In [1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import scipy.stats
from scipy.stats import norm
```

1) Evaluate the following expected values and probabilities:

1) $p1 = P(Y_2 > 5)$

2) $e1 = E[X_2^{1/3}]$

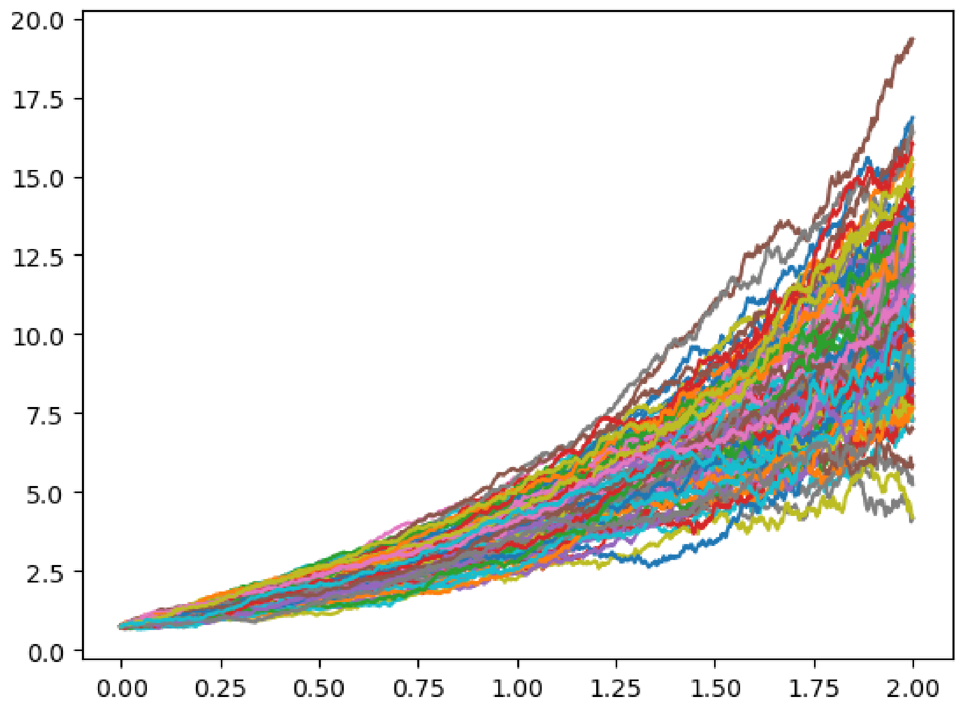
3) $e2 = E[Y_3]$

4) $e3 = E[X_2 \times Y_2 \times 1(X_2 > 1)]$

In [2]:
```
rand_norm_matrix = []
for i in range(100):
    rand_norm_temp = np.random.normal(size=1000)
    rand_norm_matrix.append(rand_norm_temp)
```

In [3]:
```
# p1
def Y_t(y_prev, t, zi, dt, j):
    def a(y0, t, j):
        return ((2/(1 + (j*dt)))*y_prev) + ((1+(j*dt)**3)/3)
    def b(y0, t, j):
        return (1+(j*dt)**3)/3
    yt = y_prev + (a(y_prev, t, j)*dt) + (b(y_prev, t, j)*np.sqrt(dt)*zi)
    return yt
```

In [4]:
```
all_y2 = []
y0 = 0.75
t = 2
time_step = t/1000
for i in range(len(rand_norm_matrix)):
    temp_y2 = [y0]
    for j in range(len(rand_norm_matrix[i])):
        y_t = Y_t(temp_y2[-1], 2, rand_norm_matrix[i][j], time_step, j)
        temp_y2.append(y_t)
    all_y2.append(temp_y2)
for i in all_y2:
    plt.plot(np.linspace(0, 2, 1000), i[1:])
```

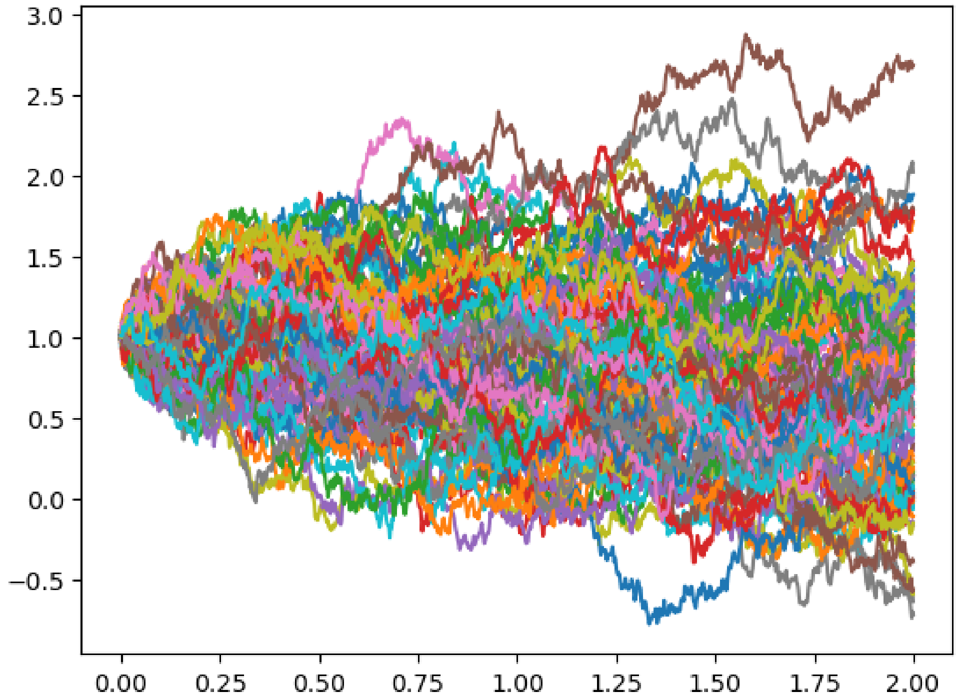


In [5]:
```
count = 0
for i in range(len(all_y2)):
    if all_y2[i][-1] > 5:
        count += 1
print("p1:", count/len(all_y2))
```

