CM2005
Object Oriented Programming
Coursework 1 Report

10211835

Kwang Kian Hui

Contents

Assumptions and understanding:

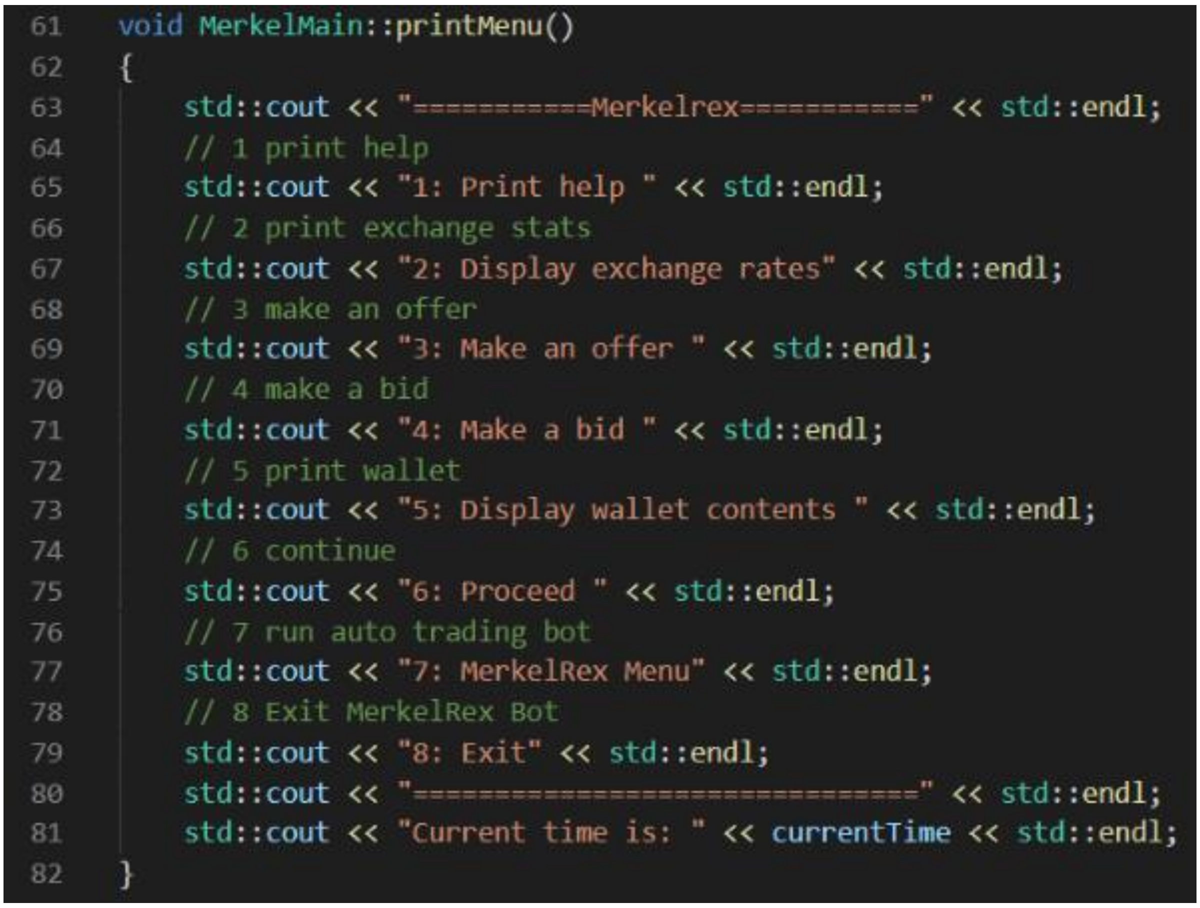
1) My understanding of R1A requirement of “retrieve the live order book” means to extract data from the live order book that will be used for market analysis instead of simply importing data from csv file as that part is already done in the skeleton code itself, despite some learning peers saying that is the requirement and thus, my explanation will be on what and how I will extract for use in market analysis.
2) Assuming that USDT is the main currency when checking asset value over various timestamps.

R1: Market Analysis

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

(Achieved)

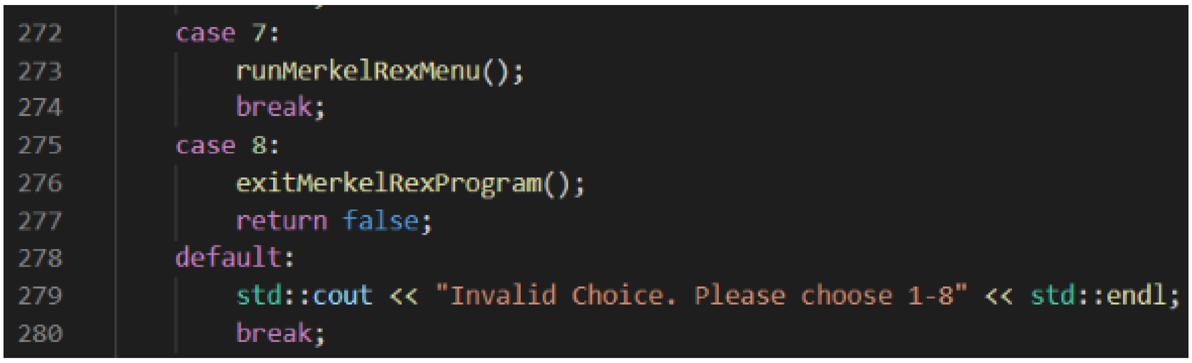
Program steps before retrieving data from live order book:



```
61  void MerkelMain::printMenu()
62  {
63      std::cout << "==========Merkelrex==========" << std::endl;
64      // 1 print help
65      std::cout << "1: Print help " << std::endl;
66      // 2 print exchange stats
67      std::cout << "2: Display exchange rates" << std::endl;
68      // 3 make an offer
69      std::cout << "3: Make an offer " << std::endl;
70      // 4 make a bid
71      std::cout << "4: Make a bid " << std::endl;
72      // 5 print wallet
73      std::cout << "5: Display wallet contents " << std::endl;
74      // 6 continue
75      std::cout << "6: Proceed " << std::endl;
76      // 7 run auto trading bot
77      std::cout << "7: MerkelRex Menu" << std::endl;
78      // 8 Exit MerkelRex Bot
79      std::cout << "8: Exit" << std::endl;
80      std::cout << "==============================" << std::endl;
81      std::cout << "Current time is: " << currentTime << std::endl;
82  }
```

Figure 1.1A – MerkelMain.cpp (line 61 – 82)

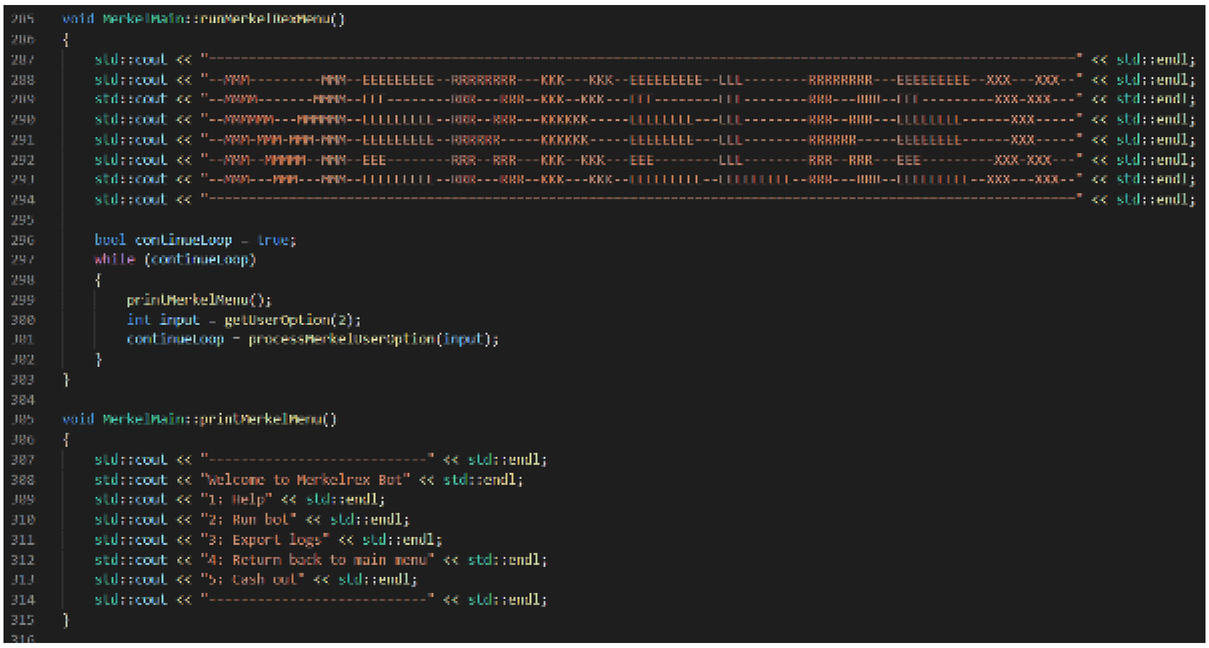
A new option '7' is created for the user to select as seen in Figure 1.1A. It will open up a menu with several options.



```
272     case 7:
273         runMerkelRexMenu();
274         break;
275     case 8:
276         exitMerkelRexProgram();
277         return false;
278     default:
279         std::cout << "Invalid Choice. Please choose 1-8" << std::endl;
280         break;
```

Figure 1.1B – MerkelMain.cpp (line 272 – 280)

Figure 1.1B shows a part of the processUserOption() function in the MerkelMain.cpp file from line 247 to 283. Switch case function is used to process the input and to run the appropriate functions. In this case, option 7 will run the runMerkelRexMenu() function.



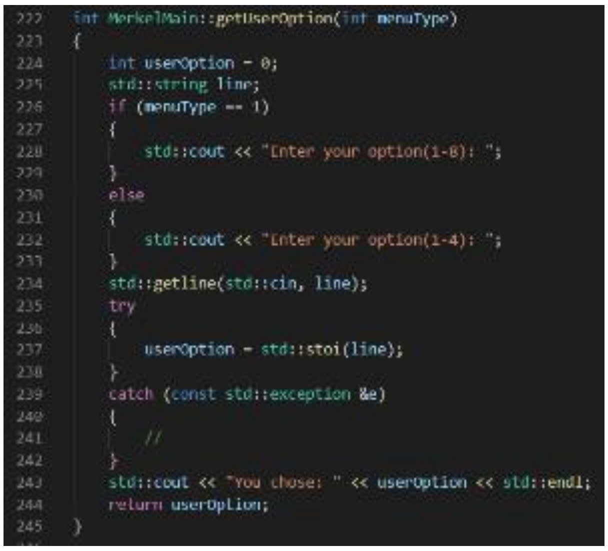
```
285  void MerkelMain::runMerkelRexMenu()
286  {
287      std::cout << "-----------------------------------------------------------------------------------------------------------" << std::endl;
288      std::cout << "--MMM--------MMM--EEEEEEEEE--RRRRRRRR---KKK---KKK--EEEEEEEEE--LLL--------RRRRRRRR---EEEEEEEEE--XXX---XXX--" << std::endl;
289      std::cout << "--MMMM------MMMM--EEE--------RRR---RRR--KKK--KKK---EEE--------LLL--------RRR---RRR--EEE----------XXX-XXX---" << std::endl;
290      std::cout << "--MMMMMM---MMMMMM--EEEEEEEE--RRR--RRR---KKKKKK-----EEEEEEEE---LLL--------RRR--RRR---EEEEEEEE------XXX-----" << std::endl;
291      std::cout << "--MMM-MMM-MMM-MMM--EEEEEEEE--RRRRRR-----KKKKKK-----EEEEEEEE---LLL--------RRRRRR-----EEEEEEEE------XXX-----" << std::endl;
292      std::cout << "--MMM--MMMMM--MMM--EEE--------RRR--RRR---KKK--KKK---EEE--------LLL--------RRR--RRR---EEE----------XXX-XXX---" << std::endl;
293      std::cout << "--MMM---MMM---MMM--EEEEEEEEE--RRR---RRR--KKK---KKK--EEEEEEEEE--LLLLLLLLL--RRR---RRR--EEEEEEEEE--XXX---XXX--" << std::endl;
294      std::cout << "-----------------------------------------------------------------------------------------------------------" << std::endl;
295
296      bool continueLoop = true;
297      while (continueLoop)
298      {
299          printMerkelMenu();
300          int input = getUserOption(2);
301          continueLoop = processMerkelUserOption(input);
302      }
303  }
304
305  void MerkelMain::printMerkelMenu()
306  {
307      std::cout << "--------------------------" << std::endl;
308      std::cout << "Welcome to Merkelrex Bot" << std::endl;
309      std::cout << "1: Help" << std::endl;
310      std::cout << "2: Run bot" << std::endl;
311      std::cout << "3: Export logs" << std::endl;
312      std::cout << "4: Return back to main menu" << std::endl;
313      std::cout << "5: Cash out" << std::endl;
314      std::cout << "--------------------------" << std::endl;
315  }
```

Figure 1.1C – MerkelMain.cpp (line 285 – 315)

Figure 1.1C shows both the runMerkelRexMenu() and printMerkelMenu() function in the MerkelMain class.

The line of code from 287 to 294 is only for aesthetic purposes to simulate a real trading bot use interface. It can be commented or removed without affecting the functionality of the other codes.

Line 296 to line 303 will produce a similar functional menu as the main menu seen previously. printMerkelMenu() function prints the menu as seen in line 305 to 315.

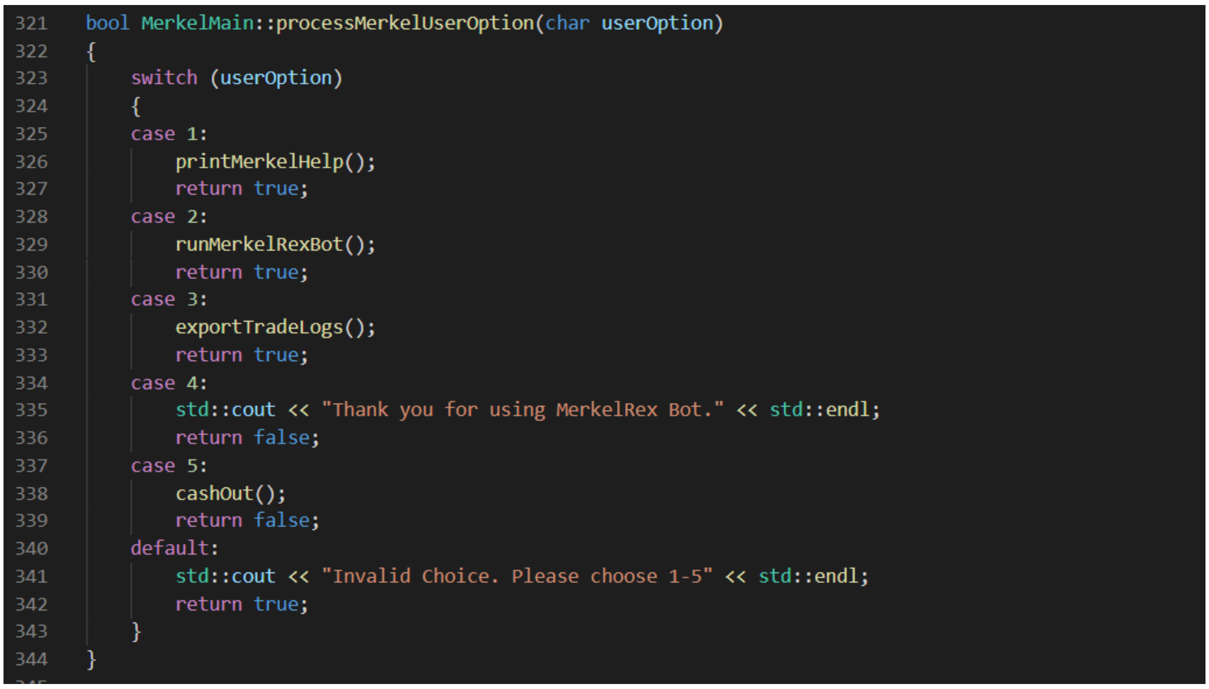


```
int MerkelMain::getUserOption(int menuType)
{
    int userOption = 0;
    std::string line;
    if (menuType == 1)
    {
        std::cout << "Enter your option(1-8): ";
    }
    else
    {
        std::cout << "Enter your option(1-4): ";
    }
    std::getline(std::cin, line);
    try
    {
        userOption = std::stoi(line);
    }
    catch (const std::exception &e)
    {
        //
    }
    std::cout << "You chose: " << userOption << std::endl;
    return userOption;
}
```

Figure 1.1D – MerkelMain.cpp (line 222 – 245)

As shown in figure 1.1D, getUserOption() function is modified to take in an integer input. Depending on the integer value, the message printed will be different but will function the same by attempting to convert the string input value to integer before returning it.

The returned integer value will be used as the input parameter for the processMerkelUserOption() function shown in figure 1.1C, line 301.



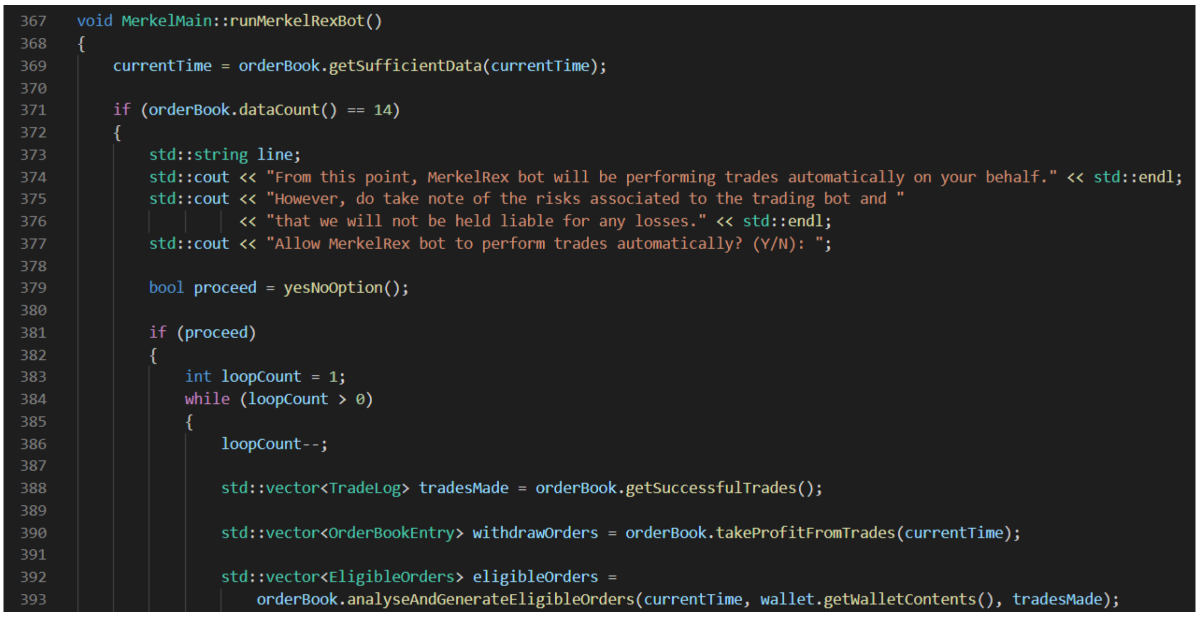
```cpp
bool MerkelMain::processMerkelUserOption(char userOption)
{
    switch (userOption)
    {
    case 1:
        printMerkelHelp();
        return true;
    case 2:
        runMerkelRexBot();
        return true;
    case 3:
        exportTradeLogs();
        return true;
    case 4:
        std::cout << "Thank you for using MerkelRex Bot." << std::endl;
        return false;
    case 5:
        cashOut();
        return false;
    default:
        std::cout << "Invalid Choice. Please choose 1-5" << std::endl;
        return true;
    }
}
```

Figure 1.1E – MerkelMain.cpp (line 321 – 344)

Figure 1.1E shows the processMerkelUserOption() function in MerkelMain class. Switch case is used to process the input again. The parameter char userOption converts the integer value into a char value. Each case returns a boolean value where if true, the menu will loop again, else it will exit from the runMerkelRexMenu() and return to the main menu.

Option 2 runs the MerkelRex bot which performs the analysis and predictions.

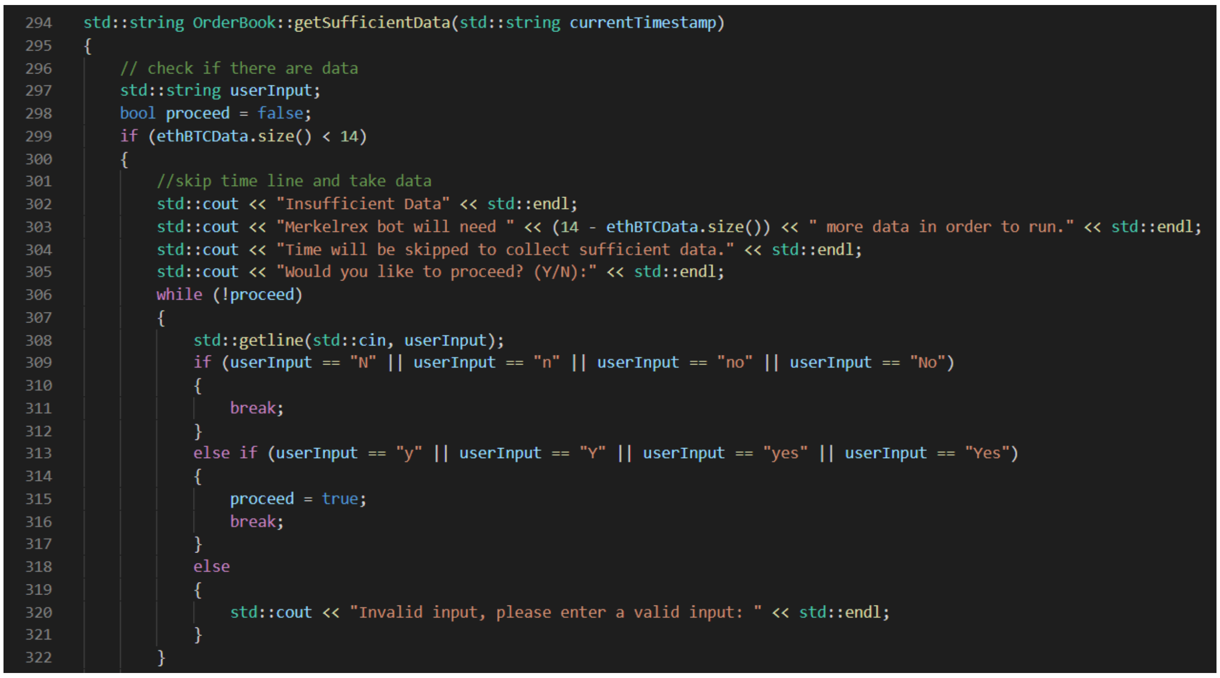
Start data collection:



```cpp
void MerkelMain::runMerkelRexBot()
{
    currentTime = orderBook.getSufficientData(currentTime);

    if (orderBook.dataCount() == 14)
    {
        std::string line;
        std::cout << "From this point, MerkelRex bot will be performing trades automatically on your behalf." << std::endl;
        std::cout << "However, do take note of the risks associated to the trading bot and "
                  << "that we will not be held liable for any losses." << std::endl;
        std::cout << "Allow MerkelRex bot to perform trades automatically? (Y/N): ";

        bool proceed = yesNoOption();

        if (proceed)
        {
            int loopCount = 1;
            while (loopCount > 0)
            {
                loopCount--;

                std::vector<TradeLog> tradesMade = orderBook.getSuccessfulTrades();

                std::vector<OrderBookEntry> withdrawOrders = orderBook.takeProfitFromTrades(currentTime);

                std::vector<EligibleOrders> eligibleOrders =
                    orderBook.analyseAndGenerateEligibleOrders(currentTime, wallet.getWalletContents(), tradesMade);
```

Figure 1.1F – MerkelMain.cpp (line 367 – 393)

In line 369, we call the OrderBook object, orderBook to run the getSufficientData() function with currentTime as its input parameter. It will be explained in the next part as to why the return value of the getSufficientData() function is assigned to currentTime.



```
294 std::string OrderBook::getSufficientData(std::string currentTimestamp)
295 {
296     // check if there are data
297     std::string userInput;
298     bool proceed = false;
299     if (ethBTCData.size() < 14)
300     {
301         //skip time line and take data
302         std::cout << "Insufficient Data" << std::endl;
303         std::cout << "Merkelrex bot will need " << (14 - ethBTCData.size()) << " more data in order to run." << std::endl;
304         std::cout << "Time will be skipped to collect sufficient data." << std::endl;
305         std::cout << "Would you like to proceed? (Y/N):" << std::endl;
306         while (!proceed)
307         {
308             std::getline(std::cin, userInput);
309             if (userInput == "N" || userInput == "n" || userInput == "no" || userInput == "No")
310             {
311                 break;
312             }
313             else if (userInput == "y" || userInput == "Y" || userInput == "yes" || userInput == "Yes")
314             {
315                 proceed = true;
316                 break;
317             }
318             else
319             {
320                 std::cout << "Invalid input, please enter a valid input: " << std::endl;
321             }
322         }
```

Figure 1.1G – OrderBook.cpp (line 294 – 322)

Figure 1.1G shows the getSufficientData in OrderBook.cpp from line 294 to 333.

The function has string currentTimestamp as its input parameter. It first creates a string variable named userInput and bool variable named proceed and initialises it with false value.

The first if statement checks if the vector of AnalysisData - ethBTCData.size() has at least 14 or more data. The AnalysisData class takes the product(string), avgAsk(double), askVolume(double), avgBid(double) and bidVolume(double) as its class variables.

// start-reasoning

The aim of holding these data in a new AnalysisData class is to run our analysis and algorithm later with those data. How those data will be collected will be explained in the later part of this function.

The if statement to check the data size of the vector need not be the ethBTCData vector. It can be any of the five AnalysisData vectors that I created in OrderBook.h for each respective product available for exchange.

The data size need not be 14 but I have chosen this number previously as I wanted to use the relative strength index(RSI) as one of the factors to be used in the algorithm to decide whether to buy or sell a product. Written in Investopedia, the standard is to use 14 periods to calculate the initial RSI value.[1] JASON FERNANDO. (2020, November 17). Relative Strength Index (RSI). Investopedia. https://www.investopedia.com/terms/r/rsi.asp]

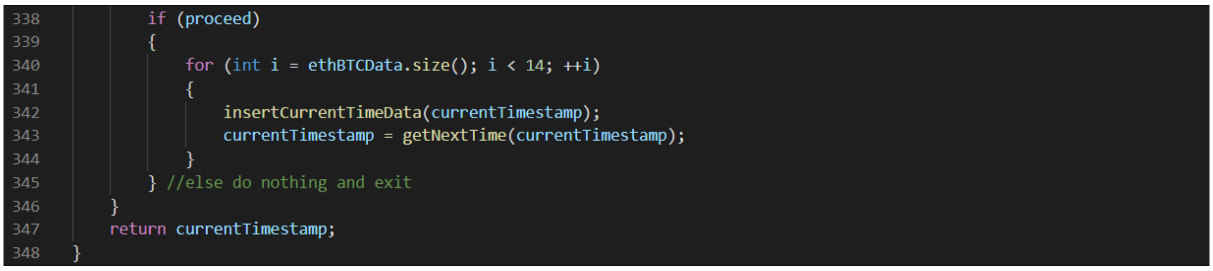
RSI is usually used to identify trends with 14 periods as the default setting and can be changed but a lower period will produce readings that are much more sensitive. The RSI indicates that any stock with RSI rating below 30 indicates that it is oversold while RSI rating above 70 indicates that it is overbought.

However, data of each time period would typically be recorded separated by a day while the data provided is separated by 5 seconds. During my tests, I have found that the RSI of every products would range from 34 – 62. Each product's RSI over the time periods does not change much despite having different data in each timestamp.

e.g. RSI of BTC/USDT hovers around 50+ to 60+ while DOGE/USDT hovers around 30+.

Thus, I have decided to not use RSI as factor for my algorithm but will continue to use the data size of 14 for other calculations and analysis.

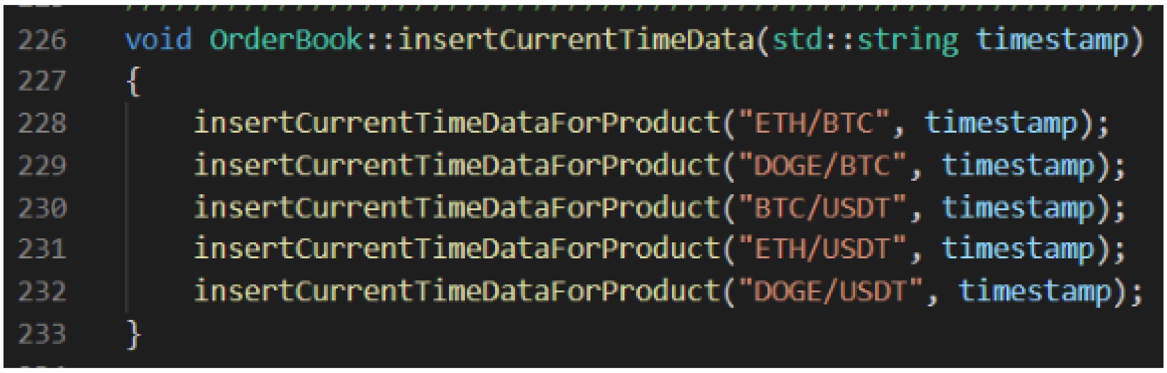
// end-reasoning



```
338             if (proceed)
339             {
340                 for (int i = ethBTCData.size(); i < 14; ++i)
341                 {
342                     insertCurrentTimeData(currentTimestamp);
343                     currentTimestamp = getNextTime(currentTimestamp);
344                 }
345             } //else do nothing and exit
346         }
347         return currentTimestamp;
348     }
```

Figure 1.1H – OrderBook.cpp (line 338 – 348)

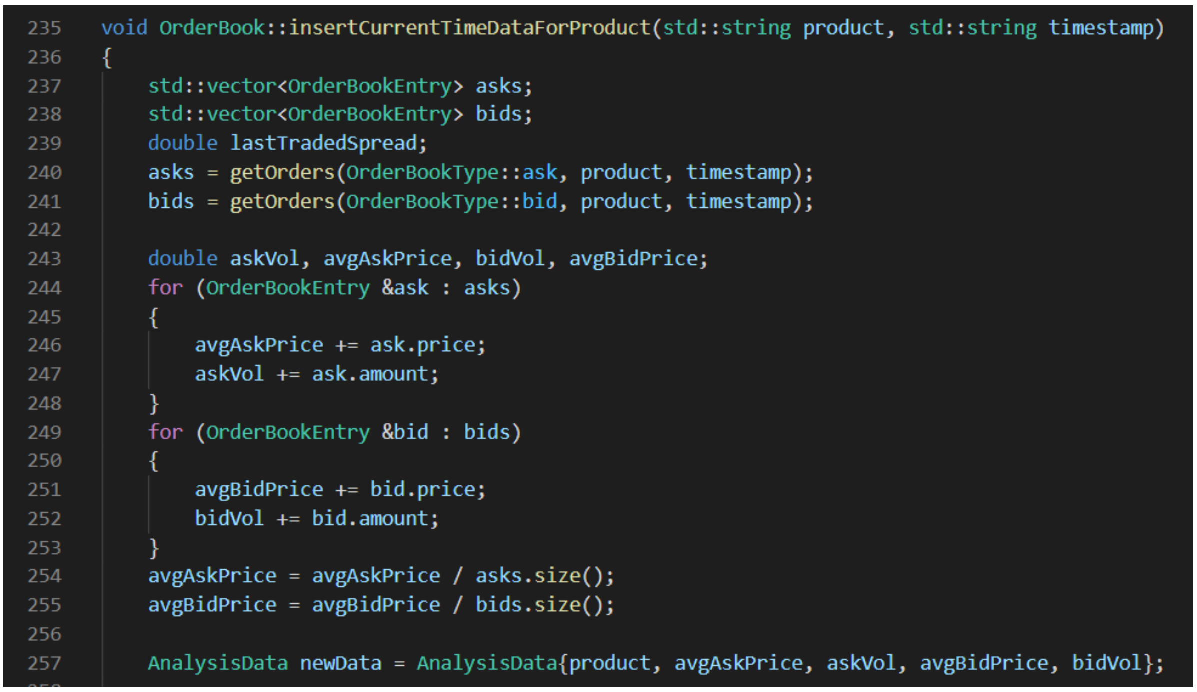
If user enters any of the yes inputs ("y","Y","yes","Yes"), it will proceed to perform the insertion of data as shown in figure 1.1H. The for loop starts with the i value equal to the size of the data that has already been collected and runs until the i value reaches 14. It inserts the data from current time stamp which can be seen in the OrderBook.cpp from line 226 to 282.



```
226   void OrderBook::insertCurrentTimeData(std::string timestamp)
227   {
228       insertCurrentTimeDataForProduct("ETH/BTC", timestamp);
229       insertCurrentTimeDataForProduct("DOGE/BTC", timestamp);
230       insertCurrentTimeDataForProduct("BTC/USDT", timestamp);
231       insertCurrentTimeDataForProduct("ETH/USDT", timestamp);
232       insertCurrentTimeDataForProduct("DOGE/USDT", timestamp);
233   }
```

Figure 1.1I – OrderBook.cpp (line 226 – 233)

insertCurrentTimeData() function calls insertCurrentTimeDataForProduct() function with all the different products and timestamp as its parameters.

