UNIVERSITY OF LONDON

# CM2005: OBJECT ORIENTED PROGRAMMING


**Student Name:** Lawrence Ho Sheng Jin

**Date Submitted:** 11th January 2020

**Degree Title:** Computer Science

**Local Institution:** Singapore Institute of Management

**Student ID:** 10205927

# Table of Contents

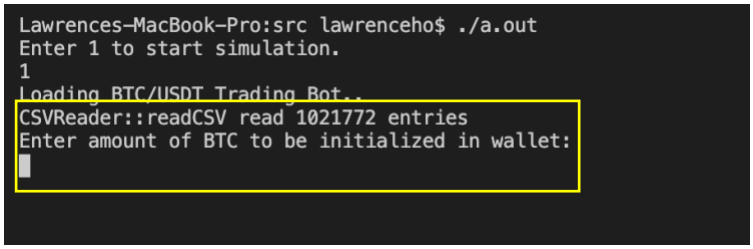
# Introduction of Bot

Currently, the bot only deals with a single product: BTC/USDT. Its profit percentage is set to a 0.01% increment, and the bot will stop running once it hits the target amount. After every timestamp, the wallet's balance will be displayed for the user to monitor changes. At the end of the bot, there will be a logging function upon the user's choice to display transaction histories or not.

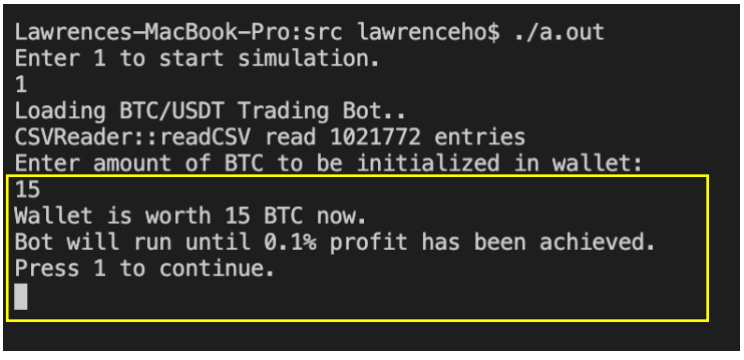
An overview of the bot's execution phase will be shown in the following screenshots:

1. Firstly, the bot will prompt user to enter '1' to start the simulation. Upon entering '1' by the user, the bot will retrieve the live order book from the Merklerex exchange simulation and the console will print out the loading text.

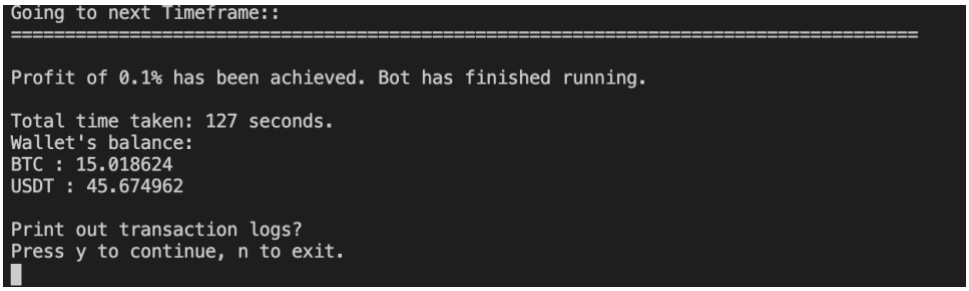

2. When it has finished retriving orders from the live order book, it will display the number of entries read. Then, the bot will prompt the user to enter the desired BTC balance in the wallet.



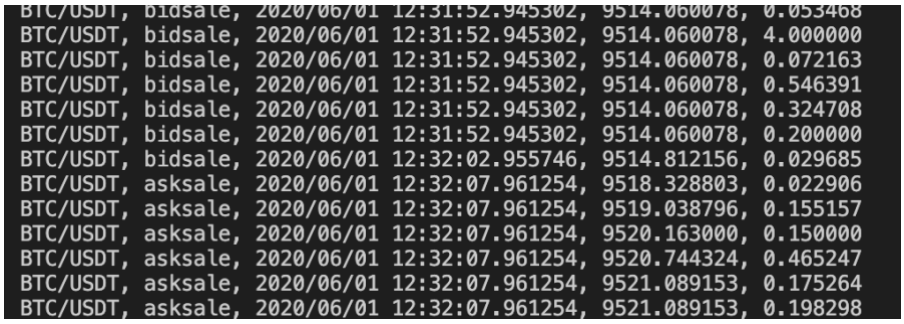

3. Once the user has entered an amount, the bot will print out the balance in the wallet, and tell the user the terminating condition for the bot, which in this case is a 0.01% profit in BTC. Then, it prompts the user to enter '1' again to execute the trading phase. From then on, the bot will start its automated trading process.



