

---

# Pricing Asian Options on Commodities with GARCH Model

Lue Shen<sup>1</sup>, Leheng Chen<sup>1</sup>, Chu Song<sup>1</sup> and Jing Qian<sup>1</sup>

<sup>1</sup>Department of Mathematics, the Hong Kong University of Science and Technology, Hong Kong SAR, China

---

May 14th, 2019

**T**his report studies log return time series for 16 commodities, including energy, agriculture and metals. Basic time series analysis, i.e. statistics and correlation analysis, are conducted for all commodity series. Binomial tree method, Monte Carlo simulations with log-normal assumptions and Monte Carlo simulations with GJR-GARCH model are introduced to price Asian options on crude oil, ethanol, natural gas, gold and silver. Multiple conclusions are derived based on the pricing results. Future work is also discussed at the end of this report.

## 1 Introduction

Asian option, also called average value option, is essentially an exotic option, whose payoff depends on the path of the underlying price. As presented in equation 1, the payoff is determined by the strike price and the average underlying price (either arithmetic or geometric) during the option's life at the maturity. This averaging feature makes it popular among derivative traders, because the payoff will not be affected dramatically from unpredictable oscillations caused by the underlying's market performance close to maturity, and these kinds of options also provide a nice hedging choice to companies exposed to average prices, say oil production firms and farm producers. Most common underlying for Asian option are commodities and foreign exchange rate. In this report, Asian options on commodities with high liquidity will be utilized to construct numerical examples and examine proposed pricing methods.

$$\begin{aligned} C_T &= (A_T(t, T) - K)^+ \\ P_T &= (K - A_T(t, T))^+ \end{aligned} \quad (1)$$

Since their invention in late seventies, several pricing methods have been proposed, and generally speaking, they can be classified into three categories: semi-analytical, approximation and Monte Carlo. The semi-analytical pricing approach assumes the underlying price following a geometric Brownian motion (GBM). Taking logarithms, the average value of the GBM can be expressed in terms of the sum of normal random variables. Using fast Fourier transform (FFT) and convolutions, the final payoff can be calculated in an analytical way. (Carverhill and Clewlow, 1990) Benhamou applied the same method for discrete Asian options on underlying with non-normal returns in 2000. (Benhamou, 2000) Approximation methods approximate the real distribution of the average underlying price at the maturity with tractable ones, such as Edgeworth series expansion (Turnbull and Wakeman, 1991) and lognormal distributions (Levy, 1992). The Monte Carlo approach is a combination of stochastic models and Monte Carlo simulations. Kemna and Vorst proposed a Monte Carlo simulation scheme for the arithmetic average, as well as a variance reduction technique in their paper (Kemna and Vorst, 1990).

This report introduces an innovative Monte Carlo scheme to price a discretely monitored Asian options (average value options) on commodities by assuming that log returns of the underlying price follow GJR-GARCH model. Other non-GARCH models, i.e. binomial tree model and Monte Carlo simulations with constant volatility, are also implemented as a comparison to the proposed pricing method.

## 2 Theory

### 2.1 Commodities

A commodity, such as crude oil or wheat, is a basic good used in commerce that is interchangeable with other commodities of the same type. Commodities can be classified into a few categories according to their usages and values, say raw materials, basic resources, agricultural, and mining products, etc. The price of a commodity good is typically determined as a function of its market as a whole: established physical commodities possess actively traded spot and derivative markets.

Commodity futures are agreements to buy or sell a raw material at a specific date in the future at a particular price. The intrinsic value of one commodity future, at any time  $t$  during its life, can be expressed by equation 2. Since its intrinsic value is a linear function of its underlying price and the value of the contract is zero, we assume that the future price is also a linear function of the underlying price with a coefficient close to 1 and it contains all the volatility and underlying price information for us to price their options. Due to data source limitations, prices of commodities are not available to us, hence we study their future prices as an alternative data source for underlying prices, regardless of its minor delays.

$$F_t = S_t - f_t \quad (2)$$

In this project, 16 commodities from 3 broad categories, namely energy, agriculture and metals, are selected for analysis. For each of them, price data of the most liquid/major future contract is collected for analysis. Details as follow:

#### Energy:

- Crude Oil: WTI Financial Futures
- Natural Gas: Henry Hub Natural Gas Futures
- Refined Products: RBOB Gasoline Futures
- Biofuels: Chicago Ethanol (Platts) Futures
- Coal: Coal (API2) CIF ARA (ARGUS-McCloskey) Futures

#### Agriculture:

- Corns: Corns Futures
- Wheats: Chicago SRW Wheat Futures
- Soybean: Soybean Futures
- Soybean Meal: Soybean Meal Futures
- Soybean Oil: Soybean Oil Futures
- Livestock: Live Cattle Futures
- Livestock: Lean Hog Futures

#### Metals:

- Gold: Gold Futures
- Silver: Silver Futures
- Platinum: Platinum Futures
- Copper: Copper Futures

### 2.2 Asian Options

Asian option, also named average value option, is one of the most popular path dependent options among derivative traders. Its payoff is determined by the average underlying price over some pre-set period of time. The average price of the underlying asset can either determine the underlying settlement price or the option strike price. This is different from the case of usual European options and American options, whose payoff depends on underlying price at the exercise date. Hence Asian option is a typical form of exotic options. These exotic features provide risk reduction of market manipulation of the underlying instrument close to maturity (Kemna and Vorst, 1990). On top of that, the cost of Asian options is usually much lower than European or American options with same strike and maturity.

Suppose  $S_t$  is the underlying price at time  $t$ ,  $A_T(0, T)$  is the arithmetic average underlying price at maturity  $T$ , then we have

$$A_T(0, T) = \frac{1}{T} \sum_{t=0}^T S_t$$

in discrete form, or in continuous form like this

$$A_T(0, T) = \frac{1}{T} \int_0^T S_t dt$$

For geometric average underlying price, say  $\tilde{A}_T(0, T)$ , can be expressed as

$$\tilde{A}_T(0, T) = \left( \prod_{t=0}^T S_t \right)^{\frac{1}{T}}$$

in discrete form, or in continuous form like this

$$\tilde{A}_T(0, T) = \exp\left(\frac{1}{T} \int_0^T \log S_t dt\right)$$

For a European Asian call or put option with strike price  $K$ , its payoff becomes

$$C_T = (A_T(0, T) - K)^+$$

or

$$P_T = (K - A_T(0, T))^+$$

In this report, only arithmetic average underlying price in discrete form is taken into considerations for final payoff. We use strike prices and maturities from 5 commodity option contracts to construct numerical examples, say European Asian call/put options, for proposed pricing models. Considering different specifications, there are 319 option contracts in total, with maturity varying from 5 days to 5 years. Their contract specifications, together with latest settlement prices, are collected as reference. Here is the list:

- WTI Crude Oil Asian Option

- Chicago Ethanol (Platts) Asian Option
- Gold American Option
- Silver American Option
- Natural Gas European Option

## 2.3 Forward Risk Free Interest Rates

In any pricing model based on numerical methods, the risk free rate always plays an important part and will affect the final pricing results significantly. Since all of the interested options as well as their underlyings are traded in US exchanges, US treasury yield curve on the day we collect price data will be regarded as risk free rate in numerical models. To implement risk free rate and simulate movements of underlying prices, the forward curve can be constructed as the following:

$$\begin{aligned} f(t_i, t_{i+1}) &= \frac{\log(e^{r_{t_{i+1}}t_{i+1}}/e^{r_{t_i}t_i})}{t_{i+1} - t_i} \\ &= \frac{r_{t_{i+1}}t_{i+1} - r_{t_i}t_i}{t_{i+1} - t_i} \end{aligned} \quad (3)$$

In equation 3,  $r_{t_i}$  is the yield of a treasury instrument maturing at  $t_i$ , and it can be regarded as the risk free rate for  $(0, t_i)$ .  $f(t_i, t_{i+1})$  is the forward rate in  $(t_i, t_{i+1})$  derived by two treasury instruments with adjacent maturity dates.

## 2.4 Asian Option Pricing Models

### 2.4.1 Binomial Tree Model

The binomial tree model uses tree structure to simulate the path of the underlying asset over the whole life of Asian option and applies the forward shooting grid method to avoid the exponential growth of average price. In this case, we add extra grids to record the possible average price and restrict the number of possible average price to be linear growth. To update the average value when the asset value jumps, we use grid function to determine the updated average price.

Under Black-Scholes world, we assume that the volatility of asset value is constant. For the binomial tree, let the jump factor  $u, d$  to be  $u = \exp(\sigma\sqrt{\Delta t})$ ,  $d = \exp(-\sigma\sqrt{\Delta t})$  where  $\sigma$  is constant volatility, and  $\Delta t$  is the time step.

Denote  $S_j^n$  and  $A_k^n$  to be the asset value jumping upward  $j$  times and average price with index  $k$  at  $n$ -th time level, respectively. To restrict the possible values for  $F$  to a certain set of predetermined values, we limit the number of averaging values to some multiple of the number of values assumed by the asset price: Assume the coefficient to be  $1/\rho \in \mathbb{N}$ . For a given time step  $\Delta t$ , we let the asset value  $S_j^n$  and the average value  $A_k^n$  to be:

$$\begin{aligned} S_j^n &= S_0 e^{j\Delta W}, \quad A_k^n = S_0 e^{k\Delta Y}, \\ \Delta W &= \sigma\sqrt{\Delta t}, \quad \Delta Y = \rho\Delta W_j, \quad k \in \mathbb{N} \end{aligned} \quad (4)$$

Then consider at  $(n, j)$  node, for a upward jump from  $(n, j)$  to  $(n+1, j+1)$ , the asset price will change from  $S_j^n$  to  $S_{j+1}^{n+1}$ . Let  $A_{k^\pm(j)}^{n+1}$  to be the updated value changing from  $A_k^n$  by the upward move. Then by usual computation of updating average value, we have:

$$A_{k^\pm(j)}^{n+1} = \frac{(n+1)A_k^n + S_{j\pm 1}^{n+1}}{n+2} \quad (5)$$

Note that  $A_{k^\pm(j)}^{n+1}$  do not coincide with  $A_{k'}^{n+1} = S_0 e^{k'\Delta Y}$  for some  $k' \in \mathbb{N}$  in general, indicating that  $k^\pm(j)$  may not be integer. Recall  $A_{k^\pm(j)}^{n+1} = S_0 e^{k^\pm(j)\Delta Y}$  and  $S_{j\pm 1}^{n+1} = S_0 e^{(j \pm 1)\Delta W}$ , by (5), we equate the two parts and have the grid function:

$$k^\pm(j) = \frac{\ln \frac{(n+1) \exp(k\Delta Y) + \exp((j\pm 1)\Delta W)}{n+2}}{\Delta Y} \quad (6)$$

Let  $K'$  denote a subset of integer s.t.  $A_{k'}^{n+1} \leq A_{k^\pm(j)}^{n+1}$ . Define  $k_{floor}^\pm$  s.t.  $A_{k_{floor}^\pm}^{n+1} = \max_{k' \in K'} S_{k'}^{n+1}$ . In other words,  $k_{floor}^\pm = \lfloor k^\pm(j) \rfloor$ , then  $k_{ceil}^\pm = k_{floor}^\pm + 1$ . Define  $k_{floor} = \lfloor k \rfloor$  and  $k_{ceil} = k_{floor} + 1$ . At  $n$ th step, we have  $A_t \in [-S_n^n, S_n^n]$  and thus  $-n/\rho \leq k \leq n/\rho$ . This helps us to restrict the size of  $k$ . For a  $k \leq |S_n^n|$ , we have  $A_k^n \in [A_{k_{floor}}^n, A_{k_{ceil}}^n]$  and  $-n/\rho \leq k_{floor} < k_{ceil} \leq n/\rho$  at  $n$ th time step.

