Computations in Finance Assessment 1

201605543

February 26, 2026

Contents

Contents

1 Testing a Binomial Market Model

If we want to price a path independent European put option, the industry typical approach is to obtain an arbitragy free price under the Black-Scholes model. One method of approximating this computationally is by using an adjusted binomial tree stock price model like the Cox-Ross-Rubenstein (CRR) model. This can be made relatively simply by obtaining final values for each unique possible outcome and back calculating for the initial price. In order to test our adjusted CRR function, it is useful to compare the output of the function to the Black-Scholes model itself. To directly calculate the price of a put under the Black-Scholes model, we can use the formulas:

$$P_t = Ke^{-rT} \cdot \Phi(-d_2) - S_0 \cdot \Phi(-d_1)$$

Where:

$$d_1 = \frac{\log(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

And Φ is the standard normal cumulative distribution function. For an example test, we can use the inputs:

1. Strike Price (K) of £90,

2. 18 months till maturity ($T = 1.5$),
3. An initial price (S_0) of £80,
4. A volatility (σ) of 30%,
5. A risk-free rate (r) of 2.5%,
6. 50 iterations ($M = 50$).

We can write a basic Python function that calculates the exact value and then compares it to the output of our basic binomial modeller. Doing this, we get the output:

```
Price using equation: 15.777390569960389,
Price using binomial model 15.767610655878515
Error: 0.000619868921828804%
```

Implying that our model is accurate enough for a 2 digit decimal place currency. Unit testing with other known outputs shows that the approximation is reasonably accurate:

```
  import pytest
  from Main import compute_binomial_algorithm1_2

  known_output = [
    [100, 0.025, 30, 1.5, 90, 200, 7.951759601479267],
    [10, 0.06, 20, 1, 14, 5, 3.2719288406522917],
    [100, 0.05, 30, 0.5, 105, 1280, 9.805954112130756],
  ]

  @pytest.mark.parametrize(["S0", "r", "sigma", "T", "K", "M",
  "result"], known_output)
def test_6 (S0, r, sigma, T, K, M, result):
    test_val = compute_binomial_algorithm1_2 (S0, r, sigma, T, K, M)
    assert abs((test_val - result) / result) < 0.01
```

For our 4 test cases, the approximation has come within a 1% error of the true value under the Black-Scholes model.

2 Benchmarking a Binomial Market Model

In addition to accuracy, we need to consider time efficiency. The CRR approximation theoretically converges around the Black-Scholes model price value as our number of iterations, *M*, increases. However, the current time complexity of the model is $\mathcal{O}(M^2)$. Due to this, if we were to use 10,000 steps, Python needs to make 10×10^8 operations, which can easily stall the interpreter for seconds. If we were to attempt 1,000,000 iterations, Python would need to make 10×10^{12} operations. As we can observe in Figure 1, on a log scale plot, the time complexity grows linearly, since the time complexity is exponential. Consequently, there is a huge trade off between increasing certainty of convergence and time complexity. This becomes an obvious problem if the user needs to calculate multiple European put options to a reasonable level of accuracy.

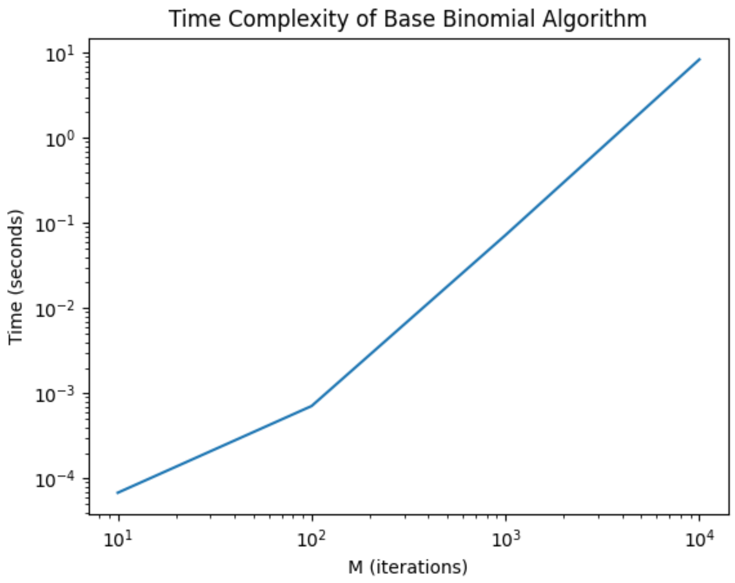

Figure 1: Time Complexity of Basic Binomial Model

3 Towards More Optimal Models

3.1 Single Array Binomial Model

If we want to optimise the function, we can begin by optimising memory usage. Since the price of a European option is path-independent, there is no need to store a list of every value the stock could take. Instead, we can optimise our memory by juggling values in a single array rather than an $M \times M$ matrix.

Suppose we fill up an array of length M with the final payoff values for the European put, in descending order. If we begin a primary loop from the second to last period returning to the first, and a second loop inside iterating from the foremost value to the period the first loop is on, we can calculate the current value of the possible outcome at the time of the first loop by taking the risk neutral measure of the outcome we are on and the next outcome. Since the second loop moves down the array, this does not alter the list in a detrimental way, allowing us to save memory.

If we observe Figure 2, we can see that the memory optimised function takes slightly less time than the base binomial model, but they have the same exponential growth rate. While the second model optimises memory use to a single M long array of floats, we still rely on two nested loops. As a result, we have exponential complexity, and the function is still $\mathcal{O}(M^2)$. In practice, since we use a small number of iterations, there is a time gain in the second. This is likely due to lower memory requirements reducing the load on the memory threshold Python 3.13 faces on my laptop. If we were to test at higher values of M, we would observe a convergence in the lines.

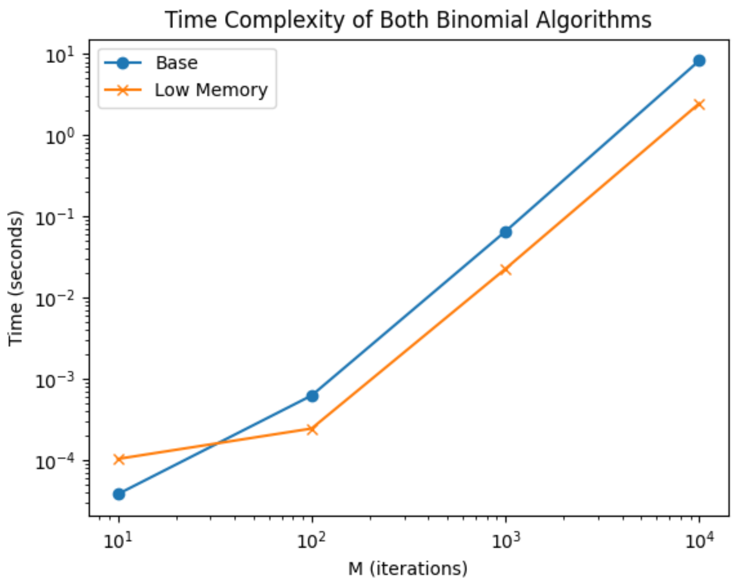

Figure 2: Comparison of base and memory optimised models

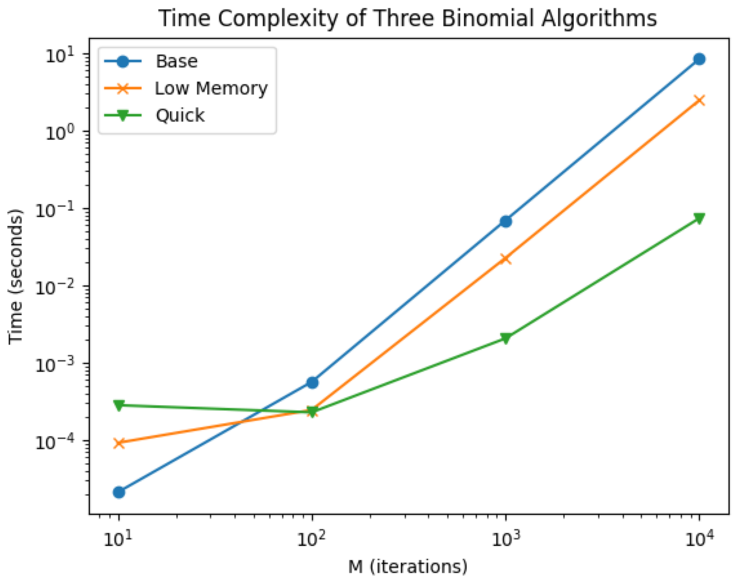

Figure 3: Comparison of all three models

3.2 Numpy Optimised Binomial Model

Unfortunately, there is little we can do to get around the nested loop calculation. Back calculating the binomial tree requires following each node, even if the nodes decrease returning to `S_0`, the gain is negligible as M tends to infinity. However, we can employ the optimised linear algebra methods of the numpy library to gain speed on the direct calculations. For efficiency, we can use the arange function declare a numpy array of length $M+1$ called j. This creates a list from 0 to M. We can then directly calculate the final period values using j, numpy will efficiently create an array with the transformation. For maximal speed on the back calculation, we can take slices of the arrays to save as much memory as possible. We can see the speed gain from the quick algorithm in Figure 3. Unit testing on the quick algorithm passes in all known test cases, and we can take an experimental average of accuracy. As we can see in Figure 4, the models have the same accuracy, meaning the speed and memory gain between models comes at little cost.

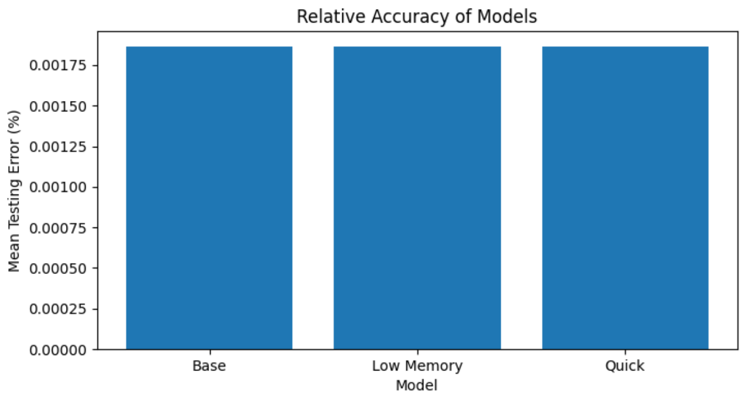

Figure 4: Relative Accuracies

4 Tilted Trees and Richardson Extrapolation

Convergence can be irregular on the typical CRR model. To fix this, we can employ a twisted tree, which has the centring calibration:

$$u = e^{\sigma\sqrt{\Delta t} + \frac{1}{M}\log(K/S_0)}$$
$$d = -e^{-\sigma\sqrt{\Delta t} + \frac{1}{M}\log(K/S_0)}$$

For the twisted tree specifically, it can be shown that:

$$2P(M) - P(M/2) = P_{\text{ex}} + \mathcal{O}(M^{-2})$$

Where $P(M)$ is the price from a binomial model using the tilted tree calibration and P_{ex} is the exact price of the put option at time 0. If we observe time complexity of the Richard extrapolation in Figure 5, we can observe that the approximation has the same gradient as the numpy model rather than the other binomial models. While the extrapolation is extremely fast, the accuracy is lower than the other models. Testing the accuracy again in 6, we can see that while the algorithm iis fast, it is not accurate like the other modles, and fits better as an approximation to be made when M becomes tremely

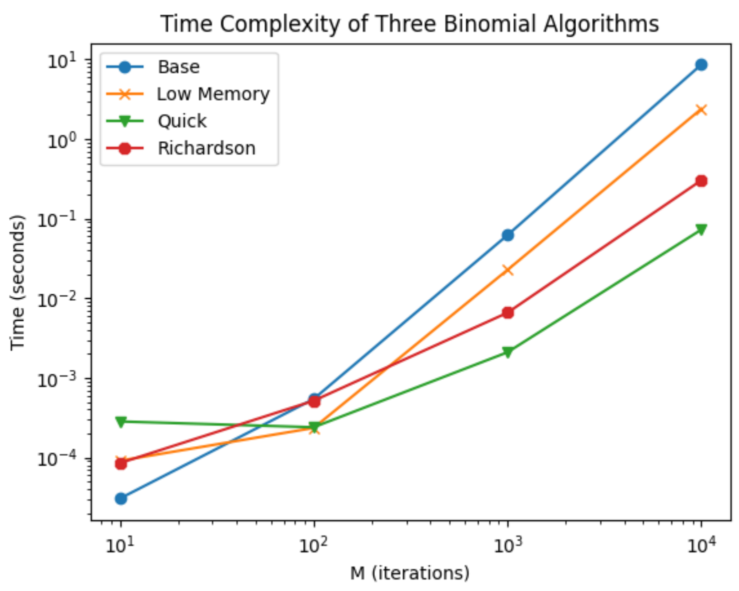

Figure 5: Time Complexity of Richardson Extrapolation

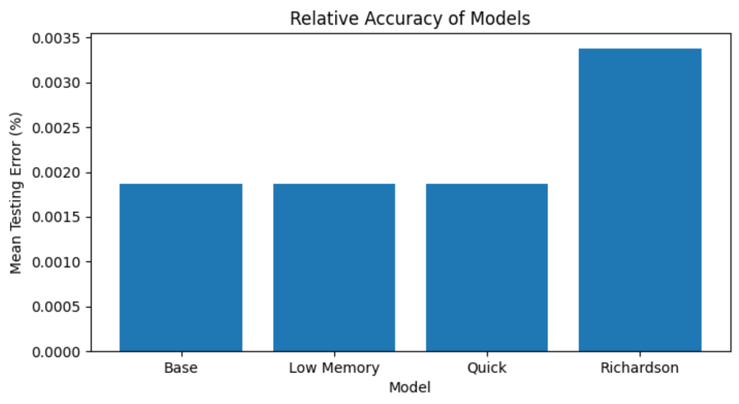

Figure 6: Accuracy of Richardson Extrapolation

5 Conclusion

Overall, we have pursued several different algorithms to price European put options. We made a base case algorithm that simply back calculated from the final values. We made a memory efficient version of this, and later a highly optimised version using the numpy package. Finally, we employed the Richardson extrapolation for $\mathcal{O}(M^{-2})$ solutions. However, these models all had their own flaws. Two of the models had exponential $(\mathcal{O}(M^2))$ time efficiency (Figure 5), while the best two have an efficiency converging on $(\mathcal{O}(M^-2))$. One of these faster algorithms was slightly more inaccurate than the others (averaging around 0.35% inaccuracy against an average less than 0.2%). Despite this, all of the models performed well enough to not have a tolerance noticeable in any currency with two decimal points. One function that employed numpy linear algebra solving notably outperformed the others on efficiency and accuracy (Figure 6).

This function could be optimised further, either by employing the Numbas compiler for Python and numpy, or by employing low level optimisations in a language like C++, but the function is close to peak optimisation in Python alone. Despite this, the speed of the algorithm is still rather slow, and the model cannot price American options at all. Due to this, pursuing avenues like Monte Carlo simulators could prove more efficient, as they could price any manner of exotic option and could potentially out-speed the CRR approximation.