4. Once it has reached the end of the bot's trading process, it will prompt the user whether to display all of the transaction histories.




5. Upon entering 'y', the bot will print out all the transaction logs in the console, as well as streaming it to a separate .txt file.




Otherwise, if the user enters 'n', the bot will stop running and will print out a message in the console indicating that the bot has terminated.

# R1: Market Analysis

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

I have added an OrderBook instance into my bot.h file, which will call the class constructor and pass in the CSV file "20200317.csv". The constructor then calls the readCSV method from the class CSVReader which converts the lines in the file to OrderBookEntry objects.

```
OrderBook orderBook{"20200317.csv"}; //read CSV file name
/** construct, reading a csv data file */
OrderBook::OrderBook(std::string filename)
{
    orders = CSVReader::readCSV(filename);
}
```

Fig 1.1

---

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

My objective for the bot to earn a profit would be to extract the average lowest and highest prices over a period of time to buy low and sell high. As such, I have implemented two methods to help predict future market exchange rates.

1. Take the Simple Moving Average (SMA) of ask/bid orders as the bot loops through the exchange.
    a. For looping through ask orders in the exchange, the bot will gather data on the past lowest prices and take the average of it to determine whether to place a bid or not.
    b. For looping through bid orders in the exchange, the bot will gather data on the past highest prices and take the average of it to determine whether to place an ask or not.
    c. This method continuously takes in a fixed amount of values to determine the SMA, similar to that of a sliding window(sub-vector) moving across a vector.

The method to calculate SMA would be located in the bot.cpp file. Due to the need to reference many values in the OrderBookEntries, I have created a number of instance variables (vector<double> and double) to store these values for different methods to be able to access them.

The instance variables used are listed below (Fig 1.2):

```
//SMA
int windowSize = 10; //double to store moving window size for SMA and moving slope (LR/SMA)
double askSMA=0, bidSMA=0; //double to store current SMA for ask/bid (SMA)
double sumAskSMA, sumBidSMA; //double to store sum of ask/bid over a period of time before dividing with the
number of ask/bid prices to get the current Timestamp's SMA
```

```
std::vector<double> priceVecAsks, priceVecBids; //vector to store values of minimum price for asks and
maximum price of bids in the exchange
```

Fig 1.2

The class function to calculate SMA is listed below (Fig 1.3):

*The code below was referenced from Samgak, who replied to a query on Stack Overflow regarding calculating simple moving averages (2015).*

```
//takes in current counter of cycles the bot has gone through
void Bot::calcSMA(int n)
{
    //adds the min price of the latest OrderBookEntries to sumAskSMA and sumBidSMA
    sumAskSMA += priceVecAsks[n];
    sumBidSMA += priceVecBids[n];

    //if the no. of cycles the bot has gone through > windowSize,
    if(n >= windowSize)
    {
        //check for the oldest index within the vector to subtract
        int replaceIndex = n - windowSize;
        //subtracts the min and max price of the oldest OrderBookEntries
        sumAskSMA -= priceVecAsks[replaceIndex];
        sumBidSMA -= priceVecBids[replaceIndex];
    }

    //when the bot has gone through the same number of cycles as the value of windowSize or more
    if(n >= (windowSize-1))
    {
        //calculates the current SMA values by dividing with the number of elements (windowSize)
        askSMA = sumAskSMA / windowSize;
        bidSMA = sumBidSMA / windowSize;
    }
}
```

Fig 1.3

The method takes in the current number of cycles the bot has gone through. The latest min and max price is then added into a sum variable. If the current number of cycles is greater than the set value of windowSize, it subtracts the oldest min and max price from the sum variable.

While the number of cycles is in range of the windowSize, it continuously calculates the new SMA for the current timestamp.

2. Applying Linear Regression (LR) to the prices to determine the trend of increasing or decreasing prices. The bot will take the slope gradient that into consideration in determining whether to place an order.

   If the trend shows an increment in prices over a set window of time, it indicates that the future prices might not continue going up, and now is a good time to place an ask to sell it at the best possible price. Likewise, if the trend shows a decrement in prices over a set window of time, it indicates that the future prices might not continue going down, and that now is a good time to place a bid to buy at the best possible price.

I have placed the algorithm code for linear regression in a separate class named Algo, as it only requires the reference of two vectors to be passed into its parameters to get the job done. This distribution makes it easier to debug any issues with the SMA or linear regression functions.