```
p1: 0.98
```

In [6]:
```
# e1
def X_t(x_prev, zi, dt):
    def a(x_prev):
        value = (1/5) - ((1/2)*x_prev)
        return value
    temp = (x_prev + (a(x_prev)*dt) + ((2/3)*np.sqrt(dt)*zi))
    return temp
```

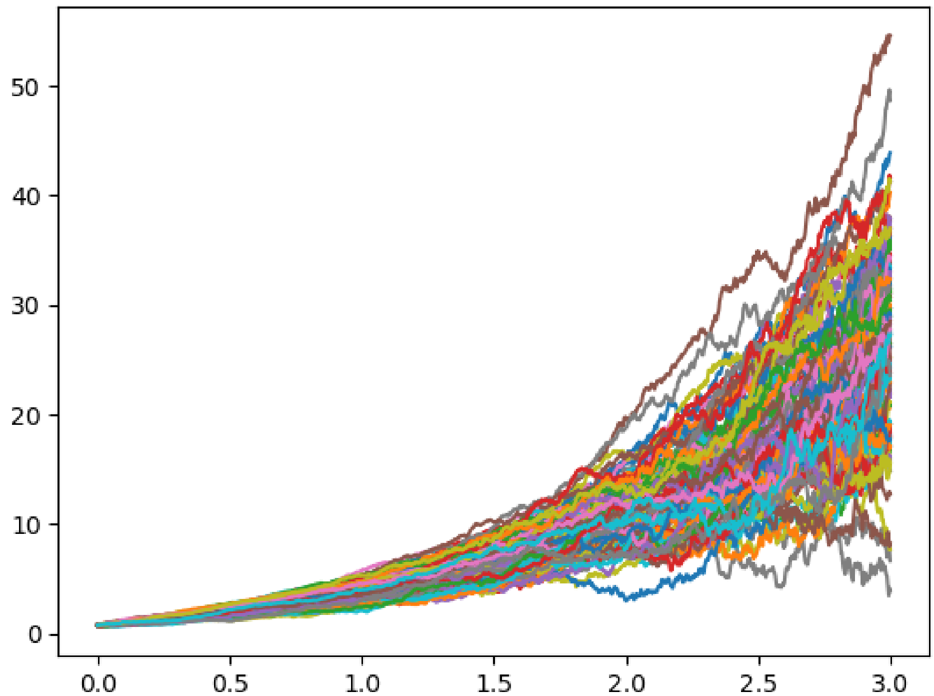
In [7]:
```
all_x2 = []
x0 = 1
t = 2
time_step = t/1000
for i in range(len(rand_norm_matrix)):
    temp_x2 = [x0**(1/3)]
    for j in range(len(rand_norm_matrix[i])):
        x_t = X_t(temp_x2[-1], rand_norm_matrix[i][j], time_step)
        temp_x2.append(x_t)
    all_x2.append(temp_x2)
for i in all_x2:
    plt.plot(np.linspace(0, 2, 1000), i[1:])
```



In [8]:
```
e1_list = [np.cbrt(all_x2[i][-1]) for i in range(len(all_x2))]
print("e1:", np.mean(e1_list))
```

```
e1: 0.6938301430966738
```

In [9]:
```
# e2
all_y3 = []
y0 = 0.75
t = 3
time_step = t/1000
for i in range(len(rand_norm_matrix)):
    temp_y3 = [y0]
    for j in range(len(rand_norm_matrix[i])):
        y_t = Y_t(temp_y3[-1], 3, rand_norm_matrix[i][j], time_step, j)
        temp_y3.append(y_t)
    all_y3.append(temp_y3)
for i in all_y3:
    plt.plot(np.linspace(0, 3, 1000), i[1:])
```



In [10]:
```
e2_list = [all_y3[i][-1] for i in range(len(all_x2))]
print("e2:", np.mean(e2_list))
```

```
e2: 26.943346723550565
```

In [11]:
```
# e3
x2y2 = [all_x2[i][-1] * all_y2[i][-1] for i in range(len(all_x2))]
x2y2 = [x2y2[i] if all_x2[i][-1]>1 else 0 for i in range(len(x2y2))]
print("e3:", np.mean(x2y2))
```

```
e3: 6.174785433120284
```

2) Estimate the following expected values:

1) $e1 = E[1 + X_3]^{1/3}$

2) $e2 = E[X_1 \times Y_1]$

Since we know that W_t and Z_t are both independent standard normal processes, $\rho = 0$ and there is no need to implement the Cholesky Decomposition

In [12]:
```
z1_matrix, z2_matrix = [], []
for i in range(100):
    z1_matrix.append(np.random.normal(size=1000))
    z2_matrix.append(np.random.normal(size=1000))
```

In [13]:
```
def Xt(x_prev, z1, z2, dt):
    next_x = x_prev + (0.25 * x_prev * dt) + ((1/3)*x_prev*np.sqrt(dt)*z1) - (0.75*x_pre
    return next_x
def Yt(z1, z2, dt, j):
    next_y = np.exp((-0.08 * dt * j) + ((1/3)*np.sqrt(dt)*z1) + (0.75 * np.sqrt(dt) * z
    return next_y
```

In [14]:
```
# e1
all_x3 = []
x0 = 1
t = 3
time_step = t/1000
for i in range(len(z1_matrix)):
    x3_temp = [x0]
    for j in range(len(z1_matrix[i])):
        next_xt = Xt(x3_temp[-1], z1_matrix[i][j], z2_matrix[i][j], time_step)
        x3_temp.append(next_xt)
    all_x3.append(x3_temp)
```

In [15]:
```
one_plus_x3 = [1 + all_x3[i][-1] for i in range(len(all_x3))]
e1 = np.cbrt(np.mean(one_plus_x3))
print("e1:", e1)
```

