For this project, two additional options were added to the main menu of the simulation:

- Let Bot trade this timestamp, which lets the bot analyze and do trades only for the current time frame, so that the simulation can be run manually while still letting the bot trade. Both the bot and the manual user share the same wallet.

- Let Bot take over the rest of the simulation, which sets it to automated mode and iterates through all the time frames until it goes back to the first, analyzing and trading in every one of them, and generating a log file in the end. This method is the main focus of the assessment.

Market Analysis (R1)

The bot uses a Strategy object for each different product, which is the `StandardDeviationStrategy` class, found in the corresponding files. This strategy accumulates the orders received in each timestamp and keeps track of a given number of them, as configured in its constructor. The orders are passed by the bot, which receives the `orderBook` class object and uses it to retrieve the current orders (the `MerkelBot::processNewTimestamp()` function):

```cpp
void MerkelBot::processNewTimestamp(OrderBook& orderBook,
std::list<std::shared_ptr<OrderBookEntry>>& listOfOrdersPlaced)
{
    std::string currentTimestamp = orderBook.getCurrentTimestamp();

    // This **MUST** be a reference pair or it will create copies of the strategy items,
    // so they'll never get updated
    for (auto& pair : productStrategies)
    {
        // Set aliases for our pair for easier understanding
        std::string product = pair.first;
        StandardDeviationStrategy& strategy = pair.second;

        // Clean pending orders and insert them into the withdrawnOrders list
        std::list<OrderBookEntry> withdrawnOrders =
            strategy.cleanPendingOrders(currentTimestamp);

        // Record all the withdrawn orders into our log
        for (OrderBookEntry& orderWithdrawn : withdrawnOrders)
        {
            recordWithdrawnOrder(orderWithdrawn);
        }

        // Get all the orders from the current timestamp in the orderbook
        std::list<std::shared_ptr<OrderBookEntry>> currentAsks =
            orderBook.getCurrentOrders(OrderBookType::ask, product);
        std::list<std::shared_ptr<OrderBookEntry>> currentBids =
            orderBook.getCurrentOrders(OrderBookType::bid, product);

        // Pass them to our strategy to update its metrics and then make the necessary
        // trades, if any, which will be inserted into the listOfOrdersPlaced that we
        // are passing by reference
        strategy.updateMetrics(currentAsks, currentBids);
        strategy.tradeCurrentTimestamp(listOfOrdersPlaced);
    }
}
```

With these orders, it will calculate the mean, variance, and standard deviations of both the ask and bids separately (following a linear regression model). Then, it will find the lowest priced ask and the highest priced bid in each timestamp, and if they are far enough from the mean (these are the `isAskSignal()` and `isBidSignal()` functions within the strategy), it will place a bid against the lowest ask and an ask against the highest bid, matching their price:

```
bool StandardDeviationStrategy::isAskSignal()
{
    // If highest bid before current one is lower (price is on the rise), and current
    // one has crossed the +1 deviation, then we sell.
    return highestBids[1]->price < highestBids[0]->price &&
           highestBids[0]->price >= bidsMean + bidsOneStdDeviation;
}
```

```cpp
bool StandardDeviationStrategy::isBidSignal()
{
    // If lowest ask before current one is higher (price is falling), and current one
    // has crossed the -1 deviation, then we buy.
    return lowestAsks[1]->price > lowestAsks[0]->price &&
           lowestAsks[0]->price <= asksMean - asksOneStdDeviation;
}
```

Following this model, a price that is found in +- 1 standard deviation of the mean starts to become an outlier, and thus is a good point to trade, as it is expected to regress towards the trend sooner or later.

