Short Introduction

These example below is taken from the Class: GRA6535 Derivatives, at BI Norwegian Business school. The task was to program different scenarios for the options. This was an excellent exercise to combine knowledge with programming to understand how to build pricing functions for options.

Exercise 1

Task description:

Find the price of an American put with S0=K= 30,r= 0.04, σ= 0.2, and maturity in 7 months using the binomial method. the option numerically (without using any analyticalformulas of the greeks).

In [4]:
```
# First we need to import necessary modules in this case we only need the math
import math


# Building our Binomial Tree (this is our method for pricing the American put option)
def put_binomial(s0, K, r, sigma, T, n):
    # Calculating the time interval
    dt = T / n

    # Calculating the discount factor (Our risk-free)
    rf_discount = math.exp(-r * dt)

    # Calculate the up and down factors
    u = math.exp(sigma * math.sqrt(dt))
    d = 1 / u

    # Calculating the risk-neutral probability p
    p = (math.exp(r * dt) - d) / (u - d)

    # Creating our binomial price tree with a double for loop
    tree = [[0 for j in range(i + 1)] for i in range(n + 1)]
    tree[0][0] = s0

    # Forward calculating the stock price in our tree based on u and d
    for i in range(1, n + 1):
        tree[i][0] = tree[i - 1][0] * u  # Upward prices
        for j in range(1, i + 1):
            tree[i][j] = tree[i - 1][j - 1] * d  # Downward prices

    # Calculating the option values at the maturity, the end of our trees
    options_in_tree = [[0 for j in range(i + 1)] for i in range(n + 1)]
    for j in range(n + 1):
        options_in_tree[n][j] = max(0, K - tree[n][j])

    # Then we need to work our way backwards, using the backward induction method
    for i in range(n - 1, -1, -1):
        for j in range(i + 1):
            # Since this is an American option we need to take the early exercise into account
            exercise_value = max(0, K-tree[i][j])
            # Finding the PV by discounting the price
            continuation_value = rf_discount * (p*options_in_tree[i + 1][j + 1])
            # Choose either the early exercise price or the continuation value
            options_in_tree[i][j] = max(exercise_value, continuation_value)
```

```
    return options_in_tree[0][0] # Returning the option value of the stock


# Inserting our values into the function for finding the price for the American option
option_price = put_binomial(s0=30, K=30, r=0.04, T=7/12, sigma=0.2, n=4)
print(f"The American put option price is: ${round(option_price,4)}")
```

```
The American put option price is: $1.1386
```

Exercise 2

Task description:

Find the price of a European call withS0=K= 50,r= 0.03,σ= 0.25, and maturityin 5 months using the Monte Carlo method. Compute the delta, gamma, and vega of theoption numerically (without using any analytical formulas of the greeks)

In [5]:
```
# We need to import the numpy library for different calculations
# As well as the statistical package from Scipy
import numpy as np
import scipy.stats as sts

# Defining our global variables with values
s_0 = 50
K_price = 50
rate = 0.03
T_ = 5/12
sigma_ = 0.25
num_simulations = 10000


# Finding the price for European Call option
def call_pricing(s0, K, r, T, sigma,
                 num_simulations):
    # Our Wiener process
    z = np.random.standard_normal(num_simulations)
    st = s0 * np.exp((r-(0.5 * sigma**2))*T + np.sqrt(T)*z)
    payoffs = np.maximum(st-K, 0)
    price = np.exp(-r*T)*np.mean(payoffs)
    return price


call_price = call_pricing(s0=s_0, K=K_price,
                          r=rate, T=T_, sigma=sigma_,
                          num_simulations=num_simulations)
print(f"The European Call option price is: ${round(call_price, 3)}")


def calculate_greeks(s0=s_0, K=K_price, r=rate,
                     sigma=sigma_, T=T_):
    # We calclate the ND1, which is the norm.CDF of D1
    d1 = ((np.log(s0/K)) + (r+(0.5*sigma**2)) * T)/(sigma*np.sqrt(T))
    n_d1 = sts.norm.cdf(d1)

    # Calculating our three greeks
    delta = n_d1
    gamma = sts.norm.pdf(d1)/(s0*sigma*np.sqrt(T))
    vega = (s_0 * np.sqrt(T) * sts.norm.pdf(d1)/100)

    # printing out the result
```

```
    print(f"The Delta is: {round(delta, 4)}")
    print(f"The Gamma is: {round(gamma, 4)}")
    print(f"The Vega is: {round(vega, 4)}")


if __name__ == "__main__":
    calculate_greeks()
```

```
The European Call option price is: $21.283
The Delta is: 0.5628
The Gamma is: 0.0488
The Vega is: 0.1272
```