The main idea of linear regression is to plot the points of the trend of prices into a linear graph:

- Y = mx + c
- m = gradient (slope)
- c = y-intercept

By calculation the gradient of the slope, the bot is able to determine the rise or drop in prices over a set window of time. Along with the y-intercept, the bot is able to predict future prices based on the equation by taking the latest gradient, y-intercept and a future x value.

The instance variables used are listed below (Fig 1.4):

```
//LR
//vector to store values for LR (analysis)
std::vector<double> priceVecAsks, priceVecBids, askTimeX, bidTimeX;


//double to store slope gradient for ask/bid (LR)
double slopeAsk, slopeBid;


//double to store y-intercept for ask/bid (LR)
double interceptAsk, interceptBid;


//vector to store all the slope gradients (LR)
std::vector<double> slopeAskVec, slopeBidVec;
```

Fig 1.4

The class function to calculate LR comprises of some other helper methods. These helper methods are listed below (Fig 1.5):

```
/**calculating the gradient for linear regression*/
double slope(const vector<double>& x, const vector<double>& y);
/**calculating y-intercept*/
double intercept(const vector<double>& x, const vector<double>& y, double gradient);
```

Fig 1.5a (methods in algo.h)

```
/**get a section of an array to be used as the moving window of prices*/
std::vector<double> getSubVector(std::vector<double> &priceVec, int startIndex, int endIndex);
```

Fig 1.5b (methods in bot.h)

The function to calculate LR is stored in the class algo. It converts the equation (y=mx+c) into code, as shown below (Fig 1.6):

*The code below was referenced from Cassio Neri, who replied to a query on Stack Overflow regarding linear regression slope (2014).*

```
/**calculating the gradient for linear regression*/
//takes in two vectors (one for x-axis, one for y-axis)
//x-axis = ask/bid prices; y-axis = time

double Algo::slope(const std::vector<double>& x, const std::vector<double>& y) {
    double sumX  = accumulate(x.begin(), x.end(), 0.0);
    double sumY  = accumulate(y.begin(), y.end(), 0.0);
    double sumX_X = inner_product(x.begin(), x.end(), x.begin(), 0.0);
    double sumX_Y = inner_product(x.begin(), x.end(), y.begin(), 0.0);
    double slope = ((x.size() * sumX_Y - sumX * sumY) *1.0 )/ ((x.size() * sumX_X - sumX * sumX) * 1.0 );
    return slope;
}

/**calculating y-intercept*/
//takes in two vectors (one for x-axis, one for y-axis)
//x-axis = ask/bid prices; y-axis = time
double Algo::intercept(const std::vector<double>& x, const std::vector<double>& y, double slope)
{
    double x_size = x.size();
    double sumX  = accumulate(x.begin(), x.end(), 0.0);
    double sumY  = accumulate(y.begin(), y.end(), 0.0);
    return (sumY - slope * sumX) / x_size;
}
```

Fig 1.6a (algo.cpp)

In the bot class, the bot retrieves a sub-section of the prices vector to be used as the moving window of values (as the bot goes through the exchange). Fig 1.7:

```
/**get a section of an array to be used as the moving window of prices*/
std::vector<double> Bot::getSubVector(std::vector<double> &priceVec, int startIndex, int endIndex)
{
  auto first = priceVec.begin() + startIndex;
```

```
  auto last = priceVec.begin() + endIndex + 1;
  std::vector<double> vector(first, last);
  return vector;
}
```

Fig 1.7 (bot.cpp)

Lastly, the main function that groups all of these helper functions together to calculate the slope gradient (Fig 1.8):

```
/**calculate sub-section of the slopes in the graph to determine gradient of slope*/
//takes in current counter of cycles the bot has gone through
void Bot::calcMovingSlope(int n)
{
   std::vector<double> subVecAsks, subVecBids, subTimeAskX, subTimeBidX;

      //if the no. of cycles the bot has gone through > windowSize,
      if(n >= windowSize)
      {
         //gets the correct starting index for windowSize amount of values
         int startIndex = n - windowSize;
         //extracts sub vector of windowSize values for asks and bids
         subVecAsks = getSubVector(priceVecAsks, startIndex, n);
         subVecBids = getSubVector(priceVecBids, startIndex, n);
         //extracts sub vector of windowSize values for ask and bid time
         subTimeAskX = getSubVector(askTimeX, startIndex, n);
         subTimeBidX = getSubVector(bidTimeX, startIndex, n);
      }

      //when the bot has gone through the same number of cycles as the value of windowSize or more
      if(n >= (windowSize-1))
      {
         //calculate slope gradient for asks and bids
         slopeAsk = algo.slope(subTimeAskX, subVecAsks);
         slopeBid = algo.slope(subTimeBidX, subVecBids);
         //calculate y-intercept for asks and bids
         interceptAsk = algo.intercept(subTimeAskX, subVecAsks, slopeAsk);
         interceptBid = algo.intercept(subTimeBidX, subVecBids, slopeBid);
      }
}
```

