In [302…

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline, interp1d,interp2d
%matplotlib inline
plt.rcParams['figure.figsize'] =(12,8)
import warnings
warnings. filterwarnings('ignore')
```

In [303…

```
data
```

Out[303…

```
{'tenors': ['1', '2', '3', '5', '7', '10', '20', '30'],
 'rates': [5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5, 4.81]}
```

In [304…

```
data={'tenors':['1','2','3','5','7','10','20','30'],
     'rates':[5.49,5.12,4.88,4.72,4.73,4.69,5,4.81]}
term_structure=pd.DataFrame(data,)
term_structure.index=term_structure.tenors
```

In [305…

```
term_structure.drop(columns=['tenors'],inplace=True)
term_structure
```

Out[305…

| tenors | rates |
|---|---|
| **1** | 5.49 |
| **2** | 5.12 |
| **3** | 4.88 |
| **5** | 4.72 |
| **7** | 4.73 |
| **10** | 4.69 |
| **20** | 5.00 |
| **30** | 4.81 |

In [306…

```
term_structure.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 8 entries, 1 to 30
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   rates   8 non-null      float64
dtypes: float64(1)
memory usage: 128.0+ bytes
```

In [307…

```
tenor,interval=np.linspace(start=1,stop=30,retstep=True,num=8,dtype=int)
tenor=tenor
tenor
```

Out[307…

```
array([ 1,  5,  9, 13, 17, 21, 25, 30])
```

In [ ]:

Linear Interpolation:

Linear interpolation is a simple and commonly used method for estimating values between two known data points. Equation:

```
Given two points (x1, y1) and (x2, y2), the linearly interpolated
value at a point x within this range is calculated as:
                          y = y1 + (x - x1) * ((y2 - y1) / (x2 - x1))
```

Benefits: Simplicity: Linear interpolation is straightforward and easy to implement. Speed: It is computationally efficient.

Limitations: Lack of Flexibility: Linear interpolation assumes a constant rate of change between data points, which may not accurately capture complex yield curve dynamics.

When to Use: Linear interpolation is suitable when a simple approximation of values between data points is sufficient.

Quadratic Interpolation: Quadratic interpolation is an extension of linear interpolation, used when data points exhibit a curvilinear pattern. Equation: Given three points (x1, y1), (x2, y2), and (x3, y3), the quadratic interpolation formula is: y = A *x^2* + *B* x + C, where A, B, and C are coefficients determined by solving a system of equations involving the three points.

Benefits: Improved Fit: Quadratic interpolation can capture curvilinear patterns in data better than linear interpolation.

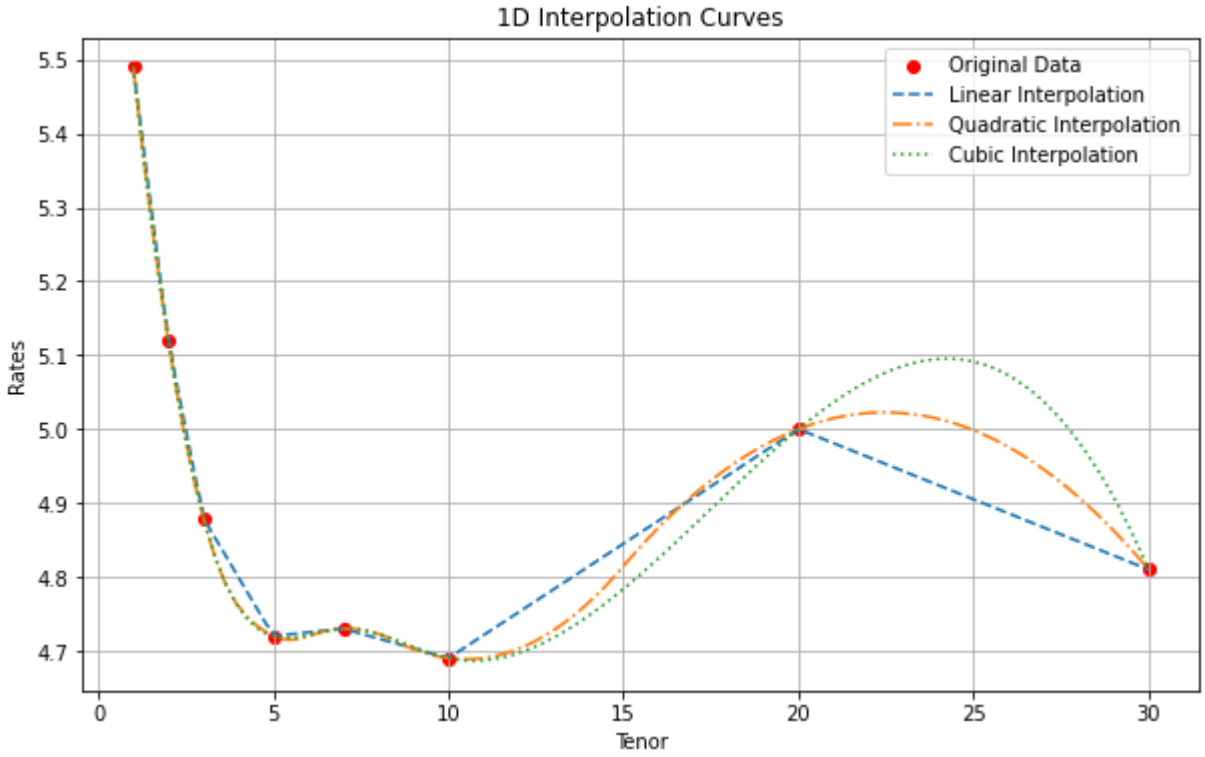
Limitations: Still Limited Flexibility: Quadratic interpolation assumes a parabolic curve, which may not suit all yield curve shapes.

When to Use: Quadratic interpolation is useful when data exhibits a smooth curvilinear behavior.

In [308…

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# Input your data here
tenors = ['1', '2', '3', '5', '7', '10', '20', '30']
rates = [5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5, 4.81]

# Convert tenors to numerical values
tenor_values = [float(tenor) for tenor in tenors]

# Define the range of x values for the plot
x_range = np.linspace(min(tenor_values), max(tenor_values), 100)

# Linear Interpolation
linear_interp = interp1d(tenor_values, rates, kind='linear')

# Quadratic Interpolation
quadratic_interp = interp1d(tenor_values, rates, kind='quadratic')

# Cubic Interpolation
```

```
cubic_interp = interp1d(tenor_values, rates, kind='cubic')

# Plot the original data and interpolation curves
plt.figure(figsize=(10, 6))
plt.scatter(tenor_values, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, linear_interp(x_range), label='Linear Interpolation', linestyle='-
plt.plot(x_range, quadratic_interp(x_range), label='Quadratic Interpolation', linest
plt.plot(x_range, cubic_interp(x_range), label='Cubic Interpolation', linestyle=':')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('1D Interpolation Curves')
plt.legend()
plt.grid(True)
plt.show()
```