```
e1: 1.4640230222617223
```

In [16]:
```
# e2
all_x1 = []
x0 = 1
t = 1
time_step = t/1000
for i in range(len(z1_matrix)):
    x3_temp = [x0]
    for j in range(len(z1_matrix[i])):
        next_xt = Xt(x3_temp[-1], z1_matrix[i][j], z2_matrix[i][j], time_step)
        x3_temp.append(next_xt)
    all_x1.append(x3_temp)

all_y1 = []
t = 1
time_step = t/1000
for i in range(len(z1_matrix)):
    y1_temp = []
    for j in range(len(z1_matrix[i])):
        next_yt = Yt(z1_matrix[i][j], z2_matrix[i][j], time_step, j)
        y1_temp.append(next_yt)
    all_y1.append(y1_temp)
```

In [17]:
```
x1y1 = [all_x1[i][-1] * all_y1[i][-1] for i in range(len(all_x1))]
print("e2:", np.mean(x1y1))
```

```
e2: 1.1754348710166769
```

Question 3

a) Write a code to compute prices of European Call options via Monte Carlo simulation. Use variance reduction techniques (e.g. Antithetic Variates) in your estimation. The code should be generic: for any input of the 5 model parameters - S_0, T, X, r, σ– the output is the corresponding price of the European call option

In [18]:
```
def european_call_option_monte_carlo(S0, K, r, sigma, T):
    price_greeks = {}
    zi = np.random.normal(size=1000)
    def price(S0, K, r, sigma, T):
        payoffs_positive = [max(0, S0 * np.exp((r - (sigma**2)/2)*(T) + (sigma)*(np.sqr
        payoffs_negative = [max(0, S0 * np.exp((r - (sigma**2)/2)*(T) + (sigma)*(np.sqr
        call_prem_antithetic_list = [(payoffs_positive[i] + payoffs_negative[i])/2 for i
        return np.exp(-1 * r * T) * np.mean(call_prem_antithetic_list)
    price_greeks['price'] = price(S0, K, r, sigma, T)

    def delta(S0, K, r, sigma, T):
        return (price(S0*1.01, K, r, sigma, T) - price(S0, K, r, sigma, T))/0.01
    price_greeks['delta'] = delta(S0, K, r, sigma, T)

    def gamma(S0, K, r, sigma, T):
        return (price(S0*1.01, K, r, sigma, T) - (2*price(S0, K, r, sigma, T)) + price(S
    price_greeks['gamma'] = gamma(S0, K, r, sigma, T)

    def vega(S0, K, r, sigma, T):
        return (price(S0, K, r, sigma*1.01, T) - price(S0, K, r, sigma, T))/0.01
    price_greeks['vega'] = vega(S0, K, r, sigma, T)

    def theta(S0, K, r, sigma, T):
        return (price(S0, K, r, sigma, T+0.004) - price(S0, K, r, sigma, T))/0.004
    price_greeks['theta'] = theta(S0, K, r, sigma, T)

    return price_greeks
```

In [19]:
```
print("c1:", np.exp(-1 * 0.05 * 0.5)*european_call_option_monte_carlo(100, 100, 0.05, 0
```

```
c1: 8.09713407787074
```

b) Write a code to compute the prices of European Call options by using the Black-Scholes formula. Use the approximation of $N(\cdot)$ described in Chapter 3. The code should be generic: for any input values of the 5 parameters - S_0, T, X, r, σ - the output is the corresponding price of the European call option.

In [20]:
```
def normal_approximation(x):
    d1, d2, d3 = 0.0498673470, 0.0211410061, 0.0032776263
    d4, d5, d6 = 0.0000380036, 0.0000488906, 0.0000053830
    if x < 0:
        x = np.negative(x)
    N = 1 - 0.5*(1 + d1*x + d2*x**2 + d3*x**3 + d4*x**4 + d5*x**5 + d6*x**6)**-16
    return N
```

In [21]:
```
def black_scholes(S, K, t, r, sigma):
    d1 = (np.log((S/K))  + ((r + (sigma**2)/2)*t))/(sigma * np.sqrt(t))
    d2 = d1 - (sigma * np.sqrt(t))
    call_price = S*normal_approximation(d1)- K*normal_approximation(d2)*np.exp(-r*t)
    return call_price
```

In [22]:
```
print("c2:", black_scholes(100, 100, 0.5, 0.05, 0.25))
```

```
c2: 8.260027941715038
```

c) Estimate the European call option's greeks - delta, gamma, theta, and vega - and graph them as functions of the initial stock price S_0. Use X = 20, σ = 0.25, r = 0.05 and T = 0.5 in your estimations. Use the range [15, 25] for S_0, with a step size of 1. You will have 4 different graphs for each of the 4 greeks. In all cases, dt (time-step) should be user-defined. Use dt=0.004 (a day) as a default value.

In [23]:
```
prices = [np.exp(-1*0.05*0.5)*european_call_option_monte_carlo(i, 20, 0.05, 0.25, 0.5)['
prices
```

Out[23]:
```
[0.08726811998981016,
 0.19127356033779067,
 0.4256916747501722,
 0.6875122090643153,
 1.0640557968598021,
 1.5608829791755836,
 2.195858757102825,
 2.992510070752141,
 3.7699397527876712,
 4.601171136222904,
 5.442687310828027]
```

In [24]:
```
def delta(S, K, t, r, sigma):
    d1 = (np.log((S/K))  + ((r - (sigma**2)/2)*t))/(sigma * np.sqrt(t))
    return norm.cdf(d1)

def gamma(S, K, t, r, sigma):
    d1 = (np.log((S/K))  + ((r + (sigma**2)/2)*t))/(sigma * np.sqrt(t))
    const = 1/(S*sigma*np.sqrt(t))
    return const * norm.pdf(d1)

def theta(S, K, t, r, sigma):
    d1 = (np.log((S/K))  + ((r + (sigma**2)/2)*t))/(sigma * np.sqrt(t))
    d2 = d1 - (sigma * np.sqrt(t))
    theta = (-S*sigma*norm.pdf(d1)/2*np.sqrt(t)) - r*K*np.exp(-r*t)*norm.cdf(d2)
    return theta

