Utah State University
DigitalCommons@USU

All Graduate Plan B and other Reports Graduate Studies

12-2017

Implementing the Heston Option Pricing Model in Object-Oriented Cython

Brandon Hardin
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/gradreports

Part of the Finance and Financial Management Commons, Management Sciences and Quantitative Methods Commons, Portfolio and Security Analysis Commons, and the Risk Analysis Commons

Recommended Citation
Hardin, Brandon, "Implementing the Heston Option Pricing Model in Object-Oriented Cython" (2017). *All Graduate Plan B and other Reports*. 1146.
https://digitalcommons.usu.edu/gradreports/1146

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Plan B and other Reports by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

Implementing the Heston Option Pricing Model in Object-Oriented Cython

Brandon C. Hardin

Master of Science
In
Financial Economics
Utah State University 2017

Implementing the Heston Option Pricing Model in Object-Oriented Cython

Master thesis in Financial Economics at
the Huntsman School of Business 2017

Committee Chair:
Dr. Tyler Brough
Other thesis committee members:
Dr. Ben Blau
Dr. Ryan Whitby

Utah State University
Huntsman School of Business
August 4, 2017

Abstract

The 1973 Black-Scholes model, a revolutionary option pricing formula whose price is 'relatively close to observed prices, makes an assumption that the volatility is constant and thus, deterministic. This causes some inefficiencies and patterns in the pricing of options due to the model providing evidence of the volatility smile' of the volatility. Many scholars have suggested that the volatility should be modelled by a stochastic process and the (1993) Heston Model is one of many proposed solutions to remedy this problem. The Heston Model allows for the 'smile' by defining the volatility as a stochastic process. This thesis considers a solution to this problem by utilizing Heston's stochastic volatility model in conjunction with Euler's discretization scheme in a simple Monte Carlo engine.

The application of this model has been implemented in object-oriented Cython, for it provides the simplicity of Python, all the while, providing C performance.

Area of review: Financial Economics
Subject Classification: Computational Methods, Monte Carlo, Heston, Python, Cython

Table of Contents

Chapter 1: Introduction……………………………………………………………1

Chapter 2: The Black-Scholes Model………………………………………2

Chapter 3: The Heston Model…………………………………………………..2

Chapter 4: The Monte Carlo Method……………………………………..3

Chapter 5: Python Programming Language……………………………….7

Chapter 6: Implementing Option Pricing Models in Python………8

Chapter 7: Object-Oriented Programming……………………………….11
7.1: Functions
7.2: Object-Oriented Python

Chapter 8: Design Patterns in Computational Finance…………….21
8.1: Facade Pattern
8.2: Strategy Pattern

Chapter 9: Cython…………………………………………………………………..26

Chapter 10: The Heston Model in Cython……………………………….29

Chapter 11: The Facade and Strategy Patterns……………………….31
11.1: Facade
11.2: Strategy

Chapter 12: Conclusion…………………………………………………………..35

References…………………………………………………...........................36

Appendices……………………………………………………………………………38

Chapter 1. Introduction

This thesis shall simulate the Heston Model by the use of Cython and thus, the reasoning behind the chosen model must be identified. The primary factor for the decision to implement the Heston Model was how it determines the evolution of volatility of the underlying asset. With the aid of continuous time diffusion models for volatility, the Heston Model derives its option price from a random process. Although the Black-Scholes model is widely supported, it is subject to inefficiencies, error and incorrectly pricing securities due to the assumption of constant volatility. If the Black-Scholes assumption were correct than, when viewing strike prices, the implied volatilities of same type options would be constant. This is not the case as patterns of volatilities varying by strike can be seen forming a smile curve or "volatility smile".

Stochastic volatility models were formulated in order to solve this problem. These models incorporate the verifiable observations, for which, the volatility of the model is a random process. In turn, volatility itself is made into stochastic process. Developed by Steven Heston, the widely used Heston (1993) model not only took time-dependent volatility into account, it also presents a stochastic process element. The Heston Model provides correlated shocks between asset returns and volatility. This assumption provides insight into the reasoning behind return skewness and strike-price biases in the Black-Scholes model. In addition, the Heston provides for the actuality of a semi-analytical resolution for European options.

In this thesis, we shall institute a simulation of one of the most widely used stochastic volatility models, the Heston Model's volatility stochastic process. The implementation is inclusive of random-number generation in a Monte Carlo engine. Monte Carlo simulation is a vital technique used in option pricing as it not only provides an improvement in the efficiency of a simulation, but it does so by sampling values randomly from all possible outcomes from the input probability distributions. The Monte Carlo simulation does this iteration as many times as specified and the result is a probability distribution of all possible outcomes. The Monte Carlo simulation implementation is quantified in Cython within the Python software. Python is a high-level programming language that is used in a variety of technical areas including finance. Although, Python is widely used for option pricing theory, the execution of the aforementioned within a Cython environment is relatively new. In the forthcoming chapters of this thesis, we shall introduce and review these methods.

Chapter 2. The Black-Scholes Model

The most recognized and widely used continuous time model is the Black-Scholes. With its simplistic nature and requirement of only five inputs: strike price, asset price, expiry, risk-free rate, and volatility, the Black-Scholes makes an assumption of an underlying asset, S, where it follows a geometric Brownian motion and we assume the drift and volatility is constant. The plain Black-Scholes model process is as follows:

$$dS_t = \mu S_t \, dt + \sigma S_t dW_t \tag{2.1}$$

where both μ, the drift and σ, the volatility, are under the assumption of being constant. As the result of asset price changes are lognormal-distributions, which means that the values are positive and they create a right skewed curve, which is a disadvantage of the model. In addition, the Black-Scholes model has demonstrated an issue with being consistent with the market as far as implied volatility is concerned. Let us allow:

$$C_{BS}(S, K, T, r, I) = C_{market} \tag{2.2}$$

to denote the Black-Scholes price for a European call option with T time to expiry, strike price K, S is the value of the underlying asset, r is the risk-free rate and implied volatility is the value of I for which, is the volatility that allows the Black-Scholes price to equal that of what is observed in the market, while making note that volatility, which has a drastic effect on price is not visible. When calculating implied volatilities from market data, the same sigma should be detectable for all options on the same underlying asset, but this is not what is observed.

The implied volatilities derived from market data are not constant as they vary with the strike price and time to expiry even when associated with the same underlying asset, which entails the formation of a skew or "volatility smile." The implied volatilities also vary over the course of time in a stochastic fashion. This directly repudiates the assumptions of constant volatility that Black-Scholes states. A stochastic volatility model can remedy the Black-Scholes of this contradiction.

Chapter 3. The Heston Model

The evolution of the volatility of an underlying asset provides the reasoning behind the creation of the Heston Model. When there is a correlation between the asset price and volatility, it produces a closed-form solution and allows the model to make the addition of stochastic interest rates. The Heston (1993) model [1] is based upon the following stochastic differential equations, which depicts the stock price and variance process diffusions under a probability measure $\mathbb{P}$ as:

$$dS_t = \mu S_t \, dt + \sqrt{v_t} S_t dw_t^1 \tag{3.1}$$

$$dv_t = \kappa[\theta - v_t]dt + \sigma\sqrt{v_t}dw_t^2 \quad (3.2)$$

Equation (3.1) assumes the underlying asset price follows the diffusion process at time t where where μ is the drift parameter, and dw_t^1 is a standard Wiener process (i.e. random walk). In equation (3.2) the volatility $\sqrt{v_t}$ itself follows a diffusion process where dw_t^2 is a Wiener process, ρ defines the correlation between dw_t^1 and dw_t^2 where, $dw_t^1 dw_t^2$ = pdt with $\rho \epsilon [-1, 1]$. Using Ito's lemma, the variance process can be written as seen in (3.2) as the volatility follows an Orstein-Uhlenbeck (mean-reverting) process.

For simplicity at this stage, under the risk-neutral measure, variations of (3.1) and (3.2) are given by:

$$dS_t = rS_t\,dt + \sqrt{v_t}S_t\,dw_t^1 \quad (3.3)$$

$$dv_t = \kappa[\theta - v_t]dt + \sigma\sqrt{v_t}dw_t^2 \quad (3.4)$$

where the (3.3) provides the dynamics of the stock price: S_t denotes the stock at time t, r is the risk-neutral drift and is found in markets, $\sqrt{v_t}$ = 0. The second equation provides the evolution of the variance where θ is the long-run mean of the variance, k is the speed of mean reversion parameter, σ is the volatility of volatility and ρ is the correlation between the two Brownian Motions W_t^1 and W_t^2 .

4. The Monte Carlo Method

In order to price an exotic security, one would be poised with the task to numerically calculate $\mathbb{E}[f(S_t)]$ for a payoff function $f : \mathbb{R}^N \to \mathbb{R}$, where a stochastic function describes the underlying asset S:

$$dS_t = a(S_t, t)dt + b(S_t, t)dW_t \quad (3.5)$$

where W_t is a normal Weiner process. One could engage a partial differential equation(PDE) criteria for the derivative, but this is ineffective, laborious and given the payoff path, could be too computationally complex. Given these circumstances and/or if the stochastic differential equations do/do not yield a closed-form solution, one could utilize a Monte Carlo simulation by discretizing the time interval and simulating the state process dynamics on this discrete-time grid. In this scenario, it would be beneficial to implement a discrete-time approximation of (3.5) to acquire a Monte Carlo estimation of $\mathbb{E}[f(S_t)]$. The Monte Carlo Simulation simulates many sample trajectories of the state variables e.g., stock price, volatility, and interest rates. The generations of trajectories and the payoff of the derivative are evaluated for each sample

trajectory, discounting and taking the mean over all trajectories gives an expected payoff $\mathbb{E}[f(S_t)]$ of the derivative price.

