FE545 Design, Patterns and Derivatives Pricing

Strategies, Decoration, and Statistics

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

02/20/2024

Overview

Design a Statistics Collection Class

- One important aspect of the Monte Carlo Simulation is to check the convergence and additional analysis of the simulation quality.
- What we need to measure the standard error, a convergence table, or return all the path data and analyze them somewhere else.
- One of the important goals of the statistics gather is to be **reusable**.
  - For example, we might have many other Monte Carlo routines such as an exotics pricier or a BGM pricer for interest rate derivatives, etc.
  - If we are developing a risk management system, we might be more interested in the ninety-fifth percentile, or in the conditional expected shortfall, than in the mean and variance.

Design a Statistics Collection Class

- How should the routine do? It must have two principal methods. The first should take in data for each path. The second must output the desired statistics.
  - We start with an abstract base class using the virtual methods, just as we did for the

    `PayOff`
  - We have to decide the precise interface for our two principal methods. They will be pure virtual functions declared in the base class and defined in the concrete inherited class.
  - Our second method will indeed return the results, and require a little more thought. We have to decide what sort of object to return the results in.
  - We may want our results to be flexible to accommodate many forms so we may choose to use a vector of vectors.

Design a Statistics Collection Class

- The **strategy pattern** (also known as the policy pattern) is a behavioral design pattern that enables selecting an algorithm at runtime.
- Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.
- A class that performs statistics gathering on incoming Monte Carlo simulation data may use the strategy pattern to select a validation algorithm depending the user choice of statistics or other discriminating factors.

  ** Strategy is one of the patterns included in the influential book Design Patterns by Gamma et al. that popularized the concept of using design patterns to describe how to design flexible and reusable object-oriented software.

A Statistics Gatherer

- Virtual Function
  - StatisticsMC.h
  - StatisticsMC.cpp

StatisticsMC.h

```
#ifndef __ch5_statistics_gatherer__MCStatisitcs__
#define __ch5_statistics_gatherer__MCStatisitcs__

#include <vector>

class StatisticsMC
{
public:
    StatisticsMC(){}

    virtual void DumpOneResult(double result)=0; // a pure virtual function
    virtual std::vector<std::vector<double>> GetResultsSoFar()const=0;
    virtual StatisticsMC* clone()const=0;// possibility of virtual copy constru
    virtual ~StatisticsMC(){}//virtual destructor

private:
};
```

Using the Statistics Gatherer

- Virtual Function
  - SimpleMC7.h
  - SimpleMC7.cpp
  - StatisticsMain1.cpp

SimpleMC7.h

```
void SimpleMonteCarlo5(const VanillaOption& TheOption,
                       ...
                       unsigned long NumberOfPaths,
                       StatisticsMC& gatherer)
{
    ...
    double discounting = exp(-r.Integral(0, Expiry));

    for (unsigned long i = 0; i<NumberOfPaths; i++) {
        ...
        double thisPayOff = TheOption.OptionPayOff(thisSpot);
        // all work on accounting the mean is wrapped
        gatherer.DumpOneResult(thisPayOff*discounting);

    }
    return;
}
```

Templates and Wrappers

- We have so far created a class hierarchy for gathering statistics. We need a wrapper class to manage the memory, just as our *PayOff* class.
- The wrapper class will provide various functionalities which are inherited to make it act like a pointer to a single object but with added responsibilities.
  - If we copy the *Wrapper* object, the pointed-to object is also copied, so that each *Wrapper* object has its own copy of the pointed-to object.
  - When the *Wrapper* object ceases to exist because of going out of scope, or being deleted, the pointed-to object is automatically deleted as well.
  - If we set one *Wrapper* object to another, then the object previously pointed-to must be deleted, and then a copy of the new object must be created so each *Wrapper* still owns precisely one object.
  - It must possible to **dereference** the *Wrapper* to obtain the underlying object, and we also want to **access the methods** of the inner object through $->$ operator.

Templates and Wrappers

- Template and Wrapper Class
  - Wrapper.h
  - Wrapper.cpp

Wrapper.h

```
// specify class T later
// eg. Wrapper<MCStatistics> TheStatsGatherer;
template <class T>

class Wrapper
{
public:
    Wrapper(){ DataPtr = 0;} // the default constructor which point to nothing
    Wrapper(const T& inner) // constructor
    {
        DataPtr = inner.clone();
    }
    ~Wrapper()
    {
        if (DataPtr != 0) // if DataPtr Point to some memory, release it
        {
            delete DataPtr;
        }
    }
    ...
```

Templates and Wrappers

- We start before each declaration with the command

  ```
  template<class T>
  ```

- The compiler will produce one copy of the code for each different sort of T that is used. Thus if we declare

  ```
  Wrapper<MCStatistics> TheStatsGatherer;
  ```

- This has side effects:
  - The first is that all the code for the *Wrapper* template is in the header file and there is no *Wrapper.cpp* file.
  - The second is that if we use the *Wrapper* class many times, we have to compile a lot more code than we might expect.
- We need to provide two different versions of the dereferencing

  ```
  operator*
  operator->
  ```

  to have both *const* and *non* − *const* versions.

Breakout Exercise: Compute More Statistics

- Write a statistics gathering class that computes the second moment of a sample - StatisticsVariance class.
  - Work as a team; one is writing the code and others are watching and then take turns to complete the design.
  - Use the $c++$ project named ch5 statistics gatherer as the starting point for your exercise.
  - Separate StatisticsVariance class from the StatisticsMC.h and StatisticsMC.cpp into StatisticsVariance.h and StatisticsVariance.cpp files.
  - Run the simulation 100 times and gather the required results.

Confidence Interval

- In order to specify how *accurate* a particular estimate $\hat{\ell}$ is, that is, how close it is to the actual unknown parameter ℓ, one needs to provide not only a point estimate $\hat{\ell}$ but a confidence interval as well.
- By the central limit theorem $\hat{\ell}$ has approximately a $N(\ell, \sigma^2/N)$ distribution, where σ^2 is the variance of $H(\mathbf{X})$.

  Usually σ^2 is unknown, but it can be estimated with the *sample variance*

$$S^2 = \frac{1}{(N-1)} \sum_{i=1}^{N} (H(\mathbf{X}_i) - \hat{\ell})^2, \tag{1}$$

  which (by the law of large numbers) tends to σ^2 as $N \to \infty$.
- Consequently, for large N we see that $\hat{\ell}$ is approximately $N(\ell, S^2/N)$ distributed.

Confidence Interval

- Thus, if z_γ denote the γ-quantile of the $N(0,1)$ distribution (this is the number such that $\Phi(z_\gamma) = \gamma$, where Φ denotes the standard normal cdf; for example $z_{0.95} = 1.645$, since $\Phi(1.645) = 0.95$), then

$$\mathbb{P}\left(\hat{\ell} - z_{1-\alpha/2}\frac{S}{\sqrt{N}} \leq \ell \leq \hat{\ell} + z_{1-\alpha/2}\frac{S}{\sqrt{N}}\right) \approx 1 - \alpha \qquad (2)$$

- In other words, an approximate $(1-\alpha)100\%$ *confidence interval* for ℓ is

$$\left(\hat{\ell} \pm z_{1-\alpha/2}\frac{S}{\sqrt{N}}\right), \qquad (3)$$

where the notation $(a \pm b)$ is shorthand for the interval $(a-b, a+b)$.

Confidence Interval

- It is a common practice in simulation to use and report the *absolute and relative* widths of this confidence interval, defined as

$$w_a = 2z_{1-\alpha/2}\frac{S}{\sqrt{N}} \text{ and } w_r = \frac{w_a}{\hat{\ell}}, \tag{4}$$

  respectively, provided that $\hat{\ell} > 0$.

- The absolute and relative widths may be used as stopping rules (criteria) to control the length of a simulation run. The relative width is particularly useful when ℓ is very small.

  For example, think of ℓ as the unreliability (1 minus the reliability) of a system in which all the components are very reliable. In such a case ℓ could be as small as $\ell \approx 10^{-10}$, so that reporting a result such as $w_a = 0.05$ is almost meaningless, while in contract, $w_r = 0.05$ is quite meaningful.

Convergence Table

- One alternative method is to use a convergence table: rather than returning statistics for the entire simulation, we instead return them for every power of two to get an idea of how the numbers are varying.
- We could just write a class directly to return such a table for the mean, but since we might want to do this for any statistic, we do it in a reusable fashion. Our class must contain a statistics gather in order to decide for which statistics to create a convergence table.
- We therefore define a class *ConvergenceTable* which is inherited from *StatisticsMC* and has a wrapper of an *StatisticsMC* object as a data member.

A Convergence Table

- A Convergence Table
  - ConvergenceTable.h
  - ConvergenceTable.cpp

ConvergenceTable.h

```
class ConvergenceTable : public StatisticsMC
{
public:
    // constructor, take in a wrapper class which point to a StatisticMC object
    ConvergenceTable(const Wrapper<StatisticsMC>& Inner_);

    virtual StatisticsMC* clone() const;
    virtual void DumpOneResult(double result);
    virtual std::vector<std::vector<double>> GetResultsSoFar() const;

private:
    Wrapper<StatisticsMC> Inner;
    std::vector<std::vector<double>> ResultSoFar;
    unsigned long StoppingPoint;
    unsigned long PathsDone;
};
```

Decoration

- The technique of this section is an example of a standard design pattern called the *decorator pattern*.
- We have added functionality to a class without changing the interface. This is process is called *decoration*.
- We can decorate as many times as we wish. It would be syntactically legal (but not useful), for example, to have a convergence table of convergence tables.
  - If we have a stream of numbers defining a time series, we often want the statistics of the successive increments instead of the numbers themselves. A decorator class could therefore do this differencing and pass the difference into the inner class.
  - We might want more than one statistic for a given set of numbers; rather than writing one class to gather many statistics, we could write a decorator class which contains a vector of statistics gatherers and passes then gathered values to each one individually.

Decorator Pattern - Advantages

- **Open-Closed Principle**: The Decorator pattern adheres to the Open-Closed Principle, which means that we can extend the functionality of objects without modifying their source code. This promotes code stability and reduces the risk of introducing new bugs when adding new features.
- **Flexibility and Extensibility**: Decorators provide a flexible way to add or remove features dynamically at runtime. We can stack multiple decorators to achieve different combinations of behavior, allowing for fine-grained customization of objects.
- **Reusability**: Decorators are reusable components. Once we create a decorator for a specific functionality, we can apply it to different objects without duplicating code. This promotes code reusability and minimizes code redundancy.

Decorator Pattern - Advantages

- **Single Responsibility Principle**: Each decorator class has a single responsibility, which makes the code more maintainable and easier to understand. This aligns with the Single Responsibility Principle (SRP), one of the SOLID principles of object-oriented design.
- **Promotes Composition over Inheritance**: The Decorator pattern is an excellent alternative to class inheritance for extending object behavior. It avoids the issues associated with deep class hierarchies and the diamond problem, which can occur in languages that support multiple inheritance.
- **Maintains Object Structure**: The Decorator pattern preserves the structure of the original object, ensuring that it remains recognizable. This makes it easier to manage and debug code.

Decorator Pattern - Disadvantages

- **Complexity and Nesting**: Overuse of decorators can lead to a complex hierarchy, making it challenging to understand the overall structure and relationships among decorators and the base component. It can result in ?decorator soup? or a convoluted class hierarchy.
- **Maintenance Overhead**: Managing decorators and their interactions can become complex and lead to increased maintenance overhead. Adding or modifying decorators may require careful consideration to avoid unintended consequences.
- **Performance Overhead**: Each decorator introduces a level of method delegation, which can lead to a minor performance overhead. If performance is a critical concern in our application, the use of decorators should be evaluated carefully.

Decorator Pattern - Disadvantages

- **Limited to Interface Inheritance**: The Decorator pattern relies on interface inheritance, which means that it can only extend the behavior of objects through interfaces or abstract classes. It cannot add new data members to the original object.
- **Limited Support for Removing Decorators**: While we can add decorators dynamically, removing them is not as straightforward. If we need to remove a specific decorator from an object, it may require additional logic or a separate mechanism.
- **Potential for Unintended Behavior**: If decorators are not designed carefully, they can lead to unintended behavior or conflicts between decorators. Ensuring that decorators do not interfere with each other requires careful planning and testing.

Key Points

- Routines can be made more flexible by using the strategy pattern.
- Making part of an algorithm be implemented by an inputtted class is called the strategy pattern.
- For code that is very similar across many different classes, we can use templates to save time in rewriting.
- If we want containers of polymorphic objects, we must use wrappers or pointers.
- Decoration is the technique of adding functionality by placing a class around a class which as the same interface; i.e. the outer class is inherited from the same base class.
- Other design patterns like the Composite pattern or Strategy pattern may provide a more natural and efficient solution than the Decoration pattern for the problem at hand.