Cubic Spline Interpolation:

Cubic spline interpolation is a more advanced method used to construct a piecewise cubic polynomial that fits a set of data points smoothly. Equation: The cubic spline interpolant between two points (xi, yi) and (xi+1, yi+1) is a cubic function of the form:

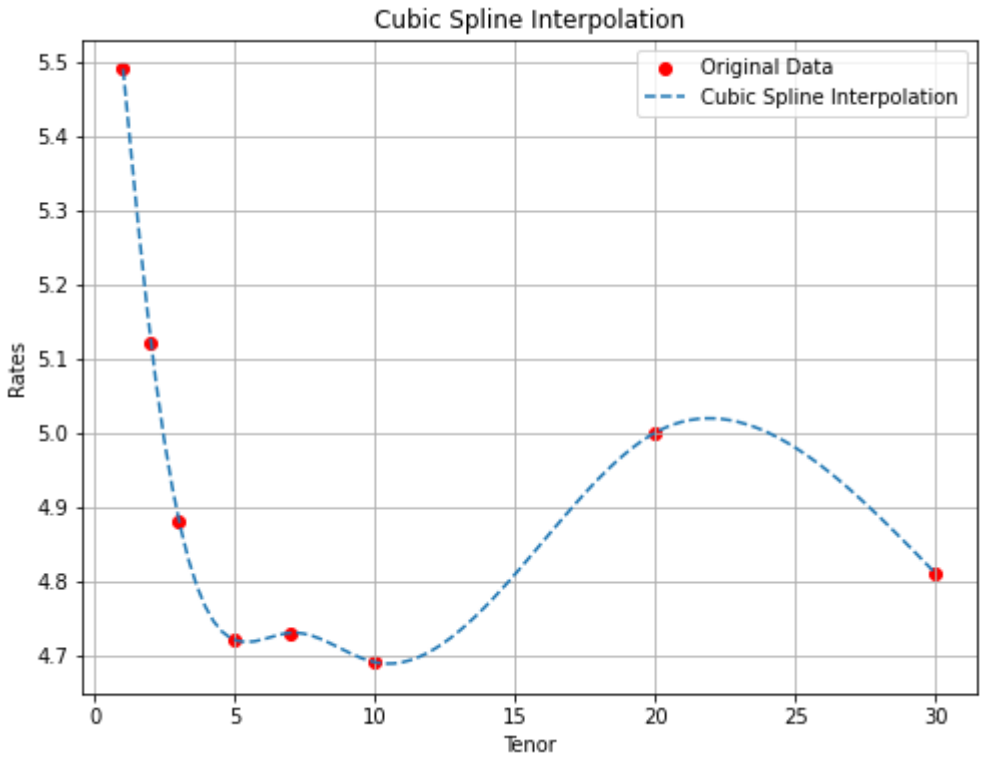
```
S(x) = a + b * (x - xi) + c * (x - xi)^2 + d * (x - xi)^3
```

The coefficients a, b, c, and d are determined to ensure continuity of the interpolant at each data point and its first and second derivatives.

Benefits: Flexibility: Cubic spline interpolation provides flexibility to fit complex yield curve shapes with smoothness. Local Accuracy: It minimizes oscillations and is locally accurate. Limitations: Complexity: The method involves solving a system of equations and can be computationally intensive. When to Use: Cubic spline interpolation is a good choice when you want a smooth, piecewise polynomial representation of the yield curve.

In [309…

```
# CubicSpline
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline

# Input your data here
tenors = ['1', '2', '3', '5', '7', '10', '20', '30']
rates = [5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5, 4.81]

# Convert tenors to numerical values
tenor_values = [float(tenor) for tenor in tenors]

# Create a cubic spline interpolation
cs = CubicSpline(tenor_values, rates,bc_type='natural')

# Define the range of x values for the plot
x_range = np.linspace(min(tenor_values), max(tenor_values), 100)

# Calculate corresponding y values using the spline
y_interp = cs(x_range)

# Plot the original data points and the cubic spline interpolation
plt.figure(figsize=(8, 6))
plt.scatter(tenor_values, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_interp, label='Cubic Spline Interpolation', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Cubic Spline Interpolation')
plt.legend()
plt.grid(True)
plt.show()
```




In [ ]:

Nelson-Siegel-Svensson (NSS) Yield Curve Model:

The NSS model is a parametric model for yield curve modeling. It uses four parameters to capture the term structure of interest rates. Equations:

Y(t) = beta0 + beta1 *(1 - exp(-lambda1* t)) / (lambda1 *t)* + *beta2* ((1 - exp(-lambda1 *t)) / (lambda1* t) - exp(-lambda1 * t))

beta0 (Beta Zero):

Interpretation: Beta0 represents the long-term or ultimate level of interest rates. It determines the horizontal or flat part of the yield curve. Role: Beta0 sets the "anchor" point for the yield curve at long maturities, indicating where rates are expected to converge in the distant future. Effect on Yield Curve: An increase in beta0 will shift the entire yield curve upwards or downwards while keeping its shape unchanged.

beta1 (Beta One):

Interpretation: Beta1 is associated with the short-term dynamics of the yield curve. It controls the initial slope of the curve. Role: Beta1 influences the steepness of the yield curve at short maturities. A higher beta1 results in a steeper initial slope. Effect on Yield Curve: Changes in beta1 primarily affect the short end of the yield curve, making it steeper or flatter.

beta2 (Beta Two):

Interpretation: Beta2 reflects the curvature or bending of the yield curve. It is responsible for capturing the hump or dip in the middle of the curve. Role: Beta2 shapes the middle part of the yield curve. A positive beta2 contributes to an upward-sloping hump, while a negative beta2 creates a downward-sloping dip. Effect on Yield Curve: Changes in beta2 mainly affect the mid-term portion of the yield curve, making it more humped or flat.

lambda1 (Lambda One):

Interpretation: Lambda1 controls the speed at which short-term interest rates converge to the long-term level (beta0). Role: Lambda1 determines the rate of mean reversion or how quickly the yield curve returns to its long-term mean. Effect on Yield Curve: A higher lambda1 implies a faster mean reversion, leading to a quicker convergence of short-term rates to beta0.

Beta0: sets the long-term level, beta1: shapes the short-term slope, beta2:introduces curvature, lambda1:controls the speed of mean reversion

Nelson-Siegel-Svensson (NSS) Yield Curve Model:

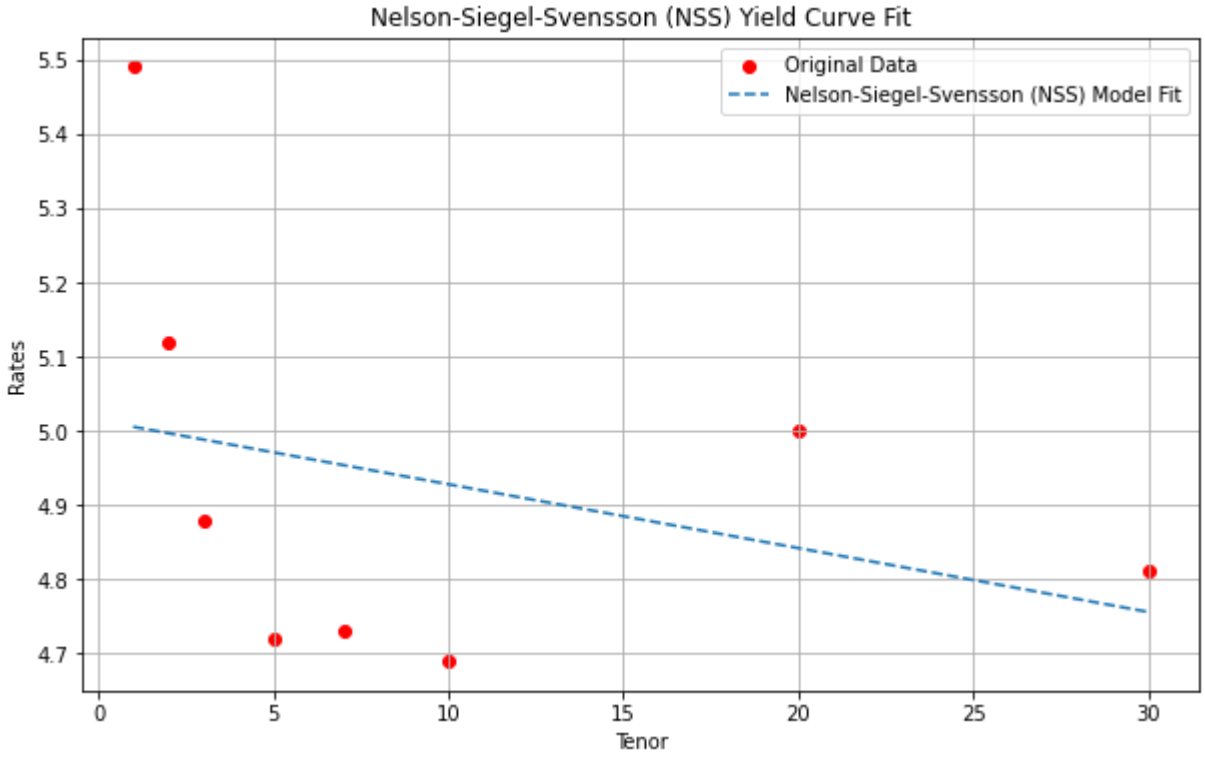
Benefits: Parametric Modeling: NSS provides a parametric model with interpretable parameters for capturing term structure features. Limitations: Simplified Representation: NSS assumes that yield curve dynamics are determined solely by exponential functions, which may not capture all market behavior. When to Use: NSS is suitable when you want a parsimonious parametric model for yield curve estimation, particularly when curve shape analysis is important.

Nelson-Siegel-Svensson Yield Curve Fit Method

In [319…

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

```
# Input your data here
tenors = [1, 2, 3, 5, 7, 10, 20, 30]
rates = [5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5.0, 4.81]

# Define the Nelson-Siegel-Svensson (NSS) function
def nss_curve(t, beta0, beta1, beta2, lambda1, lambda2):
    return beta0 + (beta1 * ((1 - np.exp(-t / lambda1)) / (t / lambda1))) + (beta2 *

# Fit the NSS model to the data
params, covariance = curve_fit(nss_curve, tenors, rates)

# Extract the fitted parameters
beta0, beta1, beta2, lambda1, lambda2 = params

# Define the range of x values for the plot
x_range = np.linspace(min(tenors), max(tenors), 100)

# Calculate corresponding y values using the fitted NSS model
y_nss = nss_curve(x_range, beta0, beta1, beta2, lambda1, lambda2)

# Plot the original data and the NSS curve
plt.figure(figsize=(10, 6))
plt.scatter(tenors, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_vasicek, label='Nelson-Siegel-Svensson (NSS) Model Fit', linesty

plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Nelson-Siegel-Svensson (NSS) Yield Curve Fit')
plt.legend()
plt.grid(True)
plt.show()

# Print the fitted parameters
print("Fitted Parameters:")
print(f"beta0: {beta0}")
print(f"beta1: {beta1}")
print(f"beta2: {beta2}")
print(f"lambda1: {lambda1}")
print(f"lambda2: {lambda2}")
```




```
Fitted Parameters:
beta0: 4.992787162771972
beta1: 1.1953238656328482
beta2: -2.4856856983598554
lambda1: 2.0431877699379624
lambda2: 1.0
```

INTEREST RATES MODELS

Vasicek Model:

The Vasicek model is a stochastic model used to describe the evolution of short-term interest rates over time. Equation: [ dr(t) = kappa *(mu - r(t))* dt + sigma *sqrt(dt)* dW(t) ]

```
The parameters are:
    mu    :the long-term mean,
    kappa :the speed of reversion to the mean,
    sigma :the volatility,
    r(t)  :the interest rate at time t
```

Benefits: Stochastic Modeling: Vasicek models the dynamics of interest rates stochastically, capturing mean reversion and volatility.

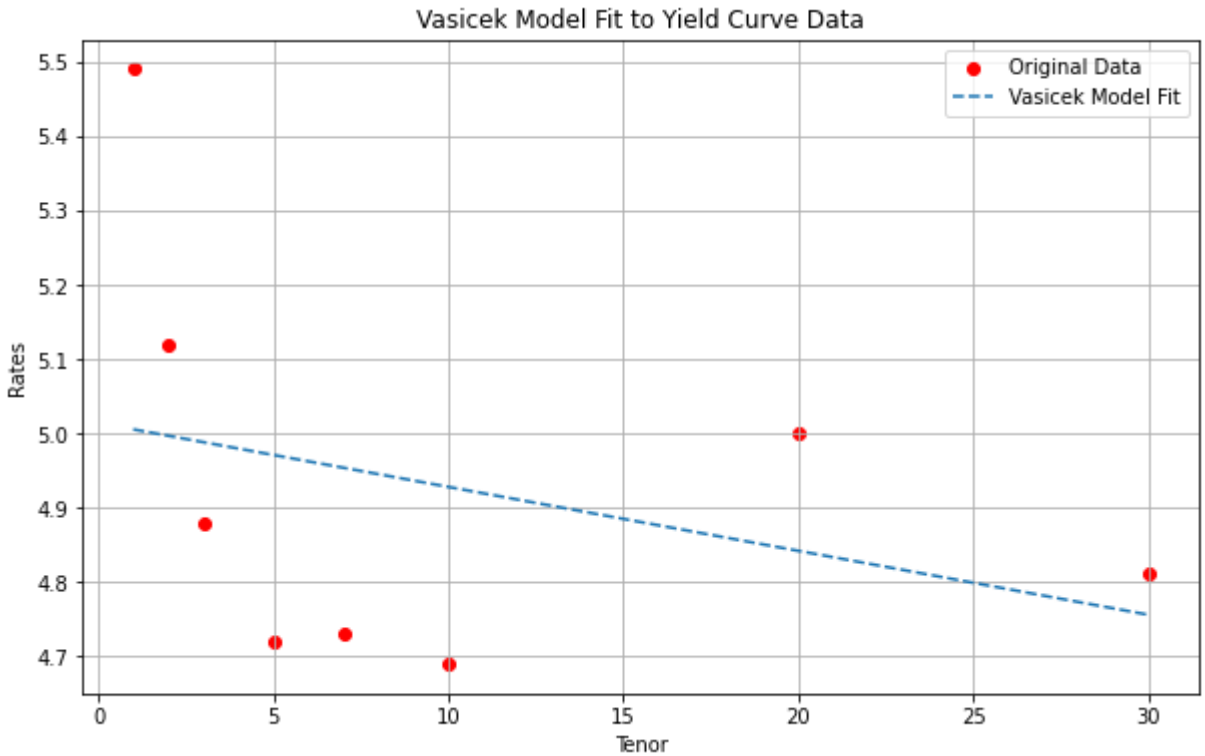
Limitations: Short-Term Focus: It is primarily designed for modeling short-term rates and may not be suitable for long-term yield curve modeling.

When to Use: Use Vasicek when modeling short-term interest rates and capturing mean reversion is crucial.

In [311…

```
# Vasicek model
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Input your data here
```

```
tenors = np.array([1, 2, 3, 5, 7, 10, 20, 30])
rates = np.array([5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5.0, 4.81])

# Define the Vasicek model function
def vasicek_model(t, mu, kappa, sigma, r0):
    dr = kappa * (mu - t)
    return dr * sigma + r0

# Fit the Vasicek model to the data
params, covariance = curve_fit(vasicek_model, tenors, rates, p0=[4.5, 0.1, 0.1, 4.5])

# Extract the fitted parameters
mu, kappa, sigma, r0 = params

# Define the range of x values for the plot
x_range = np.linspace(min(tenors), max(tenors), 100)

# Calculate corresponding y values using the fitted Vasicek model
y_vasicek = vasicek_model(x_range, mu, kappa, sigma, r0)

# Plot the original data and the Vasicek model fit
plt.figure(figsize=(10, 6))
plt.scatter(tenors, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_vasicek, label='Vasicek Model Fit', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Vasicek Model Fit to Yield Curve Data')
plt.legend()
plt.grid(True)
plt.show()

# Print the fitted parameters
print("Fitted Parameters:")
print(f"mu: {mu}")
print(f"kappa: {kappa}")
print(f"sigma: {sigma}")
print(f"r0: {r0}")
```

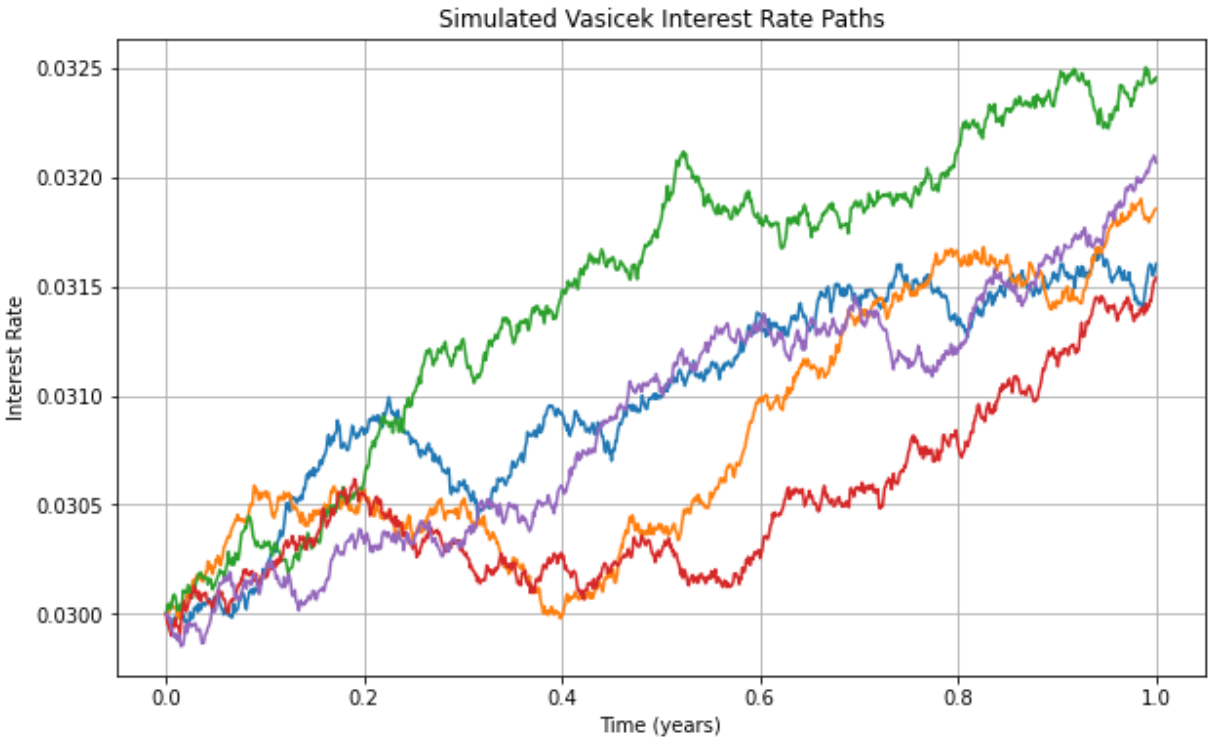



```
Fitted Parameters:
mu: 53.63581391820601
```

```
kappa: 0.17518844186556864
sigma: 0.04911743750547962
r0: 4.552370460913483
```

In [312…

```python
# Vasicek model
import numpy as np
import matplotlib.pyplot as plt

# Vasicek model parameters
mu = 0.05  # Long-term mean of the interest rate
kappa = 0.1  # Speed of reversion to the mean
sigma = 0.02  # Volatility of interest rates
r0 = 0.03  # Initial interest rate

# Simulation parameters
T = 1.0  # Time horizon (in years)
N = 1000  # Number of time steps
dt = T / N  # Time step size
num_paths = 5  # Number of interest rate paths to simulate

# Initialize arrays to store interest rate paths
rates = np.zeros((N + 1, num_paths))
rates[0, :] = r0

# Simulate interest rate paths using the Vasicek model
for i in range(num_paths):
    for t in range(1, N + 1):
        dW = np.random.normal(0, np.sqrt(dt))
        dr = kappa * (mu - rates[t - 1, i]) * dt + sigma * np.sqrt(dt) * dW
        rates[t, i] = rates[t - 1, i] + dr

# Plot interest rate paths
plt.figure(figsize=(10, 6))
plt.plot(np.arange(N + 1) * dt, rates)
plt.xlabel('Time (years)')
plt.ylabel('Interest Rate')
plt.title('Simulated Vasicek Interest Rate Paths')
plt.grid(True)
plt.show()
```



Hull-White Model:

The Hull-White model is a one-factor model used to model interest rate movements over time. Equation: [ dr(t) = (theta(t) - alpha *r(t))* dt + sigma * dW(t) ] The parameters include : alpha :the speed of reversion to the mean, sigma :the volatility, r(t) :the interest rate at time t, theta(t) :a deterministic function

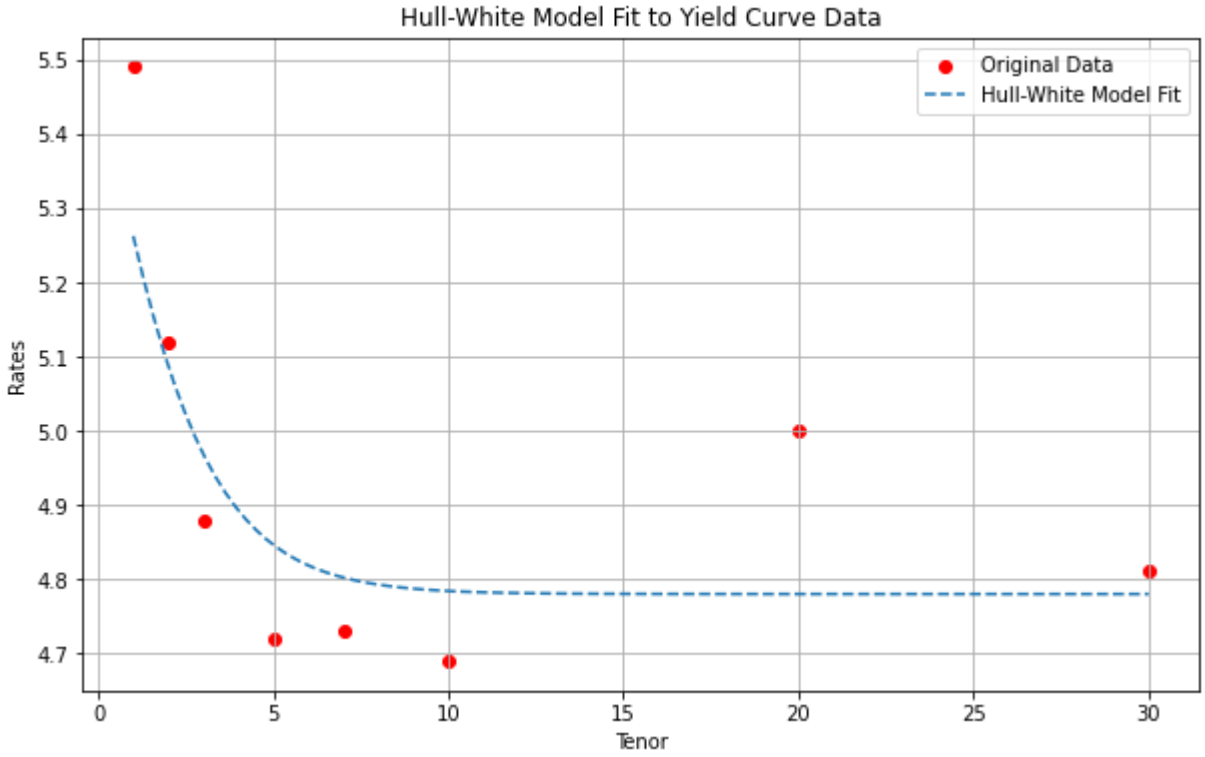
Benefits: Stochastic Modeling: Hull-White models provide a framework for modeling interest rate movements over time, including mean reversion.

Limitations: Limited Factors: It is a one-factor model and may not fully capture complex yield curve dynamics.

When to Use: Hull-White is useful for modeling interest rates when you want to capture mean reversion and are willing to accept simplifications.

In [313…

```
# Hull-White model
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Input your data here
tenors = np.array([1, 2, 3, 5, 7, 10, 20, 30])
rates = np.array([5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5.0, 4.81])

# Define the Hull-White model function
def hull_white_model(t, alpha, sigma):
    r0 = rates[0]  # Use the first data point as the initial rate
    A = alpha * (1 - np.exp(-sigma * t)) / sigma
    B = (sigma / alpha) * (1 - np.exp(-alpha * t))
    return r0 + A - B

# Fit the Hull-White model to the data
params, covariance = curve_fit(hull_white_model, tenors, rates, p0=[0.01, 0.01])

# Extract the fitted parameters
alpha, sigma = params

# Define the range of x values for the plot
x_range = np.linspace(min(tenors), max(tenors), 100)

# Calculate corresponding y values using the fitted Hull-White model
y_hw = hull_white_model(x_range, alpha, sigma)

# Plot the original data and the Hull-White model fit
plt.figure(figsize=(10, 6))
plt.scatter(tenors, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_hw, label='Hull-White Model Fit', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Hull-White Model Fit to Yield Curve Data')
plt.legend()
plt.grid(True)
plt.show()

# Print the fitted parameters
print("Fitted Parameters:")
print(f"Alpha: {alpha}")
print(f"Sigma: {sigma}")
```




```
Fitted Parameters:
Alpha: 0.5798480274407334
Sigma: 0.8211630137003364
```

Ho-Lee Model:

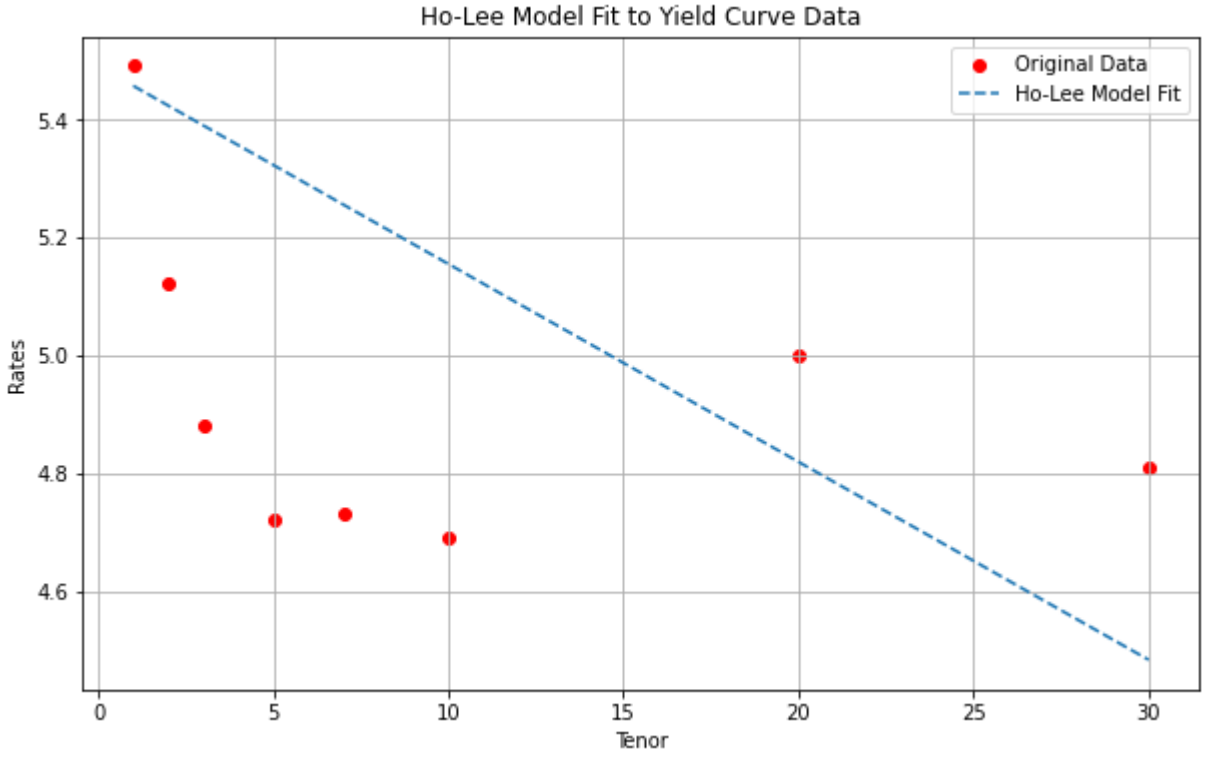
The Ho-Lee model is a simple one-factor model used for interest rate modeling. Equation: dr(t) = alpha * dt The parameter alpha represents the constant rate of change.

Benefits: Simplicity: Ho-Lee is a simple one-factor model suitable for quick interest rate modeling. Limitations: Simplified: It is overly simplistic and may not capture more complex yield curve dynamics. When to Use: Ho-Lee can be used for quick, basic interest rate modeling when simplicity is preferred.

In [314…

```python
# Ho-Lee model function
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Input your data here
tenors = np.array([1, 2, 3, 5, 7, 10, 20, 30])
rates = np.array([5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5.0, 4.81])

# Define the Ho-Lee model function
def ho_lee_model(t, alpha):
    r0 = rates[0]  # Use the first data point as the initial rate
    return r0 + alpha * t

# Fit the Ho-Lee model to the data
params, covariance = curve_fit(ho_lee_model, tenors, rates, p0=[0.01])

# Extract the fitted parameter
alpha = params[0]

# Define the range of x values for the plot
x_range = np.linspace(min(tenors), max(tenors), 100)

# Calculate corresponding y values using the fitted Ho-Lee model
```

```
y_hl = ho_lee_model(x_range, alpha)

# Plot the original data and the Ho-Lee model fit
plt.figure(figsize=(10, 6))
plt.scatter(tenors, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_hl, label='Ho-Lee Model Fit', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Ho-Lee Model Fit to Yield Curve Data')
plt.legend()
plt.grid(True)
plt.show()

# Print the fitted parameter
print("Fitted Parameter:")
print(f"Alpha: {alpha}")
```




```
Fitted Parameter:
Alpha: -0.03356182836735981
```

Cox-Ingersoll-Ross (CIR) Model:

The CIR model is a stochastic model used to describe the evolution of short-term interest rates, with mean reversion and volatility. Equation: dr(t) = alpha *(beta - r(t))* dt + sigma *sqrt(r(t))* dW(t) The parameters include : alpha :the speed of reversion to the mean, beta :the long-term mean, sigma :the volatility, r(t) :the interest rate at time t

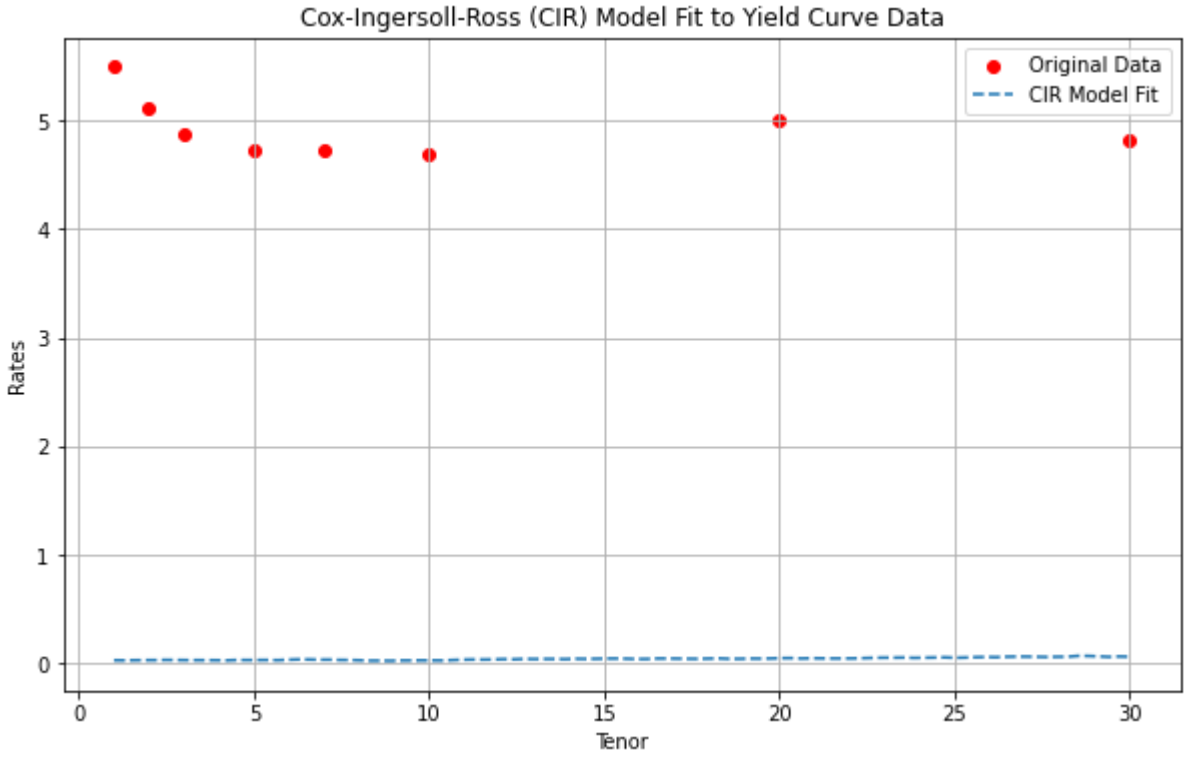
Benefits: Stochastic Modeling: CIR models are designed for modeling interest rate movements, including mean reversion and volatility.

Limitations: Short-Term Focus: Like Vasicek, it may be better suited for modeling short-term rates.

When to Use: Use CIR when you want to model interest rates with stochastic mean reversion and volatility.

In [315…

```python
# CIR model
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Input your data here
tenors = np.array([1, 2, 3, 5, 7, 10, 20, 30])
rates = np.array([5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5.0, 4.81])

# Define the CIR model function
def cir_model(t, alpha, beta, sigma, r0):
    dt = np.diff(t)
    r = [r0]
    for i in range(len(dt)):
        dW = np.random.normal(0, np.sqrt(dt[i]))
        dr = alpha * (beta - r[-1]) * dt[i] + sigma * np.sqrt(r[-1]) * dW
        r.append(r[-1] + dr)
    return np.array(r)

# Fit the CIR model to the data
params, covariance = curve_fit(cir_model, tenors, rates, p0=[0.05, 0.1, 0.02, 0.03])

# Extract the fitted parameters
alpha, beta, sigma, r0 = params

# Define the range of x values for the plot
x_range = np.linspace(min(tenors), max(tenors), 100)

# Calculate corresponding y values using the fitted CIR model
y_cir = cir_model(x_range, alpha, beta, sigma, r0)

# Plot the original data and the CIR model fit
plt.figure(figsize=(10, 6))
plt.scatter(tenors, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_cir, label='CIR Model Fit', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Cox-Ingersoll-Ross (CIR) Model Fit to Yield Curve Data')
plt.legend()
plt.grid(True)
plt.show()

# Print the fitted parameters
print("Fitted Parameters:")
print(f"Alpha: {alpha}")
print(f"Beta: {beta}")
print(f"Sigma: {sigma}")
print(f"r0: {r0}")
```




```
Fitted Parameters:
Alpha: 0.05000012960106062
Beta: 0.10000033227207261
Sigma: 0.019999905448955104
r0: 0.03000007498748917
```

In [ ]:

Constant Elasticity of Variance (CEV) Model:

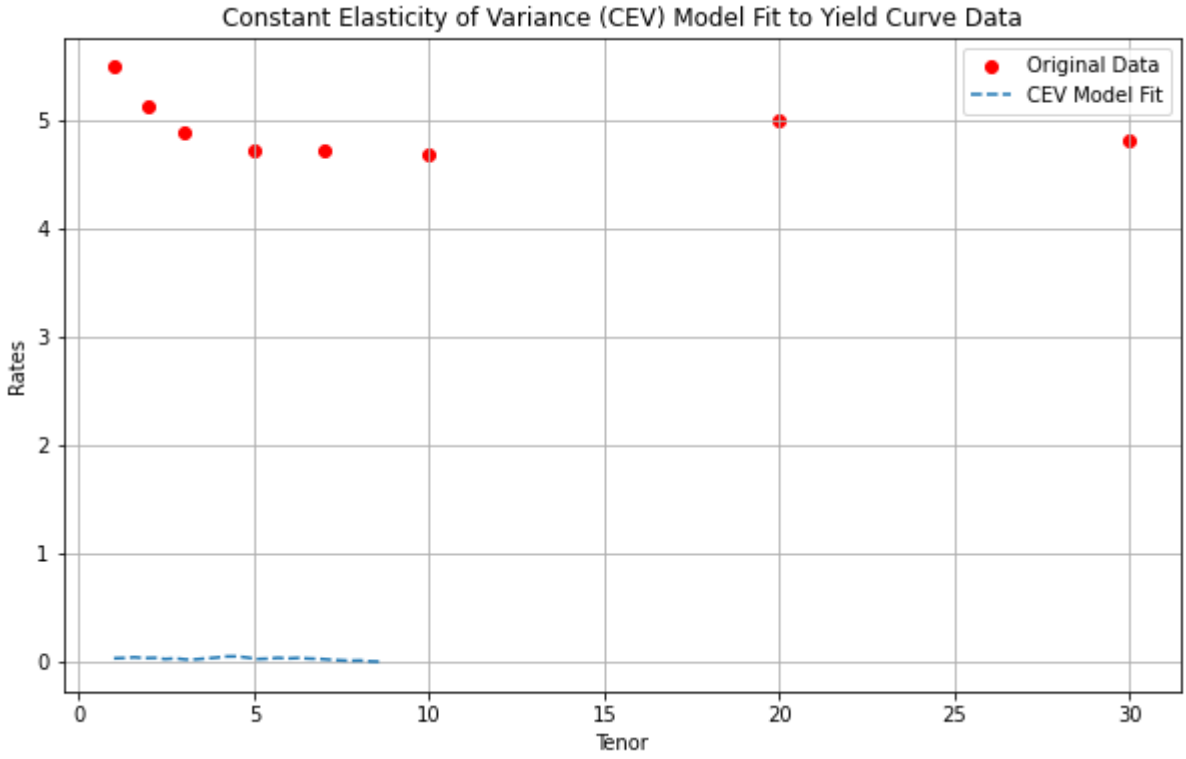
The CEV model is used for modeling interest rate processes with volatility that depends on the level of interest rates. Equation: dr(t) = alpha *(beta - r(t))* dt + sigma *r(t)^beta* dW(t) The parameters include alpha :the speed of reversion to the mean), beta :a power parameter), sigma :the volatility, and r(t) :the interest rate at time t).

Constant Elasticity of Variance (CEV) Model: Benefits: Nonlinear Volatility: CEV captures volatility that changes with the level of interest rates. Limitations: Complexity: It adds complexity with the power parameter, which may not always be justified. When to Use: CEV can be applied when capturing nonlinear volatility with a simple stochastic model is desired.

In [316…

```python
# Constant Elasticity of Variance (CEV) Model
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Input your data here
tenors = np.array([1, 2, 3, 5, 7, 10, 20, 30])
rates = np.array([5.49, 5.12, 4.88, 4.72, 4.73, 4.69, 5.0, 4.81])

# Define the CEV model function
def cev_model(t, alpha, beta, sigma, r0):
    dt = np.diff(t)
    r = [r0]
    for i in range(len(dt)):
        dW = np.random.normal(0, np.sqrt(dt[i]))
```

```
        dr = alpha * (beta - r[-1]) * dt[i] + sigma * r[-1]**beta * dW
        r.append(r[-1] + dr)
    return np.array(r)

# Fit the CEV model to the data
params, covariance = curve_fit(cev_model, tenors, rates, p0=[0.05, 0.1, 0.02, 0.03])

# Extract the fitted parameters
alpha, beta, sigma, r0 = params

# Define the range of x values for the plot
x_range = np.linspace(min(tenors), max(tenors), 100)

# Calculate corresponding y values using the fitted CEV model
y_cev = cev_model(x_range, alpha, beta, sigma, r0)

# Plot the original data and the CEV model fit
plt.figure(figsize=(10, 6))
plt.scatter(tenors, rates, label='Original Data', c='red', marker='o')
plt.plot(x_range, y_cev, label='CEV Model Fit', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title('Constant Elasticity of Variance (CEV) Model Fit to Yield Curve Data')
plt.legend()
plt.grid(True)
plt.show()

# Print the fitted parameters
print("Fitted Parameters:")
print(f"Alpha: {alpha}")
print(f"Beta: {beta}")
print(f"Sigma: {sigma}")
print(f"r0: {r0}")
```




```
Fitted Parameters:
Alpha: 0.05
Beta: 0.1
Sigma: 0.02
r0: 0.03
```

In [317…

```
plt.figure(figsize=(18,10))
# Original data
plt.scatter(tenor_values, rates, label='Original Data', c='black', marker='o')
# Linear Interpolation
plt.plot(x_range, linear_interp(x_range), label='Linear Interpolation', linestyle='-
# Quadratic Interpolation
plt.plot(x_range, quadratic_interp(x_range), label='Quadratic Interpolation', linest
# Cubic Interpolation
plt.plot(x_range, cubic_interp(x_range), label='Cubic Interpolation', linestyle=':')
# NSS Yield Curve Fit
plt.plot(x_range, y_nss, label='NSS Yield Curve Fit', linestyle='--')
# Vasicek Model Fit
plt.plot(x_range, y_vasicek, label='Vasicek Model Fit', linestyle='--')
# Hull-White Model Fit
plt.plot(x_range, y_hw, label='Hull-White Model Fit', linestyle='--')
# Ho-Lee Model Fit
plt.plot(x_range, y_hl, label='Ho-Lee Model Fit', linestyle='--')
# CIR Model Fit
# plt.plot(x_range, y_cir, label='CIR Model Fit', linestyle='--')

plt.plot(x_range, y_interp, label='Cubic Spline Interpolation', linestyle='--')
plt.xlabel('Tenor')
plt.ylabel('Rates')
plt.title(' Interpolation')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```