SM Group Work Project 1 M3 Student Group 5657

In [1]:

```
import numpy as np
import pandas as pd
from scipy.integrate import quad
from scipy.optimize import brute, fmin
import matplotlib.pyplot as plt
from numpy.fft import fft
from scipy.interpolate import splev, splrep

import warnings
warnings.filterwarnings("ignore")
```

Step1 MemberA Heston Characteristic Function

In [2]:

```
def H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    c1 = kappa_v * theta_v
    c2 = -np.sqrt(
        (rho * sigma_v * u * 1j - kappa_v) ** 2 - sigma_v**2 * (-u * 1j - u**2)
    )
    c3 = (kappa_v - rho * sigma_v * u * 1j + c2) / (
        kappa_v - rho * sigma_v * u * 1j - c2
    )
    H1 = r * u * 1j * T + (c1 / sigma_v**2) * (
        (kappa_v - rho * sigma_v * u * 1j + c2) * T
        - 2 * np.log((1 - c3 * np.exp(c2 * T)) / (1 - c3))
    )
    H2 = (
        (kappa_v - rho * sigma_v * u * 1j + c2)
        / sigma_v**2
        * ((1 - np.exp(c2 * T)) / (1 - c3 * np.exp(c2 * T)))
    )

    char_func_value = np.exp(H1 + H2 * v0)

    return char_func_value
```

Integral Value in Lewis (2001)

In [3]:

```
def H93_int_func(u, S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    char_func_value = H93_char_func(
        u - 1j / 2, T, r, kappa_v, theta_v, sigma_v, rho, v0
    )
    int_func_value = (
        1 / (u**2 + 0.25) * (np.exp(1j * u * np.log(S0 / K)) * char_func_value).real
    )
    return int_func_value

def H93_put_value(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    int_value = quad(
        lambda u: H93_int_func(u, S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0),
        0,
        np.inf,
        limit=250,
    )[0]
    call_value = max(0, S0 - np.exp(-r * T) * np.sqrt(S0 * K) / np.pi * int_value)
    put_value = call_value + K * np.exp(- r * T) - S0 # Put-Call parity
    return put_value
```

Heston Calibration

In [4]:

```
S0 = 232.90
r0 = 1.5/100

data = pd.read_csv("MScFE 622_Stochastic Modeling_GWP1_Option data.xlsx - 1.csv")
data["r"] = r0
data["T"] = data["Days to maturity"] / 250 # 250days / year

options = data[(data["Days to maturity"] == 15) & (data["Type"] == "P")] # Put Option with DTM is 15
options
```

Out[4]:

	Days to maturity	Strike	Price	Type	r	T
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

In [5]:

```
def H93_error_function(p0):
    global i, min_MSE
    kappa_v, theta_v, sigma_v, rho, v0 = p0
    if kappa_v < 0.0 or theta_v < 0.005 or sigma_v < 0.0 or rho < -1.0 or rho > 1.0:
        return 500.0
    if 2 * kappa_v * theta_v < sigma_v**2:
        return 500.0
    se = []
    for row, option in options.iterrows():
        model_value = H93_put_value(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
        )

        se.append((model_value - option["Price"]) ** 2)

    MSE = sum(se) / len(se)
    min_MSE = min(min_MSE, MSE)
    if i % 25 == 0:
        print("%4d |" % i, np.array(p0).round(2), "| %7.3f | %7.3f" % (MSE, min_MSE))
    i += 1
    return MSE

def H93_calibration_full():
    p0 = brute(
      H93_error_function,
```

```python
        (
            (1.5, 6.5, 5.0),
            (0.1, 0.4, 0.1),
            (0.01, 0.03, 0.01),
            (-0.5, 0.25, 0.25),
            (0.04, 0.09, 0.01),
        ),
        finish=None,
    )
    opt = fmin(
        H93_error_function, p0, xtol=0.00001, ftol=0.00001, maxiter=750, maxfun=900
    )
    return opt
```

In [6]:

```python
i = 0
min_MSE = 500

params_H93 = H93_calibration_full()
```

```
   0 | [ 1.5    0.1    0.01 -0.5    0.04] |    3.488 |    3.488
  25 | [1.5   0.1   0.02 0.    0.04] |    3.490 |    0.055
  50 | [ 1.5    0.2    0.02 -0.25   0.04] |    2.704 |    0.034
  75 | [ 1.5    0.3    0.02 -0.5    0.04] |    2.046 |    0.034
 100 | [1.5   0.4   0.01 0.    0.04] |    1.503 |    0.034
 125 | [ 1.5    0.4    0.02 -0.5    0.07] |    0.048 |    0.034
 150 | [ 1.48   0.4    0.02 -0.52   0.07] |    0.034 |    0.034
 175 | [ 1.28   0.43   0.03 -0.68   0.07] |    0.033 |    0.033
 200 | [ 0.95   0.49   0.05 -1.     0.07] |    0.031 |    0.031
 225 | [ 0.96   0.37   0.07 -0.98   0.08] |    0.029 |    0.029
 250 | [ 1.07   0.02   0.13 -0.96   0.09] |    0.026 |    0.026
 275 | [ 0.98   0.01   0.14 -0.98   0.09] |    0.024 |    0.024
 300 | [ 1.08   0.01   0.14 -0.99   0.09] |    0.023 |    0.023
 325 | [ 1.09   0.01   0.14 -1.     0.09] |    0.023 |    0.023
 350 | [ 1.08   0.01   0.14 -1.     0.09] |    0.023 |    0.023
 375 | [ 1.08   0.01   0.14 -1.     0.09] |    0.023 |    0.023
 400 | [ 1.08   0.01   0.14 -1.     0.09] |    0.023 |    0.023
 425 | [ 1.08   0.01   0.14 -1.     0.09] |    0.023 |    0.023
 450 | [ 1.08   0.01   0.14 -1.     0.09] |    0.023 |    0.023
Optimization terminated successfully.
         Current function value: 0.023166
         Iterations: 287
         Function evaluations: 508
```

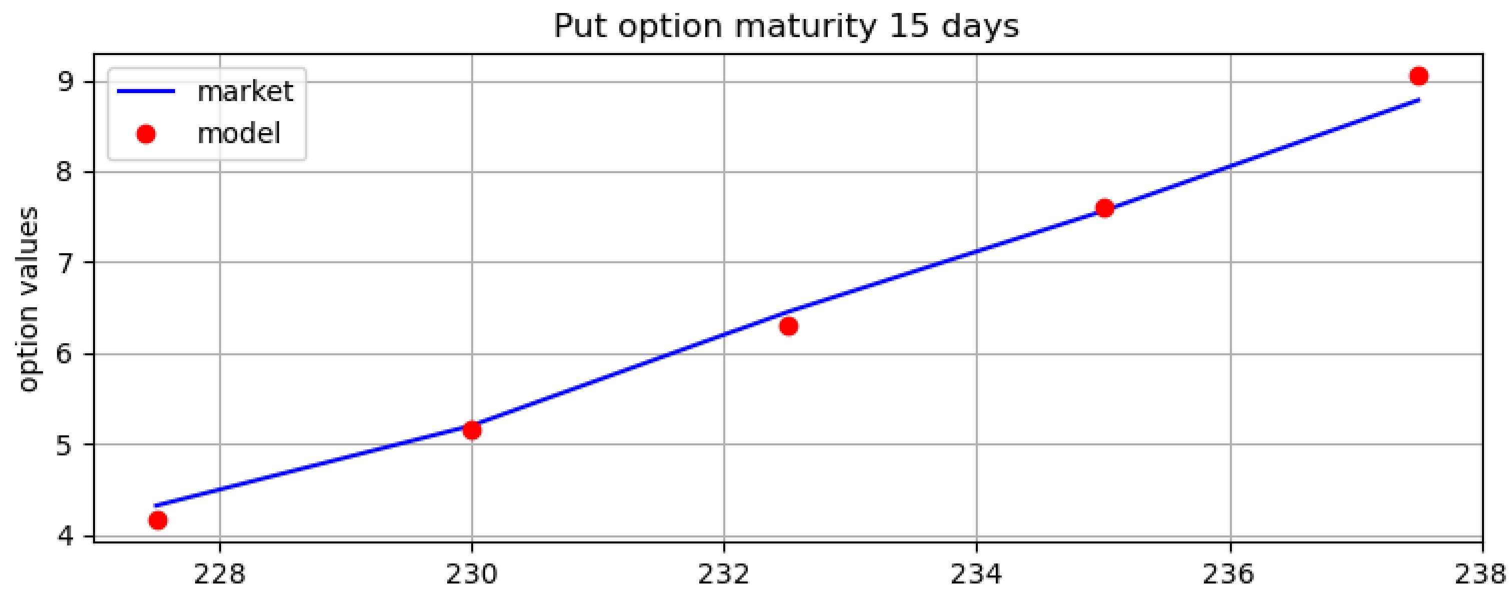
In [7]:

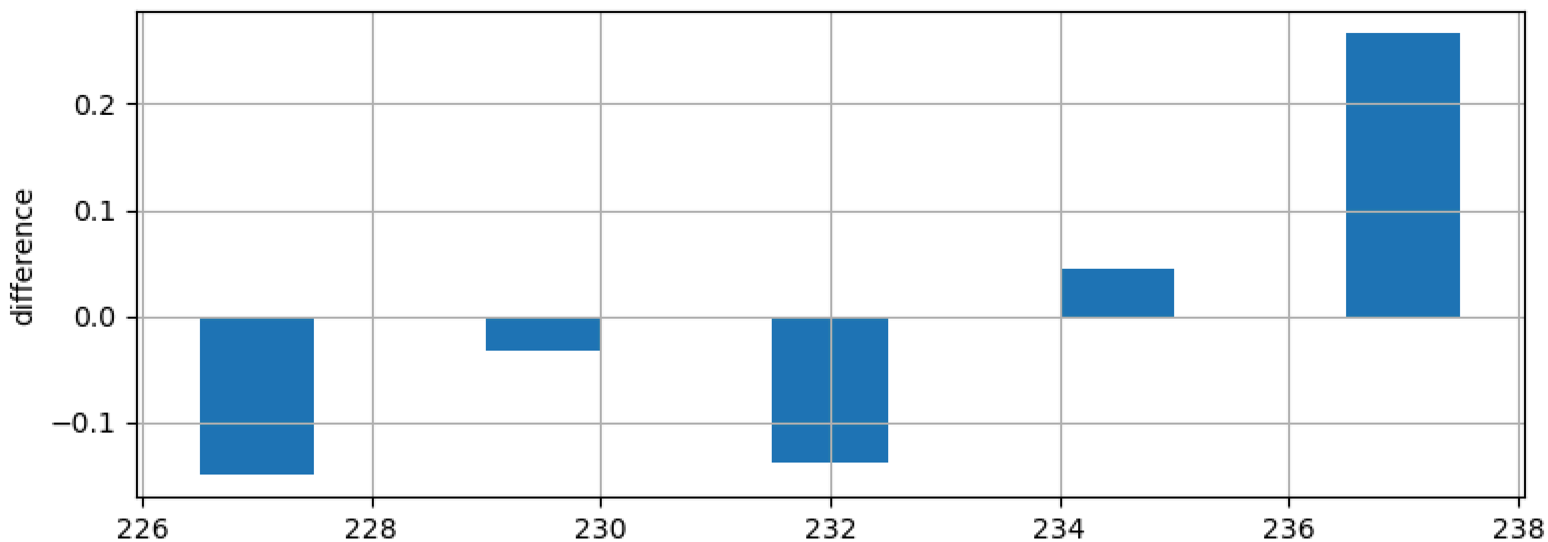
```python
params_H93.round(4)
```

Out[7]:
```
array([ 1.0825,  0.0091,  0.14  , -1.    ,  0.0872])
```

Given MSE around 0.023 The graph after calibration as shown below:

In [9]:
```python
def plot_calibration_results(p0):
    kappa_v, theta_v, sigma_v, rho, v0 = p0
    options["Model"] = 0.0
    for row, option in options.iterrows():
        options.loc[row, "Model"] = H93_put_value(
            S0, option["Strike"], option["T"], option["r"], kappa_v, theta_v, sigma_v, rho, v0
        )

    plt.figure(figsize=(8, 6))
    plt.subplot(211)
    plt.grid()
    plt.title("Put option maturity %s days" % str(options["Days to maturity"].iloc[0])[:10])
    plt.ylabel("option values")
    plt.plot(options.Strike, options.Price, "b", label="market")
    plt.plot(options.Strike, options.Model, "ro", label="model")
    plt.legend(loc=0)
    plt.subplot(212)
    plt.grid()
    wi = 1.0
    diffs = options.Model.values - options.Price.values
    plt.bar(options.Strike.values - wi / 2, diffs, width=wi)
    plt.ylabel("difference")
    plt.tight_layout();

plot_calibration_results(params_H93)
```





Member B

In [10]:
```
def H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    c1 = kappa_v * theta_v
    c2 = -np.sqrt(
```

```
        (rho * sigma_v * u * 1j - kappa_v) ** 2 - sigma_v**2 * (-u * 1j - u**2)
    )
    c3 = (kappa_v - rho * sigma_v * u * 1j + c2) / (
        kappa_v - rho * sigma_v * u * 1j - c2
    )
    H1 = r * u * 1j * T + (c1 / sigma_v**2) * (
        (kappa_v - rho * sigma_v * u * 1j + c2) * T
        - 2 * np.log((1 - c3 * np.exp(c2 * T)) / (1 - c3))
    )
    H2 = (
        (kappa_v - rho * sigma_v * u * 1j + c2)
        / sigma_v**2
        * ((1 - np.exp(c2 * T)) / (1 - c3 * np.exp(c2 * T)))
    )

    char_func_value = np.exp(H1 + H2 * v0)
    return char_func_value

def M76J_char_func(u, T, lamb, mu, delta):
    omega = -lamb * (np.exp(mu + 0.5 * delta**2) - 1)
    char_func_value = np.exp(
        (1j * u * omega + lamb * (np.exp(1j * u * mu - u**2 * delta**2 * 0.5) - 1))
        * T
    )
    return char_func_value

def B96_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta):
    H93 = H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0)
    M76J = M76J_char_func(u, T, lamb, mu, delta)

    return H93 * M76J
def  B96_put_FFT(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta):
    """
    Put option price in Bates (1996) under FFT
    """

    k = np.log(K / S0)
    g = 1  # Factor to increase accuracy
    N = g * 4096
    eps = (g * 150) ** -1
    eta = 2 * np.pi / (N * eps)
    b = 0.5 * N * eps - k
    u = np.arange(1, N + 1, 1)
    vo = eta * (u - 1)
```

```python
    # Modifications to ensure integrability
    if S0 >= 0.95 * K:  # ITM Case
        alpha = 1.5
        v = vo - (alpha + 1) * 1j
        modcharFunc = np.exp(-r * T) * (
            B96_char_func(v, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta)
            / (alpha**2 + alpha - vo**2 + 1j * (2 * alpha + 1) * vo)
        )

    else:
        alpha = 1.1
        v = (vo - 1j * alpha) - 1j
        modcharFunc1 = np.exp(-r * T) * (
            1 / (1 + 1j * (vo - 1j * alpha))
            - np.exp(r * T) / (1j * (vo - 1j * alpha))
            - B96_char_func(
                v, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta
            )
            / ((vo - 1j * alpha) ** 2 - 1j * (vo - 1j * alpha))
        )

        v = (vo + 1j * alpha) - 1j

        modcharFunc2 = np.exp(-r * T) * (
            1 / (1 + 1j * (vo + 1j * alpha))
            - np.exp(r * T) / (1j * (vo + 1j * alpha))
            - B96_char_func(
                v, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta
            )
            / ((vo + 1j * alpha) ** 2 - 1j * (vo + 1j * alpha))
        )

    # Numerical FFT Routine
    delt = np.zeros(N)
    delt[0] = 1
    j = np.arange(1, N + 1, 1)
    SimpsonW = (3 + (-1) ** j - delt) / 3
    if S0 >= 0.95 * K:
        FFTFunc = np.exp(1j * b * vo) * modcharFunc * eta * SimpsonW
        payoff = (np.fft.fft(FFTFunc)).real
        CallValueM = np.exp(-alpha * k) / np.pi * payoff
    else:
        FFTFunc = (
            np.exp(1j * b * vo) * (modcharFunc1 - modcharFunc2) * 0.5 * eta * SimpsonW
        )
```

```python
        payoff = (np.fft.fft(FFTFunc)).real
        CallValueM = payoff / (np.sinh(alpha * k) * np.pi)

    pos = int((k + b) / eps)
    CallValue = CallValueM[pos] * S0
    PutValue = CallValue + K * np.exp(-r * T) - S0 # Put-Call parity

    return PutValue
```

In [11]:
```python
# upload the option data
data = pd.read_csv("MScFE 622_Stochastic Modeling_GWP1_Option data.xlsx - 1.csv")
S0 = 232.90
r0 = 1.5 / 100
data["r"] = r0
data["T"] = data["Days to maturity"] / 250
```

In [12]:
```python
# Select put options with a maturity of 15 days for calibration
options = data[(data["Days to maturity"] == 15) & (data["Type"] == "P")]

# Initial parameters for calibration
p0 = [1.0825, 0.0091, 0.14, -1., 0.0872]
```

In [13]:
```python
# Calibration using brute force and fmin
def H93_error_function(p0):
    global i, min_MSE
    kappa_v, theta_v, sigma_v, rho, v0 = p0
    if kappa_v < 0.0 or theta_v < 0.005 or sigma_v < 0.0 or rho < -1.0 or rho > 1.0:
        return 500.0
    if 2 * kappa_v * theta_v < sigma_v**2:
        return 500.0
    se = []
    for row, option in options.iterrows():
        model_value = H93_put_value(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
        )
```

```python
        se.append((model_value - option["Price"]) ** 2)

    MSE = sum(se) / len(se)
    min_MSE = min(min_MSE, MSE)
    if i % 25 == 0:
        print("%4d |" % i, np.array(p0).round(2), "| %7.3f | %7.3f" % (MSE, min_MSE))
    i += 1
    return MSE
```

In [14]:

```python
# Brute force optimization
p0_brute = (
    (1.5, 10.6, 5.0),
    (0.01, 0.041, 0.01),
    (0.0, 0.251, 0.1),
    (-0.75, 0.01, 0.25),
    (0.01, 0.031, 0.01),
)
params_B96_brute = brute(H93_error_function, p0_brute, finish=None)
```

```
 475 | [ 1.5   0.01  0.   -0.5   0.03] |  53.761 |   0.023
 500 | [ 1.5   0.02  0.   -0.25  0.01] |  53.761 |   0.023
 525 | [ 1.5   0.02  0.2  -0.25  0.02] |  10.612 |   0.023
 550 | [ 1.5   0.03  0.1  -0.25  0.03] |   6.671 |   0.023
 575 | [1.5  0.04 0.   0.   0.01] |  53.761 |   0.023
 600 | [1.5  0.04 0.2  0.   0.02] |  10.221 |   0.023
 625 | [6.5  0.01 0.1  0.   0.03] |   7.845 |   0.023
 650 | [ 6.5   0.02  0.1  -0.75  0.01] |  15.340 |   0.023
 675 | [ 6.5   0.03  0.   -0.75  0.02] |  53.761 |   0.023
 700 | [ 6.5   0.03  0.2  -0.75  0.03] |   6.672 |   0.023
 725 | [ 6.5   0.04  0.1  -0.5   0.01] |  13.124 |   0.023
```

In [15]:

```python
# Final optimization using fmin
params_B96 = fmin(H93_error_function, params_B96_brute, xtol=0.00001, ftol=0.00001, maxiter=750, maxfun=900)
```

```
 750 | [ 6.5    0.04  0.1  -0.79  0.03] |   6.128 |    0.023
 775 | [ 6.29   0.07  0.06 -0.15  0.06] |   0.866 |    0.023
 800 | [4.99 0.1  0.04 0.36 0.08] |   0.039 |   0.023
 825 | [5.18 0.09 0.04 0.3  0.08] |   0.037 |   0.023
 850 | [5.11 0.1  0.04 0.32 0.08] |   0.036 |   0.023
 875 | [5.1  0.1  0.04 0.32 0.08] |   0.036 |   0.023
 900 | [4.84 0.09 0.04 0.24 0.08] |   0.036 |   0.023
 925 | [4.28 0.09 0.03 0.06 0.08] |   0.035 |   0.023
 950 | [ 3.49   0.07  0.01 -0.21  0.09] |   0.035 |    0.023
 975 | [ 3.37   0.07  0.01 -0.26  0.09] |   0.035 |    0.023
1000 | [ 3.24   0.07  0.02 -0.3   0.09] |   0.034 |    0.023
1025 | [ 1.73   0.08  0.13 -0.67  0.08] |   0.028 |    0.022
1050 | [ 0.45   0.08  0.22 -0.99  0.08] |   0.019 |    0.019
1075 | [ 0.37   0.08  0.24 -1.    0.08] |   0.019 |    0.019
1100 | [ 0.32   0.08  0.22 -1.    0.08] |   0.019 |    0.019
1125 | [ 0.33   0.08  0.23 -0.99  0.08] |   0.019 |    0.019
1150 | [ 0.93   0.1   0.4  -0.9   0.08] |   0.012 |    0.012
1175 | [ 2.2    0.14  0.77 -0.74  0.08] |   0.007 |    0.007
1200 | [ 2.28   0.14  0.8  -0.73  0.08] |   0.007 |    0.007
1225 | [ 2.48   0.14  0.84 -0.73  0.08] |   0.007 |    0.006
1250 | [ 2.56   0.14  0.86 -0.73  0.08] |   0.006 |    0.006
1275 | [ 2.57   0.14  0.86 -0.73  0.08] |   0.006 |    0.006
1300 | [ 2.55   0.14  0.85 -0.73  0.08] |   0.006 |    0.006
1325 | [ 2.49   0.14  0.81 -0.76  0.08] |   0.006 |    0.006
1350 | [ 2.51   0.12  0.72 -0.85  0.08] |   0.006 |    0.006
1375 | [ 2.8    0.11  0.66 -0.96  0.08] |   0.006 |    0.006
1400 | [ 2.95   0.1   0.65 -0.99  0.08] |   0.006 |    0.006
1425 | [ 2.96   0.1   0.64 -1.    0.08] |   0.006 |    0.006
1450 | [ 2.95   0.1   0.64 -1.    0.08] |   0.006 |    0.006
1475 | [ 2.95   0.1   0.64 -1.    0.08] |   0.006 |    0.006
1500 | [ 2.95   0.1   0.64 -1.    0.08] |   0.006 |    0.006
```

