In [1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
```

Question 1

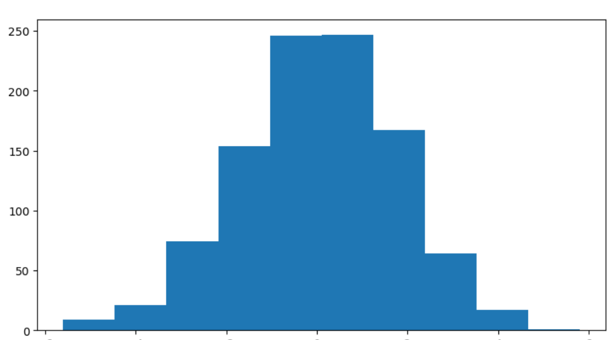
In [97]:
```
# Define the linear congruential function
def LCG(N, S):
    a = 7**5
    m = 2**31 - 1
    def f(S):
        return (a*S) % m
    U = []
    for i in range(N):
        S = f(S)
        U += [S/m]
    return U
```

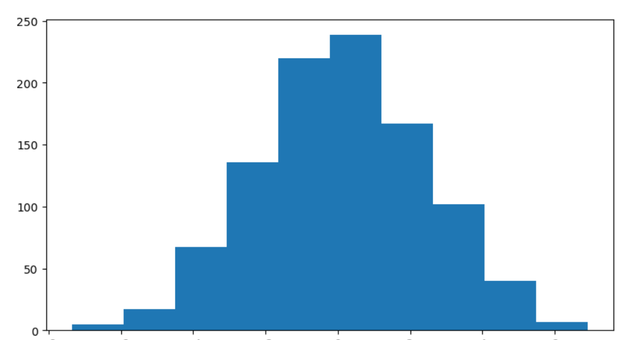
In [98]:
```
# Implement Box-Muller to calculate 1000 standard normals for Xi and Yi
def box_muller(u1, u2):
    z1 = np.sqrt(-2 * np.log(u1)) * math.cos(2 * np.pi * u2)
    z2 = np.sqrt(-2 * np.log(u1)) * math.sin(2 * np.pi * u2)
    return (z1, z2)
```

In [123]:
```
# We will call the LCG function to give us 2000 uniform values that we will split into two to give us
# for each Xi and Yi.
rand_num_uniform = LCG(2000, 100)
u1_list = rand_num_uniform[0:1000]
u2_list = rand_num_uniform[1000:2000]
rand_num_normal = [box_muller(u1_list[i], u2_list[i]) for i in range(len(u1_list))]
z1 = [i[0] for i in rand_num_normal]
z2 = [i[1] for i in rand_num_normal]
```

In [124]:
```
a = -0.7
mean = [0,0]
cov_matrix = [[3, a], [a, 5]]
sigma_1 = cov_matrix[0][0]
sigma_2 = cov_matrix[1][1]
rho = cov_matrix[0][1]/(np.sqrt(sigma_1) * np.sqrt(sigma_2))
x_norm, y_norm = [], []
for i in range(len(z1)):
    X = mean[0] + (np.sqrt(sigma_1)*z1[i])
    Y = mean[1] + (np.sqrt(sigma_2)*rho*z1[i]) + (np.sqrt(sigma_2)*np.sqrt(1-rho**2)*z2[i])
    x_norm.append(X)
    y_norm.append(Y)
```

In [125]:
```
# We plot our X and Y arrays to check whether they are normal and we find from the graphs below that
# resemble a normal distribution
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.hist(x_norm)
ax2.hist(y_norm)
plt.gcf().set_size_inches(20, 5)
```





In [126]:
```
def correlation(x, y):
    x_mean = np.mean(x)
    y_mean = np.mean(y)
    xi_x = [(i - x_mean) for i in x]
    yi_y = [(i - y_mean) for i in y]
    num = (1/(len(x)-1))*np.sum(np.multiply(xi_x, yi_y))
    den_1 = np.sqrt((1/(len(x)-1))*np.sum(np.power(xi_x, 2)))
    den_2 = np.sqrt((1/(len(y)-1))*np.sum(np.power(yi_y, 2)))
    return num/(den_1*den_2)
```

In [127]:
```
# We see that the our estimate for the correlation value is very close to the true correlation value.
print("Correlation Estimator: ", correlation(x_norm, y_norm))
print("True Correlation value: ", rho)
```

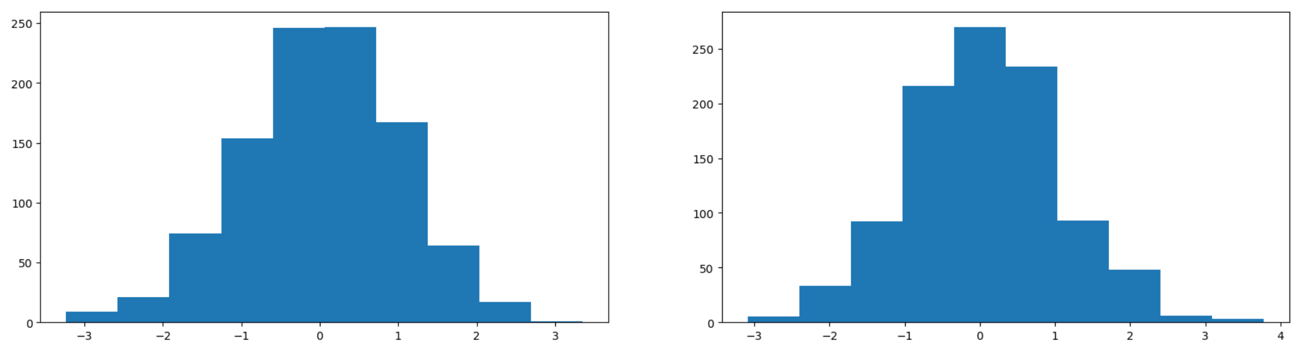
```
Correlation Estimator:  -0.21716481689533618
True Correlation value:  -0.18073922282301277
```

Question 2

In [128]:
```
def payoff(x, y):
    return max(0, (y**3 + math.sin(y) + (x**2*y)))
```

In [129]:
```
# We will reuse the same z1 and z2 we generated in Question 1
mean = [0,0]
cov_matrix = [[1, 0.6], [0.6, 1]]
sigma_1 = cov_matrix[0][0]
sigma_2 = cov_matrix[1][1]
rho = cov_matrix[0][1]/(np.sqrt(sigma_1) * np.sqrt(sigma_2))
x_norm, y_norm = [], []
for i in range(len(z1)):
    X = mean[0] + (np.sqrt(sigma_1)*z1[i])
    Y = mean[1] + (np.sqrt(sigma_2)*rho*z1[i]) + (np.sqrt(sigma_2)*np.sqrt(1-rho**2)*z2[i])
    x_norm.append(X)
    y_norm.append(Y)
```

In [130]:
```
# We see that our new X and Y values also follow a normal distribution
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.hist(x_norm)
ax2.hist(y_norm)
plt.gcf().set_size_inches(20, 5)
```



In [131]:
```
payoffs = [payoff(x_norm[i], y_norm[i]) for i in range(len(x_norm))]
print("Premium of exotic option:", np.mean(payoffs))
```