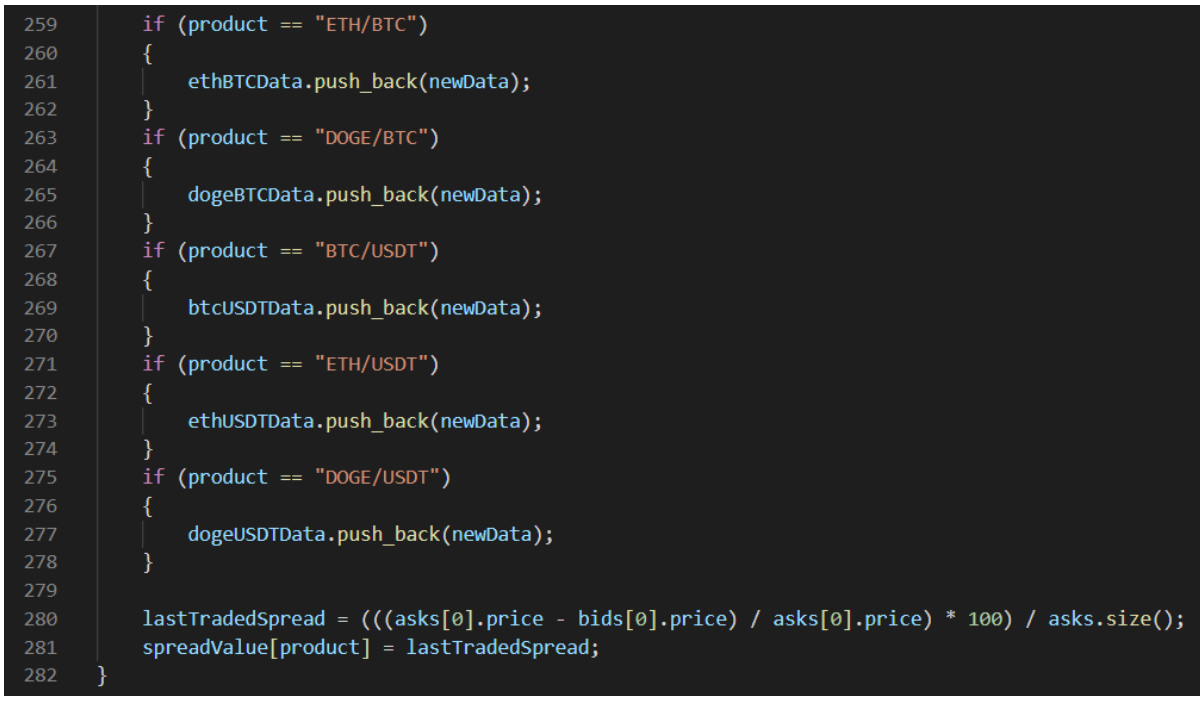


```cpp
235  void OrderBook::insertCurrentTimeDataForProduct(std::string product, std::string timestamp)
236  {
237      std::vector<OrderBookEntry> asks;
238      std::vector<OrderBookEntry> bids;
239      double lastTradedSpread;
240      asks = getOrders(OrderBookType::ask, product, timestamp);
241      bids = getOrders(OrderBookType::bid, product, timestamp);
242
243      double askVol, avgAskPrice, bidVol, avgBidPrice;
244      for (OrderBookEntry &ask : asks)
245      {
246          avgAskPrice += ask.price;
247          askVol += ask.amount;
248      }
249      for (OrderBookEntry &bid : bids)
250      {
251          avgBidPrice += bid.price;
252          bidVol += bid.amount;
253      }
254      avgAskPrice = avgAskPrice / asks.size();
255      avgBidPrice = avgBidPrice / bids.size();
256
257      AnalysisData newData = AnalysisData{product, avgAskPrice, askVol, avgBidPrice, bidVol};
```

Figure 1.1J – OrderBook.cpp (line 235 – 257)

Data was collected by going through each asks and bids, summing up the total ask and bid volume (amount), and summing up all of the prices. The average ask price and bid price can then be derived on as shown in line 244 and 253 in figure 1.1J.

A new AnalysisData class variable is created with the new data collected.



```cpp
    if (product == "ETH/BTC")
    {
        ethBTCData.push_back(newData);
    }
    if (product == "DOGE/BTC")
    {
        dogeBTCData.push_back(newData);
    }
    if (product == "BTC/USDT")
    {
        btcUSDTData.push_back(newData);
    }
    if (product == "ETH/USDT")
    {
        ethUSDTData.push_back(newData);
    }
    if (product == "DOGE/USDT")
    {
        dogeUSDTData.push_back(newData);
    }

    lastTradedSpread = (((asks[0].price - bids[0].price) / asks[0].price) * 100) / asks.size();
    spreadValue[product] = lastTradedSpread;
}
```

Figure 1.1K – OrderBook.cpp (line 259 – 282)

It proceeds to check which product data of AnalysisData vector to be inserted into.

The last traded spread is calculated by subtracting the lowest ask price and highest bid price before dividing it by the lowest ask price and multiplying by 100. The spread is the difference between the ask and bid price in % for the product at that time stamp.

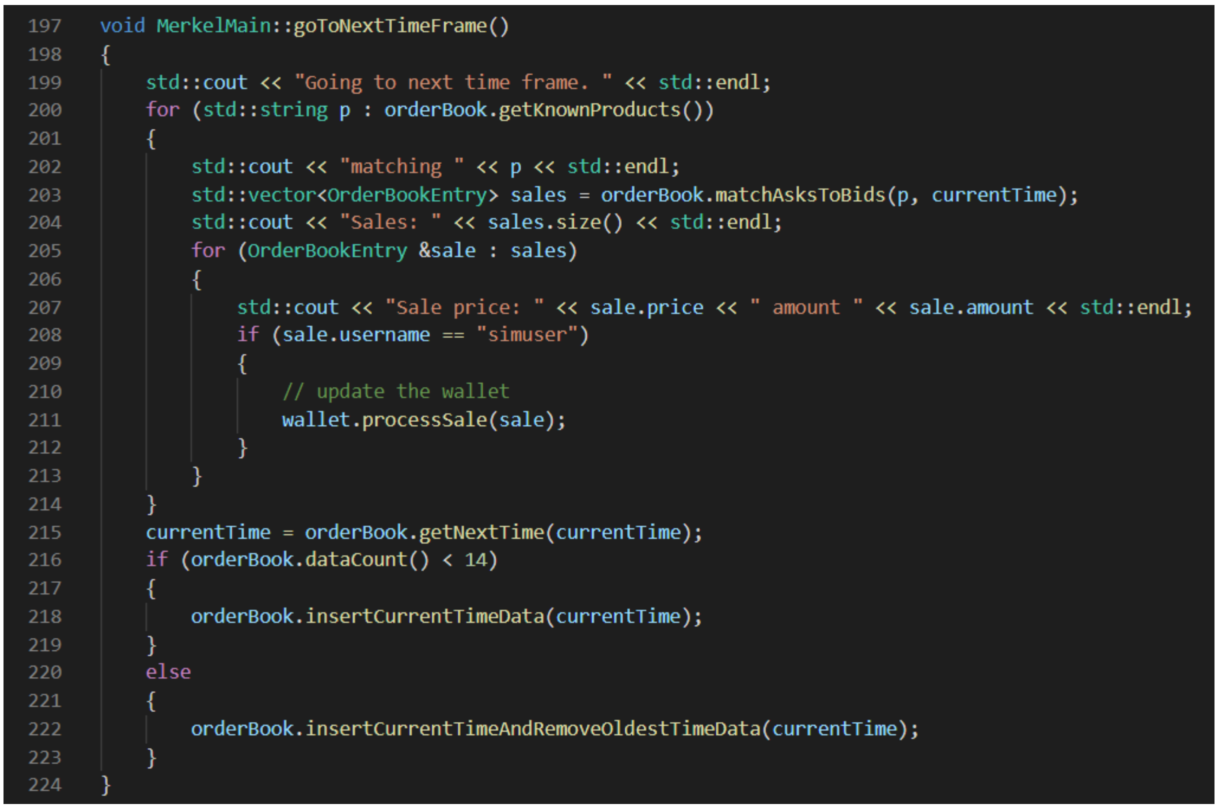
However, I divided the spread by the asks.size() additionally which will be explained during the explanation of the analysis in R1B when the spreadValue map is used.

spreadValue is a map of <string, double> that holds the spread values for all products.

The insertCurrentTimeDataForProducts() function ends after this.

After inserting data for each product in the current time stamp, we will move to the next timestamp as shown in line 340 in figure 1.1H. This process repeats until for loop is exited. After exiting, we will return the new current timestamp which will be received by the currentTime in the MerkelMain class.

That is one of the ways that the program can collect data from the live order book. The other method would be to select option 6 in the main menu which will run the goToNextTimeFrame() function in MerkelMain class seen in figure 1.1L below.

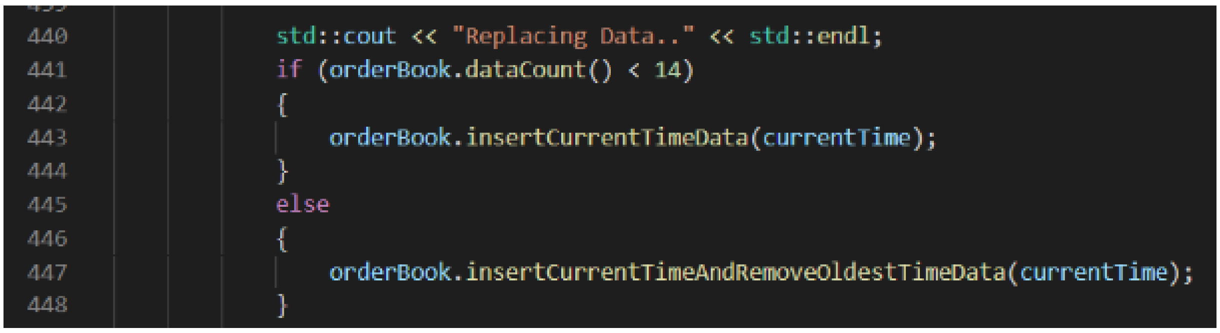


```
197  void MerkelMain::goToNextTimeFrame()
198  {
199      std::cout << "Going to next time frame. " << std::endl;
200      for (std::string p : orderBook.getKnownProducts())
201      {
202          std::cout << "matching " << p << std::endl;
203          std::vector<OrderBookEntry> sales = orderBook.matchAsksToBids(p, currentTime);
204          std::cout << "Sales: " << sales.size() << std::endl;
205          for (OrderBookEntry &sale : sales)
206          {
207              std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
208              if (sale.username == "simuser")
209              {
210                  // update the wallet
211                  wallet.processSale(sale);
212              }
213          }
214      }
215      currentTime = orderBook.getNextTime(currentTime);
216      if (orderBook.dataCount() < 14)
217      {
218          orderBook.insertCurrentTimeData(currentTime);
219      }
220      else
221      {
222          orderBook.insertCurrentTimeAndRemoveOldestTimeData(currentTime);
223      }
224  }
```

Figure 1.1L – MerkelMain.cpp (line 197 – 224)

The goToNextTimeFrame() is modified with new line of 216 to 224. It checks if the orderBook data count is less than 14 or more. dataCount() returns the data size of ethBTCData ("ethBTCData.size()").

If the data size is lesser than 14, it will insert current time data similar to the previous method. If the data size is 14 or more, it will first remove the oldest time data before inserting the new ones. This function can be found in OrderBook.cpp from line 284 to 292.



```
440             std::cout << "Replacing Data.." << std::endl;
441             if (orderBook.dataCount() < 14)
442             {
443                 orderBook.insertCurrentTimeData(currentTime);
444             }
445             else
446             {
447                 orderBook.insertCurrentTimeAndRemoveOldestTimeData(currentTime);
448             }
```

Figure 1.1M – MerkelMain.cpp (line 440 – 448)

The other time that data is retrieved is in the runMerkelRexBot() function in MerkelMain class from line 367 to 454. It performs the same checks and actions as the one in figure 1.1L.

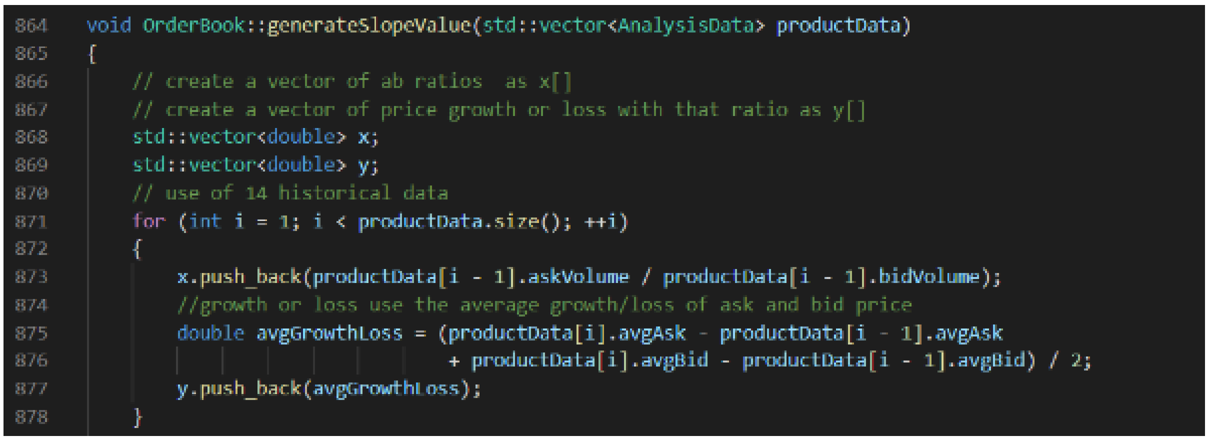
R1A (achieved).

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

(Achieved)

Throughout the analyseAndGenerateEligibleOrders() function in OrderBook class from line 335 to 587 in OrderBook.cpp, linear regression will be performed to predict the likely future price of the products. The criteria and factors to which will be analysed will be discussed in R2A and R3A.

We will jump straight into the linear regression model:



```
864  void OrderBook::generateSlopeValue(std::vector<AnalysisData> productData)
865  {
866      // create a vector of ab ratios  as x[]
867      // create a vector of price growth or loss with that ratio as y[]
868      std::vector<double> x;
869      std::vector<double> y;
870      // use of 14 historical data
871      for (int i = 1; i < productData.size(); ++i)
872      {
873          x.push_back(productData[i - 1].askVolume / productData[i - 1].bidVolume);
874          //growth or loss use the average growth/loss of ask and bid price
875          double avgGrowthLoss = (productData[i].avgAsk - productData[i - 1].avgAsk
876                                  + productData[i].avgBid - productData[i - 1].avgBid) / 2;
877          y.push_back(avgGrowthLoss);
878      }
```

Figure 1.2A – OrderBook.cpp (line 864 – 878)

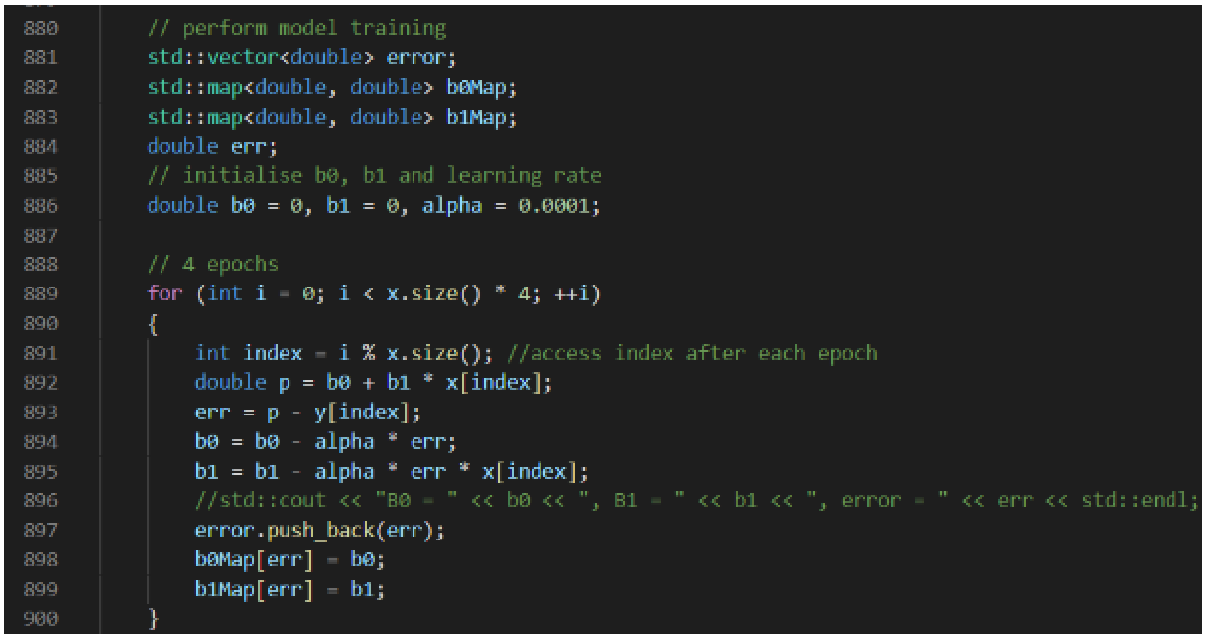
Using linear regression, we can predict the growth or loss value of the currency based on the ab ratio. ab ratio is the ask/bid volume ratio which represents ask volume over the bid volume for the product. The volume is the sum of the amounts.

When the ask volume is higher than bid volume, the prices of the products are likely to increase as it indicates a strong buying pressure. A higher bid volume will indicate that the selling pressure is higher.[2] ADAM MILTON. (2020, July 21). Buying and Selling Volume. The Balance. https://www.thebalance.com/buying-and-selling-volume-1031027 Further explanation on how we calculate and store the values will be explained in later sections.

Vector x will be the factor we are going to make the prediction based on, while vector y will be the growth or loss factor that we will try to predict.

Line 871 to 878 in figure 1.2A shows how we assign the x and y value to the x and y vector. x is calculated by using the product data to get the ab ratio over the different timestamps.

y is calculated by getting the average ask of current timestamp minus the average ask of previous timestamp, summing it with the average bid of current timestamp minus the average bid of previous timestamp. The sum is then divided by 2 to get the average gain or loss of the bid and ask price.



```cpp
    // perform model training
    std::vector<double> error;
    std::map<double, double> b0Map;
    std::map<double, double> b1Map;
    double err;
    // initialise b0, b1 and learning rate
    double b0 = 0, b1 = 0, alpha = 0.0001;

    // 4 epochs
    for (int i = 0; i < x.size() * 4; ++i)
    {
        int index = i % x.size(); //access index after each epoch
        double p = b0 + b1 * x[index];
        err = p - y[index];
        b0 = b0 - alpha * err;
        b1 = b1 - alpha * err * x[index];
        //std::cout << "B0 = " << b0 << ", B1 = " << b1 << ", error = " << err << std::endl;
        error.push_back(err);
        b0Map[err] = b0;
        b1Map[err] = b1;
    }
```

Figure 1.2B – OrderBook.cpp (line 880 – 900)

Why:

Our goal is to get the b0 and b1 value that has minimal error and is as accurate as it can be to get the best fit for future price predictions. The method used is the gradient descent algorithm where we attempt to find the minimum of a function and in this case, the error function. The b0 and b1 value can be seen as m and c variable of the formula y = mx + c. It will be discussed further later in R2A when the predicted growth or loss is used.

Variables:

We will create a vector to hold all error values along with maps of <double, double> to hold b0 and b1 values. We will first declare double values of err, b0, b1 and alpha, where alpha will be the learning rate of the model. We start with random b0 and b1 values and update them based on the error of each instance.

The alpha value typically should have a small value to achieve greater accuracy. I have tried different values and found that 0.0001 seem to have a better prediction accuracy.

Steps:

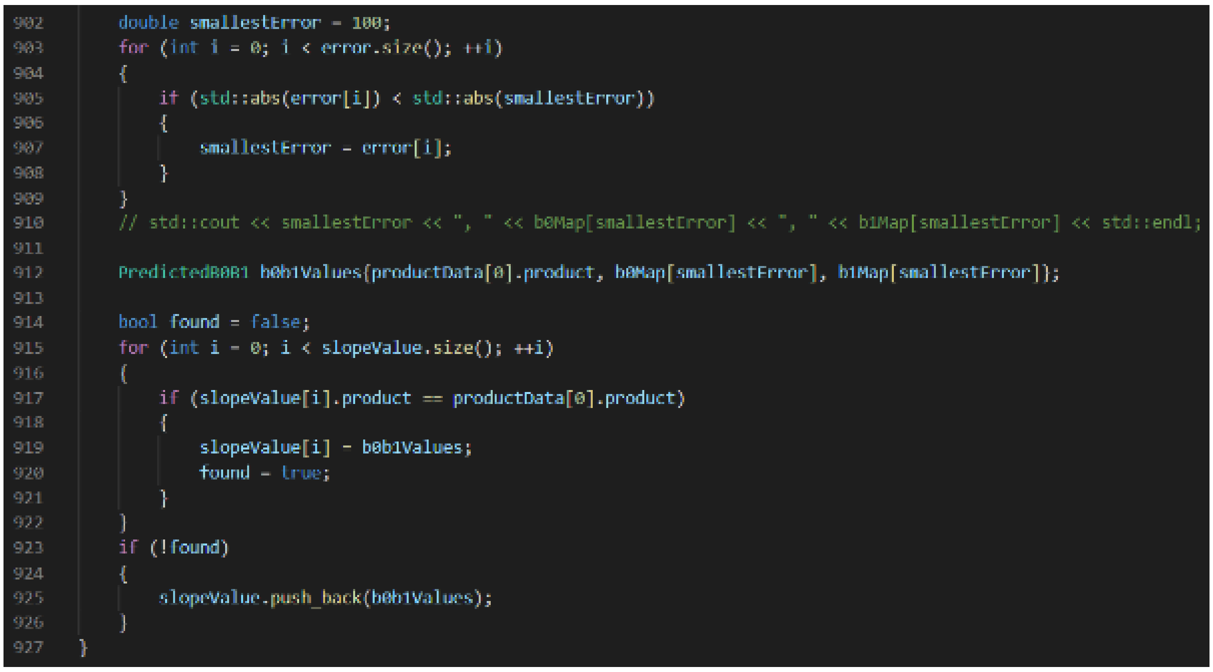
We will need a for loop to iterate through the x and y loop to train the linear regression model. I have decided to iterate through data batch size 4 times (4 epochs). This number can be changed to any number. If we iterate through the data batch once, we may not get the best fit values but may not necessarily get better accuracy values even if we run large number of runs.

Since we are iterating through the x vector 4 times, we will need to get the correct index as the i value increases. This can be achieved by declaring an index variable takes the result of i %(modulus) the batch size.

We start the prediction with the formula in line 868 and initialise the predicted value to variable p. The error is calculated by subtracting the p variable by the actual gain or loss result stored in the y vector. The b0 and b1 values are then updated by subtracting the (learning rate * error of previous prediction) and (learning rate * error of previous prediction) * (x value of the instance) to itself.

After updating the b0 and b1 value, push the err variable into the error vector and map the b0 and b1 values into the respective map variables with the err value.

Repeat these steps until 4 epochs have been completed.



```
902         double smallestError = 100;
903         for (int i = 0; i < error.size(); ++i)
904         {
905             if (std::abs(error[i]) < std::abs(smallestError))
906             {
907                 smallestError = error[i];
908             }
909         }
910         // std::cout << smallestError << ", " << b0Map[smallestError] << ", " << b1Map[smallestError] << std::endl;
911 
912         PredictedB0B1 b0b1Values{productData[0].product, b0Map[smallestError], b1Map[smallestError]};
913 
914         bool found = false;
915         for (int i = 0; i < slopeValue.size(); ++i)
916         {
917             if (slopeValue[i].product == productData[0].product)
918             {
919                 slopeValue[i] = b0b1Values;
920                 found = true;
921             }
922         }
923         if (!found)
924         {
925             slopeValue.push_back(b0b1Values);
926         }
927     }
```

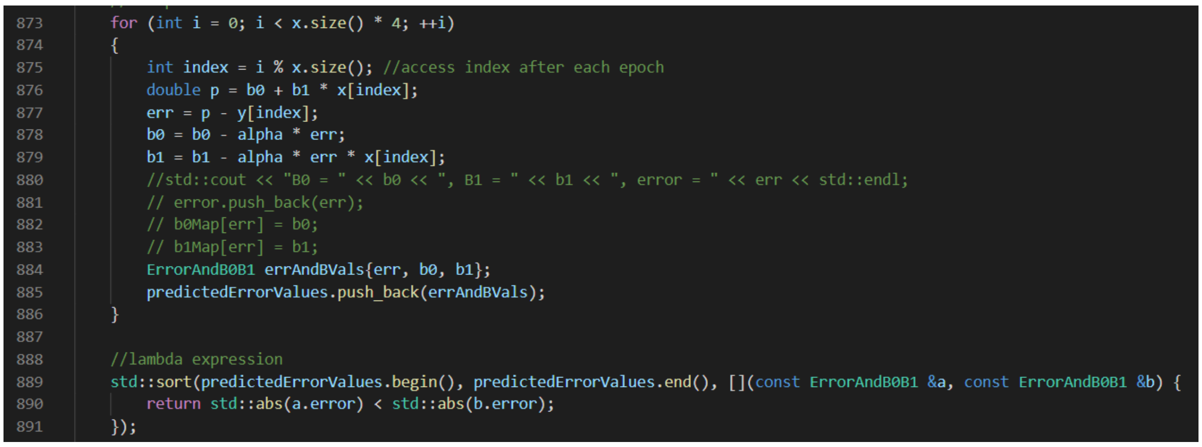
Figure 1.2C.1 – OrderBook.cpp (line 902 – 927)

The next step is to find the smallest error out of all the errors in the error vector by iterating through the error vector with a for loop. Inside the for loop, check if the error value of the instance is smaller than the smallest error and replace the smallestError variable if the error value is smaller. During the check, abs() function is needed as the error values could be negative or positive.

The smallest error in this case would be the one closest to 0 and thus, we have to use the absolute function to remove and negative signs before comparing them.

This method works in most cases but if a scenario occurs where 2 error values are the same, the b0 and b1 value retrieved from the map will not be correct. Thus, I have created a new class “ErrorAndB0B1” to store the 3 double values.

A vector of that class called predictedErrorValues was declared and in each iteration, we will be inserting the err, b0 and b1 values into the vector.



```
873     for (int i = 0; i < x.size() * 4; ++i)
874     {
875         int index = i % x.size(); //access index after each epoch
876         double p = b0 + b1 * x[index];
877         err = p - y[index];
878         b0 = b0 - alpha * err;
879         b1 = b1 - alpha * err * x[index];
880         //std::cout << "B0 = " << b0 << ", B1 = " << b1 << ", error = " << err << std::endl;
881         // error.push_back(err);
882         // b0Map[err] = b0;
883         // b1Map[err] = b1;
884         ErrorAndB0B1 errAndBVals{err, b0, b1};
885         predictedErrorValues.push_back(errAndBVals);
886     }
887
888     //lambda expression
889     std::sort(predictedErrorValues.begin(), predictedErrorValues.end(), [](const ErrorAndB0B1 &a, const ErrorAndB0B1 &b) {
890         return std::abs(a.error) < std::abs(b.error);
891     });
```

Figure 1.2C.2 – OrderBook.cpp (line 873 – 891)

Once the model completes its training, we will perform a sort using a custom sort to sort the error values in ascending order. We will be using lambda expressions to perform the sort. a will be the earlier element of the vector while b is the later element. The comparison sort will be based on the absolute value of the error, which was previously explained.

Next, we create a PredictedB0B1 class variable with {string product, double b0, double b1}.

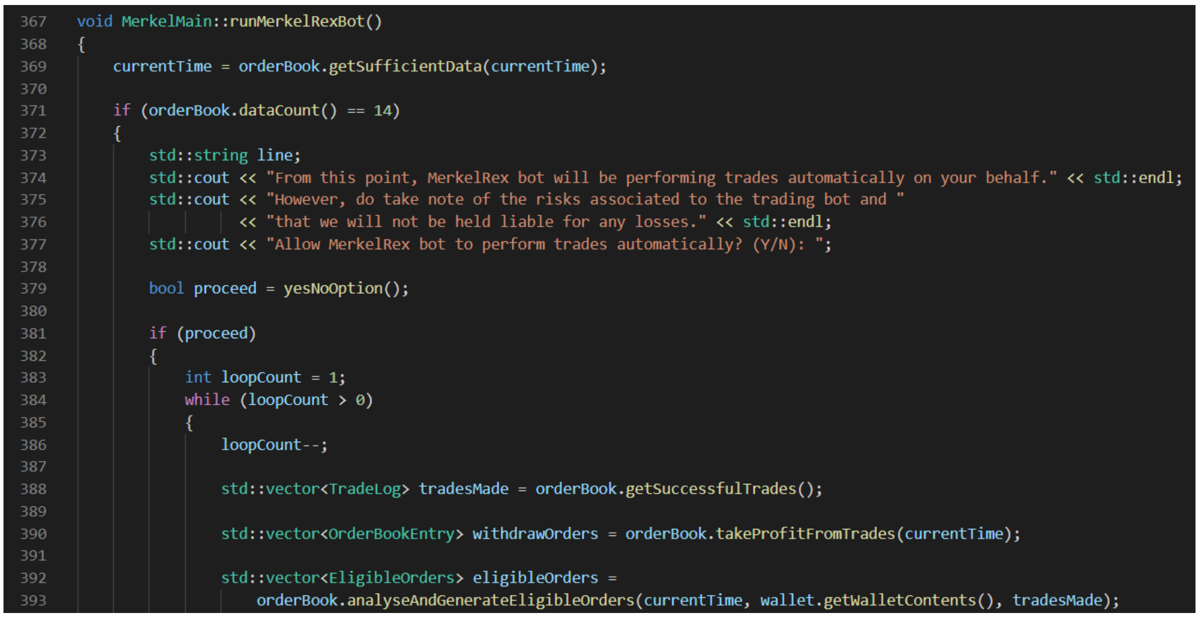
Lastly, from line 905 to 917, we will replace the old predicted b0 and b1 values for the product of the instance with the newly created ones.

(R1B Achieved)

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

Steps leading up to generating orders:



```
367 void MerkelMain::runMerkelRexBot()
368 {
369     currentTime = orderBook.getSufficientData(currentTime);
370 
371     if (orderBook.dataCount() == 14)
372     {
373         std::string line;
374         std::cout << "From this point, MerkelRex bot will be performing trades automatically on your behalf." << std::endl;
375         std::cout << "However, do take note of the risks associated to the trading bot and "
376                   << "that we will not be held liable for any losses." << std::endl;
377         std::cout << "Allow MerkelRex bot to perform trades automatically? (Y/N): ";
378 
379         bool proceed = yesNoOption();
380 
381         if (proceed)
382         {
383             int loopCount = 1;
384             while (loopCount > 0)
385             {
386                 loopCount--;
387 
388                 std::vector<TradeLog> tradesMade = orderBook.getSuccessfulTrades();
389 
390                 std::vector<OrderBookEntry> withdrawOrders = orderBook.takeProfitFromTrades(currentTime);
391 
392                 std::vector<EligibleOrders> eligibleOrders =
393                     orderBook.analyseAndGenerateEligibleOrders(currentTime, wallet.getWalletContents(), tradesMade);
```

Figure 1.1F – MerkelMain.cpp (line 367 – 393)

Resuming from figure 1.1F from R1A, in MerkelMain.cpp, currentTime now receives a new time stamp.

If the datacount is not equal to 14, it means that the user has selected the n option previously in sufficient data and thus, we will not perform any analysis and return back to the merkelrex menu. However, if true we will continue to perform the following actions.

Declaration of string variable 'line' in line 371 will be used later. The message in line 372 to 374 acts as a terms and conditions section for the bot and can be removed or commented out without affecting the functionality of the codes.

Boolean variable proceed is initialised with the return value from yesNoOption(). This function can be found in MerkelMain.cpp from line 528 to 550. It retrieves user input and returns true or false based on their input.

If user chooses to proceed, loopCount will be initialised with the value 1 and proceeds to the while loop. The while loop will run as long as the loopCount is greater than 1.

