FE545 Design, Patterns and Derivatives Pricing

An Exotics Engine and the Template Pattern

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

03/04/2024

Overview

Exotic Option Pricing

- Our objective is to develop a flexible Monte Carlo pricer for exotic options which pay off at some future date according to the value of spot on a finite number of dates.
- We will work within a deterministic interest rate world, and assume the Black-Scholes model of stock price evolution.
- We assume that our derivative is discrete, i.e. that it depends upon the value of spot on a discrete set of times.
  - i Our derivative is associated to a set of times, $t_1, t_2, \ldots, t_n$ and
  - ii pays at some time T a function $f(S_{t_1}, \ldots, S_{t_n})$ of the value of spot at those times.
- For example, a one-year Asian call option struck at K with monthly resets would pay

$$\left(\frac{1}{12}\sum_{j=1}^{12} S_{t_j} - K\right)_+ ,$$

Exotic Option Pricing

- All options give the holder the right, but not the obligation, to buy or sell an underlying asset at a specific price, called the strike, before or at the expiration date.
- Typically the price of the underlying asset is trading at is compared to the strike price to determine profitability. But in a path dependent option, what price is used to determine profitability can vary. Profitability may be based on an average price, or a high or low price, for example.
- More generally, the derivative could possibly pay cash-flows at more than one time. For example, a discrete barrier knock-out option could pay an ordinary vanilla pay-off at the time of expiry, and a rebate at the time of knock-out otherwise.
- We do not consider American or Bermudan options here as the techniques involved are quite different. Note however, that once an exercise strategy has been chosen the option is really just a path dependent derivative.

Understanding Path Dependent Options

- A path dependent option is an exotic option whose payout that can vary based on the path of the underlying asset's price takes over its life or at certain times during the option's life. There are two varieties of path dependent options:

  1. **Soft path dependent option:** bases its value on a single price event that occurred during the life of the option. It could be the highest or lowest traded price of the underlying asset or it could be a triggering event such as the underlying touching a specific price. Option types in this group include barrier options, lookback options, and chooser options.
  2. **Hard path dependent option:** takes into account the entire trading history of the underlying asset. Some options take the average price, sampled at specific intervals. Option types in this group include Asian options, which are also known as average options.

Path Dependent Option

- Here is a brief rundown on several types of path-dependent options:
  - **Barrier Options:** This category includes many sub-varieties, but for all of them, the payoff depends on whether or not the underlying asset reaches or exceeds a predetermined price. A barrier option can be a knock-out option, meaning it can expire worthless if the underlying exceeds a certain price. It can also be a knock-in option, meaning it has no value until the underlying reaches a certain price. Barriers can be below the strike price, above it, or both.
  - **Lookback Options:** Also known as hindsight options, lookback options allow the holder the advantage of knowing history when determining when to exercise their option. This type of option reduces uncertainties associated with the timing of market entry and reduces the chances the option will expire worthless. Lookback options are expensive to execute, so these advantages come at a cost.

Path Dependent Option

- ... continue on types of path -dependent options:
  - **Russian Options:** A Russian option is type of lookback option that does not have an expiration so the life of the option is whatever the holder chooses it to be. They are also known as reduced regret options.
  - **Chooser Options:** This type of option allows the holder to decide whether it is a call or put prior to the expiration date. Chooser options usually have the same exercise price and expiration date regardless of what decision the holder ultimately makes. Because they don't specify that the movement in the underlying asset be positive or negative, chooser options provide investors a great deal of flexibility and tend to be more expensive than conventional options.
  - **Asian Options:** Here, the payoff depends on the average price of the underlying asset over a certain period of time as opposed to standard vanilla options. These options allow the buyer to purchase (or sell) the underlying asset at the average price instead of the spot price.

Asian Option

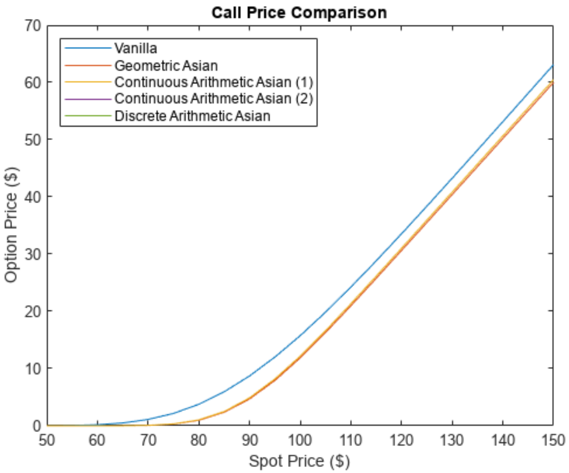

Figure: Average Strike Put Option

Asian Option: Identifying Components

- To generate a pay-off for a path, we have to know the path of stock prices at the relevant times, plug this path into the pay-off function of the option to obtain the cash-flows, and then discount these cash-flows back to the start to obtain the prices for that path
    - i The generation of the stock price path;
    - ii The generation of cash-flows given a stock price path;
    - iii The discounting and summing of cash-flows for a given path;
    - iv The averaging of the prices over all the paths.
- An obvious component for our model is therefore a path-dependent exotic option class which will encapsulate the information which would be written in the *term-sheet* for the option.

