



## **TOC QUINTET**

**Creating Order Matching System with Specific Type of Orders**

***Submitted by,***

**B.N.S Vishal, Kalyani .M, Yukta Bhatia, Sarmishta Jain, Pawani Trivedi**

## OBJECTIVE:

Creating live order matching software that is used for trading of **Microsoft shares** using all types of orders. The order matching software is used to match buy orders with sell orders.

The screenshot shows a web application titled "ORDER MATCHING SYSTEM" for Microsoft Corporation Common Stock (MSFT). The page displays the following market data:

- Open Price: \$210.62
- Total Volume: 26372464
- Today's High/Low: \$210.65/\$204.64
- Circuit Band: 10% (\$189.55 - \$231.68)
- Last Close Price: \$205.05

A green button labeled "Open Market" is visible below the market data.

The "Trade Sheets" section contains a table with the following columns: ORDER ID, PRICE TYPE (MARKET/ LIMIT), ORDERTYPE, PRICE (\$), QUANTITY, and TIME. The table is currently empty.

At the bottom of the page, there is an order placement form with the following fields and buttons:

- Select number of orders you'd like to place: 1 (dropdown menu)
- Send (button)
- \*Maximum 5 trades are allowed in a day (for 1 iteration)
- Order Type: BUY (dropdown menu)
- Quantity: (input field)
- Limit/ Market Price: LIMIT (dropdown menu)
- Share Price: (input field)
- Place Order (button)
- Clear (button)

A red note at the bottom states: "You're order might take a few minutes to reflect on the screen."

Figure1 - Home Page

## INTRODUCTION:

An order matching engine operates on an order book to match buyers and sellers, resulting in a series of trades. The matches happen when compatible buy orders and sell orders for the same security are submitted in proximity of price and time, a buy order and a sell order are compatible if the maximum price of the buy order matches or exceeds the minimum price of the sell order.

FIFO matching algorithm is used for this project. Under a basic FIFO algorithm, or price-time-priority algorithm, the earliest active buy order at the highest price takes priority over any subsequent order at that price, which in turn takes priority over any active buy order at a lower price and if multiple orders are present at the same price level the oldest order will be filled first. This is the same principle as a FIFO queue: first in, first filled.

Order matching algorithm is implemented using the Python programming language. Two python filenames used :

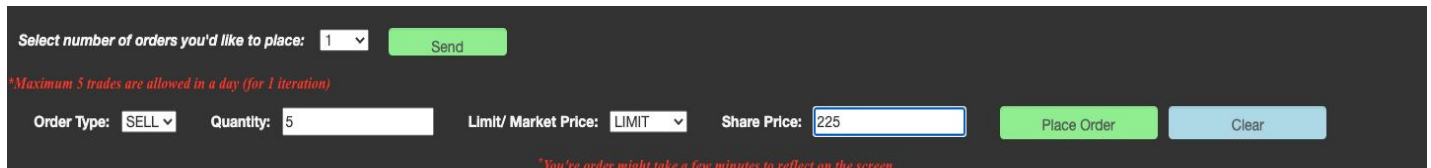
1. **Main\_trigger.py** – In this file we fetch the user input order details, initiate the random order generation and append all these details to a Python List and send it to match engine function.
2. **Matching\_engine.py** – In this file we implement the FIFO matching algorithm. we match the user input orders with the dummy orders generated. If the matching is successful, the trade details are displayed.

For the front end of the website, Angular 8 was used with Bootstrap 3 for styling.

Flask APIs were created to submit the order form using the HTTP POST method and to display real time data on the main page i.e. the last close price, open price, total volume and today's high low. Yahoo finance API was used in python to retrieve the data that was wrapped as a JSON object which was then retrievable by Angular using the HTTP GET method for which another Flask API was created.

Google sheets API was used to display our sheetbook itself on the main page, so all the dummy orders, user input generated orders, trades executed and cancelled orders were fed into the sheets in the live environment and reflected on our main page. Figure 1 shows the main page with a frame for the google sheetbook with tabs for the different sheets.

The user has been given a choice to enter how many trades he/she would like to do in the day (for 1 iteration) and only after those many orders, the main\_trigger function would call the random order generator that creates the random orders after which matching gets done.