Fig 1.8

The window size I have set for my bot is 10, and the bot will then calculate the moving slope gradient for 10 timestamps based on the minimum price for asks (push into vector *priceVecAsks)* and maximum price for bids (push into vector *priceVecBids)* in the exchange. The counter for the number timestamps the bot has gone through will be pushed back into vectors askTimeX and bidTimeX.

*(The price vectors will be the y-axis of the graph, and the number of timestamps will be the x-axis. )*

A sub vector (*subVecAsks*, *subVecBids*) of 10 values (1 value for each timestamp) will then be extracted from the main *priceVecAsks* and *priceVecBids* vector. Similarly, the same will be done for the time vectors. From there, these arguments will be passed into the slope calculation algorithm, and the value of the gradient for asks and bids prices over a course of 10 timestamps will be collected to serve as a condition for the bot to place orders.

Hence, to summarize my bot's trading algorithm, is to calculate the simple moving averages of prices for both bids (max) and asks (min), and apply linear regression on the prices for a set window size of 10. Once the listing price is lower than the simple moving average and trend of the slope gradient calculated over 10 timestamps, the bot will then see fit to place an order directly proportionate to that particular listing.

# 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

The bot first loops through the market and filters out the ask listings (Fig 2.1) :

```
/**go through diff currency ASK prices in the market*/
void Bot::loopThruExchangeAskPrices()
{
    for(std::string p : botCurrencies)
    {
        std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookType::ask, p, currentTime);
        for(int k=0; k<entries.size(); k++)
        {
            //if price lower than askSMA and trend of prices going down
            if(entries[k].price <= askSMA && negativeSlope(slopeAsk))
            {
                //if the current offer is better than prev one, then remove the prev order
                //so if there is a streak of prices going down, take the lowest once in the curve for maximum value
                if(entries[k].price < prevAskPrice)
                    orderBook.removeOrder();

                //enter bot bid based on the entries[k] data, exact match to listing in exchange
```

```
            enterBotBid(p, entries[k].price, entries[k].amount);
        }
        prevAskPrice = entries[k].price;
    }
  }
}
```

Fig 2.1

The bot gathers the asks in the exchange into a local vector, and then loops through the listing details. If there is a suitable listing, the bot will then take reference from the listing details and place an order based on the data in that particular listing for an exact match later on. This is to avoid any carry over into the next timestamp, and for an efficient and fast matching process later on.

---

- R2B: Pass the bids to the exchange for matching

Once it has gone through the many ask listings in the market, it checks for two conditions before placing an bid:

- Whether the current OrderBookEntry price is lesser than (ask SMA)
- Whether the slope gradient indicates a downward trend (LR)

If both conditions are true, the bot then places a bid listing through the enterBotBid() function, which is similar to the enterBid() function in the original MerkelMain.cpp file. The bot will also print out the details of the order it is placing to the console, for e.g *Buying 4 BTC for 38071 USDT*. (Fig 2.2):

```
/**enter bot's BID in the exchange*/
void Bot::enterBotBid(std::string product, double price, double amount)
{
    std::vector<std::string> splitProductName = CSVReader::tokenise(product, '/');
    std::string outgoingCurr = splitProductName[0];
    std::string incomingCurr = splitProductName[1];

    OrderBookEntry obe = CSVReader::stringsToOBE(std::to_string(price),std::to_string(amount), currentTime,
product, OrderBookType::bid);
    obe.username = "simuser";

    if (wallet.canFulfillOrder(obe))
    {
        std::cout << "Buying " << amount << " "<< outgoingCurr << " for: " << price*amount << incomingCurr <<
std::endl;
```

```
            orderBook.insertOrder(obe); //wont sort yet (moved to a diff function)
            //data log for bot bids placed
            historicalDataBidsPlaced.push_back(obe);
        }
        else
            std::cout << "Bot::wallet has insufficient funds to place BID. " << std::endl;
}
```

Fig 2.2

The function passes in the necessary parameters to create a new OrderBookEntry of type bid in regards to that particular ask in the market. The funds are then sent and deducted through the wallet's canFulfillOrder() function. The console then prints out a statement indicating the amount of currency being bought at said price.

