Requirements:

R1A:

In this requirement we are getting the information from a csv document. We brought the information through document dealing with to perform further activities on it. The information put away in the csv record is a time series data with respect to offers and requests bitcoin costs and amount.

R1B:

To forecast the BTC's future values, the linear regression model is used. So that, according to the future values, we may make an offer or sell BTC. Via time series analysis, the future value prediction was performed. Taking values of a particular size of the window and training the model to predict the sequence's next upcoming values.

R2A:

We will make a purchase when the forecast value for the future goes up. To forecast the future values, we used the above-mentioned linear regression model. If the graph/ future expected values are rising, the bot asks to bid.

R2B:

If the graphs go up in the future forecast, you are asked to enter bids for the exchange that will verify the availability of wallet numbers.

R2C:

It passes through the wallet and BTC quantity availability after entering the bid values. It will verify all items and proceed to purchase if all specifications are fully met.

R2D:

R1B and R2C met this requirement as we forecast the future values first and check anything required for it after entering the quantity and quantity. If anything is available, it continues otherwise that the transaction is not made.

R3A:

The future prediction algorithm in which we can forecast the future value and then produce the asking value according to it also fulfills this requirement.

R3B:

After receiving the submission, it is submitted for the further phase in the MerkleMain.

R3C:

It went through the complete process, including the wallet and orderbook, after obtaining the asked values to verify whether further processing is feasible or not. If all of the conditions are met, we will continue with this action.

R3D:

R1B and R2C met this requirement as we forecast the future values first and check anything required for it after entering the quantity and quantity. If everything is open, it continues otherwise it does not make any sales.

R4A:

All data is stored in the csv format and the amount, price and type of BTC time sequence is appended to the incoming data.

R4B:

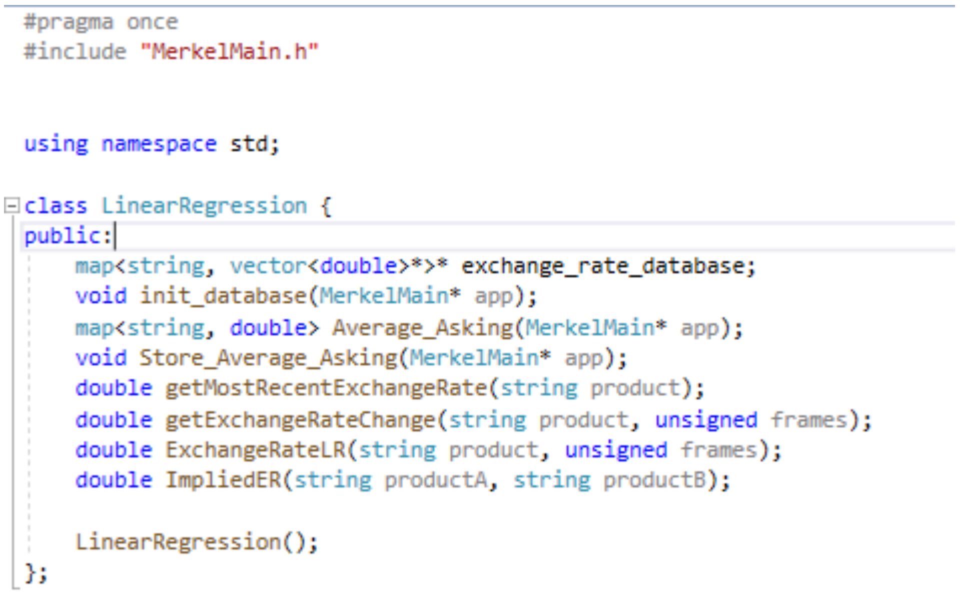
Both Buying and Selling had different files that held all behavior that happened to them.

