The Structure and Behaviour of the Continuous Double Auction

Sanyam Vyas[1]

[1] University of Bristol, Bristol BS8 1TH, United Kingdom
sv17657@bristol.ac.uk

Abstract. Automated trading strategies have been extensively researched in the last three decades, simulated agent models have been utilised and have improved over time. This paper implements an automated trading algorithm which is inspired by the Gjerstad-Dickhaut (GD) algorithm. Important modifications have been made to the implemented GD algorithm in an attempt to improve the performance of the model in a continuous double auction (CDA) environment. The results of this algorithm performance have been conducted through statistical comparisons with another trading strategy. The algorithm implementation along with statistical analysis have been conducted using Bristol Stock Exchange that is a minimal simulation of a limit-order-book financial exchange.

Keywords: Automated Trading, Gjerstad-Dickhaut, Auction Markets.

1 Background

Double auctions have been extensively used in the stock markets of several countries, including NYSE, recognised as New York Stock Exchange. It is also used in markets for financial instruments such as options and futures. Within a double auction trading period, a seller or a buyer an submit an offer that is monitored by every other buyer and a seller who is in the market. Offers are accepted at any time during the trading period by the buyer while a sellers ask can be accepted at any time by the buyer depending on the respective allocated shares/budget of the traders.

Traditionally, human traders interacted amongst each other in a continuous double auctions to finalise transactions a given share. However, human traders have been replaced by automated trading systems, allowing flexibility to handle very high daily order flows. Several investment banking organisations along with hedge funds utilise these trading algorithms and also aim to hire individuals who design the best automated traders. Hence, in order for large firms maximise their profits in this form of trading, the algorithms that produced are not published.

As an algorithm is designed, it requires experimental investigation which focuses on its behaviour and market performance in the CDA. The first medium of experimental investigation in CDA was designed by Smith in 1962 [5] Supply and demand conditions were introduced in the market by giving buyers a redemption value for each unit of an abstract commodity purchased. A surplus is given to the buyers that is equivalent to the difference between the purchase price negotiated by a seller and the redemption value. The sellers receive a surplus equal to the difference between their unit cost and the buyer unit purchase price. For every trading session, competitive equilibrium price in achieved allowing for the quality of traders being compared.

Several trading algorithms have been published over the last three decades and have been proved to outperform human traders in laboratory-style experiments in trading markets. Trading strategies include ZI ,ZIC [7], ZIP [6], GD [2], MGD [3] and AA [4]. In this paper, a trading algorithm with a foundation of GD algorithm has been produced in an attempt to further optimise the quality of trades, appropriate statistical analysis has been conducted against DIMM algorithm (implemented by D.Cliff and J.Cartlidge) to observe the quality of the implemented trading algorithm.
The rest of the paper is structured as follows. Section 2 describes the trading environment used in this paper in order to run the implemented trader against benchmark algorithms already implemented in the algorithm, while section 3 focuses on the how the trading algorithm has been implemented. Section 4 evaluates results and statistical analysis of the implemented algorithm and section 5 presents the conclusions of the implemented model quality and the possible direction that could be applied to the model as an attempt to solve its shortcomings.

2 Bristol Stock Exchange

The implemented trading algorithm utilises Bristol Stock Exchange (BSE) [1] which is a novel minimal simulation of a centralised financial market, based on a Limit Order Book (LOB), a technique that is commonly utilised in several major stock exchanges. The motivation for the implementation of BSE was to allow experimental probing of an abstract market while functions. The BSE was originally developed on Python and successfully used as resource for students looking to expand their understanding on the contemporary financial market systems, or for researchers and teachers looking to achieve learning outcomes in students).

The BSE comprises of and maintains a single LOB for recording limit orders in a single abstract type of tradeable item, a feature that is used in real financial exchanges (usually for up to thousands of tradeable items). In particular, a trader is able to issue a new order that immediately replaces any previous order that the trader had on the lob allowing the trader to have at most one order on a LOB at a specific time. Trades are possible as soon as a new order is issued i.e. has zero latency in communications between the traders and the exchange. All trades are then immediately updated in the LOB before any other trader could issue another order.

