Midterm Examination
FE 621: Computational Methods in Finance
Instructor: Ionut Florescu
3/31/2019

Rukmal Weerawarana
rweerawa@stevens.edu | 104-307-27
Department of Financial Engineering
Stevens Institute of Technology

Overview

This is my solution manuscript for the FE 621 Midterm Examination.

Unless otherwise stated, all solutions in the manuscript assume that the number of days in a year to be 365.

The content of this manuscript is divided into four sections; the first addresses Problem 1 (Numerical Integration). The second section addresses Problem 2, pricing options with Trinomial Additive Trees. The third section addresses Problem 3, and computes various ranges of possible option prices. Finally, the fourth section answers Problem 4, and discretizes a partial differential equation describing a stochastic option pricing model.

See Appendix A for specific question implementations, and the project GitHub repository[1] *for full source code of the* `fe621` *Python package.*

1. Weerawarana 2019

Contents

1 Question 1

Problem 1: Numerical Integration

(a) Numerically compute the integral $\int_0^2 e^{x^2}\,dx$ using the trapezoid method with 100 steps.

(b) Numerically compute the integral $\int_0^2 e^{x^2}\,dx$ using Simpson's quadrature rule, with 100 steps.

(c) Please compare the two results obtained. Comment.

1.1 Part (a) and (b)

To answer this question, I am using implementations of Simpson's and Trapezoidal Quadrature rules from the `fe621` package. The complete methodology for these two quadrature rules are outlined in Homework 1.[2] For clarity, the quadrature rule approximation equations for both Simpson's rule and the Trapezoidal rule, $S_N(f)$ and $T_N(f)$, respectively, are reproduced below:

$$\text{Let data} = \boldsymbol{x}$$
$$\text{Let } i^{\text{th}} \text{ element of } \boldsymbol{x} = x_i$$
$$\boldsymbol{x}_{\text{mid}} = \left(\frac{x_{i-1} + x_i}{2}\right)$$

$$\Rightarrow S_N(f) \approx \frac{h}{6}(2f(\boldsymbol{x}) - (f(x_0) + f(x_N)) + 4f(\boldsymbol{x}_{\text{mid}}))$$
$$\Rightarrow T_N(f) = hf(\boldsymbol{x}) - \frac{h}{2}(f(x_0) + f(x_N))$$

To compute the integral in the question, implementations from the `fe621` package was used for both of the quadrature rules. The source code for both Simpson's and Trapezoidal quadrature rules are reproduced in Appendix B.1 and Appendix B.2, respectively. Furthermore, the source code for the computation and table output of the solutions to *Part (a)* and *Part (b)* is reproduced in Appendix A.1.

2. Weerawarana 2019

1.2 Part (c)

Quadrature Rule	Estimated Area
Trapezoidal Rule	16.460054216142815
Simpson's Rule	16.452628043283323

Table 1: Numerical approximations for the integral $\int_0^2 e^{x^2}\,dx$ with the Simpson's and Trapezoidal quadrature rules.

The approximations for the integral $\int_0^2 e^{x^2}\,dx$ under both Simpson's and Trapezoidal quadrature rules are reproduced in Table 1.

It can be seen that the approximated integrals under both of the quadrature rules are extremely close, differing by < 0.01. This indicates convergence under the two rules, and may be attributed to the large number of steps, $N = 100$, for the small interval 0 to 2.

Utilizing the Wolfram—Alpha computation platform[3], I found that the true estimated value of the integral ≈ 16.4526. Analyzing the computed approximations through the lens of this solution, it is clear that the Simpson's Quadrature Rule assumption is significantly closer to the Wolfram—Alpha approximation, compared to the Trapezoidal Quadrature Rule.

A potential explanation of this may be the interpolating behavior of Simpson's Quadrature Rule. The function e^{x^2} is better approximated quadratically than linearly in small intervals between 0 and 2. Thus, the quadratic interpolating behavior of Simpson's Rule is a better approximation heuristic for the function with 100 steps. Despite this shortcoming, the Trapezoidal quadrature rule approximation is also extremely close to the Wolfram—Alpha computed area. Theoretically, both approximations will converge with the true value as the number of steps, $N \to \infty$.

3. Wolfram—Alpha 2019

2 Question 2

Problem 2: Option Pricing using a Trinomial Tree
Construct a Trinomial tree to price an American put option. To this end, start with the following given parameters: $S_0 = 100$, $K = 120$, maturity $T = 8$ months, $r = 0$, $\delta = 0$, volatility $\sigma = 30\%$, and time steps $N = 200$.

(a) What is the best choice for Δx to obtain the best order of convergence? Calculate Δx.

(b) Calculate the American Put option price using the tree.

(c) Estimate Gamma of the American Put at time $t = 0$.

For this question, I will be using the `AdditiveTree` Trinomial tree model, outlined in Homework 2.[4] This builds on the `GeneralTree` generalized tree implementation, also discussed in Homework 2. The source code for these two modules from the `fe621` package are reproduced in Appendix B.3 and Appendix B.4, respectively.

2.1 Part (a)

In order to guarantee a convergent trinomial tree, the following condition was imposed on the jump of each step on the tree, Δx[5]:

$$\Delta x \geq \sigma\sqrt{3\Delta t}$$

Utilizing the option parameters specified in the question, the lower bound on the jump, Δx can be computed:

$$\Delta t = T/N = \frac{8}{12} \cdot \frac{1}{200}$$
$$\sigma = 0.3$$
$$\Rightarrow \Delta x = \sigma\sqrt{3\Delta t} = 0.3\sqrt{3 \cdot \frac{8}{12} \cdot \frac{1}{200}} = 0.3\sqrt{\frac{3}{200} \cdot \frac{2}{3}} = 0.3\sqrt{0.01}$$
$$\therefore \Delta x = 0.03$$

2.2 Part (b)

Utilizing the jump size computed above, an Additive Trinomial Tree was used to compute the price of an American Put option with the parameters stated in the question:

American Put Estimated Price
23.5330268874819

Table 2: Price of an American Put option, computed with a Trinomial Additive Tree.

4. Weerawarana 2019
5. Florescu 2019

2.3 Part (c)

To compute the Gamma, Γ, of the American Put option at time $t = 0$, the Central Finite Difference Method was used. Γ is defined as the second derivative of the option value, V, with respect to the underlying asset price, S:

$$\Gamma = \frac{\partial^2 V}{\partial S^2}$$

The second-order central finite difference method for an arbitrary three-times differentiable function, f, in an interval around the point a is[6]:

$$\text{Let } h > 0$$
$$\Rightarrow f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2} + O(h^2)$$

This can be applied to the tree-pricing methodology utilized to compute the initial price for the American Put option. By setting $f(a)$ to be equal to the trinomial-tree computed price of the option, given initial stock price a, an approximation for the Gamma, Γ of the option can be computed.