R4C:

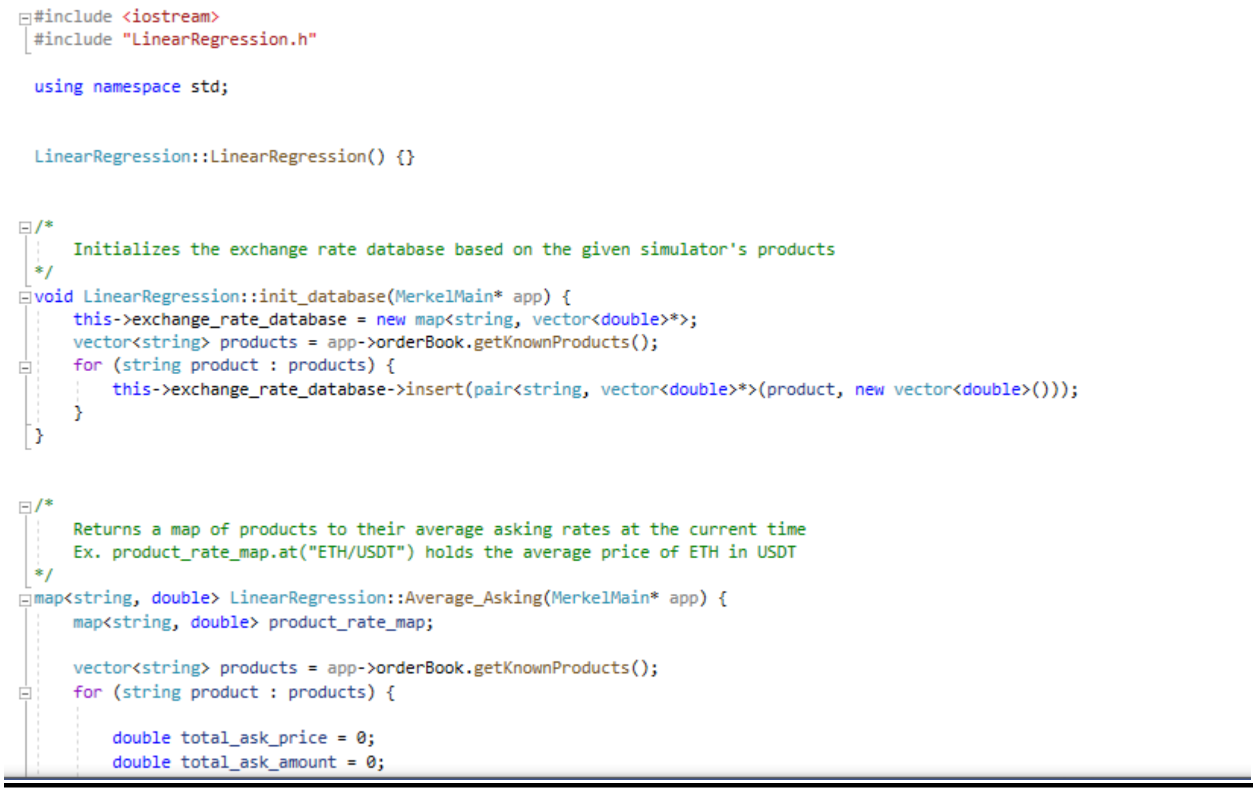
Only successful bids and requests are stored in both the purchase and request of the csv file.

Algorithms:

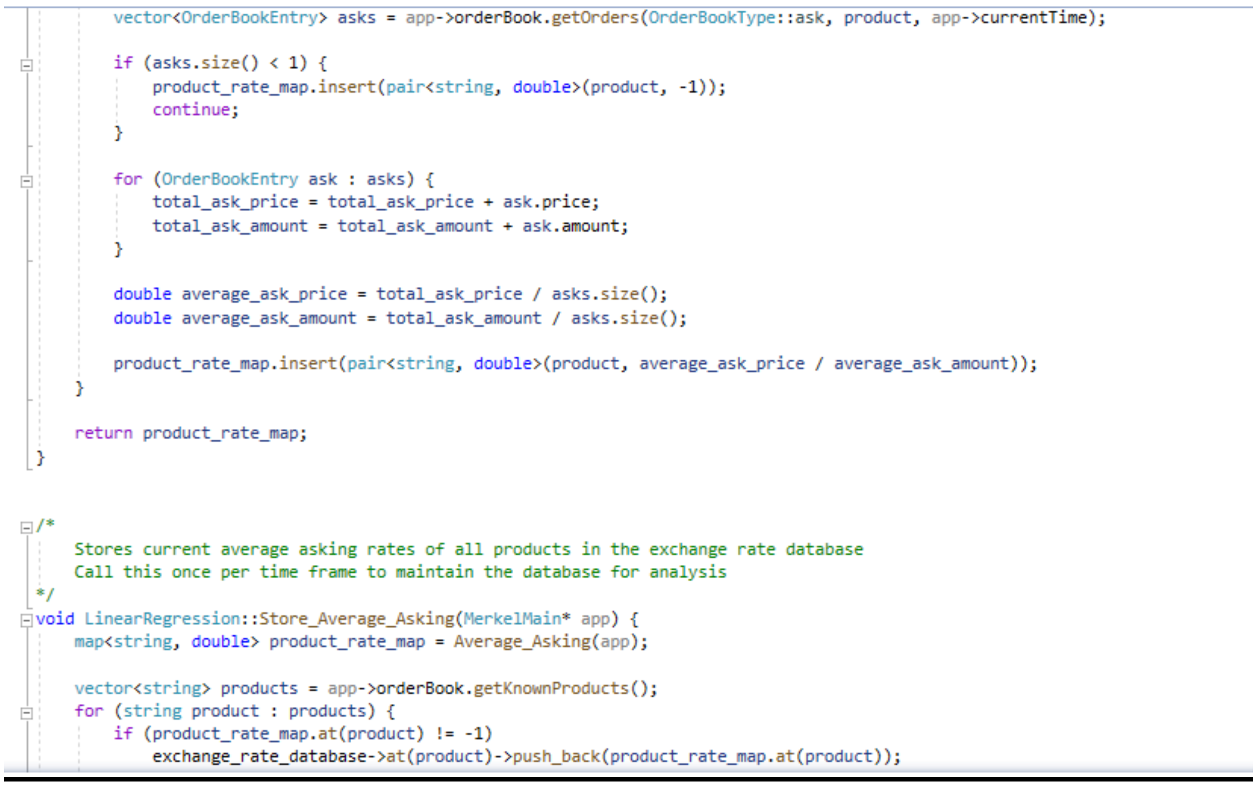
Linear Regression:



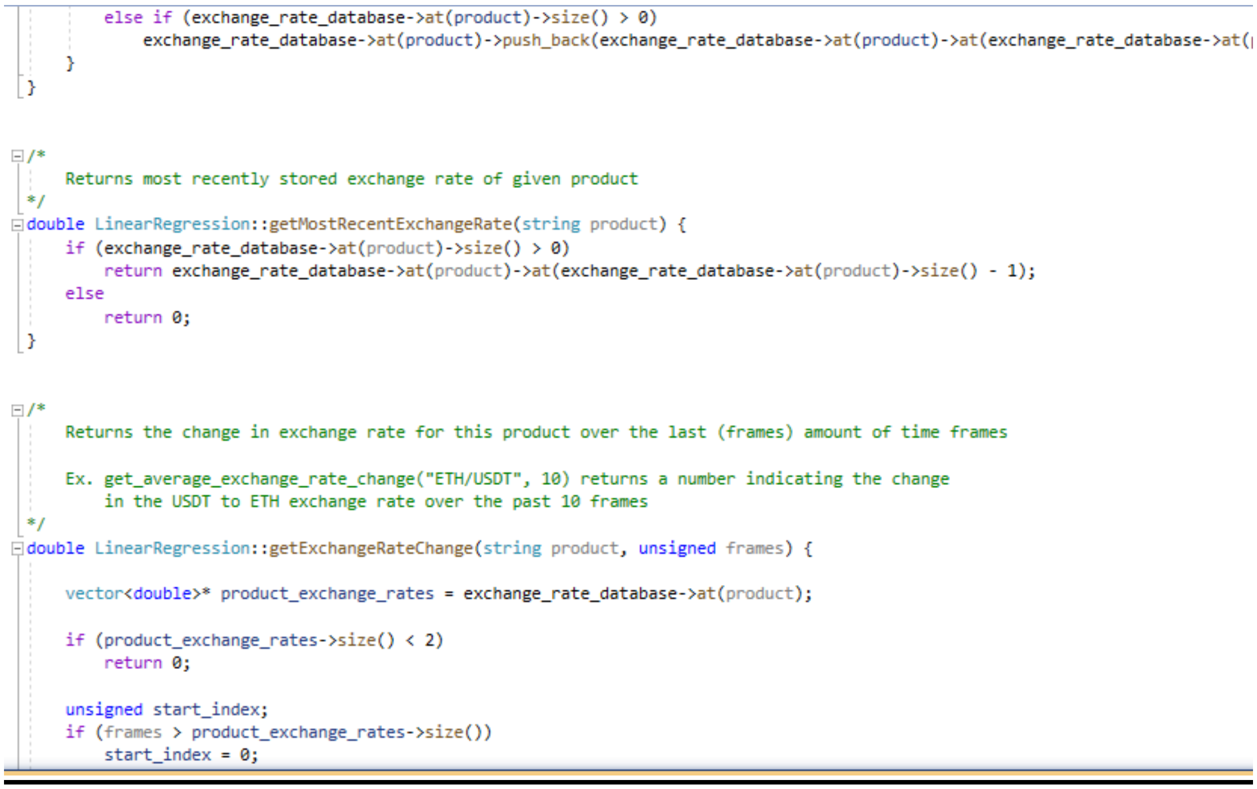
```
#pragma once
#include "MerkelMain.h"


using namespace std;

class LinearRegression {
public:
    map<string, vector<double>*>* exchange_rate_database;
    void init_database(MerkelMain* app);
    map<string, double> Average_Asking(MerkelMain* app);
    void Store_Average_Asking(MerkelMain* app);
    double getMostRecentExchangeRate(string product);
    double getExchangeRateChange(string product, unsigned frames);
    double ExchangeRateLR(string product, unsigned frames);
    double ImpliedER(string productA, string productB);

    LinearRegression();
};
```



