GROUP A & B – EXACT PRICING METHODS

1. **Output from execution of Group A&B code, along with any commentary.**

<u>Exact Solutions of One-Factor Plain Options</u>

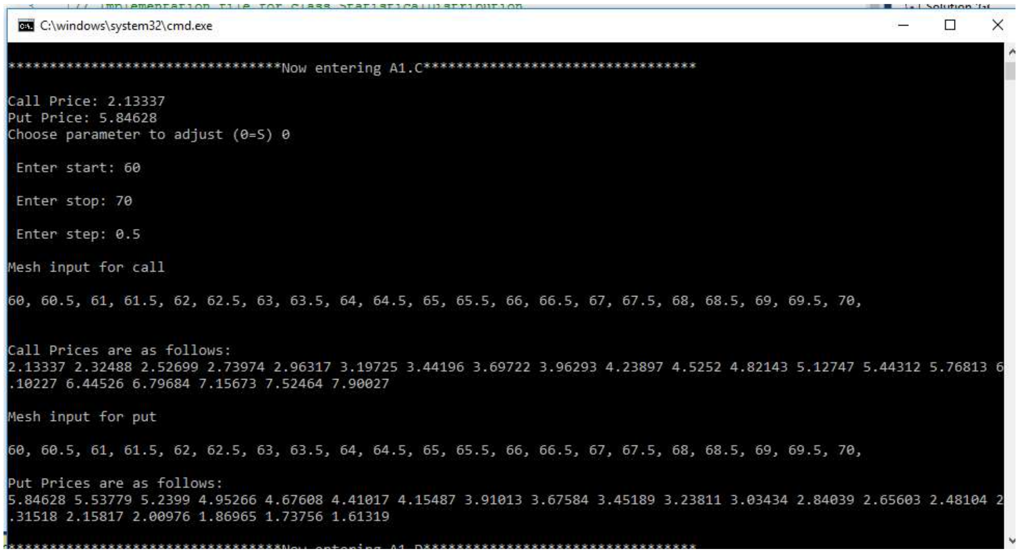
a) *Implement the above formulae for call and put option pricing using the data sets Batch 1 to Batch 4. Check your answers, as you will need them when we discuss numerical methods for option pricing.*

Refer to b)

b) *Apply the put-call parity relationship to compute call and put option prices. For example, given the call price, compute the put price based on this formula using Batches 1 to 4. Check your answers with the prices from part a). Note that there are two useful ways to implement parity: As a mechanism to calculate the call (or put) price for a corresponding put (or call) price, or as a mechanism to check if a given set of put/call prices satisfy parity. The ideal submission will neatly implement both approaches.*

```
C:\windows\system32\cmd.exe
******************************BATCH 1*************************************
Call Price: 2.13337
Call price from put-call parity: 2.13337
Put Price: 5.84628
Put price from put-call parity: 5.84628
put-call parity test: Put Call Parity satisfied for batch


******************************BATCH 2*************************************
Call Price: 7.96557
Call price from put-call parity: 7.96557
Put Price: 7.96557
Put price from put-call parity: 7.96557
put-call parity test: Put Call Parity satisfied for batch


******************************BATCH 3*************************************
Call Price: 0.204058
Call price from put-call parity: 0.204058
Put Price: 4.07326
Put price from put-call parity: 4.07326
put-call parity test: Put Call Parity satisfied for batch


******************************BATCH 4*************************************
Call Price: 92.1757
Call price from put-call parity: 92.1757
Put Price: 1.2475
Put price from put-call parity: 1.2475
put-call parity test: Put Call Parity satisfied for batch
```

c) *Say we wish to compute option prices for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector. This entails calling the option pricing formulae for each value S and each computed option price will be stored in a std::vector<double> object. It will be useful to write a global function that produces a mesh array of doubles separated by a mesh size h.*



d) *Now we wish to extend part c and compute option prices as a function of i) expiry time, ii) volatility, or iii) any of the option pricing parameters. Essentially, the purpose here is to be able to input a matrix (vector of vectors) of option parameters and receive a matrix of option prices as the result. Encapsulate this functionality in the most flexible/robust way you can think of.*

C:\windows\system32\cmd.exe

```
**********************************Now entering A1.D**********************************

Choose parameter to adjust (0=S, 1=K, 2=r, 3=T, 4=sig, 5=b) 1

 Enter start: 65

 Enter stop: 75

 Enter step: 1


*****Now printing matrix of parameters from mesher prior to pricing*********************************


S    K    r     T   sig   b
60, 65, 0.08, 0.25, 0.3, 0.08,

60, 66, 0.08, 0.25, 0.3, 0.08,

60, 67, 0.08, 0.25, 0.3, 0.08,

60, 68, 0.08, 0.25, 0.3, 0.08,

60, 69, 0.08, 0.25, 0.3, 0.08,

60, 70, 0.08, 0.25, 0.3, 0.08,

60, 71, 0.08, 0.25, 0.3, 0.08,

60, 72, 0.08, 0.25, 0.3, 0.08,
```

C:\windows\system32\cmd.exe

```
60, 73, 0.08, 0.25, 0.3, 0.08,

60, 74, 0.08, 0.25, 0.3, 0.08,

60, 75, 0.08, 0.25, 0.3, 0.08,


Now printing vector of call prices

2.13337, 1.83988, 1.57982, 1.3507, 1.14994, 0.97498, 0.823313, 0.692506, 0.580247, 0.484371, 0.402866,


Now printing vector of put prices

5.84628, 6.53299, 7.25313, 8.0042, 8.78364, 9.58889, 10.4174, 11.2668, 12.1348, 13.0191, 13.9178,


******************************Option Sensitivites Section************************************
Call Delta: 0.594629  Put Delta: -0.356601
Gamma: 0.0134936

******************************A2.b************************************
```

Option Sensitivities, aka the Greeks

a) *Implement the above formulae for call and put for gamma for call and put future option pricing using the data set: K = 100, S = 105, T = 0.5, r = 0.1, b = 0 and sig = 0.36. (exact delta call = 0.5946, delta put = -0.3566).*

```
******************************Option Sensitivites Section************************************
Call Delta: 0.594629  Put Delta: -0.356601
Gamma: 0.0134936
```

b) *We now use the code in part a to compute call delta price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and it entails calling the above formula for a call delta for each value S and each computed option price will be store in a std::vector<double> object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.*

```
******************************A2.b************************************


Choose parameter to adjust (0=S) 0
Enter start: 105
Enter stop: 115
Enter step: 1



Mesh input:

105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,


Delta Call:
0.594629 0.607976 0.621025 0.633767 0.646196 0.658306 0.670094 0.681556 0.692691 0.703497 0.713974

Delta Put :
-0.356601 -0.343253 -0.330205 -0.317463 -0.305034 -0.292923 -0.281135 -0.269673 -0.258539 -0.247733 -0.237256
```

c) *Incorporate this into your above matrix pricer code, so you can input a matrix of option parameters and receive a matrix of either Delta or Gamma as the result.*

```
******************************A2.C************************************


Choose parameter to adjust (0=S, 1=K, 2=r, 3=T, 4=sig, 5=b) 2
Enter start: 0.1
Enter stop: 0.2
Enter step: 0.01




*****Now printing matrix of parameters from mesher prior to pricing**********************************


S    K    r     T    sig   b
105, 100, 0.1, 0.5, 0.36, 0,

105, 100, 0.11, 0.5, 0.36, 0,

105, 100, 0.12, 0.5, 0.36, 0,

105, 100, 0.13, 0.5, 0.36, 0,

105, 100, 0.14, 0.5, 0.36, 0,

105, 100, 0.15, 0.5, 0.36, 0,

105, 100, 0.16, 0.5, 0.36, 0,
```

```
105, 100, 0.17, 0.5, 0.36, 0,

105, 100, 0.18, 0.5, 0.36, 0,

105, 100, 0.19, 0.5, 0.36, 0,



Delta Call:
0.594629 0.591663 0.588712 0.585776 0.582854 0.579947 0.577055 0.574177 0.571313 0.568464

Delta Put:
-0.356601 -0.354822 -0.353053 -0.351292 -0.34954 -0.347796 -0.346062 -0.344336 -0.342618 -0.340909
Gamma:
0.0134936 0.0134263 0.0133594 0.0132927 0.0132264 0.0131605 0.0130948 0.0130295 0.0129645 0.0128999
```

d) *We now use divided differences to approximate option sensitivities. In some cases, an exact formula may not exist (or is difficult to find) and we resort to numerical methods. In general, we can approximate first and second-order derivatives in S by 3-point second order approximations, for example:*

$$\Delta = \frac{V(S+h)-V(S-h)}{2h}$$

$$\Gamma = \frac{V(S+h)-2V(S)+V(S-h)}{h^2}.$$

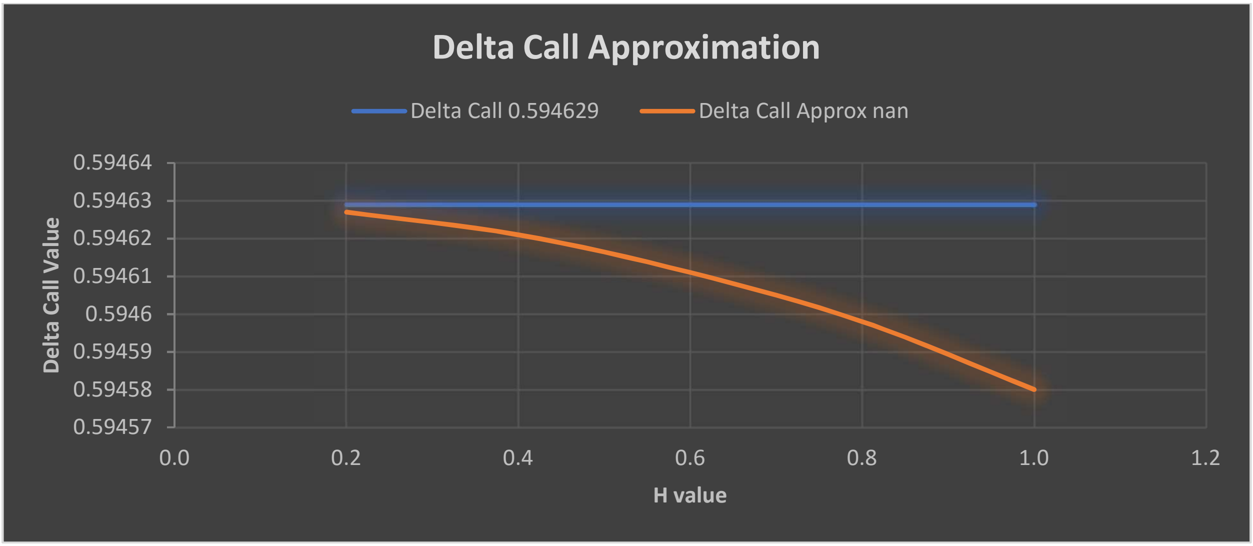
In this case the parameter h is 'small' in some sense. By Taylor's expansion you can show that the above approximations are second order accurate in h to the corresponding derivatives.

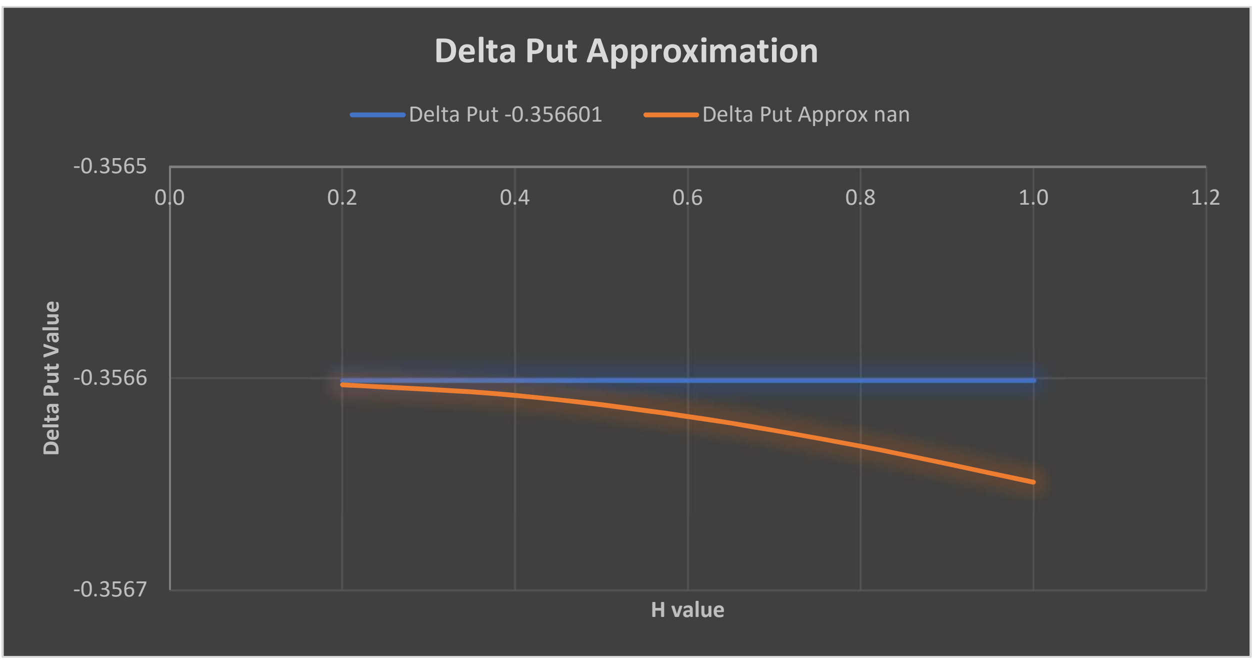
The objective of this part is to perform the same calculations as in parts a and b, but now using divided differences. Compare the accuracy with various values of the parameter h (In general, smaller values of h produce better approximations but we need to avoid round-offer errors and subtraction of quantities that are very close to each other). Incorporate this into your well-designed class structure.

Analysis:

As shown below in the plot data and the Excel graphs (following page), we can observe that as H increases in increments of 0.2, we can see the accuracy gradually degrade – more so in the first-order derivatives than second-order derivatives. We also observe at H=0, the issue associated with round-offer errors. Therefore, it's typically best to avoid approximating at H=0.

H	Delta Call	Delta Call Approx	Delta Put	Delta Put Approx	Gamma	Gamma Call Approx	Gamma Put Approx
0.0	0.594629	nan	-0.3566	nan	0.01349	nan	nan
0.2	0.594629	0.594627	-0.3566	-0.356603	0.01349	0.0134936	0.0134936
0.4	0.594629	0.594621	-0.3566	-0.356608	0.01349	0.0134935	0.0134935
0.6	0.594629	0.594611	-0.3566	-0.356618	0.01349	0.0134935	0.0134935
0.8	0.594629	0.594598	-0.3566	-0.356632	0.01349	0.0134931	0.0134931
1.0	0.594629	0.59458	-0.3566	-0.356649	0.01349	0.0134928	0.0134928


Delta Call Approximation
Delta Call 0.594629
Delta Call Approx nan
0.59464
0.59463
0.59462
0.59461
0.5946
0.59459
0.59458
0.59457
Delta Call Value
0.0
0.2
0.4
0.6
0.8
1.0
1.2
H value


Delta Put Approximation
Delta Put -0.356601
Delta Put Approx nan
-0.3565
-0.3566
-0.3567
Delta Put Value
0.0
0.2
0.4
0.6
0.8
1.0
1.2
H value

```
*******************************A2.d - Divided Differences*********************************



*************************HOLDING S CONSTANT, WITH VARYING LEVELS OF H BETWEEN [0,1]*******************

Enter step: 0.15

Call Delta: 0.594629  Put Delta: -0.356601
Gamma: 0.0134936

Parameter S: 0.1 with H value of:  0
Delta Call: 0.594629
Delta Call Approx: -nan(ind)
Delta Put: -0.356601
Delta Put Approx: -nan(ind)
Gamma: 0.0134936
Gamma Call Approx: -nan(ind)
Gamma Put Approx: -nan(ind)

Parameter S: 0.1 with H value of:  0.15
Delta Call: 0.594629
Delta Call Approx: 0.594628
Delta Put: -0.356601
Delta Put Approx: -0.356602
Gamma: 0.0134936
Gamma Call Approx: 0.0134936
```

C:\windows\system32\cmd.exe

```
Gamma Put Approx: 0.0134936

Parameter S: 0.1 with H value of:  0.3
Delta Call: 0.594629
Delta Call Approx: 0.594624
Delta Put: -0.356601
Delta Put Approx: -0.356605
Gamma: 0.0134936
Gamma Call Approx: 0.0134936
Gamma Put Approx: 0.0134936

Parameter S: 0.1 with H value of:  0.45
Delta Call: 0.594629
Delta Call Approx: 0.594619
Delta Put: -0.356601
Delta Put Approx: -0.356611
Gamma: 0.0134936
Gamma Call Approx: 0.0134935
Gamma Put Approx: 0.0134935

Parameter S: 0.1 with H value of:  0.6
Delta Call: 0.594629
Delta Call Approx: 0.594611
Delta Put: -0.356601
Delta Put Approx: -0.356618
Gamma: 0.0134936
Gamma Call Approx: 0.0134933
Gamma Put Approx: 0.0134933

Parameter S: 0.1 with H value of:  0.75
```

```
Parameter S: 0.1 with H value of:  0.75
Delta Call: 0.594629
Delta Call Approx: 0.594602
Delta Put: -0.356601
Delta Put Approx: -0.356628
Gamma: 0.0134936
Gamma Call Approx: 0.0134932
Gamma Put Approx: 0.0134932

Parameter S: 0.1 with H value of:  0.9
Delta Call: 0.594629
Delta Call Approx: 0.59459
Delta Put: -0.356601
Delta Put Approx: -0.35664
Gamma: 0.0134936
Gamma Call Approx: 0.013493
Gamma Put Approx: 0.013493
```

Perpetual American Options

a) *Program the above formulae, and incorporate into your well-designed options pricing classes.*

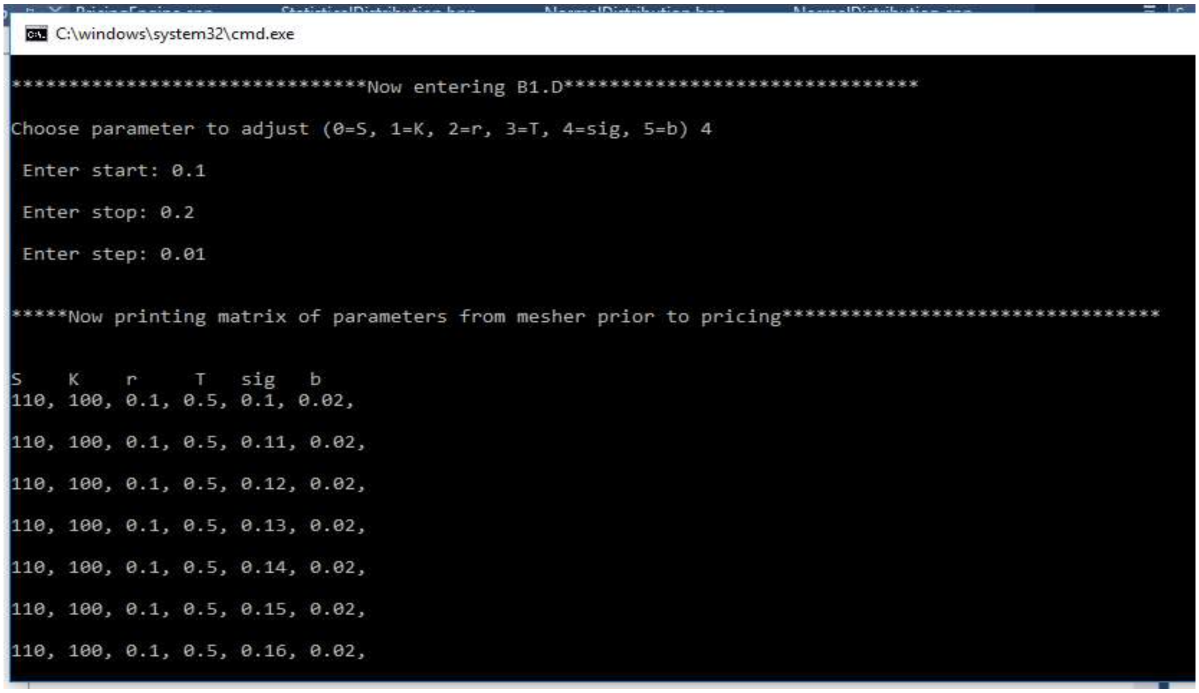
b) *Test the data with K = 100, sig = 0.1, r = 0.1, b = 0.02, S = 110 (check C = 18.5035, P = 3.03106).*

```
*****************************Perpetual American Options************************************
Call Price: 18.5035
Put Price: 3.03106
```

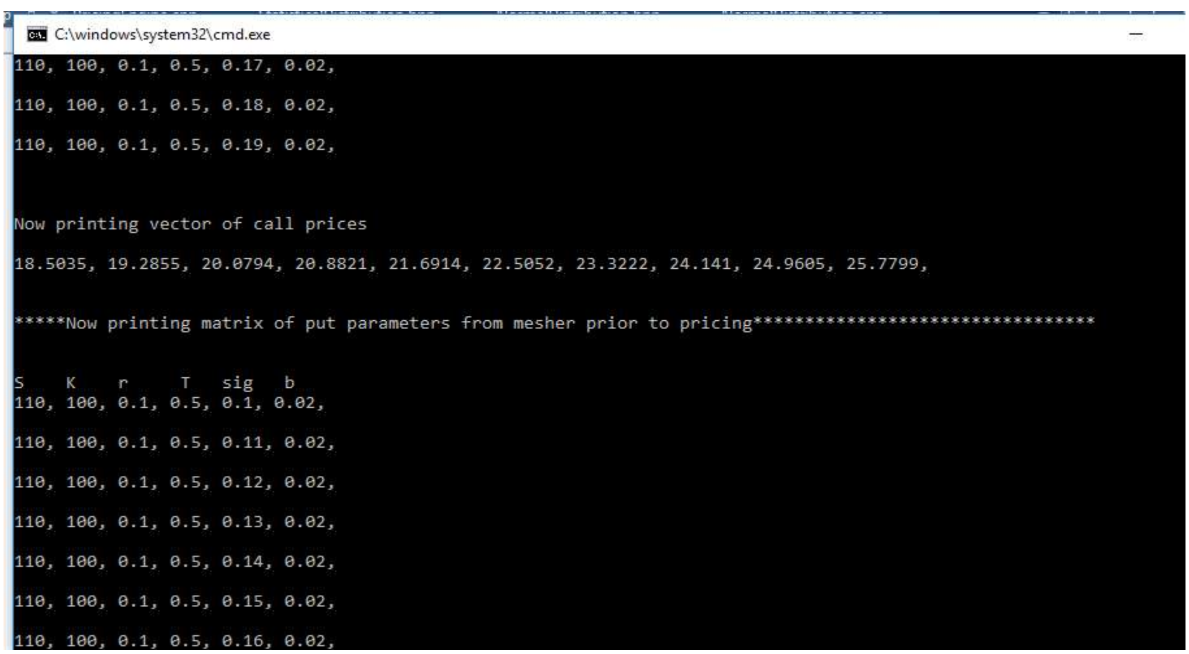
c) *We now use the code in part a) to compute call and put option price for a monotonically increasing range of underlying values of S, for example 10, 11, 12, …, 50. To this end, the output will be a vector and this exercise entails calling the option pricing formulae in part a) for each value S and each computed option price will be stored in a std::vector<double> object. It will be useful to reuse the above global function that produces a mesh array of double separated by a mesh size h.*

```
********************************Now entering B1.C*********************************

Choose parameter to adjust (0=S) 0

 Enter start: 110

 Enter stop: 120

 Enter step: 1

Mesh input for call

110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,


Call Prices are as follows:
18.5035 19.0501 19.6078 20.1765 20.7566 21.3481 21.951 22.5656 23.192 23.8302 24.4804

Mesh input for put

110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,

Put Prices are as follows:
3.03106 2.86523 2.70985 2.56416 2.42748 2.29919 2.1787 2.06548 1.95904 1.85891 1.76467
```

d) Incorporate this into your above matrix pricer code, so you can input a matrix of option parameters and receive a matrix of Perpetual American option prices.



```
C:\windows\system32\cmd.exe
*******************************Now entering B1.D********************************

Choose parameter to adjust (0=S, 1=K, 2=r, 3=T, 4=sig, 5=b) 4

 Enter start: 0.1

 Enter stop: 0.2

 Enter step: 0.01

*****Now printing matrix of parameters from mesher prior to pricing*********************************

S    K    r    T   sig   b
110, 100, 0.1, 0.5, 0.1, 0.02,

110, 100, 0.1, 0.5, 0.11, 0.02,

110, 100, 0.1, 0.5, 0.12, 0.02,

110, 100, 0.1, 0.5, 0.13, 0.02,

110, 100, 0.1, 0.5, 0.14, 0.02,

110, 100, 0.1, 0.5, 0.15, 0.02,

110, 100, 0.1, 0.5, 0.16, 0.02,
```



```
C:\windows\system32\cmd.exe
110, 100, 0.1, 0.5, 0.17, 0.02,

110, 100, 0.1, 0.5, 0.18, 0.02,

110, 100, 0.1, 0.5, 0.19, 0.02,


Now printing vector of call prices

18.5035, 19.2855, 20.0794, 20.8821, 21.6914, 22.5052, 23.3222, 24.141, 24.9605, 25.7799,

*****Now printing matrix of put parameters from mesher prior to pricing*********************************

S    K    r    T   sig   b
110, 100, 0.1, 0.5, 0.1, 0.02,

110, 100, 0.1, 0.5, 0.11, 0.02,

110, 100, 0.1, 0.5, 0.12, 0.02,

110, 100, 0.1, 0.5, 0.13, 0.02,

110, 100, 0.1, 0.5, 0.14, 0.02,

110, 100, 0.1, 0.5, 0.15, 0.02,

110, 100, 0.1, 0.5, 0.16, 0.02,
```

C:\windows\system32\cmd.exe

```
110, 100, 0.1, 0.5, 0.17, 0.02,

110, 100, 0.1, 0.5, 0.18, 0.02,

110, 100, 0.1, 0.5, 0.19, 0.02,


Now printing vector of put prices

3.03106, 3.75233, 4.50092, 5.27011, 6.05489, 6.85143, 7.65679, 8.46863, 9.28509, 10.1046,
```

2. Class Design

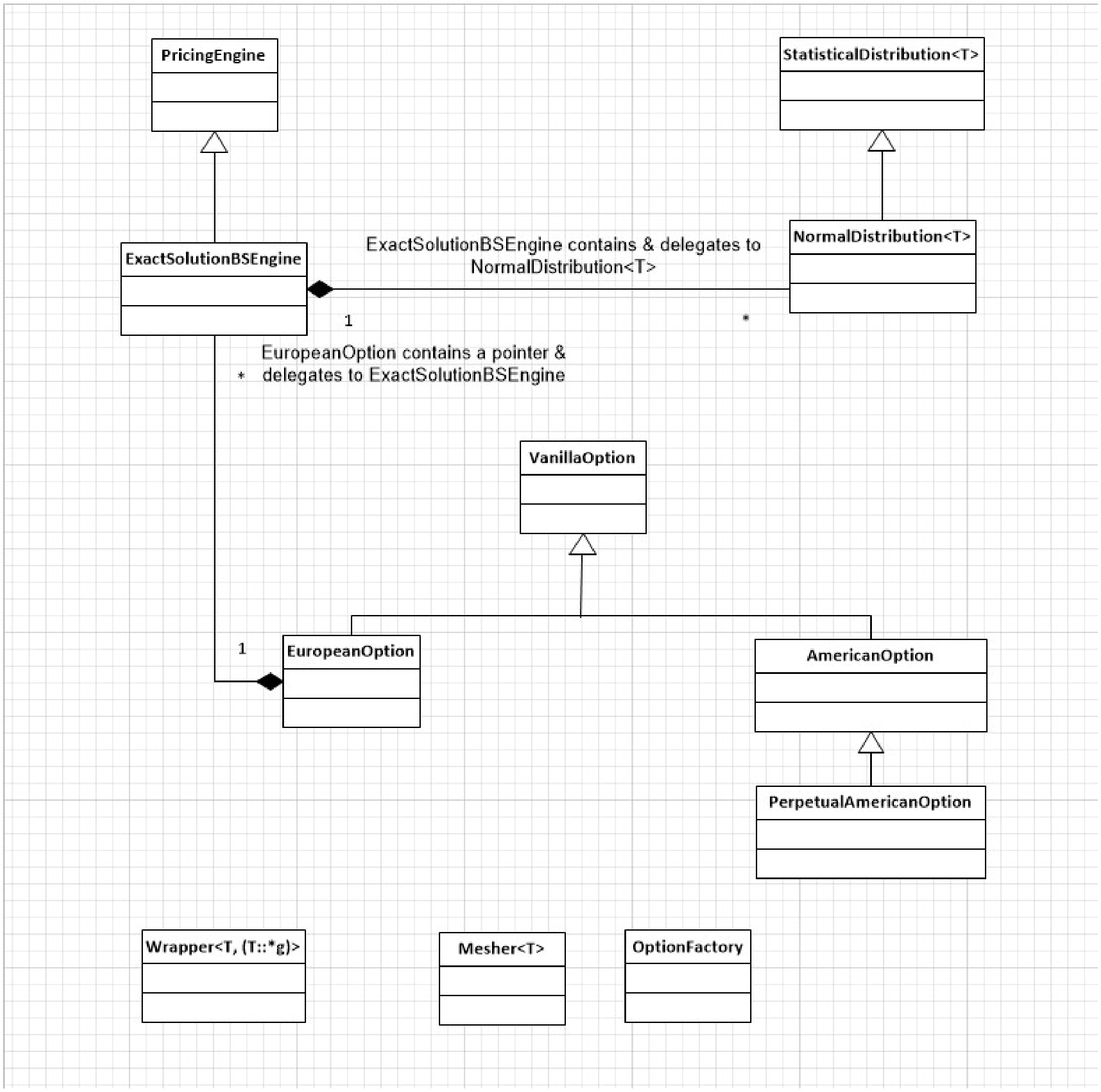


Figure 1 Class Design

My approach towards designing the classes per the TA requirements is shown above.

- *PricingEngine:* In an effort to accommodate other pricing methods (i.e., Monte Carlo, FDM, Lattice, etc.) apart from closed-form Black Scholes, I created this as a base class. Although this class currently doesn't serve much purpose, with future enhancements, I intend to transform it into an API that can, among other things, indicate what input arguments are required by a consumer to run a calculation for any pricing methodology (i.e., Black Scholes, Monte Carlo, etc.) that derives itself from this base class.

- *ExactSolutionBSEngine:* This class contains all functionality to price a vanilla European option, including Greeks. This class does NOT cater to the PerpetualAmericanOption class, since that class has its own particular calculation requirements.

  This class can receive either a vector or matrix of option parameters as an input, and return to the caller a vector of option prices:

  - *CalculateVector():* Returns a vector of option prices as a function of a monotonically increasing range of S values. This method <u>only</u> accommodates a monotonically increasing range of <u>**S values**</u>. It does not accommodate permutations to the other option parameters such as K, T, etc. I had a difficult time deciphering the requirements of the Group A&B write-up associated with this. I chose therefore, to limit this method only to a permutating S parameter, and defer complete permutation flexibility to the *CalculateMesh()* function (described below).

  - *CalculateMesh():*This function can calculate an option price as a function of ANY option parameter permutation. It parses a “matrix” of option parameters (PxN), and returns a (1xN matrix) of option prices, one for each row.

- *NormalDistribution<T>:* This class contains the Probability Density and Cumulative Distribution functions associated with a Normal (aka Gaussian Distribution). This class serves as a wrapper around the boost library - Normal Gaussian Distribution. This class provides flexibility associated with T, which represents ‘class RealType’ in the boost library. You can also specify the mean and standard deviation. If left to the default settings, the normal distribution will initialize with a mean of 0 and standard deviation one is 1, and therefore, would be considered as a *Standard Normal Distribution*.

- *StatisticalDistribution<T>:* This class serves as an abstract base class for NormalDistribution<T>, and potentially future distributions provided in the boost/math/distributions libraries. Per boost documentation, all boost/math/distributions implement the Probability Density and Cumulative Distribution functions. Therefore, these methods have been declared as pure virtual in this class, given that they will vary depending on the distribution being implemented. I did, however, implement Mean and Standard Deviation in this class.

- *EuropeanOption*: This class is derived from base class VanillaOption. It contains methods to price itself, check put-call parity, and mesh. In an effort to decouple an option class from its calculation, this class delegates the Black Scholes calculations to class ExactSolutionBSEngine. As part of a future enhancement, the intent would be to abstract this even further such that *any pricing engine* could be passed through EuropeanOption, or any of the “option” classes.

- *AmericanOption*: Since the exact solution Black Scholes equation does not apply to this class, there’s very little functionality contained in this class. But as mentioned in the EuropeanOption section, future design considerations would call for passing any calculation methodology through this class, such as binomial model.

- *PerpetualAmericanOption*: I created a separate class for the perpetual american option, since this particular option differentiates itself from an American Option such that the Perperpetual American Option does not have an expiry date. Pricing calculations to price itself are also contained in this class.

  I wanted to abstract mesh functions (i.e., CalculateVector(), CalculateMesh(), etc.) contained both in this class and ExactSolutionBSEngine, and put them into the Mesher class. But doing so proved to be too complex for me. Specifically, I already pass a function pointer (see screenshot on following page) in an effort to avoid code duplication, and use std::invoke() to achieve this. But trying to also abstract the *call back class name* as a pointer proved too difficult at the present moment.

```
// Returns a vector of option prices as a function of i) expiry time, ii) volatility, or iii) any of the option pricing parameters.
// Mesher object is passed into the CalculateMesh method, along with a function pointer based on whether the object is a call or put.
// Struct MeshParamData is passed in with all necessary mesh parameters needed to instatiate Mesher object.
vector<double> EuropeanOption::MeshPriceMatrix(const MeshParamData& _mesh_param_data) const
{
    Mesher<double> m_mesh_(this->option_vector_data(), _mesh_param_data._start, _mesh_param_data._end, _mesh_param_data._step, _mesh_param_data._pr

    if (IsCall())  // Call
        return bs->CalculateMesh(m_mesh_, &DevonKaberna::Engine::ExactSolutionBSEngine::CalculateCallPrice);   // Call CalculateMesh function in pr
    else
        return bs->CalculateMesh(m_mesh_, &DevonKaberna::Engine::ExactSolutionBSEngine::CalculatePutPrice);   // Call CalculateMesh function in pri
}
```

Figure 2 Caller function

```
// Calculcates option price as a function of i) expiry time, ii) volatility, or iii) any of the option pricing parameters.
// Mesher object provides a "matrix" (i.e., vector of vectors) of option parameters to this function.
// CallPrice/PutPrice/CallDelta/PutDelta/Gamma passed in as a function pointer in argument vcm.
vector<double> ExactSolutionBSEngine::CalculateMesh(const Mesher<double>& mesh, MeshModel mm) const
{
    std::size_t numberOfRows = mesh.MeshVector().size();  // Size is driven from user input of mesh size
    int start = 0;
    int row_arr = 0;

    vector<vector<double> > _OptionParamMatrix = mesh.MeshParamMatrix();  // Mesh object returns a "matrix" (i.e., vector of vectors) of

    std::vector<double> result(numberOfRows, start);  // Will store vector of option prices
    vector<vector<double> >::const_iterator vvi_iterator;   // STL iterator that iterates through each row of the matrix - each row cont

    for (vvi_iterator = _OptionParamMatrix.begin(); vvi_iterator != _OptionParamMatrix.end(); ++vvi_iterator)  // Loop through each row
    {
        result[row_arr] = std::invoke(mm, this, *vvi_iterator);  // CallPrice/PutPrice/CallDelta/PutDelta/Gamma functions is passed in v
        row_arr++;
    }
    return result;   // Return vector of option prices
}
```

Figure 3 CalculateMesh() function in ExactSolutionBSEngine

- *Mesher<T>*: This class provides functionality required by this assignment for creating meshes.
    - *MeshVector():* Outputs a mesh array of double separated by mesh size h. A consumer class such as EuropeanOption would then pass this vector onto *ExactSolutionBSEngine* (as an example), which would in turn, provide back a vector of option prices.

```
*********************************Now entering B1.C*********************************

Choose parameter to adjust (0-5) 0

 Enter start: 110

 Enter stop: 120

 Enter step: 1

Mesh input for call

110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
```

- *MeshParamMatrix()*: Outputs a matrix (vector of vectors) of option parameters. End user can choose any parameter to permutate, such as what's shown on the following page. A consumer class such as EuropeanOption would then pass this matrix of option parameters onto *ExactSolutionBSEngine* (as an example), which would in turn, provide back a vector of option prices (1xN).

```
******************************A2.C************************************


Choose parameter to adjust (0=S, 1=K, 2=r, 3=T, 4=sig, 5=b) 2
Enter start: 0.1
Enter stop: 0.2
Enter step: 0.01



*****Now printing matrix of parameters from mesher prior to pricing*********************************


S    K    r     T    sig   b
105, 100, 0.1, 0.5, 0.36, 0,

105, 100, 0.11, 0.5, 0.36, 0,

105, 100, 0.12, 0.5, 0.36, 0,

105, 100, 0.13, 0.5, 0.36, 0,

105, 100, 0.14, 0.5, 0.36, 0,

105, 100, 0.15, 0.5, 0.36, 0,

105, 100, 0.16, 0.5, 0.36, 0,
```

- *OptionFactory*: Although not required as part of this project, I created this class more out of curiosity, and wanted to experiment with the Factory Method design pattern. There's not much to it currently, other than to defer instantiation of any class derived from VanillaOption until run-time.

- _Wrapper<T, (T::*g)>:_ This was also created out of pure curiosity and experimentation. I use it in ExactSolutionBSEngine when retrieving cdf and pdf from the boost libraries.