The code is written in C++.

Question 1

Using the LGM method, generate Uniformly distributed random numbers on [0,1] to do the following:

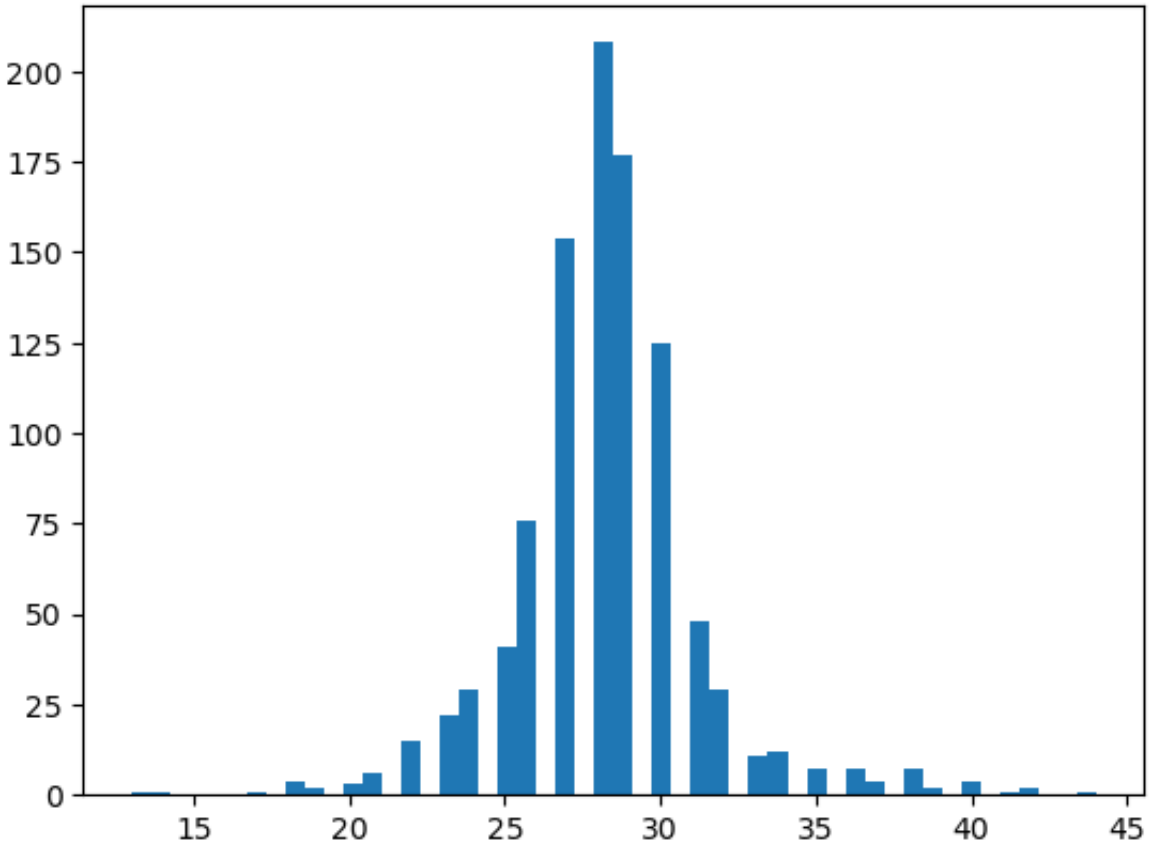
(a) Generate 1,000 random numbers with Binomial distribution with n = 44 and p = 0.64. Compute the probability that the random variable X, that has Binomial (44, 0.64) distribution, is at least 40: $P(X \geq 40)$.

In [ ]:
```
void exponential(float lambda, float X1, float X2, int N = 10000)
{
    // // simulate exponentially distributed random numbers
    float arr[N], std=0, mean;
    for(int i = 0; i < N; i++)
    {
        arr[i] = - lambda * log(1-uniform());
    }
    mean = arr_mean(arr,N);
    for(int i = 0; i < N; i++)
    {
        std += pow(arr[i]-mean,2);
    }
    cout << "The empirical mean is " << mean << " and the standard deviat
    cout << "P(X >= 1) = " << prob(arr, X1, N) << ", P(X >= 4) = " << pro
    // // print 10,000 Exponentially random numbers
    cout << "The 10,000 Exponentially random numbers: " ;
    for (int i = 0; i < N; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

OUTPUT_1_a

- P(X >= 35) = 0.035

In [25]:
```
data = [28, 29, 27, 29, 30, 26, 30, 29, 26, 29, 30, 27, 27, 28, 27, 28, 2
plt.hist(data,bins = 50)
plt.show()
```



(b) Generate 10,000 Exponentially distributed random numbers with parameter λ = 1.5. Estimate $P(X \geq 1)$; $P(X \geq 4)$; and compute the empirical mean and the standard deviation of the sequence of 10,000 numbers. Draw the histogram by using the 10,000 numbers you have generated.

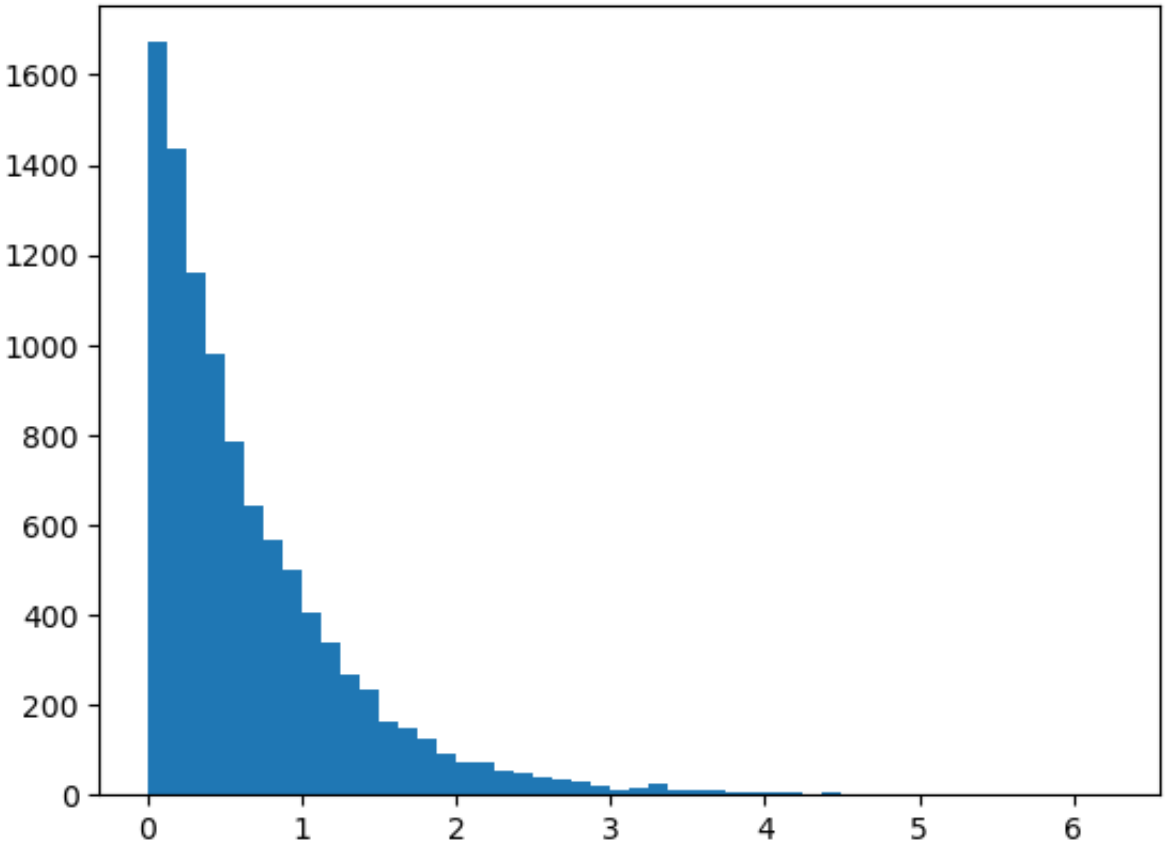
In [ ]:
```
template<typename T>
T arr_mean(T array[], int N)
{
    # // int N = sizeof(array) / sizeof(array[0]);
    T sum = 0;
    for(int i = 0; i < N; i++)
    {
        sum += array[i];
    }
    return sum/N;
}

void exponential(float lambda, float X1, float X2, int N = 10000)
{
    # // simulate exponentially distributed random numbers
    float arr[N], std=0, mean;
    for(int i = 0; i < N; i++)
    {
        arr[i] = - lambda * log(1-uniform());
    }
    mean = arr_mean(arr,N);
    for(int i = 0; i < N; i++)
```

```
    {
        std += pow(arr[i]-mean,2);
    }
    cout << "The empirical mean is " << mean << " and the standard deviat
    cout << "P(X >= 1) = " << prob(arr, X1, N) << ", P(X >= 4) = " << pro
    // // print 10,000 Exponentially random numbers
    cout << "The 10,000 Exponentially random numbers: " ;
    for (int i = 0; i < N; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

OUTPUT_1_b

- The empirical mean is 0.670216 and the standard deviation is 0.66926
- P(X >= 1) = 0.2259, P(X >= 4) = 0.0025

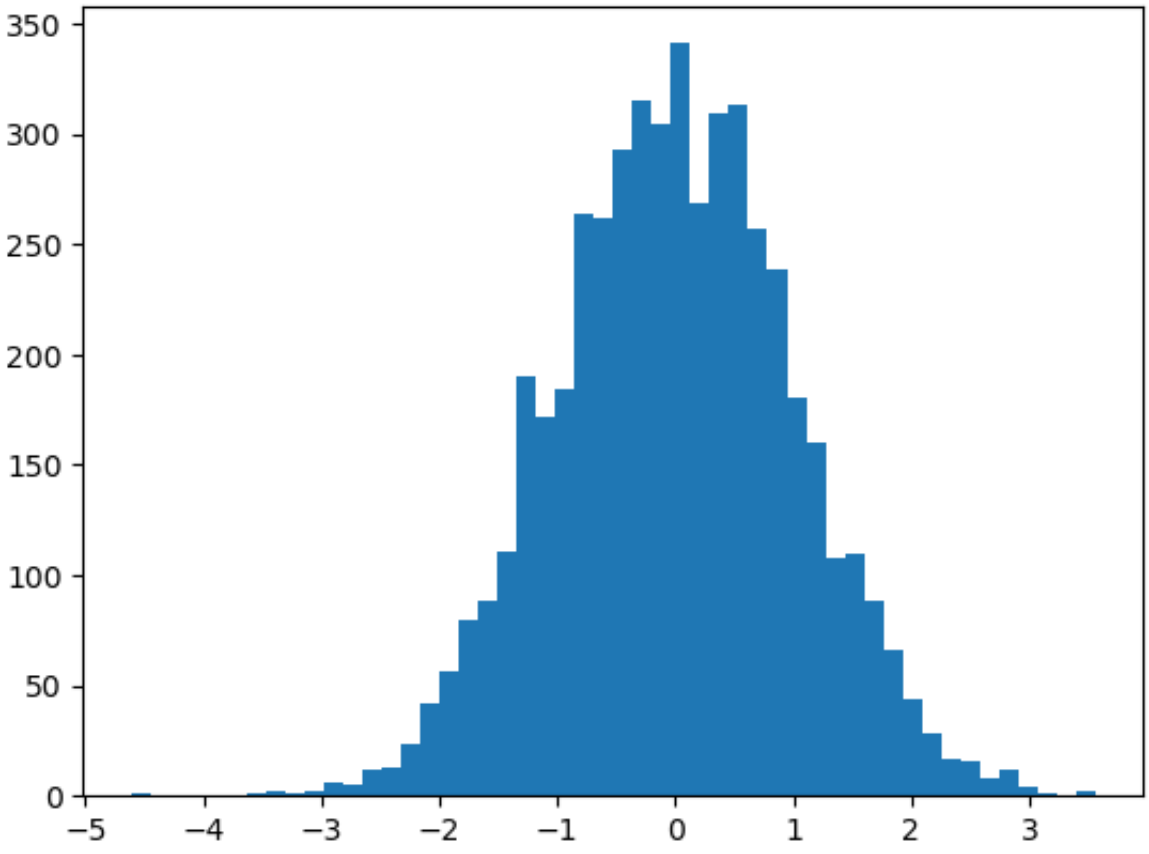
In [22]:
```
data = [0.215565, 0.509547, 1.0093, 1.01181, 1.01981, 0.499768, 0.22514,
plt.hist(data,bins = 50)
plt.show()
```



(c) Generate 5,000 Normally distributed random numbers with mean 0 and variance 1, by using the Box- Muller Method.

In [ ]:
```
float box_muller(int N)
{
    float arr[2*N], u1, u2;
    for(int i = 0; i < N; i++)
    {
        u1 = uniform();
        u2 = uniform();
        arr[2*i] = sqrt(-2*log(u1))*cos(2*M_PI*u2);
        arr[2*i+1] = sqrt(-2*log(u1))*sin(2*M_PI*u2);
    }
    if(N > 1)
    {
        cout << "The 5,000 box_muller random numbers: " ;
        for (int i = 0; i < 2*N; ++i)
        {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
    return arr[0];
}
```

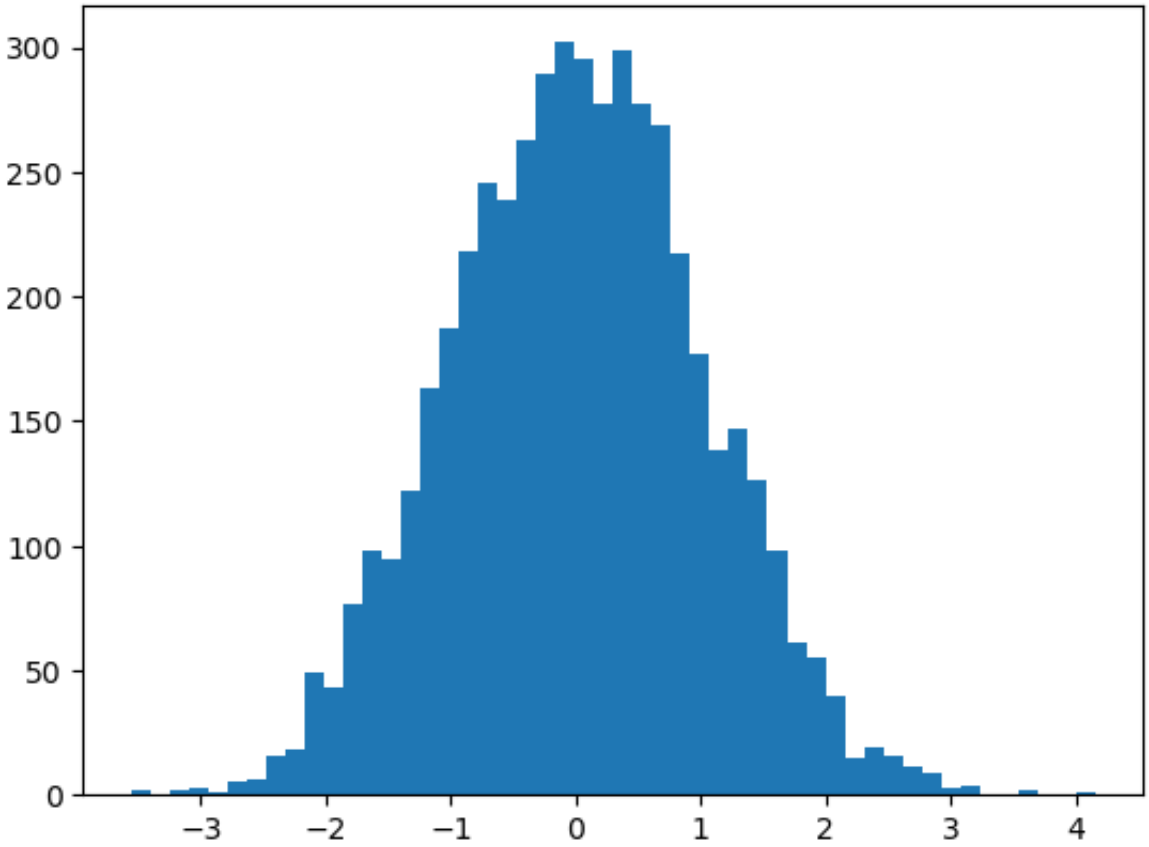
In [23]:
```
data = [0.744701, -0.797455, -0.311815, 0.514636, -0.327482, 0.926324, 0.
plt.hist(data,bins = 50)
plt.show()
```



(d) Generate 5,000 Normally distributed random numbers with mean 0 and variance 1, by using the Polar-Marsaglia method.

In [ ]:
```
float polar_marsaglia(int N)
{
    float arr[2*N];
    float u1, u2, v1, v2, w;
    int i = 0;
    while(i < 2*N)
    {
        u1 = uniform();
        u2 = uniform();
        v1 = 2*u1-1;
        v2 = 2*u2-1;
        w = pow(v1,2)+pow(v2,2);
        if (w <= 1) {
            float multiplier = sqrt((-2 * log(w)) / w);
            arr[i] = v1 * multiplier; // First normal variable
            arr[i+1] = v2 * multiplier; // Second normal variable
            i += 2;
        }
    }
    cout << "The 5,000 polar_marsaglia random numbers: " ;
    for (int i = 0; i < 2*N; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    return arr[0];
}
```

In [24]:
```
data = [-1.29837, -1.45004, -0.559208, -0.0820156, -0.457915, 0.17363, 0.
plt.hist(data,bins = 50)
plt.show()
```



(e) Now compare the efficiencies of the two above-algorithms, by comparing the execution times to generate 5,000 normally distributed random numbers by the two methods. Which one is more efficient? If you do not see a clear difference, you need to increase the number of generated realizations of random variables to 10,000, 20,000, etc

In [ ]:
```
auto time_1 = chrono::high_resolution_clock::now();
cout << "1(c):" << endl;
box_muller(2500);
auto time_2 = chrono::high_resolution_clock::now();
cout << "1(d):" << endl;
polar_marsaglia(2500);
auto time_3 = chrono::high_resolution_clock::now();
cout << "1(e)" << endl;
chrono::duration<double> delta_1 = time_2 - time_1;
chrono::duration<double> delta_2 = time_3 - time_2;
cout << "The time of box_muller is  " << delta_1.count() << " s\n";
cout << "The time of polar_marsaglia is  " << delta_2.count() << " s\n";
```

OUTPUT_1_e

- The time of box_muller is 0.000334289 s
- The time of polar_marsaglia is 0.00057104 s
- We can see that Box Muller is more efficient.

Question 2

(a) Estimate the following expected values by simulation:
$A(t) = E(W_t^2 + sin(W_t))$, $B(t) = E(e^{t/2}cos(W_t))$, for t = 1,3,5. Here, W_t is a Standard Wiener Process.

In [ ]:
```
void calculate_AB(int t)
{
    double w, A[N_simulate], B[N_simulate];
    for(int i = 0; i < N_simulate; i++)
    {
        w = sqrt(t) * dis(gen);
        //cout << w << endl;
        A[i] = pow(w,2.0) + sin(w);
        B[i] = exp(t/2.0) * cos(w);
    }
    cout << "At time " << t << " - A(t):" << arr_mean(A,N_simulate) << ",
}
```

OUTPUT_2_a

- t time 1 - A(t):0.993443, B(t):0.99963
- At time 3 - A(t):2.99192, B(t):0.988174
- At time 5 - A(t):4.96999, B(t):1.03677

(b) How are the values of $B(t)$ (for the cases t = 1,3,5) related?

The value of B(t) is around 1 since $B(t) = e^{t/2}E[cos(W_t)]$, $e^{t/2}$ will be offset by $E[cos(W_t)]$, generating a theoretical answer of 1.

(c) Now use a variance reduction technique (whichever you want) to compute the expected value $B(5)$. Do you see any improvements? Comment on your findings.

Antithetic Variates

- $A(t) = \frac{1}{N}\sum((W_t^2 + sin(W_t) + (-W_t)^2 + sin(-W_t))/2)/N = \frac{1}{N}\sum(W_t^2)$
- $B(t) = \frac{1}{2N}\sum(e^{t/2} * cos(W_t) + e^{t/2} * cos(-\frac{1}{2}W_t + \frac{\sqrt{3}}{2}W_t'))$

In [ ]:
```
float B_anti_vars(int t)
{
    float w, u, B[N_simulate];
    for(int i = 0; i < N_simulate; i++)
    {
        w = sqrt(t) * dis(gen);
        u = - 0.3 * w + sqrt(1-pow(0.3,2.0)) * t * dis(gen);
        B[i] = (exp(t/2.0)*cos(w) + exp(t/2.0)*cos(w))/2.0;
    }
```

```
    return arr_mean(B, N_simulate);
}
```

OUTPUT_2_c

- B(5) = 1.01844

- B(5) is closer to 1.

Question 3

Let S_t be a Geometric Brownian Motion process: $S_t = S_0 e^{\sigma W_t + (r - \sigma^2/2)t}$, where $r = 0.055, \sigma = 0.2, S_0 = \$100$; W_t is a Standard Brownian Motion process (Standard Wiener process).

(a) Estimate the price c of a European Call option on the stock with T=5,X=\$100 by using Monte Carlo simulation.

In [ ]:
```
template<typename T1, typename T2>
auto max(T1 a, T2 b) -> typename std::common_type<T1, T2>::type {
    return (a > b) ? a : b;
}
float call(float S0, float r, float sigma, int T, float X)
{
    float arr[N_simulate];
    for(int i = 0; i < N_simulate; i++)
    {
        arr[i] = max(exp(-r*T)*(S0*exp(sigma*sqrt(T)*dis(gen)+(r-pow(sigm
    }
    return arr_mean(arr, N_simulate);
}
```

OUTPUT_3_a

- 3(a): 33.3826

(b) Compute the exact value of the option c using the Black-Scholes formula.

In [ ]:
```
double norm_cdf(double value)
{
    return 0.5 * erfc(-value * M_SQRT1_2);
}

float bs_call(float S0, float r, float sigma, int t, float X)
{
    double d1 = (log(S0 / X) + (r + 0.5 * sigma * sigma) * t) / (sigma *
```

```
    double d2 = d1 - sigma * sqrt(t);
    return S0 * norm_cdf(d1) - X * exp(-r * t) * norm_cdf(d2);
}
```

OUTPUT_3_b

- 3(b): 33.6102

(c) Use variance reduction techniques (whichever one(s) you want) to estimate the price in part (a) again using the same number of simulations. Did the accuracy improve? Compare your findings and comment.

In [ ]:
```
float anti_vars(float S0, float r, float sigma, int t, float X)
{
    float arr[N_simulate], w;
    for(int i = 0; i < N_simulate; i++)
    {
        w = dis(gen);
        arr[i] = (max(exp(-r*t)*(S0*exp(sigma*sqrt(t)*w + (r-pow(sigma,2.
    }
    return arr_mean(arr, N_simulate);
}
```

OUTPUT_3_c

- 3(c): 33.6645

Question 4

(a) For each integer number n from 1 to 10, use 1,000 simulations of Sn to estimate $E(Sn)$, where St is a Geometric Brownian Motion process: $St=S0e(\sigma Wt+(r-\sigma^2/2)t)$, where r=0.055,σ=0.20,$S0$=$88. Plot all of the above $E(Sn)$, for n ranging from 1 to 10, in one graph.

In [ ]:
```
void ES_n(float r, float sigma, float S0)
{
    float ES[10];
    for(int n = 1; n <= 10; n++)
    {
        ES[n-1] = 0;
        for(int i = 0; i < N_simulate; i++)
        {
            ES[n-1] += S0*exp(sigma*dis(gen)*sqrt(n) + (r-pow(sigma,2.0)/
        }
        ES[n-1] /= float(N_simulate);
    }
    for(int n = 1; n <= 10; n++)
```

```
    {
        cout << ES[n-1] << ", ";
    }
    cout << endl;
}
```

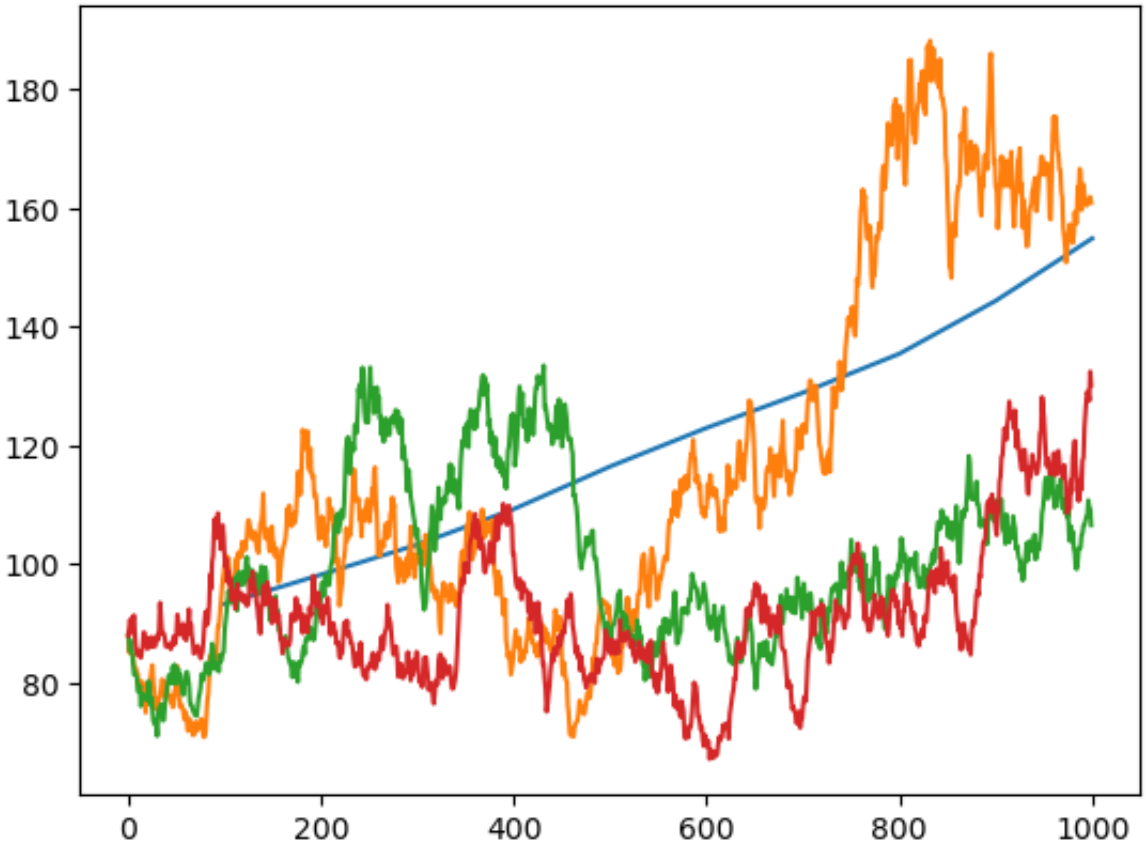
(b) Now simulate 3 paths of St for $0 \leq t \leq 10$ (defined in part (a)) by dividing up the interval [0, 10] into 1,000 equal parts

In [ ]:
```
void simulate_paths(float r, float sigma, float S0, float n)
{
    float S[int(n)][1000], w, dt = 0.01;
    for(int i = 0; i < n; i++)
    {
        S[i][0] = S0;
        //cout << "[ " << S0;
        for(int j = 1; j < 1000; j++)
        {
            w = sqrt(dt) * dis(gen);
            S[i][j] = S[i][j-1] + S0*exp(sigma*w + (r-pow(sigma,2.0)/2.0)
            //cout << ", " << S[i][j];
        }
        //cout << "]" << endl;
    }
}
```

(c) Plot your data from parts (a) and (b) in one graph.

In [55]:
```
ES_n = [93.3449, 98.2692, 103.326, 109.322, 116.506, 123.008, 128.943, 13
S_1 = [ 88, 85.1966, 86.7557, 85.0539, 84.9187, 84.6688, 83.0591, 82.5993
S_2 = [ 88, 87.1072, 85.3365, 86.9905, 85.2612, 83.7175, 81.4202, 81.6282
S_3 = [ 88, 89.2298, 90.7146, 89.1407, 88.4221, 91.1559, 91.4137, 87.8369
plt.plot(np.arange(100,1100,100), ES_n)
plt.plot(range(1000), S_1)
plt.plot(range(1000), S_2)
plt.plot(range(1000), S_3)
```

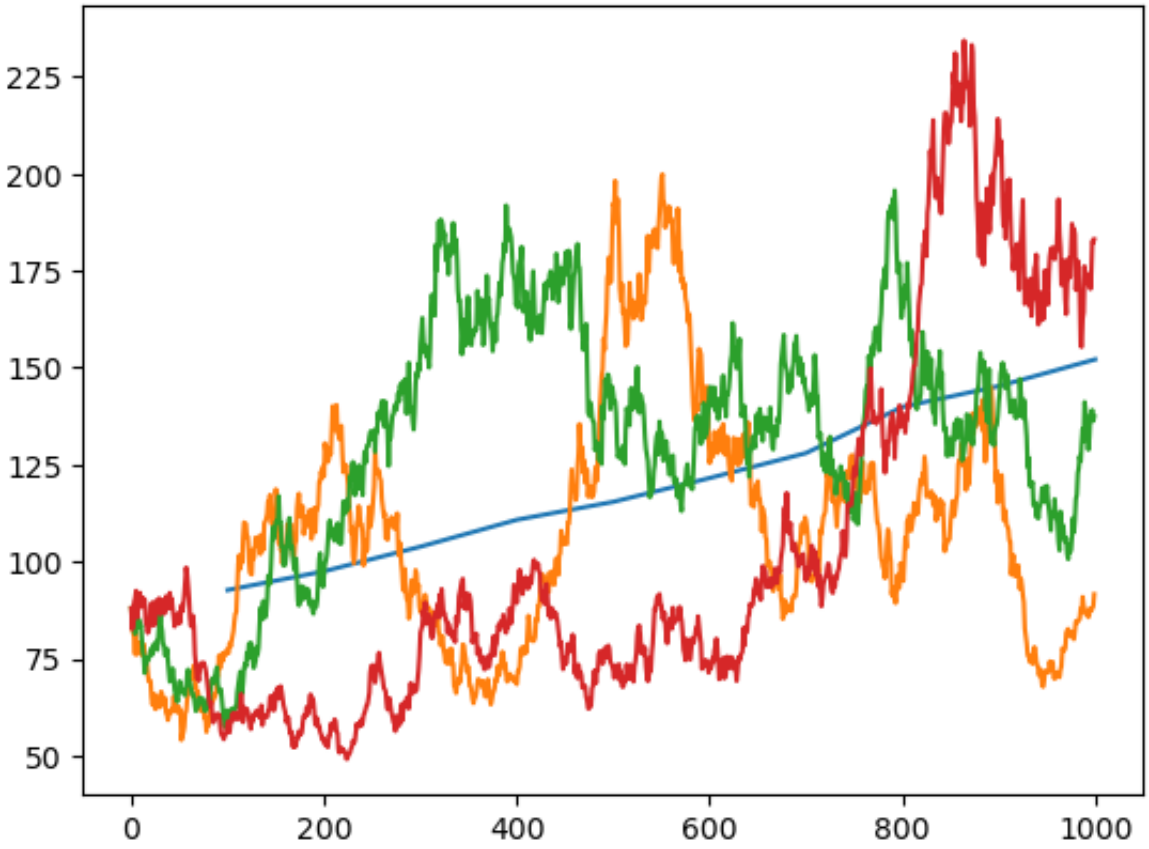
Out[55]:
```
[<matplotlib.lines.Line2D at 0x13d4da6d0>]
```



(d) What would happen to the $E(Sn)$ graph if you increased σ from 20% to 30%? What would happen to the 3 plots of St for $0 \leq t \leq 10$, if you increased σ from 20% to 30%?

In [54]:
```
ES_n =  [92.6725, 97.5122, 103.857, 110.792, 115.416, 121.553, 127.99, 13
S_1 = [ 88, 82.5414, 81.6572, 80.5193, 76.1402, 82.0553, 76.0874, 78.5979
S_2 = [ 88, 84.4422, 85.3829, 84.4275, 81.3728, 82.5924, 85.449, 84.5795,
S_3 = [ 88, 82.6173, 84.4434, 89.5197, 90.514, 92.4622, 85.5091, 88.8796,
plt.plot(np.arange(100,1100,100), ES_n)
plt.plot(range(1000), S_1)
plt.plot(range(1000), S_2)
plt.plot(range(1000), S_3)
```

Out[54]:
```
[<matplotlib.lines.Line2D at 0x13d4e5350>]
```



Question 5

(a) Write a code to compute prices of European Call options via Monte Carlo simulation of paths of the stock price process. Use Euler's discretization scheme to discretize the SDE for the stock price process. The code should be generic: for any input of the 5 model parameters - $S0$, T, X, r, σ – the output is the corresponding price of the European call option and the standard error of the estimate.

In [ ]:
```
void euler_call(float S0, float T, float X, float r, float sigma, float d
{
    float S, call[N_simulate], std=0;
    # // dS = rSdt + sigmaSdW
    for(int i = 0; i < N_simulate; i++)
    {
        S = S0;
        for(float t = 0; t < T; t = t + dt)
        {
            S += r*S*dt + sigma*S*sqrt(dt)*dis(gen);
        }
        call[i] = max(S-X,0);
    }
    float mean = arr_mean(call, N_simulate);
    cout << " | " << mean;
    for(int i = 0; i < N_simulate; i++)
    {
```

```
        std += pow(call[i]-mean,2.0);
    }
    cout << " | " << sqrt(std) << " | " << endl;
}
```

(b) Write a code to compute prices of European Call options via Monte Carlo simulation of paths of the stock price process. Use Milshtein's discretization scheme to discretize the SDE for the stock price process. The code should be generic: for any input of the 5 model parameters - $S0$, T, X, r, σ – the output is the corresponding price of the European call option and the standard error of the estimate.

In [ ]:
```
void milshtein_call(float S0, float T, float X, float r, float sigma, flo
{
    # // dS = rSdt + sigmaSdW + sigma*sigma*S*(w**2-dt)/2
    float S, call[N_simulate], std=0, w;
    for(int i = 0; i < N_simulate; i++)
    {
        S = S0;
        for(float t = 0; t < T; t = t + dt)
        {
            w = sqrt(dt)*dis(gen);
            S += r*S*dt + sigma*S*w + pow(sigma,2.0)*S*(pow(w,2.0)-dt)*0.
        }
        call[i] = max(S-X,0);
    }
    double mean = arr_mean(call, N_simulate);
    cout << " | " << mean;
    for(int i = 0; i < N_simulate; i++)
    {
        std += pow(call[i]-mean,2.0);
    }
    cout << " | " << sqrt(std) << " | " << endl;
}
```

(c) Write 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 - $S0$, T, X, r, σ - the output is the corresponding price of the European call option.

In [ ]:
```
double cdf_N(double x)
{
    if(x>=0)
        return 1 - pow(1+ 0.0498673470*x+ 0.0211410061*pow(x,2.0)+ 0.0032
    else
        return 1 - cdf_N(-x);
}
double bs_call_cdf(float S0, float r, float sigma, float t, float X)
{
    double d1 = (log(S0 / X) + (r + 0.5 * sigma * sigma) * t) / (sigma *
    double d2 = d1 - sigma * sqrt(t);
```

```
    return S0 * cdf_N(d1) - X * exp(-r * t) * cdf_N(d2);
}
```

(d) Use the results of (a) to (c) to compare the two schemes of parts (a) and (b) by computing the following European call option prices: X = 100, σ = 0.25, r = 0.055, T = 0.5, and the range [95, 104] for $S0$, with a step size of 1.

- **euler_call:**

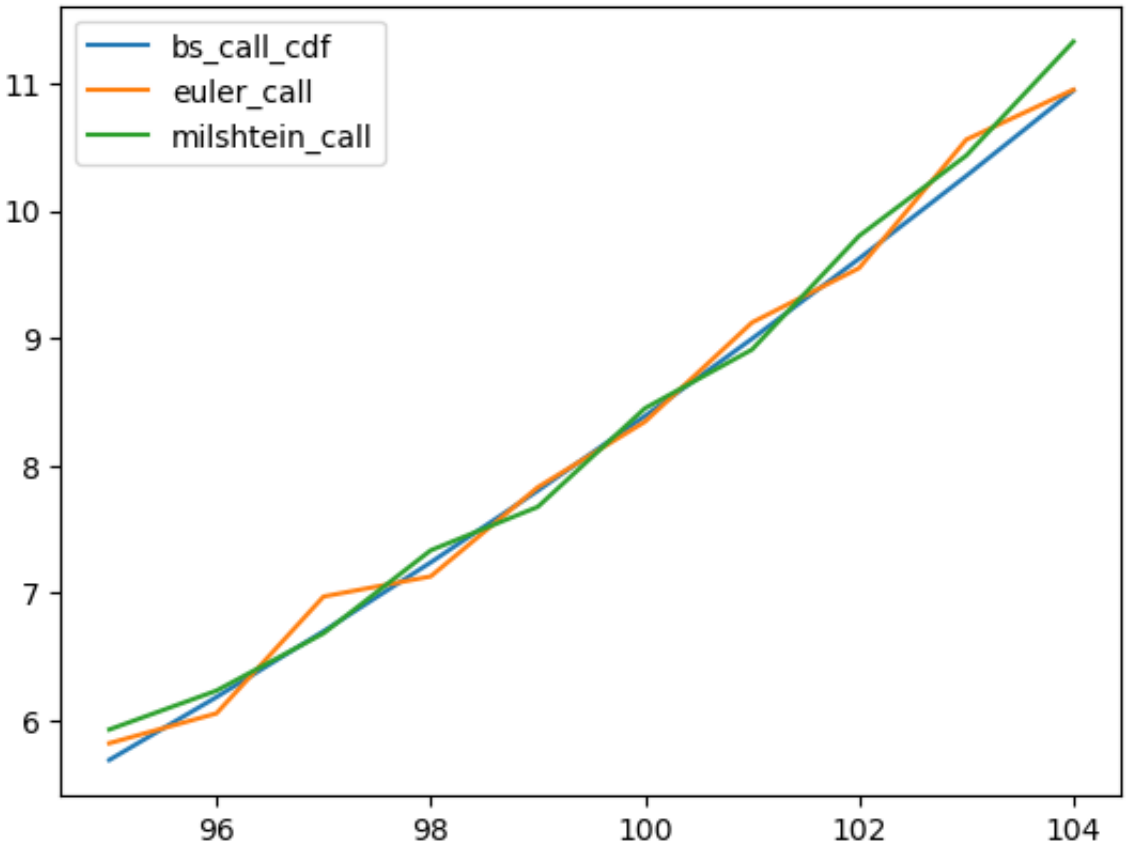
BSM	The estimate	standard error
5.68895	5.82012	1029.92
6.18233	6.05451	1038.06
6.69922	6.97165	1133.74
7.23933	7.12947	1134.03
7.80228	7.8291	1196.89
8.38763	8.34329	1233.06
8.99486	9.12197	1299.38
9.62345	9.54928	1309.42
10.2728	10.5573	1385.15
10.9423	10.9486	1406.23

- **milshtein_call**

BSM	The estimate	standard error
5.68895	5.92768	1054.07
6.18233	6.23274	1058.18
6.69922	6.68288	1112.46
7.23933	7.33404	1160.12
7.80228	7.67535	1172.48
8.38763	8.44866	1244.33
8.99486	8.90975	1263.98
9.62345	9.80168	1346.34
10.2728	10.4321	1370.88
10.9423	11.3264	1447.72

In [56]:
```
bs_call_cdf = [5.68895, 6.18233, 6.69922, 7.23933, 7.80228, 8.38763, 8.99
euler_call = [5.82012, 6.05451, 6.97165, 7.12947, 7.8291, 8.34329, 9.1219
milshtein_call = [5.92768, 6.23274, 6.68288, 7.33404, 7.67535, 8.44866, 8
plt.plot(range(95,105),bs_call_cdf,label='bs_call_cdf')
plt.plot(range(95,105),euler_call,label='euler_call')
plt.plot(range(95,105),milshtein_call,label='milshtein_call')
plt.legend()
```

Out[56]:
```
<matplotlib.legend.Legend at 0x13d4db010>
```



- They are really close to the BSM value.

OUTPUT_5_d

(e) Estimate the European call option's greeks – delta (Δ), gamma (Γ), theta (θ), and vega (ν) - and graph them as functions of the initial stock price $S0$. Use X = 100, σ = 0.25, r = 0.055 and T = 0.5 in your estimations. Use the range [95, 105] for $S0$, 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 = 0.05 as a default value.

In [ ]:
```
void greeks(float r, float sigma, float T, float X, float dt = 0.05)
{
    float delta[11], gamma[11], theta[11], vega[11];
    int S0_list[11] = {95,96,97,98,99,100,101,102,103,104,105};
    cout << "[" ;
```

```
    for(int i = 0; i <= 10; i++)
    {
        delta[i] = (bs_call_cdf(S0_list[i]+1,r,sigma,T,X) - bs_call_cdf(S
        gamma[i] = (bs_call_cdf(S0_list[i]+2,r,sigma,T,X) - bs_call_cdf(S
        theta[i] = (bs_call_cdf(S0_list[i],r,sigma,T-0.1,X) - bs_call_cdf
        vega[i] = (bs_call_cdf(S0_list[i],r,sigma,T,X) - bs_call_cdf(S0_l
        cout << "[ " << delta[i] << " , " << gamma[i]  << " , " << theta[
    }
    cout << "]" ;
}
```

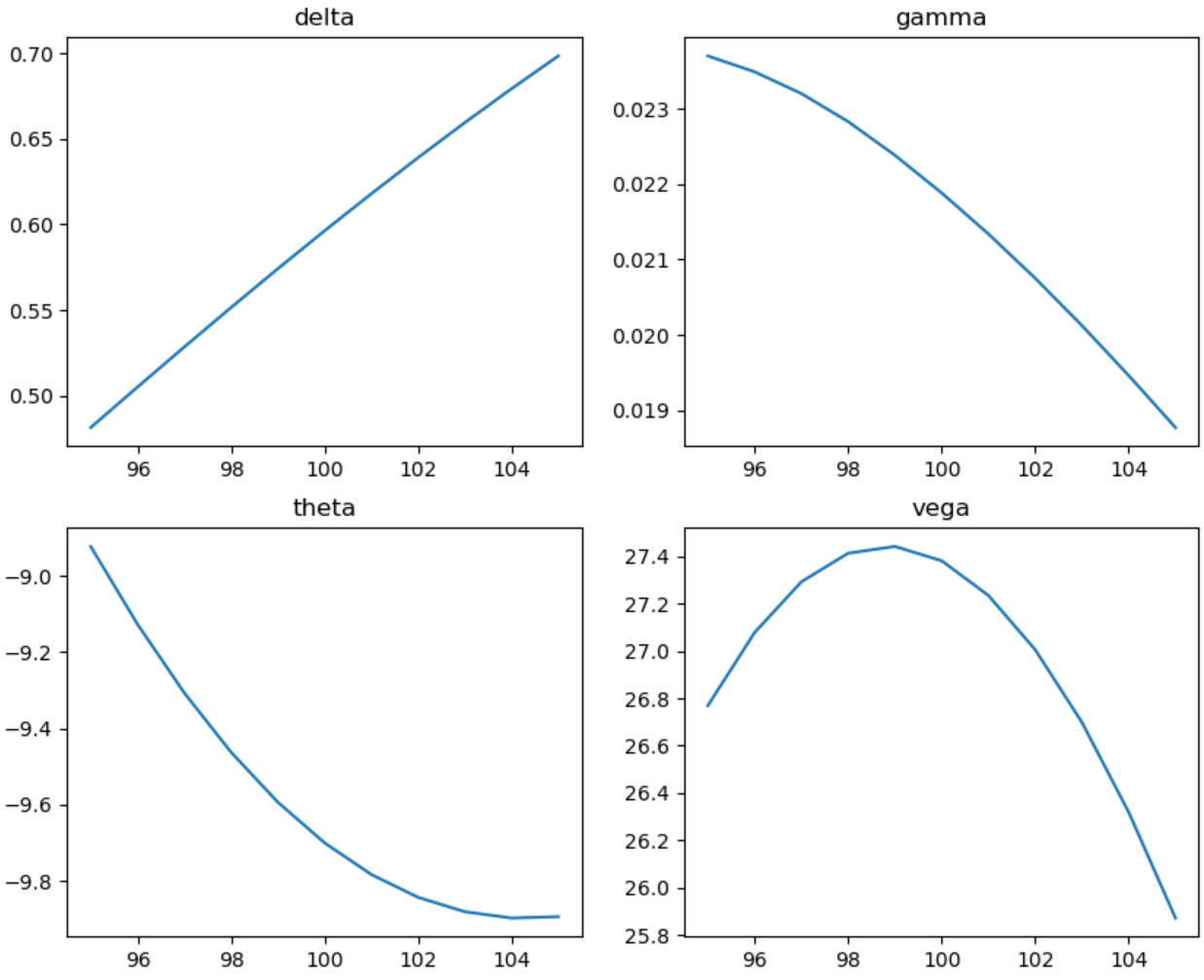
OUTPUT_5_d

S_0	delta	gamma	theta	vega
95	0.481527	0.0236961	-8.92209	26.7698
96	0.505136	0.0234856	-9.12631	27.0785
97	0.528498	0.0231972	-9.3065	27.293
98	0.55153	0.0228255	-9.46224	27.4137
99	0.574149	0.0223798	-9.59345	27.4423
100	0.59629	0.0218798	-9.70037	27.3818
101	0.617909	0.0213364	-9.78352	27.235
102	0.638963	0.0207493	-9.84358	27.0058
103	0.659407	0.0201216	-9.88138	26.6987
104	0.679206	0.0194597	-9.89793	26.3188
105	0.698327	0.0187698	-9.89437	25.8714

In [52]:
```
import pandas as pd
import matplotlib.pyplot as plt
greeks = pd.DataFrame([[ 0.481527, 0.0236961, -8.92209, 26.7698 ],
[ 0.505136, 0.0234856, -9.12631, 27.0785 ],
[ 0.528498, 0.0231972, -9.3065, 27.293 ],
[ 0.55153, 0.0228255, -9.46224, 27.4137 ],
[ 0.574149, 0.0223798, -9.59345, 27.4423 ],
[ 0.59629, 0.0218798, -9.70037, 27.3818 ],
[ 0.617909, 0.0213364, -9.78352, 27.235 ],
[ 0.638963, 0.0207493, -9.84358, 27.0058 ],
[ 0.659407, 0.0201216, -9.88138, 26.6987 ],
[ 0.679206, 0.0194597, -9.89793, 26.3188 ],
[ 0.698327, 0.0187698, -9.89437, 25.8714 ]
])
greeks.columns = ['delta','gamma','theta','vega']
greeks['S0'] = np.arange(95,106)
fig, axs = plt.subplots(2, 2, figsize=(10, 8))
axs[0, 0].plot(greeks['S0'],greeks['delta'])
```

```
axs[0, 0].set_title('delta')

axs[0, 1].plot(greeks['S0'],greeks['gamma'])
axs[0, 1].set_title('gamma')

axs[1, 0].plot(greeks['S0'],greeks['theta'])
axs[1, 0].set_title('theta')

axs[1, 1].plot(greeks['S0'],greeks['vega'])
axs[1, 1].set_title('vega')
```

Out[52]: Text(0.5, 1.0, 'vega')



Question 6

Consider the following 2-factor model for stock prices with stochastic volatility:dSt = $rSt\ dt + \sqrt{Vt}\ St\ dWt$^1, $dVt = \alpha(\beta - Vt)dt + \sigma\sqrt{Vt}\ dWt$^2 where the Brownian Motion processes above are correlated: $dWt^{1}dWt$^2 = ρdt, where the correlation ρ is a constant in [−1,1]. Estimate the price of a European Call option (via Monte Carlo simulation) that has a strike price of X and matures in T years. Use the following default parameters of the model: ρ = −0.6, r = 0.055, $S0 = 100$, X =100, $V0$ = 0.05, σ = 0.42, α = 5.8, β = 0.0625, dt = 0.05, N = 10,000. Use the Full Truncation, Partial

Truncation, and Reflection methods, and provide 3 price estimates by using the tree methods.

In [ ]:
```
void sto_vol(float rho, float r, float S0, float X, float V0, float sigma
{
    double S_ft, V_ft, c_ft[N_simulate], S_pt, V_pt, c_pt[N_simulate], S_
    for(int j = 0; j < N_simulate; j++)
    {
        # //initiate S equals S0, V equals V0
        S_ft = S_pt = S_r = S0;
        V_ft = V_pt = V_r = V0;
        for(int i = 1; i < N; i++)
        {
            z1 = dis(gen);
            z2 = rho * z1 + sqrt(1-pow(rho,2.0))*dis(gen);
            # // full truncation
            S_ft += r*S_ft*dt + sqrt(max(V_ft,0))*S_ft*sqrt(dt)*z1;
            V_ft += alpha*(beta-max(V_ft,0))*dt + sigma * sqrt(max(V_ft,0
            # // partial truncation
            S_pt += r*S_pt*dt + sqrt(max(V_pt,0))*S_pt*sqrt(dt)*z1;
            V_pt += alpha*(beta-V_pt)*dt + sigma * sqrt(max(V_pt,0)*dt)*z
            # // reflection methods
            S_r += r*S_r*dt + sqrt(abs(V_r))*S_r*sqrt(dt)*z1;
            V_r = abs(V_r) + alpha*(beta-abs(V_r))*dt + sigma * sqrt(abs(
        }
        # // save call option price max(S-X,0)
        c_ft[j] = max(S_ft-X,0);
        c_pt[j] = max(S_pt-X,0);
        c_r[j] = max(S_r-X,0);
    }
    cout << "Full Truncation: " << exp(-r*dt*N)*arr_mean(c_ft, N_simulate
    cout << "Partial Truncation: " << exp(-r*dt*N)*arr_mean(c_pt, N_simul
    cout << "Reflection method: " << exp(-r*dt*N)*arr_mean(c_r, N_simulat
}
```

OUTPUT_6

- T = N * dt = 500
- Full Truncation: 3.16829
- Partial Truncation: 3.18658
- Reflection method: 3.07766

Question 7

The objective of this exercise is to compare a sample of Pseudo-Random numbers with a sample of Quasi-Monte Carlo numbers of Uniform[0,1]x[0,1]: Use 2-dimensional Halton sequences to estimate the following integral:

$$I = \int\int e^{-xy}(sin6\pi x + cos^{\frac{1}{3}}2\pi y)dxdy$$

Default parameter values: N=10,000; (2,3) for bases.

In [ ]:
```
class HaltonSequence {
public:
    HaltonSequence(int base) : base(base) {}
    double next()
    {
        double f = 1, r = 0;
        int i = num;
        while (i > 0)
        {
            f = f / base;
            r = r + f * (i % base);
            i = i / base;
        }
        num++;
        return r;
    }
private:
    int base;
    int num = 0;
};

double func(double x, double y)
{
    if(cos(2 * M_PI * y) > 0)
        return exp(-x*y)*(sin(6 * M_PI * x) + pow(cos(2 * M_PI * y),1.0/3
    else
        return exp(-x*y)*(sin(6 * M_PI * x) - pow(-cos(2 * M_PI * y),1.0/
}

double integral_7()
{
    HaltonSequence x(2), y(3);
    double ans = 0;
    x.next();
    y.next();
    for(int i = 0; i < N_simulate; i++)
    {
        ans += func(x.next(),y.next());
    }
    return ans/N_simulate;
}
```

OUTPUT_7

- 0.026262

In [ ]: