

## Homework Assignment 2

*FE 621: Computational Methods in Finance*

*Instructor: Ionut Florescu*

3/25/2019

**Rukmal Weerawarana**

*rweerawa@stevens.edu | 104-307-27*

Department of Financial Engineering

Stevens Institute of Technology

---

## Overview

In this Homework Assignment, we explore various tree construction methods, and price various option contracts. I implement a highly generalized Tree, that is extended and utilized throughout the assignment.

Unless otherwise stated, data is from the same dataset used in Homework 1:

- **DATA2** - Thursday, February 7 2019 (2/7/19).

The content of this Homework Assignment is divided into four sections; the first discusses tree construction. The second contains various computations with the Trigeorgis Binomial Tree, and the third discusses additive Trinomial pricing trees. Finally, the fourth section explores the pricing of exotic options with trees.

*See Appendix E for specific question implementations, and the project GitHub repository<sup>1</sup> for full source code of the `fe621` Python package.*

---

1. Weerawarana 2019

## Contents

<b>1</b>	<b>Tree Implementation</b>	<b>1</b>
1.1	General Tree Construction . . . . .	1
1.2	Binomial Tree . . . . .	8
1.3	Trinomial Tree . . . . .	12
<b>2</b>	<b>Binomial Tree Operations</b>	<b>16</b>
2.1	Computed Option Prices . . . . .	16
2.2	Absolute Error Analysis . . . . .	16
2.3	Implied Volatility Computation . . . . .	18
<b>3</b>	<b>Trinomial Tree Operations</b>	<b>19</b>
3.1	Arbitrary Option Price Computation . . . . .	19
3.2	Computed Option Prices . . . . .	20
3.3	Absolute Error Analysis . . . . .	20
3.4	Implied Volatility Computation . . . . .	22
<b>4</b>	<b>Exotic Option Pricing</b>	<b>23</b>
4.1	Barrier Option Tree Description . . . . .	23
4.2	Barrier Option Tree Source Code . . . . .	24
4.3	Analytical Barrier Option Pricing . . . . .	28
4.4	Barrier Option Computation . . . . .	30
<b>A</b>	<b>Binomial Tree Option Prices</b>	<b>32</b>
<b>B</b>	<b>Binomial Tree Implied Volatility</b>	<b>34</b>
<b>C</b>	<b>Trinomial Tree Option Prices</b>	<b>36</b>
<b>D</b>	<b>Trinomial Tree Implied Volatility</b>	<b>38</b>
<b>E</b>	<b>Solution Source Code</b>	<b>39</b>
E.1	Question 1 Implementation . . . . .	39
E.1.1	Binomial Tree Price Computation . . . . .	39
E.1.2	Binomial Tree Absolute Error Analysis . . . . .	40
E.1.3	Binomial Tree Implied Volatility Optimization . . . . .	41
E.2	Question 2 Implementation . . . . .	43
E.2.1	Trinomial Tree Arbitrary Price . . . . .	43
E.2.2	Trinomial Tree Price Computation . . . . .	43
E.2.3	Trinomial Tree Absolute Error Analysis . . . . .	45
E.2.4	Trinomial Tree Implied Volatility Optimization . . . . .	46
E.3	Question 3 Implementation . . . . .	48
E.3.1	Barrier EU Call Option Utilities . . . . .	48
E.3.2	Complete Solution Implementation . . . . .	48

# 1 Tree Implementation

## 1.1 General Tree Construction

To simplify the construction of all tree-like structures, I implemented a fully generalized tree structure. This structure handles tree construction and traversal, completely encapsulating all required functionality.

The class intelligently exposes price and value tree variables during traversal, and abstract methods are designed to be overridden to implement a specific tree pricing algorithm. This general tree class is reproduced below. Furthermore, the class also utilizes DOK (i.e. *Dictionary-of-Keys*) matrices whenever possible. This provides quick  $O(1)$  access to elements, and minimizes overall space complexity, as zero-valued fields are not store explicitly.

```

1 from abc import ABC, abstractmethod
2 from scipy import sparse
3 import numpy as np
4
5
6 class GeneralTree(ABC):
7     """Abstract class enabling efficient implementation of any generalized
8         binomial or trinomial tree pricing or analysis algorithm.
9
10    This implementation of a general tree follows the algorithm outlined in
11    my notes. See: http://bit.ly/2WjfkJu.
12
13    This class may be inherited by a subclass that implements a specific pricing
14    algorithm, while this abstract class handles tree construction, reverse
15    traversal and price computation, given implementations of functions for
16    computing price of children from a current node, the value of the last
17    column (i.e. bottom row of leaf nodes) of a constructed price tree before
18    recombination, and the value of a node given the children values.
19
20    This generalized tree computation methodology allows this class to be used
21    as a base for any arbitrary tree pricing or analysis tool, including
22    multiplicative and additive trees. Tree values are strategically exposed at
23    runtime when building and traversing the tree for added flexibility. Details
24    of specific exposed runtime variables are discussed further in the
25    specific function docstrings.
26
27    Requires that 'GeneralTree.childrenPrice',
28    'GeneralTree.instrumentValueAtNode', and 'GeneralTree.valueFromLastCol'
29    be overridden and implemented. Specific requirements for these abstract
30    methods are outlined in their respective docstrings below.
31
32    Raises:
33        NotImplementedError -- Raised when not implemented.
34    """
35
36    # Need to add documentation to this; explain persistent variables, etc.
37    def __init__(self, price_tree_root: float, steps: int=1,
38                 build_price_tree: bool=True, build_value_tree: bool=True):
39        """Initialization method for the abstract 'GeneralTree' class.
40
41        Constructs both the price and value tree, and isolates the instrument
42        price from the computed value tree.
43
44        Provides flags to suppress the construction of the price tree and the
45        value tree for flexibility. This option allows for an externally
46        constructed price or value tree to be used by setting it to the
47        'price_tree' and 'value_tree' class variables respectively.

```

```

48
49     Arguments:
50         price_tree_root {float} -- Value of the root of the price tree.
51
52     Keyword Arguments:
53         steps {int} -- Number of steps to construct (default: {1}).
54         build_price_tree {bool} -- Price tree flag (default: {True}).
55         build_value_tree {bool} -- Value tree flag (default: {True}).
56
57     Raises:
58         ValueError -- Raised when the number of steps is invalid.
59         RuntimeError -- Raised when invalid sequence is attempted. That is,
60                        if the value tree is attempted to be constructed
61                        without a price tree being constructed first.
62     """
63
64     self.price_tree_root = price_tree_root
65     self.steps = steps
66
67     # Check steps
68     if self.steps < 1:
69         raise ValueError('Must have a step size of at least 1.')
70
71     # Computing shape of matrix representing the tree
72     self.nrow = (2 * self.steps) + 1
73     self.ncolumn = self.steps + 1
74
75     # Construct the price tree
76     if build_price_tree:
77         self.price_tree = self._constructPriceTree()
78
79     # Construct value tree (check that price tree is constructed first)
80     if build_value_tree:
81         try:
82             self.price_tree
83         except NameError:
84             raise RuntimeError('Price tree not constructed yet.')
85
86     # Price tree exists, continue
87     self.value_tree = self._constructValueTree()
88
89     @abstractmethod
90     def valueFromLastCol(self, last_col: np.array) -> np.array:
91         """Abstract function to compute the instrument values, given the last
92         column of the price matrix. That is, the bottom row of leaf nodes on
93         the price tree.
94
95         At runtime, the implementing class can access the current price tree
96         from 'self.price_tree'.
97
98         See documentation for 'GeneralTree._constructValueTree' for more.
99
100         It is required that the returned array has the same dimensions as
101         argument 'last_col'.
102
103         Arguments:
104             last_col {np.array} -- Last column of the price tree. That is, the
105                                   bottom row of leaf nodes on the price tree.
106
107         Raises:
108             NotImplementedError -- Raised when not implemented.

```

```

109
110     Returns:
111         np.array -- Array of size equal to argument 'last_col'.
112     """
113
114     raise NotImplementedError
115
116 @abstractmethod
117 def instrumentValueAtNode(self) -> float:
118     """Abstract function to compute the instrument value at a given node.
119
120     The implementing class can access the current indexes, current node
121     price, current child indexes, and current child values from the
122     variables 'self._current_row', 'self._current_col',
123     'self._current_val', 'self._child_indexes', and 'self._child_values',
124     respectively.
125
126     See documentation for 'GeneralTree._constructValueTree' for more.
127
128     Raises:
129         NotImplementedError -- Raised when not implemented.
130
131     Returns:
132         float -- Value to be set at the current node.
133     """
134
135     raise NotImplementedError
136
137 @abstractmethod
138 def childrenPrice(self) -> np.array:
139     """Abstract function to compute the price of child nodes, from the
140     position of the current node.
141
142     The implementing class can access the current indexes, current node
143     price, and current child indexes from the variables 'self._current_row',
144     'self._current_col', 'self._current_val', and 'self._child_indexes',
145     respectively.
146
147     See documentation for 'GeneralTree._constructPriceTree' for more.
148
149     It is required that the returned array has size 3, with the format
150     [up_child_price, mid_child_price, down_child_price].
151
152     Raises:
153         NotImplementedError -- Raised when not implemented.
154
155     Returns:
156         np.array -- Array of length 3 with format [up_child_price,
157         mid_child_price, down_child_price].
158     """
159
160     raise NotImplementedError
161
162 def getPriceTree(self) -> np.array:
163     """Get the constructed price tree.
164
165     Raises:
166         RuntimeError -- Raised when the price tree is not constructed yet,
167         note that this only happens if the tree construction
168         flags are used in the initialization method.
169

```

```

170     Returns:
171         np.array -- Constructed price tree (matrix representation).
172     """
173
174     try:
175         return self.price_tree.toarray()
176     except NameError:
177         raise RuntimeError('Price tree not constructed yet.')
178
179 def getValueTree(self) -> np.array:
180     """Get the constructed value tree.
181
182     Raises:
183         RuntimeError -- Raised when the value tree is not constructed yet,
184             note that this only happens if the tree construction
185             flags are used in the initialization method.
186
187     Returns:
188         np.array -- Constructed value tree (matrix representation).
189     """
190
191     try:
192         return self.value_tree.toarray()
193     except NameError:
194         raise RuntimeError('Value tree not constructed yet.')
195
196 def getInstrumentValue(self) -> float:
197     """Get the value of the instrument as implied by the value tree.
198
199     Raises:
200         RuntimeError -- Raised when the value tree is not constructed yet,
201             note that this only happens if the tree construction
202             flags are used in the initialization method.
203
204     Returns:
205         float -- Value of the instrument as implied by the value tree.
206     """
207
208     try:
209         return self.value_tree[self.mid_row_index, 0]
210     except NameError:
211         raise RuntimeError('Value tree not constructed yet.')
212
213
214 def _constructPriceTree(self) -> sparse.dok_matrix:
215     """Constructs the price tree.
216
217     It is instantiated as a dictionary of keys matrix (DOK) for efficiency.
218     The rows and columns are set to (2 * steps) + 1 and N + 1 respectively.
219     For more on the DOK matrix, see: http://bit.ly/2HygbCT.
220
221     The price tree is constructed following the algorithm outlined in my
222     notes. See: http://bit.ly/2WhyFem.
223
224     This function calls 'childrenPrice' to get the price to set at
225     the child nodes. To aid in this process, select variables are exposed
226     and can be accessed via the 'self' object in the class implementing
227     the 'childrenPrice' abstract method.
228
229     Specifically, the following variables are static and set once:
230     'self.nrow' -- Number of rows of the price tree matrix.

```

```

231         'self.ncolumn' -- Number of columns of the price tree matrix.
232         'self.mid_row_index' -- Index of the middle row of the matrix.
233
234     The following variables are updated on each iteration, and deleted on
235     completion of the price tree construction:
236         'self._current_row' -- Current row of the iteration.
237         'self._current_col' -- Current column of the iteration.
238         'self._current_val' -- Price value at the current node.
239         'self._child_indexes' -- Current indexes of the children nodes. Has
240                                format [up_idx, mid_idx, low_idx].
241
242     Returns:
243         sparse.dok_matrix -- Correctly sized DOK sparse matrix to store the
244                             price tree.
245     """
246
247     # Instantiate sparse matrix with correct size and type
248     price_tree = sparse.dok_matrix((self.nrow, self.ncolumn), dtype=float)
249
250     # Setting root of tree to given value
251     self.mid_row_index = np.floor(self.nrow / 2)
252     price_tree[self.mid_row_index, 0] = self.price_tree_root
253
254     # Iterate over columns
255     for j in range(0, self.ncolumn - 1):
256         # NOTE: The following optimization iterates only over the non-zero
257         #       rows. Determined using the triangular pattern of tree data.
258         #       Ensures that we will never encounter a node with value 0
259         offset = row_low = self.steps - j
260         row_high = self.nrow - offset
261
262         # Iterate over rows:
263         for i in range(row_low, row_high):
264             # Skip to next iteration if current node is 0
265             if price_tree[i, j] == 0:
266                 continue
267
268             # Making current i, j, and value global for external visibility
269             self._current_row = i
270             self._current_col = j
271             self._current_val = price_tree[i, j]
272
273             # Update children indexes
274             self._updateChildIndexes()
275             # Get deltaX
276             deltaX = self.childrenPrice()
277             # Update child values
278             for idx, child_delX in zip(self._child_indexes, deltaX):
279                 price_tree[idx[0], idx[1]] = child_delX
280
281     # Delete intermediate exposed variables
282     del self._current_row
283     del self._current_col
284     del self._current_val
285     del self._child_indexes
286
287     # Return final price tree
288     return price_tree
289
290 def _constructValueTree(self) -> sparse.dok_matrix:
291     """Constructs the value tree.

```

```

292 This tree is also represented as a dictionary of keys matrix (DOK) for
293 efficiency. It has the same dimensions as the price tree.
294
295
296 The value tree is constructed following the algorithm outlined in my
297 notes. See: http://bit.ly/2WrByt9.
298
299 This function calls 'valueFromLastCol' and 'instrumentValueAtNode' to
300 compute the initial last-row (i.e. bottom leaf nodes of the tree) values
301 and the value of a given node at traversal, respectively. To aid in this
302 process, select variables are exposed and can be accessed via the 'self'
303 object in the class implementing the 'valueFromLastCol' and
304 'instrumentValueAtNode' abstract methods.
305
306 The following variables are updated on each iteration, and deleted on
307 completion of the value tree construction:
308     'self._current_row' -- Current row of the iteration.
309     'self._current_col' -- Current column of the iteration.
310     'self._current_val' -- Price value at the current node.
311     'self._child_values' -- Value of the current children. Has format
312                           [up_child, mid_child, down_child].
313     'self._child_indexes' -- Current indexes of the children nodes. Has
314                           format [up_idx, mid_idx, low_idx].
315
316 Returns:
317     sparse.dok_matrix -- Value tree DOK sparse matrix with the same
318                       dimensions as 'self.price_tree'.
319 """
320
321 # Creating copy of price tree for the value tree
322 value_tree = sparse.dok_matrix((self.nrow, self.ncolumn), dtype=float)
323
324 # Applying value function to the last column of child price nodes
325 last_row = self.valueFromLastCol(
326     last_col=self.price_tree[:, self.ncolumn - 1].toarray()
327 )
328
329 # Updating last column values
330 # NOTE: I realize that the loop here is inefficient, but dok_matrix does
331 #       not support sliced value setting (as far as I can tell)
332 for i in range(0, self.nrow):
333     value_tree[i, self.ncolumn - 1] = last_row[i]
334
335 # Iterate over columns (starting with the one-before-last column)
336 for j in reversed(range(0, self.ncolumn - 1)):
337     # NOTE: The following optimization iterates only over the non-zero
338     #       rows. Determined using the triangular pattern of tree data.
339     #       Ensures that we will never encounter a node with value 0
340     offset = row_low = self.steps - j
341     row_high = self.nrow - offset
342
343     for i in range(row_low, row_high):
344         # Expose corresponding current node price from 'price_tree'
345         self._current_val = self.price_tree[i, j]
346
347         # Skip to next iteration if current node in price tree is 0
348         if self._current_val == 0:
349             continue
350
351         # Making current i, j and value global for external visibility
352         self._current_row = i

```



```

353         self._current_col = j
354
355         # Update children indexes
356         self.__updateChildIndexes()
357
358         # Building 3x1 array of child values, making globally visible
359         child_row_range = range(self._child_indexes[0][0],
360                                self._child_indexes[2][0] + 1)
361         self._child_values = value_tree[child_row_range, j + 1]\
362                               .toarray()
363
364         # Set value of current node
365         value_tree[i, j] = self.instrumentValueAtNode()
366
367         # Delete intermediate exposed variables
368         del self._current_row
369         del self._current_col
370         del self._current_val
371         del self._child_indexes
372         del self._child_values
373
374         # Return final value tree
375         return value_tree
376
377     def __updateChildIndexes(self) -> np.array:
378         """Function to update the 'self._child_indexes' with the correct values,
379         given the current row index (i), 'self._current_row', and the current
380         column index (j), 'self._current_col'. 'self._child_indexes' is set to a
381         tuple (len 3) of tuples (len 2; indexes) with the values,
382         corresponding to: ((up_i, up_j), (mid_i, mid_j), (down_i, down_j)).
383
384         Arguments:
385             row_idx {int} -- Current row index.
386             col_idx {int} -- Current column index.
387         """
388
389         self._child_indexes = (
390             [self._current_row - 1, self._current_col + 1],
391             [self._current_row, self._current_col + 1],
392             [self._current_row + 1, self._current_col + 1]
393         )

```

../fe621/tree\_pricing/general\_tree.py

## 1.2 Binomial Tree

Following this implementation strategy, the additive TrigeorgisTrigeorgis 1991 tree was implemented. Required methods of the abstract `GeneralTree` class were overridden to implement the Trigeorgis tree for both Call and Put options, of both American and European options in a single class.

Furthermore, optimizations were made to the Trigeorgis tree class, such that a European option can be computed from the same price tree as constructed for an American option, and vice versa. This functionality enables efficient cross-style option value computations with minimal space complexity.

The Trigeorgis tree implementation is reproduced below.

```

1 from ..general_tree import GeneralTree
2
3 import numpy as np
4
5
6 class Trigeorgis(GeneralTree):
7     """Binomial tree option pricing with the Trigeorgis tree. This method is
8     outlined in http://bit.ly/2FAT3S0.
9
10    Implemented with the 'GeneralTree' abstract class.
11    """
12
13    def __init__(self, current: float, strike: float, ttm: float, rf: float,
14                 volatility: float, opt_type: str, opt_style: str,
15                 steps: int=1):
16        """Initialization method for the 'Trigeorgis' class.
17
18        Arguments:
19            current {float} -- Current asset price.
20            strike {float} -- Strike price of the option.
21            ttm {float} -- Time to maturity of the option (in years).
22            rf {float} -- Risk-free rate (annualized).
23            volatility {float} -- Volatility of the underlying asset price.
24            opt_type {str} -- Option type, 'C' for Call, 'P' for Put.
25            opt_style {str} -- Option style, 'E' for European, 'A' for American.
26
27        Keyword Arguments:
28            steps {int} -- Number of steps to construct (default: {1}).
29        """
30
31        # Ensuring valid option type and style
32        if opt_type not in ['C', 'P'] or opt_style not in ['A', 'E']:
33            raise ValueError('opt_type must be \'C\' or \'P\' and \'opt_style\' \
34                               must be \'A\' or \'E\'')
35
36        # Setting class variables
37        self.opt_type = opt_type
38        self.opt_style = opt_style
39        self.rf = rf
40        self.volatility = volatility
41        self.strike = strike
42
43        # Computing deltaT
44        deltaT = ttm / steps
45
46        # Computing upward and downward jumps for children
47        # Do this only once so it doesn't have to be recomputed each time
48        # Upward additive deltaX
49        self.deltaXU = np.sqrt((np.power(rf - (np.power(volatility, 2) / 2), 2) \
50                                * np.power(deltaT, 2)) + (np.power(volatility,

```

```

51         2) * deltaT))
52     # Down deltaX = -1 * upDeltaX
53     self.deltaXD = -1 * self.deltaXU
54
55     # Computing jump probabilities for value tree construction
56     # Do this only once so it doesn't have to be recomputed each time
57     self.jumpU = 0.5 + (0.5 * (rf - (np.power(volatility, 2) / 2)) * deltaT\
58                       / self.deltaXU)
59     self.jumpD = 1 - self.jumpU
60
61     # Define discount factor for each jump
62     self.disc = np.exp(-1 * rf * deltaT)
63
64     # Initializing GeneralTree, with root set to log price for Trigeorgis
65     super().__init__(price_tree_root=np.log(current), steps=steps)
66
67     def childrenPrice(self) -> np.array:
68         """Function to compute the price of children nodes, given the price at
69         the current node.
70
71         Returns:
72             np.array -- Array of length 3 corresponding to [up_child_price,
73                     mid_child_price, down_child_price].
74         """
75
76         # Computing up and downward child additive values (mid is 0)
77         up_child_price = self._current_val + self.deltaXU
78         down_child_price = self._current_val + self.deltaXD
79
80         return np.array([up_child_price, 0, down_child_price])
81
82     def instrumentValueAtNode(self) -> float:
83         """Function to compute the instrument value at the given node.
84
85         Intelligently adapts to the specified option style ('self.opt_style')
86         and type ('self.opt_type') to work with both European options, and the
87         path-dependent American option style.
88
89         Returns:
90             float -- Value of the option at the given node.
91         """
92
93         # Value implied by children
94         child_implied_value = self.disc * ((self.jumpU * self._child_values[0])\
95                                           + (self.jumpD * self._child_values[2]))
96
97         # American option special case
98         # NOTE: It is path dependent, so evaluate option value at current node
99         # and return if higher than 'child_implied_value'
100         if self.opt_style == 'A':
101             # Computing value of option if exercised at current node
102             # NOTE: Using 'valueFromLastCol' here as it is the same computation;
103             # casting current node value to array and passing thru
104             option_value = self.valueFromLastCol(last_col=np.array([
105                 self._current_val]))[0]
106
107             # If value is higher than 'child_implied_value', exercise now
108             if option_value > child_implied_value:
109                 return option_value
110
111         return child_implied_value

```

```

112
113 def valueFromLastCol(self, last_col: np.array) -> np.array:
114     """Function to compute the option value of the last column (i.e. last
115     row of leaf nodes) of the price tree.
116
117     Arguments:
118         last_col {np.array} -- Last column of the price tree.
119
120     Returns:
121         np.array -- Value of the option corresponding to the input prices.
122     """
123
124     # Call option (same for European and American)
125     if self.opt_type == 'C':
126         # Computing non-floored call option value
127         non_floor_val = np.exp(last_col) - self.strike
128
129     # Put option (same for European and American)
130     if self.opt_type == 'P':
131         # Computing non-floored put option value
132         non_floor_val = self.strike - np.exp(last_col)
133
134         # Replacing values equal to (self.strike - 1) with 0. This is to
135         # adjust for the fact that zero nodes would have this value in
136         # the tree.
137         # This is a special case adjustment that must be made to
138         # computation. This is purely for clarity.
139         non_floor_val = np.where(non_floor_val == (self.strike - 1), 0,
140                                non_floor_val)
141
142     # Floor to 0 and return
143     return np.where(non_floor_val > 0, non_floor_val, 0)
144
145 def getPriceTree(self) -> np.array:
146     """Function to get the price tree. Overrides superclass function of the
147     same name to return the real price tree as opposed to the
148     log-price tree.
149
150     Returns:
151         np.array -- Constructed price tree.
152     """
153
154     # Getting log price tree from superclass method
155     log_price_tree = super().getPriceTree()
156     # Computing real price tree
157     price_tree_unadj = np.exp(log_price_tree)
158
159     # Replacing all instances of value '1' with zero, as it would have
160     # previously been a zero node before exponentiation
161     return np.where(price_tree_unadj == 1, 0, price_tree_unadj)
162
163 def computeOtherStylePrice(self, opt_style: str) -> float:
164     """Function to compute the 'other' option style (i.e. American or
165     European), given the constructed price tree. Note that this modifies the
166     current instance 'self.opt_type' and 'self.value_tree' variables.
167
168     This is possible for this specific implementation, as the same
169     constructed price tree is utilized for both option value calculations.
170
171     This function calls internal functions from abstract class 'GeneralTree'
172     to recompute the option value, given a change in style.

```

```
173
174     Arguments:
175         opt_style {str} -- Option style, 'E' for European, 'A' for American.
176
177     Returns:
178         float -- Option value of the desired style.
179     """
180
181     # Ensuring valid option style
182     if opt_style not in ['A', 'E']:
183         raise ValueError('opt_style must be \'A\' or \'E\'.')
184
185     # If desired option style matches current style, return price
186     if opt_style == self.opt_style:
187         return self.getInstrumentValue()
188
189     # Setting new option style
190     self.opt_style = opt_style
191
192     # Rebuilding value tree (calling superclass internal function here)
193     self.value_tree = self._constructValueTree()
194
195     return self.getInstrumentValue()
```

../fe621/tree-pricing/binomial/trigeorgis.py

### 1.3 Trinomial Tree

Similar to the Trigeorgis tree above, a generalized additive Trinomial tree was implemented utilizing the same abstract `GeneralTree` class. The implementation of a Trinomial tree utilizing the same abstract class illustrates its versatility, with constructed DOK price and value trees intelligently adapting to the different degree of the tree.

This was accomplished entirely by overriding prescribed abstract methods of the `GeneralTree` class, without any direct modification of the generalized tree class. The Trinomial Additive Tree implementation is reproduced below.

```

1 from ..general_tree import GeneralTree
2
3 import numpy as np
4
5
6 class TrinomialAdditivePriceTree(GeneralTree):
7     """Trinomial tree option pricing with an additive tree. This method is
8     outlined in https://en.wikipedia.org/wiki/Trinomial\_tree.
9
10    Implemented with the 'GeneralTree' abstract class.
11    """
12
13    def __init__(self, current: float, strike: float, ttm: float, rf: float,
14                 volatility: float, opt_type: str, opt_style: str,
15                 dividend: float=0, steps: int=1):
16        """Initialization method for the 'TrinomialAdditivePriceTree' class.
17
18        Arguments:
19            current {float} -- Current asset price.
20            strike {float} -- Strike price of the option.
21            ttm {float} -- Time to maturity of the option (in years).
22            rf {float} -- Risk-free rate (annualized).
23            volatility {float} -- Volatility of the underlying asset price.
24            opt_type {str} -- Option type, 'C' for Call, 'P' for Put.
25            opt_style {str} -- Option style, 'E' for European, 'A' for American.
26
27        Keyword Arguments:
28            dividend {float} -- Cont. div. yield (annualized) (default: {0}).
29            steps {int} -- Number of steps to construct (default: {1}).
30        """
31
32        # Ensuring valid option type and style
33        if opt_type not in ['C', 'P'] or opt_style not in ['A', 'E']:
34            raise ValueError('opt_type must be \'C\' or \'P\' and \'opt_style\' \
35                               must be \'A\' or \'E\'')
36
37        # Setting class variables
38        self.opt_type = opt_type
39        self.opt_style = opt_style
40        self.rf = rf
41        self.volatility = volatility
42        self.strike = strike
43        self.nu = (rf - dividend) - (0.5 * np.power(volatility, 2))
44
45        # Computing deltaT
46        deltaT = ttm / steps
47
48        # Setting upward and downward jumps for children
49        # Setting equal to the convergence condition for now
50        self.deltaXU = volatility * np.sqrt(3 * deltaT)

```

```

51     self.deltaXD = -1 * self.deltaXU
52
53     # Computing upward, middle and downward jumps (additive)
54     self.jumpU = 0.5 * (((np.power(volatility, 2) * deltaT) + (np.power(
55         self.nu, 2) * np.power(deltaT, 2))) / np.power(self.deltaXU, 2)) + \
56         (self.nu * deltaT / self.deltaXU))
57     self.jumpD = 0.5 * (((np.power(volatility, 2) * deltaT) + (np.power(
58         self.nu, 2) * np.power(deltaT, 2))) / np.power(self.deltaXU, 2)) - \
59         (self.nu * deltaT / self.deltaXU))
60     self.jumpM = 1 - self.jumpU - self.jumpD
61
62     # Discount factor for each jump
63     self.disc = np.exp(-1 * rf * deltaT)
64
65     # Initializing GeneralTree, with root set to log price for Additive tree
66     super().__init__(price_tree_root=np.log(current), steps=steps)
67
68     def childrenPrice(self) -> np.array:
69         """Function to compute the price of children nodes, given the price at
70         the current node.
71
72         Returns:
73             np.array -- Array of length 3 corresponding to [up_child_price,
74                 mid_child_price, down_child_price].
75         """
76
77         # Computing upward and downward child additive values (mid is same)
78         up_child_price = self._current_val + self.deltaXU
79         down_child_price = self._current_val + self.deltaXD
80
81         return np.array([up_child_price, self._current_val, down_child_price])
82
83     def instrumentValueAtNode(self) -> float:
84         """Function to compute the instrument value at the given node.
85
86         Intelligently adapts to the specified option style ('self.opt_style')
87         and type ('self.opt_type') to work with both European options, and the
88         path-dependent American option style.
89
90         Returns:
91             float -- Value of the option at the given node.
92         """
93
94         # Value implied by children
95         child_implied_value = self.disc * ((self.jumpU * self._child_values[0]) \
96             + (self.jumpM * self._child_values[1]) \
97             + (self.jumpD * self._child_values[2]))
98
99         # American option special case
100         # NOTE: It is path dependent, so evaluate option value at current node
101         # and return if higher than 'child_implied_value'
102         if self.opt_style == 'A':
103             # Computing value of option if exercised at current node
104             # NOTE: Using 'valueFromLastCol' here as it is the same computation;
105             # casting current node value to array and passing thru
106             option_value = self.valueFromLastCol(last_col=np.array([
107                 self._current_val]))[0]
108
109             # If value is higher than 'child_implied_value', exercise now
110             if option_value > child_implied_value:
111                 return option_value

```

```

112         return child_implied_value
113
114
115     def valueFromLastCol(self, last_col: np.array) -> np.array:
116         """Function to compute the option value of the last column (i.e. last
117         row of leaf nodes) of the price tree.
118
119         Arguments:
120             last_col {np.array} -- Last column of the price tree.
121
122         Returns:
123             np.array -- Value of the option corresponding to the input prices.
124         """
125
126         # Call option (same for European and American)
127         if self.opt_type == 'C':
128             # Computing non-floored call option value
129             non_floor_val = np.exp(last_col) - self.strike
130
131         # Put option (same for European and American)
132         if self.opt_type == 'P':
133             # Computing non-floored put option value
134             non_floor_val = self.strike - np.exp(last_col)
135
136             # Replacing values equal to (self.strike - 1) with 0. This is to
137             # adjust for the fact that zero nodes would have this value in
138             # the tree.
139             # This is a special case adjustment that must be made to
140             # computation. This is purely for clarity.
141             non_floor_val = np.where(non_floor_val == (self.strike - 1), 0,
142                                     non_floor_val)
143
144         # Floor to 0 and return
145         return np.where(non_floor_val > 0, non_floor_val, 0)
146
147     def getPriceTree(self) -> np.array:
148         """Function to get the price tree. Overrides superclass function of the
149         same name to return the real price tree as opposed to the
150         log-price tree.
151
152         Returns:
153             np.array -- Constructed price tree.
154         """
155
156         # Getting log price tree from superclass method
157         log_price_tree = super().getPriceTree()
158         # Computing real price tree
159         price_tree_unadj = np.exp(log_price_tree)
160
161         # Replacing all instances of value '1' with zero, as it would have
162         # previously been a zero node before exponentiation
163         return np.where(price_tree_unadj == 1, 0, price_tree_unadj)
164
165     def computeOtherStylePrice(self, opt_style: str) -> float:
166         """Function to compute the 'other' option style (i.e. American or
167         European), given the constructed price tree. Note that this modifies the
168         current instance 'self.opt_type' and 'self.value_tree' variables.
169
170         This is possible for this specific implementation, as the same
171         constructed price tree is utilized for both option value calculations.
172

```



```
173     This function calls internal functions from abstract class 'GeneralTree'
174     to recompute the option value, given a change in style.
175
176     Arguments:
177         opt_style {str} -- Option style, 'E' for European, 'A' for American.
178
179     Returns:
180         float -- Option value of the desired style.
181     """
182
183     # Ensuring valid option style
184     if opt_style not in ['A', 'E']:
185         raise ValueError('opt_style must be \'A\' or \'E\'')
186
187     # If desired option style matches current style, return price
188     if opt_style == self.opt_style:
189         return self.getInstrumentValue()
190
191     # Setting new option style
192     self.opt_style = opt_style
193
194     # Rebuilding value tree (calling superclass internal function here)
195     self.value_tree = self._constructValueTree()
196
197     return self.getInstrumentValue()
```

../fe621/tree\_pricing/trinomial/trinomial\_price.py

## 2 Binomial Tree Operations

### 2.1 Computed Option Prices

Option prices were computed, utilizing data from Homework Assignment 1's SPY **DATA2** dataset. To begin, both American and European style options were computed for options of various strike prices, with expiration dates varying from 1 to 3 months of the data gathering date.

This data is reproduced in Appendix A, and the source code for this computation is reproduced in Appendix A.

As seen in the tables, the computed values for the European style option with the Binomial tree agree with the analytically computed Black Scholes price. This behavior is to be expected, as the Binomial Tree price converges to the Black Scholes price as the step size,  $N \rightarrow \infty$ .

Furthermore, it can also be noted that the prices of the American style options are consistently higher. Note that some significant figures may be truncated in the presentation of the table in this document.

This behavior is also expected. Under the efficient market hypothesis, and the risk-neutral assumption of option pricing, risk is compensated equally. Thus, the higher cost of the American style options can be attributed to the fact that the holder must pay for the *optionality* provided by the early-exercise feature of American style options, compared to their European style counterparts.

### 2.2 Absolute Error Analysis

To better understand the behavior of the Binomial Tree pricing under varying step sizes, the following error function was plotted for various values of the step size,  $N$ . The source code for this computation is reproduced in Appendix E.1.2.

$$N \in \{10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400\}$$

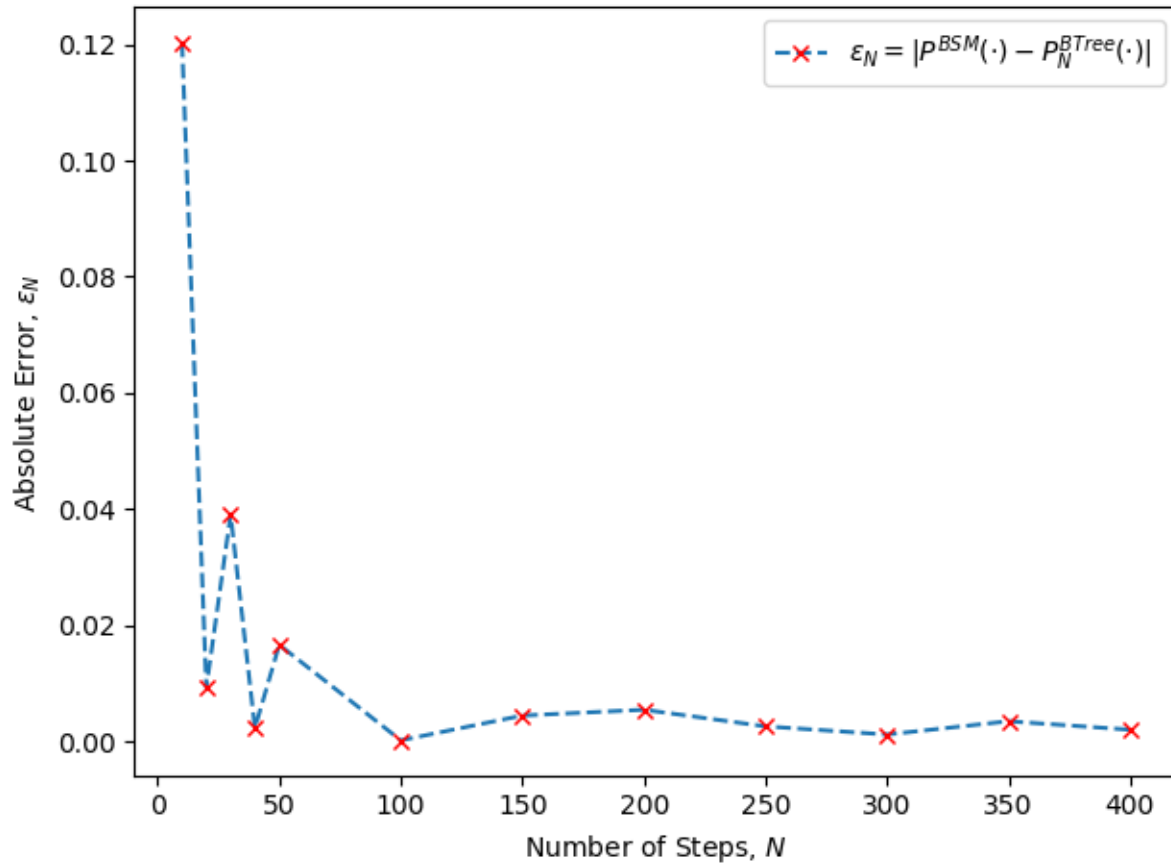
$$\epsilon_N = |P^{BSM}(\cdot) - P_N^{BTree}(\cdot)|$$

Steps	Abs Error
10	0.12032084404936327
20	0.00928024797387872
30	0.03904593917569654
40	0.002298521477286819
50	0.016632059389977805
100	0.0001823855735061386
150	0.004449571374874672
200	0.0054544157620779465
250	0.0025911545465095998
300	0.00118413784583149
350	0.0034705840704774005
400	0.00201394954165357

**Table 1:** Absolute error of Binomial Tree Put Option price computation, with respect to a range of varying step sizes,  $N$ .

Where  $P_N^{BTree}$  is the price of a put option computed with a binomial tree of  $N$  steps. The table of absolute

errors is reproduced in Table 1. Additionally, a graphical representation of this data is also presented in Figure 1.



**Figure 1:** Absolute error analysis of the Binomial Tree price computation convergence, with respect to varying step sizes.

As evidenced by the graphic, there is a clear pattern of convergence to the analytically computed value as the number of steps,  $N$  increases. This is to be expected, as at the limit as  $N \rightarrow \infty$ , the Binomial Tree process will perfectly approximate a continuous geometric brownian motion.

### 2.3 Implied Volatility Computation

Following the algorithm outlined in Homework 1, I computed the implied volatility for the selection of option contracts used in this Homework Assignment, using the Binomial Option Tree, driven by a Bisection Search Algorithm. Source code for this computation is reproduced in Appendix E.1.3.

Compared to the implied volatilities computed in Homework Assignment 1, the volatilities computed here are marginally higher. This too can be attributed to the fact that the holder is compensated fairly for the added optionality provided under the American Option heuristic, used in this computation.

### 3 Trinomial Tree Operations

#### 3.1 Arbitrary Option Price Computation

As outlined in the question, the Trinomial Tree was utilized to compute the price of an option with both Call and Put types, under both the European and American pricing heuristic. The parameters of this option were as follows:

$$S_0 = 100$$

$$K = 100$$

$$T = 1$$

$$\sigma = 25\%$$

$$r = 6\%$$

$$\delta = 0.03$$

With the convergence condition,  $\Delta x \geq \sigma\sqrt{3\Delta t}$

Option Type	Value
European Call	11.001323377114792
American Call	11.001472532311025
European Put	8.133223385060965
American Put	8.500485529931007

**Table 2:** Arbitrary option computation with the Trinomial tree.

The results of this computation are presented in Table 2, and the corresponding source code is reproduced in Appendix E.2.1.

As seen in the previous cases, the price of the American style option is consistently higher than that of its European style counterpart. Again, this is due to the added optionality provided by the American Option pricing heuristic.

### 3.2 Computed Option Prices

Option prices were computed, utilizing data from Homework Assignment 1's SPY **DATA2** dataset. To begin, both American and European style options were computed for options of various strike prices, with expiration dates varying from 1 to 3 months of the data gathering date.

This data is reproduced in Appendix C, and the source code for this computation is reproduced in Appendix C.

As seen in the tables, the computed values for the European style option with the Trinomial tree agree with the analytically computed Black Scholes price. This behavior is to be expected, as the Trinomial Tree price converges to the Black Scholes price as the step size,  $N \rightarrow \infty$ .

Furthermore, it can also be noted that the prices of the American style options are consistently higher. Note that some significant figures may be truncated in the presentation of the table in this document.

As with the Binomial Option, this behavior is also expected. Under the efficient market hypothesis, and the risk-neutral assumption of option pricing, risk is compensated equally. Thus, the higher cost of the American style options can be attributed to the fact that the holder must pay for the *optionality* provided by the early-exercise feature of American style options, compared to their European style counterparts.

### 3.3 Absolute Error Analysis

To better understand the behavior of the Trinomial Tree pricing under varying step sizes, the following error function was plotted for various values of the step size,  $N$ . The source code for this computation is reproduced in Appendix E.2.3.

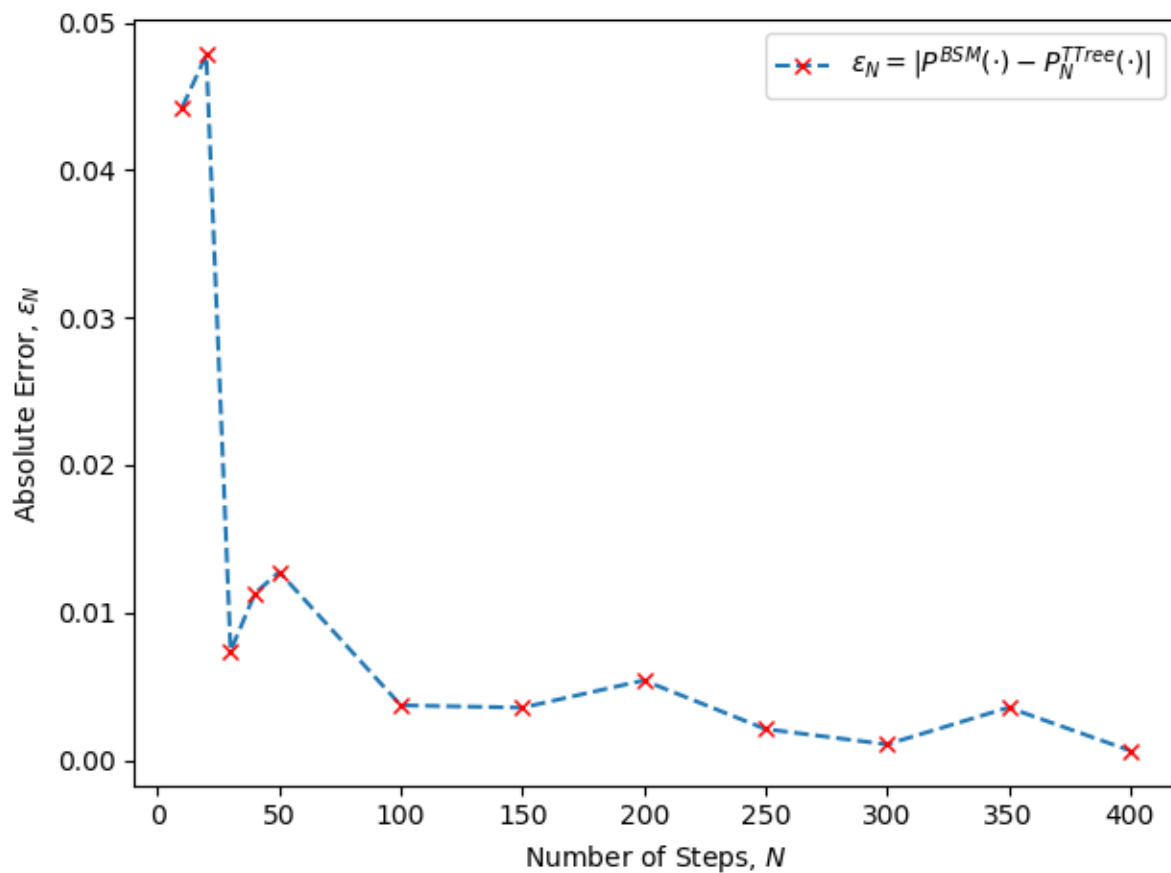
$$N \in \{10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400\}$$

$$\epsilon_N = |P^{BSM}(\cdot) - P_N^{TTree}(\cdot)|$$

Steps	Abs Error
10	0.04422557918418901
20	0.04780683355937887
30	0.007400369564857456
40	0.011312814102280022
50	0.012755416240123996
100	0.003714871983870438
150	0.0035567296842411444
200	0.005407098479568884
250	0.0021066467843509074
300	0.00106554619970467
350	0.0035641295101265236
400	0.0006457655324361156

**Table 3:** Absolute error of Trinomial Tree Put Option price computation, with respect to a range of varying step sizes,  $N$ .

Where  $P_N^{TTree}$  is the price of a put option computed with a trinomial tree of  $N$  steps. The table of absolute errors is reproduced in Table 3. Additionally, a graphical representation of this data is also presented in Figure 2.



**Figure 2:** Absolute error analysis of the Trinomial Tree price computation convergence, with respect to varying step sizes.

As evidenced by the graphic, there is a clear pattern of convergence to a the analytically computed value as the number of steps,  $N$  increases. This is to be expected, as at the limit as  $N \rightarrow \infty$ , (similar to the Binomial Tree), the Trinomial Tree process will perfectly approximate a continuous geometric brownian motion.

### 3.4 Implied Volatility Computation

Following the algorithm outlined in Homework 1, I computed the implied volatility for the selection of option contracts used in this Homework Assignment, using the Trinomial Option Tree, driven by a Bisection Search Algorithm. Source code for this computation is reproduced in Appendix E.1.3.

Compared to the implied volatilities computed in Homework Assignment 1, the volatilities computed here are marginally higher. This too can be attributed to the fact that the holder is compensated fairly for the added optionality provided under the American Option heuristic, used in this computation.

Furthermore, the implied volatilities computed here were also higher compared to their Binomial counterparts. This too is to be expected, as we were evaluating the differing recombination strategies to the same analytical price, thus implying that the given implied volatility was distributed across a larger range of possible price paths.



## 4 Exotic Option Pricing

### 4.1 Barrier Option Tree Description

In this section, we explore the power of tree pricing methods through the lens of exotic option pricing. Specifically, we attempt to price a series of Barrier options, using a Binomial tree, and standard barrier conditions.

As with the Binomial and Trinomial tree, this exotic option pricing tree class too was built on the abstract **GeneralTree** class discussed in Section 1.1.

The implementation of this class builds on the extreme generality and extensibility of the ‘GeneralTree’ class I designed above. It intelligently modifies the control flow of the standard ‘GeneralTree’ class (functionality provided by design) to price all variants of Put and Call type, Up and Down barrier, In and Out barrier type options, under both American and European pricing heuristics.

The ‘Barrier’ class utilizes Python’s multiple inheritance to its advantage, maintaining - in some cases - two simultaneous trees to utilize *In-Out* Parity to compute In barrier type options. Furthermore, as this builds on the excellent scalability of the DOK matrix-driven ‘GeneralTree’ class, it is extremely space efficient, and provides access to tree nodes in  $O(1)$  time complexity.

## 4.2 Barrier Option Tree Source Code

The full source code for the Barrier Option Binomial Tree is reproduced below. Note that it inherits 'GeneralTree', discussed in depth in Section 1.1.

```

1 from ..general_tree import GeneralTree
2 from .trigeorgis import Trigeorgis
3
4 import numpy as np
5
6
7 class Barrier(GeneralTree):
8     """Barrier option pricing with a Trigeorgis tree.
9
10    Implemented with the 'GeneralTree' abstract class.
11    """
12
13    def __init__(self, current: float, strike: float, ttm: float, rf: float,
14                  volatility: float, barrier: float, barrier_type: str,
15                  opt_type: str, opt_style: str, steps: int=1):
16        """Initialization method for the 'Barrier' class.
17
18        Arguments:
19            current {float} -- Current asset price.
20            strike {float} -- Strike price of the option.
21            ttm {float} -- Time to maturity of the option (in years).
22            rf {float} -- Risk-free rate (annualized).
23            volatility {float} -- Volatility of the underlying asset price.
24            barrier {float} -- Barrier of the option (sometimes called 'H').
25            barrier_type {str} -- Barrier type, 'O' for out and 'I' for in.
26            opt_type {str} -- Option type, 'C' for Call, 'P' for Put.
27            opt_style {str} -- Option style, 'E' for European, 'A' for American.
28
29        Keyword Arguments:
30            steps {int} -- Number of steps to construct (default: {1}).
31        """
32
33        # Ensuring valid option type and style
34        if opt_type not in ['C', 'P'] or opt_style not in ['A', 'E']:
35            raise ValueError('opt_type must be \'C\' or \'P\' and \'opt_style\' \
36                             must be \'A\' or \'E\'')
37
38        # Ensuring valid barrier option type
39        if barrier_type not in ['I', 'O']:
40            raise ValueError('barrier_type must be \'I\' for In type options, \
41                             or \'O\' for Out type options.')
42
43        # Inferring barrier option characteristic from current price
44        # i.e. either Up or Down
45        if barrier > current:
46            self.barrier_characteristic = 'U'
47        else:
48            self.barrier_characteristic = 'D'
49
50        # Setting class variables
51        self.opt_type = opt_type
52        self.opt_style = opt_style
53        self.rf = rf
54        self.volatility = volatility
55        self.strike = strike
56        self.barrier = barrier

```

```

57     self.barrier_log = np.log(barrier)
58
59     # Override barrier type if option style is European
60     if opt_style == 'E':
61         self.barrier_type = '0' # Necessary for this to work
62     else:
63         self.barrier_type = barrier_type
64
65     # Computing deltaT
66     deltaT = ttm / steps
67
68     # Computing upward and downward jumps for children
69     # Do this only once so it doesn't have to be recomputed each time
70     # Upward additive deltaX
71     self.deltaXU = np.sqrt((np.power(rf - (np.power(volatility, 2) / 2), 2) \
72                             * np.power(deltaT, 2)) + (np.power(volatility,
73                             2) * deltaT))
74     # Down deltaX = -1 * upDeltaX
75     self.deltaXD = -1 * self.deltaXU
76
77     # Computing jump probabilities for value tree construction
78     # Do this only once so it doesn't have to be recomputed each time
79     self.jumpU = 0.5 + (0.5 * (rf - (np.power(volatility, 2) / 2)) * deltaT \
80                       / self.deltaXU)
81     self.jumpD = 1 - self.jumpU
82
83     # Define discount factor for each jump
84     self.disc = np.exp(-1 * rf * deltaT)
85
86     # Initializing GeneralTree, with root set to ln price for Trigeorgis
87     super().__init__(price_tree_root=np.log(current), steps=steps)
88
89     if (barrier_type == 'I'):
90         # Special case for 'In' barrier type
91         vanilla_tree = Trigeorgis(current=current, strike=strike, ttm=ttm,
92                                   rf=rf, volatility=volatility, opt_type=opt_type,
93                                   opt_style=opt_style, steps=steps)
94         self.value_tree = (vanilla_tree.value_tree - self.value_tree) \
95             .todok(copy=True)
96
97     def childrenPrice(self) -> np.array:
98         """Function to compute the price of children nodes, given the price at
99         the current node.
100
101         Returns:
102             np.array -- Array of length 3 corresponding to [up_child_price,
103                     mid_child_price, down_child_price].
104         """
105
106         # Computing up and downward child additive values (mid is 0)
107         up_child_price = self._current_val + self.deltaXU
108         down_child_price = self._current_val + self.deltaXD
109
110         # Computing barrier indicator for each of the child prices
111         up_child_price = up_child_price \
112             * self.barrierIndicator(np.exp(up_child_price))
113         down_child_price = down_child_price \
114             * self.barrierIndicator(np.exp(down_child_price))
115
116         return np.array([up_child_price, 0, down_child_price])
117

```

```

118 def instrumentValueAtNode(self) -> float:
119     """Function to compute the instrument value at the given node.
120
121     Intelligently adapts to the specified option style ('self.opt_style')
122     and type ('self.opt_type') to work with both European options, and the
123     path-dependent American option style.
124
125     Returns:
126         float -- Value of the option at the given node.
127     """
128
129     # Value implied by children
130     child_implied_value = self.disc * ((self.jumpU * self._child_values[0])\
131         + (self.jumpD * self._child_values[2]))
132
133     # American option special case
134     # NOTE: It is path dependent, so evaluate option value at current node
135     # and return if higher than 'child_implied_value'
136     if self.opt_style == 'A':
137         # Computing value of option if exercised at current node
138         # NOTE: Using 'valueFromLastCol' here as it is the same computation;
139         # casting current node value to array and passing thru
140         option_value = self.valueFromLastCol(last_col=np.array([
141             self._current_val]))[0]
142
143         # If value is higher than 'child_implied_value', exercise now
144         if option_value > child_implied_value:
145             return option_value
146
147     return child_implied_value
148
149 def valueFromLastCol(self, last_col: np.array) -> np.array:
150     """Function to compute the option value of the last column (i.e. last
151     row of leaf nodes) of the price tree.
152
153     Arguments:
154         last_col {np.array} -- Last column of the price tree.
155
156     Returns:
157         np.array -- Value of the option corresponding to the input prices.
158     """
159
160     # Computing prices of each of the values
161     underlying_prices = np.array([0 if i == 0 else np.exp(i)
162         for i in last_col])
163
164     # Computing barrier indicator function values for last_col values
165     indicator_val = np.array([[self.barrierIndicator(i)]
166         for i in underlying_prices])
167
168     # Call option (same for European and American)
169     if self.opt_type == 'C':
170         # Computing non-floored call option value
171         non_floor_val = np.exp(last_col) - self.strike
172
173     # Put option (same for European and American)
174     if self.opt_type == 'P':
175         # Computing non-floored put option value
176         non_floor_val = self.strike - np.exp(last_col)
177
178     # Replacing values equal to (self.strike - 1) with 0. This is to

```

```

179         # adjust for the fact that zero nodes would have this value in
180         # the tree.
181         # This is a special case adjustment that must be made to
182         # computation. This is purely for clarity.
183         non_floor_val = np.where(non_floor_val == (self.strike - 1), 0,
184                                 non_floor_val)
185
186         # Floor to 0
187         floor_val = np.where(non_floor_val > 0, non_floor_val, 0)
188
189         # Return element-wise product of floor_val and barrier indicator
190         return np.multiply(indicator_val, floor_val)
191
192     def barrierIndicator(self, underlying_price: float) -> int:
193         """Indicator for the barrier option.
194
195         Returns the corresponding value for the barrier indicator function,
196         depending on the barrier type and barrier characteristic.
197
198         Arguments:
199             underlying_price {float} -- Log price of the underlying asset.
200
201         Returns:
202             int -- 1 or 0 depending on indicator function.
203         """
204
205         # Down and out
206         if (self.barrier_characteristic == 'D') and (self.barrier_type == 'O'):
207             return 1 if underlying_price > self.barrier else 0
208
209         # Down and in
210         if (self.barrier_characteristic == 'D') and (self.barrier_type == 'I'):
211             return 1 if underlying_price <= self.barrier else 0
212
213         # Up and out
214         if (self.barrier_characteristic == 'U') and (self.barrier_type == 'O'):
215             return 1 if underlying_price < self.barrier else 0
216
217         # Up and in
218         if (self.barrier_characteristic == 'U') and (self.barrier_type == 'I'):
219             return 1 if underlying_price >= self.barrier else 0
220
221     def getPriceTree(self) -> np.array:
222         """Function to get the price tree. Overrides superclass function of the
223         same name to return the real price tree as opposed to to the
224         log-price tree.
225
226         Returns:
227             np.array -- Constructed price tree.
228         """
229
230         # Getting log price tree from superclass method
231         log_price_tree = super().getPriceTree()
232         # Computing real price tree
233         price_tree_unadj = np.exp(log_price_tree)
234
235         # Replacing all instances of value '1' with zero, as it would have
236         # previously been a zero node before exponentiation
237         return np.where(price_tree_unadj == 1, 0, price_tree_unadj)

```

../fe621/tree-pricing/binomial/barrier.py

### 4.3 Analytical Barrier Option Pricing

Additionally, the analytical formula for computing the price of a barrier option were also implemented. This source code is reproduced below.

```

1 from .util import AnalyticalUtil
2
3 from scipy.stats import norm
4 import numpy as np
5
6
7 def callUpAndIn(S: float, H: float, volatility: float, ttm: float,
8                 K: float, rf: float, dividend: float=0) -> float:
9     """Analytical formula to compute the value of an up and in Barrier option.
10
11     See formula 5.1 in http://bit.ly/2JHoVbQ for more.
12
13     Arguments:
14         S {float} -- Current price.
15         H {float} -- Barrier price.
16         volatility {float} -- Volatility of the underlying.
17         ttm {float} -- Time to maturity (in years).
18         K {float} -- Strike price.
19         rf {float} -- Risk-free rate (annualized).
20
21     Keyword Arguments:
22         dividend {float} -- Dividend yield (default: {0}).
23
24     Returns:
25         float -- Analytical value of up and in call option.
26     """
27
28     util = AnalyticalUtil(volatility=volatility, ttm=ttm, rf=rf,
29                           dividend=dividend)
30
31     return np.power((H / S), 2 * util.nu / np.power(volatility, 2)) * \
32         (util.pBS(np.power(H, 2) / S, K) - util.pBS(np.power(H, 2) / S, H) + \
33          ((H - K) * np.exp(-1 * rf * ttm) * norm.cdf(-1 * util.dBS(H, S)))) + \
34         util.cBS(S, H) + ((H - K) * np.exp(-1 * rf * ttm) * norm.cdf(
35             util.dBS(S, H)))
36
37
38 def callUpAndOut(S: float, H: float, volatility: float, ttm: float, K: float,
39                  rf: float, dividend: float=0) -> float:
40     """Analytical formula to compute the value of an up and out barrier call
41     option.
42
43     See formula 5.2 in http://bit.ly/2JHoVbQ for more.
44
45     Arguments:
46         S {float} -- Current price.
47         H {float} -- Barrier price.
48         volatility {float} -- Volatility of the underlying.
49         ttm {float} -- Time tp maturity (in years).
50         K {float} -- Strike price.
51         rf {float} -- Risk-free rate (annualized).
52
53     Keyword Arguments:
54         dividend {float} -- Dividend yield (default: {0}).
55
56     Returns:

```

```
57         float -- Analytical price of up and out barrier call option.
58         """
59
60         util = AnalyticalUtil(volatility=volatility, ttm=ttm, rf=rf,
61                               dividend=dividend)
62
63         return util.cBS(S, K) - util.cBS(S, H) - ((H - K) * np.exp(-1 * rf * ttm)\
64            * norm.cdf(util.dBS(S, H))) - (np.power(H / S, 2 * util.nu / np.power(
65            volatility, 2)) * (util.cBS(np.power(H, 2) / S, K) - util.cBS(np.power(
66            H, 2) / S, H) - ((H - K) * np.exp(-1 * rf * ttm)) * norm.cdf(util.dBS(
67            H, S))))
```

../fe621/black\_scholes/barrier/call.py

#### 4.4 Barrier Option Computation

Here, I present a table with options computed using my Tree implementation, *In-Out* parity, and the analytical formula. The parameters of the computed options are as follows:

$$S_0 = 10$$

$$K = 10$$

$$T = 0.3$$

$$\sigma = 0.2$$

$$r = 0.01$$

$$H = 11$$

$$N = 200$$

Type	Value
Tree Up-and-Out EU Call	0.06262109819997087
Analytical Up-and-Out EU Call	0.05309279660325303
Tree Up-and-In EU Call	0.38812231889718535
Analytical Up-and-In EU Call	0.3981948482776454
Tree Up-and-In A Call	0.4507434170971562

**Table 4:** Values of various barrier options, computed with my Barrier Option Tree, *In-Out* parity, and the analytical formula.



## References

- Trigeorgis, Lenos. 1991. "A Log-Transformed Binomial Numerical Analysis Method for Valuing Complex Multi-Option Investments." *The Journal of Financial and Quantitative Analysis* 26, no. 3 (September): 309. ISSN: 00221090. doi:10.2307/2331209. <https://www.jstor.org/stable/2331209?origin=crossref>.
- Weerawarana, Rukmal. 2019. *FE 621 Homework - rukmal - GitHub*. Accessed February 20, 2019. <https://github.com/rukmal/FE-621-Homework>.

## A Binomial Tree Option Prices

Option Name	Strike	Implied Volatility	Binomial (A)	Binomial (E)	BS (E)
SPY190215C00265000	265.0000	0.1596	6.1422	6.1422	6.1400
SPY190215P00265000	265.0000	0.1580	0.7983	0.7961	0.7900
SPY190215C00266000	266.0000	0.1557	5.3300	5.3300	5.3300
SPY190215P00266000	266.0000	0.1519	0.9537	0.9509	0.9500
SPY190215C00267000	267.0000	0.1476	4.4952	4.4952	4.4900
SPY190215P00267000	267.0000	0.1470	1.1622	1.1585	1.1600
SPY190215C00268000	268.0000	0.1392	3.6939	3.6939	3.6900
SPY190215P00268000	268.0000	0.1409	1.3968	1.3919	1.3900
SPY190215C00269000	269.0000	0.1365	3.0327	3.0327	3.0300
SPY190215P00269000	269.0000	0.1359	1.6994	1.6927	1.6900
SPY190215C00270000	270.0000	0.1330	2.4190	2.4190	2.4200
SPY190215P00270000	270.0000	0.1310	2.0652	2.0562	2.0600
SPY190215C00271000	271.0000	0.1293	1.8709	1.8709	1.8700
SPY190215P00271000	271.0000	0.1267	2.5102	2.4976	2.5000
SPY190215C00272000	272.0000	0.1248	1.3908	1.3908	1.3900
SPY190215P00272000	272.0000	0.1227	3.0419	3.0251	3.0200
SPY190215C00273000	273.0000	0.1209	0.9908	0.9908	0.9900
SPY190215P00273000	273.0000	0.1195	3.6612	3.6382	3.6400
SPY190215C00274000	274.0000	0.1181	0.6897	0.6897	0.6900
SPY190215P00274000	274.0000	0.1175	4.3802	4.3497	4.3500
SPY190215C00275000	275.0000	0.1160	0.4615	0.4615	0.4600
SPY190215P00275000	275.0000	0.1162	5.1718	5.1317	5.1300
SPY190315C00265000	265.0000	0.1528	8.6051	8.6051	8.6000
SPY190315P00265000	265.0000	0.1637	3.1554	3.1328	3.1300
SPY190315C00266000	266.0000	0.1461	7.7469	7.7469	7.7400
SPY190315P00266000	266.0000	0.1613	3.4355	3.4099	3.4100
SPY190315C00267000	267.0000	0.1443	7.0625	7.0625	7.0600
SPY190315P00267000	267.0000	0.1589	3.7504	3.7210	3.7200
SPY190315C00268000	268.0000	0.1466	6.5428	6.5428	6.5400
SPY190315P00268000	268.0000	0.1564	4.0795	4.0460	4.0500
SPY190315C00269000	269.0000	0.1395	5.7430	5.7430	5.7400
SPY190315P00269000	269.0000	0.1541	4.4506	4.4122	4.4100
SPY190315C00270000	270.0000	0.1378	5.1431	5.1431	5.1400
SPY190315P00270000	270.0000	0.1523	4.8555	4.8115	4.8100
SPY190315C00271000	271.0000	0.1356	4.5763	4.5763	4.5700
SPY190315P00271000	271.0000	0.1505	5.3071	5.2570	5.2500
SPY190315C00272000	272.0000	0.1323	3.9870	3.9870	3.9900
SPY190315P00272000	272.0000	0.1487	5.7717	5.7130	5.7200
SPY190315C00273000	273.0000	0.1300	3.4812	3.4812	3.4800
SPY190315P00273000	273.0000	0.1471	6.2969	6.2317	6.2300
SPY190315C00274000	274.0000	0.1273	2.9953	2.9953	2.9900
SPY190315P00274000	274.0000	0.1463	6.8635	6.7881	6.7900
SPY190315C00275000	275.0000	0.1253	2.5657	2.5657	2.5700
SPY190315P00275000	275.0000	0.1453	7.4641	7.3797	7.3800

Option Name	Strike	Implied Volatility	Binomial (A)	Binomial (E)	BS (E)
SPY190418C00265000	265.0000	0.1393	10.2397	10.2397	10.2400
SPY190418P00265000	265.0000	0.1660	5.0641	5.0063	5.0100
SPY190418C00266000	266.0000	0.1370	9.5104	9.5104	9.5100
SPY190418P00266000	266.0000	0.1639	5.3774	5.3152	5.3100
SPY190418C00267000	267.0000	0.1364	8.8839	8.8839	8.8800
SPY190418P00267000	267.0000	0.1622	5.7124	5.6439	5.6400
SPY190418C00268000	268.0000	0.1352	8.2343	8.2343	8.2400
SPY190418P00268000	268.0000	0.1604	6.0684	5.9920	5.9900
SPY190418C00269000	269.0000	0.1337	7.6214	7.6214	7.6100
SPY190418P00269000	269.0000	0.1582	6.4299	6.3471	6.3400
SPY190418C00270000	270.0000	0.1311	6.9524	6.9524	6.9600
SPY190418P00270000	270.0000	0.1563	6.8054	6.7131	6.7200
SPY190418C00271000	271.0000	0.1293	6.3673	6.3673	6.3600
SPY190418P00271000	271.0000	0.1548	7.2461	7.1449	7.1400
SPY190418C00272000	272.0000	0.1277	5.7999	5.7999	5.8000
SPY190418P00272000	272.0000	0.1527	7.6666	7.5556	7.5500
SPY190418C00273000	273.0000	0.1260	5.2622	5.2622	5.2600
SPY190418P00273000	273.0000	0.1519	8.1680	8.0438	8.0500
SPY190418C00274000	274.0000	0.1242	4.7439	4.7439	4.7400
SPY190418P00274000	274.0000	0.1507	8.6900	8.5556	8.5500
SPY190418C00275000	275.0000	0.1228	4.2763	4.2763	4.2700
SPY190418P00275000	275.0000	0.1481	9.1491	8.9991	9.0000

## B Binomial Tree Implied Volatility

Option Name	Strike	Type	Binomial Implied Volatility
SPY190215C00265000	265.0000	265.0000	0.1592
SPY190215P00265000	265.0000	265.0000	0.1572
SPY190215C00266000	266.0000	266.0000	0.1556
SPY190215P00266000	266.0000	266.0000	0.1513
SPY190215C00267000	267.0000	267.0000	0.1472
SPY190215P00267000	267.0000	267.0000	0.1468
SPY190215C00268000	268.0000	268.0000	0.1387
SPY190215P00268000	268.0000	268.0000	0.1402
SPY190215C00269000	269.0000	269.0000	0.1361
SPY190215P00269000	269.0000	269.0000	0.1350
SPY190215C00270000	270.0000	270.0000	0.1333
SPY190215P00270000	270.0000	270.0000	0.1309
SPY190215C00271000	271.0000	271.0000	0.1285
SPY190215P00271000	271.0000	271.0000	0.1254
SPY190215C00272000	272.0000	272.0000	0.1247
SPY190215P00272000	272.0000	272.0000	0.1211
SPY190215C00273000	273.0000	273.0000	0.1212
SPY190215P00273000	273.0000	273.0000	0.1184
SPY190215C00274000	274.0000	274.0000	0.1179
SPY190215P00274000	274.0000	274.0000	0.1150
SPY190215C00275000	275.0000	275.0000	0.1154
SPY190215P00275000	275.0000	275.0000	0.1117
SPY190315C00265000	265.0000	265.0000	0.1520
SPY190315P00265000	265.0000	265.0000	0.1625
SPY190315C00266000	266.0000	266.0000	0.1459
SPY190315P00266000	266.0000	266.0000	0.1609
SPY190315C00267000	267.0000	267.0000	0.1445
SPY190315P00267000	267.0000	267.0000	0.1579
SPY190315C00268000	268.0000	268.0000	0.1459
SPY190315P00268000	268.0000	268.0000	0.1547
SPY190315C00269000	269.0000	269.0000	0.1390
SPY190315P00269000	269.0000	269.0000	0.1525
SPY190315C00270000	270.0000	270.0000	0.1381
SPY190315P00270000	270.0000	270.0000	0.1513
SPY190315C00271000	271.0000	271.0000	0.1353
SPY190315P00271000	271.0000	271.0000	0.1486
SPY190315C00272000	272.0000	272.0000	0.1317
SPY190315P00272000	272.0000	272.0000	0.1463
SPY190315C00273000	273.0000	273.0000	0.1302
SPY190315P00273000	273.0000	273.0000	0.1451
SPY190315C00274000	274.0000	274.0000	0.1270
SPY190315P00274000	274.0000	274.0000	0.1442
SPY190315C00275000	275.0000	275.0000	0.1249
SPY190315P00275000	275.0000	275.0000	0.1421

Option Name	Strike	Type	Binomial Implied Volatility
SPY190418C00265000	265.0000	265.0000	0.1395
SPY190418P00265000	265.0000	265.0000	0.1650
SPY190418C00266000	266.0000	266.0000	0.1373
SPY190418P00266000	266.0000	266.0000	0.1621
SPY190418C00267000	267.0000	267.0000	0.1360
SPY190418P00267000	267.0000	267.0000	0.1599
SPY190418C00268000	268.0000	268.0000	0.1345
SPY190418P00268000	268.0000	268.0000	0.1581
SPY190418C00269000	269.0000	269.0000	0.1333
SPY190418P00269000	269.0000	269.0000	0.1562
SPY190418C00270000	270.0000	270.0000	0.1317
SPY190418P00270000	270.0000	270.0000	0.1549
SPY190418C00271000	271.0000	271.0000	0.1292
SPY190418P00271000	271.0000	271.0000	0.1525
SPY190418C00272000	272.0000	272.0000	0.1271
SPY190418P00272000	272.0000	272.0000	0.1496
SPY190418C00273000	273.0000	273.0000	0.1255
SPY190418P00273000	273.0000	273.0000	0.1486
SPY190418C00274000	274.0000	274.0000	0.1244
SPY190418P00274000	274.0000	274.0000	0.1476
SPY190418C00275000	275.0000	275.0000	0.1228
SPY190418P00275000	275.0000	275.0000	0.1453

## C Trinomial Tree Option Prices

Option Name	Strike	Implied Volatility	Trinomial (A)	Trinomial (E)	BS (E)
SPY190215C00265000	265.0000	0.1596	6.1409	6.1409	6.1400
SPY190215P00265000	265.0000	0.1580	0.7970	0.7951	0.7900
SPY190215C00266000	266.0000	0.1557	5.3275	5.3275	5.3300
SPY190215P00266000	266.0000	0.1519	0.9522	0.9497	0.9500
SPY190215C00267000	267.0000	0.1476	4.4923	4.4923	4.4900
SPY190215P00267000	267.0000	0.1470	1.1594	1.1559	1.1600
SPY190215C00268000	268.0000	0.1392	3.6890	3.6890	3.6900
SPY190215P00268000	268.0000	0.1409	1.3918	1.3870	1.3900
SPY190215C00269000	269.0000	0.1365	3.0310	3.0310	3.0300
SPY190215P00269000	269.0000	0.1359	1.6974	1.6910	1.6900
SPY190215C00270000	270.0000	0.1330	2.4184	2.4184	2.4200
SPY190215P00270000	270.0000	0.1310	2.0642	2.0556	2.0600
SPY190215C00271000	271.0000	0.1293	1.8724	1.8724	1.8700
SPY190215P00271000	271.0000	0.1267	2.5110	2.4989	2.5000
SPY190215C00272000	272.0000	0.1248	1.3883	1.3883	1.3900
SPY190215P00272000	272.0000	0.1227	3.0385	3.0220	3.0200
SPY190215C00273000	273.0000	0.1209	0.9925	0.9925	0.9900
SPY190215P00273000	273.0000	0.1195	3.6628	3.6405	3.6400
SPY190215C00274000	274.0000	0.1181	0.6883	0.6883	0.6900
SPY190215P00274000	274.0000	0.1175	4.3785	4.3485	4.3500
SPY190215C00275000	275.0000	0.1160	0.4611	0.4611	0.4600
SPY190215P00275000	275.0000	0.1162	5.1710	5.1314	5.1300
SPY190315C00265000	265.0000	0.1528	8.6043	8.6043	8.6000
SPY190315P00265000	265.0000	0.1637	3.1439	3.1220	3.1300
SPY190315C00266000	266.0000	0.1461	7.7432	7.7432	7.7400
SPY190315P00266000	266.0000	0.1613	3.4381	3.4135	3.4100
SPY190315C00267000	267.0000	0.1443	7.0586	7.0586	7.0600
SPY190315P00267000	267.0000	0.1589	3.7413	3.7125	3.7200
SPY190315C00268000	268.0000	0.1466	6.5433	6.5433	6.5400
SPY190315P00268000	268.0000	0.1564	4.0805	4.0482	4.0500
SPY190315C00269000	269.0000	0.1395	5.7386	5.7386	5.7400
SPY190315P00269000	269.0000	0.1541	4.4451	4.4078	4.4100
SPY190315C00270000	270.0000	0.1378	5.1424	5.1424	5.1400
SPY190315P00270000	270.0000	0.1523	4.8537	4.8108	4.8100
SPY190315C00271000	271.0000	0.1356	4.5732	4.5732	4.5700
SPY190315P00271000	271.0000	0.1505	5.3027	5.2539	5.2500
SPY190315C00272000	272.0000	0.1323	3.9891	3.9891	3.9900
SPY190315P00272000	272.0000	0.1487	5.7743	5.7179	5.7200
SPY190315C00273000	273.0000	0.1300	3.4755	3.4755	3.4800
SPY190315P00273000	273.0000	0.1471	6.2894	6.2252	6.2300
SPY190315C00274000	274.0000	0.1273	2.9895	2.9895	2.9900
SPY190315P00274000	274.0000	0.1463	6.8647	6.7918	6.7900
SPY190315C00275000	275.0000	0.1253	2.5685	2.5685	2.5700
SPY190315P00275000	275.0000	0.1453	7.4579	7.3741	7.3800

Option Name	Strike	Implied Volatility	Trinomial (A)	Trinomial (E)	BS (E)
SPY190418C00265000	265.0000	0.1393	10.2406	10.2406	10.2400
SPY190418P00265000	265.0000	0.1660	5.0602	5.0049	5.0100
SPY190418C00266000	266.0000	0.1370	9.5056	9.5056	9.5100
SPY190418P00266000	266.0000	0.1639	5.3680	5.3070	5.3100
SPY190418C00267000	267.0000	0.1364	8.8804	8.8804	8.8800
SPY190418P00267000	267.0000	0.1622	5.7129	5.6463	5.6400
SPY190418C00268000	268.0000	0.1352	8.2361	8.2361	8.2400
SPY190418P00268000	268.0000	0.1604	6.0586	5.9839	5.9900
SPY190418C00269000	269.0000	0.1337	7.6171	7.6171	7.6100
SPY190418P00269000	269.0000	0.1582	6.4238	6.3427	6.3400
SPY190418C00270000	270.0000	0.1311	6.9518	6.9518	6.9600
SPY190418P00270000	270.0000	0.1563	6.8027	6.7124	6.7200
SPY190418C00271000	271.0000	0.1293	6.3642	6.3642	6.3600
SPY190418P00271000	271.0000	0.1548	7.2406	7.1418	7.1400
SPY190418C00272000	272.0000	0.1277	5.7931	5.7931	5.8000
SPY190418P00272000	272.0000	0.1527	7.6581	7.5488	7.5500
SPY190418C00273000	273.0000	0.1260	5.2604	5.2604	5.2600
SPY190418P00273000	273.0000	0.1519	8.1687	8.0481	8.0500
SPY190418C00274000	274.0000	0.1242	4.7392	4.7392	4.7400
SPY190418P00274000	274.0000	0.1507	8.6809	8.5486	8.5500
SPY190418C00275000	275.0000	0.1228	4.2739	4.2739	4.2700
SPY190418P00275000	275.0000	0.1481	9.1502	9.0029	9.0000

## D Trinomial Tree Implied Volatility

Option Name	Strike	Type	Binomial Implied Volatility
SPY190215C00265000	265.0000	C	0.1598
SPY190215P00265000	265.0000	P	0.1576
SPY190215C00266000	266.0000	C	0.1555
SPY190215P00266000	266.0000	P	0.1515
SPY190315C00265000	265.0000	C	0.1524
SPY190315P00265000	265.0000	P	0.1627
SPY190315P00266000	266.0000	P	0.1605
SPY190315P00267000	267.0000	P	0.1587
SPY190315P00268000	268.0000	P	0.1553
SPY190315P00269000	269.0000	P	0.1528
SPY190315P00270000	270.0000	P	0.1514
SPY190418P00265000	265.0000	P	0.1653
SPY190418P00266000	266.0000	P	0.1629
SPY190418P00267000	267.0000	P	0.1605
SPY190418P00268000	268.0000	P	0.1585
SPY190418P00269000	269.0000	P	0.1565
SPY190418P00270000	270.0000	P	0.1551
SPY190418P00271000	271.0000	P	0.1529
SPY190418P00272000	272.0000	P	0.1501



## E Solution Source Code

### E.1 Question 1 Implementation

#### E.1.1 Binomial Tree Price Computation

```

1 from context import fe621
2 from config import cfg
3
4 import pandas as pd
5
6
7 # Loading homework 2 data
8 hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)
9
10 # Container to store prices
11 computed_prices = pd.DataFrame()
12
13 # Steps for tree construction
14 steps = 200
15
16 # Flags
17 counter = 0
18
19 # Iterating through each of the options, computing tree prices
20 for idx, row in hw2_data.iterrows():
21     # Dictionary to store new row data
22     price_data = dict()
23
24     # Isolating name
25     price_data['name'] = idx
26     # Assigning black scholes price
27     price_data['bs_price'] = row['bs_price']
28
29     # Initializing tree
30     tree = fe621.tree_pricing.binomial.Trigeorgis(current=cfg.data2_price,
31                                                    strike=row['strike'],
32                                                    ttm=row['ttm'],
33                                                    rf=cfg.data2_rf,
34                                                    volatility=row['implied_vol'],
35                                                    opt_type=row['opt_type'],
36                                                    opt_style='E',
37                                                    steps=steps)
38
39     # Setting implied volatility used
40     price_data['implied_vol'] = row['implied_vol']
41     price_data['opt_type'] = row['opt_type']
42     price_data['strike'] = row['strike']
43     # Assigning European and American price
44     price_data['binomial_E'] = tree.getInstrumentValue()
45     price_data['binomial_A'] = tree.computeOtherStylePrice(opt_style='A')
46
47     # Appending new row to output DataFrame
48     computed_prices = computed_prices.append(price_data, ignore_index=True)
49
50     # Log
51     counter += 1
52     print('%f%% Complete - Binomial tree price for EU option %s is %f' % \
53           (counter / len(hw2_data.index) * 100, idx, price_data['binomial_E']))
54
55 # Setting index to option name

```

```

55 computed_prices = computed_prices.set_index('name')
56
57 # Saving to CSV
58 computed_prices.to_csv('Homework 2/bin/binomial_data2_prices.csv', index=True,
59                        float_format='%.4f')

```

question\_solutions/question\_1\_prices.py

### E.1.2 Binomial Tree Absolute Error Analysis

```

1  from context import fe621
2  from config import cfg
3
4  import matplotlib.pyplot as plt
5  import numpy as np
6  import pandas as pd
7
8
9  # Loading homework 2 data
10 hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)
11
12 # Container to store prices
13 computed_prices = pd.DataFrame()
14
15 # Steps for tree construction
16 steps = [10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400]
17
18 # Candidate Put option metadata
19 option_name = 'SPY190315P00265000'
20 strike = 265.0
21 ttm = fe621.util.getTTM(name=option_name, current_date=cfg.data2_date)
22 implied_vol = hw2_data.loc[option_name]['implied_vol']
23
24 option_bs_price = fe621.black_scholes.put(current=cfg.data2_price,
25                                           volatility=implied_vol,
26                                           ttm=ttm,
27                                           strike=strike,
28                                           rf=cfg.data2_rf)
29
30 def computeAbsError() -> np.array:
31     tree_prices = list()
32
33     # Iterate through steps
34     for step in steps:
35         # Constructing tree
36         candidate_tree = fe621.tree_pricing.binomial.Trigeorgis(
37             current=cfg.data2_price, strike=strike, ttm=ttm, rf=cfg.data2_rf,
38             volatility=implied_vol, opt_type='P', opt_style='E', steps=step
39         )
40
41         # Adding price to array for analysis
42         tree_prices += [candidate_tree.getInstrumentValue()]
43
44     # Casting to numpy array
45     tree_prices = np.array(tree_prices)
46
47     # Computing absolute error
48     abs_error = np.abs(tree_prices - option_bs_price)
49

```

```

50     # Building output dataframe
51     abs_error_df = pd.DataFrame({'Steps': steps, 'Abs Error': abs_error})\
52
53     # Saving to CSV
54     abs_error_df.to_csv(
55         'Homework 2/bin/binomial_tree_abs_error.csv', index=False)
56
57     return abs_error_df
58
59
60 def plotAbsError(steps: np.array, abs_error: np.array):
61     # Equation label
62     eq_label = r'\epsilon_N=\left|P^{\{BSM\}}(\cdot)-P^{\{BTree\}}_N(\cdot)\right|'
63
64     # Plotting points
65     plt.plot(steps, abs_error, 'x--', label=eq_label, markeredgecolor='r')
66
67     ax = plt.gca() # Getting current axes
68
69     # Setting x and y labels
70     ax.set_xlabel(r'Number of Steps, $N$')
71     ax.set_ylabel(r'Absolute Error, $\epsilon_N$')
72
73     # Setting layout to tight
74     plt.tight_layout()
75
76     # Adding plot legend
77     plt.legend(loc='upper right')
78
79     # Save to file
80     plt.savefig(fname='Homework 2/bin/binomial_abs_error_plot.png')
81
82     # Close plot
83     plt.close()
84
85 if __name__ == '__main__':
86     # Compute/load absolute error (uncomment relevant line)
87     abs_error = computeAbsError()
88     # abs_error = pd.read_csv('Homework 2/bin/binomial_tree_abs_error.csv',
89     #                         index_col=False, header=0)
90
91     # Plot graph of absolute error
92     plotAbsError(steps=abs_error['Steps'].values,
93                 abs_error=abs_error['Abs Error'].values)

```

question\_solutions/question\_1\_abs\_error.py

### E.1.3 Binomial Tree Implied Volatility Optimization

```

1 from context import fe621
2 from config import cfg
3
4 import pandas as pd
5
6
7 # Loading HW2 data
8 hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)
9
10 # Container to store implied volatilities

```

```

11 implied_vol = pd.DataFrame()
12
13 # Iterating through options
14 for idx, row in hw2_data.iterrows():
15     # Dictionary to store new row data
16     imp_vol_data = dict()
17
18     # Isolating name
19     imp_vol_data['name'] = idx
20
21     # Isolating type
22     imp_vol_data['type'] = row['opt_type']
23
24     imp_vol_data['strike'] = row['strike']
25
26     # Setting steps
27     steps = 50
28
29     # Defining function to be optimized
30     def optimFunc(x: float) -> float:
31         # Building tree
32         tree = fe621.tree_pricing.binomial.Trigeorgis(current=cfg.data2_price,
33                                                         strike=row['strike'],
34                                                         ttm=row['ttm'],
35                                                         rf=cfg.data2_rf,
36                                                         volatility=x,
37                                                         opt_type=row['opt_type'],
38                                                         opt_style='A',
39                                                         steps=steps)
40
41         return row['bs_price'] - tree.getInstrumentValue()
42
43     try:
44         imp_vol_data['binomial_vol'] = fe621.optimization.bisectionSolver(
45             f=optimFunc, a=0.0, b=0.3, tol=0.001
46         )
47     except Exception:
48         print('WARNING: No implied vol solution found for %s' % idx)
49         continue
50
51     # Appending to array
52     implied_vol = implied_vol.append(imp_vol_data, ignore_index=True)
53
54 # Setting index to option name
55 implied_vol = implied_vol.set_index('name')
56
57 # Saving to CSV
58 implied_vol.to_csv('Homework 2/bin/binomial_implied_vol.csv',
59                   float_format='%.4f')

```

question\_solutions/question\_1\_imp\_vol.py

## E.2 Question 2 Implementation

### E.2.1 Trinomial Tree Arbitrary Price

```

1 from context import fe621
2 from config import cfg
3
4 import pandas as pd
5
6 # Arbitrary option metadata
7 strike = 100
8 current = 100
9 ttm = 1
10 volatility = 0.25
11 rf = 0.06
12 dividend = 0.03
13 steps = 200
14
15 # Constructing arbitrary tree price
16 call_tree = fe621.tree_pricing.trinomial.AdditiveTree(current=current,
17                                                         strike=strike,
18                                                         ttm=ttm, rf=rf,
19                                                         volatility=volatility,
20                                                         opt_type='C',
21                                                         opt_style='E',
22                                                         dividend=dividend,
23                                                         steps=steps)
24
25 prices = pd.DataFrame()
26
27 # Call Option prices
28 prices['European Call'] = [call_tree.getInstrumentValue()]
29 prices['American Call'] = [call_tree.computeOtherStylePrice(opt_style='A')]
30
31 put_tree = fe621.tree_pricing.trinomial.AdditiveTree(current=current,
32                                                         strike=strike,
33                                                         ttm=ttm, rf=rf,
34                                                         volatility=volatility,
35                                                         opt_type='P',
36                                                         opt_style='E',
37                                                         dividend=dividend,
38                                                         steps=steps)
39
40 # Put option prices
41 prices['European Put'] = [put_tree.getInstrumentValue()]
42 prices['American Put'] = [put_tree.computeOtherStylePrice(opt_style='A')]
43
44 # Writing results to CSV
45 prices.T.to_csv('Homework 2/bin/trinomial_arbitrary_price.csv',
46                 index_label='Option Type', header=['Value'])

```

question\_solutions/question\_2\_arb\_option.py

### E.2.2 Trinomial Tree Price Computation

```

1 from context import fe621
2 from config import cfg
3
4 import pandas as pd

```

```

5
6
7 # Loading homework 2 data
8 hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)
9
10 # Container to store prices
11 computed_prices = pd.DataFrame()
12
13 # Steps for tree construction
14 steps = 200
15
16 # Flags
17 counter = 0
18
19 # Iterating through each of the options, computing tree prices
20 for idx, row in hw2_data.iterrows():
21     # Dictionary to store new row data
22     price_data = dict()
23
24     # Isolating name
25     price_data['name'] = idx
26     # Assigning black scholes price
27     price_data['bs_price'] = row['bs_price']
28
29     # Initializing tree
30     tree = fe621.tree_pricing.trinomial.AdditiveTree(
31         current=cfg.data2_price,
32         strike=row['strike'],
33         ttm=row['ttm'],
34         rf=cfg.data2_rf,
35         volatility=row['implied_vol'],
36         opt_type=row['opt_type'],
37         opt_style='E',
38         steps=steps
39     )
40
41     # Setting implied volatility used
42     price_data['implied_vol'] = row['implied_vol']
43     price_data['opt_type'] = row['opt_type']
44     price_data['strike'] = row['strike']
45     # Assigning European and American price
46     price_data['trinomial_E'] = tree.getInstrumentValue()
47     price_data['trinomial_A'] = tree.computeOtherStylePrice(opt_style='A')
48
49     # Appending new row to output DataFrame
50     computed_prices = computed_prices.append(price_data, ignore_index=True)
51
52     # Log
53     counter += 1
54     print('%f%% Complete - Trinomial tree price for EU option %s is %f' % \
55           (counter / len(hw2_data.index) * 100, idx, price_data['trinomial_E']))
56
57 # Setting index to option name
58 computed_prices = computed_prices.set_index('name')
59
60 # Saving to CSV
61 computed_prices.to_csv('Homework 2/bin/trinomial_data2_prices.csv', index=True,
62                       float_format='%%.4f')

```

question\_solutions/question.2\_prices.py

## E.2.3 Trinomial Tree Absolute Error Analysis

```

1 from context import fe621
2 from config import cfg
3
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import pandas as pd
7
8
9 # Loading homework 2 data
10 hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)
11
12 # Container to store prices
13 computed_prices = pd.DataFrame()
14
15 # Steps for tree construction
16 steps = [10, 20, 30, 40, 50, 100, 150, 200, 250, 300, 350, 400]
17
18 # Candidate Put option metadata
19 option_name = 'SPY190315P00265000'
20 strike = 265.0
21 ttm = fe621.util.getTTM(name=option_name, current_date=cfg.data2_date)
22 implied_vol = hw2_data.loc[option_name]['implied_vol']
23
24 option_bs_price = fe621.black_scholes.put(current=cfg.data2_price,
25                                           volatility=implied_vol,
26                                           ttm=ttm,
27                                           strike=strike,
28                                           rf=cfg.data2_rf)
29
30 def computeAbsError() -> np.array:
31     tree_prices = list()
32
33     # Iterate through steps
34     for step in steps:
35         # Constructing tree
36         candidate_tree = fe621.tree_pricing.trinomial.AdditiveTree(
37             current=cfg.data2_price, strike=strike, ttm=ttm, rf=cfg.data2_rf,
38             volatility=implied_vol, opt_type='P', opt_style='E', steps=step
39         )
40
41         # Adding price to array for analysis
42         tree_prices += [candidate_tree.getInstrumentValue()]
43
44     # Casting to numpy array
45     tree_prices = np.array(tree_prices)
46
47     # Computing absolute error
48     abs_error = np.abs(tree_prices - option_bs_price)
49
50     # Building output dataframe
51     abs_error_df = pd.DataFrame({'Steps': steps, 'Abs Error': abs_error})\
52
53     # Saving to CSV
54     abs_error_df.to_csv(
55         'Homework 2/bin/trinomial_tree_abs_error.csv', index=False)
56
57     return abs_error_df
58
59

```

```

60 def plotAbsError(steps: np.array, abs_error: np.array):
61     # Equation label
62     eq_label = r'$\epsilon_N = \left| P^{\{BSM\}}(\cdot) - P^{\{TTree\}}_N(\cdot) \right| $'
63
64     # Plotting points
65     plt.plot(steps, abs_error, 'x--', label=eq_label, markeredgecolor='r')
66
67     ax = plt.gca() # Getting current axes
68
69     # Setting x and y labels
70     ax.set_xlabel(r'Number of Steps, $N$')
71     ax.set_ylabel(r'Absolute Error, $\epsilon_N$')
72
73     # Setting layout to tight
74     plt.tight_layout()
75
76     # Adding plot legend
77     plt.legend(loc='upper right')
78
79     # Save to file
80     plt.savefig(fname='Homework 2/bin/trinomial_abs_error_plot.png')
81
82     # Close plot
83     plt.close()
84
85 if __name__ == '__main__':
86     # Compute/load absolute error (uncomment relevant line)
87     abs_error = computeAbsError()
88     # abs_error = pd.read_csv('Homework 2/bin/trinomial_tree_abs_error.csv',
89     #                          index_col=False, header=0)
90
91     # Plot graph of absolute error
92     plotAbsError(steps=abs_error['Steps'].values,
93                  abs_error=abs_error['Abs Error'].values)

```

question\_solutions/question\_2\_abs\_error.py

## E.2.4 Trinomial Tree Implied Volatility Optimization

```

1 from context import fe621
2 from config import cfg
3
4 import pandas as pd
5
6
7 # Loading HW2 data
8 hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)
9
10 # Container to store implied volatilities
11 implied_vol = pd.DataFrame()
12
13 # Iterating through options
14 for idx, row in hw2_data.iterrows():
15     # Dictionary to store new row data
16     imp_vol_data = dict()
17
18     # Isolating name
19     imp_vol_data['name'] = idx
20

```



```
21 # Isolating type
22 imp_vol_data['type'] = row['opt_type']
23
24 imp_vol_data['strike'] = row['strike']
25
26 # Setting steps
27 steps = 50
28
29 # Defining function to be optimized
30 def optimFunc(x: float) -> float:
31     # Building tree
32     tree = fe621.tree_pricing.trinomial.AdditiveTree(
33         current=cfg.data2_price,
34         strike=row['strike'],
35         ttm=row['ttm'],
36         rf=cfg.data2_rf,
37         volatility=x,
38         opt_type=row['opt_type'],
39         opt_style='A',
40         steps=steps
41     )
42
43     return row['bs_price'] - tree.getInstrumentValue()
44
45 try:
46     imp_vol_data['trinomial_vol'] = fe621.optimization.bisectionSolver(
47         f=optimFunc, a=0.0, b=0.3, tol=0.001
48     )
49 except Exception:
50     print('WARNING: No implied vol solution found for %s' % idx)
51     continue
52
53 # Appending to array
54 implied_vol = implied_vol.append(imp_vol_data, ignore_index=True)
55
56 # Setting index to option name
57 implied_vol = implied_vol.set_index('name')
58
59 # Saving to CSV
60 implied_vol.to_csv('Homework 2/bin/trinomial_implied_vol.csv',
61                   float_format='%.4f')
```

question\_solutions/question\_2\_imp\_vol.py

## E.3 Question 3 Implementation

### E.3.1 Barrier EU Call Option Utilities

```

1 from ..call import blackScholesCall
2 from ..put import blackScholesPut
3 from ..util import computeD1D2
4
5 import numpy as np
6
7
8 class AnalyticalUtil():
9     """Helper class for analytical barrier option pricing.
10
11     See chapter 5 in http://bit.ly/2JHoVbQ for more.
12     """
13
14     def __init__(self, volatility: float, ttm: float, rf: float,
15                  dividend: float=0):
16         self.volatility = volatility
17         self.ttm = ttm
18         self.rf = rf
19         self.dividend = dividend
20         self.nu = self.rf - self.dividend - (np.power(self.volatility, 2) / 2)
21
22     def cBS(self, current: float, strike: float) -> float:
23         return blackScholesCall(current=current,
24                                  volatility=self.volatility,
25                                  ttm=self.ttm,
26                                  strike=strike,
27                                  rf=self.rf)
28
29     def pBS(self, current: float, strike: float) -> float:
30         return blackScholesPut(current=current,
31                                 volatility=self.volatility,
32                                 ttm=self.ttm,
33                                 strike=strike,
34                                 rf=self.rf)
35
36     def dBS(self, current: float, strike: float) -> float:
37         return (np.log(current / strike) + (self.nu * self.ttm)) /\
38             (self.volatility * np.sqrt(self.ttm))

```

../fe621/black\_scholes/barrier/util.py

### E.3.2 Complete Solution Implementation

```

1 from context import fe621
2
3 import pandas as pd
4
5
6 # Option parameters
7 current = 10
8 strike = 10
9 ttm = 0.3
10 volatility = 0.2
11 rf = 0.01
12 H = 11

```

```

13 steps = 200
14
15 q3_answers = pd.DataFrame()
16
17 # Part (a)
18 barrier_tree = fe621.tree_pricing.binomial.Barrier(
19     current=current,
20     strike=strike,
21     ttm=ttm,
22     rf=rf,
23     volatility=volatility,
24     barrier=H,
25     barrier_type='O',
26     opt_type='C',
27     opt_style='E',
28     steps=steps
29 )
30
31 q3_answers['Tree Up-and-Out EU Call'] = [barrier_tree.getInstrumentValue()]
32
33
34 # Part (b)
35 analytical_upandout = fe621.black_scholes.barrier.callUpAndOut(
36     S=current,
37     H=H,
38     volatility=volatility,
39     ttm=ttm,
40     K=strike,
41     rf=rf
42 )
43
44 q3_answers['Analytical Up-and-Out EU Call'] = [analytical_upandout]
45
46
47 # Part (c)
48 barrier_tree = fe621.tree_pricing.binomial.Barrier(
49     current=current,
50     strike=strike,
51     ttm=ttm,
52     rf=rf,
53     volatility=volatility,
54     barrier=H,
55     barrier_type='I',
56     opt_type='C',
57     opt_style='E',
58     steps=steps
59 )
60
61 q3_answers['Tree Up-and-In EU Call'] = [barrier_tree.getInstrumentValue()]
62
63 analytical_upandin = fe621.black_scholes.barrier.callUpAndIn(
64     S=current,
65     H=H,
66     volatility=volatility,
67     ttm=ttm,
68     K=strike,
69     rf=rf
70 )
71
72 q3_answers['Analytical Up-and-In EU Call'] = [analytical_upandin]
73

```

```
74
75 # Part (d)
76 barrier_tree = fe621.tree_pricing.binomial.Barrier(
77     current=current,
78     strike=strike,
79     ttm=ttm,
80     rf=rf,
81     volatility=volatility,
82     barrier=H,
83     barrier_type='I',
84     opt_type='C',
85     opt_style='A',
86     steps=steps
87 )
88
89 q3_answers['Tree Up-and-In A Call'] = [barrier_tree.getInstrumentValue()]
90
91
92 # Writing results to CSV
93 q3_answers.T.to_csv('Homework 2/bin/barrier_option_values.csv',
94                     index_label='Type', header=['Value'])
```

question\_solutions/question\_3.py