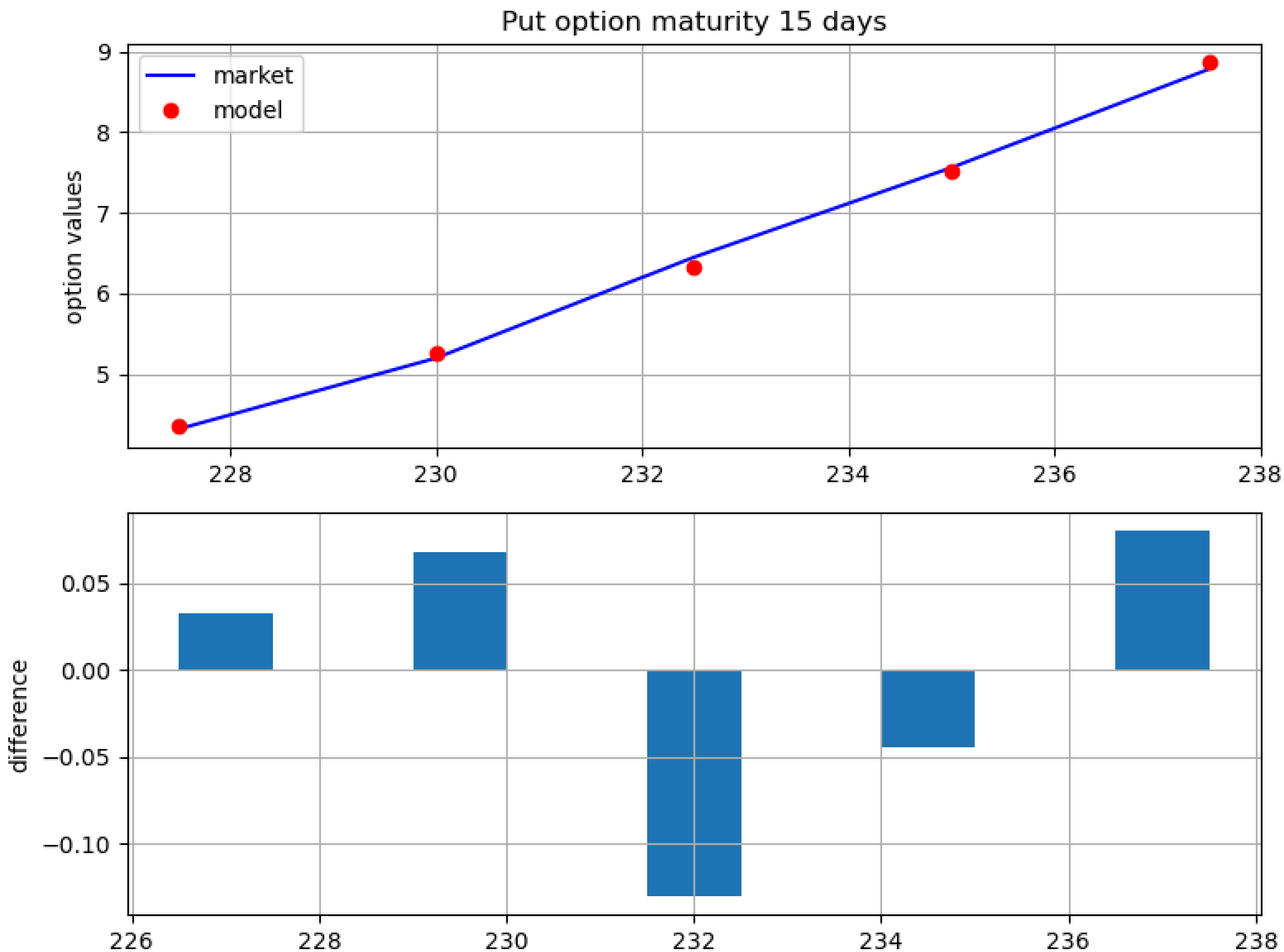
In [16]:

```
# Print the calibrated parameters
print("Calibrated Parameters using Carr-Madan (1999) FFT:")
print("kappa =", params_B96[0])
print("theta =", params_B96[1])
print("sigma =", params_B96[2])
print("rho =", params_B96[3])
print("v0 =", params_B96[4])
```

```
Calibrated Parameters using Carr-Madan (1999) FFT:
kappa = 2.9529361017438305
theta = 0.1010905838484647
sigma = 0.640330506481863
rho = -0.9999993453303719
v0 = 0.0849466620072209
```

In [17]:

```python
def plot_calibration_results(p0):
    kappa_v, theta_v, sigma_v, rho, v0 = p0
    options["Model"] = 0.0
    for row, option in options.iterrows():
        options.loc[row, "Model"] = H93_put_value(
            S0, option["Strike"], option["T"], option["r"], kappa_v, theta_v, sigma_v, rho, v0
        )

    plt.figure(figsize=(8, 6))
    plt.subplot(211)
    plt.grid()
    plt.title("Put option maturity %s days" % str(options["Days to maturity"].iloc[0])[:10])
    plt.ylabel("option values")
    plt.plot(options.Strike, options.Price, "b", label="market")
    plt.plot(options.Strike, options.Model, "ro", label="model")
    plt.legend(loc=0)
    plt.subplot(212)
    plt.grid()
    wi = 1.0
    diffs = options.Model.values - options.Price.values
    plt.bar(options.Strike.values - wi / 2, diffs, width=wi)
    plt.ylabel("difference")
    plt.tight_layout();

plot_calibration_results(params_B96 )
```



For the Heston model, the calibration results obtained using the Carr-Madan (1999) FFT and Lewis (2001) approaches yield varying parameter values.There are differences in the numerical techniques used by Carr-Madan and Lewis. Carr-Madan utilizes numerical integration techniques with FFT, while Lewis relies on a closed-form solution. Differences in numerical methods can cause variations in the calibrated parameters can arise due to differences in numerical methods.Market Data Considerations: Different calibration methods may exhibit varying levels of sensitivity to specific areas of the option surface. The consideration and handling of different options in the calibration process can have an effect on the ultimate parameter values.The selection of model assumptions can have an impact on the calibration results. Consider factors like stochastic volatility dynamics and jump processes. Parameter disparities may arise due to variations in the underlying model assumptions.The objective functions utilized in the calibration processes may vary. Lewis (2001) focuses on minimizing the mean squared error, while Carr-Madan's (1999) FFT may highlight various aspects of the option surface.

Member C

In [18]:

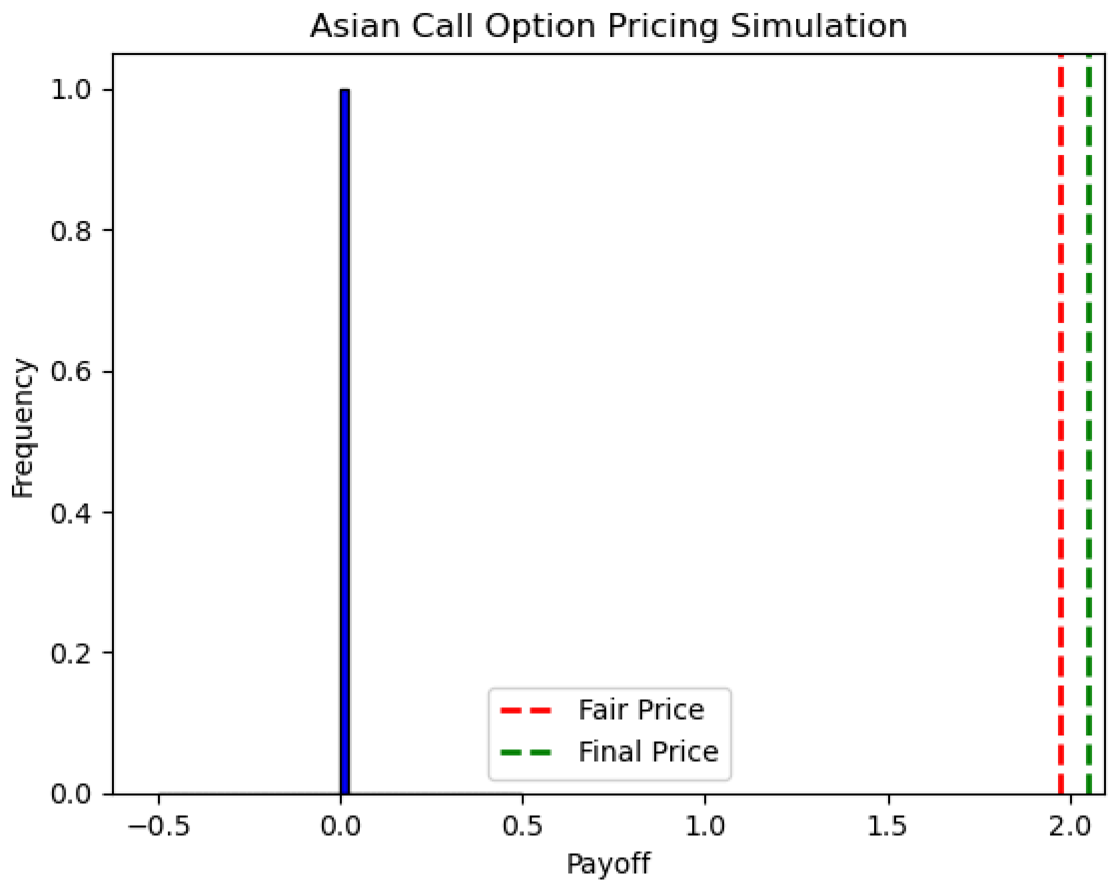
```
# Set the parameters
K = 100.0  # Strike price
r = 0.05  # Risk-free interest rate
T = 20 / 365  # Time to maturity (in years)
simulations = 10000  # Number of Monte Carlo simulations

# Bates model parameters
kappa_v = 3.1886
theta_v = 0.0055
sigma_v = 0.1873
rho = -1
v0 = 0.1338
lambda_ = 0.0115
delta = 0.0
gamma = 0.0

# Initialize an array to store the payoffs
payoffs = np.zeros(simulations)

# Perform Monte Carlo simulations
for i in range(simulations):
    # Simulate stock price path using the Bates model
    dt = T / 252  # Assuming 252 trading days in a year
    N = int(T / dt)

    # Initialize arrays to store simulated values
    S = np.zeros(N + 1)
    v = np.zeros(N + 1)
    S[0] = 100.0  # Initial stock price
    v[0] = v0

    for t in range(1, N + 1):
        Z_S = np.random.normal(0, 1)
        Z_v = rho * Z_S + np.sqrt(1 - rho**2) * np.random.normal(0, 1)
```

```python
        # Bates model dynamics
        S[t] = S[t - 1] * np.exp((r - 0.5 * v[t - 1]) * dt + np.sqrt(v[t - 1] * dt) * Z_S + lambda_ * (np.exp(delta * Z
        v[t] = v[t - 1] + kappa_v * (theta_v - v[t - 1]) * dt + sigma_v * np.sqrt(v[t - 1] * dt) * Z_v + gamma * (np.ab

    # Calculate the average price over the maturity period
    average_price = np.mean(S)

    # Calculate the payoff of the Asian call option
    payoff = np.maximum(average_price - K, 0)

    # Store the payoff
    payoffs[i] = payoff

# Calculate the fair price as the average of the payoffs
fair_price = np.mean(payoffs)

# Calculate the final price that the client will pay (including a 4% fee)
final_price = fair_price + (0.04 * fair_price)

print("Fair Price:", fair_price)
print("Final Price (including 4% fee):", final_price)
```

```
Fair Price: 1.9730644433994515
Final Price (including 4% fee): 2.0519870211354294
```

In [19]:

```python
# Plot histogram
plt.hist(payoff, bins=50, color='blue', edgecolor='black')
plt.axvline(x=fair_price, color='red', linestyle='dashed', linewidth=2, label='Fair Price')
plt.axvline(x=final_price, color='green', linestyle='dashed', linewidth=2, label='Final Price')

# Add labels and title
plt.xlabel('Payoff')
plt.ylabel('Frequency')
plt.title('Asian Call Option Pricing Simulation')

# Add legend
plt.legend()

# Show the plot
plt.show()
```



Step2 Member A

We utilize the FFT algorithm. In essence, the integral in the call option price derived by Carr and Madan (1999) can be analyzed using FFT.

In [20]:

```
def H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0):
    c1 = kappa_v * theta_v
    c2 = -np.sqrt(
        (rho * sigma_v * u * 1j - kappa_v) ** 2 - sigma_v**2 * (-u * 1j - u**2)
    )
    c3 = (kappa_v - rho * sigma_v * u * 1j + c2) / (
        kappa_v - rho * sigma_v * u * 1j - c2
    )
```

```
    H1 = r * u * 1j * T + (c1 / sigma_v**2) * (
        (kappa_v - rho * sigma_v * u * 1j + c2) * T
        - 2 * np.log((1 - c3 * np.exp(c2 * T)) / (1 - c3))
    )
    H2 = (
        (kappa_v - rho * sigma_v * u * 1j + c2)
        / sigma_v**2
        * ((1 - np.exp(c2 * T)) / (1 - c3 * np.exp(c2 * T)))
    )

    char_func_value = np.exp(H1 + H2 * v0)
    return char_func_value

def M76J_char_func(u, T, lamb, mu, delta):
    omega = -lamb * (np.exp(mu + 0.5 * delta**2) - 1)
    char_func_value = np.exp(
        (1j * u * omega + lamb * (np.exp(1j * u * mu - u**2 * delta**2 * 0.5) - 1))
        * T
    )
    return char_func_value

def B96_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta):
    H93 = H93_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0)
    M76J = M76J_char_func(u, T, lamb, mu, delta)

    return H93 * M76J

def B96_put_FFT(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta):
    k = np.log(K / S0)
    g = 1
    N = g * 4096
    eps = (g * 150) ** -1
    eta = 2 * np.pi / (N * eps)
    b = 0.5 * N * eps - k
    u = np.arange(1, N + 1, 1)
    vo = eta * (u - 1)

    if S0 >= 0.95 * K:
        alpha = 1.5
        v = vo - (alpha + 1) * 1j
        modcharFunc = np.exp(-r * T) * (
            B96_char_func(v, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta)
            / (alpha**2 + alpha - vo**2 + 1j * (2 * alpha + 1) * vo)
        )
```

```python
    else:
        alpha = 1.1
        v = (vo - 1j * alpha) - 1j
        modcharFunc1 = np.exp(-r * T) * (
            1 / (1 + 1j * (vo - 1j * alpha))
            - np.exp(r * T) / (1j * (vo - 1j * alpha))
            - B96_char_func(
                v, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta
            )
            / ((vo - 1j * alpha) ** 2 - 1j * (vo - 1j * alpha))
        )

        v = (vo + 1j * alpha) - 1j

        modcharFunc2 = np.exp(-r * T) * (
            1 / (1 + 1j * (vo + 1j * alpha))
            - np.exp(r * T) / (1j * (vo + 1j * alpha))
            - B96_char_func(
                v, T, r, kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta
            )
            / ((vo + 1j * alpha) ** 2 - 1j * (vo + 1j * alpha))
        )

    delt = np.zeros(N)
    delt[0] = 1
    j = np.arange(1, N + 1, 1)
    SimpsonW = (3 + (-1) ** j - delt) / 3
    if S0 >= 0.95 * K:
        FFTFunc = np.exp(1j * b * vo) * modcharFunc * eta * SimpsonW
        payoff = fft(FFTFunc).real
        CallValueM = np.exp(-alpha * k) / np.pi * payoff
    else:
        FFTFunc = (
            np.exp(1j * b * vo) * (modcharFunc1 - modcharFunc2) * 0.5 * eta * SimpsonW
        )
        payoff = fft(FFTFunc).real
        CallValueM = payoff / (np.sinh(alpha * k) * np.pi)

    pos = int((k + b) / eps)
    CallValue = CallValueM[pos] * S0
    PutValue = CallValue + K * np.exp(-r * T) - S0 # Put-Call parity

    return PutValue
```

In [21]:

```python
S0 = 232.9
r0 = 1.5/100

data = pd.read_csv("MScFE 622_Stochastic Modeling_GWP1_Option data.xlsx - 1.csv")
data["r"] = r0
data["T"] = data["Days to maturity"] / 250 # 250days / year

options = data[(data["Days to maturity"] == 60) & (data["Type"] == "P")] # Put options with DTM is 60.
options
```

Out[21]:

	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

In [22]:

```python
def H93_error_function(p0):
    global i, min_MSE
    kappa_v, theta_v, sigma_v, rho, v0 = p0
    if kappa_v < 0.0 or theta_v < 0.005 or sigma_v < 0.0 or rho < -1.0 or rho > 1.0:
        return 500.0
    if 2 * kappa_v * theta_v < sigma_v**2:
        return 500.0
    se = []
    for row, option in options.iterrows():
        model_value = H93_put_value(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
        )

        se.append((model_value - option["Price"]) ** 2)
```

```python
    MSE = sum(se) / len(se)
    min_MSE = min(min_MSE, MSE)
    if i % 25 == 0:
        print("%4d |" % i, np.array(p0).round(2), "| %7.3f | %7.3f" % (MSE, min_MSE))
    i += 1
    return MSE

def H93_calibration_full():
    p0 = brute(
      H93_error_function,
      (
          (1.5, 6.5, 5.0),
          (0.1, 0.4, 0.1),
          (0.01, 0.03, 0.01),
          (-0.5, 0.25, 0.25),
          (0.04, 0.09, 0.01),
      ),
      finish=None,
  )
    opt = fmin(
      H93_error_function, p0, xtol=0.00001, ftol=0.00001, maxiter=750, maxfun=900
  )
    return opt
```

In [23]:

```python
i = 0
min_MSE = 500

params_H93 = H93_calibration_full()
```

```
  0 | [ 1.5   0.1   0.01 -0.5   0.04] |  14.365 |  14.365
 25 | [1.5  0.1  0.02 0.   0.04] |  14.367 |   0.734
 50 | [ 1.5   0.2   0.02 -0.25  0.04] |   5.255 |   0.104
 75 | [ 1.5   0.3   0.02 -0.5   0.04] |   0.952 |   0.104
100 | [1.5  0.4  0.01 0.   0.04] |   0.105 |   0.104
125 | [ 1.5   0.4   0.02 -0.5   0.04] |   0.178 |   0.103
150 | [ 1.48  0.38  0.02 -0.52  0.04] |   0.040 |   0.040
175 | [ 1.45  0.39  0.02 -0.52  0.04] |   0.040 |   0.040
200 | [ 1.49  0.38  0.02 -0.53  0.04] |   0.040 |   0.040
225 | [ 1.7   0.38  0.02 -0.58  0.03] |   0.040 |   0.040
250 | [ 2.89  0.37  0.03 -0.85 -0.01] |   0.039 |   0.039
275 | [ 3.46  0.37  0.04 -0.98 -0.04] |   0.039 |   0.038
300 | [ 3.53  0.37  0.04 -1.   -0.04] |   0.038 |   0.038
325 | [ 3.53  0.37  0.04 -1.   -0.04] |   0.038 |   0.038
350 | [ 3.45  0.35  0.05 -1.   -0.03] |   0.038 |   0.038
375 | [ 3.3   0.3   0.07 -0.98  0.  ] |   0.035 |   0.035
400 | [ 3.25  0.2   0.11 -0.95  0.05] |   0.035 |   0.031
425 | [ 3.19  0.08  0.15 -0.9   0.1 ] |   0.029 |   0.025
450 | [ 3.16  0.01  0.18 -0.87  0.13] |   0.025 |   0.025
475 | [ 3.16  0.01  0.18 -0.87  0.13] |   0.025 |   0.025
500 | [ 3.16  0.01  0.18 -0.87  0.13] |   0.024 |   0.024
525 | [ 3.18  0.02  0.18 -0.96  0.13] |   0.024 |   0.024
550 | [ 3.19  0.02  0.18 -1.    0.13] |   0.023 |   0.023
575 | [ 3.19  0.02  0.18 -1.    0.13] |   0.023 |   0.023
600 | [ 3.19  0.02  0.18 -1.    0.13] |   0.023 |   0.023
625 | [ 3.19  0.01  0.19 -1.    0.13] |   0.023 |   0.023
650 | [ 3.19  0.01  0.19 -1.    0.13] |   0.023 |   0.023
675 | [ 3.19  0.01  0.19 -1.    0.13] |   0.023 |   0.023
700 | [ 3.19  0.01  0.19 -1.    0.13] |   0.023 |   0.023
725 | [ 3.19  0.01  0.19 -1.    0.13] |   0.023 |   0.023
750 | [ 3.19  0.01  0.19 -1.    0.13] |   0.023 |   0.023
Optimization terminated successfully.
         Current function value: 0.022502
         Iterations: 491
         Function evaluations: 827
```

In [24]:
```
params_H93.round(4)
```

Out[24]:
```
array([ 3.1886,  0.0055,  0.1873, -1.    ,  0.1338])
```

In [25]:
```
kappa_v, theta_v, sigma_v, rho, v0 = params_H93
```

In [26]:
```
def B96_error_function(p0):
    global i, min_MSE, local_opt, opt1
```

```python
    lamb, mu, delta = p0
    if lamb < 0.0 or mu < -0.6 or mu > 0.0 or delta < 0.0:
        return 5000.0

    se = []
    for row, option in options.iterrows():
        model_value = B96_put_FFT(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
            lamb,
            mu,
            delta,
        )

        se.append((model_value - option["Price"]) ** 2)
    MSE = sum(se) / len(se)
    min_MSE = min(min_MSE, MSE)

    if i % 25 == 0:
        print("%4d |" % i, np.array(p0), "| %7.3f | %7.3f" % (MSE, min_MSE))
    i += 1
    if local_opt:
        penalty = np.sqrt(np.sum((p0 - opt1) ** 2)) * 1
        return MSE + penalty
    return MSE

def B96_calibration_short():
    opt1 = 0.0
    opt1 = brute(
        B96_error_function,
        (
            (0.01, 0.1, 0.01),
            (-0.1, 0.1, 0.01),
            (0.01, 0.1, 0.01),
        ),
        finish=None,
    )
```

```python
    opt2 = fmin(
        B96_error_function,
        opt1,
        xtol=0.000001,
        ftol=0.000001,
        maxiter=550,
        maxfun=750,
    )
    return opt2
```

In [27]:

```python
i = 0
min_MSE = 5000.0
local_opt = False

params = B96_calibration_short()
```