def vega(S, K, t, r, sigma):
    d1 = (np.log((S/K))  + ((r + (sigma**2)/2)*t))/(sigma * np.sqrt(t))
    vega = S*np.sqrt(t)*norm.pdf(d1)
    return vega
```

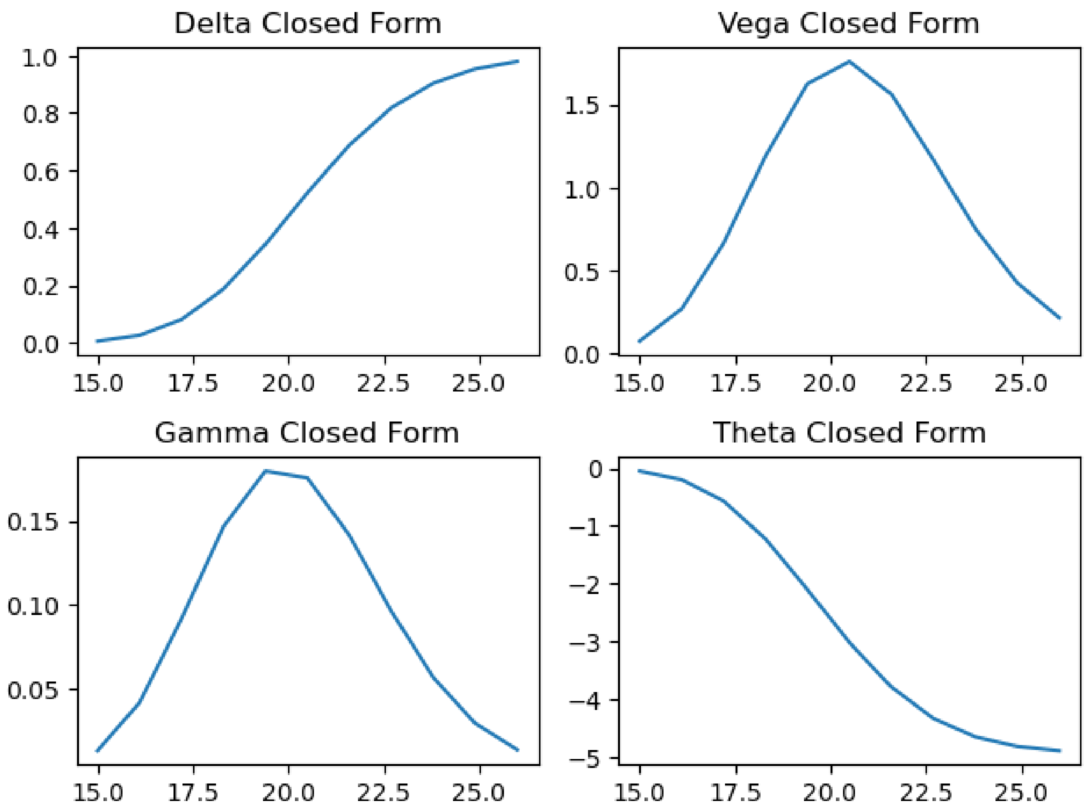
In [25]:
```
delta_list_closed_form = [delta(i, 20, 0.05, 0.25, 0.5) for i in range(15, 26)]
vega_list_closed_form = [vega(i, 20, 0.05, 0.25, 0.5) for i in range(15, 26)]
gamma_list_closed_form = [gamma(i, 20, 0.05, 0.25, 0.5) for i in range(15, 26)]
theta_list_closed_form = [theta(i, 20, 0.05, 0.25, 0.5) for i in range(15, 26)]
```

In [26]:
```
# Plots of all closed form Greeks
fig, axs = plt.subplots(2, 2)
axs[0,0].plot(np.linspace(15, 26, 11), delta_list_closed_form)
axs[0,0].set_title("Delta Closed Form")

axs[0,1].plot(np.linspace(15, 26, 11), vega_list_closed_form)
axs[0,1].set_title("Vega Closed Form")

axs[1,0].plot(np.linspace(15, 26, 11), gamma_list_closed_form)
axs[1,0].set_title("Gamma Closed Form")