Asian Option: Identifying Components

- Note that by defining the concept we model by the term-sheet, we make it clear that the class will not involve interest rates nor knowledge of volatility nor any aspect of the stock price process.
  - A consequence of this is that the option class can only ever say what cash-flows occur and when; it cannot say anything about their discounted values because that would require knowledge of interest rates.
- Note also the general point here that, by defining the concept in real-world terms, it becomes very easy to decide what should and should not be contained in the class.
  - Another consequence is that the component is more easily reusable; if we decided to do jump-diffusion pricing or stochastic interest rates, this class will be reusable without modification.

Asian Option: Identifying Components

- There are more than one way to handle the two main tasks: path generation and cash-flow accounting. The latter will be the same for any deterministic interest rate model.

- Therefore it is natural to include it as part of the main engine class. We can require path generation to be an input to our main class, and therefore define it in terms of its own class hierarchy, or via a virtual method of the base class.

- The option we will pursue is to make path generation a virtual method of the base class. This is an example of the *template* design pattern.

  - The idea is that the base class sets up a structure with methods that control everything - in this case it would be methods to run the simulation and account for each path - though the main work of actually generating the path is not defined in the base class.

Communication between the Components

- Now we need to assess what information has to be passed between them, and we need to decide how to carry out the communication.

  i. The path generator asks the product what times it needs spot for, and it passes back an array.
  ii. The accounting part of the engine asks the product what cash-flow times are possible, and it passes back an array. The engine then computes all the possible discount factors.
  iii. The accounting part of the engine asks the product the maximum number of cash flows it can generate, and sets up a vector of that size.
  iv. For each path, the engine gets an array of spot values from the path generator.
  v. The array of spot values is passed into the product, which passes back the number of cash-flows, and puts their values into the vector.
  vi. The cash-flows are discounted appropriately and summed, and the total value is passed into the statistics gatherer.
  vii. After all the looping is done, the final results are obtained from the statistics gatherer.

The Base Class

PathDependent.h

```
#include "Arrays.h"
#include <vector>


// define the CashFlow and the PathDependent classes which give our path-depend

class CashFlow
{
public:
    double Amount;
    unsigned long TimeIndex;

    CashFlow(const long TimeIndex_ = 0, double Amount_ =0.0): TimeIndex(TimeInd
};
```

The Base Class

PathDependent.h

```
class PathDependent
{
public:
    PathDependent(const MJArray& LookAtTimes); // pass in times need to return

    const MJArray& GetLookAtTimes() const;

    virtual unsigned long MaxNumberOfCashFlows() const=0;
    virtual MJArray PossibleCashFlowTimes() const=0;
    virtual unsigned long CashFlows(const MJArray& SpotValues, std::vector<Cash
    virtual PathDependent* clone() const=0; // virtual copy construction
    virtual ~PathDependent(){} // avoid memory leaking from wrongly destroy bas

private:
    MJArray LookAtTimes;
};

#endif /*
```

The Base Class

PathDependent.cpp

```
#include "PathDependent.h"

PathDependent::PathDependent(const MJArray& LookAtTimes_): LookAtTimes(LookAtTi
{

}

const MJArray& PathDependent::GetLookAtTimes() const
{
    return  LookAtTimes;
}
```

The ExoticEngine Class

ExoticEngine.h

```
...
// GetOnePath() can defined under different model

class ExoticEngine
{
public:
    ExoticEngine(const Wrapper<PathDependent>& TheProduct_, const Parameters& r

    // pure virtual method-> defined in inherited class to adopt various stocha
    virtual void GetOnePath(MJArray& SpotValues)=0; // return array of spot by
    void DoSimulation(StatisticsMC& TheGatherer, unsigned long NumberOfPaths);
    virtual ~ExoticEngine(){}
    double DoOnePath(const MJArray& SpotValues) const; // get the sum of PV of

private:
    Wrapper<PathDependent> TheProduct; // to store the option product
    Parameters r; // store IR
    MJArray Discounts; // store the discount factors
    mutable std::vector<CashFlow> TheseCashFlows; // use "mutable", it can be c
};
```

The ExoticEngine Class

ExoticEngine.cpp

```
#include "ExoticEngine.h"
#include <cmath>


ExoticEngine::ExoticEngine(const Wrapper<PathDependent>& TheProduct_, const Par
{
    // array Discounts first store time, then compute the discount factor first
    for (unsigned long i =0; i<Discounts.size(); i++)
    {
        Discounts[i] = exp(-r.Integral(0.0, Discounts[i]));
    }

    TheseCashFlows.resize(TheProduct->MaxNumberOfCashFlows());
}
...
```

The ExoticEngine Class

ExoticEngine.cpp

```
  ...
void ExoticEngine::DoSimulation(StatisticsMC &TheGatherer, unsigned long Number
{
    MJArray SpotValues(TheProduct-> GetLookAtTimes().size());

    TheseCashFlows.resize(TheProduct->MaxNumberOfCashFlows());
    double thisValue;

    for (unsigned long i=0; i<NumberOfPaths; i++)
    {
        GetOnePath(SpotValues); // simulation one path, possible spot stored in
        thisValue = DoOnePath(SpotValues); // calcuate the PV of payoff cashflo
        TheGatherer.DumpOneResult(thisValue);
    }

    return;
}
...
```

The ExoticEngine Class

ExoticEngine.cpp

```
...
double ExoticEngine::DoOnePath(const MJArray &SpotValues) const
{
    unsigned long NumberFlows = TheProduct->CashFlows(SpotValues, TheseCashFlow

    double Value = 0.0;

    // perform discounting om cash flows
    for (unsigned long i=0; i<NumberFlows; i++)
    {
        Value += TheseCashFlows[i].Amount * Discounts[TheseCashFlows[i].TimeInd
    }

    return Value;
}
...
```

An Arithmetic Asian Option

- Let's look at this new project with the PathDependentAsian class:
  - PathDependentAsian.h
  - PathDependentAsian.cpp
  - ExoticEngine.h
  - ExoticEngine.cpp
- The methods defined are just the ones required by the base class. We pass in the averaging times as an array and we provide a separate delivery time to allow for the possibility that the pay-off occurs at some time after the last averaging date.
- Note that the use of PayOffBridge class means that the memory handling is handled internally, and this class does not need to worry about assignment, copying and destruction.

Breakout Exercise: Geometric Brownian Motion

- Implement a function to generate a Geometric Brownian Motion price path

$$S(t_i) = S(t_{t-1})\exp[(r - \frac{1}{2}\sigma^2 \Delta t) + \sigma\sqrt{\Delta t}\epsilon] \tag{1}$$

  where ϵ is a Gaussian random varaible, and then calculate arithmetic mean

$$A(0,T) = \frac{1}{N}\sum_{i=1}^{N}(S_{t_i}) \tag{2}$$

  and geometric mean

$$A(0,T) = \exp\left(\frac{1}{N}\sum_{i=1}^{N}\log(S_{t_i})\right) \tag{3}$$

- Test your functions with $S_0 = 50$ and $\Delta t = 0.004$. Not that you can time things using the *clock*() function.

A Black-Scholes Path Generation Engine

- The Black-Scholes engine will produce paths from the risk-neutral Black-Scholes process. The paths will be an array of spot values at the times specified by the product.
- We allow the possibility of variable interest rates and dividend rates, as well as variable but deterministic volatility. The stock price therefore follows the process

$$dS_t = (r((t) - d(t))S_t dt + \sigma(t) S_t dW_t, \tag{4}$$

  with S_0 given. To simulate this process at times $t_0, t_1, \ldots, t_{n-1}$, we need n independent $N(0,1)$ variates W_j and we set

$$\begin{aligned} \log S_{t_0} &= \log S_0 \\ &+ \int_0^{t_0} \left(r(s) - d(s) - \frac{1}{2}\sigma(s)^2 \right) ds \\ &+ \sqrt{\int_0^{t_0} \sigma(s)^2 ds} W_0, \end{aligned} \tag{5}$$

A Black-Scholes Path Generation Engine

- We can put a recursive price path as:

$$
\begin{aligned}
\log S_{t_j} &= \log S_{t_{j-1}} \\
&+ \int_{t_{j-1}}^{t_j} \left(r(s) - d(s) - \frac{1}{2}\sigma(s)^2 \right) ds \\
&+ \sqrt{\int_{t_{j-1}}^{t_j} \sigma(s)^2 ds} W_j,
\end{aligned}
\tag{6}
$$

The ExoticEngine Class

ExoticBSEngine.h

```
...
// path is a array of spot values at times specified by the product
class ExoticBSEngine : public ExoticEngine
{
public:
    ExoticBSEngine(const Wrapper<PathDependent>& TheProduct_,
                   const Parameters& R_,
                   const Parameters& D_,
                   const Parameters& Vol_,
                   const Wrapper<RandomBase>& TheGenerator_,
                   double Spot_);

    virtual void GetOnePath(MJArray& SpotValues);
    virtual ~ExoticBSEngine(){}

private:
    Wrapper<RandomBase> TheGenerator;
    MJArray Drifts;
    MJArray StandardDeviations;
    double LogSpot;
    unsigned long NumberOfTimes;
    MJArray Variates;
```

The ExoticEngine Class

ExoticBSEngine.cpp

```
#include "ExoticBSEngine.h"
#include <cmath>

void ExoticBSEngine::GetOnePath(MJArray &SpotValues)
{
    TheGenerator -> GetGaussians(Variates); // get normal

    double CurrentLogSpot = LogSpot;

    for (unsigned long j=0; j<NumberOfTimes; j++)
    {
        // according to the BS model
        // logSt = logS(t-1) + drift + sqrt(variacne)*z
        CurrentLogSpot += Drifts[j];
        CurrentLogSpot += StandardDeviations[j]*Variates[j];
        SpotValues[j] = exp(CurrentLogSpot); // raise to exp, get spot value of
    }
    return;

}
...
```

The ExoticEngine Class

ExoticEngine.cpp

```
  ...
// pass in a wrapper object on RandomBase, so that we can plug in any random nu
//
ExoticBSEngine::ExoticBSEngine(const Wrapper<PathDependent>& TheProduct_,
                               const Parameters& R_,
                               const Parameters& D_,
                               const Parameters& Vol_,
                               const Wrapper<RandomBase>& TheGenerator_,
                               double Spot_):
                                ExoticEngine(TheProduct_,R_),
                                TheGenerator(TheGenerator_)
{
    MJArray Times(TheProduct_ ->GetLookAtTimes());
    NumberOfTimes = Times.size(); // time points needed within one simulated pa

    TheGenerator -> ResetDimensionality(NumberOfTimes);
    Drifts.resize(NumberOfTimes);
    StandardDeviations.resize(NumberOfTimes);

    ...
}
```

The ExoticEngine Class

ExoticEngine.cpp

```
  ...
// pass in a wrapper object on RandomBase, so that we can plug in any random nu
//
ExoticBSEngine::ExoticBSEngine(const Wrapper<PathDependent>& TheProduct_,
                               const Parameters& R_,
                               const Parameters& D_,
                               const Parameters& Vol_,
                               const Wrapper<RandomBase>& TheGenerator_,
                               double Spot_):
                                ExoticEngine(TheProduct_,R_),
                                TheGenerator(TheGenerator_)
{
  ...
     // pre compute the drift term and variacne within each time steps
    double Variance = Vol_.IntegralSquare(0, Times[0]); // first time point
    // compute drift term within first time point
    Drifts[0] = R_.Integral(0, Times[0]) - D_.Integral(0, Times[0]) - 0.5*Varia
    StandardDeviations[0] = sqrt(Variance);

    for (unsigned long j=1; j<NumberOfTimes; j++)
    {
        double thisVariance = Vol_.IntegralSquare(Times[j-1], Times[j]);
        Drifts[j] = R_.Integral(Times[j-1],Times[j])
```

Putting All Together

- Let's look at this new project with the PathDependentAsian Option class:
    - PathDependentAsian.h
    - PathDependentAsian.cpp
    - PathDependent.h
    - PathDependent.cpp
    - ExoticBSEngine.h
    - ExoticBSEngine.cpp
    - PayOffBridge.h
    - PayOffBridge.cpp
    - PayOff3.h
    - PayOff3.cpp

      ..
    - AntiThetic.h
    - AntiThetic.cpp
    - EquityFXMain.cpp
- The compiler silently converts it for us into the *PayoffBridge* object which is then passed to the *PathDependentAsian* constructor.

Key Points

- An important part of the design process is identifying the necessary components and specifying how they talk to each other.
- The template pattern involves deferring the implementation of an important part of an algorithm to an inherited class.
- If an option class knows nothing that is not specified in the term-sheet then it is much easier to reuse.
- We can reuse the PayOff class to simplify the coding of our more complicated path-dependent derivatives.