```
Premium of exotic option: 1.759630825797371
```

Question 3

a)

In [132]:
```
def a(t, z):
    wt = np.sqrt(t) * z
    return (wt**2) + math.sin(wt)
def b(t, z):
    wt = np.sqrt(t)*z
    return np.mean(np.exp(t/2) * np.cos(wt))
```

In [142]:
```
a_payoff_1 = np.mean([a(1, z1[i]) for i in range(len(z1))])
a_payoff_3 = np.mean([a(3, z1[i]) for i in range(len(z1))])
a_payoff_values_5 = [a(5, z1[i]) for i in range(len(z1))]
a_payoff_5 = np.mean(a_payoff_values_5)
print("Payoff of A when t = 1:", a_payoff_1)
print("Payoff of A when t = 3:", a_payoff_3)
print("Payoff of A when t = 5:", a_payoff_5)
```

```
Payoff of A when t = 1: 1.0113446367990022
Payoff of A when t = 3: 2.992222542026916
Payoff of A when t = 5: 4.951738314384789
```

In [134]:
```
b_payoff_1 = np.mean([b(1, z1[i]) for i in range(len(z1))])
b_payoff_3 = np.mean([b(3, z1[i]) for i in range(len(z1))])
b_payoff_5 = np.mean([b(5, z1[i]) for i in range(len(z1))])
print("Payoff of B when t = 1:", b_payoff_1)
print("Payoff of B when t = 3:", b_payoff_3)
print("Payoff of B when t = 5:", b_payoff_5)
```

```
Payoff of B when t = 1: 1.010187193854828
Payoff of B when t = 3: 1.0561076799498101
Payoff of B when t = 5: 1.1530777389021012
```

b)

We see that the payoff values for $B(t)$ is increasing when our t increases. This is because we know that the first variation of our Weiner process scales at $\sqrt{t}$ so as our t increases, the length that the Weiner process travels will increase causing the values that our payoff takes to also increase. In our earlier plots of our z_1 and z_2 we see that the plot is not perfectly normal. We are scaling this imperfect normal distribution by $\sqrt{t}$ which is causing the error in our W_t to also increase

c)

In [136]:
```
# The variance reduction technique we will use will be antithetic variates
z1_negative = [-1*z1[i] for i in range(len(z1))]
```

In [144]:
```
new_a_payoff_1 = np.mean([(a(1, z1[i]) + a(1, z1_negative[i]))/2 for i in range(len(z1))])
new_a_payoff_3 = np.mean([(a(3, z1[i]) + a(3, z1_negative[i]))/2 for i in range(len(z1))])
new_a_payoff_values_5 = [(a(5, z1[i]) + a(5, z1_negative[i]))/2 for i in range(len(z1))]
new_a_payoff_5 = np.mean(new_a_payoff_values_5)
print("New Payoff of A for t=1 using antithetic:", new_a_payoff_1)
print("New Payoff of A for t=3 using antithetic:", new_a_payoff_3)
print("New Payoff of A for t=5 using antithetic:", new_a_payoff_5)
```

```
New Payoff of A for t=1 using antithetic: 0.984954373814069
New Payoff of A for t=3 using antithetic: 2.954863121442207
New Payoff of A for t=5 using antithetic: 4.924771869070346
```

In [145]:
```
new_b_payoff_1 = np.mean([(b(1, z1[i]) + b(1, z1_negative[i]))/2 for i in range(len(z1))])
new_b_payoff_3 = np.mean([(b(3, z1[i]) + b(3, z1_negative[i]))/2 for i in range(len(z1))])
new_b_payoff_5 = np.mean([(b(5, z1[i]) + b(5, z1_negative[i]))/2 for i in range(len(z1))])
print("New Payoff of B for t=1 using antithetic:", new_b_payoff_1)
print("New Payoff of B for t=3 using antithetic:", new_b_payoff_3)
print("New Payoff of B for t=5 using antithetic:", new_b_payoff_5)
```

```
New Payoff of B for t=1 using antithetic: 1.010187193854828
New Payoff of B for t=3 using antithetic: 1.0561076799498101
New Payoff of B for t=5 using antithetic: 1.1530777389021012
```

In [146]:
```
a_var_5 = np.var(a_payoff_values_5)
new_a_var_5 = np.var(new_a_payoff_values_5)
print("Variance without Antithetic Variates:", a_var_5)
print("variance with Antithetic Variates:", new_a_var_5)
```

```
Variance without Antithetic Variates: 50.05305818133142
variance with Antithetic Variates: 49.72597681452127
```

Yes, we see that our variance has improved indicating that our antithetic variates are indeed reducing our variance.

Question 4

a)

In [166]:
```
S0, K, delta_t, r, sigma = 100, 110, 5, 0.05, 0.28

def stock_evolution(S0, K, r, sigma, delta_t, z):
    S_next = S0 * np.exp((r - (sigma**2)/2)*(delta_t) + (sigma)*(np.sqrt(delta_t))*z)
    return S_next
def call_option(St, K):
    return max(0, St-K)
```

In [172]:
```
ST_list = [stock_evolution(S0, K, r, sigma, delta_t, z1[i]) for i in range(len(z1))]
call_prem_list = [call_option(ST_list[i], K) for i in range(len(ST_list))]
call_prem = (np.exp(-r * delta_t))*np.mean(call_prem_list)
print("Call option premium:", "$", call_prem)
```