axs[1,1].plot(np.linspace(15, 26, 11), theta_list_closed_form)
axs[1,1].set_title("Theta Closed Form")
fig.tight_layout(pad=1.0)
```



In [27]:
```
delta_list_fd = [european_call_option_monte_carlo(i, 20, 0.05, 0.25, 0.5)['delta'] for i
vega_list_fd = [european_call_option_monte_carlo(i, 20, 0.05, 0.25, 0.5)['vega'] for i i
gamma_list_fd = [european_call_option_monte_carlo(i, 20, 0.05, 0.25, 0.5)['gamma'] for i
theta_list_fd = [european_call_option_monte_carlo(i, 20, 0.05, 0.25, 0.5)['theta'] for i
```

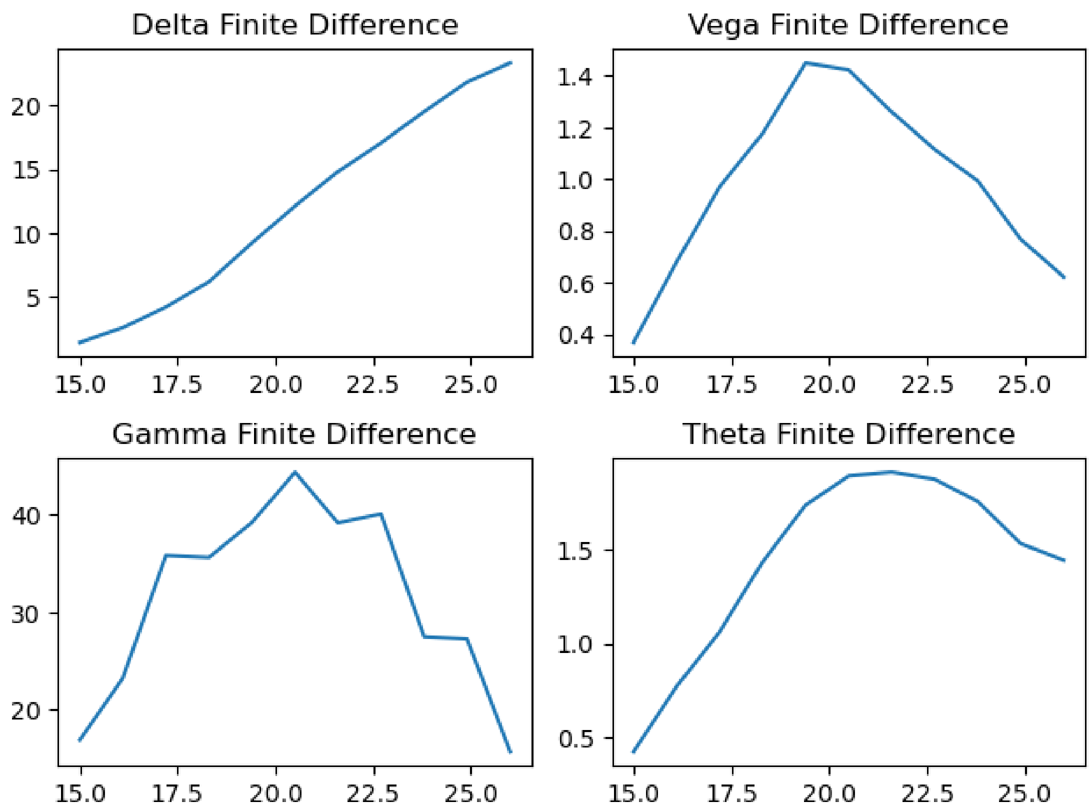
In [28]:
```
# Plots of all finite difference Greeks
fig, axs = plt.subplots(2, 2)
axs[0,0].plot(np.linspace(15, 26, 11), delta_list_fd)
axs[0,0].set_title("Delta Finite Difference")

axs[0,1].plot(np.linspace(15, 26, 11), vega_list_fd)
axs[0,1].set_title("Vega Finite Difference")

axs[1,0].plot(np.linspace(15, 26, 11), gamma_list_fd)
axs[1,0].set_title("Gamma Finite Difference")

axs[1,1].plot(np.linspace(15, 26, 11), theta_list_fd)
axs[1,1].set_title("Theta Finite Difference")

fig.tight_layout(pad=1.0)
```



Question 4

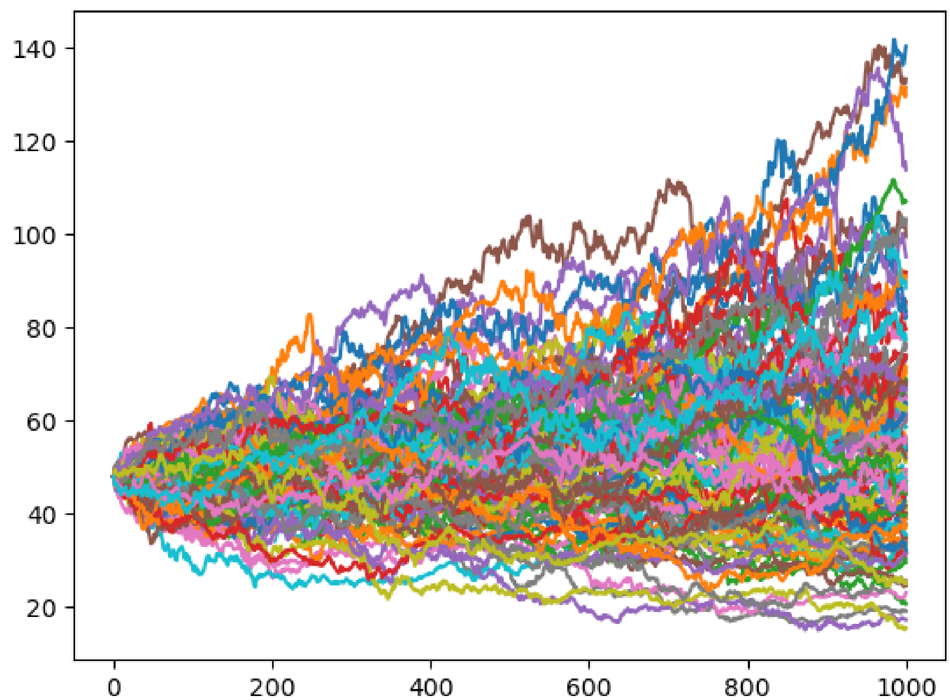
In [29]:
```
# Generate correlated standard normal variables
rho = -0.6
T = 3
z1_matrix, z2_matrix = [], []
for i in range(100):
    z1 = np.random.normal(size=1000)
    z2 = np.random.normal(size=1000)
    z2 = [z1[i]*rho + z2[i]*np.sqrt(1 - rho**2) for i in range(len(z1))]
    z1_matrix.append(z1)
    z2_matrix.append(z2)

S0, v0, alpha, beta, sigma, dt, K, r = 48, 0.05, 5.8, 0.0625, 0.42, T/1000, 50, 0.03
```

In [30]:
```
# Full Truncation Method
# Estimate the volatility path
def V_t(prev_v, alpha, beta, sigma, dt, z2_i):
    next_v = prev_v + ((alpha * (beta - max(0, prev_v)))*dt) + (sigma * np.sqrt(max(0, 
    return next_v
# Estimate the stock path
def S_t(prev_s, prev_v, dt, z1_i, r):
    next_s = prev_s + (r * prev_s * dt) + (np.sqrt(max(0, prev_v)) * prev_s * np.sqrt(d
    return next_s

