Design Documentation

Prakhar Srivastava
+919517186482
prakhar.bitsup@gmail.com

Contents

How to run ?

Run binary 2 options

1. **./main_release.o (default simconfig path "./Simconfig.csv")**
2. **./main_release.o Simconfig.csv (../main_release.o <simconfigpath>)**
3.

In case compilation is needed use -> g++ *.cpp (version 11.3)

1. stime, etime, **use_multiple_threads(0/1)** can be changed by editing line1 of Simconfig.csv (global params)
2. Other lines are simulation_strategy specific.

Performance:

1. With multithreading.

real	**0m2.829s**
user	0m5.966s
sys	0m0.339s

2. Without multithreading:

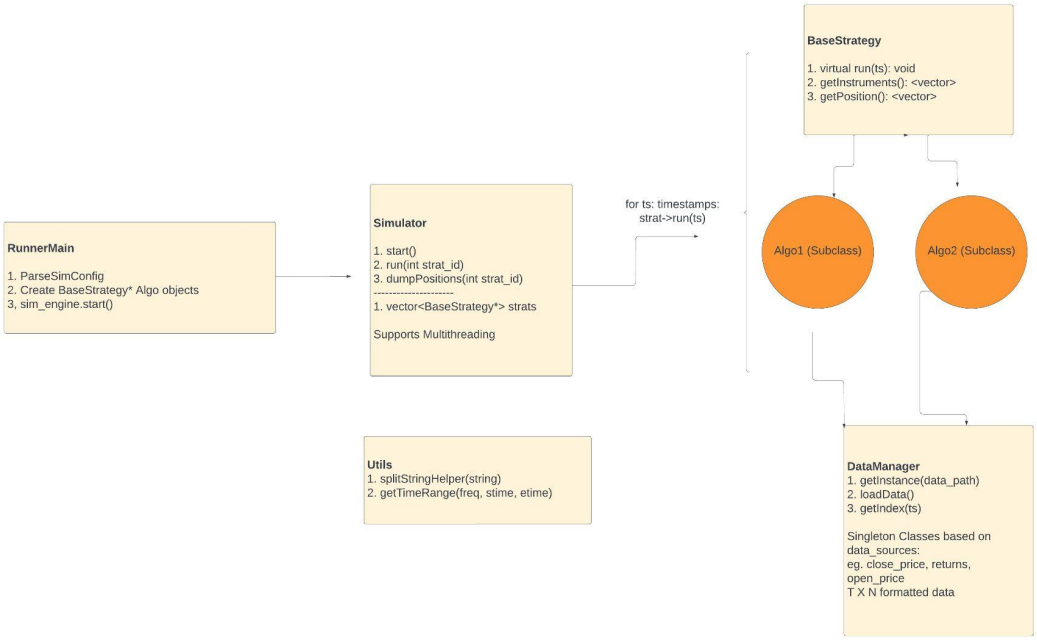
real	**0m6.074s**
user	0m5.693s
sys	0m0.312s

Simconfig.csv

```
stime,2021-04-01 00:00:00,etime,2022-04-30, 23:00:00,use_multiple_threads,1,bar_freq,60
//[key, value, key, value] Global Sim Params
SimSample1,Algo1,close_price_path,Close.csv
SimSample2,Algo1,close_price_path,Close.csv
SimSample3,Algo2,close_price_path,Close.csv,maw,5
SimSample4,Algo2,close_price_path,Close.csv,maw,10
//[simulation_name, algo_name, key, value, key, value…..] Simulation specific infos
```

- Can provide **close_price_path** or any path like **returns_path** etc in the simconfig itself.

Diagram of flow among component



Components

1. RunnerMain

- Start of the program
- Parses Simconfig.csv. Firstline of Simconfig.csv is the global information required by the simulator. These include (stime, etime, bar_freq, use_multiple_threads). The file also has Simulation Algo based information from next lines.

  stime,2021-04-01 00:00:00,etime,2022-04-30, 23:00:00,use_multiple_threads,1,bar_freq,60
  //[key, value, key, value] Global Sim Params
  SimSample1,Algo1,close_price_path,Close.csv
  SimSample2,Algo1,close_price_path,Close.csv
  SimSample3,Algo2,close_price_path,Close.csv,maw,5
  SimSample4,Algo2,close_price_path,Close.csv,maw,10
  //[simulation_name, algo_name, key, value, key, value…..] Simulation specific infos

- 4 positions csv files are generated 2 with Algo1 and 2 with Algo2 with different moving average params.
- Creates Simulator object with
  - **Global_params** -> (stime: 20200401 00:00:00, etime: 20220430 00:00:00, use_multiple_threads: 1…..)

- **Sim_names**: list of simulation names (Simulation1, Simulation2, Simulation3…)
- **Strategies**: list of Strategy objects referenced by basename, in correspondence with sim_names.

```
Simulator(
std::unordered_map<std::string, std::string> global_params,
std::vector<std::string> sim_names,
std::vector<BaseStrategy*> strategies);
```

- Make use of **algo_name -> algo1** from simconfig.csv to create **Algo objects referenced by BaseStrategy*.** This allows for support of dynamic polymorphism on run() method. Thus abstracting child strategies so that traders are concerned about writing logic, all the internals abstracted in BaseStrategy.
- **Precautions**: Simconfig.csv is hardcoded in the program by default, you can pass it as command line argument, refer to **How To run ? section**

---

2. Simulator

- Execution starts when RunnerMain() calls start() public function.
- Simulator is responsible for **generating all valid time_range_strings,** starting from stime to etime. Eg [“20220101 00:00:00”, “20220101 01:00:00”.....] and so on based on frequency (bar_freq) provided by the user.
- Iterates over each timestamp and calls the Strategy run() method. It fetches the positions (1 * N) vector populated by strategy after the run() method by calling strat->getPosition().
- Positions_map[sim_name] -> vector<vector<double>> values. Is appended with latest position fetched from strategy
  Positions_map[strat_name] -> dictionary of vector<vector<double>>

- **Multi Threaded runs:**
  - **startMultipleThreads()** spawns a thread for each of the simulation algo that needs to be run identified from Simconfig.csv.
  - **run(strat_id)** fn spawned. It Iterates over timestamps and calls run(ti) of each of the strategies.
  - Below code snippet showcases that iterations as well as dumpPositions() are designed in such a way that **everything inside it can be run in a threaded environment.**

```
void Simulator::run(int strat_id) {
    std::cout << "Simulation started for: " << sim_names[strat_id] << std::endl;
    auto& strat = strategies[strat_id];
    auto& sim_name = sim_names[strat_id];
    for (auto ti: timestamps) {
```

```
            strat->run(ti);
            positions_map[sim_name].push_back(strat->getPosition());
        }
        dumpPositions(strat_id);
        std::cout << "Simulation ends for: " << sim_name << std::endl;
    }
```

- Positions are dumped using **dumpPositions(strat_id)**
- **Precaution:** Avoid running lots of strategies together as an equal number of threads would be created. Thus can slow down processes due to lots of threads with each having very less task to do and a lot of overhead in sharing hardware resources.
  Can be avoided by fixing the number of threads to an appropriate constant and round robin the strategy runs.

3.BaseStrategy

- BaseStrategy is the parent class of all the strategies written. This contains 2 member variables
  - std::vector<double> position
    - 1 X N size position vector populated by concrete strategy
  - std::vector<std::string> instruments
    - List of instruments. Fetched from any data source.
- getPosition(), used by simulator on each iteration to fetch filled position vector
- 2 concrete algorithms as samples are provided. These are inherited from BaseStrategy. Algo1 && Algo2

---

4. Utils

- Utilities class that encapsulates utility functions as static methods

---

5. DataManager

- DataManger is memory efficient designed class that can represent T*N dataset (Close.csv, Returns.csv). This structure makes sure that for a given data like Close.csv, every **simulation instance uses just 1 overall data instance based on data_source_path** created and cached in memory
- We fetch **read only reference** of values, this ensures that even with single instance of Close_price data referenced by all, the data cannot be changed by any of the strats.

```
const std::vector<std::vector<double>>& DataManager::getValues() {
```

```
    return values;
}
```

- Instances are cached based on data_source_path

```
static std::unordered_map<std::string, DataManager*> instances;
DataManager* DataManager::getInstance(std::string path) {
    if (instances.find(path) == instances.end()) {
        instances[path] = new DataManager(path);
    }
    return instances[path];
}
```

- Singleton class behavior for given data_source_path.
- This function can represent any datasource of the format similar to Close.csv. I.e TXN matrix with the first line being symbols, others being (Timestamp, value, value2….).
- **Returns.csv or Close.csv both can be represented by separate instance of same class**
- **Precaution**:
    - This class only supports TXN objects, thus any custom dataset needs to be handled independently by the user.

---

6. Algo1 && Algo2

Concrete strategies with logic mentioned in comments of codebase

---

Question & Answers

Many good design related arguments can be well presented using a question answer based info. Thus here are some important design features highlighted as answers

Q1. Why not use the Timestamps provided in Close.csv to iterate over each date ?

SImulator class **getTimeRange** function is used by passing stime, etime and bar_freq which fetches a list of timestamps. It iterates over this, and use this as TRUE timestamp sources. Reasons for it being.

- **In the real world any data source can have buggy/inconsistent data, thus dates might be missing in the data.**
- All datasets (Close.csv, Returns.csv) might not have data for each timestamps. **It is not obvious to know the correct timestamp list.**

Q2. What purpose is bar_freq used for ?

There are 2 benefits of using bar_freq

- Easy to work with data of different frequencies
- Can extend the program to work on strategies that work on data with frequency that is multiple of bar_freq. Thus a strategy can listen to bar of 5 hrs and another of 3 hrs accumulated.
- Timestamps are generated using stime, etime and **bar_freq**. Thus can help in reporting any missing data

Q3. How is Close.csv or any data_source memory optimized ?

Close.csv or any TXN dataset is represented by a unique instance of the DataManager class. This is **cached in static mapping on data_source_path** and strategies reference the object by calling **getInstance(path)**. Thereby only 1 instance of each data source is created.

Q4. With all instances sharing the close_price data, how is data correctness maintained ?

By keeping data private and exposing its **constant reference with function getValues()**

```
const std::vector<std::vector<double>>& DataManager::getValues() {
    return values;
}
```

Q5. How are algos abstracted ?

Each Algorithm is abstracted by making it a child class of BaseStrategy. **Thus allowing concrete strategies to only be concerned about ideas and not worry about internals**, which would be placed in baseStrategy.

Issues and their resolutions on memory usage and simulation speed

#Issue1. **Loading large dataset on minutely and subminutely bars.** Currently Close.csv with hourly data is consuming 18M of space, which would increase by 60x -> 1080M with minutely data. The size of data would get exponential when dealing with options dataset with all strikes, and it would be impossible to load all data in memory.
Problem1:

1. Very large data to be stored in memory
2. Significant time wasted in loading whole data. Thus every time on simulation run, lot of time would be wasted in preprocessing and any simple bug of strategy would only be known after code breaks (which would occur only after fixed large time)

#Resolution1: Store the data in **Tar file structure which gives directory structure with data partitioned according to date.**

1. Don't load the whole data at one go. Rather bring only chunk of data to memory (based on dates)
2. After a small data load, pass the control to the strategy and then load the next chunk of data in memory. Thus any error would be encountered very quickly.

#Issue2. **Storing all the position vectors in memory for all the strategies before dumping.**

#Resolution2. Similar to solution1, we can think of dumping the positions accumulated so far in the file after an interval and reset the positions vector (clearing the memory)

#Issue3. **Simulation speed optimizations.**

#Resolution3. The code is designed to make sure no copies of vectors etc are being done when the data source is being read.

1. One must also ensure to not involve a lot of std::string copies and instead use std::string& wherever necessary.
2. Encourage use of vectors wherever possible instead of maps.
3. **Not all strategies might be interested in iterating over all symbols, and might be concerned about just a few symbols.** Thus their positions vector can be made of their interested symbols list size only, making simulations faster.

How do you scale up the number of strategies, and what issues you might encounter ?

With the given design structure, it's very easy to scale up to multiple strategies with the strategies concerned only with writing their alphas and internals being taken care of by BaseStrategy. DataManger is also designed to make the most efficient use of memory.

One concern is that the current design spawns as many threads as there are strategies involved, which should be avoided by fixing the number of threads, and strategies working on

only the given constant of threads. As a lot of threads with too little work would add a lot of overhead and simulation time increase of the code.

Endnotes

Hey! Thanks for making it to the end. I appreciate the fact that you made it till this point of documentation. (I know reading documentation is not fun 🙂). Thus have also provided a flow chart to make things a bit more interesting.

The current design is structured after giving much thought on each of the components with always keeping speed and memory usage in mind, though it can never be perfect hence suggestions on making the given design awesomer 😀 would be helpful and appreciated.

Thank you.