QF633 Course Project

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

The historical raw tick data is provided as csv files, and each tick (one row in the csv file) contains the below fields:

contractName	BTC-29APR22-41000-C
time	2022-04-18T00:00:00.015Z
msgType	update
priceCcy	BTC
bestBid	0.023
bestBidAmount	8.9
bestBidIV	52.38
bestAsk	0.024
bestAskAmount	8.8
bestAskIV	53.87
markPrice	0.0236156
markIV	53.29
underlyingIndex	BTC-29APR22
underlyingPrice	39667.63
interest_rate	0
lastPrice	0.0225
open_interest	268.1
vega	26.61952
theta	-62.58772
rho	4.39256
gamma	0.0001
delta	0.38025

Figure 1: Deribit option tick data format

The fields relevant to us are:

- `contractName`: the symbol of the exchange traded contract: `Underlying_ExpiryDate_Strike_OptionTYpe`
- `time`: timestamp of the update, in UTC.
- `msgType`: `update` or `snap`
  - `update`: a best level update of a particular instrument, an update message should be applied on top of the currently maintained snapshot

  - `snap`: a snapshot of the whole market, when a snapshot is received, the locally maintained snapshot can be disgarded to avoid error accumulation. When we process a historical data csv file, we need to disgard all updates before the first snapshot.

- `bestBid`: best bid price in unit of BTC (bitcoin)
- `bestBidAmount`: best bid notional, in unit of BTC.
- `bestBidIV`: implied volatility from best bid price.
- `bestAsk`: best ask price in unit of BTC (bitcoin)
- `bestAskAmount`: best ask notional, in unit of BTC.
- `bestAskIV`: implied volatility from best ask price.
- `markPrice`: we can consider this as the mid price.
- `markIV`: implied volatility from the mark price.
- `underlyingIndex`: the underlying futures contract name. Some underlying index starts with 'SYN.', this indicate that the underlying Futures contract is not listed, so the underlying Futures price could be an interpolated price. For this project, we do not need to worry about whether the underlying index is synthetic or not.
- `underlyingPrice`: the price underlying futures contract.
- `lasstPrice`: last transaction price of the contract.
- `open_interest`: current open interest of the instrument, in unit of BTC.

We can ignore the rest of the fields in this project.

Step 1. Read input events from CSV

The first step is to complete the `CsvFeeder` class, that consumes the tick event data. The `CsvFeeder` class needs to be constructed using the below constructor

```
CsvFeeder::CsvFeeder(const std::string ticker_filename,
                     FeedListener feed_listener,
                     std::chrono::minutes interval,
                     TimerListener timer_listener)
```

- `ticker_filename`: the filename of the input csv file,
- `feed_listener`: the function to be called inside `Step()`, after an update message is loaded,
- `interval`: the time interval to call `timer_listener`,
- `timer_listener`: the function to be called at a given frequency.

`FeedListener` and `TimerListener` are our own types defined as follows:

```
using FeedListener = std::function<void(const Msg& msg)>;
using TimerListener = std::function<void(uint64_t ms_now)>;
```

The class `Msg` is given in `Msg.h`. You may modify or extend it. Your main job for step 1 is to complete the step-wise processing functions:

```
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
    return true;
}
```

Note that multiple rows in the csv files that has exactly the same timestamp is considered as one `Msg`. `ReadNextMsg()` should load a complete `Msg` until timestamp changes. It is possible that timestamp is not in strictly increasing order. But we do not need to make special treatment for this.

Note that the time stamp is parsed to unix epoch time (`https://en.wikipedia.org/wiki/Unix_time`) in millisecond precision, which is represented as a `uint64_t`. You can use the function `TimeToUnixMS` implemented in `CsvFeeder.cpp` to parse the input string to the Unix epoch timestamp. A test script `test_ts_parse.cpp` is there for you to test what the parser deos. You can build it using `make test_ts_parser`. You may use the function `getline` (`https://www.cplusplus.com/reference/istream/istream/getline/`) to read a line from the file, and parse the line in your code by separating each field with ",". We have also added a constructor to `datetime_t` that takes a unix epoch timestamp in second (note that the unix is in second, so if you use it with millisecond timestamp, you need to divide it by 1000) to construct our own type `datetime_t`.

It is also possible that certain fields are empty for some ticks. In that case, you can set the field to `NaN` using `std::numeric_limits<double>::quiet_NaN()` defined in `<limits>` header.

Read `step1.cpp`, which contains a dummy `FeedListener` that prints basic information of the `Msg` given, and `TimerListener` that simply prints the timestamp. Build `step1` using `make step1` under the `project` folder. The run it: `bin/step1.out <test_csv_filename>`. Without the implementation of `ReadNextMsg` function, the code will compile but will not function properly. Once `ReadNextMsg` is correctly implemented, `step1` should process the input csv file and print out basic information of the input events.

Step 2: Maintain the latest market snapshot

`VolSurfBuilder` keeps track of the latest snapshot of the option market using the member variable `std::map<std::string, TickData> currentSurfaceRaw` — the latest tick data of all option contracts in the market. Implement the below two functions in `VolSurfBuilder.h`:

```
template <class Smile>
void VolSurfBuilder<Smile>::Process(const Msg& msg) {
    if (msg.isSnap) {
        // discard currently maintained market snapshot, and construct a
           new copy based on the input Msg
    } else {
        // update the currently maintained market snapshot
    }
}

template <class Smile>
void VolSurfBuilder<Smile>::PrintInfo() {
    // TODO: you may print out information about VolSurfBuilder's
       currentSnapshot to test
}
```

`Process` is called by `FeedListener` in `step2.cpp`. Every time `VolSurfBuilder` takes a `Msg`,, it checks if it is a snapshot or an update. If the message is a snapshot, it will discard the previously maintained market, to avoid error accumulation. If the message is an update, it will just apply the updatees to the currently maintained market snapshot.

`PrintInfo` is called by `TimerListener` in `step2.cpp`. It should pprint out the information of the market snapshot at the given interval.

Test `step2.cpp` with `make step2` and `bin/step2.out <test_csv_filename>`.

Note: the class `VolSurfBuilder` is a template class that could take different smile models.

Step 3: Fit smile at regular time interval, and stream the marks and fitting error to output files

The raw snapshot of the market contains strike to price/implied volatility pairs. We want to fit our own volatility smile models to the raw market data so it is easier for pricing and signal research. The smile model we want to use here is cubic spline, implemented as `CubicSmile` (see lecture notes for more details of this smile model). The parameters of the model will be ATMVOL, BF25, RR25, BF10, and RR10 for each expiry date.

Complete the `FitSmiles` function in `VolSurfBuilder.h`. It groups the market tick data by expiry date, pass the data of each expiry to the `FitSmile` function in the smile model to fit the model to the market data, and calculate the fitting error. The fitting error is defined as the mean (or weighted) square error (see `https://en.wikipedia.org/wiki/Mean_squared_error` and lecture notes) of all the contracts' mid implied volatility versus the model volatility at the corresponding strike. For simplicity, at this step we can take the average of `BestBidIV` and `BestAskIV` for fitting, without looking at other fields. You may start with equal weights for all the contracts, and refine weights later (optional). If you choose to assign different weights to different contracts, please describe clearly the formula for the weights in the report.

`FitSmiles` is called by `TimerListener` in `step3.cpp` to generate the volatility surface on our specified time frequency. You should also complete the `time_listener` in `step3.cpp` that streams the volatility marks and fitting errors to the output file. Below is the example format of the output csv file:

```
TIME,EXPIRY,FUT_PRICE,ATM,BF25,RR25,BF10,RR10
2022-04-18T00:00:00.139Z,29-APR-2022,38667.63,0.6,0.02,0.05,0.0335,0.067
```

Implement the `FitSmile` function in `CubicSmile.h`. The function returns a `CubicSmile` that fits the tick data of a given expiry date. The fitting algorithm could be a least square fit using L-BFGS (`https://en.wikipedia.org/wiki/Limited-memory_BFGS`). The project folder includes a header-only library for L-BFGS (`https://lbfgspp.statr.me/`) in the `Solver` subfolder. You can look at the `test_lbfgs` build rule in `Makefile` for how to use it. You may also use algorithm of your own choice (it can also be simple algorithm that does not involve a solver) to fit the input data.

To build step 3: `make step3`

To run step 3: `bin/step3 <input.csv> <output.csv>`

[Bonus point] Consider make minor changes to the CubicSmile model or fitting error measurement to improve the fitting quality. You should describe the rationale and test results if you do so.

Project Submission and Evaluation Criteria

Zip your project PDF report with your project source code folder, submit a single zip file. Indicate your group members in the report. In the report, you shoud describe your test cases and show the test results in tables or charts. Tables or charts does not have to be generated using C++. Only one member of the group needs to submit the report on eLearn.

The evaluation of the project is based on

- Clearness of the presentation
- Thoroughness of the tests
- Correctness, readability, and performance of the code
- Smile fitting error