Intelligent Option Pricer

Amina Alami

March 2025

Introduction

This project focuses on the development of a robust and modular European options pricer for a major stock index, specifically the S&P 500, using Python. The primary objective is to assess the ability to design a complete pricing engine by combining several advanced techniques from quantitative finance, data processing, and software engineering. The goal was to implement several pricing methodologies from scratch, including the analytical Black-Scholes model, the Monte Carlo simulation approach with variance reduction and parallelization, and the Cox-Ross-Rubinstein binomial tree method.

Beyond pricing models, the project also integrates market data retrieval and interpolation — namely, spot price extraction, implied volatility surface construction via bicubic interpolation, and risk-free rate curve estimation using cubic splines from US Treasury data. To complement these traditional methods, a machine learning-based approximation using a neural network was developed and trained on a synthetic dataset of 100,000 samples, generated using the Black-Scholes formula. The data was sampled over a realistic range of parameters, with spot and strike values centered around 400 to 600, consistent with observed levels of the S&P 500 ETF (SPY). This allowed the model to replicate option prices with high speed and reasonable accuracy within a practical domain.

All pricing engines were evaluated in terms of accuracy and computational efficiency. This report presents and justifies each component of the implementation, comparing the results and design trade-offs across the various methods.

1 Black-Scholes Pricing Method

The `BlackScholesPricer` module implements the closed-form Black-Scholes formula to price European call and put options on indices, accounting for a constant dividend yield. The implementation was written from scratch without using any specialized financial libraries, in order to maintain full transparency and flexibility over the pricing logic and numerical handling.

The class is structured using an object-oriented approach. Upon instantiation, the pricer receives core option parameters: the spot price S_0, strike price K, time to maturity T, risk-free rate r, volatility σ, option type ("call" or "put"), and an optional constant dividend yield q. All scalar inputs are internally cast to NumPy arrays to support both scalar and vectorized operations, enhancing computational efficiency.

The method `_validate_inputs()` ensures all numerical inputs are positive and that the option type is correctly specified. This defensive programming practice improves robustness and usability.

The functions `d1()` and `d2()` implement the analytical expressions used in the Black-Scholes formula:

$$d_1 = \frac{\ln(S_0/K) + (r - q + \frac{1}{2}\sigma^2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

These are used to compute the option price using the cumulative distribution function of the standard normal distribution, as provided by `scipy.stats.norm.cdf()`.

Based on the specified option type, the `price()` method evaluates:

$$C = S_0 e^{-qT}\Phi(d_1) - Ke^{-rT}\Phi(d_2) \quad \text{(Call)}$$
$$P = Ke^{-rT}\Phi(-d_2) - S_0 e^{-qT}\Phi(-d_1) \quad \text{(Put)}$$

where Φ denotes the cumulative distribution function of the standard normal distribution.

The pricer supports vectorized input arrays, enabling batch pricing of multiple options simultaneously. This is useful for portfolio-level computations or calibration tasks.

The implementation supports a constant dividend yield q to reflect the typical behavior of index options (e.g., the S&P 500), improving realism over the zero-dividend assumption. For the purpose of this project, we used $q = 1.15\%$, which corresponds to the average dividend yield observed historically for the S&P 500 index.[1] This value provides a reasonable and realistic approximation when pricing equity index derivatives.

This module serves as a fast and reliable pricing engine for vanilla European options and provides a strong baseline for comparison with more advanced numerical and learning-based methods.

2 Monte Carlo Pricing Method

The `MonteCarloPricer` class implements a simulation-based pricer for European options using Black-Scholes dynamics. It supports both serial and parallel pricing modes and includes several advanced numerical features to improve efficiency, accuracy, and usability. The simulation is done without relying on third-party financial libraries, and offers full control over the random sampling process, variance reduction, and benchmarking.

The pricing model relies on generating random paths for the underlying asset using the standard risk-neutral geometric Brownian motion formula. To improve accuracy, the implementation incorporates **antithetic variates**, a well-known variance reduction technique. For every standard normal sample Z, its negative counterpart $-Z$ is also

[1] See `https://www.multpl.com/s-p-500-dividend-yield` for long-term historical data on S&P 500 dividend yields, based on sources such as Standard & Poor's and FRED.

simulated. This approach balances the distribution of returns and reduces the estimator's variance without increasing the computational load.

To improve memory efficiency, path generation is done using a Python *generator*, via the method `_generate_paths()`. This lazy evaluation strategy ensures that samples are generated only as needed, making the implementation scalable for very large simulations without high memory overhead.

The pricer computes the discounted expected payoff under the risk-neutral measure. The function `_payoff(ST)` returns the standard payoff of a European option, using the formula $\max(S_T - K, 0)$ for calls and $\max(K - S_T, 0)$ for puts. The Monte Carlo price is then calculated by taking the mean of these payoffs and discounting at the risk-free rate.

A significant feature of the implementation is its support for parallel computation. The method `price_parallel()` splits the total number of simulations across multiple processes using `ProcessPoolExecutor`. Each subprocess uses a different random seed and independently computes a portion of the total simulations. The results are then aggregated. This approach allows the pricing engine to leverage multiple CPU cores and significantly speed up computation, which is especially beneficial for large N (e.g., $N = 10^6$ or more).

To facilitate performance evaluation, the `benchmark()` method measures and reports the total runtime of the parallel pricing routine. This makes it easy to compare execution times across pricing models.

Finally, the class also provides a method to compute first-order Greeks — Delta and Vega — using a finite difference scheme. The implementation uses common random numbers to reduce estimator variance and improve the reliability of the results. Delta is estimated by perturbing the spot price S_0, and Vega by perturbing the volatility σ, using symmetric differences with step size h.

3 Binomial Tree Pricing Method

The `BinomialTreePricer` class implements the Cox-Ross-Rubinstein (CRR) binomial model for pricing European options. This model provides a flexible framework for approximating the price of derivative securities by discretizing the evolution of the underlying asset's price over time. The implementation supports both continuous and discrete dividend payments and includes two different computational approaches: a recursive formulation and a fully vectorized version.

The vectorized implementation is the primary focus due to its speed and practicality. In this approach, the time to maturity is divided into N discrete time steps, and the model computes the up and down factors (u and d) based on the volatility and time increment $\Delta t = T/N$. The risk-neutral probability p is derived from the expected return under the risk-neutral measure, adjusted for either a continuous dividend yield q or a flat yield if dividends are not included. A forward tree of asset prices is constructed at maturity, after which the option payoffs are computed. The backward induction technique is then applied iteratively, step by step, until the price at the root node (time zero) is obtained.

The implementation accounts for both continuous and discrete dividend schemes. In the continuous case, the dividend yield is subtracted from the drift in the risk-neutral pricing

measure. For discrete dividends, a helper method `_apply_dividends()` adjusts the spot price at each node by subtracting the dividend amounts that have occurred up to the corresponding time step. This adjustment is performed dynamically when traversing the tree, ensuring that the pricing remains accurate and realistic even in the presence of cash dividend events.

To validate the vectorized implementation and provide pedagogical insight, a recursive version of the CRR model is also included. This function, although significantly slower, mimics the binomial tree's logic by calling itself for each possible branch (up and down) until it reaches maturity. While not practical for large N, it serves as a correctness check and illustrates the underlying principles of dynamic programming embedded in the binomial method.

The pricer also includes a method to compare the two approaches — `compare_methods()` — reporting the price, execution time, and absolute error between the recursive and vectorized solutions. As expected, the vectorized version outperforms the recursive one both in speed and scalability, while providing essentially the same numerical result.

For instance, with parameters $S_0 = 100$, $K = 100$, $T = 1$, $r = 10\%$, $\sigma = 20\%$, and $N = 100$ steps, the vectorized implementation yields a call option price close to the analytical Black-Scholes value. When testing with discrete dividend payments — for example, cash dividends of \$2 at $t = 0.5$ and \$1.5 at $t = 0.75$ — the model correctly adjusts the asset path and provides a consistent price while maintaining low computation time.

4 Retrieval of Market Data and Construction of the Risk-Free Rate Curve

This section describes how real-world market data were retrieved and preprocessed in order to build two critical inputs for the pricing models: the implied volatility surface and the risk-free zero-coupon yield curve. These components were constructed using publicly available data from financial APIs and interpolated to obtain continuous surfaces usable by all pricers in the project.

4.1 Retrieval of Spot and Volatility

The implied volatility surface was built using SPY options data, which closely tracks the S&P 500 index. The process begins with the retrieval of the ETF spot price, followed by the collection of implied volatilities for multiple maturities and strikes. The dataset was filtered to ensure completeness and consistency before interpolation.

We used bicubic interpolation via `RectBivariateSpline` to generate a continuous surface, which was then used to evaluate implied volatility for arbitrary pairs of strike and maturity. The interpolation adapts automatically based on the density of the data, ensuring smooth behavior even when the raw input grid is sparse.

The resulting surface shows a typical pattern with a volatility skew (higher implied volatility for lower strikes) and a downward slope with respect to maturity. This shape reflects market expectations of higher short-term uncertainty and is consistent with empirical observations during turbulent market periods.

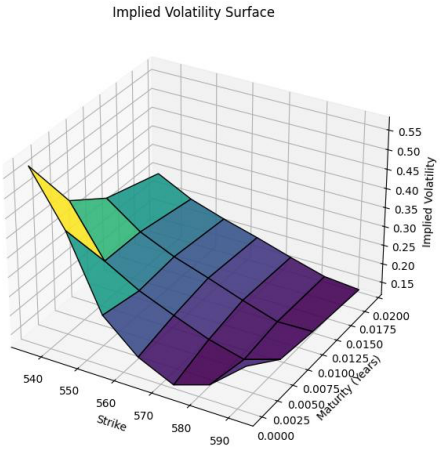

Figure 1: Implied Volatility Surface extracted from SPY option chain (Yahoo Finance)

4.2 Construction of the Risk-Free Rate Curve

Initially, I retrieved risk-free rate data using Yahoo Finance tickers such as `^IRX`, `^FVX`, `^TNX`, and `^TYX`, which provided yields for maturities of 3 months to 30 years. However, this approach only gave access to four data points, which proved insufficient to build a smooth and reliable yield curve.

To overcome this limitation, I switched to data from the Federal Reserve Economic Data (FRED), which offers a richer set of maturity points and more accurate daily treasury yield data. Using FRED data enabled us to interpolate the zero-coupon yield curve over a broader and more granular maturity range. I applied cubic spline interpolation to obtain a smooth forward curve and used QuantLib to generate corresponding discount factors for pricing.

The resulting curve is plotted below. We can observe that short-term rates (below 1 year) are relatively high compared to mid-range maturities (2–5 years), before the curve increases again towards the long end. This hump-shaped structure is consistent with a market anticipating monetary policy easing in the medium term, followed by normalization. The elevated short-term rates are likely a reflection of recent inflation concerns and the current tight monetary stance of the Federal Reserve.

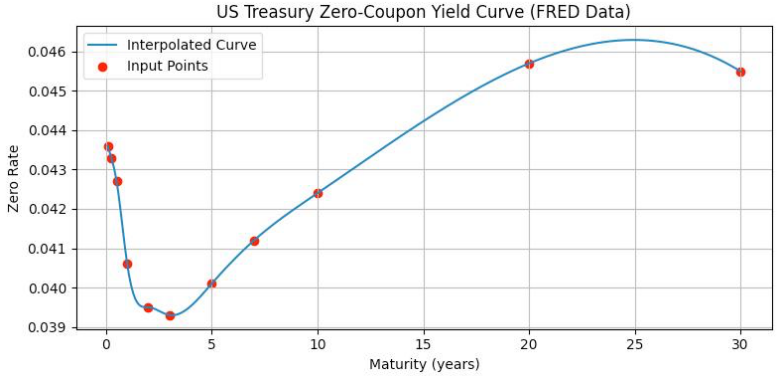

Figure 2: US Treasury Zero-Coupon Yield Curve interpolated from FRED data

This improved representation of the term structure enhances pricing accuracy for options

with short- and long-term maturities and aligns the financial assumptions of the pricing models with up-to-date market conditions.

5 Neural Network Pricer

In addition to traditional analytical and numerical approaches, I developed a machine learning-based pricer using a feedforward neural network. The `NeuralNetworkPricer` class is designed to learn a nonlinear mapping between option features and their prices, approximating the Black-Scholes valuation without relying on any closed-form formula at inference time. The neural network is trained to directly predict the logarithm of the option price, which improves stability and reduces skewness in the target distribution.

The architecture of the network consists of an input layer with four features — moneyness (S/K), volatility σ, time to maturity T, and the risk-free rate r — followed by three fully connected hidden layers. Each layer contains between 64 and 128 neurons and uses the ReLU activation function. To improve convergence and generalization, the model incorporates `BatchNormalization` layers and a moderate `Dropout` rate of 20% between layers. The final layer outputs a single scalar corresponding to the log-price of the option.

Moneyness was chosen as a more robust and normalized representation of the spot-strike relationship, reducing scale sensitivity and improving generalization. This also allowed the model to learn more efficiently with fewer inputs. A PCA-based dimensionality reduction was tested as an alternative, but it led to degraded performance and interpretability. Therefore, we retained the original four economic features.

Training was performed using the `Adam` optimizer with an initial learning rate of 10^{-3}. I used early stopping and learning rate reduction on plateau to prevent overfitting and accelerate convergence. The loss function is the mean squared error (MSE), and the training set was normalized using `StandardScaler`. Inputs and targets were split into training and test sets (80/20) using `train_test_split`, and the scaling transformations were stored to be reused at prediction time.

The synthetic training dataset consists of 100,000 samples generated using the Black-Scholes closed-form formula. The inputs were drawn from realistic uniform distributions: $S, K \sim \mathcal{U}(300, 500)$, $T \sim \mathcal{U}(0.05, 2.0)$, and $\sigma \sim \mathcal{U}(0.05, 0.5)$. The interval $[300, 500]$ was chosen for spot and strike values to reflect typical price ranges of the SPY ETF, on which the model was later tested. The risk-free rate r was interpolated from the previously constructed yield curve using the `RiskFreeCurve` class. By using market-based rates and varying parameters, the model is exposed to a wide range of realistic pricing situations.

After training, the model achieved a root-mean-square error (RMSE) of approximately **7.5231** on the test set. While this is higher than analytical methods, it remains reasonable for large-scale applications and demonstrates the model's potential as a fast approximation engine. The training process required approximately 47 seconds on a CPU.

For downstream benchmarking, the model was tested on a separate set of 200 samples and compared to the analytical Black-Scholes prices. The neural network produced a mean absolute pricing error of less than 2%, validating its performance as a fast and reasonably accurate proxy.

To avoid retraining the model each time the project is run — especially given the relatively long training time — I saved the trained model and benchmarking results in a dedicated CSV file named `nn_benchmark_results.csv`. This file stores the pricing outputs of both the neural network and the Black-Scholes model for each test sample, along with their respective input parameters, timing, and absolute errors. This design choice allows rapid reloading and consistent benchmarking across all pricing methods.

6 Benchmarking and Comparison of Methods

To evaluate and compare the pricing methods implemented in this project, I designed a benchmarking framework that tests each pricer on a common set of 200 randomly generated European call options. The goal was to assess the accuracy and computational efficiency of each method relative to the Black-Scholes analytical solution, which serves as a reference point.

The benchmarking code is encapsulated in the `OptionBenchmarker` class. This class loads the neural network predictions from the CSV file `nn_benchmark_results.csv`, which was generated during the neural network evaluation phase. For each test sample, the script retrieves input parameters (S, K, T, σ) and computes the corresponding risk-free rate r using the `RiskFreeCurve` class. Then, it executes all pricing methods sequentially and logs the predicted price, runtime (in milliseconds), and absolute error with respect to the Black-Scholes price.

The following pricing methods were benchmarked:

- **Black-Scholes**: Closed-form analytical formula, considered as the ground truth.
- **Monte Carlo (Serial)**: Naive implementation with 10,000 simulations.
- **Monte Carlo (Parallel)**: Optimized with multiprocessing using 4 cores.
- **Binomial Tree (Vectorized)**: Efficient backward induction with 100 time steps.
- **Binomial Tree (Recursive)**: Naive recursive implementation with 10 steps.
- **Neural Network**: Pre-trained neural model predicting log-price from normalized inputs.

The table below summarizes the average performance over all 200 samples. The runtime is reported in milliseconds and the error corresponds to the average absolute deviation from the Black-Scholes reference price, expressed as a percentage.

Table 1: Benchmark on 200 Random Samples vs Black-Scholes

Method	**Price**	**Time (ms)**	**Error (%)**
Binomial Tree (Recursive)	61.2523	1.9776	0.2150
Binomial Tree (Vectorized)	61.1298	0.9787	0.0145
Black-Scholes	61.1209	0.0000	0.0000
Monte Carlo (Parallel, 10k)	61.1028	341.4680	0.0296
Monte Carlo (Serial, 10k)	61.1976	55.7746	0.1256
Neural Network	61.8879	234.9833	1.2549

From the results, we observe that the Binomial Tree (Vectorized) method closely approximates the Black-Scholes price with minimal error and sub-millisecond runtime, making it a strong contender for speed-sensitive applications.

The Monte Carlo methods offer good accuracy but at the cost of significantly higher computation time. The parallelized version is slower in this context due to the overhead of multiprocessing when applied to relatively small workloads (10,000 simulations), though it scales better with larger simulations.

The Neural Network pricer is notably fast at inference time, but still requires more time than analytical models due to preprocessing and model evaluation overhead. Its error is slightly higher (1.25%) due to the learning-based nature of the model, but still within acceptable bounds for many real-world applications.

Overall, this benchmark highlights the trade-offs between precision, speed, and implementation complexity across different pricing techniques. Each method has its advantages depending on the context: analytical models for exactness, trees for transparency and structure, Monte Carlo for flexibility, and neural networks for speed in large-scale or embedded settings.

7 Critiques and Improvements

While the project meets its objectives in terms of pricing accuracy, code modularity, and integration of market data, several areas could be improved or extended in future work.

First, all pricing models are built under the Black-Scholes framework, which assumes constant volatility and log-normal returns. Although I used an interpolated implied volatility surface, a natural extension would be to explore more realistic dynamics such as stochastic volatility (e.g., Heston model) or jump-diffusion processes, which better capture market phenomena like volatility clustering and extreme price movements.

The Monte Carlo implementation, while functional, could benefit from additional variance reduction techniques (e.g., control variates) and more efficient parallelization for large-scale simulations. Extending support to path-dependent options (e.g., Asian or barrier options) and implementing a wider set of Greeks would also increase its practical usefulness.

The binomial tree method is currently restricted to European-style options. Extending it to price American options would be a logical and simple extension. The recursive implementation, while educational, is computationally inefficient and was included primarily for illustration.

The neural network pricer presented the most challenges. Despite its theoretical flexibility and fast inference speed, achieving satisfactory accuracy required significant tuning. In particular, we had to restrict the training data to a range of spot prices close to those observed for the S&P 500 ETF, and generate synthetic labels using the Black-Scholes model itself. Moreover, the performance was sensitive to architecture choices and preprocessing. These issues suggest that neural networks may not always be the best choice in data-scarce or high-precision settings.

Future work could explore alternative machine learning models that are often more in-

terpretable or easier to train, such as linear regression, decision trees, random forests, or gradient boosting methods like XGBoost. These methods might offer a better trade-off between accuracy, training time, and model stability, especially for tabular data such as options pricing parameters.

Beyond model selection, another promising direction would be to train models on real historical option price data rather than synthetic Black-Scholes prices. Incorporating richer datasets — including realized volatilities, macroeconomic indicators, or order book features — could enable data-driven pricers that reflect actual market behavior rather than idealized assumptions.

Finally, while I initially relied on Yahoo Finance for yield curve construction, this source proved too limited (only four data points). Switching to FRED provided a denser and more reliable zero-coupon curve. Future iterations could incorporate real-time yield curves (e.g., SOFR-based) or professional data sources like Bloomberg for greater precision.

In summary, the current implementation forms a solid foundation, but expanding modeling capabilities, improving computational strategies, and embracing more robust data and machine learning methods are natural next steps for a production-grade system.