5-2016

Object-Oriented Programming: A Method for Pricing Options

Leonard Stewart Higham

OBJECT-ORIENTED PROGRAMMING: A METHOD FOR PRICING OPTIONS

by

Leonard Stewart Higham

A thesis submitted in partial fulfillment
Of the requirements for the degree

of

MASTER OF SCIENCE

in

Financial Economics

Approved:

Dr. Tyler Brough Major Professor	Dr. Ben Blau Committee Member

Dr. Ryan Whitby
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2016

ABSTRACT

Object-Oriented Programming: A Pricing Engine

by

Leonard Stewart Higham, Master of Science

Utah State University, 2016

Major Professor: Dr. Tyler J. Brough
Department: Finance

In the world of finance, it's becoming necessary to obtain computer programming knowledge and experience, a marketable skill that prepares one conduct quantitative analysis. The objective of this thesis is to utilize concepts in finance and computer science together to form a pricing library for financial derivatives, thus, develop a strong skillset in a specific area of financial computational methods. Through implementation of object-oriented programming and specific design patterns in Python, I develop a pricing engine for many types of options, from plain vanilla to unique and complex options, with the focus on the ability to reuse and extend various pieces of code without ruining the interface for the end user. The modules implemented range from analytical Black Scholes models to binomial option trees to help improve computational speed, power, and accuracy. I also utilize a beneficial platform called Github to facilitate the storage and application of the pricing engines and related files. The results of this project will show a dynamic, yet simple, interface for the end-user, and they will show tangible benefits of object oriented programming.

(13 Pages)

ACKNOWLEDGMENTS

I could never make it as far as I have without my family. Plain and simple.

Dr. Tyler Brough encouraged and helped me every step of the way. My experience and education will always be influenced by my time with him. I want to give him a special thank you.

Thank you for all of your separate efforts, never-ending support, and encouragement.

Leonard Stewart Higham

CONTENTS

Page

INTRODUCTION

The field of finance best suits those that have a love and interest in math and money. As time goes on, computer programming is becoming more relevant and is becoming an important part those interests. Financial economists, as well as the world, benefit from the technological advances the field of computer science provides. Academics are applying models, theories, and new research using computer programming techniques to further improve the finance field of study. The finance industry benefits from these advances by utilizing both fields to find and create comparative advantages. Using object-oriented programming (OOP) techniques to price various options is a brilliant example of this. The importance of this concept affects everyone in the industry; it applicable in various careers. This paper will go over why OOP is an important concept to understand as a recent graduate in finance and economics, the specific design patterns that my project covers and possible extensions to be added to the pricing library, and the results of how studying these topics interact give the user a marketable skillset that has helped jumpstart a career.

In this particular paper, I utilize OOP, implement various design patterns, and set the interface for the end user to price financial options. I create four different files, each with a portion of the necessary code, that interact with each other as the user calls them to receive the option's price. The files contain generic and specific code to be able to dynamically implement the option the user wants to price at run time. For the scope of this paper, the functionality of the designs I created is satisfactory, however, there may be more optimal approaches than the ones I have implemented, and as such, the original code can be modified without the necessity of changing the entire framework of the program. This is the importance of OOP. I will discuss the specifics of the design and functionality in a later section. The files I included are an abstract

factory file that is the basic framework for the OOP process and starts the interaction via a price function, a pricing engine file that contains all of the generic formulas or code to price specific derivative options called through the first file, a market data file in which I set up the ability to retrieve financial data necessary to price an option or pass that responsibility on to the user at runtime, and a payoff function that returns a price based off the specific inputs from the user. I use the abstract factory, façade, and strategy design patterns to standardize and facilitate the structure of the code. I also use a setup tool in Python to abstract the code away from the user, to provide flexibility of use. A computer that initially doesn't have the code can access and run the code with normal functionality with a simple import command. The written code is stored on a Github repository where one can pull the code and modify asneeded.

