Level 9 Homework
Groups C&D: Monte Carlo Pricing Methods

Student Kaushik Aryan R
Date August 17th 2025

Introduction

This report presents the analysis and implementation of Monte Carlo methods for pricing European options. The implementation leverages the Euler-Maruyama scheme to simulate stochastic differential equations and demonstrates comprehensive testing across various parameter combinations. The analysis includes both theoretical code examination and practical accuracy assessments using different numbers of time steps and simulation paths.

Part C: Monte Carlo 101

a. Source Code Analysis

The provided C++ code in `TestMC.cpp` implements the Monte Carlo method for pricing European options using the Euler-Maruyama scheme. The implementation relates directly to the theoretical framework through several key components:

SDE Definition and Theory Connection

The code parametrizes the stochastic differential equation (SDE):

$$dX = \alpha X\,dt + bX\,dW \tag{1}$$

The functions in the namespace `SDEDefinition` implement the theoretical framework:

```cpp
namespace SDEDefinition
{ // Defines drift + diffusion + data

    OptionData* data;   // The data for the option MC

    double drift(double t, double X)
    { // Drift term
        return (data->r) * X; // r - D
    }

    double diffusion(double t, double X)
    { // Diffusion term
        double betaCEV = 1.0;
        return data->sig * pow(X, betaCEV);
    }

    double diffusionDerivative(double t, double X)
    { // Diffusion term, needed for the Milstein method
        double betaCEV = 1.0;
        return 0.5 * (data->sig) * (betaCEV)* pow(X, 2.0 * betaCEV - 1.0);
    }

} // End of namespace
```

Where:

- **Drift function**: `drift(t, X)` implements $\alpha X = rX$ (risk-free rate times current price)
- **Diffusion function**: `diffusion(t, X)` implements $bX = \sigma X$ (volatility times current price)
- **Diffusion derivative**: `diffusionDerivative(t, X)`Used for the Milstein method (though code uses Euler-Maruyama)

Main Function

```
OptionData myOption;
myOption.T = 0.25;
myOption.K = 65.0;
myOption.sig = 0.30;
myOption.r = 0.08;
double S_0 = 60;
double expectedPrice = 2.13337;
myOption.type = 1;        // Put -1, Call +1

long N = nt[i];
std::cout << "Number of subintervals in time: ";
std::cin >> N;
```

Option parameters are set, along with the type of option and number of subintervals N that need to be created.

Monte Carlo Implementation

The implementation follows the theoretical Monte Carlo process:

1. **Time discretization**: Creates a uniform mesh with $N_T + 1$ points:

```
Range<double> range(0.0, myOption.T);
vector<double> x = range.mesh(NT);
double k = myOption.T / double(NT);
double sqrk = sqrt(k);
```

The time grid is represented by `range`, which is an object of the `range<double>` class. The $N+1$ grid points are generated by `x = range.mesh(N)` which gives a vector of doubles.

$$0 = t_0 < t_1, \ldots < t_n < t_{n+1} < \ldots < t_N = T \tag{2}$$

2. **Random number generation**: Uses `BoostNormal()` for Gaussian pseudo-random numbers

```
// NormalGenerator is a base class
NormalGenerator* myNormal = new BoostNormal();
```

3. **Euler-Maruyama scheme**:

```
VNew = VOld + (k * drift(x[index - 1], VOld))
    + (sqrk * diffusion(x[index - 1], VOld) * dW);
```

This implements: $Y_{n+1} = Y_n + aY_n\Delta t_n + bY_n\Delta W_n$

4. **Monte Carlo Process**:

```
for (long i = 1; i <= NSim; ++i)
{ // Calculate a path at each iteration

        if ((i / 10000) * 10000 == i)
        {// Give status after each 1000th iteration

                //std::cout << i << std::endl;
        }

        VOld = S_0;
        for (unsigned long index = 1; index < x.size(); ++index)
        {

                // Create a random number
                dW = myNormal->getNormal();

                // The FDM (in this case explicit Euler)
                VNew = VOld + (k * drift(x[index - 1], VOld))
                        + (sqrk * diffusion(x[index - 1], VOld) * dW);

                VOld = VNew;

                // Spurious values
                if (VNew <= 0.0) coun++;
        }

        double tmp = myOption.myPayOffFunction(VNew);
        price += (tmp) / double(NSim);
}

// D. Finally, discounting the average price
price *= exp(-myOption.r * myOption.T);

// Cleanup; V2 use scoped pointer
delete myNormal;
```

The process involves:

1. Outer loop that runs $NSIM$ simulations and computes MC price $NSIM$ times
2. Inner loop simulates stock path from $t = 0$ to $t = T$. Because eurpoean options do not have an early exercise feature, we only need to consider the payout of the option in period T.
3. Calculation of payoff at expiry: `myOption.myPayOffFunction(VNew)`
4. Average the price `price += (tmp) / double(NSim)` and discount it to present value: `price *= exp(-myOption.r * myOption.T)`
5. Delete the object `myNormal` from memory, and display the results of the MC simulation

b. Accuracy Analysis for Batches 1 & 2

Batch 1 Analysis ($T = 0.25$, $K = 65$, $\sigma = 0.30$, $r = 0.08$, $S = 60$)

Reference Call Price = 2.13337

N_T \NSIM	100	1,000	10,000	100,000	1,000,000
2	2.632e-01	2.421e-01	1.066e-01	8.943e-02	8.545e-02
5	6.718e-01	1.044e-02	8.194e-03	3.694e-02	3.395e-02
10	2.519e-01	5.336e-02	1.040e-02	2.220e-02	1.663e-02
100	3.845e-02	5.927e-02	4.430e-03	2.945e-03	6.578e-04
500	4.221e-02	1.101e-01	6.304e-03	1.526e-02	2.663e-03
1000	4.188e-01	3.704e-02	1.747e-03	1.128e-02	1.190e-03

Table 1: Absolute Error of MC Simulation for Batch 1 Calls

N_T \NSIM	100	1,000	10,000	100,000	1,000,000
2	4.194e-01	2.238e-02	6.010e-02	7.979e-02	6.359e-02
5	7.385e-01	1.630e-01	2.674e-03	7.224e-03	2.046e-02
10	1.782e-01	1.818e-01	4.515e-03	6.821e-04	5.536e-03
100	7.415e-01	2.234e-03	6.179e-02	2.693e-02	4.966e-03
500	1.056e-01	1.867e-01	9.126e-02	8.103e-03	5.034e-03
1000	4.566e-01	2.278e-01	9.165e-02	2.438e-02	1.043e-02

Table 2: Absolute Error of MC Simulation for Batch 1 Puts

Batch 2 Analysis ($T = 1.0$, $K = 100$, $\sigma = 0.20$, $r = 0$, $S = 100$)

Reference Call Price = 7.96557, Reference Put Price = 7.96557

N_T \NSIM	100	1,000	10,000	100,000	1,000,000
2	3.485e-01	3.754e-01	1.672e-02	1.803e-02	2.091e-02
5	1.988	1.403e-02	7.291e-02	8.823e-03	1.129e-02
10	4.771e-01	9.580e-02	1.512e-02	4.270e-03	3.654e-03
100	1.904e-01	2.300e-01	2.460e-02	2.195e-02	3.069e-03
500	1.696e-02	3.409e-01	3.299e-02	4.699e-02	4.147e-03
1000	9.024e-01	9.393e-02	4.331e-02	1.627e-02	9.719e-03

Table 3: Absolute Error of MC Simulation for Batch 2 Calls

N_T \NSIM	100	1,000	10,000	100,000	1,000,000
2	7.300e-01	8.801e-02	6.749e-02	2.638e-02	5.689e-02
5	1.297	2.913e-01	4.251e-02	6.845e-02	3.594e-02
10	1.400e-01	3.948e-01	1.822e-02	4.113e-02	2.541e-02
100	1.483	8.063e-02	9.779e-02	4.233e-02	8.824e-03
500	1.442e-01	3.213e-01	1.718e-01	7.624e-03	8.943e-03
1000	9.188e-01	3.213e-01	1.575e-01	4.117e-02	1.575e-02

Table 4: Absolute Error of MC Simulation for Batch 2 Puts

Key Observations from Experimental Results

- **Call vs Put Comparison:**
  - Both calls and puts show similar convergence patterns
  - Put options sometimes show higher absolute errors, particularly at low simulation counts
  - The symmetry in errors suggests the implementation correctly handles both option types
- **Time Steps (N_T) Impact:**
  - **Diminishing returns**: Beyond $N_T = 100$, improvements are minimal
  - **Optimal range**: $N_T = 100 - 500$ provides good balance between accuracy and computation
  - **Low N_T sufficiency**: Even $N_T = 2$ can achieve reasonable accuracy with sufficient simulations ($NSIM$)

- **Simulations ($NSIM$) Impact:**
  - **Dominant factor**: $NSIM$ has much greater impact than N_T on accuracy
  - **Convergence rate**: Errors generally decrease as $NSIM$ increases
  - **Target simulations**: 100,000+ simulations typically needed for practical accuracy
- **Batch Comparison:**
  - **Batch 1 (shorter maturity)**: Generally more stable, faster convergence
  - **Batch 2 (longer maturity)**: More volatile results, requires higher simulation counts
  - **Parameter sensitivity**: Longer time horizons amplify discretization errors

c. Stress Testing (Batch 4)

For Batch 4 parameters ($T = 30.0$, $K = 100$, $\sigma = 0.30$, $r = 0.08$, $S = 100$), achieving two decimal place accuracy (error < 0.01) requires:

- **Extreme computational requirements**: $N_T = 1,000$, $NSIM = 1,000,000,000$
- **Error scaling**: With error of 3.23×10^{-1} at $N_T = 1,000$, $NSIM = 100,000,000$, approximately $10\times$ more simulations needed
- **Practical limitation**: This demonstrates MC method's slow convergence for challenging parameter sets

The simulation output shows:

```
Simulation number: 99930000
Simulation number: 99940000
Simulation number: 99950000
Simulation number: 99960000
Simulation number: 99970000
Simulation number: 99980000
Simulation number: 99990000
Simulation number: 100000000
Pricing Error: 3.230237e-01
Number of times origin is hit: 0
```

Based on this result, I estimate that $N_T = 1,000$ and $NSIM = 1,000,000,000$ should achieve the required accuracy of 1×10^{-2}, as $10^{-3} \approx 3.23 \times 10^{-1}$.

Part D: Advanced Monte Carlo

a. Implementation of Standard Deviation and Standard Error

Two generic template functions were implemented in the `TestMC.cpp` file:

```cpp
// Function to compute standard deviation
template<typename NumType>
NumType computeSD(vector<NumType> draws, NumType r, NumType T)
{
        // This function computes the standard deviation of the Monte Carlo
        //        pricing algorithm based on a vector of draws, the risk-free rate,
        //        and time-to-expiry.
        //
        // Extract drawCount
        unsigned drawCount = draws.size();

        // Compute sum_C := \sum_j C_{T,j}, and sum_sq_C := \sum_j C_{T,j}^2
        NumType sumDraws = 0;
        NumType sumDrawsSquared = 0;
        for (unsigned i = 0; i < drawCount; ++i)
        {
                sumDraws += draws[i];
                sumDrawsSquared += pow(draws[i], 2);
        }

        return sqrt((sumDrawsSquared - (pow(sumDraws, 2)
            / drawCount)) / (drawCount - 1)) * exp(-r * T);
}

// Function to compute standard error
template<typename NumType>
NumType computeSE(vector<NumType> draws, NumType r, NumType T)
{
        // This function computes the standard error of the Monte Carlo
        //        pricing algorithm based on a vector of draws, the risk-free rate,
        //        and time-to-expiry.

        // Extract drawCount
        unsigned drawCount = draws.size();

        // Compute standard deviation
        NumType stdDev = computeSD(draws, r, T);

        return stdDev / sqrt(drawCount);
}
```

Key modifications to main loop:

```cpp
vector<double> Cs;
Cs.reserve(NSIM);

// Main Monte Carlo simulation loop
for (long i = 0; i < NSIM; ++i) {
    // ... simulation code ...

    double payoff = myOption.myPayOffFunction(VNew);    // Compute C_{T,j}
    Cs.push_back(payoff);                               // Store in vector
    price += payoff;
}
```

```
// Compute standard deviation and standard error
double std_dev = computeSD(Cs, myOption.r, myOption.T);
double std_err = computeSE(Cs, myOption.r, myOption.T);
```

b. Statistical Analysis from Output Data

N_T	NSIM	Abs. Error	Std. Deviation	Std. Error
2	100	2.632417e-01	3.702759e+00	3.702759e-01
2	1000	9.203648e-02	4.205938e+00	1.330034e-01
2	10000	1.200790e-01	4.134713e+00	4.134713e-02
2	100000	8.510446e-02	4.149745e+00	1.312265e-02
2	1000000	8.340218e-02	4.170402e+00	4.170402e-03
5	100	6.264057e-02	4.366236e+00	4.366236e-01
5	1000	1.951929e-01	4.375735e+00	1.383729e-01
5	10000	9.015876e-02	4.362438e+00	4.362438e-02
5	100000	6.333460e-02	4.317874e+00	1.365432e-02
5	1000000	2.476275e-02	4.388547e+00	4.388547e-03
10	100	3.690039e-02	4.167179e+00	4.167179e-01
10	1000	9.809618e-02	4.397541e+00	1.390624e-01
10	10000	1.478055e-02	4.496954e+00	4.496954e-02
10	100000	1.067153e-03	4.483778e+00	1.417895e-02
10	1000000	1.295198e-02	4.454495e+00	4.454495e-03
100	100	2.866685e-01	4.547652e+00	4.547652e-01
100	1000	8.267394e-04	4.335316e+00	1.370947e-01
100	10000	3.999401e-03	4.485819e+00	4.485819e-02
100	100000	7.285199e-03	4.517929e+00	1.428695e-02
100	1000000	5.307240e-03	4.508666e+00	4.508666e-03
500	100	1.870652e-01	4.087508e+00	4.087508e-01
500	1000	8.817638e-02	4.505500e+00	1.424764e-01
500	10000	4.365284e-02	4.463176e+00	4.463176e-02
500	100000	1.928258e-02	4.474316e+00	1.414903e-02
500	1000000	4.158966e-03	4.516903e+00	4.516903e-03
1000	100	5.072205e-02	4.069782e+00	4.069782e-01
1000	1000	4.779176e-02	4.464908e+00	1.411928e-01
1000	10000	9.760834e-02	4.639763e+00	4.639763e-02
1000	100000	3.135009e-02	4.565208e+00	1.443646e-02
1000	1000000	6.992468e-04	4.515452e+00	4.515452e-03

Table 5: Complete Results from TestMC.cpp Output

Key Statistical Observations

- **Standard Deviation Patterns:**
  - **Consistency across N_T**: SD remains relatively stable ($\sim 4.1 - 4.6$) regardless of time steps
  - **Convergence with NSIM**: SD converges to stable value ($\sim 4.4 - 4.5$) as simulations increase
  - **Theoretical expectation**: SD represents intrinsic option price volatility, should be independent of discretization
- **Standard Error Patterns:**
  - **$\sqrt{M}$ scaling**: SE decreases by factor of $\sqrt{10} \approx 3.16$ when NSIM increases by $10\times$
  - **Examples from data**:
    - $N_T = 2$: SE drops from 3.70×10^{-1} (100 sims) to 1.33×10^{-1} (1000 sims) to 4.17×10^{-3} (1M sims)
    - **Scaling verification**: $3.70 \times 10^{-1}/\sqrt{10} \approx 1.17 \times 10^{-1} \approx 1.33 \times 10^{-1}$

NSIM	**Expected SE Ratio**	**Observed SE Ratio ($N_T = 2$)**
$100 \to 1000$	$1/\sqrt{10} \approx 0.316$	$1.330/3.703 \approx 0.359$
$1000 \to 10000$	$1/\sqrt{10} \approx 0.316$	$0.413/1.330 \approx 0.311$
$10000 \to 100000$	$1/\sqrt{10} \approx 0.316$	$0.131/0.413 \approx 0.317$
$100000 \to 1000000$	$1/\sqrt{10} \approx 0.316$	$0.042/0.131 \approx 0.318$

Table 6: Standard Error Scaling Analysis

Accuracy vs. Statistical Measures:

- **Absolute Error**: Generally decreases with both N_T and NSIM increases
- **Standard Error relationship**: When SE $\approx$ absolute error, we're near optimal simulation count
- **Efficiency insight**: Higher N_T has diminishing returns compared to higher NSIM

Conclusions

- **Practical Conclusions:**
  - **NSIM dominates accuracy**: Increasing simulations more effective than increasing time steps
  - **SE as convergence indicator**: SE provides theoretical bound on expected error
  - **Optimal resource allocation**: Focus computational budget on NSIM rather than N_T
  - **Monte Carlo efficiency**: Method shows characteristic $O(1/\sqrt{M})$ convergence rate
- **Pattern Recognition:**
  - **Time step sufficiency**: Even $N_T = 2$ provides reasonable bias with sufficient simulations
  - **Simulation requirements**: $100,000+$ simulations typically needed for practical accuracy
  - **Resource trade-off**: $10\times$ increase in NSIM gives $\sim 3\times$ improvement in precision

Looking at the complete analysis, both increases in grid size (N_T) and number of simulations ($NSIM$) tend to decrease error. While increasing the number of simulations does not bias the estimate, we are much more likely to get a smaller draw from the distribution of absolute error when the number of simulations is larger. Increasing the number of grid points does tend to result in lower bias (smaller error), but the results are minimal. Even with $N_T = 2$ we get a good approximation to prices with very large number of simulations.

The distribution of prices itself should converge to a stable distribution, which we observe in the standard deviation values. The standard error decreases by a factor of roughly $\sqrt{\text{NSIM}}$ as we increase the order of magnitude of NSIM, confirming the theoretical $\sqrt{M}$ convergence rate of Monte Carlo methods.