```
Call option premium: $ 30.3138440009054
```

b)

In [195]:
```
from scipy.stats import norm
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*norm.cdf(d1)- K*norm.cdf(d2)*np.exp(-r*t)
    return call_price
```

In [199]:
```
print("Black Scholes Price:", "$", black_scholes(S0, K, delta_t, r, sigma))
```

```
Black Scholes Price: $ 30.649382950089553
```

c)

In [208]:
```
# Antithetic Variates for call option
ST_list_antithetic = [stock_evolution(S0, K, r, sigma, delta_t, z1_negative[i]) for i in range(len(z1_
call_prem_negative = [call_option(ST_list_antithetic[i], K) for i in range(len(ST_list_antithetic))]
call_prem_antithetic_list = [(call_prem_list[i] + call_prem_negative[i])/2 for i in range(len(call_pre
call_prem_antithetic = (np.exp(-r * delta_t))*np.mean(call_prem_antithetic_list)
print("Call Premium from Antithetic Variates:", "$", call_prem_antithetic)
```

```
Call Premium from Antithetic Variates: $ 30.211641880041142
```

No, we see that the accuracy does not improve due to antithetic variaties. Perhaps if we try anither variance reduction technique such as control variates we would find that the accuracy improves.

Question 5

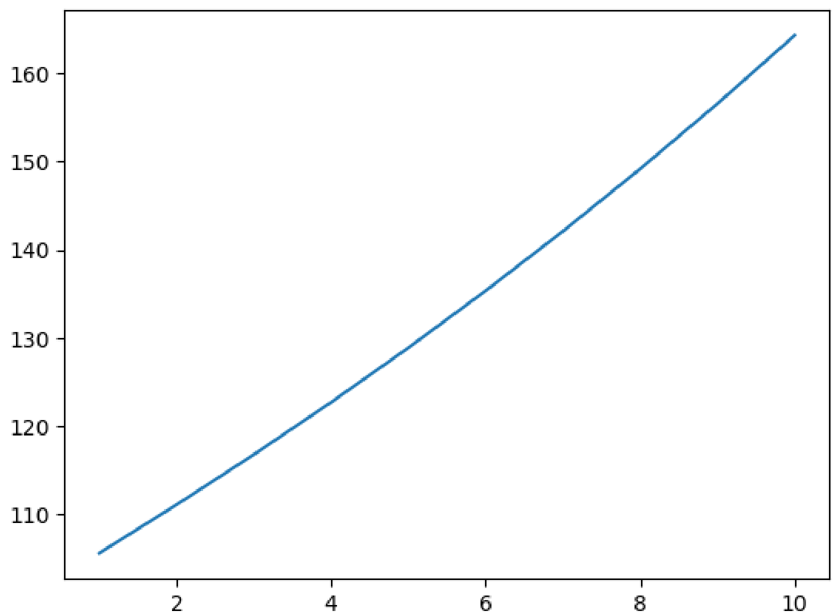
a)

In [214]:
```
S0, K, delta_t, r, sigma = 100, 110, 5, 0.05, 0.28