To solve the problem that that  $k^\pm(j)$  may not be integer, we define the linear interpolation formula that will be used in our FSG method.

Let  $c_{j,l}^n$  denote the numerical approximation to the Asian call value at  $(n, j)$  node with the averaging state variable assuming the value  $A_l^n$ . Similar notations for  $c_{j,l_{floor}}^n$  and  $c_{j,l_{ceil}}^n$ . For  $j \in \mathbb{R}/\mathbb{N}$ , the  $c_{j,l}^n$  will be approximated by linear interpolation:

$$c_{j,l}^n = c_{j,l_{floor}}^n + \varepsilon_l (c_{j,l_{ceil}}^n - c_{j,l_{floor}}^n) \quad (7)$$

where  $\varepsilon_l$  is the fraction of one step  $\Delta Y$  between  $\ln A_{l_{ceil}}^n$  and  $A_{l_{floor}}^n$ :

$$\varepsilon_l = \frac{\ln \frac{A_l^n}{A_{l_{floor}}^n}}{\Delta Y}, \quad A_l^n = A_{l_{floor}}^n e^{\varepsilon_l \Delta Y} \quad (8)$$

Finally we have the pricing formula under a binomial tree schemes is given by:

$$\begin{aligned} c_{j,k}^n &= \frac{1}{R} \left[ p c_{j+1, k^+(j)}^{n+1} + (1-p) c_{j-1, k^-(j)}^{n+1} \right] \\ &\approx \frac{1}{R} \left\{ p \left[ \varepsilon_{k^+(j)} c_{j+1, k_{ceil}^+}^{n+1} + (1 - \varepsilon_{k^+(j)}) c_{j+1, k_{floor}^+}^{n+1} \right] \right. \\ &\quad \left. + (1-p) \left[ \varepsilon_{k^-(j)} c_{j-1, k_{ceil}^-}^{n+1} + (1 - \varepsilon_{k^-(j)}) c_{j-1, k_{floor}^-}^{n+1} \right] \right\} \\ n &= N-1, \dots, 0, j = -n, -n+2, \dots, n, \\ k &\in \mathbb{N} \cap \left[ -\frac{n}{p}, \frac{n}{p} \right] \end{aligned} \quad (9)$$

where the risk neutral probability of jump upward is:

$$p = \frac{R-d}{u-d}, \quad R = e^{r_t(T-t)} \quad (10)$$

where  $r_t$  is the forward rate at time  $t$ , Finally, the terminal payoff of put and call are:

$$\begin{aligned} p_{j,k}^N &= \max(K - S_0 e^{k\Delta Y}, 0) \\ c_{j,k}^N &= \max(S_0 e^{k\Delta Y} - K, 0), \\ j &= -N, -N + 2, \dots, N \end{aligned} \quad (11)$$

#### 2.4.2 Monte Carlo Simulations with Constant Volatility

In this method, the underlying commodity price  $S_t$  is assumed to follow the stochastic process below:

$$dS_t = (r_t - \frac{1}{2}\sigma^2)S_t dt + \sigma S_t dW_t \quad (12)$$

where  $W_t$  is a standard Wiener process,  $r_t$  is the risk free interest rate at time  $t$ ,  $\sigma$  is a constant, representing the volatility of the underlying price's log return.

With US treasury yield curve, the forward the rate at time  $t$  can be used to represent  $r_t$  for simulations, and its calculation method is covered in section 2.3 already. With historical future prices of the commodity's major future contract, the volatility  $\sigma$  can be calculated as

$$\begin{aligned} r_i &= \log S_{t_i} - \log S_{t_{i-1}} \\ \sigma &= \sqrt{\frac{1}{N-1} \sum_{i=1}^N (r_i - \bar{r})^2} \end{aligned} \quad (13)$$

Given a commodity start price  $S_0$ , the underlying price movements can be simulated by the dynamics of its log returns  $\{r_i\}$ :

$$\begin{aligned} dS_t &= (r_t - \frac{1}{2}\sigma^2)S_t dt + \sigma S_t dW_t \\ d \log S_t &= (r_t - \frac{1}{2}\sigma^2)dt + \sigma dW_t \\ r_i &= r_{i-1} + (r_t - \frac{1}{2}\sigma^2)dt + \sigma dW_t \\ S_{t_i} &= S_{t_{i-1}} \exp(r_i \Delta t) = S_0 \exp(\sum_{k=0}^i r_k \Delta t) \end{aligned} \quad (14)$$

The average price  $A_T(0, T)$  at the maturity is essentially

$$A_T(0, T) = \frac{1}{N} \sum_{i=0}^N S_{t_i}$$

where  $t_0 = 0, t_N = T$ .

Without loss of generality, Asian call option is used to illustrate the Monte Carlo scheme, and the put option can be easily derived thereafter. The Asian call option payoff becomes

$$V_T = (A_T(0, T) - K)^+$$

With  $n$  paths of simulated underlying prices, the fair payoff at the maturity can be expressed as an average of the payoff generated from each path.

$$\bar{V}_T = \frac{1}{n} \sum_{i=1}^n V_T^i$$

Multiplying with corresponding discount factor, the value of the contract at present can be derived.

$$\begin{aligned} df(0, T) &= \exp(-\sum_{i=0}^{N-1} r_{t_i} \Delta t) \\ \bar{V}_0 &= df(0, T) \bar{V}_T \end{aligned} \quad (15)$$

where  $r_{t_i}$  is the forward risk free interest rate at  $t_i$ , and  $\Delta t = 1/252$ .

Since the underlying is only traded on business days, BUS/BUS date convention is implemented in the simulation scheme.

#### 2.4.3 Monte Carlo Simulations with GJR-GARCH model

As a comparison to the constant volatility MC scheme, GJR-GARCH model defines the log return of underlying price as a GARCH process. For a given time series  $\{S_t\}$ ,

$$\begin{aligned} r_t &= \log S_t - \log S_{t-1} \\ r_t &= \mu + \epsilon_t \\ \epsilon_t &= \sigma_t Z_t \end{aligned} \quad (16)$$

where  $Z_t$  follows a standard normal distribution and

$$\begin{aligned} \sigma_t^2 &= \omega + \sum_i \alpha_i \epsilon_{t-i}^2 + \sum_j \beta_j \sigma_{t-j}^2 \\ &+ \sum_k \gamma_k \epsilon_{t-k}^2 I_{\epsilon_{t-k} < 0} \end{aligned} \quad (17)$$

This model will illustrate the volatility behavior of the time series, and since the volatility is the source engine to generate simulated price paths, we assume that a more subtle volatility model will contribute to more accurate pricing results.

In this report, for each commodity, the best model configuration, say the number of  $\alpha, \beta, \gamma$  and whether using zero mean, is selected based on the following steps.

##### Step one: zero mean vs constant mean

Constant mean model will be considered at the first place. If the p-value of  $\mu$  is larger than 0.05, then the mean is considered as insignificant, and it will be removed to refit the model again as a consequence.

##### Step two: Ljung-Box test

The fitted model will be plugged into historical data, and the series of  $\{Z_t\}$  is derived. If  $\{Z_t\}$  passes Ljung-Box test at a significance level of 0.05, then it proves that the model well explains the historical time series. Otherwise, the model will be rejected.

##### Step three: BIC

Bayesian information criterion (BIC) is implemented as a criterion for model selection, instead of Akaike information criterion (AIC). This is because BIC has

more punishments on number of parameters when the data set is large comparing AIC.

$$\begin{aligned} AIC &= 2k - 2\ln(\hat{L}) \\ BIC &= \ln(n)k - 2\ln(\hat{L}) \end{aligned} \quad (18)$$

where  $\hat{L}$  is the maximized value of the likelihood function of the model,  $k$  is the number of parameters of the model to estimate, and  $n$  is the sample size.

If one model has a less BIC and less number of parameters than the other one, it will be selected as the temporary best model. Alternatively, if BIC of one model is 5% less than the other one, it will also win the selection.

With the best fitted GJR-GARCH model, the volatility of future log returns will be predicted. A series of  $\{\sigma_0(t)\}$  can be derived, since

$$\sigma_0(t) = E_0[\sigma(t)]$$

and then a path of underlying price can be derived from the generated log return series  $r_t$ , where

$$r_t = \mu + \sigma_0(t)Z_t$$

The rest pricing procedure is the same as the constant volatility scheme.

## 3 Process

### 3.1 Data Source

There are two types of data involved in this project, say commodity future price and option specifications. All of commodity future price data are retrieved from investing.com, a global financial portal owned by Fusion Media Limited. The time span starts from 2000-01-03 and ends at 2019-03-22. However, some futures, like silver and corn, may not have such a long price history. In this case, we take as earlier as we can, and all the future prices are available after year 2009. On the other hand, the option contract specifications and corresponding latest settle prices are recorded manually from CME group website, and there are  $319 \times 2 = 638$  options in total (319 call put pairs).

### 3.2 Data Preprocess

The commodity prices are retrieved one by one, hence to further analyze or utilize them, one must concat price series into one data table with respect to dates. The concatenation generates plenty of missing values due to the difference in trading day conventions. For example, futures of silver, platinum and US soybean oil can be traded on Sundays, and all other future contracts will have missing prices on those days. Besides, the time span also varies from one commodity to another, which produces missing values for those with shorter time span before their first date point.

Two preprocessing rules are developed, one for time series analysis while the other for pricing.

#### Preprocessing for time series analysis:

When two time series are selected for correlation analysis or collinearity analysis, only date points when both have data will be kept and all the date points with missing values will be removed.

#### Preprocessing for pricing:

Since Asian options only have one underlying, every historical time series will be handled independently. Thus date points with missing values will be dropped in this case.

For option specification data, the currency unit for strike price is different from each other, some are using USD cent, some are 0.001 USD. A currency divisor dictionary is specially created, and the strike price will all be converted to USD before pricing.

## 3.3 Correlation Analysis

Correlation analysis is a method of statistical evaluation used to study the strength of a relationship between two, numerically measured, continuous variables. In this step, the correlation coefficients among future prices and price log returns are calculated respectively by equation 19. Heat maps are used to reflect their relationships. The results are shown in section 4.1.

$$\begin{aligned} \rho_{xy} &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y} \\ &= \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2 \sum_{i=1}^n (y_i - \bar{y})^2}} \end{aligned} \quad (19)$$

## 3.4 Non-GARCH Model Pricing

This part illustrates the process of Non-GARCH model pricing.

### 3.4.1 Binomial Tree Method

We use binomial tree model to price the current Asian option value. We first calculate the historical volatility of log return. Then we compute the forward rate based the U.S. treasury yield curve. We will use the historical volatility and forward rate as the volatility and risk free rate of the model. For choosing the time step, we consider the total number of business days from the spot date to maturity date. If the time length reaches over 1 year, we restrict the time steps to 252, so that the computational speed could be faster. Since the underlying asset is futures and each futures contract contains more than one unit of goods, hence we choose the price the option that the underlying asset contains 1 unit good.

### 3.4.2 Monte Carlo Simulations

After data preprocessing, the option specifications are stored in one data table, and the underlying table is stored in the other. To price one option with Monte Carlo simulations, the option specifications are retrieved as strike price  $K$ , start date  $t$ , maturity date  $T$ , and underlying name.

Using the underlying name, the fixed volatility  $\sigma$  can be derived from historical underlying price data, as mentioned in section 2. The forward risk free interest rates are also calculated from treasury yield curve before simulations. The step size of Monte Carlo simulation is one business date, so the number of steps can be derived from Julian date difference of  $t$  and  $T$ .

For every simulation, the underlying price path can be generated by following equation 14. Then the average price at the maturity, say  $A_T(0, T)$  can be calculated based on the prices along the path. And the final discounted final payoff can be derived correspondingly, the average value of which will be the value priced by Monte Carlo simulation method.

### 3.5 GARCH Model Pricing

Before pricing the option, the best GARCH model needs to be selected from 2000 possible models ( $2 \times 10 \times 10 \times 10$ ). There are 10 possible values, say from 0 to 9, for each of  $\alpha$ ,  $\beta$  and  $\gamma$ , and the model can be either zero mean or constant mean. The details about model selection criterion is mentioned in section 2.4.3. With the best fitted GARCH model, predictions for mean of volatilities can be derived from the model. Since the step size is one business day, the number of predictions equal to the number of business days from commencing date and the maturity of the option.

The rest pricing procedure are the same as those of Monte Carlo simulations mentioned in last section, except for the simulation scheme for underlying prices, which follows in section 2.4.3.

## 4 Results

### 4.1 Correlation Results

The heap maps are shown in the appendices.

Figure 1 is the correlation coefficient heat map of future prices. It is clear that there are many strong positive linear relationships between some contracts, such as crude oil and gasoline or silver and platinum or soybean products. There is almost no strong negative linear relationships, but two commodities, say natural gas and live cattle, are less correlated to all other commodities. Except for those commodities made from same materials and those with competitive relationships, most of correlation coefficients are very high, say close to right color in heat map. These present randomness of correlations in commodity prices.

By contrast, figure 2 shows the correlation coefficient heat map of future log returns. Strong correlations are only presented between commodities within the same category, especially in agriculture or metals, and the log returns in same category are positively correlated.

### 4.2 Model results

| Underlying    | $(p, o, q)$ | $Q(20)$ | p-value | BIC   |
|---------------|-------------|---------|---------|-------|
| Crude Oil WTI | (1,1,1)     | 13.69   | 0.84    | 21129 |
| Ethanol       | (1,1,1)     | 39.91   | 0.005   | 13715 |
| Gold          | (1,0,1)     | 27.53   | 0.12    | 14403 |
| Natural gas   | (1,0,1)     | 23.48   | 0.2     | 24608 |
| Silver        | (1,0,5)     | 29.42   | 0.07    | 12151 |

Table 1: Model result: GJR model for each underlying futures

Table 1 shows the model we fit based on BIC criteria. The p-value of Ljung-Box test shows that the GJR-GARCH model for Crude Oil, Gold, Natural gas and Silver is adequate but the model for Ethanol is not. The order of first four GJR-GARCH models are all less than 1, indicates that the selected model is simple. The order of the last model(Silver) is (1, 0, 5), indicating that the volatility of Silver contract has long memory compared to other underlying futures.

**Convergence of the Monte Carlo simulation** Figure 3, 4, 6, 5 shows the sample mean of the call option payoff on Monte-Carlo simulation, The x-axis shows the simulation times and y-axis shows the estimated option value. These figures shows that the option value convergences to a single value and hence the option value from the Monte-Carlo simulation is rational. The prediction of the volatility is shown in appendix.

For the three pricing model(GARCH-MC, MC, BT), we use the following criterion to evaluate the performance of the model:

**ARE** We use ARE to evaluate the difference between market price and fair price derived from the model. The formula of ARE is:

$$ARE = \frac{1}{N} \sum_{j=1}^N \frac{|V_j^{model} - V_j^{market}|}{V_j^{market}} \times 100 \quad (20)$$

where  $N$  is the total number of the samples,  $V_j^{model}$  and  $V_j^{market}$  are the fair price from the model and the market price of the  $j$ -th sample, respectively. The results are shown in table 3 (Zhu and Ling, 2015).

Compared to market price, GARCH-MC model outperforms to other model when the underlying assets are Ethanol and Silver, and call option on Gold, put option on Crude Oil. The MC model outperform on call option on Crude Oil, put option on Natural gas. The BT model outperforms only on call option on Natural gas. Therefore, the fair price from GARCH-MC model is the closets price to the market price.

**Fair price of European option** Instead of market price, the European option price calculated from BS model can also be used as criterion. The analytical value calculated from the BS formula is also a fair price under BS world. Since Asian option uses the average price of the whole path to compute the payoff while European option only consider the asset price on maturity, the risk of holding one unit of European option should be larger than holding one unit of Asian option. Hence the fair price of the Asian option should be smaller than the European option value with the same specification of options.

Denote  $N_s$  as the total number that the fair price of Asian option is smaller than the corresponding European option with the same specification and  $N$  as the total number of samples. We use the proportion  $PS = N_s/N$  as the statistic to evaluate the model, the result is:

| Models   | $PS_{call}$ | $PS_{put}$ |
|----------|-------------|------------|
| GARCH-MC | 1           | 1          |
| MC       | 1           | 1          |
| BT       | 1           | 1          |

**Table 2:** Proportion of number of Asian option value smaller than European option value with same specification

From table 2, we can conclude that all the value from our model are rational.

## 5 Discussions

### 5.1 Data Limitations

Commodity futures are the assessment of raw materials trading in open market. Future prices are essentially estimates for the value of future commodities, instead of the spot price. The difference between two prices, say future price and spot price, will be affected by delivery time, risk-free rate, storage cost and convenience yield. Therefore, it will bring inaccuracies into the pricing process when using futures price not the commodity spot prices.

Besides, only WTI Crude Oil Asian option and Chicago Ethanol (Platts) Asian option are Asian options and all the rest options are either American or European options, specifications of which are used to construct numerical examples for the proposed pricing models. In future studies, more traded Asian options settings should be collected to examine these models.

### 5.2 Model Limitations

Chicago Ethanol (Platinum) futures has many duplicate prices on adjacent dates in the early stage, resulting in zero log returns. Moreover, there is no fitted GJR-GARCH model for it, and GJR-GARCH(1,1,1) is used to fit its time series of log returns. However, ARIMA(0,0,1)

can sufficiently explain the time series, we may consider applying ARIMA GARCH to fit the historical data and embedding into Monte Carlo scheme in further studies.

### 5.3 Monte Carlo Limitations

In Monte Carlo GARCH method, although Monte Carlo simulations are executed 10000 times, the standard deviation of payoffs generated is relatively large compared to constant volatility Monte Carlo method. This indicates that the number of paths might not be enough. Hence in GARCH Monte Carlo simulations, the final payoff could be a partial solution, and it may take million paths to obtain an output with lower variance. On top of that, the payoff variance from the proposed GARCH Monte Carlo simulation might underestimate the real variance, especially when the number of simulated path is insufficient. In this case, probability bounds analysis (PBA) can be implemented as an examination for partial information. Using PBA, the sparse simulations will provide far-apart upper and lower bounds, and this will well estimate the risk in pricing results.

### 5.4 Advanced GARCH Models

Affine GARCH model is described as the following:

$$r_t = r - \frac{1}{2}\sigma_t^2 + \sigma_t Z_t \tag{21}$$

$$\sigma_t^2 = \omega + \alpha_1(Z_{t-1} - \lambda\sigma_{t-1})^2 + \beta_1\sigma_{t-1}^2$$

According to Lorenzo, a semi-analytical formula for geometric Asian options can be provided when the underlying follows an affine GARCH process and the extreme asymmetry of this kind of models with non-normal innovations will price more accurately for options with very short time to maturity(Lorenzo, 2011). Hence, affine GARCH might provides a better pricing result if it is implemented to Monte Carlo simulation scheme.

## 6 Conclusions

In this report, 16 commodity future prices are studied, and 5 options of them are priced by three pricing methods, say binomial tree model, Monte Carlo simulations with constant volatility, and Monte Carlo simulations with GJR-GARCH volatility. Strong sector correlations are discovered in the 16 time series of commodity log returns. According to pricing result analysis, the ARE criterion shows that GJR-GARCH model is the most appropriate pricing model in three models implemented in this report. However, there are also a few limitations in this report, say the data limitations and Monte Carlo convergence limitations. There are also other advanced

GARCH models, say affine GARCH, to examine in the future studies.

## References

- Benhamou, Eric (2000). "Fast Fourier transform for discrete Asian options". In: *EFMA 2001 Lugano Meetings*.
- Carverhill, A. P. and L. J Clewlow (1990). "Valuing Average Rate (Asian) Options". In: *Risk* 3, pp. 33–36.
- Kemna, Angelien GZ and Antonius CF Vorst (1990). "A pricing method for options based on average asset values". In: *Journal of Banking & Finance* 14.1, pp. 113–129.
- Levy, Edmond (1992). "Pricing European average rate currency options". In: *Journal of International Money and Finance* 11.5, pp. 474–491.
- Lorenzo, Mercuri (2011). "Pricing Asian options in affine Garch models". In: *International Journal of Theoretical and Applied Finance* 14.02, pp. 313–333.
- Turnbull, Stuart M and Lee Macdonald Wakeman (1991). "A quick algorithm for pricing European average options". In: *Journal of financial and quantitative analysis* 26.3, pp. 377–389.
- Zhu, Ke and Shiqing Ling (2015). "Model-based pricing for financial derivatives". In: *Journal of Econometrics* 187.2, pp. 447–457.

## 7 Appendices

### 7.1 Figures

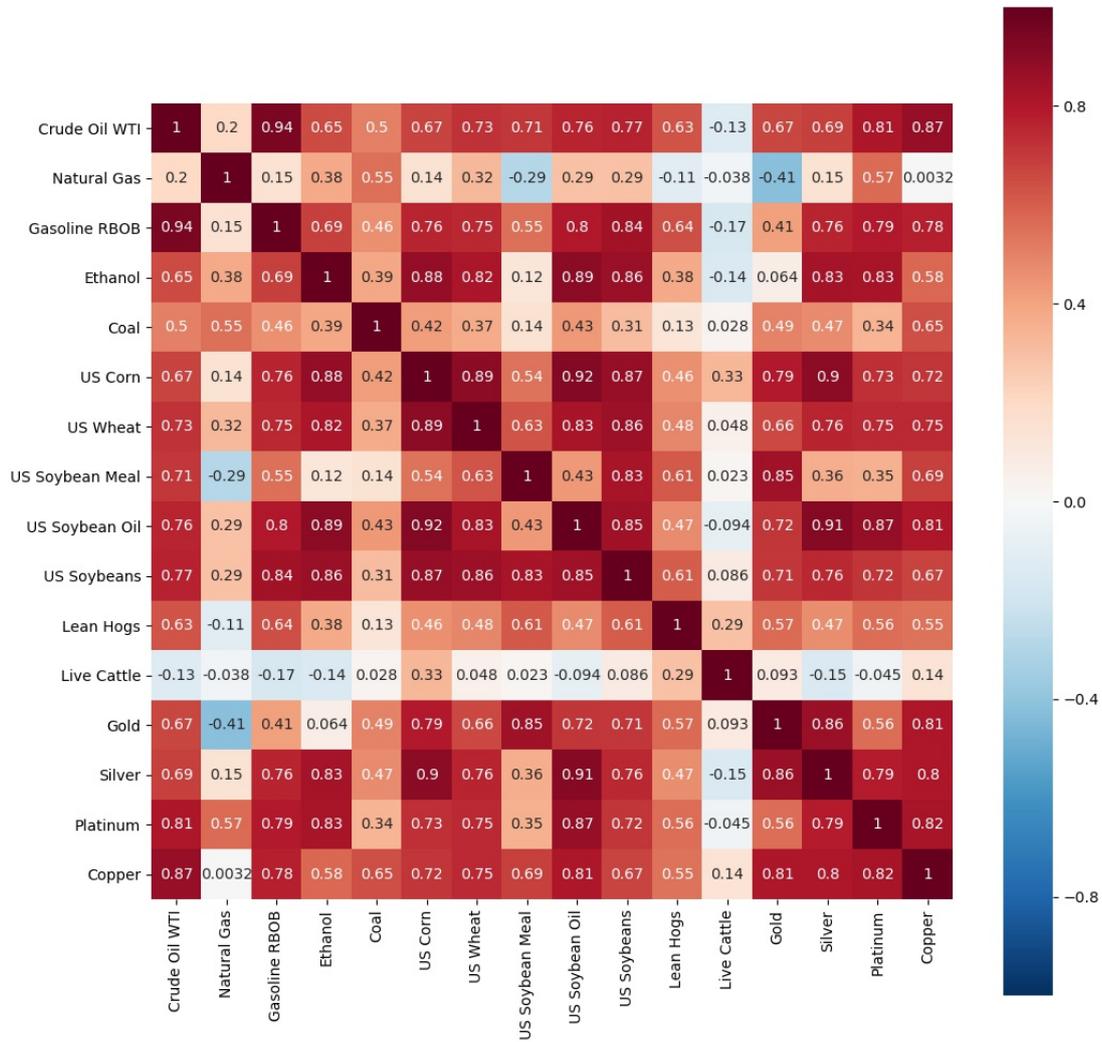


Figure 1: Correlation Analysis of Commodity Prices

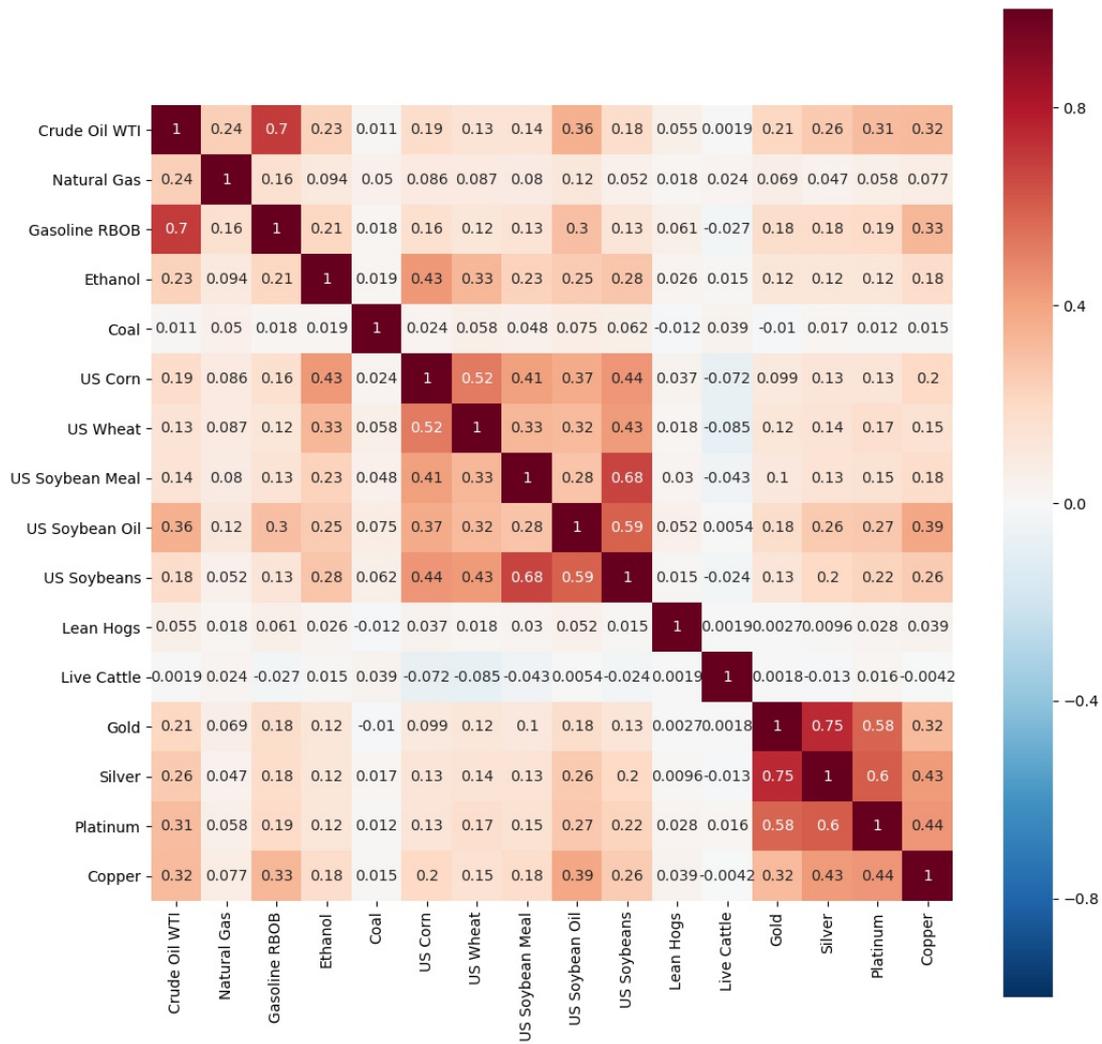


Figure 2: Correlation Analysis of Commodity Log Returns

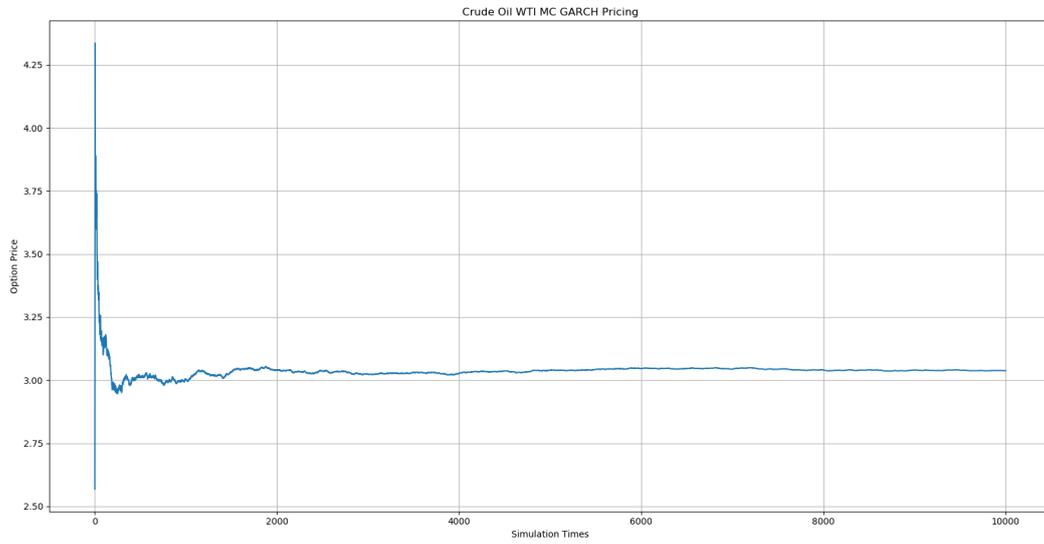


Figure 3: Convergence Plot for a Crude Oil Asian Option

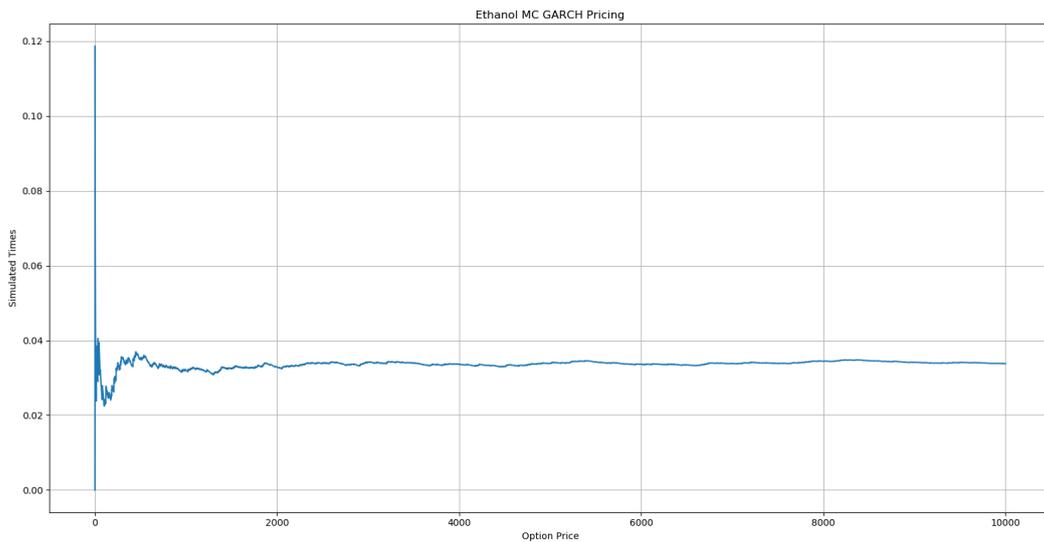


Figure 4: Convergence Plot for an Ethanol Option

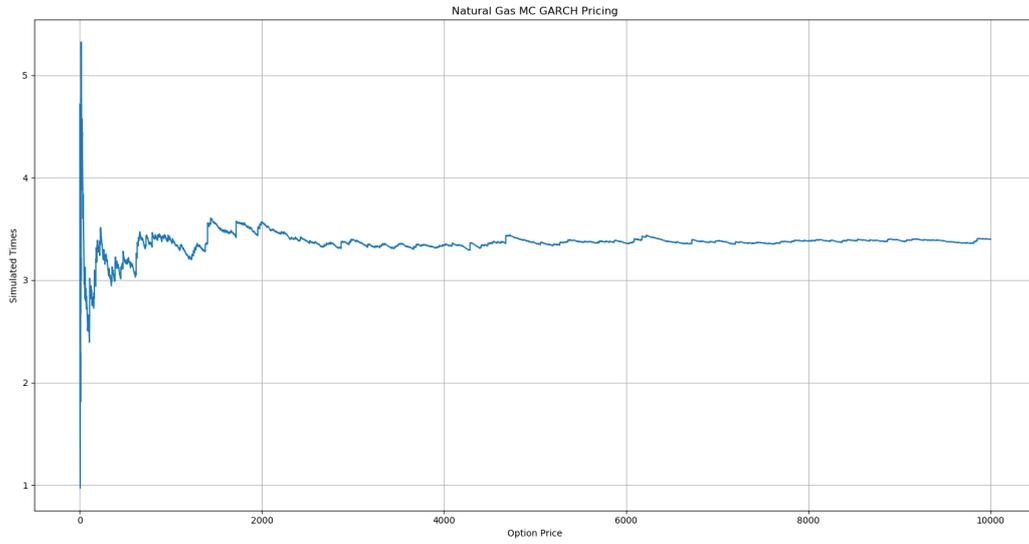


Figure 5: Convergence Plot for a Natural Gas Asian Option

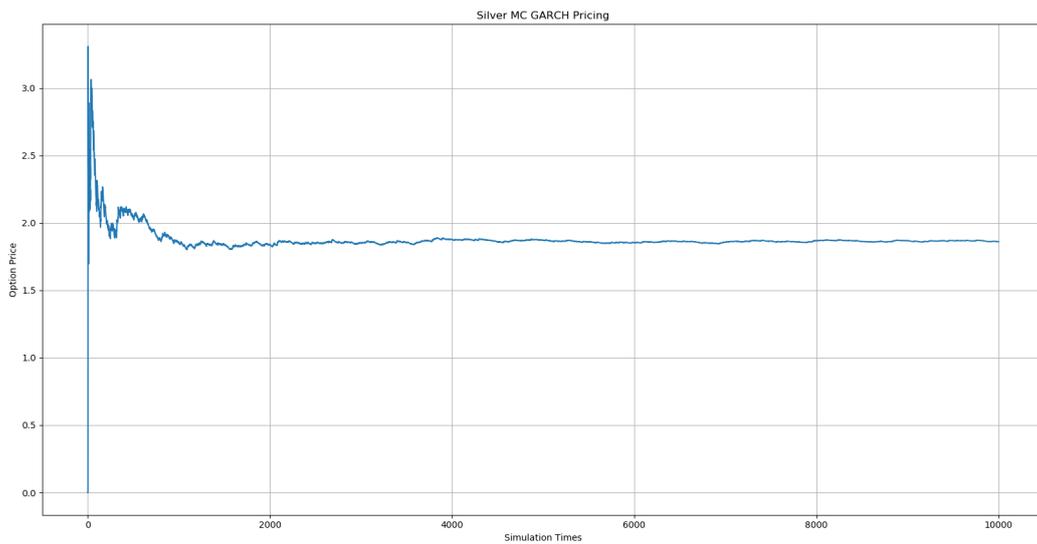
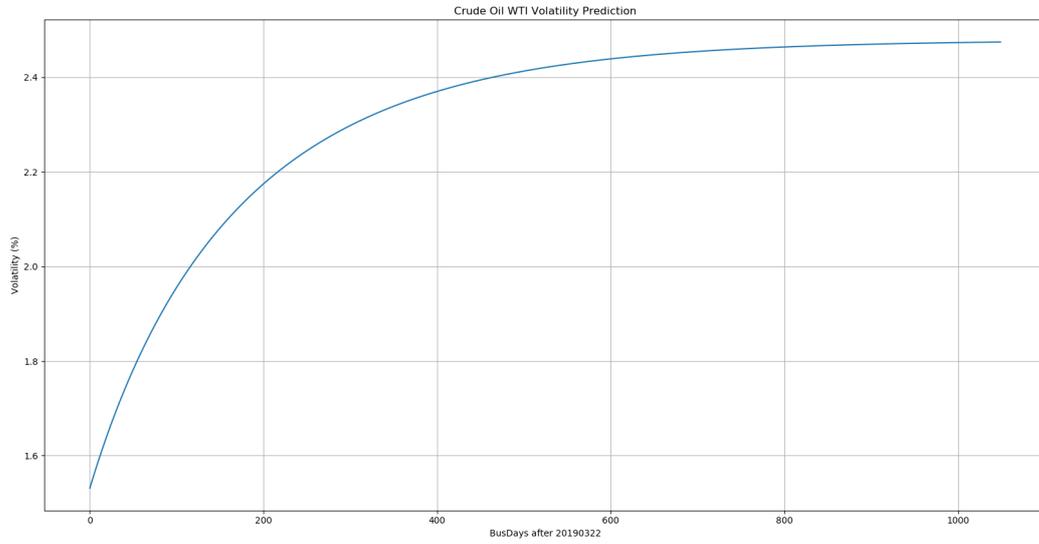
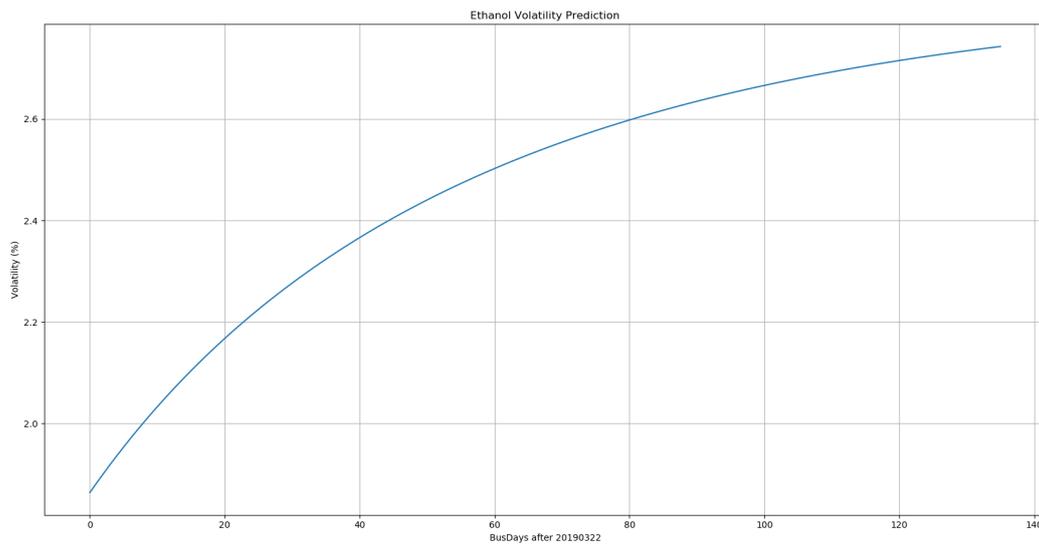


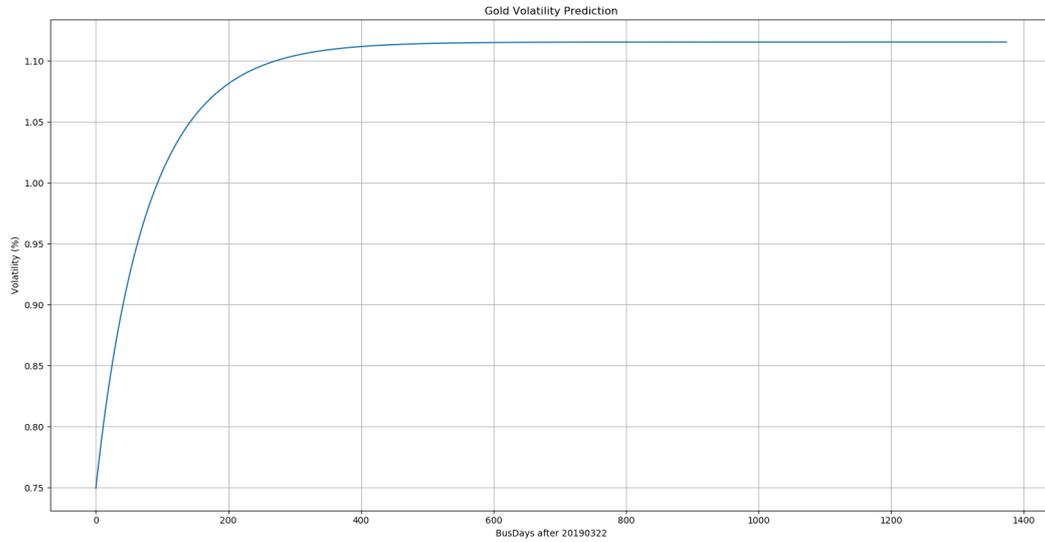
Figure 6: Convergence Plot for a Silver Asian Option



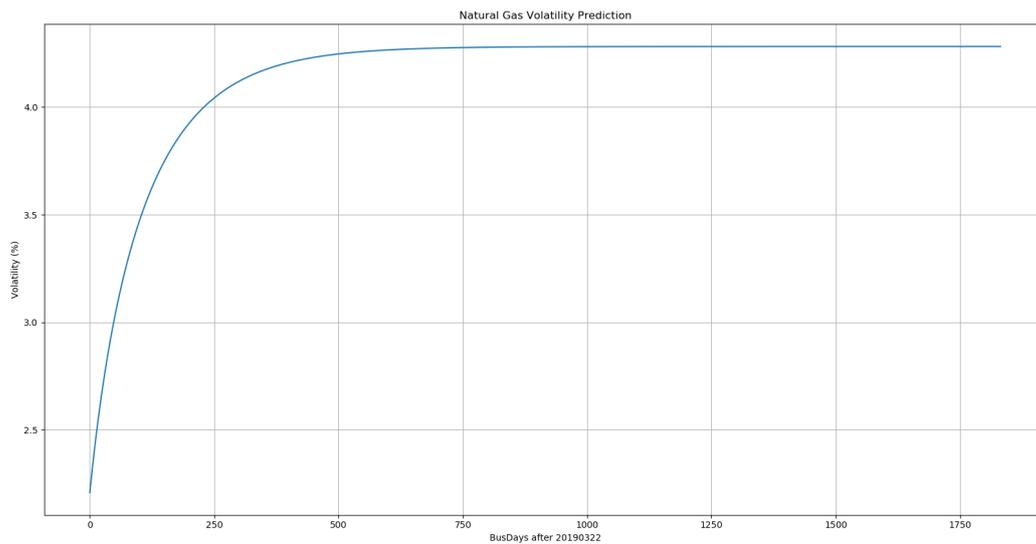
**Figure 7: Volatility Prediction on Crude Oil**



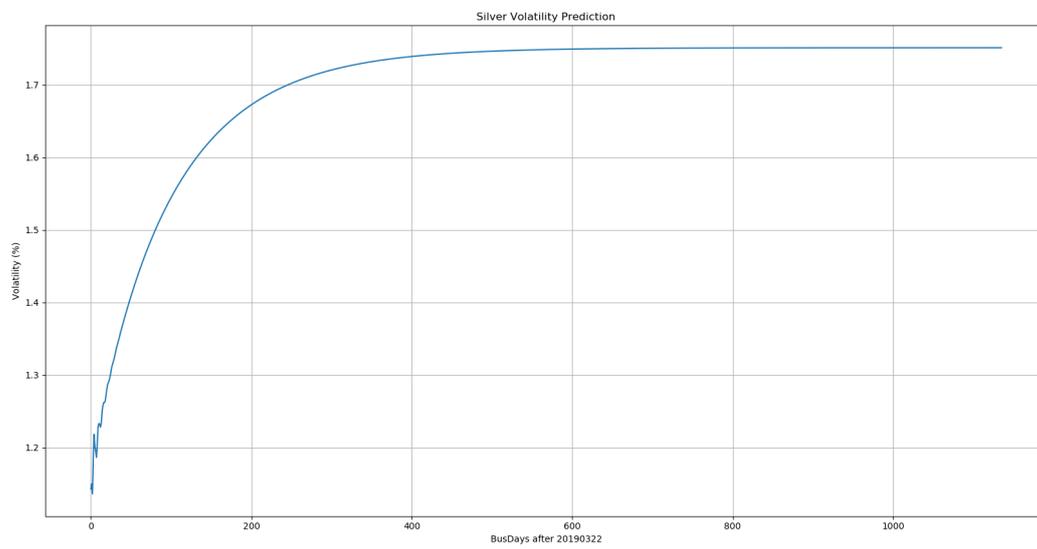
**Figure 8: Volatility Prediction on Ethanol**



**Figure 9:** Volatility Prediction on Gold



**Figure 10:** Volatility Prediction on Natural Gas



**Figure 11:** Volatility Prediction on Silver

## 7.2 Tables

| Models   | Crude Oil WTI |         | Ethanol |        | Gold    |         | Silver |         | Natural gas |         |
|----------|---------------|---------|---------|--------|---------|---------|--------|---------|-------------|---------|
|          | put           | call    | put     | call   | put     | call    | put    | call    | put         | call    |
| GARCH-MC | 46.679        | 304.798 | 10.757  | 25.258 | 72.799  | 84.586  | 17.455 | 26.597  | 215.232     | 324.858 |
| MC       | 83.121        | 122.238 | 56.035  | 73.732 | 52.874  | 91.129  | 64.379 | 89.397  | 64.891      | 84.962  |
| BT       | 402.088       | 819.106 | 45.110  | 62.466 | 106.801 | 149.124 | 83.969 | 107.557 | 264.242     | 72.697  |

**Table 3:** ARE for the Asian put and call option for each underlying futures.

## 7.3 Scripts

### 7.3.1 corr.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Sun April 12 15:53:00 2019
5
6  @author: jingqian
7  """
8
9  import pandas as pd
10 import numpy as np
11 import seaborn as sns
12 import matplotlib.pyplot as plt
13
14 data = pd.read_csv("../Cleaned Data/Cleaned Underlying Data.csv", index_col = 0)
15 newCols = ['Crude Oil WTI', 'Natural Gas', 'Gasoline RBOB', 'Ethanol', 'Coal',
16           'US Corn', 'US Wheat', 'US Soybean Meal', 'US Soybean Oil', 'US Soybeans',
17           'Lean Hogs', 'Live Cattle', 'Gold', 'Silver', 'Platinum', 'Copper']
18
19
20 price = data[newCols]
21
22 corr_price = price.corr()
23 corr_logRt = price.corr()
24
25 for col1 in newCols:
26     for col2 in newCols:
27         tmpdf = data[[col1, col2]].dropna()
28         corr_price[col1][col2] = tmpdf[[col1]].corrwith(tmpdf[col2]).values[0]
29         logdf = data[[col1, col2]].dropna()
30         logdf[col1] = (np.log(logdf[col1]) - np.log(logdf[col1].shift(1))) * 100
31         logdf[col2] = (np.log(logdf[col2]) - np.log(logdf[col2].shift(1))) * 100
32         logdf = logdf.dropna()
33         corr_logRt[col1][col2] = logdf[[col1]].corrwith(logdf[col2]).values[0]
34
35
36 plt.subplots(figsize = (12, 12))
37 sns_plot = sns.heatmap(corr_price, annot=True, vmin= -1, vmax= 1, square=True, cmap="RdBu_r")
38 fig = sns_plot.get_figure()
39 fig.savefig('corr_price.jpg')
40
41 plt.subplots(figsize=(12, 12))
42 sns_plot = sns.heatmap(corr_logRt, annot=True, vmin= -1, vmax= 1, square=True, cmap="RdBu_r")
43 fig = sns_plot.get_figure()
44 fig.savefig('corr.jpg')

```

*corr.py*

### 7.3.2 binomialTreePrice.py

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu April 14 10:23:02 2019
4
5  @author: Leheng Chen
6  """
7
8  import numpy as np
9
10 class asianOptionBinomialTree:
11
12     def __init__(self, num_steps, volatility, time_period, oneOverRho, interest_rate):
13         self.num_steps = num_steps
14         self.volatility = volatility
15         self.time_period = time_period
16         self.oneOverRho = oneOverRho
17         self.interest = np.array(interest_rate)
18         self.discount_factor = np.exp(-1 * self.interest * self.time_period)

```

```

19     self.half_len_grid = self.num_steps * self.oneOverRho
20
21     self.averagePriceTree = np.zeros(2 * self.num_steps * oneOverRho + 1)
22     self.assetPriceTree = np.zeros((self.num_steps + 1, self.num_steps + 1))
23     self.optionPriceTree = np.zeros((self.num_steps + 1, 2 * self.num_steps * self.
24         oneOverRho + 1))
25
26     def forwardInduction(self):
27         self.up_factor = np.exp(self.volatility * np.sqrt(self.time_period))
28
29         for i in range(self.num_steps + 1):
30             lower_bound = -i
31             for j in range(i + 1):
32                 self.assetPriceTree[i, j] = self.init_price * (self.up_factor ** lower_bound)
33                 lower_bound += 2
34
35         for j in range(2 * self.num_steps * self.oneOverRho + 1):
36             jump = j - self.half_len_grid
37             self.averagePriceTree[j] = self.init_price * (self.up_factor ** (jump / self.
38                 oneOverRho))
39
40         for s in range(self.num_steps + 1):
41             for k in range(2 * self.num_steps * self.oneOverRho + 1):
42                 self.optionPriceTree[s, k] = max(self.averagePriceTree[k] - self.strike, 0)
43
44     def grid(self, n, k, j, plus):
45         numerator = np.zeros((len(j), len(k)))
46         denominator = self.volatility * np.sqrt(self.time_period) / self.oneOverRho
47         for jj in j:
48             numerator[jj] = (n + 1) * self.up_factor ** (k / self.oneOverRho) + self.up_
49                 factor ** (jj + plus)
50             numerator[jj] = np.log(numerator[jj] / (n + 2))
51
52         return numerator / denominator
53
54     def backwardInduction(self):
55         delta_y = self.volatility * np.sqrt(self.time_period) / self.oneOverRho
56         proba_up = (1 / self.discount_factor - 1 / self.up_factor) / (self.up_factor - 1 /
57             self.up_factor)
58         # print(np.round(self.optionPriceTree))
59         for n in reversed(range(self.num_steps)):
60             k_idx = np.array([k for k in range(-n * self.oneOverRho, n * self.oneOverRho + 1)
61                 ])
62             j_idx = np.array([j for j in range(n + 1)])
63             k_up = self.grid(n, k_idx, j_idx, 1)
64             k_down = self.grid(n, k_idx, j_idx, -1)
65
66             j_idx_ext = np.repeat(j_idx[:, np.newaxis], len(k_idx), axis=1)
67
68             k_up_floor = np.maximum(np.floor(k_up + self.half_len_grid).astype(int), 0)
69             k_up_ceil = np.minimum(k_up_floor + 1, self.half_len_grid * 2)
70
71             # average_price_up = ((n + 1) * self.averagePriceTree[n, k] + self.assetPriceTree[
72                 n + 1, i + 1]) / (n + 2)
73             average_price_up = self.init_price * self.up_factor ** (k_up / self.oneOverRho)
74             factor_interpolation_up = np.log(average_price_up / self.averagePriceTree[k_up_
75                 floor]) / delta_y
76
77             option_price_up = factor_interpolation_up[0:] * self.optionPriceTree[j_idx_ext
78                 [0:], k_up_ceil[0:]] + \
79                 (1 - factor_interpolation_up[0:]) * self.optionPriceTree[j_idx_ext[0:], k_up_
80                 floor[0:]]
81
82             k_down_floor = np.maximum(np.floor(k_down + self.half_len_grid).astype(int), 0)
83             k_down_ceil = np.minimum(k_down_floor + 1, self.half_len_grid * 2)
84
85             # average_price_down = ((n + 1) * self.averagePriceTree[n, k] + self.
86                 assetPriceTree[n + 1, i - 1]) / (n + 2)
87             average_price_down = self.init_price * self.up_factor ** (k_down / self.oneOverRho
88                 )
89             factor_interpolation_down = np.log(average_price_down / self.averagePriceTree[k_
90                 down_floor]) / delta_y
91
92             # assert self.averagePriceTree[n + 1, k_down_floor] != 0

```

```

81     option_price_down = factor_interpolation_down[:(n + 1)] * self.optionPriceTree[j_
82         idx_ext[:(n + 1)], k_down_ceil[:(n + 1)]] + \
            (1 - factor_interpolation_down[:(n + 1)]) * self.optionPriceTree[j_idx_ext[:(n
            + 1)], k_down_floor[:(n + 1)]]
83
84     self.optionPriceTree[j_idx_ext[:(n + 1)], k_idx + self.half_len_grid] = proba_up[n
85         ] * option_price_up + (1 - proba_up[n]) * option_price_down
86     self.optionPriceTree[j_idx_ext[:(n + 1)], k_idx + self.half_len_grid] *= self.
87         discount_factor[n]
88     # print(np.round(self.optionPriceTree[j_idx_ext[:(n + 1)], k_idx + self.half_len_
89         grid]))
90
91     def getOptionPrice(self, init_price, strike):
92         self.init_price = init_price
93         self.strike = strike
94         self.forwardInduction()
95         self.backwardInduction()
96         return self.optionPriceTree[0, self.half_len_grid]

```

*binomialTreePricer.py*

### 7.3.3 biTreePriceSimulation.py

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu May  9 13:46:08 2019
4
5  @author: Leheng Chen
6  """
7
8  from binomialTreePricer import asianOptionBinomialTree
9  import pandas as pd
10 import numpy as np
11 from datetime import datetime, timedelta
12
13 uly_names = ['Crude Oil WTI', 'Ethanol', 'Gold', 'Silver', 'Natural Gas']
14 uly_init = df_uly[uly_names].tail(1)
15 df_opt['bdays'] = 1 + np.busday_count(df_opt['Start Date'].values.astype('datetime64[D]'), df_
16     opt['Maturity Date'].values.astype('datetime64[D]'))
17
18 df_uly_vol = df_uly[uly_names].std(skipna=True)
19
20 oneOverRho = 3
21 df_vols = pd.DataFrame([[0.3, 0.01, 0.4, 0.1, 0.001]], columns = uly_names)
22 df_units = pd.DataFrame([[0.01, 0.0001, 1, 0.001, 0.01]], columns = uly_names)
23 bdays_year = 252
24
25 # =====
26 # Define risk free rate, reference to US treasury yield curve as of 20190322
27 # https://www.treasury.gov/resource-center/data-chart-center/interest-rates/pages/TextView.
28 #   aspx?data=yieldYear&year=2019
29 # 1m, 2m, 3m, 6m, 1y, 2y, 3y, 5y, 7y, 10y, 20y, 30y
30 # =====
31 # Define risk free rate according to US
32 yieldCurveDict = {
33     '2019-04-22': 2.49,
34     '2019-05-22': 2.48,
35     '2019-06-22': 2.46,
36     '2019-09-22': 2.48,
37     '2020-03-22': 2.45,
38     '2021-03-22': 2.31,
39     '2022-03-22': 2.24,
40     '2024-03-22': 2.24,
41     '2026-03-22': 2.34,
42     '2029-03-22': 2.44,
43     '2039-03-22': 2.69,
44     '2049-03-22': 2.88
45 }
46
47 # Derive forward rates from US treasury yield curve

```

```

46 curvePoints = ['2019-03-22'] + list(yieldCurveDict.keys())
47
48 forwardCurveDict = {}
49 for i in range(len(yieldCurveDict)):
50     datePoint1 = curvePoints[i]
51     datePoint2 = curvePoints[i + 1]
52     if (datePoint1 == curvePoints[0]):
53         forwardCurveDict[datePoint2] = yieldCurveDict[datePoint2]
54     else:
55         yieldAtDate1 = yieldCurveDict[datePoint1]
56         yieldAtDate2 = yieldCurveDict[datePoint2]
57         busDateDiff1 = np.busday_count(curvePoints[0], datePoint1)
58         busDateDiff2 = np.busday_count(curvePoints[0], datePoint2)
59         forwardCurveDict[datePoint2] = float((yieldAtDate2 * busDateDiff2 - yieldAtDate1 *
60             busDateDiff1) / (busDateDiff2 - busDateDiff1))
61
62 # Function to get risk free rate given a date (datetime.date object)
63 def getRiskFreeRate(inputDate):
64     input_date = inputDate.date()
65     for i in range(len(forwardCurveDict)):
66         datePoint1 = datetime.strptime(curvePoints[i], '%Y-%m-%d').date()
67         datePoint2 = datetime.strptime(curvePoints[i + 1], '%Y-%m-%d').date()
68         if (input_date >= datePoint1 and input_date < datePoint2):
69             return forwardCurveDict[curvePoints[i + 1]]
70     return 0
71
72 for row in df_opt.index:
73     # Retrieve the name of the underlying
74     tmp_uly = df_opt['Underlying'][row][:-8]
75     tmp_strike = df_opt['Strike'][row]
76     tmp_maturity = df_opt['Maturity Date'][row]
77     tmp_steps = df_opt['bdays'][row]
78     if tmp_steps > bdays_year:
79         tmp_steps = bdays_year
80     tmp_init = uly_init[tmp_uly][0]
81     tmp_time_period = 1 / bdays_year
82     tmp_vol = df_uly_vol[tmp_uly]
83     tmp_ir = get_interest_rate(tmp_steps)
84     tmp_rates = [getRiskFreeRate(tmp_maturity - timedelta(d)) for d in range(tmp_steps)]
85
86     tmp_call = df_opt['Call'][row]
87     tmp_unit = df_units[tmp_uly][0]
88
89     pricer = asianOptionBinomialTree(tmp_steps, tmp_vol, tmp_time_period, oneOverRho, tmp_
90         rates)
91     sim = pricer.getOptionPrice(tmp_init, tmp_strike * tmp_unit)
92     print('underlying: %s; bdays: %d, strike: %6.3f, init: %6.3f --> simulate: %6.3f; actual
93         call: %6.3f' \
94         % (tmp_uly, tmp_steps, tmp_strike * tmp_unit, tmp_init, sim, tmp_call))

```

biTreePriceSimulation.py

### 7.3.4 garchPricer.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Apr 5 17:22:55 2019
5
6  @author: lueshen
7
8  get_best_model function is based on a script from Chu Song
9  """
10
11 import pandas as pd
12 import numpy as np
13 import statistics
14 import progressbar
15 from datetime import datetime
16 from arch import arch_model
17 from statsmodels.stats.diagnostic import acorr_ljungbox

```

```

18 import matplotlib.pyplot as plt
19 import warnings
20 warnings.filterwarnings('ignore')
21 warnings.simplefilter("ignore")
22 # =====
23 # Define risk free rate, reference to US treasury yield curve as of 20190322
24 # https://www.treasury.gov/resource-center/data-chart-center/interest-rates/pages/TextView.aspx?data=yieldYear&year=2019
25 # 1m, 2m, 3m, 6m, 1y, 2y, 3y, 5y, 7y, 10y, 20y, 30y
26 # =====
27 # Define risk free rate according to US
28 yieldCurveDict = {
29     '2019-04-22': 2.49,
30     '2019-05-22': 2.48,
31     '2019-06-22': 2.46,
32     '2019-09-22': 2.48,
33     '2020-03-22': 2.45,
34     '2021-03-22': 2.31,
35     '2022-03-22': 2.24,
36     '2024-03-22': 2.24,
37     '2026-03-22': 2.34,
38     '2029-03-22': 2.44,
39     '2039-03-22': 2.69,
40     '2049-03-22': 2.88
41 }
42
43 # Derive forward rates from US treasury yield curve
44 curvePoints = ['2019-03-22'] + list(yieldCurveDict.keys())
45
46 forwardCurveDict = {}
47 fwdCurveDict = {0:0}
48 for i in range(len(yieldCurveDict)):
49     datePoint1 = curvePoints[i]
50     datePoint2 = curvePoints[i + 1]
51     busDateDiff1 = np.busday_count(curvePoints[0], datePoint1)
52     busDateDiff2 = np.busday_count(curvePoints[0], datePoint2)
53     if (datePoint1 == curvePoints[0]):
54         forwardCurveDict[datePoint2] = yieldCurveDict[datePoint2]
55         fwdCurveDict[busDateDiff2] = yieldCurveDict[datePoint2]
56     else:
57         yieldAtDate1 = yieldCurveDict[datePoint1]
58         yieldAtDate2 = yieldCurveDict[datePoint2]
59         forwardCurveDict[datePoint2] = float((yieldAtDate2 * busDateDiff2 - yieldAtDate1 *
60             busDateDiff1) / (busDateDiff2 - busDateDiff1))
61         fwdCurveDict[busDateDiff2] = float((yieldAtDate2 * busDateDiff2 - yieldAtDate1 *
62             busDateDiff1) / (busDateDiff2 - busDateDiff1))
63
64 # Function to get risk free rate given a date (datetime.date object)
65 def getRiskFreeRateByDate(inputDate):
66     for i in range(len(forwardCurveDict)):
67         datePoint1 = datetime.strptime(curvePoints[i], '%Y-%m-%d').date()
68         datePoint2 = datetime.strptime(curvePoints[i + 1], '%Y-%m-%d').date()
69         if (inputDate >= datePoint1 and inputDate < datePoint2):
70             return forwardCurveDict[curvePoints[i + 1]]
71     return 0
72
73 # Function to get risk free rate given a business date count from 20190322
74 def getRiskFreeRate(dayCounts):
75     dayCountPoints = list(fwdCurveDict.keys())
76     for i in range(len(dayCountPoints)-1):
77         dayCount1 = dayCountPoints[i]
78         dayCount2 = dayCountPoints[i + 1]
79         if (dayCounts >= dayCount1 and dayCounts < dayCount2):
80             return fwdCurveDict[dayCount2]
81     return 0
82
83 # =====
84 # convenience yield - storage cost
85 # =====
86 commodityYieldDict = {
87     'Crude Oil WTI': 0.01,
88     'Ethanol': 0.01,
89     'Gold': 0.01,
90     'Natural Gas': 0.01,

```

```

89         'Silver':          0.01
90     }
91
92 # =====
93 # Utility functions
94 # =====
95 commodityCurrDict = {
96     'Crude Oil WTI':      0.01,
97     'Ethanol':           0.0001,
98     'Gold':              1,
99     'Natural Gas':       0.001,
100    'Silver':            0.01
101 }
102
103 def get_currency_divisor(commodity):
104     return commodityCurrDict[commodity]
105
106 # Get best GJR GARCH model for the log return series
107 def get_best_model(logRtSeries, pLimit, oLimit, qLimit, predictDays):
108     best_bic = np.inf
109     best_order = None
110     best_mdl = None
111     best_numParams = np.inf
112     isZeroMean = False
113
114     for pValue in range(pLimit + 1):
115         for oValue in range(oLimit + 1):
116             for qValue in range(qLimit + 1):
117                 isZeroMean = False
118                 try:
119                     tmp_mdl = arch_model(y = logRtSeries,
120                                         p = pValue,
121                                         o = oValue,
122                                         q = qValue,
123                                         dist = 'Normal')
124                     tmp_res = tmp_mdl.fit(update_freq=5, disp='off')
125
126                     # Remove mean if it's not significant
127                     if tmp_res.pvalues['mu'] > 0.05:
128                         isZeroMean = True
129                         tmp_mdl = arch_model(y = logRtSeries,
130                                             mean = 'Zero',
131                                             p = pValue,
132                                             o = oValue,
133                                             q = qValue,
134                                             dist = 'Normal')
135                         tmp_res = tmp_mdl.fit(update_freq=5, disp='off')
136
137                     tmp_bic = tmp_res.bic
138                     tmp_numParams = tmp_res.num_params
139                     tmp_wn_test = tmp_res.resid / tmp_res._volatility
140                     [lbvalue, pvalue] = acorr_ljungbox(tmp_wn_test, lags = 20)
141
142                     # Make sure the model pass Ljungbox Test, and fit the time series
143                     if pvalue[19] >= 0.05:
144                         if best_bic / tmp_bic > 1.05:
145                             best_bic = tmp_bic
146                             best_order = [pValue, oValue, qValue]
147                             best_mdl = tmp_res
148                             # Choose simpler model
149                         elif tmp_bic <= best_bic and tmp_numParams <= best_numParams:
150                             best_bic = tmp_bic
151                             best_order = [pValue, oValue, qValue]
152                             best_mdl = tmp_res
153                 except:
154                     continue
155
156 # Handle situations when all models don't pass Ljungbox Test
157 if (best_mdl == None):
158     tmp_mdl = arch_model(y = logRtSeries,
159                         p = 1,
160                         o = 1,
161                         q = 1,
162                         dist = 'Normal')

```

```

163     best_mdl = tmp_mdl.fit(update_freq=5, disp='off')
164
165     # Remove mean if it's not significant
166     if best_mdl.pvalues['mu'] > 0.05:
167         isZeroMean = True
168         tmp_mdl = arch_model(y = logRtSeries,
169                             mean = 'Zero',
170                             p = 1,
171                             o = 1,
172                             q = 1,
173                             dist = 'Normal')
174         best_mdl = tmp_mdl.fit(update_freq=5, disp='off')
175
176     best_bic = best_mdl.bic
177     best_order = [1, 1, 1]
178
179
180     # Test for first 20-lag
181     wn_test = best_mdl.resid / best_mdl._volatility
182     [lbvalue, pvalue] = acorr_ljungbox(wn_test, lags = 20)
183
184     output = {}
185     output['Zero Mean Model'] = isZeroMean
186     output['Best BIC'] = best_bic
187     output['Best Order'] = best_order
188     output['Best Model'] = best_mdl
189     volForecasts = best_mdl.forecast(horizon=predictDays)
190     output['Vol Predictions'] = np.sqrt(volForecasts.residual_variance.iloc[-1].values)
191     output['Ljungbox Test Statistics'] = lbvalue[19]
192     output['Ljungbox Test pvalue'] = pvalue[19]
193
194     return output
195
196 # Get affine garch model [not implemented yet]
197 def get_affine_garch(logRtSeries):
198     tmp_mdl = arch_model(y = logRtSeries, p = 1, q = 1, dist = 'Normal')
199     tmp_res = tmp_mdl.fit(update_freq=5, disp='off')
200     tmp_bic = tmp_res.aic
201
202     output = {}
203     output['Best AIC'] = tmp_bic
204     output['Best Model'] = tmp_res
205
206     return output
207
208 # Use GARCH vol to price Asian option (Monte Carlo)
209 def garchPricer(startPrice, strikePrice, garchModel, expBusDays, numPath):
210     sumCallPrice = 0
211     sumPutPrice = 0
212     res = garchModel['Best Model']
213     mu = 0
214     if garchModel['Zero Mean Model'] == False:
215         mu = res.params['mu']
216     vol = garchModel['Vol Predictions']
217     simulatedUlyPrices = []
218     callPrices = []
219     putPrices = []
220
221     for i in range(numPath):
222         simulatedUlyPrice = [startPrice]
223         ulyPrice = startPrice
224         sumUlyPrice = 0
225         dt = 1 / 252
226         randomGenerator = np.random.normal(0, 1, expBusDays)
227         discountRate = 0
228
229         # Simulated one path of underlying price
230         for j in range(expBusDays):
231             zt = randomGenerator[j]
232             rt = getRiskFreeRate(j) / 100
233             dLogSt = mu + vol[j] / 100 * zt
234             discountRate += rt
235             ulyPrice *= np.exp(dLogSt)
236             simulatedUlyPrice.append(ulyPrice)

```

```

237         sumUlyPrice += ulyPrice
238
239     avgUlyPrice = sumUlyPrice / expBusDays
240     simulatedUlyPrices.append(simulatedUlyPrice)
241
242     # True for call, false for put
243     callPrice = max(avgUlyPrice - strikePrice, 0) * np.exp(-discountRate * dt)
244     putPrice = max(strikePrice - avgUlyPrice, 0) * np.exp(-discountRate * dt)
245     callPrices.append(callPrice)
246     putPrices.append(putPrice)
247     sumCallPrice += callPrice
248     sumPutPrice += putPrice
249
250     output = {
251         'Call': sumCallPrice / numPath,
252         'Put': sumPutPrice / numPath,
253         'Call Prices': callPrices,
254         'Put Prices': putPrices,
255         'Call STD': statistics.stdev(callPrices),
256         'Put STD': statistics.stdev(putPrices),
257         'Simulated Price': simulatedUlyPrices
258     }
259
260     return output
261
262 # Pricing Asian Option with fixed vol (Monte Carlo)
263 def nonGarchPricer(startPrice, strikePrice, vol, costYield, expBusDays, numPath):
264     sumCallPrice = 0
265     sumPutPrice = 0
266     simulatedUlyPrices = []
267     callPrices = []
268     putPrices = []
269
270     for i in range(numPath):
271         simulatedUlyPrice = [startPrice]
272         ulyPrice = startPrice
273         sumUlyPrice = 0
274         dt = 1 / 252
275         randomGenerator = np.random.normal(0, np.sqrt(dt), expBusDays)
276         discountRate = 0
277
278         # Simulated one path of underlying price
279         for j in range(expBusDays):
280             dWt = randomGenerator[j]
281             rt = getRiskFreeRate(j) / 100
282             dLogSt = (rt - costYield - (vol / 100)**2 / 2) * dt + vol / 100 * dWt
283             discountRate += rt
284             ulyPrice *= np.exp(dLogSt)
285             simulatedUlyPrice.append(ulyPrice)
286             sumUlyPrice += ulyPrice
287
288         avgUlyPrice = sumUlyPrice / expBusDays
289         simulatedUlyPrices.append(simulatedUlyPrice)
290
291         callPrice = max(avgUlyPrice - strikePrice, 0) * np.exp(-discountRate * dt)
292         putPrice = max(strikePrice - avgUlyPrice, 0) * np.exp(-discountRate * dt)
293         callPrices.append(callPrice)
294         putPrices.append(putPrice)
295         sumCallPrice += callPrice
296         sumPutPrice += putPrice
297
298     output = {
299         'Call': sumCallPrice / numPath,
300         'Put': sumPutPrice / numPath,
301         'Call Prices': callPrices,
302         'Put Prices': putPrices,
303         'Call STD': statistics.stdev(callPrices),
304         'Put STD': statistics.stdev(putPrices),
305         'Simulated Price': simulatedUlyPrices
306     }
307
308     return output
309
310 # =====

```

```

311 # Data Preprocessing
312 # =====
313
314 # Load underlying data from git
315 df_uly = pd.read_csv("../Underlying Data/Underlying Data.csv", sep=',')
316
317 # Preprocess dataframe, set up index, fill nan with latest previous values
318 df_uly.index = pd.to_datetime(df_uly['Date']).dt.date
319 df_uly = df_uly.drop('Date', axis = 1)
320
321 # Load option data from git
322 df_opt = pd.read_csv("../Option Price Data/Option Data.csv", sep=',')
323
324 # Preprocess dataframe, convert dates, calculate days to maturity
325 df_opt.columns = ['Start Date', 'Maturity Date', 'Strike', 'Put', 'Call', 'Underlying']
326 df_opt['Maturity Date'] = pd.to_datetime(df_opt['Maturity Date']).dt.date
327 df_opt['Start Date'] = '2019-3-25'
328 df_opt['Start Date'] = pd.to_datetime(df_opt['Start Date']).dt.date
329 try:
330     df_opt['Exp BusDays'] = np.busday_count(df_opt['Start Date'], df_opt['Maturity Date']) + 1
331 except:
332     # Sometimes line above may not work, use method below as an alternative
333     tmp_list = []
334     for i in range(len(df_opt['Start Date'])):
335         tmp_list.append(np.busday_count(df_opt['Start Date'][i], df_opt['Maturity Date'][i]) +
336                        1)
337     df_opt['Exp BusDays'] = tmp_list
338 # =====
339 # Get best GARCH model and other info for underlyings whose options we will price later
340 # =====
341 masterObj = {}
342
343 ulyList = list(np.unique(df_opt['Underlying']))
344 for underlying in progressbar.progressbar(ulyList):
345     tmp_uly = underlying[:-8]
346     TS_uly = df_uly[tmp_uly].dropna()
347     TS_logRt = (np.log(TS_uly) - np.log(TS_uly.shift(1))).dropna() * 100
348     TS_logRt = TS_logRt[TS_logRt!=0]
349     max_expDays = max(df_opt[df_opt['Underlying']==underlying]['Exp BusDays'])
350     masterObj[tmp_uly] = {
351         'Start Price': TS_uly[-1],
352         'Volatility': statistics.stdev(TS_logRt),
353         'Garch Model': get_best_model(TS_logRt, 10, 10, 10, max_expDays)
354     }
355 # =====
356 # Execute garchPricer and collect results
357 # =====
358 nonGarchFairPriceCall = []
359 nonGarchFairPricePut = []
360 garchFairPriceCall = []
361 garchFairPricePut = []
362 nonGarchCallStd = []
363 nonGarchPutStd = []
364 garchCallStd = []
365 garchPutStd = []
366 resultsObj = {}
367
368
369 # Number of Monte Carlo simulated paths
370 numPath = 10000
371
372 # Loop through options
373 for row in progressbar.progressbar(df_opt.index):
374     # Retrieve the name of the underlying
375     tmp_uly = df_opt['Underlying'][row]
376     tmp_strike = df_opt['Strike'][row] * get_currency_divisor(tmp_uly)
377     tmp_maturity = df_opt['Maturity Date'][row]
378     tmp_expBusDays = df_opt['Exp BusDays'][row]
379
380     # Retrieve the underlying historical data
381     tmp_s0 = masterObj[tmp_uly]['Start Price']
382     tmp_vol = masterObj[tmp_uly]['Volatility']

```

```

384 tmp_model = masterObj[tmp_uly]['Garch Model']
385
386 nonGarchResults = nonGarchPricer(tmp_s0, tmp_strike, tmp_vol, commodityYieldDict[tmp_uly],
    tmp_expBusDays, numPath)
387 nonGarchFairPriceCall.append(nonGarchResults['Call'])
388 nonGarchFairPricePut.append(nonGarchResults['Put'])
389 nonGarchCallStd.append(nonGarchResults['Call STD'])
390 nonGarchPutStd.append(nonGarchResults['Put STD'])
391
392 garchResults = garchPricer(tmp_s0, tmp_strike, tmp_model, tmp_expBusDays, numPath)
393 garchFairPriceCall.append(garchResults['Call'])
394 garchFairPricePut.append(garchResults['Put'])
395 garchCallStd.append(garchResults['Call STD'])
396 garchPutStd.append(garchResults['Put STD'])
397
398 resultsObj[row] = {
399     'GARCH MC': garchResults,
400     'Non-GARCH MC': nonGarchResults
401 }
402
403 df_opt['Put (MC non-GARCH)'] = nonGarchFairPricePut
404 df_opt['Put (MC non-GARCH) STD'] = nonGarchPutStd
405 df_opt['Call (MC non-GARCH)'] = nonGarchFairPriceCall
406 df_opt['Call (MC non-GARCH) STD'] = nonGarchCallStd
407 df_opt['Put (MC GARCH)'] = garchFairPricePut
408 df_opt['Put (MC GARCH) STD'] = garchPutStd
409 df_opt['Call (MC GARCH)'] = garchFairPriceCall
410 df_opt['Call (MC GARCH) STD'] = garchCallStd

```

*garchPricer.py*