STEP1

In step1, we implement the ReadNextMsg() function as shown below:

```cpp
bool ReadNextMsg(std::ifstream& file, Msg& msg) {
    if (file.eof()) {
        return false;
    }
    // TODO: your implementation to read file and create the next Msg into the variable msg

    std::string line;
    std::vector<TickData> updates;
    uint64_t lastUpdateTimeStamp = 0;
    int counter = 0;
    msg.isSet = true;

    while (std::getline(file, line)) {
        std::istringstream iss(line);
        std::string field;

        // Parse the CSV line and store each field in the 'field' variable
        std::vector<std::string> fields;
        while (std::getline(iss, field, ',')) {
            fields.push_back(field);
        }

        // Discard first row (headers)
        if (fields[0] == "contractName") {
            continue;
        }

        // Discard any data before the first snapshot
        if (fields[2] == "update" && msg.timestamp == 0) {
            continue;
        }

        TickData update;
        update.LastUpdateTimeStamp = TimeToUnixMS(fields[1]);

        // If the current timestamp is different from the previous one, return true
        if (update.LastUpdateTimeStamp != lastUpdateTimeStamp && counter >= 1) {
            msg.timestamp = update.LastUpdateTimeStamp;
            msg.Updates = updates;
            return true;
        }

        // Set the timestamp for the message
        lastUpdateTimeStamp = update.LastUpdateTimeStamp;
        msg.timestamp = update.LastUpdateTimeStamp;

        // Determine if it's a snapshot
        msg.isSnap = (fields[2] == "snap");

        // Store data inside the TickData structure
        update.ContractName = fields[0];
        update.BestBidPrice = fields[4].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[4]);
        update.BestBidAmount = fields[5].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[5]);
        update.BestBidIV = fields[6].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[6]);
        update.BestAskPrice = fields[7].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[7]);
        update.BestAskAmount = fields[8].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[8]);
        update.BestAskIV = fields[9].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[9]);
        update.MarkPrice = fields[10].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[10]);
        update.MarkIV = fields[11].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[11]);
        update.UnderlyingIndex = fields[12];
        update.UnderlyingPrice = fields[13].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[13]);
        update.LastPrice = fields[15].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[15]);
        update.OpenInterest = fields[16].empty() ? std::numeric_limits<double>::quiet_NaN() : std::stod(fields[16]);

        updates.push_back(update);
        counter++;
    }

    msg.timestamp = lastUpdateTimeStamp;
    msg.Updates = updates;

    return true;
}
```

Introduction:

The purpose of this report is to describe the implementation of the 'ReadNextMsg' function, which is responsible for reading a file and creating a message ('Msg') based on the data extracted from the file. The function is designed to process a CSV file containing tick data for financial contracts and extract relevant information to populate the 'Msg' structure.

Function Overview:

The 'ReadNextMsg' function takes two parameters: a reference to an 'std::ifstream' object representing the input file and a reference to a Msg object. It returns a boolean value indicating the success of reading the next message from the file. If successful, the function updates the 'Msg' object with the extracted data and returns 'true'. Otherwise, if the end of the file is reached, it returns 'false'.

Function Implementation:

1. The function first checks if the end of the file ('file.eof()') has been reached. If so, it returns 'false', indicating that there are no more messages to read from the file.
2. Inside the function, several variables are declared, including 'line' (string), 'updates' (vector of 'TickData' objects), 'lastUpdateTimeStamp' (uint64_t), and 'counter' (int). These variables will be used to store and manipulate the extracted data.
3. The 'isSet' member of the 'msg' object is set to true, indicating that the message is being populated.
4. The function enters a loop that reads each line from the file using 'std::getline'.
5. For each line, the line is parsed into individual fields by splitting it using a comma delimiter. The fields are stored in the 'fields' vector.
6. The first row (headers) is discarded by checking if the value of the first field is "contractName". If it is, the loop continues to the next iteration.
