MFE C++ Spring 2022 – Assignment 8

Due 4pm PST, Feb 17, 2022

Please upload your code as separate files and include a PDF/Word document with your writeup. DO NOT UPLOAD YOUR CSV FILES.

Students must make a good faith effort on Assignment 8 to receive the certificate. Further, students are strongly advised to read the entire assignment (maybe a couple of times) before starting coding.

Descriptions of some parts of the assignment are left vague because we expect students to come up with their own solution. However, "vague" is never intended to be misleading so please do not hesitate to ask questions. Given the breadth of the assignment, we may need to update wording to clarify questions.

Market Simulation

In this assignment, we are going to simulate a single security market by translating net trades into a price. Specifically, let's define a variable called ***net***. Further, let's define a security price S as a function of *net*.

$$S(net_i) = e^{p*net_i}$$

where p is a scaling factor.

Imagine a market where N participants (agents) decide whether to buy or sell securities based on their own individual *heuristics* (decision-making processes). We calculate *net* by summing each agent's individual trading amount.

$$net_i = net_{i-1} + \sum_{j=1}^{N} T_{i,j}$$

where $T_{i,j}$ is the j'th agent's trading amount of the security at time i.

Thus, $S_i(net_i) = e^{p*net_i}$

We assume that $net_0 = 0$ so $S_0 = 1$

NOTE: you will run your program using a cpp file downloaded from study.net. This file includes a header file called "agents.h". It is advised (but not required) that you put your class declarations in agents.h and the code implementation in agents.cpp.

Part 1 – CPP code

We will define three different heuristics that must be implement as derived classes of an **Agent** class.

1. Create an abstract Agent class with the following declaration (you may need to add more methods & properties as well):

```
class Agent {
public:
   // the probability the agent trades during a period
   double tradeProb;

   // a scaling factor for the size of the trade
   double tradeScale;



public:
   Agent(double tradeProb, double tradeScale);
   virtual ~Agent() {};
   virtual double tick(double price)=0;
   virtual void reset() {}
};
```

   a. The constructor should set the tradeProb and tradeScale properties.
   b. **reset()** should reset properties (if any) for the agent that are used to track market behavior. We only expect the trend agent described below to implement a **reset()** method of its own. This method is called at the beginning of a simulation run.
   c. **tick()** will be implemented by derived classes. When tick() is called, the agent decides to buy or sell shares based on its heuristic.

2. Create a Result class to hold simulation results.

```
class Result {
public:
	int sim;
	int period;
	double price;
public:
	Result() {};
	Result(int sim, int period, double price)
		: sim(sim), period(period), price(price) {}
};
```

3. Create a **Dealer** class with the following declaration:

```
class Dealer {
private:
       vector<Agent*> agents;
       double priceScale;

public:
   Dealer(double priceScale);
   virtual ~Dealer();
   void addAgent(Agent* a);

   void runSimulation(int numSimulations, int numPeriods,
                      vector< Result > & simResults);

   double getPrice(double net);
}
```

   a. Your constructor should store the **priceScale** property.
   b. Your destructor should deallocate any resources your Dealer allocated. If none, the destructor will be empty.
   c. **addAgent()** should add the agent to the **agents** vector. NOTE: it is not the Dealer's responsibility to **delete** the agents since they are allocated elsewhere.
   d. **getPrice(net)** – returns the price based on net

$$price = e^{priceScale * net}$$

e. **runSimulation** should simulate a market for the securities using the following pseudo-code. The caller of runSimulation() passes in a vector of Result objects in which results will be stored. For each period in the simulation, the dealer keeps track of the *price* at the beginning of the period and updates net as it calls each agent. The dealer only updates the price **after** each agent's tick() method has been called (or passed over).

In the pseudo code below, **price** refers to the current price and is passed into the tick() method of each agent.

Pseudo code for runSimulation:

```
run simulation {
   clear the simResults vector

   for sim number = 1 to numSimulations {
      reset all agents
      set price = 1
      set net = 0
      Add a starting Result object to simResults with sim number, 0 for
                                                          period and 1 for
                                                          price
       Add the starting Result object to simResults

      for period = 1 to numPeriods {
             for each agent {
                 generate a uniform random variable u value in [0,1)
                 if u < agent->tradeProb {
                     net = net + agent->tick(price)
                 }
             }
             price = getPrice(net)
             Create a Result object with sim number, period and price
             Add the result object to simResults
      }
    }
}
```

4. Create a **ValueAgent** class that is a derived class of agent.

```
class ValueAgent : public Agent {
public:
	ValueAgent(double tradeProb,
		double tradeScale
		) :
		Agent(tradeProb, tradeScale) {}

	double tick(double price);

};
```

The ValueAgent sells when its index is positive and buys when its index is negative. It should implement the following heuristic in tick():

double ValueAgent::tick(price) {

double netToTrade = $\boldsymbol{tradeScale} * \left[-\mathbf{0.5} + \frac{\mathbf{1.0}}{\mathbf{2.0}+\boldsymbol{log}(\boldsymbol{price})}\right]$

return netToTrade

}

5. Create a **TrendAgent** class

```
class TrendAgent : public Agent {
protected:
	double prevPrice = 1.0;

public:
	TrendAgent(double tradeProb,
		double tradeScale
	) : Agent(tradeProb, tradeScale) {}

	double tick(double price);
	void reset();
};
```

The TrendAgent looks at its previously-observed price and buys if the previous price is lower than the current price and sells if the previous price is higher.

It heuristic is as follows:

double TrendAgent::tick(price) {

double netToTrade = $\boldsymbol{tradeScale} * (\boldsymbol{log}(\boldsymbol{price}) - \boldsymbol{log}(\boldsymbol{prevPrice}))$

prevPrice = price

return netToTrade

}

The trend agent should implement the virtual **reset()** method. It should call Agent's reset() method by calling **Agent::reset()** and then set **prevPrice** to 1.

6. Create a **NoiseAgent** class that is a derived class of agent.

```
class NoiseAgent : public Agent {

public:
	NoiseAgent(double tradeScale) :
		Agent(1.0, tradeScale) {}

	double tick(double price);


};
```

The NoiseAgent represents random buying and selling in the market. It should implement the following heuristic in tick():

```
double NoiseAgent::tick(price) {
   netToTrade = random standard normal * tradeScale
   return netToTrade
}
```

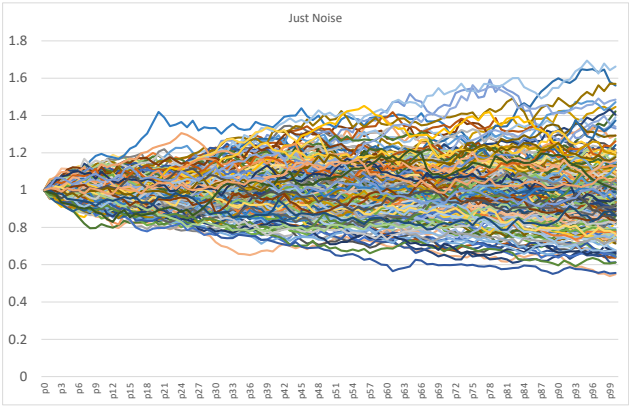
7. Download a**ssignment8.cpp** from Files on bCourses and add it to your project with agents.h and agents.cpp. Run the program. It runs 4 separate simulations and creates six files:
   a. **sim_just_noise.csv** – only noise traders simulation results
   b. **sim_noise_and_value.csv** **-** noise and value traders in the simulation
   c. **sim_noise_and_trend.csv –** noise and trend traders in the simulation
   d. **sim_all_agents.csv** – noise and value and trend traders in the simulation
   e. **agent_implied_vols.csv** – implied volatility curves for the 4 simulations run.
   f. **agent_prices.csv –** price curves for the 4 simulations run.

   Each of the first four files contains the results with ending prices for the simulation. Each simulation file should have 201 lines (a header line and the results, period by period, of 200 simulations). Each line in these files represents a single run of the simulation.

   The two "agent" files contain a T column with the time period analyzed (each row is a different t) and 4 columns with implied volatilities or prices for each of the simulations.

Part 2 – Analyze Results

1. Using Excel, Python, R or your favorite data analysis tool, create line charts for the 4 simulation results. The X-axis should be period and the Y-axis will be price. A sample chart is shown below.



QUESTION 1 Paste your four simulation charts into a document that contains your writeup. Qualitatively, describe the behavior of each simulation. What do you think are the important features of each result? A paragraph of a few sentences will suffice.

2. Using Excel, Python, R or your favorite analysis tool create line charts for the agent implied vol and price csv files. Your X-axis should be time (T) and the Y-axis should be implied vol or price depending on the file.

   Note: the implied vol values are capped at 5 (500%). If you see a 5, that means the actual implied value is higher (possibly much higher). Feel free to cap the implied vol chart's Y-axis at 2 so that the other simulation's vol curves are easier to discern.

   Note 2: "Implied Vol" is the Black Scholes volatility for an option with a given price, time to expiry, strike and risk-free rate. Our simulations do not follow the assumption that vol is constant and, thus, volatilities may be different between strategies and time to expiry.

   Note 3: Your "noise" vol curve should be essentially flat apart from inconsistencies due to simulation. If it is not, then you may have an issue with your agents or dealer implementation. "Essentially flat" means "close to 0.2" or 0.2 +/- 0.01

   Paste your line charts into your writeup along with the raw csv file values as tables.

   QUESTION 2: When looking at the vol curves, how do the value and trend curves compare? What do you think is causing the difference?

   QUESTION 3: How does the "all agents" curve compare to the "noise" vol curve? Describe what happens over time and what you think the contribution of the value and trend agents are having as time progresses.

   These are fairly open-ended questions. We'd like you to think a bit about what you are seeing and why you are seeing it. You will not lose points if you provide thoughtful responses. Also, we do not expect you to write pages and pages. A paragraph is sufficient for each question.