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[1] *for full source code of the* `fe621` *Python package.*

1. Weerawarana 2019

Contents

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.

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


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

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

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

    This generalized tree computation methodology allows this class to be used
    as a base for any arbitrary tree pricing or analysis tool, including
    multiplicative and additive trees. Tree values are strategically exposed at
    runtime when building and traversing the tree for added flexibility. Details
    of specific exposed runtime variables are discussed further in the
    specific function docstrings.

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

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

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

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

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

```

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

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

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

        self.price_tree_root = price_tree_root
        self.steps = steps

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

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

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

        # Construct value tree (check that price tree is constructed first)
        if build_value_tree:
            try:
                self.price_tree
            except NameError:
                raise RuntimeError('Price tree not constructed yet.')

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

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

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

        See documentation for `GeneralTree._constructValueTree` for more.

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

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

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

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

        raise NotImplementedError

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

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

        See documentation for `GeneralTree._constructValueTree` for more.

        Raises:
            NotImplementedError -- Raised when not implemented.

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

        raise NotImplementedError

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

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

        See documentation for `GeneralTree._constructPriceTree` for more.

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

        Raises:
            NotImplementedError -- Raised when not implemented.

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

        raise NotImplementedError

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

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

```

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

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

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

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

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

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

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

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

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

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


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

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

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

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

        Specifically, the following variables are static and set once:
            `self.nrow` -- Number of rows of the price tree matrix.
```

```
            `self.ncolumn` -- Number of columns of the price tree matrix.
            `self.mid_row_index` -- Index of the middle row of the matrix.

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

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

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

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

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

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

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

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

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

        # Return final price tree
        return price_tree

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

```

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

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

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

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

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

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

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

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

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

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

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

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

```
                self._current_col = j

                # Update children indexes
                self.__updateChildIndexes()

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

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

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

        # Return final value tree
        return value_tree

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

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

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

../fe621/tree_pricing/general_tree.py

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.

```
from ..general_tree import GeneralTree

import numpy as np


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

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

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

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

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

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

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

        # Computing deltaT
        deltaT = ttm / steps

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

```
                                  2) * deltaT))
        # Down deltaX = -1 * upDeltaX
        self.deltaXD = -1 * self.deltaXU

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

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

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

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

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

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

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

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

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

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

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

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

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

        return child_implied_value
```

```python

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

        # Setting new option style
        self.opt_style = opt_style

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

        return self.getInstrumentValue()
```

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

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.

```python
from ..general_tree import GeneralTree

import numpy as np


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

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

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

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

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

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

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

        # Computing deltaT
        deltaT = ttm / steps

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

```
        self.deltaXD = -1 * self.deltaXU

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

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

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

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

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

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

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

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

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

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

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

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

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

```python

        return child_implied_value

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

        # Setting new option style
        self.opt_style = opt_style

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

        return self.getInstrumentValue()
```

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

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, $N \to \infty$.

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.

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

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

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.

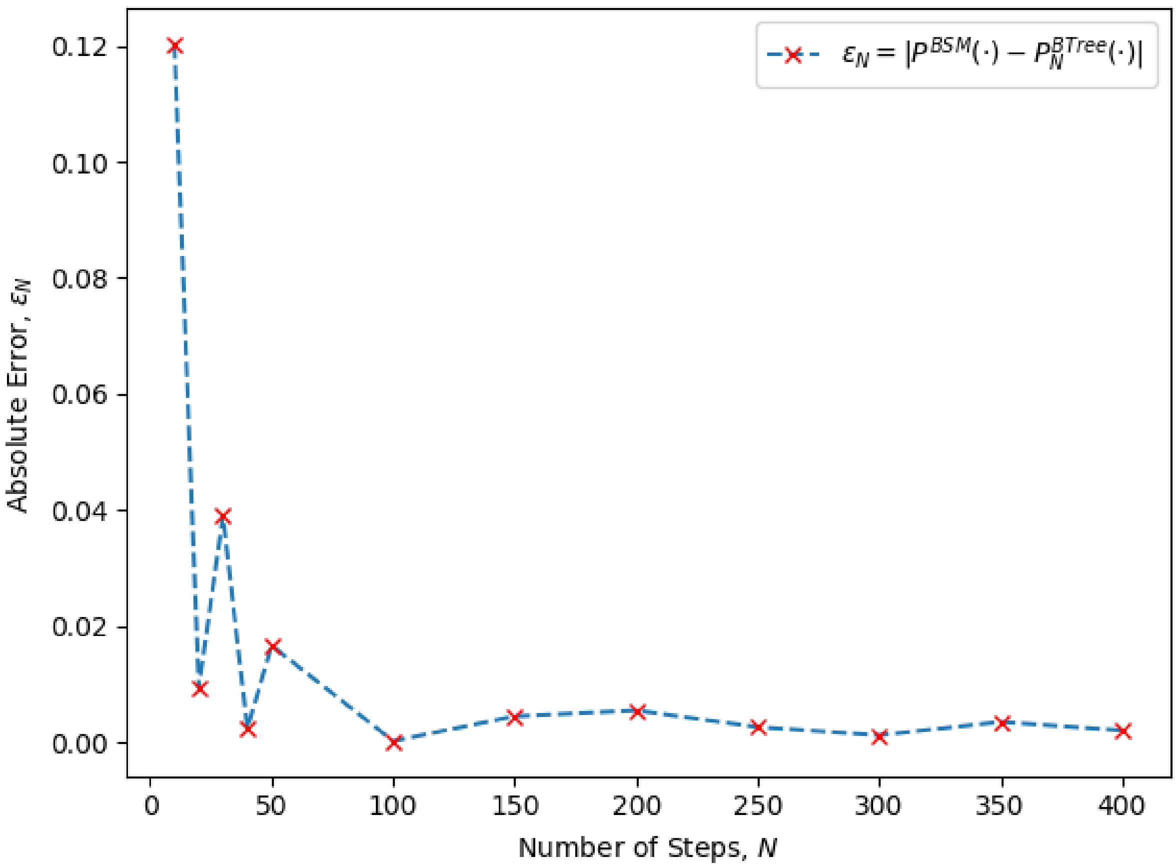

Figure 1: Absolute error analysis of the Binomial Tree price computation convergence, with respect to varying step sizes.

As evidenced by the graphic, there is a clear pattern of convergence to a the analytically computed value as the number of steps, N increases. This is to be expected, as at the limit as $N \rightarrow \infty$, 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, $N \to \infty$.

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.

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

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

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.

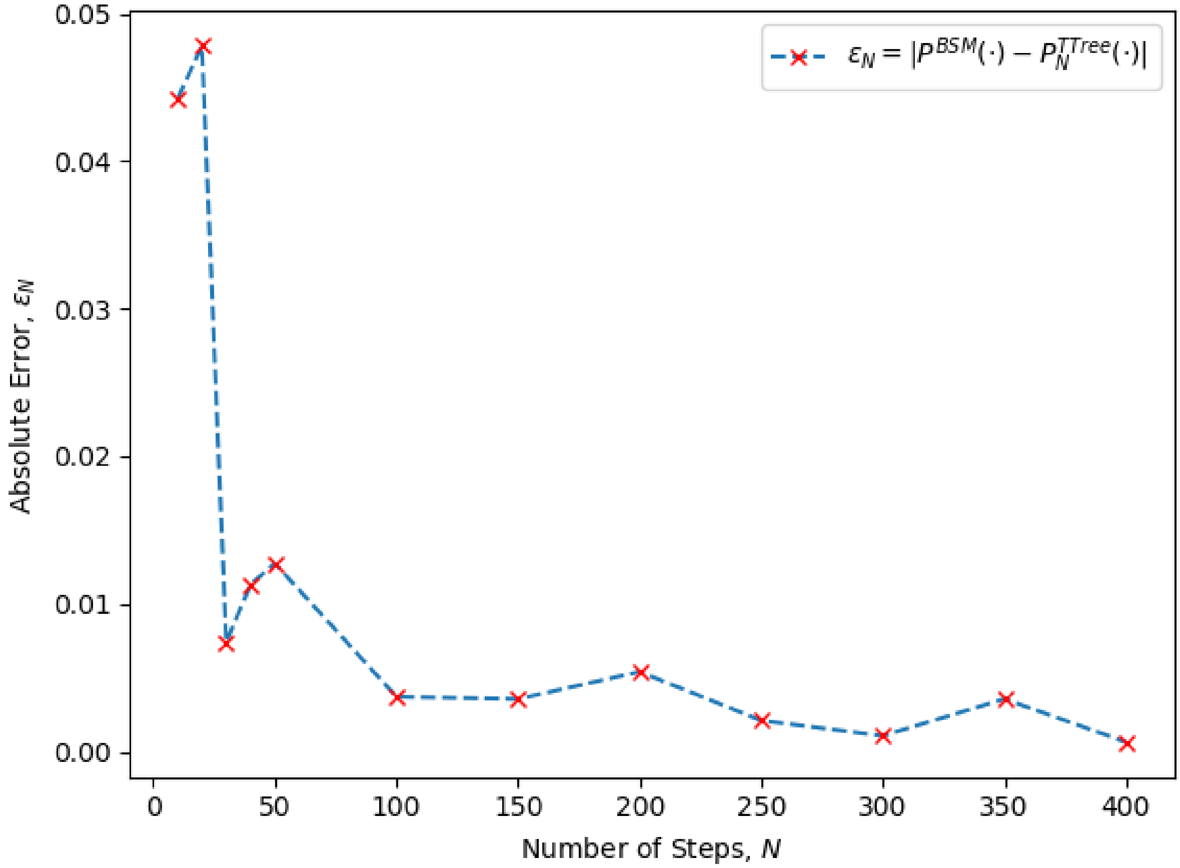

Figure 2: Absolute error analysis of the Trinomial Tree price computation convergence, with respect to varying step sizes.

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 $N \to \infty$, (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.

```
from ..general_tree import GeneralTree
from .trigeorgis import Trigeorgis

import numpy as np


class Barrier(GeneralTree):
    """Barrier option pricing with a Trigeorgis tree.

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

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

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

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

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

        # Ensuring valid barrier option type
        if barrier_type not in ['I', 'O']:
            raise ValueError('`barrier_type` must be \'I\' for In type options,\
                or \'O\' for Out type options.')

        # Inferring barrier option characteristic from current price
        # i.e. either Up or Down
        if barrier > current:
            self.barrier_characteristic = 'U'
        else:
            self.barrier_characteristic = 'D'

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

```
        self.barrier_log = np.log(barrier)

        # Override barrier type if option style is European
        if opt_style == 'E':
            self.barrier_type = 'O'  # Necessary for this to work
        else:
            self.barrier_type = barrier_type

        # Computing deltaT
        deltaT = ttm / steps

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

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

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

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

        if (barrier_type == 'I'):
            # Special case for 'In' barrier type
            vanilla_tree = Trigeorgis(current=current, strike=strike, ttm=ttm,
                rf=rf, volatility=volatility, opt_type=opt_type,
                opt_style=opt_style, steps=steps)
            self.value_tree = (vanilla_tree.value_tree - self.value_tree)\
                .todok(copy=True)

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

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

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

        # Computing barrier indicator for each of the child prices
        up_child_price = up_child_price \
            * self.barrierIndicator(np.exp(up_child_price))
        down_child_price = down_child_price \
            * self.barrierIndicator(np.exp(down_child_price))

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

```

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

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

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

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

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

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

        return child_implied_value

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

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

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

        # Computing prices of each of the values
        underlying_prices = np.array([0 if i == 0 else np.exp(i)
                                      for i in last_col])

        # Computing barrier indicator function values for last_col values
        indicator_val = np.array([[self.barrierIndicator(i)]
                                  for i in underlying_prices])

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

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

            # Replacing values equal to (self.strike - 1) with 0. This is to
```

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

        # Floor to 0
        floor_val = np.where(non_floor_val > 0, non_floor_val, 0)

        # Return element-wise product of floor_val and barrier indicator
        return np.multiply(indicator_val, floor_val)

    def barrierIndicator(self, underlying_price: float) -> int:
        """Indicator for the barrier option.

        Returns the corresponding value for the barrier indicator function,
        depending on the barrier type and barrier characteristic.

        Arguments:
            underlying_price {float} -- Log price of the underlying asset.

        Returns:
            int -- 1 or 0 depending on indicator function.
        """

        # Down and out
        if (self.barrier_characteristic == 'D') and (self.barrier_type == 'O'):
            return 1 if underlying_price > self.barrier else 0

        # Down and in
        if (self.barrier_characteristic == 'D') and (self.barrier_type == 'I'):
            return 1 if underlying_price <= self.barrier else 0

        # Up and out
        if (self.barrier_characteristic == 'U') and (self.barrier_type == 'O'):
            return 1 if underlying_price < self.barrier else 0

        # Up and in
        if (self.barrier_characteristic == 'U') and (self.barrier_type == 'I'):
            return 1 if underlying_price >= self.barrier else 0

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

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

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

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

../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.

```python
from .util import AnalyticalUtil

from scipy.stats import norm
import numpy as np


def callUpAndIn(S: float, H: float, volatility: float, ttm: float,
                K: float, rf: float, dividend: float=0) -> float:
    """Analytical formula to compute the value of an up and in Barrier option.

    See formula 5.1 in http://bit.ly/2JHoVbQ for more.

    Arguments:
        S {float} -- Current price.
        H {float} -- Barrier price.
        volatility {float} -- Volatility of the underlying.
        ttm {float} -- Time to maturity (in years).
        K {float} -- Strike price.
        rf {float} -- Risk-free rate (annualized).

    Keyword Arguments:
        dividend {float} -- Dividend yield (default: {0}).

    Returns:
        float -- Analytical value of up and in call option.
    """

    util = AnalyticalUtil(volatility=volatility, ttm=ttm, rf=rf,
                          dividend=dividend)

    return np.power((H / S), 2 * util.nu / np.power(volatility, 2)) *\
        (util.pBS(np.power(H, 2) / S, K) - util.pBS(np.power(H, 2) / S, H) +\
        ((H - K) * np.exp(-1 * rf * ttm) * norm.cdf(-1 * util.dBS(H, S)))) +\
        util.cBS(S, H) + ((H - K) * np.exp(-1 * rf * ttm) * norm.cdf(
        util.dBS(S, H)))


def callUpAndOut(S: float, H: float, volatility: float, ttm: float, K: float,
                 rf: float, dividend: float=0) -> float:
    """Analytical formula to compute the value of an up and out barrier call
    option.

    See formula 5.2 in http://bit.ly/2JHoVbQ for more.

    Arguments:
        S {float} -- Current price.
        H {float} -- Barrier price.
        volatility {float} -- Volatility of the underlying.
        ttm {float} -- Time tp maturity (in years).
        K {float} -- Strike price.
        rf {float} -- Risk-free rate (annualized).

    Keyword Arguments:
        dividend {float} -- Dividend yield (default: {0}).

    Returns:
```

```
        float -- Analytical price of up and out barrier call option.
    """

    util = AnalyticalUtil(volatility=volatility, ttm=ttm, rf=rf,
                          dividend=dividend)

    return util.cBS(S, K) - util.cBS(S, H) - ((H - K) * np.exp(-1 * rf * ttm)\
        * norm.cdf(util.dBS(S, H))) - (np.power(H / S, 2 * util.nu / np.power(
        volatility, 2)) * (util.cBS(np.power(H, 2) / S, K) - util.cBS(np.power(
        H, 2) / S, H) - ((H - K) * np.exp(-1 * rf * ttm)) * norm.cdf(util.dBS(
        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:

$$
\begin{aligned}
S_0 &= 10 \\
K &= 10 \\
T &= 0.3 \\
\sigma &= 0.2 \\
r &= 0.01 \\
H &= 11 \\
N &= 200
\end{aligned}
$$

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

```
from context import fe621
from config import cfg

import pandas as pd


# Loading homework 2 data
hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)

# Container to store prices
computed_prices = pd.DataFrame()

# Steps for tree construction
steps = 200

# Flags
counter = 0

# Iterating through each of the options, computing tree prices
for idx, row in hw2_data.iterrows():
    # Dictionary to store new row data
    price_data = dict()

    # Isolating name
    price_data['name'] = idx
    # Assigning black scholes price
    price_data['bs_price'] = row['bs_price']

    # Initializing tree
    tree = fe621.tree_pricing.binomial.Trigeorgis(current=cfg.data2_price,
                                                  strike=row['strike'],
                                                  ttm=row['ttm'],
                                                  rf=cfg.data2_rf,
                                                  volatility=row['implied_vol'],
                                                  opt_type=row['opt_type'],
                                                  opt_style='E',
                                                  steps=steps)
    # Setting implied volatility used
    price_data['implied_vol'] = row['implied_vol']
    price_data['opt_type'] = row['opt_type']
    price_data['strike'] = row['strike']
    # Assigning European and American price
    price_data['binomial_E'] = tree.getInstrumentValue()
    price_data['binomial_A'] = tree.computeOtherStylePrice(opt_style='A')

    # Appending new row to output DataFrame
    computed_prices = computed_prices.append(price_data, ignore_index=True)

    # Log
    counter += 1
    print('%f%% Complete - Binomial tree price for EU option %s is %f' % \
          (counter / len(hw2_data.index) * 100, idx, price_data['binomial_E']))

# Setting index to option name
```

```
computed_prices = computed_prices.set_index('name')

# Saving to CSV
computed_prices.to_csv('Homework 2/bin/binomial_data2_prices.csv', index=True,
                       float_format='%.4f')
```

question_solutions/question_1_prices.py

E.1.2 Binomial Tree Absolute Error Analysis

```
from context import fe621
from config import cfg

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


# Loading homework 2 data
hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)

# Container to store prices
computed_prices = pd.DataFrame()

# Steps for tree construction
steps = [10, 20, 30, 40, 50,100, 150, 200, 250, 300, 350, 400]

# Candidate Put option metadata
option_name = 'SPY190315P00265000'
strike = 265.0
ttm = fe621.util.getTTM(name=option_name, current_date=cfg.data2_date)
implied_vol = hw2_data.loc[option_name]['implied_vol']

option_bs_price = fe621.black_scholes.put(current=cfg.data2_price,
                                          volatility=implied_vol,
                                          ttm=ttm,
                                          strike=strike,
                                          rf=cfg.data2_rf)

def computeAbsError() -> np.array:
    tree_prices = list()

    # Iterate through steps
    for step in steps:
        # Constructing tree
        candidate_tree = fe621.tree_pricing.binomial.Trigeorgis(
            current=cfg.data2_price, strike=strike, ttm=ttm, rf=cfg.data2_rf,
            volatility=implied_vol, opt_type='P', opt_style='E', steps=step
        )

        # Adding price to array for analysis
        tree_prices += [candidate_tree.getInstrumentValue()]

    # Casting to numpy array
    tree_prices = np.array(tree_prices)

    # Computing absolute error
    abs_error = np.abs(tree_prices - option_bs_price)

```

```
    # Building output dataframe
    abs_error_df = pd.DataFrame({'Steps': steps, 'Abs Error': abs_error})\

    # Saving to CSV
    abs_error_df.to_csv(
        'Homework 2/bin/binomial_tree_abs_error.csv', index=False)

    return abs_error_df


def plotAbsError(steps: np.array, abs_error: np.array):
    # Equation label
    eq_label = r'$\epsilon_N=\left|P^{BSM}(\cdot)-P^{BTree}_N(\cdot)\right|$'

    # Plotting points
    plt.plot(steps, abs_error, 'x--', label=eq_label, markeredgecolor='r')

    ax = plt.gca()  # Getting current axes

    # Setting x and y labels
    ax.set_xlabel(r'Number of Steps, $N$')
    ax.set_ylabel(r'Absolute Error, $\epsilon_N$')

    # Setting layout to tight
    plt.tight_layout()

    # Adding plot legend
    plt.legend(loc='upper right')

    # Save to file
    plt.savefig(fname='Homework 2/bin/binomial_abs_error_plot.png')

    # Clsoe plot
    plt.close()

if __name__ == '__main__':
    # Compute/load absolute error (uncomment relevant line)
    abs_error = computeAbsError()
    # abs_error = pd.read_csv('Homework 2/bin/binomial_tree_abs_error.csv',
    #                         index_col=False, header=0)

    # Plot graph of absolute error
    plotAbsError(steps=abs_error['Steps'].values,
                 abs_error=abs_error['Abs Error'].values)
```

question_solutions/question_1_abs_error.py

E.1.3 Binomial Tree Implied Volatility Optimization

```
from context import fe621
from config import cfg

import pandas as pd


# Loading HW2 data
hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)

# Container to store implied volatilities
```

```
implied_vol = pd.DataFrame()

# Iterating through options
for idx, row in hw2_data.iterrows():
    # Dictionary to store new row data
    imp_vol_data = dict()

    # Isolating name
    imp_vol_data['name'] = idx

    # Isolating type
    imp_vol_data['type'] = row['opt_type']

    imp_vol_data['strike'] = row['strike']

    # Setting steps
    steps = 50

    # Defining function to be optimized
    def optimFunc(x: float) -> float:
        # Building tree
        tree = fe621.tree_pricing.binomial.Trigeorgis(current=cfg.data2_price,
                                                      strike=row['strike'],
                                                      ttm=row['ttm'],
                                                      rf=cfg.data2_rf,
                                                      volatility=x,
                                                      opt_type=row['opt_type'],
                                                      opt_style='A',
                                                      steps=steps)

        return row['bs_price'] - tree.getInstrumentValue()

    try:
        imp_vol_data['binomial_vol'] = fe621.optimization.bisectionSolver(
            f=optimFunc, a=0.0, b=0.3, tol=0.001
        )
    except Exception:
        print('WARNING: No implied vol solution found for %s' % idx)
        continue

    # Appending to array
    implied_vol = implied_vol.append(imp_vol_data, ignore_index=True)

# Setting index to option name
implied_vol = implied_vol.set_index('name')

# Saving to CSV
implied_vol.to_csv('Homework 2/bin/binomial_implied_vol.csv',
                   float_format='%.4f')
```

question_solutions/question_1_imp_vol.py

E.2 Question 2 Implementation

E.2.1 Trinomial Tree Arbitrary Price

```
from context import fe621
from config import cfg

import pandas as pd

# Arbitrary option metadata
strike = 100
current = 100
ttm = 1
volatility = 0.25
rf = 0.06
dividend = 0.03
steps = 200

# Constructing arbitrary tree price
call_tree = fe621.tree_pricing.trinomial.AdditiveTree(current=current,
                                                      strike=strike,
                                                      ttm=ttm, rf=rf,
                                                      volatility=volatility,
                                                      opt_type='C',
                                                      opt_style='E',
                                                      dividend=dividend,
                                                      steps=steps)

prices = pd.DataFrame()

# Call Option prices
prices['European Call'] = [call_tree.getInstrumentValue()]
prices['American Call'] = [call_tree.computeOtherStylePrice(opt_style='A')]

put_tree = fe621.tree_pricing.trinomial.AdditiveTree(current=current,
                                                     strike=strike,
                                                     ttm=ttm, rf=rf,
                                                     volatility=volatility,
                                                     opt_type='P',
                                                     opt_style='E',
                                                     dividend=dividend,
                                                     steps=steps)

# Put option prices
prices['European Put'] = [put_tree.getInstrumentValue()]
prices['American Put'] = [put_tree.computeOtherStylePrice(opt_style='A')]

# Writing results to CSV
prices.T.to_csv('Homework 2/bin/trinomial_arbitrary_price.csv',
                index_label='Option Type', header=['Value'])
```

question_solutions/question_2_arb_option.py

E.2.2 Trinomial Tree Price Computation

```
from context import fe621
from config import cfg

import pandas as pd
```

```python


# Loading homework 2 data
hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)

# Container to store prices
computed_prices = pd.DataFrame()

# Steps for tree construction
steps = 200

# Flags
counter = 0

# Iterating through each of the options, computing tree prices
for idx, row in hw2_data.iterrows():
    # Dictionary to store new row data
    price_data = dict()

    # Isolating name
    price_data['name'] = idx
    # Assigning black scholes price
    price_data['bs_price'] = row['bs_price']

    # Initializing tree
    tree = fe621.tree_pricing.trinomial.AdditiveTree(
        current=cfg.data2_price,
        strike=row['strike'],
        ttm=row['ttm'],
        rf=cfg.data2_rf,
        volatility=row['implied_vol'],
        opt_type=row['opt_type'],
        opt_style='E',
        steps=steps
    )

    # Setting implied volatility used
    price_data['implied_vol'] = row['implied_vol']
    price_data['opt_type'] = row['opt_type']
    price_data['strike'] = row['strike']
    # Assigning European and American price
    price_data['trinomial_E'] = tree.getInstrumentValue()
    price_data['trinomial_A'] = tree.computeOtherStylePrice(opt_style='A')

    # Appending new row to output DataFrame
    computed_prices = computed_prices.append(price_data, ignore_index=True)

    # Log
    counter += 1
    print('%f%% Complete - Trinomial tree price for EU option %s is %f' % \
          (counter / len(hw2_data.index) * 100, idx, price_data['trinomial_E']))

# Setting index to option name
computed_prices = computed_prices.set_index('name')

# Saving to CSV
computed_prices.to_csv('Homework 2/bin/trinomial_data2_prices.csv', index=True,
                       float_format='%.4f')
```

question_solutions/question_2_prices.py

E.2.3 Trinomial Tree Absolute Error Analysis

```
from context import fe621
from config import cfg

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


# Loading homework 2 data
hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)

# Container to store prices
computed_prices = pd.DataFrame()

# Steps for tree construction
steps = [10, 20, 30, 40, 50,100, 150, 200, 250, 300, 350, 400]

# Candidate Put option metadata
option_name = 'SPY190315P00265000'
strike = 265.0
ttm = fe621.util.getTTM(name=option_name, current_date=cfg.data2_date)
implied_vol = hw2_data.loc[option_name]['implied_vol']

option_bs_price = fe621.black_scholes.put(current=cfg.data2_price,
                                          volatility=implied_vol,
                                          ttm=ttm,
                                          strike=strike,
                                          rf=cfg.data2_rf)

def computeAbsError() -> np.array:
    tree_prices = list()

    # Iterate through steps
    for step in steps:
        # Constructing tree
        candidate_tree = fe621.tree_pricing.trinomial.AdditiveTree(
            current=cfg.data2_price, strike=strike, ttm=ttm, rf=cfg.data2_rf,
            volatility=implied_vol, opt_type='P', opt_style='E', steps=step
        )

        # Adding price to array for analysis
        tree_prices += [candidate_tree.getInstrumentValue()]

    # Casting to numpy array
    tree_prices = np.array(tree_prices)

    # Computing absolute error
    abs_error = np.abs(tree_prices - option_bs_price)

    # Building output dataframe
    abs_error_df = pd.DataFrame({'Steps': steps, 'Abs Error': abs_error})\

    # Saving to CSV
    abs_error_df.to_csv(
        'Homework 2/bin/trinomial_tree_abs_error.csv', index=False)

    return abs_error_df


```

```python
def plotAbsError(steps: np.array, abs_error: np.array):
    # Equation label
    eq_label = r'$\epsilon_N=\left|P^{BSM}(\cdot)-P^{TTree}_N(\cdot)\right|$'

    # Plotting points
    plt.plot(steps, abs_error, 'x--', label=eq_label, markeredgecolor='r')

    ax = plt.gca()  # Getting current axes

    # Setting x and y labels
    ax.set_xlabel(r'Number of Steps, $N$')
    ax.set_ylabel(r'Absolute Error, $\epsilon_N$')

    # Setting layout to tight
    plt.tight_layout()

    # Adding plot legend
    plt.legend(loc='upper right')

    # Save to file
    plt.savefig(fname='Homework 2/bin/trinomial_abs_error_plot.png')

    # Clsoe plot
    plt.close()

if __name__ == '__main__':
    # Compute/load absolute error (uncomment relevant line)
    abs_error = computeAbsError()
    # abs_error = pd.read_csv('Homework 2/bin/trinomial_tree_abs_error.csv',
    #                         index_col=False, header=0)

    # Plot graph of absolute error
    plotAbsError(steps=abs_error['Steps'].values,
                 abs_error=abs_error['Abs Error'].values)
```

question_solutions/question_2_abs_error.py

E.2.4 Trinomial Tree Implied Volatility Optimization

```python
from context import fe621
from config import cfg

import pandas as pd


# Loading HW2 data
hw2_data = pd.read_csv('Homework 2/hw2_data2.csv', index_col=0)

# Container to store implied volatilities
implied_vol = pd.DataFrame()

# Iterating through options
for idx, row in hw2_data.iterrows():
    # Dictionary to store new row data
    imp_vol_data = dict()

    # Isolating name
    imp_vol_data['name'] = idx

```

```
    # Isolating type
    imp_vol_data['type'] = row['opt_type']

    imp_vol_data['strike'] = row['strike']

    # Setting steps
    steps = 50

    # Defining function to be optimized
    def optimFunc(x: float) -> float:
        # Building tree
        tree = fe621.tree_pricing.trinomial.AdditiveTree(
            current=cfg.data2_price,
            strike=row['strike'],
            ttm=row['ttm'],
            rf=cfg.data2_rf,
            volatility=x,
            opt_type=row['opt_type'],
            opt_style='A',
            steps=steps
        )

        return row['bs_price'] - tree.getInstrumentValue()

    try:
        imp_vol_data['trinomial_vol'] = fe621.optimization.bisectionSolver(
            f=optimFunc, a=0.0, b=0.3, tol=0.001
        )
    except Exception:
        print('WARNING: No implied vol solution found for %s' % idx)
        continue

    # Appending to array
    implied_vol = implied_vol.append(imp_vol_data, ignore_index=True)

# Setting index to option name
implied_vol = implied_vol.set_index('name')

# Saving to CSV
implied_vol.to_csv('Homework 2/bin/trinomial_implied_vol.csv',
                   float_format='%.4f')
```

question_solutions/question_2_imp_vol.py

E.3 Question 3 Implementation

E.3.1 Barrier EU Call Option Utilities

```
from ..call import blackScholesCall
from ..put import blackScholesPut
from ..util import computeD1D2

import numpy as np


class AnalyticalUtil():
    """Helper class for analytical barrier option pricing.

    See chapter 5 in http://bit.ly/2JHoVbQ for more.
    """

    def __init__(self, volatility: float, ttm: float, rf: float,
        dividend: float=0):
        self.volatility = volatility
        self.ttm = ttm
        self.rf = rf
        self.dividend = dividend
        self.nu = self.rf - self.dividend - (np.power(self.volatility, 2) / 2)

    def cBS(self, current: float, strike: float) -> float:
        return blackScholesCall(current=current,
                                volatility=self.volatility,
                                ttm=self.ttm,
                                strike=strike,
                                rf=self.rf)

    def pBS(self, current: float, strike: float) -> float:
        return blackScholesPut(current=current,
                                volatility=self.volatility,
                                ttm=self.ttm,
                                strike=strike,
                                rf=self.rf)

    def dBS(self, current: float, strike: float) -> float:
        return (np.log(current / strike) + (self.nu * self.ttm)) /\
            (self.volatility * np.sqrt(self.ttm))
```

../fe621/black_scholes/barrier/util.py

E.3.2 Complete Solution Implementation

```
from context import fe621

import pandas as pd


# Option parameters
current = 10
strike = 10
ttm = 0.3
volatility = 0.2
rf = 0.01
H = 11
```

```
steps = 200

q3_answers = pd.DataFrame()

# Part (a)
barrier_tree = fe621.tree_pricing.binomial.Barrier(
    current=current,
    strike=strike,
    ttm=ttm,
    rf=rf,
    volatility=volatility,
    barrier=H,
    barrier_type='O',
    opt_type='C',
    opt_style='E',
    steps=steps
)

q3_answers['Tree Up-and-Out EU Call'] = [barrier_tree.getInstrumentValue()]


# Part (b)
analytical_upandout = fe621.black_scholes.barrier.callUpAndOut(
    S=current,
    H=H,
    volatility=volatility,
    ttm=ttm,
    K=strike,
    rf=rf
)

q3_answers['Analytical Up-and-Out EU Call'] = [analytical_upandout]


# Part (c)
barrier_tree = fe621.tree_pricing.binomial.Barrier(
    current=current,
    strike=strike,
    ttm=ttm,
    rf=rf,
    volatility=volatility,
    barrier=H,
    barrier_type='I',
    opt_type='C',
    opt_style='E',
    steps=steps
)

q3_answers['Tree Up-and-In EU Call'] = [barrier_tree.getInstrumentValue()]

analytical_upandin = fe621.black_scholes.barrier.callUpAndIn(
    S=current,
    H=H,
    volatility=volatility,
    ttm=ttm,
    K=strike,
    rf=rf
)

q3_answers['Analytical Up-and-In EU Call'] = [analytical_upandin]

```

```

# Part (d)
barrier_tree = fe621.tree_pricing.binomial.Barrier(
    current=current,
    strike=strike,
    ttm=ttm,
    rf=rf,
    volatility=volatility,
    barrier=H,
    barrier_type='I',
    opt_type='C',
    opt_style='A',
    steps=steps
)

q3_answers['Tree Up-and-In A Call'] = [barrier_tree.getInstrumentValue()]


# Writing results to CSV
q3_answers.T.to_csv('Homework 2/bin/barrier_option_values.csv',
                    index_label='Type', header=['Value'])
```

question_solutions/question_3.py