Palestine Journal of Mathematics

Vol. 9(2)(2020) , 326–344

AN OBJECT-ORIENTED FRAMEWORK USING DESIGN PATTERNS FOR NUMERICAL OPTION PRICING

Jeetendre Narsoo and Sameer Sunhaloo

Communicated by Suheel Khoury

MSC 2020 Classifications: Primary 68N19; Secondary 91G20

Keywords and phrases: Object-oriented programming, design patterns, numerical option pricing.

Abstract Nowadays most software used for development purposes has an environment that supports object-oriented concepts and there has been a major shift from the functional approach to object-oriented programming. The software industry has grown exponentially, covering all major areas of businesses. Among these areas, we have also the financial market. Many financial institutions and investors make use of option contracts to speculate on trends in the stock market or to keep their level of risks from other investments under control. The price of an option is therefore an important factor. Many researchers related to financial options have come up with several algorithms to calculate option values. These algorithms have been implemented in different computer languages or packages. In this paper, we propose an object-oriented framework using design patterns for pricing European options in *MATLAB*, under the Black-Scholes framework.

1 Introduction

Computation of a fair price of an option is of utmost importance in the market of financial derivatives. An option is a contract which gives the owner the right but not the obligation to buy or sell an underlying asset at a prescribed exercise price E on or before the expiry date T. Consider the stock price process [7]

$$dS = \mu S dt + \sigma S dW,$$

where S is the stock price at time t, μ is the expected return on stock, σ is the constant volatility of the stock price and W follows a Wiener process. With v denoting the price of an option, the Black-Scholes-Merton differential equation is given by [2]

$$\frac{\partial v}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 v}{\partial S^2} + rS\frac{\partial v}{\partial S} - rv = 0, \tag{1.1}$$

where r is the risk-free interest rate. By imposing appropriate boundary and initial conditions, (1.1) can be solved to determine the price of an option. However, it is not always possible to obtain closed-form expressions for the values of the options. Thus, numerical techniques must be employed. A powerful tool of pricing options numerically is the finite difference approach.

Object-oriented concepts are linked to the work of Ole-Johan Dahl and Kristen Nygaard on the design of the SIMULA language. Other object-oriented languages became popular in the late 80's. The craze towards object-oriented programming languages started in the 90's. Nowadays most programming languages and packages support an object-oriented environment. The main idea behind object-oriented concepts [1, 4, 8, 9, 10] is to think in terms of objects and manipulate them as they are represented in the real world. Object-oriented programming languages include the facilities to define objects through classes. A class is a blueprint for objects creation. Programming languages and packages like C++, C♯, JAVA, ASP.net, VB.net, php and *MATLAB*, among others support an object-oriented environment.

Design patterns originated from the work of Christopher Alexander, an architect, in the early 60's. Gamma et al. [3] have come up with 23 design patterns that can be used in software design. Design patterns have been used in many software development areas to optimise object-oriented programs to make them more flexible and re-usable. Design patterns have been applied in commercial applications, game development, graphical user interfaces among others. Heer & Agrawala [5] have made use of a few design patterns for information visualisation domain which also includes the representation of data in graphics form. They finally had a design in a reusable form to facilitate software design, implementation and evaluation.

In this paper, we have identified design patterns and proposed an object-oriented framework for the implementation of numerical option pricing algorithms using *MATLAB*. European option prices have been computed in the finite difference setting. It is to be noted that the algorithms discussed in this paper have also been implemented by other researchers using functional approaches. To the best of our knowledge, a few of them have used an object-oriented approach that does not include design patterns. Our approach is expected to provide more flexibility.

2 European Options

A European option can be exercised at the expiry date only. A European call option gives its holder the right to buy an underlying asset and its value at the expiry date is $\max(S(T) - E,\, 0)$. On the other hand, a European put option gives it holder the right to sell an underlying asset and its value at the expiry date is $\max(E - S(T),\, 0)$. It is be noted that

$$c\,(S,\, t) + Ee^{-r(T-t)} = p\,(S,\, t) + S, \tag{2.1}$$

where $c\,(S,\, t)$ and $p\,(S,\, t)$, respectively, denote the call and put values at asset price S and time t.

Let τ denote the time to the expiry date. Then, the value of a European put option satisfies

$$\frac{\partial p}{\partial \tau} = \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 p}{\partial S^2} + rS\frac{\partial p}{\partial S} - rp, \tag{2.2}$$

with initial initial condition $p\,(S,\, 0) = \max\,(S - E,\, 0)$ and boundary conditions

$$p\,(0,\, \tau) = Ee^{-r\tau} \qquad \text{and} \qquad p\,(S,\, \tau) \approx 0, \quad \text{for large} \quad S,$$

for all $\tau \in [0,\, T]$.

It can be shown that the Black-Scholes formula for the value of a European put option is given by

$$p\,(S,\, t) = Ee^{-r(T-t)}N\,(d_1) - SN\,(d_2)\,, \tag{2.3}$$

where

$$d_1 = -\frac{1}{\sigma\sqrt{T-t}}\left(\ln\,(S/E) + \left(r - \frac{1}{2}\sigma^2\right)(T-t)\right),$$

$d_2 = d_1 - \sigma\sqrt{T-t}$ and $N\,(\cdot)$ is the distribution function of the standard normal variable. The Black-Scholes surface for a European put option is shown in Figure 1. The Black-Scholes formula for the value of a European call option can be obtained using (2.1) and (2.3).

3 Finite Difference Formulation

Finite difference methods for solving (2.2) consist of the discretisation of the temporal domain τ over the interval $[0,\, T]$ with N interior points and the discretisation of the spatial domain S over the interval $[0,\, L]$ with M interior points.

Let ΔS and $\Delta\tau$, represent the constant cell spacings along the time and space axes, respectively. Moreover, let $p_j^k = p\,(j\Delta S,\, k\Delta\tau)$. The first temporal derivative is approximated as follows:

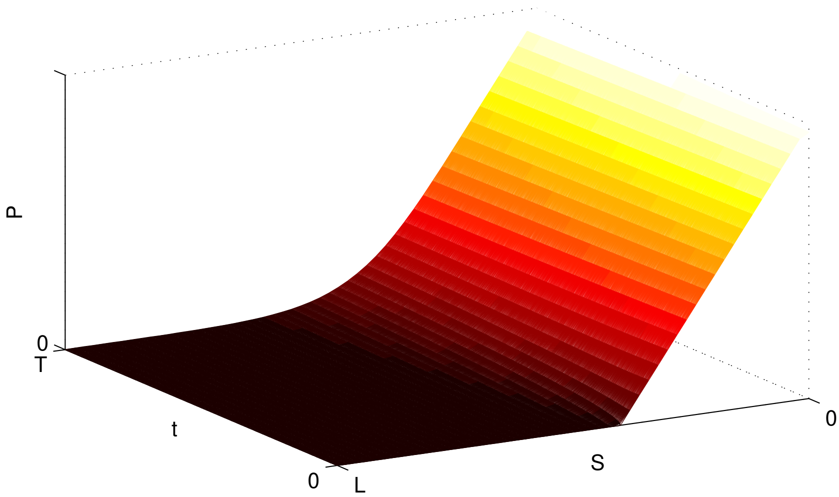

Figure 1: Black-scholes surface for a European put option.

- Forward difference approximation

$$\left(\frac{\partial p}{\partial \tau}\right)_{j,\,k} \approx \frac{1}{\Delta\tau}\left(p_j^{k+1} - p_j^k\right).$$

- Backward difference approximation

$$\left(\frac{\partial p}{\partial \tau}\right)_{j,\,k} \approx \frac{1}{\Delta\tau}\left(p_j^{k} - p_j^{k-1}\right).$$

The first and second spatial derivatives are approximated by

$$\left(\frac{\partial p}{\partial S}\right)_{j,\,k} \approx \frac{1}{\Delta S}\left(p_{j+1}^{k} - p_{j-1}^{k}\right),$$

and

$$\left(\frac{\partial^2 p}{\partial S^2}\right)_{j,\,k} \approx \frac{1}{\Delta S^2}\left(p_{j+1}^{k} - 2p_j^k + p_{j-1}^{k}\right),$$

respectively.

3.1 Matrix-Vector Representation

Application of the Forward Difference in Time and Central Difference in Space (FTCS) scheme to (2.2) gives rise to the linear system

$$\begin{pmatrix} p_1^{k+1} \\ p_2^{k+1} \\ \vdots \\ p_{M-1}^{k+1} \\ p_M^{k+1} \end{pmatrix} = \begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{M-1} & b_{M-1} & c_{M-1} \\ & & & a_M & b_M \end{pmatrix} \begin{pmatrix} p_1^k + \phi_1 \\ p_2^k \\ \vdots \\ P_{M-1}^k \\ p_M^k \end{pmatrix}, \tag{3.1}$$

where $a_j = j\Delta\tau\left(\sigma^2 j - r\right)/2$, $b_j = 1 - r\Delta\tau - \sigma^2 j^2 \Delta\tau$ and $c_j = j\Delta\tau\left(\sigma^2 j + r\right)/2$ for $j = 1, \ldots, M$, and $\phi_1 = a_1 E e^{-rk\Delta\tau}$.

The linear system corresponding to the Backward Difference in Time and Central Difference in Space (FTCS) scheme is

$$\begin{pmatrix} \hat{b}_1 & -c_1 & & & \\ -a_2 & \hat{b}_2 & -c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & -a_{M-1} & \hat{b}_{M-1} & -c_{M-1} \\ & & & -a_M & \hat{b}_M \end{pmatrix} \begin{pmatrix} p_1^{k+1} \\ p_2^{k+1} \\ \vdots \\ p_{M-1}^{k+1} \\ p_M^{k+1} \end{pmatrix} = \begin{pmatrix} p_1^k + \psi_1 \\ p_2^k \\ \vdots \\ p_{M-1}^k \\ P_M^k \end{pmatrix}, \tag{3.2}$$

where $\hat{b}_j = 2 - b_j$ for $j = 1, \ldots, M$, and $\psi_1 = a_1 E e^{-r(k+1)\Delta\tau}$.

The matrix-vector representation of the Crank-Nicolson scheme is obtained by adding (3.1) and (3.2).

Following [6], we define the diagonal matrices D_1 and D_2 and the tridiagonal matrices T_1 and T_2 as follows:

$$D_1 = \begin{pmatrix} 1 & & & & \\ & 2 & & & \\ & & \ddots & & \\ & & & M-1 & \\ & & & & M \end{pmatrix}, \qquad D_2 = \begin{pmatrix} 1^2 & & & & \\ & 2^2 & & & \\ & & \ddots & & \\ & & & (M-1)^2 & \\ & & & & M^2 \end{pmatrix},$$

$$T_1 = \begin{pmatrix} 0 & 1 & & & \\ -1 & 0 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 0 & 1 \\ & & & -1 & 0 \end{pmatrix} \quad \text{and} \quad T_2 = \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix}.$$

Thus, (3.1) can be written as

$$\mathbf{p}^{k+1} = F\mathbf{p}^k + \mathbf{v}_f, \tag{3.3}$$

where $F = (1 - r\Delta\tau) I + \frac{1}{2}\Delta\tau \left(\sigma^2 D_2 T_2 + r D_1 T_1\right)$, $\mathbf{p}^k = \begin{pmatrix} p_1^k & p_2^k & \cdots & p_M^k \end{pmatrix}^T$ and $\mathbf{v}_f = \begin{pmatrix} \phi_1 & 0 & \cdots & 0 \end{pmatrix}^T$. Similarly, (3.2) can be expressed as

$$B\mathbf{p}^{k+1} = \mathbf{p}^k + \mathbf{v}_b, \tag{3.4}$$

where $B = (1 + r\Delta\tau) I - \frac{1}{2}\Delta\tau \left(\sigma^2 D_2 T_2 + r D_1 T_1\right)$ and $\mathbf{v}_b = \begin{pmatrix} \psi_1 & 0 & \cdots & 0 \end{pmatrix}^T$.

Algorithm 3.1 generates European put option values using the FTCS scheme.

Algorithm 3.1 The FTCS algorithm to compute European put option values.

Input the parameters E, σ, r, M, N and T
Set $L = 2E$
Define $\Delta S = L/(M+1)$ and $\Delta\tau = T/(N+1)$
Compute F
Set $\mathbf{p}^0 = \max\left(E - (\Delta S : \Delta S : L - \Delta S)^T, 0\right)$
Define $\mathbf{v}_f = \text{zeros}(M, 1)$
for $k = 0$ to N **do**
  Compute $\phi_1 = a_1 E e^{-rk\Delta\tau}$
  Compute $\mathbf{p}^{k+1} = F\mathbf{p}^k + \mathbf{v}_f$
end for

Algorithm 3.2 computes European put option values using the BTCS scheme.

Algorithm 3.2 The BTCS algorithm to compute European put option values.

Input the parameters E, σ, r, M, N and T
Set $L = 2E$
Define $\Delta S = L/(M+1)$ and $\Delta\tau = T/(N+1)$
Compute B
Set $\mathbf{p}^0 = \max\left(E - (\Delta S : \Delta S : L - \Delta S)^T, 0\right)$
Define $\mathbf{v}_b = \text{zeros}(M, 1)$
for $k = 0$ to N **do**
  Compute $\psi_1 = a_1 E e^{-r(k+1)\Delta\tau}$
  Solve $B\mathbf{p}^{k+1} = \mathbf{p}^k + \mathbf{v}_b$
end for

Algorithm 3.3 generates European put option values using the CN scheme.

Algorithm 3.3 The CN algorithm to compute European put option values.

Input the parameters E, σ, r, M, N and T
Set $L = 2E$
Define $\Delta S = L/(M+1)$ and $\Delta\tau = T/(N+1)$
Compute F and B
Set $\mathbf{p}^0 = \max\left(E - (\Delta S : \Delta S : L - \Delta S)^T, 0\right)$
Define $\mathbf{v}_f = \text{zeros}(M, 1)$ and $\mathbf{v}_f = \text{zeros}(M, 1)$
for $k = 0$ to N **do**
  Compute $\phi_1 = a_1 E e^{-rk\Delta\tau}$ and $\psi_1 = a_1 E e^{-r(k+1)\Delta\tau}$
  Solve $(I + B)\,\mathbf{p}^{k+1} = (I + F)\,\mathbf{p}^k + \mathbf{v}_f + \mathbf{v}_f$
end for

It is to be noted that the linear systems inside the `for` loops in Algorithm 3.2 and Algorithm 3.3 can be solved using the LU-factorisation method.

It can be observed from Algorithms 3.1, 3.2 and 3.3, that the FTCS, BTCS and CN schemes share common characteristics and features. In the sequel, we have proposed a framework using object-oriented concepts and design patterns suitable for option pricing models and that can be used and implemented using the *MATLAB* environment.

4 Object-Oriented Concepts in *MATLAB*

In this section, we introduce an object-oriented approach to implement option pricing algorithms that have been coded using the *MATLAB* environment, which fully supports object-oriented concepts. *MATLAB* R2008 and later versions incorporate an object-oriented framework.

4.1 Class Definition and Object Creation

The object-oriented main concept is to create a class and then defines objects from the class which encapsulates data members and functions together. Data members refer to variables and functions are also known as methods. A class is considered to be a blueprint for objects creation and since classes cannot be used directly, objects are used to manipulate them. The process of objects creation is also called instantiation and an object is said to be an instance of a class. The general structure of a class as defined in *MATLAB* is shown in Listing 1.

Listing 1: The General Structure of a class as defined in *MATLAB*.

```
classdef ClassName
properties
\% This section consists of the declaration of variables
\% also called data members.
end
methods
\% This section consists of declaration of functions
\% also called methods.
end
end
```

The definition of a class is specified by using the `classdef` keyword and `ClassName` is an identifier which is the name given to the class. A class consists of two main blocks where the first block is the `properties` block, which includes the definition of variables and as compared to `C` or `Java` languages, data types are omitted. The second block is the `methods` block in which functions are declared. Data members and methods can be defined as public, private or protected. Private data members or methods can be accessed only within the class they have been defined whereas protected data members or methods can be accessed within the class or sub-classes created through inheritance. Public data members or methods can be accessed within and also outside the class. When data members or methods are declared as private or protected, access to them is only through the use of other methods called accessors. This includes extra function calls which of course will increase the execution time. For this reason, all data members and methods are kept public in our implementation of the option pricing algorithms.

5 Inheritance

One of the advantages of using the object-oriented approach is re-use and this can be shown by creating a base class also known as a super class or parent class. The latter consists of the common features. Sub-classes or children may then be created from the parent class though a process called inheritance. There are two types of inheritance. The creation of sub-classes from a single parent is called simple inheritance. The creation of sub-classes from more than one parent is called multiple inheritance.

6 Design Patterns

A design pattern may be a single class or a group of classes. The idea behind design patterns is to create something that can be used over and over again, that is, a design pattern can be used in the same software or system or in other related software at least more than once. Design patterns are classified into creational, structural and behavioural patterns. Design patterns are described by specifying the *pattern name*, *intent*, *motivation*, *applicability*, *structure*, *participants*, *collaborations*, *consequences*, *implementation*, *sample code* and *known uses*.

7 Proposed Object Oriented Framework for Option Pricing Algorithms

We have identified two design patterns namely, creational and structural, for the option pricing algorithms. The creational patterns are dataModel, Error, Factory, Initialise and European. The structural pattern is a composite pattern named GenericOpPricingModel. Given below is the description of each pattern.

7.1 Creational Pattern: *dataModel*

Pattern Name

dataModel

Intent

The *dataModel* pattern encloses common properties or data used in the option pricing algorithms.

Motivation

We have observed that most option pricing algorithms make use of similar parameters, that is, a common set of data. For example, the FTCS, BTCS and CN schemes use the same parameters, that is, exercise price (E), volatility (σ), interest rate (r) and expiry time (T), among others.

Applicability

This pattern is meant to be used for the option pricing algorithms but it can also be modified for other systems.

Structure

The class diagram of the *dataModel* pattern is given in Figure 2.

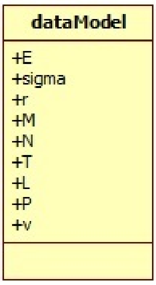

Figure 2: Creational Pattern: dataModel.

Participants

There are no participants.

Collaborations

There are no collaborators since this pattern is a single class.

Consequences

There is no need to declare variables in each option pricing algorithm and all algorithms will make use of the same set of data.

Implementation

Classes will implement the data *dataModel* class through inheritance as shown in Listing 2. The class definition is given in Listing 3.

Listing 2: Implementation of class `dataModel`.

```
classdef bsBtcs < dataModel
...
end
```

The class *bsBtcs* is created from the class *dataModel*.

Sample Code

Listing 3: class `dataModel`.

```
classdef dataModel
% The class dataModel consists of
% the various parameters used in finance models
properties
S;          % Stock Price
E;          % Exercise Price
r;          % Interest Rate
T;          % Expiry Time
sigma;      % Volatility
P;          % Put Option Values
v;
end
end
```

Known Uses

All algorithms discussed in this paper make use of the class *dataModel*.

The next design pattern derived is called *Error*.

7.2 Creational Pattern: Error

Pattern Name

Error

Intent

The *Error* pattern captures parameter input errors.

Motivation

A common *Error* class is used to capture the parameter errors. The parameters are: E, σ, r, M, N and T. The class *Error* consists of the variable Err which is a vector of size 6. Each index refers to a specific error. For example $Err(1)$ refers to error concerning parameter E. Table 1 lists all the possible errors.

Error	Parameter	Description
Err(1)	E	E cannot be less than or equal to zero and cannot be blank
Err(2)	σ	σ cannot be less than or equal to zero and cannot be blank
Err(3)	r	r cannot be less than or equal to zero and cannot be blank
Err(4)	M	M cannot be less or equal to zero and cannot be blank
Err(5)	N	N cannot be less or equal to zero and cannot be blank
Err(6)	T	T cannot be less than or equal to zero and cannot be blank

Table 1: Error List.

Each option pricing algorithm will set the error according to the parameters they are using.

Applicability

This pattern is meant to be used for the option pricing algorithms or can be modified to capture errors for any other system.

Structure

The structure of the class *Error* is given in Figure 3.

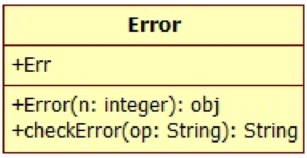

Figure 3: Creational Pattern: Error.

Participants

There are no participants.

Collaborations

There are no collaborators since this pattern is a single class.

Consequences

All the option pricing algorithms will make use of the same *Error* class.

Implementation

Classes will implement the data *Error* class through inheritance as shown in Listing 4.

Listing 4: Implementation of class `Error`.

```
classdef bsBtcs < dataModel < Error
...
end
```

The *Err* variable is set in the *init* method as shown in Listing 5.

Listing 5: code extract from `init` method of the `BTCS class`.

```
...
function obj=init(obj)
if isempty(obj.E)
obj.Err(1) = 1;
end
if obj.E <= 0
obj.Err(1) = 1;
end
if isempty(obj.sigma)
obj.Err(2) = 1;
end
if obj.sigma <= 0
obj.Err(2) = 1;
end
if isempty(obj.r)
obj.Err(3) = 1;
end
if obj.r <= 0
obj.Err(3) = 1;
end
if isempty(obj.M)
obj.Err(4) = 1;
end
```

```
if obj.M <= 0
obj.Err(4) = 1;
end
if isempty(obj.N)
obj.Err(5) = 1;
end
if obj.N <= 0
obj.Err(5) = 1;
end
if isempty(obj.T)
obj.Err(6) = 1;
end
if obj.T <= 0
obj.Err(6) = 1;
end

if sum(obj.Err) > 0
return
end
end
...
```

Sample Code

Sample code used to implement the class *Error* is given in Listing 6.

Listing 6: class `Error`.

```
classdef Error
properties
Err;
end
methods
function [obj]=Error(obj,n)
obj.Err = zeros(n,1);
end
function [mess]=checkError(obj,op)
mess = '';
if obj.Err(1) == 1
mess = [mess 'E'];
end
if obj.Err(2) == 1
if ~isempty(mess)
mess = [mess ',sigma'];
else
mess = [mess 'sigma'];
end
end
if obj.Err(3) == 1
if ~isempty(mess)
mess = [mess ',r'];
else
mess = [mess 'r'];
end
end
if obj.Err(4) == 1
if ~isempty(mess)
mess = [mess ',M'];
```

```
else
mess = [mess 'M'];
end
end
if obj.Err(5) == 1
if ~isempty(mess)
mess = [mess ',N'];
else
mess = [mess 'N'];
end
end
if obj.Err(6) == 1
if ~isempty(mess)
mess = [mess ',T'];
else
mess = [mess 'T'];
end
end
if ~isempty(mess)
mess = [op ': ' mess ' cannot be zero or less or character'];
end
end
end
```

Known Uses

All algorithms discussed so far make use of the class *Error*.

The `Factory` pattern is another design pattern used.

7.3 Factory

A factory is responsible for the creation of objects with same of different types at run-time. Based of the requirement at run-time, the factory will decide which object(s) to instantiate.

Pattern Name

Factory

Intent

The *Factory* pattern is used to create objects of the different option pricing algorithms.

Motivation

We are using a single class to create a set of different objects.

Applicability

This pattern is meant to be used for the option pricing algorithms but it can also be modified for other systems.

Structure

The class diagram and the class definition of the *Factory* pattern are given in Figure 4 and Listing 7, respectively.

Figure 4: Creational Pattern: Factory.

Participants

There are no participants.

Collaborations

There are no collaborators since this pattern is a single class.

Consequences

Creation of objects will be made through the *Factory* pattern.

Implementation

The class *Factory* consists of the *Create* method with one argument. The latter is the name of the object we want to create. For example, *Factory.Create(bsBtcs)* will create an object of type *bsBtcs* class.

Sample Code

Listing 7: class `Factory`.

```
classdef Factory
methods(Access = private)
function obj=Factory(obj)
end
end
methods(Static)
function obj=Create(opt)
obj = eval(opt);
end
end
end
```

Known Uses

The class *Factory* is used to create objects of the different option pricing algorithms at run-time.

We also make use of abstract factory in our implementation.

7.4 Abstract Factory

Abstract factories may be considered to be interfaces that should be implemented by concrete classes. The abstract classes *Initialise* and *European* consist of abstract methods which should be implemented by their respective concrete classes.

Pattern Name

Initialise

Intent

The *Initialise* pattern is used to force all classes related to option pricing models to implement the *init* method. All default values and initialisation are done through this method.

Motivation

All classes will have the same polymorphic method for initialisation.

Applicability

This pattern is meant to be used for option pricing algorithms but it can also be modified for other systems.

Structure

The class diagram and the class definition of the *Initialise* pattern are given in Figure 5 and Listing 8, respectively.

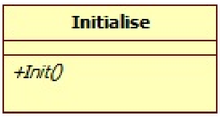

Figure 5: Creational Pattern: Initialise.

Participants

There are no participants.

Collaborations

There are no collaborators since this pattern is a single class.

Consequences

All the option pricing algorithms will implement the *Initialise* pattern.

Implementation

All the option pricing algorithms will implement the *Initialise* pattern.

Sample Code

Listing 8: class `Factory`.

```
classdef Initialise
% Abstract class Initialise
% with abstract method init to be implemented
% by concrete classes
methods(Abstract)
obj=init(obj);
end
end
```

Known Uses

All classes that implement an option pricing algorithm shall implement the *Initialise* class.

Pattern Name

European

Intent

The *European* pattern is used to force all classes related to the option pricing algorithms to implement the *EurPut* method. After implementing the static method *EurPut*, a European option value will be computed.

Motivation

All classes will have the same polymorphic method for the calculation of a European option value.

Applicability

This pattern is meant to be used for the option pricing algorithms but it can also be modified for other systems.

Structure

The class diagram and the class definition of the *European* pattern are given in Figure 6 and Listing 9, respectively.

Figure 6: Creational Pattern: European.

Participants

There are no participants.

Collaborations

There are no collaborators since this pattern is a single class.

Consequences

All option pricing algorithms will implement the *European* pattern.

Implementation

All option pricing algorithms will implement the *European* pattern.

Sample Code

Listing 9: class `European`.

```
classdef European
% Abstract class European
% with abstract method EurPut to be implemented
%  by concrete classes
methods(Abstract)
obj=EurPut(obj);
end
end
```

Known Uses

All classes that implement an option pricing algorithm shall implement the *Initialise* class.

7.5 Generic Option Pricing Model

The generic option pricing model is a composite design pattern and is the framework that can be used while implementing the option pricing algorithms.

Intent

The *Generic Option Pricing Model* pattern is used to provide a common template for the option pricing algorithms.

Motivation

All classes will have to implement the the different interfaces and make use of a common data model and error objects.

Applicability

This pattern is meant to be used for the option pricing algorithms but it can also be modified for other systems.

Structure

The class diagram and the class definition of the *Generic Option Pricing Model* pattern are given in Figure 7 and Listing 10, respectively.

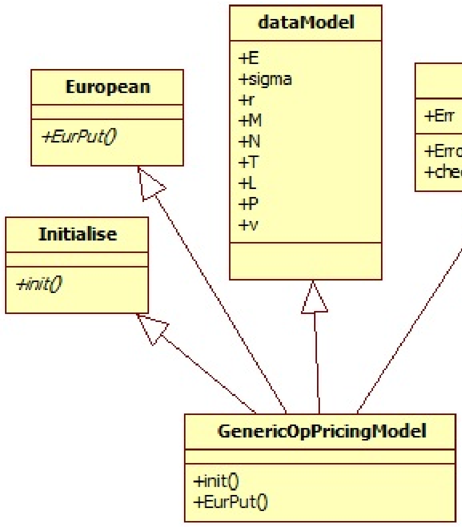

Figure 7: Generic Option Pricing class.

Participants

Participants are European, Initialise, dataModel and Error classes.

Collaborations

The collaborators are same as the participants.

Consequences

All option pricing algorithms will implement the *European* and *Initialise* patterns.

Implementation

All option pricing algorithms will implement the *European* and *Initialise* patterns.

Sample Code

Listing 10: class `Generic Option Pricing Model`.

```
classdef opModel < dataModel & Initialise & European & Error
methods
function obj=init(obj)
# To be implemented by concrete classes
end
function obj=EurPut(obj)
# To be implemented by concrete classes
end
end
end
```

Listing 11: Example bsBtcs class.

```
classdef bsBtcs < dataModel & Initialise & European & Error
methods
function obj=init(obj)
# capture error and set Err variable if any
# initialise other variables
end
function obj=EurPut(obj)
# compute an European option value as per algorithm
end
end
end
```

Known Uses

All classes that implement an option pricing algorithm should implement abstract methods inherited.

8 Numerical Experiments

All numerical experiments in this paper have been performed on an HP laptop with Intel(R) Core(TM) i7-3520M CPU @ 2.90 GHz and Windows 8 environment. *MATLAB* R2012a environment has been used to write and execute the *MATLAB* programs.

8.1 Classical Finite Difference Methods

The classes that implement the FTCS, BTCS and CN schemes to solve the Black-Scholes partial differential equation for a European put option value have been tested with different sets of parameters. As shown in Table 2, we tested the `FTCS`, `BTCS`, and `CN` objects with $N = \{2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}\}$, $E = 100$, $S = 100$, $r = 0.06$, $\Delta S = 0.2$, $T = 1$ and $\sigma = 0.3$.

From Table 2, it can be observed that the FTCS scheme is not stable for all combinations of $\Delta\tau$ and ΔS. We set $\Delta S = 0.2$ for all the three schemes and when smaller values of h are used, the execution times increases and more memory space is required. The CN scheme gives a four decimal point accuracy in fewer times step. The execution time, Etime, is measured in seconds and Error is the difference between the value obtained from the Black-Scholes (BS) formula and the three finite difference schemes studied.

	FTCS			BTCS			True Value
N	Value	Etime	Error	Value	Etime	Error	BS
2^7	NaN	0.20		8.881968	0.75	1.16E-02	8.893526
2^8	NaN	0.19		8.887709	0.09	5.82E-03	8.893526
2^9	NaN	0.27		8.890580	0.11	2.95E-03	8.893526
2^{10}	NaN	0.53		8.892016	0.16	1.51E-03	8.893526
2^{11}	NaN	0.99		8.892734	0.32	7.92E-04	8.893526
2^{12}	NaN	1.97		8.893093	0.53	4.33E-04	8.893526
2^{13}	NaN	3.91		8.893273	0.96	2.53E-04	8.893526
2^{14}	NaN	7.84		8.893362	1.88	1.63E-04	8.893526
2^{15}	NaN	15.86		8.893407	3.63	1.19E-04	8.893526
2^{16}	NaN	18.03		8.893430	7.20	9.61E-05	8.893526
2^{17}	8.893463	63.59	6.24E-05	8.893441	14.47	8.49E-05	8.893526

Table 2: Results of the FTCS and BTCS schemes to compute a European put option value from the Black-Scholes partial differential equation with $E = 100$, $S = 100$, $r = 0.06$, $\Delta S = 0.2$, $T = 1$ and $\sigma = 0.3$.

In Tables 3-5, $S = \{80, 85, 90, 95, 100, 105, 110, 115, 120\}$, $T = \{1, 3\}$, $\sigma = \{0.1, 0.3\}$ and $r = 0.06$.

	BTCS		CN		True Value
S	Value	Error	Value	Error	BS
80	18.955521	8.39E-05	18.955558	4.74E-05	18.955605
85	15.882539	2.09E-04	15.882691	5.66E-05	15.882748
90	13.192400	3.14E-04	13.192651	6.30E-05	13.192714
95	10.870815	3.89E-04	10.871137	6.73E-05	10.871204
100	8.893093	4.33E-04	8.893454	7.24E-05	8.893526
105	7.227820	4.49E-04	7.228188	8.09E-05	7.228269
110	5.840140	4.49E-04	5.840490	9.91E-05	5.840589
115	4.694402	4.50E-04	4.694714	1.38E-04	4.694852
120	3.756097	4.73E-04	3.756359	2.11E-04	3.756570

Table 3: Results of the BTCS and CN schemes to compute European put option values from the Black-Scholes partial differential equation with $E = 100$, $S = 100$, $r = 0.06$, $\Delta S = 0.2$, $T = 1$, $\sigma = 0.3$ and $N = 2^{12}$ (for BTCS) and $N = 2^9$ (for CN).

	BTCS		CN		True Value
S	Value	Error	Value	Error	BS
80	18.569997	1.05E-02	18.570227	1.02E-02	18.580477
85	16.530407	1.57E-02	16.530760	1.53E-02	16.546060
90	14.705421	2.25E-02	14.705877	2.21E-02	14.727959
95	13.075037	3.15E-02	13.075574	3.09E-02	13.106519
100	11.619952	4.28E-02	11.620545	4.23E-02	11.662797
105	10.321899	5.70E-02	10.322538	5.64E-02	10.378902
110	9.163855	7.43E-02	9.164518	7.37E-02	9.238188
115	8.130136	9.52E-02	8.130808	9.45E-02	8.225341
120	7.206422	1.20E-01	7.207089	1.19E-01	7.326407

Table 4: Results of the BTCS and CN schemes to compute a European put option value from the Black-Scholes partial differential equation with $E = 100$, $S = 100$, $r = 0.06$, $\Delta S = 0.2$, $T = 3$, $\sigma = 0.3$ and $N = 2^{12}$ (for BTCS) and $N = 2^9$ (for CN).

	BTCS		CN		True Value
S	Value	Error	Value	Error	BS
80	14.363833	-2.99E-04	14.363602	6.76E-05	14.363534
85	9.886541	-3.02E-04	9.886259	1.99E-05	9.886239
90	6.131633	-8.64E-05	6.131441	1.06E-04	6.131547
95	3.374278	1.77E-04	3.374244	2.11E-04	3.374455
100	1.635493	2.83E-04	1.635557	2.19E-04	1.635776
105	0.698049	2.22E-04	0.698117	1.54E-04	0.698271
110	0.263635	1.14E-04	0.263670	7.95E-05	0.263749
115	0.088801	3.91E-05	0.088808	3.19E-05	0.088840
120	0.026926	6.16E-06	0.026923	9.14E-06	0.026932

Table 5: Results of the BTCS and CN schemes to compute a European put option value from the Black-Scholes partial differential equation with $E = 100$, $S = 100$, $r = 0.06$, $\Delta S = 0.2$, $T = 1$, $\sigma = 0.1$ and $N = 2^{12}$ (for BTCS) and $N = 2^9$ (for CN).

9 Remarks

We have exploited the programming environment of *MATLAB*, which provides both functional and object-oriented approaches to programming. It is to be noted that programs which are designed using an object-oriented approach does not necessarily improve performance. An object-oriented approach using design patterns provides a structured way to write and implement flexible and robust programs. The proposed framework provides a common template for implementing option pricing algorithms. Moreover, the approach presented in this paper can easily be extended to other mathematical models arising in finance and engineering, amongst others.

10 Conclusion

Object-oriented concepts have been used to implement some generic numerical algorithms for pricing financial options under the Black-Scholes framework. In particular, the Forward Difference in Time and Central Difference in Space, Backward Difference in Time and Central Difference in Space and the Crank-Nicolson schemes have been employed to approximate the Black-Scholes equation. The object-oriented framework using design patterns was implemented and tested in *MATLAB*. It should be noted that our proposed strategy does not necessarily improve execution time as most microprocessors execute one instruction at a time. As future work we are proposing to design a flexible graphical user interface to integrate several option pricing models under one common environment.

References

[1] S. M. Biju, Difficulties in Understanding Object Oriented Programming Concepts, *Lecture Notes in Electrical Engineering* **152**, 319–326 (2012).

[2] F. Black and M. Scholes, The Pricing of Options and Corporate Liabilities', *Journal of Political Economy* **81**, 637–654 (1973).

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman Publishing Co., Inc., USA (1995).

[4] S. Georgantaki and S. Retalis, Using Educational Tools for Teaching Object Oriented Design and Programming', *Journal of Information Technology Impact* **7**, 111–130 (2007).

[5] J. Heer and M. Agrawala, Software Design Patterns for Information Visualization, *IEEE Transactions on Visualization and Computer Graphics* **12**, 853–860 (2006).

[6] D. J. Higham, *An Introduction to Financial Option Valuation: Mathematics, Stochastics and Computation*, Cambridge University Press (2004).

[7] J. C. Hull, *Options, Futures, and Other Derivatives*, Pearson (2017).

[8] A. Rani, M. Kavana, M. D. Parvathy and S. J. Shreelakshmi, Object-Oriented Programming and its Concepts, *International Journal for Scientific Research & Development* **5**, 840–842 (2017).

[9] R. Subburaj, G. Jekese and C. Hwata, Impact of Object Oriented Design Patterns on Software Development, *International Journal of Scientific & Engineering Research* **6**, 961–967 (2015).

[10] A. Urdhwareshe, Object-Oriented Programming and its Concepts, *International Journal of Innovation and Scientific Research* **26**, 1–6 (2016).

Author information

Jeetendre Narsoo and Sameer Sunhaloo, School of Innovative Technologies and Engineering, University of Technology, Mauritius, Mauritius.
E-mail: jnarsoo@umail.utm.ac.mu

Received: April 27, 2020.
Accepted: September 2, 2020.