---

- 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

At the end of looping through all the exchange listings in the timestamp, the bot will then proceed to match all appropriate pairings of asks and bids in the listings based on the orders. When there is a successful pair, meaning that the sale is successful, the bot will print out the details of the sale to the console. The wallet's balance will then be adjusted accordingly as well. The matching function is shown below in (Fig 2.3):

```
/**MerkelMain's gotoNextTimeFrame*/
void Bot::nextTimeframe()
{
    //removing the p:products part from Merkel
    //only one currency in it for now
    for (std::string p : botCurrencies)
    {
        std::cout << "matching " << p << std::endl;
        std::vector<OrderBookEntry> sales =  orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;
        for (OrderBookEntry& sale : sales)
        {
            std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
            if (sale.username == "simuser")
            {
```

```
            if(wallet.processSale(sale)) //bool
            {
                std::cout<<"Sale successful."<<std::endl;
                //data log for sales
                historicalSales.push_back(sale);
            }
        }
    }
    std::cout<<currentTime<<std::endl;
    walletSummary();
    std::cout<<"Going to next Timeframe::"<<std::endl;

std::cout<<"================================================================================
========\n"<<std::endl;
    currentTime = orderBook.getNextTime(currentTime);
}
```

Fig 2.3

---

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

As the bot loops through the different amount of prices, it has an additional checker in loopThruExchangeAskPrices():

```
//if the current offer is better than prev one, then remove the prev order
            //so if there is a streak of prices going down, take the lowest once in the curve for maximum value
            if(entries[k].price < prevAskPrice)
                orderBook.removeOrder();
```

This is to ensure that the bot is placing and order of better value in terms of temporal locality, in this case, the previous entry price in the exchange. If the current price is a better value than the previous one, then the bot will withdraw its previous order, and then place a new order with the current price, which is the better price. Having this additional condition has proved to yield better results than without it.

# 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

The bot first loops through the market and filters out the bid listings (Fig 3.1) :

```
/**go through diff currency BID prices in the market*/
void Bot::loopThruExchangeBidPrices()
{
    for(std::string p : botCurrencies)
    {
        std::vector<OrderBookEntry> entries = orderBook.getOrders(OrderBookType::bid, p, currentTime);
        for(int k=0; k<entries.size(); k++)
        {
            //if price greater than bidSMA and trend of prices going up
            if(entries[k].price >= bidSMA && positiveSlope(slopeBid))
            {
                //if the current offer is better than prev one, then remove the prev order
                //so if there is a streak of prices going down, take the highest once in the curve for maximum profits
                if(entries[k].price > prevBidPrice)
                    orderBook.removeOrder();

                //enter bot ask based on the entries[k] data, exact match to listing in exchange
                enterBotAsk(p, entries[k].price, entries[k].amount);
            }
            prevBidPrice = entries[k].price;
        }
    }
}
```

Fig 3.1

The bot gathers the *bids* in the exchange into a local vector, and then loops through the listing details. If there is a suitable listing, the bot will then take reference from the listing details and place an order based on the data in that particular listing for an exact match later on. This is to avoid any carry over into the next timestamp, and for an efficient and fast matching process later on.

---

- R3B: Pass the bids to the exchange for matching

Once it has gone through the many bid listings in the market, it checks for two conditions before placing an ask:

- Whether the current OrderBookEntry price is greater than (bid SMA)
- Whether the slope gradient indicates an upward trend (LR)

If both conditions are true, the bot then places an ask listing through the enterBotAsk() function, which is similar to the enterAsk() function in the original MerkelMain.cpp file. The bot will also print out the details of the order it is placing to the console, for e.g *Selling 4 BTC for 38071 USDT*. (Fig 3.2):

```
/**enter bot's ASK in the exchange*/
void Bot::enterBotAsk(std::string product, double price, double amount)
{
    std::vector<std::string> splitProductName = CSVReader::tokenise(product, '/');
    std::string outgoingCurr = splitProductName[0];
    std::string incomingCurr = splitProductName[1];

    OrderBookEntry obe = CSVReader::stringsToOBE(std::to_string(price),std::to_string(amount), currentTime,
product, OrderBookType::ask);
    obe.username = "simuser";

    if (wallet.canFulfillOrder(obe))
    {
        std::cout << "Selling " << amount << " "<< outgoingCurr << " for: " << price*amount << incomingCurr <<
std::endl;
        orderBook.insertOrder(obe); //wont sort yet (moved to a diff function)
        //data log for bot asks placed
        historicalDataAsksPlaced.push_back(obe);

    }
    else
        std::cout << "Bot::wallet has insufficient funds to place ASK. " << std::endl;
}
```

Fig 3.2

The function passes in the necessary parameters to create a new OrderBookEntry of type *ask* in regards to that particular *bid* in the market. The funds are then sent and deducted through the wallet's canFulfillOrder() function. The console then prints out a statement indicating the amount of currency being sold at said price.