To accomplish this, the central finite difference method from the `fe621` package (outlined in Homework 1) was utilized. The source code for this approximation method is reproduced in Appendix B.5. As with the previous question, the source code for this computation is reproduced in Appendix A.2.

American Put Gamma Approximation
0.013575511874115875

Table 3: Estimated Gamma, Γ of the American Put at time $t = 0$.

6. Stefanica 2011

3 Question 3

Problem 3: Option Price Range
Assume a stock is at \$23.35. You look at the market to a European Call option with strike \$22.50 and maturity of 8 weeks. Assume $r = 0.01$. The listed best bid is \$3.20, and the best ask is \$3.80. Use the code you turned in the assignments to answer the following questions:

(a) Calculate an interval of possible values for the European Put.

(b) Calculate an interval of possible values for the American Call.

Note: For this question, I am assuming that the instructor wants us to compute an interval of possible option prices, utilizing the best bid and ask values as upper and lower bounds on the price of the original call option. I am making this assumption due to the fact that the instructor has not provided corresponding volumes for the best bid and ask offer values, thus making a volume-weighted average price computation impossible.

Under this assumption, I computed upper and lower bounds on the implied volatility of the option, using the best bid and ask price of the call option, in conjunction with a Bisection-method optimization on the Black-Scholes option pricing formula. The corresponding `fe621` package source code for the Black-Scholes Call Price and the Bisection Optimization Method are reproduced in Appendix B.6, and Appendix B.7, respectively.

The source code for this computation is reproduced in Appendix A.3. Utilizing the assumption outlined above, upper and lower bounds on the implied volatility were computed and are reproduced in Table 4.

Initial Price	Computed Implied Volatility
Bid	0.7670173645019531
Ask	0.9381752014160156

Table 4: Upper and lower bounds on the implied volatility of the European Call Option, using the Best Bid as the Lower Bound price, and Best Ask as the Upper Bound price for the Bisection optimizer.

3.1 Part (a)

The upper and lower bounds on the implied volatility (see Table 4) were used to compute a range of possible option prices for a European Put option, using the Black-Scholes formula.

The `fe621` package source code for computing the Black-Scholes European Put option price is reproduced in Appendix B.8. The source code for this computation is reproduced in Appendix A.3. The upper and lower bounds on the price of a European Put option are reproduced in Table 5.

	Black-Scholes EU Put
Lower Bound	2.3155038666427252
Upper Bound	2.9154999137462685

Table 5: Upper and lower bounds on the price of a European Put option.

3.2 Part (b)

Similar to Part (a), the upper and lower bounds on the implied volatility (see Table 4) were used to compute a range of possible prices for an American Call option, using the Trigeorgis Binomial Pricing Tree.

The `fe621` package source code for the Trigeorgis tree, and the `GeneralTree` generalized tree on which it is based is reproduced in Appendix B.9, and Appendix B.4, respectively. The source code for this computation is reproduced in Appendix A.3. The upper and lower bounds on the price of an American Call option are reproduced in Table 6.

	Trigeorgis Tree American Call
Lower Bound	3.1997826473835826
Upper Bound	3.8029268355407826

Table 6: Upper and lower bounds on the price of an American Call option.

4 Question 4

Problem 4
We know that an option price under a certain stochastic model satisfies the following PDE:

$$\frac{\partial V}{\partial t} + 2\cos(S)\frac{\partial V}{\partial S} + 0.2S^{\frac{3}{2}}\frac{\partial^2 V}{\partial S^2} - rV = 0.$$

Assume you have an equidistant grid with points of the form $(i, j) = (i\Delta t, j\Delta x)$, where $i \in \{1, 2, \ldots, N\}$, and $j \in \{-N_S, N_S\}$. Let $V_{i,j} = (i\Delta t, j\Delta x)$. Discretize the derivatives and give finite difference equation for an Explicit scheme. Use the notation introduced above.

$$\frac{\partial V}{\partial t} + 2\cos(S)\frac{\partial V}{\partial S} + 0.2S^{\frac{3}{2}}\frac{\partial^2 V}{\partial S^2} - rV = 0$$

$$\text{Let } \ln(S) = x$$
$$\Rightarrow S = e^x$$
$$\Rightarrow \frac{dS}{dx} = e^x$$
$$\therefore dS = e^x dx = S dx$$

We can use this substitution to reorganize the initial PDE:

$$-\frac{\partial V}{\partial t} = \frac{2\cos(S)}{S}\frac{\partial V}{\partial x} + \frac{0.2}{\sqrt{S}}\frac{\partial^2 V}{\partial x^2} - rV$$

We can discretize the stock process, S, and the value of the option, V.

Utilizing finite difference methods:

$$\Rightarrow \frac{\partial V_{i,j}}{\partial t} = \frac{V_{i+1,j} - V_{i,j}}{\Delta t}$$
$$\Rightarrow \frac{\partial V_{i,j}}{\partial x} = \frac{V_{i+1,j+1} - V_{i+1,j-1}}{2\Delta x}$$
$$\Rightarrow \frac{\partial^2 V_{i,j}}{\partial x^2} = \frac{V_{i+1,j+1} - 2V_{i+1,j} + V_{i+1,j-1}}{\Delta x^2}$$

Substituting this in the reorganized discretized PDE:

$$-\frac{\partial V}{\partial t} = \frac{2\cos(S)}{S}\frac{\partial V}{\partial x} + \frac{0.2}{\sqrt{S}}\frac{\partial^2 V}{\partial x^2} - rV$$

$$\Rightarrow -\frac{V_{i+1,j} - V_{i,j}}{\Delta t} = \frac{2\cos(S_{i,j})}{S_{i,j}}\frac{V_{i+1,j+1} - V_{i+1,j-1}}{2\Delta x} + \frac{0.2}{\sqrt{S_{i,j}}}\frac{V_{i+1,j+1} - 2V_{i+1,j} + V_{i+1,j-1}}{\Delta x^2} - rV_{i+1,j}$$

We can now reorganize the equation, and solve for $V_{i,j}$

$$-\frac{V_{i+1,j} - V_{i,j}}{\Delta t} = \frac{2\cos(S_{i,j})}{S_{i,j}} \frac{V_{i+1,j+1} - V_{i+1,j-1}}{2\Delta x} + \frac{0.2}{\sqrt{S_{i,j}}} \frac{V_{i+1,j+1} - 2V_{i+1,j} + V_{i+1,j-1}}{\Delta x^2} - rV_{i+1,j}$$

$$\Rightarrow V_{i,j} = \Delta t \left(\frac{2\cos(S_{i,j})}{S_{i,j}} \frac{V_{i+1,j+1} - V_{i+1,j-1}}{2\Delta x} + \frac{0.2}{\sqrt{S_{i,j}}} \frac{V_{i+1,j+1} - 2V_{i+1,j} + V_{i+1,j-1}}{\Delta x^2} - rV_{i+1,j} \right) + V_{i+1,j}$$

Reorganizing the equation to isolate p_u, p_m, and p_d:

$$\Rightarrow V_{i,j} = V_{i+1,j+1} \left(\frac{\Delta t 0.2}{\sqrt{S_{i,j}}\Delta x^2} + \frac{2\Delta t \cos(S_{i,j})}{2\Delta x S_{i,j}} \right) + V_{i+1,j} \left(1 - r\Delta t - \frac{2\Delta t 0.2}{\sqrt{S_{i,j}}\Delta x^2} \right)$$
$$+ V_{i+1,j-1} \left(\frac{\Delta t 0.2}{\sqrt{S_{i,j}}\Delta x^2} - \frac{2\Delta t \cos(S_{i,j})}{2\Delta x S_{i,j}} \right)$$

From the equation above, we can isolate the jump probabilities of the tree:

$$p_u = \frac{\Delta t 0.2}{\sqrt{S_{i,j}}\Delta x^2} + \frac{2\Delta t \cos(S_{i,j})}{2\Delta x S_{i,j}} = \frac{\Delta t}{\sqrt{S_{i,j}}\Delta x} \left(\frac{0.2}{\Delta x} + \frac{\cos(S_{i,j})}{\sqrt{S_{i,j}}} \right)$$

$$p_m = 1 - r\Delta t - \frac{2\Delta t 0.2}{\sqrt{S_{i,j}}\Delta x^2} = 1 - \Delta t \left(r + \frac{0.4}{\sqrt{S_{i,j}}\Delta x^2} \right)$$

$$p_d = \frac{\Delta t 0.2}{\sqrt{S_{i,j}}\Delta x^2} - \frac{2\Delta t \cos(S_{i,j})}{2\Delta x S_{i,j}} = \frac{\Delta t}{\sqrt{S_{i,j}}\Delta x} \left(\frac{0.2}{\Delta x} - \frac{\cos(S_{i,j})}{\sqrt{S_{i,j}}} \right)$$

$$\therefore V_{i,j} = p_u \cdot V_{i+1,j+1} + p_m \cdot V_{i+1,j} + p_d \cdot V_{i+1,j-1}$$

References

Florescu, Ionut. 2019. "6.5 Trinomial tree method and other considerations." Chap. 6 - Tree M in *Computational Methods in Finance,* 136–139. Hoboken, NJ.

Stefanica, Dan. 2011. *A Primer for the Mathematics of Financial Engineering.* First Edit. 89–96. New York, NY: FE Press. ISBN: 0-9797576-2-2.

Weerawarana, Rukmal. 2019. *FE 621 Homework - rukmal - GitHub.* Accessed February 20, 2019. `https://github.com/rukmal/FE-621-Homework`.

Wolfram—Alpha. 2019. *Wolfram—Alpha Computational Intelligence.* Accessed March 31, 2019. `https://www.wolframalpha.com/`.

A Solution Source Code

A.1 Question 1 Solution

```
from context import fe621

import numpy as np
import pandas as pd


# Defining objective function
def f(x: float) -> float:
    return np.exp(np.power(x, 2))

# Configuration variables for the quadrature rule approximations
start = 0
stop = 2
steps = 100

# Building DataFrame to store results
q1_res = pd.Series()

# Computing integral with the Trapezoidal rule
# Part (a)
q1_res.at['Trapezoidal Rule'] = fe621.numerical_integration.trapezoidalRule(
    f=f, N=steps, start=start, stop=stop
)

# Computing integral with Simpson's Rule
# Part (b)
q1_res.at['Simpson\'s Rule'] = fe621.numerical_integration.simpsonsRule(
    f=f, N=steps, start=start, stop=stop
)

# Updating Index label
q1_res.index.name = 'Quadrature Rule'

# Casting to DataFrame, saving to CSV
pd.DataFrame({'Estimated Area': q1_res}).to_csv(
    'Midterm Exam/bin/question_1.csv'
)
```

question_solutions/question_1.py

A.2 Question 2 Solution

```
from context import fe621

import pandas as pd


# Configuring option data for tree construction
current = 100
strike = 120
ttm = 8 / 12  # Fraction of a year (assuming 365 days)
rf = 0
volatility = .3
N = 200
```

```python


# Constructing Trinomial tree to price American Put option
tree = fe621.tree_pricing.trinomial.AdditiveTree(
    current=current, strike=strike, ttm=ttm, rf=rf, volatility=volatility,
    opt_type='P', opt_style='A', steps=N
)

# Part (b)
pd.DataFrame({'American Put Estimated Price': [tree.getInstrumentValue()]})\
    .to_csv('Midterm Exam/bin/question_2_price.csv', index=False)


# Part (c)

# Defining objective function for second derivative (Gamma) computation
def f(x: float) -> float:
    # Building trinomial tree for the given strike price, `x`, keeping
    # all other parameters the same
    tree = fe621.tree_pricing.trinomial.AdditiveTree(
        current=x, strike=strike, ttm=ttm, rf=rf, volatility=volatility,
        opt_type='P', opt_style='A', steps=N
    )
    return tree.getInstrumentValue()

# Computing estimate for Gamma
gamma = fe621.numerical_differentiation.secondDerivative(f=f, x=current, h=3)

# Writing to CSV
pd.DataFrame({'American Put Gamma Approximation': [gamma]}).to_csv(
    'Midterm Exam/bin/question_2_gamma.csv', index=False
)
```

question_solutions/question_2.py

A.3 Question 3 Solution

```python
from context import fe621

import numpy as np
import pandas as pd


# Option metadata
current = 23.35
strike = 22.5
ttm = (8 * 7) / 365  # In years, assuming 365 days per year
rf = 0.01

# Defining bid and ask prices from question
best_bid = 3.2
best_ask = 3.8

# Defining range of values
a = 0
b = 2

# Computing possible implied volatilities using the bisection solver, and the
# Black-Scholes call option formula
```

```python

implied_vol = pd.Series()

# Defining function to be optimized (bid)
def f_bid(x: float) -> float:
    return best_bid - fe621.black_scholes.call(
        current=current, volatility=x, ttm=ttm, strike=strike, rf=rf
    )

# Defining function to be optimized (ask)
def f_ask(x: float) -> float:
    return best_ask - fe621.black_scholes.call(
        current=current, volatility=x, ttm=ttm, strike=strike, rf=rf
    )


implied_vol.at['Bid'] = fe621.optimization.bisectionSolver(
    f=f_bid, a=a, b=b, tol=1e-4
)

implied_vol.at['Ask'] = fe621.optimization.bisectionSolver(
    f=f_ask, a=a, b=b, tol=1e-4
)

# Labeling Index
implied_vol.index.name = 'Initial Price'

# Saving implied volatility range to CSV
pd.DataFrame({'Computed Implied Volatility': implied_vol}).to_csv(
    'Midterm Exam/bin/question_3_imp_vol.csv'
)


# European Put Option Range - Part (a)

eu_put_range = pd.Series()

# Lower Bound
eu_put_range.at['Lower Bound'] = fe621.black_scholes.put(
    current=current, volatility=implied_vol['Bid'], ttm=ttm,
    strike=strike, rf=rf
)

# Upper Bound
eu_put_range.at['Upper Bound'] = fe621.black_scholes.put(
    current=current, volatility=implied_vol['Ask'], ttm=ttm,
    strike=strike, rf=rf
)

# Writing to CSV
pd.DataFrame({'Black-Scholes EU Put': eu_put_range}).to_csv(
    'Midterm Exam/bin/question_3_eu_put_prices.csv'
)


# American Call Option Range - Part (b)

a_call_range = pd.Series()

N = 200  # Steps for the tree

```

```python
# Lower Bound
a_call_range.at['Lower Bound'] = fe621.tree_pricing.binomial.Trigeorgis(
    current=current, strike=strike, ttm=ttm, rf=rf,
    volatility=implied_vol['Bid'], opt_type='C', opt_style='A', steps=N
).getInstrumentValue()

# Upper Bound
a_call_range.at['Upper Bound'] = fe621.tree_pricing.binomial.Trigeorgis(
    current=current, strike=strike, ttm=ttm, rf=rf,
    volatility=implied_vol['Ask'], opt_type='C', opt_style='A', steps=N
).getInstrumentValue()

pd.DataFrame({'Trigeorgis Tree American Call': a_call_range}).to_csv(
    'Midterm Exam/bin/question_3_american_call_prices.csv'
)
```

question_solutions/question_3.py

B fe621 Package Code

B.1 Simpson's Quadrature Rule

```python
from typing import Callable
import numpy as np


def simpsonsRule(f: Callable, N: float, start: float=-1e6,
                 stop: float=1e6) -> float:
    """Function to approximate the numeric integral of a function, f, using
    Simpson's rule.

    Arguments:
        f {Callable} -- Function for which the integral is to be estimated.
        N {float} -- Number of nodes to consider.

    Keyword Arguments:
        start {float} -- Starting point (default: {-1e6}).
        stop {float} -- Stopping point (default: {1e6}).

    Returns:
        float -- Approximation of the area under the function.
    """

    # Building values for approximation, and getting step size
    x, h = np.linspace(start=start, stop=stop, num=N, retstep=True)

    # Computing midpoints
    x_mid = np.array([(x[i - 1] + x[i]) / 2 for i in range(1, N)])

    # Estimating using Simpson's rule
    area = np.sum(2 * f(x)) - (f(start) + f(stop)) + (4 * np.sum(f(x_mid)))

    # Scaling area
    area *= (h / 6)

    return area
```

../fe621/numerical_integration/simpsons.py

B.2 Trapezoidal Quadrature Rule

```python
from typing import Callable
import numpy as np


def trapezoidalRule(f: Callable, N: float, start: float=-1e6,
                    stop: float=1e6) -> float:
    """Function to approximate the numeric integral of a function, f, using
    the Trapezoidal rule.

    Arguments:
        f {Callable} -- Function who's integral is to be estimated.
        N {int} -- Number of nodes to consider.

    Keyword Arguments:
        start {float} -- Starting point (default: {-1e6}).
```

```
        stop {float} -- Stopping point (default: {1e6}).

    Returns:
        float -- Approximation of the area under the function.
    """

    # Building values for approximation, and getting step size
    x, h = np.linspace(start=start, stop=stop, num=N, retstep=True)

    # Estimating area using trapezoidal rule, return
    return np.sum((h * f(x))) - ((h / 2) * (f(start) + f(stop)))
```

../fe621/numerical_integration/trapezoidal.py

B.3 Additive Tree Trinomial Tree

```
from ..general_tree import GeneralTree

import numpy as np


class TrinomialAdditivePriceTree(GeneralTree):
    """Trinomial tree option pricing with an additive tree. This method is
    outlined in https://en.wikipedia.org/wiki/Trinomial_tree.

    Implemented with the `GeneralTree` abstract class.
    """

    def __init__(self, current: float, strike: float, ttm: float, rf: float,
                 volatility: float, opt_type: str, opt_style: str,
                 dividend: float=0, steps: int=1):
        """Initialization method for the `TrinomialAdditivePriceTree` class.

        Arguments:
            current {float} -- Current asset price.
            strike {float} -- Strike price of the option.
            ttm {float} -- Time to maturity of the option (in years).
            rf {float} -- Risk-free rate (annualized).
            volatility {float} -- Volatility of the underlying asset price.
            opt_type {str} -- Option type, 'C' for Call, 'P' for Put.
            opt_style {str} -- Option style, 'E' for European, 'A' for American.

        Keyword Arguments:
            dividend {float} -- Cont. div. yield (annualized) (default: {0}).
            steps {int} -- Number of steps to construct (default: {1}).
        """

        # Ensuring valid option type and style
        if opt_type not in ['C', 'P'] or opt_style not in ['A', 'E']:
            raise ValueError('`opt_type` must be \'C\' or \'P\' and `opt_style`\
                must be \'A\' or \'E\'.')

        # Setting class variables
        self.opt_type = opt_type
        self.opt_style = opt_style
        self.rf = rf
        self.volatility = volatility
        self.strike = strike
        self.nu = (rf - dividend) - (0.5 * np.power(volatility, 2))
```

```python

        # Computing deltaT
        deltaT = ttm / steps

        # Setting upward and downward jumps for children
        # Setting equal to the convergence condition for now
        self.deltaXU = volatility * np.sqrt(3 * deltaT)
        self.deltaXD = -1 * self.deltaXU

        # Computing upward, middle and downward jumps (additive)
        self.jumpU = 0.5 * ((((np.power(volatility, 2) * deltaT) + (np.power(
            self.nu, 2) * np.power(deltaT, 2))) / np.power(self.deltaXU, 2)) +\
            (self.nu * deltaT / self.deltaXU))
        self.jumpD = 0.5 * ((((np.power(volatility, 2) * deltaT) + (np.power(
            self.nu, 2) * np.power(deltaT, 2))) / np.power(self.deltaXU, 2)) -\
            (self.nu * deltaT / self.deltaXU))
        self.jumpM = 1 - self.jumpU - self.jumpD

        # Discount factor for each jump
        self.disc = np.exp(-1 * rf * deltaT)

        # Initializing GeneralTree, with root set to log price for Additive tree
        super().__init__(price_tree_root=np.log(current), steps=steps)

    def childrenPrice(self) -> np.array:
        """Function to compute the price of children nodes, given the price at
        the current node.

        Returns:
            np.array -- Array of length 3 corresponding to [up_child_price,
                        mid_child_price, down_child_price].
        """

        # Computing upward and downward child additive values (mid is same)
        up_child_price = self._current_val + self.deltaXU
        down_child_price = self._current_val + self.deltaXD

        return np.array([up_child_price, self._current_val, down_child_price])

    def instrumentValueAtNode(self) -> float:
        """Function to compute the instrument value at the given node.

        Intelligently adapts to the specified option style (`self.opt_style`)
        and type (`self.opt_type`) to work with both European options, and the
        path-dependent American option style.

        Returns:
            float -- Value of the option at the given node.
        """

        # Value implied by children
        child_implied_value = self.disc * ((self.jumpU * self._child_values[0])\
            + (self.jumpM * self._child_values[1])\
            + (self.jumpD * self._child_values[2]))

        # American option special case
        # NOTE: It is path dependent, so evaluate option value at current node
        #       and return if higher than `child_implied_value`
        if self.opt_style == 'A':
            # Computing value of option if exercied at current node
            # NOTE: Using `valueFromLastCol` here as it is the same computation;
```

```
            #       casting current node value to array and passing thru
            option_value = self.valueFromLastCol(last_col=np.array([
                self._current_val]))[0]

            # If value is higher than `child_implied_value`, exercise now
            if option_value > child_implied_value:
                return option_value

        return child_implied_value

    def valueFromLastCol(self, last_col: np.array) -> np.array:
        """Function to compute the option value of the last column (i.e. last
        row of leaf nodes) of the price tree.

        Arguments:
            last_col {np.array} -- Last column of the price tree.

        Returns:
            np.array -- Value of the option corresponding to the input prices.
        """

        # Call option (same for European and American)
        if self.opt_type == 'C':
            # Computing non-floored call option value
            non_floor_val = np.exp(last_col) - self.strike

        # Put option (same for European and American)
        if self.opt_type == 'P':
            # Computing non-floored put option value
            non_floor_val = self.strike - np.exp(last_col)

            # Replacing values equal to (self.strike - 1) with 0. This is to
            # adjust for the fact that zero nodes would have this value in
            # the tree.
            # This is a special case adjustment that must be made to
            # computation. This is purely for clarity.
            non_floor_val = np.where(non_floor_val == (self.strike - 1), 0,
                                     non_floor_val)

        # Floor to 0 and return
        return np.where(non_floor_val > 0, non_floor_val, 0)

    def getPriceTree(self) -> np.array:
        """Function to get the price tree. Overrides superclass function of the
        same name to return the real price tree as opposed to to the
        log-price tree.

        Returns:
            np.array -- Constructed price tree.
        """

        # Getting log price tree from superclass method
        log_price_tree = super().getPriceTree()
        # Computing real price tree
        price_tree_unadj = np.exp(log_price_tree)

        # Replacing all instances of value '1' with zero, as it would have
        # previously been a zero node before exponentiation
        return np.where(price_tree_unadj == 1, 0, price_tree_unadj)

    def computeOtherStylePrice(self, opt_style: str) -> float:
```

```
        """Function to compute the 'other' option style (i.e. American or
        European), given the constructed price tree. Note that this modifies the
        current instance `self.opt_type` and `self.value_tree` variables.

        This is possible for this specific implementation, as the same
        constructed price tree is utilized for both option value calculations.

        This function calls internal functions from abstract class `GeneralTree`
        to recompute the option value, given a change in style.

        Arguments:
            opt_style {str} -- Option style, 'E' for European, 'A' for American.

        Returns:
            float -- Option value of the desired style.
        """

        # Ensuring valid option style
        if opt_style not in ['A', 'E']:
            raise ValueError('`opt_style` must be \'A\' or \'E\'.')

        # If desired option style matches current style, return price
        if opt_style == self.opt_style:
            return self.getInstrumentValue()

        # Setting new option style
        self.opt_style = opt_style

        # Rebuilding value tree (calling superclass internal function here)
        self.value_tree = self._constructValueTree()

        return self.getInstrumentValue()
```

../fe621/tree_pricing/trinomial/trinomial_price.py

B.4 `GeneralTree` Generalized Tree

```
from abc import ABC, abstractmethod
from scipy import sparse
import numpy as np


class GeneralTree(ABC):
    """Abstract class enabling efficient implementation of any generalized
    binomial or trinomial tree pricing or analysis algorithm.

    This implementation of a general tree follows the algorithm outlined in
    my notes. See: http://bit.ly/2WjfkJu.

    This class may be inherited by a subclass that implements a specific pricing
    algorithm, while this abstract class handles tree construction, reverse
    traversal and price computation, given implementations of functions for
    computing price of children from a current node, the value of the last
    column (i.e. bottow row of leaf nodes) of a constructed price tree before
    recombination, and the value of a node given the children values.

    This generalized tree computation methodology allows this class to be used
    as a base for any arbitrary tree pricing or analysis tool, including
    multiplicative and additive trees. Tree values are strategically exposed at
```

```
    runtime when building and traversing the tree for added flexibility. Details
    of specific exposed runtime variables are discussed further in the
    specific function docstrings.

    Requires that `GeneralTree.childrenPrice`,
    `GeneralTree.instrumentValueAtNode`, and `GeneralTree.valueFromLastCol`
    be overridden and implemented. Specific requirements for these abstract
    methods are outlined in their respective docstrings below.

    Raises:
        NotImplementedError -- Raised when not implemented.
    """

    # Need to add documentation to this; explain persistent variables, etc.
    def __init__(self, price_tree_root: float, steps: int=1,
                 build_price_tree: bool=True, build_value_tree: bool=True):
        """Initialization method for the abstract `GeneralTree` class.

        Constructs both the price and value tree, and isolates the instrument
        price from the computed value tree.

        Provides flags to suppress the construction of the price tree and the
        value tree for flexibility. This option allows for an externally
        constructed price or value tree to be used by setting it to the
        `price_tree` and `value_tree` class variables respectively.

        Arguments:
            price_tree_root {float} -- Value of the root of the price tree.

        Keyword Arguments:
            steps {int} -- Number of steps to construct (default: {1}).
            build_price_tree {bool} -- Price tree flag (default: {True}).
            build_value_tree {bool} -- Value tree flag (default: {True}).

        Raises:
            ValueError -- Raised when the number of steps is invalid.
            RuntimeError -- Raised when invalid sequence is attempted. That is,
                            if the value tree is attempted to be constructed
                            without a price tree being constructed first.
        """

        self.price_tree_root = price_tree_root
        self.steps = steps

        # Check steps
        if self.steps < 1:
            raise ValueError('Must have a step size of at least 1.')

        # Computing shape of matrix representing the tree
        self.nrow = (2 * self.steps) + 1
        self.ncolumn = self.steps + 1

        # Construct the price tree
        if build_price_tree:
            self.price_tree = self._constructPriceTree()

        # Construct value tree (check that price tree is constructed first)
        if build_value_tree:
            try:
                self.price_tree
            except NameError:
```

```python
                raise RuntimeError('Price tree not constructed yet.')

            # Price tree exists, continue
            self.value_tree = self._constructValueTree()

    @abstractmethod
    def valueFromLastCol(self, last_col: np.array) -> np.array:
        """Abstract function to compute the instrument values, given the last
        column of the price matrix. That is, the bottom row of leaf nodes on
        the price tree.

        At runtime, the implementing class can access the current price tree
        from `self.price_tree`.

        See documentation for `GeneralTree._constructValueTree` for more.

        It is required that the returned array has the same dimensions as
        argument `last_col`.

        Arguments:
            last_col {np.array} -- Last column of the price tree. That is, the
                                  bottom row of leaf nodes on the price tree.

        Raises:
            NotImplementedError -- Raised when not implemented.

        Returns:
            np.array -- Array of size equal to argument `last_col`.
        """

        raise NotImplementedError

    @abstractmethod
    def instrumentValueAtNode(self) -> float:
        """Abstract function to compute the instrument value at a given node.

        The implementing class can access the current indexes, current node
        price, current child indexes, and current child values from the
        variables `self._current_row`, `self._current_col`,
        `self._current_val`, `self._child_indexes`, and `self._child_values`,
        respectively.

        See documentation for `GeneralTree._constructValueTree` for more.

        Raises:
            NotImplementedError -- Raised when not implemented.

        Returns:
            float -- Value to be set at the current node.
        """

        raise NotImplementedError

    @abstractmethod
    def childrenPrice(self) -> np.array:
        """Abstract function to compute the price of child nodes, from the
        position of the current node.

        The implementing class can access the current indexes, current node
        price, and current child indexes from the variables `self._current_row`,
        `self._current_col`, `self._current_val`, and `self._child_indexes`,
```

```
        respectively.

        See documentation for `GeneralTree._constructPriceTree` for more.

        It is required that the returned array has size 3, with the format
        [up_child_price, mid_child_price, down_child_price].

        Raises:
            NotImplementedError -- Raised when not implemented.

        Returns:
            np.array -- Array of length 3 with format [up_child_price,
                        mid_child_price, down_child_price].
        """

        raise NotImplementedError

    def getPriceTree(self) -> np.array:
        """Get the constructed price tree.

        Raises:
            RuntimeError -- Raised when the price tree is not constructed yet,
                            note that this only happens if the tree construction
                            flags are used in the initialization method.

        Returns:
            np.array -- Constructed price tree (matrix representation).
        """

        try:
            return self.price_tree.toarray()
        except NameError:
            raise RuntimeError('Price tree not constructed yet.')

    def getValueTree(self) -> np.array:
        """Get the constructed value tree.

        Raises:
            RuntimeError -- Raised when the value tree is not constructed yet,
                            note that this only happens if the tree construction
                            flags are used in the initialization method.

        Returns:
            np.array -- Constructed value tree (matrix representation).
        """

        try:
            return self.value_tree.toarray()
        except NameError:
            raise RuntimeError('Value tree not constructed yet.')

    def getInstrumentValue(self) -> float:
        """Get the value of the instrument as implied by the value tree.

        Raises:
            RuntimeError -- Raised when the value tree is not constructed yet,
                            note that this only happens if the tree construction
                            flags are used in the initialization method.

        Returns:
            float -- Value of the instrument as implied by the value tree.
```

```
        """

        try:
            return self.value_tree[self.mid_row_index, 0]
        except NameError:
            raise RuntimeError('Value tree not constructed yet.')


    def _constructPriceTree(self) -> sparse.dok_matrix:
        """Constructs the price tree.

        It is instantiated as a dictionary of keys matrix (DOK) for efficiency.
        The rows and columns are set to (2 * steps) + 1 and N + 1 respectively.
        For more on the DOK matrix, see: http://bit.ly/2HygbCT.

        The price tree is constructed following the algorithm outlined in my
        notes. See: http://bit.ly/2WhyFem.

        This function calls `childrenPrice` to get the price to set at
        the child nodes. To aid in this process, select variables are exposed
        and can be accessed via the `self` object in the class implementing
        the `childrenPrice` abstract method.

        Specifically, the following variables are static and set once:
            `self.nrow` -- Number of rows of the price tree matrix.
            `self.ncolumn` -- Number of columns of the price tree matrix.
            `self.mid_row_index` -- Index of the middle row of the matrix.

        The following variables are updated on each iteration, and deleted on
        completion of the price tree construction:
            `self._current_row` -- Current row of the iteration.
            `self._current_col` -- Current column of the iteration.
            `self._current_val` -- Price value at the current node.
            `self._child_indexes` -- Current indexes of the children nodes. Has
                                     format [up_idx, mid_idx, low_idx].

        Returns:
            sparse.dok_matrix -- Correctly sized DOK sparse matrix to store the
                                 price tree.
        """

        # Instantiate sparse matrix with correct size and type
        price_tree = sparse.dok_matrix((self.nrow, self.ncolumn), dtype=float)

        # Setting root of tree to given value
        self.mid_row_index = np.floor(self.nrow / 2)
        price_tree[self.mid_row_index, 0] = self.price_tree_root

        # Iterate over columns
        for j in range(0, self.ncolumn - 1):
            # NOTE: The following optimization iterates only over the non-zero
            #       rows. Determined using the triangular pattern of tree data.
            #       Ensures that we will never encounter a node with value 0
            offset = row_low = self.steps - j
            row_high = self.nrow - offset

            # Iterate over rows:
            for i in range(row_low, row_high):
                # Skip to next iteration if current node is 0
                if price_tree[i, j] == 0:
                    continue
```

```
                # Making current i, j, and value global for external visibility
                self._current_row = i
                self._current_col = j
                self._current_val = price_tree[i, j]

                # Update children indexes
                self.__updateChildIndexes()
                # Get deltaX
                deltaX = self.childrenPrice()
                # Update child values
                for idx, child_delX in zip(self._child_indexes, deltaX):
                    price_tree[idx[0], idx[1]] = child_delX

        # Delete intermediate exposed variables
        del self._current_row
        del self._current_col
        del self._current_val
        del self._child_indexes

        # Return final price tree
        return price_tree

    def _constructValueTree(self) -> sparse.dok_matrix:
        """Constructs the value tree.

        This tree is also represented as a dictionary of keys matrix (DOK) for
        efficiency. It has the same dimensions as the price tree.

        The value tree is constructed following the algorithm outlined in my
        notes. See: http://bit.ly/2WrByt9.

        This function calls `valueFromLastCol` and `instrumentValueAtNode` to
        compute the initial last-row (i.e. bottom leaf nodes of the tree) values
        and the value of a given node at traversal, respectively. To aid in this
        process, select variables are exposed and can be accessed via the `self`
        object in the class implementeing the `valueFromLastCol` and
        `instrumentValueAtNode` abstract methods.

        The following variables are updated on each iteration, and deleted on
        completion of the value tree construction:
            `self._current_row` -- Current row of the iteration.
            `self._current_col` -- Current column of the iteration.
            `self._current_val` -- Price value at the current node.
            `self._child_values` -- Value of the current children. Has format
                                    [up_child, mid_child, down_child].
            `self._child_indexes` -- Current indexes of the children nodes. Has
                                     format [up_idx, mid_idx, low_idx].

        Returns:
            sparse.dok_matrix -- Value tree DOK sparse matrix with the same
                                 dimensions as `self.price_tree`.
        """

        # Creating copy of price tree for the value tree
        value_tree = sparse.dok_matrix((self.nrow, self.ncolumn), dtype=float)

        # Applying value function to the last column of child price nodes
        last_row = self.valueFromLastCol(
            last_col=self.price_tree[:, self.ncolumn - 1].toarray()
        )
```

```
        # Updating last column values
        # NOTE: I realize that the loop here is inefficient, but dok_matrix does
        #       not support sliced value setting (as far as I can tell)
        for i in range(0, self.nrow):
            value_tree[i, self.ncolumn - 1] = last_row[i]

        # Iterate over columns (starting with the one-before-last column)
        for j in reversed(range(0, self.ncolumn - 1)):
            # NOTE: The following optimization iterates only over the non-zero
            #       rows. Determined using the triangular pattern of tree data.
            #       Ensures that we will never encounter a node with value 0
            offset = row_low = self.steps - j
            row_high = self.nrow - offset

            for i in range(row_low, row_high):
                # Expose corresponding current node price from `price_tree`
                self._current_val = self.price_tree[i, j]

                # Skip to next iteration if current node in price tree is 0
                if self._current_val == 0:
                    continue

                # Making current i, j and value global for external visiblity
                self._current_row = i
                self._current_col = j

                # Update children indexes
                self.__updateChildIndexes()

                # Building 3x1 array of child values, making globally visible
                child_row_range = range(self._child_indexes[0][0],
                                        self._child_indexes[2][0] + 1)
                self._child_values = value_tree[child_row_range, j + 1]\
                                               .toarray()

                # Set value of current node
                value_tree[i, j] = self.instrumentValueAtNode()

        # Delete intermediate exposed variables
        del self._current_row
        del self._current_col
        del self._current_val
        del self._child_indexes
        del self._child_values

        # Return final value tree
        return value_tree

    def __updateChildIndexes(self) -> np.array:
        """Function to update the `self._child_indexes` with the correct values,
        given the current row index (i), `self._current_row`, and the current
        column index (j), `self._current_col`. `self._child_indexes` is set to a
        tuple (len 3) of tuples (len 2; indexes) with the values,
        corresponding to: ((up_i, up_j), (mid_i, mid_j), (down_i, down_j)).

        Arguments:
            row_idx {int} -- Current row index.
            col_idx {int} -- Current column index.
        """

```

```python
        self._child_indexes = (
            [self._current_row - 1, self._current_col + 1],
            [self._current_row, self._current_col + 1],
            [self._current_row + 1, self._current_col + 1]
        )
```

../fe621/tree_pricing/general_tree.py

B.5 Second-Order Central Finite Difference

```python
from typing import Callable


def secondDerivative(f: Callable, x: float, h: float=1e-7) -> float:
    """Function to numerically approximate the second derivative about a point
    `x`, given a function `f(x)` which takes a single float as its argument.
    This function uses the central finite difference method, computing the slope
    of a nearby secant curve passing through the points
    `(x - h)`, `x`, and `(x - h)`.

    Arguments:
        f {Callable} -- Objective function who's second derivative is computed.
        x {float} -- Point about which the second derivative is computed.

    Keyword Arguments:
        h {float} -- Step size (default: {1e-7}).

    Returns:
        float -- Approxmation of the second derivative of `f` about point `x`.
    """

    return (f(x + h) - (2 * f(x)) + f(x - h)) / (h ** 2)
```

../fe621/numerical_differentiation/second_derivative.py

B.6 Black-Scholes Call Option Price

```python
from .util import computeD1D2

from scipy.stats import norm
import numpy as np


def blackScholesCall(current: float, volatility: float, ttm: float,
                     strike: float, rf: float) -> float:
    """Function to compute the Black-Scholes-Merton price of a European Call
    Option, parameterized by the current underlying asset price, volatility,
    time to expiration, strike price, and risk-free rate.

    Arguments:
        current {float} -- Current price of the underlying asset.
        volatility {float} -- Volatility of the underlying asset price.
        ttm {float} -- Time to expiration (in years).
        strike {float} -- Strike price of the option contract.
        rf {float} -- Risk-free rate (annual).

    Returns:
```

```
        float -- Price of a European Call Option contract.
    """

    d1, d2 = computeD1D2(current, volatility, ttm, strike, rf)

    call = (current * norm.cdf(d1)) \
        - (strike * np.exp(-1 * rf * ttm) * norm.cdf(d2))

    return call
```

../fe621/black_scholes/call.py

B.7 Bisection Method Optimizer

```
from typing import Callable
import numpy as np


def bisectionSolver(f: Callable, a: float, b: float,
                    tol: float=10e-6) -> float:
    """Bisection method solver, implemented using recursion.

    Arguments:
        f {Callable} -- Function to be optimized.
        a {float} -- Lower bound.
        b {float} -- Upper bound.

    Keyword Arguments:
        tol {float} -- Solution tolerance (default: {10e-6}).

    Raises:
        Exception -- Raised if no solution is found.

    Returns:
        float -- Solution to the function s.t. f(x) = 0.
    """

    # Compute midpoint
    mid = (a + b) / 2

    # Check if estimate is within tolerance
    if (b - a) < tol:
        return mid

    # Evaluate function at midpoint
    f_mid = f(mid)

    # Check position of estimate, move point and re-evaluate
    if (f(a) * f_mid) < 0:
        return bisectionSolver(f=f, a=a, b=mid)
    elif (f(b) * f_mid) < 0:
        return bisectionSolver(f=f, a=mid, b=b)
    else:
        raise Exception("No solution found.")
```

../fe621/optimization/bisection.py

B.8 Black-Scholes Put Option Price

```
from .util import computeD1D2

from scipy.stats import norm
import numpy as np


def blackScholesPut(current: float, volatility: float, ttm: float,
                    strike: float, rf: float) -> float:
    """Function to compute the Black-Scholes-Merton price of a European Put
    Option, parameterized by the current underlying asset price, volatility,
    time to expiration, strike price, and risk-free rate.

    Arguments:
        current {float} -- Current price of the underlying asset.
        volatility {float} -- Volatility of the underlying asset price.
        ttm {float} -- Time to expiration (in years).
        strike {float} -- Strike price of the option contract.
        rf {float} -- Risk-free rate (annual).

    Returns:
        float -- Price of a European Put Option contract.
    """

    d1, d2 = computeD1D2(current, volatility, ttm, strike, rf)

    put = (strike * np.exp(-1 * rf * ttm) * norm.cdf(-1 * d2)) \
        - (current * norm.cdf(-1 * d1))

    return put
```

../fe621/black_scholes/put.py

B.9 Trigeorgis Binomial Tree

```
from ..general_tree import GeneralTree

import numpy as np


class Trigeorgis(GeneralTree):
    """Binomial tree option pricing with the Trigeorgis tree. This method is
    outlined in http://bit.ly/2FAT3S0.

    Implemented with the `GeneralTree` abstract class.
    """

    def __init__(self, current: float, strike: float, ttm: float, rf: float,
                 volatility: float, opt_type: str, opt_style: str,
                 steps: int=1):
        """Initialization method for the `Trigeorgis` class.

        Arguments:
            current {float} -- Current asset price.
            strike {float} -- Strike price of the option.
            ttm {float} -- Time to maturity of the option (in years).
            rf {float} -- Risk-free rate (annualized).
            volatility {float} -- Volatility of the underlying asset price.
```

```
            opt_type {str} -- Option type, 'C' for Call, 'P' for Put.
            opt_style {str} -- Option style, 'E' for European, 'A' for American.

        Keyword Arguments:
            steps {int} -- Number of steps to construct (default: {1}).
        """

        # Ensuring valid option type and style
        if opt_type not in ['C', 'P'] or opt_style not in ['A', 'E']:
            raise ValueError('`opt_type` must be \'C\' or \'P\' and `opt_style`\
                must be \'A\' or \'E\'.')

        # Setting class variables
        self.opt_type = opt_type
        self.opt_style = opt_style
        self.rf = rf
        self.volatility = volatility
        self.strike = strike

        # Computing deltaT
        deltaT = ttm / steps

        # Computing upward and downward jumps for children
        # Do this only once so it doesn't have to be recomputed each time
        # Upward additive deltaX
        self.deltaXU = np.sqrt((np.power(rf - (np.power(volatility, 2) / 2), 2)\
                                * np.power(deltaT, 2)) + (np.power(volatility,
                                2) * deltaT))
        # Down deltaX = -1 * upDeltaX
        self.deltaXD = -1 * self.deltaXU

        # Computing jump probabilities for value tree construction
        # Do this only once so it doesn't have to be recomputed each time
        self.jumpU = 0.5 + (0.5 * (rf - (np.power(volatility, 2) / 2)) * deltaT\
                              / self.deltaXU)
        self.jumpD = 1 - self.jumpU

        # Define discount factor for each jump
        self.disc = np.exp(-1 * rf * deltaT)

        # Initializing GeneralTree, with root set to log price for Trigeorgis
        super().__init__(price_tree_root=np.log(current), steps=steps)

    def childrenPrice(self) -> np.array:
        """Function to compute the price of children nodes, given the price at
        the current node.

        Returns:
            np.array -- Array of length 3 corresponding to [up_child_price,
                        mid_child_price, down_child_price].
        """

        # Computing up and downward child additive values (mid is 0)
        up_child_price = self._current_val + self.deltaXU
        down_child_price = self._current_val + self.deltaXD

        return np.array([up_child_price, 0, down_child_price])

    def instrumentValueAtNode(self) -> float:
        """Function to compute the instrument value at the given node.

```

```
        Intelligently adapts to the specificed option style (`self.opt_style`)
        and type (`self.opt_type`) to work with both European options, and the
        path-dependent American option style.

        Returns:
            float -- Value of the option at the given node.
        """

        # Value implied by children
        child_implied_value = self.disc * ((self.jumpU * self._child_values[0])\
                                + (self.jumpD * self._child_values[2]))

        # American option special case
        # NOTE: It is path dependent, so evaluate option value at current node
        #       and return if higher than `child_implied_value`
        if self.opt_style == 'A':
            # Computing value of option if exercied at current node
            # NOTE: Using `valueFromLastCol` here as it is the same computation;
            #       casting current node value to array and passing thru
            option_value = self.valueFromLastCol(last_col=np.array([
                self._current_val]))[0]

            # If value is higher than `child_implied_value`, exercise now
            if option_value > child_implied_value:
                return option_value

        return child_implied_value

    def valueFromLastCol(self, last_col: np.array) -> np.array:
        """Function to compute the option value of the last column (i.e. last
        row of leaf nodes) of the price tree.

        Arguments:
            last_col {np.array} -- Last column of the price tree.

        Returns:
            np.array -- Value of the option corresponding to the input prices.
        """

        # Call option (same for European and American)
        if self.opt_type == 'C':
            # Computing non-floored call option value
            non_floor_val = np.exp(last_col) - self.strike

        # Put option (same for European and American)
        if self.opt_type == 'P':
            # Computing non-floored put option value
            non_floor_val = self.strike - np.exp(last_col)

            # Replacing values equal to (self.strike - 1) with 0. This is to
            # adjust for the fact that zero nodes would have this value in
            # the tree.
            # This is a special case adjustment that must be made to
            # computation. This is purely for clarity.
            non_floor_val = np.where(non_floor_val == (self.strike - 1), 0,
                                     non_floor_val)

        # Floor to 0 and return
        return np.where(non_floor_val > 0, non_floor_val, 0)

    def getPriceTree(self) -> np.array:
```

```
        """Function to get the price tree. Overrides superclass function of the
        same name to return the real price tree as opposed to to the
        log-price tree.

        Returns:
            np.array -- Constructed price tree.
        """

        # Getting log price tree from superclass method
        log_price_tree = super().getPriceTree()
        # Computing real price tree
        price_tree_unadj = np.exp(log_price_tree)

        # Replacing all instances of value '1' with zero, as it would have
        # previously been a zero node before exponentiation
        return np.where(price_tree_unadj == 1, 0, price_tree_unadj)

    def computeOtherStylePrice(self, opt_style: str) -> float:
        """Function to compute the 'other' option style (i.e. American or
        European), given the constructed price tree. Note that this modifies the
        current instance `self.opt_type` and `self.value_tree` variables.

        This is possible for this specific implementation, as the same
        constructed price tree is utilized for both option value calculations.

        This function calls internal functions from abstract class `GeneralTree`
        to recompute the option value, given a change in style.

        Arguments:
            opt_style {str} -- Option style, 'E' for European, 'A' for American.

        Returns:
            float -- Option value of the desired style.
        """

        # Ensuring valid option style
        if opt_style not in ['A', 'E']:
            raise ValueError('`opt_style` must be \'A\' or \'E\'.')

        # If desired option style matches current style, return price
        if opt_style == self.opt_style:
            return self.getInstrumentValue()

        # Setting new option style
        self.opt_style = opt_style

        # Rebuilding value tree (calling superclass internal function here)
        self.value_tree = self._constructValueTree()

        return self.getInstrumentValue()
```

../fe621/tree_pricing/binomial/trigeorgis.py