Exercise 3

Task description:

Find the price of a European call with S0=K= 40,r= 0.05, and maturity in 8 months. The risk-neutral dynamics of the stock price is dSt=rStdt+σtStdzt. Here the volatility of the stock price is stochastic and: dσt=uσtdt+vσtdwt, σ0= 0.3,where u= 0.3,v= 0.2, and wt is a Wiener process independent to zt.

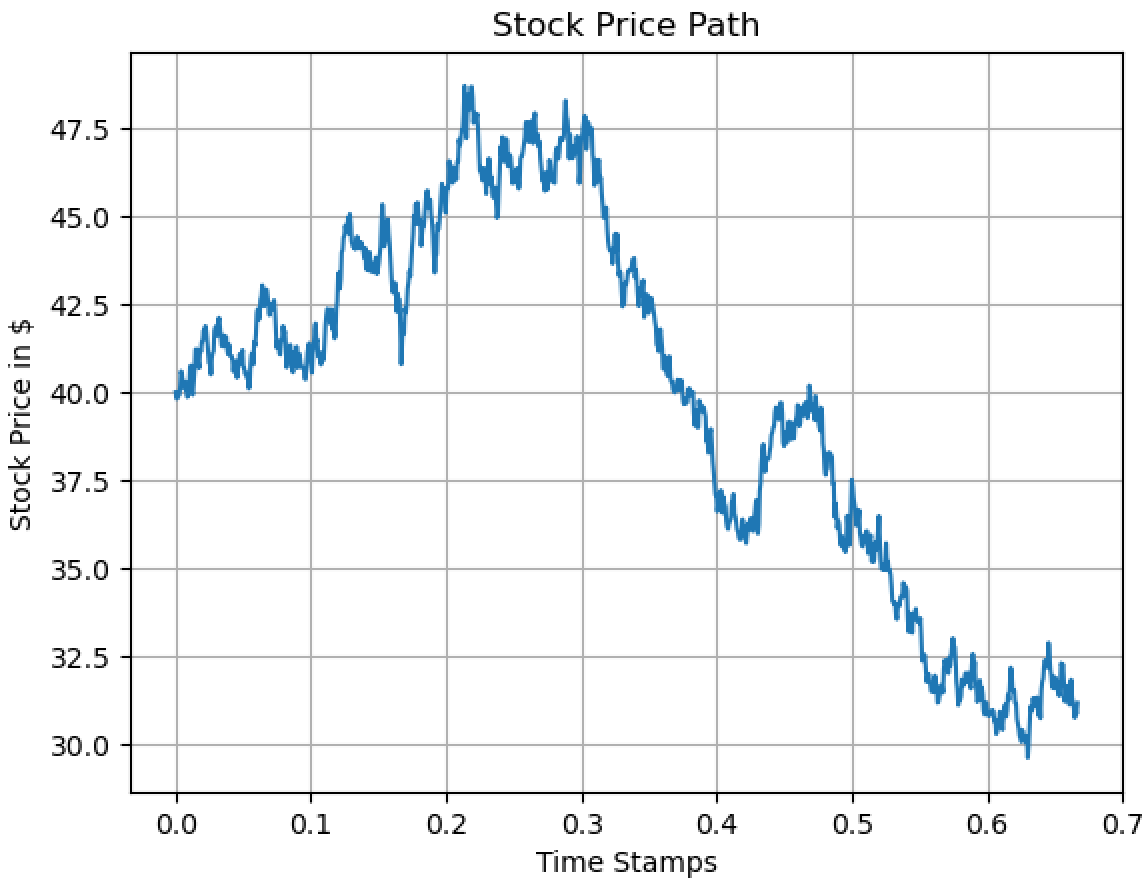
Choose the appropriate numerical method. Compute the delta, gamma, and vega of the option numerically(without using any analytical formulas of the greeks)