Let us refer to (3.3), Heston's underlying asset price:

$$dS_t = rS_t\,dt + \sqrt{v_t}S_t\,dw_t^1 \tag{3.6}$$

where there are no dividends and r is deterministic. In order to smoothen the discretization transition, let us work with the underlying asset's log stock price $x_t = ln[S_t]$. With the application of Ito's lemma to (3.6), in conjunction with the variance process instituted earlier, the Heston model we will look at is:

$$dx_t = \left[r - \frac{1}{2}v_t\right]dt + \sqrt{v_t}dW_t^1 \tag{3.7}$$

$$dv_t = \kappa\left[\theta - v_t\right]dt + \sigma\sqrt{v_t}dW_t^2 \tag{3.8}$$

We have to discretize these SDE's in order to simulate them but, we must ensure that we pinpoint the correct time-discretization of (3.7) and (3.8). Although there are no closed-form solutions for equations (3.7) and (3.8), we can use the most uncomplicated and painless method, the Euler discretization technique. While it is easy to implement, it does not come without its problems. When transitioning from continuous-time to discrete-time processes, the draws the variance equation produces will be negative values of volatility for which, posses a significant problem. While there are many ways to rectify this, we shall work directly with the natural logarithm of the variance. By utilizing Ito's lemma once more, we get the dynamics of $\ln(v_t)$ as follows:

$$d\ln(v_t) = \tfrac{1}{v_t}(\kappa(\theta - v_t) - \tfrac{1}{2}\sigma^2)dt + \sigma\tfrac{1}{\sqrt{v_t}}dW_t^2 \tag{3.7}$$

Now we can execute the Euler discretization technique and we now have the following discrete time solutions:

$$\ln(S_{t+\Delta t}) = \ln(S_t) + \left(r - \frac{1}{2}v_t\right)\Delta t + \sqrt{v_t}\sqrt{\Delta t}\varepsilon_{S,t+1} \tag{3.8}$$

$$\ln(v_{t+\Delta t}) = \ln(v_t) + \frac{1}{v_t}\left(\kappa(\theta - v_t) - \frac{1}{2}\sigma^2\right)\Delta t + \sigma\frac{1}{\sqrt{v_t}}\sqrt{\Delta t}\varepsilon_{v,t+1}$$

Shocks to the volatility, $\varepsilon_{v,t+1}$ are correlated with the shocks to the stock price process, $\varepsilon_{S,t+1}$. This is denoted by $\rho = Corr(\varepsilon_{v,t+1}, \varepsilon_{S,t+1})$ and the relationship between shocks can be written:

$$\varepsilon_{v,t+1} = \rho\varepsilon_{S,t+1} + \sqrt{1-\rho^2}\varepsilon_{t+1} \quad (3.9)$$

where the ε_{t+1} are iid Standard Normal variables that each have zero correlation with $\varepsilon_{S,t+1}$.

While we only looked at the basic Euler discretization scheme, there are plethora of ways to overcome the negative value of volatility such as the Full Truncation Euler, Milstein, etc... It is worth noting that in Brodie and Kaya's model [3], they offer a method where the sample stock price and variance from the exact distribution is utilized that prompts a neutral estimator of the price of a derivative. The Brodie and Kaya method, when compared with the Euler discretization, achieves a faster convergence rate of the error. While their model converges faster and is more accurate, it is very complicated and difficult implement. Let us take a brief look at these "improvements" to the basic Euler scheme.

A simple scheme, Lord, Koekkoek, and Dijks' Full Truncation Euler [4][18], mitigates the negative variance by making absorption ($V(t) = 0$) and reflection $(V(t) = |V_{(t)}|)$ and takes the following form:

$$X_{t+\Delta} = X_t + \left(r - \frac{V_t^+}{2}\right)\Delta + \sqrt{V_t^+\Delta}W^1 \quad (4.0)$$

$$V_{t+\Delta} = V_t + k(\theta - V_t^+)\Delta + \sigma\sqrt{V_t^+\Delta}W^2 \quad (4.1)$$

The procedure for *V* is allowed to become negative and so it follows that the next time step for *V* becomes deterministic with an upslope drift of $\kappa\theta \geq 0$. It turns out that this model is really simple and easily executed. In addition, it has been found to produce the smallest discretization bias of all plain Euler schemes.

Another "improvement" of the Euler scheme is Milstein's. While there are numerous variations of the Milstein methods, we shall look at the (1974) version [5][6]:

$$V_{t+\Delta} = V_t + k(\theta - V_t^+)\Delta + \sigma\sqrt{V_t^+\Delta}W^2 + \frac{\sigma^2}{4}\Delta(W_2^2 - 1) \quad (4.2)$$

where Milstein adds an additional term for the variance procedure for which, has an improved strong order of convergence as opposed to Euler's scheme.

Nevertheless, whichever method is chosen to approximate a continuous-time process by a discrete-time process will incorporate bias into the simulation estimator. In Brodie and Kaya's model, they provide a non-bias model that generates the process from the distribution S_t, by constraining the values simulated by the variance process. This notion is fairly straightforward and is demonstrated by, revising (3.3) and (3.4) as the following:

$$S_t = S_u exp[r(t-u) - \frac{1}{2}\int_u^t V_s ds + \rho \int_u^t \sqrt{V_s} dW_s^1 + \sqrt{1-\rho^2}\int_u^t \sqrt{V_s} dW_s^2]$$

$$V_t = V_u + \kappa\theta(t-u) - \kappa \int_u^t V_s ds + \sigma_v \int_u^t \sqrt{V_s} dW_s^1 \quad (4.3)$$

where in order to sample from the distribution of (S_t,V_t) we,

1. Generate V_t given V_u (4.4)

2. Generate $\int_u^t V_s ds$ given V_t and V_u

3. Calculate $\int_u^t \sqrt{V_s} dW_s^1$ from (4.3)

4. Generate S_t given $\int_u^t \sqrt{V_s} dW_s^1$ and $\int_u^t V_s ds$

The predominant aspect of their model that they depend upon is given V_u, $u < t$, V_t is up to a scale factor, a non-central chi-squared [7]:

$$V_t = \frac{\sigma_v^2(1-e^{-k(t-u)})}{4\kappa} X_d^2(\frac{4\kappa e^{-k(t-u)})}{\sigma_v^2(1-e^{-k(t-u)}} V_u) \quad (4.4)$$

In spite of the fact that, Brodie and Kaya have created an exact method that eliminates bias and it works, it comes a a high price, where it is a extrememly complex and tedious process among other noted drawbacks. While the exact method works and is deserving of an honerable mention for providing a method that excludes bias, it shall not be mentioned throughout the remainder of upcoming chapters.

While we mentioned different discretization methods in this chapter, we will only be working with the Euler scheme for the remaining chapters. For simplicity, the Euler scheme was chosen as it performs "reasonably well" and is easy to implement and thus, we will explore it to a greater extent in later chapters.

5.Python Programming Language

As the complexity in financial modeling arises, so does the need of the technology that executes a variety of methods, ranging from simple models i.e. Euler to complex exact method models. Typically, these models are executed in Fortran, C or C++. Although, these programming languages excel at performance, they lack in other important aspects of being a “complete” package. As there are consistent innovations in technology, shouldn’t your programming language be a reflection of those innovations?

The financial industry is in a fascinating time. Now, it is a cliché to say that in order for a bank to survive and thrive, they will have to embrace technology and innovation. They have already bought into this idea as they have essentially become technology firms. So, if that holds true, then the system/application software should be a reflection of that, which is why we are seeing an increase in demand for Python in finance. The reasoning behind this increase in the demand for Python for finance is because it allows for quick, powerful and effortless structuring of programs. Ideas translate quickly to the computer, which is why Python has been called’’ programming at the speed of thought [8].”

From the Python website, you can find Python's executive summary [9]:

> Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python’s simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

This adequately sums up the reasoning behind Pythons evolution to becoming a paramount programming language as of late. Additional reasoning for the support of Python is because of its characteristics, libraries and tools. Python is open source, interpreted, multi-paradigm, multipurpose, cross-platform, dynamically typed, indention aware and garbage collects[10]. Python has all of the scientific computing tools one could need. Python is a “one-size-fits-all” program. It supports a wide range of programmers such as your casual programmer, scientific developers and professional software developers.

As we mentioned earlier, technology is ever-evolving and the financial industry is diversifying and becoming more technological as opposed to being *simply* a financial institution. Technology is becoming a significant investment tool that has yield a prospective competitive edge or lack thereof. Not only has technology increased innovation, efficiency, speed, within the financial sector, it also increased the demand for computational power and real-time analytics.

6.Implementing Option Pricing Models in Python

The prior chapter provided context pertaining to technologies advancement of the financial sector. In this chapter we shall provide a more in-depth examination as to how Python contributes to the growth of the financial industry by providing insight and the implementation of an option pricing model in Python with the ultimate goal being the creation of a simple, yet interacted, Heston option pricing model originating from the culmination of the forthcoming chapters.

Now, allow us to demonstrate the ease of usability of Python in a financial context. Notice that Python's syntax is really close to the mathematical syntax utilized to outline financial algorithms and because the desired use of this code is intended to be reused often, this example often gets organized into modules (or scripts), which are single Python (i.e., text) files with the suffix .py. Modules of this type typically look like the model in (4.5) and could be saved as a file named Naive_MC.py. We will examine a simple naïve Monte Carlo option pricing model as it can effortlessly manage high-dimensional problems where the complication and computational requirements, both, increase in sequential manner.

The caveat of the Monte Carlo method is that it quite computationally strenuous. Thus, it follows that, it is imperative to implement Monte Carlo algorithms as efficiently as possible. The following examples explicates multiple implementation strategies in Python and provides three different viable implementation options for a Monte Carlo-based valuation of a European option. The three methods are[10]:

Pure Python

Coding in pure Python utilizes the basic library—i.e., the use of built-in only Python capabilities that encapsulates the standard packages and libraries in order to implement the Monte Carlo valuation.

Vectorized with NumPy

This implementation utilizes the capabilities of *NumPy* to make the implementation more compact, easier to read (and maintain), and significantly faster execution times.

Fully Vectorized NumPy

A combination of different mathematical formulations with the vectorization capabilities of *NumPy* to get an even more compact version of the same algorithm.

While we will only focus on the vectorized and fully vectorized versions with *NumPy*. It is worth noting that, in regards to writing code in Pure Python, the only difference in coding between the forthcoming vectorized Monte Carlo with *NumPy* and a Pure Python solution is the importation of *NumPy* in conjunction with omitting a few simple lines of code that would be a required input in Pure Python. Thus, it is more beneficial to reap the benefits of *NumPy* by importing it whilst eliminating a few lines of code. Let us begin with the simple naïve Monte Carlo pricing algorithm:

For i = 1, 2, ….,M

1.Set $S_{0,i}$ = spot price

2.Simulate from: $S_{T,i} = S_0 exp[(r - \frac{1}{2}\sigma^2) + (\sigma\sqrt{T}\varepsilon)]$

3. Apply the option payoff (in this case for a Call): $C_{T,i} = max(S_{T,i} - K, 0)$

4. $\hat{C}_0 = \exp - r \times T \left[\frac{1}{M} \sum_{i=1}^{M} C_{T,i} \right]$

where #4 provides the numerical Monte Carlo estimator for the value of the call option and the parameters are as follows:

- spot = 41
- strike = 40
- expiry = 1
- rate = .08
- sigma = .30
- M = 50000

Vectorized with NumPy and in a loop (4.5)

```
import numpy as np

## set up of the parameters
spot = 41.0
strike = 40.0
expiry = 1.0
rate = 0.08
sigma = 0.30
M = 50000

## The main simulation loop
spotT = np.empty((M, ))
callT = np.empty((M, ))

for i in range(M):
    z = np.random.normal(size=1)   ## pseudorandom numbers
    spotT[i] = spot * np.exp((rate - 0.5 * sigma * sigma)* expiry + sigma * n
p.sqrt(expiry) * z)
    callT[i] = np.maximum(spotT[i] - strike, 0.0)

price = np.exp(-rate * expiry) * callT.mean()
print("The Call Option Price is: {0:.3f}".format(price))
```

```
The Call Option Price is: 6.947
```

Take notice that the estimated option value itself relies on the pseudorandom numbers generated and of the importation of *NumPy*.

The major components we will consider in this code is the for loop and the importation of *NumPy*. The for loop implements the repeated execution of code based on a loop variable and in this case, M. The Monte Carlo estimator is then calculated by utilizing Python's list comprehension syntax. Python's list comprehension syntax is much more compact and is pretty close to the actual mathematical notation of the Monte Carlo estimator.

NumPy, short for Numerical Python, is a third-party package for high performance scientific computing in Python provides a powerful multidimensional array data types, called ndarrays, as well as a comprehensive set of functions and methods to manipulate them and implement(complex) operations on such objects. To hone in on it a little further, there are essentially two substantial benefits of using *NumPy*[10]:

Syntax

NumPy essentially permits implementations that are more compact as opposed to, when done in pure Python and that are simpler to read and maintain. With that being said it provides standard mathematical functions for quick operations (vectorization) and tools that allow access to very efficient low-level C, C++, and Fortran codes.

Speed

Because *NumPy* code is predominately implemented in C or Fortran, this makes *NumPy*, considerably faster than pure Python.

The root of a more compact syntax arises from the characteristics of *NumPy* as it provides Python with robust vectorization and broadcasting capabilities. This is comparable to using vector notation that is used in mathematical vectors for large vectors or matrices. Let us see how we can make this code more compact, faster and more efficient by omitting the for loop for full vectorization:

Fully Vectorized NumPy (4.6)

```
import numpy as np

spot = 41.0
strike = 40.0
expiry = 1.0
rate = 0.08
sigma = 0.30
M = 1000000
z = np.random.normal(size=M)
spotT = spot * np.exp((rate - 0.5 * sigma * sigma)* expiry + sigma * np.sqrt(
expiry) * z)
callT = np.maximum(spotT - strike, 0.0)
```

```
price = np.exp(-rate * expiry) * callT.mean()
print("The Call Option Price is: {0:.3f}".format(price))
The Call Option Price is: 6.965
```

Vectorization is not only substantially faster than pure Python, it is also noticeably more compact. The estimated Monte Carlo call option value is very close to the standard that was set from (4.5).

The vectorization is quite clear now as we notice the removal of the for loop and a substantial increase in numbers to be generated by our pseudorandom numbers generator. The number generation requirement has increased from 50000 to 1000000 numbers. All of which, have been done so by a single line of code:

z = np.random.normal(size=M)

The result of the vectorization via *NumPy* is code that is typically more compact, simpler to read and maintain and faster in regards to execution times. These characteristics are necessities when applied to financial applications. Now that we understand the importance of vectorization and *NumPy,* let us expand our knowledge of Python programming and its benefits to finance.

7.Object-Oriented Programming

Now we see the benefits that *NumPy* and vectorization have on the application of complex mathematical and financial algorithms, we shall build upon that knowledge by introducing what is considered the foundation of modern programming languages, object-oriented programming (OOP). But before we do, we must introduce Python's functions in order to completely understand the relevance of OOP and to further our required skillset targeted towards our goal of implementing the Heston option pricing model in object-oriented Cython. Take note that in this chapter as well as the following two chapters, we will provide basic examples of the chapter subjects with the intention of applying those subjects in a financial application at the completion of chapter 9.

7.1 Functions

Functions are organized reusable pieces of code. They provide you with the ability to assign a name to a block of statements. Thus, allowing you to run a block of code utilizing the designated name anywhere in your program and as often as you would like. Functions underline the separation of the utility of a program into individualistic, interchangeable **modules**, such that each module holds every requisite to execute only one feature of the desired functionality. This is what is recognized as *calling* the function. Functions are essentially *the* single most crucial element in any programming language and it

remains true for Python as well. Although there are various aspects of functions, we shall focus on the aspects that we will utilize in the upcoming chapters.

Functions are defined using the keyword def followed by the *identifying* name for the function, then by parentheses (( )) which may surround some input parameters or arguments, and then the insertion of a colon (:) that ends the line of code. Now we implement the block of statements that are are encapsulated within the function. Let us provide a general illustration of this were we employ the widely recognized Hello World example and save it as well:

Example (save module as helloworld.py): (4.7)

```
def saying_hello():
    ## Block belonging to the function
    Print('hello world')
    ## End of the function

saying_hello()  ## Calling the function

$ python helloworld.py

hello world     ## Output
```

The syntax above provides the definition of the function called saying_hello. The saying_hello function takes no parameters or arguments and thus providing the justification for the emptiness expressed between the parentheses. Parameters assigned to functions are simply inputs where we can assign different values to it and obtain the corresponding results.

7.2 Object-Oriented Python

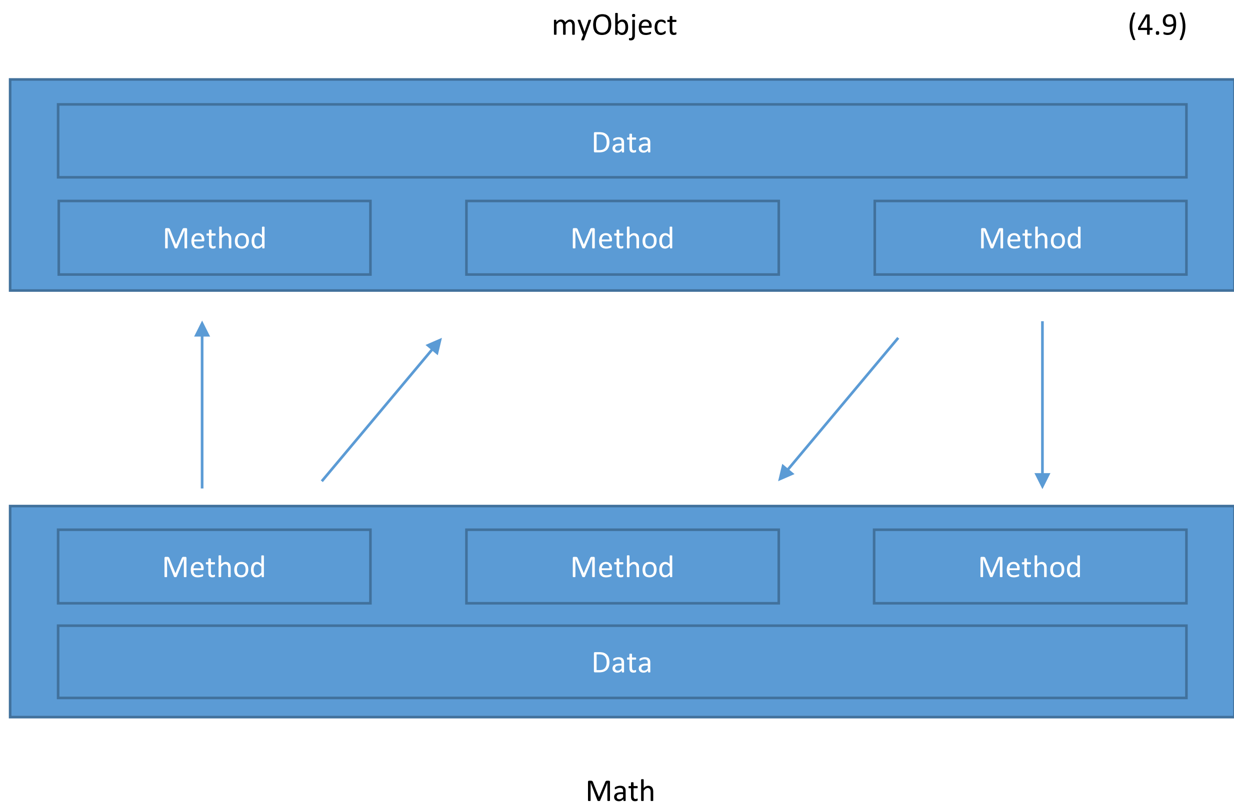
Program (4.7), was built around a function and this method of construction is called procedure programming. One can organize their program in a different manor, which is to combine *both* data and behavior. This, in its basic definition is called an *object*. Object-oriented programming (OOP) allows objects to fully utilize other objects' services as well as inherit their functionality, promotes code portability and reuse [11]. While there is nothing intrinsically wrong with procedure programming, there are definitely opportunities and instances that merits the use of OOP. For instance, when writing large programs or reusing code, object-oriented programming is extremely valuable.

This section provides an overview of the basic OO concepts. While the concepts expressed here are the basic essentials, we shall explore how each concept operates and implement them in our Heston option pricing model in later chapters. But first let us differentiate procedural versus OOP.

Procedural Versus OOP

In procedural programming, for example, code is designated towards completely different functions or procedures[11]. Then, these procedures or functions (behaviors) become, what Weisfield calls, "black boxes," for which, inputs go in and outputs are the result. The data is deposited into unconnected structures and is controlled by these behaviors as shown in example (4.8)[11]:

where, on the other hand, one can easily see how OOP provides more developer options such as allowing some members of an object, both attributes and methods, to be hidden from other objects in the course of interaction. Let us look at the best demonstration of object-to-object communication[11]:



For instance, an object **Math** holds two integers, **Itsint1** and **Itsint2**. Additionally, the **Math** object holds the required methods to set and recover the values of **Itsint1** and **Itsint2** while

possibly holding a method called sum ( ) that combines the two integers. When the combination of attributes and methods meet in the same entity, this is called encapsulation and we can regulate access to the data in the Math object. When we restrict access to particular attributes and methods, we are *data hiding*.

So, the best way to comprehend the object-to-object communication in (4.9) is to imagine that-for instance, **myObject**-would like to have access to the sum of **Itsint1** and **Itsint2.** It asks the math object: **myObject** sends a message to the **Math** object. Ultimately, it is just sending a message to the **Math** object's **sum** method. Then the **sum** method provides the value to **myObject**. This is extremely useful because **myObject** has no need to know how the sum was calculated. Now, with this methodology, we can change how the **Math** object calculates the sum withholding the need to change **myObject** because all we desire is the sum. We *do not care* about how it was calculated. Now that we understand the differences between procedure programming and OOP, we come to the conclusion that the primary benefit of OOP is that the data and the operations that control the code are both encapsulated within the object. But, what is an object?

Objects

Objects are the essential foundation of OOP. OOP in itself is essentially an assortment of objects. These objects are comprised of object data and object behavior.

Object data are the *attributes* of the data that is contained within an object that depicts the condition of the object. For example, let us look at a teacher. A teachers' attributes could be date of birth, gender, phone number, salary, and so on. Attributes contain information that distinguishes different objects from one another. We shall analyze attributes more closely later in the discussion. Object *behaviors* are behaviors of an object that tell what the the objects abilities are or what it can do. These behaviors are held within *methods*, for which you solicit methods by calling to it. Really, though, when it comes down to it, attributes and behaviors are essentially variables (class and instance) and functions that pertain to an object or a class.

Class

Classes and objects go hand-in-hand. They depend on one another to complete a task. Although, objects are listed first in this thesis, they do not come first. The class comes first because it is the blueprint for an object. A class defines an object or is a template from which objects are created and is callable. When we instantiate an object (create an object), we use a class as the foundation for how the object is constructed. It is difficult to explain one without the other. Ultimately, an object cannot be instantiated without a class. Okay, let us now demonstrate a class and how it functions. An easy and straightforward example of a basic class is as follows:

```
class World:                                                      (5.0)

    pass   ## This is an empty block of code

w = World()
print(w)
```

For this example, we have created a class by using the class keyword and followed by the name of the class (World). The body of the the class is recognized by the indented block statements. For our example, this block statement is empty per our pass statement. We then created an object/instance of this class that utilizes the name of the class (World) that is then preceded by empty parentheses. Then we print the type of variable. Now that we have established a basic class example, let us refer to the fact that objects can have functionality by utilizing the functions that are apart of a class. These were called *methods*. These class methods are essentially the same as normal functions, but they have a requirement that must be fulfilled. The requirement is that they have to have an unassigned parameter value with an additional name at the beginning of the the parameter list. This variable alludes to the object *itself* and is designated by the name *self*.

The Self

The key aspects of the *self* are that it is not advisable to change its' name, even though you can, you should not and the fact that you do not have to worry about assigning it a value when calling the method as Python has taken care of this. How Python gives the value of *self* can be illustrated by the following example, for instance, let us say that we have a class names **MyClass** with **MyObject** being an instance of that class. When MyObject.method(param1, param2), a method of **MyObject** has been called, Python converts it into MyClass.method(MyObject, param1, param2) automatically. This is what is unique in regards to, the *self*. This implies that even if you have a method that does not consider any parameters or arguments, you will always have at least one, the *self*.

With the recognition that classes and objects can have methods, similar to functions with the addition of the *self* variable, let us look at an example:

```
class World:                                                      (5.1)
    def hello_world(self):
        print('Hi. How is it going?')

w = World()
w.hello_world()
Hi. How is it going?  ## Output
```

In example (5.1) we can see how the *self* operates. We can see how hello_world does not consider any parameters, but yet the within the function definition, we have the *self*.

__init_ Method

While we will not cover all of the special methods that are contained within Python and significant, we shall consider the __init__ method as we will utilize this method in our upcoming simulation.

The __init__ method is a special method in classes, that represents a constructor in Python. A constructor is a method that shares the same name as the class and has no return type[11]. The __init__ methods primary use is to initialize Python packages whenever an object class is constructed. So, whenever we instantiate an object, the first argument we will use is the *self*. Consider the example below:

(5.2)

```
class World:
    def __init__(self, name):
        self.name = name
    def hello_world(self):
        print('Hi. My name is', self.name)

w = World('Earth')
w.hello_world()
Hi. My name is Earth  ## Output
```

You can see in example (5.2) we define the __init__ method as taking the *self* and *name*. We also generated a *name.* Self.name implies that the object self has "name" apart of it and the other *name* is a local variable. When we made w, of the class World, we did so by utilizing the class name for which, the arguments contained within the parenthesis: w = World('Earth'). We did not clearly call the __init__ method and that is why this method is very unique and notable. Now, we can utilize *self.name* in our methods, for which, it is displayed in the hello_world method.

Encapsulation and Data Hiding

Encapsulation as stated earlier, is when the combination of attributes and methods meet in the same entity. Thus, data hiding is a major component of encapsulation. One of the main benefits of using objects is that they do not have to completely disclose all of their attributes and behaviors. In OOP construction, an object should only disclose the necessary interfaces needed in order for other objects to communicate with it.

To provide context, for example, let us say we have an object that calculates the multiplication of two numbers. The object has to provide an interface in order to acquire the result. But, the internal attributes and set of rules specifying how to multiply the numbers does not need to be readily available to petitioning object. Classes constructed with encapsulation at the forefront are considered extremely powerful. Now, we will look more closely at interface and implementation as they are underlying foundation of encapsulation.

The interface provides communication between objects where each class design determines the interfaces for the correct instantiation and operation of objects. Any behaviors that the object presents, has to be implored by a message sent by one of the supplied interfaces. This interface should outline how users of the class communicate with the class. Methods that are components of interface should not be constructed as **public** but as **private**. Every attribute should be recognized as **private** in order for data hiding to function effectively because then the attributes are under no circumstances part of the interface. If attributes are expressed as being **public**, the notion of data hiding shall cease to exist. It is worth mentioning that there are interfaces to the classes along with the methods. But, do not mistake one for the other. Public methods are the interfaces to the classes and the interfaces to the methods are associated with how you implore them. Regarding the implementation, solely the public attributes and methods are regarded as the interface.

The end-user should not be able to see any portion of the implantation. They should only be able to interact with the object only via class interfaces. Although, there are often times when the methods should be hidden as well. Resuming the example of the object that calculates the addition of two numbers, the end-user ultimately is not concerned with how the multiplication was calculated, provided that the answer was right. Thereby, the implementation is allowed to change, but not at the expense of having an impact on the end-user's code[11]. This would allow the firm that makes the software that calculates the multiplication example from above to have the ability to change their formula (maybe because their new software update provides quicker calculations) and it would not affect the result.

Inheritance

One of the most notable aspects of OOP is being able to reuse code. *Inheritance* allows you to not only reuse code but it allows you define relationships between classes and thus, providing one with a comprehensive design, by allowing the organization of classes while taking similarities of different classes into account.

Inheritance allows a class to inherit characteristics (attributes and methods) of another class. Inheritance allows for the formation of a completely original class by abstracting out similar attributes and behaviors. Inheritance is best thought of as implementing a superclass and subclass relationship between the classes.

Superclasses are the derived class. This class holds all of the attributes and behaviors that are mutual to classes that inherit from it. The subclasses inherit the characteristics from the superclass and they can be used in the same fashion as if they were defined in the subclass.

Before this gets to complicated, here is an example. Say you want to code a program that has the observe the father and the son of a family. They share mutual attributes such as

name and age. They also have distinct attributes such as salary for the father and, toy for the son.

We could create two independent classes but, we would not necessarily want to do that because it could get very complex. So, let us do this in a simpler fashion. We could create a mutual class called *FamilyMember* and then would could have the father and son classes i*nherit* from the main superclass. There are quite a few benefits to this technique. For one, if we decided to change any functionality of the superclass, the subclass would recognize this change as well. An additional benefit would be that if we could refer to a father or son object as a *FamilyMember* object which could be practical in some circumstances such as computing the number of family members. This would then be what is known as *polymorphism*.

Polymorphism is one of the, if not the major benefit of OOP. It simply means many shapes. It is a principal feature of class definition in Python that is used when you have mutually named methods across various superclasses or subclasses. This is powerful because it provides access for functions to use objects of any of these polymorphic classes without the requirements of having to be aware of distinctions across the classes. Polymorphism is executed via inheritance, with subclasses inheriting from superclasses or overriding them. We will get more into polymorphism in later chapters.

Now, back to our example, the father and son class would be the subclasses and we could make the addition of distinct characteristics to these subclasses. Because the father and son both inherit from the *FamilyMember* superclass, this relationship is referred to as an *is-a* relationship because a father is a family member and son is a family member. When a subclass inherits from a superclass, it has all of the capabilities as such. Let us look at a program example[12]:

```
class FamilyMember:                                                        (5.3)
      ## Represents any family member
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print(‘ (Initialized FamilyMember: {})’.format(self.name))

    def tell(self):
      ## Tell my details
        print(‘Name:”{}” Age:”{}”’ .format(self.name, self.age), end=” “)

class Father(FamilyMember):
      ## Represents a father
    def __init__(self, name, age, salary):
        FamilyMember.__init__(self, name, age)
        self.salary = salary
             print(‘ (Initialized Father: {})’.format(self.name))
    def tell(self):
        FamilyMember.tell(self)
        Print(‘Salary: “{.d}”’.format(self.salary))
```

```
class Son(FamilyMember):
            ## Represents a son
    def __init__(self, name, age, toy):
        FamilyMember.__init__(self, name, age)
        self.toy = toy
        print(' (Initialized Son: {})'.format(self.name))

    def tell(self):
        FamilyMember.tell(self)
        print('Toy: "{.d}"'.format(self.toy))

f = Father('Mr. Davis', 50, 50000)
s = Son('Ronald', 10, 200)

# Print a blank line
print()

members = [f, s]
For member in members:
      ## Works for both Father and Son
      member.tell()

Output:

(Initialized FamilyMember: Mr. Davis)
(Initialized Father: Mr. Davis)
(Initialized FamilyMember: Ronald)
(Initialized Son: Ronald)
Name:"Mr. Davis" Age:"50" Salary:"50000"
Name:"Ronald" Age:"10" Toy:"200"
```

So, you now see. In order to use inheritance, we specified the superclass names in a tuple following the class name in the class definition. We then saw that the __init__ method of the superclass was clearly called utilizing the self variable so that we could then initialize the superclass portion of the object. We saw that the methods of the superclass can be called by prefixing the class name to the method called and then by passing through the *self* variable with any parameters.

Take note that we can treat instances of *Father* or *Son* as just instances of *FamilyMember* when we utilize the tell method of the *FamilyMember* class. Also, notice that we call the *tell* method of the subclass as opposed to the *tell* method of the superclass. This is easy to comprehend in the sense that Python consistently begins looking for methods in the actual type and in our scenario, it does. If it cannot find the method, it begins searching through the superclass, one after the other in the order that they were identified in the tuple of the class definition.

So far, we have seen inheritance from one class or single-inheritance. It is worth mentioning that we can have inheritance from multiple classes. This is called multiple-inheritance. It is an aspect of OOP that allows a class to inherit attributes and methods from more than one superclass. Let us have a brief look:

```
class MyClass(Superclass1, Superclass2, Superclass3,…):        (5.4)
    def __init__(self):
        Superclass1.__init__(self)
        Superclass2.__init__(self)
```

It is pretty intuitive since we have just completed an example in inheritance. But make no mistake about it, it can become extremely complex really quick. We will discuss this further later on in upcoming chapters as we utilize this feature in our simulation. Back to our example. The *end* parameter is utilized in when we call the print function in the FamilyMemeber's *tell()* method to print a line and to permit the next print to proceed on the same line. This is a simple ploy to make *print* not print a new line symbol at the completion of printing. Now, we can see how inheritance can become quite large. So, let us briefly talk about abstraction.

Abstraction

When the *FamilyMember* and *Father* are complete, other family members, such as sons (or mothers, daughters and cousins), can be added fairly effortlessly. The Father class can also be a superclass to other classes, making those classes subclasses. For instance, one may have to abstract the *Son* class further, to include classes for Davis who is a son, Thomas who is a son and so on. Just as with *Son*, the Father class can be the parent class for Darian who is a father and Dillan who is a father. A class can have may subclasses and this is where multiple-inheritance takes place. Now, that we have some understanding of abstraction and its place in inheritance, let us talk about composition [11].

Composition

It is common for people to imagine objects as containing other objects, an object within an object. For instance, a telephone contains a chip and a LCD. Or a video game system contains drives, chips and controllers. Even though, the video game system is an object unto itself, the chip is also a valid object unto itself too. In regards to the video game system, we could take it apart and physically hold all of the parts it contains. The thing is that, the video game system contains all of the other objects within it. It is in this sense that objects are sometimes constructed from other objects and this is what is known as composition. Similarly, as with inheritance, there is abstraction with composition [11].

Abstraction

Similar to inheritance, composition supplies a technique for constructing objects. The difference between the construction of classes from other classes in inheritance and in composition is that inheritance allows one class to inherit from another class. So, we can then abstract out attributes and behaviors for mutual classes. The relationship is a key aspect in inheritances' abstraction. Inheritance has a *is-a* relationship. In composition, classes can also construction by implanting classes in other classes.

The difference really comes down to the fact that composition relationships are of the *has-a(n)* sort. Let us refer to the example in the prior section, a telephone *has-a* chip and it *has-a* LCD. A telephone is not a chip and nor is it a LCD. Let us consider the relationship between a smart phone and its' operating system (OS). We can see benefits from separating the two as they are apparent, such as writing a favorable user-friendly OS to another phone where its current OS is not as friendly. By building the OS separately, we can use the OS in an assortment of phones. This is just one benefit of doing so. There are plenty of other advantages. The thing we cannot say is that the OS *is-a* phone. So, instead, we use the expression *has-a* to describe the composition relationship because a phone *has-a(n)* OS [11].

While there is a lot we did not cover in our exploration of OOP, we did cover quite a bit and we can exit this section with a fundamental understanding of encapsulation, inheritance, polymorphism and composition. Python is a highly OO capable program and now that we have a good grip on understanding some of these OOP concepts, we can move forward with understanding design patterns and how we will implement them in our simulation.

8. Design Patterns in Computation Finance

With everything we have just learned about OOP, we can see how our code can become very complex and cumbersome really quick. Even if you have not realized it but, just because we understand the OO essentials does not mean that we will automatically be good at building flexible, reusable and maintainable systems. So, we need a way to structure our own applications in ways that are easier to understand, more maintainable and flexible [13]. This is where the knowledge of Design Patterns becomes very useful.

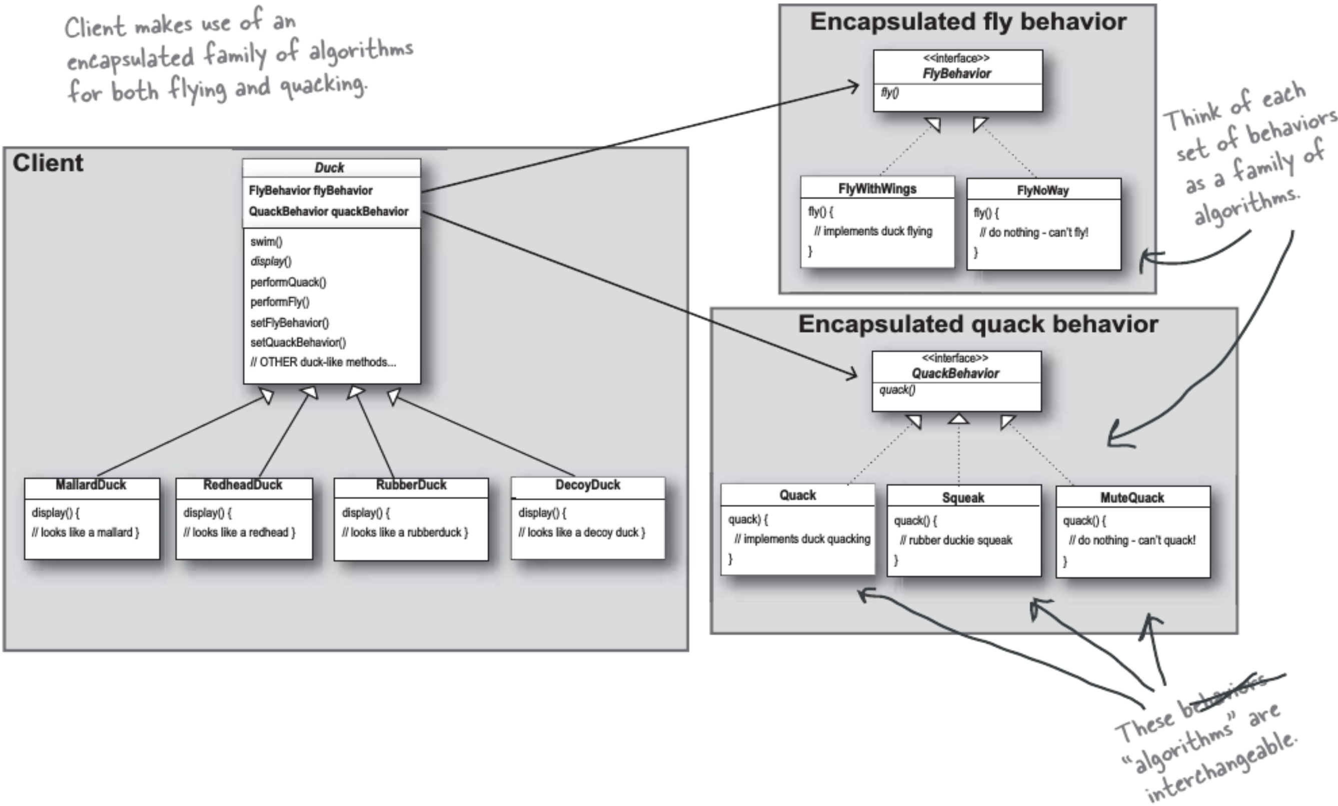
A design pattern is not something that directly goes into your code per se, it is a way of thinking. These patterns provide general solutions to design problems where once you have an understanding of how each pattern fits in with your application and OO, you can then apply them to your new designs and rework old code when you realize that that code is not flexible or easy to maintain. Design patterns show you how to construct systems that exhibit robust OO design qualities. They do this by telling us how to structure classes and objects more efficiently. While there are many design patterns, the design patterns we will focus on are the Facade and Strategy patterns.

8.1 Strategy Pattern

The strategy pattern enables the selection of an algorithm at runtime by defining a family of algorithms, encapsulating each one, and making them interchangeable [13]. The strategy pattern lets the algorithm vary independently from clients that use it [13]. In order to utilize the strategy pattern, we must be able to store a reference code in some data structure and have the ability to recover it and we can be accomplished via class or class instances in OOP [14]. Now, that we know what the strategy pattern is, let us provide an example.

For instance, a class that performs validation on incoming data may use a Strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice and other discriminating factors. These factors are not known until run-time and may require radically different validation to be performed. The validation algorithms (strategies), encapsulating separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication. The Strategy pattern is best understood in a visual manner, so let us have a look an example from Freeman & Freeman [13]:

(5.5)



Freeman & Freeman's example in (5.5) is a great example of the Strategy pattern put to use. It provides the illustration and big picture of a duck simulator design they created. The coding is not what is the big idea behind this example for us. The idea for us is to see the structure of the Strategy pattern, so when we do our simulation, we know how to organize it in a similar fashion. The example provides a completely reworked class structure for their duck simulator design. You can see how they have put last chapters OO discussion to use. You see that they have ducks extending Duck, fly behaviors implementing FlyBehavior and quack behaviors implementing QuackBehaviors. We can see how their thought process changed from thinking of duck behaviors as a *set of behaviors* to a *family of algorithms*. This illustration really provides a lot of context in regards to, the *relationships* between classes. We can really begin to establish

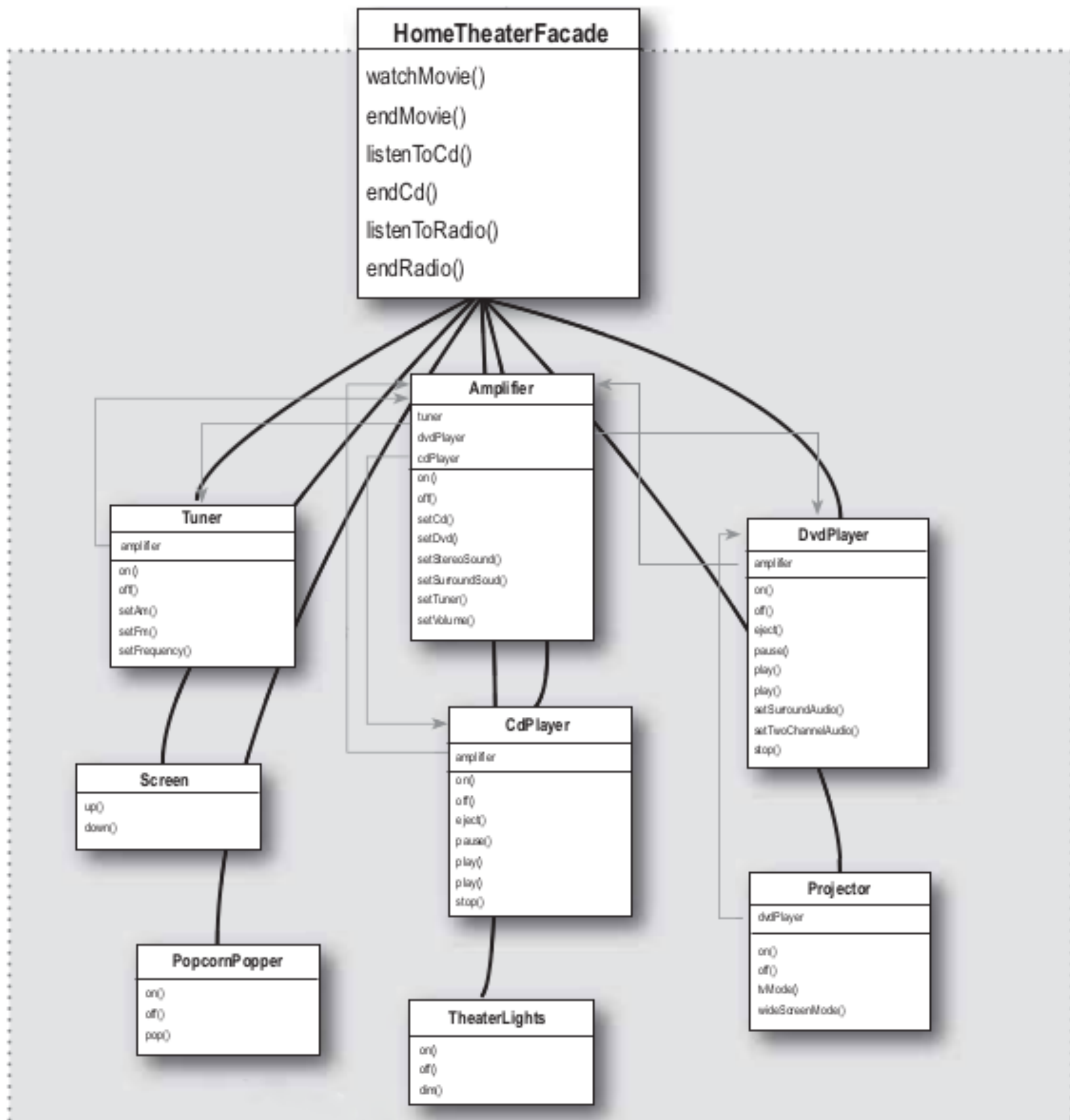
the difference between the *is-a(n)* and *has-a* relationships. Now, we can see how we should structure our complex simulation by way of the Strategy pattern, let us enhance and alter the interface. Let us hide all of the complexity of the classes behind a clean Facade pattern.

8.2 Facade Pattern

The Facade pattern provides a unified interface to a set of interfaces in a subsystem. The Facade pattern defines a higher-level interface that makes the subsystem easier to use [15]. Facade does this by making a software library easier to use, because it has convenient methods for common tasks, makes the library more readable, reduce dependencies of outside code on the inner workings a library, since most codes use the facade, thus allowing more flexibility in the developing system, wrap a poorly design collection of API's with a single well-designed API [15].

In order to utilize the Facade pattern, we must create a class that simplifies and unifies a set of more complex classes that belong to some subsystem [15]. Ultimately, we devise the facade with its subsystem and assign the subsystem to perform the work of the facade pattern. Do not mistake the Facade pattern's simplicity for weakness, it is a very powerful. Let us see it in action so we can clearly understand why it is powerful:

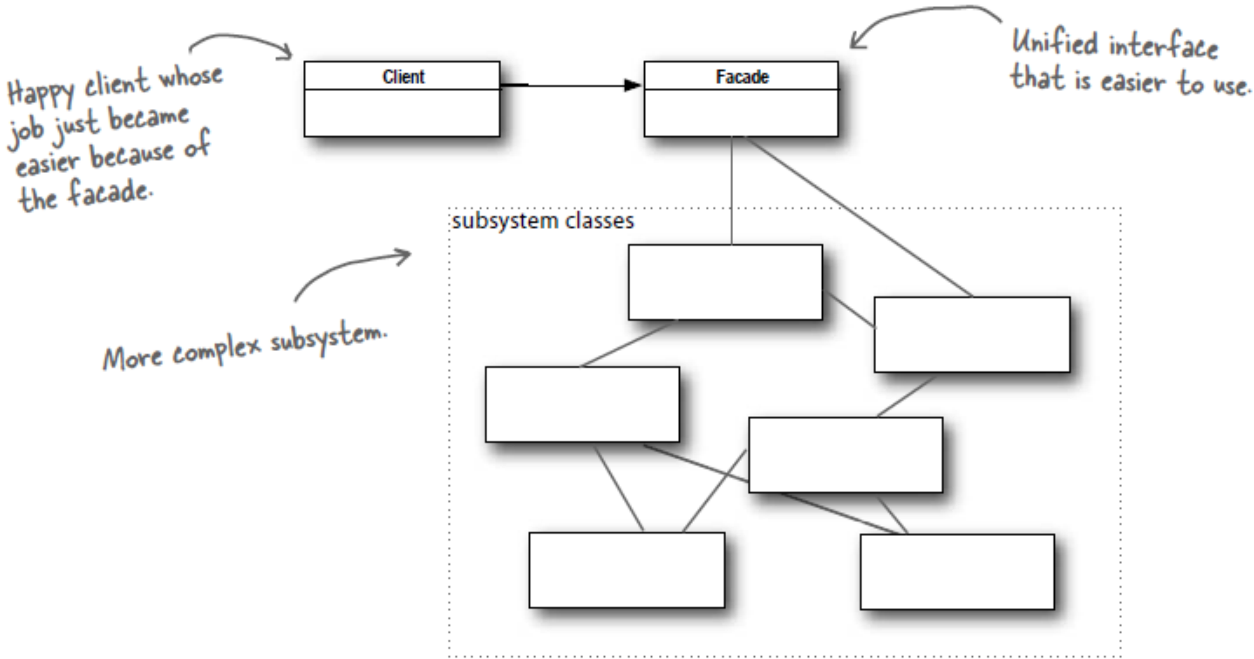
(5.6)



Freeman & Freeman provide yet, another excellent example of a Design pattern. In the example above, they have taken a complex home theater subsystem and simplified it by implementing a Facade Class that provides for a single, user-friendly interface [15]. We can see they created a HomeTheaterFacade that includes some simple methods. Then we notice the subsystem the Facade is going to make simpler and followed by the calling of the subsystem to apply the watchMovie() method. From this point, it gets even more simple. It is simple because the client calls methods on the HomeTheaterFacade and not directly through the subsystem itself. So, if we call one method from the Facade, it communicates with the the subsystem for

us. This is extremely powerful not only for it simplicity, but because you still have full direct access to the subsystem if you need. So, it is clear, what the purpose of this pattern's intention is. It is for a simplified interface that decouples a client from a subsystem of components, for which, it accomplishes and is demonstrated in the pattern's class diagram [15]:

(5.7)

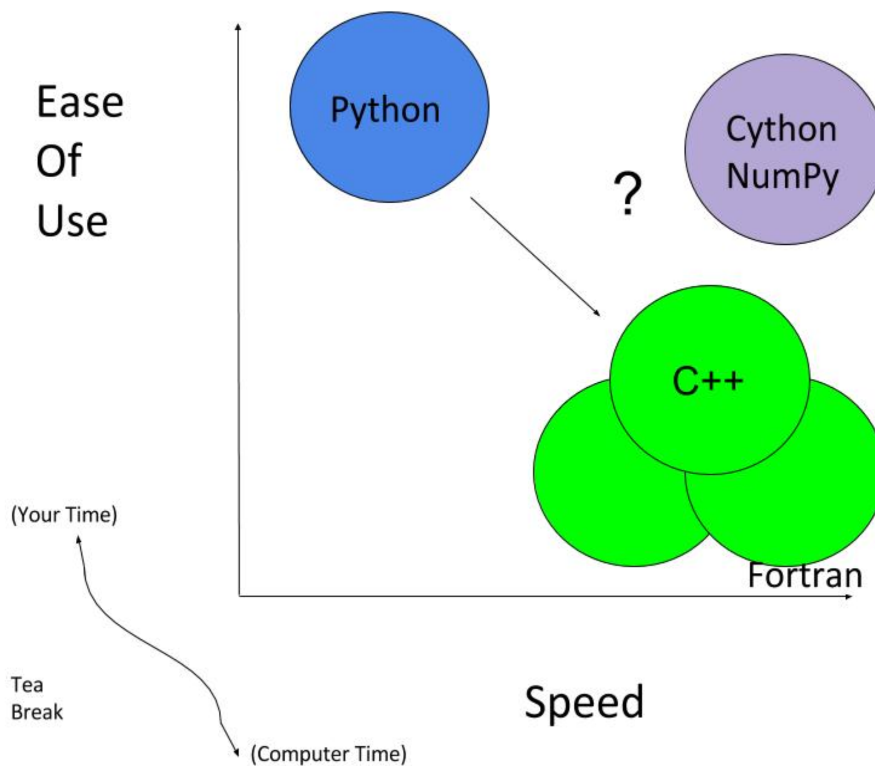


So, now that we have an understanding of how the Strategy and Facade patterns work, it is easy to see their place in computational finance. It is as simple as the patterns themselves. It is the simplification of extremely complex subsystems for which, not only become easily manageable and simple but, simpler for the client. For example, in earlier chapters we discussed technologies influence on finance and complex financial models. With the implementation of these Design patterns, we can have multiple programs of financial pricing models ready for client use. These complex models are programmed in an OO capacity and computational time is quick. A client says they want to price an exotic option. They do not care how it is calculated, they just want the price. With the implementation of these patterns, the client doesn't have to see what is going on in the background as we can restrict their access. But, what they do receive is an excellent interface that is simple and provides them with the answer they need without any regard to how it is calculated. They would simply place their input (parameters and market data) into the client portal, choose a pricing engine and then they receive their request calculated for them. Simple.

9.Cython

This purpose of this section is to provide the essentials of Cython as we will provide more details when we implement our simulation in Cython. For now, let us dive into the essentials of Cython. Cython is programming language that combines Python with the static type system of C and C++ [16]. Cython is an optimizing static compiler for both Python and the extended Python language, Cython. But, the primary feature of Cython is the ability to convert source code into optimized C/C++ source code via static type declarations that allows for very fast program execution in conjunction with an increase in developer velocity because of the utilization of Python-like syntax. This optimized source code can then be compiled into a Python extension module or a standalone executable. Cython is positioned between high-level Python and low-level C. So, we get the best of both worlds. We get Python's user-friendly syntax with the very powerful and quick speed of C. The motivation behind the use of Cython is best seen as illustrated by Stefan van der Walt's diagram from Sammi Mourya's presentation [17]:

(5.8)



Cython has secured its positioning as Python code is essentially legitimate Cython code. The difference in Python code to Cython code is very minimal. It is roughly the addition of a few key words that allows Python's language to gain access to C's type system that ultimately allows

Cython's compiler to produce quick and efficient C code. This means that if we have an understanding of Python and may come C or C++, we do not have to learn another interface. This will contribute greatly to increasing developer velocity.

Cython can contribute in many areas. We can use Cython to increase the performance of Python code or to apply Cython to C/C++ code that requires an optimized Python interface. In order make Python code faster, Cython complies Python source code with optional static type declarations to obtain *tremendous* performance improvements. We can use Cython to interface with external code and to create optimized wrappers that allows for the interfacing of C or C++ libraries with Python. Let us have a look at an example of the optimization of Python code with Cython's static type declarations. Also, taking note of the syntax differences[YouTube]:

Pure Python:

(5.9)

```
def f(x):
    return x**2-x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)

    return s * dx
```

Static Type Declarations Included:

(6.0)

```
def f(double x):
    return x**2-x

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)

    return s * dx
```

So, we can see that in (6.0), which would be saved as an .pyx file (Cython file as opposed to Python file) we have added a type to i, s and dx. So, we have made three static type declarations. We made these declarations by utilizing cdef as opposed to the def keyword. With these implementations, we saw in increase in speed of 35% over that of pure Python. This is the result of the code bypassing CPython's interface to run the code directly in C. So, the code that Cython has just generated has been optimized and now performs substantially better than its pure Python counterpart. It did this while maintain Python-like syntax. We can see it really is not that different. So, now let us briefly discuss cdef as we discussed the def keyword in an earlier chapter.

As a refresher, def is Python and is utilized for code that will be called straight from Python, utilizing Python object arguments and gives back a Python object. There are no c's before anything in this code. Now, cdef essentially places c's before its def to signify C use and it is for when we want Cython functions to be "C" functions, all types are *required* to be declared and when you want to generate code that is almost as fast as it can be. There is a middle ground in terms of the def keyword, cpdef.

Cpdef combine def and cdef. It does this by creating two function, cdef for C types and def for Python types. Cpdef takes advantage of early binding so that cpdef functions can be as fast possible when called from other Cython code but utilizes dynamic binding when passing Python objects and this can be just as slow as def. Now that we have an understanding of the declarations and cdef, let us compare Python, C, and Cython's version of an easy Python function fib example, that calculates the nth Fibonacci number[16]:

Python:

(6.1)

```
def fib(n):
    a, b = 0.0, 1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

C:

```
double cfib(int n) {
    int i;
    double a=0.0, b=1.0, tmp;
    for (i=0; i<n; ++i) {
        tmp = a; a = a + b; b = tmp;
    }
    return a;
}
```

Cython:

```
def fib(int n):
    cdef int i
    cdef double a=0.0, b=1.0
    for i in range(n):
        a, b = a + b, a
    return a
```

As we stated earlier, Python code is essentially valid Cython code and we can see the similarities in (6.1). We can also see that the C version of the code closely resembles the Python version. Thus, if we can envision the merging of C with Python code, we can see where the statically typed Cython emerges. So, how does Cython's performance measure up in this example to Python and C?

Cython attained C-level performance. It does so because the pure C portions of any algorithm will allow Cython to typically produce code as efficient as in pure C. In this example, this is demonstrated by the significant speedup of approximately 75 times that of Python in regards to, the loop runtime. So, we can see that coding in Cython is like coding in Python and C simultaneously. It is worth mentioning that, although we will not purse this example any further, we could make the addition of a few optimizations in order to squeeze out some additional performance from Cython.

With that being said, we can see the benefits of adding a few minor tweaks to Python code, such as the cdef keyword, in order to see significant performance improvements. But not all Python code complied into Cython will exhibit performance increases. We would not see any increases or the performance benefit would be decreased if functions are memory-bound or I/O–bound or network-bound operations. It is best if we can see where we can actually make improvements to Python code before looking towards Cython.

10. The Heston Model in Cython

The motivation behind this thesis was to implement an object-oriented Cython program that evaluates exotic options via the Heston option pricing model. Now, that we have developed the theory and have an understanding of programming in Python and Cython with an emphasis being placed on object-oriented design, we can apply design patterns to our Cython program called Malik that provides the end-user with a clean, user-friendly, option pricing interface.

Here, we implement the Heston option pricing model via the client experience of the unified interface (Facade) of the Malik program called test_mc.py and it is designed as such:

(6.2)

```
# make a call option
the_call_payoff = payoff.VanillaCallPayoff(40.0)
the_call = option.Option(1.0, the_call_payoff)

# make a put option
the_put_payoff = payoff.VanillaPutPayoff(40.0)
the_put = option.Option(1.0, the_put_payoff)

# make a Monte Carlo Heston engine
sigmav = 0.61
kappa  = 6.21
theta  = 0.019
rho    = -0.70
nsteps = 100
nreps  = 10000
mc_engine = engine.MCHestonEngine(500, 50)
```

```
# make a basic market data handle
mdata = marketdata.MarketData(41.0, 0.08, 0.30, 0.0)

# price the options
opt1 = facade.OptionFacade(the_call, mc_engine, mdata)
opt2 = facade.OptionFacade(the_put, mc_engine, mdata)
print("The value of the call option is: {0:0.3f}".format(opt1.price()))
print("The value of the put option is: {0:0.3f}".format(opt2.price()))

Output: 6.8061
```

Note, we could easily implement a Python graphical user interface(GUI) framework on top of our code that would allow clients and/or users to interact with our Malik program through graphical icons and visual indicators, as opposed to our currently utilized text-based user interface. But, for now, let us proceed with our model.

We can explain the various aspects of test_mc.py in the following fashion:

- We have the option to select the type of option (call or put) that we wish to evaluate the option by as demonstrated in # make a call option and the # make a put option fields. Although, there are inputs for both call and put options, we make the decision of which one to utilize at run-time.
- We have the option to select the type of engine we would like to run. In this case, we have chosen to utilize the Heston model where we list the parameters, its associated values and the engine we would like to run.
- We also have basic market data field and its inputs.
- We have the field that prices the options. Here, again we see we have both call and put options listed. It is only listed to show what the code and option looks like. But, when running the module, we would choose which one we want to use. We would then choose the associated print option. At this point, we would execute the program.
- We have an output field that provides the evaluated option price and in (6.2), we see the option price is 6.8061.

While the Heston option pricing model was successfully implemented in Cython, modelled by the test_mc.py module, the countless expansion possibilities of the Malik program are endless. For instance, we could provide the optionality of different engines for the end-user to choose from such as a Black-Scholes, a naive Monte Carlo, antithetic sampling, etc. without ever having to re-code anything. We could build out this program as far as we saw fit and it would perform at the “speed of thought” and have the performance of C/C ++. That is the unique beauty of OOP, Cython and the Malik program. It could provide the client with endless options without ever unveiling what’s behind the curtain as the client does not care about how the asset was

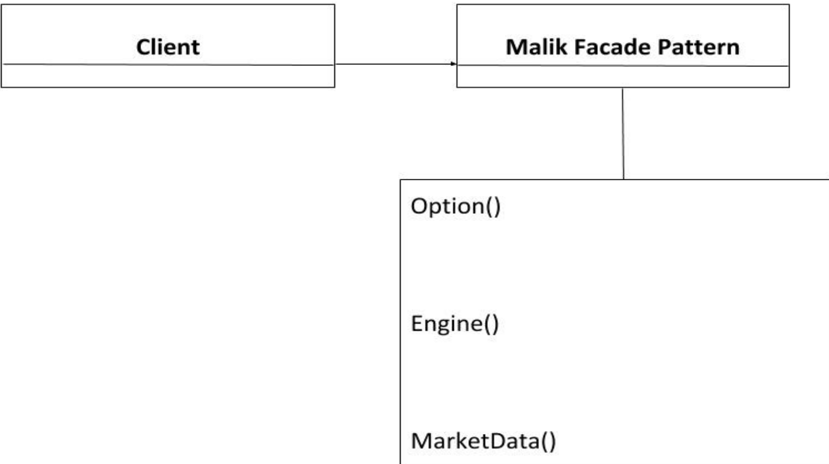
calculated. They only care about its accuracy, speed and ease-of-use. No one cares about how a calculator calculates the sum, only the result. So, it’s time to take a peak behind the curtain also known as the Facade and Strategy, of this extremely useful and buildable object-oriented Cython program.

11. The Facade and Strategy Patterns

While the background of the Malik program is neat, it is also extremely complex and intricate. So, with that in mind, the primary focus of this chapter will be to discuss some of the aspects of the Facade and Strategy Patterns used in our Malik program. We shall provide a brief overview of some of their components without going into too much detail to keep things simple and concise. In addition, we shall only discuss new, relevant, aspects of Python and Cython that we have yet to discuss with brief references of learnings from prior chapters. Let us begin with the Facade.

11.1 Facade

Our Facade is quite simple and straight-forward as clearly seen by the following illustration:
(6.3)



Our facade is comprised of three modules: Option, Engine and MarketData. We utilized Cython’s cimport where the Cython compiler locates each of these files such as Option.pxd in its’ search for included files. There are 3 file types associated with Cython. Cython consists of .pyx (implementation), .pxd(definition) and .pxi(included) file types. It is worth mentioning that when we compile the file Option.pyx, the associated Option.pxd is searched first, and if it is located, it is used before applying the .pyx file. This is easily understood because unlike Python,

Cython code has to be complied. First the .pyx file is complied into .c file that contains the code of a Python extension fie. Then .c file is complied into a .pyd file that can be imported straight into live Python. In order to build this Cython code, we wrote a distutils setup.py as this is the most natural method Cython files are distributed and built.

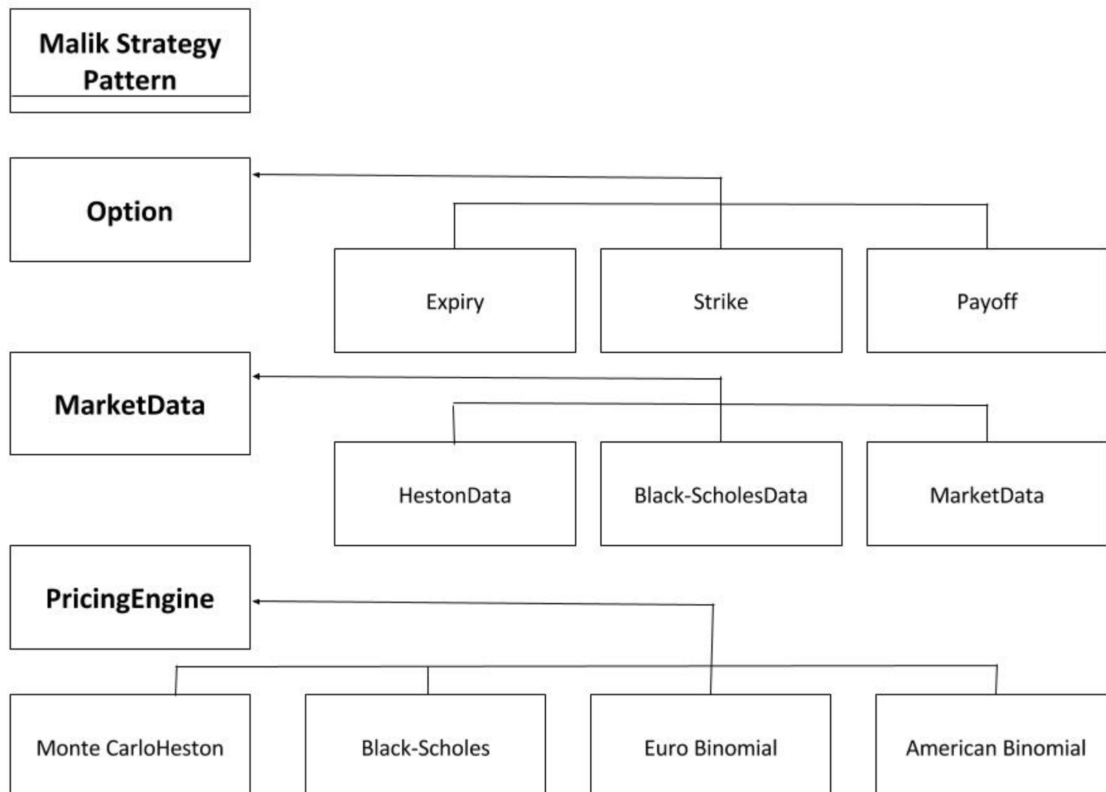
Python Distribution Utilities (distutils) are a way to distribute Python packages and extensions from the standard established Python library. The setup.py module was created where it simply complies the .pyx file into an extension module. This is similar to a Python Makefile for which, was also used. A Makefile is essentially a file that tells the program make how to compile and connect programs. It is an extremely powerful tool that is used to construct large programs and ultimately aids in the compilation of portions of code that are modified and contingent upon the modifications.

In addition to the above, we also applied cdef to a class in our Facade called OptionFacade. This facade is a facade class to price an option. Yes, we did use more than one facade pattern but, we will not get into the details in the interest of simplicity and finally we added a cpdef statement to our price. We see how simple and straight-forward our Facade really is. So, let us move on to address our more complex Strategy.

11.2 Strategy

The Strategy contains the inner workings of our Malik program. As we stated earlier, “no one cares about how a calculator calculates the sum, only the result.” But in this instance, we shall look at the thought process behind the calculation and not the formulating methods. But, in order to do so, let us provide an illustration of our Strategy so we can comprehend its components and structure. Then we shall discuss the components and their interactions. The Unified Modeling Language (UML) helps us visualize the design of our Strategy and is illustrated by the following:

(6.4)



Note: Not all classes in this illustration are implemented. Some are only used as place-holders to signify some of the different possibilities we could execute with our program.

With respect to the Facade, we can clearly see the *has-a(n)* relationship at work between the Facade and Strategy as we now see that our Facade abstracted away from the Strategy. So, allow us to take a closer look at the Strategy. In an effort to maintain simplicity, we will divide the analysis of the Strategy into three portions: Option, MarketData and PricingEngine.

Option

Our Option is quite simple. It simply contains two Cython functions, option and payoff. We created two properties of classes within the option class, the expiry and strike class. The properties behave and appear like regular attributes but, you equip procedures that control access to the attributes. We then return the payoff. It's worth mentioning that we utilize self throughout all of the Malik program so that when we create a class instance of class name and call its methods, it will be passed instinctively. Now, let us take a step back to the payoff for a

moment. We separated the payoff so that it has its own file in the interest of future endeavors. Let’s explore the payoff for a moment.

The payoff is not that much different from option in the sense of structure. We simply define more elements. In the payoff contains three Cython functions for three classes: Payoff, VanillaCallPayoff and VanillaPutPayoff where Payoff is a base class for option payoffs, VanillaCallPayoff is a concrete class for the vanilla call option payoff and VanillaPutPayoff is a concrete class for the vanilla put option payoff. We import Numy and cimport NumPy to vectorize our program by having access to NumPy Python functions and NumPy C API. We utilize the special method __init__() and create a property for the strike within the payoff. We take advantage of using the Cpdef statement so functions can be quick as we are calling from other Cython code for both the VanillaCallOption and the VanillaPutOption. We then return the result.

MarketData

MarketData consists of two Cython functions for our two classes MarketData and HestonData, where HestonData inherits from MarketData. MarketData is a class that holds market data for option pricing. The HestonData is a class that holds market data and estimated parameters for the Heston model. Both MarketData and HestonData utilize class instantiation and each class defines a special method __init__() where we have input many arguments that yielded more flexibility. The arguments in MarketData such as self, spot, rate, div, etc. were inherited by the HestonData and then the HestonData adds its arguments such as kappa, theta, rho, etc. for the estimation parameters for the Heston model. Next, we utilized properties for each argument within MarketData and HestonData with HestonData inheriting from MarketData and expanding its properties.

PricingEngine

Because the PricingEngine utilizes multiple-inheritance abstraction and composition, needless to say, it gets fairly complicated extremely quick. So, in an effort to minimize any confusion, we shall be brief and provide an extremely broad overview of our pricing engine. To begin, from libc.math we cimported exp as cexp, sqrt as csqrt, from scipy.stats we imported binom and then we imported NumPy and cimported NumPy. We utilized the learnings from prior programming chapters to create and define multiple pricing engines that interact in an OO environment where we have base, interface and concrete classes that have declared regular C ints, floats, and doubles from which the option price is derived.

Now that we have had some insight into what is behind the veil of the Malik program, we can see how the Facade and Strategy compliment our Cython financial program. We can discern how the structure or layout of the program is vital for its success and implementation. We are able to have access to key areas without too much hassle and distinctly be able to make changes and see how those changes will affect our program. Ultimately, it provides us with

accessibility to each aspect of our program in a structured manner that is easy to visualize and see how things interact.

12. Conclusion

The Heston model was successfully modeled by the Malik Cython program in conjunction with Euler's discretization scheme in a simple Monte Carlo engine and in doing so, we have remedied some inefficiencies and patterns expressed by the Black-Scholes model, namely by allowing for the 'smile' by defining the volatility as a stochastic process. While the Heston model is widely recognized, the program for which, the model was programmed in is what is truly unique and beneficial to the financial industry.

Having implemented the Heston model in object-oriented Cython is what should entice the financial industry into making Cython the normal programming language for the vast majority of their operations. It provides the simplicity or user-friendliness of Python, all the while, providing C performance. It literally is…the best of both worlds as demonstrated by the implementation of the Heston model by the Malik program. Cython proves to be beneficial to programmers and the Malik program by increasing computational efficiency and developer velocity.

Assuming you are utilizing Cython for the "right" purpose, it is just as efficient as the programming language it calls and the Malik program is the right purpose. We could have used pure Python to model the Heston but, the loss of computational efficiency is not advantageous to the Malik program nor any other mathematically complex model especially financial models use by the financial sector for which, utilize complex mathematical formulations to price similarly complex options. The increase in developer velocity stemming from OO Cython and the Malik program are substantial. Not only does it allow for code reuse which, boosts productivity, it is also blazingly quick too. Tapping into some of Python's hidden powers are as simple as adding c's before def and that is not even the best part. The best aspect of OO Cython and the Malik program is it's ease of construction.

Not only was the coding for the Malik program simple and straightforward but, it's expansion capabilities are endless. It provides the optionality of expansion without recoding anything. It is as simple as the addition of classes, subclasses and its calculation formula. This will not have any detrimental effects on the client. They will only see what you allow them to see. The added benefit is that the client has the ability to price different options. This program in conjunction with Cython, can be whatever you want. We could have it pull data from Bloomberg or CNBC Finance or whatever data source we saw fit. Any way you look at it, you cannot go wrong with OO Cython or the Malik program as they are both in the money.

References

1. L. Heston, Steven. *A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options*. The Review of Financial Studies, Vol. 6, No. 2 (1993), pp. 327-343 Oxford University Press. Sponsor: The Society for Financial Studies.
2. Hull, J. C., 1989, *Options, Futures, and Other Derivative Instruments*, Prentice-Hall, Englewood Cliffs, NJ.
3. Broadie, Mark and Kaya, Ozgur. *Exact Simulation of Stochastic Volatility and Other Affine Jump Diffusion Processes*. Operations Research, Vol. 54, No. 2, March-April 2006, pp. 217-231
4. Lord, R. Koekkoek, and D. V. Dijk. *A comparison of biased simulation schemes for stochastic volatility models*. Quantitative Finance, 10(2):177–194, 2010.
5. Mil'shtein, G. N. (1974). *Approximate Integration of Stochastic Differential Equations.* Teor. Veroyatnost. i Primenen (in Russian). 19 (3): 583–588.
6. Mil'shtejn, G. N. (1975). *Approximate Integration of Stochastic Differential Equations*. Theory of Probability & Its Applications. 19(3): 557–000.
7. Cox, J. C., J. E. Ingersoll, S. A. Ross. 1985. *A theory of the term structure of interest rates. Econometrica* 53(2) 385–407.
8. Dawson, Michael., 2010, *Python Programming, for the absolute beginner. Third Edition*. Course Technology, Boston, MA.
9. No Author (n.d.). Python. *What is Python? Executive Summary*. Retrieved August 5, 2017, from <https://www.python.org/doc/essays/blurb/>
10. Hilpisch, Yves., 2014, *Python for Finance*. O'Reilly, Sebastopol, CA.
11. Weisfield, Matt., 2013, *The Object-Oriented Thought Process. Fourth Edition*. Addison-Wesley, London.
12. Swaroop, C.H., 2015. *A Byte of Python*. CreateSpace Independent Publishing Platform, USA.
13. Freeman, Eric, Elisabeth Robson, Kathy Sierra, and Bert Bates. 2004. *Head First design patterns*. O'Reilly, Sebastopol, CA. pp. 315-420
14. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Sydnes, Zurich, Urbana, Hawthorne
15. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 185-210
16. W. Smith, Kurt., 2015, Cython. O'Reilly, Sebastopol, CA.
17. EuroPython Conference*., "Friday, 14 July – Anfiteatro 2 EuroPython 2017"YouTube*. YouTube, Web Retrieved July 29, 2017. *Scientific computing using Cython: Best of both worlds! by Simmi Mourya,* 2:05:15<https://www.youtube.com/watch?v=IZ5P12GASDQ>

18. Zhu, Jianwei. 2009. *Applications of Fourier Transform to Smile Modeling: Theory and Implementation.* Springer Science and Business Media.

19. Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith. *Cython: The Best of Both Worlds*. Computing in Science & Engineering. IEEE CS and The AIP. March/April 2011

20. Voit, Johannes. 2013. *The Statistical Mechanics of Financial Markets.* Springer Science and Business Media.

21. Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, Greg Ewing, William Stein and Gabriel Gellner. Cython. *Cython Documentation*. Retrieved August 1, 2017, from <http://docs.cython.org/en/latest/index.html>

22. Wikipedia Page https://en.wikipedia.org/wiki/Strategy_pattern

23. Wikipedia Page https://en.wikipedia.org/wiki/Facade_pattern

Appendices

Appendix A

To view code for this thesis, please visit https://github.com/brandonhardin