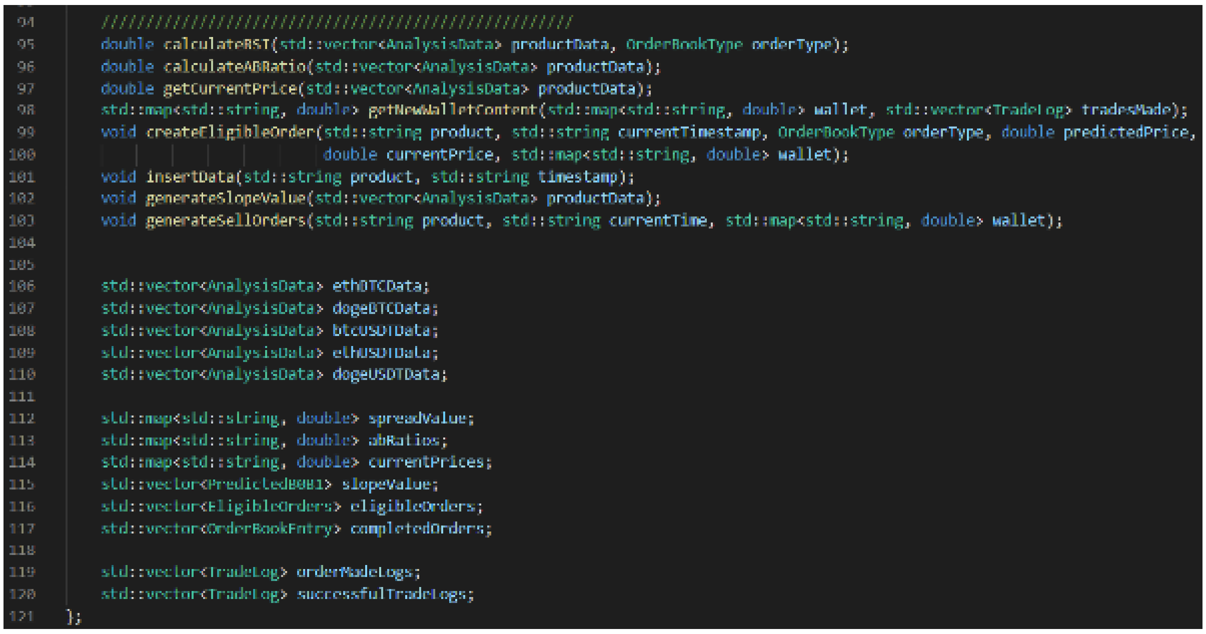
Inside the while loop, it will run analysis, order generation and other functions.

It first reduces loopCount by 1 and proceeds to retrieve any trades that have already been made successfully. Since we have not created any, tradesMade vector will be empty.

We will next proceed to create any potential withdrawal orders but will be explained later in R2D. However, we have to note that the approximate current price of all products will have been calculated and stored inside the currentPrices map in the OrderBook class when the takeProfitFromTrades() function runs.

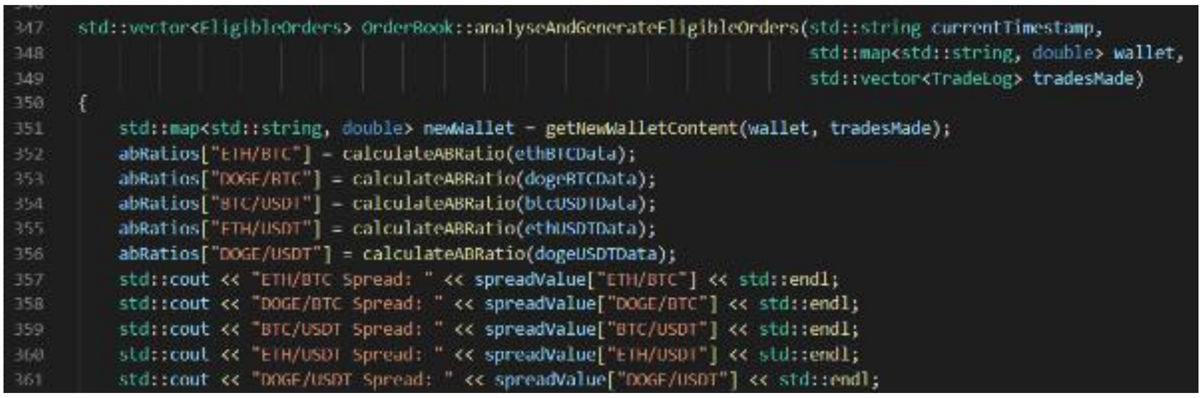
We will move on to generate eligible orders. EligibleOrders holds OrderBookEntry and a boolean value as its parameters. analyseAndGenerateEligibleOrders() takes string currentTime and map<string, double> wallet as its parameters.

Deciding when to generate orders:



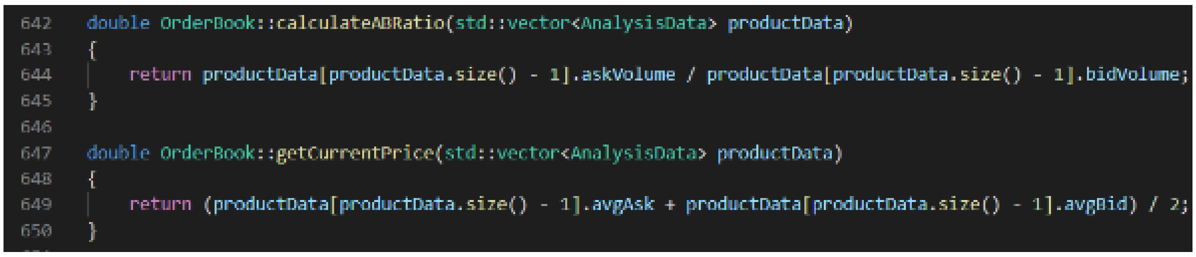
```
    ////////////////////////////////////////////////////
    double calculateBST(std::vector<AnalysisData> productData, OrderBookType orderType);
    double calculateABRatio(std::vector<AnalysisData> productData);
    double getCurrentPrice(std::vector<AnalysisData> productData);
    std::map<std::string, double> getNewWalletContent(std::map<std::string, double> wallet, std::vector<TradeLog> tradesMade);
    void createEligibleOrder(std::string product, std::string currentTimestamp, OrderBookType orderType, double predictedPrice,
                             double currentPrice, std::map<std::string, double> wallet);
    void insertData(std::string product, std::string timestamp);
    void generateSlopeValue(std::vector<AnalysisData> productData);
    void generateSellOrders(std::string product, std::string currentTime, std::map<std::string, double> wallet);


    std::vector<AnalysisData> ethBTCData;
    std::vector<AnalysisData> dogeBTCData;
    std::vector<AnalysisData> btcUSDTData;
    std::vector<AnalysisData> ethUSDTData;
    std::vector<AnalysisData> dogeUSDTData;

    std::map<std::string, double> spreadValue;
    std::map<std::string, double> abRatios;
    std::map<std::string, double> currentPrices;
    std::vector<PredictedBOB1> slopeValue;
    std::vector<EligibleOrders> eligibleOrders;
    std::vector<OrderBookEntry> completedOrders;

    std::vector<TradeLog> orderMadeLogs;
    std::vector<TradeLog> successfulTradeLogs;
};
```

Figure 2.1A – OrderBook.h (line 94 – 121)



```
std::vector<EligibleOrders> OrderBook::analyseAndGenerateEligibleOrders(std::string currentTimestamp,
                                                                        std::map<std::string, double> wallet,
                                                                        std::vector<TradeLog> tradesMade)
{
    std::map<std::string, double> newWallet = getNewWalletContent(wallet, tradesMade);
    abRatios["ETH/BTC"] = calculateABRatio(ethBTCData);
    abRatios["DOGE/BTC"] = calculateABRatio(dogeBTCData);
    abRatios["BTC/USDT"] = calculateABRatio(btcUSDTData);
    abRatios["ETH/USDT"] = calculateABRatio(ethUSDTData);
    abRatios["DOGE/USDT"] = calculateABRatio(dogeUSDTData);
    std::cout << "ETH/BTC Spread: " << spreadValue["ETH/BTC"] << std::endl;
    std::cout << "DOGE/BTC Spread: " << spreadValue["DOGE/BTC"] << std::endl;
    std::cout << "BTC/USDT Spread: " << spreadValue["BTC/USDT"] << std::endl;
    std::cout << "ETH/USDT Spread: " << spreadValue["ETH/USDT"] << std::endl;
    std::cout << "DOGE/USDT Spread: " << spreadValue["DOGE/USDT"] << std::endl;
```

Figure 2.1B – OrderBook.cpp (line 347 – 380)



```
double OrderBook::calculateABRatio(std::vector<AnalysisData> productData)
{
    return productData[productData.size() - 1].askVolume / productData[productData.size() - 1].bidVolume;
}

double OrderBook::getCurrentPrice(std::vector<AnalysisData> productData)
{
    return (productData[productData.size() - 1].avgAsk + productData[productData.size() - 1].avgBid) / 2;
}
```

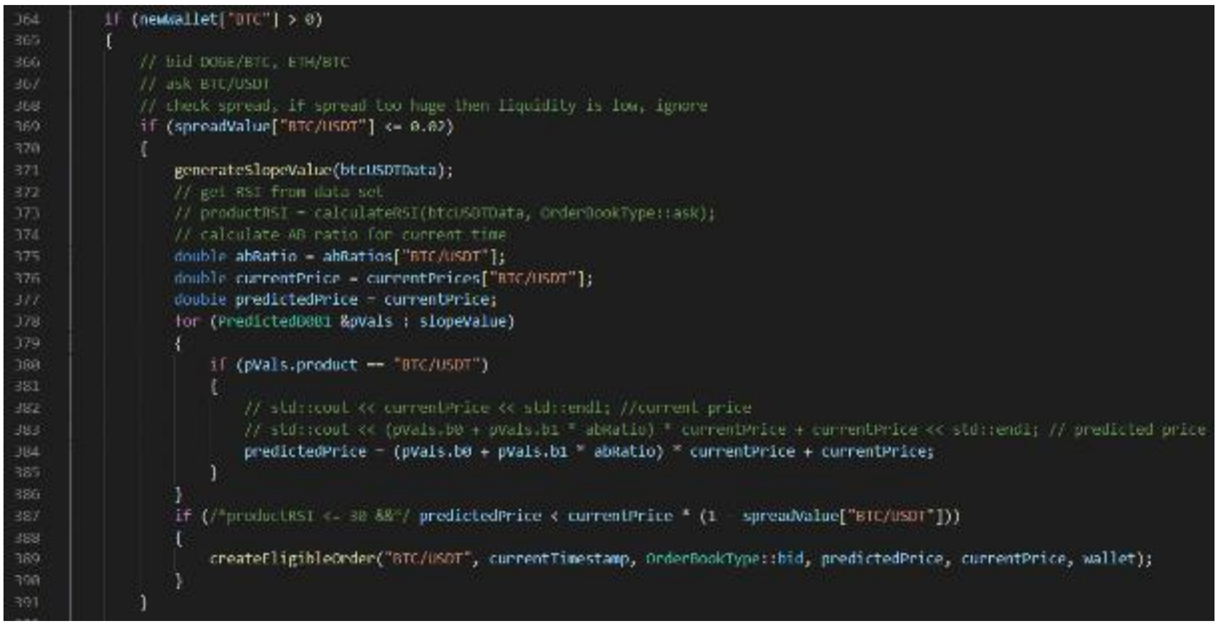
Figure 2.1C – OrderBook.cpp (line 644 – 652)

analyseAndGenerateEligibleOrders() function can be found in OrderBook.cpp from line 347 to 597.

As seen in figure 2.1B, analyseAndGenerateEligibleOrders() function starts by calculating the ab ratio of each product.

In this program, we are using the median of the average ask and average bid to get the current price. As mentioned previously current price is calculated when the takeProfitFromTrades() function run.

Line 357 to 361 in figure 2.1B prints out the spread of each products.



```cpp
    if (newWallet["BTC"] > 0)
    {
        // bid DOGE/BTC, ETH/BTC
        // ask BTC/USDT
        // check spread, if spread too huge then liquidity is low, ignore
        if (spreadValue["BTC/USDT"] <= 0.02)
        {
            generateSlopeValue(btcUSDTData);
            // get RSI from data set
            // productRSI = calculateRSI(btcUSDTData, OrderBookType::ask);
            // calculate AB ratio for current time
            double abRatio = abRatios["BTC/USDT"];
            double currentPrice = currentPrices["BTC/USDT"];
            double predictedPrice = currentPrice;
            for (PredictedD001 &pVals : slopeValue)
            {
                if (pVals.product == "BTC/USDT")
                {
                    // std::cout << currentPrice << std::endl; //current price
                    // std::cout << (pVals.b0 + pVals.b1 * abRatio) * currentPrice + currentPrice << std::endl; // predicted price
                    predictedPrice = (pVals.b0 + pVals.b1 * abRatio) * currentPrice + currentPrice;
                }
            }
            if (/*productRSI <= 30 &&*/ predictedPrice < currentPrice * (1 - spreadValue["BTC/USDT"]))
            {
                createEligibleOrder("BTC/USDT", currentTimestamp, OrderBookType::bid, predictedPrice, currentPrice, wallet);
            }
        }
```

Figure 2.1D – OrderBook.cpp (line 364 – 391)

Once the ab ratios and current prices have been calculated and stored, we will proceed with the algorithm as seen in figure 2.1D. We will first check if the wallet has more than 0 for each currency as we will only want to analyse products that we have sufficient currency to perform trades on.

The next if check will check that the spread value for that product is lesser than or equal to 0.02. We want to trade products that have lower spread as it translates to lower costs or fees. The check will only be for products that are limited to the currency on hand. e.g. if wallet contains more than 0 BTC, check the spread of BTC/USDT, DOGE/BTC and ETH/BTC.

How I arrived on the value 0.02 is that I want to be trading products that have less than 1% spread, but as I have divided the spread by the asks.size() as mentioned in R1A, I will have to divide 1 by 50 which equals to 0.02. However, in the data, DOGE/BTC only has 38 ask and 38 bid orders. So the calculation should have been 1/38 which is equal to 2.631…. but to simplify things, we will standardise it at 0.02.

If the product spread meets the criteria, we will perform linear regression to calculate the slope value of the product by calling the generateSlopeValue() function with the relevant product data as the input.

I declared 3 double variables in line 375 to 377 of Figure 2.1D. abRatio and currentPrice variable is not necessary but I felt that it will be easier to write the formula to calculate the predicted price as seen in line 384.

The predicted price can be calculated with the formula of:

Linear regression equation: $y = b0 + b1*x$ || $y = a + bx$

$$\therefore \text{ predictedPrice} = \underbrace{\underbrace{(b0 + b1 * abRatio)}_{\boxed{\text{predicted change in \%}}} * \text{currentPrice}}_{\boxed{\text{predicted change}}} + \text{currentPrice}$$

It is required to add the currentPrice as the y value will only represent the change in % of the currentPrice and multiplying the currentPrice will represent the predicted actual change in the same unit value of the product.

Line 389 next if():

After deriving on a predicted price of the product, we will check if the predicted price is less than the current price multiplied by (1 – spread value of product). We want to check if the predicted price will fall lower than the cost of exchanging the currencies and thus, current price – current price * spread value of product.

Depending on what currency you are holding and the product to be exchanged, it could be an if statement of:

(predictedPrice > currentPrice * (1 + spread value of product) // bid

or

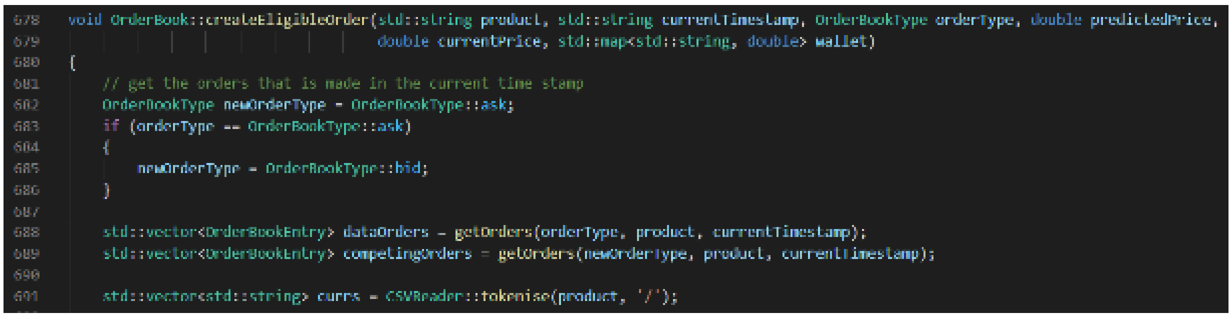
(predictedPrice < currentPrice * (1 – spread value of product) // ask

If the currency held in the wallet is the currency on the left of the exchange product, we will only be able to make an ask offer.

An example would be if the currency held is BTC, in the exchange product of BTC/USDT, we will only be able to make an ask as a bid offer for BTC/USDT will mean that we want to buy BTC with USDT.

When the only option is to make a bid offer, we want the price to grow in the future and the price to fall if we can only make an ask offer.

If the criteria are met, we will move on to create an eligible order with the createEligibleOrder() function. Eligible orders are not actual orders that are put up in the order book and will have to go through further checks before an order is made at a later time.



```
678  void OrderBook::createEligibleOrder(std::string product, std::string currentTimestamp, OrderBookType orderType, double predictedPrice,
679                                     double currentPrice, std::map<std::string, double> wallet)
680  {
681      // get the orders that is made in the current time stamp
682      OrderBookType newOrderType = OrderBookType::ask;
683      if (orderType == OrderBookType::ask)
684      {
685          newOrderType = OrderBookType::bid;
686      }
687
688      std::vector<OrderBookEntry> dataOrders = getOrders(orderType, product, currentTimestamp);
689      std::vector<OrderBookEntry> competingOrders = getOrders(newOrderType, product, currentTimestamp);
690
691      std::vector<std::string> currs = CSVReader::tokenise(product, '/');
```

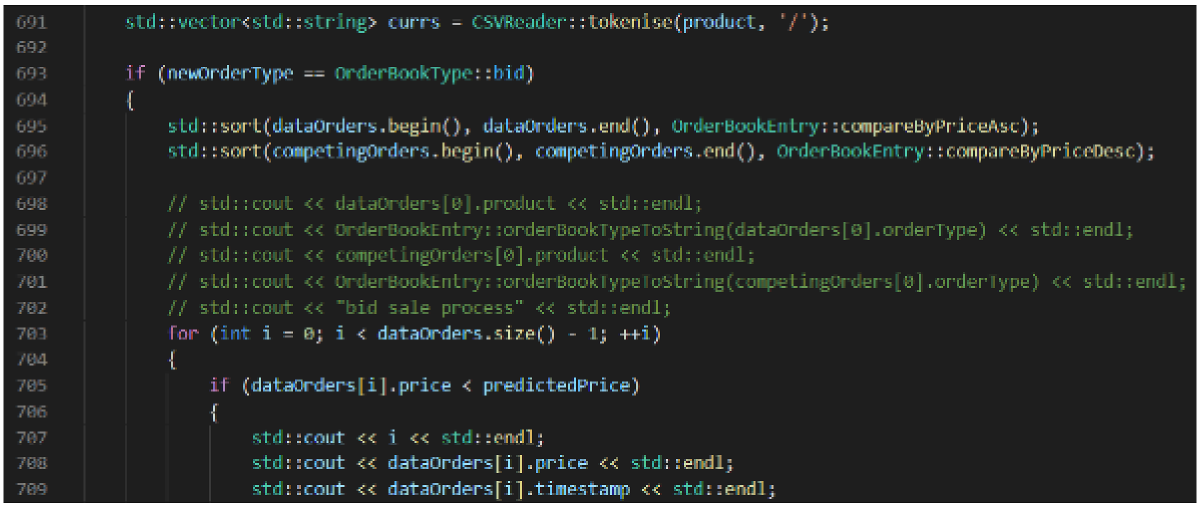
Figure 2.1E – OrderBook.cpp (line 678 – 691)

Inside the createEligibleOrder() function in OrderBook class, I started off by declaring an OrderBookType variable, newOrderType and initialise it with OrderBookType::ask. Next, we check that the orderType variable that was passed in to see if it is OrderBookType::ask and change the newOrderType to a bid type if true, else remain as an ask type.

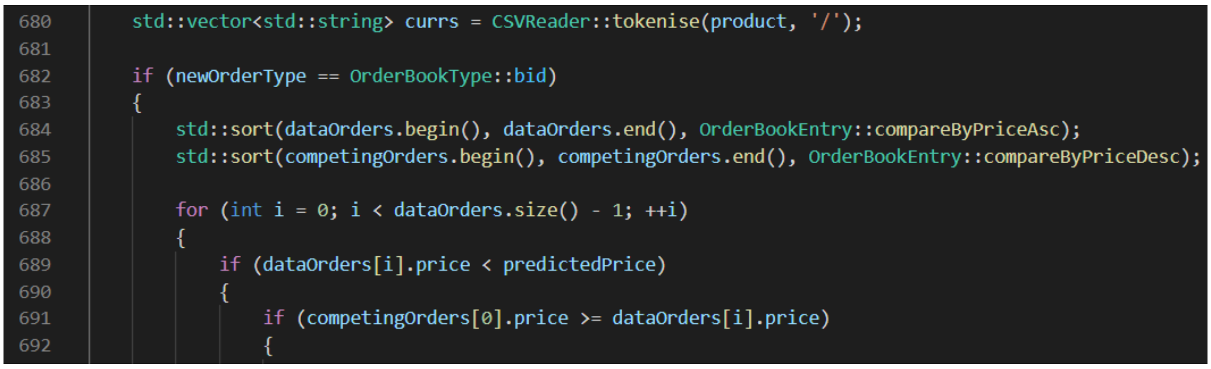
Next, we will declare two OrderBookEntry vectors named dataOrders and competingOrders. The dataOrders vector will be holding all the orders of the opposite OrderBookType of the same timestamp while competingOrders will hold orders of the same OrderBookType.

String vector currs is declared and initialised with the 2 currencies of the product achieved by using the tokenise function in CSVReader that separates string values by a delimiter, returning a vector of strings separated by that delimiter.

Criteria for creating eligible orders:



```
691      std::vector<std::string> currs = CSVReader::tokenise(product, '/');
692
693      if (newOrderType == OrderBookType::bid)
694      {
695          std::sort(dataOrders.begin(), dataOrders.end(), OrderBookEntry::compareByPriceAsc);
696          std::sort(competingOrders.begin(), competingOrders.end(), OrderBookEntry::compareByPriceDesc);
697
698          // std::cout << dataOrders[0].product << std::endl;
699          // std::cout << OrderBookEntry::orderBookTypeToString(dataOrders[0].orderType) << std::endl;
700          // std::cout << competingOrders[0].product << std::endl;
701          // std::cout << OrderBookEntry::orderBookTypeToString(competingOrders[0].orderType) << std::endl;
702          // std::cout << "bid sale process" << std::endl;
703          for (int i = 0; i < dataOrders.size() - 1; ++i)
704          {
705              if (dataOrders[i].price < predictedPrice)
706              {
707                  std::cout << i << std::endl;
708                  std::cout << dataOrders[i].price << std::endl;
709                  std::cout << dataOrders[i].timestamp << std::endl;
```

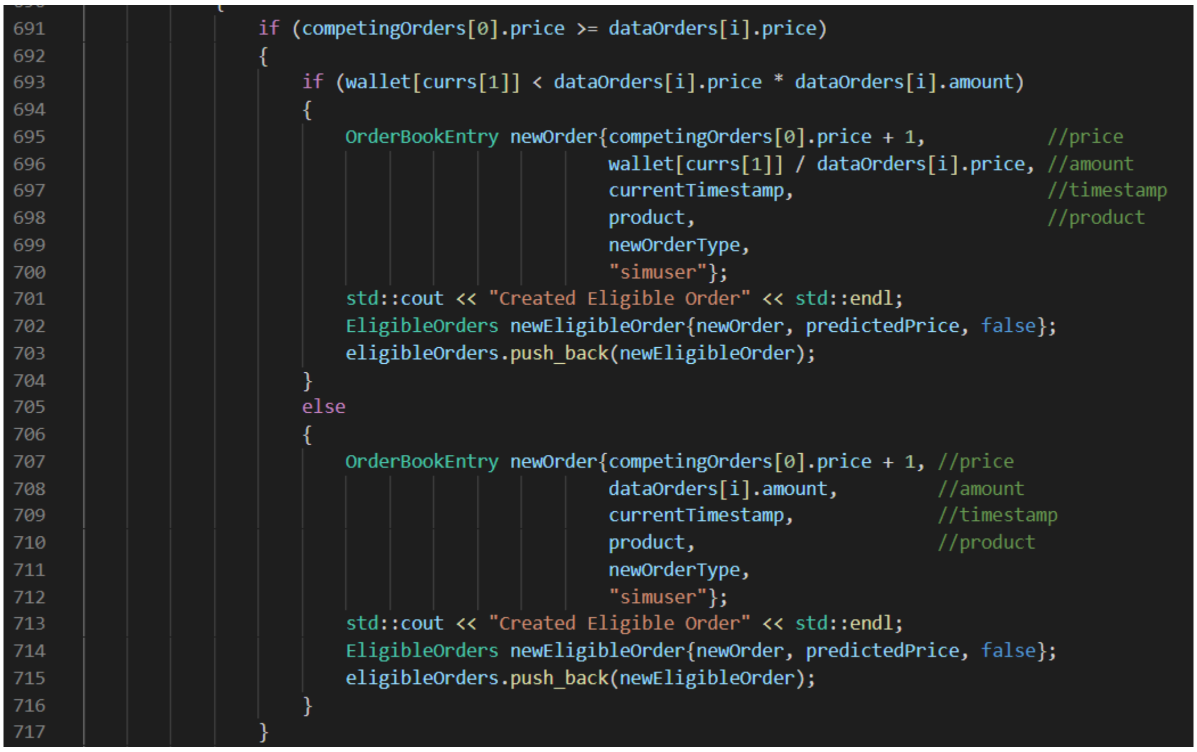


```cpp
    std::vector<std::string> curr = CSVReader::tokenise(product, '/');

    if (newOrderType == OrderBookType::bid)
    {
        std::sort(dataOrders.begin(), dataOrders.end(), OrderBookEntry::compareByPriceAsc);
        std::sort(competingOrders.begin(), competingOrders.end(), OrderBookEntry::compareByPriceDesc);

        for (int i = 0; i < dataOrders.size() - 1; ++i)
        {
            if (dataOrders[i].price < predictedPrice)
            {
                if (competingOrders[0].price >= dataOrders[i].price)
                {
```

Figure 2.1F – OrderBook.cpp (line 680 – 692)

We will go through the criteria for creating eligible orders. Depending on the OrderBookType of the newOrderType, we will be sorting the dataOrders and competingOrders differently. For bid orders, we will be sorting the dataOrders in ascending order and competingOrders in descending order as we want to start checking the lowest ask price and the highest bid price.

After sorting the orders, we will begin to iterate through the dataOrders vector with a for loop. Inside the for loop, we will check if the order price of index i of the dataOrders vector is lower than the predictedPrice. We want to check if the price is lower than the predicted price as it means that we can potentially make a profit by buying now and selling it in the future when it reaches the predicted price.



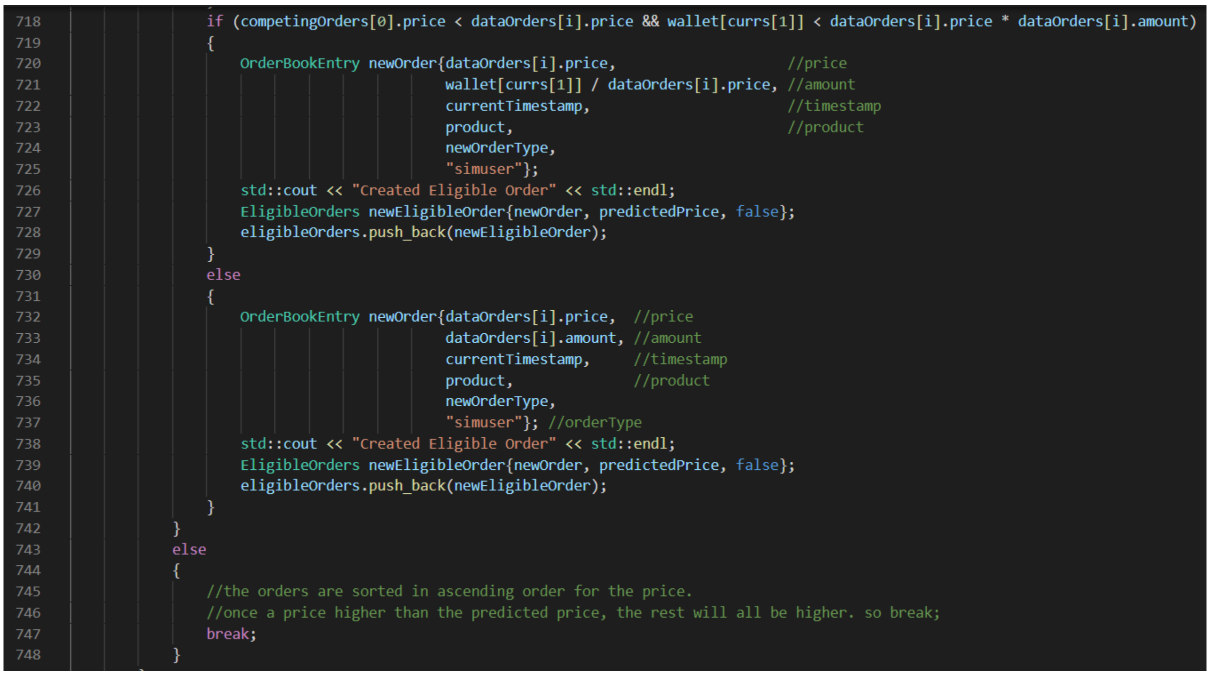
```cpp
                if (competingOrders[0].price >= dataOrders[i].price)
                {
                    if (wallet[curr[1]] < dataOrders[i].price * dataOrders[i].amount)
                    {
                        OrderBookEntry newOrder{competingOrders[0].price + 1,           //price
                                                wallet[curr[1]] / dataOrders[i].price, //amount
                                                currentTimestamp,                       //timestamp
                                                product,                                //product
                                                newOrderType,
                                                "simuser"};
                        std::cout << "Created Eligible Order" << std::endl;
                        EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
                        eligibleOrders.push_back(newEligibleOrder);
                    }
                    else
                    {
                        OrderBookEntry newOrder{competingOrders[0].price + 1, //price
                                                dataOrders[i].amount,         //amount
                                                currentTimestamp,             //timestamp
                                                product,                      //product
                                                newOrderType,
                                                "simuser"};
                        std::cout << "Created Eligible Order" << std::endl;
                        EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
                        eligibleOrders.push_back(newEligibleOrder);
                    }
                }
```

Figure 2.1G – OrderBook.cpp (line 691 – 717)

The next check will check if the highest competing bid order has a higher price than the ask order of dataOrders. This means that if we create a bid order that matches the ask order price, the orders will not match as the higher bid will be matched.

Finally, we want to check if the wallet amount of the right currency is less than what the order requires. If true, the newOrder will be created with the price of the highest competing order + 1. This will ensure that our bid order will be prioritised as we have the highest bid. The amount will be the currency we have left divided by the ask order price. We do not have to divide it by our bid price as the sale price will match the asking price instead of the bidding price.

Else, the newOrder will have the amount equal to the dataOrders amount as our wallet will have sufficient currency to complete that order. We will push the new eligible orders into the eligibleOrders vector that holds all eligible orders.



```
718                     if (competingOrders[0].price < dataOrders[i].price && wallet[currs[1]] < dataOrders[i].price * dataOrders[i].amount)
719                     {
720                         OrderBookEntry newOrder{dataOrders[i].price,                    //price
721                                                 wallet[currs[1]] / dataOrders[i].price, //amount
722                                                 currentTimestamp,                       //timestamp
723                                                 product,                                //product
724                                                 newOrderType,
725                                                 "simuser"};
726                         std::cout << "Created Eligible Order" << std::endl;
727                         EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
728                         eligibleOrders.push_back(newEligibleOrder);
729                     }
730                     else
731                     {
732                         OrderBookEntry newOrder{dataOrders[i].price,  //price
733                                                 dataOrders[i].amount, //amount
734                                                 currentTimestamp,     //timestamp
735                                                 product,              //product
736                                                 newOrderType,
737                                                 "simuser"}; //orderType
738                         std::cout << "Created Eligible Order" << std::endl;
739                         EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
740                         eligibleOrders.push_back(newEligibleOrder);
741                     }
742                 }
743                 else
744                 {
745                     //the orders are sorted in ascending order for the price.
746                     //once a price higher than the predicted price, the rest will all be higher. so break;
747                     break;
748                 }
```

Figure 2.1H – OrderBook.cpp (line 718 – 748)

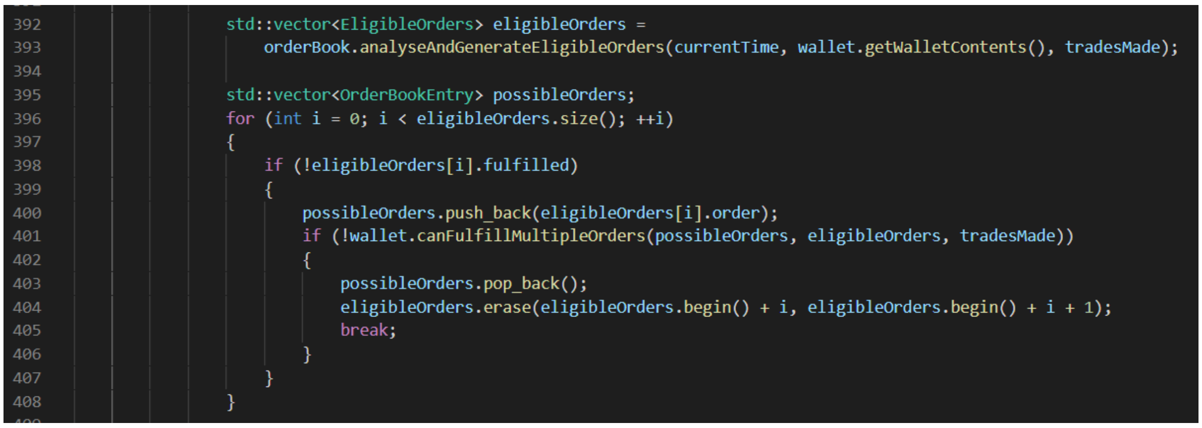
If the competing bid price is not higher than the ask price as seen in the if check in figure 2.1G, we will check if the highest competing bid price is lesser than the ask price and that the wallet have insufficient currency to match the ask order. If true, we will create a new order matching the ask price and an amount that the currency on hand could trade. The new order is then pushed into the eligibleOrders vector.

