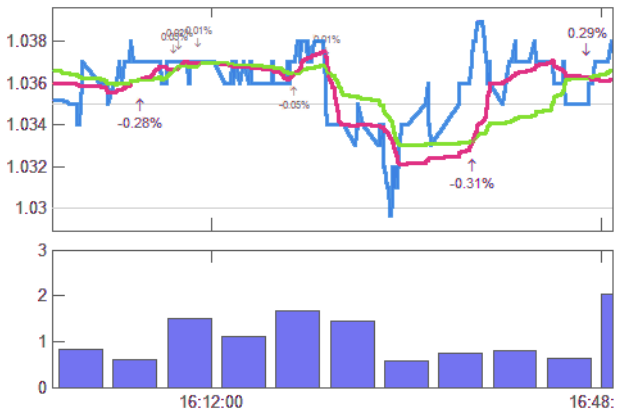

Fig. 1. Plot of price, volume and moving averages of A2A

measure describing how much prices varies in a specified interval of time and it is commonly used in the process of risk evaluation.

```
>> [sigma, acc]= volatility(spm,30)
sigma = 0.0518
acc = 1.5048
```

In this function, volatility is computed as the standard deviation σ of prices and is considered accurate if prices are distributed like a Normal distribution, thus I_{acc} is a function of the Bhattacharya distance between the distribution of prices and a discretized Normal.

III. Automatic Testing

Since one of the main goals of MatTrader is to help researchers develop and test new methodologies and trading systems, the package is shipped with two classes that simplify those tasks. These are abstract Trading Machines $\mathcal{M}$ providing a high level of abstraction and automatically taking care of object allocation, order handling, event subscriptions and creation of basic plots. Trading Machines are initialized with a ticker $\mathcal{T}$, and an handler to a Matlab function $\mathcal{A}$ implementing an algorithm for the trading strategy. $\mathcal{A}$ takes as parameters price, volume, timestamp and the state variables of the system and returns the triplet of buy/sell price, desired number of shares, and updated state variables. $\mathcal{A}$ is agnostic to weather it is operating in real time or on historical data. The skeleton of an algorithm is as follows:

```
% definition of the algorithm
function ...
  [tradeVolume, tradePrice, newState] = ...
    controlFunction( ...
        typeEvent, ...
        inputPrice, ...
        inputVolume, ...
        inputTimestamp, ...
        state, ...
        machine )
    ... % body of the function
end
```

This structure can be thought as a feedback system, where inputs are generated by the market as prices and trade events, outputs are sent to the market as orders, the controller is the algorithm $\mathcal{A}$, the system component is represented by the machine $\mathcal{M}$, and the feedback f consist in the set of variables containing the state of the system (see Figure 2).

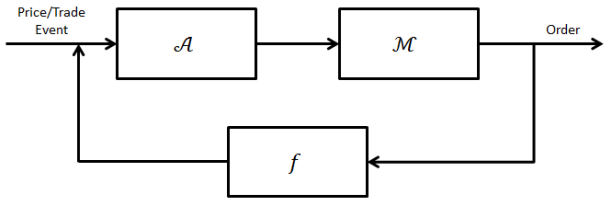

Fig. 2. A feedback-like structure: $\mathcal{A}$ is the algorithm, $\mathcal{M}$ is the trading machine and f is the state.

A. Backtesting

The first step for evaluating the performance of a strategy is to evaluate how it would have performed if applied to past scenarios. To perform such test, the backtesting machine expects as inputs a ticker, the period of analysis, an handler to the function implementing the strategy, and optionally an initial state. Historical data is automatically requested from Directa servers.[2] and, one price at a time, they are passed to the function along with the timestamp of the historical event; in such a way the function is always agnostic to the machine which is feeding data. The function may eventually return the intention to submit an order; the machine will check if in that historical instant the order may be instantly resolved, else the order will be queued. While the historical data are parsed and new prices are evaluated, the machine will check if those new prices satisfy current orders in queue; if in this new scenario an order can be evaluated the machine calls the function passing as a first parameter the string `'order'` together with the other trade data.

There are some limitations to the capability of this machine, currently it will keep only one order at time in its queue, flushing away older orders when a new one is received. Public properties of this object are:

- `ticker`: identifying the security the machine is trading on;
- `state`: current state of the trading system;
- `gain`: current gain in EUR of the trading system;
- `timeTradeStart`: the timestamp of the first price evaluated;
- `priceAvgPortfolio`: average price of shares of this security in portfolio;
- `qtyPortfolio`: quantity of shares in portfolio for this security;
- `qtyDirecta`: quantity of shares in short position;
- `qtyNegotiation`: quantity of shares in queue;
- `plotter`: a reference to the plotting system of this machine, useful for plotting custom curves.

[2]Some other sources of data for back-testing may be obtained, for instance, at `http://pitrading.com`, `http://www.nasdaq.com`, `http://www.csidata.com`, `https://quantquote.com`

The following command will start an analysis session of the last day prices detailed at second, for Saipem `Ticker` using the function `randomSample` (see section 4.1)

```
>> m = MachineHist(spm,@randomSample,1,1)
...
>> m.stop
>> clear m
```

B. Real-time machine

Once an algorithm has been tested and tuned on historical data, it may be ready to be used in the real market. If the function has been built in such a way as to being agnostic of the trading machine $\mathcal{M}$, the function $\mathcal{A}$ implementing the trading system will work automatically in this environment. In general a requirement for the function to be agnostic is to not use `tic, toc` and other current time Matlab calls; information about time must be extracted from the `timestamp` parameter.

Public properties of the machine and parameters of the function are the same as those described in the previous section, moreover this machine will also keep alive one order at time. There are differences in the management of orders, as in real time trading orders may need confirmation or generate errors (e.g., not enough budget available). The following command will start a trading session on Saipem `Ticker` using the function `randomSample` (see section 4.1); the machine will start to place orders on the market based on the strategy and will provide a real time plot of the evolution of the security. To stop the execution the method `stop` is called.

```
>> m = MachineRT(spm,@randomSample,1,1)
...
>> m.stop
>> clear m
```

IV. Simple trading systems

Some examples of implementation of simple trading strategies are provided with the Matlab package. We discuss some of them in this section. In the following snippets of code `r_volume`, `r_price` and `r_state` refers to `tradeVolume`, `tradePrice` and `newState` respectively (see Section III).

A. Random strategy

This is a random strategy that will take long or short positions without actually performing any analysis of the market, it is a kind of *null hypothesis* that can be used to evaluate performances of other strategies (which should, hopefully, do better that the random strategy). When a price is received, if a casual amount of time has elapsed then a buy or sell order is fired, depending on the timestamp. A snippet of the code is listed here:

```
function [ ... ] = randomSample(...)
    %load state variables:
    [delay, lastTrade] = state{:};

    %if we received a new price:
    if(strcmpi(typeEvent,'price') == true
        % if enough time has elapsed
        if(timeEslapsedInSeconds( ...
                lastTrade, timestamp) > ...
                        poissrnd(delay))
            % buy or sell 1 share
            r_volume = mod(sum(timestamp),3)-1;
            r_price = price;
            if(r_volume ~= 0)
                lastTrade = timestamp;
            end
        end
    end
    % update state
    r_state = {delay, lastTrade};
```

Gains for a sample intraday session at different delays and for different tickers are shown in Table I, it can be noted that if the parameter `delay` is small, gains will strictly follow the trend of the underlying security.

TABLE I
GAINS FOR RANDOMSAMPLE STRATEGY FOR AN INTRADAY SESSION

Ticker	Net Change %	30s	120s	3600s
SPM	7.19	2.12	0.6	-0.08
A2A	6.54	2.5	0.5	1.1
CPR	1.40	0.58	0.3	-1.4

B. Moving averages

A popular trading strategy used by practitioners consists in checking the crossing instant between moving averages of the stock price calculated over different time windows. The `movingAverages` function implements this technique.

```
function [ ... ] = movingAverages(...)
    %load state variables:
    [budget] = state{:};
    % build moving averages
    [mean1, mean2] = ...
        buildMovingAverages();
    % check crossover
    if( crossover() )
        cross = ~cross;
    end
    % update the plot with new means
    machine.plotter.plot('mean1',time,mean1);
    machine.plotter.plot('mean2',time,mean2);

    if(cross)
        [r_volume, r_price] = placeOrder(budget);
    end
end
```

Figure 3 shows the plot, automatically generated by the machine, of prices of Saipem together with two moving averages, respectively at 10 and 300 seconds, over a period of half an hour, togheter with a plot of gains. New curves can be programmatically drawn on the plot, as shown in the above example by calling `machine.plotter.plot('mean1',time,mean1)` the machine automatically plots the moving average associated with the evolution over time of the variable `mean1`.

Table II shows gains resulting from the variation of period of the means, for three distinct stocks in an intra-day scenario.

C. VWAP breaking

VWAP (volume-weighted average price) is a measure of the average price which takes in account the evolution of

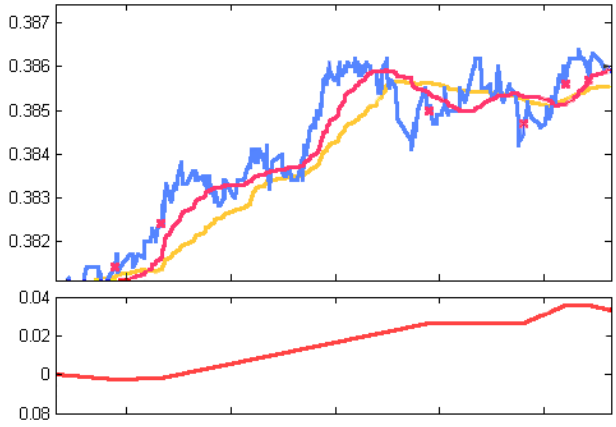

Fig. 3. Price and two moving averages (first plot), gain (EUR) (second plot)

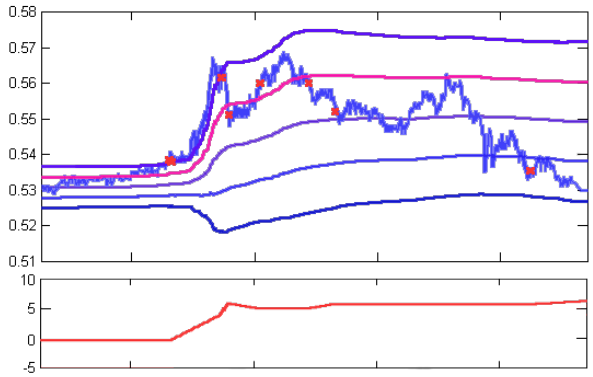

Fig. 4. Price and VWAP bounds (first plot), gain(%) (second plot), for BMPS.IT ticker for an intraday session

prices and volumes over a time window. Alongside with this measure, additional ones, called bands, are built as the composition of the VWAP together with standard deviation. Fast falling and rising trends may be analyzed by the mean of VWAP analysis and the dynamic construction of supports and resistances. The `vwapBound` function implements a strategy in which VWAP bands are used as resistances and orders are placed when price breaks through those bands.

```
function [ ... ] = vwapBound(...)
    %load state variables:
    [budget, i, N, D, S] = state{:};
    ...
    % build VWAP
    i = i + 1;
    N = N + volume*price;
    D = D + volume;
    vwap = N/D;
    S = S + (price - vwap)^2;
    sigma = sqrt(S/i);

    % if price has broken second resistance
    if(price > vwap+2*sigma ...
            || price < vwap-2*sigma)
        cross2 = true;
        ...
    elseif(price > vwap+sigma ...
            || price < vwap-sigma)
        cross1 = true;
        cross2 = false;
        ...
    else
        cross1 = false;
        cross2 = false;
        ...
    end
    [r_volume, r_price] = ...
        calcOrder(cross1, cross2, budget);
    %update state variables:
    r_state = {budget, i, N, D, S};
end
```

TABLE II

GAIN FOR MOVINGAVERAGES FOR DIFFERENT PERIODS OF THE MEANS FOR AN INTRADAY SESSION

Ticker	Net Change %	[40s,80s]	[10s,300s]	[120s,300s]
SPM	7.19	4.12	6.23	4.47
A2A	6.54	-2.33	6.34	5.76
CPR	1.40	2.83	2.41	2.3

V. FEEDBACK TRADING

In the next subsection we present an example of construction of a trading system based on the idea of feedback. A trading system can be seen as a controller trying to achieve consistent performance. We show a very simple example here, but more complex methods can be used to generate sophisticated control-based trading strategies.

A. The Barmish paradigm

Interesting results can be obtained by approaching equity trading as a control problem. For example [6] uses a feedback law to modulate the amount invested in a stock over time. The controller will seek to obtain robust returns given an uncertain input (the price of an equity).

In the example below we show a simplified model that controls the amount invested $I(t)$ based on the performance of the stock $g(t)$, which represents the gain or loss of the investment at time t. When the stock is performing well ($g(t) > 0$) we increase our position, when the price starts going down we decrease our position to limit losses. The amount invested is increased up to a saturation limit I_{max}.

Outside of the saturation regime the amount invested is controlled by the law:

$$\frac{dI}{dt} = K\frac{dg}{dt}$$

Where K is a given gain.

The invested amount I, varies from $-I_{max}$ to $+I_{max}$ where I_{max} is the maximum amount of money we are willing to invest in the stock (negative values of I mean that we are shorting the stock).

In practice, since brokerage includes fees, we do not vary our amount invested continuously, but only if the difference between the amount of shares desired and the amount owned is over a certain threshold.

With our package it is easy to write a function implementing the strategy, and then use the same code for both backtesting and real-time trading. The code below shows how to implement such a strategy within MatTrader.

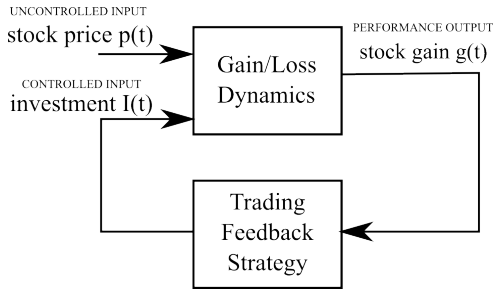

Fig. 5. Block diagram of the feedback trading strategy presented in [6].

```
function [ ... ] = barmish(...)
    %load state variables:
    [last_price, I] = state{:};

    %if we received a new price:
    if(strcmpi(typeEvent,'price') == true)

        g = (price-last_price)/last_price;
        %update I:
        I = I+K*g;

        %saturation:
        if I>Imax
            I=Imax;
        elseif I<-Imax
            I=-Imax;
        end

        %buy shares if necessary
        qty = obj.qtyPortfolio
        desired_shares = round(I/price) - qty;
        if abs(desired_shares) > 400
            %buy shares
            r_volume = desired_shares;
            r_price = price;
        end

        %update state variables:
        r_state = {price, I};
    end
end
```

Using this function we can backtest the system against any stock, calling:

```
MachineHist(ticker,@barmish,nDays)
```

Some results produced by this system are shown in Figure 6. Once we obtain satisfying results during backtesting, the algorithm can be easily deployed in real time calling:

```
MachineRT(ticker,@barmish)
```

The feedback control strategy tries to diminish the amount invested during periods when the stock price is falling, which limits drawdowns and leads, in this example, to higher returns compared to a simple buy-and-hold strategy. In this case we constrained $I(t)$ to be positive, but negative amounts can be included if we want to allow the shorting of the underlying stock.

VI. Conclusions

We presented a new open source Matlab toolbox based on a Java framework that makes it easy to access and analyze financial data and to implement any user-defined real-time trading strategy on the real market. Using data structures provided by the framework, trading strategies can

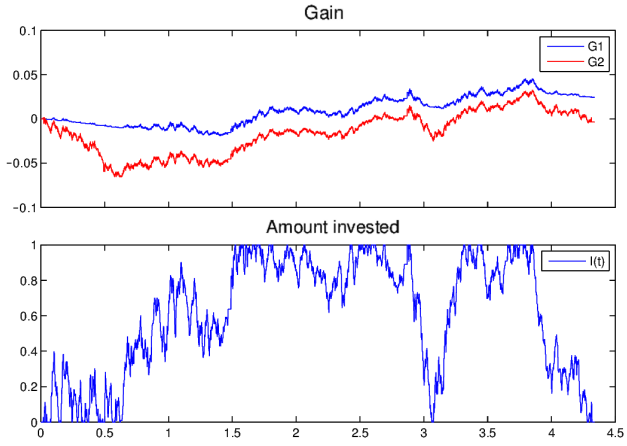

Fig. 6. Feedback trading strategy (G1) compared to a buy-and-hold baseline (G2). The amount invested is modulated by I(t).

be easily implemented as Matlab functions accepting pricing information and state variables as input, and providing execution signals as output. This is particularly suited for trading systems relying on control theory principles. We have described the main features of the package and shown some examples of implementation of simple trading strategies.

The package, its source code, and the full documentation are freely available at [7].

For future work, we plan to include in the MatTrader Matlab toolbox some time-series analysis, optimization, and machine-learning functionality, while the Java API can be extended for compatibility with other programming languages, and for providing redistributable interfaces for different online brokers.

References

[1] Iwarere, S. and Barmish, B. Ross, A confidence interval triggering method for stock trading via feedback control, *Proceedings of the American Control Conference*, 2010, pp 6910–6916.

[2] Bemporad et al., Scenario-based stochastic model predictive control for dynamic option hedging, *Decision and Control (CDC), 2010 49th IEEE Conference on*, 2010, pp 6089–6094.

[3] Calafiore, G.C. and Monastero, B., Triggering Long-Short Trades on Indexes, *International Journal of Trade, Economics and Finance*, vol. 1, 2010, pp 289–296.

[4] Barmish, B. Ross and Primbs, J.A., On market-neutral stock trading arbitrage via linear feedback, *American Control Conference (ACC)*, vol. 1, 2012, pp 3693–3698.

[5] Barmish, B. Ross, On trading of equities: a robust control paradigm, *Proceedings of the IFAC World Congress, Seoul, Korea*, vol. 1, 2008, pp 1621-1626.

[6] Barmish, B. Ross and Primbs, J.A., On a New Paradigm for Stock Trading Via a Model-Free Feedback Controller, *IEEE Transactions on Automatic Control*, vol. 61, 2016, pp 662–676.

[7] Calafiore, G.C. and Poletti, L., MatTrader source code, *github.com*, `https://github.com/kalup/MatTrader`.