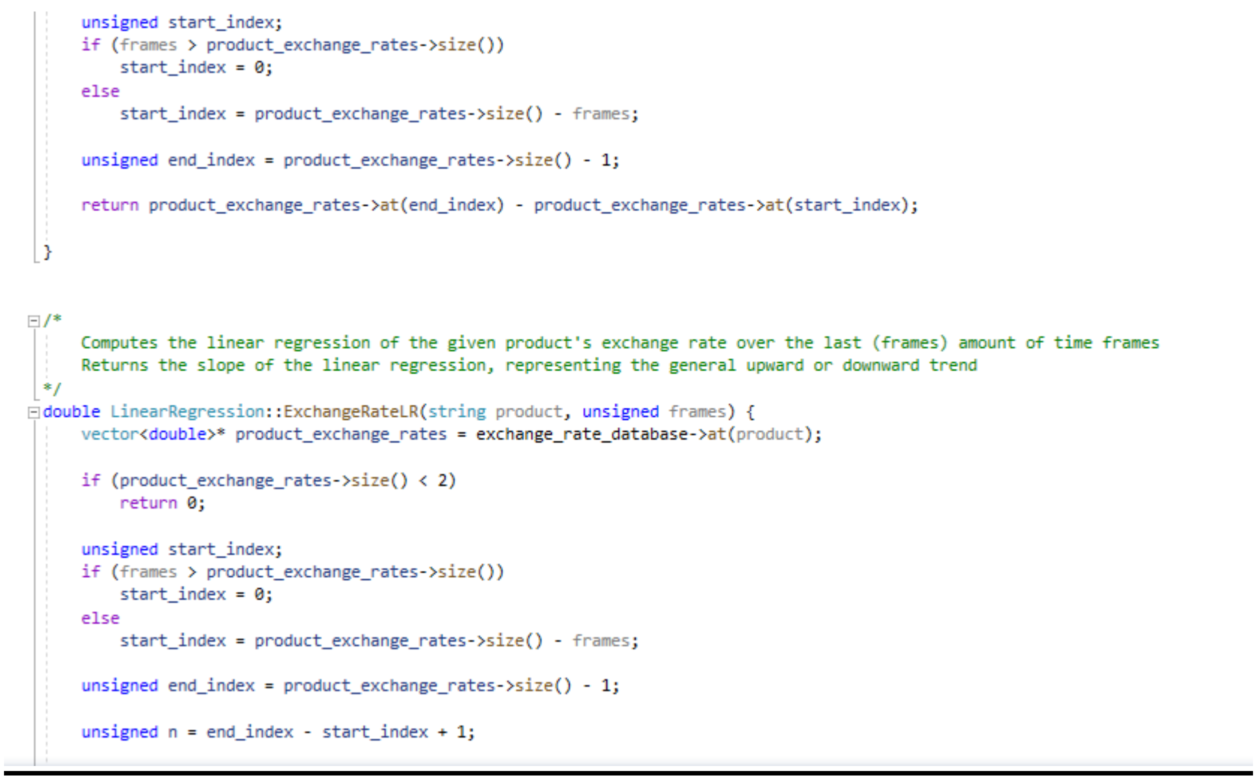
```
#include <iostream>
#include "LinearRegression.h"

using namespace std;


LinearRegression::LinearRegression() {}


/*
    Initializes the exchange rate database based on the given simulator's products
*/
void LinearRegression::init_database(MerkelMain* app) {
    this->exchange_rate_database = new map<string, vector<double>*>;
    vector<string> products = app->orderBook.getKnownProducts();
    for (string product : products) {
        this->exchange_rate_database->insert(pair<string, vector<double>*>(product, new vector<double>()));
    }
}


/*
    Returns a map of products to their average asking rates at the current time
    Ex. product_rate_map.at("ETH/USDT") holds the average price of ETH in USDT
*/
map<string, double> LinearRegression::Average_Asking(MerkelMain* app) {
    map<string, double> product_rate_map;

    vector<string> products = app->orderBook.getKnownProducts();
    for (string product : products) {

        double total_ask_price = 0;
        double total_ask_amount = 0;
```



