Monte Carlo Pricing of Temperature Weather Derivatives.

By Jorge C Gibert

Submitted as Partial Fulfillment of the Graduate Certification in

Quantitative Environmental Finance

Department of Mathematics and Statistics, Florida International University, Spring 2024

1. INTRODUCTION

A weather derivative is a financial instrument designed to hedge or speculate on the risks associated with adverse weather conditions. Unlike traditional financial assets such as stocks or bonds, the value of weather derivatives is linked to atmospheric variables like temperature, precipitation, wind speed, or other meteorological factors (Brewer, 2000). Industries that are vulnerable to weather variability use weather derivatives to hedge against weather-related risks. These industries include agriculture, energy, tourism, and insurance.

Weather derivative contracts typically involve two parties: a contract buyer and a contract seller. The buyer gets protection against weather-related risks by securing a prearranged payout based on specific weather conditions. The seller assumes the associated risk in exchange for a premium from the buyer (Hartmann & Geyer, 2005). Usually, weather derivatives exist, each tied to specific weather parameters like Temperature, Precipitation, Wind, etc.

Temperature Derivatives: Temperature derivatives rely on temperature indexes such as Heating Degree Days (HDD) or Cooling Degree Days (CDD) to measure deviations from predetermined temperature thresholds. For example, during the winter season, the buyer of an HDD contract might be entitled to a payout if the average temperature falls below a certain level during the winter months, being compensated for the loss associated with reduced heating demand.

Precipitation Derivatives: These derivatives are linked to measures of rainfall or snowfall in specific regions. They could be used by agricultural producers to hedge against losses resulting from drought or excessive precipitation.

Wind Derivatives: Wind derivatives are associated with wind speed or wind energy production and are frequently used by wind farm operators to manage revenue volatility stemming from fluctuations in wind conditions.

By allowing market participants to transfer and manage these risks, weather derivatives contribute to overall market efficiency and stability (Weng & Zeng, 2017). The purpose of this paper is to develop a method, to price Temperature Weather derivatives using Montecarlo Modeling.

Brief History of Temperature Weather Derivatives

The history of temperature weather derivatives dates back to the last decade of the 20th century when financial markets began to recognize the economic impact of weather fluctuations on various industries.

The inception of weather derivatives began in July 1996 with the first over-the-counter (OTC) trade written by Aquila Energy, which developed a dual-commodity hedge for Consolidated Edison Co. Since then, OTC trading of weather derivatives has persisted, gaining momentum from 1997 onward. As market demand expanded, the Chicago Mercantile Exchange (CME) played a pivotal role by introducing exchange-traded weather futures and options, including contracts based on temperature indexes on September 22, 1999 (Brewer, 2000). These contracts provided market participants with a standardized mechanism for managing temperature-related risks. After its introduction, Temperature weather derivatives gained traction among corporations seeking to mitigate the financial impact of weather fluctuations. These derivatives became increasingly popular in industries sensitive to temperature variations, such as energy utilities, agricultural producers, and retail businesses. The early 2000s witnessed further innovation and expansion in the temperature derivatives market, with the development of sophisticated pricing models and risk management strategies.

It is estimated that about 30% of the U.S. economy is directly affected by the weather. CME Group offers Weather futures and options which are index-based products geared to average seasonal and monthly weather in 18 cities around the world (CME Group Inc., 2021)

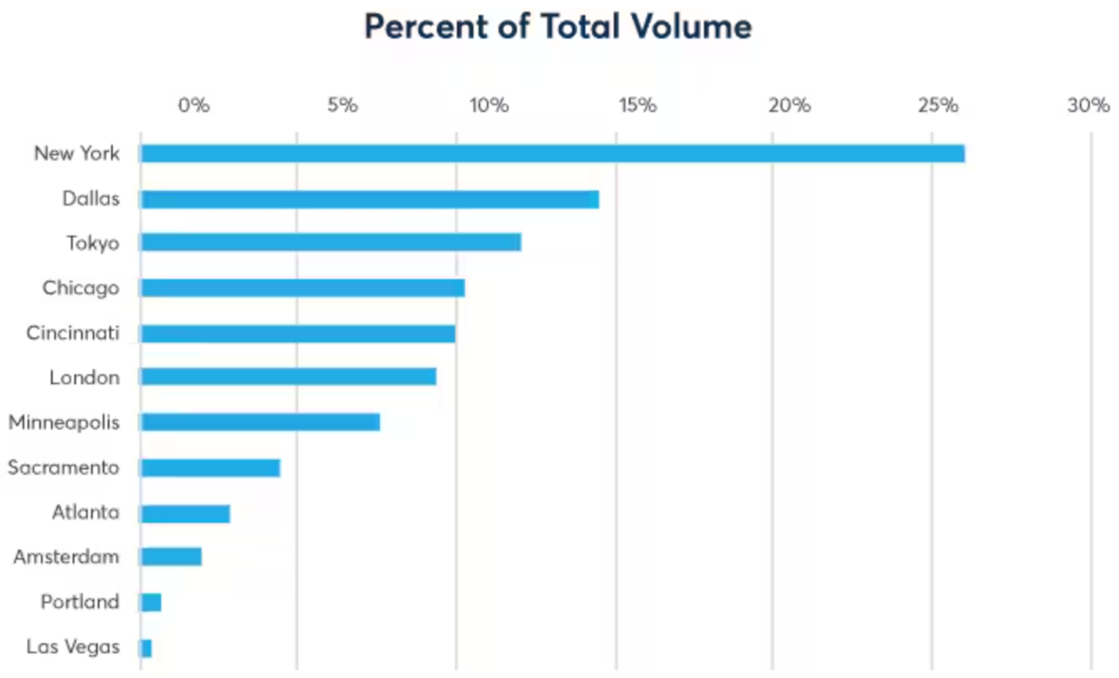


Fig. 1.1. Weather derivative percent of Total value by market. Source: CME Group Inc, 2021

Academic research contributed to the advancement of mathematical models for pricing temperature derivatives, enhancing the understanding of temperature dynamics and their impact on derivative values (Hartmann & Geyer, 2005) (Considine, 2000). As the temperature derivatives market matured, it diversified to include a range of contract types and underlying indexes. Heating Degree Days (HDD) and Cooling Degree Days (CDD) emerged as the predominant indexes used in temperature derivatives, reflecting the demand for heating and cooling services based on temperature thresholds.

In recent years, temperature weather derivatives have become an integral component of risk management strategies for businesses exposed to temperature-related risks. The market continues to evolve with the introduction of new products and innovations aimed at addressing the diverse needs of market participants

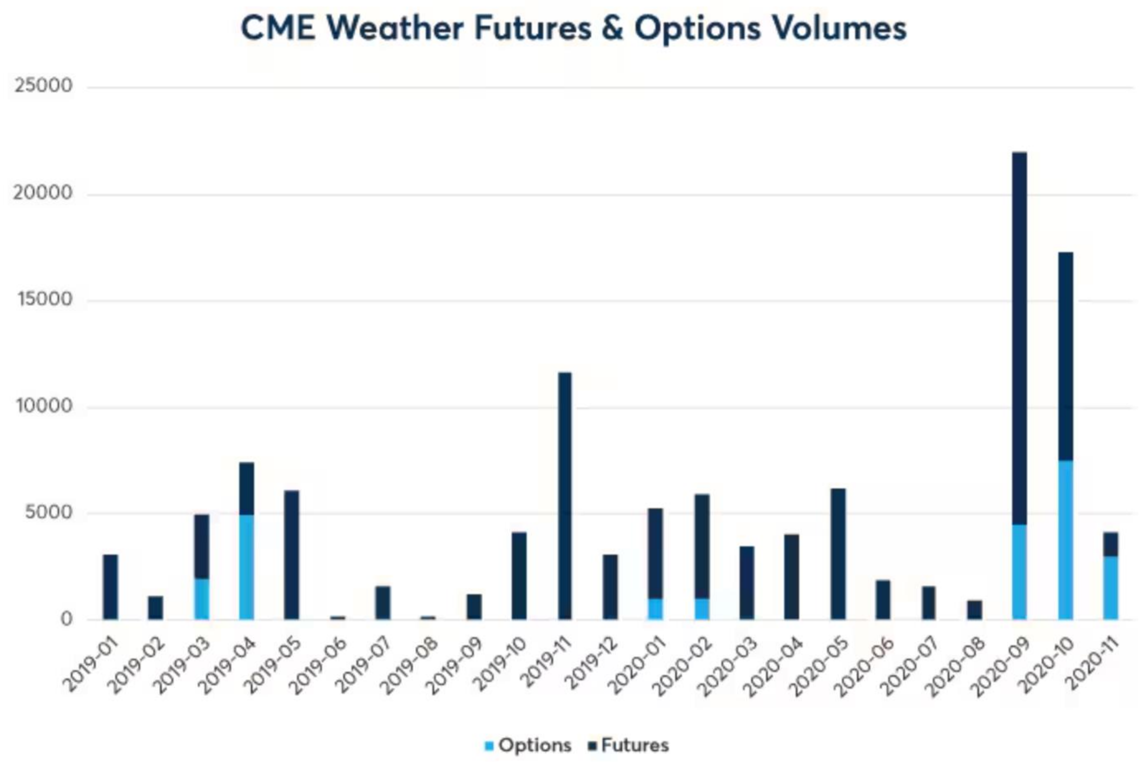


Figure 1.2. CME Option and Future Traded Values 2019-2020. Source CME Group Inc., 2021

Features of Temperature Weather Derivatives Contracts

Temperature derivatives are characterized by several features:

1. **Settlement Mechanism:** Settlement of temperature derivatives can occur in cash or physical delivery. Cash settlement involves a payment based on the difference between the actual and reference temperature multiplied by a predetermined contract value.
2. **Underlying Index:** The most common indexes used in temperature derivatives are Heating Degree Days (HDD) and Cooling Degree Days (CDD). HDD measures deviations below a reference

temperature, indicating heating demand, while CDD measures deviations above a reference temperature, indicating cooling demand. For a given day $\in N$

$$CDD_n = (T_n - 65)^+ \qquad 1.1$$

$$HDD_n = (65 - T_n)^+ \qquad 1.2$$

Where T_n is the average of the maximum daily temperature (T_{max}) and the minimum daily temperature. In this paper, T_n was calculated as

$$T_n = \frac{T_{max} - T_{min}}{2}. \qquad 1.3$$

3. **Contract Specifications:** Temperature derivatives specify the reference temperature, contract period, and payout structure. For example, a contract may pay out if the cumulative HDD during winter months exceeds a certain threshold (usually 65 degrees Fahrenheit in the US). The buyer of the option will receive a payout that is a function of the cumulative index over a period with a number of days *N*, i.e.

$$\xi = f(DD) \qquad 1.4$$

So DD for Heating (H_n, usually May 15 to October 15 for OTC) and Cooling season (C_n, usually November 15 to March 15 for OTC)

$$DD = H_n = HDD^N = \sum_{n=1}^{N}(T_n - 65)^+ \qquad 1.5$$
$$DD = C_n = CDD^N = \sum_{n=1}^{N}(T_n - 65)^+ \qquad 1.6$$

Two popular payoff functions are call options with caps and put options with floors. In the case of a call option with a cap, the payoff function is

$$\xi = min\{\alpha(DD - K)^+, C\} \qquad 1.7$$

Where α is the payout rate (Typically 2,500 or 5000 USD), and the cap *C*, 500,000 or 1000,000 USD, and K_C is the strike. See Figure 1.2 (*see Appendix 1 for Python code*)

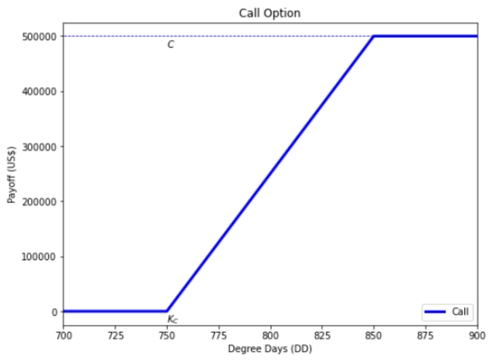


Figure 1.2. Call option with K = $750 and C = $500,000

The payoff function for a put option with a payoff function

$$\xi = min\{\alpha(DD - K)^{+}, C\} \qquad 1.8$$

is shown in Figure 1.3.

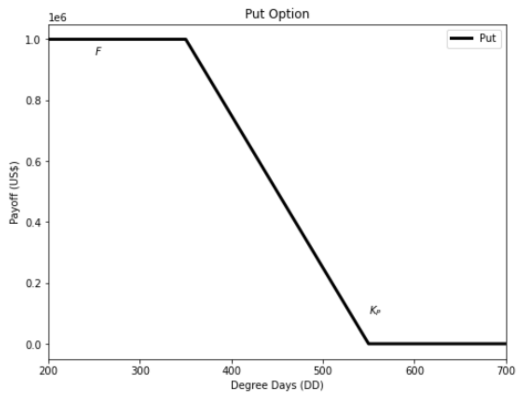


Figure 1.3. Put option with Strike K_P = \$550 and Floor *F = \$1000,000*

2. MEAN-REVERSION AND THE ORNSTEIN-UHLENBECK PROCESS

2.1 Mean Reversion: Asset prices will eventually return to their long-term mean or average. This concept is grounded in the belief that asset prices and historical returns will gravitate toward a long-term average over time.

