OBJECT ORIENTED PROGRAMMING
COURSEWORK FOR MIDTERM: MERKLEREXBOT
Student ID: 190488303

REPORT

R1: MARKET ANALYSIS

R1A: Retrieve the live order book from the Merklerex exchange simulation

```
OrderBook orderBook{"20200317.csv"};
OrderBook orderBook{ "20200601.csv" };
```

These two lines of code can be found in MerkelMain.h and it creates the object orderBook which retrieves the live order book in the Merklerex exchange simulation from both csv files.

R1B: Generate predictions of likely future market exchange rates using defined algorithms, for example, linear regression

To explain the algorithm for linear regression, I will be using code from the function automateBuying() where the algorithm was implemented to facilitate the bot in deciding when and how to generate bids by taking into account likely future market price.

```
for (std::string const& p : orderBook.getKnownProducts()) {
```

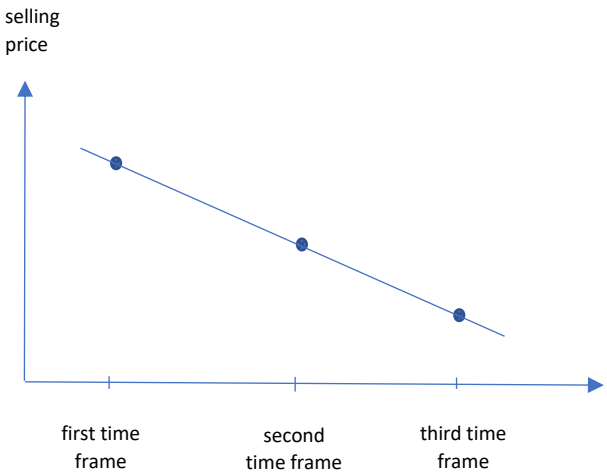
The for loop is used to generate products from the orderbook entry to be used later.

```
std::vector<OrderBookEntry> ask_entries = orderBook.getOrders(OrderBookType::ask, p, currentTime);
//get ask entries in the current timeframe
double currentPrice = orderBook.getLowPrice(ask_entries);
//get the lowest price of the ask entries in this timeframe
std::vector<OrderBookEntry> next_ask_entries = orderBook.getOrders(OrderBookType::ask, p, orderBook.getNextTime(currentTime));
//get ask entries in the next timeframe
double nextPrice = orderBook.getLowPrice(next_ask_entries);
//get the lowest price of the ask entries in next timeframe
```

The first line of code is used to call all entries that are of a certain type within the first timeframe from the orderbook vector. It extracts its type, currency, and current timeframe and places it in the vector. The second line then looks for the lowest price from that vector for the current or first time frame, and stores it in the double, which in this case is currentPrice, to be used later.

This is then repeated for the next time frame in the third and fourth line of code, where third line calls all entries that are of the same type within the second timeframe from the orderbook vector. It extracts its type, currency, and next timeframe and places it into another vector. The fourth line of code will then look for the lowest price in the next or second time frame from that vector, and store it in the double, which in this case is nextPrice, to be used later.

For linear regression, I used the line y = mx + c to find the predicted price in the future, or third time frame.



```
double gradient = (nextPrice - currentPrice) / 1;
//gradient
double constant = currentPrice - gradient * 1;
//constant
double predictedPrice = gradient * 3 + constant;
//predicted low price
```

gradient = m = (nextPrice - currentPrice) / (second timeframe – first timeframe)

= (nextPrice - currentPrice) / (2-1)

= (nextPrice - currentPrice) / 1

constant = c = y – mx //at first timeframe, x = 1

= currentPrice – gradient*1

Thus, we can use y = mx + c to find the predicted price in the third timeframe.

predictedPrice = y = mx + c // at third timeframe, x = 3

= gradient*3 + constant

R2: BIDDING AND BUYING FUNCTIONALITY

R2A: Decide when and how to generate bids using a defined algorithm which takes account of the current and likely future market price

When buying, we will generate bids based on the suggested linear regression algorithm. Firstly, to generate a bid, we will need to see what others are asking for in the live order book. We will then proceed to request for the lowest price in each timeframe as a basis to use linear regression to predict the lowest ask amount in the third timeframe. This is because we want to buy at the lowest price.

```
std::vector<OrderBookEntry> ask_entries = orderBook.getOrders(OrderBookType::ask, p, currentTime);
//get ask entries in the current timeframe
double currentPrice = orderBook.getLowPrice(ask_entries);
//get the lowest price of the ask entries in this timeframe
std::vector<OrderBookEntry> next_ask_entries = orderBook.getOrders(OrderBookType::ask, p, orderBook.getNextTime(currentTime));
//get ask entries in the next timeframe
double nextPrice = orderBook.getLowPrice(next_ask_entries);
//get the lowest price of the ask entries in next timeframe
```

The first line of code is used to call all entries that are of asking type within the first timeframe from the orderbook vector. It extracts its type, currency, and current timeframe and places it in the vector ask_entries. The second line then looks for the lowest price from the vector ask_entries for the current or first time frame, and stores it into the double currentPrice.

This is then repeated for the next time frame in the third and fourth line of code, where third line calls all entries that are of asking type within the second timeframe from the orderbook vector. It extracts its type, currency, and next timeframe and places it in the vector next_ask_entries. The fourth line of code will then look for the lowest price from the vector next_ask_entries in the next or second time frame, and stores it in the double nextPrice.

```
double gradient = (nextPrice - currentPrice) / 1;
//gradient
double constant = currentPrice - gradient * 1;
//constant
double predictedPrice = gradient * 3 + constant;
//predicted low price
```

The algorithm is then used to generate and predict the lowest ask price in the third timeframe, which is the value stored in predictedPrice.

```
if (nextPrice > predictedPrice)
{
```

Since we want to buy at a lower price, the bot will only generate bids when the predictedPrice in the third timeframe is lower than nextPrice in the second timeframe. Hence, this allows us to decide when and how to generate bids by taking into account the current and likely future market price.

R2B: Pass the bids to the exchange for matching

```
try {
    OrderBookEntry obe = CSVReader::stringsToOBE(
        std::to_string(predictedPrice),
        "100",
        currentTime,
        p,
        OrderBookType::ask
    );
```

To generating bids using the algorithm, a new object is used to store the new data in vector form. There are five elements in the vector to be stored, which is the new predicted price in the third time frame, the amount of exchanges we are willing to make, the current time at which the bid takes place, the currency, as well as the type, which in this case is ask, all done respectively. These are all to be passed into the exchange later for matching.

Here, I have hard coded the amount of exchanges made to “100” as it is a good balance for the amount of exchanges that can be made in order to earn money.

```
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
    std::cout << obe.price << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;
```

If there is sufficient money in the wallet, which is checked in the if statement in the first line of code, the bids are passed to the exchange for matching. This can be seen where our bids are matched to the asks in the market in the fifth line of code.

R2C: Receive the results of the exchange's matching engine (which decides which bids have been accepted) which might involve exchanging assets according to the bid and the matching, and the cost of the exchange generated by the simulation

```
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
    std::cout << obe.price << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;
```

Assuming that it occurs, when the bids are matched and assets are exchanged, the results of the exchange's matching engine will then be stored in a new vector called sales in the order book entry, which can be seen in the fifth line of code.

```
for (OrderBookEntry& sale : sales)
{
    std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
    std::cout << "reached" << std::endl;
    myFile << "Sale product:" << sale.product << " ,Sale price: " << sale.price << " ,amount: " << sale.amount << std::endl;

    // update the wallet
    wallet.processSale(sale);
```

For each sale made which is stored in the orderbook entry which can be seen in the for loop, the price and amount made during sales will be printed out, which can be seen in the second line of code. Hence, this prints out the cost of the exchange generated by the simulation. The wallet will be updated in the last line of code as shown above.

R2D: Using the live order book from the exchange, decide if it should withdraw its bids at any point in time

```
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
    std::cout << obe.price << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;

    for (OrderBookEntry& sale : sales)
    {
        std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
        std::cout << "reached" << std::endl;
        myFile << "Sale product:" << sale.product << " ,Sale price: " << sale.price << " ,amount: " << sale.amount << std::endl;

        // update the wallet
        wallet.processSale(sale);
    }
}
else
{
    std::cout << "Wallet has insufficient funds . " << std::endl;
}
```

According to the code above, the bot will stop bidding when there is insufficient money in the wallet and it cannot fulfil the order, which will be printed out in the else loop. Hence more purchases are withdrawn once it hits the wallet's limit.

R3: OFFERING AND SELLING

R3A: Generate offers using a defined algorithm which takes account of the current and likely future market price and pass the offers to the exchange for matching

When selling, we will generate offers based on the suggested linear regression algorithm. Firstly, to generate a offer, we will need to see what others are bidding for in the live order book. We will then proceed to request for the highest price in each timeframe as a basis to use linear regression to predict the highest bid amount in the third timeframe. This is because we want to sell at the highest price.

```
std::vector<OrderBookEntry> bid_entries = orderBook.getOrders(OrderBookType::bid, p, currentTime);
//get bid entries in the current timeframe
double currentPrice = orderBook.getHighPrice(bid_entries);
//get the highest price of the bid entries in this timeframe
std::vector<OrderBookEntry> next_bid_entries = orderBook.getOrders(OrderBookType::bid, p, orderBook.getNextTime(currentTime));
//get bid entries in the next timeframe
double nextPrice = orderBook.getHighPrice(next_bid_entries);
//get the highest price of the bid entries in the next timeframe
```

The first line of code is used to call all entries that are of bidding type within the first timeframe from the orderbook vector. It extracts its type, currency, and current timeframe and places it in the vector bid_entries. The second line then looks for the highest price from the vector bid_entries for the current or first time frame, and stores it into the double currentPrice.

This is then repeated for the next time frame in the third and fourth line of code, where third line calls all entries that are of bidding type within the second timeframe from the orderbook vector. It extracts its type, currency, and next timeframe and places it in the vector next_bid_entries. The fourth line of code will then look for the lowest price from the vector next_bid_entries in the next or second time frame, and stores it in the double nextPrice.

```
double gradient = (nextPrice - currentPrice) / 1;
//gradient
double constant = currentPrice - gradient * 1;
//constant
double predictedPrice = gradient * 3 + constant;
//predicted low price
```

The algorithm is then used to generate and predict the highest bid price in the third timeframe, which is the value stored in predictedPrice.

```
if (nextPrice < predictedPrice)
```

Since we want to sell at a higher price, the bot will only generate offers when the predictedPrice in the third timeframe is higher than nextPrice in the second timeframe. Hence, this allows us to decide when and how to generate offers by taking into account the current and likely future market price.

R3B: Pass the bids to the exchange for matching

```
try {
    OrderBookEntry obe = CSVReader::stringsToOBE(
        std::to_string(predictedPrice),
        "100",
        currentTime,
        p,
        OrderBookType::bid
    );
```

To generating offers using the algorithm, a new object is used to store the new data in vector form. There are five elements in the vector to be stored, which is the new predicted price in the third time frame, the amount of exchanges we are willing to make, the current time at which the offer takes place, the currency, as well as the type, which in this case is bid, all stored respectively. These are all to be passed into the exchange later for matching.

Here, I have hard coded the amount of exchanges made to “100” as it is a good balance for the amount of exchanges that can be made in order to earn money.

```
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
    std::cout << obe.price << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;
```

If there is sufficient money in the wallet, which is checked in the if statement in the first line of code, the bids are passed to the exchange for matching. This can be seen where our asks are matched to the bids in the market in the fifth line of code.

R3C: Receive the results of the exchange's matching engine (which decides which offers have been accepted) which might involve exchanging assets according to the offer and the matching, and the cost of the exchange generated by the simulation

```
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
    std::cout << obe.price << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;
```

Assuming that it occurs, when the offers are matched and assets are exchanged, the results of the exchange's matching engine will then be stored in a new vector called sales in the order book entry, which can be seen in the fifth line of code.

```
for (OrderBookEntry& sale : sales)
{
    std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
    std::cout << "reached" << std::endl;
    myFile << "Sale product:" << sale.product << " ,Sale price: " << sale.price << " ,amount: " << sale.amount << std::endl;

    // update the wallet
    wallet.processSale(sale);
```

For each sale made which is stored in the orderbook entry which can be seen in the for loop, the price and amount made during sales will be printed out, which can be seen in the second line of code. Hence, this prints out the cost of the exchange generated by the simulation. The wallet will be updated in the last line of code as shown above.

R3D: Using the live order book from the exchange, decide if it should withdraw its offers at any point in time

```
if (wallet.canFulfillOrder(obe))
{
    std::cout << "Wallet looks good. " << std::endl;
    orderBook.insertOrder(obe);
    std::cout << obe.price << std::endl;

    std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
    std::cout << "Sales: " << sales.size() << std::endl;

    for (OrderBookEntry& sale : sales)
    {
        std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
        std::cout << "reached" << std::endl;
        myFile << "Sale product:" << sale.product << " ,Sale price: " << sale.price << " ,amount: " << sale.amount << std::endl;

        // update the wallet
        wallet.processSale(sale);
    }
}
else
{
    std::cout << "Wallet has insufficient funds . " << std::endl;
}
```

According to the code above, the bot will stop offering when there is insufficient money in the wallet and it cannot fulfil the order, which will be indicated and printed out in the else loop. Hence offers are withdrawn once it hits the wallet's limit.

R4: LOGGING

R4A: Maintain a record of its assets and how they change over time

```
#include <fstream>
using namespace std;
```

To create a new file, these two lines of code must be declared so that they can be called in any function later.

For buying:

```
ofstream assetRecord("Wallet.txt");
ofstream myFile("Automate asking.csv");
myFile << "Ask csv " << endl;
cout << "Going to start asking" << endl;
```

For selling:

```
ofstream assetRecord("Wallet.txt");
ofstream myFile("Automate bidding.csv");
myFile << "Bid csv " << endl;
cout << "Going to start bidding" << endl;
```

For both buying and selling, a file is created for both to maintain records of its assets and how it changes over time. They are called “Automate asking” and “Automate bidding”.

```
myFile << "" << endl;
myFile << "Wallet at " << currentTime << endl;
myFile << wallet.toString() << endl;
assetRecord << "Wallet at " << currentTime << endl;
assetRecord << wallet.toString() << endl;
```

For both buy and selling functions, they include the codes above, and these lines of code prints out the timing at which purchases are made. To do so, the amount in the wallet has to change datatype from double to string, which is done in the second line of code above. Hence this records how assets changes over time.

R4B: Maintain a record of the bids and offers it has made in a suitable file format

R4C: Maintain a record of successful bids and offers it has made, along with the context (e.g. exchange average offer and bid) in a suitable file format

```
for (OrderBookEntry& sale : sales)
{
    std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
    std::cout << "reached" << std::endl;
    myFile << "Sale product:" << sale.product << " ,Sale price: " << sale.price << " ,amount: " << sale.amount << std::endl;

    // update the wallet
    wallet.processSale(sale);
}
```

These codes can be found in both functions for buying and selling, and it is responsible for recording all the bids and offers it has made and stores them into the corresponding file. According to line four of the code, the sales product, price and amount are the elements to be stored into the file.