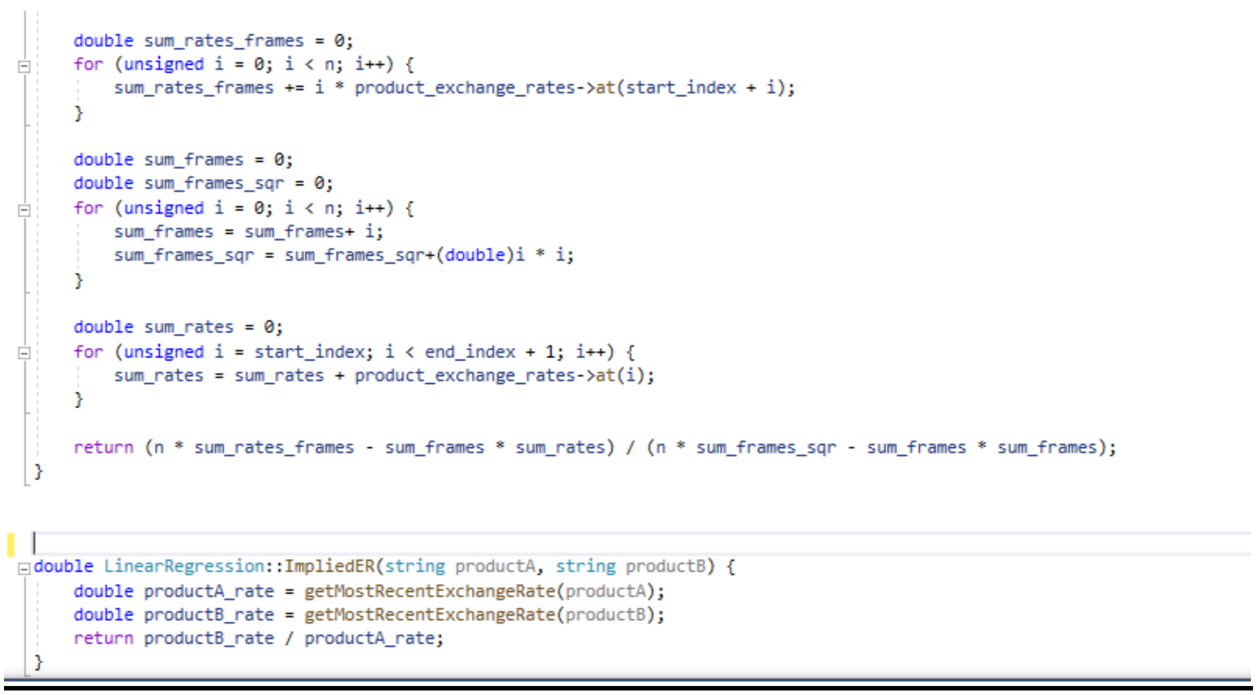
```cpp
        vector<OrderBookEntry> asks = app->orderBook.getOrders(OrderBookType::ask, product, app->currentTime);

        if (asks.size() < 1) {
            product_rate_map.insert(pair<string, double>(product, -1));
            continue;
        }

        for (OrderBookEntry ask : asks) {
            total_ask_price = total_ask_price + ask.price;
            total_ask_amount = total_ask_amount + ask.amount;
        }

        double average_ask_price = total_ask_price / asks.size();
        double average_ask_amount = total_ask_amount / asks.size();

        product_rate_map.insert(pair<string, double>(product, average_ask_price / average_ask_amount));
    }

    return product_rate_map;
}


/*
    Stores current average asking rates of all products in the exchange rate database
    Call this once per time frame to maintain the database for analysis
*/
void LinearRegression::Store_Average_Asking(MerkelMain* app) {
    map<string, double> product_rate_map = Average_Asking(app);

    vector<string> products = app->orderBook.getKnownProducts();
    for (string product : products) {
        if (product_rate_map.at(product) != -1)
            exchange_rate_database->at(product)->push_back(product_rate_map.at(product));
```



```cpp
        else if (exchange_rate_database->at(product)->size() > 0)
            exchange_rate_database->at(product)->push_back(exchange_rate_database->at(product)->at(exchange_rate_database->at(
    }
}


/*
    Returns most recently stored exchange rate of given product
*/
double LinearRegression::getMostRecentExchangeRate(string product) {
    if (exchange_rate_database->at(product)->size() > 0)
        return exchange_rate_database->at(product)->at(exchange_rate_database->at(product)->size() - 1);
    else
        return 0;
}


/*
    Returns the change in exchange rate for this product over the last (frames) amount of time frames

    Ex. get_average_exchange_rate_change("ETH/USDT", 10) returns a number indicating the change
        in the USDT to ETH exchange rate over the past 10 frames
*/
double LinearRegression::getExchangeRateChange(string product, unsigned frames) {

    vector<double>* product_exchange_rates = exchange_rate_database->at(product);

    if (product_exchange_rates->size() < 2)
        return 0;