How It's Used:

Statistical Analysis: Investors use statistical tools like Z-scores to measure how far an asset price has deviated from its mean.

Volatility: Some traders and investors use mean reversion in the context of volatility, buying options when volatility is high with the expectation that it will revert to the mean.

Risk Management: Stop-loss orders and take-profit points can be typically set around the mean to manage potential losses and secure gains.

Algorithmic Trading: Quantitative analysts use mean reversion in algorithmic trading strategies, often using complex mathematical models to predict price movements.

Limitations of Mean-eversion

- Market Conditions: Mean reversion is less effective in strongly trending markets, where prices may not revert to the mean for extended periods.
- Transaction Costs: The strategy often involves frequent trading, which has a propensity for higher transaction costs.
- False Signals: Shorter time frames in particular are susceptible to market noise, which can generate false mean-reverting signals.
- Economic Events: Economic shocks or sudden news can disrupt mean-reverting patterns, leading to potential losses.
- Lack of Direction: Unlike trend-following strategies, mean reversion is non-directional, which may not suit all trading styles.

2.2 Ornstein–Uhlenbeck Process (U-O)

The Ornstein–Uhlenbeck process is a stationary Gauss–Markov process, which means that it is a Gaussian process, a Markov similar to a discrete-time AR(1) process (Doob, 1942).

The Black-Scholes equation (Øksendal, 1998), commonly used to price financial options, assumes that the underlying asset's price follows a geometric Brownian motion. This means that the price of the underlying moves in a continuous and normally distributed fashion over time. However, weather derivatives are based on underlying assets like temperature, rainfall, or wind speed, which do not exhibit the same characteristics as financial assets.

Weather variables often have non-normal distributions and exhibit patterns like seasonality and autocorrelation, which makes them more complex than financial assets (Wigley, 2006). Additionally, weather data can have jumps or discontinuities, such as sudden temperature changes due to weather events like storms, which are not captured by the continuous and smooth movements assumed in the Black-Scholes model.

Therefore, applying the Black-Scholes equation directly to weather derivatives would not accurately reflect the unique characteristics of weather-related data, leading to inaccurate pricing and risk assessment. Instead, models specifically designed for weather derivatives take into account the statistical properties and dynamics of weather variables to provide more accurate pricing and risk management.

The cyclical nature of the temperature time series justifies the use of a mean-reverting process in modeling its dynamics which can be done as an Ito process with a mean reverting O-U process.

$$\boldsymbol{dT_t = \kappa(\overline{T}_t - T_t)dt + \sigma_t dW_t t} \qquad \textbf{2.1}$$

Dornier and Queruel (Dornier and Queruel, 2000). Benth and Šaltytė-Benth (Benth, et. al., 2008), Benth and Benth (Benth and Saltyte-Benth, 2005)], Alaton et al. (Alaton, et. al., 2002) observed that in general the speed of mean reversion should be a function of time. However, the authors provided no evidence as no studies had been undertaken to compute the daily mean reversion as the process is very complex.

To capture fully the mean-reverting dynamics of the temperature, it is important to have the Expectation of the temperature $\boldsymbol{T_t}$ to approximate the long-term reverting average temperature $\boldsymbol{\overline{T}_t}$. i.e.

$$\boldsymbol{Exp[T_t] \approx \overline{T}_t} \qquad \textbf{2.2}$$

Ito-Doeblin Formula

$$\boldsymbol{df(t,x) = f_t(t,x)dt + f_x(t,x)dx + \tfrac{1}{2}f_{xx}(t,x)dxdx} \qquad \textbf{2.3}$$

The function to be used is similar to the one used for the Cox-Ingersoll_Ross (CIR) (Bensoussan and Brouste, 2016) interest rate model, i.e.:

$$\boldsymbol{f(t,x) = e^{\kappa t}x, \qquad f_t(t,x) = \kappa e^{\kappa t}, f_x(t,x) = e^{\kappa t}, \qquad f_{xx}(t,x) = 0}$$

$$\boldsymbol{d(e^{\kappa t}T_t) = df_t(t,T_t) + f_t(t,T_t)dt + f_x(t,T_t)dT_t + \tfrac{1}{2}f_{xx}(t,x)dT_tdT_t} \qquad \textbf{2.4}$$

$$\boldsymbol{d(e^{\kappa t}T_t) = \kappa e^{\kappa t}T_tdt + e^{\kappa t}dT_t + \frac{1}{2}(0)dT_tdT_t}$$

$$\boldsymbol{d(e^{\kappa t}T_t) = \kappa e^{\kappa t}T_tdt + e^{\kappa t}\kappa(\overline{T}_t - T_t)dt + e^{\kappa t}\sigma_t dW_t}$$

$$\boldsymbol{d(e^{\kappa t}T_t) = e^{\kappa t}\kappa(\overline{T}_t)dt + e^{\kappa t}\sigma_t dW_t} \qquad \textbf{2.5}$$

Integrating over $\boldsymbol{u \in, t > s}$

$$\boldsymbol{\int_s^t d(e^{\kappa t}T_t) = e^{\kappa t}T_t - e^{\kappa s}T_s = \int_s^t \kappa e^{\kappa u}\overline{T}_u du + \int_s^t e^{\kappa u}\sigma_t dW_u} \qquad \textbf{2.6}$$

$$\boldsymbol{e^{\kappa t}T_t = e^{\kappa s}T_s + \int_s^t \kappa e^{\kappa u}\overline{T}_u du + \int_s^t e^{\kappa u}\sigma_t dW_u} \qquad \textbf{2.7}$$

Changing the base on the integral to $\boldsymbol{d\overline{T}_t}$ and dividing by $\boldsymbol{e^{\kappa t}}$

$$\boldsymbol{T_t = e^{-\kappa(t-s)}T_s + \int_s^t \kappa e^{-\kappa(t-u)}d\overline{T}_u + \int_s^t e^{-\kappa(t-u)}\sigma_t dW_u} \qquad \textbf{2.8}$$

Integrating the second term on the right,

$$\boldsymbol{T_t = T_s e^{-\kappa(t-s)} + T_t e^{-\kappa(t-t)} - T_s e^{-\kappa(t-s)} + \int_s^t e^{-\kappa(t-u)}\sigma_t dW_u} \qquad \textbf{2.9}$$

Grouping similar terms,

$$\boldsymbol{T_t = \overline{T}_t + (T_t - \overline{T}_s)e^{-\kappa(t-s)} + \int_s^t e^{-\kappa(t-u)}\sigma_t dW_u} \qquad \textbf{2.10}$$

Since

$$\boldsymbol{Exp}\left[\int_{s}^{t} \boldsymbol{e^{-\kappa(t-u)}\sigma_t dW_u}\right] = \mathbf{0}$$

$$\boldsymbol{Exp[T_t] = \overline{T}_t + (T_t - \overline{T}_s)e^{-\kappa(t-s)} \neq \overline{T}_t} \qquad \textbf{2.11}$$

This issue arises from the fact that $\boldsymbol{\kappa}$ is not constant but it changes over time.

This issue was corrected by adding a term $\frac{d\overline{T}_t}{dt}$ so the O-U model was modified to be written as

$$\boldsymbol{dT_t} = [\frac{d\overline{T}_t}{dt} + \boldsymbol{\kappa(\overline{T}_t - T_t)]dt + \sigma_t dW_t} \qquad \textbf{2.10}$$

The solution used to mitigate this issue was initially proposed by Dornier and Querel (Dornier and Queruel, 2000) and further developed by Dzupire, Ngare, and Odongo (Dzupire, Ngare and Odongo, 2019) who used a Lévy process-driven Ornstein-Uhlenbeck daily temperature model that took into account a time-dependent speed of mean reversion, $\boldsymbol{\kappa}$. The long average temperature $\boldsymbol{\overline{T}_t}$ was modeled as the sum of a seasonal term (sinusoidal) and a term that incorporates a small trend in the temperature data which can be due to global warming and urban heating effects (linear).

The SDE is solved by multiplying the equation by the factor $e^{\int_0^t \boldsymbol{\kappa} du}$ so we have

$$e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{d\overline{T}_u} - e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\kappa(\overline{T}_u - T_u)du} + e^{\int_0^t \boldsymbol{\kappa} du} dT_u = e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\sigma_t dW_u} \qquad \textbf{2.11}$$

$$d[e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\kappa(\overline{T}_u - T_u)]du} = e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\sigma_t dW_u} \qquad \textbf{2.12}$$

Let $Z_t = e^{\int_0^t \boldsymbol{\kappa} du}(\boldsymbol{\overline{T}_u - T_u})$, then $dZ_t = d\left[e^{\int_0^t \boldsymbol{\kappa} du}(\boldsymbol{\overline{T}_u - T_u})\right] = e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\sigma_t dW_u}$ with

$$Z_t = Z_o - \int_0^t e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\sigma_t dW_u} \qquad \textbf{2.13}$$

Substituting $Z_t = e^{\int_0^t \boldsymbol{\kappa} du}(\boldsymbol{\overline{T}_u - T_u})$ with $\boldsymbol{\overline{T}_0 - T_0}$ into **2.11**

$e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\kappa(\overline{T}_t - T_t)} = e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\kappa(\overline{T}_o - T_o)} - \int_0^t e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\sigma_t dW_u}$ rearranging terms

$$\boldsymbol{T_t = \overline{T}_t} + e^{-\int_0^t \boldsymbol{\kappa} du}\int_0^t e^{\int_0^t \boldsymbol{\kappa} du}\boldsymbol{\sigma_t dW_u} \qquad \textbf{2.12}$$

So $\boldsymbol{E[T_t] = \overline{T}_t}$ **2.13**

Temperature Model for Trend and Seasonality.

$$\boldsymbol{T_{trend} = a + bt}$$

$$\boldsymbol{T_{season} = \alpha sin(\omega t + \phi)}$$

$$\boldsymbol{\overline{T}_t = a + bt + \alpha sin(\omega t + \phi)} \qquad \text{with } \boldsymbol{\omega = \frac{2\pi}{365}}$$

3. MODIFIED ORNSTEIN-UHLENBECK PROCESS WITH MONTE CARLO SIMULATION FOR WEATHER DERIVATIVE PRICING.

Weather derivatives, financial instruments whose values are derived from weather variables such as temperature, precipitation, or wind speed, have gained significant traction in recent years as tools for managing weather-related risks. These derivatives allow businesses to hedge against adverse weather conditions that might impact their operations, such as farmers protecting against crop damage due to droughts or energy companies hedging against revenue losses from mild winters affecting energy demand.

One of the commonly used models for pricing weather derivatives is the modified Ornstein-Uhlenbeck process, a stochastic model that describes the mean-reverting behavior of a variable over time. This model, coupled with Monte Carlo simulation techniques, offers a robust framework for pricing weather derivatives accurately (Stein, Lopes, and Medino, 2021).

Understanding the Modified Ornstein-Uhlenbeck Process

The Ornstein-Uhlenbeck process is a stochastic process that models the evolution of a variable over time. It is characterized by two parameters: the long-term mean to which the variable reverts and a parameter controlling the speed of reversion. The modified Ornstein-Uhlenbeck process introduces additional parameters to account for seasonality and trends in the data, making it particularly suitable for modeling weather variables that exhibit such characteristics.

In the context of weather derivatives, the modified Ornstein-Uhlenbeck process can be used to model various weather variables such as temperature or precipitation. By calibrating the model to historical weather data, analysts can estimate the parameters governing the mean-reverting behavior of the variable, allowing them to simulate future scenarios.

4. MONTE CARLO SIMULATION FOR WEATHER DERIVATIVE PRICING

Monte Carlo simulation is a computational technique used to estimate the probability distribution of outcomes by repeatedly sampling from a probability distribution of input variables (Oetomo and Stevenson, 2005). In the context of weather derivative pricing, Monte Carlo simulation is employed to generate a large number of possible future paths for the weather variable of interest based on the modified Ornstein-Uhlenbeck process.

To price a weather derivative using Monte Carlo simulation and the modified Ornstein-Uhlenbeck process, the following steps are typically followed:

1. Calibration: Estimate the parameters of the modified Ornstein-Uhlenbeck process using historical weather data. This involves fitting the model to observed weather patterns to determine the long-term mean, speed of reversion, seasonality, and trend parameters.

2. Simulation: Generate a large number of simulated paths for the weather variable using the calibrated modified Ornstein-Uhlenbeck process. Each path represents a possible evolution of the weather variable over time.

3. Derivative Valuation: For each simulated path, calculate the payoff of the weather derivative based on its contract specifications. This could involve comparing the simulated weather variable against a predetermined strike level or using more complex payoff structures.

Monte Carlo Integration: Aggregate the payoffs from all simulated paths and calculate the average payoff. This average represents the expected value of the weather derivative under the modified Ornstein-Uhlenbeck process.