- 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.

At the end of looping through all the exchange listings in the timestamp, the bot will then proceed to match all appropriate pairings of asks and bids in the listings based on the orders. When there is a successful pair, meaning that the sale is successful, the bot will print out the details of the sale to the console. The wallet's balance will then be adjusted accordingly as well. The matching function is shown below in (Fig 2.3):

```
/**MerkelMain's gotoNextTimeFrame*/
void Bot::nextTimeframe()
{
    //removing the p:products part from Merkel
    //only one currency in it for now
    for (std::string p : botCurrencies)
    {
        std::cout << "matching " << p << std::endl;
        std::vector<OrderBookEntry> sales =  orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;
        for (OrderBookEntry& sale : sales)
        {
            std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
            if (sale.username == "simuser")
            {
                if(wallet.processSale(sale)) //bool
                {
                    std::cout<<"Sale successful."<<std::endl;
                    //data log for sales
                    historicalSales.push_back(sale);
                }
            }
        }
    }
    std::cout<<currentTime<<std::endl;
    walletSummary();
    std::cout<<"Going to next Timeframe::"<<std::endl;

std::cout<<"================================================================================
========\n"<<std::endl;
```

```
    currentTime = orderBook.getNextTime(currentTime);
}
```

Fig 2.3

---

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

As the bot loops through the different amount of prices, it has an additional checker in loopThruExchangeBidPrices():

```
//if the current offer is better than prev one, then remove the prev order
//so if there is a streak of prices going down, take the highest once in the curve for maximum profits
            if(entries[k].price > prevBidPrice)
                orderBook.removeOrder();
```

This is to ensure that the bot is placing an order of better value in terms of temporal locality; In this case, the previous entry price in the exchange. If the current price is a better value than the previous one, then the bot will withdraw its previous order, and then place a new order with the current price, which is the better price. Having this additional condition has proved to yield better results than without it.

# R4: Logging

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

After going through every single listing in the market for each timestamp, the bot prints out the wallet's assets to the console, displaying the current amount of currencies it has through a function walletSummary(). This function can be found near the end of the nextTimeframe() function. When bids/asks are placed by the bot, the funds are subsequently deducted from the wallet, and printed out at the end of that timestamp (Fig 4.1):

```
/**MerkelMain's gotoNextTimeFrame*/
void Bot::nextTimeframe()
{
    //removing the p:products part from Merkel
    //only one currency in it for now
    for (std::string p : botCurrencies)
    {
        std::cout << "matching " << p << std::endl;
        std::vector<OrderBookEntry> sales =  orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;
        for (OrderBookEntry& sale : sales)
        {
            std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
            if (sale.username == "simuser")
            {
                if(wallet.processSale(sale)) //bool
                {
                    std::cout<<"Sale successful."<<std::endl;
                    //data log for sales
                    historicalSales.push_back(sale);
                }
            }
        }
    }
        std::cout<<currentTime<<std::endl;
        //data record of wallet's balance over time
    walletSummary();
    std::cout<<"Going to next Timeframe::"<<std::endl;
```

```
std::cout<<"================================================================================
========\n"<<std::endl;
    currentTime = orderBook.getNextTime(currentTime);
}
```

Fig 4.1 (walletSummary() highlighted in blue)

---

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

The bot class has 3 instance vectors variables that stores all of the listed past asks, bids and sales (Fig 4.2):

```
//LOG

//vectors to store transaction histories
std::vector<OrderBookEntry> historicalDataAsksPlaced, historicalDataBidsPlaced, historicalSales;
```

Fig 4.2

At the end of every ask and bid placed by the bot, that particular listing will be push backed (as per Fig 2.2 and Fig 3.2) into the respective vector variables shown in (Fig 4.2). When the bot reaches the end of its runtime, it prompts the user for a print out of all the transaction logs, and if the user agrees, the function log() in Bot will print out all the transaction logs to a new file.

The following function writes out the data stored in the historicalDataAsksPlace vector to a new file (Fig 4.3):

```
/**prints out all asks placed by bot to new file*/
void Bot::printHistoricalDataAskFile()
{
    //writing to file for all asks placed by bot
    ofstream historicalDataAskfile;
    historicalDataAskfile.open("historicalDataAsksPlaced");

    //write to file
    historicalDataAskfile<<"Asks placed: \n" ;
    historicalDataAskfile<<"PRODUCT | ASKS | TIMESTAMP              |PRICE      |AMOUNT\n";
    for(OrderBookEntry& e : historicalDataAsksPlaced)
```

```
    {
        historicalDataAskfile << e.toString() + "\n";
        std::cout<< e.toString() <<std::endl;
    }
    historicalDataAskfile.close();
}
```

Fig 4.3a

The following function writes out the data stored in the historicalDataBidsPlace vector to a new file (Fig 4.3):

```
/**prints out all bids placed by bot to new file*/
void Bot::printHistoricalDataBidFile()
{
    //writing to file for all bids placed by bot
    ofstream historicalDataBidfile;
    historicalDataBidfile.open ("historicalDataBidsPlaced");

    //write to file
    historicalDataBidfile<<"Bids placed: \n";
    historicalDataBidfile<<"PRODUCT | BIDS | TIMESTAMP              |PRICE     |AMOUNT\n";
    for(OrderBookEntry& e : historicalDataBidsPlaced)
    {
        historicalDataBidfile << e.toString() + "\n";
        std::cout<< e.toString() <<std::endl;
    }
    historicalDataBidfile.close();
}
```

Fig 4.3b

---

- 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

Whenever the bot goes to the next timestamp via the nextTimeframe() function, it pushes back all of the successful sales made into the vector historicalSales (Fig 4.4):

```
void Bot::nextTimeframe()
{
    for (std::string p : botCurrencies)
    {
        std::cout << "matching " << p << std::endl;
        std::vector<OrderBookEntry> sales =  orderBook.matchAsksToBids(p, currentTime);
        std::cout << "Sales: " << sales.size() << std::endl;
        for (OrderBookEntry& sale : sales)
        {
            std::cout << "Sale price: " << sale.price << " amount " << sale.amount << std::endl;
            if (sale.username == "simuser")
            {
                if(wallet.processSale(sale)) //bool
                {
                    historicalSales.push_back(sale); //data log
                }
            }
        }
    }

std::cout<<"================================================================================
========"<<std::endl;
    currentTime = orderBook.getNextTime(currentTime);
}
```

Fig 4.4

Then, in the function printHistoricalSalesFile(), it writes all of the successful sales data including the average ask/bid sale price to a new file (Fig 4.5):

```
/**prints out all sales and average successful sale price for asks/bids placed by bot to new file*/
void Bot::printHistoricalSalesFile()
{
    //writing to file for all successful sales
    ofstream historicalSalesFile;
    historicalSalesFile.open ("historicalSales");
```

```
    //write to file
    historicalSalesFile<<"Succesful sales: \n";
    historicalSalesFile<<"Average asksale price: " + std::to_string(getAvgAskSalePrice()) + "\n";
    historicalSalesFile<<"Average bidsale price: " + std::to_string(getAvgBidSalePrice()) + "\n";
    historicalSalesFile<<"PRODUCT |ORDERTYPE |TIMESTAMP                |PRICE      |AMOUNT\n";

    for(OrderBookEntry& e : historicalSales)
    {
        historicalSalesFile << e.toString() + "\n";
        std::cout<< e.toString() <<std::endl;
    }
    historicalSalesFile.close();
}
```

Fig 4.5

It calls in two helper functions to get the avgAskSalePrice and avgAskBidPrice (Fig 4.6):

```
/**gets average successful ask sale price*/
double Bot::getAvgAskSalePrice()
{
    double sumAskSalePrice, sumAskCounter;
    for(OrderBookEntry&e : historicalSales)
    {
        if(e.orderType == OrderBookType::asksale)
        {
            sumAskSalePrice += e.price;
            sumAskCounter++;
        }
    }
    return sumAskSalePrice / sumAskCounter;
}
```

Fig 4.6a (get the average ask sale price)

```
/**gets average successful bid sale price*/
double Bot::getAvgBidSalePrice()
{
    double sumBidSalePrice, sumBidCounter;
    for(OrderBookEntry&e : historicalSales)
```

```
    {
        if(e.orderType == OrderBookType::bidsale)
        {
            sumBidSalePrice += e.price;
            sumBidCounter++;
        }
    }
    return sumBidSalePrice / sumBidCounter;
}
```

4.6b (get the average bid sale price)

# Challenge Requirement

In the OrderBook class, there is a sort function in the insertOrders() function. With a large dataset, every time an order is inserted into the orderBook, the sort will be called and it will be extremely time consuming. The sort function is not necessary in the trading process, as it simply reorganises the orders by their timestamps. My bot enters the specific amount for any bid or ask listing in the exchange, hence there will be no carry over to next timestamps and as such, the timestamp sort is rendered redundant in my case.