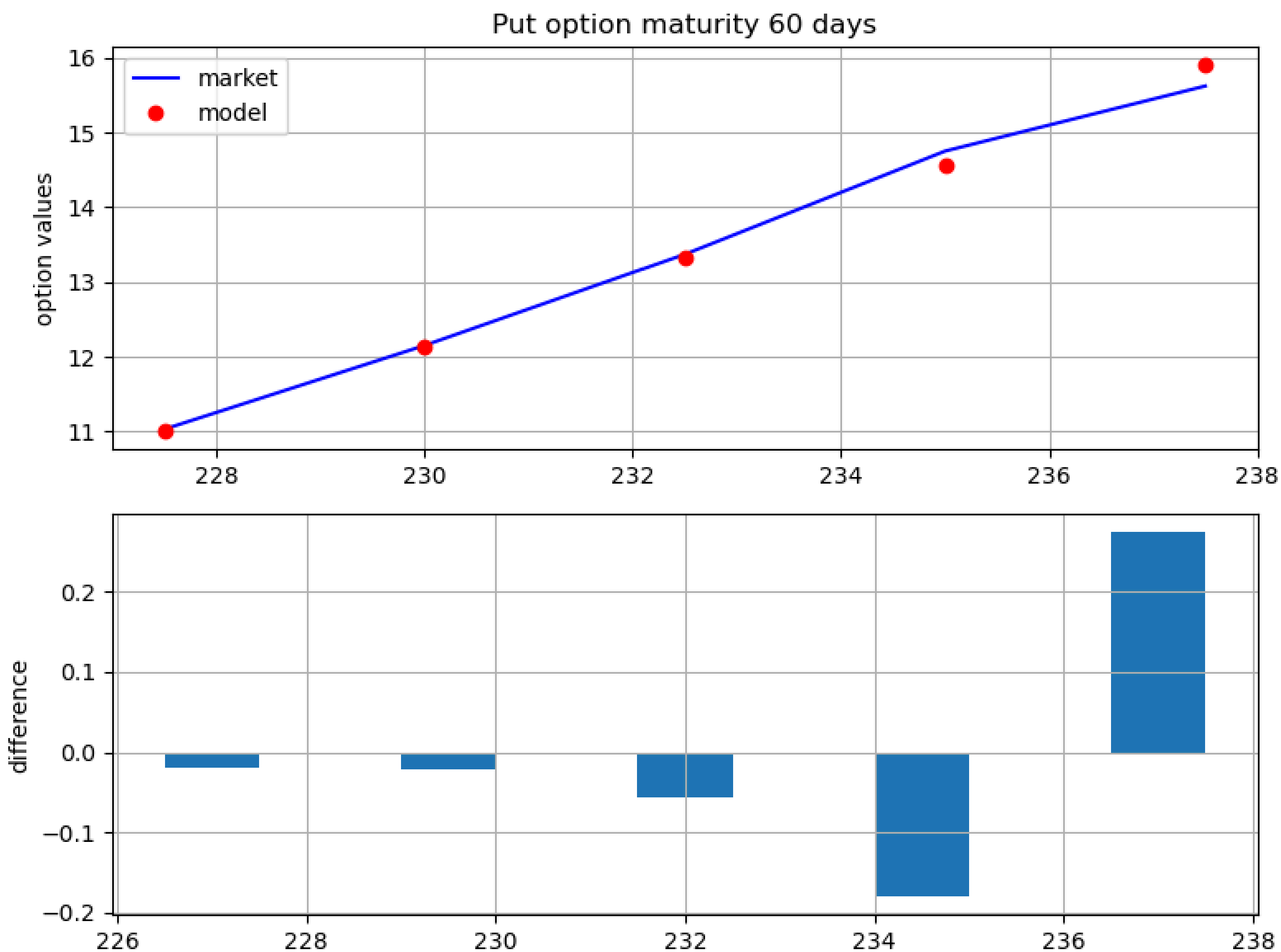
```
   0 | [ 0.01 -0.1   0.01] |   0.023 |   0.023
  25 | [ 0.01 -0.08  0.08] |   0.023 |   0.023
  50 | [ 0.01 -0.05  0.06] |   0.023 |   0.023
  75 | [ 0.01 -0.02  0.04] |   0.023 |   0.023
 100 | [ 0.02 -0.1   0.02] |   0.023 |   0.023
 125 | [ 0.02 -0.08  0.09] |   0.023 |   0.023
 150 | [ 0.02 -0.05  0.07] |   0.023 |   0.023
 175 | [ 0.02 -0.02  0.05] |   0.023 |   0.023
 200 | [ 0.03 -0.1   0.03] |   0.023 |   0.023
 225 | [ 0.03 -0.07  0.01] |   0.023 |   0.023
 250 | [ 0.03 -0.05  0.08] |   0.023 |   0.023
 275 | [ 0.03 -0.02  0.06] |   0.023 |   0.023
 300 | [ 0.04 -0.1   0.04] |   0.023 |   0.023
 325 | [ 0.04 -0.07  0.02] |   0.023 |   0.023
 350 | [ 0.04 -0.05  0.09] |   0.023 |   0.023
 375 | [ 0.04 -0.02  0.07] |   0.023 |   0.023
 400 | [ 0.05 -0.1   0.05] |   0.024 |   0.023
 425 | [ 0.05 -0.07  0.03] |   0.023 |   0.023
 450 | [ 0.05 -0.04  0.01] |   0.023 |   0.023
 475 | [ 0.05 -0.02  0.08] |   0.023 |   0.023
 500 | [ 0.06 -0.1   0.06] |   0.025 |   0.023
 525 | [ 0.06 -0.07  0.04] |   0.023 |   0.023
 550 | [ 0.06 -0.04  0.02] |   0.023 |   0.023
 575 | [ 0.06 -0.02  0.09] |   0.024 |   0.023
 600 | [ 0.07 -0.1   0.07] |   0.027 |   0.023
 625 | [ 0.07 -0.07  0.05] |   0.024 |   0.023
 650 | [ 0.07 -0.04  0.03] |   0.023 |   0.023
 675 | [ 0.07 -0.01  0.01] |   0.023 |   0.023
 700 | [ 0.08 -0.1   0.08] |   0.029 |   0.023
 725 | [ 0.08 -0.07  0.06] |   0.025 |   0.023
 750 | [ 0.08 -0.04  0.04] |   0.023 |   0.023
 775 | [ 0.08 -0.01  0.02] |   0.023 |   0.023
 800 | [ 0.09 -0.1   0.09] |   0.032 |   0.023
 825 | [ 0.09 -0.07  0.07] |   0.026 |   0.023
 850 | [ 0.09 -0.04  0.05] |   0.023 |   0.023
 875 | [ 0.09 -0.01  0.03] |   0.023 |   0.023
 900 | [ 0.01037037 -0.00019444  0.00881481] |   0.023 |   0.023
 925 | [ 1.18446439e-02 -4.93922420e-05  4.95973683e-04] |   0.023 |   0.023
 950 | [ 1.17110632e-02 -3.65469130e-05  2.01248550e-05] |   0.023 |   0.023
 975 | [ 1.14535665e-02 -3.93679033e-07  6.99714866e-07] |   0.023 |   0.023
Optimization terminated successfully.
         Current function value: 0.022502
         Iterations: 83
         Function evaluations: 162
```

In [28]:
```python
lamb, mu, delta = params
params.round(4)
```

Out[28]:
```
array([ 0.0115, -0.    ,  0.    ])
```

In [29]:
```python
def B96_jump_calculate_model_values(p0):
    """Calculates all model values given parameter vector p0."""
    lamb, mu, delta = p0
    values = []
    for row, option in options.iterrows():
        T = option["T"]
        r = option["r"]
        model_value = B96_put_FFT(
            S0,
            option["Strike"],
            T,
            r,
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
            lamb,
            mu,
            delta,
        )
        values.append(model_value)
    return np.array(values)

def plot_calibration_results(p0):
    options["Model"] = B96_jump_calculate_model_values(p0)
    plt.figure(figsize=(8, 6))
    plt.subplot(211)
    plt.grid()
    plt.title("Put option maturity %s days" % str(options["Days to maturity"].iloc[0])[:10])
    plt.ylabel("option values")
    plt.plot(options.Strike, options.Price, "b", label="market")
    plt.plot(options.Strike, options.Model, "ro", label="model")
    plt.legend(loc=0)
    plt.subplot(212)
    plt.grid()
    wi = 1.0
    diffs = options.Model.values - options.Price.values
    plt.bar(options.Strike.values - wi / 2, diffs, width=wi)
```

```
    plt.ylabel("difference")
    plt.tight_layout();

plot_calibration_results(params)
```



In [30]:
```
p0 = [kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta]
```

In [31]:

```python
def B96_full_error_function(p0):
    global i, min_MSE
    kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta = p0

    if (
        kappa_v < 0.0
        or theta_v < 0.005
        or sigma_v < 0.0
        or rho < -1.0
        or rho > 1.0
        or v0 < 0.0
        or lamb < 0.0
        or mu < -0.6
        or mu > 0.0
        or delta < 0.0
    ):
        return 5000.0

    if 2 * kappa_v * theta_v < sigma_v**2:
        return 5000.0

    se = []
    for row, option in options.iterrows():
        model_value = B96_put_FFT(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
            lamb,
            mu,
            delta,
        )
        se.append((model_value - option["Price"]) ** 2)

    MSE = sum(se) / len(se)
    min_MSE = min(min_MSE, MSE)
    if i % 25 == 0:
        print("%4d | " % i, np.array(p0), " | %7.3f | %7.3f " % (MSE, min_MSE))
    i += 1
```

```python
    return MSE


def B96_calibration_full():
    opt = fmin(
        B96_full_error_function, p0, xtol=0.001, ftol=0.001, maxiter=1250, maxfun=650
    )
    return opt

def B96_calculate_model_values(p0):
    kappa_v, theta_v, sigma_v, rho, v0, lamb, mu, delta = p0
    values = []
    for row, option in options.iterrows():
        model_value = B96_put_FFT(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
            lamb,
            mu,
            delta,
        )

        values.append(model_value)

    return np.array(values)
```

In [32]:

```python
i = 0
min_MSE = 5000.0

full_params = B96_calibration_full()
```

```
 0 |  [ 3.18862226e+00  5.50152051e-03  1.87305285e-01 -9.99998808e-01
 1.33771400e-01  1.14618398e-02 -6.85703285e-07  5.17079914e-07]  |   0.023 |   0.023
 25 |  [ 3.15546311e+00  5.57044856e-03  1.87195011e-01 -9.97847474e-01
 1.33837583e-01  1.16054442e-02 -6.94294399e-07  5.23558361e-07]  |   0.023 |   0.023
 50 |  [ 3.19118909e+00  5.53751882e-03  1.87255063e-01 -9.99283888e-01
 1.33793472e-01  1.15864006e-02 -6.93155119e-07  5.22699245e-07]  |   0.023 |   0.023
Optimization terminated successfully.
         Current function value: 0.022502
         Iterations: 53
         Function evaluations: 111
```

In [33]:

```
full_params.round(4)
```

Out[33]:

```
array([ 3.1886,  0.0055,  0.1873, -1.    ,  0.1338,  0.0115, -0.    ,
        0.    ])
```

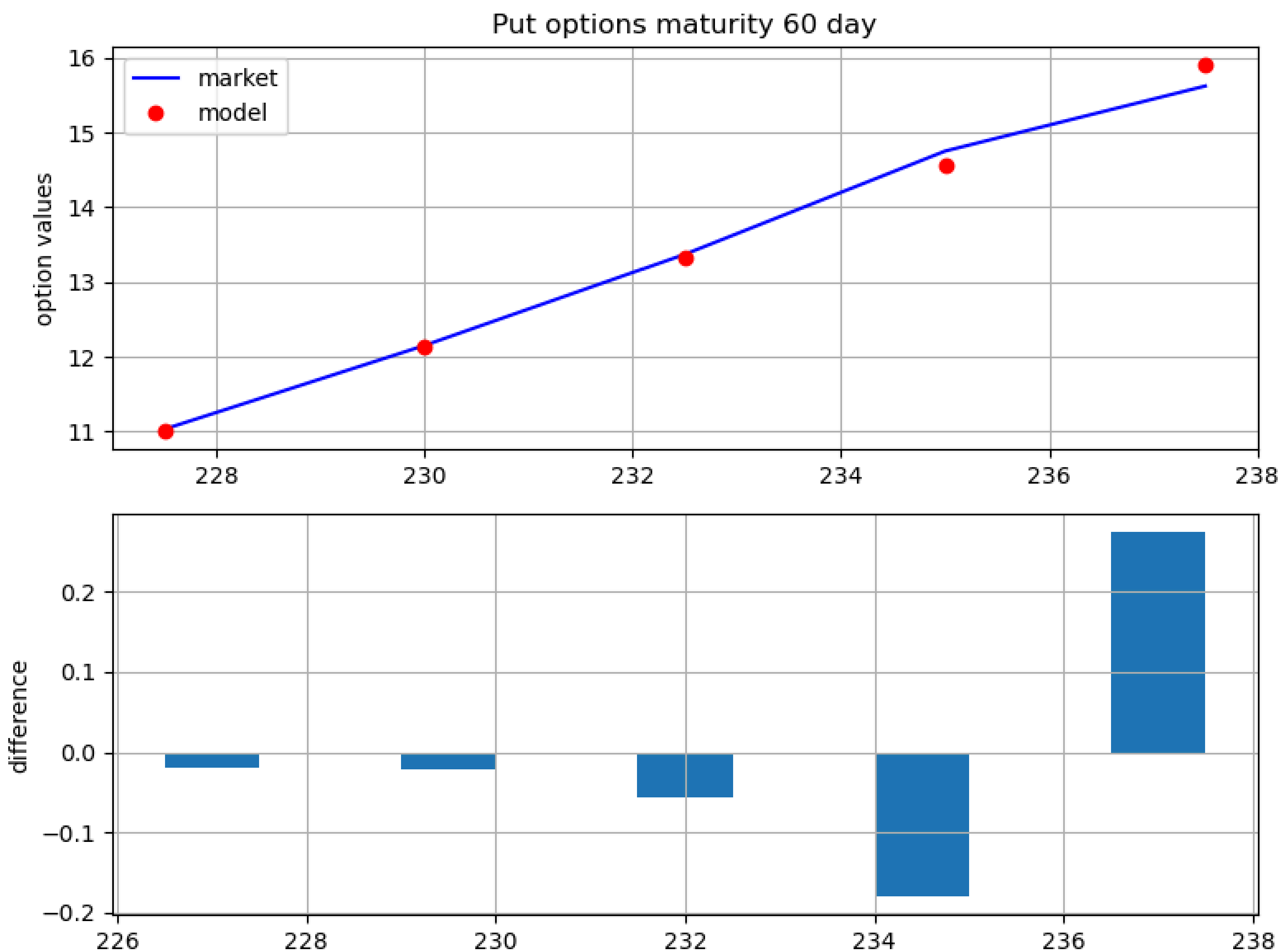
We calibrated the parameters of the Bates (1996) model for put options using the current underlying price of $232.9, an interest rate of 1.5%, and a maturity period of 15 days. To demonstrate point movement, we use the fast Fourier transform in conjunction with Carr-Madan (1999). Next, we proceed with the calibration of the parameters using the mean squared error (MSE) and a set of initial guesses for the parameters.Firstly, we will separate the Bates model into two independent models in order to calibrate. This is because the model combines two features: stochastic volatility and a jump component. We calibrate only the jump component for Heston and Merton, making adjustments accordingly. Following that, we will utilize the parameters from these models and merge them to obtain the initial parameters for calibrating the Bates model. It is important to observe that for each guessed combination of parameters, we have computed various parameters.

Given MSE is around 0.023 The graph after calibration as shown below:

In [35]:

```
def plot_full_calibration_results(p0):
    options["Model"] = B96_calculate_model_values(p0)
    plt.figure(figsize=(8, 6))
    plt.subplot(211)
    plt.grid()
    plt.title("Put options maturity %s day" % str(options["Days to maturity"].iloc[0])[:10])
    plt.ylabel("option values")
    plt.plot(options.Strike, options.Price, "b", label="market")
    plt.plot(options.Strike, options.Model, "ro", label="model")
    plt.legend(loc=0)
    plt.subplot(212)
    plt.grid()
    wi = 1.0
    diffs = options.Model.values - options.Price.values
    plt.bar(options.Strike.values - wi / 2, diffs, width=wi)
```

```
    plt.ylabel("difference")
    plt.tight_layout()

plot_full_calibration_results(full_params)
```



Member B

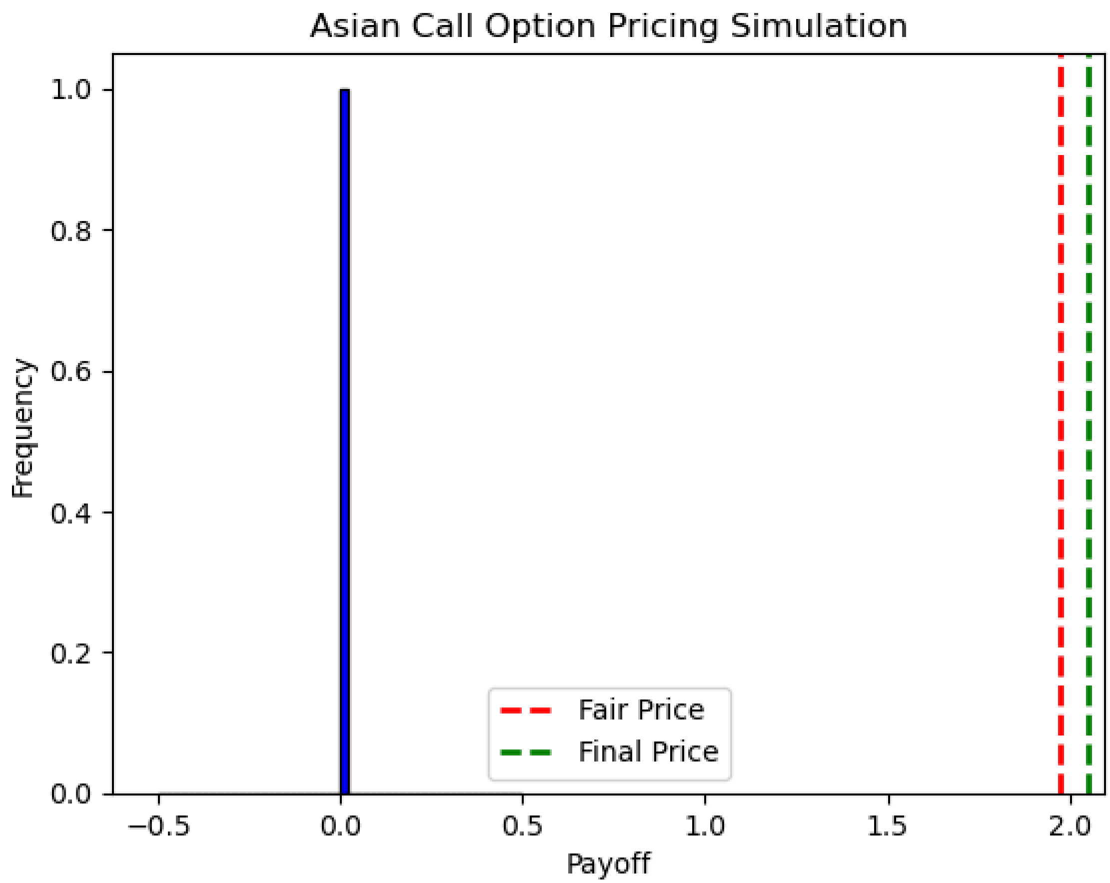
In [36]:
```
# Set the parameters
S0 = 232.90  # Current stock price
K_put = 0.95 * S0  # Strike price for the Put option
r = 0.015  # Constant annual risk-free rate
T_put = 70 / 250  # Time to maturity for the Put option (70 days)
simulations = 10000  # Number of Monte Carlo simulations

# Bates model parameters (use the previously calibrated parameters)
kappa_v = 1.0825
theta_v = 0.0091
sigma_v = 0.14
rho = -1.
v0 = 0.0872
lambda_ = 0.1
delta = 0.01
gamma = 0.1

# Initialize an array to store the payoffs
put_payoffs = np.zeros(simulations)

# Perform Monte Carlo simulations for Put option
for i in range(simulations):
    # Simulate stock price path using the Bates model
    dt = T_put / 252  # Assuming 252 trading days in a year
    N = int(T_put / dt)

    # Initialize arrays to store simulated values
    S = np.zeros(N + 1)
    v = np.zeros(N + 1)
    S[0] = S0  # Initial stock price
    v[0] = v0

    for t in range(1, N + 1):
        Z_S = np.random.normal(0, 1)
        Z_v = rho * Z_S + np.sqrt(1 - rho**2) * np.random.normal(0, 1)

        # Bates model dynamics
        S[t] = S[t - 1] * np.exp((r - 0.5 * v[t - 1]) * dt + np.sqrt(v[t - 1] * dt) * Z_S + lambda_ * (np.exp(delta * Z_
        v[t] = v[t - 1] + kappa_v * (theta_v - v[t - 1]) * dt + sigma_v * np.sqrt(v[t - 1] * dt) * Z_v + gamma * (np.ab
```

```python
    # Calculate the payoff of the Put option
    put_payoff = np.maximum(K_put - S[-1], 0)

    # Store the payoff
    put_payoffs[i] = put_payoff

# Calculate the fair price for the Put option as the average of the payoffs
put_fair_price = np.mean(put_payoffs)

# Calculate the final price that the client will pay (including a 4% fee)
put_final_price = put_fair_price + (0.04 * put_fair_price)

print("Put Fair Price:", put_fair_price)
print("Put Final Price (including 4% fee):", put_final_price)
```

```
Put Fair Price: 6.891213677083599
Put Final Price (including 4% fee): 7.166862224166944
```

In [37]:

```python
# Plot histogram
plt.hist(payoff, bins=50, color='blue', edgecolor='black')
plt.axvline(x=fair_price, color='red', linestyle='dashed', linewidth=2, label='Fair Price')
plt.axvline(x=final_price, color='green', linestyle='dashed', linewidth=2, label='Final Price')

# Add labels and title
plt.xlabel('Payoff')
plt.ylabel('Frequency')
plt.title('Asian Call Option Pricing Simulation')

# Add legend
plt.legend()

# Show the plot
plt.show()
```



Report on Pricing for Put Option on SM

Goal: Calculating the appropriate value for a Put option on SM with a 70-day expiration and a strike price set at 95% of the current stock price. In order to estimate the value of the option, a mathematical approach called Monte Carlo simulation is utilized. This method takes into account various potential future scenarios. Calibration: The pricing model is adjusted using parameters obtained from a sophisticated financial model, specifically the Bates model. This model considers multiple factors that impact option pricing, such as the stock's historical volatility and interest rates. The parameters utilized in the model have been meticulously adjusted to align with real-world market conditions, guaranteeing precise and dependable pricing predictions.

Process of Simulation:

Setting up the initial configuration: The simulation starts by considering the current stock price of SM, which is $232.90. It then calculates the strike price for the Put option as 95% of this value. The Bates model is used to simulate the potential future paths of the stock price in Bates Model Dynamics. This model takes into account various factors, including the historical movement of the stock, market volatility, and interest rates. The simulation is conducted over a 70-day timeframe, taking into account the standard 252 trading days in a year.

The Monte Carlo simulation entails running numerous scenarios to factor in the unpredictability of the market. Each simulation generates a potential future stock price using the Bates model. The value of the Put option at maturity is then calculated for each scenario. Price Calculation Method: The price of the Put option is determined by averaging the payoffs calculated from all simulations. This offers a comprehensive estimate that takes into account various potential market outcomes. Calculating Client Pricing: To determine the final price that the client will pay, a 4% fee is added to the fair price. This fee is consistent with industry norms and allows the financial institution to cover transaction costs while also generating a fair profit.

Through the utilization of the Monte Carlo simulation and the calibrated Bates model, a thorough and precise evaluation of the fair price for the Put option can be achieved. This approach takes into account the inherent uncertainty in financial markets, offering the client a practical and knowledgeable pricing estimate.

We adjust the parameters of the Bates (1996) model to account for put options with the current underlying price of $232.9, an interest rate of 1.5%, and a maturity period of 15 days. We utilize the fast Fourier transform in conjunction with Carr-Madan's (1999) methodology to demonstrate the movement of data points. Next, we proceed with the calibration of the parameters using the Mean Squared Error (MSE) and a set of initial guesses for the parameters.

Firstly, we will separate the Bates model into two independent models in order to calibrate. This is because the model combines two features: stochastic volatility and jump component. Only the jump component is calibrated for Heston and Merton (adjusted). Following that, we will utilize the parameters derived from these models and merge them to obtain the initial parameters for calibrating the Bates model. It is important to observe that for each guessed combination of parameters, we have computed various parameters.

Member C

In [38]:
```
# Load the data
S0 = 232.9
r0 = 1.5 / 100


data = pd.read_csv("MScFE 622_Stochastic Modeling_GWP1_Option data.xlsx - 1.csv")
data["r"] = r0
```

```python
data["T"] = data["Days to maturity"] / 250  # Assuming 250 days in a year

options = data[(data["Days to maturity"] == 60)]  # Select 60-day maturity instruments

# Bates characteristic function
def Bates_char_func(u, T, r, kappa_v, theta_v, sigma_v, rho, v0, lambda_, delta, gamma):
    # Modify the characteristic function to include jumps
    c1 = kappa_v * theta_v
    c2 = -np.sqrt(
        (rho * sigma_v * u * 1j - kappa_v) ** 2 - sigma_v**2 * (-u * 1j - u**2)
    )
    c3 = (kappa_v - rho * sigma_v * u * 1j + c2) / (
        kappa_v - rho * sigma_v * u * 1j - c2
    )
    J = np.exp(lambda_ * (np.exp(delta + 0.5 * gamma**2) - 1) * T * (np.exp(1j * u) - 1))

    H1 = r * u * 1j * T + (c1 / sigma_v**2) * (
        (kappa_v - rho * sigma_v * u * 1j + c2) * T
        - 2 * np.log((1 - c3 * np.exp(c2 * T)) / (1 - c3))
    )
    H2 = (
        (kappa_v - rho * sigma_v * u * 1j + c2)
        / sigma_v**2
        * ((1 - np.exp(c2 * T)) / (1 - c3 * np.exp(c2 * T)))
    )

    char_func_value = np.exp(H1 + H2 * v0) * J

    return char_func_value

# Bates option valuation function
def Bates_option_value(S0, K, T, r, kappa_v, theta_v, sigma_v, rho, v0, lambda_, delta, gamma):
    u = np.linspace(0, 200, 4000)  # Discretize the integral
    integrand = (
        np.exp(-1j * u * np.log(K / S0))
        * Bates_char_func(u - 1j / 2, T, r, kappa_v, theta_v, sigma_v, rho, v0, lambda_, delta, gamma)
    ).real

    # Perform numerical integration using the trapezoidal rule
    integral_value = np.trapz(integrand, u) / np.pi

    # Calculate option value
    call_value = max(0, S0 - np.exp(-r * T) * np.sqrt(S0 * K) / np.pi * integral_value)
    put_value = call_value + K * np.exp(- r * T) - S0
```

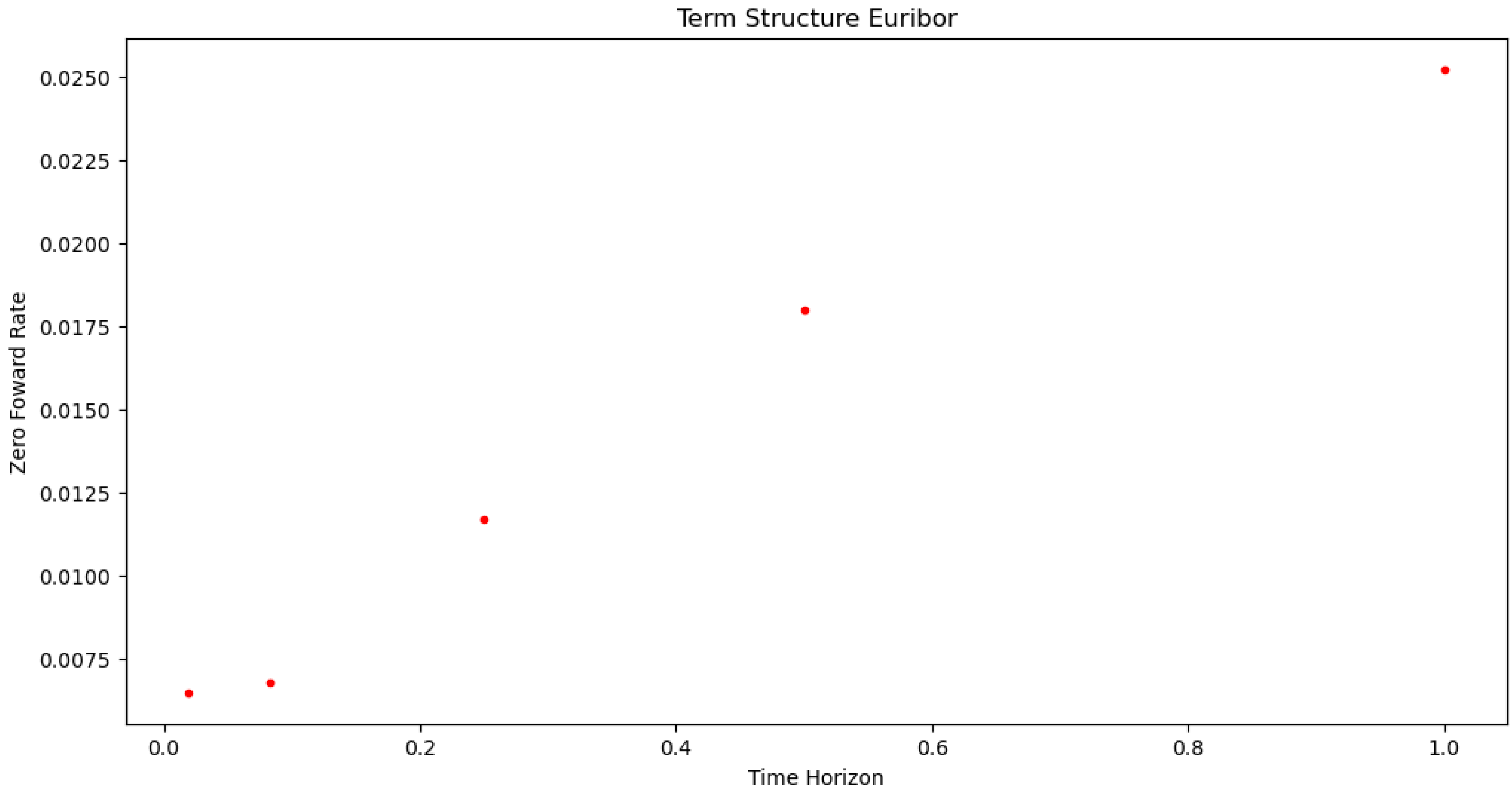
```python
    return put_value

# Error function for Bates calibration
i = 0
min_MSE = 500

def Bates_error_function(p0):
    global i, min_MSE
    kappa_v, theta_v, sigma_v, rho, v0, lambda_, delta, gamma = p0

    if kappa_v < 0.0 or theta_v < 0.005 or sigma_v < 0.0 or rho < -1.0 or rho > 1.0:
        return 500.0
    if 2 * kappa_v * theta_v < sigma_v**2:
        return 500.0

    se = []
    for row, option in options.iterrows():
        model_value = Bates_option_value(
            S0,
            option["Strike"],
            option["T"],
            option["r"],
            kappa_v,
            theta_v,
            sigma_v,
            rho,
            v0,
            lambda_,
            delta,
            gamma,
        )

        se.append((model_value - option["Price"]) ** 2)

    MSE = sum(se) / len(se)
    min_MSE = min(min_MSE, MSE)
    if i % 25 == 0:
        print("%4d |" % i, np.array(p0).round(2), "| %7.3f | %7.3f" % (MSE, min_MSE))
    i += 1
    return MSE

# Calibration function for Bates model
def Bates_calibration_full():
    p0 = brute(
        Bates_error_function,
```

```python
        (
            (2.5, 10.6, 5.0),
            (0.01, 0.041, 0.01),
            (0.05, 0.251, 0.1),
            (-0.75, 0.01, 0.25),
            (0.01, 0.031, 0.01),
            (0.01, 0.1, 0.01),
            (-0.1, 0.1, 0.01),
            (0.01, 0.1, 0.01),
        ),
        finish=None,
    )

    opt = fmin(
        Bates_error_function, p0, xtol=0.00001, ftol=0.00001, maxiter=750, maxfun=900
    )

    return opt

# conduct Bates calibration for the 60-day maturity instruments
#params_Bates = Bates_calibration_full()
#params_Bates.round(2)

# Generate and display the plot
def generate_plot_Bates(params, options):
    kappa_v, theta_v, sigma_v, rho, v0, lambda_, delta, gamma = params
    options["Model"] = 0.0
    for row, option in options.iterrows():
        options.loc[row, "Model"] = Bates_option_value(
            S0, option["Strike"], option["T"], option["r"], kappa_v, theta_v, sigma_v, rho, v0, lambda_, delta, gamma
        )

    mats = sorted(set(options["Days to maturity"]))
    options = options.set_index("Strike")
    for i, mat in enumerate(mats):
        options[options["Days to maturity"] == mat][["Price", "Model"]].plot(
            style=["b-", "ro"], title="%s days" % str(mat)[:10]
        )
        plt.ylabel("Option Value")

# Generate and display the plot for Bates model
#generate_plot_Bates(params_Bates, options)
```

Step 3

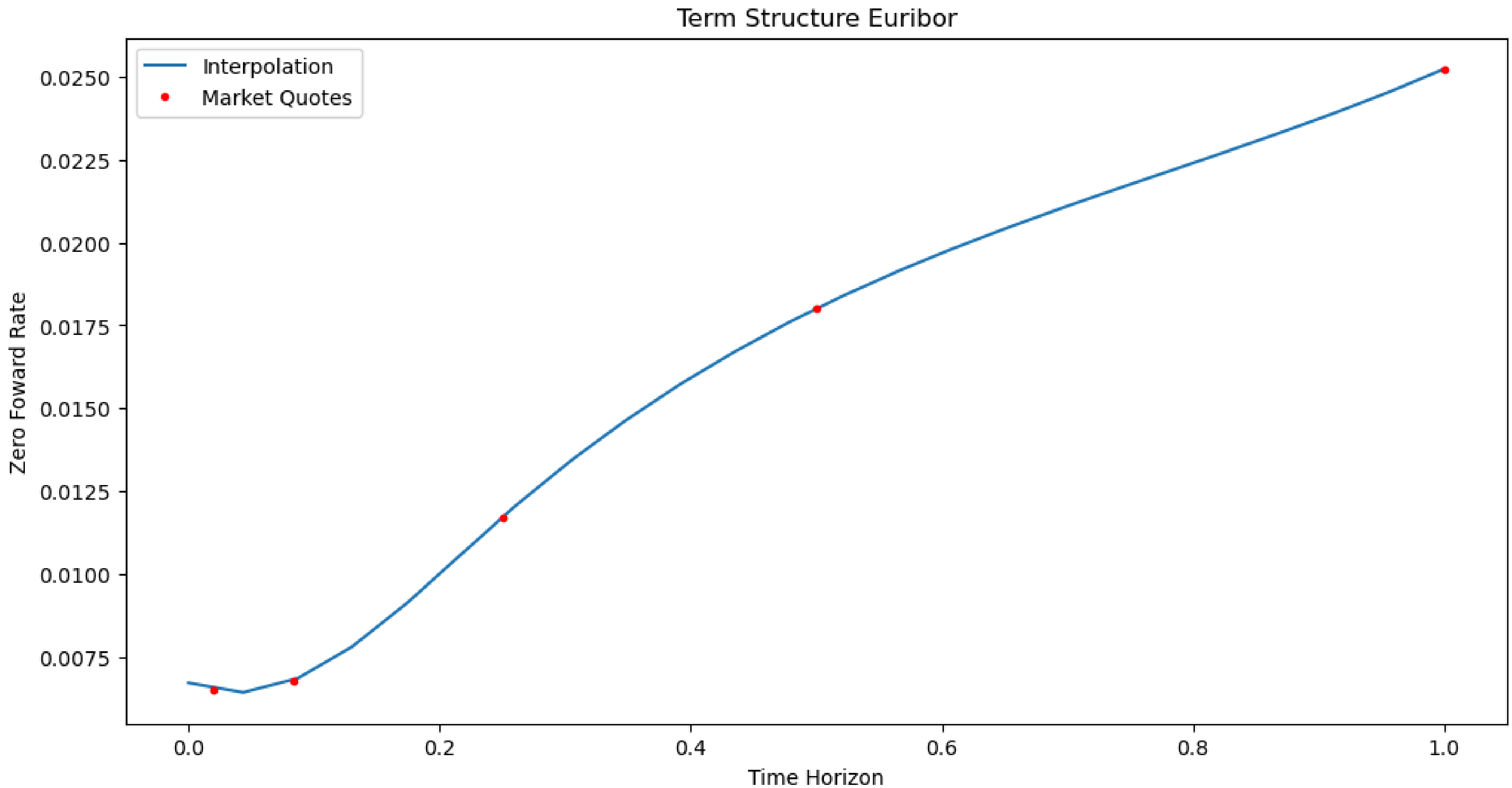
By utilizing Euribor rates, we have established a set of time periods that correspond to the quotes for the risk-free rate. Let's establish the current short-term rate (r0), the capitalization factors, and the zero-forward rates implied by the Euribor rates observed in the market.

In [39]:
```
mat_list = np.array((7, 30, 90, 180, 360)) / 360
rate_list = (
    np.array((0.648, 0.679, 1.173, 1.809, 2.556)) / 100
)

r0 = rate_list[0]
factors = 1 + mat_list * rate_list
zero_rates = 1 / mat_list * np.log(factors)

plt.figure(figsize=(12,6))
plt.plot(mat_list, zero_rates, "r.")
plt.xlabel("Time Horizon")
plt.ylabel("Zero Foward Rate")
plt.title("Term Structure Euribor");
```



Next, we utilize the cubic spline method to accurately interpolate the rate curve by seamlessly filling in the missing data points.

In [40]:
```
bspline = splrep(mat_list, zero_rates, k=3)
mat_list_n = np.linspace(0.0, 1.0, 24)
inter_rates = splev(mat_list_n, bspline, der=0)
first_der = splev(mat_list_n, bspline, der=1)
f = inter_rates + first_der * mat_list_n

plt.figure(figsize=(12,6))
plt.plot(mat_list_n, inter_rates, label="Interpolation")
plt.plot(mat_list, zero_rates, "r.", label="Market Quotes")
plt.xlabel("Time Horizon")
plt.ylabel("Zero Foward Rate")
plt.title("Term Structure Euribor")
plt.legend();
```



In [41]:

```
def CIR_forward_rate(alpha):
    kappa_r, theta_r, sigma_r = alpha

    t = mat_list_n
    g = np.sqrt(kappa_r**2 + 2 * sigma_r**2)

    s1 = (kappa_r * theta_r * (np.exp(g * t) - 1)) / (
        2 * g + (kappa_r + g) * (np.exp(g * t) - 1)
    )
    s2 = r0 * (
        (4 * g**2 * np.exp(g * t))
        / (2 * g + (kappa_r + g) * (np.exp(g * t) - 1)) ** 2
    )
    return s1 + s2

# Error function
```

```python
def CIR_error_function(alpha):
    kappa_r, theta_r, sigma_r = alpha
    if 2 * kappa_r * theta_r < sigma_r**2:
        return 100
    if kappa_r < 0 or theta_r < 0 or sigma_r < 0.001:
        return 100
    forward_rates = CIR_forward_rate(alpha)
    MSE = np.sum((f - forward_rates) ** 2) / len(f)
    return MSE

def CIR_calibration():
    opt = fmin(
        CIR_error_function,
        [0.5, 0.05, 0.2],
        xtol=0.00001,
        ftol=0.00001,
        maxiter=300,
        maxfun=500,
    )
    return opt
```

In [42]:

```python
params = CIR_calibration()
params.round(4)
```

```
Optimization terminated successfully.
         Current function value: 0.000003
         Iterations: 166
         Function evaluations: 299
```

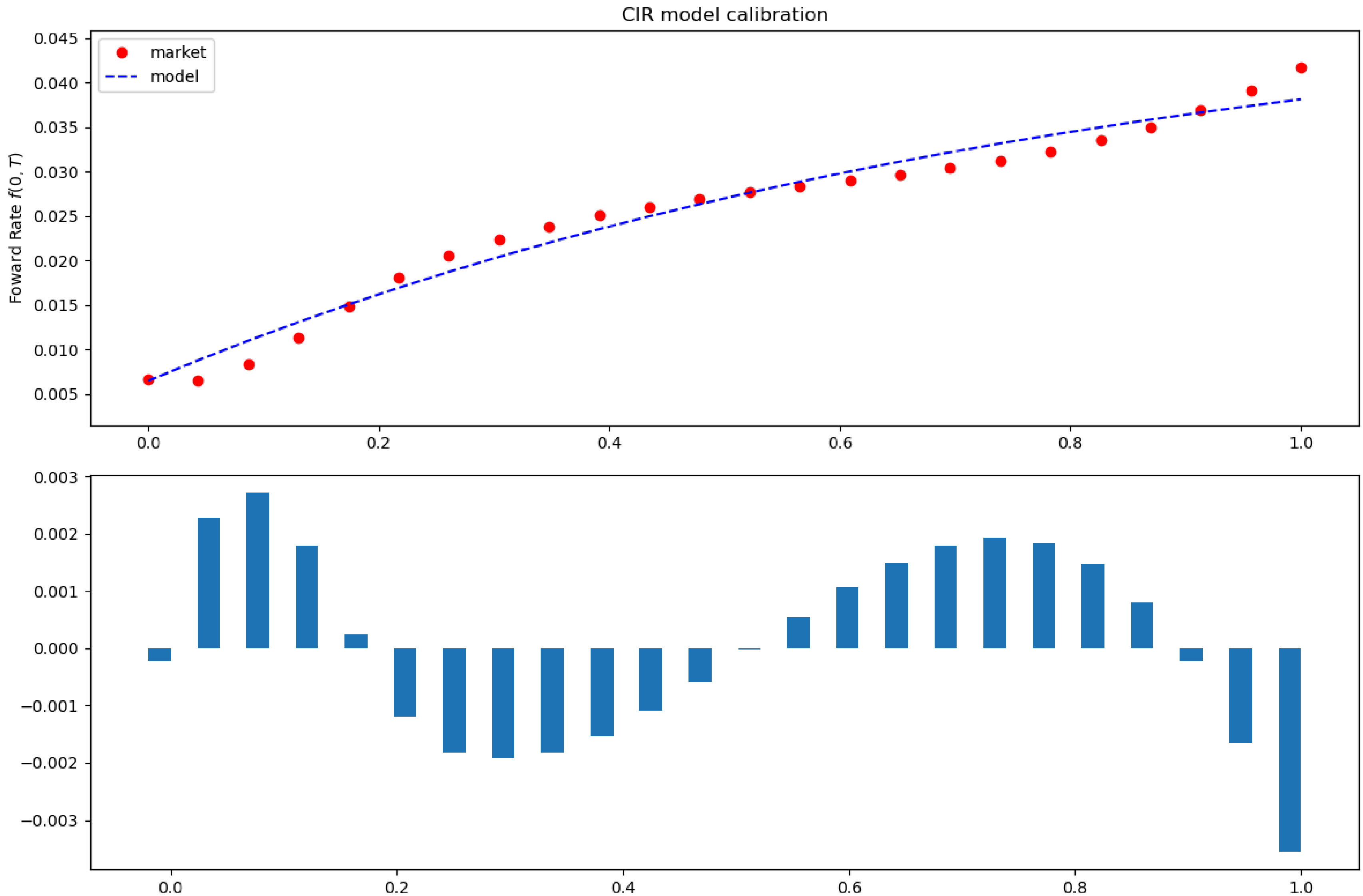
Out[42]:

```
array([1.2227, 0.1026, 0.0105])
```

In [43]:

```python
def plot_calibrated_frc(opt):
    forward_rates = CIR_forward_rate(opt)
    plt.figure(figsize=(12, 8))
    plt.subplot(211)
    plt.title("CIR model calibration")
    plt.ylabel("Foward Rate $f(0,T)$")
    plt.plot(mat_list_n, f, "ro", label="market")
    plt.plot(mat_list_n, forward_rates, "b--", label="model")
    plt.legend(loc=0)
    plt.axis(
        [min(mat_list_n) - 0.05, max(mat_list_n) + 0.05, min(f) - 0.005, max(f) * 1.1]
    )
    plt.subplot(212)
    wi = 0.02
```

```
    plt.bar(mat_list_n - wi / 2, forward_rates - f, width=wi)
    plt.tight_layout()

plot_calibrated_frc(params)
```



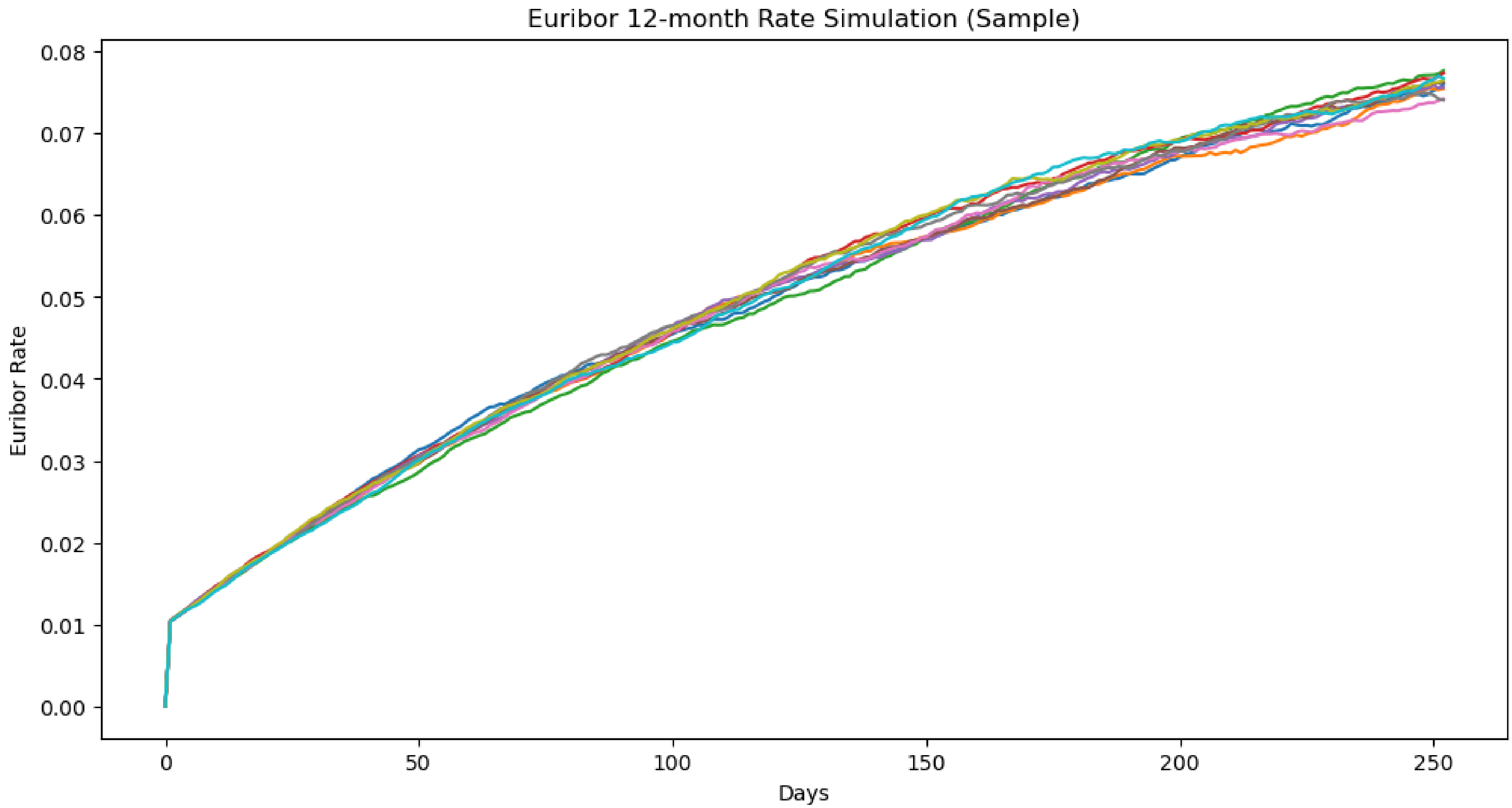
After calibrating, we obtained the following results: κ = 1.2227. θ = 0.1026. The value of σ is 0.0105.

In [44]:
```
# CIR model parameters
kappa = 1.2227
```

```python
theta = 0.1026
sigma = 0.0105

# Simulation parameters
num_simulations = 100000
num_days = 252  # Assuming 252 business days in a year

# Function to simulate CIR process
def cir_simulation(kappa, theta, sigma, num_days, num_simulations):
    dt = 1 / num_days
    rates = np.zeros((num_simulations, num_days + 1))

    for i in range(num_simulations):
        rate = 0.01
        for j in range(1, num_days + 1):
            dW = np.random.normal(0, np.sqrt(dt))
            rate += kappa * (theta - rate) * dt + sigma * np.sqrt(rate) * dW
            rates[i, j] = rate

    return rates
```

In [48]:

```python
plt.figure(figsize=(12, 6))
for i in range(10):  # Plotting 10 sample simulations
    plt.plot(np.arange(num_days + 1), euribor_simulations[i, :])

plt.title('Euribor 12-month Rate Simulation (Sample)')
plt.xlabel('Days')
plt.ylabel('Euribor Rate')
plt.show()

# Calculate statistics and discuss the results
confidence_interval = np.percentile(euribor_simulations[:, -1], [2.5, 97.5])
expected_value = np.mean(euribor_simulations[:, -1])

print(f'i. Confidence Interval (95%): {confidence_interval}')
print(f'ii. Expected Value after 1 year: {expected_value:.4%}')
```




```
i. Confidence Interval (95%): [0.0723379  0.07853569]
ii. Expected Value after 1 year: 7.5415%
```

The 95% confidence interval [0.0723, 0.0785] indicates a precise range for the projected future 12-month Euribor rates. The level of precision demonstrated here is quite valuable when it comes to decision-making. It provides a higher level of confidence in the estimated range.

```
In [ ]:
```