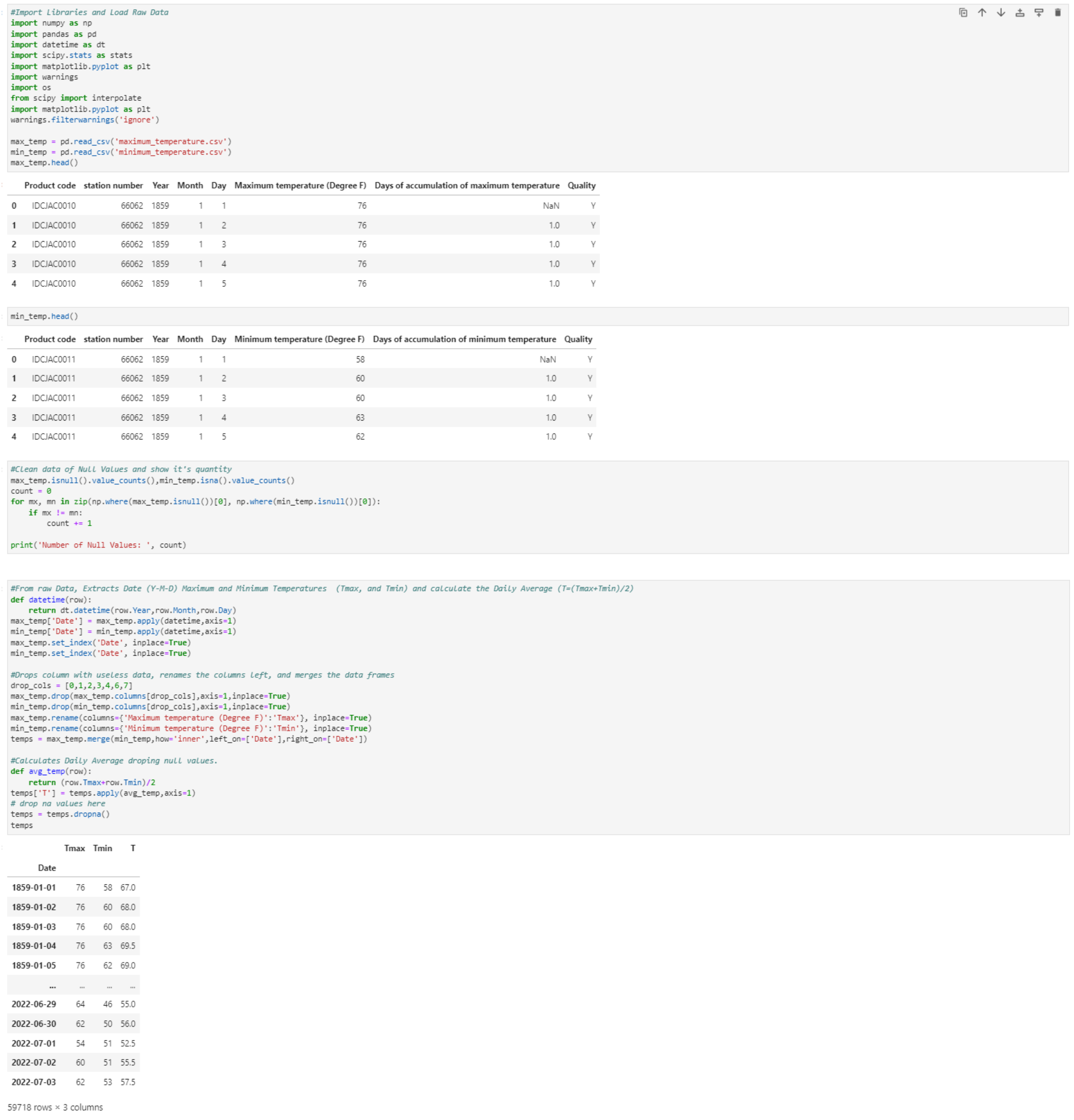
The entire programming process of fitting the Temperature Data and Volatility, applying the model, and simulating predicting the option price by Monte Carlo simulation are shown in the Python code below. The coding process consisted of two parts. An initial code was developed to create a time series based on historical temperature data to obtain fitting parameters. These parameters were used in the second code to estimate the price function of the temperature weather derivative using a Monte Carlo simulation. The results from the Monte Carlo simulation using the O-U model were graphed in conjunction with the results from the Black-Scholes model. In this paper, the comparison was presented for visual inspection only. No deep analysis was carried out.

5. MONTE CARLO SIMULATION FOR PRICING WEATHER DERIVATIVE OPTION WITH PYTHON.

5.1. Code 1. Data retrieval, cleaning, organization, and fitting.

Code Title (WDDtata Processing.ipynb)

5.1.1 Python’s library import, Data Frame uploading, and cleaning. Creation of Temperature time series with calculated Daily Time Average (DTA)



```python
#Import Libraries and Load Raw Data
import numpy as np
import pandas as pd
import datetime as dt
import scipy.stats as stats
import matplotlib.pyplot as plt
import warnings
import os
from scipy import interpolate
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')

max_temp = pd.read_csv('maximum_temperature.csv')
min_temp = pd.read_csv('minimum_temperature.csv')
max_temp.head()
```

	Product code	station number	Year	Month	Day	Maximum temperature (Degree F)	Days of accumulation of maximum temperature	Quality
0	IDCJAC0010	66062	1859	1	1	76	NaN	Y
1	IDCJAC0010	66062	1859	1	2	76	1.0	Y
2	IDCJAC0010	66062	1859	1	3	76	1.0	Y
3	IDCJAC0010	66062	1859	1	4	76	1.0	Y
4	IDCJAC0010	66062	1859	1	5	76	1.0	Y

```python
min_temp.head()
```

	Product code	station number	Year	Month	Day	Minimum temperature (Degree F)	Days of accumulation of minimum temperature	Quality
0	IDCJAC0011	66062	1859	1	1	58	NaN	Y
1	IDCJAC0011	66062	1859	1	2	60	1.0	Y
2	IDCJAC0011	66062	1859	1	3	60	1.0	Y
3	IDCJAC0011	66062	1859	1	4	63	1.0	Y
4	IDCJAC0011	66062	1859	1	5	62	1.0	Y

```python
#Clean data of Null Values and show it's quantity
max_temp.isnull().value_counts(),min_temp.isna().value_counts()
count = 0
for mx, mn in zip(np.where(max_temp.isnull())[0], np.where(min_temp.isnull())[0]):
    if mx != mn:
        count += 1

print('Number of Null Values: ', count)
```

```python
#From raw Data, Extracts Date (Y-M-D) Maximum and Minimum Temperatures  (Tmax, and Tmin) and calculate the Daily Average (T=(Tmax+Tmin)/2)
def datetime(row):
    return dt.datetime(row.Year,row.Month,row.Day)
max_temp['Date'] = max_temp.apply(datetime,axis=1)
min_temp['Date'] = min_temp.apply(datetime,axis=1)
max_temp.set_index('Date', inplace=True)
min_temp.set_index('Date', inplace=True)

#Drops column with useless data, renames the columns left, and merges the data frames
drop_cols = [0,1,2,3,4,6,7]
max_temp.drop(max_temp.columns[drop_cols],axis=1,inplace=True)
min_temp.drop(min_temp.columns[drop_cols],axis=1,inplace=True)
max_temp.rename(columns={'Maximum temperature (Degree F)':'Tmax'}, inplace=True)
min_temp.rename(columns={'Minimum temperature (Degree F)':'Tmin'}, inplace=True)
temps = max_temp.merge(min_temp,how='inner',left_on=['Date'],right_on=['Date'])

#Calculates Daily Average droping null values.
def avg_temp(row):
    return (row.Tmax+row.Tmin)/2
temps['T'] = temps.apply(avg_temp,axis=1)
# drop na values here
temps = temps.dropna()
temps
```

	Tmax	Tmin	T
Date			
1859-01-01	76	58	67.0
1859-01-02	76	60	68.0
1859-01-03	76	60	68.0
1859-01-04	76	63	69.5
1859-01-05	76	62	69.0
...	...	...	...
2022-06-29	64	46	55.0
2022-06-30	62	50	56.0
2022-07-01	54	51	52.5
2022-07-02	60	51	55.5
2022-07-03	62	53	57.5

59718 rows × 3 columns

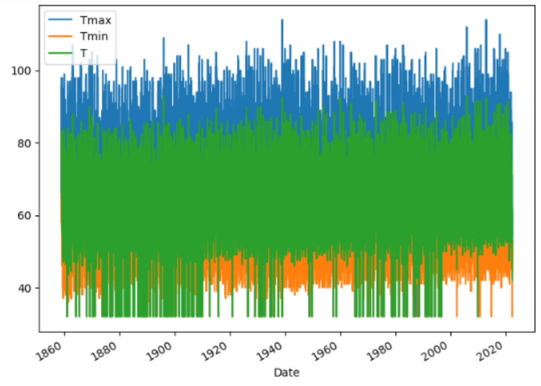
5.1.2. The Data Frame was divided into winter and summer seasons and a histogram was plotted.

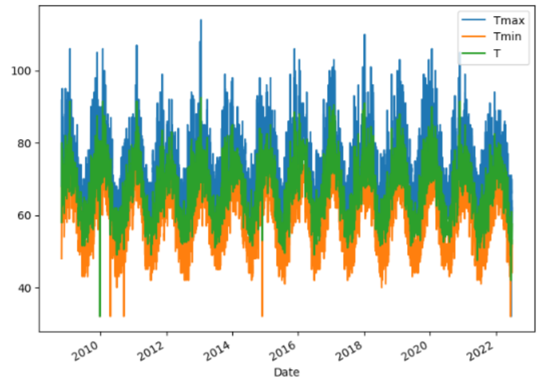
```
#Divides temperature in seasons wih Summer between May and September and Winter between October and April
temps_season = temps.copy(deep=True)
temps_season['month'] = temps_season.index.month
mask = (temps_season['month'] >= 5) & (temps_season['month'] <= 10)
temps_season['winter'] = np.where(mask,1,0)
temps_season['summer'] = np.where(temps_season['winter'] != 1,1,0)
temps_season
```

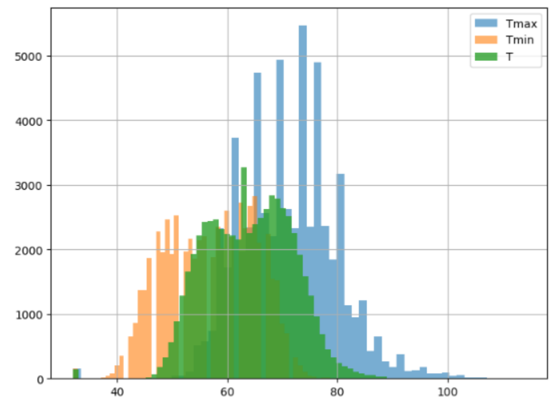
	Tmax	Tmin	T	month	winter	summer
Date						
1859-01-01	76	58	67.0	1	0	1
1859-01-02	76	60	68.0	1	0	1
1859-01-03	76	60	68.0	1	0	1
1859-01-04	76	63	69.5	1	0	1
1859-01-05	76	62	69.0	1	0	1
...	...	...	...	...	...	...
2022-06-29	64	46	55.0	6	1	0
2022-06-30	62	50	56.0	6	1	0
2022-07-01	54	51	52.5	7	1	0
2022-07-02	60	51	55.5	7	1	0
2022-07-03	62	53	57.5	7	1	0

59718 rows × 6 columns

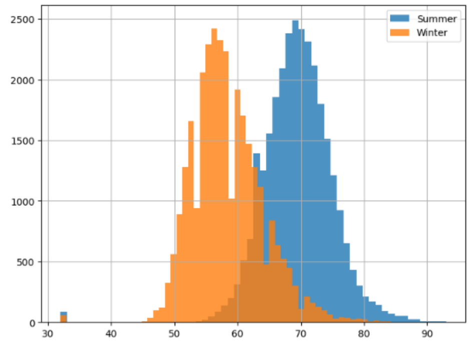
```
#Creates timeseries with Tmax, Tmin and, Daily Average, and plots: All data, Last 14 years, and Histogram.
temps[:].plot(figsize=(8,6))
plt.show()

#Timeseries plot - ~14 years
temps[-5000:].plot(figsize=(8,6))
plt.show()

#Temperature distributions in histogram
plt.figure(figsize=(8,6))
temps.Tmax.hist(bins=60, alpha=0.6, label='Tmax')
temps.Tmin.hist(bins=60, alpha=0.6, label='Tmin')
temps['T'].hist(bins=60, alpha=0.8, label='T')
plt.legend()
plt.show()
```







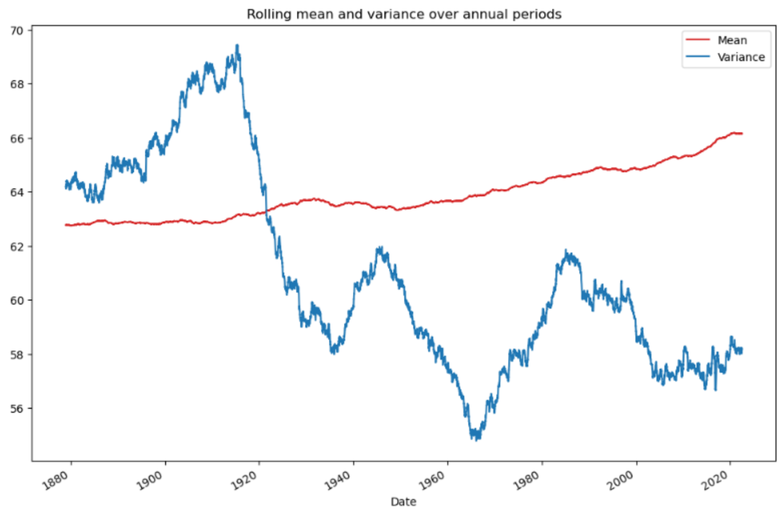
```
#Plots histograms of summer and winter temperatures.
plt.figure(figsize=(8,6))
temps_season[temps_season['summer'] == 1]['T'].hist(bins=60, alpha=0.8, label='Summer')
temps_season[temps_season['winter'] == 1]['T'].hist(bins=60, alpha=0.8, label='Winter')
plt.legend()
plt.show()
```



5.1.3. Visualization of Trend and Seasonability.

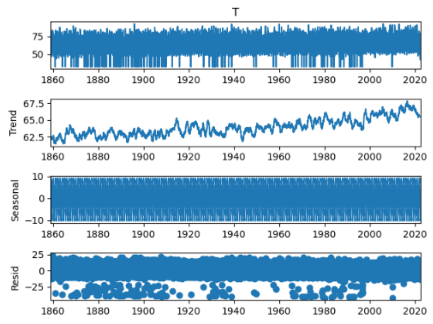
```
#Imports Q-Q plot, ADFuller, Seasonal Decompose, ACF, and P-ACF tools.
from statsmodels.graphics.api import qqplot
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.ar_model import AutoReg, ar_select_order, AutoRegResults
```

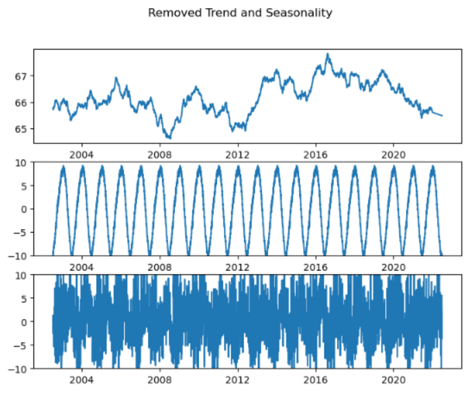
```
#Plots Rolling mean over Annual Period (Red) and Rolling Variance over Annual period (Blue).
temps.sort_index(inplace=True)
Years=20
temps['T'].rolling(window = 365*Years).mean().plot(figsize=(12,8), color="tab:red", title="Rolling mean over annual periods", label = 'Mean')
temps['T'].rolling(window = 365*Years).var().plot(figsize=(12,8), color="tab:blue", title="Rolling mean and variance over annual periods", label = 'Variance');
plt.legend()
plt.show()
```



5.1.4. Seasonal decomposition and visualization of trend, seasonality, and residuals.

```
#Apply decompose Seasonal to time series and plot, Data, Trend, Seasonal, and Residual
decompose_result = seasonal_decompose(temps['T'], model='additive', period=int(365), extrapolate_trend='freq')
trend = decompose_result.trend
seasonal = decompose_result.seasonal
residual = decompose_result.resid

#Visualise All Data, Trend, Seasonal, and Residuals
decompose_result.plot()
plt.show()

#Visualise x-Years
Years = 20
years_examine = 365*Years
fig, axs = plt.subplots(3, figsize=(8,6))
fig.suptitle('Removed Trend and Seasonality')
axs[0].plot(trend[-years_examine:])
axs[1].plot(seasonal[-years_examine:])
axs[1].set_ylim([-10,10])
axs[2].plot(residual[-years_examine:])
axs[2].set_ylim([-10,10])
```

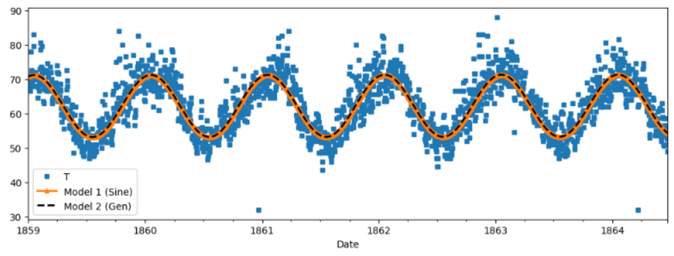


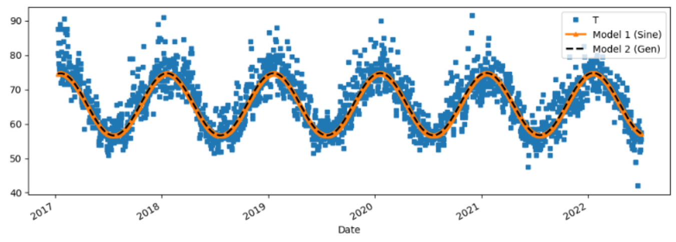


5.1.5. Fitting models to daily average temperature time series were applied, and model parameters were obtained and compared.

```python
#Apply fitting models and calculate parameters with Residual Sum of Squares (RSS)
#Model 1 Tt¯=a+bt+asin(ωt+ϑ) here y_pred = a+bt+a1sin((omega)x+b1)
#Model 2 Tt¯=a+bt+asin(ωt+ϑ)+βcos(ωt+ϕ) here y_pred = a+bt+a1sin((omegha)x+theta)+b1cos((omega)x+phi)
from scipy.stats import norm
from scipy.optimize import curve_fit
temp_t = temps['T'].copy(deep=True)
temp_t = temp_t.to_frame()

#Define Models
#Model 1 (Sine)
def model_fit(x, a, b, a1, b1):
    omega = 2*np.pi/365.25
    y_pred = a + b*x + a1*np.cos(omega*x) + b1*np.sin(omega*x)
    return y_pred
def RSS(y, y_pred):
    return np.sqrt( (y - y_pred)**2 ).sum()

#Model 2 (Generalized)
def model_fit_general(x, a, b, a1, b1, theta, phi):
    omega = 2*np.pi/365.25
    y_pred = a + b*x + a1*np.cos(omega*x + theta) + b1*np.sin(omega*x + phi)
    return y_pred

if isinstance(temp_t.index , pd.DatetimeIndex):
    first_ord = temp_t.index.map(dt.datetime.toordinal)[0]
    temp_t.index=temp_t.index.map(dt.datetime.toordinal)

params, cov = curve_fit(model_fit, xdata = temp_t.index-first_ord, ydata = temp_t['T'], method='lm')
param_list = ['a', 'b', 'a1', 'b1']
print('\n        Model 1 Parameters \n')
std_dev = np.sqrt(np.diag(cov))
for name, p, sd in zip( param_list, params, std_dev):
    print('{0} :  {1:0.3}  CI ~normally [{2:0.2e},{3:0.2e}]'.format(name, p, p-1.96*sd,p+1.96*sd))

temp_t['Model 1 (Sine)'] = model_fit(temp_t.index-first_ord, *params)
if isinstance(temp_t.index , pd.DatetimeIndex):
    temp_t.index=temp_t.index.map(dt.datetime.toordinal)
params1, cov1 = curve_fit(model_fit_general, xdata = temp_t.index-first_ord, ydata = temp_t['T'], method='lm')
param_list = ['a', 'b', 'a1', 'b1', 'theta', 'phi']
print('\n        Model 2 Parameters \n')
std_dev = np.sqrt(np.diag(cov1))
for name, p, sd in zip( param_list, params1, std_dev):
    print('{0} :  {1:0.3}  CI ~normally [{2:0.2e},{3:0.2e}]'.format(name, p, p-1.96*sd,p+1.96*sd))
temp_t['Model 2 (Gen)'] = model_fit_general(temp_t.index-first_ord, *params1)
if not isinstance(temp_t.index , pd.DatetimeIndex):
    temp_t.index=temp_t.index.map(dt.datetime.fromordinal)
temp_t[:2000].plot(figsize=(12,4), style=['s','^-','k--'] , markersize=4, linewidth=2 )
temp_t[-2000:].plot(figsize=(12,4), style=['s','^-','k--'] , markersize=4, linewidth=2 )
RSS(temp_t['T'], temp_t['Model 2 (Gen)'])

print('\n     Residual Sum of Squares (RSS)  \n')
print('  RSS Model 1 (Sine):', round(RSS(temp_t['T'], temp_t['Model 1 (Sine)']),2))
print('  RSS Model 2 (Gen):', round(RSS(temp_t['T'], temp_t['Model 2 (Gen)']),2))
```

```
        Model 1 Parameters 

a :  62.1  CI ~normally [6.21e+01,6.22e+01]
b :  6.08e-05  CI ~normally [5.87e-05,6.29e-05]
a1 :  8.6  CI ~normally [8.55e+00,8.66e+00]
b1 :  2.73  CI ~normally [2.68e+00,2.78e+00]

        Model 2 Parameters 

a :  62.1  CI ~normally [6.21e+01,6.22e+01]
b :  6.08e-05  CI ~normally [5.87e-05,6.29e-05]
a1 :  -27.6  CI ~normally [nan,nan]
b1 :  23.6  CI ~normally [nan,nan]
theta :  -2.16e+02  CI ~normally [nan,nan]
phi :  -1.26e+02  CI ~normally [nan,nan]

     Residual Sum of Squares (RSS)  

  RSS Model 1 (Sine): 196268.41
  RSS Model 2 (Gen): 196268.43
```

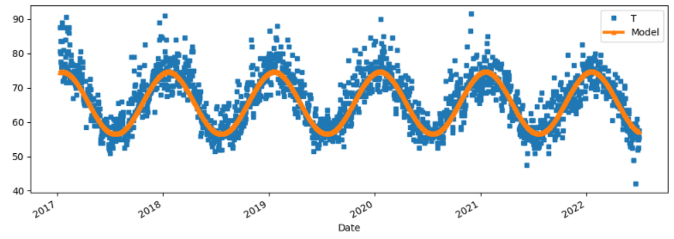




5.1.6. Fitting Parameters were calculated and applied to fitted data.

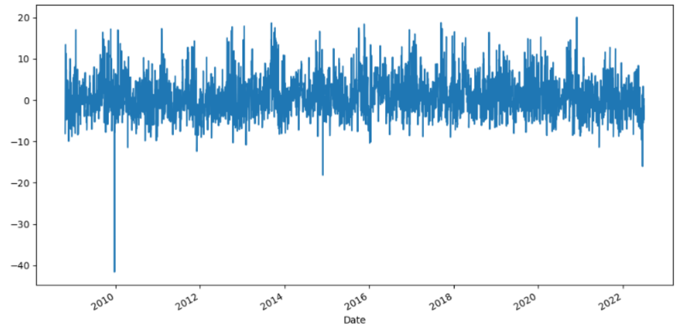
```python
#Calculate parameters with data and fit curve
temp_t = temps['T'].copy(deep=True)
temp_t = temp_t.to_frame()
def model(x, params):
    a,b,a1,b1 = params
    omega = 2*np.pi/365.25
    theta = np.arctan(a1/b1)
    alpha = np.sqrt( a1**2 + b1**2)
    print('Parameters:\n     a {0:0.3}\n     b {1:0.3}\n alpha {2:0.3}\n theta {3:0.3}'.format(a,b,alpha,theta))
    y_pred = a + b*x + alpha*np.sin(omega*x + theta)
    return y_pred
def model_fit(x, a, b, a1, b1):
    omega = 2*np.pi/365.25
    y_pred = a + b*x + a1*np.cos(omega*x) + b1*np.sin(omega*x)
    return y_pred
if isinstance(temp_t.index , pd.DatetimeIndex):
    first_ord = temp_t.index.map(dt.datetime.toordinal)[0]
    temp_t.index=temp_t.index.map(dt.datetime.toordinal)

params_all, cov = curve_fit(model_fit, xdata = temp_t.index-first_ord, ydata = temp_t['T'], method='lm')

temp_t['Model'] = model(temp_t.index-first_ord, params_all)
if not isinstance(temp_t.index , pd.DatetimeIndex):
    temp_t.index=temp_t.index.map(dt.datetime.fromordinal)

temp_t[-2000:].plot(figsize=(12,4), style=['s','^-','k-'] , markersize=4, linewidth=3 )
plt.show()
```

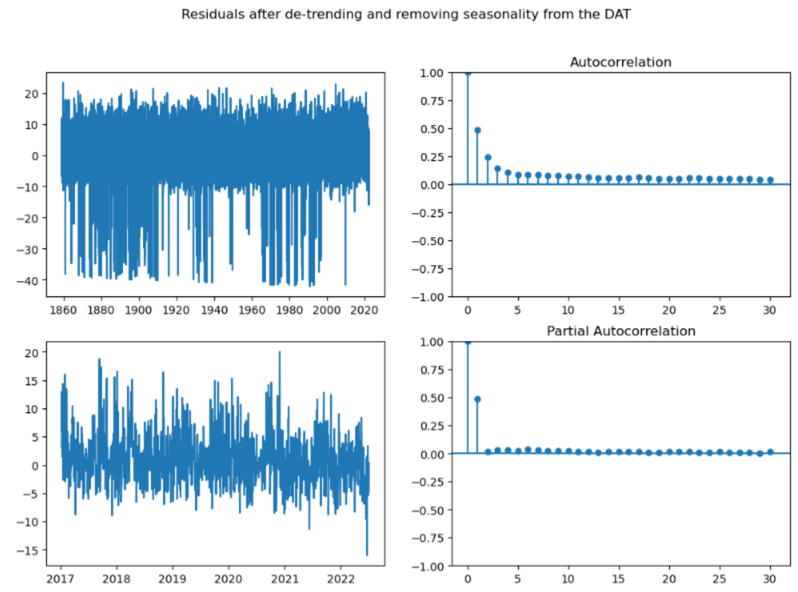
```
Parameters:
     a 62.1
     b 6.08e-05
 alpha 9.03
 theta 1.26
```



5.1.7. Autocorrelation was applied and residuals were analyzed.

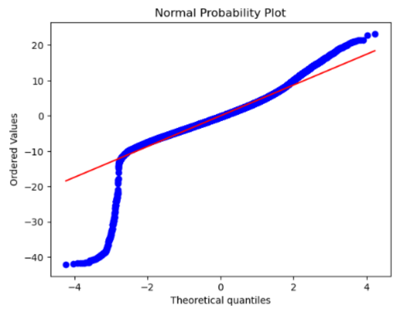
```python
#Visualize Residuals after detrending and removing seasonality from the Daily Average Temperature (DAT) 12 years.
if not isinstance(temp_t.index , pd.DatetimeIndex):
    temp_t.index=temp_t.index.map(dt.datetime.fromordinal)
temp_t['res'] = temp_t['T']-temp_t['Model']
temp_t['res'][-5000:].plot(figsize=(12,6))
plt.show()
#Visualizes autotcorrelation for all the time series (TLags) and the the last few years (PLags).
TLags = 30
PLags = 30
fig, axs = plt.subplots(2,2, figsize=(12,8))
fig.suptitle('Residuals after de-trending and removing seasonality from the DAT')
axs[0,0].plot(temp_t['res'])
axs[1,0].plot(temp_t['res'][-2000:])
plot_acf(temp_t['res'], lags = TLags, ax=axs[0,1])
plot_pacf(temp_t['res'], lags = PLags, ax=axs[1,1])
plt.show()
```



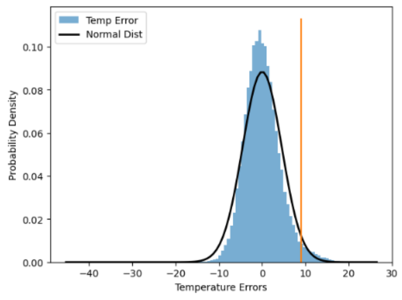


5.1.8. Probability distribution was calculated and plotted.

```
#Plots Probability Distribution
stats.probplot(temp_t['res'], dist="norm", plot=plt)
plt.title("Normal Probability Plot")
plt.show()

#Plot probability distribution of temperature errors and compares it to the normal distribution.
mu, std = norm.fit(temp_t['res'])
z = (temp_t['res'] - mu)/std
plt.hist(temp_t['res'], density=True, alpha=0.6, bins=100, label='Temp Error')
xmin, xmax = plt.xlim()
ymin, ymax = plt.ylim()
x = np.linspace(xmin, xmax, 100)
p = norm.pdf(x, mu, std)
data = np.random.randn(100000)
plt.plot(x, p, 'k', linewidth=2, label='Normal Dist')
plt.plot([std*2,std*2],[0,ymax])
print('P(Z > 2): {:0.3}% vs. Normal Distibution: {:0.3}% '.format(len(z[z >= 2])/len(z)*100, (1-norm.cdf(2))*100))
print('Skew    : {:0.3}'.format(stats.skew(z)))
print('Kurtosis: {:0.3}'.format(stats.kurtosis(z)+3))
plt.ylabel('Probability Density')
plt.xlabel('Temperature Errors')
plt.legend()
plt.show()
```

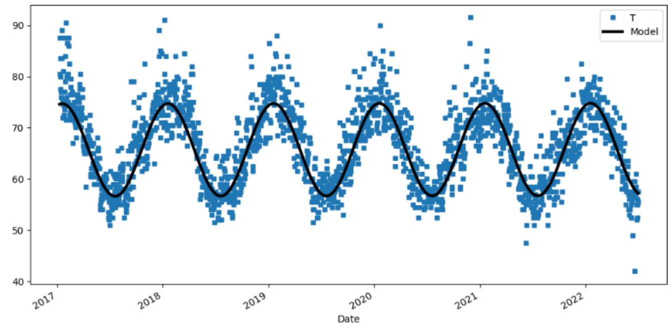


```
P(Z > 2): 3.09% vs. Normal Distibution: 2.28%
Skew    : -0.567
Kurtosis: 12.2
```



5.1.9. Visualization of Fitted Data.

```python
#Plots Partial Data and Fit Model
rows=2000
temp_t[['T','Model']][-rows:].plot(figsize=(12,6), style=['s','k-'] , markersize=4, linewidth=3 )
plt.show()
```



5.1.10. Autoregression was applied to temperature residuals.

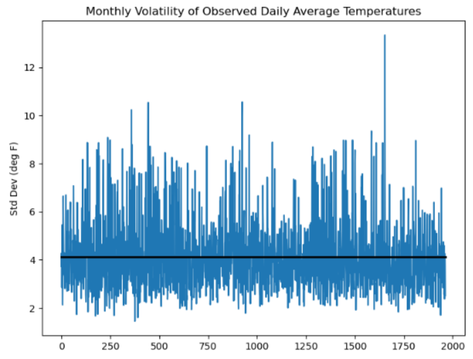
```python
# Performs an Autogregression of the Residuals and print parameters
residuals = temp_t['res']
residuals.index = pd.DatetimeIndex(residuals.index).to_period('D')
model = AutoReg(residuals, lags=1, old_names=True,trend='n')
model_fit  = model.fit()
coef = model_fit.params
res = model_fit.resid
res.index = res.index.to_timestamp()
print(model_fit.summary())
```

```
                            AutoReg Model Results
==============================================================================
Dep. Variable:                    res   No. Observations:                59718
Model:                     AutoReg(1)   Log Likelihood             -166627.381
Method:               Conditional MLE   S.D. of innovations              3.941
Date:                Tue, 30 Apr 2024   AIC                         333258.762
Time:                        10:33:44   BIC                         333276.757
Sample:                    01-02-1859   HQIC                        333264.353
                         - 07-03-2022
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
res.L1         0.4880      0.004    136.609      0.000       0.481       0.495
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            2.0494           +0.0000j            2.0494            0.0000
-----------------------------------------------------------------------------
```

5.1.11. The monthly volatility of the DAT was estimated and plotted.

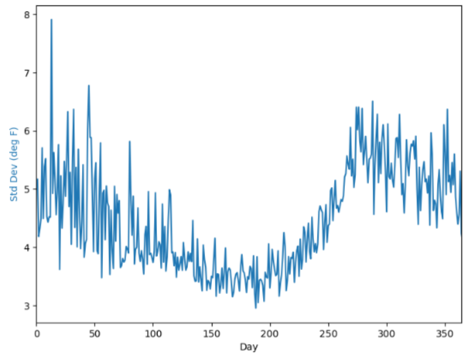
```python
##Estimate volatility based on the quadratic variation of temperature process. Plot Monthly Volatility of DAT
temp_t['Day'] = temp_t.index.dayofyear
temp_t['month'] = temp_t.index.month
temp_t['year'] = temp_t.index.year
vol = temp_t.groupby(['year','month'])['T'].agg(['mean','std'])
vol = vol.reset_index()
vol['std'].plot(figsize=(8,6))
plt.plot([0, len(vol)], [vol['std'].mean(),vol['std'].mean()], 'k', linewidth=2)
plt.ylabel('Std Dev (deg F)')
plt.title('Monthly Volatility of Observed Daily Average Temperatures', color='k',)
print('Trend or long term volatility is easy: ~', round(vol['std'].mean(),3))
plt.show()
```

```
Trend or long term volatility is easy: ~ 4.121
```



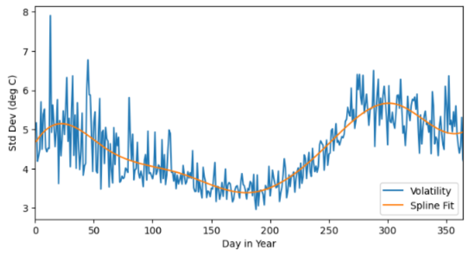
5.1.12. The daily volatility of the DAT was estimated and plotted.

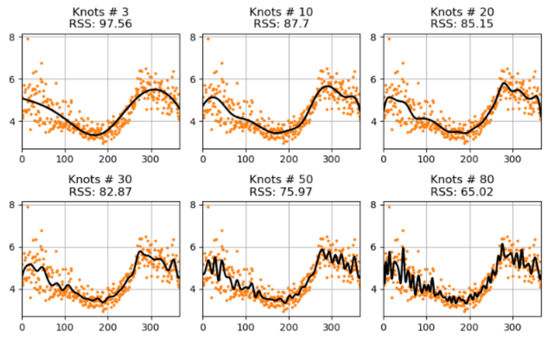
```
#Estimate Daily Volatility of DAT, and Plot
vol = temp_t.groupby(['Day'])['T'].agg(['mean','std'])
vol['std'].plot(color='tab:blue', figsize=(8,6))
plt.ylabel('Std Dev (deg F)',color='tab:blue')
plt.xlim(0,364)
plt.show()
```



5.1.13. A 5-knot (optimal) B-splined was applied to the daily volatility.

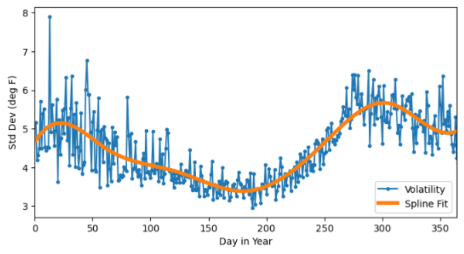
```
#Plot Spline Fit of volatility

from scipy import interpolate
x = np.array(vol['std'].index)
y = np.array(vol['std'].values)
knot_numbers = 5
x_new = np.linspace(0, 1, knot_numbers+2)[1:-1]
q_knots = np.quantile(x, x_new)
t,c,k = interpolate.splrep(x, y, t=q_knots, s=1)
yfit = interpolate.BSpline(t,c,k)(x)
plt.figure(figsize=(8,4))
plt.plot(x, y, label='Volatility')
plt.plot(x, yfit, label='Spline Fit')
plt.ylabel('Std Dev (deg C)')
plt.xlabel('Day in Year')
plt.xlim(0,364)
plt.legend(loc='lower right')
plt.show()
def spline(knots, x, y):
    x_new = np.linspace(0, 1, knots+2)[1:-1]
    t, c, k = interpolate.splrep(x, y, t=np.quantile(x, x_new), s=3)
    yfit = interpolate.BSpline(t,c, k)(x)
    return yfit
knots = [3, 10, 20, 30, 50, 80]
i = 0
fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(8, 5))
for row in range(2):
    for col in range(3):
        ax[row][col].plot(x, y, '.',c='tab:orange', markersize=4)
        yfit = spline(knots[i], x, y)
        rss = np.sum( np.square(y-yfit) )
        ax[row][col].plot(x, yfit, 'k', linewidth=2)
        ax[row][col].set_title("Knots # "+str(knots[i])+"\nRSS: "+str(round(rss,2)), color='k')
        ax[row][col].set_xlim(0,366)
        ax[row][col].grid()
        i=i+1

plt.tight_layout()
plt.show()
```





5.1.14. Volatility function was obtained from the quadratic variation of the Temperature process.

```python
#Estimate Volatility from the quadratic variation of the Temperature Process
temp_vol = temps['T'].copy(deep=True)
temp_vol = temp_vol.to_frame()
temp_vol['Day'] = temp_vol.index.dayofyear
temp_vol['month'] = temp_vol.index.month
vol = temp_vol.groupby(['Day'])['T'].agg(['mean','std'])
days = np.array(vol['std'].index)
T_std = np.array(vol['std'].values)
#Fits Std Dev to 5-knots spline
def spline(knots, x, y):
    x_new = np.linspace(0, 1, knots+2)[1:-1]
    t, c, k = interpolate.splrep(x, y, t=np.quantile(x, x_new), s=3)
    yfit = interpolate.BSpline(t,c, k)(x)
    return yfit
volatility = spline(5, days, T_std)
plt.figure(figsize=(8,4))
plt.plot(days, T_std, marker='.',label='Volatility')
plt.plot(days, volatility, linewidth=4, label='Spline Fit')
plt.ylabel('Std Dev (deg F)')
plt.xlabel('Day in Year')
plt.xlim(0,364)
plt.legend(loc='lower right')
plt.show()
```

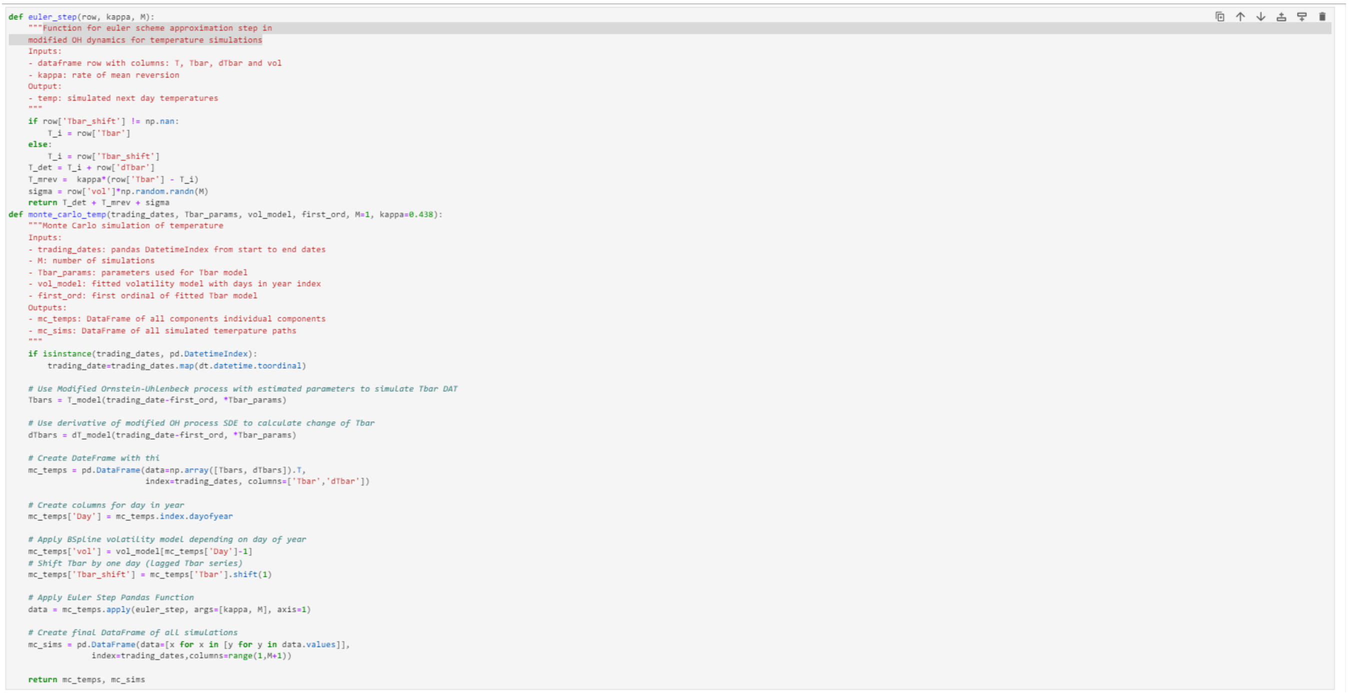


5.1.15. The volatility (κ) of the volatility process was estimated using the quadratic variation of σ.

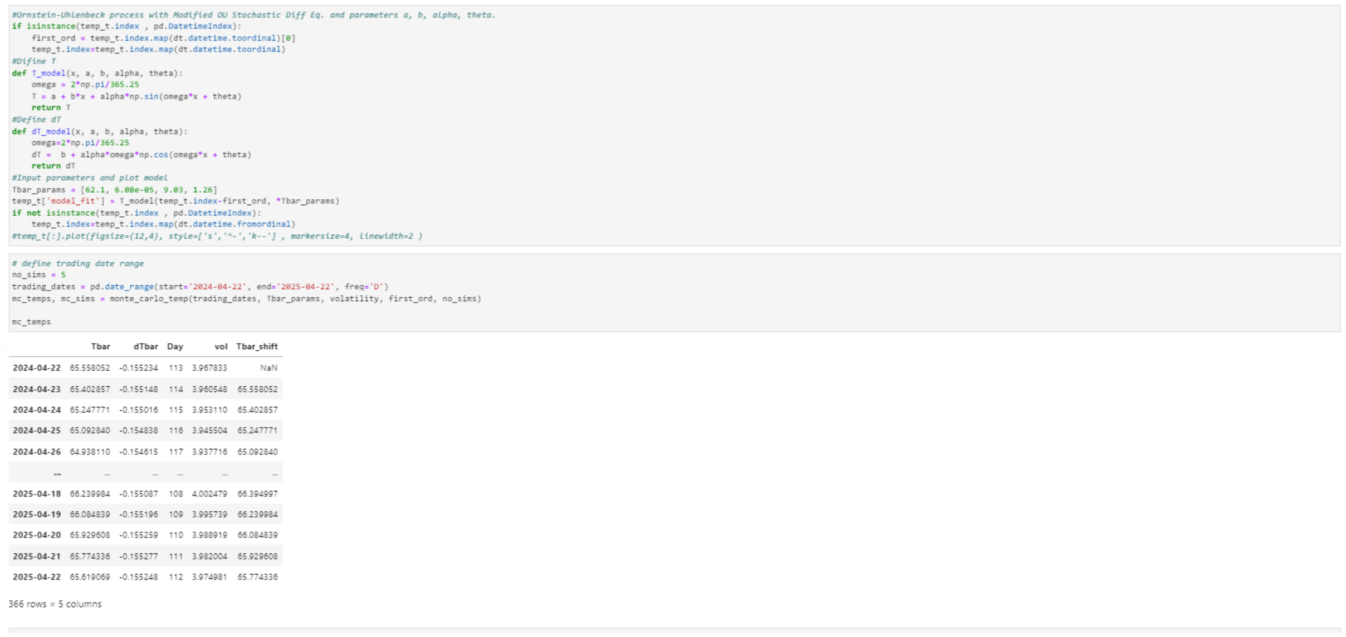
```python
# Estimate volatility of the volatility process by using the quadratic variation of sigma.
print('Gamma is: ', round(vol['std'].std(),3))
model = AutoReg(vol['std'], lags=1, old_names=True,trend='n')
model_fit  = model.fit()
coef = model_fit.params
res = model_fit.resid
print('Rate of mean reversion of volatility process is : ', coef[0])
print(model_fit.summary())
```

```
Gamma is:  0.876
Rate of mean reversion of volatility process is :  0.9885299831341818
                            AutoReg Model Results
==============================================================================
Dep. Variable:                    std   No. Observations:                  366
Model:                     AutoReg(1)   Log Likelihood                -378.671
Method:               Conditional MLE   S.D. of innovations              0.683
Date:                Sun, 28 Apr 2024   AIC                            761.343
Time:                        22:28:52   BIC                            769.143
Sample:                             1   HQIC                           764.443
                                  366
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
std.L1         0.9885      0.008    126.244      0.000       0.973       1.004
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            1.0116           +0.0000j            1.0116            0.0000
-----------------------------------------------------------------------------
```

5.1.15. The data frame was approximated by applying an Euler-Maruyama scheme. O-U Parameters were obtained by using a Monte Carlo approximation.



```python
def euler_step(row, kappa, M):
    """Function for euler scheme approximation step in
    modified OH dynamics for temperature simulations
    Inputs:
    - dataframe row with columns: T, Tbar, dTbar and vol
    - kappa: rate of mean reversion
    Output:
    - temp: simulated next day temperatures
    """
    if row['Tbar_shift'] != np.nan:
        T_i = row['Tbar']
    else:
        T_i = row['Tbar_shift']
    T_det = T_i + row['dTbar']
    T_mrev =  kappa*(row['Tbar'] - T_i)
    sigma = row['vol']*np.random.randn(M)
    return T_det + T_mrev + sigma
def monte_carlo_temp(trading_dates, Tbar_params, vol_model, first_ord, M=1, kappa=0.438):
    """Monte Carlo simulation of temperature
    Inputs:
    - trading_dates: pandas DatetimeIndex from start to end dates
    - M: number of simulations
    - Tbar_params: parameters used for Tbar model
    - vol_model: fitted volatility model with days in year index
    - first_ord: first ordinal of fitted Tbar model
    Outputs:
    - mc_temps: DataFrame of all components individual components
    - mc_sims: DataFrame of all simulated temerpature paths
    """
    if isinstance(trading_dates, pd.DatetimeIndex):
        trading_date=trading_dates.map(dt.datetime.toordinal)

    # Use Modified Ornstein-Uhlenbeck process with estimated parameters to simulate Tbar DAT
    Tbars = T_model(trading_date-first_ord, *Tbar_params)

    # Use derivative of modified OH process SDE to calculate change of Tbar
    dTbars = dT_model(trading_date-first_ord, *Tbar_params)

    # Create DateFrame with thi
    mc_temps = pd.DataFrame(data=np.array([Tbars, dTbars]).T,
                            index=trading_dates, columns=['Tbar','dTbar'])

    # Create columns for day in year
    mc_temps['Day'] = mc_temps.index.dayofyear

    # Apply BSpline volatility model depending on day of year
    mc_temps['vol'] = vol_model[mc_temps['Day']-1]
    # Shift Tbar by one day (lagged Tbar series)
    mc_temps['Tbar_shift'] = mc_temps['Tbar'].shift(1)

    # Apply Euler Step Pandas Function
    data = mc_temps.apply(euler_step, args=[kappa, M], axis=1)

    # Create final DataFrame of all simulations
    mc_sims = pd.DataFrame(data=[x for x in [y for y in data.values]],
                 index=trading_dates,columns=range(1,M+1))

    return mc_temps, mc_sims
```



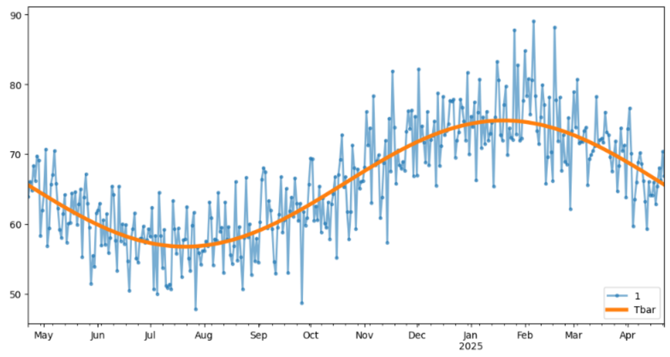
```python
#Ornstein-Uhlenbeck process with Modified OU Stochastic Diff Eq. and parameters a, b, alpha, theta.
if isinstance(temp_t.index , pd.DatetimeIndex):
    first_ord = temp_t.index.map(dt.datetime.toordinal)[0]
    temp_t.index=temp_t.index.map(dt.datetime.toordinal)
#Difine T
def T_model(x, a, b, alpha, theta):
    omega = 2*np.pi/365.25
    T = a + b*x + alpha*np.sin(omega*x + theta)
    return T
#Define dT
def dT_model(x, a, b, alpha, theta):
    omega=2*np.pi/365.25
    dT =  b + alpha*omega*np.cos(omega*x + theta)
    return dT
#Input parameters and plot model
Tbar_params = [62.1, 6.08e-05, 9.03, 1.26]
temp_t['model_fit'] = T_model(temp_t.index-first_ord, *Tbar_params)
if not isinstance(temp_t.index , pd.DatetimeIndex):
    temp_t.index=temp_t.index.map(dt.datetime.fromordinal)
#temp_t[:].plot(figsize=(12,4), style=['s','^-','k--'] , markersize=4, linewidth=2 )
```

```python
# define trading date range
no_sims = 5
trading_dates = pd.date_range(start='2024-04-22', end='2025-04-22', freq='D')
mc_temps, mc_sims = monte_carlo_temp(trading_dates, Tbar_params, volatility, first_ord, no_sims)

mc_temps
```

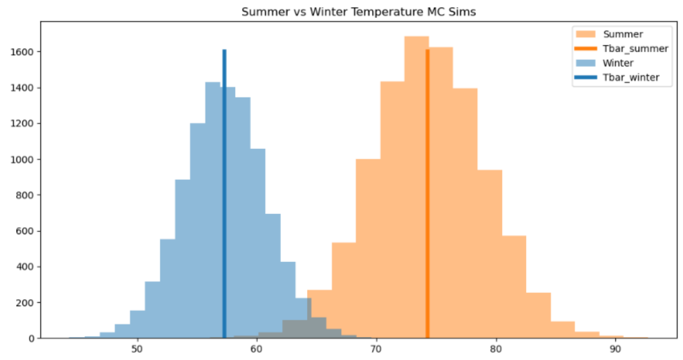
	Tbar	dTbar	Day	vol	Tbar_shift
2024-04-22	65.558052	-0.155234	113	3.967833	NaN
2024-04-23	65.402857	-0.155148	114	3.960548	65.558052
2024-04-24	65.247771	-0.155016	115	3.953110	65.402857
2024-04-25	65.092840	-0.154838	116	3.945504	65.247771
2024-04-26	64.938110	-0.154615	117	3.937716	65.092840
...	...	...	...	...	...
2025-04-18	66.239984	-0.155087	108	4.002479	66.394997
2025-04-19	66.084839	-0.155196	109	3.995739	66.239984
2025-04-20	65.929608	-0.155259	110	3.988919	66.084839
2025-04-21	65.774336	-0.155277	111	3.982004	65.929608
2025-04-22	65.619069	-0.155248	112	3.974981	65.774336

366 rows × 5 columns

```python
plt.figure(figsize=(12,6))
mc_sims[1].plot(alpha=0.6,linewidth=2, marker='.')
mc_temps["Tbar"].plot(linewidth=4)
plt.legend(loc='lower right')
plt.show()
```

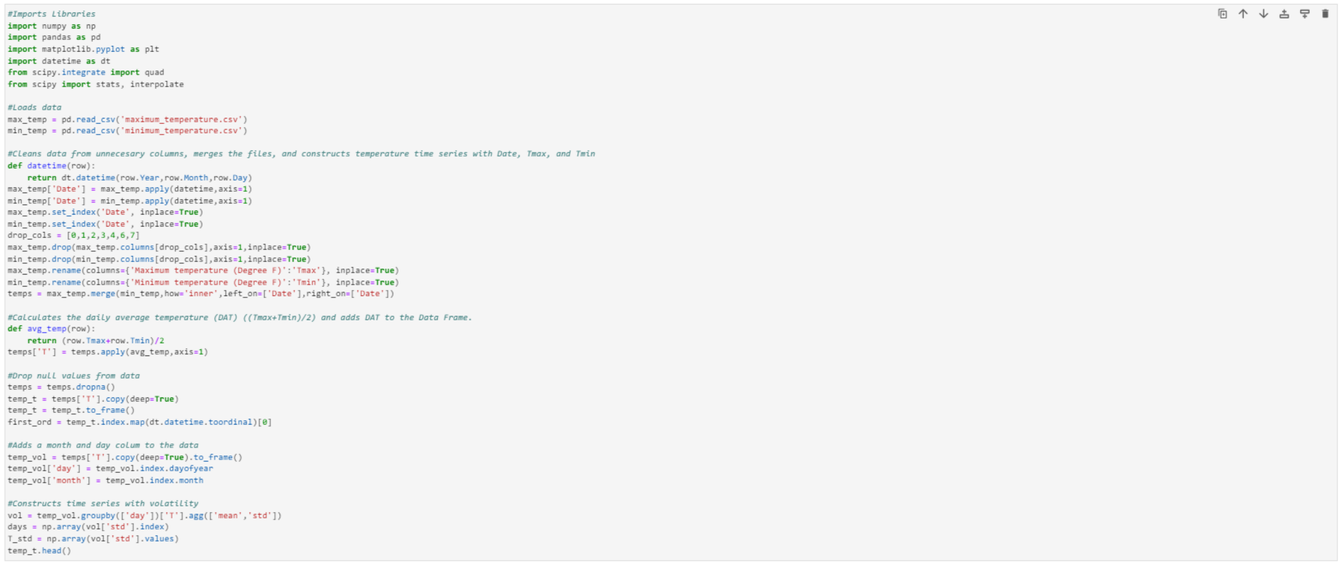


```python
no_sims = 10000
trading_dates_winter = pd.date_range(start='2023-07-01', end='2023-07-01', freq='D')
mc_temps_winter, mc_sims_winter = monte_carlo_temp(trading_dates_winter, Tbar_params, volatility, first_ord, no_sims)
trading_dates_summer = pd.date_range(start='2023-01-01', end='2023-01-01', freq='D')
mc_temps_summer, mc_sims_summer = monte_carlo_temp(trading_dates_summer, Tbar_params, volatility, first_ord, no_sims)
plt.figure(figsize=(12,6))
plt.title('Winter vs Summer Temperature MC Sims')
Tbar_summer = mc_temps_summer.iloc[-1,:]['Tbar']
Tbar_winter = mc_temps_winter.iloc[-1,:]['Tbar']
plt.hist(mc_sims_summer.iloc[-1,:],bins=20, alpha=0.5, label='Summer', color='tab:orange')
plt.plot([Tbar_summer,Tbar_summer],[0,1600], linewidth=4, label='Tbar_summer', color='tab:orange')
plt.title('Summer vs Winter Temperature MC Sims')
plt.hist(mc_sims_winter.iloc[-1,:],bins=20, alpha=0.5, label='Winter', color='tab:blue')
plt.plot([Tbar_winter,Tbar_winter],[0,1600], linewidth=4, label='Tbar_winter', color='tab:blue')
plt.legend()
plt.show()
```

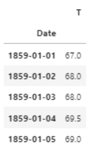


5.2. The code shown below, uses the results obtained in part 5 to price the winter temperature option with the Monte Carlo simulation and the O-U model and compares it to the results obtained with the Black-Scholes Model. Code name: WDPricing

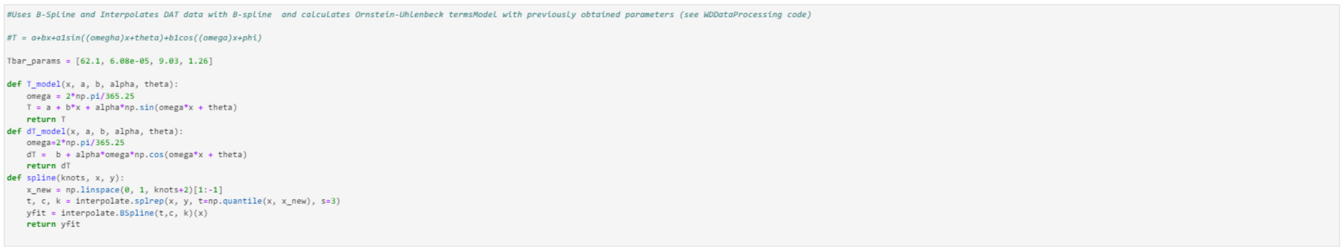
5.2.1. Data Processing.



```python
#Imports Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import datetime as dt
from scipy.integrate import quad
from scipy import stats, interpolate

#Loads data
max_temp = pd.read_csv('maximum_temperature.csv')
min_temp = pd.read_csv('minimum_temperature.csv')

#Cleans data from unnecesary columns, merges the files, and constructs temperature time series with Date, Tmax, and Tmin
def datetime(row):
    return dt.datetime(row.Year,row.Month,row.Day)
max_temp['Date'] = max_temp.apply(datetime,axis=1)
min_temp['Date'] = min_temp.apply(datetime,axis=1)
max_temp.set_index('Date', inplace=True)
min_temp.set_index('Date', inplace=True)
drop_cols = [0,1,2,3,4,6,7]
max_temp.drop(max_temp.columns[drop_cols],axis=1,inplace=True)
min_temp.drop(min_temp.columns[drop_cols],axis=1,inplace=True)
max_temp.rename(columns={'Maximum temperature (Degree F)':'Tmax'}, inplace=True)
min_temp.rename(columns={'Minimum temperature (Degree F)':'Tmin'}, inplace=True)
temps = max_temp.merge(min_temp,how='inner',left_on=['Date'],right_on=['Date'])

#Calculates the daily average temperature (DAT) ((Tmax+Tmin)/2) and adds DAT to the Data Frame.
def avg_temp(row):
    return (row.Tmax+row.Tmin)/2
temps['T'] = temps.apply(avg_temp,axis=1)

#Drop null values from data
temps = temps.dropna()
temp_t = temps['T'].copy(deep=True)
temp_t = temp_t.to_frame()
first_ord = temp_t.index.map(dt.datetime.toordinal)[0]

#Adds a month and day colum to the data
temp_vol = temps['T'].copy(deep=True).to_frame()
temp_vol['day'] = temp_vol.index.dayofyear
temp_vol['month'] = temp_vol.index.month

#Constructs time series with volatility
vol = temp_vol.groupby(['day'])['T'].agg(['mean','std'])
days = np.array(vol['std'].index)
T_std = np.array(vol['std'].values)
temp_t.head()
```



	T
Date	
1859-01-01	67.0
1859-01-02	68.0
1859-01-03	68.0
1859-01-04	69.5
1859-01-05	69.0



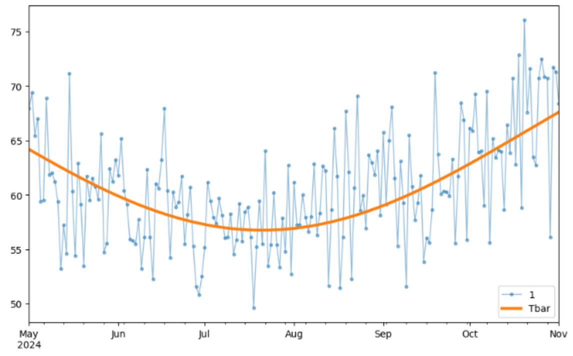
```python
#Uses B-Spline and Interpolates DAT data with B-spline  and calculates Ornstein-Uhlenbeck termsModel with previously obtained parameters (see WDDataProcessing code)

#T = a+bx+a1sin((omegha)x+theta)+b1cos((omega)x+phi)

Tbar_params = [62.1, 6.08e-05, 9.03, 1.26]

def T_model(x, a, b, alpha, theta):
    omega = 2*np.pi/365.25
    T = a + b*x + alpha*np.sin(omega*x + theta)
    return T
def dT_model(x, a, b, alpha, theta):
    omega=2*np.pi/365.25
    dT =  b + alpha*omega*np.cos(omega*x + theta)
    return dT
def spline(knots, x, y):
    x_new = np.linspace(0, 1, knots+2)[1:-1]
    t, c, k = interpolate.splrep(x, y, t=np.quantile(x, x_new), s=3)
    yfit = interpolate.BSpline(t,c, k)(x)
    return yfit
```

```python
#Uses B-Spline and Interpolates DAT data with B-spline  and calculates Ornstein-Uhlenbeck termsModel with previously obtained parameters (see WDDataProcessing code)

#T = a+bx+a1sin((omegha)x+theta)+b1cos((omega)x+phi)

Tbar_params = [62.1, 6.08e-05, 9.03, 1.26]

def T_model(x, a, b, alpha, theta):
    omega = 2*np.pi/365.25
    T = a + b*x + alpha*np.sin(omega*x + theta)
    return T
def dT_model(x, a, b, alpha, theta):
    omega=2*np.pi/365.25
    dT =  b + alpha*omega*np.cos(omega*x + theta)
    return dT
def spline(knots, x, y):
    x_new = np.linspace(0, 1, knots+2)[1:-1]
    t, c, k = interpolate.splrep(x, y, t=np.quantile(x, x_new), s=3)
    yfit = interpolate.BSpline(t,c, k)(x)
    return yfit
```

```
#Apply Euler approximation to time series.
def euler_step(row, kappa, M, lamda):
    """Function for Euler scheme approximation step in
    modified OU dynamics for temperature simulations
    Inputs:
    - dataframe row with columns: T, Tbar, dTbar and vol
    - kappa: rate of mean reversion from data processsing code
    Output:
    - temp: simulated next day temperatures
    """
    if row['T'] != np.nan:
        T_i = row['Tbar']
    else:
        T_i = row['T']
    T_det = T_i + row['dTbar']
    T_mrev =  kappa*(row['Tbar'] - T_i)
    sigma = row['vol']*np.random.randn(M)
    riskn = lamda*row['vol']
    return T_det + T_mrev + sigma - riskn

#Apply Monte Carlo approximation to time series.
def monte_carlo_temp(trading_dates, Tbar_params, vol_model, first_ord, M=1, lamda=0):
    """Monte Carlo simulation of temperature
    Inputs:
    - trading_dates: pandas DatetimeIndex from start to end dates
    - M: number of simulations
    - Tbar_params: parameters used for Tbar model
    - vol_model: fitted volatility model with days in year index
    - first_ord: first ordinal of fitted Tbar model
    Outputs:
    - mc_temps: DataFrame of all components and simulated temperatures
    """
    kappa=0.438
    if isinstance(trading_dates, pd.DatetimeIndex):
        trading_date=trading_dates.map(dt.datetime.toordinal)

    Tbars = T_model(trading_date-first_ord, *Tbar_params)
    dTbars = dT_model(trading_date-first_ord, *Tbar_params)
    mc_temps = pd.DataFrame(data=np.array([Tbars, dTbars]).T,
                            index=trading_dates, columns=['Tbar','dTbar'])
    mc_temps['day'] = mc_temps.index.dayofyear
    mc_temps['vol'] = vol_model[mc_temps['day']-1]

    mc_temps['T'] = mc_temps['Tbar'].shift(1)
    data = mc_temps.apply(euler_step, args=[kappa, M, lamda], axis=1)
    mc_sims = pd.DataFrame(data=[x for x in [y for y in data.values]],
                     index=trading_dates,columns=range(1,M+1))
    return mc_temps, mc_sims
```

