QF633 C++ for Financial Engineering

Project Report

Cai Sheng Siang, Leon

Objective

Consume tick data from a cryptocurrency option exchange (Deribit) to build our choice of volatility market representation updated at a given time frequency.

Overview

1. Step 1: Read Input Events from CSV
2. Step 2: Maintain the Latest Market Snapshot
3. Step 3: Fit Smile at Regular Time Interval and Stream the Marks and Fitting Error to Output File
4. Results and Observations

Step 1: Complete the CsvFeeder Class

As data given were not in strictly increasing order, a data wrangling script called “ETL.ipynb” was created to sort the data according to timestamp ascendingly for “20220515_BTC_Ticker.csv” and “20220515_ETH_Ticker.csv” respectively. Then, the first 100000 rows from the BTC dataframe were written into another csv file called “dev_btc.csv” for development and test case purposes. (Albeit no special treatment was needed for this, this step helped with troubleshooting greatly; the code can run with or without the sorting correctly.)

The main task for Step 1 is to complete and implement the *ReadNextMsg* function where it will read the data from the csv file line by line and parse these data into the desired *TickData* structure which will then be loaded into the *Updates* vector in the *Msg* structure.

The *ReadNextMsg* function should load a complete *Msg* until timestamp changes and multiple rows in the input csv file that have exactly the same timestamp is considered as one *Msg*. As such, we need to check whether the data is grouped according to timestamp correctly before they are passed as a *Msg* to *FeedListener*.

The test case shown below illustrates that the *ReadNextMsg* function works as described above. Also, it can be seen that *TimerListener* which is set to be trigger in every 1 minute interval is working too (highlighted in blue).

	contractName	time
0	BTC-24JUN22-29000-P	2022-05-15T00:00:00.227Z
1	BTC-30DEC22-300000-P	2022-05-15T00:00:00.310Z
2	BTC-31MAR23-140000-C	2022-05-15T00:00:00.310Z
3	BTC-30DEC22-200000-P	2022-05-15T00:00:00.310Z
4	BTC-20MAY22-44000-C	2022-05-15T00:00:00.312Z
5	BTC-27MAY22-23000-P	2022-05-15T00:00:00.313Z
6	BTC-3JUN22-20000-C	2022-05-15T00:00:00.314Z
7	BTC-29JUL22-14000-C	2022-05-15T00:00:00.314Z
8	BTC-20MAY22-50000-C	2022-05-15T00:00:00.316Z
9	BTC-31MAR23-60000-P	2022-05-15T00:00:00.543Z

```
Leon@Leons-MacBook-Air project % g++ -std=c++17 -o test step1.cpp CsvFeeder.cpp && ./test TestData/dev_raw_tick.csv
Contract: BTC-24JUN22-29000-P and timestamp: 2022-05-15T00:00:00.227Z (in epochs: 1652572800227)
Contract: BTC-30DEC22-300000-P and timestamp: 2022-05-15T00:00:00.310Z (in epochs: 1652572800310)
Timestamp: 1652572800227, isSnap = 0, numUpdates = 1

Contract: BTC-31MAR23-140000-C and timestamp: 2022-05-15T00:00:00.310Z (in epochs: 1652572800310)
Contract: BTC-30DEC22-200000-P and timestamp: 2022-05-15T00:00:00.310Z (in epochs: 1652572800310)
Contract: BTC-20MAY22-44000-C and timestamp: 2022-05-15T00:00:00.312Z (in epochs: 1652572800312)
Timestamp: 1652572800310, isSnap = 0, numUpdates = 3

Contract: BTC-27MAY22-23000-P and timestamp: 2022-05-15T00:00:00.313Z (in epochs: 1652572800313)
Timestamp: 1652572800312, isSnap = 0, numUpdates = 1

Contract: BTC-3JUN22-20000-C and timestamp: 2022-05-15T00:00:00.314Z (in epochs: 1652572800314)
Timestamp: 1652572800313, isSnap = 0, numUpdates = 1

Contract: BTC-29JUL22-14000-C and timestamp: 2022-05-15T00:00:00.314Z (in epochs: 1652572800314)
Contract: BTC-20MAY22-50000-C and timestamp: 2022-05-15T00:00:00.316Z (in epochs: 1652572800316)
Timestamp: 1652572800314, isSnap = 0, numUpdates = 2

Contract: BTC-31MAR23-60000-P and timestamp: 2022-05-15T00:00:00.543Z (in epochs: 1652572800543)
Timestamp: 1652572800316, isSnap = 0, numUpdates = 1
```

```
Contract: BTC-30SEP22-60000-P and timestamp: 2022-05-15T00:00:59.968Z (in epochs: 1652572859968)
Timestamp: 1652572859567, isSnap = 0, numUpdates = 1

Contract: BTC-30DEC22-30000-C and timestamp: 2022-05-15T00:01:00.407Z (in epochs: 1652572860407)
Timestamp: 1652572859968, isSnap = 0, numUpdates = 1

Contract: BTC-30DEC22-60000-P and timestamp: 2022-05-15T00:01:00.677Z (in epochs: 1652572860677)
Timestamp: 1652572860407, isSnap = 0, numUpdates = 1

timer_listener called: 1652572860227
```

Step 2: Maintain the Latest Market Snapshot

The *VolSurfBuilder* class keeps track of the latest snapshot of the option market using its member variable *currentSurfaceRaw*. If *Msg* is a snapshot (i.e. *isSnap* = true), its member method *Process* will discard the previously maintained market to avoid error accumulation and load the current snapshot. Else, if the *Msg* is an update (i.e. *isSnap* = false), *Process* will simply apply the updates to the currently maintained market snapshot.

The first snapshot occurred at Timestamp “2022-05-15T00:00:02.014825896Z”.

```
Timestamp: 1652572802014, isSnap = 1, numUpdates = 508
```

I have created a separate test case csv file called “test_step2.csv” to test the *Process* method.

```
test_step2_df = pd.read_csv("test_step2.csv")
```

test_step2_df

0	BTC-24JUN22-29000-P	2022-05-15T00:00:00.227Z	update	BTC	0.0800	2.9	75.12	0.0815	9.3	76.29	...	BTC-24JUN22
1	BTC-30DEC22-300000-P	2022-05-15T00:00:00.310Z	update	BTC	0.0005	5.0	0.00	0.0000	0.0	0.00	...	BTC-30DEC22
2	BTC-31MAR23-40000-C	2022-05-15T00:00:02.014825896Z	snap	BTC	0.1615	14.0	68.03	0.1685	17.4	69.90	...	BTC-31MAR23
3	BTC-30SEP22-20000-P	2022-05-15T00:00:02.014825896Z	snap	BTC	0.0535	2.4	87.78	0.0550	15.8	88.83	...	BTC-30SEP22
4	BTC-31MAR23-40000-C	2022-05-15T00:00:40.596Z	update	BTC	0.1615	14.0	68.04	0.1685	17.4	69.92	...	BTC-31MAR23
5	BTC-30DEC22-60000-P	2022-05-15T00:01:00.677Z	update	BTC	0.0100	60.0	0.00	1.3805	0.1	200.16	...	BTC-30DEC22

```
Leon@Leons-MacBook-Air project % g++ -std=c++17 -o test step2.cpp CsvFeeder.cpp && ./test TestData/test_step2.csv
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Latest Market SNAPSHOT ALERT >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

==================================== PRINTING VOL BUILDER INFO ====================================
timer_listener called: 1652572860227

<<<<<<<<<<<<<<<<<<<<<< CURRENT SNAPSHOT >>>>>>>>>>>>>>>>>>>>>>
ContractName:BTC-30DEC22-60000-P, BestBidPrice:0.01, BestBidAmount:60, BestBidIV:0, BestAskPrice:1.3805, BestAskAmount:0.1, BestAskIV:200.16, MarkPrice:0.9944, UnderlyingIndex:B
TC-30DEC22, UnderlyingPrice:30692.1, LastPrice:0.906, OpenInterest:62.4, LastUpdateTimeStamp:1652572860677

ContractName:BTC-30SEP22-20000-P, BestBidPrice:0.0535, BestBidAmount:2.4, BestBidIV:87.78, BestAskPrice:0.055, BestAskAmount:15.8, BestAskIV:88.83, MarkPrice:0.0543, UnderlyingI
ndex:BTC-30SEP22, UnderlyingPrice:30461, LastPrice:0.054, OpenInterest:1692.4, LastUpdateTimeStamp:1652572802014

ContractName:BTC-31MAR23-40000-C, BestBidPrice:0.1615, BestBidAmount:14, BestBidIV:68.04, BestAskPrice:0.1685, BestAskAmount:17.4, BestAskIV:69.92, MarkPrice:0.1649, UnderlyingI
ndex:BTC-31MAR23, UnderlyingPrice:30858.6, LastPrice:0.155, OpenInterest:111.7, LastUpdateTimeStamp:1652572840596
```

As shown on the output above, the messages before the snapshot is dropped from the current snapshot and as highlighted, contracts of the same name will be updated after the snapshot.

Step 3: Fit Smile at every 1-minute Interval

In this step, *VolSurfBuilder<Smile>::FitSmiles* method groups the market tick data by expiry date, pass the data of each expiry to *CubicSmile::FitSmile* method to fit the model to the market data, and calculate the fitting error in the *SmileError* function where it returns the mean squared error.

I have created a separate test case csv file called "test_step3.csv" to test the *VolSurfBuilder<Smile>::FitSmiles()* method. The input csv contains only 1 snapshot message to test whether market tick data is grouped by expiry date correctly.

For the below test case, as seen on the right hand side, contracts with the same expiry date in the current snapshot are grouped together and it can be verified by the test case with expiry date "31MAR23".

```
test_step3_df[['contractName','Expiry Date']].head()
```

	contractName	Expiry Date
0	BTC-31MAR23-40000-C	31MAR23
1	BTC-30SEP22-20000-P	30SEP22
2	BTC-29JUL22-38000-P	29JUL22
3	BTC-29JUL22-80000-P	29JUL22
4	BTC-30SEP22-90000-P	30SEP22

```
len(test_step3_df[test_step3_df["Expiry Date"] == '31MAR23'])
```

```
46
```

```
test_step3_df["Expiry Date"].nunique() #10 groups of expiry dates
```

```
10
```

```
Size of tickersByExpiry for current snapshot: 10
ContractName:BTC-31MAR23-10000-C, BestBidPrice:0.6835,
572802014

ContractName:BTC-31MAR23-10000-P, BestBidPrice:0.0195,
mp:1652572802014

ContractName:BTC-31MAR23-100000-C, BestBidPrice:0.018,
mp:1652572802014

⋮

ContractName:BTC-31MAR23-70000-P, BestBidPrice:0,
72802014

ContractName:BTC-31MAR23-80000-C, BestBidPrice:0.
eStamp:1652572802014

ContractName:BTC-31MAR23-80000-P, BestBidPrice:0,
2014

The number of contracts with this expiry: 46
```

Next, each group of market tick data is passed to the *CubicSmile::FitSmile* method that will return the fitted smile.

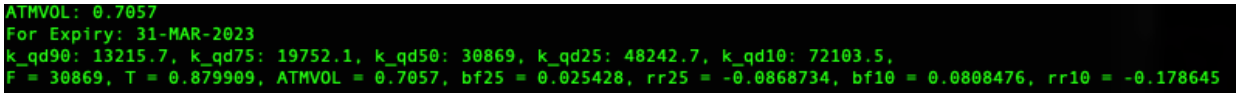
The first step is to interpolate and find the at-the-money volatility (ATMVOL) as it is a constant in the Quick Delta (QD) formula that will be used to obtain strike values of all the 5 QD points.

The method used to find ATMVOL for each expiry date in this project is via a "brute force" approach to vary ATMVOL by an increment of 0.01% up to 300%.

The idea is to pick the ATMVOL that gives the least relative error when it subtracts the interpolated market price from the analytical solution of the Black Scholes model (option price); this was done via iterations. The market price has to be interpolated as it cannot be observed from the data and linear interpolation was used for this case.

```
ERROR: 2.75308e-05
ATMVOL: 0.7057
For Expiry: 31-MAR-2023
```

Then, with the interpolated ATMVOL, calculate the remaining strikes of the other 4 QD points and interpolated the undiscounted price for each of these strikes which will be used in the *impliedVol* function that returns $QD_{0.9}$, $QD_{0.75}$, $QD_{0.25}$ and $QD_{0.1}$ which we will subsequently need to calculate BF_{25}, BF_{10}, RR_{25} and RR_{10}.



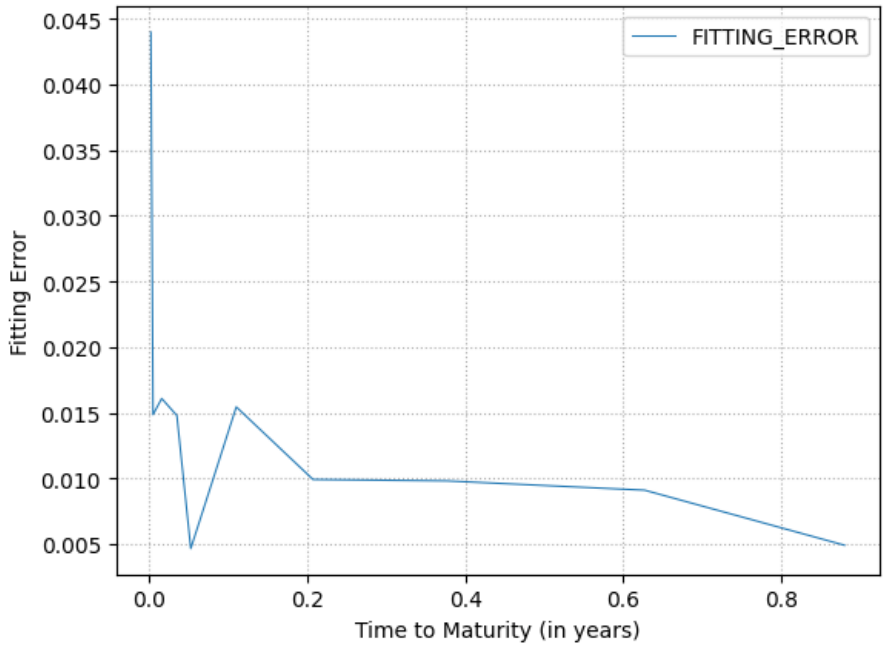
The *SmileError* function is then called to calculate the fitting error (mean squared error) from all the parameter calculated above when compared to the observe market volatility. For simplicity, contracts' mid implied volatility was taken as the average of BestBidIV and BestAskIV for fitting. Marks and fitting error are thus streamed to an output files called "BTC_output.csv" and "ETH_output.csv" for the respectively coins.

Results and Observations

The output file “BTC_output.csv” contains all parameters calculated above, including the fitting error. Below is the smile fitting error across all maturities.

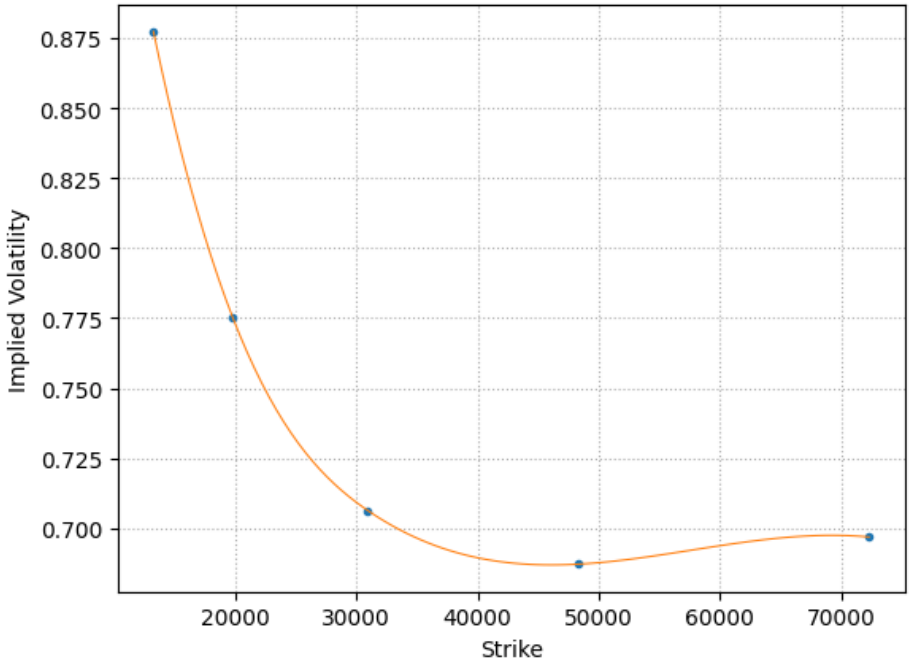
```
btc_op = pd.read_csv("BTC_output.csv")
# eth_op = pd.read_csv("ETH_output.csv")
```

```
btc_op.plot("MATURTIES", "FITTING_ERROR", lw=0.75)
plt.xlabel('Time to Maturity (in years)')
plt.ylabel('Fitting Error')
plt.grid(linestyle='dotted')
plt.show()
```



For the case of BTC, fitting error was the highest when time-to-maturity is very close 0 and decreased rapidly for further expiry dates. The smile model fits the observed market quite well as its fitting error across all expiry dates are below 5% and this is a simplistic case where all the contracts were considered as equally important. As such, fitting errors for contracts with small time-to-maturity could be optimized further if I could take into consideration more market data such as liquidity, open interest and bid-ask spread to determine and thus, optimize the weights.

Below is the interpolated smile obtained for expiry “31-MAR-2023”.



Appendix

Generate objects for each step:

1. `g++ -std=c++17 -o step1 step1.cpp CsvFeeder.cpp`
2. `g++ -std=c++17 -o step2 step2.cpp CsvFeeder.cpp`
3. `g++ -std=c++17 -o step3 step3.cpp CsvFeeder.cpp CubicSmile.cpp Date.cpp`

Commands to run each step:

1. `./step1 TestData/dev_btc.csv`
2. `./step2 TestData/dev_btc.csv`
3. `./step3 TestData/dev_btc.csv TestData/BTC_output.csv`