If false, then a new order matching the price and amount of the ask order will be matched and pushed to the eligibleOrders vector. The else statement on line 743 comes after the if check of (dataOrders[i].price < predictedPrice) as seen in figure 2.1F.

The reason why we break from the for loop when dataOrders[i].price is greater than predictedPrice is because the data orders are sorted in ascending order. If dataOrders of index i have price greater than the predictedPrice, all dataOrders of index greater than i will also have prices that are greater than the predictedPrice and thus, we will break from the loop.

As the createEligibleOrders() function have no return value, we will return to the point of analyseAndGenerateEligibleOrders() function as seen in figure 2.1D. Once we have created eligible orders for the product, we will proceed to check if the next product meets the same criteria as we have gone through.

Final check before passing orders for matching:



```
                std::vector<EligibleOrders> eligibleOrders =
                    orderBook.analyseAndGenerateEligibleOrders(currentTime, wallet.getWalletContents(), tradesMade);

                std::vector<OrderBookEntry> possibleOrders;
                for (int i = 0; i < eligibleOrders.size(); ++i)
                {
                    if (!eligibleOrders[i].fulfilled)
                    {
                        possibleOrders.push_back(eligibleOrders[i].order);
                        if (!wallet.canFulfillMultipleOrders(possibleOrders, eligibleOrders, tradesMade))
                        {
                            possibleOrders.pop_back();
                            eligibleOrders.erase(eligibleOrders.begin() + i, eligibleOrders.begin() + i + 1);
                            break;
                        }
                    }
                }
```

Figure 2.1l – MerkelMain.cpp (line 392 – 408)

After we have gone through all currencies in the wallet and created all eligible orders that have matched the criteria, we will create a new vector of OrderBookEntry to hold all orders that we will be passing to the exchange for matching.

We will iterate through the eligibleOrders with a for loop. We will first check if the eligible order is not fulfilled, where newly created eligible orders are all set to false. If not fulfilled, we will push the order instance of the eligibleOrders into the possibleOrders vector.

Our next if check attempts to check if the wallet have sufficient currency to complete the order. We will move into the wallet.cpp file of line 133 to 168, to run through how the canFulfillMultipleOrders() function work.

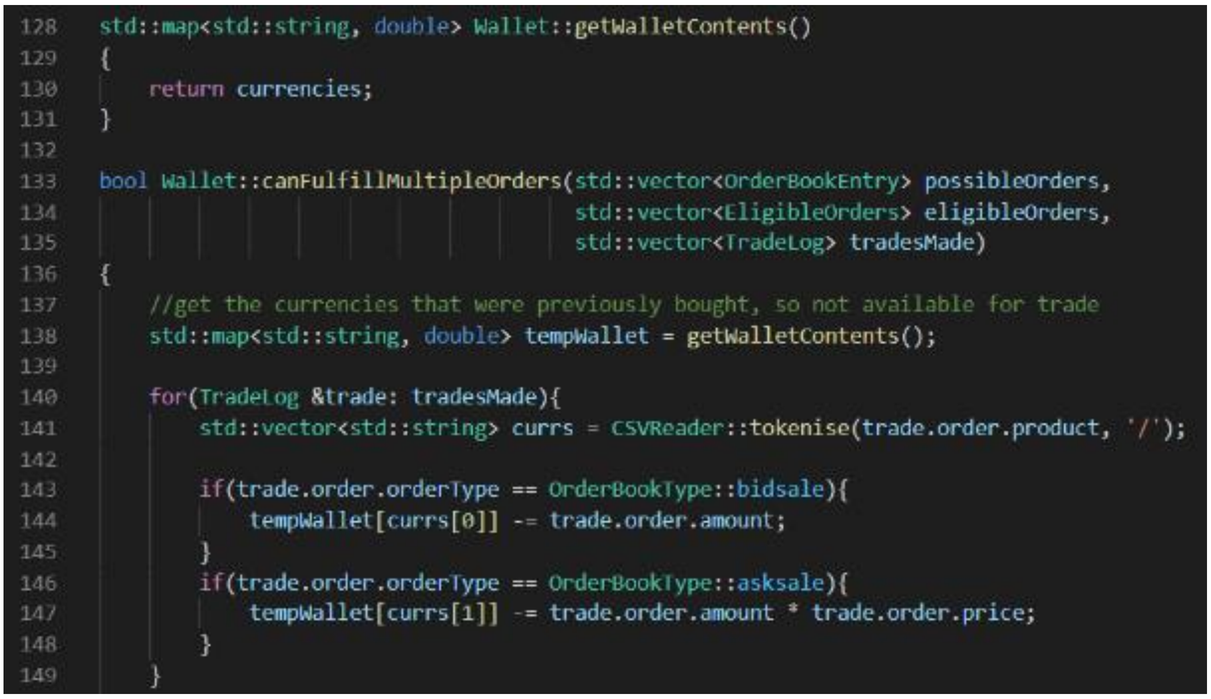
It takes in OrderBookEntry, EligibleOrders and TradeLog class vectors as its input parameters.

OrderBookEntry{ double price, double amount, string timestamp, string product, OrderBookType orderType, string username }

EligibleOrders{ OrderBookEntry order, double targetPrice, bool fulfilled }

TradeLog{ OrderBookEntry order, double avgAsk, double avgBid }

The reason why we have to perform this check is because all the eligible orders are created as long as the wallet contains more than 1 of that currency and meets the criteria. Therefore, we have to filter out the orders that cannot be made with what is left in the wallet.



```
128  std::map<std::string, double> Wallet::getWalletContents()
129  {
130      return currencies;
131  }
132
133  bool Wallet::canFulfillMultipleOrders(std::vector<OrderBookEntry> possibleOrders,
134                                        std::vector<EligibleOrders> eligibleOrders,
135                                        std::vector<TradeLog> tradesMade)
136  {
137      //get the currencies that were previously bought, so not available for trade
138      std::map<std::string, double> tempWallet = getWalletContents();
139
140      for(TradeLog &trade: tradesMade){
141          std::vector<std::string> currs = CSVReader::tokenise(trade.order.product, '/');
142
143          if(trade.order.orderType == OrderBookType::bidsale){
144              tempWallet[currs[0]] -= trade.order.amount;
145          }
146          if(trade.order.orderType == OrderBookType::asksale){
147              tempWallet[currs[1]] -= trade.order.amount * trade.order.price;
148          }
149      }
```

Figure 2.1l – Wallet.cpp (line 128 – 149)

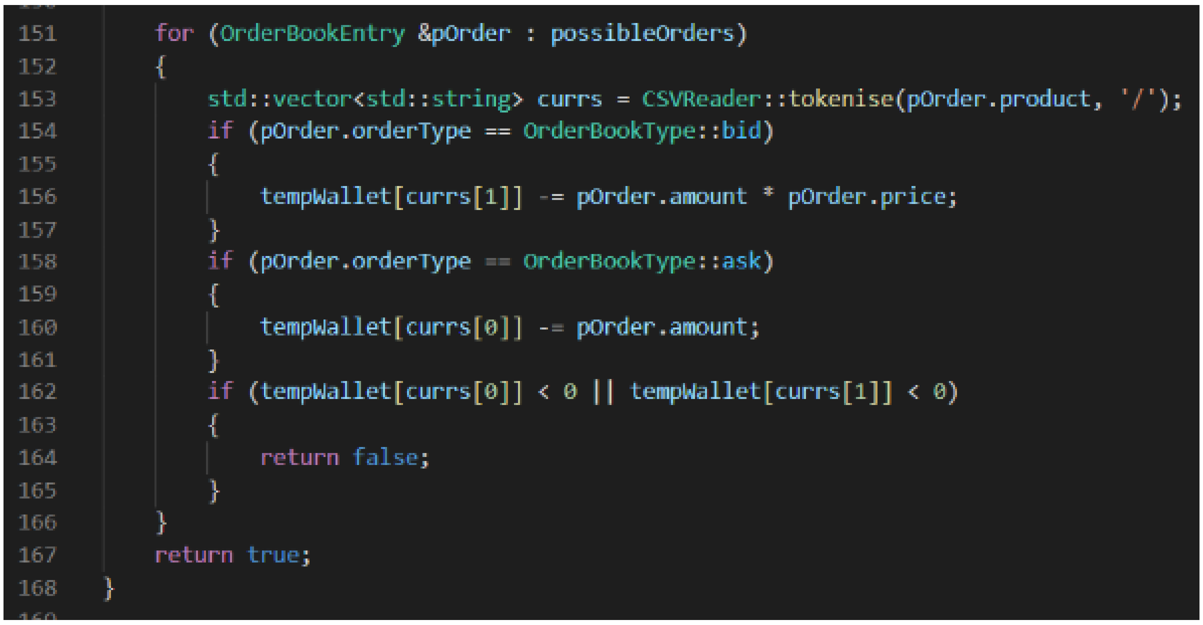
The tradesMade vector will be holding all the orders that have successfully been traded previously.

We will create a temporary wallet variable that has the same currencies as the actual wallet. We will be deducting the amount needed to perform the possible orders from the temporary wallet and if the currency reaches below 0, it will be an invalid order.

Before performing that check, we want to subtract all currencies that was obtained from any previous trades as we will not want to trade them before it hits the predicted target price of the product. As seen in figure 2.1l from line 140 to 149, we will check if the order type is a bidsale or asksale and subtract the respective currencies traded from the temporary wallet.

When the order is a bid, we will be using the currency on the right of the product to trade for the currency on the left and thus, we will need to have sufficient currency to trade back to the original currency.

The bid amount can be retrieved easily by getting the order amount while the ask amount must be calculated by using the order amount multiplied by the price.



```cpp
    for (OrderBookEntry &pOrder : possibleOrders)
    {
        std::vector<std::string> currs = CSVReader::tokenise(pOrder.product, '/');
        if (pOrder.orderType == OrderBookType::bid)
        {
            tempWallet[currs[1]] -= pOrder.amount * pOrder.price;
        }
        if (pOrder.orderType == OrderBookType::ask)
        {
            tempWallet[currs[0]] -= pOrder.amount;
        }
        if (tempWallet[currs[0]] < 0 || tempWallet[currs[1]] < 0)
        {
            return false;
        }
    }
    return true;
}
```

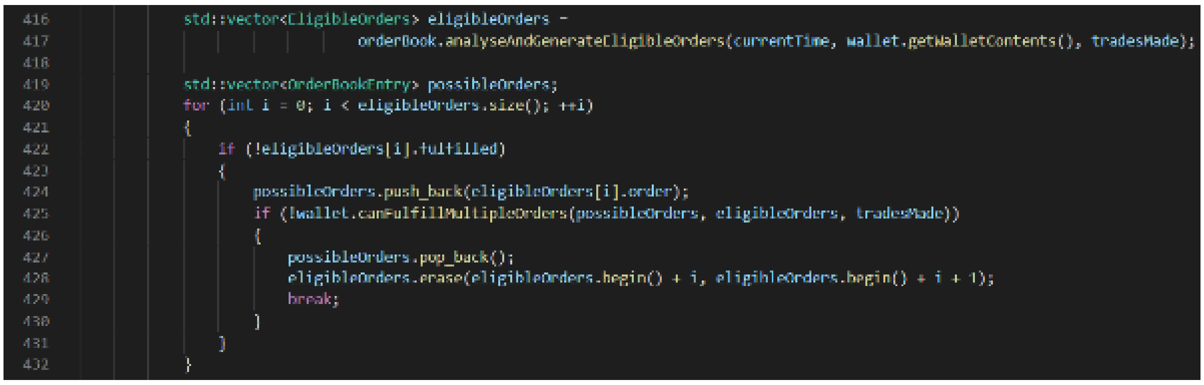
Figure 2.1J – Wallet.cpp (line 151 – 168)

We will now proceed to iterate through the possibleOrders in a for loop to access each of those orders. We will separate the product with the '/' delimiter and store the 2 currencies into the currs string vector.

If the order is a bid order type, we will have to multiply the order amount and price before subtracting the currency by the calculated multiplication value as the order amount is the amount of currency to be traded while the price represents the exchange rate. If the price is 1.5, it means to bid for the currency on the left of the product, we will have to pay 1.5 with the currency on the right of the product (ETH(1) : BTC(price)).

After subtracting the amount of currency, we will check if any of the currencies of the instance in the temporary wallet is below 0. If true, we will return false, which means that the wallet will not be able to fulfil all of the orders in the possibleOrders vector and return true if the for loop finishes and currencies have not reached below 0.

We will move back to figure 2.1I back at the if statement.



```cpp
std::vector<EligibleOrders> eligibleOrders =
                    orderBook.analyseAndGenerateEligibleOrders(currentTime, wallet.getWalletContents(), tradesMade);

std::vector<OrderBookEntry> possibleOrders;
for (int i = 0; i < eligibleOrders.size(); ++i)
{
    if (!eligibleOrders[i].fulfilled)
    {
        possibleOrders.push_back(eligibleOrders[i].order);
        if (!wallet.canFulfillMultipleOrders(possibleOrders, eligibleOrders, tradesMade))
        {
            possibleOrders.pop_back();
            eligibleOrders.erase(eligibleOrders.begin() + i, eligibleOrders.begin() + i + 1);
            break;
        }
    }
}
```

Figure 2.1I – MerkelMain.cpp (line 416 – 432)

If the return value from canFulfillMultipleOrders() function is false, we will use the pop_back() function to remove the last element of the vector in the possibleOrders vector. Even though it is not necessary, we will be also erase the eligible order from the eligibleOrders()as we will not be needing it anymore. However, there will be a function at a later part of the code that removes all eligible orders that are unable to be fulfilled.

Improvements:

1)

An improvement of this code could be that when we get a false value as the wallet is unable to fulfilled the last order in the possibleOrders vector, we could change the amount to fit what is left in the wallet. If this improvement is implemented, there will not be a break in line 429.

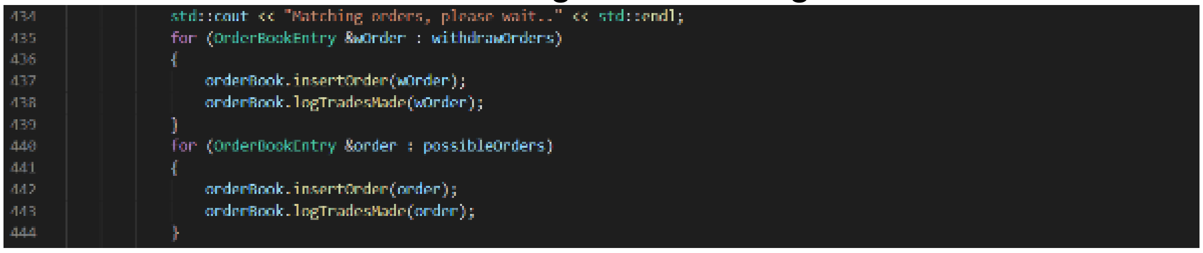
As the improvement have not been implemented, without a break, we could possibly be able to fulfil a smaller order with a higher price which would not be the most logical way to trade and thus, I have inserted a break line to break out of the for loop once we have found the maximum full eligible orders that the currencies in the wallet can fulfil.

2)

Another improvement in terms of program running time is that the program could have checked the wallet currency contents before making an eligible order. This could improve the run time of the program as we do not have to loop through the eligibleOrders created to check them. However, doing so will make the code confusing with many if statements bundled together and thus, I have chosen to separate them.

(R2A Achieved)

R2B: Pass the bids to the exchange for matching



```
434            std::cout << "Matching orders, please wait.." << std::endl;
435            for (OrderBookEntry &wOrder : withdrawOrders)
436            {
437                orderBook.insertOrder(wOrder);
438                orderBook.logTradesMade(wOrder);
439            }
440            for (OrderBookEntry &order : possibleOrders)
441            {
442                orderBook.insertOrder(order);
443                orderBook.logTradesMade(order);
444            }
```

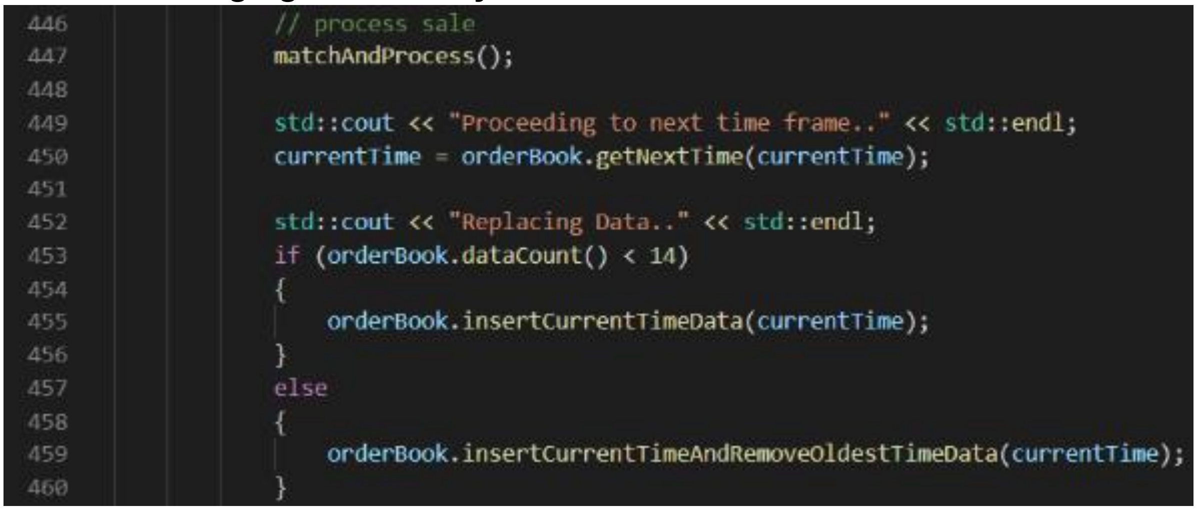
Figure 2.2A – MerkelMain.cpp (line 434 – 444)

I will be explaining the withdrawal orders in R2D so I will skip line 435 to 439.

In line 440 to 444, we will be passing all orders into the exchange for matching by inserting them into the orderBook. The logTradesMade() function will be talked about in R4 of logging.

(R2B Achieved)

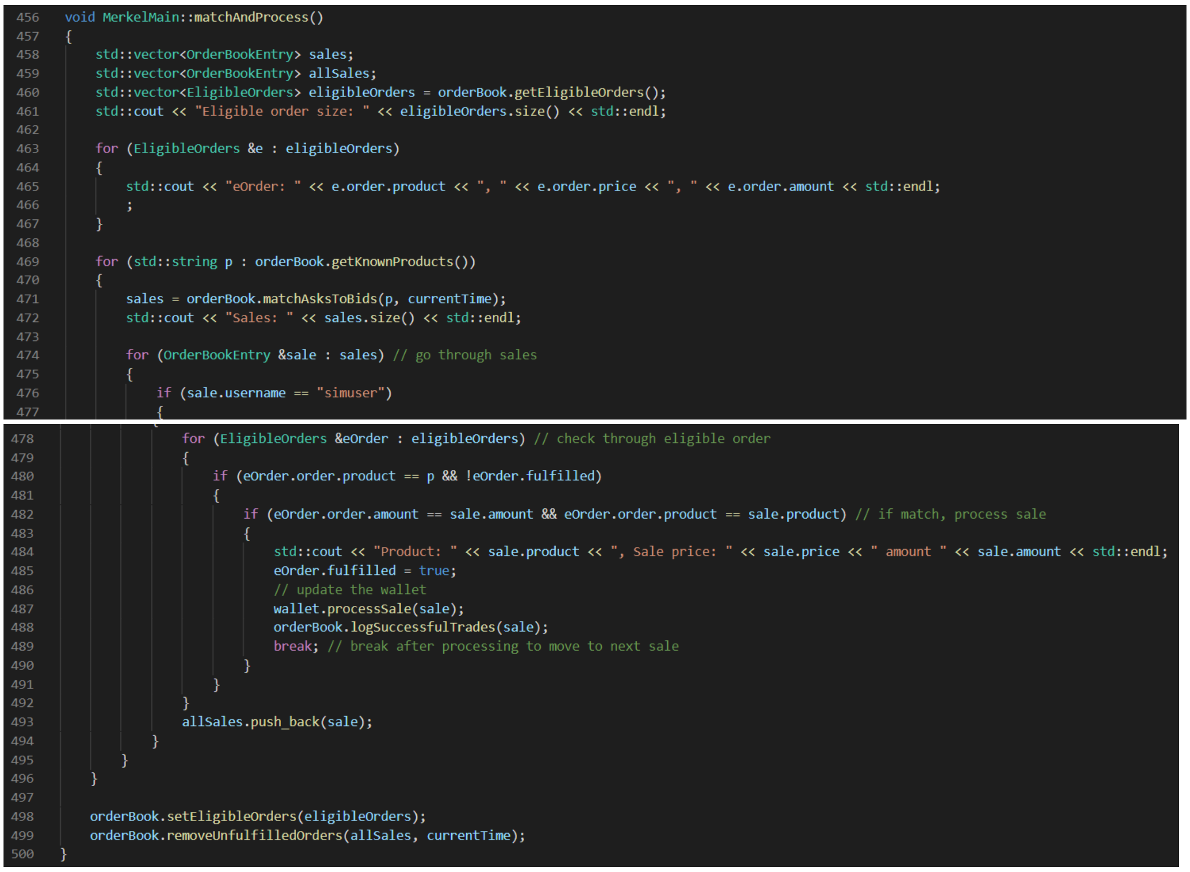
R2C: Receive the results of the exchange’s matching engine (which decides which bids have been accepted) which might involve exchanging assets according to the bid and the matching, and the cost of the exchange generated by the simulation



```cpp
            // process sale
            matchAndProcess();

            std::cout << "Proceeding to next time frame.." << std::endl;
            currentTime = orderBook.getNextTime(currentTime);

            std::cout << "Replacing Data.." << std::endl;
            if (orderBook.dataCount() < 14)
            {
                orderBook.insertCurrentTimeData(currentTime);
            }
            else
            {
                orderBook.insertCurrentTimeAndRemoveOldestTimeData(currentTime);
            }
```

Figure 2.3A – MerkelMain.cpp (line 446 – 460)

After orders have been inserted into the orderBook, We will begin run the matching engine by calling the matchAndProcess() function.



```cpp
void MerkelMain::matchAndProcess()
{
    std::vector<OrderBookEntry> sales;
    std::vector<OrderBookEntry> allSales;
    std::vector<EligibleOrders> eligibleOrders = orderBook.getEligibleOrders();
    std::cout << "Eligible order size: " << eligibleOrders.size() << std::endl;

    for (EligibleOrders &e : eligibleOrders)
    {
        std::cout << "eOrder: " << e.order.product << ", " << e.order.price << ", " << e.order.amount << std::endl;
        ;
    }

    for (std::string p : orderBook.getKnownProducts())
    {
        sales = orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;

        for (OrderBookEntry &sale : sales) // go through sales
        {
            if (sale.username == "simuser")
            {
                for (EligibleOrders &eOrder : eligibleOrders) // check through eligible order
                {
                    if (eOrder.order.product == p && !eOrder.fulfilled)
                    {
                        if (eOrder.order.amount == sale.amount && eOrder.order.product == sale.product) // if match, process sale
                        {
                            std::cout << "Product: " << sale.product << ", Sale price: " << sale.price << " amount " << sale.amount << std::endl;
                            eOrder.fulfilled = true;
                            // update the wallet
                            wallet.processSale(sale);
                            orderBook.logSuccessfulTrades(sale);
                            break; // break after processing to move to next sale
                        }
                    }
                }
                allSales.push_back(sale);
            }
        }
    }

    orderBook.setEligibleOrders(eligibleOrders);
    orderBook.removeUnfulfilledOrders(allSales, currentTime);
}
```

Figure 2.3B – MerkelMain.cpp (line 456 – 500)

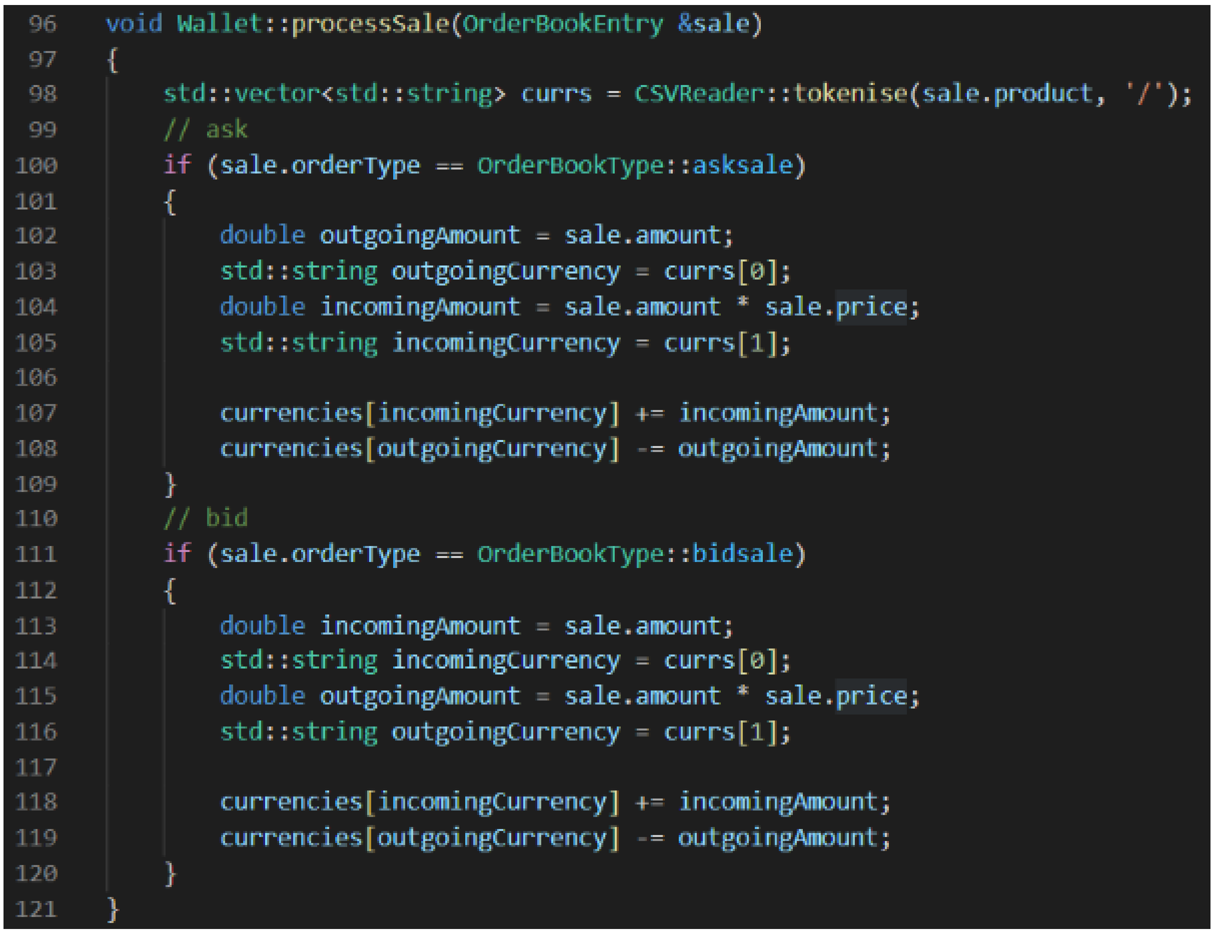
In the matchAndProcess() function, we will declare a sale and allSales vector of OrderBookEntry along with a EligibleOrders vector to retrieve all eligible orders as seen in line 345. Next, we will iterate through each product to run the matching engine.

At the start of every iteration, we will set sales equals to the return value of matchAsksToBids(), inputting product and current time as the input parameters.

We will proceed to iterate through each eligible order with for loop and we will check if the instance of the eligible orders is equal to the product p. We will also check if the eligible order is fulfilled.

If true, we will iterate through each sale of the sales vector to check if the sale has the username of “simuser”. We will also check if the eligibleOrder amount and product matches the sale amount and product. We do not check if the prices match as our bid price may not match the sale price due to the matching engine’s rule of sale price following the ask price when there is a match between ask and bid.

If the sale matches these criteria, we will set the fulfilled value of the eligible order of the instance to true. Next, we will call the processSale() function in Wallet class and pass on the current sale as the input.



```
 96  void Wallet::processSale(OrderBookEntry &sale)
 97  {
 98      std::vector<std::string> currs = CSVReader::tokenise(sale.product, '/');
 99      // ask
100      if (sale.orderType == OrderBookType::asksale)
101      {
102          double outgoingAmount = sale.amount;
103          std::string outgoingCurrency = currs[0];
104          double incomingAmount = sale.amount * sale.price;
105          std::string incomingCurrency = currs[1];
106
107          currencies[incomingCurrency] += incomingAmount;
108          currencies[outgoingCurrency] -= outgoingAmount;
109      }
110      // bid
111      if (sale.orderType == OrderBookType::bidsale)
112      {
113          double incomingAmount = sale.amount;
114          std::string incomingCurrency = currs[0];
115          double outgoingAmount = sale.amount * sale.price;
116          std::string outgoingCurrency = currs[1];
117
118          currencies[incomingCurrency] += incomingAmount;
119          currencies[outgoingCurrency] -= outgoingAmount;
120      }
121  }
```

Figure 2.3C – Wallet.cpp (line 96 – 121)

We will need to know which currency to deduct, so we will have to split the product by the delimiter of ‘/’ and get the 2 currencies involved.

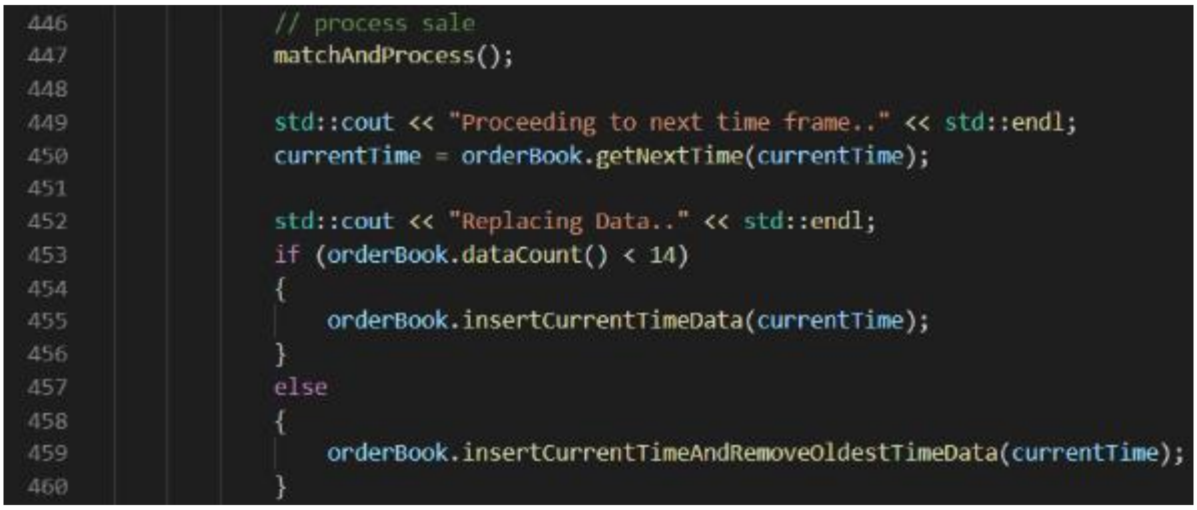
If the created sale is a bidsale, we will get the incoming amount of the first currency and outgoing amount of the second currency. Lastly subtract the amounts from the respective currencies and exit the function.

After successfully processing the sale, we will log the successful trade. The function will be talked about in R4C.

The break line is called as we will want to proceed to check another sale, and the sale is then pushed into allSales vector. At the end of all the loops, all sales with username of “simuser” will be present in the allSales vector.

We will update the eligibleOrders using setEligibleOrders() to update the eligibleOrders that have been fulfilled. We will then remove any unfulfilled orders which we be talked about in R2D.

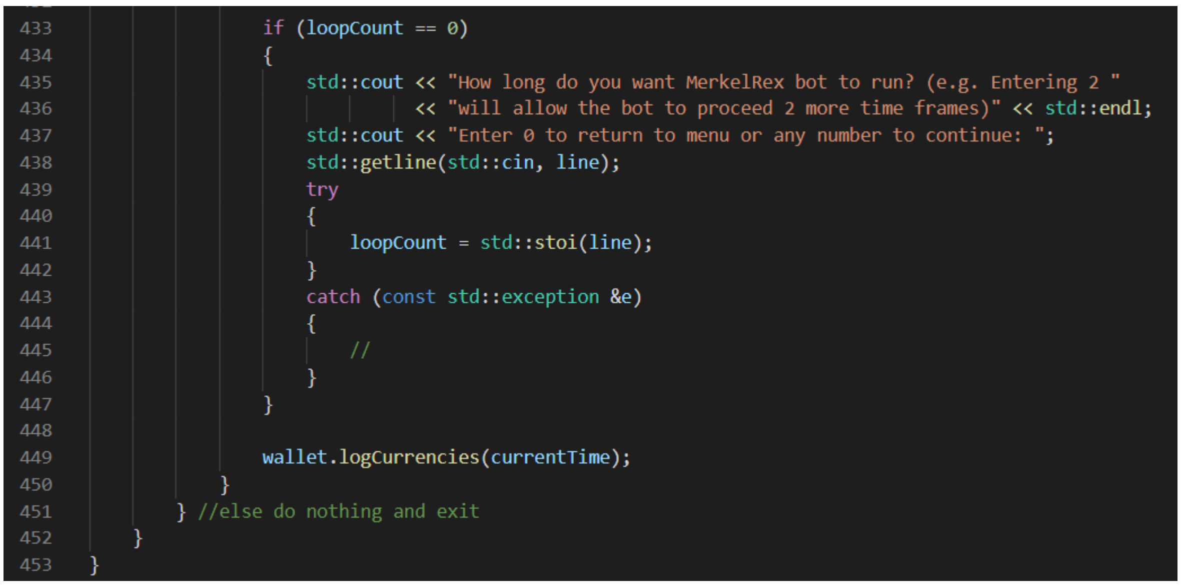
Before moving on to R2D, we will continue into the explanation of the final parts of runMerkelRexBot() function after the matchAndProcess() function.



```
// process sale
matchAndProcess();

std::cout << "Proceeding to next time frame.." << std::endl;
currentTime = orderBook.getNextTime(currentTime);

std::cout << "Replacing Data.." << std::endl;
if (orderBook.dataCount() < 14)
{
    orderBook.insertCurrentTimeData(currentTime);
}
else
{
    orderBook.insertCurrentTimeAndRemoveOldestTimeData(currentTime);
}
```

Figure 2.3A – MerkelMain.cpp (line 446 – 460)

After matching orders, we will proceed to the next time stamp and when doing so, we will perform the necessary action of inserting data and removing the oldest data if necessary.



```
                if (loopCount == 0)
                {
                    std::cout << "How long do you want MerkelRex bot to run? (e.g. Entering 2 "
                              << "will allow the bot to proceed 2 more time frames)" << std::endl;
                    std::cout << "Enter 0 to return to menu or any number to continue: ";
                    std::getline(std::cin, line);
                    try
                    {
                        loopCount = std::stoi(line);
                    }
                    catch (const std::exception &e)
                    {
                        //
                    }
                }

                wallet.logCurrencies(currentTime);
            }
        } //else do nothing and exit
    }
}
```

Figure 2.3D – MerkelMain.cpp (line 462 –480)

After replacing the data, we will check if the loopCount is 0. If true, we will ask user if they want to continue running the program with the option of entering the number of timestamps they want to proceed.

After getting the user input, we will attempt to convert the user input into an integer to be passed to the loopCount variable. We will catch any error so that the program does not crash if the user enters an invalid input.

Right before the end of the loop, we will log the currencies/assets which will be discussed in R4A.

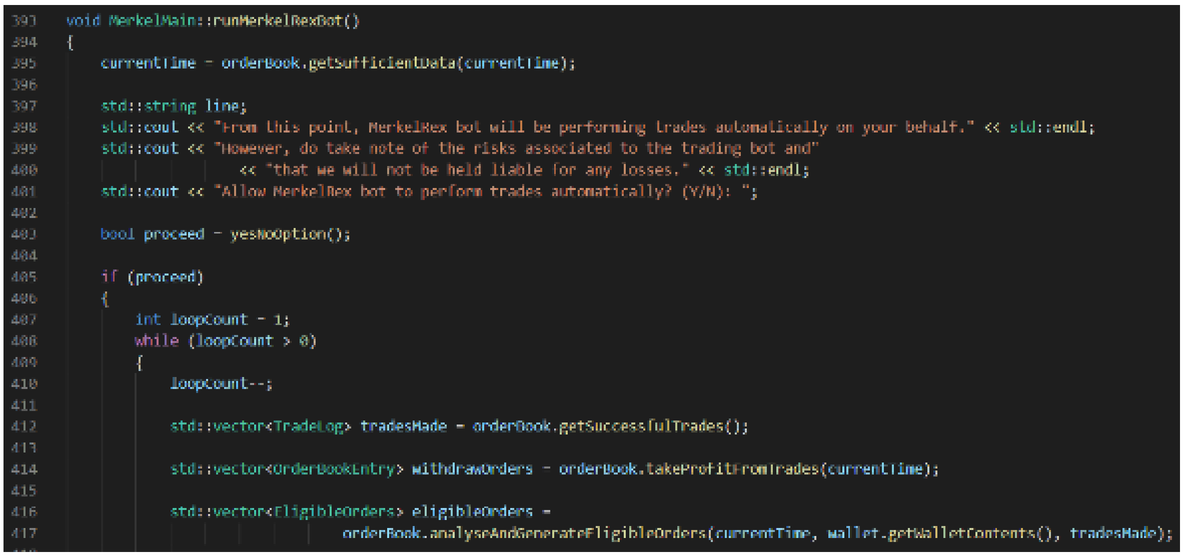
Finally, depending on the loopCount value that the user entered previously, the loop will run continuously for n times where n is the user input until the user is prompted to enter a value again. If the input is 0, the program will exit from the runMerkelRexBot() function and return to the main menu.

(R2C Achieved)

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

I am unsure of whether withdrawal simply means removing the unfulfilled orders from the orderBook and thus, I will attempt to do both taking profits and removal of order from orderBook for R2D and R3D.

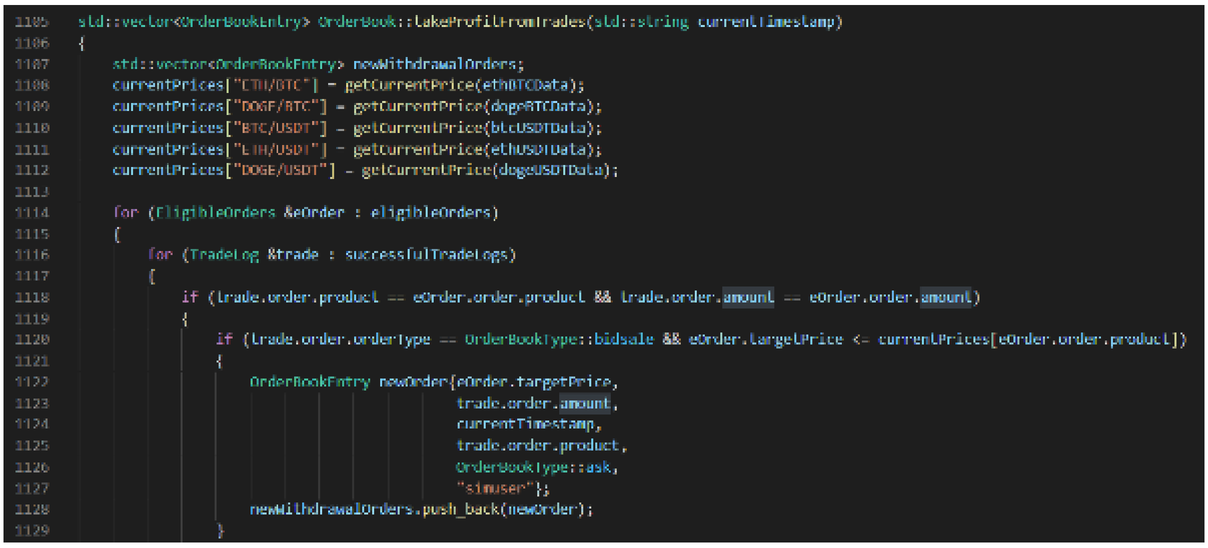
Withdrawal (taking profits):



```cpp
void MerkelMain::runMerkelRexBot()
{
    currentTime = orderBook.getSufficientData(currentTime);

    std::string line;
    std::cout << "From this point, MerkelRex bot will be performing trades automatically on your behalf." << std::endl;
    std::cout << "However, do take note of the risks associated to the trading bot and"
                << "that we will not be held liable for any losses." << std::endl;
    std::cout << "Allow MerkelRex bot to perform trades automatically? (Y/N): ";

    bool proceed = yesNoOption();

    if (proceed)
    {
        int loopCount = 1;
        while (loopCount > 0)
        {
            loopCount--;

            std::vector<TradeLog> tradesMade = orderBook.getSuccessfulTrades();

            std::vector<OrderBookEntry> withdrawOrders = orderBook.takeProfitFromTrades(currentTime);

            std::vector<EligibleOrders> eligibleOrders =
                          orderBook.analyseAndGenerateEligibleOrders(currentTime, wallet.getWalletContents(), tradesMade);
```

Figure 1.1F – MerkelMain.cpp (line 393 – 417)

As gone through previously in R1, we will get the successful trades that were made previously and we will create a withdrawOrders vector, taking in the return of the takeProfitFromTrades() function of current time stamp.



```cpp
std::vector<OrderBookEntry> OrderBook::takeProfitFromTrades(std::string currentTimestamp)
{
    std::vector<OrderBookEntry> newWithdrawalOrders;
    currentPrices["ETH/BTC"] = getCurrentPrice(ethBTCData);
    currentPrices["DOGE/BTC"] = getCurrentPrice(dogeBTCData);
    currentPrices["BTC/USDT"] = getCurrentPrice(btcUSDTData);
    currentPrices["ETH/USDT"] = getCurrentPrice(ethUSDTData);
    currentPrices["DOGE/USDT"] = getCurrentPrice(dogeUSDTData);

    for (EligibleOrders &eOrder : eligibleOrders)
    {
        for (TradeLog &trade : successfulTradeLogs)
        {
            if (trade.order.product == eOrder.order.product && trade.order.amount == eOrder.order.amount)
            {
                if (trade.order.orderType == OrderBookType::bidsale && eOrder.targetPrice <= currentPrices[eOrder.order.product])
                {
                    OrderBookEntry newOrder{eOrder.targetPrice,
                                            trade.order.amount,
                                            currentTimestamp,
                                            trade.order.product,
                                            OrderBookType::ask,
                                            "simuser"};
                    newWithdrawalOrders.push_back(newOrder);
                }
```

Figure 2.4A – OrderBook.cpp (line 1105 – 1130)

In the takeProfitFromTrades() function, we will create a OrderBookEntry vector to hold all new withdrawal orders and we will update the current prices of all the products with the current data.

Next, we will iterate through eligibleOrders followed by successfulTradeLogs which is a TradeLog vector that holds all orders that have been made by the bot successfully.

Firstly, we want to find the eligible order and successful trade that matches by checking if the product and amount matches. If true, we will check if successful trade has the order type of bidsale and that the target price is higher than the current price of the product. If true, we will create an order with the target price and the amount of the trade order. As a bid order was made previously, we will have to make an ask to exchange back to the original currency.

The remainder of the code checks the successful trade type of asksale which will be discussed in R3D.

Improvement:

1)

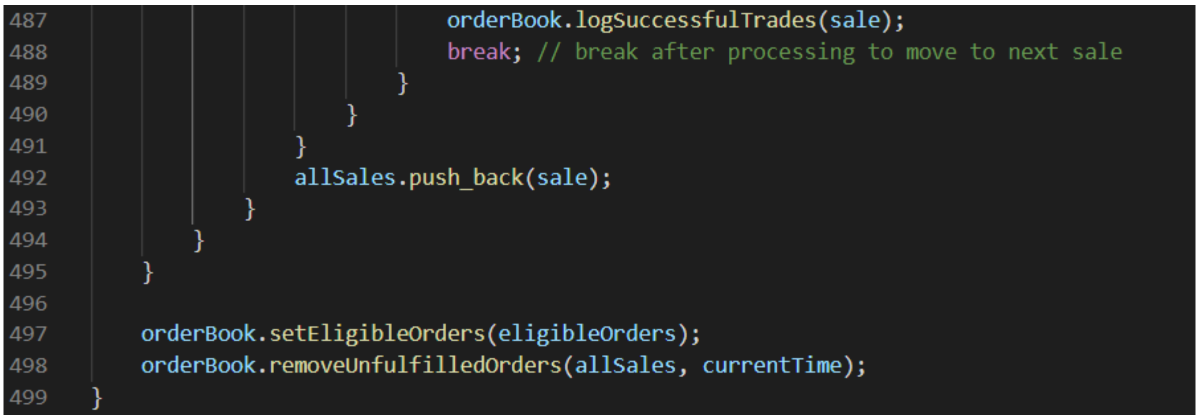
As the eligible order and successful trade is matched only by checking if the product and amount matches, there may be successful orders with same product and amount but different prices that match.

As I am unable to use the order prices to match due to the fact that the eligible bid order price may be different from the successful trade price as the matching engine matches ask price, an improvement is that I could create an id or index for each eligible order so that they can be matched to the correct orders by adding a int variable to the class and inserting it whenever I create order.

2)

As the price of the withdrawal order is a predicted target price, it may not always match an order. An improvement could be made by checking through the orders to check if the highest bid for the product so that we can make an ask order that is able to match that bid order if it is higher than the target price.

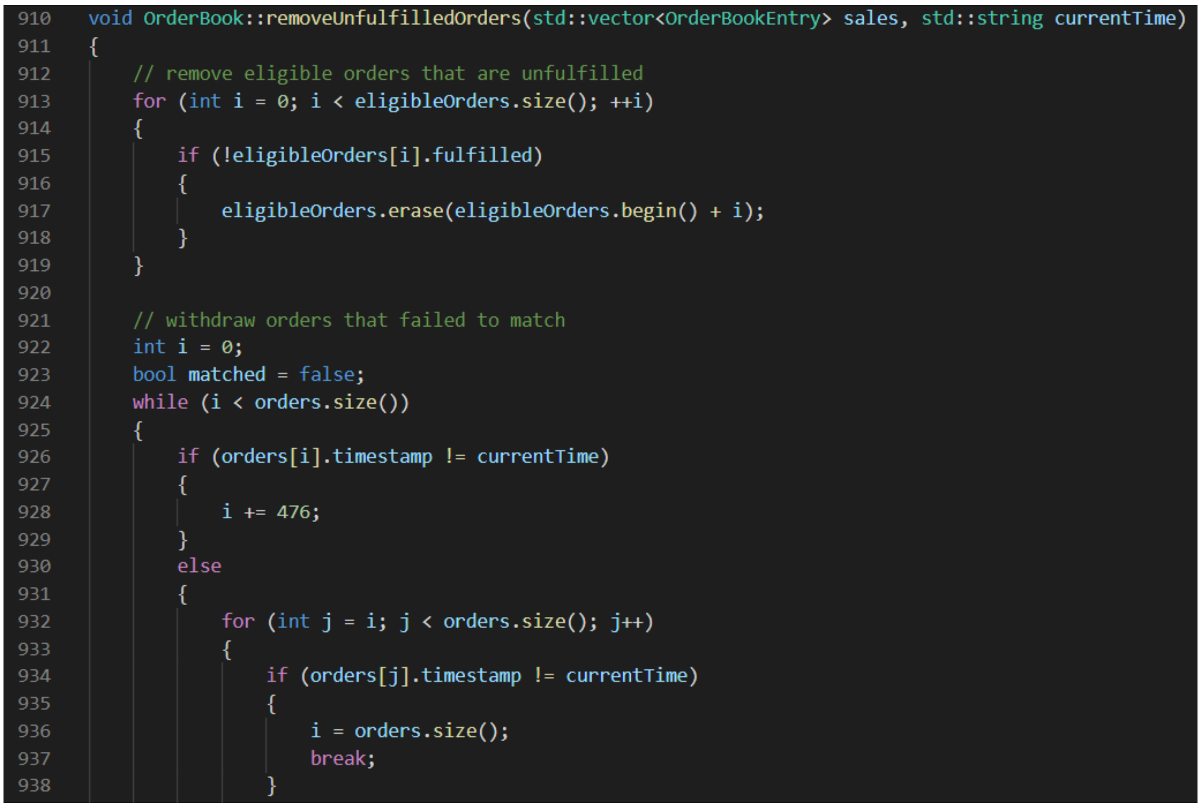
Withdrawal (remove orders from orderBook):



```cpp
                            orderBook.logSuccessfulTrades(sale);
                            break; // break after processing to move to next sale
                        }
                    }
                }
                allSales.push_back(sale);
            }
        }
    }

    orderBook.setEligibleOrders(eligibleOrders);
    orderBook.removeUnfulfilledOrders(allSales, currentTime);
}
```

Figure 2.4B – MerkelMain.cpp (line 484 – 495)

Orders that are not fulfilled in orderBook are removed by calling the removeUnfulfilledOrders() in OrderBook class. This function is called only in matchAndProcess() function of the MerkelMain class after all sale orders are processed and eligible orders that are fulfilled are updated.



```cpp
void OrderBook::removeUnfulfilledOrders(std::vector<OrderBookEntry> sales, std::string currentTime)
{
    // remove eligible orders that are unfulfilled
    for (int i = 0; i < eligibleOrders.size(); ++i)
    {
        if (!eligibleOrders[i].fulfilled)
        {
            eligibleOrders.erase(eligibleOrders.begin() + i);
        }
    }

    // withdraw orders that failed to match
    int i = 0;
    bool matched = false;
    while (i < orders.size())
    {
        if (orders[i].timestamp != currentTime)
        {
            i += 476;
        }
        else
        {
            for (int j = i; j < orders.size(); j++)
            {
                if (orders[j].timestamp != currentTime)
                {
                    i = orders.size();
                    break;
                }
```

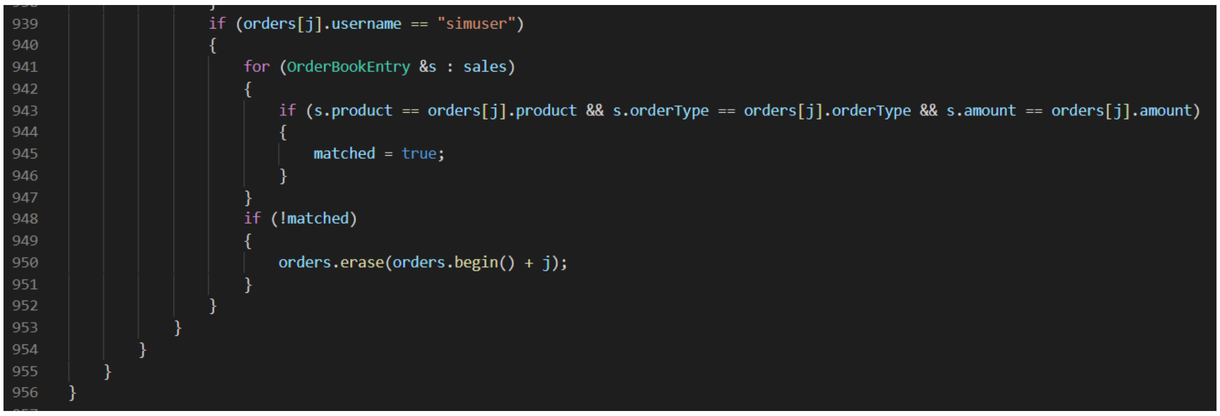
Figure 2.4C – OrderBook.cpp (line 910 – 938)

In the removeUnfulfilledOrders() function, we will iterate through eligibleOrders and remove any unfulfilled orders.

Next, we will iterate through the orders to find orders made by simuser and if it does not match any of the sales, we will remove them.

We first check if the timestamp of the order of that iteration is not equal to the current time. If true, we will increment i by 476 because the data has 476 data in each timestamp(100 for all products except DOGE/BTC which has 76).

If the timestamp matches, we will iterate through the orders with index j that starts with at the index of i. At any point of time where the orders timestamp of index j is not equal to currentTime, we will let index i be equal to the size of the orderBook and break from the loop. This will result in the program exiting both for and while loop.



```
939                 if (orders[j].username == "simuser")
940                 {
941                     for (OrderBookEntry &s : sales)
942                     {
943                         if (s.product == orders[j].product && s.orderType == orders[j].orderType && s.amount == orders[j].amount)
944                         {
945                             matched = true;
946                         }
947                     }
948                     if (!matched)
949                     {
950                         orders.erase(orders.begin() + j);
951                     }
952                 }
953             }
954         }
955     }
956 }
```

Figure 2.4D – OrderBook.cpp (line 939 – 956)

if the timestamp matches the currentTime, and if the current order username is “simuser”, we will iterate through all sales to check if any of the sales matches the order. We will check if product, amount and orderType matches but not the price as a high bid price may match but the sale’s price will be equal to the matched ask.

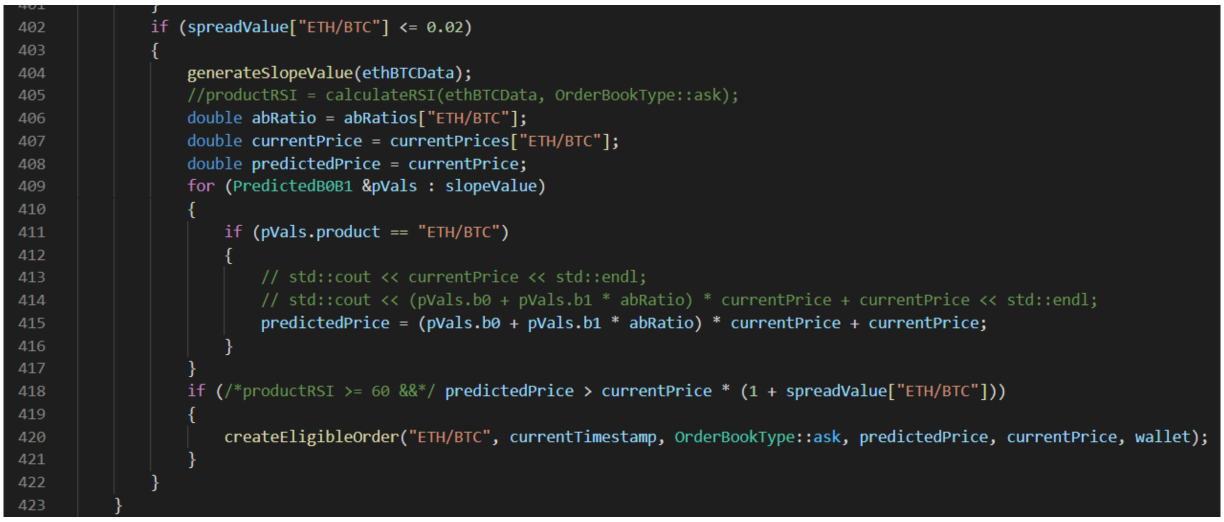
If matched, we will let matched be true. At the end of the sales for loop, we will check if matched is false. If matched variable is false, we will erase the order of j index as it means that no sale was made with that order.

This function runs every timestamp after the matchAndProcess() function and any orders that does not match are removed.

(R2D Achieved)

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



```cpp
        if (spreadValue["ETH/BTC"] <= 0.02)
        {
            generateSlopeValue(ethBTCData);
            //productRSI = calculateRSI(ethBTCData, OrderBookType::ask);
            double abRatio = abRatios["ETH/BTC"];
            double currentPrice = currentPrices["ETH/BTC"];
            double predictedPrice = currentPrice;
            for (PredictedB0B1 &pVals : slopeValue)
            {
                if (pVals.product == "ETH/BTC")
                {
                    // std::cout << currentPrice << std::endl;
                    // std::cout << (pVals.b0 + pVals.b1 * abRatio) * currentPrice + currentPrice << std::endl;
                    predictedPrice = (pVals.b0 + pVals.b1 * abRatio) * currentPrice + currentPrice;
                }
            }
            if (/*productRSI >= 60 &&*/ predictedPrice > currentPrice * (1 + spreadValue["ETH/BTC"]))
            {
                createEligibleOrder("ETH/BTC", currentTimestamp, OrderBookType::ask, predictedPrice, currentPrice, wallet);
            }
        }
    }
```

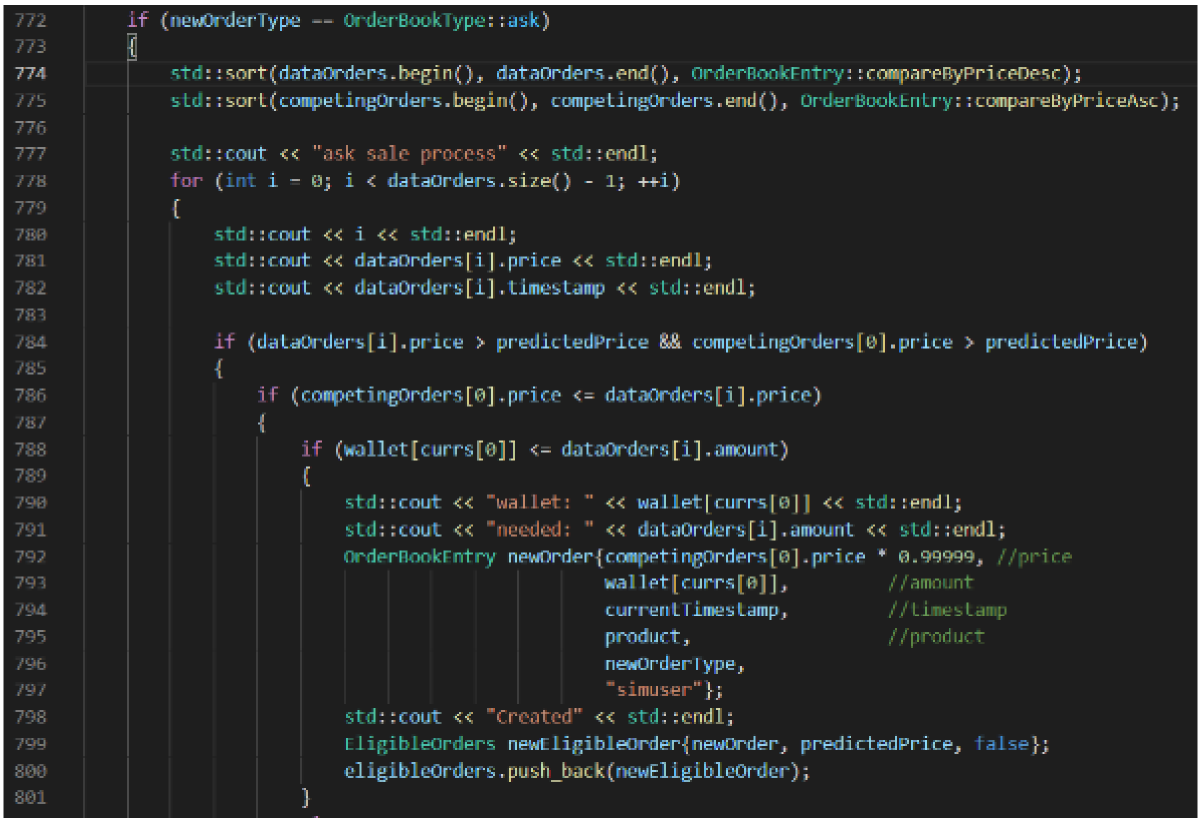
Figure 3.1A – OrderBook.cpp (line 393 – 413)

As the procedure and steps are very similar to the description in R2A, I will be skipping them and will only talk about the differences between them.

Similarly, the current prices of each product would have already been calculated and stored inside the currentPrices map in the OrderBook class, while the ab ratios will be calculated at the start of the analyseAndGenerateEligibleOrders() function.

Predicted price will still be calculated with the same formula, but the subsequent if check will be different. We will check if the predicted price is higher than the current price of the product + the cost or spread of the currency.

This means that if we predict that the price will rise, we should create a bid order to exchange to the currency on the left of the product, else if we predict that the price will fall, we should create an ask order to exchange to the currency on the right of the product.



```cpp
    if (newOrderType == OrderBookType::ask)
    {
        std::sort(dataOrders.begin(), dataOrders.end(), OrderBookEntry::compareByPriceDesc);
        std::sort(competingOrders.begin(), competingOrders.end(), OrderBookEntry::compareByPriceAsc);

        std::cout << "ask sale process" << std::endl;
        for (int i = 0; i < dataOrders.size() - 1; ++i)
        {
            std::cout << i << std::endl;
            std::cout << dataOrders[i].price << std::endl;
            std::cout << dataOrders[i].timestamp << std::endl;

            if (dataOrders[i].price > predictedPrice && competingOrders[0].price > predictedPrice)
            {
                if (competingOrders[0].price <= dataOrders[i].price)
                {
                    if (wallet[curr s[0]] <= dataOrders[i].amount)
                    {
                        std::cout << "wallet: " << wallet[currs[0]] << std::endl;
                        std::cout << "needed: " << dataOrders[i].amount << std::endl;
                        OrderBookEntry newOrder{competingOrders[0].price * 0.99999, //price
                                                wallet[currs[0]],         //amount
                                                currentTimestamp,         //timestamp
                                                product,                  //product
                                                newOrderType,
                                                "simuser"};
                        std::cout << "Created" << std::endl;
                        EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
                        eligibleOrders.push_back(newEligibleOrder);
                    }
```

Figure 3.1B – OrderBook.cpp (line 772 – 800)

In the createEligibleOrders() function, we start by checking the orderType passed in and updating the newOrderType with the opposite OrderBookType of orderType variable. We then proceed to get the respective order types or dataOrders vector and competingOrders vector.

Similar to R2A, the sorting of dataOrders and competingOrders will be done after the if statement to check the newOrderType. In a bid type, we will want to sort the dataOrders in ascending and competingOrders in descending order, but in an ask order, we want to sort dataOrders in descending order while the competingOrders in ascending order.

The reason is that when making an ask order, we want to check from the highest to lowest bid, which translate to the dataOrders being sorted in descending order while we want to check from the lowest competing asks orders which translate to the competingOrders being sorted in ascending order.

After sorting, we will iterate through the dataOrders to check through each order.

The first if statement checks if the price of the bid order from dataOrders is higher than the predicted price and the lowest competing ask order is higher than the predicted price.

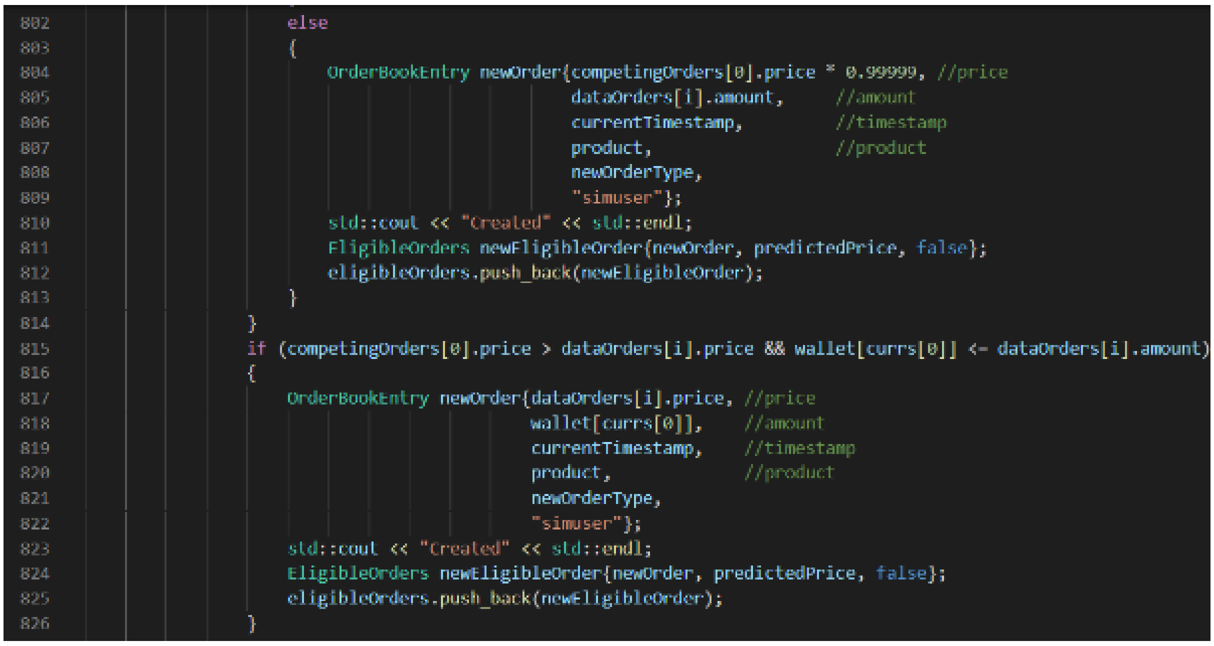
This if statement checks for a scenario where both the bidding price and competing ask price is above the predicted price.