    unsigned start_index;
    if (frames > product_exchange_rates->size())
        start_index = 0;
```



```cpp
    unsigned start_index;
    if (frames > product_exchange_rates->size())
        start_index = 0;
    else
        start_index = product_exchange_rates->size() - frames;

    unsigned end_index = product_exchange_rates->size() - 1;

    return product_exchange_rates->at(end_index) - product_exchange_rates->at(start_index);

}


/*
    Computes the linear regression of the given product's exchange rate over the last (frames) amount of time frames
    Returns the slope of the linear regression, representing the general upward or downward trend
*/
double LinearRegression::ExchangeRateLR(string product, unsigned frames) {
    vector<double>* product_exchange_rates = exchange_rate_database->at(product);

    if (product_exchange_rates->size() < 2)
        return 0;

    unsigned start_index;
    if (frames > product_exchange_rates->size())
        start_index = 0;
    else
        start_index = product_exchange_rates->size() - frames;

    unsigned end_index = product_exchange_rates->size() - 1;

    unsigned n = end_index - start_index + 1;
```



```cpp
    double sum_rates_frames = 0;
    for (unsigned i = 0; i < n; i++) {
        sum_rates_frames += i * product_exchange_rates->at(start_index + i);
    }

    double sum_frames = 0;
    double sum_frames_sqr = 0;
    for (unsigned i = 0; i < n; i++) {
        sum_frames = sum_frames+ i;
        sum_frames_sqr = sum_frames_sqr+(double)i * i;
    }

    double sum_rates = 0;
    for (unsigned i = start_index; i < end_index + 1; i++) {
        sum_rates = sum_rates + product_exchange_rates->at(i);
    }

    return (n * sum_rates_frames - sum_frames * sum_rates) / (n * sum_frames_sqr - sum_frames * sum_frames);
}


double LinearRegression::ImpliedER(string productA, string productB) {
    double productA_rate = getMostRecentExchangeRate(productA);
    double productB_rate = getMostRecentExchangeRate(productB);
    return productB_rate / productA_rate;
}
```

Code Optimization:

The machine learning portion, the linear regression technique used to predict future values, is optimized to get the future prediction before any request and bid without any wait, so we will know how much to do after getting the future values before bidding and selling BTC.