A Discussion of OOP

There are numerous benefits to OOP. I utilize of the benefits in a finance setting to estimate an option price. An example of an advantage of OOP is being able to abstract away pieces, generally the more complex portion of a formula or code, from the client or main function. For instance, I abstract away the Black Scholes formula, how to run a Monte Carlo simulation, how to calculate certain variables. This brings more simplicity to the end user, can be less computationally expensive, dependent on structure, and advances the reusability of the code. Mark S. Joshi, Ph.D., explains in his book, *C++ Design Patterns and Derivatives Pricing*, that the functionality of OOP resembles that to the natural mental maps of people (Joshi 2010). Another example that I applied to the project is the ability to implement suboptimal, yet functional, code in the time of necessity and have the ability to modify and improve the abstracted code without changing other files, as needed. The versatility of OOP allows meto

add on to code with extended variations and techniques. OOP gives coding structure and reliability that otherwise would be difficult to produce every time a new code is needed. For example, an option needs a model to price it, data, and a payoff function. This structure makes sense and allows the analyst to focus on each part of the option without difficulty. The reliability comes from the code becoming cleaner and easier for others to read and work together without error.

"One of the key elements of OOP is inheritance, which allows you to base a new class on an existing one. It's like getting all of the work that went into writing the existing class for free!" (Joshi 2010) I used specific types of pricing engines, or models that were able to inherit from a generic pricing engine. The generic engine is able to collect the common information from all of the subclasses that existed such as a calculate function. It is important to understand the objective of the inheritance concept. Inheritance abstracts complicated methods and objects away from the end user which creates a simpler environment in which he/she utilizes and reuses.

DESIGN PATTERNS

To build on an object-oriented design, I use various design patterns to add more structure and construct standardized solutions to my code. The creation of design patterns is accredited to a group popularly known as the 'Gang of Four' through a software engineering book that was written to solve recurring functional and design problems. While I could use the whole list of patterns, the patterns that best fit the bill helped organize my pricing library. The main design pattern applied, called the façade design pattern, is an essential piece used to facilitate the pricing of the option for the end user while creating structure under the hood of

the engine. Other important designs that I have in my code are the abstract factory pattern and the strategy pattern.

Façade Pattern

In finance terms, the façade pattern takes the complexities of a pricing engine, payoff function, and the necessary data together and is able to make them interact in the right way without the end user needing to know where the information is coming from. The purpose of a façade pattern is to build a simple interface and hide the complexity of the low-level code that is not necessary to know and understand. A downside to this pattern is that the end user will forego some control over the general code. The main point in using this pattern is to simplify usage and speed up the process for the end user that is pricing options. In Figure 1 the Façade class takes the initializer function to set up the three other files and how they will interact once instantiated from the analyst.

Figure 1

```
import abc

class OptionFacade(object, metaclass=abc.ABCMeta):
    """An option. -- Using Facade design pattern.  This instantiates the price method for the price engine.
    Also requires a payoff method to be used.  Requires an option, a pricing engine, and the data.

    """

    def __init__(self, option, engine, data):
        self.option = option
        self.engine = engine
        self.data = data


    def price(self):
        return self.engine.calculate(self.option, self.data)
```

Abstract Factory Pattern

In my project, I use an abstract factory to call the calculate function that is required by the façade pattern, which is passed on to each concrete factory of specific pricing engines to

calculate the encapsulated function specifically. The function behind using an abstract factory pattern in the sense of a pricing engine is that the abstract factory pattern is designed to create complex objects that are composed of other objects where the composed objects are all of one particular family (Summerfield 2014). This basically means that the abstract factory is passing on or assigning certain responsibilities to another object or class, known as loose coupling. This is also a good way of testing concrete price engines. One downfall of the abstract factory pattern can be when the code or concrete factories get to be sufficiently large, the central point of abstraction, or the responsibility being passed on, may start to change or modify, making more levels of complexity and abstraction necessary. The reason behind using this pattern is to pass on responsibility of deciding which pricing engine in the library to use until runtime, which greatly benefits the client user. In Figure 2 one sees that the abstract factory class called "PricingEngine" contains a calculate function that passes on its responsibility on to a concrete factory class. There is an example of this in the second class mentioned in the figure that contains the calculate function at the bottom.

Figure 2

```
import abc
import enum
import numpy as np
from scipy.stats import binom, norm

class PricingEngine(object, metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def calculate(self):
        """A method to implement a pricing model.  Called from Facade, passed through here.  instantiated in the specific Pricing Engines.
           The pricing method may be either an analytic model (i.e.
           Black-Scholes), a PDF solver such as the finite difference method,
           or a Monte Carlo pricing algorithm.
        """
        pass

class BinomialPricingEngine(PricingEngine):
    def __init__(self, steps, pricer):
        self.__steps = steps
        self.__pricer = pricer

    @property
    def steps(self):
        return self.__steps

    @steps.setter
    def steps(self, new_steps):
        self.__steps = new_steps

    def calculate(self, option, data):
        return self.__pricer(self, option, data)
```

Strategy Pattern

The different payoff functions which are a part of a particular payoff class that is called from the correct pricing engine are dynamically decided at runtime. Other examples include the different types of Monte Carlo variance reduction techniques. Strategy patterns encapsulate the different algorithms that can be used interchangeably depending on the user's needs (Summerfield 2014). The user only needs to know that there exists an abstraction interface and the actual implementation (payoff or engine) chosen will do the same thing, but in different ways to come up with the right answer, but the interface is identical (Phillips 2010). The payoff function in Figure 3 shows the call and put payoffs being referenced by the abstract method in the payoff class above it. This shows inheritance as well as the vanilla payoff class inherits from the payoff class, but because we do not need to decide on a payoff until runtime, we have the ability to call the specific function dynamically.

```
class Payoff(object, metaclass=abc.ABCMeta):
    @property
    @abc.abstractmethod
    def expiry(self):
        """Get the expiry date"""
        pass

    @expiry.setter
    @abc.abstractmethod
    def expiry(self, newExpiry):
        """Set the expiry date"""
        self.__expiry = newExpiry
        pass

    @abc.abstractmethod
    def payoff(self):
        pass

class VanillaPayoff(Payoff):
    def __init__(self, expiry, strike, payoff):
        self.__expiry = expiry
        self.__strike = strike
        self.__payoff = payoff

    @property
    def expiry(self):
        return self.__expiry

    @expiry.setter
    def expiry(self, new_expiry):
        self.__expiry = new_expiry

    @property
    def strike(self):
        return self.__strike

    @strike.setter
    def strike(self, new_strike):
        self.__strike = new_strike

    def payoff(self, spot):
        return self.__payoff(self, spot)


def call_payoff(option, spot):
    return maximum(spot - option.strike, 0.0)

def put_payoff(option, spot):
    return maximum(option.strike - spot, 0.0)
```

Figure 3

SETUP

Python has a package that enables a user to install a third party package into Python called setuptools. This means that a user with a computer that is not connected to the physical code can install the package via Python and it generates the interface necessary for the end user without intensive knowledge of how or why Python works. There is a need for experience in using the package, necessary knowledge of what needs to be called within the package, and the ability to use it correctly. The format used in this project utilized a folder that contained all four of the necessary modules and an initializer module. At the same level as the aforementioned folder were two other modules, a setup module and a test module to assure that the format functions correctly. Below in Figure 4 one sees that once in in the Thesis work folder, there doesn't exist a Probo folder, but with a simple git clone command with the correct URL, or holding place for the pricing library. After the command, one can see highlighted in blue that Probo (the pricing library) is now installed in the Thesis work folder. In Figure 5, the last command uses Python to call the setup file which imports the pricing library package, where the analysts can now price the options in the library.

Figure 4

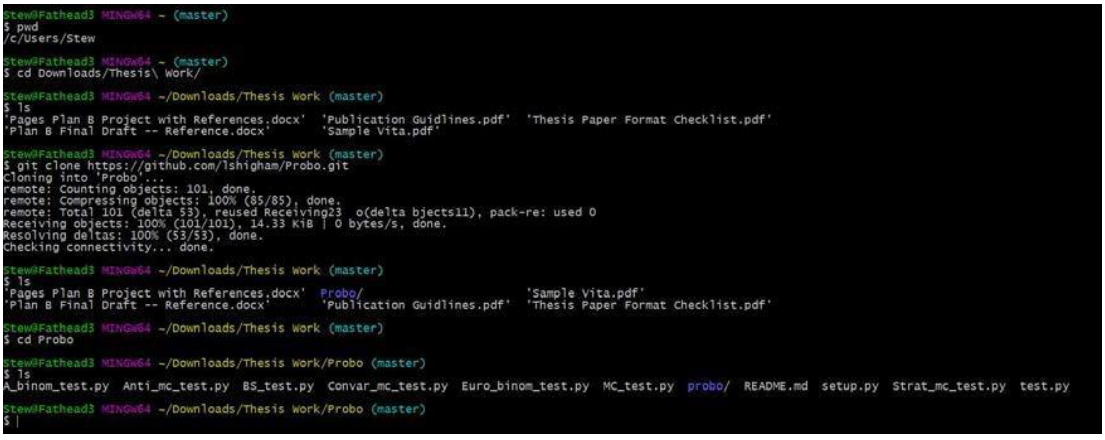


Figure 5

THE CODE

The benefit of the ability to abstract away the payoff function from the specific options, the pricing engine formulas are generic. Although generic, the user has the ability to call the code to estimate the price of a variety options using the binomial option pricing method, Black Scholes, and the Monte Carlo pricing method along with adding variance reduction techniques such as antithetic sampling, stratified sampling, and control variate sampling, and the ability to extend the code. The payoff class consists of a vanilla payoff, which is the maximum of the difference between the spot and strike, depending on the type of option, orzero (Figure____. There is also a Black Scholes payoff and I have the exotic payoff initiated and will extend this project to encompass multiple exotic pricing engines. I also have produced a data class that is a blueprint for pulling data from a predetermined source to price the options with “live” data. I will extend this as well using a package in Python called pandas.

Figures 6 and 7 are examples of the Monte Carlo code and a Monte Carlo simulation with a control variate to reduce variance.

Figure 6

```
class MonteCarloPricingEngine(PricingEngine):
    def __init__(self, replications, time_steps, pricer):
        self.__replications = replications
        self.__time_steps = time_steps
        self.__pricer = pricer

    @property
    def replications(self):
        return self.__replications

    @replications.setter
    def replications(self, new_replications):
        self.__replications = new_replications

    @property
    def time_steps(self):
        return self.__time_steps

    @time_steps.setter
    def time_steps(self, new_time_steps):
        self.__time_steps = new_time_steps

    def calculate(self, option, data):
        return self.__pricer(self, option, data)

def Naive_Monte_Carlo_Pricer(engine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, volatility, dividend) = data.get_data()
    time_steps = engine.time_steps
    replications = engine.replications
    discount_rate = np.exp(-rate * expiry)
    delta_t = expiry / time_steps
    z = np.random.normal(size = time_steps)

    nudt = (rate - 0.5 * volatility * volatility) * expiry
    sidt = volatility * np.sqrt(expiry)

    spot_t = np.zeros((replications, ))
    payoff_t = 0.0
    for i in range(replications):
        spot_t = spot * np.exp(nudt + sidt * z[i])

        payoff_t += option.payoff(spot_t)

    payoff_t /= replications
    price = discount_rate * payoff_t

    return price
```

Figure 7

```
Stew@Fathead3 MINGW64 ~/Downloads/Thesis Work/Probo (master)
$ python MC_test.py
The Call Price is 3.079
```

Figure 8

```
def BlackScholesDelta(spot, t, strike, expiry, volatility, rate, dividend):
    tau = expiry - t
    d1 = (np.log(spot/strike) + (rate - dividend + 0.5 * volatility * volatility) * tau) / (volatility * np.sqrt(tau))
    BS_delta = np.exp(-dividend * tau) * norm.cdf(d1)
    return BS_delta

def ControlVariatePricer(engine, option, data):
    expiry = option.expiry
    strike = option.strike
    (spot, rate, volatility, dividend) = data.get_data()
    time_steps = engine.time_steps
    replications = engine.replications
    delta_t = expiry / time_steps
    nudt = (rate - dividend - 0.5 * volatility * volatility) * delta_t
    sigsdt = volatility * np.sqrt(delta_t)
    erddt = np.exp((rate - dividend) * delta_t)
    beta = -1.0
    cash_flow_t = np.zeros((replications, ))
    price = 0.0

    for j in range(replications):
        spot_t = spot
        convar = 0.0
        z = np.random.normal(size=time_steps)

        for i in range(int(engine.time_steps)):
            t = i * delta_t
            BS_delta = BlackScholesDelta(spot, t, strike, expiry, volatility, rate, dividend)
            spot_tn = spot_t * np.exp(nudt + sigsdt * z[i])
            convar = convar + BS_delta * (spot_tn - spot_t * erddt)
            spot_t = spot_tn

        cash_flow_t[j] = option.payoff(spot_t) + beta * convar

    price = np.exp(-rate * expiry) * cash_flow_t.mean()
    #stderr = cash_flow_t.std() / np.sqrt(replications)
    return price
```

It's noticeable that I used the Black Scholes Delta formula as the control variate and it's resultof 3.410 compared to the naïve Monte Carlo engine's result of 3.079, as seen in Figures 8 and9.

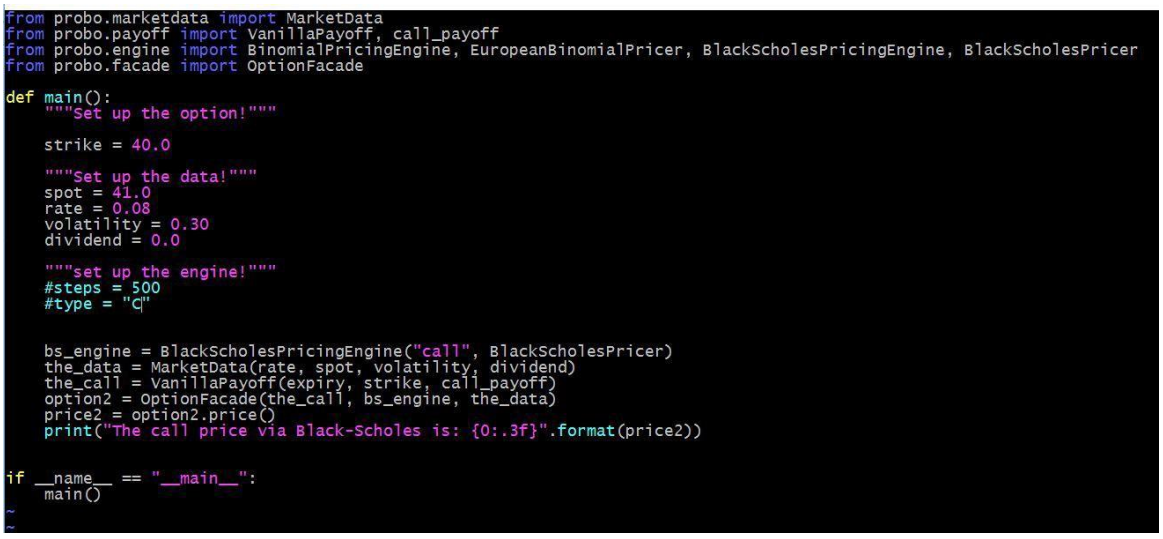
Figure 9

```
Installed c:\users\stew\anaconda3\lib\site-packages\probo-0.1-py3.4.egg
Processing dependencies for probo==0.1
Finished processing dependencies for probo==0.1

Stew@Fathead3 MINGW64 ~/Downloads/Thesis Work/Probo (master)
$ python Convar_mc_test.py
The Call Price via Control Variate Monte Carlo is 3.410
```

Below is examples of the file that will execute the Black Scholes and European Binomial pricing method as the interface for the analyst. First, in figure 10 we see the Black Scholes

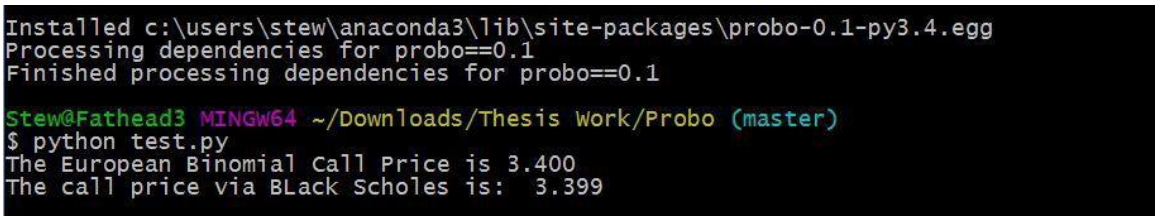
formula and its result versus the European option being priced by the binomial method and the corresponding results in Figure 11.

Figure 11



```
from probo.marketdata import MarketData
from probo.payoff import VanillaPayoff, call_payoff
from probo.engine import BinomialPricingEngine, EuropeanBinomialPricer, BlackScholesPricingEngine, BlackScholesPricer
from probo.facade import OptionFacade

def main():
    """Set up the option!"""

    strike = 40.0

    """Set up the data!"""
    spot = 41.0
    rate = 0.08
    volatility = 0.30
    dividend = 0.0

    """set up the engine!"""
    #steps = 500
    #type = "C"


    bs_engine = BlackScholesPricingEngine("call", BlackScholesPricer)
    the_data = MarketData(rate, spot, volatility, dividend)
    the_call = VanillaPayoff(expiry, strike, call_payoff)
    option2 = OptionFacade(the_call, bs_engine, the_data)
    price2 = option2.price()
    print("The call price via Black-Scholes is: {0:.3f}".format(price2))


if __name__ == "__main__":
    main()
~
~
```

Figure 12



```
Installed c:\users\stew\anaconda3\lib\site-packages\probo-0.1-py3.4.egg
Processing dependencies for probo==0.1
Finished processing dependencies for probo==0.1

Stew@Fathead3 MINGW64 ~/Downloads/Thesis Work/Probo (master)
$ python test.py
The European Binomial Call Price is 3.400
The call price via BLack Scholes is:  3.399
```

The end user can utilize whichever method deemed most useful or optimal at runtime to price an option. As the modules were executed, benchmark prices from the McDonald textbook *Derivative Markets* were used to check accuracy of the estimated prices. If the user needs a specific option that doesn't exist, all that is needed is an implementation of apricing

engine and a payoff. If the user wants a better computational turnaround time for a current method of pricing, the software engineering team works on an enhanced algorithm that increases the computational speed without compromising the OOP and design pattern concepts. This will not break any code for the end user and increases the efficiency of the library.

EXTENSIONS

One of the most intriguing aspects of this project is the fact that a countless number of variations and additions can be done to extend and modify the original code, customizing it to one's needs without compromising the framework of the code. One can amend the pricing engine code by using other design patterns, or implementing more patterns, like a builder or decorator pattern, as the code grows. Adding methods like the Brownian Bridge, the Heston model, and a trinomial tree model are examples of additions that can be made. A linter can be added to find errors in the programming code. Even a graphical user interface (GUI) can be implemented for the end user to see the various methods in which the option can be priced. Different types of variance reduction techniques such as importance sampling and Low discrepancy sequences (McDonald 2006). The option pricing library can even be implemented for real options, depending on the size of the scope. This list is not comprehensive and is meant to be a building block to add on to the library.

CONCLUSION

By implementing the techniques discussed in this paper one will be able to accurately price options using conventional methods with the ability to have reusable code the can be enhanced and extended to take on more complexities and volume. Coding is becoming more commonplace in the financial industry and its application can be a marketable skill for recent graduates. While there are existing software packages in the world that do this type of work automatically, a quick search for quantitative finance jobs required compute software engineering skills will show that there is a demand for these related skillsets. Skills associated with being able to make unique adjustments and variations to be more applicable to a specific need, or to be able to fix various problems or bugs. These are what provide the most value to me.

REFERENCES

Joshi, Mark S. 2010. *C++ Design Patterns and Derivatives Pricing.* 2nd. New York, New York: Cambridge University Press. Accessed March 2016.

McDonald, Robert L. 2006. *Derivatives Markets.* 2nd . Boston, MA: Pearson Education, Inc. Accessed Feb 2016.

Phillips, Dusty. 2010. *Python 3 Object Oriented Programming.* Birmingham: Packt Publishing Ltd. Accessed April 2016.

Summerfield, Mark. 2014. *Python in Practice.* Crawfordsville, Indiana: Pearson Education, Inc. Accessed March 2016.