AN INTRODUCTION TO APPLIED QUANTITATIVE FINANCE

Market Dynamics and Quantitative Analysis

Contents

1. Market Analysis

2. Statistical Analysis

3. Forecasting Techniques

4. Pricing Techniques

REQUIREMENTS TO READ THIS BOOK

Prerequisites

This book assumes that readers have:

1. A basic understanding of mathematics.
2. Fundamental knowledge of statistics and finance.
3. A working knowledge of Python and basic programming principles.

Tools and Software

To follow along with the examples in this book, you will need the following tools:

4. Python (3.11.7): The primary language used for all code examples.

Libraries:

5. NumPy: For numerical computations.
6. Pandas: For data manipulation and analysis.

7. Plotly: For interactive visualizations.

8. SciPy: For scientific computing, including optimization and statistics.

The book is designed to be interactive, and you should feel free to experiment with the code examples.

Market Analysis

Market analysis is the process of studying and interpreting various factors that influence the behavior of financial markets. It is a key component of both investment strategy and risk management, providing insights into how prices of assets such as stocks, bonds, commodities, and currencies are likely to move over time. Market analysis involves examining a wide range of data, from historical price trends to economic indicators, in order to make informed decisions about market behavior. There are two primary types of market analysis: fundamental analysis and technical analysis. Fundamental Analysis focuses on understanding the underlying economic factors that drive market movements. This includes analyzing company financials, economic reports, interest rates, inflation, and geopolitical events. In finance, fundamental analysis seeks to assess the intrinsic value of an asset and determine whether it is overvalued or undervalued. Technical Analysis relies on historical market data, particularly price and volume, to identify patterns and trends. Technical analysts believe that market prices reflect all available information and that past price movements can predict future trends. This analysis often involves the use of charts, indicators, and statistical models to forecast price movements. Quantitative market analysis, which blends elements of both fundamental and technical analysis, uses mathematical models and statistical techniques to predict market behavior. It often involves the development of algorithms, backtesting trading strategies, and -

using advanced tools like machine learning to improve forecasting accuracy. In this book, we will explore the dynamics of financial markets using applied quantitative models, leveraging tools such as Python, data science libraries, and financial theory to provide a systematic approach to market analysis. By combining both traditional and modern techniques, we aim to gain deeper insights into how markets operate and how we can better understand and predict their movements.

The Stochastic Volatility Hypothesis

The Stochastic Volatility Hypothesis proposes that volatility is not constant but rather evolves in a random, unpredictable manner. This stands in contrast to traditional models like Black-Scholes, which assume constant volatility. Financial markets often exhibit volatility clustering, where high-volatility periods tend to be followed by high-volatility periods, and low-volatility periods tend to cluster as well.

The Heston Model

One of the most widely used stochastic volatility models is the Heston Model, developed by Steven Heston in 1993. This model assumes that volatility itself is driven by a mean-reverting process, where the volatility tends to return to a long-term average over time. Model Dynamics: In the Heston model, volatility follows a mean-reverting square-root process:

$$dV^{t} = \kappa(\theta - V^{t})dt + \sigma\sqrt{V^{t}}\,dW^{t^{V}}$$

where:

- (V^t) is the instantaneous variance (volatility squared),
- (κ) is the speed of mean reversion,
- (θ) is the long-term variance level,
- (σ) is the volatility of volatility (also known as the "volatility of variance"),
- (dW^{t^V}) is a Wiener process (a random noise term).

The Heston model incorporates two key features:

1. **Mean reversion**: The volatility tends to revert to a long-term average, (θ), over time, representing the stabilization of market conditions.

2. **Volatility of volatility**: The model allows for volatility to vary over time, with large fluctuations (i.e., periods of high volatility) potentially followed by low-volatility periods, thus capturing observed market behavior.

```
import numpy as np
import plotly.graph_objects as go

import pandas as pd
```

```
# Load SPY data
data =  pd.read_csv("SPY_daily_data_2020_2024.csv", index_col="Date",
parse_dates=True)
S0 = data['Close'].iloc[-1]  # Use the latest closing price as the starting price

# Heston model parameters
V0 = 0.04       # Initial variance
rho = -0.7      # Correlation
kappa = 1.5     # Mean reversion speed
theta = 0.04    # Long-term variance level
sigma_v = 0.3   # Volatility of volatility
T = 1           # Time in years
dt = 1/252      # 1-day steps
N = int(T / dt) # Total steps
n_simulations = 10  # Number of simulated paths

simulated_prices = np.zeros((N, n_simulations))
simulated_volatilities = np.zeros((N, n_simulations))

# Initialize the first row with initial conditions
simulated_prices[0, :] = S0
simulated_volatilities[0, :] = V0

# Simulate paths
for i in range(1, N):
dW_s = np.random.normal(0, np.sqrt(dt), n_simulations)  # Wiener process
```

```
dW_v = rho * dW_s + np.sqrt(1 - rho ** 2) * np.random.normal(0,
np.sqrt(dt), n_simulations)  # Correlated Wiener process

# Variance update
simulated_volatilities[i, :] = (
simulated_volatilities[i-1, :]
+ kappa * (theta - simulated_volatilities[i-1, :]) * dt
+ sigma_v * np.sqrt(simulated_volatilities[i-1, :]) * dW_v
)

# Ensure non-negative variances
simulated_volatilities[i, :] = np.maximum(simulated_volatilities[i, :], 0)

# Price update
simulated_prices[i, :] = simulated_prices[i-1, :] * np.exp(
(-0.5 * simulated_volatilities[i-1, :] * dt) +
np.sqrt(simulated_volatilities[i-1, :] * dt) * dW_s
)


future_dates = pd.date_range(start=data.index[-1] + pd.Timedelta(days=1),
periods=N, freq='B')


fig = go.Figure()
```

```python
fig.add_trace(go.Scatter(
    x=data.index,
    y=data['Close'],
    mode='lines',
    name='Historical SPY Prices',
    line=dict(color='orange', width=2)
))


for j in range(n_simulations):
    fig.add_trace(go.Scatter(
        x=future_dates,
        y=simulated_prices[:, j],
        mode='lines',
        name=f'Simulated Path {j+1}',
        line=dict(width=1)
    ))


fig.update_layout(
    title="SPY Historical Prices with Heston Model Simulated Paths",
    xaxis_title="Date",
    yaxis_title="Price (USD)",
    template="plotly_white"
)
```

```
fig.show()
```

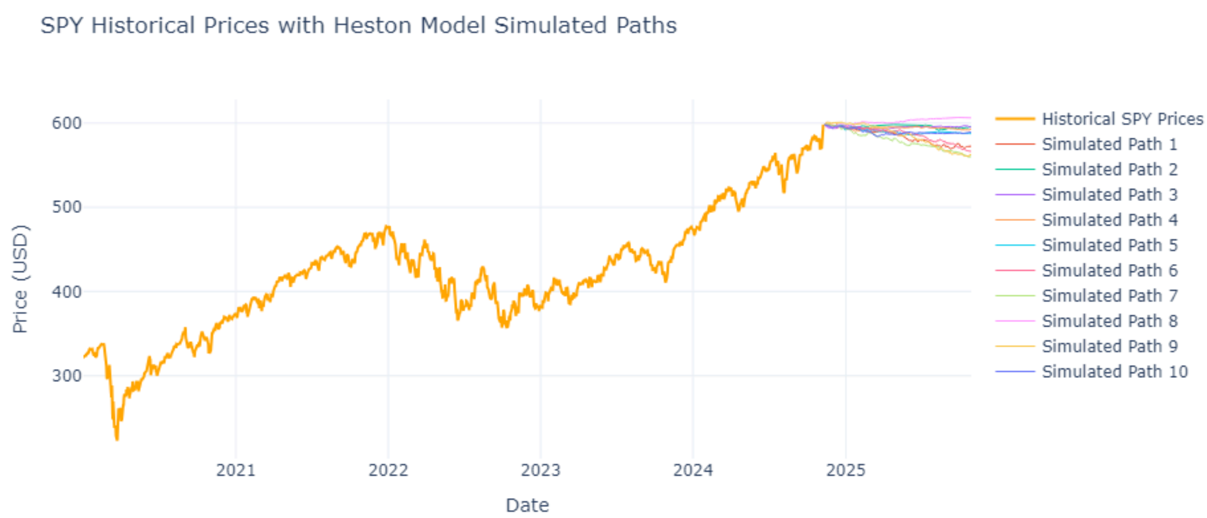

Figure 1.1

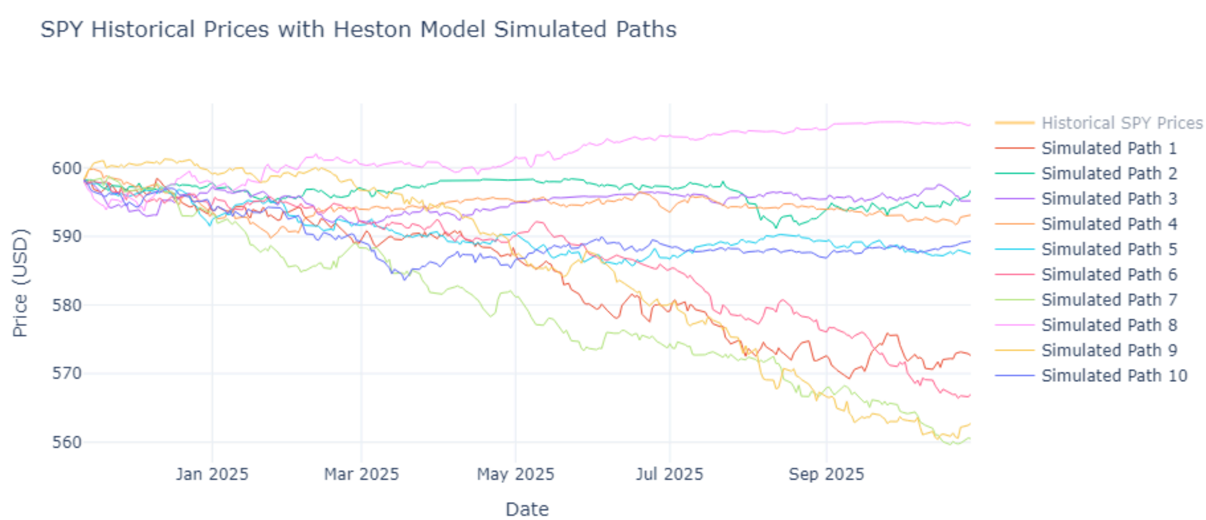

Figure 1.2

```
import numpy as np
import pandas as pd
```

```python
import plotly.graph_objects as go


data = pd.read_csv("SPY_daily_data_2020_2024.csv", index_col="Date", parse_dates=True)


data['Returns'] = data['Close'].pct_change()

# Calculate the rolling volatility
data['Rolling_Volatility'] = data['Returns'].rolling(window=30).std() * np.sqrt(252)  # Annualize volatility

# Initial variance (V0): Variance of the first 30 days' returns
V0 = data['Rolling_Volatility'].iloc[30] ** 2
print(f"Initial Variance (V0): {V0}")

# Long-term variance level: Mean of the rolling volatilities
theta = data['Rolling_Volatility'].mean() ** 2
print(f"Long-term Variance (theta): {theta}")

# Estimate the volatility of volatility: Standard deviation of the rolling volatilities
sigma_v = data['Rolling_Volatility'].std()
print(f"Volatility of Volatility (sigma_v): {sigma_v}")
```

```
data['Rolling_Covariance'] =
data['Returns'].rolling(window=30).cov(data['Rolling_Volatility'])
data['Rolling_Variance'] =
data['Rolling_Volatility'].rolling(window=30).var()
data['Rolling_Correlation'] = data['Rolling_Covariance'] /
np.sqrt(data['Rolling_Variance'] *
data['Rolling_Volatility'].rolling(window=30).var())
rho = data['Rolling_Correlation'].mean()
print(f"Correlation (rho): {rho}")


# Calculate the log differences of the rolling volatility over time and apply
linear regression to estimate kappa
log_diff_vol = np.log(data['Rolling_Volatility'].dropna()).diff().dropna()
kappa = -log_diff_vol.mean() / np.mean(data['Rolling_Volatility'].dropna())
print(f"Estimated Mean Reversion Speed (kappa): {kappa}")



S0 = data['Close'].iloc[-1]

# Heston model parameters
V0 = V0       # Initial variance
rho = rho     # Correlation
kappa = kappa # Mean reversion speed
theta = theta # Long-term variance level
sigma_v = sigma_v # Volatility of volatility
```

```python
T = 1            # Time in years
dt = 1/252       # 1-day steps
N = int(T / dt) # Total steps
n_simulations = 10  # Number of simulated paths


simulated_prices = np.zeros((N, n_simulations))
simulated_volatilities = np.zeros((N, n_simulations))


simulated_prices[0, :] = S0
simulated_volatilities[0, :] = V0

# Simulate paths
for i in range(1, N):
dW_s = np.random.normal(0, np.sqrt(dt), n_simulations)  # Wiener process

dW_v = rho * dW_s + np.sqrt(1 - rho ** 2) * np.random.normal(0,
np.sqrt(dt), n_simulations)  # Correlated Wiener process

# Variance update
simulated_volatilities[i, :] = (
simulated_volatilities[i-1, :]
+ kappa * (theta - simulated_volatilities[i-1, :]) * dt
+ sigma_v * np.sqrt(simulated_volatilities[i-1, :]) * dW_v
)
```

```
# Ensure non-negative variances
simulated_volatilities[i, :] = np.maximum(simulated_volatilities[i, :], 0)

# Price update
simulated_prices[i, :] = simulated_prices[i-1, :] * np.exp(
(-0.5 * simulated_volatilities[i-1, :] * dt) +
np.sqrt(simulated_volatilities[i-1, :] * dt) * dW_s
)

future_dates = pd.date_range(start=data.index[-1] + pd.Timedelta(days=1),
periods=N, freq='B')


fig = go.Figure()


fig.add_trace(go.Scatter(
x=data.index,
y=data['Close'],
mode='lines',
name='Historical SPY Prices',
line=dict(color='orange', width=2)
))


for j in range(n_simulations):
fig.add_trace(go.Scatter(
```

```
x=future_dates,
y=simulated_prices[:, j],
mode='lines',
name=f'Simulated Path {j+1}',
line=dict(width=1)
))

fig.update_layout(
title="SPY Historical Prices with Heston Model Simulated Paths",
xaxis_title="Date",
yaxis_title="Price (USD)",
template="plotly_white"
)

fig.show()
```

Below we will dive deeper into our code.