The next if statement checks if the lowest competing ask price is lesser than or equal to bid price. We want to check if there will be any matches occurring in the order book so that we are able to compete.

The following if statement checks if the currency in the wallet is less than or equal to the order amount. If true, we will have to create an ask that will partially fulfil the ask order.

We will use the competing price and multiply it by 0.99999 which equates to subtracting 0.001% of the price so that our ask will be lower than the lowest ask price without giving up too much of our profits. The amount will match whatever currency that is left inside the wallet.

We will then push the eligible order into the eligibleOrders vector.

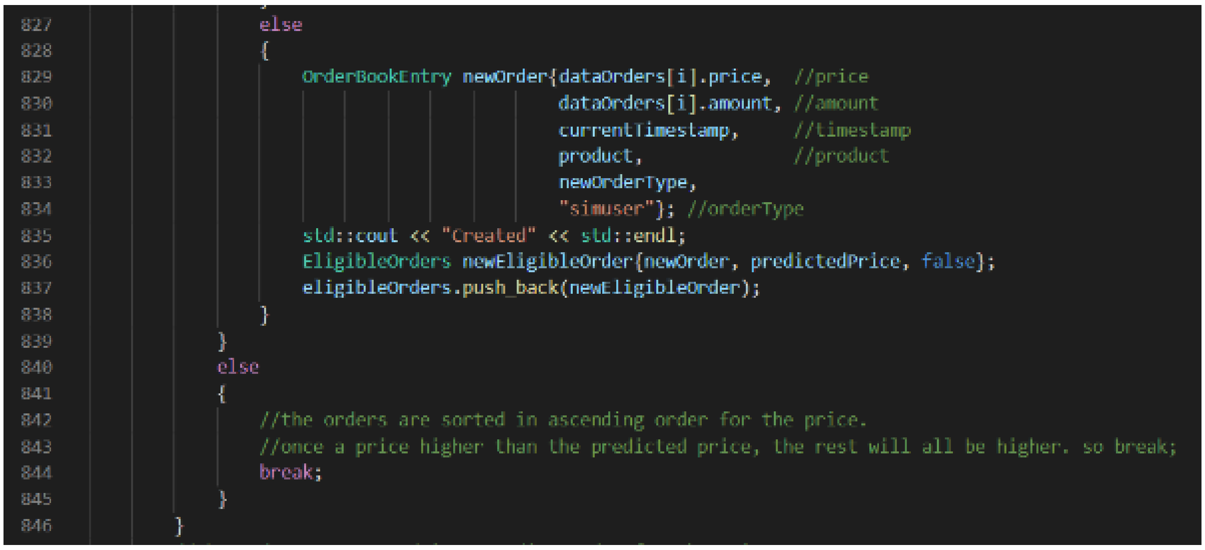


```
                    else
                    {
                        OrderBookEntry newOrder{competingOrders[0].price * 0.99999, //price
                                                dataOrders[i].amount,      //amount
                                                currentTimestamp,          //timestamp
                                                product,                   //product
                                                newOrderType,
                                                "simuser"};
                        std::cout << "Created" << std::endl;
                        EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
                        eligibleOrders.push_back(newEligibleOrder);
                    }
                }
                if (competingOrders[0].price > dataOrders[i].price && wallet[currs[0]] <= dataOrders[i].amount)
                {
                    OrderBookEntry newOrder{dataOrders[i].price, //price
                                            wallet[currs[0]],    //amount
                                            currentTimestamp,    //timestamp
                                            product,             //product
                                            newOrderType,
                                            "simuser"};
                    std::cout << "Created" << std::endl;
                    EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
                    eligibleOrders.push_back(newEligibleOrder);
                }
```

Figure 3.1C – OrderBook.cpp (line 802 – 826)

If currency in wallet is not more than the order amount, the else statement in figure 3.1C will run. We will create an OrderBookEntry of with 0.001% lower price than the lowest competing price and match the order amount.

The next if statement checks if the competing ask price is higher than the order price and that the currency is lower than or equal to the order amount. If both are true, we will create a new eligible order matching the bid price as there are no competing asks that have a lower ask and the amount will match the amount of currency available in the wallet.



```
        else
        {
            OrderBookEntry newOrder{dataOrders[i].price,  //price
                                    dataOrders[i].amount, //amount
                                    currentTimestamp,     //timestamp
                                    product,              //product
                                    newOrderType,
                                    "simuser"}; //orderType
            std::cout << "Created" << std::endl;
            EligibleOrders newEligibleOrder{newOrder, predictedPrice, false};
            eligibleOrders.push_back(newEligibleOrder);
        }
    }
    else
    {
        //the orders are sorted in ascending order for the price.
        //once a price higher than the predicted price, the rest will all be higher. so break;
        break;
    }
}
```

Figure 3.1D – OrderBook.cpp (line 827 – 846)

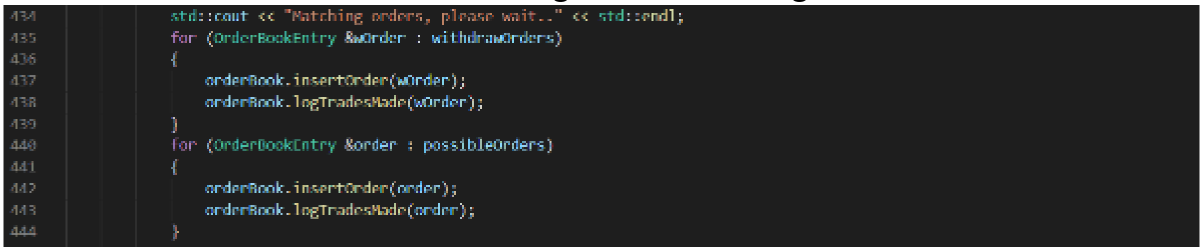
If the lowest competing ask price is higher than the orders prices but the currency in the wallet has more than sufficient to complete the order, we will create a new eligible order matching the price and amount of the order in dataOrders.

The final else() is linked to the if statement that checks if dataOrder and competingOrder prices are higher than the predictedPrice. As the data orders are sorted in descending order, if the order price at index i is lower than the predicted price, the order price at index greater than i, the if statement will always be false and thus, we will break from the loop.

The program will proceed to runMerkelRexBot() in MerkelMain class and check through the eligible orders and insert orders that can be fulfilled into possibleOrders. The possible orders will be the final orders that will be passed into the orderBook for matching.

(R3A Achieved)

R3B: Pass the bids to the exchange for matching



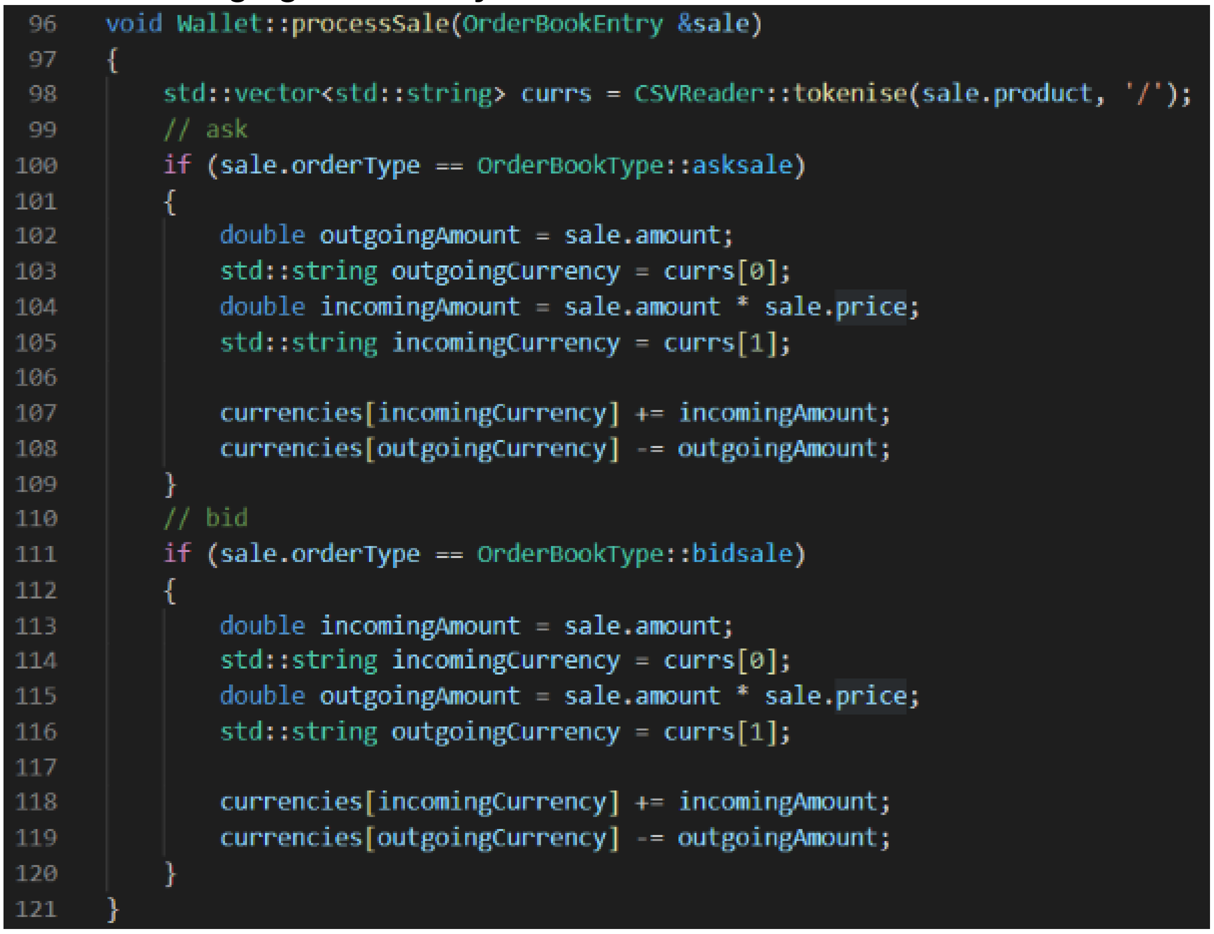
```cpp
        std::cout << "Matching orders, please wait.." << std::endl;
        for (OrderBookEntry &wOrder : withdrawOrders)
        {
            orderBook.insertOrder(wOrder);
            orderBook.logTradesMade(wOrder);
        }
        for (OrderBookEntry &order : possibleOrders)
        {
            orderBook.insertOrder(order);
            orderBook.logTradesMade(order);
        }
```

Figure 2.2A – MerkelMain.cpp (line 434 – 444)

As explained in R2B, the possible orders will be inserted into the orderBook.

(R3B Achieved)

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



```cpp
void Wallet::processSale(OrderBookEntry &sale)
{
    std::vector<std::string> currs = CSVReader::tokenise(sale.product, '/');
    // ask
    if (sale.orderType == OrderBookType::asksale)
    {
        double outgoingAmount = sale.amount;
        std::string outgoingCurrency = currs[0];
        double incomingAmount = sale.amount * sale.price;
        std::string incomingCurrency = currs[1];

        currencies[incomingCurrency] += incomingAmount;
        currencies[outgoingCurrency] -= outgoingAmount;
    }
    // bid
    if (sale.orderType == OrderBookType::bidsale)
    {
        double incomingAmount = sale.amount;
        std::string incomingCurrency = currs[0];
        double outgoingAmount = sale.amount * sale.price;
        std::string outgoingCurrency = currs[1];

        currencies[incomingCurrency] += incomingAmount;
        currencies[outgoingCurrency] -= outgoingAmount;
    }
}
```

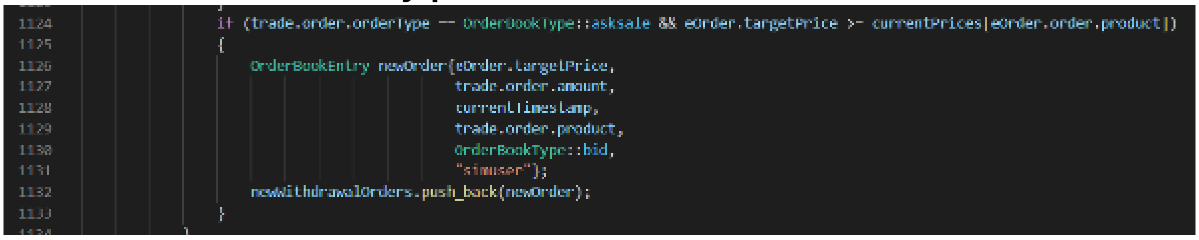
Figure 2.3C – Wallet.cpp (line 96 – 121)

Codes for R3C are the same as previously described in R2C. The difference is that the processSale() function will perform the calculations differently.

If the created sale is an asksale, we will get the incoming amount of the second currency and outgoing amount of the first currency. Lastly subtract the amounts from the respective currencies and exit the function.

(R3C Achieved)

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



```
1124        if (trade.order.orderType == OrderBookType::asksale && eOrder.targetPrice >= currentPrices[eOrder.order.product])
1125        {
1126            OrderBookEntry newOrder{eOrder.targetPrice,
1127                                    trade.order.amount,
1128                                    currentTimestamp,
1129                                    trade.order.product,
1130                                    OrderBookType::bid,
1131                                    "simuser"};
1132            newWithdrawalOrders.push_back(newOrder);
1133        }
```

Figure 3.4A – OrderBook.cpp (line 1124 – 1133)

The takeProfitFromTrades() function will decide if trades should be withdrawn as explained in R2D. However, we will be checking if the successful trade is an asksale OrderBookType and if the target price is greater than or equal to the current price of the product.

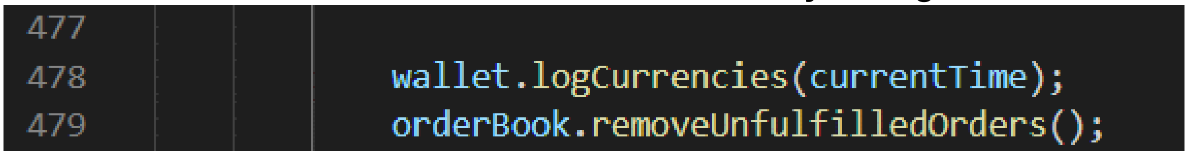
If true, we will create an order with the target price and the amount of the trade order. As an ask order was made previously, we will have to make a bid to exchange back to the original currency.

Removal of unsuccessful orders from the orderBook will be the same for both bids and asks.

(R3D Achieved)

R4: Logging

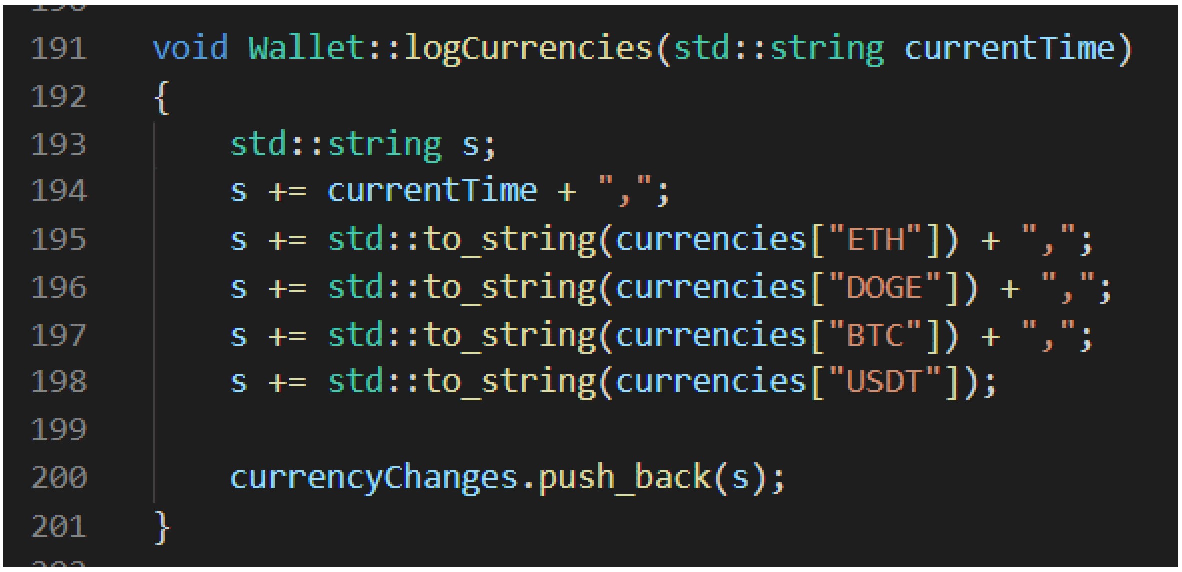
R4A: Maintain a record of its assets and how they change over time



```cpp

            wallet.logCurrencies(currentTime);
            orderBook.removeUnfulfilledOrders();
```

Figure 4.1A – MerkelMain.cpp (line 477 – 479)

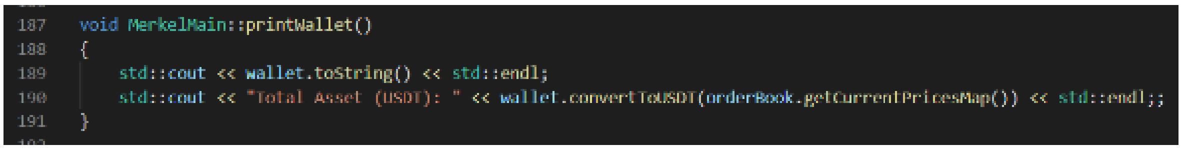
At the end of every time stamp, the logCurrencies() function in Wallet class will be called to keep a record of all currencies on hand at the end of each time stamp.



```cpp
void Wallet::logCurrencies(std::string currentTime)
{
    std::string s;
    s += currentTime + ",";
    s += std::to_string(currencies["ETH"]) + ",";
    s += std::to_string(currencies["DOGE"]) + ",";
    s += std::to_string(currencies["BTC"]) + ",";
    s += std::to_string(currencies["USDT"]);

    currencyChanges.push_back(s);
}
```

Figure 4.1B – Wallet.cpp (line 191 – 201)

In the logCurrencies() function, string s will hold the current time, amount of ETH, amount of DOGE, amount of BTC and amount of USDT in this order, separated by comma. The string is pushed into the string vector of currencyChanges.



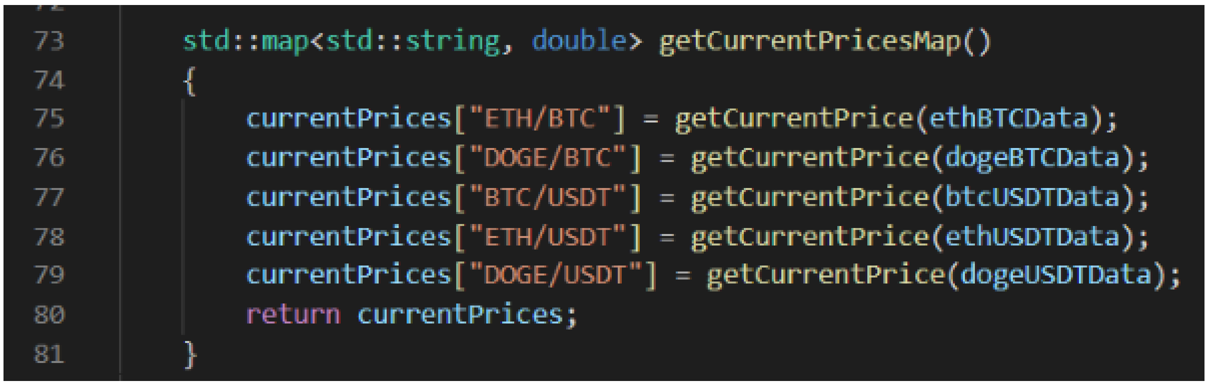
```cpp
void MerkelMain::printWallet()
{
    std::cout << wallet.toString() << std::endl;
    std::cout << "Total Asset (USDT): " << wallet.convertToUSDT(orderBook.getCurrentPricesMap()) << std::endl;;
}
```

Figure 4.1C – MerkelMain.cpp (line 187 – 191)

I have modified the printWallet() function to print out the estimated value of all currencies in USDT.

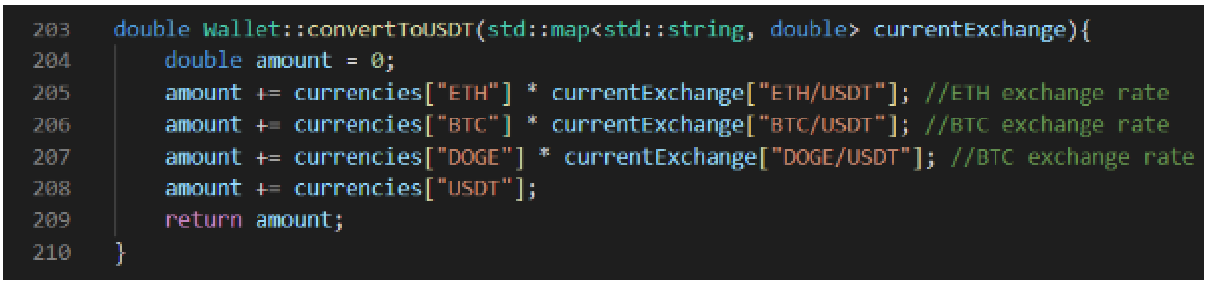
getCurrentPricesMap() retrieves a map of <string, double> consisting of the current price as double and the product as string in OrderBook.h file.

The map is then passed to convertToUSDT() function in Wallet class.



```cpp
    std::map<std::string, double> getCurrentPricesMap()
    {
        currentPrices["ETH/BTC"] = getCurrentPrice(ethBTCData);
        currentPrices["DOGE/BTC"] = getCurrentPrice(dogeBTCData);
        currentPrices["BTC/USDT"] = getCurrentPrice(btcUSDTData);
        currentPrices["ETH/USDT"] = getCurrentPrice(ethUSDTData);
        currentPrices["DOGE/USDT"] = getCurrentPrice(dogeUSDTData);
        return currentPrices;
    }
```

Figure 4.1D – OrderBook.h (line 73 – 81)

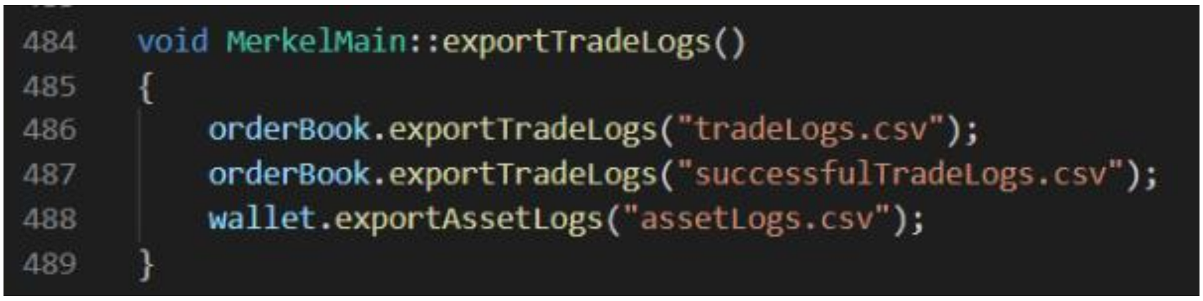


```cpp
double Wallet::convertToUSDT(std::map<std::string, double> currentExchange){
    double amount = 0;
    amount += currencies["ETH"] * currentExchange["ETH/USDT"]; //ETH exchange rate
    amount += currencies["BTC"] * currentExchange["BTC/USDT"]; //BTC exchange rate
    amount += currencies["DOGE"] * currentExchange["DOGE/USDT"]; //BTC exchange rate
    amount += currencies["USDT"];
    return amount;
}
```

Figure 4.1E – Wallet.cpp (line 203 – 210)

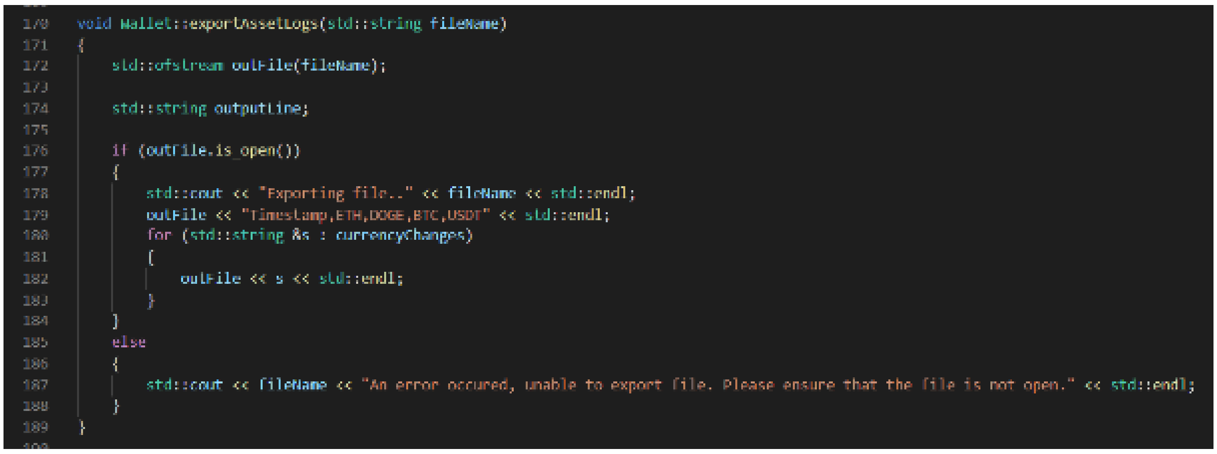
The total amount in USDT is then calculated and returned.

All of the currency information can be exported by entering option 3 in MerkelMenu which will run the exportTradeLogs() function to log all information into a csv file.



```cpp
void MerkelMain::exportTradeLogs()
{
    orderBook.exportTradeLogs("tradeLogs.csv");
    orderBook.exportTradeLogs("successfulTradeLogs.csv");
    wallet.exportAssetLogs("assetLogs.csv");
}
```

Figure 4.1F – MerkelMain.cpp (line 484 – 489)



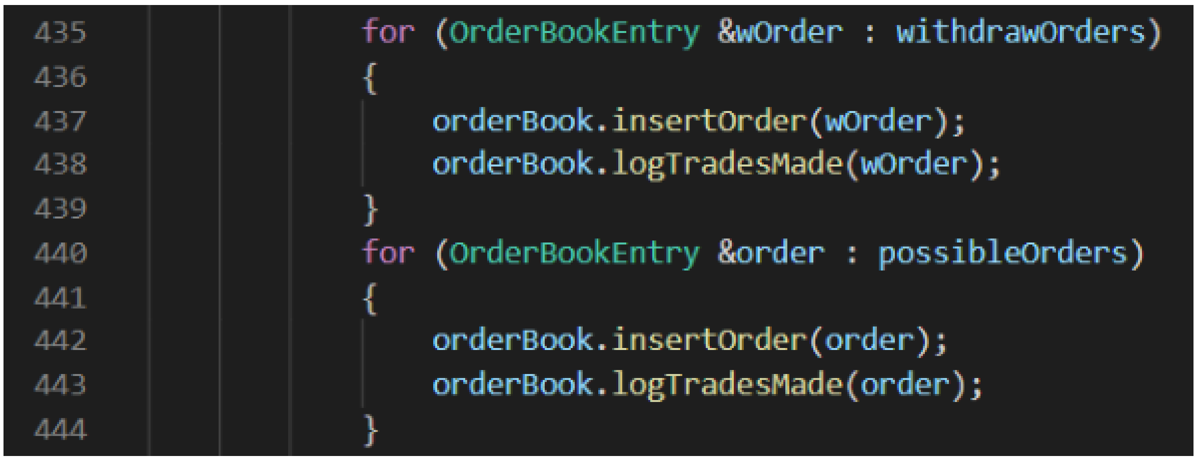
```cpp
void Wallet::exportAssetLogs(std::string fileName)
{
    std::ofstream outFile(fileName);

    std::string outputLine;

    if (outFile.is_open())
    {
        std::cout << "Exporting file.." << fileName << std::endl;
        outFile << "Timestamp,ETH,DOGE,BTC,USDT" << std::endl;
        for (std::string &s : currencyChanges)
        {
            outFile << s << std::endl;
        }
    }
    else
    {
        std::cout << fileName << "An error occured, unable to export file. Please ensure that the file is not open." << std::endl;
    }
}
```

Figure 4.1G – Wallet.cpp (line 170 – 189)

exportAssetLogs function in Wallet class prints all currencies into the assetLogs.csv.

(R4A Achieved)

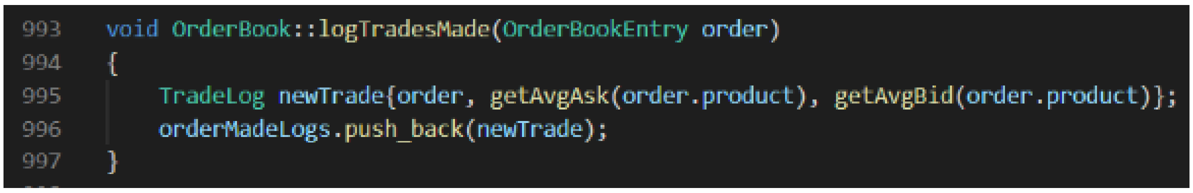
R4B: Maintain a record of the bids and offers it has made in a suitable file format



```
for (OrderBookEntry &wOrder : withdrawOrders)
{
    orderBook.insertOrder(wOrder);
    orderBook.logTradesMade(wOrder);
}
for (OrderBookEntry &order : possibleOrders)
{
    orderBook.insertOrder(order);
    orderBook.logTradesMade(order);
}
```

Figure 4.2A – MerkelMain.cpp (line 435 – 444)

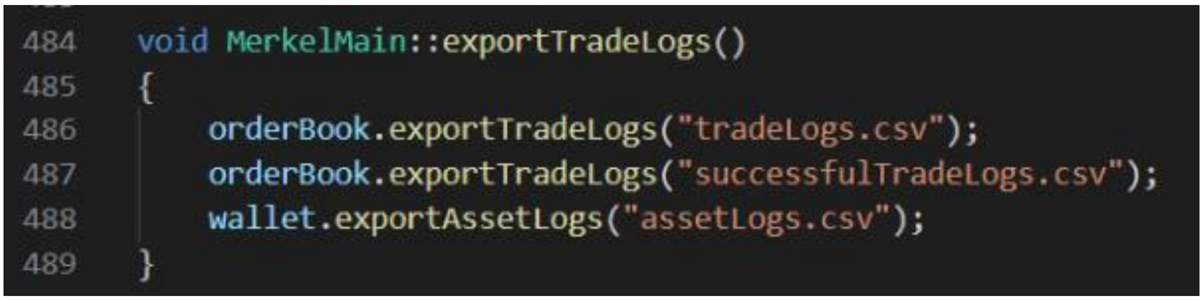
Whenever trades are created, the trade will be logged down by calling the logTradesMade() function.



```
void OrderBook::logTradesMade(OrderBookEntry order)
{
    TradeLog newTrade{order, getAvgAsk(order.product), getAvgBid(order.product)};
    orderMadeLogs.push_back(newTrade);
}
```

Figure 4.2B – OrderBook.cpp (line 993 – 997)

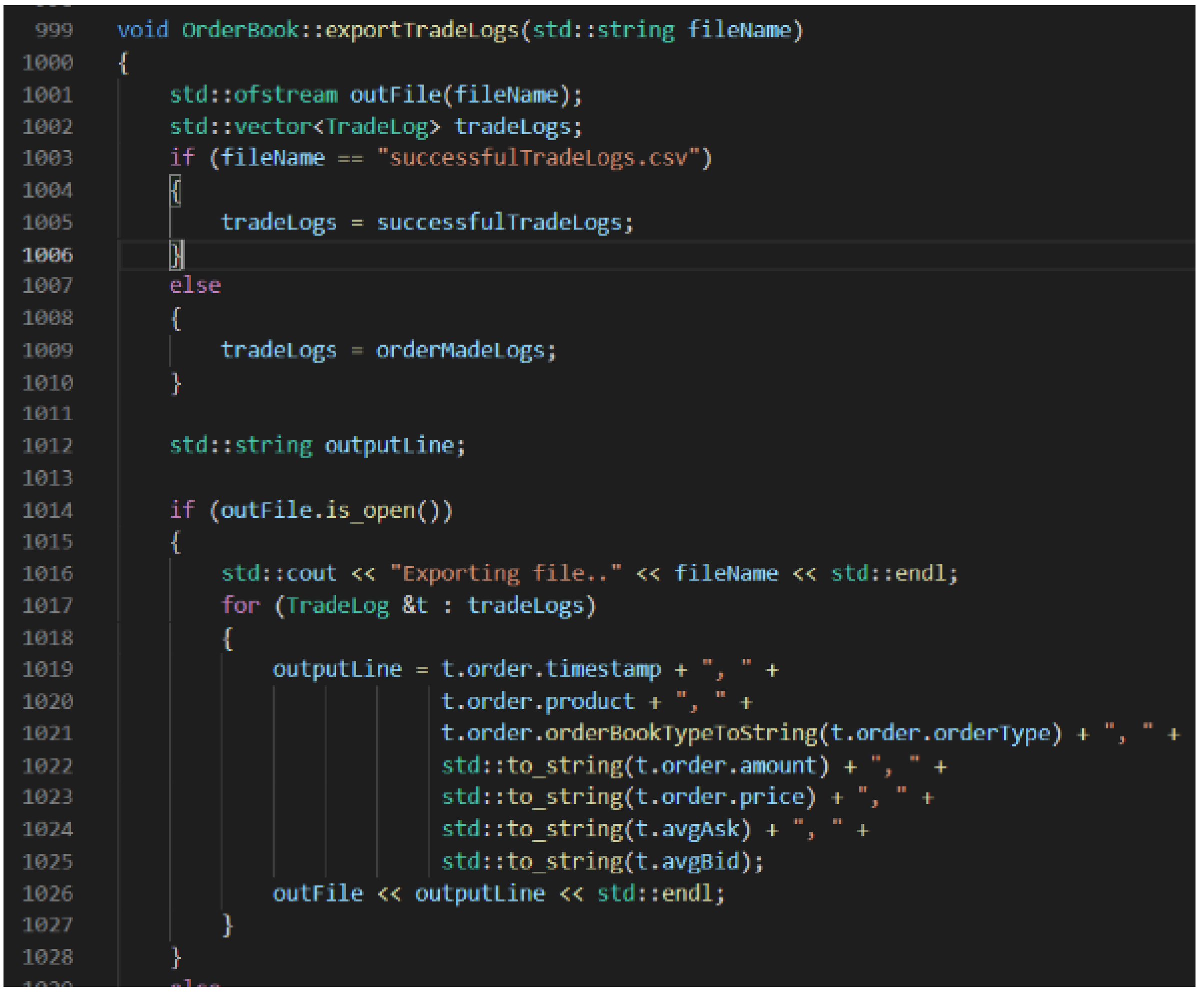
The trades are then pushed into the TradeLog vector that holds all orders that were ever made.



```
void MerkelMain::exportTradeLogs()
{
    orderBook.exportTradeLogs("tradeLogs.csv");
    orderBook.exportTradeLogs("successfulTradeLogs.csv");
    wallet.exportAssetLogs("assetLogs.csv");
}
```

Figure 4.1F – MerkelMain.cpp (line 484 – 489)

Whenever exportTradeLogs() in MerkelMain is called, the exportTradeLogs() function in OrderBook class is called.



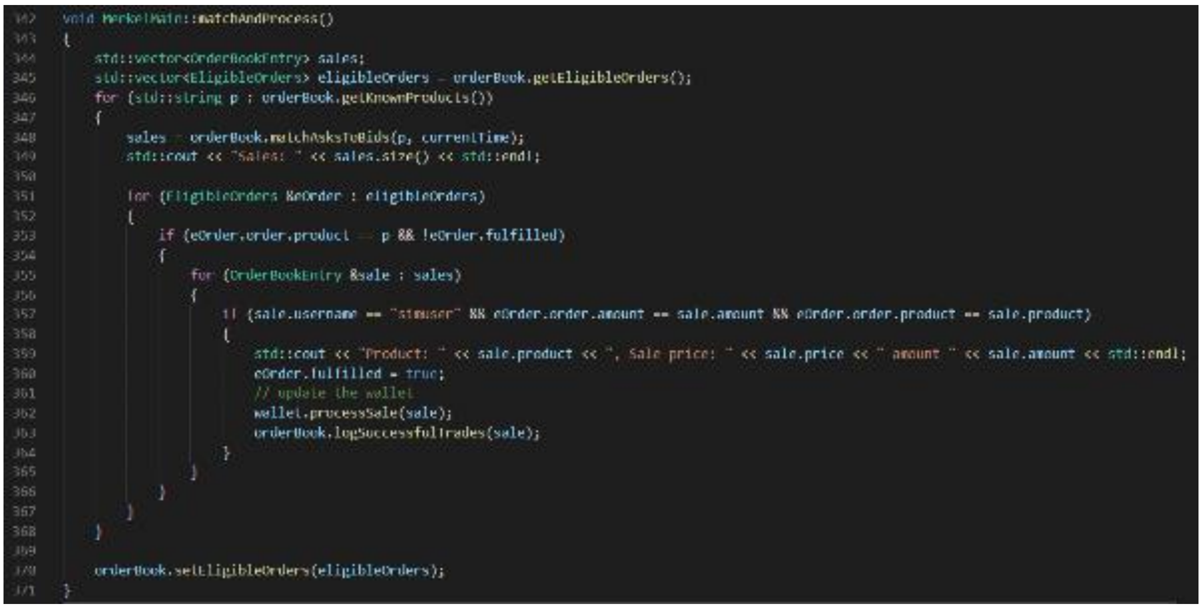
```
 999  void OrderBook::exportTradeLogs(std::string fileName)
1000  {
1001      std::ofstream outFile(fileName);
1002      std::vector<TradeLog> tradeLogs;
1003      if (fileName == "successfulTradeLogs.csv")
1004      {
1005          tradeLogs = successfulTradeLogs;
1006      }
1007      else
1008      {
1009          tradeLogs = orderMadeLogs;
1010      }
1011
1012      std::string outputLine;
1013
1014      if (outFile.is_open())
1015      {
1016          std::cout << "Exporting file.." << fileName << std::endl;
1017          for (TradeLog &t : tradeLogs)
1018          {
1019              outputLine = t.order.timestamp + ", " +
1020                           t.order.product + ", " +
1021                           t.order.orderBookTypeToString(t.order.orderType) + ", " +
1022                           std::to_string(t.order.amount) + ", " +
1023                           std::to_string(t.order.price) + ", " +
1024                           std::to_string(t.avgAsk) + ", " +
1025                           std::to_string(t.avgBid);
1026              outFile << outputLine << std::endl;
1027          }
1028      }
```

Figure 4.2C – OrderBook.cpp (line 999 – 1028)

In the exportTradeLogs() function in OrderBook class, the tradeLog vector to be retrieved is dependent on the file name passed into the function. As seen from line 1019 to 1025 of figure 4.2C, we will be storing the order information by the order of timestamp, product, order type, amount, price, average ask price and average bid price.

(R4B Achieved)

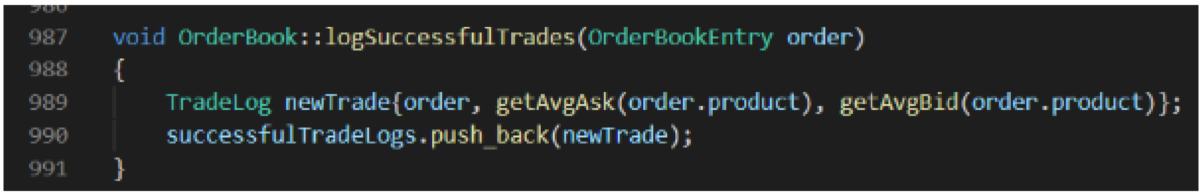
R4C: Maintain a record of successful bids and offers it has made, along with the context (e.g. exchange average offer and bid) in a suitable file format



```cpp
void MerkelMain::matchAndProcess()
{
    std::vector<OrderBookEntry> sales;
    std::vector<EligibleOrders> eligibleOrders = orderBook.getEligibleOrders();
    for (std::string p : orderBook.getKnownProducts())
    {
        sales = orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;

        for (EligibleOrders &eOrder : eligibleOrders)
        {
            if (eOrder.order.product == p && !eOrder.fulfilled)
            {
                for (OrderBookEntry &sale : sales)
                {
                    if (sale.username == "simuser" && eOrder.order.amount == sale.amount && eOrder.order.product == sale.product)
                    {
                        std::cout << "Product: " << sale.product << ", Sale price: " << sale.price << " amount " << sale.amount << std::endl;
                        eOrder.fulfilled = true;
                        // update the wallet
                        wallet.processSale(sale);
                        orderBook.logSuccessfulTrades(sale);
                    }
                }
            }
        }
    }

    orderBook.setEligibleOrders(eligibleOrders);
}
```

Figure 2.3B – MerkelMain.cpp (line 342 – 371)

As seen in figure 2.3B of part R2C, whenever a successful trade is made, the successful trade will be logged when the logSuccessfulTrade() function is called.



```cpp
void OrderBook::logSuccessfulTrades(OrderBookEntry order)
{
    TradeLog newTrade{order, getAvgAsk(order.product), getAvgBid(order.product)};
    successfulTradeLogs.push_back(newTrade);
}
```

Figure 4.3A – OrderBook.cpp (line 987 – 991)

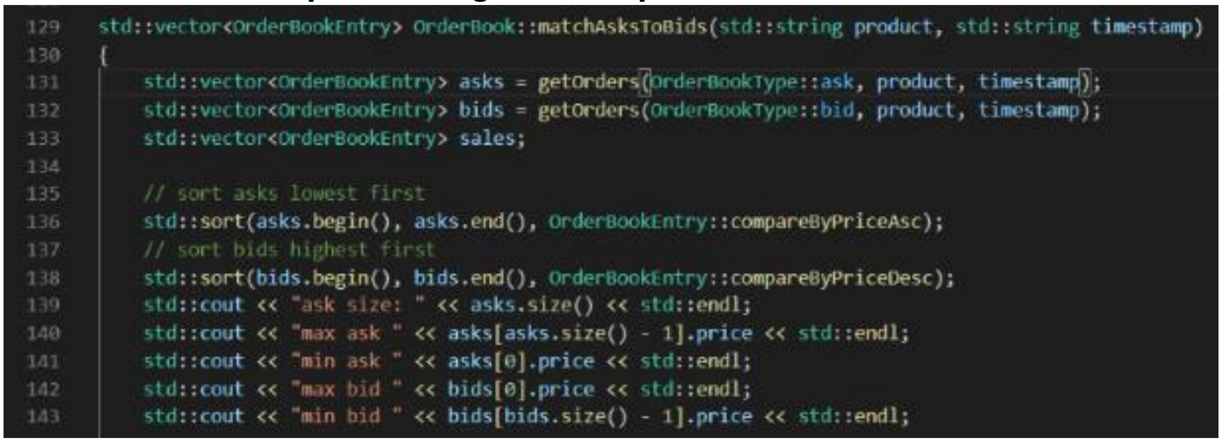
The sale will be logged with the relevant information of average ask and bid price included in the log.

Finally, as seen in figure 4.2C, successful trades will be logged into the csv file and each row represents the different orders made.

(R4C Achieved)

Challenge Requirement:

The order and bid processing before optimisation:



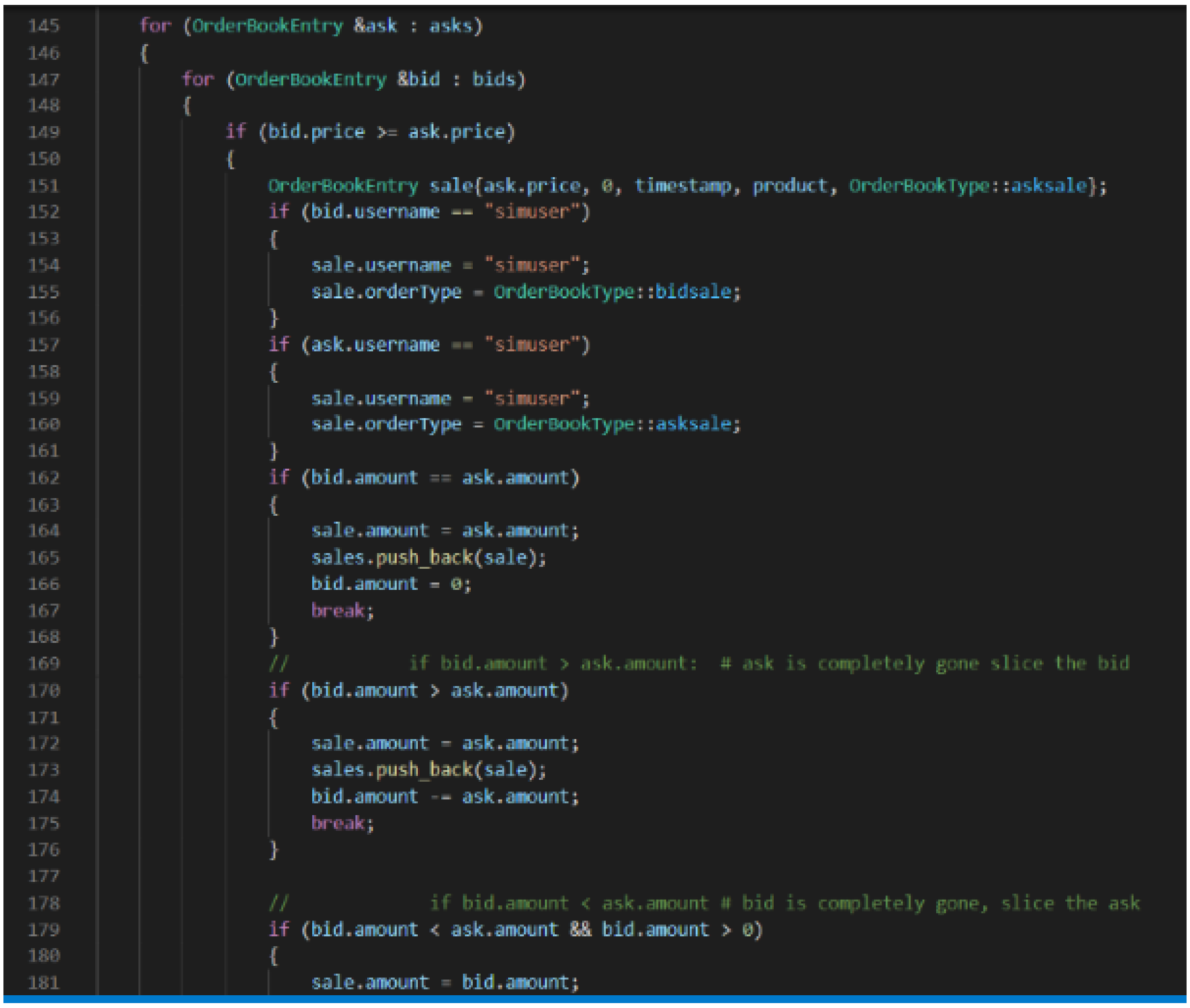
```
std::vector<OrderBookEntry> OrderBook::matchAsksToBids(std::string product, std::string timestamp)
{
    std::vector<OrderBookEntry> asks = getOrders(OrderBookType::ask, product, timestamp);
    std::vector<OrderBookEntry> bids = getOrders(OrderBookType::bid, product, timestamp);
    std::vector<OrderBookEntry> sales;

    // sort asks lowest first
    std::sort(asks.begin(), asks.end(), OrderBookEntry::compareByPriceAsc);
    // sort bids highest first
    std::sort(bids.begin(), bids.end(), OrderBookEntry::compareByPriceDesc);
    std::cout << "ask size: " << asks.size() << std::endl;
    std::cout << "max ask " << asks[asks.size() - 1].price << std::endl;
    std::cout << "min ask " << asks[0].price << std::endl;
    std::cout << "max bid " << bids[0].price << std::endl;
    std::cout << "min bid " << bids[bids.size() - 1].price << std::endl;
```

Figure 5.1A – OrderBook.cpp (line 128 – 143)

The function takes in the product and timestamp that needs to be matched as inputs.

The first 3 lines initialises 3 OrderBookEntry vectors to hold asks, bids and sales orders. The asks and bids are retrieved with the getOrder() function with their respective orderTypes, product(string) and timestamp(string) as the parameters.

The asks and bids are then sorted in ascending and descending orders respectively.



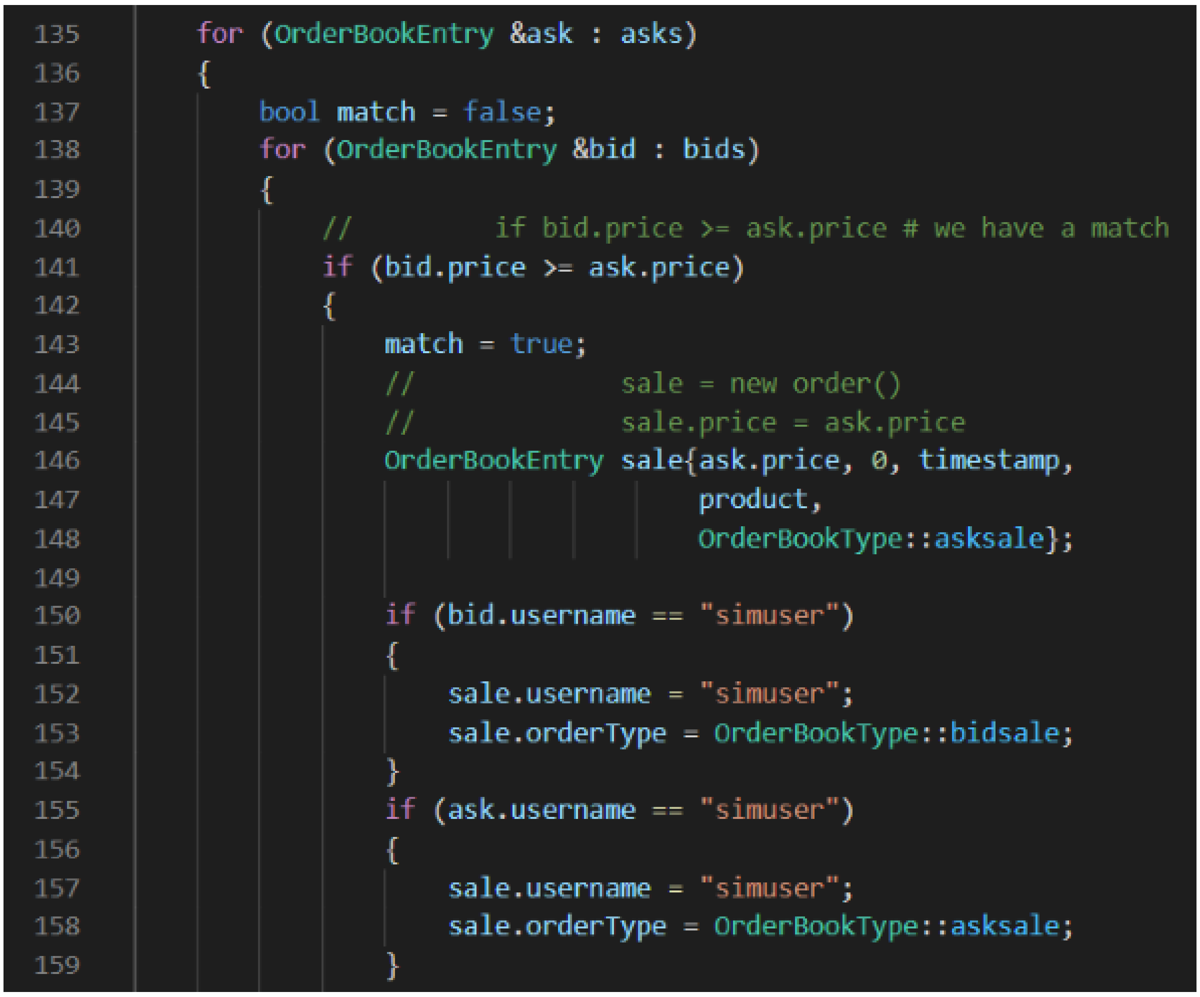
```
for (OrderBookEntry &ask : asks)
{
    for (OrderBookEntry &bid : bids)
    {
        if (bid.price >= ask.price)
        {
            OrderBookEntry sale{ask.price, 0, timestamp, product, OrderBookType::asksale};
            if (bid.username == "simuser")
            {
                sale.username = "simuser";
                sale.orderType = OrderBookType::bidsale;
            }
            if (ask.username == "simuser")
            {
                sale.username = "simuser";
                sale.orderType = OrderBookType::asksale;
            }
            if (bid.amount == ask.amount)
            {
                sale.amount = ask.amount;
                sales.push_back(sale);
                bid.amount = 0;
                break;
            }
            //          if bid.amount > ask.amount:  # ask is completely gone slice the bid
            if (bid.amount > ask.amount)
            {
                sale.amount = ask.amount;
                sales.push_back(sale);
                bid.amount -= ask.amount;
                break;
            }

            //            if bid.amount < ask.amount # bid is completely gone, slice the ask
            if (bid.amount < ask.amount && bid.amount > 0)
            {
                sale.amount = bid.amount;
```

Figure 5.1B – OrderBook.cpp (line 145 – 181)

In this matching algorithm, the code iterates through every ask and bid which will take NM time to run where N = size of asks vector and M = size of bids vector. However, as we know that the asks and bids are sorted, we know that if asks of index 'i' does not match with bids of index 'j' for all values of j where 0 <= j < bids.size(), all asks of index greater than 'i' will not match any bids since asks is arranged in ascending order.

With this information, we know that we can break from both for loops when no trades are found at the end of the loop through bids.

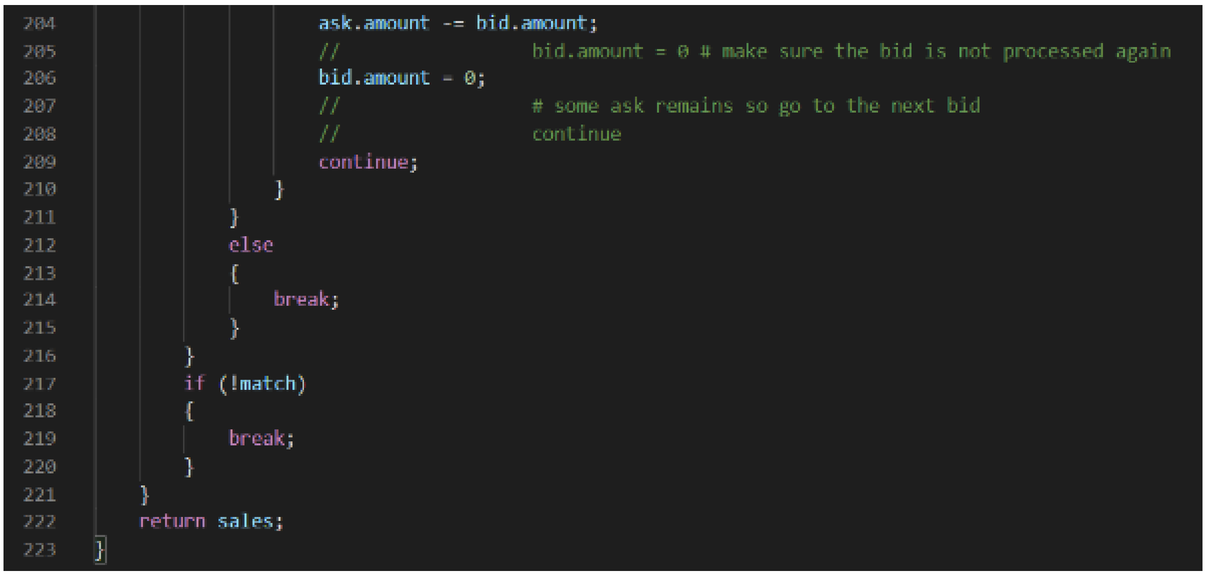
The order and bid processing after optimisation:
Implementing into code:



```
    for (OrderBookEntry &ask : asks)
    {
        bool match = false;
        for (OrderBookEntry &bid : bids)
        {
            //         if bid.price >= ask.price # we have a match
            if (bid.price >= ask.price)
            {
                match = true;
                //              sale = new order()
                //              sale.price = ask.price
                OrderBookEntry sale{ask.price, 0, timestamp,
                                    product,
                                    OrderBookType::asksale};

                if (bid.username == "simuser")
                {
                    sale.username = "simuser";
                    sale.orderType = OrderBookType::bidsale;
                }
                if (ask.username == "simuser")
                {
                    sale.username = "simuser";
                    sale.orderType = OrderBookType::asksale;
                }
```

Figure 5.2A – OrderBook.cpp (line 135 – 159)

We can simply create a bool variable and initialise it with a false value, and while running the matching algorithm, if there is a match, set it to true.



```
204                     ask.amount -= bid.amount;
205                     //                 bid.amount = 0 # make sure the bid is not processed again
206                     bid.amount = 0;
207                     //                 # some ask remains so go to the next bid
208                     //                 continue
209                     continue;
210                 }
211             }
212             else
213             {
214                 break;
215             }
216         }
217         if (!match)
218         {
219             break;
220         }
221     }
222     return sales;
223 }
```

Figure 5.2B – OrderBook.cpp (line 204 – 223)

After the if check of (bid.price >= ask.price), we can add a break for the else at the end of the if operation as seen in line 213 to 215:

if (bid.price >= ask.price){

    // actions

}else{ //213

    break; //214

} //215

Since the bids are arranged in descending order, we know that the first bid will be the highest bid. If there are no matches, we can break from the bid loop.

At the end of the bids for loop (for(OrderBookEntry &bid : bids)), write another if function to check if the match is false. When the value match is false, the code will break from the asks for loop and proceed to return sales.

If a match is found, the code will loop through the next asks order.

The bool match = false; in line 136 will run again, setting the match value to false again.

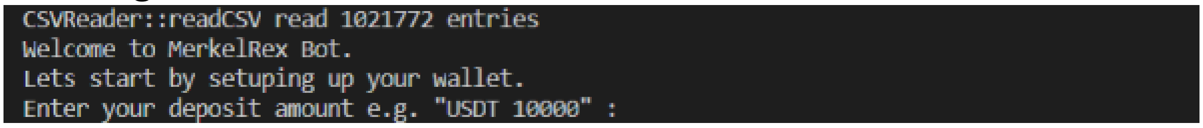
With this optimisation, the matching engine will no longer have to loop through every ask and bid orders unnecessarily and will immediately exit the engine when it knows that there will not be any possible matches for a sale to be created from that point on.

Best case time taken before optimisation: for loop runs through ask.size() * bid.size() times.

Best case time taken after optimisation: for loop runs for constant time

Best case would be that the bid price of index 0 is not higher than the ask price of index 0. In this case, the program will exit both for loop immediately.

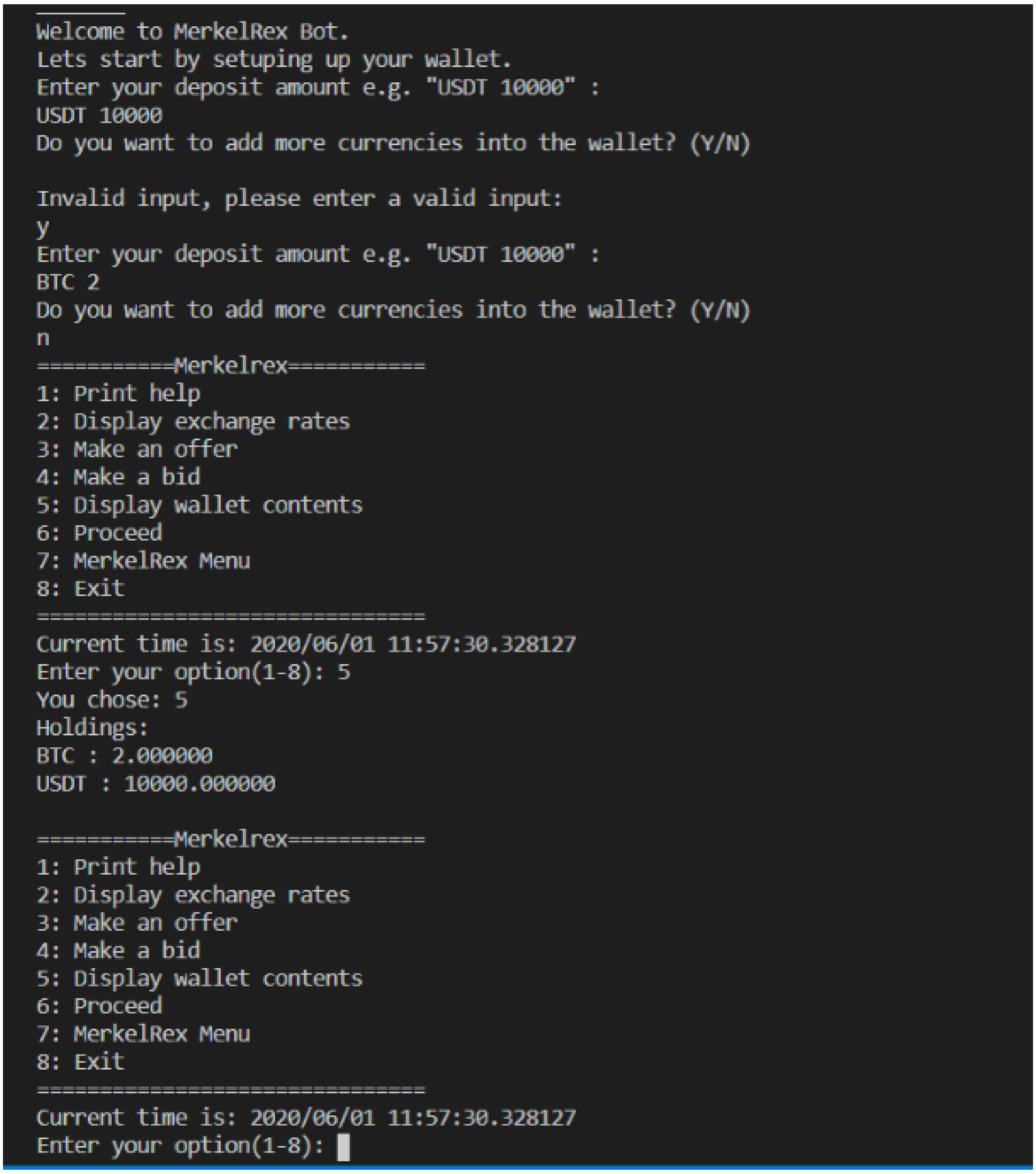
Testing:



```
CSVReader::readCSV read 1021772 entries
Welcome to MerkelRex Bot.
Lets start by setuping up your wallet.
Enter your deposit amount e.g. "USDT 10000" :
```

Figure 6.1A – MerkelRex first user input

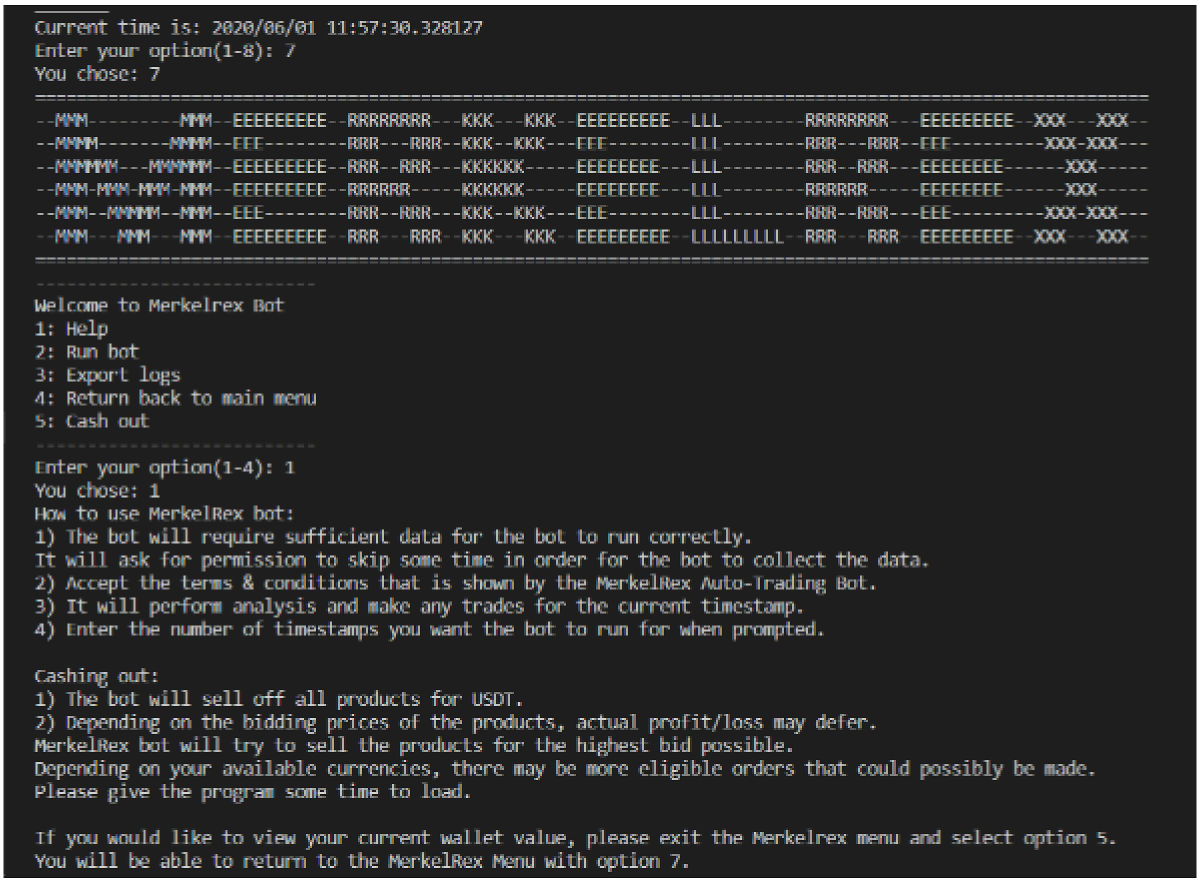
When running the program, we will be prompted to deposit currencies for the program.



```
Welcome to MerkelRex Bot.
Lets start by setuping up your wallet.
Enter your deposit amount e.g. "USDT 10000" :
USDT 10000
Do you want to add more currencies into the wallet? (Y/N)

Invalid input, please enter a valid input:
y
Enter your deposit amount e.g. "USDT 10000" :
BTC 2
Do you want to add more currencies into the wallet? (Y/N)
n
===========Merkelrex===========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
===============================
Current time is: 2020/06/01 11:57:30.328127
Enter your option(1-8): 5
You chose: 5
Holdings:
BTC : 2.000000
USDT : 10000.000000

===========Merkelrex===========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
===============================
Current time is: 2020/06/01 11:57:30.328127
Enter your option(1-8): █
```

