An Analysis of Volatility Models in Monte Carlo Simulations.

Corey Pritchard
32932472

May 15, 2025

Abstract

This study explores how incorporating dynamic sequential volatility estimates into Monte Carlo simulations affects the pricing of plain-vanilla equity options. Volatility models are calibrated to daily S&P 500 ETF data from 1993 to 2025 and used to generate risk-neutral price paths, from which option values are calculated. We find that capturing real-world volatility features alone is insufficient and that further risk-neutral adjustments must be made in order to achieve reliable pricing.

Generative AI Disclaimer I acknowledge that I have used OpenAI's ChatGPT platform as a tool in the preparation of this report. Specifically, I used ChatGPT as an aid in my research, to locate relevant references, and to refine the wording of certain passages. All final writing, analysis, and conclusions presented here are my own, and I take full responsibility for the content of this report.

Contents

1 Introduction

Volatility is a central concept in financial modelling, describing how quickly and widely asset prices can fluctuate in either direction. Markets with high volatility, such as equity markets, experience large and sudden price swings, whereas lower-volatility markets, such as bond markets, display more stable price movements.

A common practice is to work with logarithmic returns rather than raw prices directly, because log returns tend to exhibit stationarity, unlike prices, and are therefore more suitable for time series modelling. Let S_t be the asset price at time t and S_0 be an initial reference price, then the log return r_t is calculated as:

$$r_t = \ln\left(\frac{S_t}{S_0}\right) \tag{1}$$

A standard estimate of volatility is the sample standard deviation of these log returns.

In classical models such as Geometric Brownian Motion (GBM), log returns are assumed to follow a normal distribution with constant drift μ and constant volatility σ. In reality, this simplification leads to models that underestimate extreme market movements, because real-world returns are known to exhibit the following properties:

1. **Leptokurtosis**: Real world log returns tend to have fatter tails than the normal distribution, meaning that large price swings occur more frequently than a standard model would suggest.
2. **Volatility clustering**: Periods of high volatility tend to cluster together, followed by periods of relatively low volatility, contradicting the assumption of a single constant σ.
3. **Leverage effect**: Negative shocks tend to produce disproportionately larger increases in volatility compared with similarly sized positive shocks, often driven by heightened risk aversion or market fear.

In this study, we investigate various methods of simulating hypothetical paths an asset's return could follow by incorporating time-varying volatility models in Monte Carlo simulations. Presenting both classical and machine learning approaches, we aim to capture the fat-tailed, clustered nature of market returns, allowing us to simulate more accurate return distributions.

A practical way to evaluate these simulated return distributions is to compare the resulting option prices, derived from our model's return paths, with actual market quotes across the entire options chain. Since market prices embed market expectations of volatility and tail risk, the discrepancy between simulated and observed option prices provides a measure of how well our model captures the behaviour of market implied volatility.

In the following sections, we will first review the definition of options and the existing pricing methods, then discuss traditional and machine learning approaches to volatility modelling. We go on to introduce our methodology for forecasting volatility, explain how these

forecasts are integrated into Monte Carlo simulations, and finally evaluate the simulated option prices against market data.

2 Literature Review

In this section, we start by discussing the literature on option pricing to highlight why developing dynamic volatility models is relevant to producing robust Monte Carlo simulations. Next, we will review the literature surrounding volatility modelling, grouped by classical and machine learning approaches.

2.1 Foundations of Option Pricing

An option is a derivative contract that gives its holder the right, but not the obligation, to buy (a call) or sell (a put) the underlying asset at a predetermined price (the strike) on or before a specified date (the expiration). European options can only be exercised at expiration, resulting in a straightforward payoff, whereas American options may be exercised at any time prior to expiration. Exotic options introduce additional complexities, such as the Asian option whose payoff depends on the entire price path.

Option pricing theory operates under the risk-neutral measure, $\mathbb{Q}$, a probability measure under which the expected growth rate of any asset is assumed to be the risk-free interest rate r. Under risk-neutral pricing, the present fair value of an option is the expected payoff discounted at the risk-free rate. The Black-Scholes-Merton (BSM) model provides closed-form solutions for European option prices, derived under the risk-neutral measure $\mathbb{Q}$ [13]. Volatility σ is a parameter in the model whose value is assumed to be constant over the option's life while in reality, volatility is a stochastic process that quantifies the uncertainty of future prices: higher volatility increases the likelihood of extreme price moves, which generally raises the value of options, specifically those that are far out-of-the-money or have long maturities. Even a small change in volatility can significantly affect an option's theoretical price. For instance, an asset with higher volatility will have greater chance of its price moving deep-in-the-money or out-of-the-money before expiration, affecting the option's payoff distribution.

Despite its simplifying assumptions, BSM was revolutionary. Upon its introduction, the Chicago Board Options Exchange launched trading in standardised options shortly thereafter [5]. The closed-form BSM prices are:

$$C = S_0 N(d_1) \; - \; K e^{-rT} N(d_2), \tag{2}$$

$$P = K e^{-rT} N(-d_2) \; - \; S_0 N(-d_1), \tag{3}$$

where

$$d_1 = \frac{\ln\left(S_0/K\right) + \left(r + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}, \qquad d_2 = d_1 - \sigma\sqrt{T}. \tag{4}$$

The availability of closed-form "Greeks" (the partial derivatives of price) also made real-time hedging and risk management viable.

Recognising the importance of volatility, much research has focused on modelling how volatility behaves over time. Volatility in financial markets tends to be heteroscedastic,

changing over time rather than constant, and exhibits volatility clustering where periods of high volatility are followed by high volatility and periods of low volatility are followed by low volatility.

Since constant-volatility GBM fails to reproduce empirical features of returns, modern approaches replace the single parameter σ with stochastic, local, or time-dependent volatility processes. Monte Carlo simulations allow us to use such forecasts directly in pricing by simulating the underlying price path step by step, updating the volatility at each step based on a model. This flexibility means Monte Carlo simulations can handle even exotic options with path-dependent payoffs. For example, an Asian option has a payoff dependent on the average price over a period rather than the price at a single expiration price. Traditional closed-form solutions are difficult for such payoffs, but Monte Carlo simulations can solve this by simply changing the payoff function used on the resulting price paths. The framework for pricing remains the same regardless of payoff complexity, making Monte Carlo a powerful tool when combined with volatility models.

2.2 Classical Volatility Models

2.2.1 GARCH Models

The literature of econometrics present classical volatility models that has been developed to capture the observed patterns in asset return volatility including Engle's Autoregressive Conditional heteroscedasticity (ARCH) model [8] which estimates conditional volatility dependent on past shocks, ε. Bollerslev [2] generalised this approach by creating the Generalised Autoregressive Conditional heteroscedasticity (GARCH) model to include previous volatility σ_{t-1} as a parameter. The GARCH(1,1) model is a simple and widely used specification which can be written as:

$$r_t = \mu + \varepsilon_t \tag{5}$$

$$\sigma_t^2 = \omega + \alpha\varepsilon_{t-1}^2 + \beta\sigma_{t-1}^2 \tag{6}$$

where:

- σ_t^2: the conditional variance at time t.
- ε_{t-1}^2: the squared residual (shock) from the previous time step.
- σ_{t-1}^2: the variance from the previous time step.
- ω, α, β: parameters to be estimated.

The GARCH(1,1) model effectively says that today's variance is a constant ω plus a fraction of yesterday's squared shock and a fraction of yesterday's variance. This recursion produces volatility clustering, making high volatility likely to persist for several periods. GARCH

models assume a constant mean, μ. and focus on modelling the variance of returns over time, which is perfect for our goal of simulating a stochastic volatility path.

Although the standard GARCH(1,1) model is both effective and parsimonious, it exhibits notable limitations. In particular, it assumes that negative and positive return shocks of equal magnitude have the same effect on future volatility. Empirical evidence shows that large negative shocks tend to produce greater increases in volatility than comparably large positive shocks. This asymmetry, called the leverage effect, has motivated the development of several GARCH extensions such as:

- **Exponential GARCH**: Nelson [15] introduced the EGARCH model, which models the logarithm of variance to ensure positivity and includes an explicit term for the leverage effect. An EGARCH(1,1) model takes the form:

$$\ln(\sigma_t^2) = \omega + \beta \ln(\sigma_{t-1}^2) + \alpha \frac{\varepsilon_{t-1}}{\sigma_{t-1}} + \gamma \left| \frac{\varepsilon_{t-1}}{\sigma_{t-1}} \right| \tag{7}$$

  where the parameter γ allows positive and negative shocks to have different effects on volatility. By modelling $\ln\left(\sigma_t^2\right)$, EGARCH ensures that the variance is always positive without needing non-negativity constraints.

- **Threshold GARCH**: TGARCH, also known as the GJR-GARCH model after Glosten, Jagannathan and Runkle, adds a term that activates when the returns are negative to explicitly capture the leverage effect. The TGARCH(1,1) model is given by:

$$\sigma_t^2 = \omega + \alpha \varepsilon_{t-1}^2 + \gamma \varepsilon_{t-1}^2 \mathbb{I}(\varepsilon_{t-1} < 0) + \beta \sigma_{t-1}^2 \tag{8}$$

  where $\mathbb{I}(\varepsilon_{t-1} < 0)$ is an indicator that is 1 if the previous return shock was negative and 0 otherwise. This model ensures that volatility increases more after negative returns than after positive returns of equal size.

The above GARCH variants are considered classical econometric models for volatility. They are interpretable and parsimonious and have been used extensively in forecasting volatility. However, they typically assume that a single set of parameters governs the entire dataset. This can be problematic if the statistical properties of the market change over time, for example shifts between low-volatility and high-volatility periods. In such cases, a single GARCH model may not fit all periods well. This shortcoming leads us to consider models that allow for regime changes in volatility behaviour.

2.2.2 Regime-Switching Framework

Financial time series often exhibit distinct regimes, for example a stable low-volatility regime versus a turbulent high-volatility regime, corresponding to different market conditions.

Regime-switching models capture these structural changes by allowing model parameters to change depending on an unobserved state classification variable. Hamilton [11] pioneered

the use of regime-switching models in economics by introducing a Markov-switching autoregressive model to capture business cycle phases. In the context of volatility, a Regime-Switching GARCH (RS-GARCH) model extends the standard GARCH model by incorporating latent regimes. This means the parameters ω, α, β of the GARCH process can differ depending on the regime. A Markov chain is usually used to govern transitions between regimes, with a transition probability matrix defining the likelihood of switching from one state to another each period.

By allowing volatility dynamics to vary across regimes, RS-GARCH models can better accommodate nonlinear dynamics and sudden changes in volatility structure that single-regime models miss. For example, in a two-regime GARCH model, one regime might have parameters that produce low persistent volatility, capturing quiet market conditions, and another regime might have parameters for highly persistent, large volatility capturing crisis conditions. The model can switch between these sets of parameters as market conditions evolve.

Regime-switching models are powerful in capturing reality, but come with challenges. Each additional regime introduces a new set of GARCH parameters, greatly increasing the number of parameters to estimate. With limited data, this can lead to overfitting and wide parameter uncertainty. The computational complexity of estimation is also higher than single-regime models, as filtering through many possible state sequences is intensive. In our work, we consider a basic two-regime GARCH(1,1) model, a common choice in the literature [19].

2.3 Machine Learning Methods

Machine Learning (ML) techniques have been applied to financial time series for decades, aiming to detect complex patterns and improve forecasts of returns and volatility. Unlike parametric models with a fixed functional form like GARCH, machine learning models can, in principle, approximate a wider range of nonlinear relationships given sufficient data. Early applications of ML to financial forecasting involved relatively simple methods. For example, researchers used linear and nonlinear regressions to predict asset returns and volatility. These can be extended to methods like support vector regression (SVR) [17] which can capture nonlinear relationships while being robust to outliers. Tree-based models, such as decision trees and their ensembles, are also used to partition the input feature space into regions with piecewise constant predictions, which is easy to interpret but can overfit. Ensemble approaches improve on this with random forests [4] by training many trees on random subsets of data and average their predictions, reducing variance and improving generalisation. Gradient boosting machines like XGBoost [6] and LightGBM [14] build trees sequentially, each new tree focusing on the previous trees' errors, yielding a powerful predictive model at the cost of interpretability and risk of overfitting if not carefully regularised.

In recent years, deep learning has gained traction in time series analysis due to improvements in computation and the availability of large datasets. Deep learning models, such as

neural networks with multiple layers, can capture highly nonlinear and complex patterns. Recurrent Neural Networks (RNNs) are particularly relevant for sequence data as they process sequences of inputs and maintain an internal state to capture memory of past inputs. The Long Short-Term Memory (LSTM) network is a type of RNN introduced by Hochreiter and Schmidhuber [12] to address the vanishing gradient problem and works can retain information over long sequences. LSTMs have cells with gating mechanisms that learn the information to keep or discard, enabling the network to learn long-term dependencies. Fischer and Krauss [9] demonstrated that LSTM networks outperform many traditional methods in extracting signal from noisy financial time series.

Another class of deep models are Convolutional Neural Networks (CNNs), which are extremely successful in image processing but have also been applied to time series [20]. A CNN can be used on one-dimensional financial data by treating the time series like an 'image' in time and applying convolutional filters over time to extract local features or short-term patterns. These local features can then be fed into dense neural layers or combined with RNNs. In fact, the hybrid CNN-LSTM architectures have shown promise, with the CNN layers capturing short term, local structures within a rolling window of returns, and the LSTM layers capture longer-term dependencies and trends. Vidal and Kristjanpoller [18] found that a CNN-LSTM hybrid model was effective in forecasting volatility of gold prices, indicating that combining these methods can leverage both short and long-term information.

Beyond individual models, ensemble learning can be applied in the ML context by combining multiple models' predictions. For example, one might train several different ML models and then combine their volatility forecasts, either by averaging or by using these estimates as inputs in a meta-learning approach. The idea is that different models might capture different aspects of the data, so a combination could be more robust. However, ensembles can propagate biases and errors from one model to the other and become overly complex.

When applying ML methods, a crucial step is feature engineering where relevant features are constructed to contain relevant information for predicting future volatility. While many methodologies would use macro-economic features, we are limited to features derived from historical prices because each future timestep in the simulations will need a new set of inputs. The features will be discussed in detail later in the methodology.

3 Data and Preprocessing

A large amount of data is required to calibrate models and evaluate their option-pricing performance. Daily historical prices for the S&P 500 via the SPDR S&P 500 ETF (SPY) from 1993 through 2025 are collected from Yahoo Finance using the Python package 'yfinance'. Using SPY provides a continuous price series, as well as historical dividend yields, since the S&P 500 cannot be traded directly. These daily prices are used to calculate daily log returns, forming the basis for volatility modelling.

Option prices are also collected from Yahoo Finance. These include closing prices for a range of strike prices and expirations for both calls and puts.

The risk-free rate used in each simulation depends on the date to expiration, for example, to price a short-term option we use a 30-day Treasury yield, whereas for a longer-term contract we use a correspondingly longer-term Treasury yield. These yields are collected directly from the U.S. Department of the Treasury website.

After data collection, several preprocessing steps are undertaken:

- **Returns Calculation**: We convert the daily price series $\{S_0, \ldots, S_T\}$ of SPY into a series of daily log returns $r_t = \ln\left(\frac{S_t}{S_0}\right)$. Log returns are preferred due to their time additive properties and because they are expected to be stationary.
- **Stationarity Check**: To verify the assumption that log returns are stationary, we use the Augmented Dickey-Fuller [7] (ADF) test. With a test statistic of -16.60 and a p-value of 1.77×10^{-29}, the ADF test indicates that the null hypothesis of a unit root can be rejected, confirming that the log returns are stationary in mean. This justifies using models that assume a stable long-run mean, essential for GARCH modelling.
- **heteroscedasticity Check**: To justify our use of the GARCH model, we need to examine if the volatility is auto-correlated. A common test is Engle's ARCH LM test [8], which tests whether current volatility depends on the past squared residuals. The test returns an extremely large LM statistic (1991.86) and a p-value effectively equal to zero, strongly rejecting the null hypothesis of homoscedasticity. This supports the use of GARCH type models.

Since our data shows stationarity in the mean with heteroscedastic volatility, we confirm the need for dynamic volatility models and outline our methodology for generating return distributions using dynamic volatility estimates and how to use these distributions to derive option prices.

4 Methodology

Our option pricing methodology involves using Monte Carlo simulations of the underlying asset price, in our case SPY, while incorporating various volatility forecasting models to generate the volatility at each time step. We first describe the general Monte Carlo pricing framework, then detail the specific models for volatility that we compare:

- Constant volatility model as a baseline.
- A standard GARCH model.
- Regime-Switching GARCH.
- Machine learning approaches.

For each model, we explain how it is calibrated and how it is used within the Monte Carlo simulations.

4.1 Monte Carlo Option Pricing under the Risk-Neutral Measure

Monte Carlo simulation for option pricing involves generating a large number of hypothetical price paths for the underlying asset, then computing the associated option payoff for each path resulting in a distribution of payoffs for a given strike and time to expiration. Under the risk-neutral valuation we can estimate the price of an option by calculating the present value of its expected payoff. In practice, this expectation is approximated as the mean of the simulated payoff distribution. If we simulate N independent price paths for the underlying asset starting from today ($t = 0$) to expiration ($t = T$), we have a series of payoffs $P^{(i)}$ for the option along path i at maturity. Then, the option price is approximated as:

$$\text{Option Price} \approx e^{-rT} \frac{1}{N} \sum_{i=1}^{N} P^{(i)} \tag{9}$$

where r is the continuously compounded risk-free rate, assumed constant for the option's duration and given by the Treasury rate closest to the option's duration. The factor e^{-rT} discounts the expected payoff back to the present value, resulting in today's fair price.

Because we are doing simulations under the risk-neutral measure, we must ensure that the drift of the underlying price process in our simulation is given by:

$$\mu_{\mathbb{Q}} = r - q \tag{10}$$

where q is the continuous dividend yield of the underlying. This is because in a risk-neutral framework under $\mathbb{Q}$, all assets grow on average at the risk-free rate minus any payouts. Since the SPY does pay a dividend, this needs to be accounted for in the drift.

The stochastic process for the underlying asset is the same for all of our models, with the only difference being the method of calculating volatility σ_t. The stochastic differential equation for the evolution of price under risk-neutral $\mathbb{Q}$ is given by [13]:

$$dS_t = (r - q)S_t dt + \sigma_t S_t dW_t \tag{11}$$

where dW_t is a Wiener process, also known as Brownian motion. In this general form, σ_t may be time-varying, compared to the baseline GBM model that assumes $\sigma_t = \sigma$.

In our case, the simulations are discrete with daily timesteps. To implement this SDE, we must use the discrete approximation for the asset price update over one time step, given by [10]:

$$S_{t+\Delta t} = S_t \exp\left(\left(r - q - \frac{1}{2}\sigma_t^2\right)\Delta t + \sigma_t\sqrt{\Delta t}Z\right) \tag{12}$$

where $Z \sim N(0,1)$ is a sample from the standard normal distribution and $\Delta t = \frac{1}{252}$ to reflect daily timesteps, where there are 252 trading days in a year. This formula comes from the analytical solution for the GBM SDE, with the constant volatility replaced for a time-varying estimate of volatility.

The payoff $P^{(i)}$ for each simulation path depends on the type of option. For a European option with strike K, the payoff for a call is given by $P^{(i)} = \max\left(S_T^{(i)} - K, 0\right)$ and the payoff of a put is given by $P^{(i)} = \max\left(K - S_T^{(i)}, 0\right)$. Exotic options have more complex formulas for $P^{(i)}$, for example an Asian payoff replaces the final price $S_T^{(i)}$ with the average price $\bar{S}^{(i)}$, incorporating the whole simulated path rather than just the final price. Monte Carlo handles these different payoffs seamlessly by applying an arbitrary payoff function to each generated path. The flexibility of Monte Carlo simulations means vanilla and exotic options can be priced within the same simulation framework. In our case, we will use vanilla European options since the exotic market is less liquid, resulting in fewer and less reliable market prices for comparisons.

In summary, our Monte Carlo pricing procedure is:

- Simulate N independent price paths $\{S_0, \ldots, S_T\}$ using risk-neutral drift and a new Gaussian random shock at each time step. The volatility σ_t used in each step is generated by the selected model.
- Take the average payoff across the N paths and discount it by e^{-rT} to obtain the present value of the option.

The primary objective of our methodology is to use different volatility predictions in our simulations to see how they affect the resulting payoff distribution, comparing the different models against baseline constant volatility and true market prices.

4.2 Geometric Brownian Motion

Before introducing more complex volatility models, we establish a baseline using the GBM model, a standard model typically used to simulate the underlying price in Monte Carlo simulations. Under GBM, the asset's volatility is constant σ_{GBM}, estimated by the sample standard deviation of log returns, with constant drift $r - q$. From our historical data, we have $\sigma_{GBM} = 0.01181$, corresponding to an annualised volatility of $\sigma_{GBM} \times \sqrt{252} = 18.7\%$

The GBM model is simple and computationally cheap, however the key assumption of constant volatility ignores volatility clustering and regime changes that exist in the true log returns dynamics. GBM is useful for providing a benchmark as ideally any advanced model should outperform this baseline pricing method. With constant volatility, we expect these simulations to misestimate the true distribution tails, leading to mispriced options especially for strike prices that are further from the starting price and for longer-term options.

4.3 GARCH

Our first model for σ_t in the simulation is a standard GARCH(1,1) model. This model is fit to historical returns under the real-world measure $\mathbb{P}$, but for pricing we will use it under the risk-neutral measure $\mathbb{Q}$ by adjusting the drift. For simplicity, we will assume the dynamics of volatility are the same under P and Q.

To fit this model we need to estimate the parameters $\mu, \omega, \alpha, \beta$ using the function 'arch_model' from the Python package 'arch', which finds the set of parameters that maximises the log likelihood function given our historical data.

Since daily log return values are small there can be numerical instability in the optimisation. To fix this issue, we fit the model on log returns multiplied by 100, as suggested by the optimiser's output when using unscaled log returns. This scale will affect the model parameters so in order to use this model to generate unscaled returns, we need to rescale them accordingly. For the mean parameters μ, we can simply divide our estimate by the scaling constant 100 since the expected value is a linear function, meaning $\mathbb{E}(aX) = a\mathbb{E}(X)$. This mean parameter is used to calculate the shock term $\varepsilon_t = r_t - \mu$, however the risk neutral measure means this parameter is not used in the drift term $\mu_{\mathbb{Q}}$. The ω parameter is a constant intercept term in the variance equation, meaning we need to rescale by the squared constant, rather than the constant itself, since $\mathrm{Var}(aX) = a^2\mathrm{Var}(X)$. The parameters α and β are dimensionless proportions so they do not need to be scaled.

4.4 Regime-Switching GARCH

We assume the market is being governed by a hidden Markov chain with transition matrix P. In the case of two states, the matrix $P \in \mathbb{R}^{2\times 2}$ can be described with two parameters p_{00} and p_{11}:

$$P = \begin{bmatrix} p_{00} & 1-p_{00} \\ 1-p_{11} & p_{11} \end{bmatrix} \tag{13}$$

Where $p_{ii} = \mathbb{P}[k_{t+1} = i \mid k_t = i]$ is the probability of 'transitioning' from state i to state i, which means that the state has not changed since the previous time step.

During the Monte Carlo simulations, we can use this transition matrix to determine which

GARCH parameters to select when generating the volatility.

$$\sigma_{t,k}^2 = \omega_k + \alpha_k \varepsilon_{t-1}^2 + \beta_k \sigma_{t-1}^2 \tag{14}$$

. The filtering procedure follows these steps to compute the log-likelihood of a Hidden Markov Model. In this framework, the Hamilton filter recursively computes the filtered probabilities of being in each hidden state at each time step, then it computes the likelihood of the observed data given the model parameters, including the transition probabilities. We can incorporate the GARCH mechanics by defining the distribution of log returns as:

$$r_t \sim N(0, \sigma_{t,k}^2) \tag{15}$$

where $\sigma_{t,k}^2$ is the GARCH conditional volatility that depends on the state.

4.4.1 Hamilton Filter Steps

Let θ represent the entire set of parameters for the GARCH model $(\omega_k, \alpha_k, \beta_k)$, and the transition probabilities p_{ij}.

The joint probability density of the observed data can be written as:

$$p(Y_1, Y_2, \ldots, Y_T \mid \theta) = \prod_{t=1}^{T} p(Y_t \mid Y_{1:t-1}, \theta). \tag{16}$$

Since the regime k_t is latent, we must sum over all possible regimes:

$$p(Y_t \mid Y_{1:t-1}, \theta) = \sum_{k=1}^{K} p(Y_t \mid k_t = k, Y_{1:t-1}, \theta)\, \mathbb{P}(k_t = k \mid Y_{1:t-1}, \theta). \tag{17}$$

To compute $\mathbb{P}(k_t = k \mid Y_{1:t-1}, \theta)$, we use a filtering procedure to estimate the latent regimes. Define $\pi_{t|t}(k) = \mathbb{P}(k_t = k \mid Y_{1:t}, \theta)$ as the filtered probability of being in regime k at time t. The Hamilton filter proceeds as:

1. **Initialization:** Set $\pi_{0|0}$ to a uniform vector.

2. **Prediction:** Prior to observing Y_t, the predicted probabilities are:

$$\pi_{t|t-1} = \pi_{t-1|t-1} P. \tag{18}$$

3. **Update:** Define the regime-specific density:

$$f_k(Y_t) = p(Y_t \mid k_t = k, Y_{1:t-1}, \theta), \tag{19}$$

which is Gaussian with constant mean for each state and variance given by the GARCH model in regime k:

$$f_k(Y_t) = \frac{1}{\sqrt{2\pi}\sigma_{k,t}} \exp\left(-\frac{(Y_t - \mu_{k,t})^2}{2\sigma_{k,t}^2} \right), \tag{20}$$

where $\mu_{k,t}$ and $\sigma^2_{k,t}$ depend on the regime k.

The unconditional density at time t is:

$$p(Y_t \mid Y_{1:t-1}, \theta) = \sum_{k=1}^{K} \pi_{t|t-1}(k) f_k(Y_t). \tag{21}$$

Update the filtered probabilities:

$$\pi_{t|t}(k) = \frac{\pi_{t|t-1}(k) f_k(Y_t)}{\sum_{\ell=1}^{K} \pi_{t|t-1}(\ell) f_\ell(Y_t)}. \tag{22}$$

The log-likelihood is:

$$\ell(\theta) = \sum_{t=1}^{T} \log \left(\sum_{k=1}^{K} \pi_{t|t-1}(k) f_k(Y_t) \right). \tag{23}$$

Maximum Likelihood Estimation (MLE) involves finding

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta), \tag{24}$$

or equivalently minimising the negative log-likelihood

$$\hat{\theta} = \arg\min_{\theta} -\ell(\theta), \tag{25}$$

This optimisation requires numerical methods and constraints such as positive variance parameters and valid transition probabilities.

4.5 Machine Learning

In contrast to the econometric models, we employ a deep learning approach for volatility forecasting. We choose a hybrid Convolutional Neural Network-Long Short-Term Memory (CNN-LSTM) architecture. This is a complex model compared to the simpler models discussed earlier like support vector regression or decision trees. A deep network is used to learn complex, dynamic patterns hidden in the data. Recurrent Neural Networks, particularly the LSTM used, maintain an internal state to track long-term dependencies in a sequence, while the convolutional layer extracts local patterns and short-term signals in the data.

4.5.1 Feature set

All inputs are engineered from the price itself so that during Monte Carlo simulations, every feature will be available during the future steps without having to predict exogenous variables. The feature set includes:

- Log returns.
- Rolling mean, standard deviation, skewness and kurtosis of log return.

- Squared log returns as a proxy for volatility.
- Rolling mean, standard deviation, skewness and kurtosis of squared log return.

Each feature is normalised to have zero mean and unit variance, using a 'scikit-learn StandardScaler' object fitted only on the training data to prevent look-ahead bias.

The model is provided with a rolling window of the past 60 days, providing plenty of information to predict the volatility, value or class, of the next timestep.

4.5.2 CNN-LSTM architecture

The hybrid model combines a convolutional neural network at the start, to capture local temporal patterns, with a recurrent neural network, specifically LSTM, after to capture longer range patterns and regimes.

1. A one-dimensional CNN with 32 kernels of width 3 scans the sequence, learning short-run patterns such as volatility or price spikes.
2. Layer normalisation stabilises the CNN output before proceeding.
3. LSTM layer with 24 cells processes the pooled feature map and maintains an internal memory of medium and long term patterns.
4. An additional LSTM layer with 12 cells to capture any long term patterns missed by the first layer.
5. A 16 node dense layer with ReLU activation to interpret the CNN-LSTM output.
6. A dropout regularisation layer than effectively ignores 20% of the dense layer's output, reducing overfitting risk.
7. The output layer chosen depends on whether regression or classification is chosen. The regression output is a single neuron with linear activation while the classification output is three neurons with softmax activation, representing the probabilities of the three volatility classes.

4.5.3 Target Variable

We specify two closely related models, one regression and one classification, to predict next day volatility within the simulations. Both tasks share the same input features and network architecture described above. The only difference is the final layer of the network, where the regression model has a single neuron output with a linear activation and the classification model has multiple neuron outputs with softmax activation.

For the regression task the network is trained to forecast next-day variance, estimated as the realised volatility $\sigma_{t+1}^2 = r_{t+1}^2$ [1], where r_{t+1} is the forecasted log return tomorrow. By predicting realised volatility rather than log return directly, we don't have to worry about

the direction of the market as the square will remove any negative sign in front of the log returns.

However, a common challenge when forecasting continuous values like volatility is the tendency of a model to simply predict persistence, meaning the output value is close to the last observed volatility with no meaningful pattern. This can minimise error but fail to actually model any regime changes. To address this and to explicitly capture volatility regimes, we also consider a classification approach.

For the classification task, we categorise the volatility into distinct regimes. Specifically, we take the dsitribution of volatility values in the training set and partition it into three groups using the 33rd and 67th percentiles as thresholds, yielding three approximately equally sized groups of low, medium and high volatility observaions. The partitioning was done this way to help improve convergence as partitioning into unbalanced groups lead to only the dominant group being accurately predicted.

Each training observation is labelled according to which regime its next-day volatility σ_{t+1} belongs to, and the same CNN-LSTM architecture is used except now the final layer produces a probability distribution over the three classes via softmax activation instead of a single value prediction. We train the model with standard cross-entropy loss, encouraging the network to correctly classify the volatility regime of day $t+1$.

Both the regression and classification models operate on the same feature input and network structure, allowing the model to lean consistent internal representations of market volatility dynamics with the key difference being how that representation is ultimately used: either to predict a volatility forecast or classify the coming day into a regime. By examining both approaches, we can evaluate which is more effective at capturing volatility behaviour.

4.5.4 Simulations

The simulations operate sequentially where at each simulated day t we feed the model a rolling window of the price derived features, where the initial values will come from true historical values and are subsequently recomputed using the simulated price paths. For the regression model, we use its output $\hat{\sigma}^2_{t+1}$ directly in the price SDE and for the classification, each state corresponds to a fixed volatility value which is used in the SDE.

5 Pricing Results

We evaluate the pricing accuracy of each model by computing the Root Mean Squared Percentage Error (RMSPE). This metric measures the average percentage deviation of the simulated option price from the market price, with lower values indicating better accuracy. Tabled results are presented for the following models:

- The constant volatility GBM baseline.
- The single-regime GARCH(1,1) model.
- The two-regime RS-GARCH model.

Accuracy in Monte Carlo simulations tends to increase when more simulations can be used. In order to simulate the millions of timesteps needed to price the whole options chain, the algorithms had to be parallelised using the 'numpy' Python package and accelerated using 'numba', a just-in-time compiler for Python [16]. While GBM, GARCH(1,1) and RS-GARCH estimates can be generated entirely using basic mathematical operations, these models were chosen. While the deep learning models were incompatible with 'numba' meaning the large number of simulations required to price the entire options chain becomes computationally infeasible, some exploratory analysis will be presented in the following discussion.

5.1 Fitted Parameters

The GARCH(1,1) model was fitted by minimising the negative log likelihood using the 'arch_model' Python package. The unscaled parameters used are given in table 1. in particular, the sum $\alpha + \beta = 0.9855$ is close to 1 suggesting that shocks to volatility decay slowly over time. The daily unconditional variance can be estimated as $\frac{\omega}{1-\alpha-\beta} = 0.000138$, resulting in annualised volatility of $\sqrt{0.000138} \times \sqrt{252} = 18.7\%$, consistent with the sample's overall volatility level used in GBM. The GARCH model captures the clustering of volatility by allowing conditional variance to evolve with recent market turbulence, rather than remain constant. This should, in theory, lead to more responsive option pricing than the static GBM. While the table shows a value for a fitted μ, this is not the drift value that will be used to calculate the next log return value as this should be the risk neutral drift $\mu_{\mathbb{Q}}$. Instead, this value is used to calculate the shock $\varepsilon_{t-1} = r_{t-1} - \mu$ used to compute the conditional variance.

Table 1: Fitted GARCH Parameters

μ	ω	α	β
0.000719	0.000002	0.115852	0.869674

The Regime-Switching GARCH (RS-GARCH) model was calibrated using a two-state hidden Markov model for volatility, where each state has their own set of GARCH parameters. Table 2 gives the estimated GARCH parameters for each of the two regimes. Regime 1

features a much lower annualised unconditional volatility of $\sqrt{\frac{\omega}{1-\alpha-\beta} \times 252} = 4.93\%$ compared to the high 32.7% in regime 1. This shows that the constant volatility value of 19.1% overestimates the typical low volatility periods and overestimates the high volatility shocks.

The low volatility regime 1 shows higher persistence with $\alpha_1 + \beta_1 = 0.983 \approx 1$ meaning that in calm market periods,volatility tends to remain low and stable for extended periods. The high volatility regime 0 also has a high persistence of $\alpha_0 + \beta_0 = 0.9481$ meaning spikes in volatility are larger, indicated by the higher α_1, but are shorter-lived, indicated by the lower β_1. The estimated transition probabilities show this difference as the fitted transition matrix $\hat{P}$ 26 shows that the probability of staying in the low-volatility regime $p_{11} = 0.917$ is very high, while the probability of staying in the high volatility regime $p_{00} = 0.625$ is lower. These probabilities mean that for the majority of the simulations the underlying will stay in the low volatility regime and when it does eventually transition, it will likely revert back to calm conditions within a relatively short time.

$$\hat{P} = \begin{bmatrix} 0.625 & 0.375 \\ 0.083 & 0.917 \end{bmatrix} \tag{26}$$

Table 2: Fitted RS-GARCH Parameters

Regime	μ	ω	α	β
0	-0.00426	2.20×10^{-5}	0.155	0.793
1	-0.00110	1.67×10^{-7}	0.0547	0.928

5.2 Aggregate Performance

The performance of each model was evaluated by comparing the simulated option prices to actual market prices. We use the Root Mean Squared Percentage Error (RMSPE) as the metric of accuracy, which measures the average proportional deviation between model prices and market prices, where a lower value indicates more accurate prices. Table 3 summarises the RMSPE for each model across the entire option chain, meaning all strikes and maturities available on Yahoo Finance.

Table 3: Aggregate RMSPE Results

Model	**Call**	**Put**	**Total**
GBM	10.36%	16.31%	14.94%
GARCH(1,1)	10.00%	17.59%	15.84%
RS-GARCH	10.28%	19.00%	16.99%

The original hypothesis was that overall pricing accuracy, based on RMSPE, would improve with more complex volatility models. Our results show that this is not uniformly the case. In terms of RMSPE, the GBM benchmark was competitive with, and even outperformed, the both the econometric GARCH models. In fact, the RS-GARCH model achieves the highest aggregate RMSPE (16.99%) compared to GARCH(1,1) (15.84%) and GBM (14.94%).

All three models showed similar accuracy in pricing call options, but performed much worse for pricing puts such as RS-GARCH scoring 10.28% for calls, marginally better than baseline 10.36%, while also having the highest score for puts at 19.00%, worse than baseline 16.31%.

5.3 Moneyness

To investigate model accuracy across moneyness, meaning the difference between the contract strike price and the underlying price, we compared RMSPE for at-the-money (ATM) options versus deep in-the-money (ITM) and deep out-of-the-money (OTM) options, meaning contracts with strikes very far away from the current price.

Table 4 shows the RMSPE comparing only options whose strike price is close to the current price, meaning the option is at-the-money (ATM). All three of the models performed poorly here with RMSPE much higher for GBM (16.87%), GARCH(1,1) (18.55%) and RS-GARCH (22.75%) compared to their previous aggregate results. The GARCH(1,1) performed very well on calls with a RMSPE of 0.42%, while this gain was dragged by the poor put performance of 19.19%.

The poor put performance for ATM strikes could be caused by a tendency for the models to overestimate volatility, causing simulated prices to diverge further away from the current price than the market implies. Some simulated paths with large price swings cause the payoff distribution to skew for larger underlying prices, resulting in the expected call payoffs to increase and lower expected put payoff.

Table 4: ATM RMSPE Results

Model	**Call**	**Put**	**Total**
GBM	12.91%	17.01%	16.87%
GARCH(1,1)	0.42%	19.19%	18.55%
RS-GARCH	13.76%	23.08%	22.75%

More pronounced discrepancies arose for deep in-the-money (ITM) and deep out-of-the-money (OTM) contracts, meaning their strike prices are in the tails of the payoff distribution. The prices of these options are highly sensitive to the tails of the underlying return distribution.

Table 5: ITM/OTM RMSPE Results

Model	**Call**	**Put**	**Total**
GBM	6.70%	15.85%	13.12%
GARCH(1,1)	8.02%	15.84%	13.51%
RS-GARCH	6.85%	16.19%	13.41%

Table 5 shows that despite its simplicity, the GBM model (13.12%) scores the lowest RMSPE pricing deep ITM and deep OTM compared to GARCH(1,1) (13.51%) and RS-

GARCH (13.41%).

5.4 Term-structure

We also evaluated the accuracy of pricing in the term structure of expirations by comparing short-term options that expire in fewer than 50 trading days, against longer term options expiring over 50 trading days. This is to investigate how pricing is affected when dynamic volatility is given time to evolve.

Table 6 shows the results for the near-term options and can see comparable results between GBM (13.77%), GARCH(1,1) (14.22%) and RS-GARCH (14.00%). Because short maturities give volatility very little time to evolve, prices are less prone to swing in extreme directions, causing the underlying return distribution to have thin tails, just like the tails of a constant volatility return distribution, resulting in similar performance between dynamic volatility models and baseline.

Table 6: Short-Dated RMSPE Results

Model	**Call**	**Put**	**Total**
GBM	8.92%	14.58%	13.77%
GARCH(1,1)	8.58%	15.15%	14.22%
RS-GARCH	8.64%	14.22%	14.00%

Table 7 shows the pricing results for options expiring within 50 trading days and here is where we highlight the main difference between the models.

The econometric models, GARCH(1,1) (10.10%) and RS-GARCH(1,1) (10.40%), marginally outperform GBM (10.46%), although all were similar. The differences appear when we compare the put RMSPE values ranging from 16.55% for GBM to 19.57% for RS-GARCH, and 17.9%3 for GARCH(1,1).

Table 7: Long-Dated RMSPE Results

Model	**Call**	**Put**	**Total**
GBM	10.46%	16.55%	15.08%
GARCH(1,1)	10.10%	17.93%	16.04%
RS-GARCH	10.40%	19.57%	17.36%

Overall, we have seen the dynamic volatility models performing similar to the constant volatility in the bast cases, and much worse in other cases. In the following section we will discuss why this may be the case, including systematic issues in pricing puts and risk neutral assumptions.

6 Discussion

The overall pricing results show that incorporating dynamic volatility models did not consistently improve Monte Carlo option pricing relative to the GBM baseline. In particular, all the models struggled to accurately price put options, with the most complex RS-GARCH model having the highest overall RMSPE. To explain this outcome, we will have to take a deeper look at the return distributions generated by each model.

Table 8 presents the descriptive statistics of the underlying price distribution after running 30 day simulations under each of the different volatility models. We can see the GBM model produces approximately a normal distribution, evident by its negligible skew and a kurtosis close to 3, as expected. The key difference with the GARCH(1,1) and RS-GARCH models is that the resulting kurtosis is much larger, indicating heavier tails in the distribution. While we have successfully captured the leptokurtic behaviour of real market returns, this was unable to provide an advantage in pricing options as seen in our results.

Table 8: Descriptive Statistics for 30-Day Distribution

Model	**Mean**	**Std**	**Skew**	**Kurtosis**
GBM	559.6	34.98	0.1853	3.056
GARCH(1,1)	560.5	31.02	0.5369	8.386
RS-GARCH	560.6	34.08	0.2142	4.472
CNN-LSTM (reg)	587.6	1.085	0.444	2.955
CNN-LSTM (class)	587.6	4.274	0.004	3.008

Figure 1 visualises these different price distributions. Although the GARCH (green) histogram sits roughly on top of the GBM (blue) histogram in the main body of the distribution, its much higher kurtosis suggests there is far more mass in the tails compared to GBM. The RS-GARCH distribution (orange) has a sharp central spike reflecting long periods in its low-volatility regime, with heavy tails arising from the occasional regime switches, although not as heavy as single regime GARCH(1,1) where high volatility can persist forever.

None of the models include an explicit leverage-effect term. In real markets large negative price shocks are followed by a disproportionate jump in volatility, whereas our simulations treat positive and negative shocks symmetrically. This symmetry shifts probability mass toward the right-hand tail, giving the positive skew in Table 8. As a result, payoff distributions over-weight extreme call outcomes and under-weight extreme put outcomes, resulting in the mis-pricing we observe.

Figure 2 shows the results of simulating CNN-LSTM volatility for 30 days using either regression or classification. The key difference between the two modes is the resulting underlying price distribution has a much lower standard deviation than the classification mode, with both models having significantly lower standard deviation than the classical models. An explanation for the low variance in the regression is when predicting a continuous value for volatility, the model keeps its predictions close to the mean in order to reduce the loss in

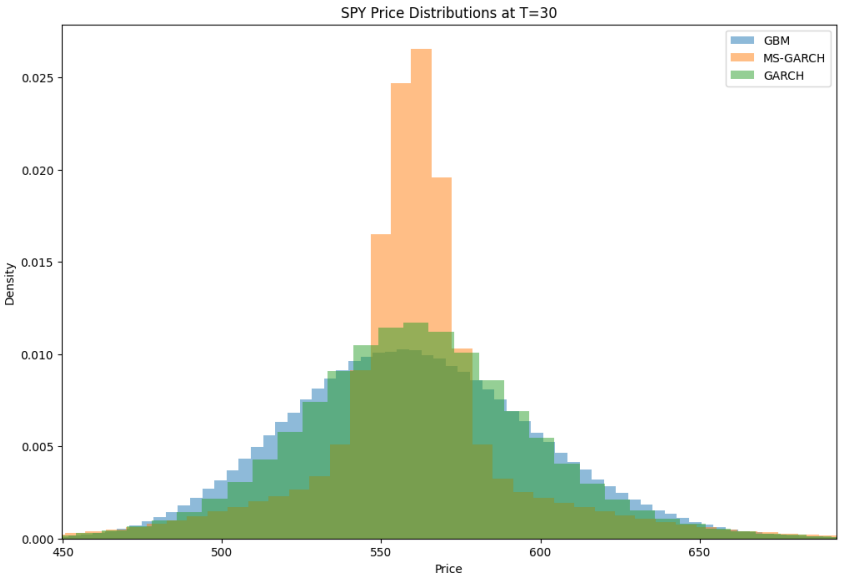

Figure 1: Simulated 30-day price distributions under GBM, GARCH(1,1) and RS-GARCH

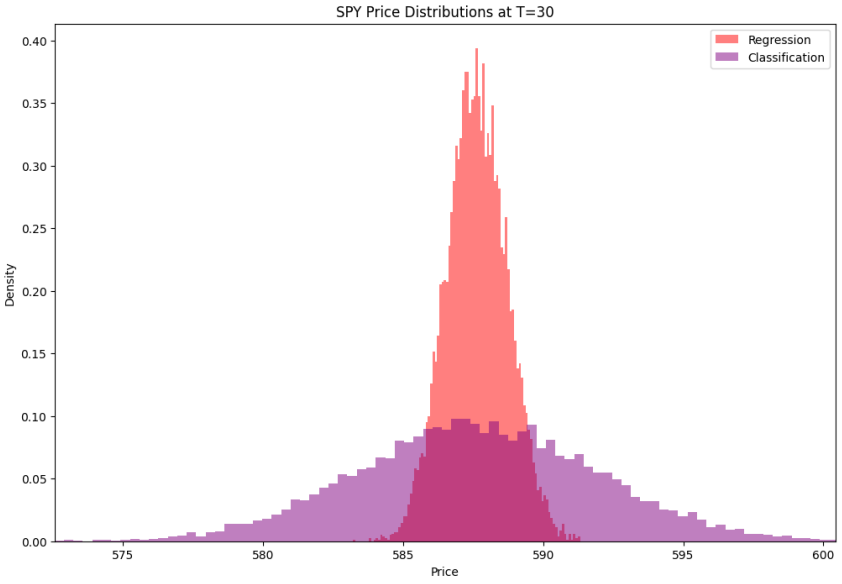

Figure 2: Simulated 30-day price distributions under CNN-LSTM

training. This was expected which is why the classification approach was also explored and we can see the price distribution for classification is more varied.

A final factor to reconcile the pricing results is the distinction between the real-world measure $\mathbb{P}$, under which our volatility models were estimated, and the risk-neutral measure $\mathbb{Q}$ required for option valuation. Although the simulations were performed with constant risk neutral drift $\mu_{\mathbb{Q}} = r - q$, the conditional volatility processes were fit using historical values under real-world measures, which means our simulations were not truly risk neutral. The literature shows that option markets demand compensation for bearing volatility risk, meaning option-implied variance systematically exceeds realised variance, creating a difference known as the volatility premium [3]. Ignoring this means our models likely underestimated implied volatility, contributing to the mispricing of options.

7 Conclusion

This study set out to evaluate whether time-varying volatility models improve Monte Carlo option pricing relative to the constant-volatility benchmark. Using daily SPY data (1993-2025), we calibrated each model and priced the full option chain via Monte Carlo simulation.

While the dynamic models successfully reproduced key empirical features of returns such as volatility clustering and leptokurtosis, they did not consistently reduce pricing error. In fact, all models struggled with put options, and the complex RS-GARCH model produced the highest aggregate RMSPE.

These findings carry two implications. First, for plain-vanilla options a parsimonious GBM can rival far more complex econometric or machine learning schemes. Secondly, capturing higher-order moments in the payoff distribution is necessary, but not sufficient for option pricing without a genuine risk-neutral transformation.

Future work should therefore:

- Focus on volatility under the risk-neutral measure $\mathbb{Q}$, introducing an adjustment that results in the simulated volatilities aligning with the implied volatility surface.
- Extend GARCH and ML to include explicit leverage effect terms to shift probability mass toward the less tail, improving put prices.
- Testing models on exotic derivatives that are path-dependent to gauge whether dynamic volatility adds value.

Ultimately, this future research will move us closer to a robust, risk-neutral simulation framework capable of accurately pricing a wide spectrum of derivative products.

Appendices

Appendix A Data Collection

All of the data for the ticker 'SPY' was collect using the Python package 'yfinance'.

```
ticker = "SPY"
timeframe = "1d"
yf_ticker = yf.Ticker(ticker)
yf_data = yf_ticker.history(period="max", interval=timeframe)
yf_data["Log Returns"] = np.log(yf_data["Close"] / yf_data["Close"].shift(1))
yf_data = yf_data[1:] # First log return is NaN
S0 = yf_data["Close"].iloc[-1]

div = yf_data["Dividends"]
prices = yf_data["Close"]
```

Options data is also collected from 'yfinance'. We need to collect all available options, including puts and calls across all strikes and expirations and store them in a single dataframe.

```
def get_all_option_prices(expiration, ticker=yf_ticker):
    option_chain = ticker.option_chain(expiration)
    calls = option_chain.calls
    puts = option_chain.puts

    merged = pd.merge(calls[['strike', 'bid', 'ask']],
                      puts[['strike', 'bid', 'ask']],
                      on='strike',
                      suffixes=('_call', '_put'))


    merged['Call_Market_Price'] = (merged['bid_call'] + merged['ask_call']) / 2
    merged['Put_Market_Price'] = (merged['bid_put'] + merged['ask_put']) / 2

    return merged[['strike', 'Call_Market_Price', 'Put_Market_Price']]

data = []
expiration_dates = yf_ticker.options

for exp in expiration_dates:
    option_prices = get_all_option_prices(exp)
    option_prices['Expiration'] = exp
    data.append(option_prices)

market_prices_df = pd.concat(data, ignore_index=True)
market_prices_df['Expiration'] = pd.to_datetime(market_prices_df['Expiration'])
market_prices_df['DTE'] = (market_prices_df['Expiration'] - today).dt.days
```

```

unique_dtes = market_prices_df['DTE'].unique()
unique_dtes = unique_dtes[unique_dtes > 0]
```

Appendix B Fitting the Models

B.1 Risk-free Rate

Under the risk neutral assumptions, the drift of an asset is the risk free rate r, taken as US T-Bills yield for the given date, adjusted for the continuous dividends of the asset q.

```
div_yield = div / prices
q = div_yield.mean() * 252
q = np.log(1 + q)
```

B.2 GBM

GBM is the simplest model to fit as σ is just the sample standard deviation of the log returns, adjsuted to be annualised.

```
sigma = np.std(yf_data["Log Returns"]) * np.sqrt(252)
```

B.3 GARCH(1,1)

The simple GARCH(1,1) model was fitted using the 'arch_model' function from the Python package 'arch'. To improve convergence, the log returns input was scaled by multiplying by 100, as suggested the warning when running the function without scaling. To account for this, the μ and ω parameters were appropriately unscaled while α and β has no units so didn't need to be rescaled.

```
garch_model = arch_model(spy_df["LogReturn"]*100, vol="Garch", p=1,
↪  q=1).fit(disp="off")
garch_model.params["mu"] /= 100
garch_model.params["omega"] /= 100**2

garch_model.params.to_csv(f"garch_params.csv")
```

B.4 RS-GARCH

The RS-GARCH model is fitted in two steps. Firstly, for a given transition matrix P, the Hamilton filter is applied to estimate GARCH parameters for each regime and then return a log likelihood value. We choose the matrix P that minimises the resulting log likelihood function.

The function is optimised using the 'numba' library which is able to optimise 'numpy' opertations by generating efficient machine code. However, in order to keep the function numba compatable, the function has to be written such that there is no Python objects, only operations compatible with machine code. Writing functions this way significantly speeds up processing time, reducing the computational cost associated with more complex models.

```
import numba
import numpy as np

@numba.jit(noPython=True)
def normal_pdf(x, mu, sigma):
    eps = 1e-16
    sigma = max(sigma, eps)  # avoid division by zero
    const = 1.0 / (np.sqrt(2.0 * np.pi) * sigma)
    z = (x - mu) / sigma
    return const * np.exp(-0.5 * z * z)

@numba.njit
def stationary_2x2(P):
    p00 = P[0, 0]
    p11 = P[1, 1]

    denom = 2.0 - p00 - p11

    pi0 = (1.0 - p11) / denom
    pi1 = (1.0 - p00) / denom

    return np.array([pi0, pi1], dtype=np.float64)

@numba.jit(noPython=True)
def rs_garch_filter(
    data,
    P,
    means,
    omega,
    alpha,
    beta,
    var0
):

    T = data.shape[0]

    pi_filt = np.zeros((T, 2))
```

```
    P_stationary = stationary_2x2(P)
    pi_filt[0,:] = P_stationary

    sigma2 = np.zeros((T, 2))
    sigma2[0, 0] = var0[0]
    sigma2[0, 1] = var0[1]

    eps = 1e-16
    f0_0 = normal_pdf(data[0], means[0], np.sqrt(sigma2[0, 0]))
    f0_1 = normal_pdf(data[0], means[1], np.sqrt(sigma2[0, 1]))
    ell_0 = pi_filt[0,0]*f0_0 + pi_filt[0,1]*f0_1

    denom = ell_0 + eps
    pi_filt[0,0] = (pi_filt[0,0] * f0_0) / denom
    pi_filt[0,1] = (pi_filt[0,1] * f0_1) / denom

    logL = np.log(ell_0 + eps)

    for t in range(1, T):
        sigma2[t, 0] = (omega[0]
                        + alpha[0] * (data[t-1] - means[0])**2
                        + beta[0]  * sigma2[t-1, 0])
        sigma2[t, 1] = (omega[1]
                        + alpha[1] * (data[t-1] - means[1])**2
                        + beta[1]  * sigma2[t-1, 1])

        pi_pred_0 = pi_filt[t-1, 0]*P[0,0] + pi_filt[t-1, 1]*P[1,0]
        pi_pred_1 = pi_filt[t-1, 0]*P[0,1] + pi_filt[t-1, 1]*P[1,1]

        f_t0 = normal_pdf(data[t], means[0], np.sqrt(sigma2[t, 0]))
        f_t1 = normal_pdf(data[t], means[1], np.sqrt(sigma2[t, 1]))

        ell_t = pi_pred_0*f_t0 + pi_pred_1*f_t1
        denom = ell_t + eps

        pi_filt[t, 0] = (pi_pred_0 * f_t0) / denom
        pi_filt[t, 1] = (pi_pred_1 * f_t1) / denom

        logL += np.log(ell_t + eps)

    return pi_filt, sigma2, logL
```

With the Hamilton filter function defined, we can fit the model using the ‘minimize’ function from ‘scipy.optimize’.

```
def objective(theta, data, P):
    """
    theta = [
        means[0], means[1],
```

```
        omega[0], omega[1],
        alpha[0], alpha[1],
        beta[0],  beta[1],
        var0[0],  var0[1],
        p00,      p11
    ]
    """
    means = np.array([theta[0], theta[1]])
    omega = np.array([theta[2], theta[3]])
    alpha = np.array([theta[4], theta[5]])
    beta  = np.array([theta[6], theta[7]])
    var0  = omega / (1 - alpha - beta)
    if np.any(var0 < 0):
        return np.inf
    if np.any(np.isnan(var0)):
        return np.inf

    pi_filt, sigma2, logL = rs_garch_filter(data, P, means, omega, alpha, beta,
    ↪  var0)
    return -logL  # negative log-likelihood

from scipy.optimise import minimize

theta0 = np.array([
    0.0,   0.0,     # means
    0.00001,   0.01,     # omega - encourage one state to have higher volatility
    ↪  than the other
    0.045,  0.15,    # alpha
    0.945,   0.65,     # beta
])

bnds = [
    (None,None),   # mean0
    (None,None),   # mean1
    (1e-6,None), (1e-6,None),   # omega
    (0,1), (0,1),               # alpha
    (0,1), (0,1),               # beta
]

lr_scaler = 100
df["Log_Returns"] = df["Log_Returns"] * lr_scaler

p00_grid = np.linspace(0, 1, 25)
p11_grid = np.linspace(0, 1, 25)
best_p00 = None
best_p11 = None
best_obj = np.inf
best_res = None
for p00 in p00_grid:
    for p11 in p11_grid:
        P = np.array([
```

```
            [p00,      1 - p00],
            [1 - p11,  p11    ]
        ])
        res = minimize(
            objective,
            theta0,
            args=(train["Log_Returns"].values, P),
            method='L-BFGS-B',
            bounds=bnds
        )
        if res.fun < best_obj:
            best_obj = res.fun
            best_p00 = p00
            best_p11 = p11
            best_res = res
            print(f"New best: p00={p00}, p11={p11}, obj={res.fun}")
```

Again, to improve convergence, the log returns were scaled by a constant 100, so we need to unscale μ and ω.

```
res = best_res
means = res.x[:2] / lr_scaler
omegas = res.x[2:4] / lr_scaler ** 2
alphas = res.x[4:6]
betas  = res.x[6:8]
var0   = omegas / (1 - alphas - betas)
P = np.array([
    [best_p00,      1 - best_p00],
    [1 - best_p11,  best_p11    ]
])

params = pd.DataFrame({
    "Mean": means,
    "Omega": omegas,
    "Alpha": alphas,
    "Beta": betas,
    "p": [best_p00, best_p11]
})
params.to_csv(f"ms_garch_params.csv")
```

Increasing the value of ‘lr_scaler’ yields near identical results.

Finally with the models fitted, we can move on to the simulations.

B.5 Machine Learning

Both the regression and classification models both aim to forecast time series information, so a recurrent neural network is chosen. To assist with short term feature patterns hidden in the data, a convolutional layer is applied, whose flattened output is given to the recurrent neural network, specifically the LSTM model.

B.5.1 Feature Engineering

During the simulations, the only features available to the model are those derived from price, because a simulation 1 year in the future will not be able to predict the next volatility based on the unknown, future exogenous variables such as GDP or VIX.

```
ticker_str = "SPY"
yf_ticker = yf.Ticker(ticker_str)
yf_data = yf_ticker.history(period="max")

yf_data["Log Return"] = np.log(yf_data["Close"] / yf_data["Close"].shift(1))
yf_data = yf_data[1:]

yf_data.head()

# features
yf_data["Log Return Sq"] = yf_data["Log Return"] ** 2

num_rolling = [4, 15, 30, 180] # 4 is minimum for kurt
for i in num_rolling:
    yf_data[f"Rolling Mean {i}"] = yf_data["Log Return"].rolling(i).mean()
    yf_data[f"Rolling Std {i}"] = yf_data["Log Return"].rolling(i).std()
    yf_data[f"Rolling Skew {i}"] = yf_data["Log Return"].rolling(i).skew()
    yf_data[f"Rolling Kurt {i}"] = yf_data["Log Return"].rolling(i).kurt()

    yf_data[f"Rolling Vol Mean {i}"] = yf_data["Log Return
    ↪  Sq"].rolling(i).mean()
    yf_data[f"Rolling Vol Std {i}"] = yf_data["Log Return Sq"].rolling(i).std()
    yf_data[f"Rolling Vol Skew {i}"] = yf_data["Log Return
    ↪  Sq"].rolling(i).skew()
    yf_data[f"Rolling Vol Kurt {i}"] = yf_data["Log Return
    ↪  Sq"].rolling(i).kurt()

yf_data = yf_data[max(num_rolling):]

features = [
    "Log Return",
    "Log Return Sq",
]


features += [f"Rolling Mean {i}" for i in num_rolling]
features += [f"Rolling Std {i}" for i in num_rolling]
```

```
features += [f"Rolling Skew {i}" for i in num_rolling]
features += [f"Rolling Kurt {i}" for i in num_rolling]

features += [f"Rolling Vol Mean {i}" for i in num_rolling]
features += [f"Rolling Vol Std {i}" for i in num_rolling]
features += [f"Rolling Vol Skew {i}" for i in num_rolling]
features += [f"Rolling Vol Kurt {i}" for i in num_rolling]

df = yf_data[features].copy()
```

B.5.2 Target

Our objective is to predict the volatility used to draw the log return of the next step in the simulation. In the regression case, the target is to predict the specific value of σ^2_{t+1}, given the feature set at time t. In the classification case, the volatility distribution is cut into two bins representing a low volatility and high volatility state.

```
yf_data["Vol"] = yf_data["Log Return"].rolling(5).std()

df["target reg"] = yf_data["Vol"].copy().shift(-1)

num_classes = 2
df["target class"] = pd.qcut(yf_data["Vol"].copy(), num_classes, labels=[i for i
↪  in range(num_classes)]).shift(-1)
```

B.5.3 Training

The model is fitted in Python as a 'Sequential' object from the 'tensorflow.keras' library:

```
from tensorflow import keras

model = keras.Sequential()
model.add(Input(shape=(lookback, n_features)))
model.add(Conv1D(32, 3, activation="relu", padding="same"))

model.add(LayerNormalization())
model.add(Dropout(0.2))

model.add(LSTM(24, return_sequences=True))
model.add(LSTM(12, return_sequences=False))

model.add(Dense(16, activation="relu"))
model.add(Dropout(0.2))

if target_type == "reg":
```

```
    model.add(Dense(len(targets), activation="linear"))
else:
    model.add(Dense(len(np.unique(df["target class"])), activation="softmax")) # class output

if target_type == "reg":
    metric = "mape"
    model.compile(optimiser=Adam(3e-4), loss="mean_squared_error", metrics=[metric]) # reg compile
else:
    metric = "accuracy"
    model.compile(optimiser=Adam(1e-4), loss="sparse_categorical_crossentropy", metrics=[metric]) # class compile
```

Appendix C Simulations

C.1 GBM

Again, the GBM function is the most simple case and also the fastest to run since there is no path dependency so the random shocks can be fully vectorised. However, for consistency, this function is still 'numba' optimised. Since 'np.random.seed()' is not numba compatable, the code had to be split into a core 'numba' optimised function and a non optimised, Python friendly wrapper.

```
@nb.njit(parallel=True, fastmath=True)
def _simulate_gbm_core(mu_Q, sigma, n_days, n_paths):
    dt      = 1.0 / 252.0
    drift   = (mu_Q - 0.5 * sigma * sigma) * dt
    vol     = sigma * np.sqrt(dt)

    log_paths = np.empty((n_days, n_paths), dtype=np.float64)

    shocks = np.random.randn(n_days, n_paths)

    log_paths[0, :] = drift + vol * shocks[0, :]

    for t in range(1, n_days):
        log_paths[t, :] = log_paths[t - 1, :] + drift + vol * shocks[t, :]

    return log_paths


def simulate_gbm_numba(mu_Q_annual, sigma_annual, n_days=252, n_paths=10_000, seed=None):
    if seed is not None:
        np.random.seed(seed)
    log_paths = _simulate_gbm_core(mu_Q_annual, sigma_annual, n_days, n_paths)
```

```
    return log_paths
```

C.2 GARCH

Similar to GBM, the GARCH function had to be broken down into numba optimised and Python friendly functions to significantly speed up the simulation process.

```
@nb.njit(parallel=True, fastmath=True)
def _simulate_garch_core(mu_Q_annual: float,
                         mu: float,
                         omega: float,
                         alpha: float,
                         beta:  float,
                         n_days: int,
                         n_paths: int) -> np.ndarray:

    dt        = 1.0 / 252.0
    logpaths  = np.empty((n_days, n_paths), dtype=np.float64)

    # long-run variance as starting point
    if alpha + beta < 1.0:
        var0 = omega / (1.0 - alpha - beta)
    else:                          # non-stationary fallback
        var0 = omega

    for p in nb.prange(n_paths):

        prev_var   = var0
        prev_r     = 0.0
        cum_return = 0.0

        for t in range(n_days):
            prev_eps = prev_r - mu_Q_annual * dt # mu
            variance = omega + alpha * prev_eps * prev_eps + beta * prev_var
            sd       = np.sqrt(variance)              # DAILY vol
            drift    = mu_Q_annual * dt - 0.5 * variance   # DAILY drift adj.

            r_t = drift + sd * np.random.randn()
            cum_return += r_t
            logpaths[t, p] = cum_return

            prev_var = variance
            prev_r   = r_t

    return logpaths

def simulate_garch_numba(mu_Q_annual: float,
                         mu: float,
```

```
                          omega: float,
                          alpha: float,
                          beta: float,
                          n_days: int = 252,
                          n_paths: int = 100_000,
                          seed: int | None = None) -> pd.DataFrame:
    if seed is not None:
        np.random.seed(seed)

    logpaths = _simulate_garch_core(mu_Q_annual, mu, omega, alpha, beta,
                                    n_days, n_paths)

    return logpaths
```

C.3 RS-GARCH

The RS-GARCH simulations are the real reason why we choose to numba optimise the simulations. Without numba, the simulations would take multiple hours to complete, leading to very inefficient debugging. When numba is applied, the simulations complete within minutes.

```
@nb.njit(parallel=True, fastmath=True)
def _prep_params(params_df):
    return (params_df["Mean"].values.astype(np.float64),
            params_df["Omega"].values.astype(np.float64),
            params_df["Alpha"].values.astype(np.float64),
            params_df["Beta"].values.astype(np.float64),
            params_df["p"].values.astype(np.float64))


@nb.njit
def sample_discrete(p): #np.random.choice not supported
    u = np.random.rand()
    cumulative = 0.0
    for i in range(p.shape[0]):
        cumulative += p[i]
        if u < cumulative:
            return i
    return p.shape[0]


@nb.njit(parallel=True, fastmath=True)
def _simulate_ms_garch_core(mu_Q, mu, omegas, alphas, betas, p_stay,
                            n_days, n_paths):
    m = omegas.shape[0]
    dt = 1.0 / 252.0

    P = np.zeros((m, m), dtype=np.float64)
```

```
    for i in range(m):
        P[i, i] = p_stay[i]
        if m > 1:
            off = (1.0 - p_stay[i]) / (m - 1)
            for j in range(m):
                if i != j:
                    P[i, j] = off

    if m == 1:
        stationary = np.array([1.0], dtype=np.float64)
    else:
        stationary = np.ones(m, dtype=np.float64) / m
        for _ in range(50):
            stationary = stationary @ P # quick estimate of the stationary
            ↪ distribution

    out = np.empty((n_days, n_paths), dtype=np.float64)


    for path in nb.prange(n_paths):

        state = sample_discrete(stationary)

        prev_var = omegas[state] / (1.0 - alphas[state] - betas[state])

        prev_ret = 0.0
        cum_ret  = 0.0

        for t in range(n_days):
            if t > 0:
                #state = np.random.choice(m, p=P[state])
                state = sample_discrete(P[state])

            prev_eps = prev_ret - mu_Q * dt
            var   = omegas[state] + alphas[state] * prev_eps**2 + betas[state] *
            ↪ prev_var
            drift = (mu_Q * dt) - 0.5 * var * dt
            sd    = np.sqrt(var)# * dt)

            r_t    = drift + sd * np.random.randn()
            cum_ret += r_t
            out[t, path] = cum_ret

            prev_ret = r_t
            prev_var = var

    return out

def simulate_ms_garch_numba(mu_Q_annual, params_df,
                            n_days=252, n_paths=10_000, seed=None):
    if seed is not None:
```

```
        np.random.seed(seed)

    mu, omegas, alphas, betas, p_stay = _prep_params(params_df)
    log_paths = _simulate_ms_garch_core(
        mu_Q_annual, mu, omegas, alphas, betas, p_stay,
        n_days, n_paths
    )
    return log_paths
```

All of these functions simulate the path of cumulative log returns. We can transform them into the price paths by taking the exponent then multiplying by the initial price.

```
num_simulations = 1000000

for dte in unique_dtes:
    print(dte)
    mu_Q = mu_Q_dict[dte]

    gbm_return_path = simulate_gbm_numba(mu_Q, sigma, dte, num_simulations)
    gbm_final_prices = S0 * np.exp(gbm_return_path[-1, :])
    np.save(f"final prices/gbm_final_prices_{dte}.npy", gbm_final_prices)
    del gbm_return_path, gbm_final_prices
    gc.collect()

    ms_garch_return_path = simulate_ms_garch_numba(mu_Q, ms_garch_params, dte,
    ↪  num_simulations)
    ms_garch_prices = S0 * np.exp(ms_garch_return_path[-1, :])
    np.save(f"final prices/ms_garch_final_prices_{dte}.npy", ms_garch_prices)
    del ms_garch_return_path, ms_garch_prices
    gc.collect()

    garch_return_path = simulate_garch_numba(mu_Q, garch_mu, garch_omega,
    ↪  garch_alpha, garch_beta, dte, num_simulations)
    garch_prices = S0 * np.exp(garch_return_path[-1, :])
    np.save(f"final prices/garch_final_prices_{dte}.npy", garch_prices)
    del garch_return_path, garch_prices
    gc.collect()
```

References

[1] Torben G. Andersen, Tim Bollerslev, Francis X. Diebold, and Paul Labys. "Modeling and Forecasting Realized Volatility". In: *Econometrica* 71.2 (2003), pp. 579–625. DOI: https://doi.org/10.1111/1468-0262.00418. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1468-0262.00418. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-0262.00418.

[2] Tim Bollerslev. “Generalized autoregressive conditional heteroskedasticity”. In: *Journal of Econometrics* 31.3 (1986), pp. 307–327. ISSN: 0304-4076. DOI: https://doi.org/10.1016/0304-4076(86)90063-1. URL: https://www.sciencedirect.com/science/article/pii/0304407686900631.

[3] Tim Bollerslev, George Tauchen, and Hao Zhou. “Expected Stock Returns and Variance Risk Premia”. In: *Review of Financial Studies* 22.11 (2009), pp. 4463–4492. DOI: 10.1093/rfs/hhn047.

[4] L. Breiman. “Random Forests”. In: *Machine Learning* 45 (2001), pp. 5–32.

[5] Cboe Global Markets. *The Creation of Listed Options at Cboe*. Accessed: 2025-04-03. 2023. URL: https://www.cboe.com/insights/posts/the-creation-of-listed-options-at-cboe/.

[6] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 785–794. ISBN: 9781450342322. DOI: 10.1145/2939672.2939785. URL: https://doi.org/10.1145/2939672.2939785.

[7] David A. Dickey and Wayne A. Fuller and. “Distribution of the Estimators for Autoregressive Time Series with a Unit Root”. In: *Journal of the American Statistical Association* 74.366a (1979), pp. 427–431. DOI: 10.1080/01621459.1979.10482531. URL: https://doi.org/10.1080/01621459.1979.10482531.

[8] Robert F. Engle. “Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation”. In: *Econometrica* 50.4 (1982), pp. 987–1007. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1912773 (visited on 01/05/2025).

[9] Thomas Fischer and Christopher Krauss. “Deep learning with long short-term memory networks for financial market predictions”. In: *European Journal of Operational Research* 270.2 (2018), pp. 654–669. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2017.11.054. URL: https://www.sciencedirect.com/science/article/pii/S0377221717310652.

[10] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Applications of Mathematics. See Section 3.2 for Euler discretization of SDEs. New York: Springer, 2004. URL: https://www.bauer.uh.edu/spirrong/Monte_Carlo_Methods_In_Financial_Enginee.pdf.

[11] James D. Hamilton. “A New Approach to the Economic Analysis of Nonstationary Time Series and the Business Cycle”. In: *Econometrica* 57 (1989).

[12] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[13] John C. Hull. *Options, Futures, and Other Derivatives*. 9th. Pearson, 2014.

[14] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. “LightGBM: A Highly Efficient Gradient Boosting Decision Tree”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett.

Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[15] Daniel B. Nelson. "Conditional Heteroskedasticity in Asset Returns: A New Approach". In: *Econometrica* 59.2 (1991), pp. 347–370. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/2938260 (visited on 04/17/2025).

[16] Numba Developers. *Numba 5-Minute Guide*. https://numba.pydata.org/numba-doc/dev/user/5minguide.html.

[17] Francis E.H Tay and Lijuan Cao. "Application of support vector machines in financial time series forecasting". In: *Omega* 29.4 (2001), pp. 309–317. ISSN: 0305-0483. DOI: https://doi.org/10.1016/S0305-0483(01)00026-3. URL: https://www.sciencedirect.com/science/article/pii/S0305048301000263.

[18] Andrés Vidal and Werner Kristjanpoller. "Gold volatility prediction using a CNN-LSTM approach". In: *Expert Systems with Applications* 157 (2020), p. 113481. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2020.113481. URL: https://www.sciencedirect.com/science/article/pii/S0957417420303055.

[19] Huimin Wang, Songbai Song, Gengxi Zhang, and Olusola O. Ayantoboc. "Predicting daily streamflow with a novel multi-regime switching ARIMA-MS-GARCH model". In: *Journal of Hydrology: Regional Studies* 47 (2023).

[20] Bo Zhao, Haohan Lu, Kewei Chen, Jianguo Liu, and Sihong Wu. "Convolutional neural networks for time series classification". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. 2017, pp. 3578–3585.