Calculating Daily Returns

```
# Calculate daily returns
data['Returns'] =
data['Close'].pct_change()
```

We compute daily returns by calculating the percentage change in the closing price from one day to the next. These returns are essential for estimating volatility and understanding the price movements.

Rolling Volatility Calculation

```
# Calculate the rolling volatility (standard deviation of returns over a window of 30 days, for example)

data['Rolling_Volatility'] = data['Returns'].rolling(window=30).std() * np.sqrt(252)  # Annualize volatility
```

The rolling volatility represents the variability in asset returns over a specific window of time. In this case, we use a 30-day window to calculate the standard deviation of returns. Volatility is annualized by multiplying by the square root of 252 (the number of trading days in a year).

Estimating Initial Variance (V0)

```
# Initial variance (V0): Variance of the first 30 days' returns

V0 = data['Rolling_Volatility'].iloc[30] ** 2
```

The initial variance (V0) is the square of the rolling volatility from the first 30 days of data. This serves as the starting point for the stochastic volatility model.

Long-term Variance Level (Theta)

```
# Long-term variance level (theta): Mean of the rolling volatilities
theta = data['Rolling_Volatility'].mean() ** 2
```

We estimate the long-term variance level (theta) as the square of the average of the rolling volatility over the entire period. This represents the variance to which the volatility process will revert over time.

Estimating Volatility of Volatility (Sigma_V)

```
# Estimate the volatility of volatility (sigma_v): Standard deviation of the rolling volatilities
sigma_v = data['Rolling_Volatility'].std()
```

The volatility of volatility (σ_v) is the standard deviation of the rolling volatilities. It quantifies how much volatility itself fluctuates over time, providing insight into how uncertain the volatility process is.

Correlation Between Returns and Volatility (Rho)

```
# Estimate the correlation between returns and volatility (rho)
```

```
data['Rolling_Covariance'] =
data['Returns'].rolling(window=30).cov(data['Rolling_Volatility'])

data['Rolling_Variance'] =
data['Rolling_Volatility'].rolling(window=30).var()

data['Rolling_Correlation'] =
data['Rolling_Covariance'] /
np.sqrt(data['Rolling_Variance'] *
data['Rolling_Volatility'].rolling(window=30).var())

rho = data['Rolling_Correlation'].mean()
```

We estimate the correlation (rho) between returns and volatility by calculating the rolling covariance and variance between the two. This correlation represents the relationship between the asset's price movements and its volatility.

Estimating the Mean Reversion Speed (Kappa)

```
# Estimate the mean reversion speed
(kappa) based on decay rate of volatility

log_diff_vol =
np.log(data['Rolling_Volatility'].dropna()).diff().dropna()
```

```
kappa = -log_diff_vol.mean() / np.mean(data['Rolling_Volatility'].dropna())
```

The mean reversion speed (kappa) controls how quickly the volatility reverts to its long-term mean. We estimate it by looking at the decay rate of volatility over time. Specifically, we calculate the logarithmic difference of rolling volatility values, and use the average decay rate to approximate `kappa`.

Heston Model Simulation Setup

```
S0 = data['Close'].iloc[-1]  # Use the latest closing price as the starting price
```

We use the most recent closing price as the starting price for the Heston model simulation.

Simulation Parameters

```
V0 = V0        # Initial variance

rho = rho      # Correlation

kappa = kappa # Mean reversion speed

theta = theta # Long-term variance level

sigma_v = sigma_v # Volatility of volatility

T = 1              # Time in years
```

```
dt = 1/252      # 1-day steps, assuming 252 trading days in a year
N = int(T / dt) # Total steps for 1 year of simulation
n_simulations = 10  # Number of simulated paths
```

We define the key parameters needed for the Heston model simulation:

- `V0`: Initial variance
- `rho`: Correlation between returns and volatility
- `kappa`: Mean reversion speed
- `theta`: Long-term variance level
- `sigma_v`: Volatility of volatility
- `T`: The length of the simulation (1 year)
- `dt`: Time step (1 day)
- `N`: Total number of time steps in the simulation
- `n_simulations`: Number of paths to simulate

Simulating Asset Price Paths

```
# Create arrays to store simulated paths
simulated_prices = np.zeros((N, n_simulations))
```

```
simulated_volatilities = np.zeros((N, n_simulations))

# Initialize the first row with initial conditions
simulated_prices[0, :] = S0
simulated_volatilities[0, :] = V0
```

We initialize arrays to store the simulated asset prices and volatilities for each path. The first row is set to the initial conditions (latest closing price for prices and initial variance for volatilities).

```
# Simulate paths
for i in range(1, N):
    dW_s = np.random.normal(0, np.sqrt(dt), n_simulations)  # Wiener process for stock
    dW_v = rho * dW_s + np.sqrt(1 - rho ** 2) * np.random.normal(0, np.sqrt(dt), n_simulations)  # Correlated Wiener process
```

```
# Variance update
simulated_volatilities[i, :] = (
simulated_volatilities[i-1, :]
+ kappa * (theta -
simulated_volatilities[i-1, :]) * dt
+ sigma_v *
np.sqrt(simulated_volatilities[i-1, :]) *
dW_v
)

# Ensure non-negative variances
simulated_volatilities[i, :] =
np.maximum(simulated_volatilities[i, :],
0)

# Price update
simulated_prices[i, :] =
simulated_prices[i-1, :] * np.exp(
```

```
(-0.5 *
simulated_volatilities[i-1, :] * dt) +
np.sqrt(simulated_volatilities[i-1, :] *
dt) * dW_s

)
```

We use a for loop to simulate the asset prices and volatilities over time. For each time step:

- We generate a Wiener process for the stock price (`dW_s`) and for volatility (`dW_v`), which are correlated.
- The volatility is updated using the mean reversion model and the volatility of volatility.
- We ensure the volatility remains non-negative by using `np.maximum()`.
- The asset price is updated using the exponential form of the Heston mode

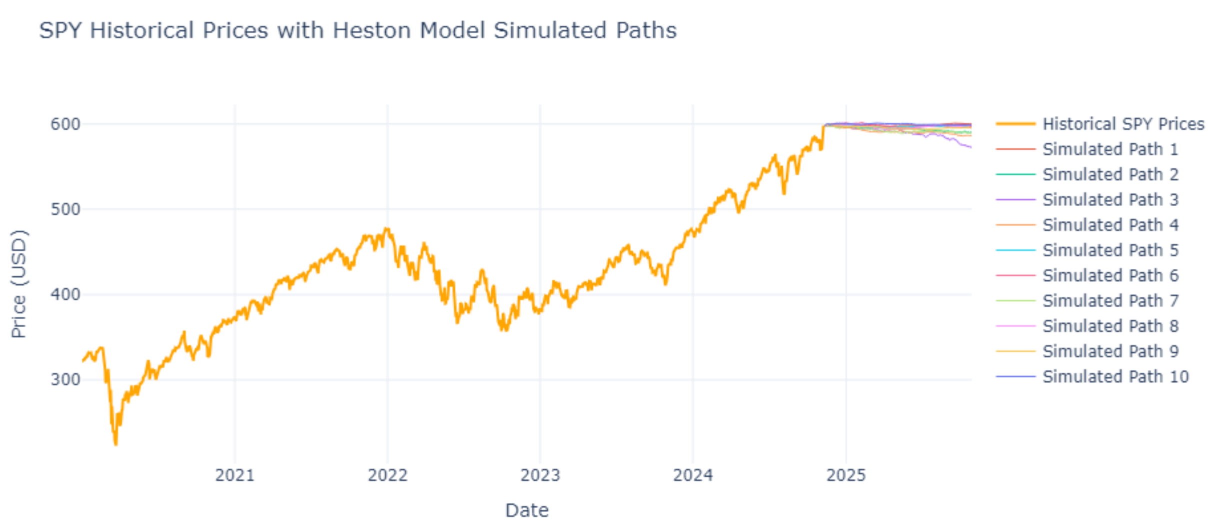

Figure 2.1

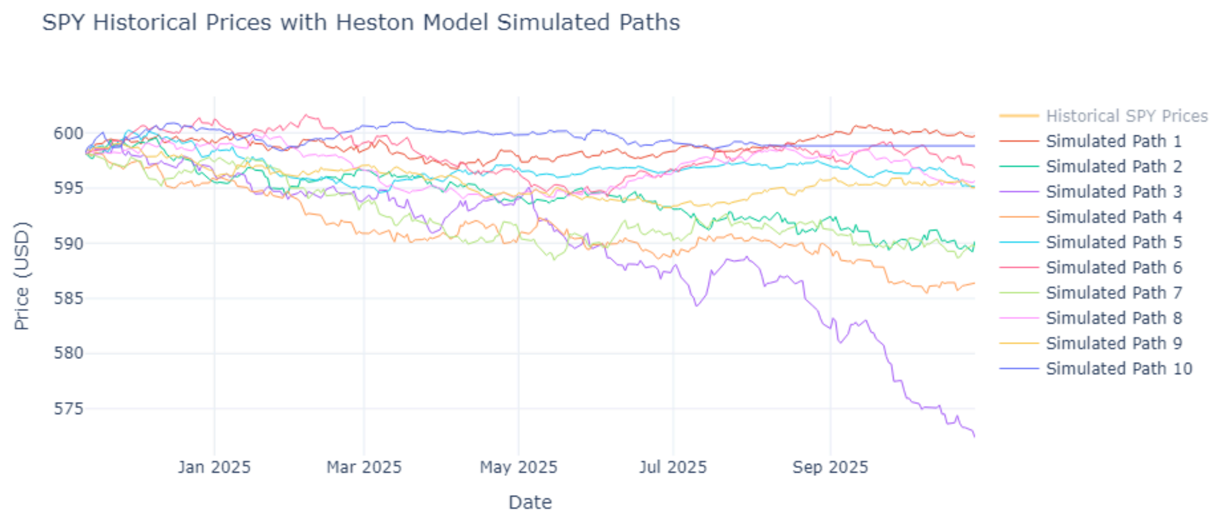

Figure 2.2

Initial Variance (V0): 0.013913262328120103
Long-term Variance (theta): 0.03290706851546082
Volatility of Volatility (sigma_v): 0.11588246558354673
Correlation (rho): 0.06633896057470169
Estimated Mean Reversion Speed (kappa): -0.00038500700729454245

Variance Gamma Model

In financial markets, asset prices do not always follow a normal distribution. Extreme market events such as crashes, bubbles, or other shocks often produce returns that exhibit both skewness and heavy tails, characteristics not captured by the basic Black-Scholes model. To address this issue, the Variance Gamma (VG) model extends the traditional Brownian motion by incorporating a Gamma process to allow for both skewness and kurtosis in the return distribution.

The VG model is based on a combination of Brownian motion and a gamma process. The price process of an asset in the VG model can be written as:

$$S^{t} = S^{o} exp((\mu - \sigma^{2}/2)\, t + \sigma W^{t} + X^{t})$$

where:

- (S^t) is the asset price at time t,
- (S^0) is the initial asset price,
- (μ) is the drift (expected return),
- (σ) is the volatility (standard deviation),
- (W^t) is a Wiener process (standard Brownian motion),
- (X^t)is a Gamma process that introduces jumps to the asset price.

Key Parameters:

1. μ: The drift of the asset price process (the expected return).
2. σ: The volatility of the asset (standard deviation of returns).

Gamma Process (X^t X^t):

This process is defined by a Gamma distribution with parameters θ (scale) and ν (shape). It introduces skewness and heavy tails into the asset price dynami cs.

The Gamma process X^t X^t is governed by the following distribution:

$$X^{t} \sim Gamma(v,\ \theta)$$

Where:

- ν controls the shape of the distribution (affects the kurtosis or the "fat tails" of the return distribution),
- θ controls the scale (affects the skewness of the return distribution).

Thus, the asset price S^t S^t in the VG model is driven by both the continuous Brownian motion component (the first term in the equation) and the jump component (the term involving X^t X^t).

The Density Function of the Variance Gamma Process

The probability density function (PDF) for the returns under the VG model, given by

$r = 1n(S^t/S^0)$, is:

$f(r) = 1/\sqrt{2\pi}$vσ ^2t exp ((r-μt)^2/2vσ^2t) * (1/r(v)) * (|r|/θ)^v-1 exp (- |r|/θ)

Where:

- (r) is the return over time t,
- (μ) is the drift parameter (expected return),
- (σ^2) is the variance (volatility squared),
- (ν) and (θ) are the shape and scale parameters of the Gamma distribution that describe the jump component of the process.

```
import numpy as np
```

```
import pandas as pd
import plotly.graph_objects as go
from scipy.stats import gamma, skew, kurtosis
from scipy.optimize import minimize


data = pd.read_csv("TSM_daily_data_2020_2024.csv",
index_col="Date", parse_dates=Tr ue)
data['Returns'] = data['Close'].pct_change()

# Set initial parameters
mu = 0.05        # Drift
sigma = 0.1      # Volatility
theta = 0.01      # Jump scale parameter
nu = 0.1          # Jump shape parameter
S0 = data['Close'].iloc[-1]  # Latest price as initial asset price

# Define the VG model simulation function
def simulate_vg_paths(S0, mu, sigma, theta, nu, T=1, dt=1/252,
n_simulations=10):
N = int(T / dt)  # Total time steps for 1 year
simulated_paths = np.zeros((N, n_simulations))
simulated_paths[0, :] = S0  # Initial price for all paths

for i in range(1, N):
gamma_increment = np.random.gamma(shape=dt/nu, scale=nu,
size=n_simulations)
```

```
brownian_increment = np.random.normal(0,
np.sqrt(gamma_increment), n_simulations)

# Update paths with the VG formula
simulated_paths[i, :] = simulated_paths[i-1, :] * np.exp(
(mu - 0.5 * sigma**2) * dt
+ sigma * brownian_increment
+ theta * gamma_increment
)
return simulated_paths

# Run the VG model simulation for 10 paths
n_simulations = 10
simulated_vg_paths = simulate_vg_paths(S0, mu, sigma, theta, nu,
T=1, n_simulations=n_simulations)


future_dates = pd.date_range(start=data.index[-1] +
pd.Timedelta(days=1), periods=simulated_vg_paths.shape[0], freq='B')


fig = go.Figure()


fig.add_trace(go.Scatter(
x=data.index,
y=data['Close'],
```

```
mode='lines',
name='Historical TSM Prices',
line=dict(color='orange', width=2)
))


for j in range(n_simulations):
    fig.add_trace(go.Scatter(
        x=future_dates,
        y=simulated_vg_paths[:, j],
        mode='lines',
        name=f'Simulated VG Path {j+1}',
        line=dict(width=1)
    ))


fig.update_layout(
title="TSM Historical Prices with Variance Gamma Model Simulated Paths",
xaxis_title="Date",
yaxis_title="Price (USD)",
template="plotly_white"
)

fig.show()
```

```python
end_prices = simulated_vg_paths[-1, :]
returns = (end_prices - S0) / S0

# Annualized expected return and volatility
annualized_mean_return = mean_return * 252
annualized_volatility = std_return * np.sqrt(252)

# Calculate skewness and kurtosis of returns
skewness = skew(returns)
excess_kurtosis = kurtosis(returns)

# VaR and CVaR at 95% confidence level
confidence_level = 0.05
VaR_95 = np.percentile(returns, confidence_level * 100)
CVaR_95 = returns[returns <= VaR_95].mean()

# Jump intensity and average jump size
jump_intensity = 1 / nu
average_jump_size = theta * nu

# Display summary statistics
print("Variance Gamma Model Simulation Statistics:")
print(f"Mean Annual Return: {mean_return:.4f}")
print(f"Standard Deviation of Annual Return: {std_return:.4f}")
print(f"Minimum Annual Return: {min_return:.4f}")
print(f"Maximum Annual Return: {max_return:.4f}")
print(f"Annualized Mean Return: {annualized_mean_return:.4f}")
```

```
print(f"Annualized Volatility: {annualized_volatility:.4f}")
print(f"Skewness: {skewness:.4f}")
print(f"Excess Kurtosis: {excess_kurtosis:.4f}")
print(f"95% VaR: {VaR_95:.4f}")
print(f"95% CVaR: {CVaR_95:.4f}")
print(f"Jump Intensity (Frequency of Jumps): {jump_intensity:.4f}")
print(f"Average Jump Size: {average_jump_size:.4f}")


import plotly.express as px
fig_hist = px.histogram(returns, nbins=50, title="Distribution of
Simulated Returns")
fig_hist.update_layout(xaxis_title="Return", yaxis_title="Frequency")
fig_hist.show()


dt = 1 / 252
avg_jump_sizes = np.mean(np.random.gamma(shape=dt/nu, scale=nu,
size=(simulated_vg_paths.shape[0], n_simulations)), axis=1)
fig_jumps = go.Figure()
fig_jumps.add_trace(go.Scatter(x=future_dates, y=avg_jump_sizes,
mode='lines', name="Average Jump Sizes"))
fig_jumps.update_layout(title="Average Jump Sizes Over Time",
xaxis_title="Date", yaxis_title="Average Jump Size")
fig_jumps.show()
```

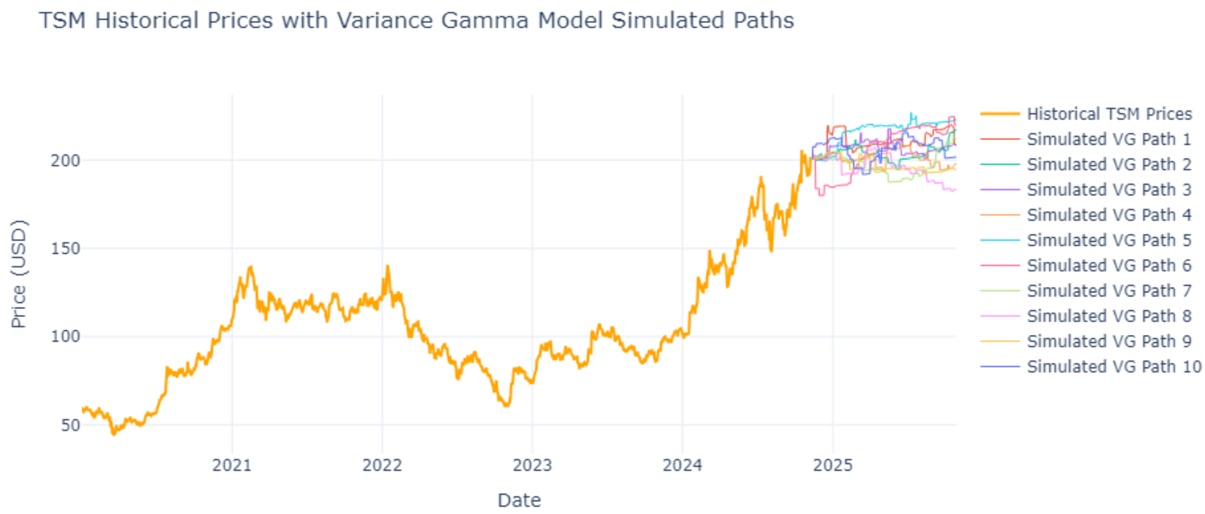

Figure 3.1

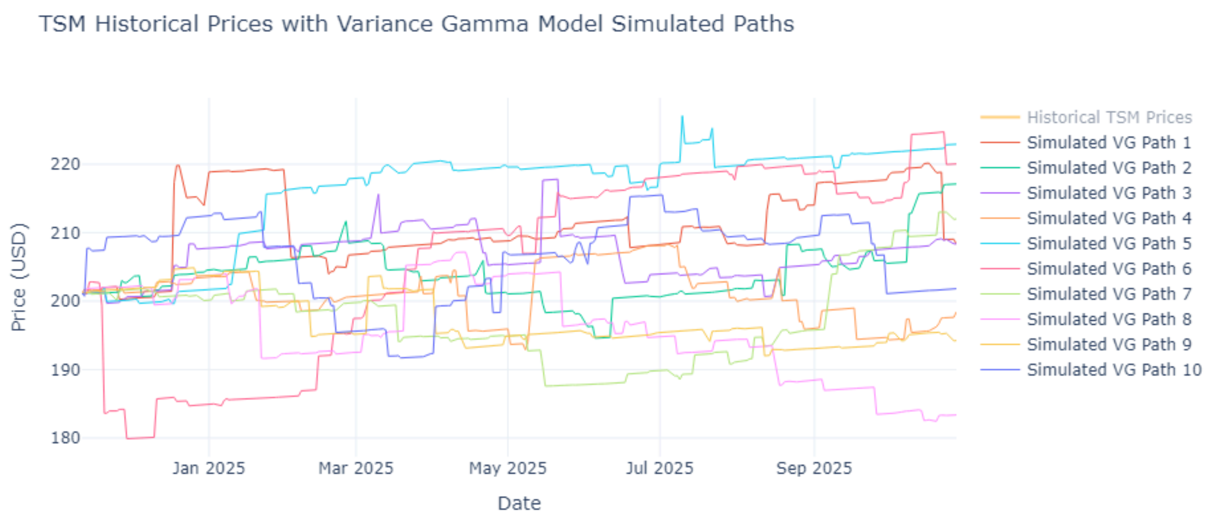

Figure 3.2

Variance Gamma Model Simulation Statistics:

Mean Annual Return: 0.0513

Standard Deviation of Annual Return: 0.1233

Minimum Annual Return: -0.1358

Maximum Annual Return: 0.2444

Annualized Mean Return: 12.9382

Annualized Volatility: 1.9574
Skewness: -0.4481
Excess Kurtosis: -0.6840
95% VaR: -0.0643
95% CVaR: -0.0886
Jump Intensity (Frequency of Jumps): 10.0000
Average Jump Size: 0.0010

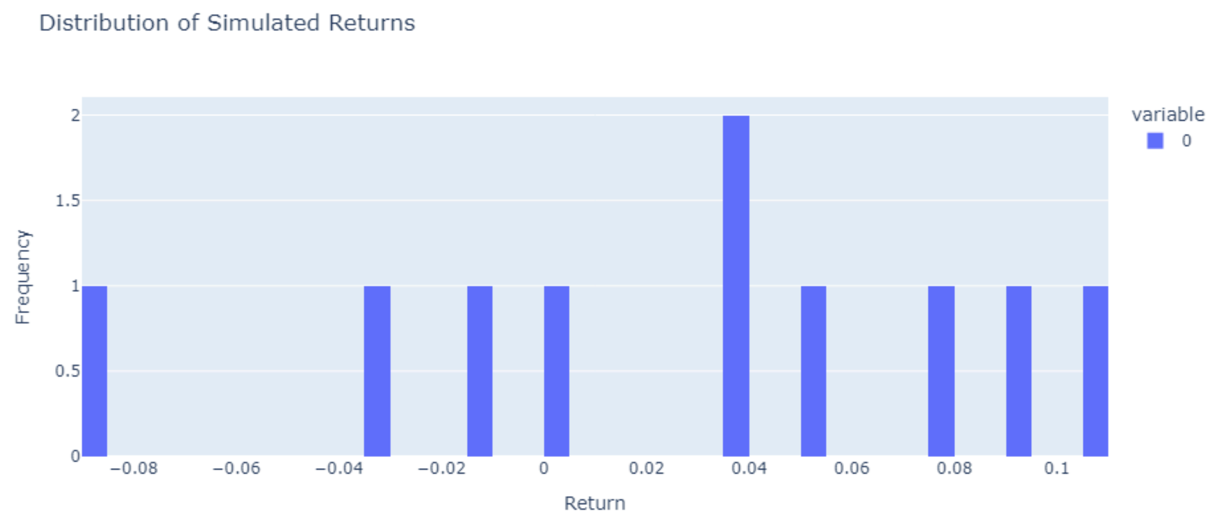

Figure 3.3

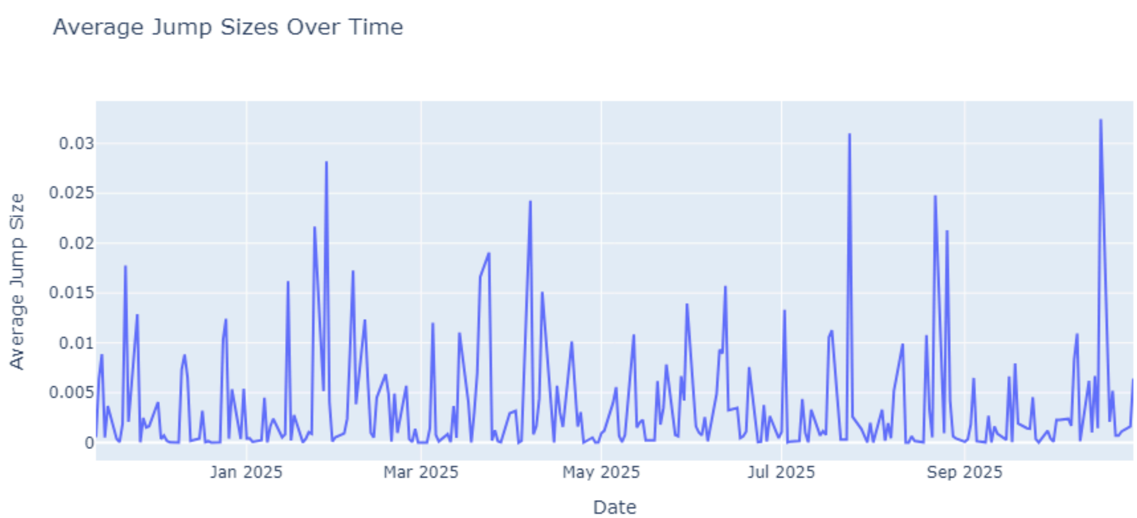

Figure 3.4

```
import numpy as np
import pandas as pd
from scipy.stats import skew, kurtosis, gamma
from scipy.optimize import minimize
import plotly.graph_objects as go


np.random.seed(42)


data = pd.read_csv("TSM_daily_data_2020_2024.csv",
index_col="Date", parse_dates=True)
data['Returns'] = data['Close'].pct_change()


data = data.dropna(subset=['Returns'])

# Estimate parameters from historical data
mu = data['Returns'].mean() * 252  # Annualized drift
sigma = data['Returns'].std() * np.sqrt(252)  # Annualized volatility
theta = skew(data['Returns'])  # Estimate for jump scale (based on
skew)
nu = kurtosis(data['Returns'])   # Excess kurtosis as jump shape
approximation (nu)

S0 = data['Close'].iloc[-1]  # Latest price as initial asset price
```

```
# Check for valid 'nu' value to prevent shape < 0
if nu <= 0:
print("Warning: 'nu' is less than or equal to zero. Adjusting nu to a small positive value.")
nu = 0.1

# Optimization function to minimize for the VG parameters
def negative_log_likelihood(params, returns):
mu, sigma, theta, nu = params


if nu <= 0:
return np.inf

N = len(returns)
likelihood = 0

for i in range(N):
# Calculate the jump component
jump_component = (returns[i] - mu) / sigma
if jump_component <= 0:
continue

# Calculate likelihood for Gamma distribution component
likelihood += np.log(gamma.pdf(jump_component, a=nu, scale=theta))
```

```
return -likelihood

# Minimize negative log-likelihood to estimate optimal parameters
initial_guess = [mu, sigma, theta, nu]
result = minimize(negative_log_likelihood, initial_guess,
args=(data['Returns'].values,))

# Check if the optimization converged
if not result.success:
print("Optimization failed:", result.message)
else:
mu_opt, sigma_opt, theta_opt, nu_opt = result.x

# Define the VG model simulation function with the optimized
parameters
def simulate_vg_paths(S0, mu, sigma, theta, nu, T=1, dt=1/252,
n_simulations=10):
N = int(T / dt)
simulated_paths = np.zeros((N, n_simulations), dtype=np.float64)

simulated_paths[0, :] = S0

for i in range(1, N):
if nu <= 0:
raise ValueError("nu must be positive.")
```

```
# Generate gamma increments for jumps
gamma_increment = np.random.gamma(shape=dt/nu, scale=nu,
size=n_simulations).astype(np.float64)

# Generate Brownian motion increments
brownian_increment = np.random.normal(0,
np.sqrt(gamma_increment), n_simulations).astype(np.float64)

# Update price paths using the VG model formula
simulated_paths[i, :] = simulated_paths[i-1, :] * np.exp(
(mu - 0.5 * sigma**2) * dt + sigma * brownian_increment +
theta * gamma_increment
)

return simulated_paths

# Simulate paths with the optimized parameters
n_simulations = 10
simulated_vg_paths = simulate_vg_paths(S0, mu_opt, sigma_opt,
theta_opt, nu_opt, T=1, n_simulations=n_simulations)


future_dates = pd.date_range(start=data.index[-1] +
pd.Timedelta(days=1), periods=simulated_vg_paths.shape[0], freq='B')


end_prices = simulated_vg_paths[-1, :]
```

```python
returns = (end_prices - S0) / S0


mean_return = np.mean(returns)
std_return = np.std(returns)
min_return = np.min(returns)
max_return = np.max(returns)
annualized_mean_return = (mean_return) * 252
annualized_volatility = std_return * np.sqrt(252)
skewness = skew(returns)
excess_kurtosis = kurtosis(returns)
VaR_95 = np.percentile(returns, 5)
CVaR_95 = returns[returns <= VaR_95].mean()
jump_intensity = 1 / nu_opt
average_jump_size = theta_opt

# Display statistics
print("Variance Gamma Model Simulation Statistics:")
print(f"Mean Annual Return: {annualized_mean_return:.4f}")
print(f"Annualized Volatility: {annualized_volatility:.4f}")
print(f"Skewness: {skewness:.4f}")
print(f"Excess Kurtosis: {excess_kurtosis:.4f}")
print(f"95% VaR: {VaR_95:.4f}")
print(f"95% CVaR: {CVaR_95:.4f}")
print(f"Jump Intensity: {jump_intensity:.4f}")
print(f"Average Jump Size: {average_jump_size:.4f}")
```

```
fig = go.Figure()


fig.add_trace(go.Scatter(
    x=data.index,
    y=data['Close'],
    mode='lines',
    name='Historical TSM Prices',
    line=dict(color='orange', width=2)
))


for j in range(n_simulations):
    fig.add_trace(go.Scatter(
        x=future_dates,
        y=simulated_vg_paths[:, j],
        mode='lines',
        name=f'Simulated VG Path {j+1}',
        line=dict(width=1)
    ))


fig.update_layout(
    title="TSM Historical Prices with Variance Gamma Model Simulated Paths",
    xaxis_title="Date",
```

```
yaxis_title="Price (USD)",
template="plotly_white"
)

fig.show()


fig_hist = px.histogram(returns, nbins=50, title="Distribution of Simulated Returns")
fig_hist.update_layout(xaxis_title="Return", yaxis_title="Frequency")
fig_hist.show()
```

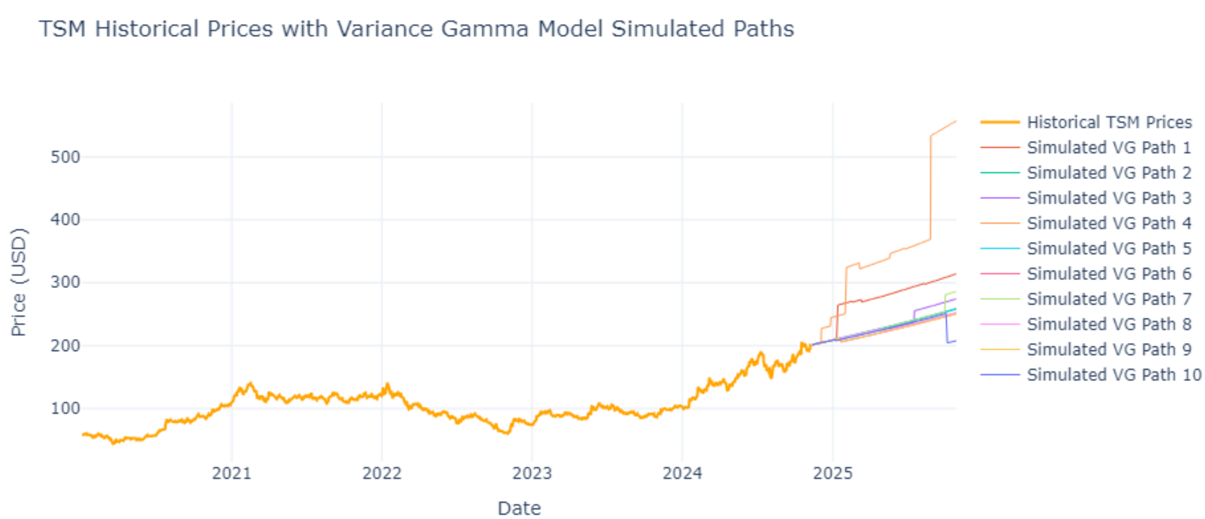

Figure 4.1

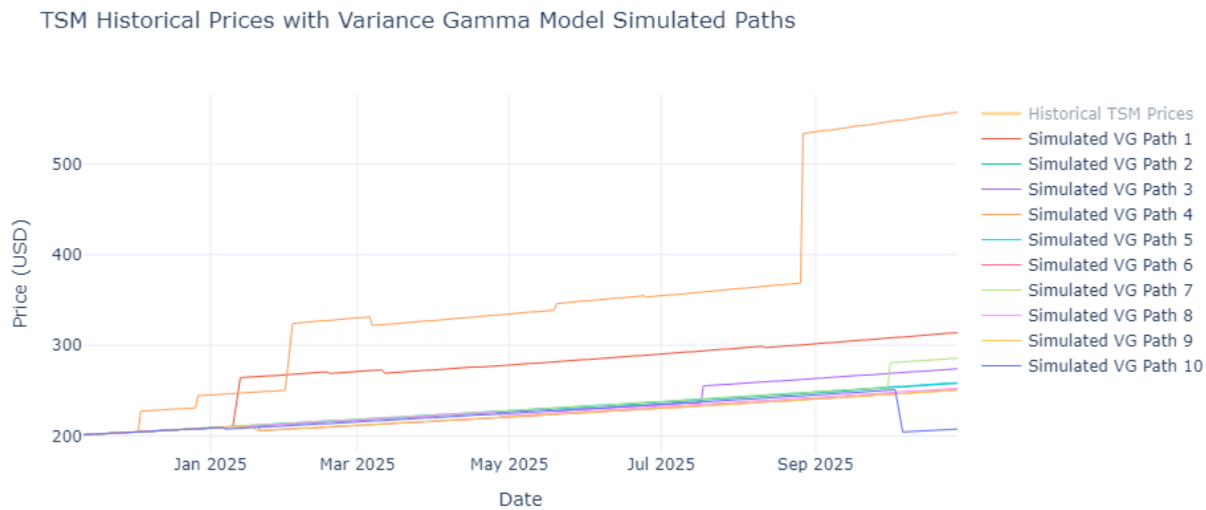

Figure 4.2

Variance Gamma Model Simulation Statistics:

Mean Annual Return: 112.4201

Annualized Volatility: 7.2985

Skewness: 2.2820

Excess Kurtosis: 3.9519

95% VaR: 0.1277

95% CVaR: 0.0316

Jump Intensity: 0.3349

Average Jump Size: 0.3595

Estimating Model Parameters

```
mu = data['Returns'].mean() * 252  # Annualized drift
sigma = data['Returns'].std() * np.sqrt(252)  # Annualized volatility
theta = skew(data['Returns'])  # Jump scale estimate
nu = kurtosis(data['Returns'])   # Jump shape estimate
```

- **Annualized drift (`mu`)**: Calculated as the average daily return multiplied by 252 (trading days in a year).
- **Annualized volatility (`sigma`)**: Calculated from daily returns' standard deviation, scaled by √252.
- **Jump scale (`theta`)** and **jump shape (`nu`)**: Estimates based on skewness and kurtosis, capturing asymmetry and fat tails in returns.

Parameter Optimization

To optimize the VG model parameters, we minimize the negative log-likelihood of the returns using initial guesses from historical data:

```
def negative_log_likelihood(params, returns):
    mu, sigma, theta, nu = params
```

```
if nu <= 0:
    return np.inf

N = len(returns)
likelihood = 0

for i in range(N):
    jump_component = (returns[i] - mu) / sigma
    if jump_component <= 0:
        continue

    likelihood += np.log(gamma.pdf(jump_component, a=nu, scale=theta))

return -likelihood

# Minimize negative log-likelihood to estimate optimal parameters
initial_guess = [mu, sigma, theta, nu]
result = minimize(negative_log_likelihood, initial_guess, args=(data['Returns'].values,))
```

This function calculates the likelihood of observed returns given the parameters `mu`, `sigma`, `theta`, and `nu`. Minimizing it provides the best-fitting VG model parameters for TSM's returns.

Simulating Future Price Paths

With optimized parameters, we simulate potential future price paths for TSM over one year:

```
def simulate_vg_paths(S0, mu, sigma, theta, nu, T=1, dt=1/252, n_simulations=10):
    N = int(T / dt)
    simulated_paths = np.zeros((N, n_simulations), dtype=np.float64)
    simulated_paths[0, :] = S0

    for i in range(1, N):
        gamma_increment = np.random.gamma(shape=dt/nu, scale=nu, size=n_simulations).astype(np.float64)
        brownian_increment = np.random.normal(0, np.sqrt(gamma_increment), n_simulations).astype(np.float64)

        simulated_paths[i, :] = simulated_paths[i-1, :] * np.exp(
```

```
(mu - 0.5 * sigma**2) * dt +
sigma * brownian_increment + theta *
gamma_increment
)

return simulated_paths

# Simulate paths with optimized parameters
simulated_vg_paths = simulate_vg_paths(S0,
mu_opt, sigma_opt, theta_opt, nu_opt, T=1,
n_simulations=n_simulations)
```

The function generates price paths using a VG model that combines Brownian motion and Gamma jumps, resulting in paths with realistic characteristics, including jumps.

Statistical Analysis

Statistical analysis is widely used for uncovering trends and relationships in data, particularly to interpret asset behaviors, assess risks, and guide decision making. In this chapter, we focus on concepts such as correlation, cointegration, and probability to enhance our understanding of how markets behave and interact.

The Mapper Algorithm

The Mapper algorithm works by transforming high-dimensional data, such as asset prices over time, into a simpler, lower-dimensional structure—a graph. This graph represents clusters of data points with similar characteristics, which can then be analyzed to understand trends and potential future movements.

let us define the following:

- Let X={x1,x2,…,xn} be the sequence of asset price observations at time steps t1,t2,…,tn, where x_i, represents the price of the asset at time t_i.
- A **covering function** $f:X\rightarrow\mathbb{R}^k$ maps the high-dimensional price data to a feature space, where each $f(x_i)$ captures the relationship between price, volatility, and momentum.
- The data is then partitioned into overlapping intervals C1,C2,…,Cm where each cluster C_i represents a group of similar price behaviors (e.g., rising, falling, or stable price).

The Mapper algorithm creates a **graph** G, where each node C^i represents a cluster, and an edge (C^i, C^j) exists between two clusters if their overlap or topological relationship indicates potential connections between price states.

If an asset's current price belongs to a cluster that frequently transitions to high volatility or sharp price movements, the model can predict a similar future behavior.

Let P(t) denote the predicted price of the asset at time t, and let C(t) represent the cluster that the asset's price currently belongs to. The prediction equation can be written as:

$$P(t + 1) = f(C(t))\cdot\Delta P(C(t)) + \epsilon$$

Where:

- P(t+1)is the predicted price at time t+1.
- f(C(t)) represents the mapping function for the current cluster C(t), capturing its historical behavior and features (such as price momentum or volatility).
- ΔP(C(t))is the expected change in price for that cluster, derived from historical transitions between clusters.
- ϵ is a small error term representing random fluctuations or noise in the market.

Interpreting the Mapper for Price Movements:

By analyzing the **graph structure** of the Mapper output, it is possible to identify **patterns** of asset price movements, such as:

- **Bullish Clusters**: Clusters where prices have historically shown upward trends. Transitions into these clusters can signal potential price increases.
- **Bearish Clusters**: Clusters representing downward price trends. Prices entering these clusters may indicate a risk of decline.
- **Stable Regimes**: Periods where prices remain stable, providing insights into low volatility and risk periods.

By understanding the transitions between these clusters, we can predict potential future price trends and movements.

```
import pandas as pd
import numpy as np
import plotly.graph_objects as go
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
import networkx as nx

spx_data = pd.read_csv('^SPX_daily_data_2020_2024.csv')
spy_data = pd.read_csv('SPY_daily_data_2020_2024.csv')

spx_data['Date'] = pd.to_datetime(spx_data['Date'])
spy_data['Date'] = pd.to_datetime(spy_data['Date'])

spx_data = spx_data.sort_values(by='Date')
spy_data = spy_data.sort_values(by='Date')
```

```
spx_data = spx_data.dropna(subset=['Close'])
spy_data = spy_data.dropna(subset=['Close'])

merged_data = pd.merge(spx_data[['Date', 'Close']], spy_data[['Date',
'Close']], on='Date', suffixes=('_SPX', '_SPY'))

merged_data['Return_SPX'] = merged_data['Close_SPX'].pct_change()
merged_data['Return_SPY'] = merged_data['Close_SPY'].pct_change()

merged_data['Volatility_SPY'] =
merged_data['Return_SPY'].rolling(window=5).std()
merged_data['Volatility_SPX'] =
merged_data['Return_SPX'].rolling(window=5).std()

merged_data = merged_data.dropna()

scaler = StandardScaler()

scaled_data_spy = scaler.fit_transform(merged_data[['Return_SPY',
'Volatility_SPY']])
scaled_data_spx = scaler.fit_transform(merged_data[['Return_SPX',
'Volatility_SPX']])

kmeans_spy = KMeans(n_clusters=3, random_state=42)
merged_data['Cluster_SPY'] = kmeans_spy.fit_predict(scaled_data_spy)

kmeans_spx = KMeans(n_clusters=3, random_state=42)
```

```
merged_data['Cluster_SPX'] = kmeans_spx.fit_predict(scaled_data_spx)

# Create 3D network graph for SPY and ^SPX based on clustering
def create_network_graph_3d(cluster_column, return_column,
volatility_column, security_name):
# Build a correlation matrix for network edges
correlation_matrix = merged_data[[return_column,
volatility_column]].corr()

# Create network graph
G = nx.Graph()

# Add nodes for each data point
for i, row in merged_data.iterrows():
node_label = f'{security_name} Cluster {row[cluster_column]}
{row["Date"]}'
G.add_node(node_label, size=5, color=row[return_column])

# Add edges between consecutive nodes
for i in range(1, len(merged_data)):
source_node = f'{security_name} Cluster
{merged_data[cluster_column].iloc[i-1]} {merged_data["Date"].iloc[i-1]}'
target_node = f'{security_name} Cluster
{merged_data[cluster_column].iloc[i]} {merged_data["Date"].iloc[i]}'

# Add edge based on correlation value
```

```
weight = correlation_matrix.iloc[0, 1] if security_name == 'SPY' else
correlation_matrix.iloc[0, 1]
G.add_edge(source_node, target_node, weight=weight)

return G


G_spy = create_network_graph_3d('Cluster_SPY', 'Return_SPY',
'Volatility_SPY', 'SPY')
G_spx = create_network_graph_3d('Cluster_SPX', 'Return_SPX',
'Volatility_SPX', '^SPX')


pos_spy = nx.spring_layout(G_spy, dim=3)
edge_weights_spy = nx.get_edge_attributes(G_spy, 'weight')


edge_trace_spy = go.Scatter3d(
x=[], y=[], z=[], line=dict(width=0.5, color='#888'),
hoverinfo='none', mode='lines')


node_trace_spy = go.Scatter3d(
x=[], y=[], z=[], text=[], mode='markers',
hoverinfo='text', marker=dict(
showscale=True, colorscale='YlGnBu', size=5, colorbar=dict(
```

```
thickness=15, title='Node Connections', xanchor='left',
titleside='right')
))

for node in G_spy.nodes():
x0, y0, z0 = pos_spy[node]
node_trace_spy['x'] += tuple([x0])
node_trace_spy['y'] += tuple([y0])
node_trace_spy['z'] += tuple([z0])
node_trace_spy['text'] += tuple([f'{node}'])

for edge in G_spy.edges():
x0, y0, z0 = pos_spy[edge[0]]
x1, y1, z1 = pos_spy[edge[1]]
edge_trace_spy['x'] += tuple([x0, x1, None])
edge_trace_spy['y'] += tuple([y0, y1, None])
edge_trace_spy['z'] += tuple([z0, z1, None])

fig_spy = go.Figure(data=[edge_trace_spy, node_trace_spy],
layout=go.Layout(
title='SPY Network Graph of Price Movements',
titlefont_size=16,
showlegend=False,
hovermode='closest',
scene=dict(
xaxis=dict(showgrid=False, zeroline=False),
yaxis=dict(showgrid=False, zeroline=False),
```

```
zaxis=dict(showgrid=False, zeroline=False)
)
))

fig_spy.show()


pos_spx = nx.spring_layout(G_spx, dim=3)
edge_weights_spx = nx.get_edge_attributes(G_spx, 'weight')


edge_trace_spx = go.Scatter3d(
x=[], y=[], z=[], line=dict(width=0.5, color='#888'),
hoverinfo='none', mode='lines')


node_trace_spx = go.Scatter3d(
x=[], y=[], z=[], text=[], mode='markers',
hoverinfo='text', marker=dict(
showscale=True, colorscale='YlGnBu', size=5, colorbar=dict(
thickness=15, title='Node Connections', xanchor='left',
titleside='right')
))

for node in G_spx.nodes():
x0, y0, z0 = pos_spx[node]
node_trace_spx['x'] += tuple([x0])
```

```
node_trace_spx['y'] += tuple([y0])
node_trace_spx['z'] += tuple([z0])
node_trace_spx['text'] += tuple([f'{node}'])

for edge in G_spx.edges():
x0, y0, z0 = pos_spx[edge[0]]
x1, y1, z1 = pos_spx[edge[1]]
edge_trace_spx['x'] += tuple([x0, x1, None])
edge_trace_spx['y'] += tuple([y0, y1, None])
edge_trace_spx['z'] += tuple([z0, z1, None])

fig_spx = go.Figure(data=[edge_trace_spx, node_trace_spx],
layout=go.Layout(
title='SPX Network Graph of Price Movements',
titlefont_size=16,
showlegend=False,
hovermode='closest',
scene=dict(
xaxis=dict(showgrid=False, zeroline=False),
yaxis=dict(showgrid=False, zeroline=False),
zaxis=dict(showgrid=False, zeroline=False)
)
))

fig_spx.show()
```

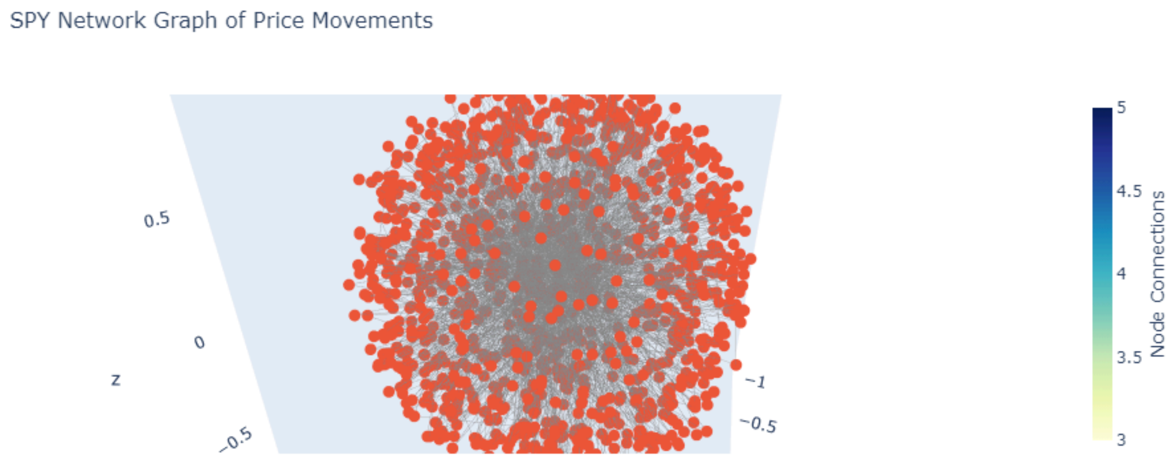

Figure 5.1

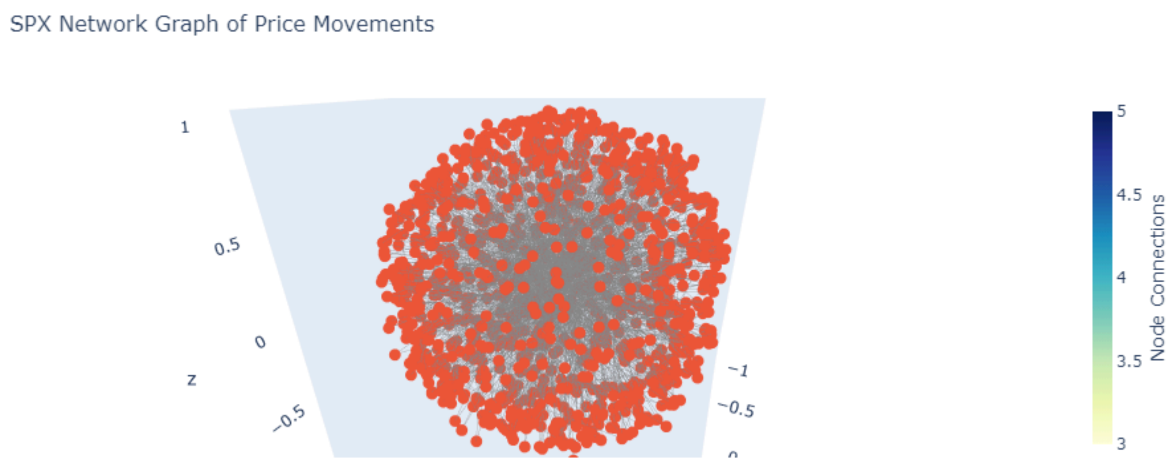

Figure 5.2

```
import pandas as pd
import numpy as np
import plotly.graph_objects as go
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
```

```
spx_data = pd.read_csv('^SPX_daily_data_2020_2024.csv')
spy_data = pd.read_csv('SPY_daily_data_2020_2024.csv')

spx_data['Date'] = pd.to_datetime(spx_data['Date'])
spy_data['Date'] = pd.to_datetime(spy_data['Date'])

spx_data = spx_data.sort_values(by='Date')
spy_data = spy_data.sort_values(by='Date')

spx_data = spx_data.dropna(subset=['Close'])
spy_data = spy_data.dropna(subset=['Close'])

merged_data = pd.merge(spx_data[['Date', 'Close']], spy_data[['Date',
'Close']], on='Date', suffixes=('_SPX', '_SPY'))

# Calculate daily returns and rolling volatility
merged_data['Return_SPX'] = merged_data['Close_SPX'].pct_change()
merged_data['Return_SPY'] = merged_data['Close_SPY'].pct_change()

# Calculate rolling volatility
merged_data['Volatility_SPY'] =
merged_data['Return_SPY'].rolling(window=5).std()
merged_data['Volatility_SPX'] =
merged_data['Return_SPX'].rolling(window=5).std()
```

```
# Drop rows with missing values
merged_data = merged_data.dropna()

# Normalize data and apply KMeans clustering
scaler = StandardScaler()
scaled_data = scaler.fit_transform(merged_data[['Return_SPY',
'Volatility_SPY', 'Return_SPX', 'Volatility_SPX']])

# Apply KMeans clustering to group the data into price behavior clusters
kmeans = KMeans(n_clusters=3, random_state=42)
merged_data['Cluster'] = kmeans.fit_predict(scaled_data)

# Predict the price movement using cluster behavior (Price change per
cluster)
merged_data['Price_Change_SPY'] = merged_data['Close_SPY'].diff()
merged_data['Price_Change_SPX'] = merged_data['Close_SPX'].diff()

# Calculate mean price change per cluster for SPY and ^SPX
cluster_changes_spy =
merged_data.groupby('Cluster')['Price_Change_SPY'].mean()
cluster_changes_spx =
merged_data.groupby('Cluster')['Price_Change_SPX'].mean()

# Predict the next 252 trading days' prices using the historical average price
changes
def predict_price(current_cluster, last_price, cluster_changes):
```

```
expected_change = cluster_changes.get(current_cluster, 0)

predicted_price = last_price + expected_change
return predicted_price


predicted_prices_spy = []
predicted_prices_spx = []


last_spy_price = merged_data['Close_SPY'].iloc[-1]
last_spx_price = merged_data['Close_SPX'].iloc[-1]

for i in range(252):
current_cluster_spy = merged_data['Cluster'].iloc[-1]
current_cluster_spx = merged_data['Cluster'].iloc[-1]

predicted_spy = predict_price(current_cluster_spy, last_spy_price,
cluster_changes_spy)
predicted_spx = predict_price(current_cluster_spx, last_spx_price,
cluster_changes_spx)

predicted_prices_spy.append(predicted_spy)
predicted_prices_spx.append(predicted_spx)

last_spy_price = predicted_spy
last_spx_price = predicted_spx
```

```
fig = go.Figure()

fig.add_trace(go.Scatter(x=merged_data['Date'],
y=merged_data['Close_SPY'], mode='lines', name='Actual SPY Price',
line=dict(color='blue')))

future_dates = pd.date_range(start=merged_data['Date'].iloc[-1],
periods=252, freq='B')[1:]
fig.add_trace(go.Scatter(x=future_dates, y=predicted_prices_spy,
mode='lines', name='Predicted SPY Price', line=dict(color='red',
dash='dash')))

fig.add_trace(go.Scatter(x=merged_data['Date'],
y=merged_data['Close_SPX'], mode='lines', name='Actual ^SPX Price',
line=dict(color='green')))

fig.add_trace(go.Scatter(x=future_dates, y=predicted_prices_spx,
mode='lines', name='Predicted ^SPX Price', line=dict(color='orange',
dash='dash')))
```

```
fig.update_layout(
title='Actual vs Predicted Prices for SPY and ^SPX',
xaxis_title='Date',
yaxis_title='Price',
template='plotly_white',
legend_title='Legend'
)

fig.show()



correlation_spy_spx = merged_data[['Return_SPY',
'Return_SPX']].corr().iloc[0, 1]



print(f"Correlation between SPY and SPX returns:
{correlation_spy_spx:.4f}")
```

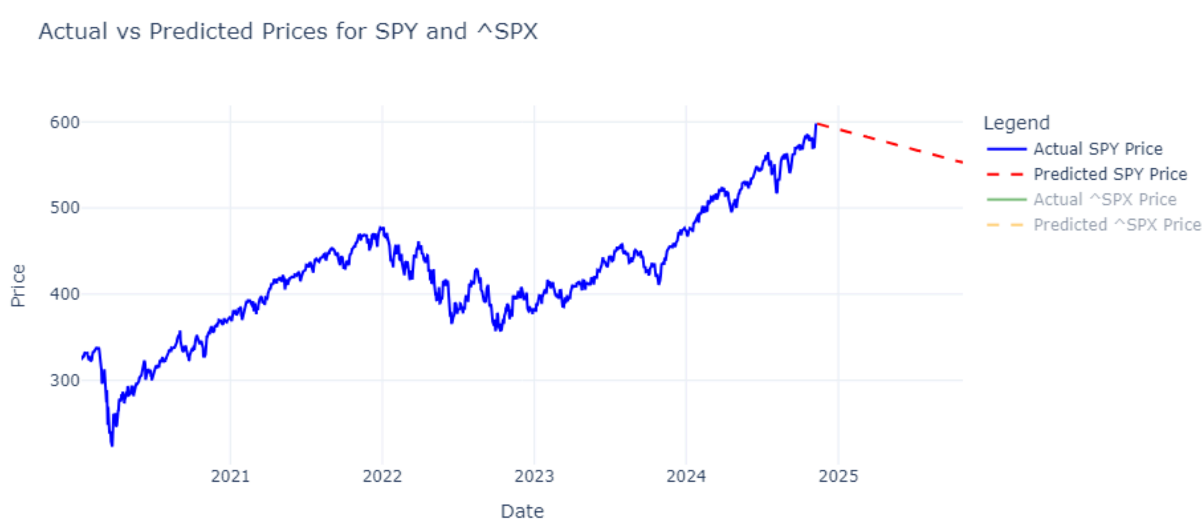

Figure 5.3

Figure 5.4

Correlation between SPY and SPX returns: 0.9981

Calculating Daily Returns and Rolling Volatility

```
merged_data['Return_SPX'] =
merged_data['Close_SPX'].pct_change()
merged_data['Return_SPY'] =
merged_data['Close_SPY'].pct_change()

merged_data['Volatility_SPY'] =
merged_data['Return_SPY'].rolling(window=5).s
td()
merged_data['Volatility_SPX'] =
merged_data['Return_SPX'].rolling(window=5).s
td()
```

- **Daily returns** for each index are calculated as percentage changes.
- **Rolling volatility** is calculated using a 5-day rolling window of the standard deviation of daily returns.

Data Normalization and Clustering

```
scaler = StandardScaler()
scaled_data = scaler.fit_transform(merged_data[['Return_SPY', 'Volatility_SPY', 'Return_SPX', 'Volatility_SPX']])

kmeans = KMeans(n_clusters=3, random_state=42)
merged_data['Cluster'] = kmeans.fit_predict(scaled_data)
```

- Normalize the return and volatility data using `StandardScaler` to bring features to a similar scale.
- Use **KMeans clustering** to identify patterns in price behavior and group data into 3 clusters.

Analyzing Cluster-Based Price Changes

```
merged_data['Price_Change_SPY'] =
merged_data['Close_SPY'].diff()
merged_data['Price_Change_SPX'] =
merged_data['Close_SPX'].diff()

cluster_changes_spy =
merged_data.groupby('Cluster')['Price_Change_
SPY'].mean()
cluster_changes_spx =
merged_data.groupby('Cluster')['Price_Change_
SPX'].mean()
```

- Calculate **daily price changes** for SPY and SPX.
- Determine the **average price change per cluster** for both indices, used to model future price changes based on cluster behavior.

Predicting Future Prices

```
def predict_price(current_cluster,
last_price, cluster_changes):
    expected_change =
cluster_changes.get(current_cluster, 0)
    predicted_price = last_price +
expected_change
    return predicted_price
```

- Define a function `predict_price` to calculate the next day's price based on the cluster's average price change.

```
predicted_prices_spy = []
predicted_prices_spx = []

last_spy_price =
merged_data['Close_SPY'].iloc[-1]
last_spx_price =
merged_data['Close_SPX'].iloc[-1]

for i in range(252):
    current_cluster_spy =
merged_data['Cluster'].iloc[-1]
    current_cluster_spx =
merged_data['Cluster'].iloc[-1]

    predicted_spy =
predict_price(current_cluster_spy,
last_spy_price, cluster_changes_spy)
    predicted_spx =
predict_price(current_cluster_spx,
last_spx_price, cluster_changes_spx)
```

```
predicted_prices_spy.append(predicted_spy)

predicted_prices_spx.append(predicted_spx)

        last_spy_price = predicted_spy
        last_spx_price = predicted_spx
```

- Predict prices for the next 252 trading days by iteratively applying the cluster-based price change model, updating the price and cluster with each step.

Visualization of Actual and Predicted Prices

```
fig = go.Figure()

fig.add_trace(go.Scatter(x=merged_data['Date'
], y=merged_data['Close_SPY'], mode='lines',
name='Actual SPY Price',
line=dict(color='blue')))

future_dates =
pd.date_range(start=merged_data['Date'].iloc[
-1], periods=252, freq='B')[1:]
```

```
fig.add_trace(go.Scatter(x=future_dates,
y=predicted_prices_spy, mode='lines',
name='Predicted SPY Price',
line=dict(color='red', dash='dash')))

fig.add_trace(go.Scatter(x=merged_data['Date'
], y=merged_data['Close_SPX'], mode='lines',
name='Actual ^SPX Price',
line=dict(color='green')))
fig.add_trace(go.Scatter(x=future_dates,
y=predicted_prices_spx, mode='lines',
name='Predicted ^SPX Price',
line=dict(color='orange', dash='dash')))

fig.update_layout(
title='Actual vs Predicted Prices for SPY
and ^SPX',
xaxis_title='Date',
yaxis_title='Price',
template='plotly_white',
legend_title='Legend'
)

fig.show()
```

- Plot both **actual and predicted prices** for SPY and SPX, with actual data in solid lines and predicted data in dashed lines, to visualize the forecasted price trends.

Correlation Analysis

```
correlation_spy_spx =
merged_data[['Return_SPY',
'Return_SPX']].corr().iloc[0, 1]
print(f"Correlation between SPY and SPX
returns: {correlation_spy_spx:.4f}")
```

- Calculate and print the **correlation** between SPY and SPX returns, indicating how closely these indices' movements align.

The Langevin Equation & The Fokker-Planck Equation

The Langevin Equation: Focuses on the evolution of individual price paths by modeling the underlying forces driving the asset price.

The Fokker-Planck Equation: Describes how the probability distribution of these prices evolves over time, offering a broader statistical view.

The **Langevin equation** is used to model the random, often noisy processes that drive asset prices. It captures both deterministic (predictable) and stochastic (random) forces impacting the asset's movement.

General Form:

$$dt/dS = \mu S + \sigma\eta(t)$$

where:

- S(t): Asset price at time t.
- μS: Drift term, representing the deterministic part of the asset's movement.
- σ: Volatility factor, scaling the impact of random fluctuations.
- η(t): Random term (white noise), capturing the unpredictable forces affecting the asset.

Interpretation:

- The Langevin equation models individual asset price paths by combining growth trends (μ) with randomness (ση(t)), making it suitable for simulations of future price paths under specific conditions.

The **Fokker-Planck equation** provides a statistical view of how the probability distribution of an asset's price evolves over time, rather than focusing on individual price paths.

General Form:

$$\partial t/\partial P(S,t) =- \partial S\partial[\mu SP(S,t)] + (1/2(\partial S^2/2\partial^2))\ [\sigma^2 S^2 P(S,t)]$$

where:

- P(S,t): Probability density function of the asset price at time t.
- μ: Drift term, representing the deterministic trend.
- σ: Volatility term, accounting for the variance in asset prices.

Interpretation:

- The Fokker-Planck equation describes how the distribution of possible prices evolves, focusing on the likelihood of various outcomes rather than specific paths. It's ideal for assessing the broader risk and uncertainty in an asset's price.

```
import numpy as np

import pandas as pd

import plotly.graph_objects as go

from scipy.stats import kurtosis, skew




data = pd.read_csv('ES=F_daily_data_2020_2024.csv')

data['Date'] = pd.to_datetime(data['Date'])

data = data.sort_values(by='Date').dropna(subset=['Close'])

data.set_index('Date', inplace=True)
```

```
data['Return'] = data['Close'].pct_change().dropna()

mean_return = data['Return'].mean()
variance_return = data['Return'].var()
std_dev_return = data['Return'].std()
skewness_return = skew(data['Return'])
kurtosis_return = kurtosis(data['Return'])
skewness_Close = skew(data['Close'])
kurtosis_Close = kurtosis(data['Close'])

mean_open = data['Open'].mean()
mean_high = data['High'].mean()
mean_low = data['Low'].mean()
mean_close = data['Close'].mean()
```

```
cleaned_returns = data['Return'].dropna()

skewness_return = skew(cleaned_returns)
kurtosis_return = kurtosis(cleaned_returns)

fig = go.Figure()

fig.add_trace(go.Scatter(x=data.index, y=data['Close'], mode='lines',
    name='ES=F Actual Close Price', line=dict(color='blue')))

# Langevin Parameters
mu = mean_return  # Drift term
sigma = std_dev_return  # Volatility
prices = [data['Close'].iloc[0]]
timesteps = len(data) - 1
```

```python
np.random.seed(0)

random_noise = np.random.normal(0, 0.5, timesteps)


# Langevin Simulation for observed period

for i in range(timesteps):

dS = mu * prices[-1] + sigma * prices[-1] * random_noise[i]

prices.append(prices[-1] + dS)




fig.add_trace(go.Scatter(x=data.index, y=prices, mode='lines',
name='Langevin Modeled Price', line=dict(color='orange')))




forecast_prices = [prices[-1]]

forecast_returns = np.random.normal(mean_return, std_dev_return, 252)

for ret in forecast_returns:

next_price = forecast_prices[-1] * (1 + ret)

forecast_prices.append(next_price)
```

```
forecast_dates = pd.date_range(start=data.index[-1], periods=252,
freq='B')[1:]

fig.add_trace(go.Scatter(x=forecast_dates, y=forecast_prices[:-1],
mode='lines', name='252-Day Forecast', line=dict(color='red', dash='dash')))



fig.update_layout(

title="ES=F Price with Langevin Model - Statistical Analysis",

xaxis_title="Date",

yaxis_title="Price",

template="plotly_white",

legend_title="Legend",

margin=dict(r=200)

)


fig.show()


fig_hist = go.Figure()
```

```python
fig_hist.add_trace(go.Histogram(x=data['Open'], nbinsx=50, histnorm='probability', name='Open Price', marker_color='purple', opacity=0.6))

fig_hist.add_trace(go.Histogram(x=data['High'], nbinsx=50, histnorm='probability', name='High Price', marker_color='green', opacity=0.6))

fig_hist.add_trace(go.Histogram(x=data['Low'], nbinsx=50, histnorm='probability', name='Low Price', marker_color='blue', opacity=0.6))

fig_hist.add_trace(go.Histogram(x=data['Close'], nbinsx=50, histnorm='probability', name='Close Price', marker_color='orange', opacity=0.6))


fig_hist.update_layout(

title="OHLC Price Distributions",

xaxis_title="Price",

yaxis_title="Probability",

barmode='overlay',

template="plotly_white",

legend_title="Price Type"

)
```

```python
fig_hist.show()


print("Statistical Summary:")
print(f"Mean Return: {mean_return:.4f}")
print(f"Variance of Return: {variance_return:.4f}")
print(f"Standard Deviation of Return: {std_dev_return:.4f}")
print(f"Skewness of Return: {skewness_return:.4f}")
print(f"Kurtosis of Return: {kurtosis_return:.4f}")
print(f"Mean Open Price: {mean_open:.4f}")
print(f"Mean High Price: {mean_high:.4f}")
print(f"Mean Low Price: {mean_low:.4f}")
print(f"Mean Close Price: {mean_close:.4f}")
```

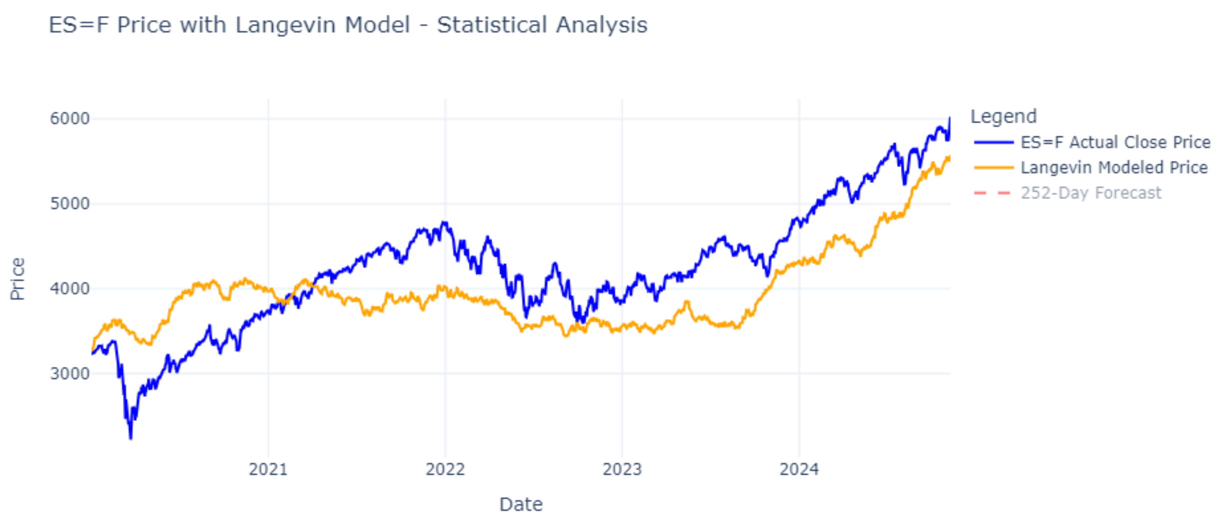

Figure 5.5

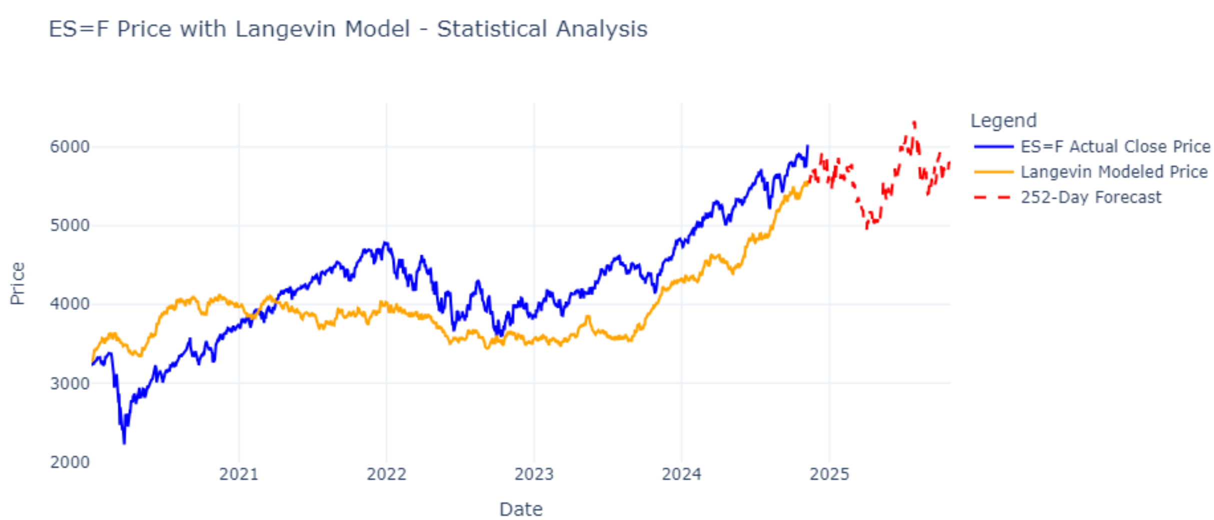

Figure 5.6

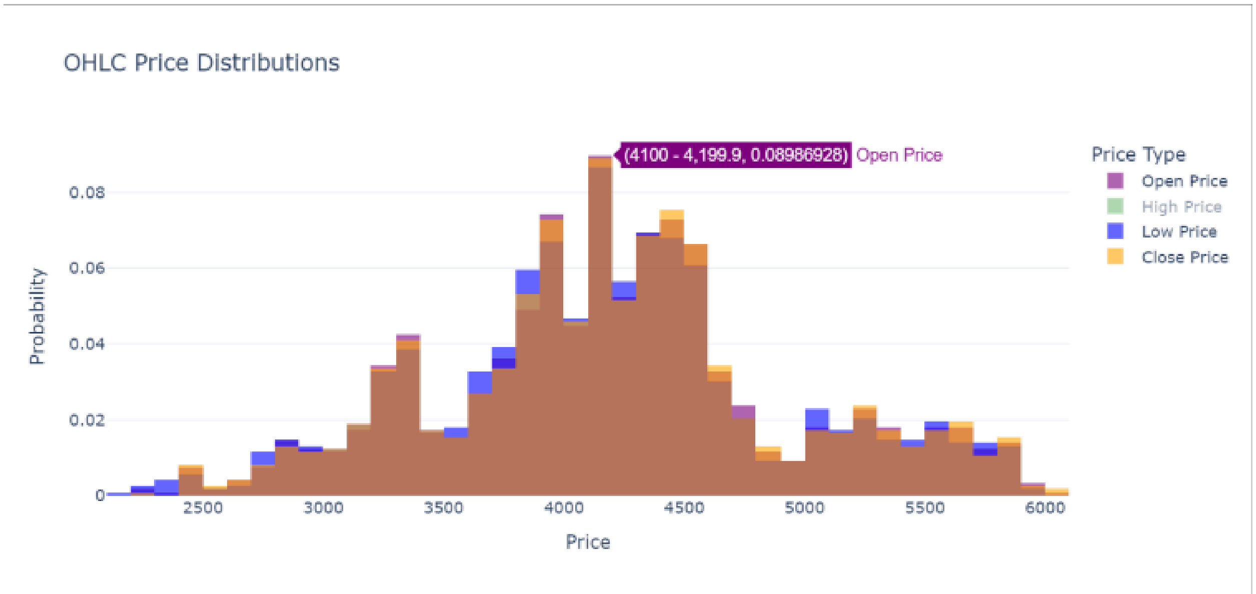

Figure 5.7

Statistical Summary:
Mean Return: 0.0006
Variance of Return: 0.0002
Standard Deviation of Return: 0.0134
Skewness of Return: -0.5793
Kurtosis of Return: 12.3423
Mean Open Price: 4215.8995
Mean High Price: 4248.7661
Mean Low Price: 4182.2107
Mean Close Price: 4218.1217

Statistical Analysis of Returns

```
mean_return = data['Return'].mean()
variance_return = data['Return'].var()
std_dev_return = data['Return'].std()
```

```
skewness_return = skew(data['Return'])
kurtosis_return = kurtosis(data['Return'])
skewness_Close = skew(data['Close'])
kurtosis_Close = kurtosis(data['Close'])
```

- **Mean Return:** Average daily return, serving as the drift parameter in the Langevin model.
- **Variance and Standard Deviation:** Variance measures return variability, while standard deviation is volatility.
- **Skewness and Kurtosis:** Measures asymmetry (skewness) and "peakedness" (kurtosis) of return distribution. Similar statistics are also calculated for `Close` prices.

Langevin Parameters and Simulation

```
mu = mean_return
sigma = std_dev_return
prices = [data['Close'].iloc[0]]
timesteps = len(data) - 1
np.random.seed(0)
random_noise = np.random.normal(0, 0.5,
timesteps)
```

- **`mu` and `sigma`**: Set up drift (mean return) and volatility (standard deviation of returns).

- **Initial Price and Timesteps**: Langevin simulation starts from the first `Close` price and proceeds for the number of time steps (days).
- **Random Noise**: Random values represent the randomness in price changes in the Langevin process.

Langevin Simulation

```
for i in range(timesteps):
dS = mu * prices[-1] + sigma * prices[-1]
* random_noise[i]
prices.append(prices[-1] + dS)
fig.add_trace(go.Scatter(x=data.index,
y=prices, mode='lines', name='Langevin
Modeled Price', line=dict(color='orange')))
```

- **Langevin Price Simulation**: Models price evolution over observed data using a stochastic process.
- **Plotting Modeled Prices**: Plots the simulated prices alongside actual prices for comparison.

Forecasting Next 252 Days

```
forecast_prices = [prices[-1]]
forecast_returns =
np.random.normal(mean_return, std_dev_return,
252)
```

```
for ret in forecast_returns:
    next_price = forecast_prices[-1] * (1 +
ret)
    forecast_prices.append(next_price)
forecast_dates =
pd.date_range(start=data.index[-1],
periods=252, freq='B')[1:]
fig.add_trace(go.Scatter(x=forecast_dates,
y=forecast_prices[:-1], mode='lines',
name='252-Day Forecast',
line=dict(color='red', dash='dash')))
```

- **Forecast Prices**: Uses the last observed price and 252 days of returns to create a projected price path.
- **Forecast Dates**: Adds new dates for each forecast day.
- **Plot Forecast**: Adds the forecast line in red to the original plot.

Histogram for OHLC Price Distributions

```
fig_hist = go.Figure()
fig_hist.add_trace(go.Histogram(x=data['Open'
], nbinsx=50, histnorm='probability',
name='Open Price', marker_color='purple',
opacity=0.6))
fig_hist.add_trace(go.Histogram(x=data['High'
], nbinsx=50, histnorm='probability',
```

```
name='High Price', marker_color='green',
opacity=0.6))
fig_hist.add_trace(go.Histogram(x=data['Low']
, nbinsx=50, histnorm='probability',
name='Low Price', marker_color='blue',
opacity=0.6))
fig_hist.add_trace(go.Histogram(x=data['Close
'], nbinsx=50, histnorm='probability',
name='Close Price', marker_color='orange',
opacity=0.6))
fig_hist.update_layout(
title="OHLC Price Distributions",
xaxis_title="Price",
yaxis_title="Probability",
barmode='overlay',
template="plotly_white",
legend_title="Price Type"
)
fig_hist.show()
```

- **Plotting Distributions**: Creates a histogram for each price type (Open, High, Low, Close), overlayed to compare their distributions.

Reversion in Stationary Random Processes

A stationary random process is one where statistical properties such as mean, variance, and autocorrelation remain constant over time. Reversion in such processes is characterized by:

- The tendency of the process to fluctuate around a long-term mean.
- The decay of deviations from the mean over time, typically modeled using an Ornstein-Uhlenbeck process or similar mean-reverting frameworks.

Key characteristics of reversion in stationary random processes:

- **Mean-reverting tendency**: Any deviation from the mean is followed by a movement back toward it.
- **Autocorrelation structure**: The degree of persistence or decay in deviations.

Frequency of Reversionary Moves

measures how often a time series crosses or approaches its long-term mean. It's closely tied to:

- The **volatility** of the process.
- The **speed of mean reversion** (θ in the OU model).

The **frequency** of reversionary moves can be estimated by examining the **crossing rate**:

- How often the process X^t crosses its mean μ.
- A process with higher volatility (σ) or slower mean reversion (θ) will have fewer reversionary moves.

Amount of Reversion

The **amount of reversion** quantifies how far the process moves back toward its mean after a deviation. This can be measured in terms of:

- The proportion of deviation corrected over a fixed period (Δt).
- The magnitude of the reversionary movement, influenced by θ (speed of mean reversion).

The **amount of reversion** quantifies how far the process moves back toward its mean after a deviation. This can be measured in terms of:

- The proportion of deviation corrected over a fixed period (Δt).
- The magnitude of the reversionary movement, influenced by θ (speed of mean reversion).

The expected reversionary movement over a small time increment (Δt) is:

$$\Delta Xt=\theta(\mu-Xt)\Delta t$$

Pure Reversion

Pure reversion refers to the theoretical movement of a process toward its mean in a perfectly stationary and deterministic system. It is driven purely by the mean-reverting force and devoid of external noise or randomness.

In the context of the OU process, pure reversion is modeled by:

$$dX_t=\theta(\mu-X_t)dt$$

Key assumption: No stochastic term ($\sigma = 0$).
The process reverts to μ smoothly and deterministically, with no perturbations.

Revealed Reversion

Revealed reversion deals with the observed or realized reversion in a system that includes both deterministic and stochastic components. Unlike pure reversion, revealed reversion accounts for:

- **Noise**: Random fluctuations that mask or distort the mean-reverting tendency.
- **Time horizon**: Reversion becomes more apparent over longer time frames as the mean-reverting force dominates noise.

Key Differences from Pure Reversion:

- Revealed reversion is observable in real-world systems and incorporates both $\theta(\mu-X_t)$ and σdW_t.
- It reflects the balance between reversionary forces and random disturbances.

To approximate the values of θ (speed of reversion) and σ (noise intensity) from real data, as well as to account for reversionary moves across percentiles (75 → 50 → 25 → 50 → 75), we'll calculate these parameters based on statistical properties of the data.

Theta (θ):

- Represents the speed of reversion toward the mean. It can be estimated using the concept of **mean reversion strength**:

$$\theta = Deviation\ from\ the\ Mean\ /\ Mean\ Reversion\ Amount$$

This requires calculating deviations from the mean and how much the price reverts over time.

Sigma (σ):

- Represents the intensity of random noise in the process. It can be approximated as the standard deviation of the residuals:

$$\sigma = std(\Delta Xt - \theta(\mu - Xt))$$

Where ΔXt is the price change and $(\mu - Xt)$ is the deviation from the mean.

Reversionary Moves Across Percentiles:

- Calculate price percentiles (25th, 50th, 75th).
- Identify transitions such as prices moving **from 75th percentile → 50th → 25th → 50th → 75th**.
- Count these transitions and use them to evaluate reversionary behavior.

```
import pandas as pd

import numpy as np

import plotly.graph_objects as go
```

```python
import plotly.express as px
from sklearn.linear_model import LinearRegression


data = pd.read_csv("AAPL_data.csv", parse_dates=["Date"])
data.set_index("Date", inplace=True)
data["Close"] = data["Close"].astype(float)


mean_price = data["Close"].mean()
percentile_25 = np.percentile(data["Close"], 25)
percentile_50 = np.percentile(data["Close"], 50)
percentile_75 = np.percentile(data["Close"], 75)


data["Deviation"] = mean_price - data["Close"]
data["Delta_Close"] = data["Close"].diff()


# Refine Theta and Sigma
X = data["Deviation"].shift(1).dropna().values.reshape(-1, 1)
```

```python
y = data["Delta_Close"].dropna().values


model = LinearRegression(fit_intercept=False)

model.fit(X, y)


theta = model.coef_[0]

residuals = y - model.predict(X)

sigma = residuals.std()


# Identify Reversionary Moves Across Percentiles

def classify_percentile(price):

if price <= percentile_25:

return "25th"

elif price <= percentile_50:

return "50th"

elif price <= percentile_75:

return "75th"
```

```
return "Above 75th"

data["Percentile"] = data["Close"].apply(classify_percentile)

# Identify transitions between percentiles
data["Transition"] = data["Percentile"].shift(1).fillna("Start") + "->" + data["Percentile"]
reversionary_moves = data["Transition"].value_counts()

# Calculate average durations within each percentile
data["Percentile_Group"] = (data["Percentile"] != data["Percentile"].shift(1)).cumsum()
percentile_durations = data.groupby(["Percentile", "Percentile_Group"]).size().reset_index(name="Duration")
average_durations = percentile_durations.groupby("Percentile")["Duration"].mean()

# Pure Reversion
```

```
pure_reversion = [data["Close"].iloc[0]]

for i in range(1, len(data)):

drift = theta * (mean_price - pure_reversion[-1])

pure_reversion.append(pure_reversion[-1] + drift)

data["Pure_Reversion"] = pure_reversion


# Revealed Reversion

revealed_reversion = [data["Close"].iloc[0]]

np.random.seed(25)

for i in range(1, len(data)):

drift = theta * (mean_price - revealed_reversion[-1])

diffusion = sigma * np.random.normal()

revealed_reversion.append(revealed_reversion[-1] + drift +
diffusion)


fig = go.Figure()
```

```
fig.add_trace(go.Scatter(x=data.index, y=data["Close"],
mode="lines", name="Actual Prices",
line=dict(color="blue")))


fig.add_trace(go.Scatter(x=data.index,
y=data["Pure_Reversion"], mode="lines", name="Pure
Reversion", line=dict(color="green")))


fig.add_trace(go.Scatter(x=data.index,
y=data["Revealed_Reversion"], mode="lines",
name="Revealed Reversion", line=dict(color="red")))


fig.add_trace(go.Scatter(x=data.index, y=[mean_price] *
len(data), mode="lines", name="Mean Price",
line=dict(color="orange", dash="dash")))


fig.update_layout(

title="Mean Reversion in AAPL",

xaxis_title="Date",
```

```
yaxis_title="Price",
template="plotly_white",
legend_title="Legend"
)

fig.show()

# Percentile Transition Heatmap
transition_matrix =
pd.crosstab(data["Percentile"].shift(1).fillna("Start"),
data["Percentile"])
heatmap_fig = px.imshow(transition_matrix, text_auto=True,
color_continuous_scale="Blues")
heatmap_fig.update_layout(title="Percentile Transition
Heatmap", xaxis_title="To", yaxis_title="From")
heatmap_fig.show()

# Results
```

```python
print(f"Estimated Theta (Speed of Reversion): {theta:.4f}")

print(f"Estimated Sigma (Noise Intensity): {sigma:.4f}")

print("\nReversionary Moves (Percentile Transitions):")

print(reversionary_moves)

print("\nAverage Duration in Each Percentile:")

print(average_durations)
```

Figure 5.8

Figure 5.9

Estimated Theta (Speed of Reversion): 0.0027
Estimated Sigma (Noise Intensity): 2.6954

Reversionary Moves (Percentile Transitions):

Transition	
25th->25th	284
Above 75th->Above 75th	277
50th->50th	269
75th->75th	262
75th->Above 75th	18
Above 75th->75th	17
50th->75th	15
75th->50th	14
25th->50th	11
50th->25th	10
Start->25th	1

Average Duration in Each Percentile:

Percentile	
25th	26.818182
50th	11.760000

75th	**9.187500**
Above 75th	**16.388889**

Reversionary Moves (Percentile Transitions):

- These transitions describe how often the stock price moved between different percentile ranges:
  - `25th->25th: 284`: The price remained in the 25th percentile range for 284 time steps.
  - `Above 75th->Above 75th: 277`: The price stayed in the above-75th percentile range for 277 time steps.
  - `50th->50th: 269`, `75th->75th: 262`: Similar patterns are observed for the other ranges.
- These high values suggest that the price often remained within the same percentile range rather than frequently transitioning.

Transitions Between Percentiles:

  - `75th->Above 75th: 18`, `Above 75th->75th: 17`: These are relatively rare transitions, showing price movement between the highest percentile and just below it.
  - `50th->75th: 15`, `75th->50th: 14`: The price crossed between the 50th and 75th percentiles a few times.

- ○ `25th->50th: 11`, `50th->25th: 10`: Similarly, movements between the lower percentiles were infrequent.
- The relatively small number of transitions across percentiles suggests that the stock exhibited **persistent clustering** within certain ranges.

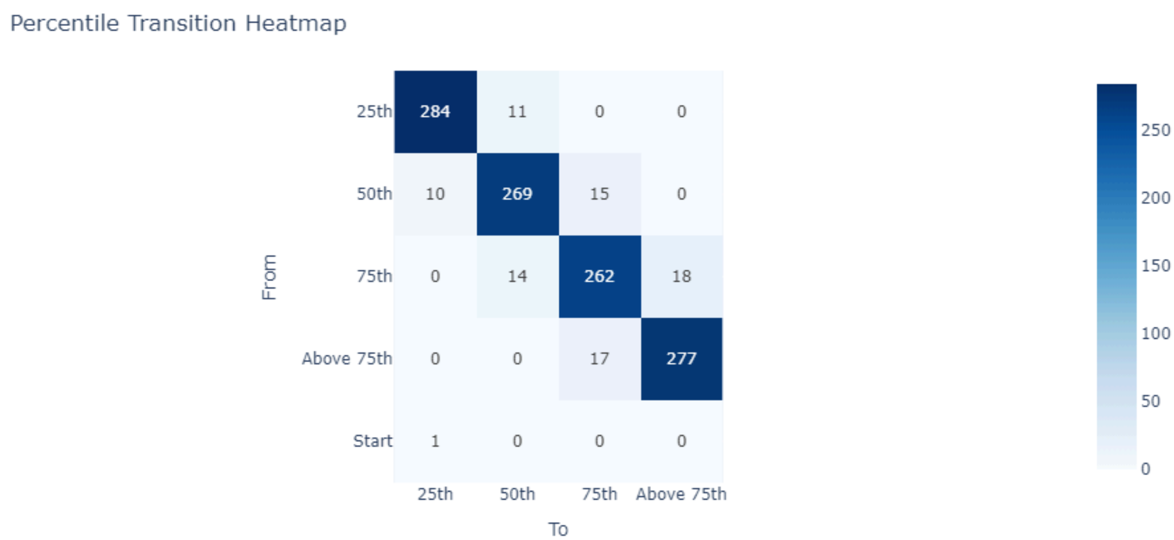

Figure 6.1

Create Derived Features

```
data["Deviation"] = mean_price - data["Close"]  # Difference from the mean.
data["Delta_Close"] = data["Close"].diff()  # Daily price change.
```

- **Deviation**: Measures how far each price is from the mean.
- **Delta_Close**: Captures the day-to-day price movement.

Fit Mean-Reversion Model

```
X = data["Deviation"].shift(1).dropna().values.reshape(-1, 1)  # Lagged deviation.
y = data["Delta_Close"].dropna().values  # Daily changes.

model = LinearRegression(fit_intercept=False)  # Initialize linear regression without intercept.
model.fit(X, y)  # Fit the model.

theta = model.coef_[0]  # Extract theta (mean-reversion speed).
residuals = y - model.predict(X)  # Calculate residuals.
sigma = residuals.std()  # Standard deviation of residuals (noise intensity).
```

- Fits a linear regression model to estimate:
    - **Theta (θ)**: The speed at which prices revert to the mean.
    - **Sigma (σ)**: The magnitude of random noise.

Classify Percentiles

```
def classify_percentile(price):
    if price <= percentile_25:
        return "25th"
    elif price <= percentile_50:
        return "50th"
    elif price <= percentile_75:
        return "75th"
    return "Above 75th"

data["Percentile"] =
data["Close"].apply(classify_percentile)  #
Classify each price.
```

- Categorizes prices into percentile groups (e.g., below 25th percentile).

Analyze Transitions

```
data["Transition"] =
data["Percentile"].shift(1).fillna("Start") +
"->" + data["Percentile"]
reversionary_moves =
data["Transition"].value_counts()  # Count
transitions between percentiles.
```

- Tracks transitions between percentiles, e.g., "25th -> 50th".

Compute Average Duration

```
data["Percentile_Group"] =
(data["Percentile"] !=
data["Percentile"].shift(1)).cumsum()
percentile_durations =
data.groupby(["Percentile",
"Percentile_Group"]).size().reset_index(name=
"Duration")
average_durations =
percentile_durations.groupby("Percentile")["D
uration"].mean()
```

- Calculates how long prices stay within each percentile group on average.

Simulate Price Series

```
pure_reversion = [data["Close"].iloc[0]]
for i in range(1, len(data)):
    drift = theta * (mean_price - pure_reversion[-1])
    pure_reversion.append(pure_reversion[-1] + drift)
data["Pure_Reversion"] = pure_reversion  # Add pure reversion to the data.

revealed_reversion = [data["Close"].iloc[0]]
np.random.seed(25)
for i in range(1, len(data)):
    drift = theta * (mean_price - revealed_reversion[-1])
    diffusion = sigma * np.random.normal()

revealed_reversion.append(revealed_reversion[-1] + drift + diffusion)
data["Revealed_Reversion"] = revealed_reversion  # Add revealed reversion.
```

- **Pure Reversion**: Models prices revert to the mean based solely on drift.
- **Revealed Reversion**: Adds random noise to mimic market behavior.

Below we will observe the differences with data on a 1 minute interval.

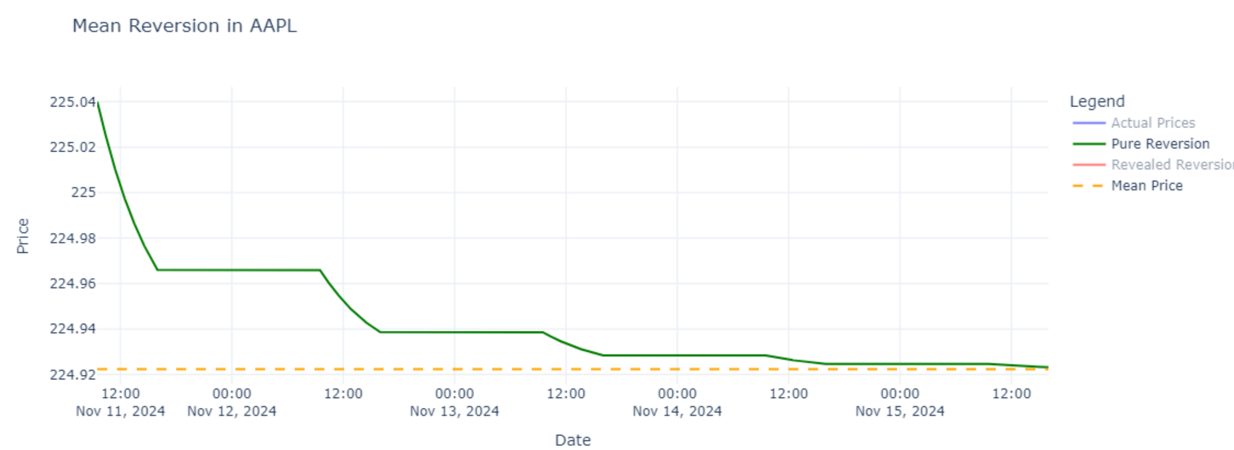

Figure 6.2

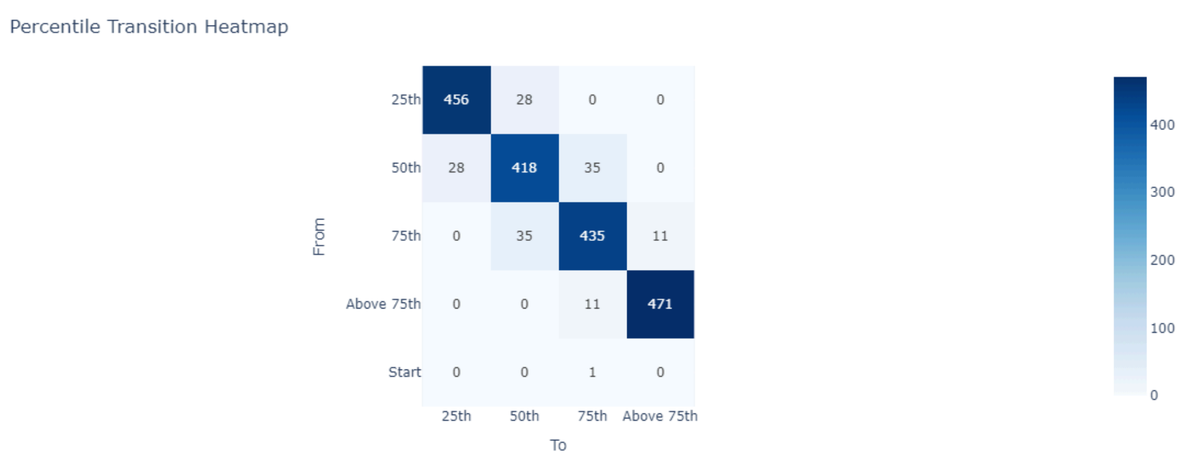

Figure 6.3

Estimated Theta (Speed of Reversion): 0.0026
Estimated Sigma (Noise Intensity): 0.1134

Reversionary Moves (Percentile Transitions):

Transition	
Above 75th->Above 75th	471
25th->25th	456
75th->75th	435
50th->50th	418
75th->50th	35
50th->75th	35
50th->25th	28
25th->50th	28
75th->Above 75th	11
Above 75th->75th	11
Start->75th	1

Average Duration in Each Percentile:

Percentile	
25th	17.285714
50th	7.634921
75th	10.255319
Above 75th	43.818182