S1 = np.mean([stock_evolution(S0, K, r, sigma, 1, z1[j]) for j in range(len(z1))])
S2 = np.mean([stock_evolution(S0, K, r, sigma, 2, z1[j]) for j in range(len(z1))])
S3 = np.mean([stock_evolution(S0, K, r, sigma, 3, z1[j]) for j in range(len(z1))])
S4 = np.mean([stock_evolution(S0, K, r, sigma, 4, z1[j]) for j in range(len(z1))])
S5 = np.mean([stock_evolution(S0, K, r, sigma, 5, z1[j]) for j in range(len(z1))])
S6 = np.mean([stock_evolution(S0, K, r, sigma, 6, z1[j]) for j in range(len(z1))])
S7 = np.mean([stock_evolution(S0, K, r, sigma, 7, z1[j]) for j in range(len(z1))])
S8 = np.mean([stock_evolution(S0, K, r, sigma, 8, z1[j]) for j in range(len(z1))])
S9 = np.mean([stock_evolution(S0, K, r, sigma, 9, z1[j]) for j in range(len(z1))])
S10 = np.mean([stock_evolution(S0, K, r, sigma, 10, z1[j]) for j in range(len(z1))])
expected_stock = [S1, S2, S3, S4, S5, S6, S7, S8, S9, S10]
```

In [225]:
```
plt.plot(np.linspace(1, 10, 10), expected_stock)
```

Out[225]:
```
[<matplotlib.lines.Line2D at 0x7f8b626fb050>]
```



b)

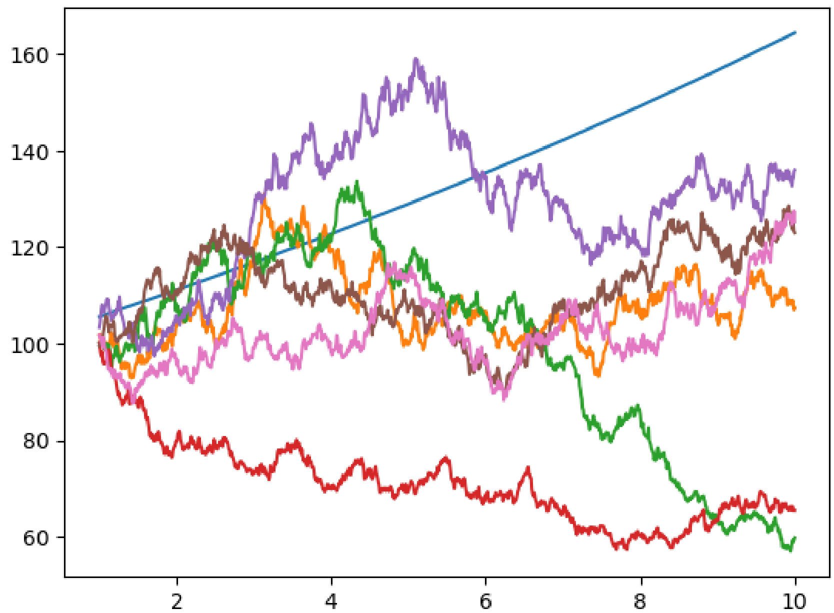
In [235]:
```
def generate_normals(N, S):
    def LCG(N, S):
        a = 7**5
        m = 2**31 - 1
        def f(S):
            return (a*S) % m
        U = []
        for i in range(N):
            S = f(S)
            U += [S/m]
        return U
    def box_muller(u1, u2):
        z1 = np.sqrt(-2 * np.log(u1)) * math.cos(2 * np.pi * u2)
