Exploration into Hybrid Automated Trading Agents

Daniel Lamb
Department of Computer Science
University of Bristol
Bristol, BS8 1TR,
Email: dl0909@my.bristol.ac.uk,
Candidate Number: 39545

***Abstract*—I explore different trading algorithms for use in a continuous double auction before developing a hybrid method from the most effective methods. This new method, named GDP, is statistically analysed to see whether it is an improvement on current work, and if so, to gauge whether the improvement is significant. The algorithm is a hybrid of the GD and ZIP trading algorithms. The trader observes market activity in order to form *belief tables* over a range of possible prices, where the price chosen to bid/ask at is an estimate of which price will bring the most profit. This price is then used as a target price for the algorithm using a *learning rate* specific to the trader. I will show how this new algorithm performs significantly better than the ZIP trading agent, as well as others, in a simple and controlled simulation environment.**

I. Introduction

In 2001, a research team from IBM presented results that would go on to change the way the world's financial markets are operated forever. Automated traders were not a new concept, originating with the zero intelligence (ZI) and ZI-constrained (ZI-C) traders by [1], with some more advanced work coming in 1997/8 with the ZI-Plus (ZIP) algorithm [2] and Gjerstad and Dickhaut's (GD) algorithm [3]. However, the significance of the algorithm's success was not fully appreciated until the work of IBM's research team.

They conducted a series of experiments between human and automated traders, using the environment of a continuous double auction (CDA) to simulate systems implemented by most financial markets worldwide. CDAs are *continuous* auctions because the traders can send bids and offers to the market where they can wait to be accepted or taken; a process known as persistent offers, since the bids remain active until they are accepted by a seller or until the bidder makes a new offer. They are *double* auctions because each trader has the ability to both buy and sell stock, at different quotes. The market then provides each trading agent with feedback of the current state of the market; the quantity of information varies in different systems.

The results published by IBM showed that the automated trading agents (specifically the ones using ZIP and GD) could consistently outperform the human trading agents. They predicted that their results would have a huge impact on the global financial market "that might be measured in billions of dollars annually." A pretty profound statement, but one which instantly sparks clarity and understanding in every businessman's brain - using an automated trader can make more profit more quickly with much less effort.

IBM's experiment is the inspiration for this paper. After finding out that automated traders are more profitable than humans, the next obvious question is which algorithm is best? Both the ZIP algorithm and the GD algorithm have been proven to be very good, to roughly the same level. More recently, the Adaptive-Aggressive (AA) algorithm [4] developed in 2006 as part of a PhD thesis was shown to be statistically significantly better than the main competition [5].

However, the effectiveness of a trading algorithm depends just as much on the algorithm itself as it does on all of the other agents. A trivial example being that if all the traders in a market used the exact same algorithm then you would expect their success to converge to being equal. Another illustration of this is given by Todd Kaplan's 'Sniper' algorithm, which won a trading algorithm competition among far more complex and sophisticated algorithms simply by waiting ('lurking') until a deal is about to be done and swooping in to steal the deal. Among a market of other snipers, this algorithm is almost ineffective as they all wait until the end of the session, but with lots of other traders to leech off it does surprisingly well.

This demonstrates perfectly the need for different trading algorithms within a market, and so part of the problem in designing an algorithm becomes being different to the competition, for the sake of being different, as a new algorithm has to be different in order to improve on the original, even if that risks being worse.

By applying learning techniques from the ZIP algorithm to the GD algorithm, I hope to combine the strengths of the two. In theory, this hybrid approach should help to create an algorithm which is better at finding the market equilibrium price (p_0) than the GD algorithm, as the learning techniques adopted from ZIP will provide the mechanism to slowly traverse towards p_0, as opposed to making large jumps in price and never converging properly.

Equally, using GD as a base for the algorithm should provide an advantage over ZIP in that the algorithm will base its prices on all of the recent market activity, whereas ZIP only considers information from completed trades at prices more competitive than its own. The hypothesis of this paper is that my hybrid algorithm, hereafter to be referred to as GD-Plus (GDP), will outperform other automated trading agents in the Bristol Stock Exchange (BSE) environment.

II. Background

A. Bristol Stock Exchange

As mentioned prebiously, the experiments conducted in this paper were performed using the BSE [6] as the trading environment. Written in python, it is a useful mechanism as a simulation of the structure of modern day real-world financial exchanges. However it makes several assumptions in order to simplify the system, making it more applicable for simple testing than realistic testing, as detailed below:

- At any time, a trader can have at most one order on a limit order book (LOB). This means that any new orders will replace previous orders.
- The BSE assumes zero latency of communications between trader and market. In reality, market prices may have already changed by the time a trader receives information from the system.
- Whenever a trader issues an order, this order can potentially be executed as a transaction before any other agents can issue another order. The updated LOB is then sent to all traders.
- Perhaps the biggest abstraction of complexity from real financial exchanges is the fact that the BSE contains just one stock to be traded.

For these reasons, while the BSE provides a good simulation of real-world financial markets, any results gained from it must be tested on more advanced simulations before any real conclusions can be drawn. However, for the purposes of exploring the performance of new algorithms against existing algorithms, it is ideal.

There are four algorithms built-in to the BSE: Giveaway, ZIC, Sniper, Shaver and ZIP. An adaptation of Vytelingum's AA algorithm has also been developed for the BSE [7], though it has not been used in these experiments as its author is "suspicious about poor performance" compared to Vytelingum's original experiments.

B. Giveaway

The giveaway algorithm is as basic as a trader could possibly be. Upon receiving an order, it simply tries to complete the order at the limit price, without trying to make a transaction with some profit margin, as shown in Listing 1.

Listing 1. How Giveaway algorithm generates its prices

```
def getorder(self, time, countdown, lob):
    if len(self.orders) < 1:
        order = None
    else:
        price = self.orders[0].price
```

C. Zero Intelligence Constrained

ZIC is slightly more intelligent - although not by much. Gode & Sunder's algorithm [1] generates its order prices using a random distribution, meaning that it has a chance of doing a good deal, but usually it probably will not. It is *constrained* by the limit price of the customer, meaning that it will at least never make a loss on a deal.

In this implementation of ZIC, the seller (bidder resp.) also has an upper (lower) bound on the range of distribution that the random price comes from, which is the worst ask (best bid) existing in the LOB so far, shown in Listing 2. The algorithm will generally rely more on luck of the randomisations than anything else.

Listing 2. Generating prices in ZIC

```
minprice = lob['bids']['worst']
maxprice = lob['asks']['worst']
limit = self.orders[0].price
otype = self.orders[0].otype
if otype == 'Bid':
    price = random.randint(minprice,limit)
else:
    price = random.randint(limit,maxprice)
```

D. Shaver

The shaver algorithm uses information from the LOB in order to 'shave' a penny off the best bid/ask at the moment, without going over the limit price.

Again it is a relatively simple method, since the algorithm does not use information of recent market activity in order to find the market equilibrium p_0. Instead it simply aims to have the best offer on the market to ensure that it makes a trade, as shown in Listing 3, which just shows how the bidding agent undercuts the best offer. This can mean that it does not exploit the current p_0 to make a better profit margin.

Listing 3. Shaver aims to have the best offers at all times

```
if otype == 'Bid':
    quoteprice = lob['bids']['best'] + 1
    if quoteprice > limitprice :
        quoteprice = limitprice
```

E. 'Sniper'

Snipers exploit the format of the market session to go in just before the market closes with more urgency in order to steal the final deals.

The sniper algorithm in the BSE is a very basic interpretation of Todd Kaplan's contest-winning Sniper algorithm, and is more of a conversion from the shaver algorithm to include sniping capability. Listing 4 shows how the amount to undercut the best offer ($shave$) is inversely proportional to the time left ($countdown$), meaning it increases exponentially as time runs out, ensuring a deal is stolen.

For example, if $countdown = 10$ then $shave = 0.06$, but if $countdown = 0.1$ then $shave = 5.66$.

Listing 4. Shave rate increases as time runs out

```
lurk_threshold = 0.2
shavegrowthrate = 3
shave = int(1.0 / (0.01 + countdown /
    (shavegrowthrate * lurk_threshold)))
```

It goes on to check the new price against the customer's limit price, so as not to make a loss, but otherwise this method pretty much gurantees that one more profit making trade will be made just before the market closes, leaving no time for any other trader to react.

F. Zero Intelligence Plus

When the ZIP algorithm was invented in 1996, its fundamental breakthrough was the fact that it responded to events in the market, such as whether the best offer has been improved or a trade has just occurred, before using machine learning techniques to 'tend towards' this price, in order to provide a dynamic pricing strategy that would follow the market equilibirum, shown in Listing 5.

The version of ZIP in the BSE has been adapted as closely as possible from the original C implementation. The values of parameters used in the algorithm, such as the learning rate variables $beta$ and $momntm$, or profit margin making variables $margin\_buy$ and $margin\_sell$ were all copied from the original code.

Listing 5. Learning rate calculations in ZIP

```
def profit_alter(price):
    oldprice = self.price
    diff = price - oldprice
    change = ((1.0-self.momntm)*
        (self.beta*diff)) + (self.momntm*
        self.prev_change)
    self.prev_change = change
    newmargin = ((self.price + change)/
        self.limit) - 1.0
```

G. GDP

GDP is the algorithm developed during this research, and it uses a combination of belief functions from GD and learning techniques from ZIP. Belief functions represent the algorithm's expectancy that, for a particular price p, a bid/ask will be accpeted at that price. This expectancy is based on the history H of market activity that has occurred in the last n timesteps. The belief functions are calculated using the following function:

$$f(p) = \frac{TBL(p) + AL(p)}{TBL(p) + AL(p) + RBG(p)}$$

where

- $TBL(p)$ represents the number of taken bids at a $price \leq p$ in the last n timesteps.
- $AL(p)$ represents the number of active offers at a $price \leq p$ in the last n timesteps.
- $RBg(p)$ represents the number of rejected bids at a $price \leq p$ in the last n timesteps.

Since the notion of *rejected bids* does not really exist in the BSE, as offers are persistent until the trader replaces it with a new offer, rejected bids are classed as offers that have remained active for a certain *grace period*. This is based on one of the alterations made by Das and Tesauro [8] in their modified-GD algorithm (MGD) in order to make the belief function work with this interpretation of a CDA.

GDP uses the same learning techniques as ZIP, shown in Listing 5 in order to learn the market equilibrium dynamically, so the only thing left to explain is how to store the information needed to calculate the belief functions, and where that information comes from.

time=94.5	'bids'	'asks'
'lob'	[[23, 1], [101, 1], [102, 1], [103, 1]]	[[105, 1], [106, 1], [236, 1]]
'worst'	1	1000
'best'	103	105
'n'	4	4

TABLE I. EXAMPLE INFORMATION ABOUT THE CURRENT STATE OF THE LOB, PASSED TO AN AGENT AT EVERY TIMESTEP

'party2'	'B01'
'price'	103
'party1'	'S04'
'qty'	1
'time'	90.9375

TABLE II. EXAMPLE INFORMATION PASSED TO AN AGENT WHEN A TRADE OCCURS

On each timestep, each trader is given a copy of the current LOB, which contains a list of all current bids and asks with their respective prices, as well as a list of any trades that have been completed at this timestep. An example LOB is shown in Table I while an example trade is shown in Table II.

From this information, we build history tables; one called $self.TB$ which stores all recent trades, and one called $self.alloffers$, which stores all recent offers made in the LOB, from which it can be established whether the offer belongs in the $self.A$ table or the $self.RB$ table, depending on whether the offer has existed for the grace period or not. The updating of the TB table is shown in Listing 6, but it is a similar process for the $alloffers$ table. In this code, $tupaccepted(tup)$ checks whether any offers that used to be in the $alloffers$ table on the previous timestep have been accepted (hence changed to being classed as *taken bids*).

Listing 6. Updating the values in the TB table

```
prev_alloffers = self.alloffers
self.alloffers = []
# store all offers that are still within
# (n+grace) recent trades
for tup in prev_alloffers:
if tup[0] >= (time-(self.nrecent +
  self.graceperiod)) and
  not self.tupaccepted(tup):
    self.alloffers.append(tup[0], tup[1])

# and also store any new offers
for tup in lob['bids']['lob']:
  if not self.alreadystored(tup[0]):
    self.alloffers.append((time, tup[0]))
```

We then need to be able to get the number of entries in these tables below a certain price p to execute the belief function $f(p)$. This is where the bottleneck of the program is, as we need to calculate the number of entries for each of TB, A and RB for each price p in a range from $minprice$ to $maxprice$. Since the market equilibirum is not known beforehand (or at any point), it is necessary for this range to be quite large.

I therefore had to optimise this section of the code in a couple of ways. Firstly, I made the functions that calculate the number of entries at a specific price p using dynamic programming techniques, so the entire list is only accessed once. To do this, I store the index of the list that we got to for price $p-1$, as we know that all of the items in the list that are $\leq p-1$ will also be $\leq p$. Using this index, it is possible to then

start the next iteration from this point and break out as soon as we access an item in the list that is $> p$, returning p as the index for the next iteration. The only additional overhead is sorting the list beforehand, which is done using Python's `sorted` method. The code for the TB table is shown in Listing 7, where $self.TB\_a$ is the TB table sorted in ascending order of price.

Listing 7. Get number of entries in TB table with price $\leq p$

```
def lenTB(self, price, prev_lenTB):
    start = prev_lenTB
    tb = start

    for i in range(start,len(self.TB_a)):
        if self.TB_asc[i][1] <= price:
            tb+=1
        else:
            break
    return tb
```

This still took a long time to execute, so more optimisation was needed. Since trades only persist for one timestep, the TBL history must be updated at each timestep so as not to miss any information. However, as the bids and asks in the LOB are persistent, we can optimise their storage by only updating the history of AL and RBG every x timesteps, reducing the amount of work to be done by a factor of $x * num\_GDP\_agents$. Setting this to update every 0.5 seconds seemed to work well, and this does not lose much accuracy as the market is unlikely to change significantly in less than half a second.

III. Experimental Setup

The setup of the experiment was to compare the performance of GDP against all the other trading agents in the BSE. The traditional method for doing this is to exhaustively vary the ratios of the different agents in the system. In this experiment, 4 different types of agent have been used divided into 8 agents, so the ratios of each agent might look something like this:

[5, 1, 1, 1]

[4, 1, 2, 1]

[3, 1, 3, 1]

[2, 1, 4, 1]

[1, 1, 5, 1]

[4, 1, 1, 2]

etc...

In general, the other 3 agent types used in these experiments were Shaver, ZIC and ZIP, for the simple reasons that Giveaway is too 'dumb' and this implementation of Sniper is not as much more basic than Kaplan's Sniper, so it can give more unpredictable or unexpected results.

IV. Experimental Results

Using the method of exhaustively varying the combinations of ratios, I ran the experiment 5 times and took the average of all 5*35 sets of results.

Agent Type	Total Balance	Num	Average Balance
GDP	483	4	120.750000
SHVR	325	4	81.250000
ZIC	331	4	82.750000
ZIP	450	4	112.500000

TABLE III. Example of the output from a market session.

Agent Type	1	2	3	4	5	Mean	Std Dev
GDP	130.06	129.65	128.31	134.22	130.26	130.50	20.53
SHVR	102.97	101.54	107.54	97.74	99.44	101.85	15.24
ZIC	88.23	89.62	85.30	94.60	84.64	88.48	19.34
ZIP	121.52	124.69	122.65	121.87	120.68	122.28	17.34

TABLE IV. Average balance on iteration i of experiment, with overall mean and standard deviation over all sessions (35 sessions per iteration).

Table III shows an example of the output from 1 market session, where the ratio of all agents is equal, just to give an idea of what happens in a single market session.

It shows that GDP performed best on this round, but that could easily be an anomaly, so next we take a look at the average result of all tests in Table IV. This table shows the average balance for each agent on each iteration of the experiment (1→5), where an iteration has 35 separate tests varying the proportions of each agent. As can be seen, the average balance for each iteration remains remarkably similar throughout each iteration for all 4 agents used here. This is perhaps due to the fact that there is only a small amount of randomness in the generation of orders and decisions of profit margins for each agent.

We can then see the overall mean for all sections in the 'Mean' column, as well as the standard deviation over all $5 * 35$ sessions. The overall mean balance shows the GDP does perform pretty well against these opponents. As expected, ZIP also does well compared to Shaver and ZIC - neither of which use any information of market activity - however it was not expected to lose to GDP so comfortably.

Expressed as percentages, GDP takes on average 29.45% of the profit, ZIP takes 27.6%, SHVR takes 22.99% and ZIC takes 19.97%. Normalising these percentages gives a factor that represents a multiplier of how much profit we can expect an agent to make:

GDP 1.17

ZIP 1.1

SHVR 0.92

ZIC 0.79

The standard deviation also gives interesting results. Despite being very consistent with overall iteration balances, GDP has the highest cross-iteration deviation. This can suggest that the trader may perform very differently when there are differing proportions of itself inside the market. For example, if there is only 1 GDP agent, does this give it a different success rate compared to when there are 5 GDP agents? It could also suggest that it can perform better/worse depending on the proportions of one or more of the other agents. Figure

1 shows the average performances of GDP when it is the minority agent in a market session compared to when it is the majority. As shown, the GDP makes on average 10 pence more over 50 market sessions when it is the *minority* trading agent in the market - about 8% more. Although looking at the graph shows that there is a lot more noise than this statistic suggests, and that GDP is probably more consistent when it is the majority.

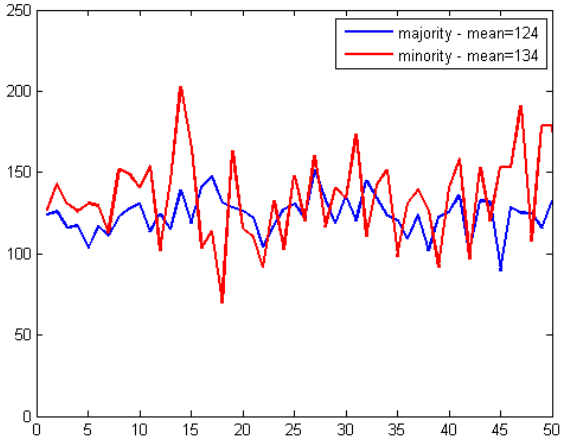

Fig. 1. Comparison between GDPs performance against other trading agents when it is the majority and minority agent. Agent balance shown on Y axis, agent number on X axis.

There are several possible reasons for this; firstly, when GDP is the majority, there are more GDP agents' balances to average, so the average is more likely to be similar in each market session. Secondly, when it is the majority, each GDP agent will be competing against many other GDP agents, meaning that the performances are likely to be similar in each session. In contrast, when GDP is the minority, the majority agent could be different in each session, making the performances more unpredictable.

To determine the true behaviour of an agent during a market session, it can also be useful to put it up against itself, in order to show how the market equilibrium will progress when each agent uses the same (give or take a few minor randomisations in the learning rate) strategy. To do this I ran 5 market sessions, each involving 4 GDP agents, and recorded information of each transaction made by all of the $5 * 4$ agents. The result of this experiment is shown in Figure 2.

As shown, over the 180 second period, most trading happens at very close to the true equilibrium price of £1.00, which is good, although there is some strange behaviour that needs explaining. At the beginning of all 5 sessions in the experiment, there was not a single transaction made during the first ~ 30 seconds of the session.

This could be because each agent is waiting for some more information of recent market activity in order to calculate their own belief function, but every other agent is also waiting so nothing happens. This shows a potential flaw in the algorithm that would need to be ironed out in the future. While it will not prove problematic in most markets, where you could expect with reasonable confidence levels that there will be several

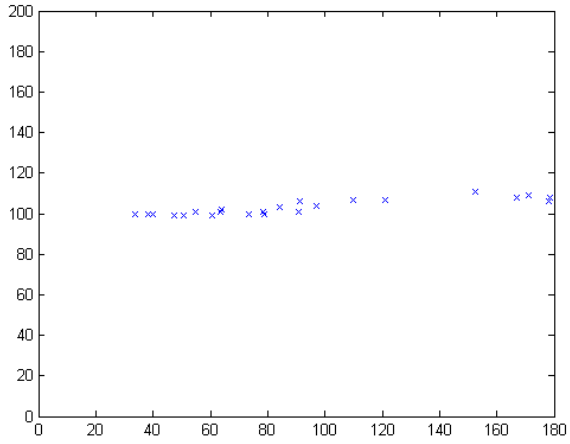

Fig. 2. Transaction history for market populated by 20 GDP traders, over 180s of trading. Most trading seems to occur around the equilibrium price of £1.00.

types of agents that will create market activity in this time, it is still worth fixing so that GDP could exploit moments where everyone else in the market is also waiting. One possible method to do this would be for GDP to adopt Shaver-like behavious while there is nothing happening in the market, knocking a penny off the best price on each iteration until a transaction is made that it can use to calculate belief functions. It is also an advocation for more aggressive algorithms, such as AA [4], which would seize the initiative a bit more in situations like this to drive the market.

Comparing this market activity to a market filled with ZIP agents (Figure 3) shows a surprisingly large difference. The ZIP agents behave much more predictably than GDP, with many transactions occurring just after new orders are made (at an interval of 30 seconds). To begin with, the ZIP agents perform transactions at a wide variety of prices, however they very quickly converge to the true market equilibrium price of £1.00. This is due to the learning techniques used in their pricing strategy, which cause each agent's estimations to get closer to the market equilibirium (and further from their original profit making margins) with each timestep. This is not ideal in this situation, leading to each agent making very little profit, but it is unlikely in a real market situation that every trading agent would use the same algorithm, so this should not be a problem in reality.

The more eccentric behaviour of GDP compared to ZIP is exaggerated further when looking at what happens when we extend the running time of a market session from 180 seconds to 600 seconds, as shown in Figure 4. Here we can see how the market equilibrium extends and amplifies in a sine wave over time. It steadies a little towards the end, peaking at around £1.40 every time and troughing at around £1.00.

This peculiar behaviour could be caused by the belief functions. Since the most suitable price is selected as the largest product of profit to be made and belief value, a range of prices with the same belief value will choose the one that makes the most profit. This could cause the fluctuations in market equilibrium as each agent tries to maximise their profit

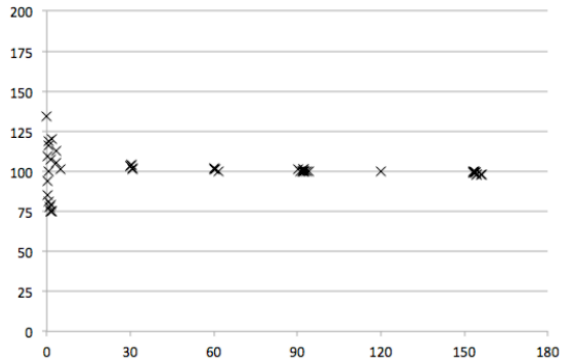

Fig. 3. Transaction history for market populated by ZIP traders, over 180s of trading. ZIP traders can rapidly converge on the equilibrium price. Graph taken from BSEGuide [6]

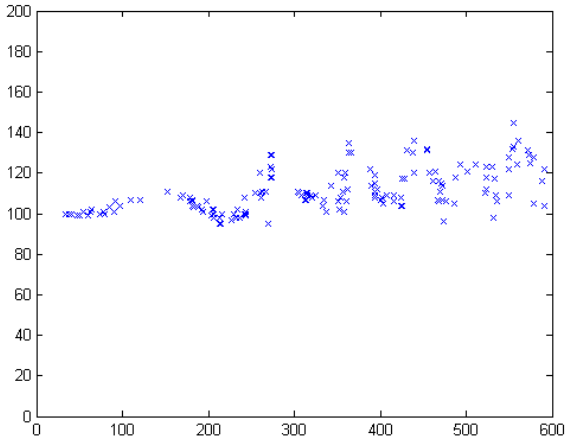

Fig. 4. Transaction history for market populated by 20 GDP traders, over a full market session (600s of trading). The trading seems to get much more erratic towards the end of the session. [6]

margin until eventually no other agent will sell at that price, so the price goes back up again.

I ran the same experiment that generated the data in Table IV using the other two trading agents, Sniper and Giveaway, in place of the two lowest performing agents in the previous experiment, ZIC and Shaver. The results of this experiment are shown in Table V.

The surprise here is that the Giveaway algorithm does so well, much better than ZIP. This could be just because it goes straight to the limit price, so it will be able to do a lot more transactions than any other algorithm. Nevertheless it is still beaten by GDP, whose mean is very slightly improved from the previous experiment, probably due to how poorly Sniper does.

Agent Type	1	2	3	4	5	Mean	Std Dev
GDP	136.66	132.94	142.99	134.95	130.35	135.58	22.59
GVWY	124.54	123.27	129.77	121.05	123.67	124.46	18.41
SNPR	36.19	38.46	36.37	35.32	36.90	36.65	11.28
ZIP	112.95	107.46	109.37	109.9	108.47	109.63	18.08

TABLE V. AVERAGE BALANCE ON ITERATION i OF EXPERIMENT, WITH OVERALL MEAN AND STANDARD DEVIATION OVER ALL SESSIONS (35 SESSIONS PER ITERATION).

We now have data of the performances of GDP and ZIP against all other trading agents supported in BSE, which is ideal for performing a statistical test, to see if GDP is significantly better than ZIP for this sample set. This paper [9] shows the Robust Rank-Order test to be a more reliable measure of significance than the Mann-Whitney U test, for "when the samples come from populations with dierent higherorder moments". Since that is not the case with this dataset, it is sufficient to use the more simple Mann-Whitney U test.

By this time computing the average values from each configuration rather than each iteration, we get 35 values for both GDP and ZIP from both experiments, giving 70 data points for each sample. We want to know if we can reject the null hypothesis h_0, that the data from GDP and ZIP could have come from the same distribution, to show that GDP is significantly better (*or* significantly worse) than ZIP, so we will use a two-tailed test, with a significance level of 0.01.

After calculating the ranks of all 140 samples, the summed ranks are as follows:

$$GDP : 6296; ZIP : 3574$$

Where data has been ranked lowest to highest. Using the formula:

$$U_X = N_1 * N_2 + N_X * \frac{N_X + 1}{2} - T_X$$

Where T_X is the total rank for distribution X, we get U values of:

$$GDP : 1089; ZIP : 3811$$

Using the in-built Matlab function ranksum [10] returns a p value of 0.000000014258 for these U-values. Since this is ≤ 0.01, we can reject the null hypothesis and statistically show that GDP is significantly different (and in this case, better) than ZIP in this environment.

V. FUTURE WORK

- If there is no data of other deals, it will wait rather than seizing initiative. This can be improved.
- Test performance when trading multiple stocks, not just one.
- Test against other leading trading algorithms, such as AA and GDX.

VI. CONCLUSION

I have explored the way a number of different trading algorithms work and analysed their strengths and weaknesses in order to design a new algorithm, GDP, that combines belief functions from GD [3] with learning mechanisms from ZIP [2] to create a possible improved alternative. Using the BSE [6] I was able to simulate how GDP fared up against a number of different algorithms in varying ratios and statistically showed that it was significantly better than ZIP in this environment.

By analysing GDPs behaviour in a market filled only with GDP agents, I was able to learn how the belief functions affect the market equilibrium, causing it to fluctuate, as well as identify areas for improving the algorithm, such as some way of taking action when there is no market activity. As stated earlier, due to the abstractions of complexity in BSE, these results must be tested in more advanced simulations before any real conclusions can be drawn, but in terms of a proof of concept, I believe I have shown that this algorithm has potential to be an improvement on some current work.

References

[1] Dhananjay K Gode and Shyam Sunder. "Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality". In: *Journal of political economy* (1993), pp. 119–137.

[2] Dave Cliff, Janet Bruten, et al. "Zero Not Enough: On The Lower Limit of Agent Intelligence For Continuous Double Auction Markets". In: *HP Laboratories Technical Report HPL* (1997).

[3] Steven Gjerstad and John Dickhaut. "Price formation in double auctions". In: *Games and economic behavior* 22.1 (1998), pp. 1–29.

[4] Perukrishnen Vytelingum, Dave Cliff, and Nicholas R Jennings. "Strategic bidding in continuous double auctions". In: *Artificial Intelligence* 172.14 (2008), pp. 1700–1729.

[5] Marco De Luca and Dave Cliff. "Human-agent auction interactions: Adaptive-aggressive agents dominate". In: *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*. AAAI Press. 2011, pp. 178–185.

[6] Dave Cliff. "Bristol Stock Exchange". In: (). URL: https://github.com/davecliff/BristolStockExchange.

[7] Ash Booth. "Source-code for an AA implementation". In: (). URL: https://github.com/ab24v07/BristolStockExchange.

[8] Rajarshi Das et al. "Agent-human interactions in the continuous double auction". In: *International Joint Conference on Artificial Intelligence*. Vol. 17. 1. LAWRENCE ERLBAUM ASSOCIATES LTD. 2001, pp. 1169–1178.

[9] Nick Feltovich. "Nonparametric tests of differences in medians: comparison of the Wilcoxon–Mann–Whitney and robust rank-order tests". In: *Experimental Economics* 6.3 (2003), pp. 273–297.

[10] In: (). URL: http://www.mathworks.co.uk/help/stats/ranksum.html.

VII. Appendix

Listing 8. GDP source code ©Daniel Lamb 2013

```
class Trader_GDP(Trader):

        def __init__(self, ttype, tid, balance):
                self.ttype = ttype
                self.tid = tid
                self.balance = balance
                self.blotter = []
                self.orders = []
                self.alloffers = []
                self.TB = []
                self.A = []
                self.RB = []
                # same lists need to be sorted by price for optimisations
                self.TB_asc = []
                self.A_asc = []
                self.RB_asc = []

                self.nrecent = 60
                self.graceperiod = 15
                self.job = None
                self.limit = None
                self.active = False
                self.margin_buy = -1.0*(0.05 + 0.3*random.random())
                self.margin_sell = 0.05 + 0.3*random.random()
                self.momntm = 0.1*random.random()
                self.beta = 0.1 + 0.4*random.random()
                self.prev_change = 0
                self.min = 1
                self.max = 200

        def getorder(self, time, countdown, lob):
                if len(self.orders) < 1:
                        self.active = False
                        order = None
                else:
                        self.active = True
                        self.limit = self.orders[0].price
                        self.job = self.orders[0].otype
                        if self.job == 'Bid':
                                # currently a buyer (working a bid ord
                                self.margin = self.margin_buy
                        else:
                                # currently a seller (working a sell o
                                self.margin = self.margin_sell
                        quoteprice = int(self.limit * (1 + self.margin
                        self.price = quoteprice

                        order=Order(self.tid, self.job, quoteprice, se

                return order

        def tupaccepted(self, tup):
                # return true if the argument tup is an offer that has
                accepted = False
                for tup2 in self.TB:
                        if tup[0] == tup2[0] and tup[1] == tup2[1]:
                                accepted = True
                return accepted

        def alreadystored(self, price):
                for tup in self.alloffers:
                        if tup[1] == price:
                                return True
                return False

        def updatealloffers(self, lob, time):
                prev_alloffers = self.alloffers
                self.alloffers = []
                # store all offers that are still within (n+grace) rec
                for tup in prev_alloffers:
                        if tup[0] >= (time-(self.nrecent + self.gracep
                                self.alloffers.append((tup[0], tup[1])

                # and also store any new offers
                for tup in lob['bids']['lob']:
                        if not self.alreadystored(tup[0]):
                                self.alloffers.append((time, tup[0]))


        def calculateTBL(self, trade, time):
                prev_TB = self.TB
                self.TB = []
                # store old trades that are still within n recent trad
                for tup in prev_TB:
                        if tup[0] >= (time-self.nrecent):
                                self.TB.append((tup[0], tup[1]))

                # and also store any new completed trades
                if trade != None:
                        self.TB.append((trade['time'], trade['price'])

                self.TB_asc = sorted(self.TB, key=lambda x: x[1])

        def calculateRBL_AL(self, lob, time):
                # rejected bids are offers that have existed for a cer
                # so store offers that are older than time-grace but y
                self.RB = []
```

```
        self.A = []

        # self.alloffers is already filtered to stuff
        # that is within the last (nrecent+graceperiod) seconds
        for tup in self.alloffers:
                if (time-self.graceperiod) > tup[0]:
                        # all offers that have existed for grace period
                        # -> count as rejected
                        self.RB.append((tup[0], tup[1]))
                else:
                        # else it is an open offer -> count as offer
                        self.A.append((tup[0], tup[1]))
        self.RB_asc = sorted(self.RB, key=lambda x: x[1])
        self.A_asc = sorted(self.A, key=lambda x: x[1])

    def lenTB(self, price, prev_lenTB):
        start = prev_lenTB
        tb = start

        for i in range(start, len(self.TB_asc)):
                if self.TB_asc[i][1] <= price:
                        tb+=1
                else:
                        break
        return tb


    def lenRB(self, price, prev_lenRB):
        start = prev_lenRB
        rb = start

        for i in range(start, len(self.RB_asc)):
                if self.RB_asc[i][1] <= price:
                        rb+=1
                else:
                        break
        return rb


    def lenA(self, price, prev_lenA):
        start = prev_lenA
        a = start

        for i in range(start, len(self.A_asc)):
                if self.A_asc[i][1] <= price:
                        a+=1
                else:
                        break
        return a


    def calculatebelieffunction(self):
        optimum_price = None
        optimum = 0
        prev_belief = 0
        prev_lenTB = 0
        prev_lenA = 0
        prev_lenRB = 0
        for p in range(self.min, self.max):
                lenTB = self.lenTB(p,prev_lenTB)
                lenRB = self.lenRB(p,prev_lenRB)
                lenA = self.lenA(p,prev_lenA)
                prev_lenTB = lenTB
                prev_lenA = lenA
                prev_lenRB = lenRB

                denom = (lenTB+lenA+lenRB)
                if denom == 0:
                        belief = 0
                else:
                        belief = float(lenTB + lenA) / denom

                if self.job == 'Ask':
                        belief = 1 - belief

                profit = self.profit_function(p)
                product = belief * profit
                if (self.job == 'Bid' and product > optimum and not lenTB == 0):
                        optimum = product
                        optimum_price = p
                        prev_belief = belief
                elif (self.job == 'Ask' and product > optimum and belief > prev_belief):
                        optimum = product
                        optimum_price = p
                        prev_belief = belief
        return optimum_price

    def profit_function(self, price):
        if self.job=='Bid':
                return self.limit - price
        elif self.job == 'Ask':
                return price - self.limit
        return None

    def respond(self, time, lob, trade, verbose):

        def profit_alter(price):
                oldprice = self.price
                diff = price - oldprice
                change = ((1.0-self.momntm)*(self.beta*diff))
                self.prev_change = change
                newmargin = ((self.price + change)/self.limit)

                if self.job=='Bid':
                        if newmargin < 0.0 :
                                self.margin_buy = newmargin
                                self.margin = newmargin
                else :
                        if newmargin > 0.0 :
                                self.margin_sell = newmargin
                                self.margin = newmargin

                #set the price from limit and profit-margin
                self.price = int(round(self.limit*(1.0+self.ma


        if (self.job == 'Bid' or self.job == 'Ask'):
                self.calculateTBL(trade, time)
                if (time%0.5 == 0):
                        self.calculateTBL(trade, time)
                        self.updatealloffers(lob, time)
                        # self.calculateAL(lob, time)
                        self.calculateRBL_AL(lob, time)

                        final_price = self.calculatebelieffunc
                        if not final_price == None:
                                profit_alter(final_price)
```