7. Any data before the first snapshot is discarded by checking if the value of the third field is "update" and if the 'timestamp' member of the 'msg' object is 0. If both conditions are true, the loop continues to the next iteration.
8. A new 'TickData' object, 'update', is created to store the extracted data for the current line.
9. The 'LastUpdateTimeStamp' member of the update object is set by converting the value of the second field to a Unix timestamp using the 'TimeToUnixMS' function.
10. If the current timestamp ('update.LastUpdateTimeStamp') is different from the previous one ('lastUpdateTimeStamp') and the counter ('counter') is greater than or equal to 1, it means that a new message is being started. In this case, the function updates the 'timestamp' and 'Updates' members of the 'msg' object with the collected data and returns 'true' to indicate a successful message extraction.
11. The 'lastUpdateTimeStamp' is updated with the current timestamp, and the 'timestamp' member of the 'msg' object is set to the current timestamp.
12. The 'isSnap' member of the 'msg' object is set based on the value of the second field ('fields[2]'). If it is "snap", the value is set to 'true'; otherwise, it is set to 'false'.
13. The remaining fields of the 'update' object are populated with the corresponding values from the 'fields' vector. Empty fields are handled by assigning

'std::numeric_limits<double>::quiet_NaN()' to indicate missing or unavailable data.

14. The 'update' object is added to the 'updates' vector.
15. The 'counter' is increased.

Example screenshot of the result of step1:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:49, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:49, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:50, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:51, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 2
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 2
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:52, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:53, isSnap = 0, numUpdates = 1
2022-05-16 07:59:54, isSnap = 0, numUpdates = 1
2022-05-16 07:59:54, isSnap = 0, numUpdates = 1
2022-05-16 07:59:54, isSnap = 0, numUpdates = 1
2022-05-16 07:59:54, isSnap = 0, numUpdates = 1
2022-05-16 07:59:54, isSnap = 0, numUpdates = 1
2022-05-16 07:59:54, isSnap = 0, numUpdates = 1
```