In [7]:
```
# Import our module numpy, Jit and MatPlotLib
import numpy as np
from numba import jit
import matplotlib.pyplot as plt
import scipy.stats as sts

# Defining our necessary global variables with values
s_0 = 40
K_price = 40
rate = 0.05
sigma_ = 0.3
v = 0.2
u = 0.3
T_ = 8/12
num_simulations = 100
dt = T_/1000  # Choosing the delta time to be small timestamps


# Simulating a number of different price paths for the stock and its volatility
# Adding the NUMBA Jit function
@jit(nopython=True)
def price_path(s0=s_0, sigma_0=sigma_, r=rate, u=u,
               v=v, T=T_, dt=dt):
    # Defining our necessary values
    N = int(T/dt)
    S = np.zeros(N)
    sigma = np.zeros(N)
    S[0] = s0
    sigma[0] = sigma_0

    # Defining our paths
    for t in range(1, N):
        # Our standard normal distributed stochastic variables
        z, w = np.random.standard_normal(2)

        # Calculating the stochastic volatility
        sigma[t] = sigma[t-1] + u*sigma[t-1]*dt + v*sigma[t-1]*w*np.sqrt(dt)

        # The risk-neutral price dynamics of the stock
        S[t] = S[t-1] + r*S[t-1]*dt + sigma[t-1] * S[t-1] * z * np.sqrt(dt)
    return S[-1], sigma[-1]
```

```
# Since we have the opportunity to choose which method we
# want to utilize, I chose MC
def mc_method(s0=s_0, K=K_price, sigma_0=sigma_, r=rate,
              u=u, v=v, T=T_, num_simulations=num_simulations):
    # Our payoff calculation
    payoffs = [max(price_path(s0=s0, sigma_0=sigma_0,
                              r=r,u=u, v=v, T=T, dt=dt)[0]-K, 0)
               for _ in range(num_simulations)]
    price = np.exp(-r*T)*np.mean(payoffs)
    return price

# Our option price
option_price = mc_method()

# Calculating the necessary greeks
def calculate_greeks(s0=s_0, K=K_price, r=rate, sigma=sigma_, T=T_):
        d1 = ((np.log(s0 / K)) + (r + (0.5 * sigma ** 2)) * T) / (sigma * np.sqrt(T))
        n_d1 = sts.norm.cdf(d1)

        # Calculate the different Greeks
        delta = n_d1
        gamma = sts.norm.pdf(d1) / (s0 * sigma * np.sqrt(T))
        vega = (s_0 * np.sqrt(T) * sts.norm.pdf(d1) / 100)

        # printing out the result
        print(f"The Option price from MC simulation is: ${round(option_price, 4)}")
        print(f"The Delta is: {round(delta, 4)}")
        print(f"The Gamma is: {round(gamma, 4)}")
        print(f"The Vega is: {round(vega, 4)}")


# Since we want to visualize the price path, need to return the whole aeeay
@jit(nopython=True)
def price_path_2(s0=s_0, sigma_0=sigma_,
                 r=rate,u=u, v=v, T=T_, dt=dt):

    # Defining our necessary values
    N = int(T/dt)
    S = np.zeros(N)
    sigma = np.zeros(N)
    S[0] = s0
    sigma[0] = sigma_0
```

```
    # Defining our paths
    for t in range(1, N):
        # Our standard normal distributed stochastic variables
        z, w = np.random.standard_normal(2)

        # Calculating the stochastic volatility
        sigma[t] = sigma[t-1] + u*sigma[t-1]*dt + v*sigma[t-1]*w*np.sqrt(dt)

        # The risk-neutral price dynamics of the stock
        S[t] = S[t-1] + r*S[t-1]*dt + sigma[t-1] * S[t-1] * z * np.sqrt(dt)
    return S, sigma


# Want to visualize the computed price path of the stock
def plot_price_path():
    N = int(T_ / dt)
    S, _ = price_path_2()
    time_stamps = np.linspace(0,T_, N)

    plt.plot(time_stamps, S)
    plt.title("Stock Price Path")
    plt.xlabel("Time Stamps")
    plt.ylabel("Stock Price in $")
    plt.grid(True)
    plt.show()


if __name__ == "__main__":
    calculate_greeks()
    plot_price_path()
```

```
The Option price from MC simulation is: $4.5338
The Delta is: 0.602
The Gamma is: 0.0394
The Vega is: 0.126
```


Stock Price Path
Stock Price in $
Time Stamps
47.5
45.0
42.5
40.0
37.5
35.0
32.5
30.0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7

In [ ]: