Homework Assignment 4
FE 621: Computational Methods in Finance
Instructor: Ionut Florescu
5/8/2019

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

Overview

This is my solution manuscript for FE 621 Homework Assignment 4.

In this Homework Assignment, I explore various Monte Carlo Simulation methods, and price various option contracts. I implement a highly Monte Carlo Simulation Framework, that is extended and utilized throughout the assignment.

The content of this Homework Assignment is divided into four sections; the first discusses the Monte Carlo Model Implementations. The second contains detailed analysis and comparison of the various simulation models, and the explores portfolio modeling with multiple Monte Carlo processes. Finally, the fourth section explores the pricing of exotic basket options with Monte Carlo simulations.

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

1. Weerawarana 2019

Contents

1 Monte Carlo Simulation Framework

1.1 General Monte Carlo Simulation Driver

To simplify the various simulation methods implemented for this Homework Assignemnt, I created a generalized driver to handle the random number generation (of user-specified, arbitrary dimensions), and the loop driving the simulations, completely encapsulating all required functionality.

The driver intelligently handles dynamically specified evaluation and simulation counts. Furthermore, it also handles randomly sampling Gaussian White Noise (GWN) distributed terms for the specific simulation implementations, from an arbitrary number of independent standard Gaussian distributions. This driver also encapsulates functionality to compute a final estimate from the result set of a simulation, as well as other relevant statistics such as the standard error.

```python
from scipy.stats import norm
from typing import Callable
import numpy as np


def monteCarloSkeleton(sim_count: int, eval_count: int, sim_func: Callable,
    sim_dimensionality: int=1) -> np.array:
    """Function to run a simple Monte Carlo simulation. This is a highly
    generalized Monte Carlo simulation skeleton, and takes in functions as
    parameters for computation functions, and final post-processing
    functionality.

    This function uses list comprehensions to improve performance.

    Arguments:
        sim_count {int} -- Simulation count.
        eval_count {int} -- Number of evaluations per simulation.
        sim_func {Callable} -- Function to run on the random numbers
                               (per-simulation).

    Keyword Arguments
        sim_dimensionality {int} -- Dimensionality of the simulation. Affects
                                    the shape of random normals (default: {1}).

    Returns:
        np.array -- Array of simulated value outputs.
    """

    # Simulation function
    def simulation() -> float:
        """Single simulation run. This is written as a separate function so I
        can use list comprehensions in the outer loop, giving this operation
        a significant performance bump.
        """

        # Building list of normal random numbers to apply to sim_func
        rand_Ns = norm.rvs(size=(sim_dimensionality, eval_count))
        # Applying simulated function over path
        return sim_func(rand_Ns)

    # Running simulations the required number of times, returning
    return np.array([simulation() for i in range(0, sim_count)])


def monteCarloStats(mc_output: np.array) -> dict:
    """Function to compute statistics on a Monte Carlo simulation output set.
```

```

    This function computes the estimate (i.e. the mean), sample standard
    deviation (i.e. std. with delta degrees of freedom = 1), and the standard
    error of the Monte Carlo simulation output array.

    Arguments:
        mc_output {np.array} -- Array of simulated Monte Carlo values.

    Returns:
        dict -- Dictionary with summary statistics.
    """

    # Empty dictionary to store output
    output = dict()

    # Estimate
    output['estimate'] = np.mean(mc_output)
    # Standard deviation (sample)
    output['standard_deviation'] = np.std(mc_output, ddof=1)
    # Standard error
    output['standard_error'] = output['standard_deviation'] / np.sqrt(
        len(mc_output))

    # Return final output
    return output
```

../fe621/monte_carlo/monte_carlo.py

1.2 Simple Geometric Brownian Motion

The simple Geometric Brownian Motion (GBM) Monte Carlo simulation function uses the driver described above to simulate a standard Brownian Motion process for a vanilla European Option. It models the price of a Call/Put option under the Black-Scholes model heuristic, with dynamic option metadata, including dividend yields, volatilities, and strike prices.

```
from ..monte_carlo import monteCarloSkeleton, monteCarloStats

import numpy as np


def blackScholes(current: float, volatility: float, ttm: float, strike: float,
                 rf: float, dividend: float, sim_count: int, eval_count: int,
                 opt_type: str='C', **kwargs) -> dict:
    """Function to model the price of a European Option, under the Black-Scholes
    pricing model heuristic, using a Monte-Carlo simulation.

    This function simulates a simple Geometric Brownian Motion (GBM) of the
    underlying asset price, before computing the terminal contract value for a
    given number of simulated paths.

    Then, Monte Carlo simulation statistics are
    computed for each of the simulations, and a dict of results is returned.

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

```
        dividend {float} -- Dividend yield (annual).
        sim_count {int} -- Number of paths to simulate.
        eval_count {int} -- Number of evaluations per path simulation.

    Keyword Arguments:
        opt_type {str} -- Option type; must be 'C' or 'P' (default: {'C'}).

    Raises:
        ValueError: Raised if `opt_type` is not 'C' or 'P'.

    Returns:
        dict -- Formatted dictionary of Monte Carlo simulation results.
    """

    # Verify option type choice
    if opt_type not in ['C', 'P']:
        raise ValueError('Incorrect option type; must be "C" or "P".')

    # Computing delta t
    dt = ttm / eval_count
    # Computing intitial value
    init_val = np.log(current)
    # Computing nudt
    nudt = (rf - dividend - (np.power(volatility, 2) / 2)) * dt

    # Defining lambda function to model Geometric Brownian Motion (GBM)
    gbm = lambda x: nudt + (volatility * np.sqrt(dt) * x)

    # Defining simulation function
    def sim_func(x: np.array) -> float:
        if (opt_type == 'C'):
            # Call option
            return np.exp(-1 * rf * ttm) * \
                np.maximum(np.exp(init_val + np.sum(gbm(x))) - strike, 0)
        else:
            # Put option
            return np.exp(-1 * rf * ttm) * \
                np.maximum(strike - np.exp(init_val + np.sum(gbm(x))), 0)

    # Running simulation
    mc_output = monteCarloSkeleton(sim_count=sim_count,
                                   eval_count=eval_count,
                                   sim_func=sim_func)

    # Computing and returning sample statistics
    return monteCarloStats(mc_output=mc_output)
```

../fe621/monte_carlo/option_pricing/simple_gbm.py

1.3 Antithetic Variates

This function models the price of a vanilla Call/Put European Option, with a variance-reducing antithetic variate Monte Carlo simulation under the Black-Scholes model heuristic.

It achieves its variance-reducing behavior by simulating two perfectly negatively-correlated Brownian Motion processes, and computing the final payoff of the hypothetical simulated option as the arithmetic mean of the values implied by the negatively correlated processes. As with the previously described model, this too handles dynamic option metadata, and utilizes the main Monte Carlo driver described above.

```
from ..monte_carlo import monteCarloSkeleton, monteCarloStats

import numpy as np


def blackScholes(current: float, volatility: float, ttm: float, strike: float,
                 rf: float, dividend: float, sim_count: int, eval_count: int,
                 opt_type: str='C', **kwargs) -> dict:
    """Function to model the price of a European Option, under the
    Black-Scholes pricing model heuristic, using an antithetic variates method
    variance-reduced Monte-Carlo simulation.

    This function simulates two perfectly negatively correlated simple Geometric
    Brownian Motion (GBM) processes of the underlying asset price, before
    computing the terminal contract value for a given number of simulated paths,
    as the arithmetic average of the payouts of each of the two GBMs.

    Then, Monte Carlo simulation statistics are computed for each of the
    simulations, and a dict of results is returned.

    Arguments:
        current {float} -- Current price of the underlying asset.
        volatility {float} -- Volatility of the underlying asset price.
        ttm {float} -- Time to expiration (in years).
        strike {float} -- Strike price of the option contract.
        rf {float} -- Risk-free rate (annual).
        dividend {float} -- Dividend yield (annual).
        sim_count {int} -- Number of paths to simulate.
        eval_count {int} -- Number of evaluations per path simulation.

    Keyword Arguments:
        opt_type {str} -- Option type; must be 'C' or 'P' (default: {'C'}).

    Raises:
        ValueError: Raised if `opt_type` is not 'C' or 'P'.

    Returns:
        dict -- Formatted dictionary of Monte Carlo simulation results.
    """

    # Verify option choice
    if opt_type not in ['C', 'P']:
        raise ValueError('Incorrect option type; must be "C" or "P".')

    # Computing delta t
    dt = ttm / eval_count
    # Computing initial value
    init_val = np.log(current)
    # Computing nudt
    nudt = (rf - dividend - (np.power(volatility, 2) / 2)) * dt

```

```
    # Defining lambda functions to model Geometric Brownian Motion (GBM);
    # for both assets (negatively correlated)
    gbm1 = lambda x: nudt + (volatility * np.sqrt(dt) * x)
    gbm2 = lambda x: nudt + (volatility * np.sqrt(dt) * (-1 * x))

    # Defining simulation function
    def sim_func(x: np.array) -> float:
        if (opt_type == 'C'):
            # Call option
            return np.exp(-1 * rf * ttm) * 0.5 * (
                np.maximum(np.exp(init_val + np.sum(gbm1(x))) - strike, 0) +
                np.maximum(np.exp(init_val + np.sum(gbm2(x))) - strike, 0))
        else:
            # Put option
            return np.exp(-1 * rf * ttm) * 0.5 * (
                np.maximum(strike - np.exp(init_val + np.sum(gbm1(x))), 0) +
                np.maximum(strike - np.exp(init_val + np.sum(gbm2(x))), 0))

    # Running simulation
    mc_output = monteCarloSkeleton(sim_count=sim_count,
                                   eval_count=eval_count,
                                   sim_func=sim_func)

    # Computing and returning sample statistics
    return monteCarloStats(mc_output=mc_output)
```

../fe621/monte_carlo/option_pricing/antithetic_variates.py

1.4 Control Variates

This function models the price of a vanilla Call/Put European option, with a variance-reducing Delta-based control variate Monte Carlo simulation under the Black-Scholes model heuristic.

It achieves this variance-reducing behavior by simulating a portfolio of the underlying asset, and a delta-hedge of the option for the duration of a given simulated path. Then, through a process of bias-correcting option-delta computation (see Appendix C.1), it *corrects* the underlying option price to reduce the variance of the resulting estimate. Similar to the previously described models, this too handles dynamic option metadata, and utilizes the main Monte Carlo driver described above.

```
from ..monte_carlo import monteCarloSkeleton, monteCarloStats
from ...black_scholes.greeks import callDelta, putDelta

import numpy as np


def deltaCVBlackScholes(current: float, volatility: float, ttm: float,
                        strike: float, rf: float, dividend: float, sim_count: int, eval_count:
                             int, beta1: float, opt_type: str='C') -> dict:
    """Function to model the price of a European Option, under the
    Black-Scholes pricing model heuristic, using a control variates method
    variance-reduced Monte-Carlo simulation.

    This function simulates a delta-hedged portfolio mimicking a call or put
    option, under the Black-Scholes pricing heuristic.

    Then, Monte Carlo simulation statistics are computed for each of the
    simulations, and a dict of results is returned.

    Arguments:
```

```
        current {float} -- Current price of the underlying asset.
        volatility {float} -- Volatility of the underlying asset price.
        ttm {float} -- Time to expiration (in years).
        strike {float} -- Strike price of the option contract.
        rf {float} -- Risk-free rate (annual).
        dividend {float} -- Dividend yield (annual).
        sim_count {int} -- Number of paths to simulate.
        eval_count {int} -- Number of evaluations per path simulation.
        beta {float} -- Beta coefficient for the delta hedge.

    Keyword Arguments:
        opt_type {str} -- Option type; must be 'C' or 'P' (default: {'C'}).

    Raises:
        ValueError: Raised if `opt_type` is not 'C' or 'P'.

    Returns:
        dict -- Formatted dictionary of Monte Carlo simulation results.
    """

    # Verify option type choice
    if opt_type not in ['C', 'P']:
        raise ValueError('Incorrect option type; must be "C" or "P".')

    # Computing delta t
    dt = ttm / eval_count
    # Computing nudt
    nudt = (rf - dividend - (np.power(volatility, 2) / 2)) * dt
    # Delta bias correction
    erddt = np.exp((rf - dividend) * dt)

    # Building vector of ttms (for option delta evaluation)
    # Note: This starts from timestep 1, to timestep eval_count.
    # Note: This is the time to maturity, the order must be flipped to match
    #       the simulated asset prices (at the first sim price
    #       it is ((ttm - dt), (ttm - 2*dt), ...)
    ttm_vec = np.flip(np.arange(start=dt, stop=(ttm + dt), step=dt))

    # Defining lambda function to model underlying Geometric Brownian Motion,
    # and Delta-based control variate
    gbm = lambda x: nudt + (volatility * np.sqrt(dt) * x)

    # Defining simulation function
    def sim_func(x: np.array) -> float:
        # Underlying price path
        st = np.cumprod(np.exp(gbm(x))) * current

        if (opt_type == 'C'):
            # Call option
            # Delta computation
            delta = callDelta(current=st,
                              volatility=volatility,
                              ttm=ttm_vec,
                              strike=strike,
                              rf=rf,
                              dividend=dividend)

            # Terminal payoff computation (future value)
            terminal_payoff = np.maximum(st[-1] - strike, 0)
        else:
            # Put option
```

```python
            # Delta computation
            delta = putDelta(current=st,
                             volatility=volatility,
                             ttm=ttm_vec,
                             strike=strike,
                             rf=rf,
                             dividend=dividend)
            # Terminal payoff computation (future value)
            terminal_payoff = np.maximum(strike - st[-1], 0)

        # Control variate computation
        cv = np.sum(delta[:-1] * (st[1:] - (st[:-1] * erddt)))

        # Adjusting estimate by control variate; returning present value
        return np.exp(-1 * rf * ttm) * (terminal_payoff + (cv * beta1))

    # Runnig simulation
    mc_output = monteCarloSkeleton(sim_count=sim_count,
                                   eval_count=eval_count,
                                   sim_func=sim_func)

    # Computing and returning sample statistics
    return monteCarloStats(mc_output=mc_output)
```

../fe621/monte_carlo/option_pricing/control_variates.py

1.5 Antithetic and Control Variates

This function is a combination of the Delta-based control variate and antithetic variate variance-reducing Monte Carlo simulations described above. This implementation covers pricing a vanilla Call/Put European Option under the Black-Scholes model heuristic.

Effectively, this function implements a Delta-based control variate within an antithetic variate framework. Similar to the traditional antithetic variate, it models two perfectly negatively correlated geometric Brownian Motions, while applying the Delta-based control variate to both. Then, it computes the final option value as the arithmetic mean of the values implied by the two processes. It too enables all of the variable option metadata functionality discussed above, and utilizes the main Monte Carlo driver.

```python
from ..monte_carlo import monteCarloSkeleton, monteCarloStats
from ...black_scholes.greeks import callDelta, putDelta

import numpy as np


def deltaCVBlackScholes(current: float, volatility: float, ttm: float,
                        strike: float, rf: float, dividend: float, sim_count: int, eval_count:
                             int, beta1: float, opt_type: str='C') -> dict:
    """Function to model the price of a European Option, under the
    Black-Scholes pricing model heuristic, using an antithetic variates and
    Delta-based control variates method variance-reduced Monte-Carlo simulation.

    This function simulates two perfectly negatively correlated simple Geometric
    Brownian Motion (GBM) processes of the underlying asset price, before
    computing the terminal contract value for a given number of simulated paths,
    as the arithmetic average of the payouts of each of the two GBMs. This
    function also performs delta hedging against a portfolio of these two
    perfectly negatively correlated GBMs, to reduce the variance of the
    estimate further.

```

```
    Then, Monte Carlo simulation statistics are computed for each of the
    simulations, and a dict of results is returned.

    Arguments:
        current {float} -- Current price of the underlying asset.
        volatility {float} -- Volatility of the underlying asset price.
        ttm {float} -- Time to expiration (in years).
        strike {float} -- Strike price of the option contract.
        rf {float} -- Risk-free rate (annual).
        dividend {float} -- Dividend yield (annual).
        sim_count {int} -- Number of paths to simulate.
        eval_count {int} -- Number of evaluations per path simulation.
        beta {float} -- Beta coefficient for the delta hedge.

    Keyword Arguments:
        opt_type {str} -- Option type; must be 'C' or 'P' (default: {'C'}).

    Raises:
        ValueError: Raised if `opt_type` is not 'C' or 'P'.

    Returns:
        dict -- Formatted dictionary of Monte Carlo simulation results.
    """

    # Verify option type choice
    if opt_type not in ['C', 'P']:
        raise ValueError('Incorrect option type; must be "C" or "P".')

    # Computing delta t
    dt = ttm / eval_count
    # Computing nudt
    nudt = (rf - dividend - (np.power(volatility, 2) / 2)) * dt
    # Delta bias correction
    erddt = np.exp((rf - dividend) * dt)

    # Building vector of ttms (for option delta evaluation)
    # Note: This starts from timestep 1, to timestep eval_count.
    # Note: This is the time to maturity, the order must be flipped to match
    #       the simulated asset prices (at the first sim price
    #       it is ((ttm - dt), (ttm - 2*dt), ...)
    ttm_vec = np.flip(np.arange(start=dt, stop=(ttm + dt), step=dt))

    # Defining lambda function to model underlying Geometric Brownian Motion,
    # and Delta-based control variate
    gbm = lambda x: nudt + (volatility * np.sqrt(dt) * x)

    # Defining simulation function
    def sim_func(x: np.array) -> float:
        # Underlying price path
        st1 = np.cumprod(np.exp(gbm(x))) * current
        st2 = np.cumprod(np.exp(gbm(-1 * x))) * current

        if (opt_type == 'C'):
            # Call option
            # Delta computation
            delta1 = callDelta(st1, volatility, ttm_vec, strike, rf, dividend)
            delta2 = callDelta(st2, volatility, ttm_vec, strike, rf, dividend)
            # Terminal payoff computation (future value)
            terminal_payoff1 = np.maximum(st1[-1] - strike, 0)
            terminal_payoff2 = np.maximum(st2[-1] - strike, 0)
        else:
```

```
            # Put option
            # Delta computation
            delta1 = putDelta(st1, volatility, ttm_vec, strike, rf, dividend)
            delta2 = putDelta(st2, volatility, ttm_vec, strike, rf, dividend)
            # Terminal payoff computation (future value)
            terminal_payoff1 = np.maximum(strike - st1[-1], 0)
            terminal_payoff2 = np.maximum(strike - st2[-1], 0)

        # Control variate computation
        cv1 = np.sum(delta1[:-1] * (st1[1:] - (st1[:-1] * erddt)))
        cv2 = np.sum(delta2[:-1] * (st2[1:] - (st2[:-1] * erddt)))

        # Adjusting estimate by control variate; returning present value
        return np.exp(-1 * rf * ttm) * 0.5 * (
            (terminal_payoff1 + (cv1 * beta1)) +
            (terminal_payoff2 + (cv2 * beta1)))

    # Runnig simulation
    mc_output = monteCarloSkeleton(sim_count=sim_count,
                                   eval_count=eval_count,
                                   sim_func=sim_func)

    # Computing and returning sample statistics
    return monteCarloStats(mc_output=mc_output)
```

../fe621/monte_carlo/option_pricing/antithetic_control_variates.py

2 Monte Carlo Simulation Methods Analysis

In this section, I explore the performance of the various Monte Carlo simulation implementations described in the previous section.

2.1 Simple GBM Monte Carlo Analysis

Utilizing the `fe621` package code reproduced above, Monte Carlo driven simulations of a simple GBM process was emulated. The simulation count, m, and the evaluation count (i.e. number of time steps), n were varied, and the standard error and time elapsed were examined.

The source code for this simple GBM analysis is reproduced in Appendix B.1. The raw dataset from this analysis is reproduced in full in Appendix A.1.

Simulation Count	n = 300	n = 400	n = 500	n = 600	n = 700
1000000	0.01371	0.01368	0.01371	0.01370	0.01369
2000000	0.00968	0.00969	0.00967	0.00970	0.00968
3000000	0.00791	0.00790	0.00791	0.00792	0.00790
4000000	0.00685	0.00684	0.00684	0.00685	0.00684
5000000	0.00612	0.00612	0.00613	0.00612	0.00612

Table 1: Standard error of estimates for various configurations of simulation count, m, and evaluation count, n for the Simple GBM Monte Carlo simulation.

Simulation Count	n = 300	n = 400	n = 500	n = 600	n = 700
1000000	31.41597	35.24046	37.61623	40.60616	43.25504
2000000	64.52212	70.15672	75.01042	81.18651	85.95727
3000000	96.59307	103.65265	111.96133	121.49709	129.37581
4000000	127.70113	138.41027	149.31797	162.48306	172.75480
5000000	160.47166	174.49779	195.68161	204.59356	216.86057

Table 2: Time elapsed (in seconds) for various configurations of simulation count, m, and evaluation count, n for the Simple GBM Monte Carlo simulation.

The standard error for each of the estimates is displayed in Table 1, and the time elapsed for computation is displayed in Table 2.

It is clear that the evaluation time of each of the Monte Carlo simulations does not vary (relatively) significantly with increasing evaluation (i.e. time) steps, n. This relatively stagnant behavior is also visible when considering the standard error, which vary significantly with increasing n.

However, there is a clear positive correlation between the number of simulated paths, m, and the time elapsed (from Table 2). Similarly, there is a significant negative correlation between m and the standard error of the estimate (from Table 1). This indicates that there is a clear increase in accuracy with increasing simulated paths, m.

2.2 Monte Carlo Methods Analysis

Similar to the previous subsection, `fe621` package code reproduced in the previous section was used to analyze the performance of the various Monte Carlo simulation methods to price vanilla European Call and Put options. The simulation count, m was set to $1,000,000$, and the evaluation count, n was set to 700 for all simulations.

The source code this MC methods analysis is reproduced in Appendix B.1, and the raw dataset is reproduced in full in Appendix A.2.

MC Method	Option Type	Estimate	Std Error	Time Elapsed
Antithetic MC	C	9.13168	0.00722	66.05935
Antithetic MC	P	6.26197	0.00464	65.72103
Antithetic and Control Delta MC	C	9.13528	0.00032	504.39522
Antithetic and Control Delta MC	P	6.26741	0.00028	508.45291
Control Delta MC	C	9.13535	0.00061	271.13860
Control Delta MC	P	6.26747	0.00098	276.63671
Simple MC	C	9.10371	0.01365	44.68048
Simple MC	P	6.25836	0.00907	45.93523

Table 3: A comparison of various Monte Carlo simulation methods.

Utilizing the results displayed in Table 3, there are significant conclusions that can be drawn regarding the performance of each of the Monte Carlo simulation methods.

It is clear from the estimated values that with the exception of the simple GBM method, all of the estimates are in extremely close proximity to each other. This is to be expected, as the variance of the other methods - again, with the exception of simple GBM - are relatively extremely small. This implies that they are significantly closer to convergence to the true value of the option, relative to the simple GBM estimate.

Analyzing the standard error of the estimates, it is clear that the optimal method (without considering computation time) is the Antithetic and Control Delta MC simulation. However, when taking in computation time to account, the best tradeoff appears to be the Control Delta MC. This method provides a standard error nearly a full order of magnitude less than the Antithetic MC, while taking approximately 5 times as long.

The standard error to computation time tradeoff of the combination of the Antithetic and Control Delta MC method pales in comparison, as it takes nearly twice as long as the Control Delta MC simulation, while only providing a standard error that is improved by a factor of approximately 2. Thus, given the computation time constraint, the optimal method for this particular option configuration appears to be the Control Delta MC simulation method.

3 Multiple Monte Carlo Processes

In this section, I utilize the framework detailed above to simulate multiple Monte Carlo processes to build a portfolio, and perform risk analytics. All source code for this question is reproduced in Appendix B.3.

3.1 Portfolio Positions

	Positions	Position Value (USD)	Position Value (CNY)
IBM Equity	50000	4000000	24400000
10-Year T-Bill	33	2970000	18117000
CNY/USD ForEx	18300000	3000000	18300000
Total	-	9970000	60817000

Table 4: Initial portfolio data from the multiple Monte Carlo process simulation.

Table 4 displays initial position data for the simulated portfolio, assuming that fractional ownership of assets is not possible. In addition to the initial positions of each of the assets, the position value is also displayed in both US Dollars (USD), and Chinese Renminbi (CNY).

3.2 Risk Analytics

	10 Day	1 Day
VaR ($)	528386.5015	167090.4830
VaR (%)	5.2839	1.6709
CVaR ($)	597910.1166	189075.7805
CVaR (%)	5.9791	1.8908

Table 5: Risk analytics performed on the portfolio, computed with a multiple Monte Carlo process simulation.

All risk metrics displayed in Table 5 for the portfolio were computed using Monte Carlo simulation data, without making any assumptions about the distribution of the underlying portfolio returns.

4 Basket Option Pricing with Correlated BM

Part (a) of this question is addressed directly in the source code, replicated in full in Appendix B.4. Additionally, unless otherwise stated, the annualized risk free rate is assumed to be 6.0%, as one was not provided in the Homework Prompt.

4.1 3-Dimensional Correlated BM Process Visualization

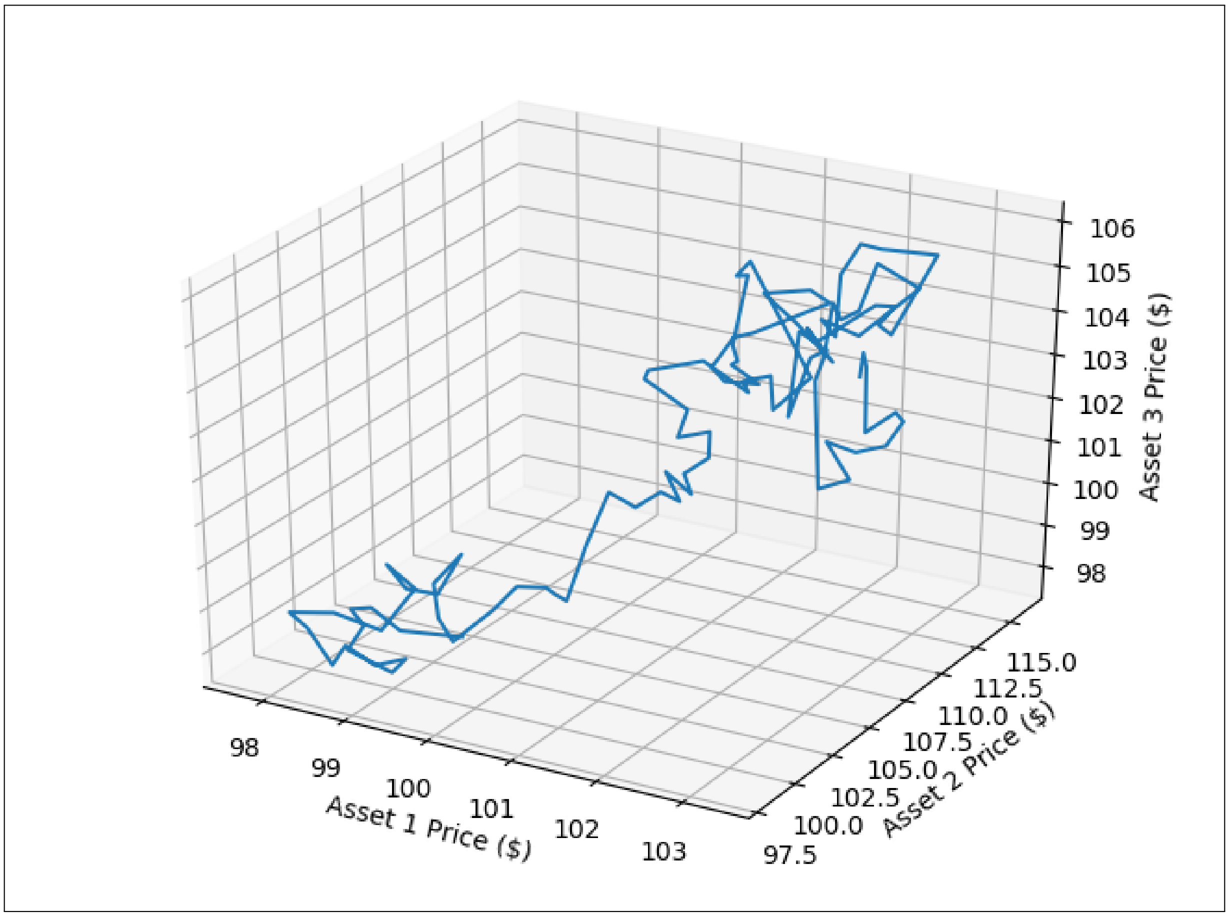

Figure 1: Sample realized path of the simulation of the correlated 3-dimensional Brownian Motion process.

Figure 1 displays a single sample simulated path from the 3-dimensional correlated Brownian Motion process.

4.2 Basket Option Pricing

In this section, I price an vanilla Call and Put option, treating the correlated 3-dimensional Brownian Motion process as the underlying basket of assets on which the option is written. The sample statistics for this option are reproduced in Table 6.

	European Call	European Put
Estimate	2.243289494600801	1.271557537236066
Standard Deviation	2.9576096946924317	2.059228990290305
Standard Error	0.09352783065023298	0.0651185383316612

Table 6: Sample statistics of a basket option priced with a simulated 3-dimensional correlated Brownian Motion process.

4.3 Exotic Basket Option Pricing

In this section, I price an exotic option (a variant of a vanilla option, with an embedded barrier for one of the basket correlates), treating the correlated 3-dimensional Brownian Motion process as the underlying basket of assets on which the option is written. The sample statistics for this option are reproduced in Table 7.

Estimate	746.2211052461753
Standard Deviation	2402.799458578407
Standard Error	75.98319049727175

Table 7: Sample statistics of an exotic basket option priced with a simulated 3-dimensional correlated Brownian Motion process.

References

Florescu, Ionut. 2019. "1.11.4 Simpson's Rule." Chap. 1 in *Computational Methods in Finance,* 25–26. Hoboken, NJ.

Shreve, Steven E. 2004. *Stochastic Calculus for Finance II.* 153–164. April. Pittsburgh, PA: Springer Finance. ISBN: 0-387-40101-6.

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

Weerawarana, Rukmal. 2016. *Homework 3 - CFRM 460 (Mathematical Methods for Computational Finance) - University of Washington - rukmal - GitHub.* Accessed February 12, 2019. `https://github.com/rukmal/CFRM-460-Homework/blob/master/Homework%203/Homework%203%20Solutions.pdf`.

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

A Raw Data

A.1 Simple GBM Analysis

Sim Count	Eval Count	Estimate ($)	Std Dev	Std Err	Time (s)
1000000.0	300.0	9.146 50	13.705 23	0.013 71	31.415 97
1000000.0	400.0	9.129 21	13.675 35	0.013 68	35.240 46
1000000.0	500.0	9.140 98	13.708 28	0.013 71	37.616 23
1000000.0	600.0	9.132 68	13.703 56	0.013 70	40.606 16
1000000.0	700.0	9.121 87	13.685 59	0.013 69	43.255 04
2000000.0	300.0	9.125 48	13.682 80	0.009 68	64.522 12
2000000.0	400.0	9.139 63	13.703 30	0.009 69	70.156 72
2000000.0	500.0	9.121 39	13.679 24	0.009 67	75.010 42
2000000.0	600.0	9.146 36	13.711 28	0.009 70	81.186 51
2000000.0	700.0	9.133 39	13.688 25	0.009 68	85.957 27
3000000.0	300.0	9.137 94	13.704 37	0.007 91	96.593 07
3000000.0	400.0	9.132 66	13.690 64	0.007 90	103.652 65
3000000.0	500.0	9.130 79	13.697 76	0.007 91	111.961 33
3000000.0	600.0	9.143 36	13.709 20	0.007 92	121.497 09
3000000.0	700.0	9.123 35	13.689 47	0.007 90	129.375 81
4000000.0	300.0	9.146 65	13.701 23	0.006 85	127.701 13
4000000.0	400.0	9.128 04	13.688 66	0.006 84	138.410 27
4000000.0	500.0	9.137 23	13.688 89	0.006 84	149.317 97
4000000.0	600.0	9.131 00	13.693 72	0.006 85	162.483 06
4000000.0	700.0	9.133 49	13.686 84	0.006 84	172.754 80
5000000.0	300.0	9.135 00	13.691 95	0.006 12	160.471 66
5000000.0	400.0	9.131 14	13.691 86	0.006 12	174.497 79
5000000.0	500.0	9.142 45	13.700 76	0.006 13	195.681 61
5000000.0	600.0	9.125 71	13.680 67	0.006 12	204.593 56
5000000.0	700.0	9.129 56	13.679 67	0.006 12	216.860 57

A.2 MC Method Analysis

Method	Opt Type	Estimate ($)	Std Dev	Std Err	Time (s)
Simple MC	C	9.103 71	13.647 54	0.013 65	44.680 48
Simple MC	P	6.258 36	9.068 55	0.009 07	45.935 23
Antithetic MC	C	9.131 68	7.217 88	0.007 22	66.059 35
Antithetic MC	P	6.261 97	4.638 44	0.004 64	65.721 03
Control Delta MC	C	9.135 35	0.613 18	0.000 61	271.138 60
Control Delta MC	P	6.267 47	0.981 60	0.000 98	276.636 71
Antithetic and Control Delta MC	C	9.135 28	0.319 66	0.000 32	504.395 22
Antithetic and Control Delta MC	P	6.267 41	0.281 27	0.000 28	508.452 91

B Solution Source Code

B.1 Question 1 Solution

```
from context import fe621

import itertools
import numpy as np
import pandas as pd
import time


# Initial parameters
current = 100
strike = 100
volatility = .2
rf = 0.06
dividend = 0.03
ttm = 1


# Part (a)
def partA():
    """Function to answer Part (a) of question 1; analyzing the performance of
    simple GBM Black-Scholes model heuristic Monte Carlo option pricing.
    """

    # Parameters for simulation analysis
    sim_counts = np.arange(start=1e6, stop=5e6 + 1, step=1e6, dtype=int)
    eval_counts = np.arange(start=300, stop=701, step=100, dtype=int)

    # Output dataframe
    output = pd.DataFrame()

    # Output monitoring stuff
    total_combos = len(sim_counts) * len(eval_counts)
    counter = 1

    # Iterating over possible simulation counts
    for sim_count in sim_counts:
        # Iterating over possible evaluation counts
        for eval_count in eval_counts:
            # Print update
            print('Starting simulation with eval count {0} and sim count {1}'.
                format(eval_count, sim_count))

            # Metadata dictionary
            meta = dict()

            # Storing simulation count and evaulation count
            meta['sim_count'] = sim_count
            meta['eval_count'] = eval_count

            # Starting timer
            start_time = time.time()

            # Running black scholes monte carlo simulation
            sim_output = fe621.monte_carlo.simple_gbm.blackScholes(
                current=current,
                volatility=volatility,
```

```
                ttm=ttm,
                strike=strike,
                rf=rf,
                dividend=dividend,
                sim_count=sim_count,
                eval_count=eval_count,
                opt_type='C'
            )

            # Recording time elapsed
            meta['time_elapsed'] = time.time() - start_time

            # Updating meta dictionary with values from simulation output
            meta.update(sim_output)

            # Adding to output dataframe
            output = output.append(meta, ignore_index=True)

            # Printing status update, update counter
            print('{0}% complete'.format(counter / total_combos * 100))
            print('Finished eval count {0} and sim count {1} in {2} minutes'.
                format(eval_count, sim_count, meta['time_elapsed'] / 60))
            counter += 1

    # Saving to CSV
    output.to_csv('Homework 4/bin/raw_simple_mc_gbm_analysis.csv', index=False)


# Part (b)
def partB():
    """Function to answer Part (b) of question 1; analyzing the performance of
    various Black-Scholes model heuristic Monte Carlo pricing simulations.
    """

    # Output dataframe
    output = pd.DataFrame()

    # Simulation settings
    sim_count = int(1e6)
    eval_count = 700
    opt_types = ['C', 'P']

    # Arguments for the simulations
    args = {
        'current': current,
        'volatility': volatility,
        'ttm': ttm,
        'strike': strike,
        'rf': rf,
        'dividend': dividend,
        'sim_count': sim_count,
        'eval_count': eval_count,
        'beta1': -1.0
    }

    # Simulation functions
    sim_functions = {
        'Simple MC': fe621.monte_carlo.simple_gbm.blackScholes,
        'Antithetic MC':
            fe621.monte_carlo.antithetic_variates.blackScholes,
        'Control Delta MC':
```

```
            fe621.monte_carlo.control_variates.deltaCVBlackScholes,
        'Antithetic and Control Delta MC':
            fe621.monte_carlo.antithetic_control_variates.deltaCVBlackScholes
    }

    # Output monitoring stuff
    total_combos = len(sim_functions.keys()) * len(opt_types)
    counter = 1

    # Iterating over all combinations of simulation functions and option types
    # See: https://docs.python.org/3/library/itertools.html#itertools.product
    for sim_method, opt_type in itertools.product(sim_functions.keys(),
        opt_types):
        # Print update
        print('Starting simulation with method {0} and option type {1}'.
            format(sim_method, opt_type))

        # Dictionary to store metadata
        meta = dict()

        # Simulation metadata (general)
        meta['method'] = sim_method
        meta['opt_type'] = opt_type

        # Setting option type in the args dictionary
        args['opt_type'] = opt_type

        # Starting timer
        start_time = time.time()

        # Running simulation
        sim_output = sim_functions[sim_method](**args)

        # Timing simulation
        meta['time_elapsed'] = time.time() - start_time

        # Updating meta dictionary with values from simulation output
        meta.update(sim_output)

        # Adding to output dataframe
        output = output.append(meta, ignore_index=True)

        # Printing status update, update counter
        print('{0}% complete'.format(counter / total_combos * 100))
        print('Finished method {0} and option type {1} in {2} minutes'.
            format(sim_method, opt_type, meta['time_elapsed'] / 60))
        counter += 1

    # Writing the output to a CSV
    output.to_csv(
        'Homework 4/bin/raw_mc_methods_analysis.csv',
        index=False,
        columns=['method', 'opt_type', 'estimate', 'standard_deviation',
                 'standard_error', 'time_elapsed']
    )

if __name__ == '__main__':
    # Part A - raw data
    partA()

    # Part B - raw data
```

```
    partB()
```

question_solutions/question_1.py

B.2 Question 1 Formatting Scripts

B.2.1 Simple MC Analysis

```
from context import fe621

import pandas as pd


# Loading raw CSV of simple GBM analysis
simple_mc_analysis = pd.read_csv(
    'Homework 4/bin/raw_simple_mc_gbm_analysis.csv')

# Creating table of evaluation time
simple_mc_eval_time = simple_mc_analysis.pivot(
    index='sim_count',
    columns='eval_count',
    values='time_elapsed'
)

# Creating table of standard error
simple_mc_std_error = simple_mc_analysis.pivot(
    index='sim_count',
    columns='eval_count',
    values='standard_error'
)

# Renaming columns and index
simple_mc_eval_time.columns = [' '.join(['n = ', str(int(i))])
    for i in simple_mc_eval_time.columns]
simple_mc_eval_time.index = pd.Index(simple_mc_eval_time.index, dtype=int)
simple_mc_std_error.columns = [' '.join(['n = ', str(int(i))])
    for i in simple_mc_std_error.columns]
simple_mc_std_error.index = pd.Index(simple_mc_std_error.index, dtype=int)

# Formatting, output to CSV
simple_mc_eval_time.to_csv('Homework 4/bin/q1_simple_mc_time.csv',
    float_format='%.5f', index_label='Simulation Count')
simple_mc_std_error.to_csv('Homework 4/bin/q1_simple_mc_std_err.csv',
    float_format='%.5f', index_label='Simulation Count')
```

question_solutions/q1_format_simple_mc.py

B.2.2 MC Methods Analysis

```
from context import fe621

import pandas as pd


# Loading raw CSV of MC methods analysis
simple_mc_analysis = pd.read_csv(
    'Homework 4/bin/raw_mc_methods_analysis.csv')
```

```

# Creating formatted table
out_df = simple_mc_analysis.pivot_table(index=['method', 'opt_type'], values=['estimate', '
    standard_error', 'time_elapsed'])

out_df.columns = ['Estimate', 'Std Error', 'Time Elapsed']

# Format, output to CSV
out_df.to_csv('Homework 4/bin/q1_mc_methods.csv', float_format='%.5f',
              index_label=['MC Method', 'Option Type'])
```

question_solutions/q1_format_mc_methods.py

B.3 Question 2 Solution

```
from context import fe621

import numpy as np
import pandas as pd


# Portfolio Metadata
port_init_val = 1e7  # Portfolio value
port_weights = np.array([.4, .3, .3])  # IBM, 10 yr Treasury, Yuan
initial_prices = np.array([80, 90000, 6.1])
asset_labels = ['IBM Equity', '10-Year T-Bill', 'CNY/USD ForEx']

# Portfolio initial stats
# Inverting CNY/USD rate as we're buying in USD
initial_prices_corrected = np.append(initial_prices[:-1], 1 / initial_prices[2])
# Flooring and casting to int as we can't buy fractional units
port_positions = np.floor((port_weights * port_init_val
    / initial_prices_corrected)).astype(int)

# Simulation data
sim_count = int(3e6)
dt = 0.001
t = 10 / 252
eval_count = int(np.ceil(t / dt))

# Processes
n_xt = lambda xt, w: xt + ((0.01 * xt * dt) + (0.3 * xt * np.sqrt(dt) * w))
n_yt = lambda yt, w, t: yt + (100 * (90000 + (1000 * t) - yt) * dt + (np.sqrt(yt)
    * np.sqrt(dt) * w))
n_zt = lambda zt, w: zt + ((5 * (6 - zt) * dt) + (0.01 * np.sqrt(zt)
    * np.sqrt(dt) * w))

# Function to compute portfolio value, given asset prices
def portfolioValue(asset_prices: np.array):
    # Making copy of asset prices (to not edit original array)
    asset_prices = np.copy(asset_prices)
    # Inverting last current price value (it is CNY/USD; we're buying in USD)
    asset_prices[2] = 1 / asset_prices[2]

    # Returning portfolio value (sum of elem-wise product across positions)
    return np.sum(np.multiply(asset_prices, port_positions))

# Simulation function
def sim_func(x: np.array) -> float:
```

```
    # Input: (3 x eval_count) matrix
    # Isolating initial prices (only need to maintain last observation)
    current_prices = np.copy(initial_prices)
    # Iterating over columns (i.e. in 3x1 chunks)
    for idx, w_vec in enumerate(x.T):
        current_prices = np.array([
            n_xt(current_prices[0], w_vec[0]),
            n_yt(current_prices[1], w_vec[1], dt * (idx + 1)),
            n_zt(current_prices[2], w_vec[2])
        ])

    return portfolioValue(current_prices)


# Running simulation
sim_data = fe621.monte_carlo.monteCarloSkeleton(
    sim_count=sim_count,
    eval_count=eval_count,
    sim_func=sim_func,
    sim_dimensionality=3
)


def exportInitialData():
    """Function to export initial portfolio data (answering question 2(a))
    """
    # Building output dictionary with necessary data
    # Specifically, positions, USD value, and CNY value
    output = dict()
    output['Positions'] = list(port_positions) + ['-']
    output['Position Value (USD)'] = np.append(np.multiply(
        initial_prices_corrected,
        port_positions
    ), portfolioValue(initial_prices))
    output['Position Value (CNY)'] = output['Position Value (USD)'] * initial_prices[2]

    # Building output dataframe, formatting and saving to CSV
    out_df = pd.DataFrame(output, index=[*asset_labels, 'Total'])
    out_df.to_csv('Homework 4/bin/q2_port_data.csv', float_format='%.0f')


def performRiskAnalytics():
    """Function to compute and export the portfolio VaR and CVaR (2(b) & 2(c))
    """

    # Output data dictionary
    output = dict()

    # VaR config
    N = 10
    alpha = 0.01

    # Computing simulation stats
    sim_stats = fe621.monte_carlo.monteCarloStats(mc_output=sim_data)

    # Computing value at risk (VaR) using the quantile method
    var = sim_stats['estimate'] - np.quantile(sim_data, alpha)
    var_daily = var / np.sqrt(N)
    output['VaR ($)'] = [var, var_daily]
    output['VaR (%)'] = np.array(output['VaR ($)']) / port_init_val * 100

```

```
    # Isolating portfolios that perform worse than the VaR risk threshold
    shortfall_ports = sim_data[sim_data <= np.quantile(sim_data, alpha)]
    # Computing conditional value at risk (cVaR) using the quantile method
    cvar = np.mean(sim_stats['estimate'] - shortfall_ports)
    cvar_daily = cvar / np.sqrt(N)
    output['CVaR ($)'] = [cvar, cvar_daily]
    output['CVaR (%)'] = np.array(output['CVaR ($)']) / port_init_val * 100

    # Building output dataframe, formatting and outputting to CSV
    out_df = pd.DataFrame(output, index=['10 Day', '1 Day']).T

    out_df.to_csv('Homework 4/bin/q2_risk_analytics.csv', float_format='%.4f')


if __name__ == '__main__':
    # Part (1)
    # exportInitialData()

    # Part (2)
    performRiskAnalytics()
```

question_solutions/question_2.py

B.4 Question 3 Solution

```
from context import fe621

import numpy as np
import pandas as pd
from scipy.linalg import cholesky
from scipy.stats import norm


# Asset basket data
init_prices = np.array([100, 101, 98])
mu_vec = np.array([0.03, 0.06, 0.02])
sigma_vec = np.array([0.05, 0.2, 0.15])
corr_mat = np.array([[1.0, 0.5, 0.2],
                     [0.5, 1.0, -0.4],
                     [0.2, -0.4, 1.0]])

# Performing Cholesky decomposition
L = cholesky(corr_mat, lower=True)

# Defining simulation parameters
dt = 1 / 365
ttm = 100 / 365
sim_count = 1000
eval_count = int(ttm / dt)
rf = 0.06

# Defining process function
st = lambda x, volatility, mu: (mu * dt) + (volatility * np.sqrt(dt) * x)


def partB():
    """Solution to 3(b)
    """

```

```
    # Defining simulation function
    def sim_func(x: np.array) -> np.array:
        return np.array([init_prices[i] * np.exp(np.cumsum(
            st(x[i], sigma_vec[i], mu_vec[i])))
            for i in range(0, 3)])

    # Running simulation
    sim_results = fe621.monte_carlo.monteCarloSkeleton(
        sim_count=sim_count,
        eval_count=eval_count,
        sim_func=sim_func,
        sim_dimensionality=3
    )

    # Reshaping as per question specs
    # (rows: time step, col: simulation, z: asset)
    sim_results = np.swapaxes(sim_results, 0, 1)  # sims to columns
    sim_results = np.swapaxes(sim_results, 0, 2)  # assets to z, time to row

    # Importing required packages for plotting
    # Note: Doing this here so I can use the debugger in other sections
    #       without the python-framework macOS installation issue

    # Importing plotting libs
    from mpl_toolkits.mplot3d import Axes3D
    import matplotlib.pyplot as plt

    # Isolating data for each axis
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    # Simulation number
    sim = 1

    x_vals = sim_results[:, sim, 0]
    y_vals = sim_results[:, sim, 1]
    z_vals = sim_results[:, sim, 2]
    # Plotting surface
    ax.plot(x_vals, y_vals, z_vals)

    # Formatting plot
    ax.set_xlabel('Asset 1 Price ($)')
    # Setting y label
    ax.set_ylabel('Asset 2 Price ($)')
    # Setting z label
    ax.set_zlabel('Asset 3 Price ($)')

    # Setting plot dimensions to tight
    plt.tight_layout()

    # Saving to file
    plt.savefig(fname='Homework 4/bin/correlated_bm_path.png')

    # Closing plot
    plt.close()


def partC():
    """Solution to 3(c)
    """

```

```
    strike = 100
    a_weights = np.array([1 / 3] * 3)

    # Defining simulation function
    def sim_func(x: np.array) -> float:
        # Computing terminal asset prices for each of the 3 correlated assets
        term_prices = np.array([init_prices[i] * np.exp(np.sum(
            st(x[i], sigma_vec[i], mu_vec[i])))
            for i in range(0, 3)])

        # Computing weighted basket price, and comparing to strike price
        term_price = np.sum(np.multiply(term_prices, a_weights))

        # Computing both put and call prices; returning
        call_price = np.exp(-1 * rf * ttm) * np.maximum(term_price - strike, 0)
        put_price = np.exp(-1 * rf * ttm) * np.maximum(strike - term_price, 0)

        return np.array([call_price, put_price])

    # Running simulation
    sim_results = fe621.monte_carlo.monteCarloSkeleton(
        sim_count=sim_count,
        eval_count=eval_count,
        sim_func=sim_func,
        sim_dimensionality=3
    )

    # Output dictionary
    output = dict()

    # Iterating over option types, computing MC stats for each
    for idx, opt_type in zip([0, 1], ['European Call', 'European Put']):
        output[opt_type] = fe621.monte_carlo.monteCarloStats(sim_results.T[idx])

    # Building output dataframe, formatting and saving to CSV
    out_df = pd.DataFrame(output)
    out_df.index = ['Estimate', 'Standard Deviation', 'Standard Error']
    out_df.to_csv('Homework 4/bin/q3_basket_option.csv')


def partD():
    """Solution to 3(d)
    """

    # Simulation constants
    strike = 100
    a_weights = np.array([1 / 3] * 3)
    barrier = 104

    # Defining simulation function
    def sim_func(x: np.array) -> float:
        # Computing asset prices for each of the 3 correlated assets
        asset_prices = np.array([init_prices[i] * np.exp(np.cumsum(
            st(x[i], sigma_vec[i], mu_vec[i])))
            for i in range(0, 3)])

        # Condition 1 - testing asset 2 against barrier
        if np.any(np.greater(asset_prices[1], barrier)):
            # Option value is equal to EU call on asset 2
            return np.exp(-1 * rf * ttm) * np.maximum(0,
                asset_prices[1][-1] - strike)
```

```

        # Condition 2 - testing max of asset 2 against max of asset 3
        if (np.max(asset_prices[1]) > np.max(asset_prices[2])):
            # Option value is (asset 2 term price ^2 - K)+
            return np.exp(-1 * rf * ttm) * np.maximum(0,
                np.power(asset_prices[1][-1], 2) - strike)

        # Condition 3 - testing average price of asset 2 against asset 3
        if (np.mean(asset_prices[1]) > np.mean(asset_prices[2])):
            # Option value is (avg asset 2 price - K)+
            return np.exp(-1 * rf * ttm) * np.maximum(0,
                np.mean(asset_prices[1]) - strike)

        # Otherwise, option is vanilla call option on the basket (same as (c))
        term_price = np.sum(np.multiply(asset_prices[:, -1], a_weights))
        return np.exp(-1 * rf * ttm) * np.maximum(term_price - strike, 0)

    # Running simulation
    sim_results = fe621.monte_carlo.monteCarloSkeleton(
        sim_count=sim_count,
        eval_count=eval_count,
        sim_func=sim_func,
        sim_dimensionality=3
    )

    # Building output dataframe with stats, formatting and saving to CSV
    out_df = pd.Series(fe621.monte_carlo.monteCarloStats(sim_results))
    out_df.index = ['Estimate', 'Standard Deviation', 'Standard Error']
    out_df.to_csv('Homework 4/bin/q3_exotic_option_mc.csv')


if __name__ == '__main__':
    # 3(b)
    # partB()

    # 3(c)
    # partC()

    # 3(d)
    partD()
```

question_solutions/question_3.py

C fe621 Package Code

C.1 Black-Scholes Analytical Greeks

```
from .util import computeD1D2

from scipy.stats import norm

import numpy as np


def callDelta(current: float, volatility: float, ttm: float, strike: float,
              rf: float, dividend: float=0) -> float:
    """Function to compute the Delta of a call option using the Black-Scholes
    formula.

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

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

    Returns:
        float -- Delta of a European Call Option contract.
    """

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

    return np.exp(-1 * dividend * ttm) * norm.cdf(d1)


def putDelta(current: float, volatility: float, ttm: float, strike: float,
             rf: float, dividend: float=0) -> float:
    """Function to compute the Delta of a put option using the Black-Scholes
    formula.

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

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

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

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

    return -1 * np.exp(-1 * dividend * ttm) * norm.cdf(-1 * d1)


def callGamma(current: float, volatility: float, ttm: float, strike: float,
```

```
              rf: float) -> float:
    """Function to compute the Gamma of a Call option using the Black-Scholes
    formula.

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

    Returns:
        float -- Delta of a European Call Opton Option contract.
    """

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

    return (norm.pdf(d1) / (current * volatility * np.sqrt(ttm)))


def vega(current: float, volatility: float, ttm: float, strike: float,
         rf: float) -> float:
    """Function to compute the Vega of an option using the Black-Scholes formula.

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

    Returns:
        float -- Vega of a European Option contract.
    """

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

    return current * np.sqrt(ttm) * norm.pdf(d1)
```

../fe621/black_scholes/greeks.py