STEP2

In step2, we implement the Process function and the PrintInfo function as shown below:

```cpp
template <class Smile>
void VolSurfBuilder<Smile>::Process(const Msg &msg)
{
    // TODO (Step 2)
    if (msg.isSnap)
    {
        // discard currently maintained market snapshot, and construct a new copy based on the input Msg
        currentSurfaceRaw.clear();
        for (auto &ticker : msg.Updates)
        {
            currentSurfaceRaw[ticker.ContractName] = ticker;
        }
    }
    else
    {
        // update the currently maintained market snapshot
        for (auto &ticker : msg.Updates)
        {
            auto it = currentSurfaceRaw.find(ticker.ContractName);
            if (it != currentSurfaceRaw.end() && it->second.LastUpdateTimeStamp <= ticker.LastUpdateTimeStamp)
            {
                currentSurfaceRaw[ticker.ContractName] = ticker;
            }
            else
            {
                currentSurfaceRaw[ticker.ContractName] = ticker;
            }
        }
    }
}


template <class Smile>
void VolSurfBuilder<Smile>::PrintInfo() {
    // TODO (Step 2): you may print out information about VolSurfBuilder's currentSnapshot to test
    std::cout << "Number of contracts in current snapshot: " << currentSurfaceRaw.size() << std::endl;
    for (const auto& entry : currentSurfaceRaw) {
        const auto& [contractName, tickData] = entry;
        std::cout << contractName << "\t, Price= " << tickData.LastPrice << "\t, IV= " << tickData.MarkIV << std::endl;
    }
}

```

Process Method:

The Process method takes a Msg object as input and updates the market snapshot accordingly. The method follows these steps:

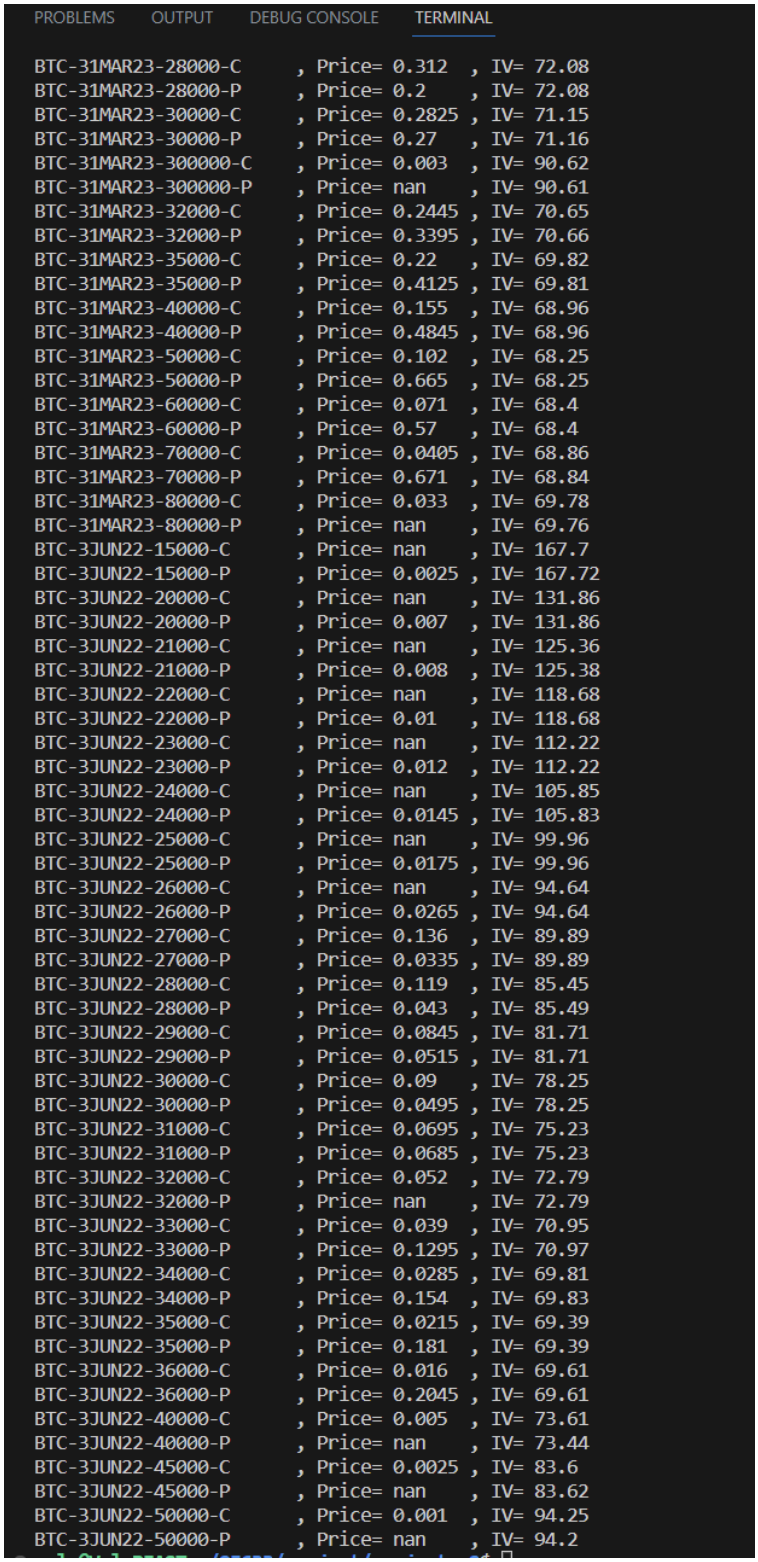
1. If the isSnap member of the input msg object is true, indicating a snapshot message, the current market snapshot is cleared (currentSurfaceRaw.clear()).
2. For each ticker in the Updates vector of the msg object, the ticker data is added or updated in the currentSurfaceRaw map.
    a. If the ticker already exists in the currentSurfaceRaw map and its LastUpdateTimeStamp is less than or equal to the LastUpdateTimeStamp of the current ticker, the existing ticker in the map is updated with the new ticker data.
    b. If the ticker does not exist in the currentSurfaceRaw map, it is added to the map with the ContractName as the key and the ticker object as the value.

PrintInfo Method:

The PrintInfo method is used to print information about the current market snapshot. The method performs the following actions:

1. It outputs the number of contracts present in the currentSurfaceRaw map using currentSurfaceRaw.size().

2. For each entry in the currentSurfaceRaw map, the contract name, last price, and implied volatility (IV) are printed.
    a. The contract name is retrieved from the map entry.
    b. The last price and IV are accessed from the tickData object stored in the map entry.
    c. The information is printed using std::cout, with appropriate formatting.



STEP3

In step3, there are roughly three parts of codes / functions to be implemented. We also add additional codes to make it compile smoothly.

Main works:

1. CubicSmile.cpp -> Extracts smile parameters by fitting a cubic smile to ticker data
2. VolSurfBuilder.h: FitSmiles -> Group data, fit the smile, calculate errors
3. step3.cpp: timer_listener -> Stream results to output file

Additional works:

1. Calculate the fitting error using time-based weights
2. Modify rfbrent function to avoid failure in searching roots

3.1 CubicSmile.cpp:

3.1.1 Get strikes and times

Before go into the function, there are some useful tools we can define to get strikes and expiry time. we have two functions: GetStrike and GetExpiryTime, which are responsible for extracting strike prices and converting expiry dates into time values, respectively.

The GetStrike function takes a contract name (cName) as input and extracts the strike price from it. It uses the position of the dashes (-) in the contract name to determine the substring that represents the strike price. By converting this substring to a floating-point number using std::stod, the function returns the extracted strike price.

The GetExpiryTime function takes an expiry date in the form of a string (expiry) and converts it into a time value. First, it splits the expiry string based on the dash character and extracts the relevant portion representing the date. Then, it determines the day, month, and year values by parsing the substring. The function uses a static map named months to convert the three-letter month abbreviation into its corresponding numerical representation.

After extracting the day, month, and year values, the function initializes a tm structure and assigns the extracted values accordingly. It sets the hour, minute, and second components to 0 and specifies that daylight saving time is not in effect (time.tm_isdst = 0). By calling mktime with the populated tm structure, the function converts the date and time information into a time value (time_t).

If any errors occur during the conversion process, such as an invalid expiry format or a failure in the mktime function, appropriate error handling is performed, including printing error messages and exiting the program if necessary.

3.1.2 CubicSmile.cpp: FitSmile:

The FitSmile function within the CubicSmile class is responsible for fitting a cubic smile to a set of tickers representing volatility data. The aim is to derive parameters that capture the shape of the volatility smile, including the at-the-money (ATM) volatility, butterfly (bf) spreads, and risk reversal (rr) spreads.

To begin, the code ensures that all tickers in the volTickerSnap vector have the same expiry and underlying asset. It identifies the latest ticker based on the LastUpdateTimeStamp and retrieves the underlying price (fwd) from that ticker.

Next, the time to expiry (T) is calculated using the difference between the expiry time and the current time, normalized to years. The code uses the GetExpiryTime function to convert the expiry date from a string to a time value, which is then multiplied by 1000 to convert it to milliseconds.

The code then proceeds to fit the parameters of the smile by iteratively solving for the implied volatility (iv) that matches the observed market prices of call options. It uses the impliedVol function, which takes into account the strike price, forward price, time to expiry, and observed option price.

To derive the other smile parameters, various strikes are calculated based on quick deltas (quickDeltaToStrike function) at specific levels, such as 90, 75, 25, and 10. Implied volatilities are then determined for these strikes using the impliedVol function with put options as well.

From these implied volatilities, the butterfly and risk reversal spreads are calculated. The butterfly spread (bf25 and bf10) represents the average difference between the implied volatilities of options at 25 and 75 delta and the ATM volatility. The risk reversal spread (rr25 and rr10) represents the difference between the implied volatilities of options at 25 and 75 delta.

By fitting the parameters and calculating these quantities, the function constructs a CubicSmile object with the resulting smile parameters (fwd, T, atmvol, bf25, rr25, bf10, rr10) and returns it.

From a financial perspective, this process is crucial for capturing the implied volatility smile, which characterizes the market's expectation of future volatility. The ATM volatility represents the baseline level of volatility, while the butterfly and risk reversal spreads provide insights into the skewness and kurtosis of the volatility distribution, respectively. These parameters are essential for pricing and hedging derivative instruments, as they capture the market sentiment and dynamics related to option prices at different strikes.

3.2 VolSurfBuilder.h: FitSmiles:

The Volatility Surface Builder is designed to group market tick data by expiry date, fit a smile model to the market data, and calculate the fitting error. The fitting error is defined as the weighted squared error between the mid implied volatility of the contracts and the model volatility at the corresponding strike. The key functionality of the Volatility Surface Builder is implemented in the FitSmiles function

For additional codes added in VolSurfBuilder class. The months map is used to map three-letter month abbreviations to their corresponding integer values. It provides a convenient

lookup mechanism for converting month abbreviations to their numeric representations. By using this map, the code can easily associate month abbreviations with their respective integers, allowing for better organization and readability in the code. The map is initialized with key-value pairs representing the 12 months of the year, with each key being a three-letter abbreviation (e.g., "JAN" for January) and the corresponding value being an integer from 1 to 12. This map enhances the code's clarity and maintainability by providing a centralized and reusable mapping for month abbreviations throughout the codebase.

3.2.1 Grouping Tickers by Expiry Date

The first step in the FitSmiles function is to group the tickers in the current market snapshot by expiry date. This is accomplished by constructing a std::map called tickersByExpiry, where the expiry date serves as the key and the tickers with the same expiry date are stored in a vector. The grouping is done using the ConvertExpiryToDate function, which converts the expiry date string from the ticker into a datetime_t value.

3.2.2 Fitting the Smile Model

Once the tickers are grouped by expiry date, the function proceeds to create a Smile instance for each expiry by calling the FitSmile function of the Smile model. The FitSmile function is a static function implemented in the Smile class and is responsible for fitting the smile model to the market data. The specific implementation of the FitSmile function depends on the type of smile model being used, and in this code snippet, it is assumed to be implemented in the CubicSmile class.

3.2.3 Calculating the Fitting Error

After obtaining the smile model for each expiry, the fitting error is calculated. We calculate the fitting error using time-based weights to evaluate the disparity between mean implied volatility and smile volatility. It iterates through a set of tickers, computing the mean implied volatility by averaging the best bid and best ask implied volatilities. The smile volatility is obtained from a fitted smile model using the ticker's strike value. Time-based weights are assigned to the tickers, giving higher weights to more recent data points. The fitting error is determined by summing the weighted squared differences between the mean implied volatility and smile volatility for each ticker. Normalization is applied by dividing the fitting error by the total weight, providing an overall measure of the fitting quality based on the tickers' implied volatilities.

For instance, suppose we have the following tickers: Ticker 1, Ticker 2, Ticker 3, while Ticker 1 is the latest ticker.To calculate the weights with normalization, we can follow these steps:

Step 1: Calculate the sum of the reciprocals of the indices:

Sum of reciprocals = 1/1 + 1/2 + 1/3 = 1.833

Step 2: Calculate the normalized weights by dividing each reciprocal by the sum:

Normalized weight for Ticker 1: (1/1) / 1.833 $\approx$ 0.545

Normalized weight for Ticker 2: (1/2) / 1.833 $\approx$ 0.273

Normalized weight for Ticker 3: (1/3) / 1.833 $\approx$ 0.182

3.2.4 Output and Logging

The results of the fitting process, including the fitting error and the parameters of the fitted smile model, are stored in a std::map called res. Each entry in the map represents a specific expiry date, and the associated value is a pair consisting of the fitted smile model and the corresponding fitting error. Additionally, the fitting error and other relevant information are logged in the console using the std::cout function.

3.2.5 Additional Utility Functions

The provided code also includes two utility functions: UnixMSToTime and DateToTime. These functions are used to convert Unix timestamps and datetime_t values to formatted time strings for logging and output purposes.

3.3 step3.cpp: timer_listener:

In this function, we have implemented the functionality to generate and stream volatility marks and fitting errors to an output file. This is achieved within the timer_listener function, which is called at a specified time frequency.

First, we create a static output file stream (fout) to manage the file handling. If the stream is not already open, we open the file in append mode, ensuring that new data is added without overwriting the existing contents. We also check if the file is empty, and if so, write a header line specifying the column names.

Next, we iterate over the smiles container, which holds the volatility marks and fitting errors. For each set of volatility marks, we write the data to the output file by formatting and writing the relevant values. We include the current timestamp, expiry date, and the various parameter values representing volatility information.

Additionally, we print the same information to the console using std::cout. This allows for immediate visibility and feedback during the program's execution.

By implementing this functionality, we ensure that the volatility marks and fitting errors are recorded and stored in a structured manner in the output file. This facilitates further analysis and examination of the generated volatility surface.

3.4 Handling Failed Root Finding with L-BFGS:

Sometimes, the L-BFGS-B solver may fail to find a root for the ATM volatility (atmvol). To address this, a modification is made in the rfbrent function. If the product of the function evaluations at the boundaries (fa and fb) is greater than zero (fa * fb > 0), it indicates that there is no sign change and a root cannot be found within the given interval. In such cases, the code assumes the market volatility as the result of the root search. This adjustment allows the code to continue running even when the L-BFGS-B solver fails to find a root for atmvol.

3.5 Ouput Example:

3.5.1 Fitting errors:

```
2022-05-16T07:55:01.356Z,16-MAY-2022,fitting error:1.12428
2022-05-16T07:55:01.356Z,17-MAY-2022,fitting error:3.03082
2022-05-16T07:55:01.356Z,20-MAY-2022,fitting error:0.897972
2022-05-16T07:55:01.356Z,27-MAY-2022,fitting error:1.95162
2022-05-16T07:55:01.356Z,3-JUN-2022,fitting error:4.12379
2022-05-16T07:55:01.356Z,24-JUN-2022,fitting error:21.0773
2022-05-16T07:55:01.356Z,29-JUL-2022,fitting error:1.26057
2022-05-16T07:55:01.356Z,30-SEP-2022,fitting error:0.392132
2022-05-16T07:55:01.356Z,30-DEC-2022,fitting error:0.379246
2022-05-16T07:55:01.356Z,31-MAR-2023,fitting error:0.53666
2022-05-16T07:56:01.356Z,16-MAY-2022,fitting error:2.72666
2022-05-16T07:56:01.356Z,17-MAY-2022,fitting error:0.856125
2022-05-16T07:56:01.356Z,20-MAY-2022,fitting error:11.2501
2022-05-16T07:56:01.356Z,27-MAY-2022,fitting error:13.4394
2022-05-16T07:56:01.356Z,3-JUN-2022,fitting error:0.86471
2022-05-16T07:56:01.356Z,24-JUN-2022,fitting error:1.01017
2022-05-16T07:56:01.356Z,29-JUL-2022,fitting error:0.479197
2022-05-16T07:56:01.356Z,30-SEP-2022,fitting error:0.474827
2022-05-16T07:56:01.356Z,30-DEC-2022,fitting error:2.10141
2022-05-16T07:56:01.356Z,31-MAR-2023,fitting error:0.19237
2022-05-16T07:57:01.356Z,16-MAY-2022,fitting error:1.13284
2022-05-16T07:59:01.356Z,16-MAY-2022,fitting error:1.18383
2022-05-16T07:59:01.356Z,17-MAY-2022,fitting error:24.5051
2022-05-16T07:59:01.356Z,20-MAY-2022,fitting error:0.696785
2022-05-16T07:59:01.356Z,27-MAY-2022,fitting error:0.922269
2022-05-16T07:59:01.356Z,3-JUN-2022,fitting error:0.821427
2022-05-16T07:59:01.356Z,24-JUN-2022,fitting error:9.68794
2022-05-16T07:59:01.356Z,29-JUL-2022,fitting error:0.503352
2022-05-16T07:59:01.356Z,30-SEP-2022,fitting error:0.402976
2022-05-16T07:59:01.356Z,30-DEC-2022,fitting error:0.352345
```

3.5.2 BTC Output csv:

A	B	C	D	E	F	G	H
TIME	EXPIRY	FUT_PRICE	ATM	BF25	RR25	BF10	RR10
2022-05-15T08	15-May-22	30039	0.9827	-0.774523	-1.08E-05	-0.605808	-3.43E-05
2022-05-15T08	16-May-22	30036.7	0.8004	-0.628392	-0.000318092	-0.489332	-0.00100848
2022-05-15T08	20-May-22	30029.3	0.8881	-0.698821	-0.00101379	-0.545572	-0.00321867
2022-05-15T08	27-May-22	30055.2	0.7506	-0.58846	-0.00118643	-0.457481	-0.00375678
2022-05-15T08	3-Jun-22	30071.2	0.83215	-0.653872	-0.00180475	-0.509635	-0.00572506
2022-05-15T08	24-Jun-22	30125.4	0.87155	-0.685506	-0.00285868	-0.534846	-0.00907639
2022-05-15T08	29-Jul-22	30247.4	0.72025	-0.564126	-0.00278813	-0.438009	-0.00883036
2022-05-15T08	30-Sep-22	30468	0.6061	-0.472836	-0.00278212	-0.365428	-0.00879171
2022-05-15T08	30-Dec-22	30711.9	0.828782	0.678447	-0.46614	1.33779	-1.16573
2022-05-15T08	31-Mar-23	30869	0.725371	0.60046	-0.427916	1.20302	-1.09646
2022-05-15T08	15-May-22	30024	0.9017	-0.709383	-9.26E-06	-0.553704	-2.94E-05
2022-05-15T08	16-May-22	30023.1	0.47515	-0.368523	-0.000125443	-0.282984	-0.000394479
2022-05-15T08	20-May-22	30020.3	0.71275	-0.55814	-0.000683727	-0.433345	-0.00216414
2022-05-15T08	27-May-22	30044	0.6445	-0.503537	-0.000903183	-0.389917	-0.00285479
2022-05-15T08	3-Jun-22	30058.4	0.63435	-0.495423	-0.00111094	-0.383466	-0.00350987
2022-05-15T08	24-Jun-22	30101.2	2.75886	2.68027	-2.6519	-0.236843	5.04383
2022-05-15T08	29-Jul-22	30222.1	0.61215	-0.477679	-0.0020856	-0.369327	-0.00658735
2022-05-15T08	30-Sep-22	30440.1	7.50497	-2.94366	9.12242	-7.50487	0
2022-05-15T08	30-Dec-22	30691.8	0.47775	-0.370571	-0.00234797	-0.284493	-0.00739226
2022-05-15T08	31-Mar-23	30844.2	0.613259	0.484616	-0.289224	0.916491	-0.677938
2022-05-15T08	15-May-22	29995	0.3443	-0.264645	-1.67E-06	-0.201107	-5.22E-06
2022-05-15T08	16-May-22	29993.8	1.05808	0.756279	-0.034856	1.32475	-0.0721186
2022-05-15T08	20-May-22	29989	9.65206	-9.65196	0	-9.65196	0
2022-05-15T08	27-May-22	30018.6	1.15171	0.832652	-0.175278	1.46823	-0.36676
2022-05-15T08	3-Jun-22	30033.7	0.63425	-0.495344	-0.0011106	-0.383403	-0.0035088
2022-05-15T08	24-Jun-22	30076.8	4.59881	-1.36402	6.46939	-0.727335	7.74275
2022-05-15T08	29-Jul-22	30195.6	0.33785	-0.259724	-0.00072578	-0.19739	-0.0022696
2022-05-15T08	30-Sep-22	30410.3	0.860261	0.668549	-0.366087	1.2439	-0.836277

3.4.3 ETH Output csv:

A	B	C	D	E	F	G	H
TIME	EXPIRY	FUT_PRICE	ATM	BF25	RR25	BF10	RR10
2022-05-15T08:(	15-May-22	2053.22	1.0895	-0.860543	-1.30E-05	-0.674736	-4.14E-05
2022-05-15T08:(	16-May-22	2053.42	1.0906	-0.861856	-0.000554053	-0.676159	-0.00176383
2022-05-15T08:(	20-May-22	2054.05	1.2801	-1.01487	-0.00195552	-0.799095	-0.00623938
2022-05-15T08:(	27-May-22	2055.46	1.00495	-0.79283	-0.00200112	-0.62078	-0.0063649
2022-05-15T08:(	3-Jun-22	2056.39	0.893	-0.702753	-0.00204784	-0.54868	-0.0065034
2022-05-15T08:(	24-Jun-22	2060.25	3.64183	-1.00767	5.26813	-0.452614	6.37824
2022-05-15T08:(	29-Jul-22	2066.86	0.6292	-0.491297	-0.00219028	-0.380133	-0.006921
2022-05-15T08:(	30-Sep-22	2078.92	0.5511	-0.428964	-0.00234853	-0.33068	-0.0074085
2022-05-15T08:(	30-Dec-22	2094.47	0.4869	-0.377845	-0.00242906	-0.290231	-0.00764801
2022-05-15T08:(	31-Mar-23	2110.91	0.45645	-0.35364	-0.0025608	-0.271105	-0.00805749
2022-05-15T08:(	15-May-22	2052.96	0.50685	-0.393502	-3.31E-06	-0.302515	-1.04E-05
2022-05-15T08:(	16-May-22	2053.18	1.05441	0.75365	-0.0346341	1.32015	-0.0716591
2022-05-15T08:(	20-May-22	2053.88	0.848709	0.607845	-0.0595332	1.06596	-0.123416
2022-05-15T08:(	27-May-22	2054.8	4.12261	3.52193	-2.74331	-0.24206	7.7609
2022-05-15T08:(	3-Jun-22	2055.97	0.65735	-0.513808	-0.00118377	-0.39807	-0.00374203
2022-05-15T08:(	24-Jun-22	2059.78	1.0994	-0.868924	-0.00433709	-0.681658	-0.0138195
2022-05-15T08:(	29-Jul-22	2066.49	0.4854	-0.376669	-0.0013797	-0.289382	-0.00434179
2022-05-15T08:(	30-Sep-22	2078.8	1.5597	1.59362	-1.68078	-0.144751	2.82969
2022-05-15T08:(	30-Dec-22	2094.41	0.894066	0.751312	-0.559922	1.5421	-1.48813
2022-05-15T08:(	31-Mar-23	2110.94	0.881363	0.797549	-0.704053	2.04913	-2.59295
2022-05-15T08:(	15-May-22	2049.49	0.3769	-0.290406	-1.96E-06	-0.221309	-6.13E-06
2022-05-15T08:(	16-May-22	2049.84	0.62445	-0.487512	-0.000203945	-0.377198	-0.000644024
2022-05-15T08:(	20-May-22	2051.24	1.08854	0.780915	-0.0981191	1.37077	-0.203738
2022-05-15T08:(	27-May-22	2051.18	0.7249	-0.567869	-0.00111451	-0.441081	-0.00352761
2022-05-15T08:(	3-Jun-22	2052.52	0.44695	-0.346123	-0.000595728	-0.265309	-0.00187214
2022-05-15T08:(	24-Jun-22	2056.59	0.77125	-0.605003	-0.00229642	-0.470617	-0.00727759
2022-05-15T08:(	29-Jul-22	2063.23	0.4885	-0.379134	-0.0013954	-0.291329	-0.00439126
2022-05-15T08:(	30-Sep-22	2075.24	0.4623	-0.358303	-0.001718	-0.274861	-0.00540371