Bidding and Buying Functionality

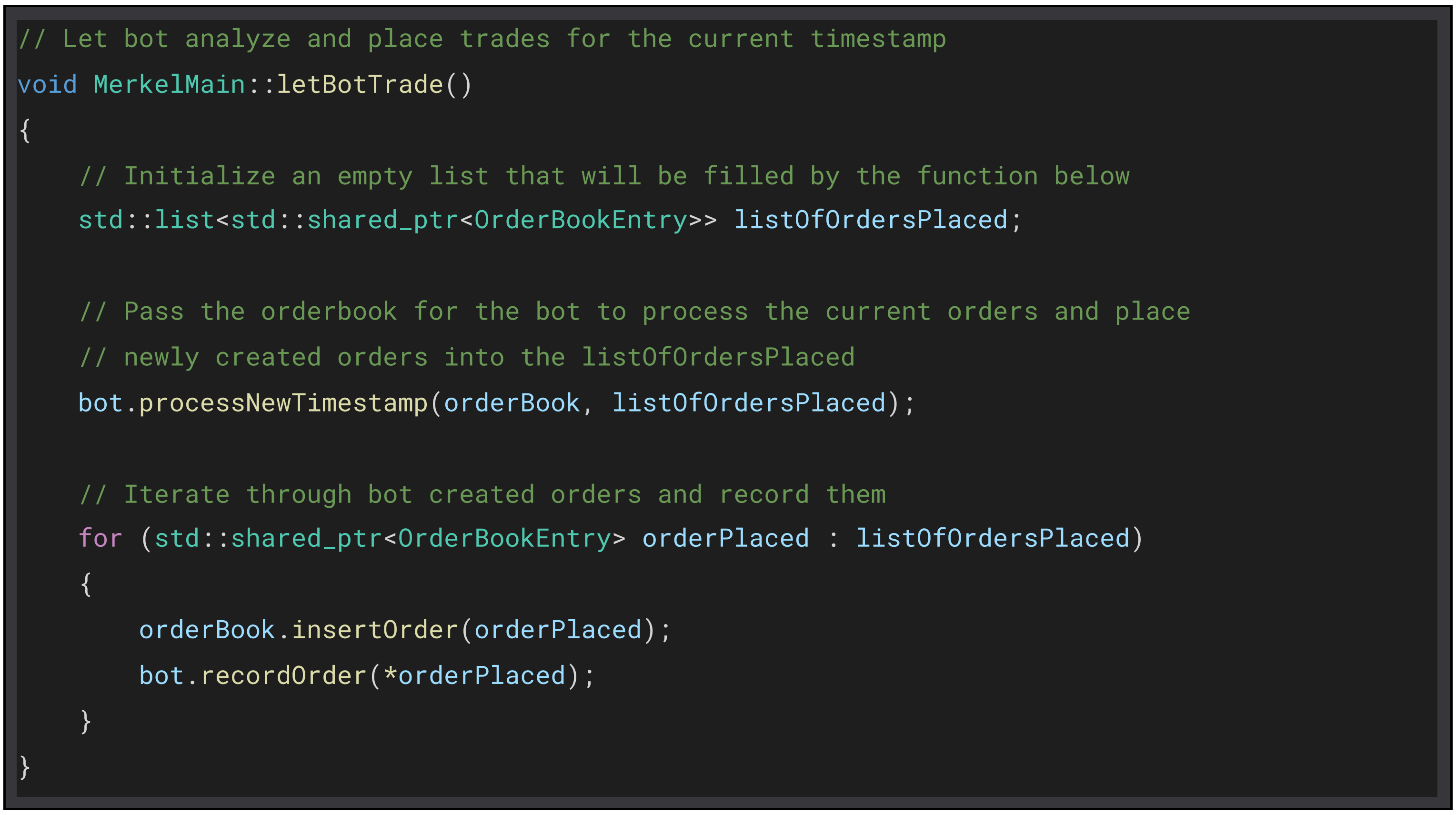
As stated above, bids will be placed when the lowest-priced ask within a timestamp hits a price that is below one standard deviation of the mean of the trend so far. The bid will be constructed to match that same price, but only bidding for 20% of the total amount it could afford. This is to ensure that not all currency goes into a single order, to respect the maxim to not keep all eggs in the same basket. To ensure that the strategies don't spend more currency than what they have available while orders are still pending resolution, a function `Wallet::lockCurrency()` was implemented, which locks away funds invested into pending orders:

```cpp
// Lock away some funds so they do not get invested while they are already used in a
// pending order
void Wallet::lockCurrency(std::string type, double amount)
{
    // If no funds of this type exist in our wallet, ignore
    if (currencies.count(type) == 0)
        return;

    if (currencies[type] < amount - 0.000001f)
    {
        std::cout << "Exception: Tried to lock " << to_string_with_precision(amount, 20)
            << " " << type << ", have " << to_string_with_precision(currencies[type], 20)
            << std::endl;
        throw std::exception{};
    }

    // Check if the type of currency exists in the map and initialize it if not
    if (lockedCurrencies.count(type) == 0)
        lockedCurrencies[type] = 0;

    currencies[type] -= amount;
    lockedCurrencies[type] += amount;
}
```

The constructed bids are assigned to a list passed to the function by reference, since it might not always decide to return an order if the time is not right. This can be seen in the `StandardDeviationStrategy::placeBid()` function, which itself is called by `StandardDeviationStrategy::tradeCurrentTimestamp()`:

```cpp
// Place a bid using the lowest ask price we can find
void StandardDeviationStrategy::placeBid(
    std::list<std::shared_ptr<OrderBookEntry>>& listOfOrdersPlaced
)
{
    // Get our current funds to bid with
    double bidFunds = wallet->getCurrentFunds(bidFundsCurrency);

    // Get the price that we need to bid at
    double priceToBidAt = lowestAsks[0]->price;

    // Determine the amount to buy based on our funds, and always buy only 20% of the full
    // amount
    double amountToBuy = std::min(lowestAsks[0]->amount, bidFunds / priceToBidAt) * .2f;
    double totalPriceToPay = amountToBuy * priceToBidAt;

    // To avoid float point precision, we will ignore any trades which would involve tiny
    // amounts
    if (amountToBuy <= 0.000000001f)
        return;

    std::shared_ptr<OrderBookEntry> pointerToBid = std::make_shared<OrderBookEntry>(
        priceToBidAt,
        amountToBuy,
        lowestAsks[0]->timestamp,
        product,
        OrderBookType::bid,
        "bot"
    );

    // Lock the currency used for this bid so we don't accidentally use it again in a
    // different order
    wallet->lockCurrency(bidFundsCurrency, totalPriceToPay);

    // Push our order into the list passed by reference
    listOfOrdersPlaced.push_back(pointerToBid);
    pendingBids.push_back(pointerToBid);
}
```

It is this function that the bot calls each timestamp to each strategy it contains, in `MerkelBot::processNewTimestamp()`, to gather the entire list of new orders that have been placed. `MerkelMain::letBotTrade()` is the caller, where the order list is finally passed over to the OrderBook class to be inserted in the list:



```cpp
// Let bot analyze and place trades for the current timestamp
void MerkelMain::letBotTrade()
{
    // Initialize an empty list that will be filled by the function below
    std::list<std::shared_ptr<OrderBookEntry>> listOfOrdersPlaced;

    // Pass the orderbook for the bot to process the current orders and place
    // newly created orders into the listOfOrdersPlaced
    bot.processNewTimestamp(orderBook, listOfOrdersPlaced);

    // Iterate through bot created orders and record them
    for (std::shared_ptr<OrderBookEntry> orderPlaced : listOfOrdersPlaced)
    {
        orderBook.insertOrder(orderPlaced);
        bot.recordOrder(*orderPlaced);
    }
}
```

After the matching is done in the `OrderBook::matchCurrentAsksToBids()`, the resulting sales are iterated through, and those which were placed by the bot originally get processed in the wallet to discount the currency (in `Wallet::processSale()`), and logged in the bot (in `MerkelBot::recordSale()`).

Bids are potentially withdrawn at the start of each new timestamp. The function `StandardDeviationStrategy::cleanPendingOrders()`, called by `MerkelBot::processNewTimestamp()` before any new trading happens, iterates through all pending orders and removes them if their amount is now 0. If not, it will check whether their price is still better than the currently known mean. If not, it will withdraw them, as they would not be profitable otherwise.

Offering and Selling Functionality

The process of generating, processing and withdrawing asks goes hand in hand with the one for bids, as all the code is found in the same places and follows the same general algorithm (but looking for high priced bids to match with an ask). The function that decides whether this happens is `StandardDeviationStrategy::isAskSignal()`, and once it returns true, `StandardDeviationStrategy::placeAsk()` is called. As with bids, only 20% of the total possible amount will be committed to the ask order:

```
// Place an ask using the highest bid price we can find
void StandardDeviationStrategy::placeAsk(
    std::list<std::shared_ptr<OrderBookEntry>>& listOfOrdersPlaced
)
{
    // Get our current funds to ask with
    double askFunds = wallet->getCurrentFunds(askFundsCurrency);

    // Get the price that we need to ask
    double priceToAsk = highestBids[0]->price;

    // Determine the amount to sell based on our funds, and always sell only 20% of the full
    // amount
    double amountToSell = std::min(askFunds, highestBids[0]->amount) * .2f;

    // To avoid float point precision, we will ignore any trades which would involve tiny
    // amounts
    if (amountToSell <= 0.000000001f)
        return;

    // Create our ask object
    std::shared_ptr<OrderBookEntry> pointerToAsk = std::make_shared<OrderBookEntry>(
        priceToAsk,
        amountToSell,
        highestBids[0]->timestamp,
        product,
        OrderBookType::ask,
        "bot"
    );
```

```cpp

    // Lock the currency used for this ask so we don't accidentally use it again in a
    // different order
    wallet->lockCurrency(askFundsCurrency, amountToSell);

    // Push our order into the list passed by reference
    listOfOrdersPlaced.push_back(pointerToAsk);
    pendingAsks.push_back(pointerToAsk);
}
```

Logging

Each placed, withdrawn and fulfilled order is reported to the bot, who will keep an `activityLog`, a vector of strings, to keep track of all those actions to date. The strings are inserted in proper formatting through the `MerkelBot::addToLog()` helper function:

```cpp
// Helper function to properly format new log entries. Each element in the received vector is
// a string which is meant to be separated from others by a tab character
void MerkelBot::addToLog(std::vector<std::string>& columns)
{
    std::string formattedStr = "";

    for (int i = 0; i < columns.size(); i++)
    {
        // When i = 1, we are logging the type of order, so no need to resize the strings to
        // a large size
        if (i == 1)
            columns[i].resize(9, ' ');

        // When i = 2, we are logging the product of the order, so no need to resize the
        // strings to a large size
        else if (i == 2)
            columns[i].resize(11, ' ');

        // Otherwise we are dealing with longer strings like the timestamp or what was
        // offered and received, so we need a size of 27 characters between each tab
        else if (columns[i].size() < 27)
            columns[i].resize(27, ' ');

        // Put all the resized "column" strings together into a single string
        formattedStr += columns[i] + "\t";
    }

    // Push the newly formatted string into the activityLog
    activityLog.push_back(formattedStr);
}
```

In turn, this is called by `MerkelBot::recordOrder()`, `MerkelBot::recordWithdrawnOrder()` and `MerkelBot::recordSale()`, which pass a vector of strings with all the necessary information to be formatted and then added to the activityLog.

Once the bot is done trading the entire simulation, the function `MerkelBot::writeLogToFile()` gets called, and a text file is created. This text file begins with the initial wallet and currencies that the bot began with in the simulation, then the history of all the order movements, and ends with the final state of the wallet after it, for an easy comparison of what it lost and what it earned:

```cpp
// Iterate through our entire activity log and write every line into a log file in the
specified path
void MerkelBot::writeLogToFile(std::string path)
{
    // Create stream to the file
    std::ofstream logFile(path + "bot_trade_log.txt");
    auto it = activityLog.begin();

    // Iterate through all the activityLog entries
    while (it != activityLog.end())
    {
        // Fetch the record at the front, stream it into the file and then erase the record
        // from the list to continue iterating
        std::string record = activityLog.front();
        logFile << record + "\n";
        it = activityLog.erase(it);
    }

    // Add the state of the final wallet to compare it to what the bot started with
    logFile << "\n\nFinal wallet: \n\n" << wallet->toString() << "\n";

    // Close the file after we're done streaming
    logFile.close();
}
```

Performance Optimizations

Many changes were made to the base program in order to improve not only performance, but also functionality:

- For the sake of keeping track of placed orders and withdrawing them, the list of orders was changed to use smart pointers to orders, as otherwise any action to insert orders into any container would create copies and lose references, making it impossible to withdraw orders from the `StandardDeviationStrategy` object. As a result, fewer copies of orders are being made in each time frame as well, improving the overall performance:

```
// Use shared smart pointers to the objects so we don't have to worry about freeing the
// memory manually
std::list<std::shared_ptr<OrderBookEntry>> CSVReader::readCSV(std::string csvFilename)
{
    // Our list to store entries
    std::list<std::shared_ptr<OrderBookEntry>> entries;

    std::ifstream csvFile{csvFilename};
    std::string line;

    if (csvFile.is_open())
    {
        while(std::getline(csvFile, line))
        {
            try
            {
                // Create our entry and assign a pointer to it
                std::shared_ptr<OrderBookEntry> entryPointer = stringsToOBE(
                    tokenise(line, ',') );
                entries.push_back(entryPointer);
            }

            catch(const std::exception& e)
            {
                std::cout << "CSVReader::readCSV read bad data." << std::endl;
            }
        }
        csvFile.close();
    }

    std::cout << "CSVReader::readCSV read " << entries.size() << " entries." << std::endl;
    return entries;
}
```

- The vector containing the orders was turned into a list instead. This allows for much faster reordering of elements where needed, and won't suffer from memory allocation performance hits if the vector becomes too big that it needs to be copied to a different memory location (since vectors are stored sequentially, whereas lists are not).

- Since the list still contains over a million elements, the first time it is iterated through at the beginning of the program is used to find the indexes at which the timestamps change, to very quickly access each new timestamp with the stored iterators. `OrderBook::timestampIterators` is a vector of iterators, each pointing to the first element in a new timestamp, and `OrderBook::currentPositionInTimestampIterators` stores the current index where the simulation is at. This way, going to the next time frame and retrieving the new orders is as simple as incrementing the index and retrieving the new iterator.

  During this initial iteration, products are also stored inside a vector, knownProducts, for faster iteration through them.

- The matching engine will now consider not only the current timestamp's orders, but also all the past ones, which considerably increases the amount of processing to be done, but also the number of sales.

- An attempt was made to not sort all the past orders when new ones were inserted into them, thus keeping them sorted ascendingly and descendingly only within their own timeframe, but this proved as slow or slower than sorting all orders till the current timestamp in every timestamp. This is because if we have all orders sorted, we can break early out of the matching loop once we get to the point where the price of our bid becomes lower than the asks, since we know it will never get matched:

```
    while (i != bids.end())
    {
        while (j != asks.end())
        {
            // Break, as the provided lists are sorted, so every next iteration
            // will result in unmatched orders, since the bid is now lower than the asks
            if ((*i)->price < (*j)->price)
                break;

            // Valid ask and bid, match them together to see if a sale is produced
            matchAskToBid(*j, *i, matches);

            // Remove empty ask and continue, since this bid still might have some amount
            // left
            if ((*j)->amount <= 0.000000001f)
                j = asks.erase(j);

            // If ask not empty, continue with next ask
            else j++;

            // Remove empty bid, which sets the iterator to the next bid,
            // and restart the asks loop for this new bid
            if ((*i)->amount <= 0.000000001f)
            {
                i = bids.erase(i);
                j = asks.begin();
            }
        }

        // Move to next bid and reset the pointer at the beginning of the asks
        i++;
        j = asks.begin();
    }
```

- Additionally, only the new set of bids and asks per each product is matched to the past set of bids and asks, since it’s unnecessary to match all the bids to all the asks again, including the older ones. If they were not matched before, they will not be matched now.

With all these optimizations, it takes ~443.43 seconds (~7 minutes) for the bot to complete the entire simulation, logging included (using the -O2 argument on compiling and running without debug mode). A log file for these trading results has been provided.

An alternative sorting function was written (`OrderBook::insertIntoSortedList()`). However, it proved slower than the standard `insert()` and `sort()` methods by a long shot (~940 seconds, ~15 minutes). The code was left commented out for reference.

Trading through the entire simulation only considering each current timestamp, without trading with older orders, now takes around ~7 seconds (less without the timestamp console prints, and using the -O2 argument on compiling and running without debug mode). Interestingly, this results in more effective trading by the bot because no other orders get matched together (since the dataset provided contains no matches of orders within each timestamp). In this case, however, the withdrawal of orders does not make much sense. Another log file for this result has been provided.

Algorithms

The trading algorithm is split into two parts: analyzing the new orders and building up the metrics (in `StandardDeviationStrategy::updateMetrics()`), and determining whether ask and bid signals are found, then placing orders (in `StandardDeviationStrategy::tradeCurrentTimestamp()`).

`StandardDeviationStrategy::updateMetrics()` algorithm

1. Insert the received asks and bids into the `observedAsks` and `observedBids` lists.

2. If the size of either list exceeds the limit set by `nbrOfEntriesToObserve`, remove the excess orders from the beginning of the lists (the older ones).

3. Calculate the new mean, variance, and standard deviations of the lists with the new elements.

4. Find the lowest ask and highest bid among the newly received ones, and insert them at the front of `lowestAsks` and `highestBids`. These will be used as reference for the trade signals.

`StandardDeviationStrategy::tradeCurrentTimestamp()` algorithm

1. Get the wallet's current funds for asks and bids of the strategy's product.

2. Check if we have accumulated enough metrics to start trading by comparing the size of our `observedAsks` and `observedBids` against the `minimumNbrOfObservedEntriesToTrade`.

3. Check if there is a bid signal found and if there are enough funds to place bids. A bid signal is found if the current time frame's lowest ask is lower than the previous one (the trend is going low) and if it is lower than one standard deviation less than the mean. The reasoning is that an order with such a price is an outlier in the overall trend, and will eventually get normalized towards a meaner price, at which point profit can be made.

    a. If so, place a bid against said lowest ask, matching the price, and requesting only a 20% of the amount that can be afforded. These funds are then locked in the wallet, in `Wallet::lockCurrency()`, since they won't actually get removed until a sale is made and we don't want them to be committed to other trades.

4. Check if there is an ask signal found and if there are enough funds to place asks. An ask signal is found if the current time frame's highest bid is higher than the previous one (the trend is rising) and if it is higher than one standard deviation more than the mean. The reasoning is that an order with such a price is an outlier in the overall trend, and will eventually get normalized towards a meaner price, at which point profit can be made.

    a. If so, place an ask against said highest bid, matching the price, and selling only a 20% of the amount that we have (or that the bid requests). These funds are then locked in the wallet, in `Wallet::lockCurrency()`, since they won't actually get removed until a sale is made and we don't want them to be committed to other trades.

5. If neither 3. nor 4. occur, then no trade is made in this time frame.

`StandardDeviationStrategy::cleanPendingOrders()` algorithm

1. Iterate through the placed Asks.

2. If the amount is equal or less than 0, erase it.

3. If the price is higher than the current mean minus 1 deviation:

    a. Unlock the funds used in what's left of the order.
    b. Log the withdrawal.
    c. Withdraw the order.

4. Repeat 1-3 with the Bids.

`OrderBook::matchCurrentAndPreviousOrders()` algorithm

1. Get the current timestamp's orders for the given product.

2. Create a new list of `OrderBookEntries` where the sales will be stored. This list does not use pointers, as it will use the orders as read-only values, and is not meant to alter anything.

3. Sort the new asks and bids, in ascending and descending order respectively.

4. Do our `mapOfAsksTillCurrent` and `mapOfBidsTillCurrent` already have an existing key?

    a. If not, initialize one.
    b. Otherwise, move on.

5. Set an alias for each map's key/list pair, pointing to the `asksTillCurrent` and `bidsTillCurrent` lists, for easier and more readable access in the rest of the function.

6. Call `OrderBook::matchAsksToBids() to` match all the asks until (but not including) the current ones with only the current bids, and insert any given sales inside our sales list.

7. Call `OrderBook::matchAsksToBids()` match only the current asks to the bids until (but not including) the current ones, and insert any given sales inside our sales list.

8. Insert the current asks and bids into the `asksTillCurrent` and `bidsTillCurrent` lists for the next time frame iteration.

9. Sort the entire `asksTillCurrent` and `bidsTillCurrent` lists ascendingly and descendingly, so the timestamp at which the orders were placed has no influence on their matching.

`OrderBook::matchAsksToBids()` algorithm

1. Declare an iterator for both asks and bids, at the start of the lists.

2. While the bids iterator has not reached the end of the list:

3. While the asks iterator has not reached the end of the list:

4. Is the current bid's price lower than the current ask price?

    a. If yes, break out of the asks loop and start it over with the next bid at step 3., since the current bid will not get matched anymore due to the sorting.

    b. If not, match the bid and ask together for a sale.

        i. After the sale, if the ask amount is 0 or less, erase it, which sets the iterator to the next ask. If not, increment the asks iterator.

        ii. After the sale, if the bid amount is 0 or less, erase it, which sets the iterator to the next bid, and set the asks iterator to the start of the list again, since the current bid is done and we must check the next one.

5. If we reach the end of the asks inner loop, then the bid has been checked against all asks and still has some amount left. Move to the next bid, and reset the asks iterator to the start, back to step 3.

6. When we reach the end of the bids loop, then all bids and asks have been matched and erased if necessary.