5.2.1. Trading range was defined and models were applied.

```
#Define trading date range
S='2024-05-01'
E='2024-11-01'

trading_dates = pd.date_range(start= S, end= E, freq='D')
volatility = spline(5, days, T_std)
mc_temps, mc_sims = monte_carlo_temp(trading_dates, Tbar_params, volatility, first_ord)
plt.figure(figsize=(10,6))
mc_sims[1].plot(alpha=0.5,linewidth=1, marker='.')
mc_temps["Tbar"].plot(linewidth=3)
plt.legend(loc='lower right')
plt.show()
```



```
#Calculate probability of no payoff within time range.
trading_dates = pd.date_range(start= S, end= E, freq='D')
volatility = spline(5, days, T_std)
mc_temps, mc_sims = monte_carlo_temp(trading_dates, Tbar_params, volatility, first_ord)
print('Probability P(max(65-Tn, 0) = 0): {0:1.1f}%'.format(len(mc_sims[mc_sims[1] >= 65])/len(mc_sims)*100))
```

```
Probability P(max(65-Tn, 0) = 0): 20.5%
```

```
def rn_mean(time_arr, vol_arr, Tbars, lamda, kappa):
    """Evaluate the risk neutral integral above for each segment of constant volatility
    Rectangular integration with step size of days
    """
    dt = 1/365.25
    N = len(time_arr)
    mean_intervals = -vol_arr*(1 - np.exp(-kappa*dt))/kappa
    return 65*N - (np.sum(Tbars) - lamda*np.sum(mean_intervals))

def rn_var(time_arr, vol_arr, kappa):
    """Evaluate the risk neutral integral above for each segment of constant volatility
    Rectangular integration with step size of days
    """
    dt = 1/365.25
6 AMar_arr = np.power(vol_arr,2)
    ar_intervals = var_arr/(2*kappa)*(1-np.exp(-2*kappa*dt))
    cov_sum = 0
    for i, ti in enumerate(time_arr):
        for j, tj in enumerate(time_arr):
            if j > i:
                cov_sum += np.exp(-kappa*(tj-ti)) * var_intervals[i]
    return np.sum(var_intervals) + 2*cov_sum

def risk_neutral(trading_dates, Tbar_params, vol_model, first_ord, lamda, kappa=0.438):
    if isinstance(trading_dates, pd.DatetimeIndex):
        trading_date=trading_dates.map(dt.datetime.toordinal)

    Tbars = T_model(trading_date-first_ord, *Tbar_params)
    dTbars = dT_model(trading_date-first_ord, *Tbar_params)
    mc_temps = pd.DataFrame(data=np.array([Tbars, dTbars]).T,
                            index=trading_dates, columns=['Tbar','dTbar'])
    mc_temps['day'] = mc_temps.index.dayofyear
    mc_temps['vol'] = vol_model[mc_temps['day']-1]
    time_arr = np.array([i/365.25 for i in range(1,len(trading_dates)+1)])
    vol_arr = mc_temps['vol'].values
    mu_rn = rn_mean(time_arr, vol_arr, Tbars, lamda, kappa)
    var_rn = rn_var(time_arr, vol_arr, kappa)
    return mu_rn, var_rn
```

```
def winter_option(trading_dates, r, alpha, K, tau, opt='c', lamda=0.0):
    """Evaluate the fair value of temperature option in winter
    Based on heating degree days only if the physical probability that
    the average temperature exceeds the Tref (65 deg F) is close to 0
    """
    mu_rn, var_rn = risk_neutral(trading_dates, Tbar_params, volatility, first_ord, lamda)
    disc = np.exp(-r*tau)
    vol_rn = np.sqrt(var_rn)
    zt = (K - mu_rn)/vol_rn
    exp = np.exp(-zt**2/2)
    if opt == 'c':
        return alpha*disc*((mu_rn - K)*stats.norm.cdf(-zt) + vol_rn*exp/np.sqrt(2*np.pi))
    else:
        exp2 = np.exp(-mu_rn**2/(2*vol_rn**2))
        return alpha*disc*((K - mu_rn)*(stats.norm.cdf(zt) - stats.norm.cdf(-mu_rn/vol_rn)) +
                           vol_rn/np.sqrt(2*np.pi)*(exp-exp2))
trading_dates = pd.date_range(start= S, end= E, freq='D')
r=0.05
K=300
alpha=2500
def years_between(d1, d2):
    d1 = dt.datetime.strptime(d1, "%Y-%m-%d")
    d2 = dt.datetime.strptime(d2, "%Y-%m-%d")
    return abs((d2 - d1).days)/365.25
start = dt.datetime.today().strftime('%Y-%m-%d')
end = E
tau = years_between(start, end)
print('Start Valuation Date:', start,
      '\nEnd of Contract Date:', end,
      '\nYears between Dates :', round(tau,3))
print('Call Price: ', round(winter_option(trading_dates, r, alpha, K, tau, 'c'),2))
print('Put Price : ', round(winter_option(trading_dates, r, alpha, K, tau, 'p'),2))
```

```
Start Valuation Date: 2024-05-01
End of Contract Date: 2024-11-01
Years between Dates : 0.504
Call Price:  1430890.03
Put Price :  0.0
```

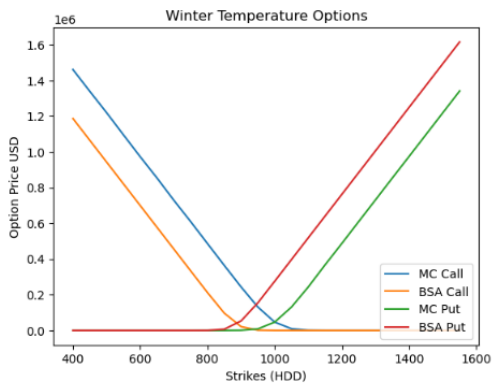
```
# define trading date range
trading_dates = pd.date_range(start= S, end= E, freq='D')
no_sims = 10000
vol_model = spline(5, days, T_std)
def temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, K, tau, first_ord, opt='c'):
    "Evaluates the price of a temperature call option"
    mc_temps, mc_sims = monte_carlo_temp(trading_dates, Tbar_params, volatility, first_ord, no_sims)
    N, M = np.shape(mc_sims)
    mc_arr = mc_sims.values
    DD = np.sum(np.maximum(mc_arr-65,0), axis=0)
    if opt == 'c':
        CT = alpha*np.maximum(DD-K,0)
    else:
        CT = alpha*np.maximum(K-DD,0)
    C0 = np.exp(-r*tau)*np.sum(CT)/M
    sigma = np.sqrt( np.sum( (np.exp(-r*tau)*CT - C0)**2) / (M-1) )
    SE = sigma/np.sqrt(M)
    return C0, SE
call = np.round(temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, K, tau, first_ord, 'c'),2)
put = np.round(temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, K, tau, first_ord, 'p'),2)
print('Call Price: {0} +/- {1} (2se)'.format(call[0], call[1]*2))
print('Put Price : {0} +/- {1} (2se)'.format(put[0], put[1]*2))
```

```
Call Price: 0.0 +/- 0.0 (2se)
Put Price : 447213.16 +/- 1110.48 (2se)
```

```
# define trading date range
trading_dates = pd.date_range(start= S, end= E, freq='D')
no_sims = 10000
vol_model = spline(5, days, T_std)
def temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, K, tau, first_ord, opt='c'):
    "Evaluates the price of a temperature call option"
    mc_temps, mc_sims = monte_carlo_temp(trading_dates, Tbar_params, volatility, first_ord, no_sims)
    N, M = np.shape(mc_sims)
    mc_arr = mc_sims.values
    DD = np.sum(np.maximum(65-mc_arr,0), axis=0)
    if opt == 'c':
        CT = alpha*np.maximum(DD-K,0)
    else:
        CT = alpha*np.maximum(K-DD,0)
    C0 = np.exp(-r*tau)*np.sum(CT)/M
    sigma = np.sqrt( np.sum( (np.exp(-r*tau)*CT - C0)**2) / (M-1) )
    SE = sigma/np.sqrt(M)
    return C0, SE
call = np.round(temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, K, tau, first_ord, 'c'),2)
put = np.round(temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, K, tau, first_ord, 'p'),2)
print('Call Price: {0} +/- {1} (2se)'.format(call[0], call[1]*2))
print('Put Price : {0} +/- {1} (2se)'.format(put[0], put[1]*2))
```

```
Call Price: 1704779.41 +/- 2338.26 (2se)
Put Price : 0.0 +/- 0.0 (2se)
```

```
#Compare Black Scholes with MC Simulations Method
strikes = np.arange(400,1600,50)
data = np.zeros(shape=(len(strikes),4))
for i, strike in enumerate(strikes):
    data[i,0] = temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, strike, tau, first_ord, 'c')[0]
    data[i,1] = winter_option(trading_dates, r, alpha, strike, tau, 'c')
    data[i,2] = temperature_option(trading_dates, no_sims, Tbar_params, vol_model, r, alpha, strike, tau, first_ord, 'p')[0]
    data[i,3] = winter_option(trading_dates, r, alpha, strike, tau, 'p')
df = pd.DataFrame({'MC Call': data[:, 0], 'BSA Call': data[:, 1], 'MC Put': data[:, 2], 'BSA Put': data[:, 3]})
df.index = strikes
plt.plot(df)
plt.title('Winter Temperature Options')
plt.ylabel('Option Price USD')
plt.xlabel('Strikes (HDD)')
plt.legend(df.columns, loc=4)
plt.show()
```



6. CONCLUSIONS AND RECOMMENDATIONS

In this study, it was demonstrated a comprehensive approach to analyzing temperature data and estimating the price of temperature weather options. Python programs were developed to process temperature data and fit the average temperature along with its volatility using B-splines. This methodology enabled us to capture the intricate patterns and fluctuations in temperature and volatility time series.

Subsequently, a Monte Carlo simulation was implemented, employing the modified Ornstein-Uhlenbeck method to estimate the price of temperature weather options. By simulating multiple paths of the temperature time series, we obtained reliable estimates of option prices, accounting for the stochastic nature of temperature dynamics.

Finally, we compared the obtained prices of temperature weather options with those derived from the Black-Scholes model for call and put options. This comparative analysis provided valuable insights into the effectiveness of our approach and highlighted the importance of incorporating temperature dynamics in pricing weather derivatives.

Overall, our findings suggest that incorporating temperature data and utilizing advanced simulation techniques can significantly enhance the accuracy of pricing temperature weather options, thereby enabling better risk management strategies for stakeholders in weather-sensitive industries.

Refinement of Simulation Models: Further research could focus on refining the simulation models used for estimating temperature dynamics. Exploring alternative stochastic processes or incorporating additional factors could potentially improve the accuracy of option price estimates.

Incorporation of Additional Data Sources: Integrating other relevant data sources such as humidity, wind speed, or precipitation could enrich the analysis and provide a more comprehensive understanding of weather-related risk factors.

Extension to Other Weather Derivatives: While our study primarily focused on temperature weather options, future research could extend the analysis to other types of weather derivatives such as rainfall or snowfall options. This expansion would broaden the applicability of the developed methodology.

Validation and Sensitivity Analysis: Conducting extensive validation studies and sensitivity analyses can enhance the robustness of the proposed approach. Investigating the impact of varying model parameters and assumptions on option prices would provide valuable insights into the reliability of the results.

Integration with Risk Management Frameworks: Exploring ways to integrate the pricing of temperature weather options into broader risk management frameworks within industries sensitive to weather fluctuations could facilitate more informed decision-making processes.

By addressing these avenues for future work, researchers can further advance the understanding and application of quantitative methods in weather derivative pricing, ultimately contributing to improved risk management practices in weather-sensitive industries

References

Alaton, P., Djehiche, B., & Stillberger, D. (2002). On modeling and pricing weather derivatives. Applied mathematical finance, 9(1), 1-20.

Bensoussan, A., & Brouste, A. (2016). Cox–Ingersoll–Ross model for wind speed modeling and forecasting. Wind Energy, 19(7), 1355-1365.

Benth, F. E., & Saltyte-Benth, J. (2012). Modeling and pricing in financial markets for weather derivatives (Vol. 17). World Scientific.

Benth, F. E., Benth, J. S., & Koekebakker, S. (2008). Stochastic modeling of electricity and related markets (Vol. 11). World Scientific.

Black, F., and Scholes, M. (1973), "The Pricing of Options and Corporate Liabilities," Journal of Political Economy, 81, 637–654.

Bloomfield, P. (1992), "Trends in Global Temperature," Climate Change, 21,1–16.

Bollerslev, T. (1986), "Generalized ARCH," Journal of Econometrics, 31,307–327.

Bollerslev, T., and Wooldridge, J. M. (1992), "Quasi-Maximum Likelihood Estimation and Inference in Dynamic Models With Time-Varying Covariances," Econometric Reviews, 11, 143–172

Brewer, M. (2000). Weather Derivatives: Risk Management Tools for Weather-Sensitive Markets. *The Journal of Futures Markets, 20*(3), 219-244.

CME Group Inc., (2021). Weather Futures and Options, Retrieved From *https://www.cmegroup.com/trading/weather/files/weather-fact-card.pdf*

Doob, J.L. (April 1942). "The Brownian Movement and Stochastic Equations". Annals of Mathematics. 43 (2): 351–369

Dornier, F., & Queruel, M. (2000). Caution to the wind. Energy & power risk management, 13(8), 30-32.

Dzupire, N. C., Ngare, P., & Odongo, L. (2019). Pricing basket weather derivatives on rainfall and temperature processes. International Journal of Financial Studies, 7(3), 35.

G. Considine (2000). Introduction to weather derivatives. Aquila Energy, http://www.

Golden, L. L., M. Wang and C. C. Yang "Handling Weather Related Risks Through the Financial Markets: Considerations of Credit Risk, Basis Risk, and Hedging." Journal of Risk & Insurance, Vol. 74, No. 2, pp. 319–346, June 2007.

Hartmann, J., & Geyer, R. (2005). Weather derivatives as part of integrated risk management concepts. *Weather, 60*(11), 320-324.

Jewson, S., A. Brix and C. Ziehmann (2005). "Weather Derivatives Valuation: The Meteorological, Statistical, Financial, and Mathematical Foundations". Cambridge, Cambridge University Press.

Mathews, J. S. (2009) "Dog Days and Degree Days", CME Group

Myers, R. 2008 "What Every CFO Needs to Know About Weather Risk Management", Storm Exchange, Inc. & CME Group

Oetomo, T., & Stevenson, M. (2005). Hot or cold? A comparison of different approaches to the pricing of weather derivatives. Journal of Emerging Market Finance, 4(2), 101-133.

Øksendal, Bernt (1998). "Option Pricing". Stochastic Differential Equations: An Introduction with Applications (5th ed.). Berlin: Springer. pp. 266–283

Stein, J., Lopes, S. R. C., & Medino, A. V. (2021). A Generalization of the Ornstein-Uhlenbeck Process: Theoretical Results, Simulations and Parameter Estimation. arXiv preprint arXiv:2108.06374.

Tang, K., Ed. (2010). "Weather Risk Management: A Guide for Corporations, Hedge Funds and Investors" Archived 2010-01-22 at the Wayback Machine, Risk Books.

Weng, Y., & Zeng, H. (2017). Modeling and Pricing of Weather Derivatives. In *Quantitative Financial Risk Management* (pp. 351-379). Springer, Cham.

Wigley, Tom M. L. (2006). "Temperature Trends in the Lower Atmosphere — Understanding and Reconciling Differences (Executive Summary)" (PDF). The US Climate Change Science Program. NOAA