Furthermore, during the matchAsksToBids() function in the orderBook, the orders will be sorted based on their prices, and that is where the order of the orders are more significant for a higher matching rate.

```
void OrderBook::insertOrder(OrderBookEntry& order)
{
    orders.push_back(order);
    //commented out the sort function here
    // std::sort(orders.begin(), orders.end(), OrderBookEntry::compareByTimestamp);
}
```

In the case of my bot's withdrawing of bids and asks, it checks for the previous listing prices to get a better value of the trade. The removeOrder() function in OrderBook simply pops back the last order if that is the case, and this is only possible without the previous sort function as mentioned (if not the order inserted will not be at the back). Hence rather than sorting the orders consistently after every single insertOrder() called, I have shifted the sort to the end of the bot instead.

For the purpose of testing, I have set a fixed loop counter of 434 times (which is the amount of times needed to make the 0.01% profit).

Before removing the sort function, the total time taken is 907 seconds:

```
Going to next Timeframe::
================================================================================

Profit of 0.1% has been achieved. Bot has finished running.

Total time taken: 907 seconds.
Wallet's balance:
BTC : 14.872918
USDT : 1402.998515

Print out transaction logs?
Press y to continue, n to exit.
```

After making the changes mentioned above, the total time has significantly reduced to 124 seconds:

```
Going to next Timeframe::
================================================================================

Profit of 0.1% has been achieved. Bot has finished running.

Total time taken: 124seconds.
Wallet's balance:
BTC : 15.018624
USDT : 45.674962

Print out transaction logs?
Press y to continue, n to exit.
```

To log the time taken, I have used the std::chrono library functions to get the starting time and ending time during the bot's execution and calculated the difference as shown:

```
//keep track of time taken for the bot
auto startTime = std::chrono::system_clock::now();
auto endTime = std::chrono::system_clock::now();
auto timeTaken = (std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime).count() / 1000);


std::cout<<"Total time taken: " << timeTaken << " seconds." << std::endl;
```

*The following code was referenced from McLeary's Github repository on C++ timer using std::chrono*

---

Additionally, in my bot's marketAnalysis() function, I am looping through the exchange asks and then bids respectively, which is a very linear approach (highlighted in blue):

```
void Bot::marketAnalysis()
{
    auto startTime = std::chrono::system_clock::now();
    int counter=0;
    int walletAmount = wallet.getCurrencyAmount("BTC");


        //loop until profit has reached
        while(!wallet.containsCurrency("BTC", walletAmount*profitPercent)) //0.01% increase (benchmark)
        {
                //keep track of how many loops
                std::cout<<"Counter: "<<counter<<std::endl;
                analysis();
                //gather data till enough for analysis
                if(counter>windowSize)
                {
                    loopThruExchangeAskPrices();
                    loopThruExchangeBidPrices();
```

```
            }
            nextTimeframe();
            counter++;
        }
        //only sort orders at the end based on timestamps
        orderBook.sortOrders();
        std::cout<<"Profit of " << (profitPercent-1)*100 << "% has been achieved. Bot has finished running.
\n"<<std::endl;
}
```

A consideration that I took upon was introducing multi-threading functions using the std::thread library in C++, such that the two functions will be able to run simultaneously, rather than consecutively.

## References


- Codesansar, Linear Regression Method Pseudocode, viewed Dec 25 2020, https://www.codesansar.com/numerical-methods/linear-regression-method-pseudocode.htm

- Cassio Neri, C++ implementation (Sep 2013), viewed Dec 25 2020, https://stackoverflow.com/questions/18939869/how-to-get-the-slope-of-a-linear-regression-line-using-c

- Samgak, Calculate moving average over an array of data (May 2015), viewed Dec 25 2020, https://stackoverflow.com/questions/30338671/i-tried-coding-my-own-simple-moving-average-in-c

- DayTrading.com, Technical Analysis: A Primer; Linear Regression Line, viewed Dec 25 2020, https://www.daytrading.com/linear-regression-line

- Yakko Majuri, Aug 2020, A step-by-step guide to building a trading bot in any language, https://blog.usejournal.com/a-step-by-step-guide-to-building-a-trading-bot-in-any-programming-language-d202ffe91569

- TD Ameritade, Dec 2020, How to use Moving Averages for Stock Trading, viewed Dec 25 2020, https://www.youtube.com/watch?v=r3Ulu0jZCJI&list=PLtjob7NnSme-CaJm8UJVvD9mEr3ABWx6p&index=3

- McLeary, Mar 2016, C++ timer using std::chrono - GitHub, viewed 10 Jan 2021, https://gist.github.com/mcleary/b0bf4fa88830ff7c882d