Figure 6.1B – MerkelRex entering currencies

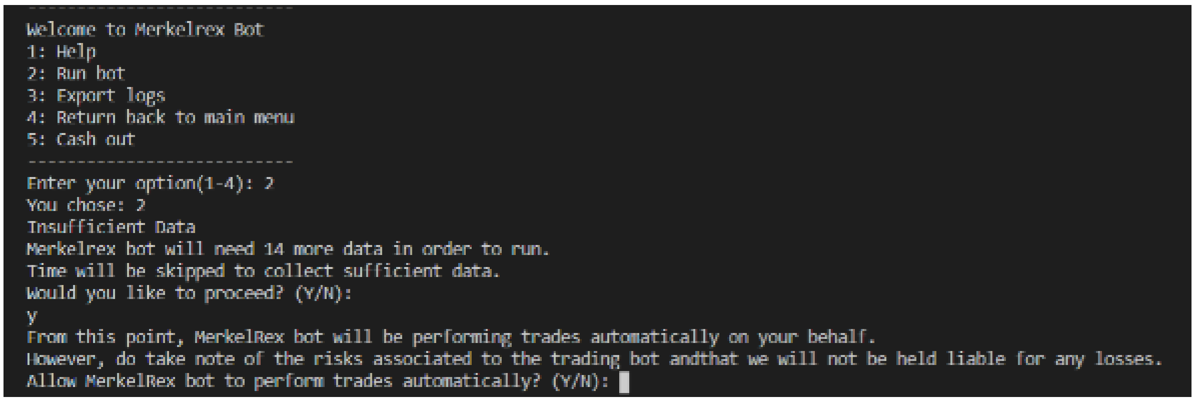
In figure 6.1B, we have inserted 10000 USDT and 2 BTC into the wallet and when we print the wallet contents, the wallet holds the correct amount. As there is no data collected yet, when printing the wallet, we will not get any approximate value of the currencies.



```
Current time is: 2020/06/01 11:57:30.328127
Enter your option(1-8): 7
You chose: 7
==========================================================================================
--MMM---------MMM--EEEEEEEEE--RRRRRRRR---KKK---KKK--EEEEEEEEE--LLL--------RRRRRRRR---EEEEEEEEE--XXX---XXX--
--MMMM-------MMMM--EEE--------RRR---RRR--KKK--KKK---EEE--------LLL--------RRR---RRR--EEE---------XXX-XXX---
--MMMMMM---MMMMMM--EEEEEEEEE--RRR--RRR---KKKKKK-----EEEEEEEE---LLL--------RRR--RRR---EEEEEEEE------XXX-----
--MMM-MMM-MMM-MMM--EEEEEEEEE--RRRRRR-----KKKKKK-----EEEEEEEE---LLL--------RRRRRR-----EEEEEEEE------XXX-----
--MMM--MMMMM--MMM--EEE--------RRR--RRR---KKK--KKK---EEE--------LLL--------RRR--RRR---EEE---------XXX-XXX---
--MMM---MMM---MMM--EEEEEEEEE--RRR---RRR--KKK---KKK--EEEEEEEEE--LLLLLLLLL--RRR---RRR--EEEEEEEEE--XXX---XXX--
==========================================================================================
--------------------------
Welcome to Merkelrex Bot
1: Help
2: Run bot
3: Export logs
4: Return back to main menu
5: Cash out
--------------------------
Enter your option(1-4): 1
You chose: 1
How to use MerkelRex bot:
1) The bot will require sufficient data for the bot to run correctly.
It will ask for permission to skip some time in order for the bot to collect the data.
2) Accept the terms & conditions that is shown by the MerkelRex Auto-Trading Bot.
3) It will perform analysis and make any trades for the current timestamp.
4) Enter the number of timestamps you want the bot to run for when prompted.

Cashing out:
1) The bot will sell off all products for USDT.
2) Depending on the bidding prices of the products, actual profit/loss may defer.
MerkelRex bot will try to sell the products for the highest bid possible.
Depending on your available currencies, there may be more eligible orders that could possibly be made.
Please give the program some time to load.

If you would like to view your current wallet value, please exit the Merkelrex menu and select option 5.
You will be able to return to the MerkelRex Menu with option 7.
```

Figure 6.1C – MerkelRex Menu Help option

Figure 6.1C shows the result of selecting option 7 and running the help option in MerkelRex menu.

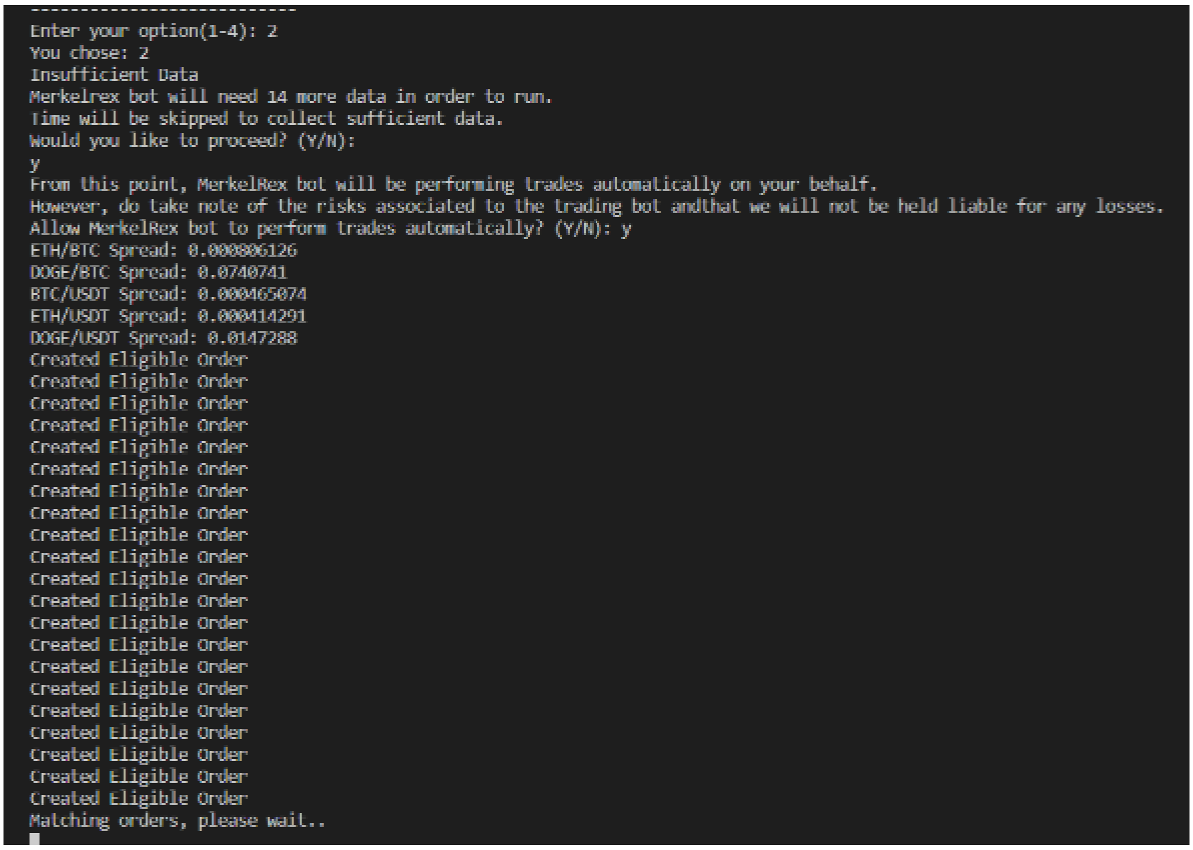


```
--------------------------
Welcome to Merkelrex Bot
1: Help
2: Run bot
3: Export logs
4: Return back to main menu
5: Cash out
--------------------------
Enter your option(1-4): 2
You chose: 2
Insufficient Data
Merkelrex bot will need 14 more data in order to run.
Time will be skipped to collect sufficient data.
Would you like to proceed? (Y/N):
y
From this point, MerkelRex bot will be performing trades automatically on your behalf.
However, do take note of the risks associated to the trading bot andthat we will not be held liable for any losses.
Allow MerkelRex bot to perform trades automatically? (Y/N):
```

Figure 6.1D – MerkelRex run bot

When we select option 2 to run the bot, program bot will ask user to allow the program to skip time to collect data.

If user allow, the bot will print terms and conditions and ask user to allow program to perform trade automatically.

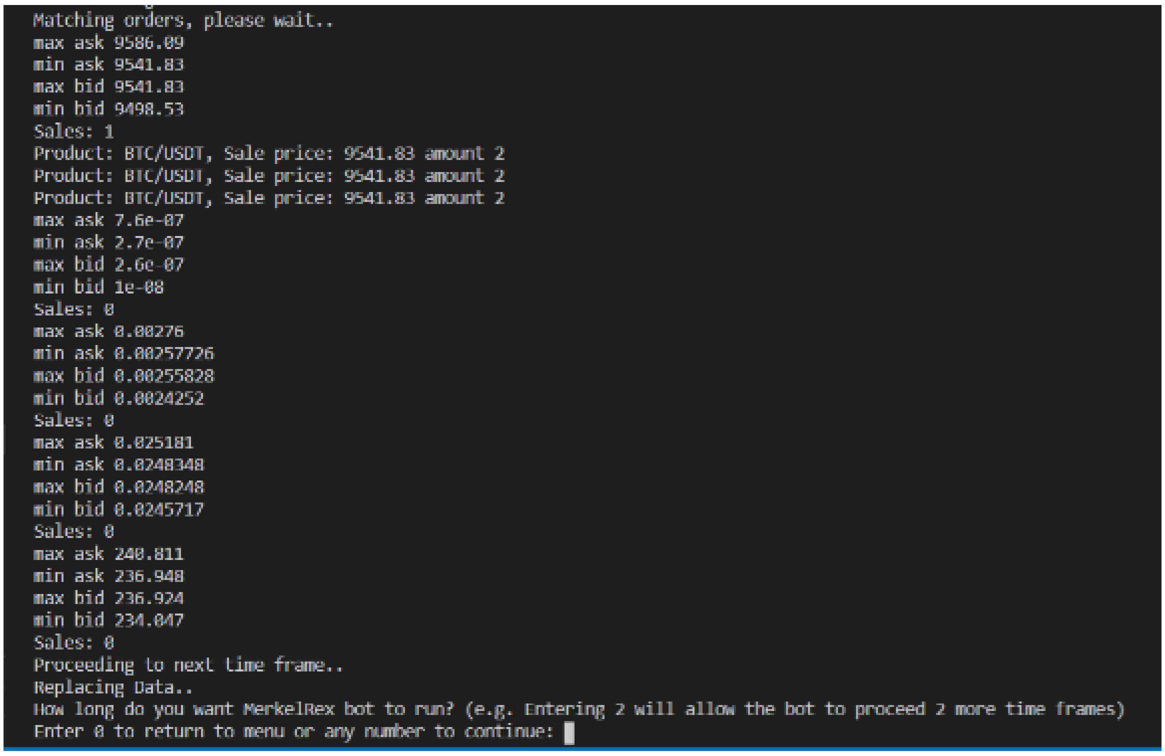


```
--------------------------
Enter your option(1-4): 2
You chose: 2
Insufficient Data
Merkelrex bot will need 14 more data in order to run.
Time will be skipped to collect sufficient data.
Would you like to proceed? (Y/N):
y
From this point, MerkelRex bot will be performing trades automatically on your behalf.
However, do take note of the risks associated to the trading bot andthat we will not be held liable for any losses.
Allow MerkelRex bot to perform trades automatically? (Y/N): y
ETH/BTC Spread: 0.000806126
DOGE/BTC Spread: 0.0740741
BTC/USDT Spread: 0.000465074
ETH/USDT Spread: 0.000414291
DOGE/USDT Spread: 0.0147288
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Created Eligible Order
Matching orders, please wait..
```

Figure 6.1E – MerkelRex run bot 2

When user enters any of the yes option (“y”, “yes, “Y”, “Yes), the bot will begin running the algorithm. Eligible orders may or may not be created. Any products with spread greater than 0.02 will not be traded.

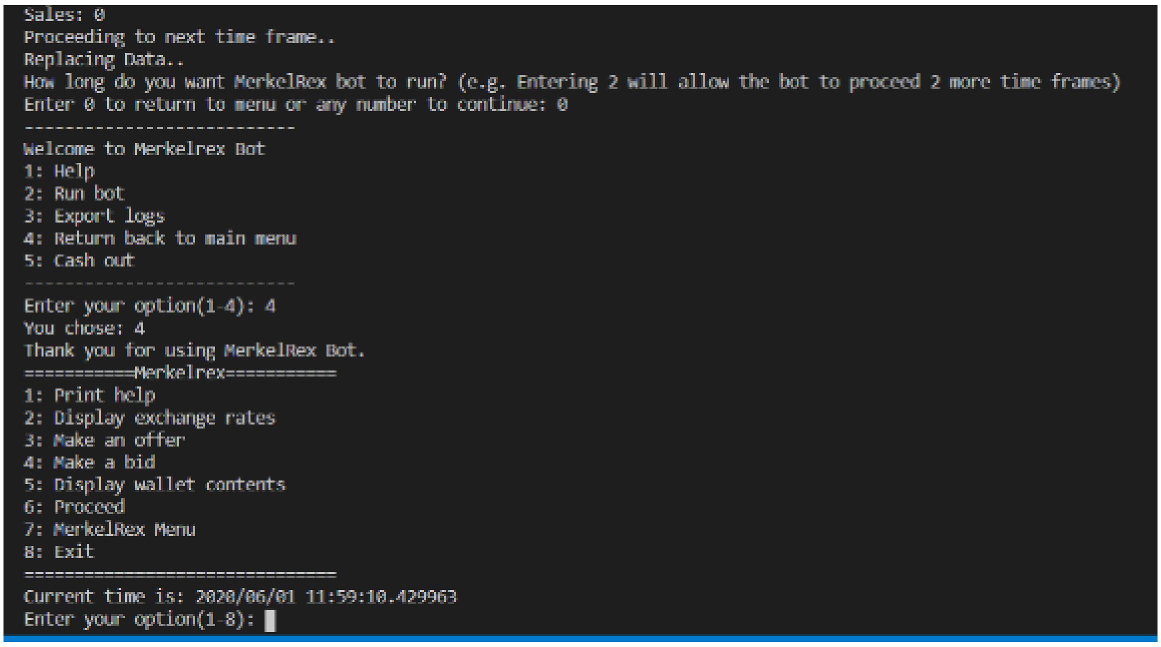
Depending on the number of orders, the matching orders process may take some time.



```
Matching orders, please wait..
max ask 9586.09
min ask 9541.83
max bid 9541.83
min bid 9498.53
Sales: 1
Product: BTC/USDT, Sale price: 9541.83 amount 2
Product: BTC/USDT, Sale price: 9541.83 amount 2
Product: BTC/USDT, Sale price: 9541.83 amount 2
max ask 7.6e-07
min ask 2.7e-07
max bid 2.6e-07
min bid 1e-08
Sales: 0
max ask 0.00276
min ask 0.00257726
max bid 0.00255828
min bid 0.0024252
Sales: 0
max ask 0.025181
min ask 0.0248348
max bid 0.0248248
min bid 0.0245717
Sales: 0
max ask 240.811
min ask 236.948
max bid 236.924
min bid 234.047
Sales: 0
Proceeding to next time frame..
Replacing Data..
How long do you want MerkelRex bot to run? (e.g. Entering 2 will allow the bot to proceed 2 more time frames)
Enter 0 to return to menu or any number to continue: ▌
```

Figure 6.1F – MerkelRex run bot 3

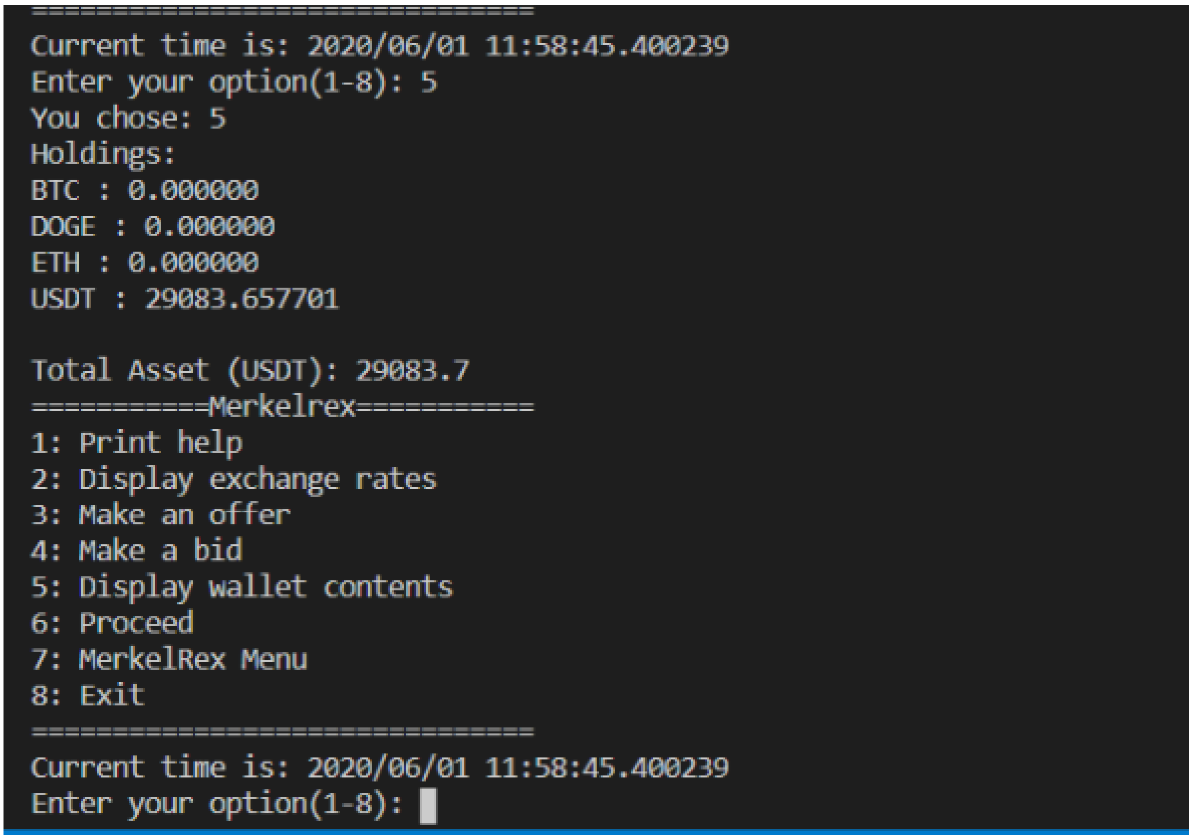
When matching is complete, any sale made will be shown. The program prompts the user for number of times they want the program to run. We will be testing 5 as the input to let the program run 5 loops.



```
Sales: 0
Proceeding to next time frame..
Replacing Data..
How long do you want MerkelRex bot to run? (e.g. Entering 2 will allow the bot to proceed 2 more time frames)
Enter 0 to return to menu or any number to continue: 0
--------------------------
Welcome to Merkelrex Bot
1: Help
2: Run bot
3: Export logs
4: Return back to main menu
5: Cash out
--------------------------
Enter your option(1-4): 4
You chose: 4
Thank you for using MerkelRex Bot.
==========Merkelrex==========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
==============================
Current time is: 2020/06/01 11:59:10.429963
Enter your option(1-8): ▌
```

Figure 6.1G – MerkelRex return to menu

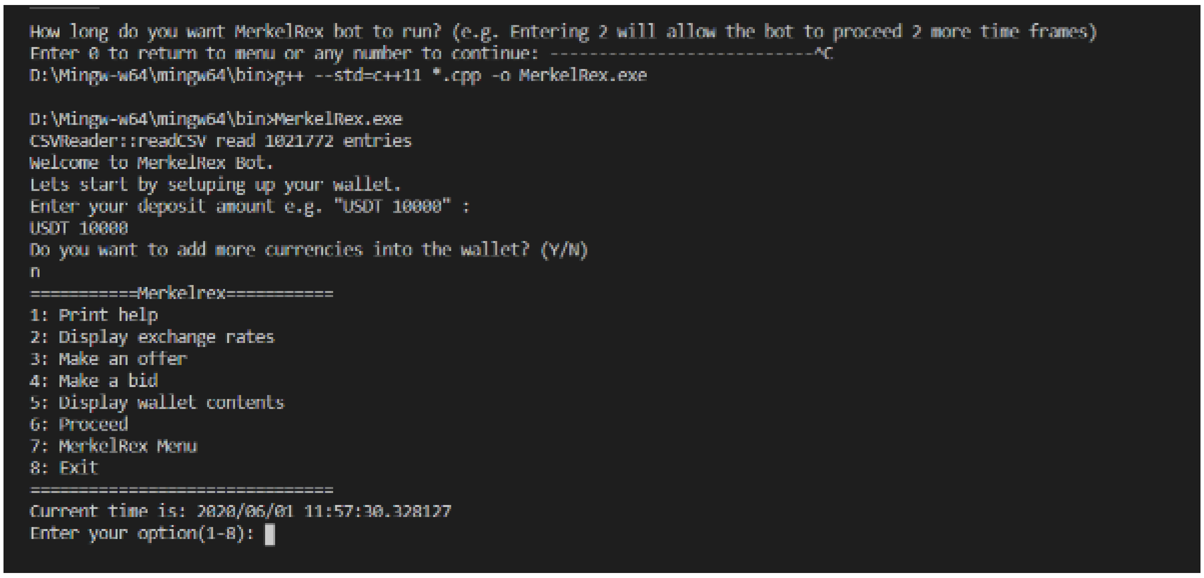
After running 5 loops, we will exit the loop by entering 0. Selecting option 4 in MerkelRex menu will allow us to return to the main menu.



```
================================
Current time is: 2020/06/01 11:58:45.400239
Enter your option(1-8): 5
You chose: 5
Holdings:
BTC : 0.000000
DOGE : 0.000000
ETH : 0.000000
USDT : 29083.657701

Total Asset (USDT): 29083.7
==========Merkelrex==========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
================================
Current time is: 2020/06/01 11:58:45.400239
Enter your option(1-8): 
```

Figure 6.1H – MerkelRex print wallet contents after run

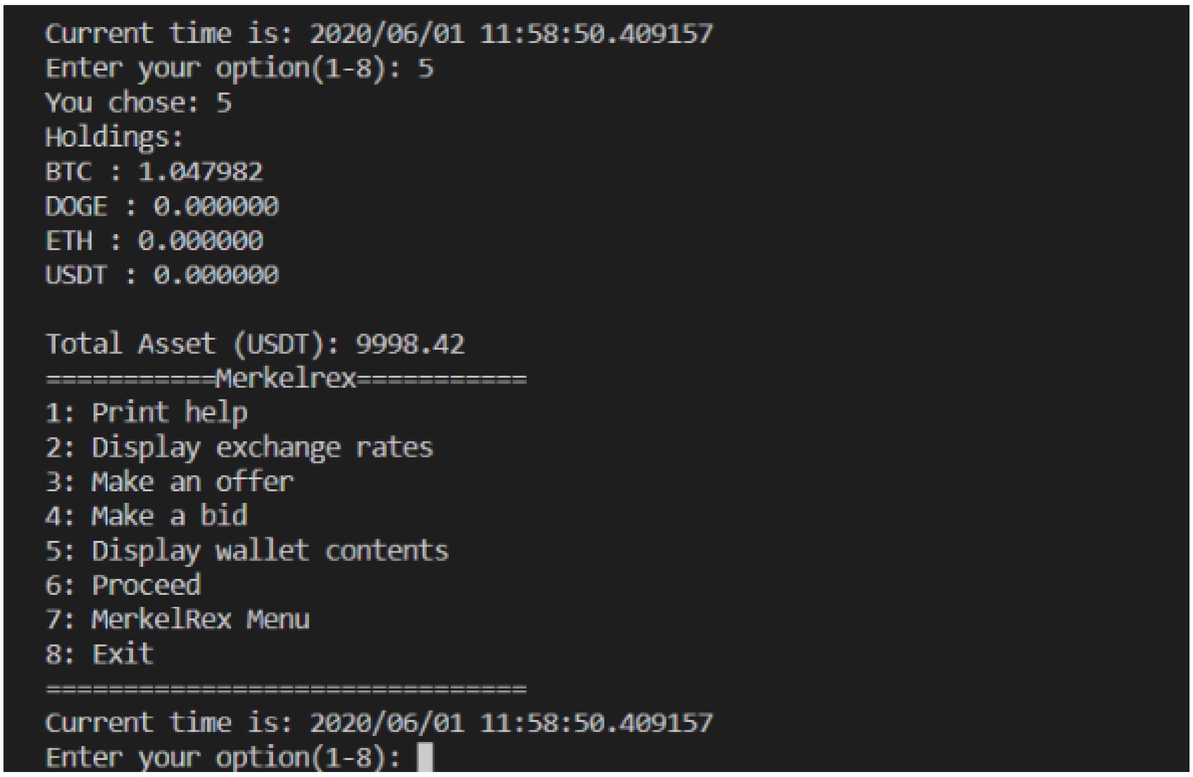
Printing wallet contents will show that we have sold BTC.



```
How long do you want MerkelRex bot to run? (e.g. Entering 2 will allow the bot to proceed 2 more time frames)
Enter 0 to return to menu or any number to continue: ----------------------------^C
D:\Mingw-w64\mingw64\bin>g++ --std=c++11 *.cpp -o MerkelRex.exe

D:\Mingw-w64\mingw64\bin>MerkelRex.exe
CSVReader::readCSV read 1021772 entries
Welcome to MerkelRex Bot.
Lets start by setuping up your wallet.
Enter your deposit amount e.g. "USDT 10000" :
USDT 10000
Do you want to add more currencies into the wallet? (Y/N)
n
==========Merkelrex==========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
=============================
Current time is: 2020/06/01 11:57:30.328127
Enter your option(1-8):
```

Figure 6.2A

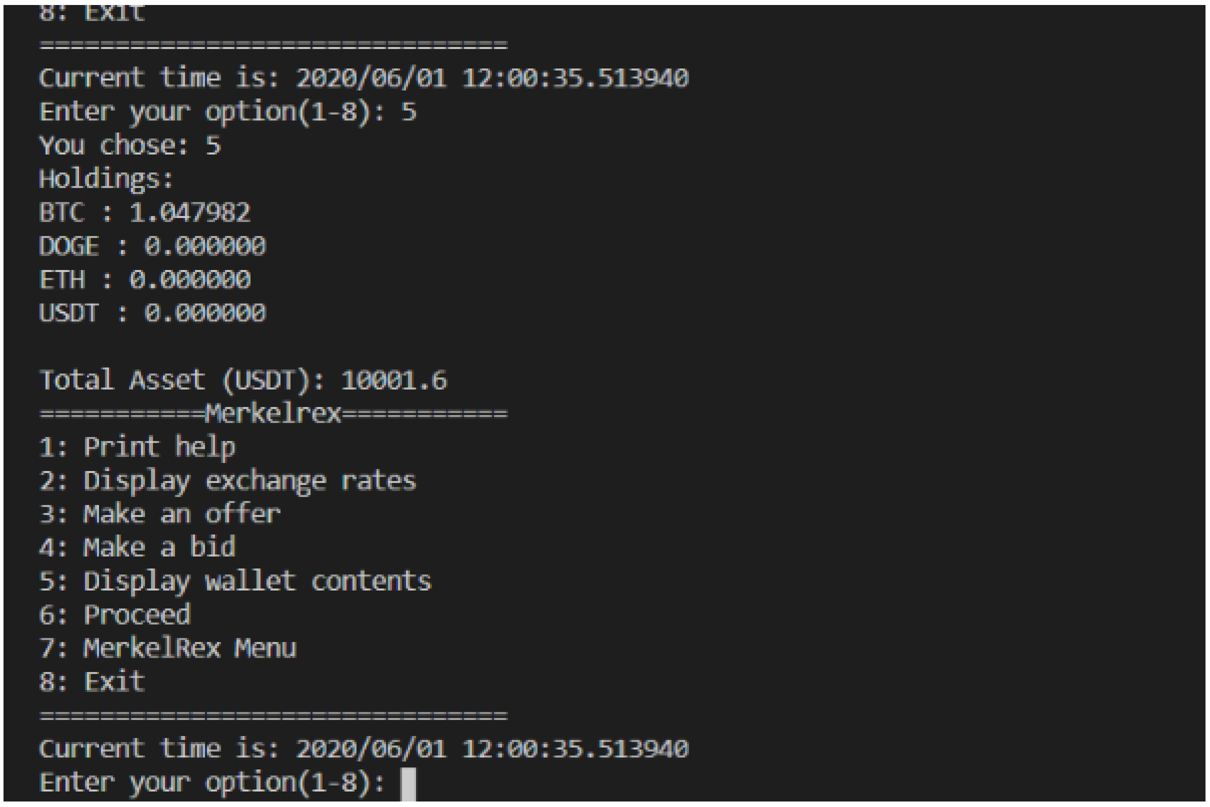
I will run the program with USDT as the starting currency to show the changes in asset value.



```
Current time is: 2020/06/01 11:58:50.409157
Enter your option(1-8): 5
You chose: 5
Holdings:
BTC : 1.047982
DOGE : 0.000000
ETH : 0.000000
USDT : 0.000000

Total Asset (USDT): 9998.42
===========Merkelrex===========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
===============================
Current time is: 2020/06/01 11:58:50.409157
Enter your option(1-8):
```

Figure 6.2B

After running 2 cycles, 10000 USDT has been exchanged 1.047982 BTC. The estimated asset value is as shown.



```
8: Exit
================================
Current time is: 2020/06/01 12:00:35.513940
Enter your option(1-8): 5
You chose: 5
Holdings:
BTC : 1.047982
DOGE : 0.000000
ETH : 0.000000
USDT : 0.000000

Total Asset (USDT): 10001.6
===========Merkelrex===========
1: Print help
2: Display exchange rates
3: Make an offer
4: Make a bid
5: Display wallet contents
6: Proceed
7: MerkelRex Menu
8: Exit
================================
Current time is: 2020/06/01 12:00:35.513940
Enter your option(1-8):
```

Figure 6.2C

After running for some time, the asset value changes as seen in the above.

Conclusion:

Throughout the testing, I have found that the program appears to only be able to make the initial trade. After running for many cycles there may be occasional eligible orders made but does not seem to have any actual sales being made. This may be due to many conditions being set such as spread value where any products that have DOGE coin involved will not be traded as the particular product has a very low liquidity and high cost of exchange.

However, by looking through the data file, I have noticed that the predicted price seems to be fairly accurate as the prices further down the timestamp tend to hit the predicted price. In order to take profits, I may have to sell products manually.

The automatic trading bot has been able to generate orders automatically, predict prices and collect data throughout the different time stamps and replace them successfully.

Model constantly updates itself with new data and it is also able to convert its assets to USDT value. Furthermore, the program ensures that all trades made are able to be paid with the currencies held in the wallet, preventing any currencies from going to negative value.

Area of improvements are discussed throughout the report.

References:

[1] JASON FERNANDO. (2020, November 17). Relative Strength Index (RSI). Investopedia. https://www.investopedia.com/terms/r/rsi.asp

[2] ADAM MILTON. (2020, July 21). Buying and Selling Volume. The Balance. https://www.thebalance.com/buying-and-selling-volume-1031027