The records in the LOB could be controlled by a wide range of dynamics of supply and demand in the BSE market. De Luca et al [9]discuss the requirement to explore trading agents in simulated markets that are more realistic used in prior experimental studies. BSE has allowed the possibility to switch between static-equilibrium supply/demand schedules to dynamic variations in the supply and demand schedules while a simultaneous replenishment of orders take place. The continuous customer orders which are given in random stream is a variation to the periodic customer orders which were previously implemented in Vernon-Smith's experiments [5]

A sample of different robot trading algorithms have been used in BSE, drawn from literatures from over four decades. The variety of these traders allow for exploring dynamics of LOB-based CDA prior to any new implementations. The current design consists of several "sales traders" which wait for an order to arrive from a customer in order to be processed afterwards, however, the program could be extended so that a "proprietary trader" could be designed i.e. where the trader has a set amount of capital for buying and selling an abstract quantity.
The program currently comprises of 5 algorithms including ZIC, Sniper (which is inspired by Rust et al's Kaplan's Sniper [10]) and ZIP.

3 Implemented Model

While the Bristol Stock Exchange repository contains code for Kaplan's Sniper (adaptation), ZIC, ZIP and AA (specifically stored in a separate file on the GitHub page [1]), there are no current implementations of the Gjerstad-Dickhaut algorithm which have been implemented within the BSE repository online on GitHub.
The implemented model is inspired from the GD algorithm and attempts to surpass its present shortcomings in order to improve its trading performance. Additionally, unlike the implemented sales traders (existing in BSE) that take orders from customers to provide a profit, the implemented model acts as a proprietary trader with a balance of 500 at the start of every trading session.

3.1 Gjerstad-Dickhaut Algorithm

Theoretically, the Gjerstad-Dickhaut (GD) algorithm is an agent architecture that uses the transaction history H_M to make its bidding and selling decisions. The orders produced by GD attempt to maximise its expected profit. The transaction history recorded is used to calculate the belief function. The belief function is shown in (1):

$$f(p) = \frac{AAG(p)+BG(p)}{AAG(p)+BG(p)+UAL(p)} \quad (1)$$

For sellers, AAG(p) is the number of accepted asks in H_M with price $\geq$ p, BG(p) is the number of bids in H_M where price $\geq$ p and UAL(p) is the number of unaccepted asks in

H_M with price ≤ p. For buyers, AAG(p) is the number of accepted bids in H_M with price ≤ p, BG(p) is the number of bids at price ≤ p and UAL(p) is the number of rejected bids at price ≥ p.

The GD agent then chooses a price from H_M that maximises the expected profit , defined as product of f(p) and the gain from trade at that price that is equal to p-l for sellers and l-p for buyers, where l is the limit price. Several shortcomings exist in this algorithm, including the complexity along with the bid/sell price it computes. Additionally, the limit price is only produced for sales traders since a customer sets the price for them. The implemented model, Trader TMGD provides selective variations to GD as an attempt to reduce the given shortcomings.

3.2 Trader TMGD

TMGD algorithm, short for Triple-M Gjerstad-Dickhaut or Modified Market Maker Gjerstad-Dickhaut algorithm, uses the strategy of the GD trader of using a (variation of) belief function and multiplying it with the gain from the trade at the specific price. TMGD contains multiple functions in the "respond" function (see appendix) which formulate the main strategy for the algorithm. In essence, the aim for the TMGD belief function, just like GD's belief function is to calculate the expected profits for all the bids and sells in history with length h, and pick the price with the highest expected profit if it is within the budget allowance. The expected profit, Ex(p), of price p is calculated as:

$$Ex(p) = b(p) * g(p) \tag{2}$$

where b(p) is the belief function calculated as:

$$b(p) = \frac{Tr(p)}{h} \tag{3}$$

where Tr(p) is the number of successful trades in auction history. The g(p) (from (2)) is the gain function equivalent to p – l for sellers and l – p for buyers, where l is the limit price. In contrast to the GD algorithm that acts as a sales trader with a given limit price, the first variation of the TMGD algorithm is the setting of the limit price. Since the market maker has been provided a set balance at the start of a trading session, setting the "limit price" as the balance will be detrimental for the generating profits since it will produce an expected profit output that will not be feasible for trading. Thus, TMGD algorithm indexes the transaction history and sets the limit price as the latest transaction value a quantity in an aim for the algorithm to realistically maximise its profits.

As a bid/sell price is selected (that achieves the expected profit), TMGD provides another variation to the original GD algorithm by including a "delta" value to the addition of the selected price. When selling a quantity, a delta value is added to the set price while for buying a quantity, a delta value is subtracted from the set price as an attempt to maximise the profit. The delta value is calculated by accessing the transaction history and calculating the absolute mean difference between all adjacent transactions of the list. After the transaction is conducted by TMGD, the value is added to the transaction history within the respond() function

As a consideration of the typical market session equilibrium, a constant value of 0.1 is multiplied to the delta value when a given number of transactions have been processed as an aim to maximise profit as market eventually reaches an equilibrium price at the end of a session.

Another variation has been added to the TMGD algorithm that aims to reduce the running time of the original GD algorithm due to the increasing auction history. Additionally, there is a high possibility for traders to change their policy at any time of a session, meaning that TMGD will only require the most recent transaction history to formulate a price that will provide an expected profit. Thus, when considering both points, a cap of 40 on the size of recent market history is included when calculating the belief function. These additions were inspired by Tesauro et al's MGD algorithm. Lastly, the strategy of TMGD was to switch between buying and selling regardless of the trends in the trading session.

4 Statistical Analysis and Results

In order to test the TMGD traders' performance against other traders, appropriate statistical analysis must be conducted. Since the trader's main aim is to increase the amount of profit it could possibly earn during a session, a performance metric of profit earned per session for a trader was used in the statistical analysis that will be conducted.

For statistical analysis of TMGD, Mann-Whitney U statistical test (U test) has been conducted in order to compare the profits of each trader with TMGD. The test provides a null hypothesis H_0 (there is no difference between the profit of both traders) and a degree of overlap in ranks between the two groups' trader profit per session. Overall, the smaller the $U_{statistic}$, the bigger the difference between the traders, while the bigger the $U_{statistic}$, the smaller the difference between traders.

In general, for a given sample A and B (from the same population), a null hypothesis is produced that sample A is equal to sample B. The U test initially combines A and B into one list and sorts it while remembering which data comes from each sample. These are then ranked from lowest to highest and every value from one sample is compared to every value from the other sample. If any values match, the an average of both ranks will be the rank for both of the values. All ranks are then added up (rank sum) for each sample and the $U_{statistic}$ value is calculated as shown below:

$$U_{statistic} = Rank\ Sum - \frac{n(n+1)}{2} \quad (4)$$

where n is the total number of samples for a sample. The lowest $U_{statistic}$ value is selected and a p-value is attained from a distribution table. If the p-value is less than $U_{statistic}$, the null hypothesis H_0 will be rejected.

In order to carry out the $U_{statistic}$ test, the TMGD proprietary trader was compared with the DIMM proprietary trader while other sales traders were present in BSE. In order to compare the quality of the trader, 1 buying and 1 selling sales traders (GVWY, SHVR, ZIC and ZIP) were set to trade alongside one TMGD and DIMM traders. The maximum price from every trading session was extracted from both traders. For each run-through of code, specifically termed in this report as a "trading day" comprised of 6 sessions. In order to increase the amount of data for comparison, 5 trading days' transactions were collected for two scenarios, the first scenario comprising of a static equilibrium trading session while the other consisting of a scheduled offset in order to change the equilibrium every session. Each trader was provided with a balance of 500 and a symmetric supply and demand curve was generated between 400 to 600. A maximum of 1 abstract quantity could be purchased by any trader.
In order to produce the statistical analysis, trading days were recorded in comma separated files and the maximum price from each session was indexed for TMGD and DIMM traders. For calculating $U_{statistic}$ and p-value, the function "mannwhitneyu " was used from "SciPy" library. A p-value was also generated where any value greater than 0.05 will mean that there is no difference between the profits between TMGD and DIMM.
Additionally, box plots were also outputted for the trading days in order to compare the performance of the respective traders by visualizing the distribution of maximum price for all trading sessions. The performance of the sales traders was not compared with the TMGD and DIMM since these are proprietary traders which are not given any capital from customers to invest. Additionally, two sales traders (one for buying and the other for selling) are required for equivalence with proprietary trader, however, multiple traders cannot be compared to one trader since the speed of buying and selling is much faster for multiple sales traders.

4.1 Results in static equilibrium

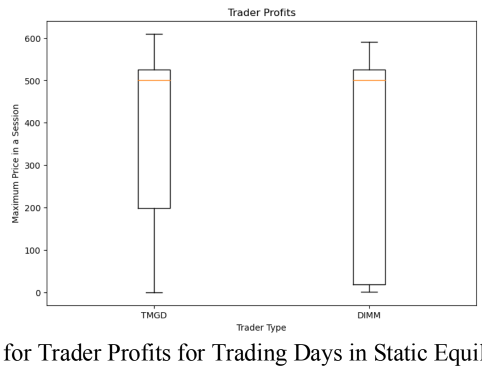

Figure 1: Box Plot for Trader Profits for Trading Days in Static Equilibrium

As shown in Figure 1, both, TMGD and DIMM traders have produced similar results in terms of the range of maximum price within a static equilibrium between 400 and

600. The prices lower than 500 are due to the traders buying a quantity and not selling them before the end of a trading session. In contrast to the similarity, the TMGD trader produced a slightly higher maximum price compared to the DIMM trader. In addition, the first quartile for TMGD is lower compared to DIMM. Overall, the results for both traders are similar as the maximum mean price at the end of every session is 500 (equivalent to no profit, or a loss). The p-value at the end of the session was 0.414.

4.2 Results in dynamic equilibrium

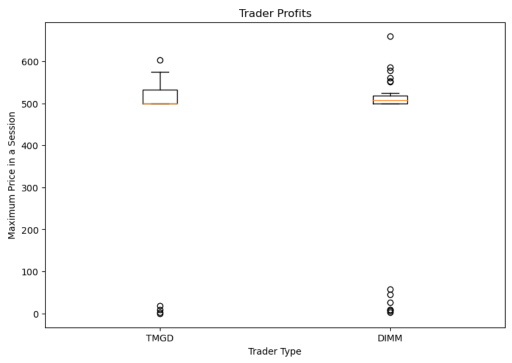

Figure 2: Box Plot for Maximum Trader Price for Trading Days in Dynamic Equilibrium

As shown in Figure 2, TMGD and DIMM have produced similar results at the end of the session in terms of the mean maximum price of a session with scheduled offset applied to the equilibrium. The prices lower than 500 are where the traders have bought a quantity and been able to sell before the end of the session. However, the TMGD has a more significant third quartile compared to DIMM which suggests that the TMGD performance was slightly superior to DIMM. The maximum price at the end of a session was also produced by TMGD. The p-value for the results is 0.345.

4.3 Analysis

With the trader results being in the same environment i.e. trading against the same traders for the same duration, the test is carried out is fair. With both of the p-values being much greater than 0.05, the theoretical interpretation is that the null hypothesis has to be accepted, which suggests that "there is no theoretical difference" between the traders. Trader TMGD produces the same profit values as DIMM, however, the TMGD has been able to produce lower overall losses per session compared to DIMM since the boxplot shows that the first quartile for both equilibrium settings is higher for TMGD compared to DIMM. This is possibly because TMGD varies the added constant to the price (generated by the expected profit function) depending on the total number of transactions which may suggest that the trader identifies the equilibrium price and is able to sell the abstract quantity it buys.

5 Conclusion and Future Work

In conclusion, the implementation of TMGD proprietary trader was an attempt to produce an adaption to the GD sales trader since it followed the similar concept of creating a bid/ask from belief function and an overall gain. More adaptations were added as an aim to reduce code run time, increase profits and identify the equilibrium of trading sessions. While the trader comparison with DIMM possibly suggests that the TMGD identifies equilibrium price better than DIMM, is not able to generate higher profits than DIMM. To possibly increase profits of a TMGD trader, different strategy of buying and selling could be applied along with further variations of generating a bid/sell price.

References

1. BSE: Bristol Stock Exchange. GitHub public source-code repository at https://github.com/davecliff/BristolStockExchange (2012).
2. Steven Gjerstad and John Dickhaut. Price Formation in Double Auction. Games and Economic Behaviour, 22(1):1-29, 1998
3. Tesauro, G., Das, R.: High-performance Bidding Agents for the Continuous Double Auction. Proceedings of the 3rd ACM Conference on Electronic Commerce, pp.206-209, (2001).
4. Vytelingum, P.: The Structure and Behaviour of the Continuous Double Auction. PhD Thesis, School of Electronics and Computer Science, University of Southampton, UK (2006).
5. Smith, V.: An Experimental Study of Competitive Market Behavior, Journal of Political Economy 70(2):111-137, (1962).
6. Cliff, D.: Minimal-Intelligence Agents for Bargaining Behaviours in Market-Based Environments. Hewlett-Packard Labs Technical Report HPL-97-91 (1997).
7. Gode, D., Sunder, S.: Allocative efficiency of markets with zero-intelligence traders. Journal of Political Economy, 101(1):119-137, (1993).
8. Gjerstad, S., Dickhaut, J.: Price Formation in Double Auctions, Games and Economic Behavior. 22(1):1-29, (1997).
9. De Luca, M., Cliff, D.: Human-Agent Auction Interactions: Adaptive-Aggressive Agents Dominate. Proceedings IJCAI-2011, pp.178-185, (2011b)
10. Rust, J. Miller, J., Palmer, R.: Behavior of Trading Automata in a Computerized Double Auction Market. In Friedman, D., Rust, J. (Eds) The Double Auction Market: Institutions, Theories, & Evidence. Addison-Wesley, pp.155-198, (1992).

LNCS Homepage, http://www.springer.com/lncs

Appendix

```
class Trader_TMGD(Trader):

    def __init__(self, ttype, tid, balance, time):
        super().__init__(ttype, tid, balance, time)  # initialise Trader superclass
        # below are new variables for the TMGD only (they are not in the superclass)
        self.job = 'Buy'  # flag switches between 'Buy' & 'Sell' to show what DIMM does
next
        self.tid= tid
        self.ttype= ttype
        self.blotter=[]
        self.orders=[]
        #this is the transaction history recorded by the trader,only trading price is
recorded for each transaction
        self.history_transac=[]

    # the following is just a copy of GVWY's getorder method
    def getorder(self, time, countdown, lob):
        if len(self.orders) < 1:
            order = None
        else:
            quoteprice = self.orders[0].price

            self.lastquote = quoteprice
            order = Order(self.tid,
                          self.orders[0].otype,
                          quoteprice,
                          self.orders[0].qty,
                          time, lob['QID'])
        return order

    def respond(self, time, lob, trade, verbose):
        #calculates the belief function from its own history_transac list
        def payOff(price):
            if self.orders == []:
                pass
            else:
                otype = self.orders[0].otype
                #transaction history limitation is set to a 40
                if len(self.history_transac) < 40:
                    #transaction history lower than 40 sets
                    #it to the history length until the max value is reached
                    h = len(self.history_transac)
                    if otype == 'Bid':
                        success = 0.0
                        for i in range(0, h):
                            value = self.history_transac[i]
                            if value <= price:
                                success += 1
                        if h == 0:
                            return 0.0
                        else:
                            return success / h
#return b(p) that is equal to Tr(p)/m where Tr is the number of successful trades in
auction history above price p

                    if otype == 'Ask':
                        success = 0.0
                        for i in range(0, h):
                            value = self.history_transac[i]
                            if value >= price:
                                success += 1
                        if h == 0.0:
                            return 0.0
                        else:
#return b(p) that is equal to Tr(p)/m where Tr is the number of successful trades in
auction history above price p
                            return success / h
        #calculates the expected profit Ex(p) by multipying b(p) given in the
        #payOff function.
        def exp_profit(price, profit):
            if self.orders == []:
                pass
            else:
                pay_off = payOff(price)
                if pay_off == None:
                    pass
                else:
                    #payOff * gain = Ex(p)
                    return pay_off * profit
    #calculates the absolute mean difference between adjacent transactions
        def delta(history_transac):
            difference = []
            for x, y in zip(history_transac[0::], history_transac[1::]):
                difference.append(y - x)
            diff = abs(np.mean(difference))
            # printing difference list
            return diff

        if self.job == 'Buy':
            profit_price = []
            temporary = []
            # see what's on the LOB
            if lob['asks']['n'] > 0:  # and price not in temp:
                # there is at least one ask on the LOB
                for price in self.history_transac:
                    if price < self.balance and price not in temporary and
lob['asks']['n'] > 0:
                        temporary.append(price)
                        profit = (self.history_transac[len(self.history_transac)-1]) -
price
                        expected_profit = exp_profit(price, profit)
                        if expected_profit != 0:
                            profit_price.append([expected_profit, price])

                profit_price.sort(reverse=True)
                bidprice = profit_price

                if bidprice == []:
                    pass
                else:
                    #sets conditions for bestprice, bid_delta*constant is subtracted
from best possible price
                    #depending on the number of transactions. Attempts to allow the
trader to
                    #change the price depending on whether an equilibrium has been
reached
                    listBest = bidprice[0]
                    bid_delta = delta(self.history_transac)
                    if 0<len(self.history_transac)<=2:
                        bestprice = listBest[1]-bid_delta*1
                    elif 3<len(self.history_transac)<=10:
                       bestprice=listBest[1]-bid_delta*0.1
                    else:
                       bestprice=listBest[1]
                    if bestprice < self.balance:
                        #place the order on the best possible price
                        order = Order(self.tid, 'Bid', bestprice, 1, time, lob['QID'])

                        self.orders = [order]
                        if verbose: print('TMGD Buy order=%s ' % (order))

        elif self.job == 'Sell':
            profit_price = []
            temporary = []
            if lob['bids']['n'] > 0:
                for price in self.history_transac:
                    if price not in temporary and lob['bids']['n'] > 0:
                        temporary.append(price)
                        profit = price - (self.history_transac[len(self.history_transac)
- 1])
                        expected_profit = exp_profit(price, profit)
                        if expected_profit != 0:
                            profit_price.append([expected_profit, price])

                profit_price.sort(reverse=True)
                bidprice = profit_price
                if bidprice == []:
                    pass
                else:
                #sets conditions for bestprice, bid_delta*constant is added to the best
possible price
                #depending on the number of transactions. Attempts to allow the trader
to
                #change the price depending on whether an equilibrium has been reached
                    listBest = bidprice[0]
                    ask_delta = delta(self.history_transac)
                    if 0<len(self.history_transac)<=2:
                        bestprice = listBest[1]+ask_delta  # ask_delta
                    elif 3<=len(self.history_transac)<=10:
                        bestprice = listBest[1]+ask_delta*0.1
                    else:
                        bestprice = listBest[1]
                    #place the order on best possible price
                    order = Order(self.tid, 'Ask', bestprice, 1, time, lob['QID'])
                #order issued to self, processed in getorder
                    self.orders = [order]
                    if verbose: print('TMGD Buy order=%s ' % (order))
                    return order
        else:
            sys.exit('FATAL: TMGD doesn\'t know self.job type %s\n' % self.job)

        def update_current_history():
            # trading price
            price = trade['price']
            # trading quantity
            count = trade['qty']
            while count != 0:
                # most recent transaction will be inserted at the start of this list
                self.history_transac.insert(0, price)
                count -= 1

    # if there was a transaction occurred in the market, then
    # we update transaction history
        if trade != None:
            update_current_history()


    def bookkeep(self, trade, order, verbose, time):
        outstr = ""

        for order in self.orders:
            outstr = outstr + str(order)
        self.blotter.append(trade)  # add trade record to trader's blotter
        # NB What follows is **LAZY** -- it assumes all orders are quantity=1
        transactionprice = trade['price']

        bidTrans = True  # did I buy? (for output logging only)
        if self.orders[0].otype == 'Bid':
            # Bid order succeeded, remember the price and adjust the balance
            self.balance -= transactionprice
            self.last_purchase_price = transactionprice
            self.job = 'Sell'# now try to sell it for a profit
        elif self.orders[0].otype == 'Ask':
            bidTrans = False  # we made a sale (for output logging only)
            # Sold! put the money in the bank
            self.balance += transactionprice
            self.last_purchase_price = 0
            self.job = 'Buy'
            #else:
            #    self.job = 'Buy'  # now go back and buy another one
        else:
            sys.exit('FATAL: TMGD doesn\'t know .otype %s\n' %
                     self.orders[0].otype)

        self.n_trades += 1

        verbose = True  # We will log to output

        if verbose:  # The following is for logging output to terminal
            if bidTrans:  # We bought some shares
                outcome = "Bght"
                owned = 1
            else:  # We sold some shares
                outcome = "Sold"
                owned = 0
            net_worth = self.balance + self.last_purchase_price
            print('%s, %s=%d; Qty=%d; Balance=%d, NetWorth=%d' %
                  (outstr, outcome, transactionprice, owned, self.balance, net_worth))

        self.del_order(order)  # delete the order
```