        z2 = np.sqrt(-2 * np.log(u1)) * math.sin(2 * np.pi * u2)
        return (z1, z2)
    rand_num_uniform = LCG(N, S)
    u1_list = rand_num_uniform[0:int(N/2)]
    u2_list = rand_num_uniform[int(N/2):N]
    rand_num_normal = [box_muller(u1_list[i], u2_list[i]) for i in range(len(u1_list))]
    z1 = [i[0] for i in rand_num_normal]
    z2 = [i[1] for i in rand_num_normal]
    return {"z1": z1, "z2":z2}
```

In [265]:
```
six_seeds_normal = []
separate_normals = generate_normals(6000, 20)
combined_normals = separate_normals['z1'] + separate_normals['z2']
for i in range(0, 6000, 1000):
    six_seeds_normal.append(combined_normals[i:i+1000])
```

In [266]:
```
# This piece of code is the bottleneck in the Monte Carlo algorithm as we see that the time complexit
S0, K, delta_t, r, sigma = 100, 110, 0.001, 0.05, 0.28
stock_matrix = []
for i in range(len(six_seeds_normal)):
    stock_path = [S0]
    for j in range(len(six_seeds_normal[i])):
        stock_price = stock_evolution(stock_path[j], K, r, sigma, delta_t, six_seeds_normal[i][j])
        stock_path.append(stock_price)
    stock_matrix.append(stock_path)
```

c)

In [274]:
```
plt.plot(np.linspace(1, 10, 10), expected_stock)
for i in stock_matrix:
    plt.plot(np.linspace(1, 10, 1000), i[1:])
```

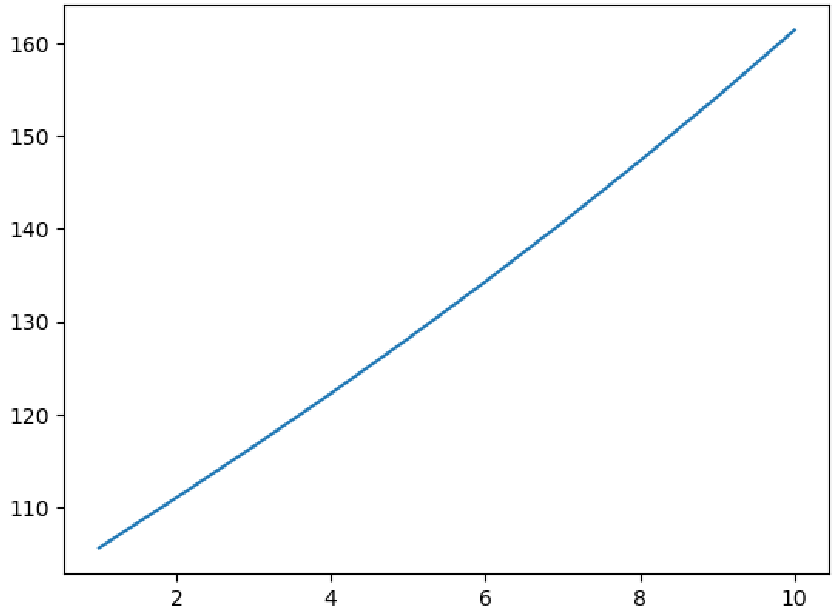


d)

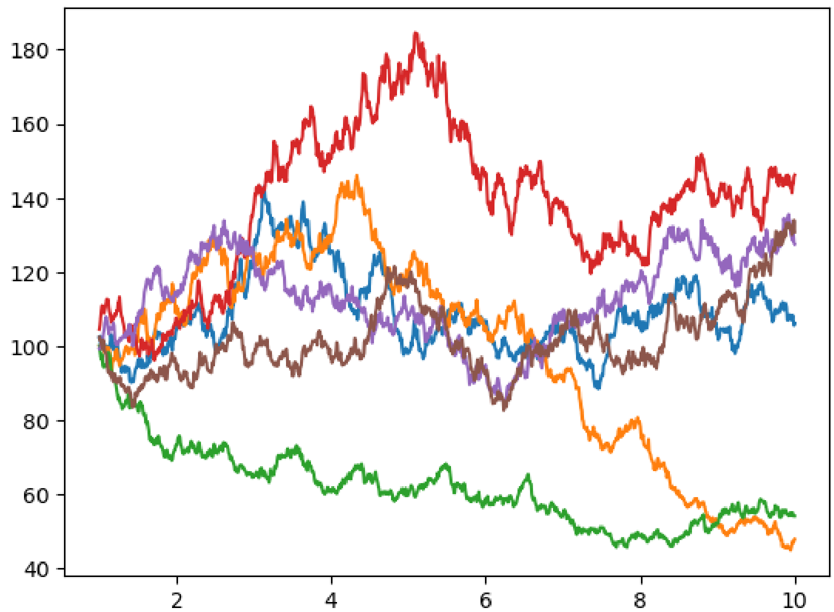
In [275]:
```
S0, K, delta_t, r, sigma = 100, 110, 5, 0.05, 0.38

S1 = np.mean([stock_evolution(S0, K, r, sigma, 1, z1[j]) for j in range(len(z1))])
S2 = np.mean([stock_evolution(S0, K, r, sigma, 2, z1[j]) for j in range(len(z1))])
S3 = np.mean([stock_evolution(S0, K, r, sigma, 3, z1[j]) for j in range(len(z1))])
S4 = np.mean([stock_evolution(S0, K, r, sigma, 4, z1[j]) for j in range(len(z1))])
S5 = np.mean([stock_evolution(S0, K, r, sigma, 5, z1[j]) for j in range(len(z1))])
S6 = np.mean([stock_evolution(S0, K, r, sigma, 6, z1[j]) for j in range(len(z1))])
S7 = np.mean([stock_evolution(S0, K, r, sigma, 7, z1[j]) for j in range(len(z1))])
S8 = np.mean([stock_evolution(S0, K, r, sigma, 8, z1[j]) for j in range(len(z1))])
S9 = np.mean([stock_evolution(S0, K, r, sigma, 9, z1[j]) for j in range(len(z1))])
S10 = np.mean([stock_evolution(S0, K, r, sigma, 10, z1[j]) for j in range(len(z1))])
expected_stock = [S1, S2, S3, S4, S5, S6, S7, S8, S9, S10]
plt.plot(np.linspace(1, 10, 10), expected_stock)
```

Out[275]:
```
[<matplotlib.lines.Line2D at 0x7f8b61efd1d0>]
```



In [279]:
```
S0, K, delta_t, r, sigma = 100, 110, 0.001, 0.05, 0.38
stock_matrix = []
for i in range(len(six_seeds_normal)):
    stock_path = [S0]
    for j in range(len(six_seeds_normal[i])):
        stock_price = stock_evolution(stock_path[j], K, r, sigma, delta_t, six_seeds_normal[i][j])
        stock_path.append(stock_price)
    stock_matrix.append(stock_path)
for i in stock_matrix:
    plt.plot(np.linspace(1, 10, 1000), i[1:])
```



We see that our $E[S_n]$ graph does not change but the graph does change. We see that there is more dispersion in the plots since we have increased our volatility. The $E[S_n]$ graph does not change since we take expectations across each time point so we don't expect this graph to change.

Question 6

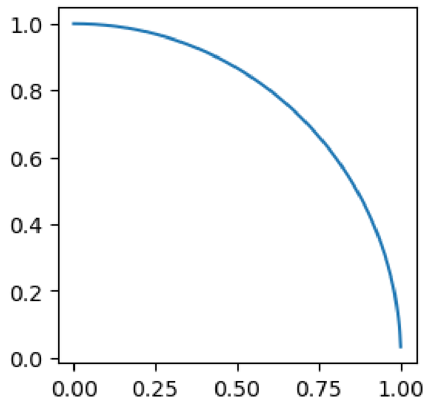
a)

In [323]:
```
# Euler discretization
f = lambda t, s: (-t)/np.sqrt(1 - t**2)
h = 0.001
t = np.arange(0.001, 1 + h, h)
s0 = 1

s = np.zeros(len(t))
s[0] = s0

for i in range(0, len(t) - 1):
    s[i + 1] = s[i] + h*f(t[i], s[i])
```

In [344]:
```
plt.plot(t, s)
plt.gcf().set_size_inches(3,3)
```



In [327]:
```
# We see that using the Euler discretization, we find the integral to be close to pi.
print("Integral evaluated:", 4*(np.sum(np.multiply(s, 0.001))))
```

```
Integral evaluated: 3.141611579122853
```

b)

In [345]:
```
rand_uniform = LCG(10000, 12)
def func(x):
    return np.sqrt(1 - x**2)
func_values = [func(rand_uniform[i]) for i in range(len(rand_uniform))]
mc_estimate_integral = 4*(1/len(rand_uniform))*np.sum(func_values)
print("Monte Carlo Estimate of Integral:", mc_estimate_integral)
```

```
Monte Carlo Estimate of Integral: 3.1448037955171424
```

c)

In [350]:
```
def importance_sampling(y, a):
    func = (math.sqrt(1 - (y**2)) * (1 - (a / 3))) / (1 - (a * y**2))
    return func

def calculate_integral_IS(a):
    array = []
    for i in range(len(rand_uniform)):
        array.append(importance_sampling(rand_uniform[i], a))
    I = 4 * np.mean(array)
    return I

integral_value = calculate_integral_IS(0.76)
print("Integral using Importance Sampling:", integral_value)
```

```
Integral using Importance Sampling: 3.1476328645546747
```

In [ ]: