AY 2020/2021 Term 2

QF205: Computing Technology For Finance

Prepared for: Dr Zhao YiBao

Python Programming and Its Applications in Option Pricing

Group 8

Submitted By:

CHENG WEI ELIZABETH	01394073
GUO XIANGSHUAI	01392149
PENG SHAOHUA	01398465
RAO NINGZHEN	01354574
SHI HAOXIANG	01317062
ZHANG SIYANG	01385799
ZHU HAORAN	01389661

Contents

1. Introduction

Python is an interpreted, high-level, general purpose programming with dynamic semantics. It emphasizes readability, and the syntax are relatively simple and easy to learn. Therefore, the cost of program maintenance is reduced. Furthermore, as Python is an open-source software with a large and comprehensive standard library, making it one of the most popular programming languages in the world. 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. Python is also considered an object-oriented programming language.

The purpose of this report is to provide guidance to readers who even have no prior knowledge in programming nor finance to understand, write and run Python code for an option price calculator application on their computer.

1.1 Our Project

The aim of our project is to teach Python beginners the basics of the beautiful programming language and bring them through step by step in creating their first program. Our choice of topic is an option pricing calculator, using Black-Scholes model and explicit method. We will first bring the readers through the basics of Python, then introduce the concept of option pricing and different methods of calculating option price. We will be elaborating the code for explicit method of option pricing and GUI construction.

The project files together with this report in the zip folder are as follow:

File Name	File Type	Description
Main.py	PY file	Contains the code to run the GUI application.
OptionPricing.py	PY file	Contains the code to run the option pricing calculator.
OptionpricingGUI.ui	UI file	Contains the design of the calculator interface.

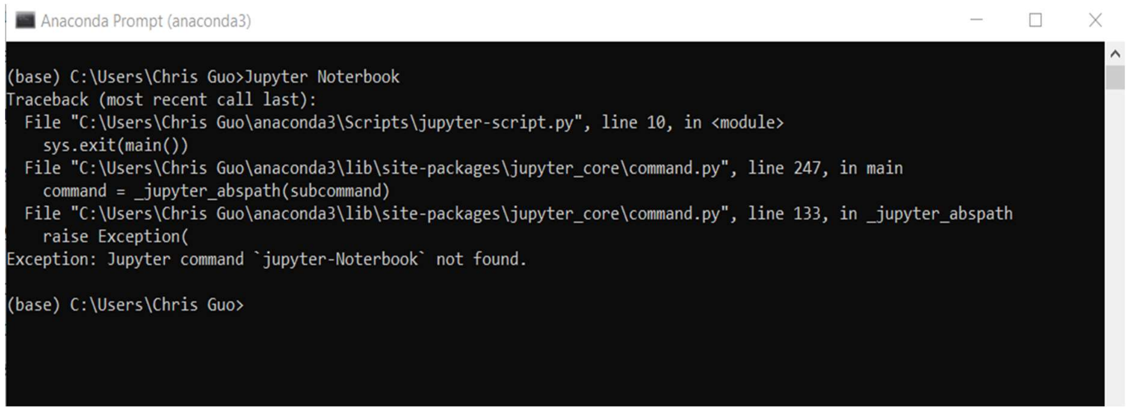
1.2 Python Installation

We will be using Python 3.8 and Anaconda3-2020.07(64bits) to demonstrate and run the option pricing calculator. The following links can be used to download and install them.

Windows: https://repo.anaconda.com/archive/Anaconda3-2020.07-Windows-x86_64.exe

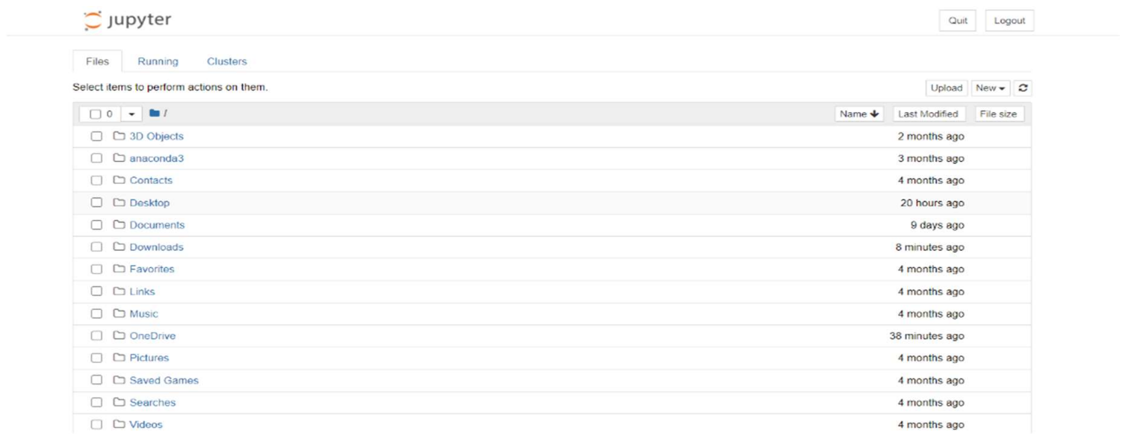
Mac OS: https://repo.anaconda.com/archive/Anaconda3-2020.07-MacOSX-x86_64.pkg

Jupiter Notebook is an open-source web application that allows users to create and share documents that contain codes, equations, visualisations and narrative text. To launch Jupiter Notebook, first open the Anaconda Prompt and enter 'Jupyter notebook', then press 'enter'.



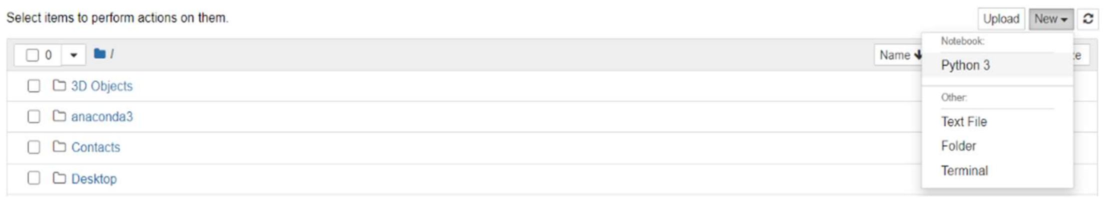
```
(base) C:\Users\Chris Guo>Jupyter Noterbook
Traceback (most recent call last):
  File "C:\Users\Chris Guo\anaconda3\Scripts\jupyter-script.py", line 10, in <module>
    sys.exit(main())
  File "C:\Users\Chris Guo\anaconda3\lib\site-packages\jupyter_core\command.py", line 247, in main
    command = _jupyter_abspath(subcommand)
  File "C:\Users\Chris Guo\anaconda3\lib\site-packages\jupyter_core\command.py", line 133, in _jupyter_abspath
    raise Exception(
Exception: Jupyter command `jupyter-Noterbook` not found.

(base) C:\Users\Chris Guo>
```

Then you will be directed to a localhost page (shown below) with your default browser.

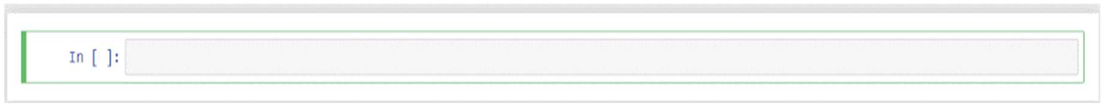


3D Objects	2 months ago
anaconda3	3 months ago
Contacts	4 months ago
Desktop	20 hours ago
Documents	9 days ago
Downloads	8 minutes ago
Favorites	4 months ago
Links	4 months ago
Music	4 months ago
OneDrive	38 minutes ago
Pictures	4 months ago
Saved Games	4 months ago
Searches	4 months ago
Videos	4 months ago

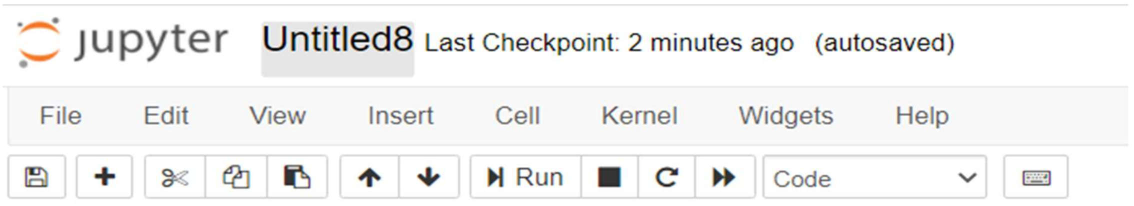
To create a new notebook, click on the 'new' button on the top right-hand corner and select 'Python 3'. Each notebook is a separate document for you to store and share your code.



Each cell in the workbook is for you to write and run your code by enter the code and press ’shift+enter’.



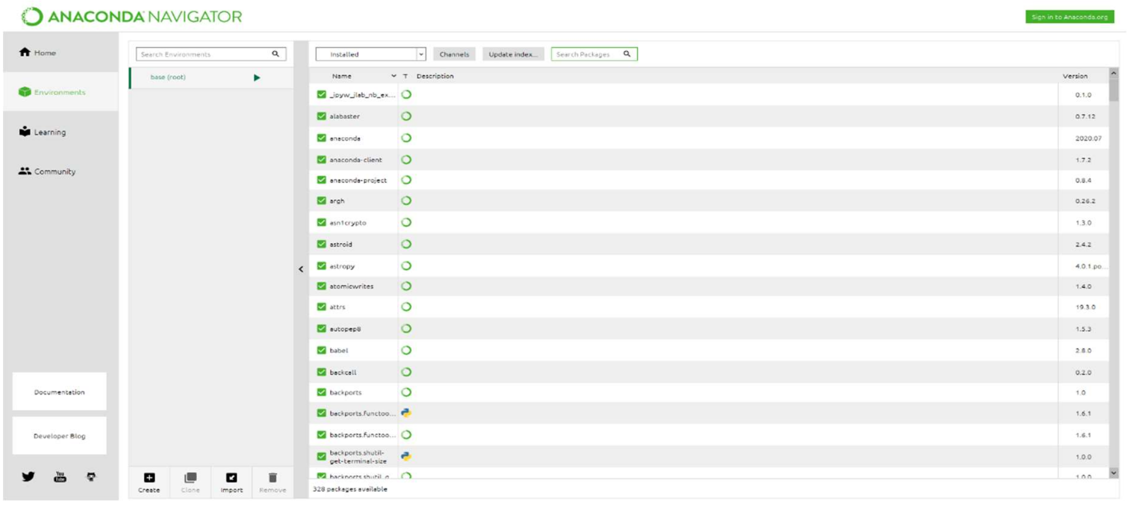
You can change the name of the file by simply click on the title.



Like any other documents, jupyter notebook can save the workbook as ‘ipynb’ file to a location of your choice and can be revisited anytime when opened using jupyter notebook. After knowing how to create a workbook to edit the code. Let us now look at how to use packages.

1.3 Packages

Package is a collection of modules which are pre-written and are available for free usage. By installing packages, we do not need to write the functions from scratch, thus making our programming much more efficient. The modern Python comes with a range of ‘system packages’ which are integrated automatically to Python. You can find the list of installed packages using Anaconda Navigator, the tab ‘environments will show you all the installed packages.

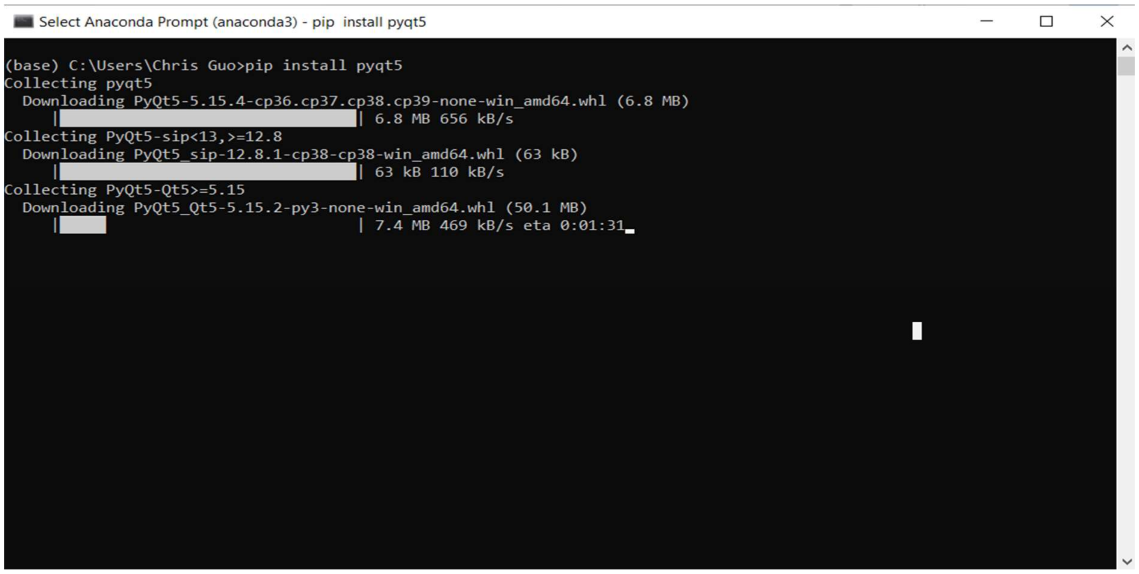


Name	Description	Version
_ipyw_jlab_nb_ex...		0.1.0
alabaster		0.7.12
anaconda		2020.07
anaconda-client		1.7.2
anaconda-project		0.8.4
argh		0.26.2
asn1crypto		1.3.0
astroid		2.4.2
astropy		4.0.1.po...
atomicwrites		1.4.0
attrs		19.3.0
autopep8		1.5.3
babel		2.8.0
backcall		0.2.0
backports		1.0
backports.functoo...		1.6.1
backports.functoo...		1.6.1
backports.shutil-get-terminal-size		1.0.0

328 packages available

If a package is not automatically installed, we must do it manually and one way is to use pip install. Pip is a commonly used package-management system used to install and manage packages. Pip itself is pre-loaded to the modern version of Python.

We can use the command 'pip install' to install a package. For example, in this project, we need to install the package 'PyQt5' which is a package used for interface designing. Therefore, we open an anaconda prompt and enter 'pip install PyQt5'.



```
(base) C:\Users\Chris Guo>pip install pyqt5
Collecting pyqt5
  Downloading PyQt5-5.15.4-cp36.cp37.cp38.cp39-none-win_amd64.whl (6.8 MB)
     |████████████████████████████████| 6.8 MB 656 kB/s
Collecting PyQt5-sip<13,>=12.8
  Downloading PyQt5_sip-12.8.1-cp38-cp38-win_amd64.whl (63 kB)
     |████████████████████████████████| 63 kB 110 kB/s
Collecting PyQt5-Qt5>=5.15
  Downloading PyQt5_Qt5-5.15.2-py3-none-win_amd64.whl (50.1 MB)
     |█████                           | 7.4 MB 469 kB/s eta 0:01:31
```

And this will start the installation of the PyQt5 package. Other packages we used in the project are pre-loaded hence do not need to install manually.

Now we have all the necessary packages installed. We will be moving on to acquire some basic knowledge of Python.

2. Basics of Python

2.1 Variables

Python variable is a reserved memory location to store values. It is very important for us to use variables to store the intermediate calculations in Option Pricing calculator. The data that is assigned to the variable can be accessed by calling the name of the variable.

```
>>> x = 100
>>> print(x)
100
```

In this case, the value 100 is assigned to the variable name x and we can access this value by calling the variable x.

2.2 Data Types

i. **Numeric Types – int**

   int type refers to positive or negative whole numbers with no decimal points. There is no limit for the length of integer literals apart form what can be stored in available memory. Also, underscores are ignored for determining the numeric value of the literal. They only can be used to group digits for enhance readability. We can use type () build in function to check the type of a given data.

   ```
   >>> x = 1
   >>> type(x)
   <class 'int'>
   ```

ii. **Numeric Types – float**

   Point -floats are specified with a decimal point. Exponent-float will be represented by the character e or E followed by a positive or negative integer.

```
>>> type (1.1)
<class 'float'>
>>> type(-5e100)
<class 'float'>
```

iii. **Numeric Types – complex**

Complex numbers are represented by a pair of floating numbers and are specified as Real part + imaginary part. We can use .real and .imag to access the real part and the imaginary part of ta complex number respectively

```
>>> type(1+2j)
<class 'complex'>
```

In this case, the complex number 1+2j has 1 is the real part and 2j is the imaginary part.

iv. **Text Types – string (str)**

A string in Python is a sequence of character and strings are immutable which means that once defined, it cannot be changed. Regardless of what is written, Python will return str if it is enclosed in quotation marks.

```
>>> type("Hello World")
<class 'str'>
>>> type('1+2')
<class 'str'>
```

The backslash (\) character is used to escape the characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character.

v. **Binary Sequence Types – bytes**

Byte type is a sequence of bytes that have been encoded and are ready to be stored in memory.

vi. **Boolean**

The Boolean data type is either True or False. Boolean data type is always returned by a comparison operation.

```
>>> type(True)
<class 'bool'>
```

Integers and floats can be converted to Boolean data type by using bool () function. Any numeric types set to 0 will return False. When set to other values, it will return True.

```
>>> bool(0)
False
>>> bool(-1)
True
```

Boolean operators in Python include and, or, not. For and operator,

```
>>> A = True
>>> B = False
>>> A and B
False
>>> A or B
True
>>> not A
False
```

vii. List

A list is a data structure in Python that is a mutable ordered sequence of elements. Lists are compounded data type made up of smaller parts and are very flexible because they can have values added, removed, and changed. List is defined by the square brackets ([]) and we can use [] to create and empty list. List allows for duplicate members.

```
>>> x = [1, "2",3.0,'hello world',(1,)]
>>> type(x)
<class 'list'>
```

List is very flexible as it can contain all the types of data and these data can be accessed using the location of each data.

viii. Tuple

A tuple is defined by comma and enclosing the elements in parentheses instead of square brackets. Tuples are ordered and immutable Python objects and tuples allow for duplicates members.

```
>>> type((1,))
<class 'tuple'>
>>> type((1))
<class 'int'>
```

A tuple is defined by the comma instead of a parenthesis.

ix. Dictionary

Dictionaries are used to store data values in key: value pairs. A dictionary is a collection which is ordered (for python 3.7), changeable and does not allow duplicates. The values in a dictionary can be accessed using the key name. Dictionaries does not allow duplicate members. Empty curly braces {} will create an empty dictionary in Python.

```
>>> x =  {"first member" : 1, "second member": "2","third member": (1,)}
>>> x['first member']
1
```

x. Set

Sets are used to store multiple items in a single variable and sets are both unordered and does not allow for duplicate members. The items a set cannot be accessed by referring to an index, since sets are unordered the items has no index. The data in a set can be accessed through a for loop. To create an empty set, we can set() function without any arguments.

```
>>> x = {1,1,1,4,"1"}
>>> type(x)
<class 'set'>
>>> for i in x:
...    print(i)
...
1
4
1
```

2.3 Arithmetic Operators

Arithmetic operators are used to perform mathematical operation and there are 7 arithmetic operators in Python.

Operator	Explanation	Example
+	Addition - Add two operands or unary plus	>>> 1+1 2
-	Subtraction - Subtract right operand from the left or unary minus	>>> 3-1 2
*	Multiplication - Multiply two operands	>>> 2*1 2
/	Division - Divide left operand by the right one (return only float type)	>>> 4/2 2.0
%	Modulus – remainder of the division of the left operand by the right	>>> 8%3 2
//	Floor division – division that results into whole number adjust to left in the number line	>>> 5//2 2
**	Exponent- **left** operand raised to the power of right	>>> 2**1 2

We need to pay special attention to the division operator as it will always return a floating number even if both inputs are integer type.

2.3.1 Operator Precedence

Operator precedence determine the grouping of terms in an expression and decides how an expression is evaluated. Operators with higher precedence will be evaluated first than operators with lower precedence.

Order	**Operation**
1	()
2	** Exponent will be executed from right to left
3	*,/,//,%
4	+,-

2.3.2 Comparison Operations

Python comparison operators are used to compare two values.

Operator	**Name**	**Example**
==	Equal	>>> 1 == 1 True
!=	Not equal	>>> 1! =2 True
>	Greater than	>>> 2>1 True
<	Less than	>>> 1<2 True
>=	Greater than or equal to	>>> 2>=2 True
<=	Less than or equal to	>>> 1<=1 True

2.4 Indexing and Slicing

Indexing means referring to an element of an iterable by its position within the iterable. We can use indexing and slicing method to access portion of data in an ordered sequence.

String	I		L	o	v	e		Q	F	2	0	5
Index	0	1	2	3	4	5	6	7	8	9	10	11

```
>>> x = 'I Love QF205'
>>> x[4]
'v'
```

For this string, each character has been assigned with an index number starting from 0 to 12 and we can access the character ‘v’ by referring to index 4 of the string.

Slicing means getting a subset of an element from and iterable based on their indices. We need to use indexing to slice a portion of an iterable objects.

```
>>> x = 'I Love QF205'
>>> x[7:12]
'QF205'
```

For slicing x[i,j], object with index i will be included and object with index j will be omitted and hence for slice x[i,j] will return j-i items in the same type as the original sequence. Indexing and slicing can also be used to other ordered sequences like lists and tuples.

2.5 Functions

A function is a block of code which only runs when it is called to achieve a specific task. We can input data as parameter into a function and the function can return data as result. There are two type of functions: built- in and user defined functions. Built in function are pre-coded and downloaded together with Python. These functions can only be used to execute basic tasks like sum (), max(),abs().

For example, sum () built in function sums start and the items of an iterable from left to right and return the total.

```
>>> sum([1,2,3])
6
```

For user-defined function, it is like the built-in function but to achieve a more complicated task. There are four blocks to define a user-defined function. 1) function blocks begin with the key word def followed by the function name and parenthesis. 2) input parameters should be placed with in the parentheses. 3) The code block within every function starts with a colon and is intended. 4) The function block can end with a return statement or passing back and expression to the caller. If there are multiple objects returned from a function, it will be returned as a tuple type and we can use indexing and slicing to access the tuple object.

```
>>> def addition(x,y):
...    a = x+y
...    return a
>>> addition(1,2)
3
```

2.6 Loops

Loops are used in python to iterate over a sequence (list, tuple, dictionary, set, or a string) by a given number of times. There are two types of loops, for loop and while loop.

i. **For loop**

   For loops helps us to iterate over a given sequence. To build a for loop, we need a sequence and for statement.

```
>>> x = "QF205"
>>> for i in x:
...    print(i)
...
Q
F
2
0
5
```

ii. While loop

While loop is used to execute a block of statemen repeatedly until a give a condition is satisfied and when the condition becomes false, the line immediately after the loop in program is executed.

```
>>> count =0
>>> while count<3:
...    count = count +1
...    print(count)
...
1
2
3
```

For this while loop, the loop will continue as long as the variable count is smaller than 3, and it loop will break after condition is no longer satisfied.

iii. "Break" and "continue" statement

The break and continue statements are used in both 'for' and 'while' loops. The break statement terminates the current loop and resumes execution next statement.

The continue statement reject all the remaining statement in the current iteration of the loop and moves the control back to the top of the loop.

3. Option Pricing

In this section, we will look at options, option pricing theory, and briefly going through the various methods and algorithms for pricing options. Here, we seek to give readers a basic understanding of the option pricing models and the different methods for implementation.

3.1 Option Contracts and Put/Call Options

An option contract refers to an agreement between two parties to facilitate a potential transaction on the underlying security at a present price, referred to as the strike price, prior to the expiration date. Purchase of an options contract grants you the right, but not an obligation, to buy or sell an underlying asset at a set price on or before a specified date.

There are two types of contracts, put and call options. A call option gives the holder the right to buy a stock at strike price and a put option gives the holder the right to sell a stock at strike price.

3.2 Option Pricing Theory

Option pricing models are used to price options by accounting for multiple variables, for instance, current market price, strike price, volatility, interest rate, and time to expiration. Commonly used option pricing models include Black-Scholes, binomial option pricing, and Monte-Carlo simulation.

Options pricing theory also derives various risk factors or sensitivities based on the inputs, which are known as "Greeks". In the rapidly changing market, the Greeks provide us with means of determining how sensitive a trade is to price fluctuations, volatility fluctuations, and time. Typical Greeks include:

1. Delta **(Δ)** – represents the rate of change between the option's price and a $1 change in the underlying asset's price
2. Theta **(Θ)** – represents the rate of change between the option's price and time
3. Gamma **(Γ)** – represents the rate of change between an option's delta and the underlying asset's price
4. Vega **(V)** – represents the rate of change between the option's value and the underlying asset's implied volatility

5. Rho **(p)** – represents the rate of change between the option's value and a 1% change in the interest rate

3.3 Black-Scholes Model

The Black-Scholes model, also known as the Black-Scholes-Merton (BSM) model, is a mathematical model for pricing an options contract. In particular, the model estimates the variation over time of financial instruments.

The BSM model makes the following assumptions:

- The option is European and can only be exercised at expiration
- No dividends are paid out during the life of the option
- Markets are efficient
- There are no transaction costs in buying the option
- The risk-free rate and volatility of the underlying are known and constant
- The returns on the underlying asset are log-normally distributed

3.4 Foundation of Finite Difference Methods

In this section, we describe the fundamental ideas of finite difference applied to solve the PDE. This section includes approximation of partial derivatives, general discussion of the grid, and the way of formulating boundary conditions.

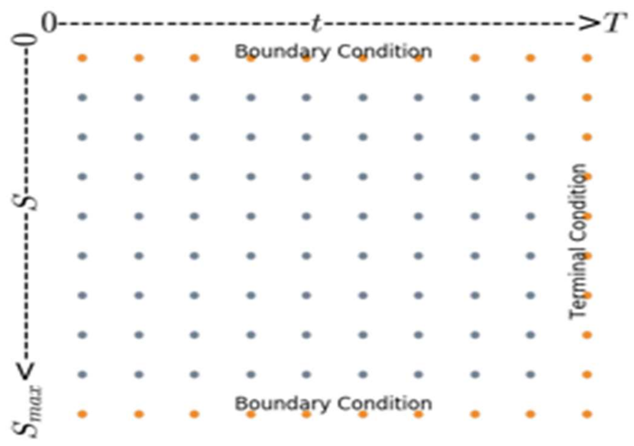
3.4.1 The Grid

Consider the Black-Scholes equation in the last section, we perform a discretization leading to a two-dimensional grid. To set up a grid, firstly, we should determine the value of Smax, which is considered as the underlying price unlikely to be reached. Then we discretize the t and S as following,

$t=0,\delta t,2\delta t,\dots,N\delta t$

$S=0,\delta S,2\delta S,\dots,M\delta S$

δt and δS are the mesh sized of the discretization of t and S. Following figure illustrates part of the entire grid in the (S,t)-plane. For each node (iδS,jδt), we use the grid notation fi,j=f(iδS,jδt)



Theoretical solution f*i,j of Black-Scholes equation is defined on the continuous space. In contrast, the approximation solution fi,j derived from the discretization of the equation is only defined for the nodes. By controlling the parameters Smax, M, N, discretization error can be reduced such that fi,j is closer to f*i,j.

3.4.2 Difference Approximation

With the help of Taylor expansion, we can discretize the derivatives in the PDE leading to a recursive formula for fi,j. Let us briefly review Taylor expansion and derivatives approximation that derived from it. Assume f∈C(n+1), h is small. For intermediate number ξ between x and x+h,

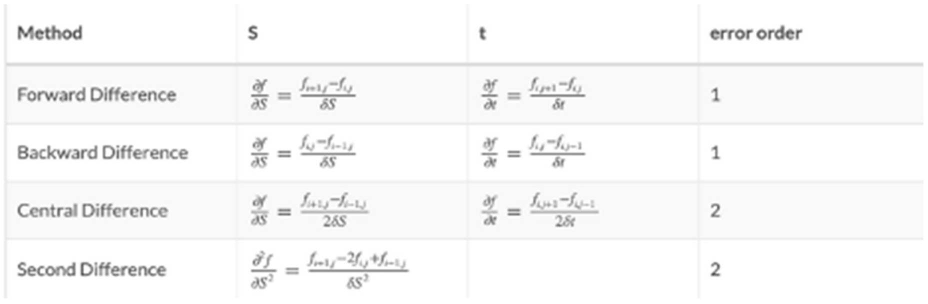
$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \ldots + \frac{h^n}{n!}f^{(n)}(x) + \frac{h^{n+1}}{(n+1)!}f^{(n+1)}(\xi)$$

Replacing terms after h2 with O(h2) and moving f'(x) to the left-hand side, we have forward difference, which has error order 1

$$f(x+h) = f(x) + hf'(x) + O(h^2)$$

$$\Rightarrow f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

Likewise, we can approximate f′ and f″ by backward difference and central difference. Following is a summary of derivatives approximation, expanding at (iδS,jδt):



Method	S	t	error order
Forward Difference	$\frac{\partial f}{\partial S} = \frac{f_{i+1,j} - f_{i,j}}{\delta S}$	$\frac{\partial f}{\partial t} = \frac{f_{i,j+1} - f_{i,j}}{\delta t}$	1
Backward Difference	$\frac{\partial f}{\partial S} = \frac{f_{i,j} - f_{i-1,j}}{\delta S}$	$\frac{\partial f}{\partial t} = \frac{f_{i,j} - f_{i,j-1}}{\delta t}$	1
Central Difference	$\frac{\partial f}{\partial S} = \frac{f_{i+1,j} - f_{i-1,j}}{2\delta S}$	$\frac{\partial f}{\partial t} = \frac{f_{i,j+1} - f_{i,j-1}}{2\delta t}$	2
Second Difference	$\frac{\partial^2 f}{\partial S^2} = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\delta S^2}$		2

3.4.3 Boundary Conditions

Before delving into the finite difference-based pricing algorithm, we need to discuss the choice of boundary conditions which is an important issue in the construction of these pricing methods. Well-chosen boundary conditions contribute to preventing any errors on the boundaries propagated through the rest of the mesh. There are several ways to specify boundary conditions:

3*.4.3.1 Dirichlet Condition*

A Dirichlet boundary condition is a boundary condition which assigns a value to ff at boundary. For example, for a call option at Smin=0 , the option value could be set to zero, since the option is worthless. However, if underlying price reaches to Smax at time t, the value of option is the discounted value of its expected payoff.

For a vanilla call option:

$$f(0, t) = 0$$

$$f(S_{max}, t) = S_{max} - Ke^{-r(T-t)}$$

For a vanilla put option:

$$f(0, t) = Ke^{-r(T-t)}$$

$$f(S_{max}, t) = 0$$

3.4.3.2 Neumann Condition

$$\frac{\partial^2 f}{\partial S^2}(\delta S, t) = 0$$

$$\frac{\partial^2 f}{\partial S^2}((N-1)\delta S, t) = 0$$

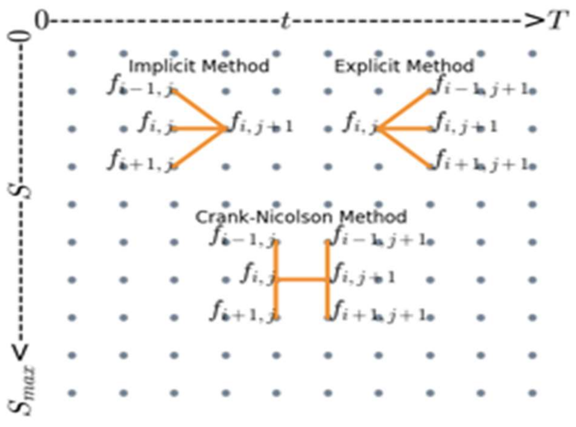
$$\Rightarrow f_{0,j} - 2f_{1,j} + f_{2,j} = 0$$

$$\Rightarrow f_{M-2,j} - 2f_{M-1,j} + f_{M,j} = 0$$

Neumann condition specifies the partial derivative of the option at the boundary. Discretizing the second derivative at the boundary by central difference gives us a more friendly equation about fi,j. In addition, the Neumann condition is inclined to be more accurate for the same boundaries than the Dirichlet condition, as the second derivatives falls off faster than the price. And Neumann condition is more universal, same for both call and put, and on both ends.

3.5 Finite Difference Methods

In this section, we discretize the B-S PDE using explicit method, implicit method and Crank-Nicolson method and construct the matrix form of the recursive formula to price the European options. Graphical illustration of these methods are shown with the grid in the following figure.

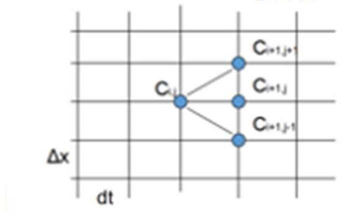


3.5.1 Explicit Method

Use backward difference approximation for t and central difference for S, expanding at (iδS,jδt),

$$-\frac{C_{i+1,j}-C_{i,j}}{dt}=\frac{1}{2}\sigma^2\frac{C_{i+1,j+1}-2C_{i+1,j}+C_{i+1,j-1}}{\Delta x^2}+v\frac{C_{i+1,j+1}-C_{i+1,j-1}}{2\Delta x}-rC_{i,j}$$

which is $$C_{i,j}=\frac{1}{1+rdt}[l_u C_{i+1,j+1}+l_m C_{i+1,j}+l_d C_{i+1,j-1}]$$



$$l_u=\frac{1}{2}\left(\frac{\sigma^2 dt}{\Delta x^2}+\frac{vdt}{\Delta x}\right)$$
$$l_d=\frac{1}{2}\left(\frac{\sigma^2 dt}{\Delta x^2}-\frac{vdt}{\Delta x}\right)$$
$$l_m=1-\frac{\sigma^2 dt}{\Delta x^2}$$

In matrix form,

$$\begin{bmatrix} 1+\beta_1 & \gamma_1 & 0 & \cdots & 0 \\ \alpha_2 & 1+\beta_2 & \gamma_2 & \cdots & 0 \\ 0 & \alpha_3 & 1+\beta_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \gamma_{M-2} \\ 0 & 0 & 0 & \alpha_{M-1} & 1+\beta_{M-1} \end{bmatrix}\begin{bmatrix} f_{1,j} \\ f_{2,j} \\ f_{3,j} \\ \vdots \\ f_{M-1,j} \end{bmatrix}+\begin{bmatrix} \alpha_1 f_{0,j} \\ 0 \\ 0 \\ \vdots \\ \gamma_{M-1} f_{M,j} \end{bmatrix}=\begin{bmatrix} f_{1,j-1} \\ f_{2,j-1} \\ f_{3,j-1} \\ \vdots \\ f_{M-1,j-1} \end{bmatrix}$$

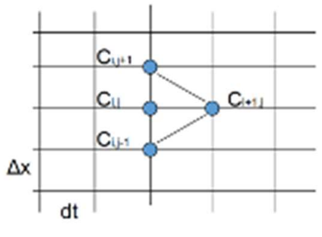
$$\begin{bmatrix} 2\alpha_1+1+\beta_1 & \gamma_1-\alpha_1 & 0 & \cdots & 0 \\ \alpha_2 & 1+\beta_2 & \gamma_2 & \cdots & 0 \\ 0 & \alpha_3 & 1+\beta_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \gamma_{M-2} \\ 0 & 0 & 0 & \alpha_{M-1}-\gamma_{M-1} & 2\gamma_{M-1}+1+\beta_{M-1} \end{bmatrix}\begin{bmatrix} f_{1,j} \\ f_{2,j} \\ f_{3,j} \\ \vdots \\ f_{M-1,j} \end{bmatrix}=\begin{bmatrix} f_{1,j-1} \\ f_{2,j-1} \\ f_{3,j-1} \\ \vdots \\ f_{M-1,j-1} \end{bmatrix}$$

3.5.2 Implicit Method

Use forward difference approximation for t and central difference for S, expanding at (iδS,jδt),

$$-\frac{C_{i+1,j}-C_{i,j}}{\Delta t}=\frac{1}{2}\sigma^2\frac{C_{i,j+1}-2C_{i,j}+C_{i,j-1}}{\Delta x^2}+v\frac{C_{i,j+1}-C_{i,j-1}}{2\Delta x}-rC_{i,j}$$

which is $$l_u C_{i,j+1}+l_m C_{i,j}+l_d C_{i,j-1}=C_{i+1,j}$$



$$l_u=-\frac{1}{2}\left(\frac{\sigma^2 dt}{\Delta x^2}+\frac{vdt}{\Delta x}\right)$$

$$l_d=-\frac{1}{2}\left(\frac{\sigma^2 dt}{\Delta x^2}-\frac{vdt}{\Delta x}\right)$$

$$l_m=1+\frac{\sigma^2 dt}{\Delta x^2}+rdt$$

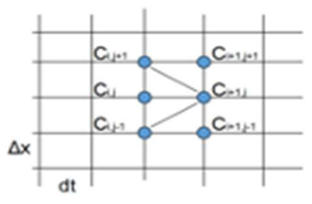
In matrix form,

$$\begin{bmatrix} 2\alpha_1+1+\beta_1 & \gamma_1-\alpha_1 & 0 & \cdots & 0 \\ \alpha_2 & 1+\beta_2 & \gamma_2 & \cdots & 0 \\ 0 & \alpha_3 & 1+\beta_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \gamma_{M-2} \\ 0 & 0 & 0 & \alpha_{M-1}-\gamma_{M-1} & 2\gamma_{M-1}+1+\beta_{M-1} \end{bmatrix}\begin{bmatrix} f_{1,j} \\ f_{2,j} \\ f_{3,j} \\ \vdots \\ f_{M-1,j} \end{bmatrix}=\begin{bmatrix} f_{1,j+1} \\ f_{2,j+1} \\ f_{3,j+1} \\ \vdots \\ f_{M-1,j+1} \end{bmatrix}$$

3.5.3 Crank-Nicolson Method

Using central difference on both t and SS, we discretize B-S equation at (iδS,(j+0.5)δt). Specifically, we use the average of first derivative on (iδS,(j+1)δt)and (iδS,jδt) to approximate the first derivative on (iδS,(j+0.5)δt). Then we discretize first and second derivatives and plug them into the B-S equation, we have

$$l_u C_{i,j+1}+l_m C_{i,j}+l_d C_{i,j-1}=-l_u C_{i+1,j+1}-(l_m-2)C_{i+1,j}-l_d C_{i+1,j-1}$$



$$l_u=-\frac{1}{4}\left(\frac{\sigma^2 dt}{\Delta x^2}+\frac{vdt}{\Delta x}\right)$$

$$l_d=-\frac{1}{4}\left(\frac{\sigma^2 dt}{\Delta x^2}-\frac{vdt}{\Delta x}\right)$$

$$l_m=1+\frac{\sigma^2 dt}{2\Delta x^2}+\frac{r}{2}dt$$

In matrix form,

$$\begin{bmatrix} -2\alpha_1+1-\beta_1 & -\gamma_1+\alpha_1 & 0 & \cdots & 0 \\ -\alpha_2 & 1-\beta_2 & -\gamma_2 & \cdots & 0 \\ 0 & -\alpha_3 & 1-\beta_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & -\gamma_{M-2} \\ 0 & 0 & 0 & -\alpha_{M-1}+\gamma_{M-1} & -2\gamma_{M-1}+1-\beta_{M-1} \end{bmatrix} \begin{bmatrix} f_{1,j} \\ f_{2,j} \\ f_{3,j} \\ \vdots \\ f_{M-1,j} \end{bmatrix} =$$

$$\begin{bmatrix} 2\alpha_1+1+\beta_1 & \gamma_1-\alpha_1 & 0 & \cdots & 0 \\ \alpha_2 & 1+\beta_2 & \gamma_2 & \cdots & 0 \\ 0 & \alpha_3 & 1+\beta_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \gamma_{M-2} \\ 0 & 0 & 0 & \alpha_{M-1}-\gamma_{M-1} & 2\gamma_{M-1}+1+\beta_{M-1} \end{bmatrix} \begin{bmatrix} f_{1,j+1} \\ f_{2,j+1} \\ f_{3,j+1} \\ \vdots \\ f_{M-1,j+1} \end{bmatrix}$$

4. Application of Python: Option Price Calculator

Now, we will put together the knowledge points from the previous sections to create an application, which is an Option Pricing Calculator. We will be implementing the explicit method algorithm that was explained Section 3 using the Python knowledge points that were explained in Section 2.

Within this section, code blocks of different colours will be used in our explanation. This is to provide a clear distinction between our actual code which will be dark-themed, and example codes which are light-themed. This is because for some concepts, our calculator only needs to utilise one way of implementation. Hence, to cover the other ways, we will use additional examples for a complete explanation.

```
#Code we used for our Option Pricing Calculator (Dark-themed)
```

```
#Code we use as examples for our explanations (Light-themed)
```

For ease of follow-through, we have also included interactive format for the codes used. The code for the Option Pricing Calculator can be accessed here, and the example codes can be accessed here.

4.1 Overview of Key Variables

Class	Method	Type	Variable name	Description
initialising	__init__()	float	T	Time to maturity (months)
			r	Risk-free rate (floating point)
			Sigma	Implied volatility

			K	Strike price
			q	Dividend rate
			S	Underlying stock price
Option_Pricing	explicit	Int	M	Number of discrete intervals for price step
			N	Number of discrete intervals for time step
		float	S_max	Maximum price of stock, assumed to be 2*K
			dt	Step-size for time step
			ds	Step-size for price step
			k	Location of underlying price in the price step
			call_price, put_price	Final value of call and put option
		numpy.ndarray	matrix_call, matrix_put	Call and put values matrix
			A	Matrix storing difference approximation values

4.2 Step-by-step Walkthrough of Explicit Method Algorithm

In this segment, we will walk-through our Python code and explain in detail the different knowledge points that are covered in our implementation.

4.2.1 Imports

First, we will use our application to explain the import system in Python. Import statements are written as the first few lines within the code, and they are used to import modules into the local namespace.

For this application, we will be importing two Python modules, numpy and math. To do so, we use the two import statements shown below:

```python
import numpy as np
from math import floor, exp
```

From here, we will use our code to explain the import system in Python, which includes Basic Import statements and import statements with the from clause.

4.2.1.1 Basic Import Statements

The first import statement we used to import numpy is a basic import statement, which has no from clause. It is executed in two steps:

1. Find the module (in this case numpy), loading and initialising where necessary.
2. Define a name or names in the local namespace for the scope where the import statement occurs.

After the requested module numpy is retrieved successfully, it will be made available in the local namespace. There are three ways by which this can be done:

```
#First way:
import numpy as np
#Second way:
import numpy
#Third way:
import matplotlib.pyplot
```

In the first way, if the module is followed by as, then the name following as is bound directly to the imported module. In our application, we have made use of the first method to import numpy module and it is bound directly to the name np.

To completely cover the knowledge points of the basic import system, we also explain the two other ways, although we did not use it in our application. In the second way, if the module imported is a top-level module and no other name is specified, the module's name is bound to the local namespace as a reference to the imported module.

Thirdly, if the module imported is not a top-level module, then the name of the top-level package that contains the module is bound in the local namespace as a reference to the top-level package. Since numpy is a top-level module, in our example we demonstrate this with the module pyplot, where matplotlib is the top-level package. For modules imported this way, it must be accessed by its full qualified name rather than directly.

4.2.1.2 Import Statements with the from clause

Import statements with from clause is executed in a more complex manner:

1. Find the module (in this case math), loading and initialising where necessary.
2. For each of the identifiers specified in the import clauses (in this case floor and exp):
    - Check if imported module has an attribute by that name.
    - If it does not, then attempt to import a submodule with that name, then check the imported module again for that attribute.
    - If the attribute is still not found, an ImportError is raised.
    - Otherwise, a reference to that value is stored in the local namespace. If as clause is present, using the name specified after the as clause, if not, just using the attribute name.

This can be seen in our Option Pricing Calculator code as shown below:

```
from math import floor, exp
```

Both floor and exp are functions provided by the math module, so math.floor() and math.exp() functions are imported, and bound to the names floor and exp respectively. The function floor(x) returns the largest integer less than or equal to x. The function exp(x) returns e=2.718281… to the power x.

However, apart from the way we used in our Option pricing Calculator, there is another way to specify the indicators, which is using the star *, as shown in the example below:

```
#Using * to specify the identifiers
from pandas import *
```

Note that this wildcard form of the import statement is only allowed at the module level – if it is used on class or function definitions, this will raise a SyntaxError. When * is used to replace the identifiers, all public names defined in the module are bound in the local namespace for the scope where the import statement occurs. This means that not everything is imported, only public names defined by the module.

Public names can be determined by a few conditions:

1. Check module namespace for variable named __all__.
2. If defined, it must be a sequence of strings which are names defined or imported by the module.

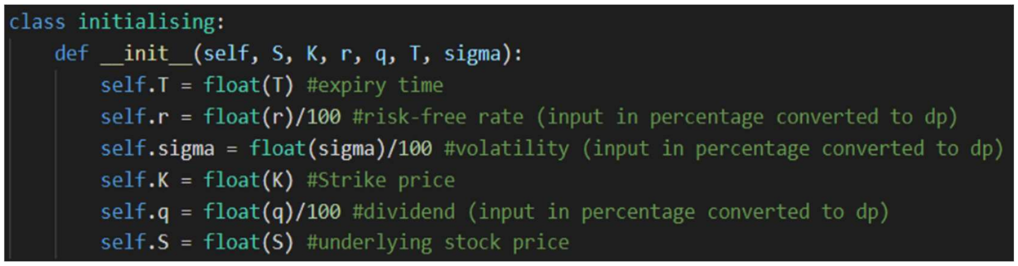
3. If not defined, the set of public names include all names found in the module's namespace not beginning with an underscore.

4.2.2 Using Class Implementation for Option Pricing Calculator

Next, our calculator will involve class implementation, and we have defined two classes: initialising and Option_Pricing. Using these two class definitions, we can demonstrate the use of instance methods, and inheritance in Python.

4.2.2.1 Instance Methods and Auto-Initialisation

The initialising class is defined as such:



```python
class initialising:
    def __init__(self, S, K, r, q, T, sigma):
        self.T = float(T) #expiry time
        self.r = float(r)/100 #risk-free rate (input in percentage converted to dp)
        self.sigma = float(sigma)/100 #volatility (input in percentage converted to dp)
        self.K = float(K) #Strike price
        self.q = float(q)/100 #dividend (input in percentage converted to dp)
        self.S = float(S) #underlying stock price
```

S, K, r, q, T, sigma are the parameters for inputs that we need from the user of the Option pricing Calculator. Using this, we can demonstrate the use of the self keyword and __init__() reserved method in python classes, which are part of using instance methods.

The self keyword is a parameter of the instance method, which points to an instance of class initialising when the method is called. Through the self parameter, instance methods can freely access attributes and methods on the same object to modify them. For example, within our Option Pricing Calculator, the formula parameters S, K, r, q, T, sigma are accessed through instance methods, like self.T points to an instance of T.

In our Option Pricing Calculator, the class initialising has also defined an __init__() method, which is a special method that creates objects with instances customised to a specific initial state. Since we have defined an __init__() method here, class instantiation automatically invokes __init__() for any newly-created class instance. In Objected-Oriented Programming, this is also called a constructor.

In this case, the __init__() method is called when S, K, r, q, T, sigma are created from initialising and allows our class to initialise the attributes accordingly. Notably, we initialise all inputs as float type using the float() function. Since in GUI we input the r, sigma, q in percentage, hence we convert all them by dividing 100.

4.2.2.2 Inheritan*ce*

The second class defined in our Option Pricing Calculator is Option_Pricing, as shown below:

```
class Option_Pricing (initialising):

    def explicit(self, M, N): …
```

Within the class, the function explicit is transformed into a method, and can now be called using the dot notation as such: Option_Pricing.explicit(M, N), where M is the number of price steps and N is the number of time steps.

The method explicit also is able to call values from the base class initialising, because Python classes support inheritance. Here, the derived class is Option_Pricing and the base class is initialising, which allows the parameters named in initialising to be referenced in Option_Pricing.explicit().

This is because when a class object is created from Option_Pricing, base class initialising is remembered. Since requested attributes like S, K, r, q, T, sigma are not found within the Option_Pricing class, they will be searched for within the base class initialising.

4.2.2.3 Object-Oriented Features of Python

Python is an object-oriented language, meaning it supports Object-Oriented Programming (OOP). OOP is a style of programming by which data and operations can be organised into classes and methods. Although our demonstration only scratches the surface, we wanted to show the Python language features that make it an object-oriented language.

The __init__() method makes code concise by automatically initialising an object's attributes whenever a new object is created.

Additionally, by defining explicit as a method within a class instead of a function, we are able to invoke the object as the active agent, rather than the function as the active agent. An example is shown below:

```
explicit(S, K, r, q, T, sigma, M, N) #function is the active agent
Option_Pricing.explicit(M, N) #object is the active agent
```

Since this report is introductory for beginners, we will not spend excessive amounts of time going into deep details of OOP. However, we felt it was relevant to at least introduce the object-oriented features in Python so that readers can have an idea of how Python can be used to support OOP.

4.2.3 Implementing Explicit Method

Our approach to implementing the explicit method can e broken down into 5 steps:

1. Discretisation
2. Create initial matrix
3. Compute matrix A
4. Compute F_i matrix
5. Find K and option price

These steps are implemented within explicit.

4.2.3.1 Discretisation

```
def explicit(self, M, N):
      #Step 1: Discretisation
        S_max = self.K*2
        dt = self.T/N
        ds = S_max/M
```

The first step is to discretise the time and price step to obtain the length of each step, stored in variables dt and ds, where S_max represents the upper limit of the stock price, assumed to be twice that of the strike price, K.

4.2.3.2 Using numpy to create matrixes

```
        #step 2: to compute F(N,j)
        matrix_call = np.zeros((M+1, N+1)) # Create F_i
        matrix_put = np.zeros((M+1, N+1))
        N_array = np.arange(M+1)
        matrix_call[:,N]= np.maximum(N_array * ds - self.K,0)
        matrix_put[:,N]= np.maximum(self.K - N_array * ds,0)
```

The following functions within the numpy module are be accessed by the following notation, since numpy was bound to the name np in our import statement:

```
np.zeros()
np.arange()
np.maximum()
```

We used np.zeros((M+1, N+1)) to create an array of M+1 by N+1 filled by zeros to initialise matrix_call and matrix_put. Then, we used np.arange(M+1) to create an array of evenly spaced values within the given interval 0 and M+1 with default step size 1, which represents j = 0, 1, 2 … M. Lastly, np.maximum() was used to express the function max(j*(ds) – K, 0) (for call options) and max(j*(ds) – K, 0) (for put options), where j = 0, 1, 2 … M.

Here, we also demonstrate how 2D slicing is supported in numpy. matrix_call[:, N] and matrix_put[:, N] selects all rows and columns up to N in the two matrixes.

Alternatively, this section of code can be written with list comprehension. In the next section, we will try to introduce the reader to list comprehension by re-expressing this part of our code with list comprehension.

4.2.3.3 List Comprehension

List comprehensions provide a concise and more Pythonic way to create lists. Commonly, they are used to make new lists where each element I the result of some operations applied to each member of another sequence of iterables to create subsequence of those elements that satisfy certain conditions.

A list comprehension consists of brackets containing an expression, followed by a for clause, then zero or more for or if clauses. Examples of list comprehension syntax shown below:

```
#Using standard for looping
A = []
for x in [3, 5, 6, 7]:
    for y in [4, 5, 8, 3]:
        if x != y:
            A.append((x, y))

#Using list comprehension
[(x, y) for x in [3, 5, 6, 7] for y in [4, 5, 8, 3] if x != y]
```

The result of list comprehension will always be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow. In this example, the output is a list of tuples:

```
[(3, 4),
 (3, 5),
 (3, 8),
 (5, 4),
 (5, 8),
 (5, 3),
 (6, 4),
 (6, 5),
 (6, 8),
 (6, 3),
 (7, 4),
 (7, 5),
 (7, 8),
 (7, 3)]
```

If an `else` clause is present, it goes after the `if` but before the `for` clauses, as shown below:

```python
#Using standard for looping with else clause
A = []
for x in [3, 5, 6, 7]:
    for y in [4, 5, 8, 3]:
        if x != y:
            A.append((x, y))
        else:
            A.append(x*y)

#Using list comprehension with else clause
[(x, y) if x != y else x*y for x in [3, 5, 6, 7] for y in [4, 5, 8, 3]]
```

The output for this example is a list of both tuples and integers:

```
[(3, 4),

 (3, 5),

 (3, 8),

 9,

 (5, 4),

 25,

 (5, 8),

 (5, 3),

 (6, 4),

 (6, 5),

 (6, 8),

 (6, 3),

 (7, 4),

 (7, 5),

 (7, 8),

 (7, 3)]
```

In essence, as long as the problem can be thought of appending to create a list, list comprehension can be used to express the problem. In the context of our Option Pricing Calculator, we can transform our original array creation as shown below:

```
        matrix_call = np.zeros((M+1, N+1))
        matrix_put = np.zeros((M+1, N+1))
        N_array = np.arange(M+1)
        matrix_call[:,N]= np.maximum(N_array * ds - self.K,0)
        matrix_put[:,N]= np.maximum(self.K - N_array * ds,0)
```

Into the following list comprehension:

```
matrix_call = [[(i * ds - self.K) if (i * ds - self.K)>0 else 0
              for j in range(M+1)]
              for i in range(N+1)]
matrix_put = [[(self.K - i * ds) if (self.K - i * ds)>0 else 0
             for j in range(M+1)]
             for i in range(N+1)]
```

The list comprehension creates a 2D array of M+1 rows and N+1 columns express the function max(j*(ds) – K, 0) (for call options) and max(j*(ds) – K, 0) (for put options), where j = 0, 1, 2 … M.

4.2.4 Using for loop to create matrix A and F_i Matrix

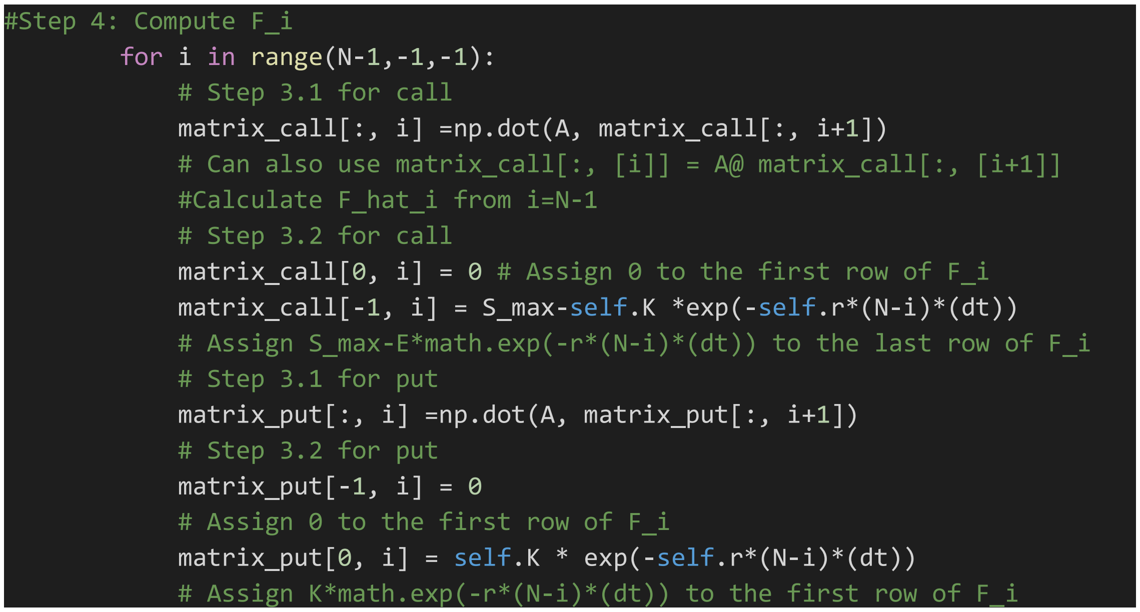
Next, in order to create the matrix A (as outlined in Section 3), we used standard for looping as explained in Section 2:

```
        #Step 3:
        #Compute matrix A
        A = np.zeros((M+1, M+1))
        A[0, 0] = 1
        for j in range(1, M):
            A[j, j-1] = 0.5*(dt)*(self.sigma**2*j**2-(self.r-self.q)*j)
            A[j, j] = 1-dt*(self.sigma**2*j**2+self.r)
            A[j, j+1] = 0.5*(dt)*(self.sigma**2*j**2+(self.r-self.q)*j)
        A[M, M] = 1
```

First, to initialise the matrix, np.zeros((M+1, M+1)) is used to create an array of M+1 by M+1 zeros. Next, through 2D array indexing, we point to the first entry in A matrix, $A_{1,1}$, through the notation

A[0,0] and assign the value from 0 to 1. Afterwards, for loop is used to recursively assign the aj, bj and cj values into the matrix. Lastly, the last entry of the matrix, $A_{M, M}$, is assigned to 1.

From here, the F_i matrix can be computed by taking the dot product of matrix_call/matrix_put and A using for looping again:



```
#Step 4: Compute F_i
        for i in range(N-1,-1,-1):
            # Step 3.1 for call
            matrix_call[:, i] =np.dot(A, matrix_call[:, i+1])
            # Can also use matrix_call[:, [i]] = A@ matrix_call[:, [i+1]]
            #Calculate F_hat_i from i=N-1
            # Step 3.2 for call
            matrix_call[0, i] = 0 # Assign 0 to the first row of F_i
            matrix_call[-1, i] = S_max-self.K *exp(-self.r*(N-i)*(dt))
            # Assign S_max-E*math.exp(-r*(N-i)*(dt)) to the last row of F_i
            # Step 3.1 for put
            matrix_put[:, i] =np.dot(A, matrix_put[:, i+1])
            # Step 3.2 for put
            matrix_put[-1, i] = 0
            # Assign 0 to the first row of F_i
            matrix_put[0, i] = self.K * exp(-self.r*(N-i)*(dt))
            # Assign K*math.exp(-r*(N-i)*(dt)) to the first row of F_i
```

This for loop uses range(N-1, -1, -1) as the iterator, which represents a range object from N-1 to 0, with a step size of -1. Hence, for i = N-1, N-2, N-3, … 0, the dot product of A and the (i+1)th column of matrix_call/matrix_put is assigned to the ith column of matrix_call/matrix_put recursively. Lastly, the first row of matrix_call/matrix_put is assigned to 0, and the last row of matrix_call is assigned to S_max-E*math.exp(-r*(N-i)*(dt)), and the last row of matrix_put is assigned to K*math.exp(-r*(N-i)*(dt)).

4.2.5 Obtaining Option Price

The final step in the calculator is to find the option price:

```
        #Step 5:
        #Find k
        k = floor(self.S/ds)

        #Option price
        call_price = matrix_call[k, 0] +
        ((matrix_call[k+1,0]-matrix_call[k,0])/ds)*(self.S-k*ds)
        put_price = matrix_put[k, 0] +
        ((matrix_put[k+1,0]-matrix_put[k,0])/ds)*(self.S-k*ds)

        return {'call':call_price,'put':put_price}
```

The variable k is the floor division of underlying stock price by ds, which helps us find the location of the underlying stock in the price step. Next, call_price and put_price stores the value of the call and oput option price, calculated using the appropriate formula outlined in Section 3.

The very last statement in our explicit() method is a return statement. This will return a dictionary with keys [‘call’, ‘put’] with the values of call_price and put_price respectively.

5. GUI Implementation

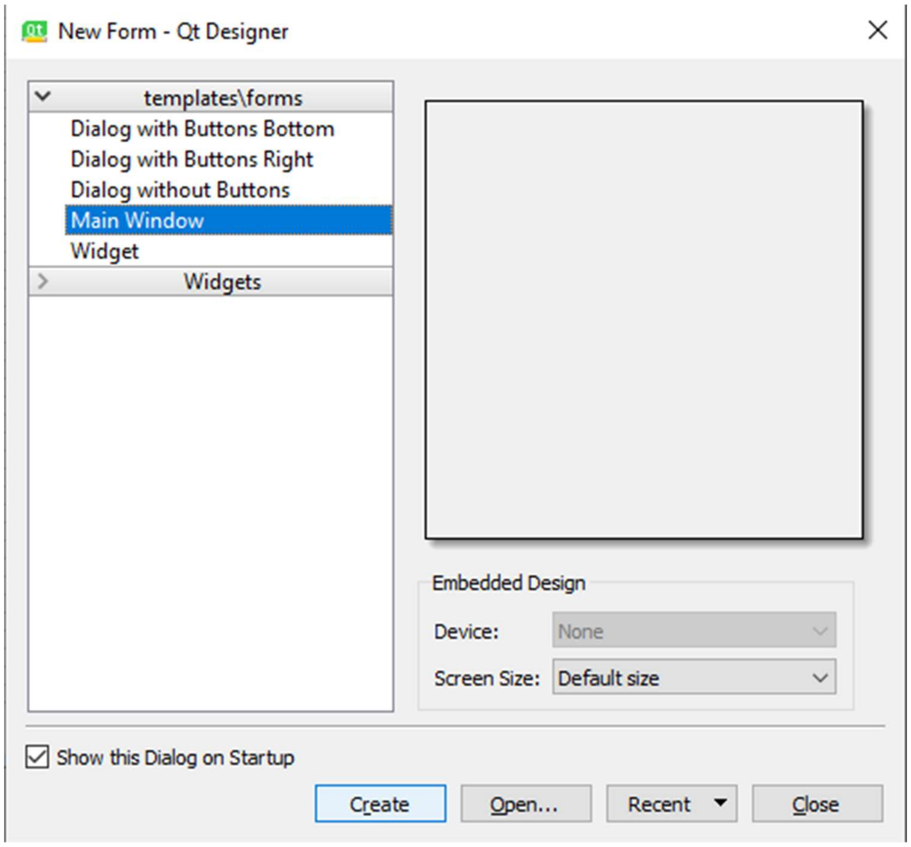
With the aim of creating a user interface, we made a Graphical User Interface (GUI) which enables easier access to the functions we built. Rather than input text-based command lines, GUIs is way more popular and user-friendly by its visualization. Moreover, we chose **Qt Designer** as a tool to build this GUI.

5.1. Qt Designer Application

The link to download Qt Designer application:

https://build-system.fman.io/qt-designer-download

After installation, open the application and selected **Main Window** and **Create**.

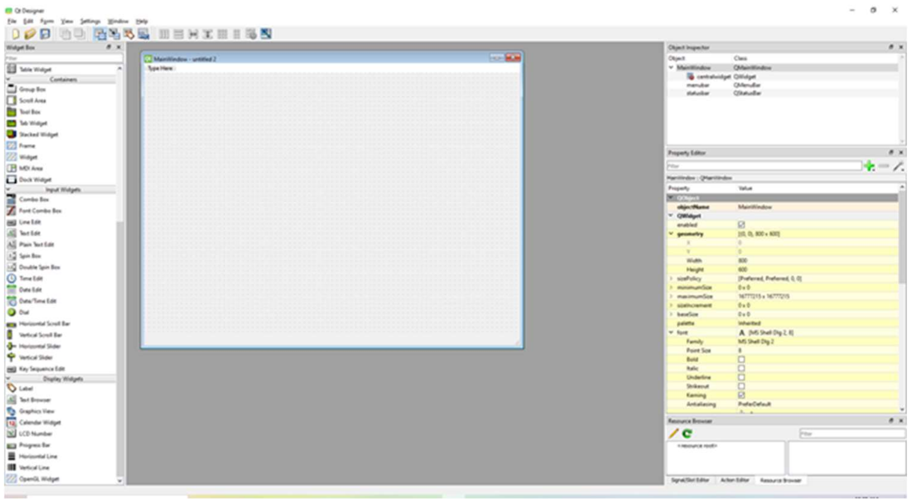


Device: None

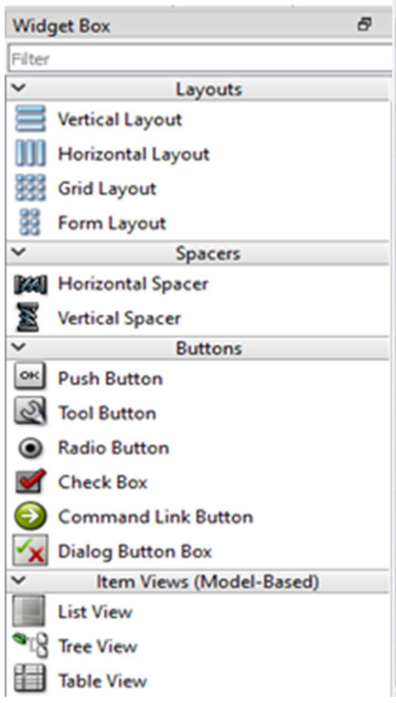
Screen Size: Default size

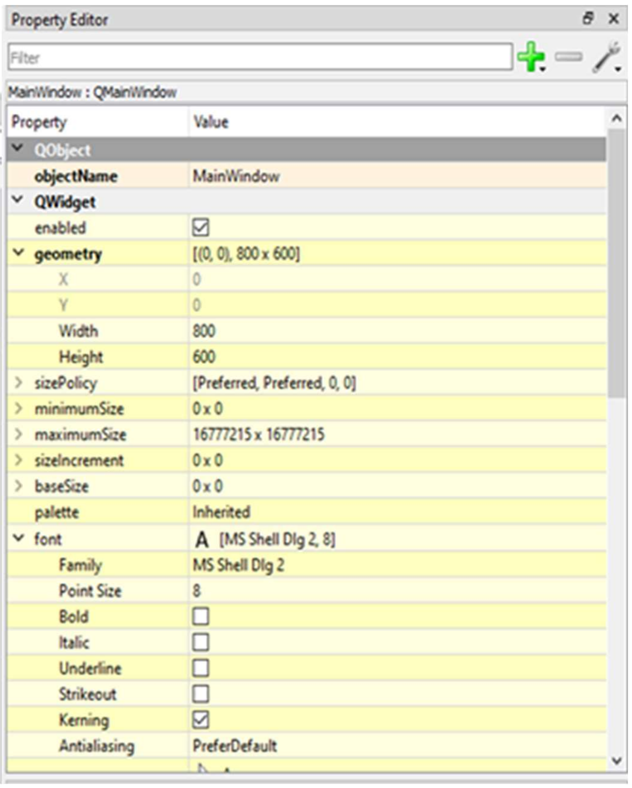
☑ Show this Dialog on Startup

This would create a new UI. file and direct user to the main designer environment, where we can build and design our GUI. There are a few key tools to design the GUI effectively: Widget Box, Property Editor, Object Inspector.



Main Designer Environment



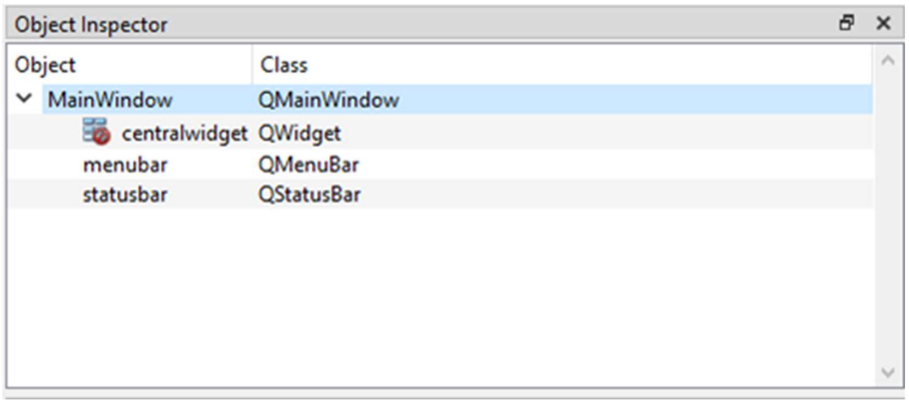

Property Editor

Filter

MainWindow : QMainWindow

Property	Value
QObject	
objectName	MainWindow
QWidget	
enabled	☑
geometry	[(0, 0), 800 x 600]
X	0
Y	0
Width	800
Height	600
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [MS Shell Dlg 2, 8]
Family	MS Shell Dlg 2
Point Size	8
Bold	☐
Italic	☐
Underline	☐
Strikeout	☐
Kerning	☑
Antialiasing	PreferDefault

Widget Box *Property Editor*


Object Inspector

Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
menubar	QMenuBar
statusbar	QStatusBar

Object Inspector

5.1.1 Widget Box

The Widget Box is on the left side of the screen and it has all the visual widgets to display the elements of GUI. These widgets icons can be dragged and dropped at any positions in the Main Window. The functions of important widgets we used were:

Display Widgets – **Label**

Button – **Push Button**

Input Widgets – **Line Edit**, **Date Edit**

The **Label** widget is used to create the title name of the application, the descriptions for the input widgets, and the blank fields to display the output of option pricing.

The **Push Button** widget is used to trigger the option price calculation.

The **Line Edit** widget is used to input the values required for the option pricing.

The **Date Edit** widget is used to provide a way of selecting option beginning date and execution date.

5.1.2 Property Editor

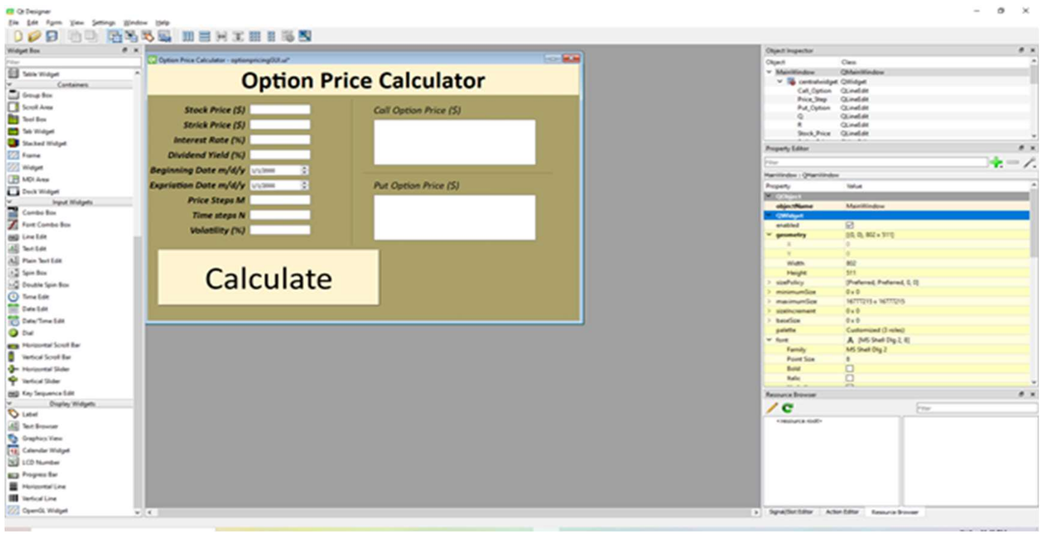
The Property Editor is on the right side of the screen, and it allows user to change the properties like size, position, and color of the objects. Moreover, it allows user to change the objects' names, which is significant for linking Python code with the GUI objects.

5.1.3 Object Inspector

The Object Inspector is on the top-right side of the screen and shows the status of the objects inside the Main Window, which allows user to check if all objects have been properly named.

5.1.4 Design

After inserting widgets and button into the Main Window with formatting, our design of the GUI was as shown below:



Final GUI Design

The left-hand side blank fields are used to input values for option price. The right-hand side blank fields are used to display the option price values. The yellow “Calculate” button is the Push Button object to trigger option calculation. The rest of elements in the window are Label objects. Finally, we named this file “optionpricingGUI.ui” and we would use it for Python code reference later.

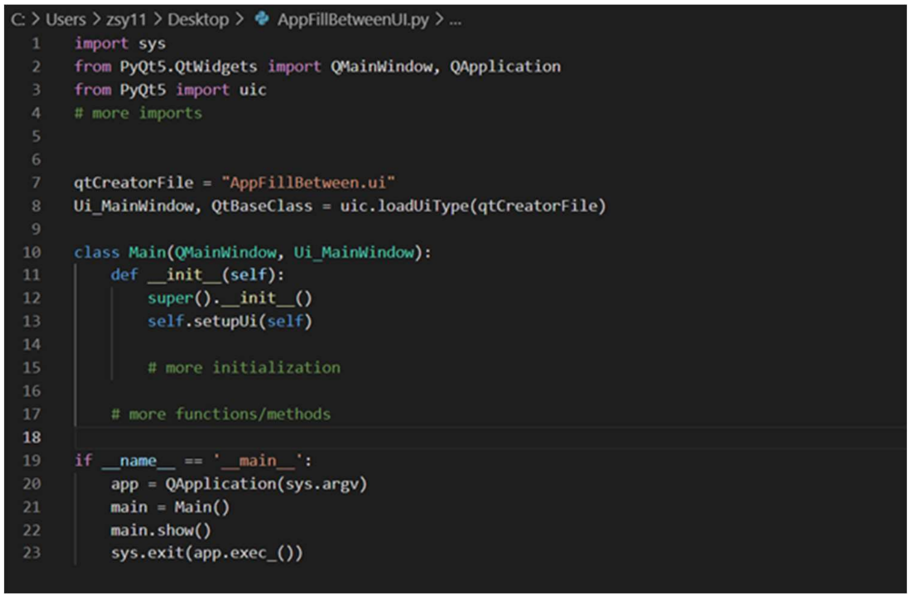
5.2. Connecting with Python Code

The next step is to link the objects to Python code. Because of the connection, the GUI can intake user inputs for option pricing, such as underlying stock price and strike price. Also, GUI can display the calculation result: call price, put price, when user clicks “calculate” button.

5.2.1 “main.py’’ file

Creating a “main.py” file to store our code for GUI implementation.

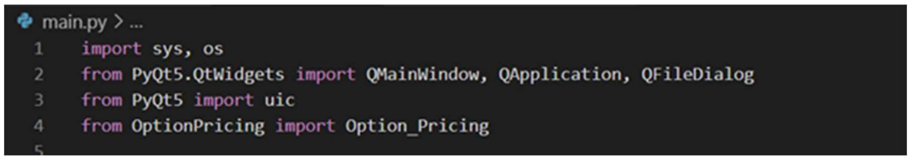
Using the following template to execute UI file loading and GUI launching.



```
C: > Users > zsy11 > Desktop > AppFillBetweenUI.py > ...
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5 import uic
# more imports


qtCreatorFile = "AppFillBetween.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        # more initialization

    # more functions/methods

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = Main()
    main.show()
    sys.exit(app.exec_())
```

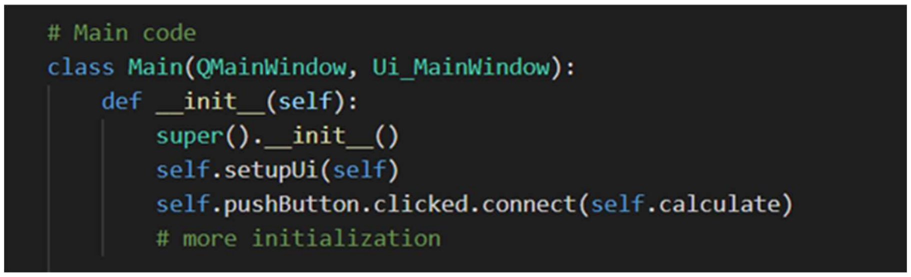
Template

Next, using from…import… argument to add in a class, Option_Pricing, for calculating option price



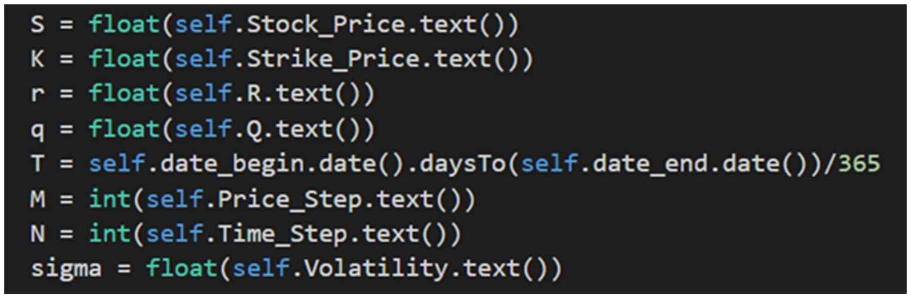
```
main.py > ...
import sys, os
from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog
from PyQt5 import uic
from OptionPricing import Option_Pricing
```

Following that, we expanded on the initialization lines under the __init__ function inside the Main class. We added in one line to allow use of Push Button widgets created in the Qt Designer by referring the Push Button object's name pushButton, to make sure the functionality of running our calculation step upon being clicked.



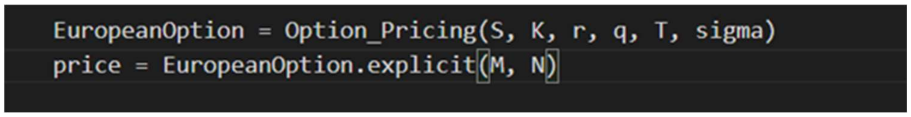
```
# Main code
class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.pushButton.clicked.connect(self.calculate)
        # more initialization
```

Under the Main class, we need to add in another function, namely calculate, to be called upon clicking the “calculate” button. In this function, we assigned the user inputs in the Line Edit objects to variables for calculation.



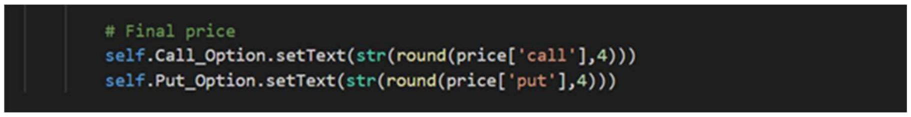
```
S = float(self.Stock_Price.text())
K = float(self.Strike_Price.text())
r = float(self.R.text())
q = float(self.Q.text())
T = self.date_begin.date().daysTo(self.date_end.date())/365
M = int(self.Price_Step.text())
N = int(self.Time_Step.text())
sigma = float(self.Volatility.text())
```

We then assign the user input to the Option_Pricing class to a variable called EuropeanOption for easier reference.



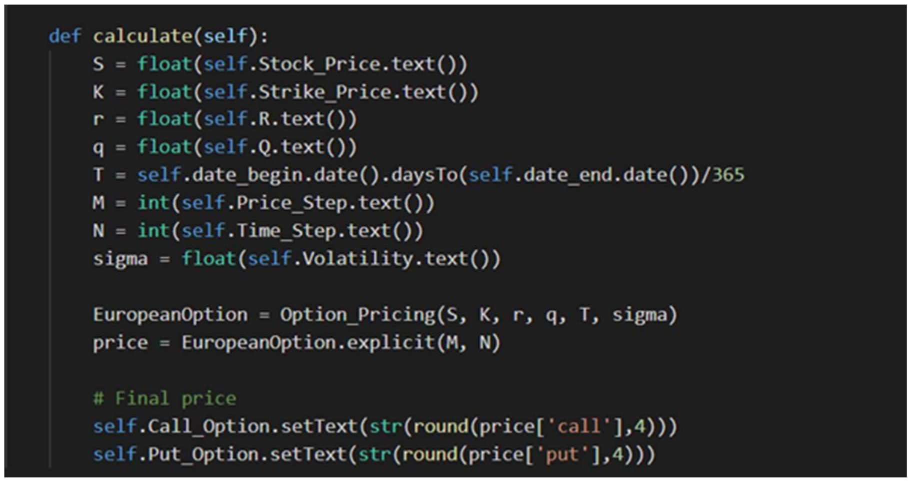
```
EuropeanOption = Option_Pricing(S, K, r, q, T, sigma)
price = EuropeanOption.explicit(M, N)
```

After the calculation is done, the outputs will be placed into two Label objects: Call_Option and Put_Option, and we rounded the outputs to 4 decimal.



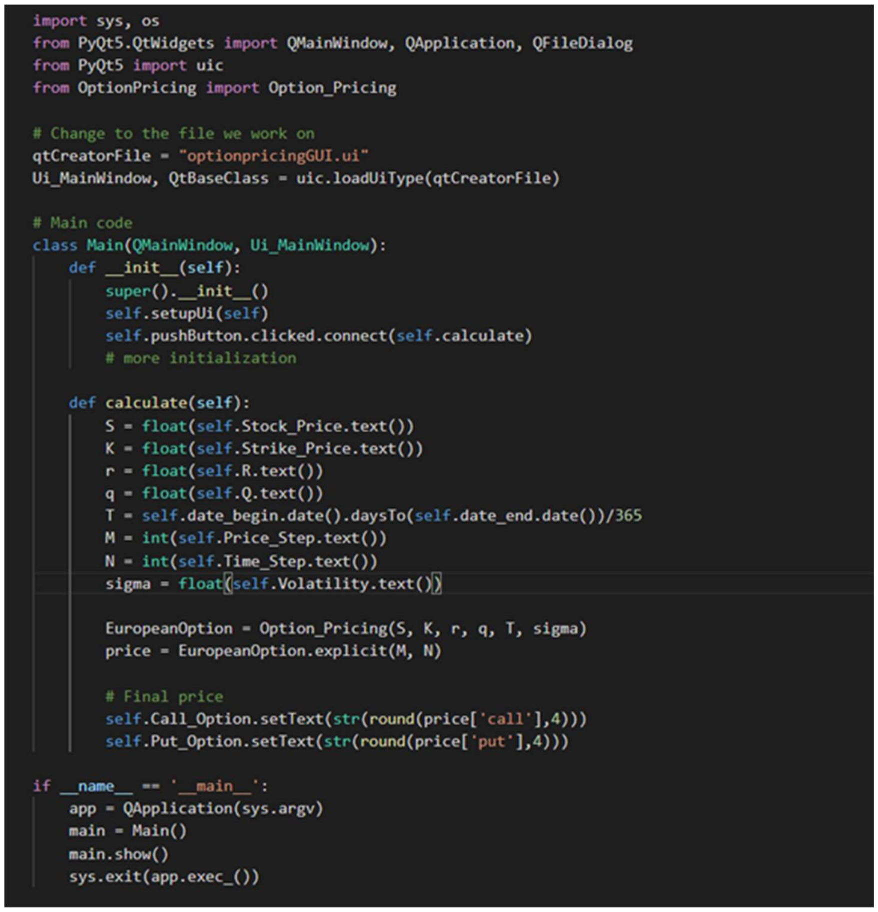
```
        # Final price
        self.Call_Option.setText(str(round(price['call'],4)))
        self.Put_Option.setText(str(round(price['put'],4)))
```

Next is an overview of calculate function code:



```
def calculate(self):
    S = float(self.Stock_Price.text())
    K = float(self.Strike_Price.text())
    r = float(self.R.text())
    q = float(self.Q.text())
    T = self.date_begin.date().daysTo(self.date_end.date())/365
    M = int(self.Price_Step.text())
    N = int(self.Time_Step.text())
    sigma = float(self.Volatility.text())

    EuropeanOption = Option_Pricing(S, K, r, q, T, sigma)
    price = EuropeanOption.explicit(M, N)

    # Final price
    self.Call_Option.setText(str(round(price['call'],4)))
    self.Put_Option.setText(str(round(price['put'],4)))
```

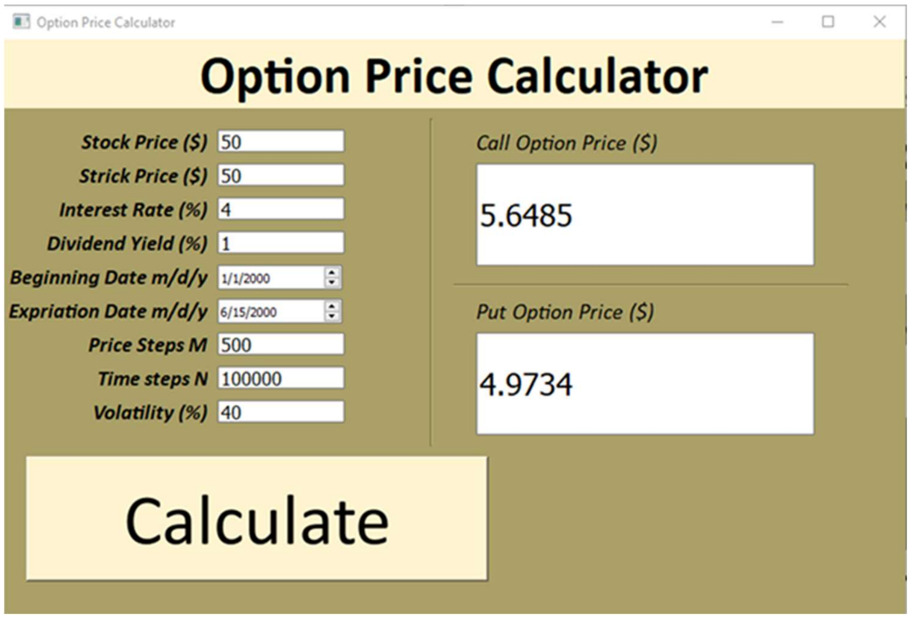
Putting these parts together, our complete GUI code is shown below:



```python
import sys, os
from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog
from PyQt5 import uic
from OptionPricing import Option_Pricing

# Change to the file we work on
qtCreatorFile = "optionpricingGUI.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

# Main code
class Main(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.pushButton.clicked.connect(self.calculate)
        # more initialization

    def calculate(self):
        S = float(self.Stock_Price.text())
        K = float(self.Strike_Price.text())
        r = float(self.R.text())
        q = float(self.Q.text())
        T = self.date_begin.date().daysTo(self.date_end.date())/365
        M = int(self.Price_Step.text())
        N = int(self.Time_Step.text())
        sigma = float(self.Volatility.text())

        EuropeanOption = Option_Pricing(S, K, r, q, T, sigma)
        price = EuropeanOption.explicit(M, N)

        # Final price
        self.Call_Option.setText(str(round(price['call'],4)))
        self.Put_Option.setText(str(round(price['put'],4)))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = Main()
    main.show()
    sys.exit(app.exec_())
```

Main.py

Finally, running the "Main.py" file in Python terminal (Note that three files "Main.py", "OptionPricing.py", "optionpricingGUI.ui" should in same directory) and GUI window will appear. Try some inputs number on the left-hand side blank fields, then the option price results will be displayed on the right-hand side blank fields. Seen as below:



Option Price Calculator

Stock Price ($)	50
Strick Price ($)	50
Interest Rate (%)	4
Dividend Yield (%)	1
Beginning Date m/d/y	1/1/2000
Expriation Date m/d/y	6/15/2000
Price Steps M	500
Time steps N	100000
Volatility (%)	40

Call Option Price ($)

5.6485

Put Option Price ($)

4.9734

User Interface (with test result)

6. References

Hayes, A. (2021, March 30). *Black-Scholes Model*. Received from https://www.investopedia.com/terms/b/blackscholes.asp

Hayes, A. (2020, February 6). *Greeks*. Received from https://www.investopedia.com/terms/g/greeks.asp

Zhang, Q. (2017, May 24). *On Pricing Options with Finite Difference Methods*. Received from https://quintus-zhang.github.io/post/on_pricing_options_with_finite_difference_methods/