The screenshot shows a dark-themed order form. At the top, it says "Select number of orders you'd like to place:" followed by a dropdown menu set to "1" and a green "Send" button. Below this, a red note states "\*Maximum 5 trades are allowed in a day (for 1 iteration)". The main form area includes "Order Type:" with a dropdown set to "SELL", "Quantity:" with a text input set to "5", "Limit/ Market Price:" with a dropdown set to "LIMIT", and "Share Price:" with a text input set to "225". There are two buttons at the bottom right: a green "Place Order" button and a light blue "Clear" button. A red note at the bottom center says "\*You're order might take a few minutes to reflect on the screen".

**Figure 2 - Order Form**

## ORDER GENERATOR:

The orders for order matching system are of two types:

### 1. Dummy Orders:

Random order generator is implemented in main\_trigger.py which will generate 30 random orders with order id ,order type (limit and market) , order category ( buy and sell) ,with randomly generated order price within the circuit band , order quantity and timestamp. The orders are generated with a time gap of half millisecond.

Code snippet for dummy orders generation -

```
def main_trigger(no_of_orders, midpoint=210.62):
    engine = m_engine.MatchingEngine()

    #random order generator
    for i in range(no_of_orders):
        ran_type = ["market"] * 25 + ["limit"] * 75
        type=random.choice(ran_type)
        category = random.choice(["buy", "sell"])
        if type=='limit':
            ranprice = random.gauss(midpoint,10)
            price=round(ranprice - math.fmod(ranprice, 0.05),2)
        if type=='market' and category=='sell':
            price=0
        elif type=='market' and category=='buy':
            price=math.inf
        quantity = random.randint(100,200)
```

```

        time.sleep(0.05)
        now = datetime.datetime.now()
        t=('02d:02d:02d.%d'%(now.hour,          now.minute,          now.second,
now.microsecond))[:-4]
        orders.append({'id': i, 'type': type, 'category': category, 'price': price,
'quantity': quantity, 'time': t})
    AllOrders = []
    for i in range(len(orders)):
        AllOrders.append([])
        for k,v in orders[i].items():
            if v == math.inf:
                AllOrders[i].append("inf")
            else:
                AllOrders[i].append(v)
    saveDataToGSheet.Export_AllOrders_To_Sheets(AllOrders)

```

## 2. Manual Orders:

Manual orders entry screen is used to enter the order details including order type, order category, order price and quantity. If the user selects the order category as 'limit', and enters the price in between the circuit band, the order gets accepted or else a pop up appears asking users to re-submit the form with the price in the circuit band. If he selects 'market', the price input box disappears and the last traded price is selected to be the price for the order. The user can enter a maximum of 5 trades in one iteration and 1 iteration is considered as 1 day.

Code snippet for manual orders-

```

def input_user_order(input_order, orderList):
    print("input order: ")
    print(input_order)
    id = orderList[0]
    category = orderList[1]
    type = orderList[3]
    quantity = orderList[2]
    price = orderList[4]
    time_of_new_order = orderList[5]
    if type=='market' and category=='sell':
        price=0
    elif type=='market' and category=='buy':
        price=math.inf

```

```

orders.append({'id':id,'type':type,'category':category,'price':price,'quantity':quantity, 'time': time_of_new_order})

print("order number: ")
global order_number
print(order_number)

if(order_number<input_order):
    order_number += 1
else:
    main_trigger(30)

```

The input\_order variable is the user input of how many orders he/she would like to do in the day which has to be between 1 and 5. Order\_number variable checks if the new order generated doesn't exceed that number. If it becomes equal to that number, the main\_trigger function is called where random orders get generated and then the match\_engine function is called. All new manual orders are then displayed on the NewOrders sheet and all orders - manual and dummy orders are displayed on the AllOrders sheet.

**ORDER MATCHING SYSTEM**

**MICROSOFT CORPORATION COMMON STOCK (MSFT)** Last Close Price: \$205.05

Open Price: \$210.62  
Total Volume: 26372464  
Today's High/Low: \$210.65/\$204.64  
Circuit Band: 10% (\$189.55 - \$231.68)

[Open Market](#)

**Trade Sheets**

ORDER ID	ORDERTYPE	QUANTITY	PRICE TYPE (MARKET/ LIMIT)	PRICE (\$)	TIME
30	buy	10.5	limit	215.15	12:40:24.98
31	sell	5	limit	225	12:41:18.03

[NewOrders](#) [TradeBook](#) [CancelledOrders](#) [AllOrders](#)

To view/ download the google sheet, [click here](#).

Select number of orders you'd like to place:  [Send](#)

\*Maximum 5 trades are allowed in a day (for 1 iteration)

Order Type:  Quantity:  Limit/ Market Price:  Share Price:  [Place Order](#) [Clear](#)

\*You're order might take a few minutes to reflect on the screen.

Figure 3 - New Orders sheet

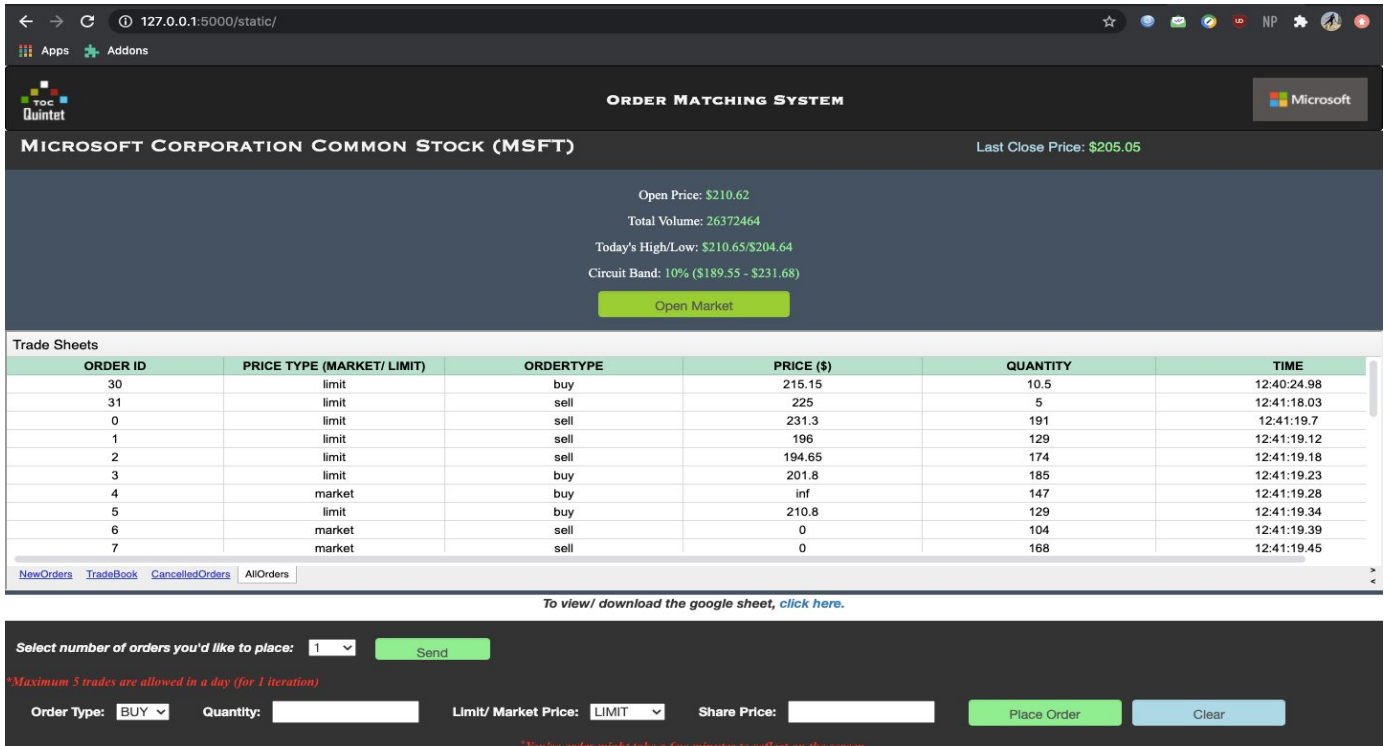


Figure 4 - All orders sheet

## ORDER MATCHING:

1. FIFO or Price/time priority, matching algorithm is implemented in this project. All orders (manual and dummy) are saved in a list called orders [].
2. Each order is sent into the matching algorithm and according to the order category they are divided into two sorted lists (bids [] and asks []). The bid list is sorted in descending order and the sell list is sorted in ascending order.
3. In price-time-priority algorithm, the earliest active buy order at the highest price takes priority over any subsequent order at that price, which in turn takes priority over any active buy order at a lower price.

4. If an order already exists in the list at the same price as the new order, then the incoming order is inserted to the right in the list using `bisect_right` inbuilt function.
5. The best sell price for a buy order is given by `best_ask ()` function and the best buy price for a sell order is given by `best_bid ()` function.
6. Market buy/sell orders are not priced and are executed immediately. Market order is implemented as a special case of limit order where the buy order price is assigned to infinity as there will always be a seller to this bid and the trade will be executed immediately and the sell order price is set to 0 for market sell . Users enter an order to buy and there are sellers at various prices and the order is executed at the lowest price.
7. The circuit band price and the tick size are checked when the user enters the price itself. If the entered price is not within the circuit band then the system throws an error and the order will not be placed.
8. The `match_order ()` function in the matching engine class implements the order matching and matched orders are removed from the order list by `remove` function and are pushed to the trades list. Cancelled order details are fetched from the bid and ask list at the end of the matching process as these orders were not successfully traded.



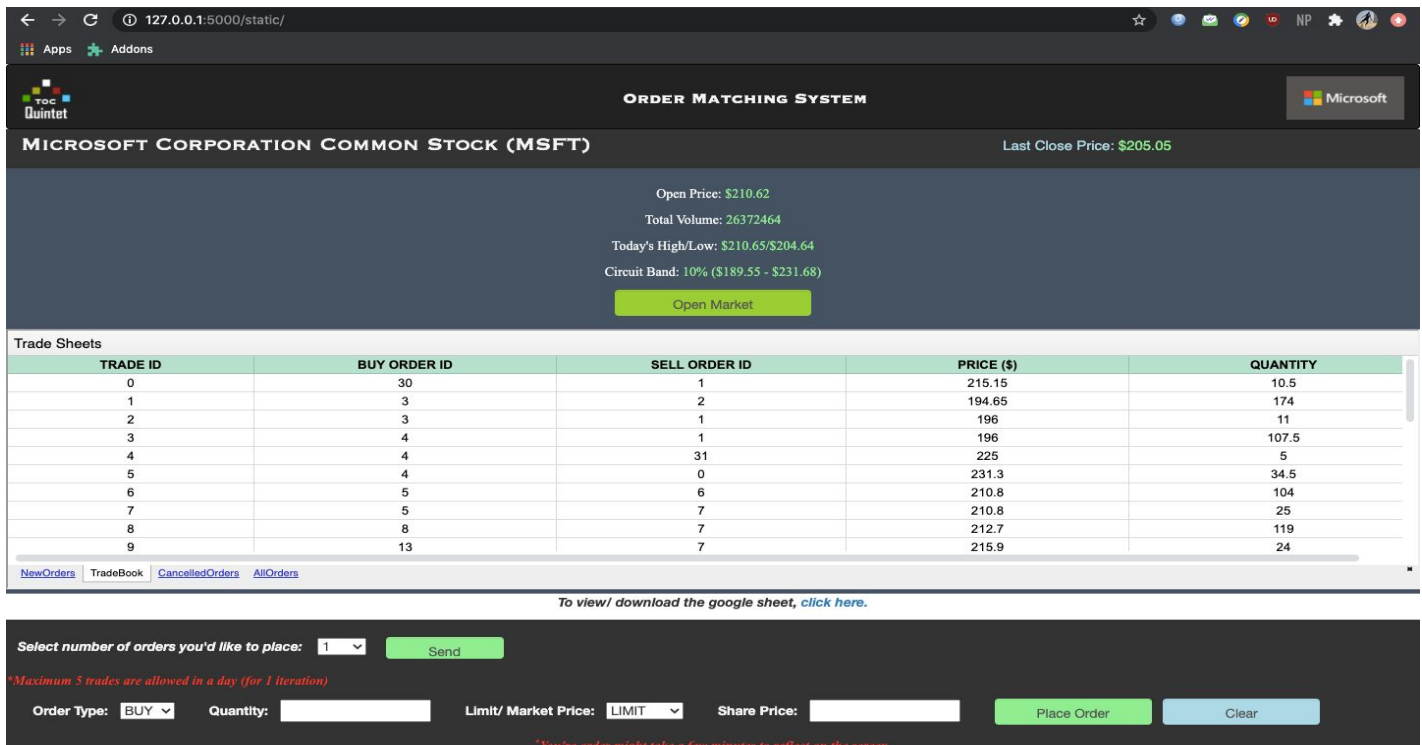


Figure 5 - TradeBook Sheets

## ORDER DISPLAY:

### 1. Trade List:

- Trade list is for orders that resulted into successful trades and each trade is identified with a unique Trade ID.
- The buy order id and sell order id with the price at which the trade is placed along with the quantity is displayed in the google sheet.
- Get\_trades () function is used to display the orders matched.

### 2. Cancelled Orders:

- Orders that are not matched are displayed in another google sheet.
- Cancel () function is used to extract the cancelled order from the lists which were left unmatched.

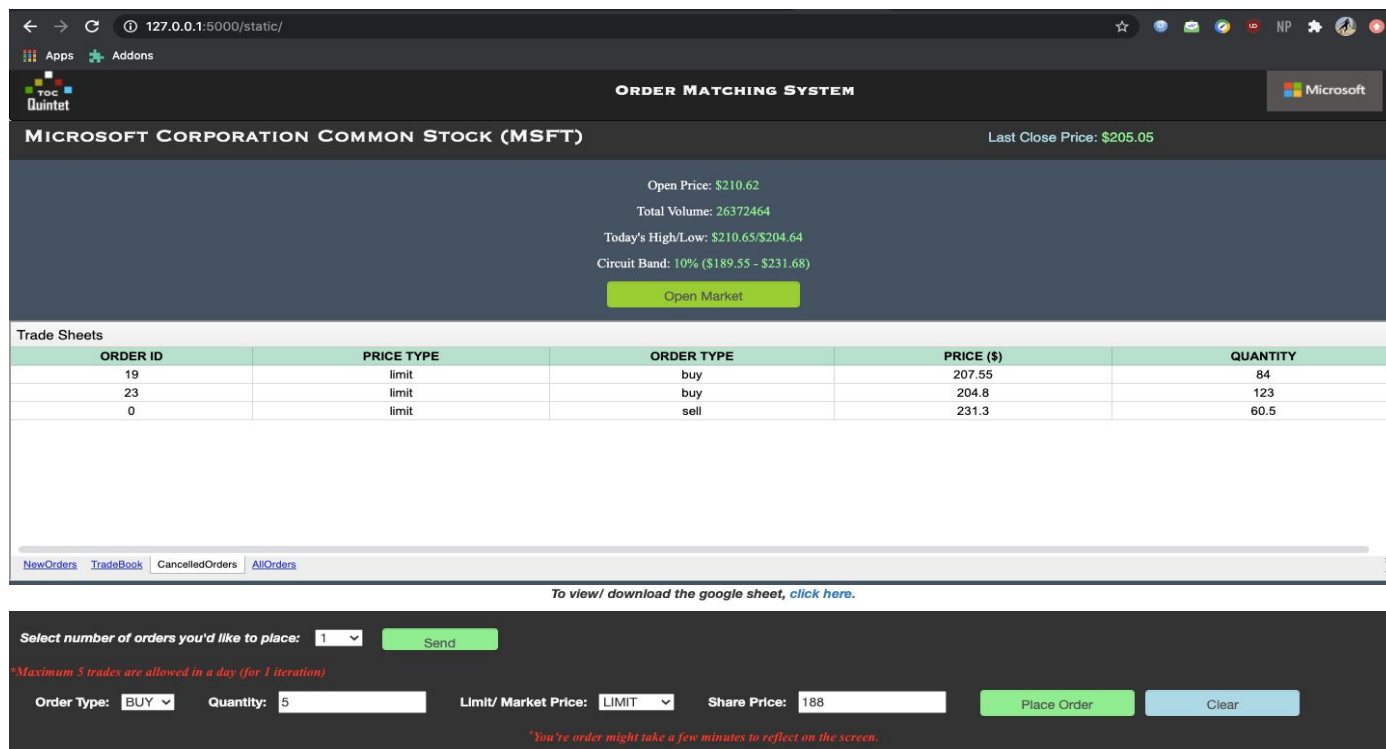


Figure 6 - Cancelled Orders Sheet

## Code for Matching Engine:

```
def match_order(self, order):
    if order.category == 'buy' and order.price >= self.orderbook.best_ask():
        # Buy order crossed the spread
        filled = 0
        consumed_asks = []
        for i in range(len(self.orderbook.asks)):
            ask = self.orderbook.asks[i]

            if ask.price == 0 and order.price == math.inf:
                break
            if ask.price > order.price:
                break # Price of ask is too high, stop filling order
            elif filled == order.quantity:
                break # Order was filled

            if filled + ask.quantity <= order.quantity: # order not yet filled,
ask will be consumed whole
```

```

        filled += ask.quantity
        if ask.price == 0:
            trade = Trade(order.id, ask.id, order.price, ask.quantity)
        else:
            trade = Trade(order.id, ask.id, ask.price, ask.quantity)
        self.trades.append(trade)
        consumed_asks.append(ask)
        elif filled + ask.quantity > order.quantity: # order is filled, ask
will be consumed partially
            volume = order.quantity - filled
            filled += volume
            if ask.price == 0:
                trade = Trade(order.id, ask.id, order.price, volume)
            else:
                trade = Trade(order.id, ask.id, ask.price, volume)
            self.trades.append(trade)
            ask.quantity -= volume

    for ask in consumed_asks:
        self.orderbook.remove(ask)

    if filled < order.quantity:

self.orderbook.add(Order(order.id, order.type, order.category, order.price, order.quantity
- filled))

    elif order.category == 'sell' and order.price <= self.orderbook.best_bid():
        # Sell order crossed the spread
        filled = 0
        consumed_bids = []
        for i in range(len(self.orderbook.bids)):
            bid = self.orderbook.bids[i]

            if order.price==0 and bid.price==math.inf:
                break
            if bid.price < order.price:
                break # Price of bid is too low, stop filling order
            if filled == order.quantity:
                break # Order was filled

            if filled + bid.quantity <= order.quantity: # order not yet filled,
bid will be consumed whole

```

```

        filled += bid.quantity
        if bid.price == math.inf:
            trade = Trade(bid.id, order.id, order.price, bid.quantity)
        else:
            trade = Trade(bid.id, order.id, bid.price, bid.quantity)
        self.trades.append(trade)
        consumed_bids.append(bid)
        elif filled + bid.quantity > order.quantity: # order is filled, bid
will be consumed partially
            volume = order.quantity - filled
            filled += volume
            if bid.price == math.inf:
                trade = Trade(bid.id, order.id, order.price, volume)
            else:
                trade = Trade(bid.id, order.id, bid.price, volume)
            self.trades.append(trade)
            bid.quantity -= volume

    for bid in consumed_bids:
        self.orderbook.remove(bid)

    if filled < order.quantity:

self.orderbook.add(Order(order.id, order.type, order.category, order.price, order.quantity
- filled))

    else:
        # Order did not cross the spread, place in order book
        self.orderbook.add(order)

```

## CONCLUSION :

In this project, an Order matching system for Microsoft Shares is implemented using Python and Angular . The order matching software is used to match buy orders with sell orders and allows users to place market orders or limit orders with a circuit band of 10% using FIFO algorithm.

### ***Thank you note..***

It was a wonderful experience in putting theoretical learning about finance into practical and technical use and it was fun collaborating with other grads virtually, working together on this project and discussing work!

We thank UBS and Imarticus for this wonderful learning experience!

Regards

Team TOC Quintet