Matching engines

An *order matching engine* operates on a limit order book to match buyers and sellers, resulting in a series of trades. It is the mechanism behind price movement; the price at which the last trade was executed usually determines the exchange rate for whatever security is being traded.

Limit order books

All incoming orders are passed on to the matching engine, which then tries to match them against the passive orders in the limit order book (LOB). The book contains all limit orders for which no matches have been found as of yet, divided in a bid side (sorted in ascending order) and an ask side (sorted in descending order). If no matches can be found for a new order it will also be stored in the order book, on the appropriate side.

A limit order book is usually summarized by the following characteristics:

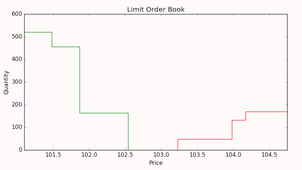
- Bid: The highest price against which a sell order can be executed
- Ask: The lowest price against which a buy order can be executed
- Spread: The difference between the lowest ask and the highest bid
- Midpoint: The price halfway between the ask and the bid ((ask+bid)/2)

Suppose we have the following LOB:

```
           LIMIT ORDER BOOK
     BID SIDE           ASK SIDE
 QUANTITY   PRICE   PRICE   QUANTITY
[131.00  -  102.54 | 103.23  -  48.00]
[32.00   -  101.87 | 103.98  -  84.00]
[293.00  -  101.48 | 104.17  -  38.00]
[65.00   -  101.18 | 104.75  -  127.00]
```

The bid is 102.54, while the ask is 103.23. The spread (ask-bid) is 0.69. The midpoint ((ask+bid)/2) is then 102.885.

The following (often used) visualisation condenses all the LOB information into one simple graph:



Matching algorithms

The most common matching algorithm is called '**Price/time priority**'. Orders in the LOB are filled primarily based on price; 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.

As an example we will use the previously used limit order book and simulate some incoming orders.

Suppose two orders come in right behind each other; the first one a limit buy order for 24 shares at $102.55 and the second one also a limit buy order for 14 shares at the same price. Seeing as the orders don't match with any asks (due to their prices being lower than the lowest ask) they are both placed in the limit order book. The first order and the second order are stored at the same price level, but the former has priority over the latter due to time priority. This basically means that the first order will be placed on top of the second order in the bid queue.

```
     BID SIDE           ASK SIDE
 QUANTITY   PRICE   PRICE   QUANTITY
[24.00   -  102.55 | 103.23  -  48.00]
[14.00   -  102.55 | 103.98  -  84.00]
[131.00  -  102.54 | 104.17  -  38.00]
[32.00   -  101.87 | 104.75  -  127.00]
```

Suppose another order comes in, a limit sell order this time, for 40 shares at $102.55. This sell order clearly should have some matches; its price is lower than the highest bid. The matching engine will then use the first two bids at price level $102.54 to fill the incoming order for 38 shares, after which it stops filling due to the limit price (even though the incoming order still has 2 shares left to be filled). The remaining order for two shares is then stored in the limit order book at the limit price, as shown here:

```
     BID SIDE           ASK SIDE
 QUANTITY   PRICE   PRICE   QUANTITY
[131.00  -  102.54 | 102.55  -   2.00]
[32.00   -  101.87 | 103.23  -  48.00]
[293.00  -  101.48 | 103.98  -  84.00]
[65.00   -  101.18 | 104.17  -  38.00]
```

Note: I have ignored *market orders* in this section. That is because a market order is in fact just a special case of limit orders. They are supposed to keep on going until they are completely filled, so it may seem like they do not have a limit price at which the order will halt. They do, in fact, but the limit prices are set high/low enough so that they will most likely never be reached while filling the order. So, in order to simulate a market order with a limit order you can just set the limit price either to 0 (for a market sell order) or to +infinity (for a market buy order).

Python implementation

We will need to implement data structures for orders, trades and the limit order book before we can implement a matching engine.

An order is simply an object with price, quantity, side (bid/ask) and order type attributes. A trade can be implemented with only a price and a quantity.

```
class Order:

    def __init__(self, order_type, side, price, quantity):
        self.type = order_type
        self.side = side.lower()
        self.price = price
        self.quantity = quantity

class Trade:

    def __init__(self, price, quantity):
        self.price = price
        self.quantity = quantity
```

A limit order book can be easily implemented as a data structure with two sorted lists containing order instances sorted by price; one sorted in ascending order (bids) and one sorted in descending order (asks).

```
import sortedcontainers

class OrderBook:

    def __init__(self, bids=[], asks=[]):
        self.bids = sortedcontainers.SortedList(bids, key = lambda order: -order.price)
        self.asks = sortedcontainers.SortedList(asks, key = lambda order: order.price)

    def __len__(self):
        return len(self.bids) + len(self.asks)

    def add(self, order):
        if order.direction == 'buy':
            self.bids.insert(self.bids.bisect_right(order), order)
        elif order.direction == 'sell':
            self.asks.insert(self.asks.bisect_right(order), order)

    def remove(self, order):
        if order.direction == 'buy':
            self.bids.remove(order)
        elif order.direction == 'sell':
            self.asks.remove(order)

    def plot(self):
        fig = plt.figure(figsize=(10,5))
        ax = fig.add_subplot(111)
        ax.set_title("Limit Order Book")

        ax.set_xlabel('Price')
        ax.set_ylabel('Quantity')

        # Cumulative bid volume
        bidvalues = [0]
        for i in range(len(self.bids)):
            bidvalues.append(sum([self.bids[x].quantity for x in range(i+1)]))
        bidvalues.append(sum([bid.quantity for bid in self.bids]))
        bidvalues.sort()

        # Cumulative ask volume
        askvalues = [0]
        for i in range(len(self.asks)):
            askvalues.append(sum([self.asks[x].quantity for x in range(i+1)]))
        askvalues.append(sum([ask.quantity for ask in self.asks]))
        askvalues.sort(reverse=True)

        # Draw bid side
        x = [self.bids[0].price] + [order.price for order in self.bids] + [self.bids[-1].price]
        ax.step(x, bidvalues, color='green')

        # Draw ask side
        x = [self.asks[-1].price] + sorted([order.price for order in self.asks], reverse=True) +
        ax.step(x, askvalues, color='red')

        ax.set_xlim([min(order.price for order in self.bids), max(order.price for order in self.a
        plt.show()
        if save:
            fig.savefig('plot.png', transparent=True)
```

Implementing the matching engine is a bit more difficult. First of all we will need two FIFO queues; one to store all incoming orders and one to store all resulting trades. We will also need a limit order book to store all orders that didn't match. This implementation also includes a threading option, which is not strictly necessary for the basic functionality.

```
from threading import Thread
from collections import deque

class MatchingEngine:

    def __init__(self, threaded=False):
        self.queue = deque()
        self.orderbook = OrderBook()
        self.trades = deque()
        self.threaded = threaded
        if self.threaded:
            self.thread = Thread(target=self.run)
            self.thread.start()
```

Orders are passed on to the engine by calling the *.process(order)* function. The resulting trades are then stored in a queue, which can then be retrieved sequentially (by iterating over the engine trade queue) or in a list by calling the *.get_trades()* function.

```
    def process(self, order):
        if self.threaded:
            self.queue.append(order)
        else:
            self.match(order)

    def get_trades(self):
        trades = list(self.trades)
        return trades
```

The matching logic looks complicated but is actually rather simple. It basically loops through the orderbook until the incoming order is completely filled. For every fill event a trade object is created and added to the list of trades. If the matching engine was not able to completely fill the order then it adds the remaining volume to the limit order book as a separate order.

```
    def match(self, order):
        if order.side == '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 > 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 c
                    filled += ask.quantity
                    trade = Trade(ask.price, ask.quantity)
                    self.trades.append(trade)
                    consumed_asks.append(ask)
                elif filled + ask.quantity > order.quantity: # order is filled, ask will be consu
                    volume = order.quantity-filled
                    filled += volume
                    trade = Trade(ask.price, volume)
                    self.trades.append(trade)
                    ask.quantity -= volume

            # Place any remaining volume in LOB
            if filled < order.quantity:
                self.orderbook.add(Order("limit", "buy", order.price, order.quantity-filled))

            # Remove asks used for filling order
            for ask in consumed_asks:
                self.orderbook.remove(ask)

        elif order.side == '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 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 c
                    filled += bid.quantity
                    trade = Trade(bid.price, bid.quantity)
                    self.trades.append(trade)
                    consumed_bids.append(bid)
                elif filled + bid.quantity > order.quantity: # order is filled, bid will be consu
                    volume = order.quantity-filled
                    filled += volume
                    trade = Trade(bid.price, volume)
                    self.trades.append(trade)
                    bid.quantity -= volume

            # Place any remaining volume in LOB
            if filled < order.quantity:
                self.orderbook.add(Order("limit", "sell", order.price, order.quantity-filled))

            # Remove bids used for filling order
            for bid in consumed_bids:
                self.orderbook.remove(bid)
        else:
            # Order did not cross the spread, place in order book
            self.orderbook.add(order)
```

In order to run the engine as a separate thread, simply call the *.run()* function. Orders can be passed on the the engine by adding them to the engine order queue. Any resulting trades can be retrieved by continually checking the trades queue for new trades.

```
    def run(self):
        while True:
            if len(self.queue) > 0:
                order = self.queue.popleft()
                self.match(order)
```