vt_matrix, st_matrix = [], []
for j in range(len(z1_matrix)):
    vt_list, st_list = [v0], [S0]
    for i in range(len(z1_matrix[j])):
        V_k_plus_one = V_t(vt_list[-1], alpha, beta, sigma, dt, z2_matrix[j][i])
        S_k_plus_one = S_t(st_list[-1], vt_list[-1], dt, z1_matrix[j][i], r)
        vt_list.append(V_k_plus_one)
        st_list.append(S_k_plus_one)
    vt_matrix.append(vt_list)
    st_matrix.append(st_list)
```

In [31]:
```
for i in st_matrix:
    plt.plot(i)
```



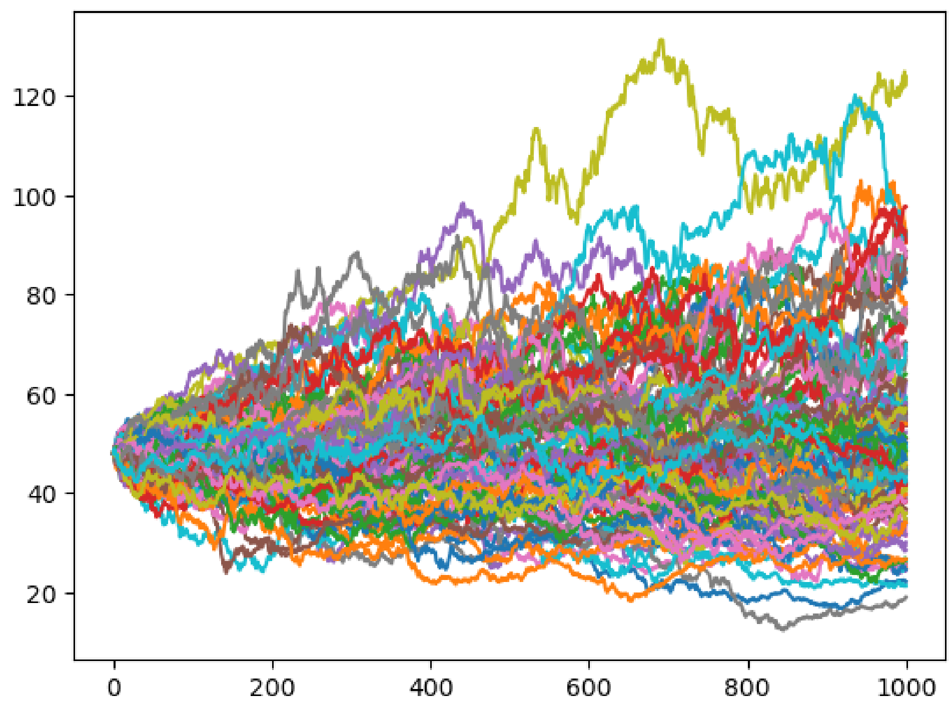
In [32]:
```
payoffs_full_trunc = [max(0, st_matrix[i][-1] - K) for i in range(len(st_matrix))]
print("C1:", np.exp(-1*r*T)*np.mean(payoffs_full_trunc))
```

```
C1: 11.607476962622192
```

In [33]:
```
# Reflection Method
z1_matrix, z2_matrix = [], []
np.random.seed(0)
for i in range(100):
    z1 = np.random.normal(size=1000)
    z2 = np.random.normal(size=1000)
    z2 = [z1[i]*rho + z2[i]*np.sqrt(1 - rho**2) for i in range(len(z1))]
    z1_matrix.append(z1)
    z2_matrix.append(z2)
# Estimate the volatility path
def V_t(prev_v, alpha, beta, sigma, dt, z2_i):
    next_v = abs(prev_v) + ((alpha*(beta - abs(prev_v)))*dt) + (sigma * np.sqrt(abs(pre
    return next_v
# Estimate the stock path
def S_t(prev_s, prev_v, dt, z1_i, r):
    next_s = prev_s + (r * prev_s * dt) + (np.sqrt(abs(prev_v)) * prev_s * np.sqrt(dt)
    return next_s

vt_matrix, st_matrix = [], []
for j in range(len(z1_matrix)):
    vt_list, st_list = [v0], [S0]
    for i in range(len(z1_matrix[j])):
        V_k_plus_one = V_t(vt_list[-1], alpha, beta, sigma, dt, z2_matrix[j][i])
        S_k_plus_one = S_t(st_list[-1], vt_list[-1], dt, z1_matrix[j][i], r)
        vt_list.append(V_k_plus_one)
        st_list.append(S_k_plus_one)
    vt_matrix.append(vt_list)
    st_matrix.append(st_list)
```

In [34]:
```
for i in st_matrix:
    plt.plot(i)
```



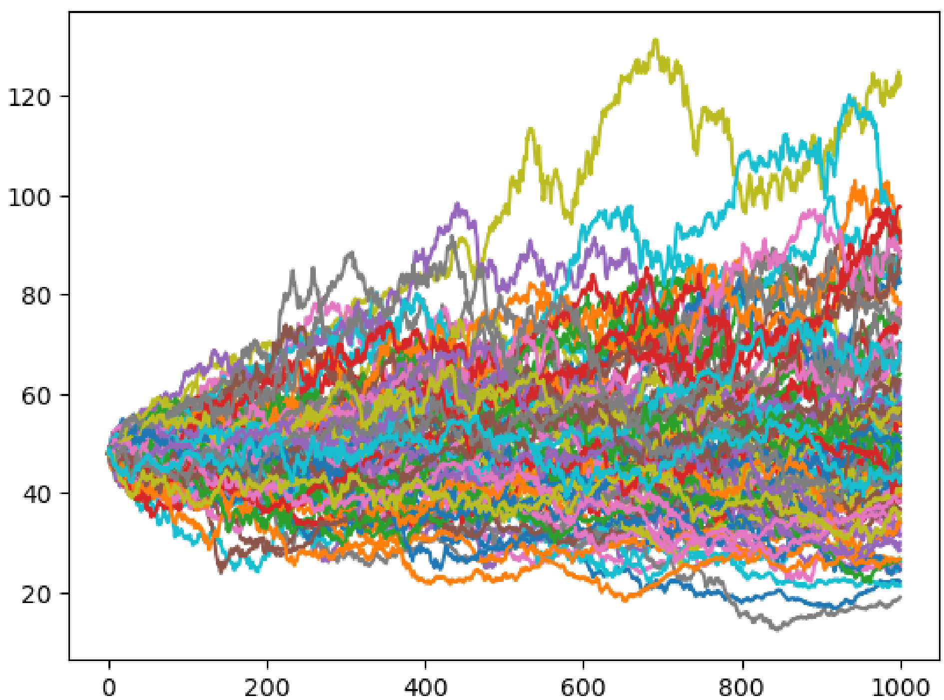
In [35]:
```
payoffs_reflection = [max(0, st_matrix[i][-1] - K) for i in range(len(st_matrix))]
print("C2:", np.exp(-1*r*T)*np.mean(payoffs_reflection))
```

```
C2: 8.337345395602302
```

In [36]:
```
# Partial Truncation Method
z1_matrix, z2_matrix = [], []
np.random.seed(0)
for i in range(100):
    z1 = np.random.normal(size=1000)
    z2 = np.random.normal(size=1000)
    z2 = [z1[i]*rho + z2[i]*np.sqrt(1 - rho**2) for i in range(len(z1))]
    z1_matrix.append(z1)
    z2_matrix.append(z2)
# Estimate the volatility path
def V_t(prev_v, alpha, beta, sigma, dt, z2_i):
    next_v = prev_v + ((alpha*(beta - prev_v))*dt) + (sigma * np.sqrt(abs(prev_v)) * np
    return next_v
# Estimate the stock path
def S_t(prev_s, prev_v, dt, z1_i, r):
    next_s = prev_s + (r * prev_s * dt) + (np.sqrt(max(0, prev_v)) * prev_s * np.sqrt(d
    return next_s

vt_matrix, st_matrix = [], []
for j in range(len(z1_matrix)):
    vt_list, st_list = [v0], [S0]
    for i in range(len(z1_matrix[j])):
        V_k_plus_one = V_t(vt_list[-1], alpha, beta, sigma, dt, z2_matrix[j][i])
        S_k_plus_one = S_t(st_list[-1], vt_list[-1], dt, z1_matrix[j][i], r)
        vt_list.append(V_k_plus_one)
        st_list.append(S_k_plus_one)
    vt_matrix.append(vt_list)
    st_matrix.append(st_list)
```

In [37]:
```
for i in st_matrix:
    plt.plot(i)
```



In [38]:
```
payoffs_partial_trunc = [max(0, st_matrix[i][-1] - K) for i in range(len(st_matrix))]
print("C3:", np.exp(-1*r*T)*np.mean(payoffs_partial_trunc))
```

```
C3: 8.337345395602302
```

Question 5

a)

In [39]:
```
# Define the linear congruential function
def LCG(N, S):
    a = 7**5
    m = 2**31 - 1
    def f(S):
        return (a*S) % m
    U = []
    for i in range(N):
        S = f(S)
        U += [S/m]
    return U
uniform_list = LCG(200, 12)
uniform_list = np.array(uniform_list).reshape(2,100)
```

b)

In [40]:
```
def getHalton(HowMany, Base):
    Seq = np.zeros(HowMany) # Column vector
    NumBits = 1 + math.ceil(np.log(HowMany)/np.log(Base))
    VetBase = 1/(Base**((np.arange(1,NumBits+1))))
    WorkVet = np.zeros(NumBits) # row vector
    for i in range(1, HowMany+1):
        j = 1
        ok = 0
        while ok == 0:
            WorkVet[j] = WorkVet[j] + 1
            if WorkVet[j] < Base:
                ok = 1
            else:
                WorkVet[j] = 0
                j += 1
        Seq[i-1] = np.dot(WorkVet, VetBase)
    return Seq
```

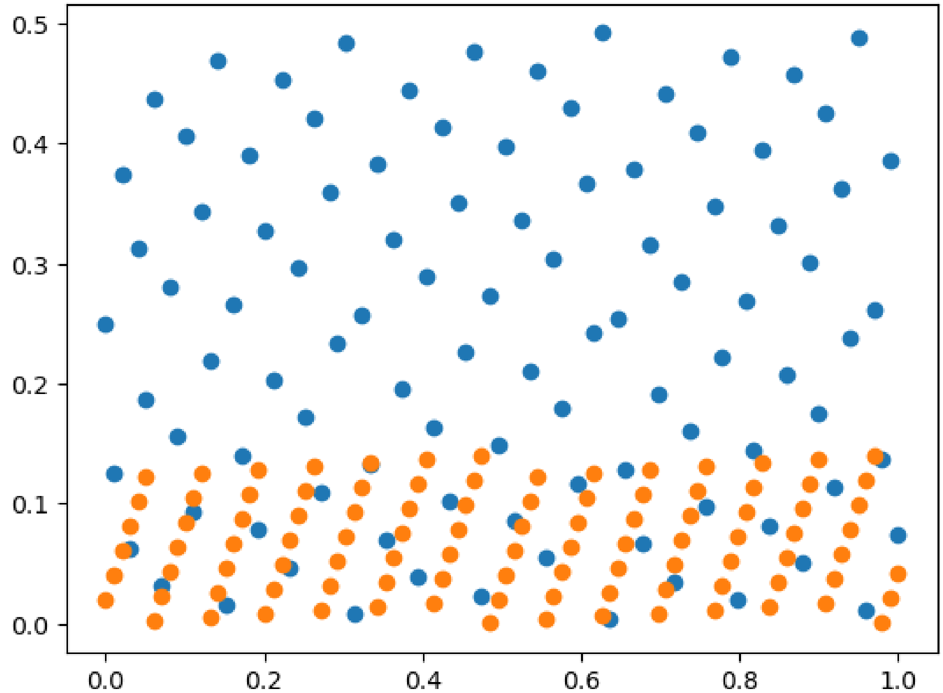
In [41]:
```
base_2 = getHalton(100, 2)
base_7 = getHalton(100, 7)
two_seven = [base_2, base_7]
```

c)

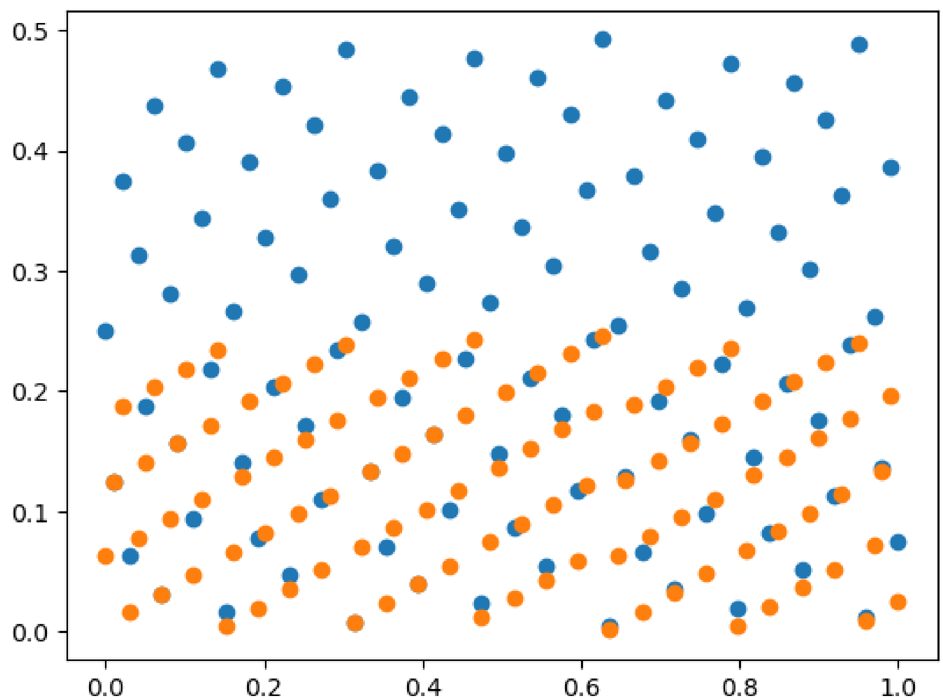
In [42]:
```
base_4 = getHalton(100, 4)
two_four = [base_2, base_4]
```

d)

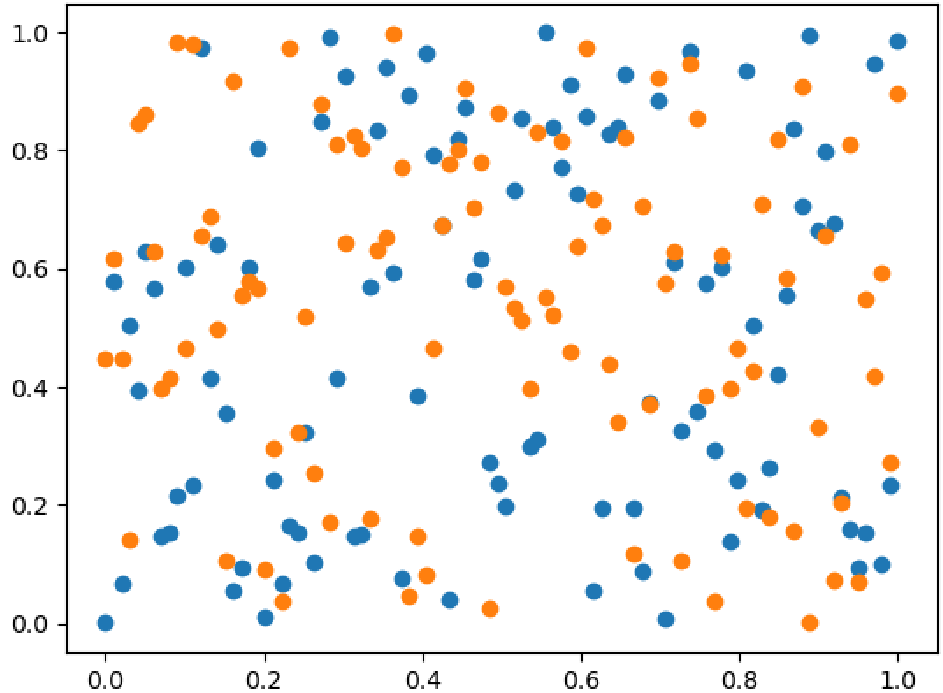
In [43]:
```
for i in range(0, 2):
    plt.scatter(np.linspace(0, 1, 100),two_seven[i])
```



In [44]:
```
for i in range(0, 2):
    plt.scatter(np.linspace(0,1,100), two_four[i])
```



In [45]:
```
for i in range(0, 2):
    plt.scatter(np.linspace(0, 1, 100),uniform_list[i])
```



We see here that if we select base = 2, we get some points that go up until 0.5. However, for bases 4 and 7, we see that the highest points are much lower. When we look at the scatter plot for the uniform distribution, we find that there is an even spread across 0 and 1. Furthermore, the uniform distribution scatter plot looks much more scattered and random while the Halton sequence gives us points that look more deterministic.

e)

In [46]:
```
def f(x, y):
    return np.exp(-x*y)*(np.sin(6*np.pi*x) + np.cbrt(np.cos(2*np.pi*y)))
# (2, 4)
base_2 = getHalton(10000, 2)
base_4 = getHalton(10000, 4)
mean_2_4 = np.mean([f(base_2[i], base_4[i]) for i in range(len(base_2))])
print("(2, 4):", mean_2_4)
```

```
(2, 4): 0.9985330544942149
```

In [47]:
```
# (2, 7)
base_7 = getHalton(10000, 7)
mean_2_7 = np.mean([f(base_2[i], base_7[i]) for i in range(len(base_2))])
print("(2, 7):", mean_2_7)
```

```
(2, 7): 1.1453558186399664
```

In [48]:
```
# (5, 7)
base_5 = getHalton(10000, 5)
mean_5_7 = np.mean([f(base_5[i], base_7[i]) for i in range(len(base_2))])
print("(5, 7):", mean_5_7)
```

```
(5, 7): 1.42402923143162
```

In [ ]: