simulations

January 10, 2022

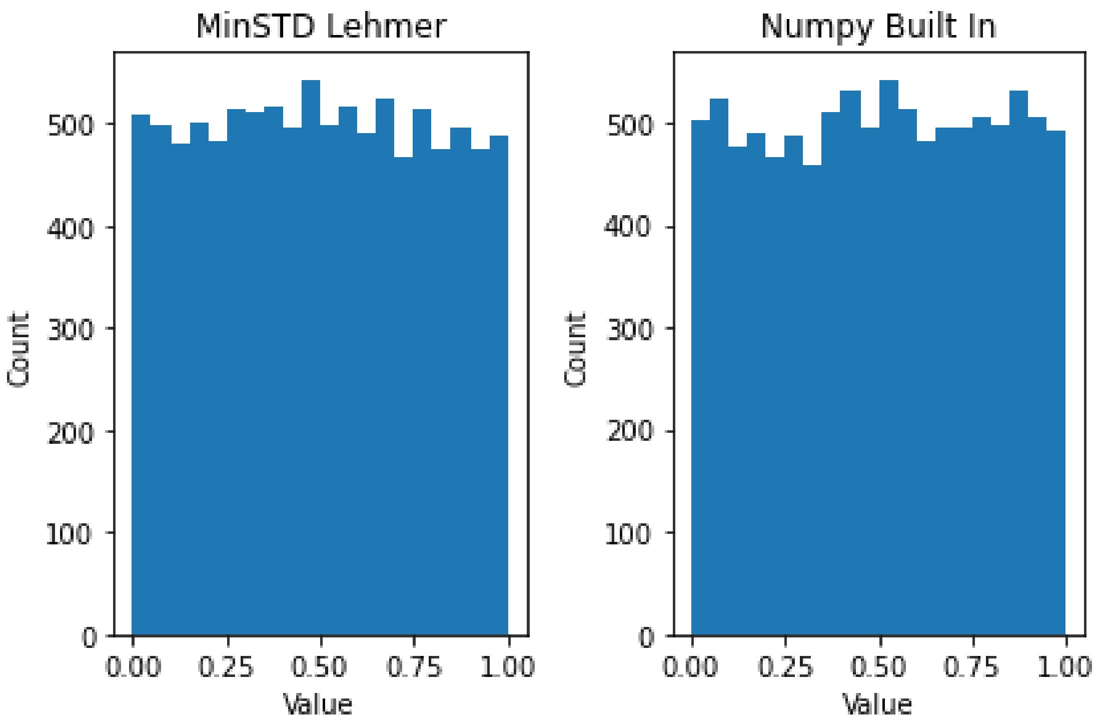
0.1 Question 1

Here, we want to generate 10000 uniform [0,1] points with two pseudorandom number generators. In this problem, the iterative sequence above is a linear congruential generator, which generates a sequence of numbers using a discontinuous piecewise linear function. This can be seen by the idea that the mod m creates cycles, where eventually numbers will be repeated.

We'll be using is the MinSTD Lehmer Random Number Generator, which has the parameters a = 7^5, m = $2^{31} - 1$. Typically, m is a prime number or a power of a prime number, a is a primitive root modulo m, which means that for every integer g that is coprime to m (the only positive divisor between them is 1), ∃ k such that a^k = g (mod m). Park and Miller suggested to use m = $2^{31} - 1$, which is a Mersenne prime or one less than a power of 2, and a = 7^5, which is a primitive root modulo m.

Numpy's built-in peusdorandom number generator is PCG, which is a variant of linear congruential generator but with larger modolus and state, and using power-of-2 modolus for better performance. We use numpy's built-in peusdorandom number generator to generate the histogram on the right, and we use a seed of 19 for the MinSTD Lehmer method, since 19 is coprime to $2^{31} - 1$. Since the Lehmer method produces numbers between 0 and $2^{31} - 1$, I divided every value by $2^{31} - 1$ to produce numbers between 0 and 1.

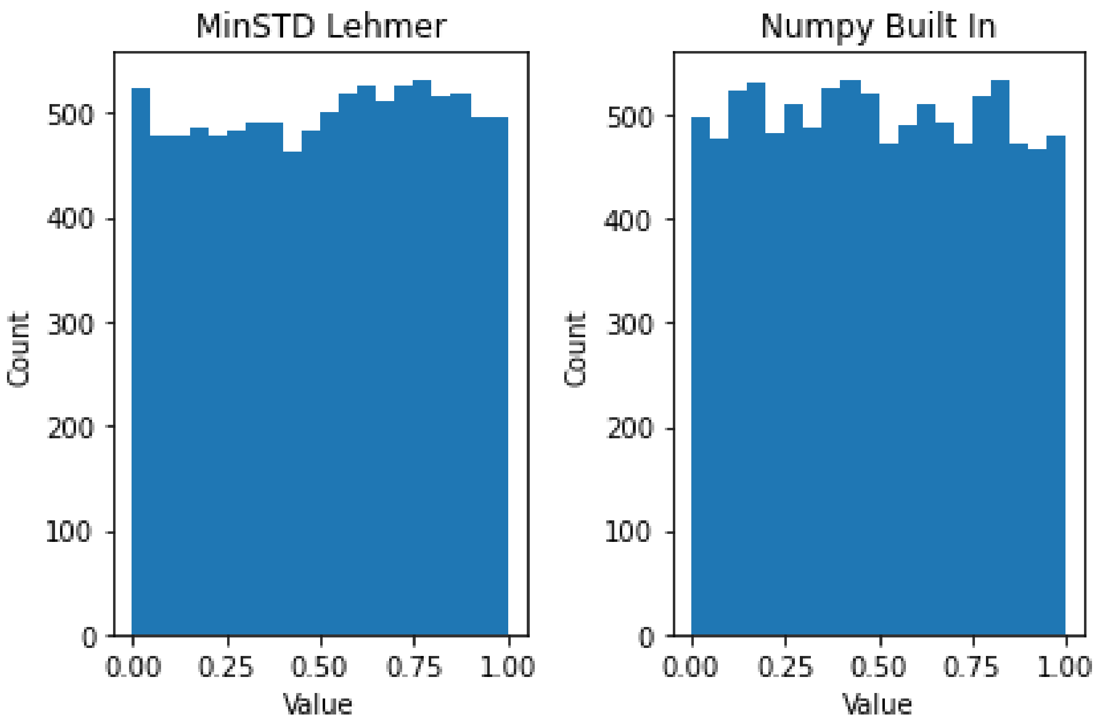
```
[47]: question1(19)
```



As we can see, the two histograms are very close to uniform, which shows that even though the number generators are peusdorandom and not truely random, they perform well enough that the results are close to random. Another point to show is that even though we take a large sample of 10000 points, since we picked a seed that is coprime to the modulo number, the MinSTD Lehmer method is guaranteed to have a full cycle, which means it will produce all the numbers between 0 and $2^{31} - 1$ before repeating. This means that the spread of our numbers will be good with no repeats, resulting in a close to uniform distribution. Similarly, numpy's built-in method also has large cycles, so with 10000 points we would not have repeats, leading to a close to uniform distribution.

If we pick a seed that is not coprime (for example 18), we will still see close to uniform, but there is no guarantee of a full cycle, which tends to lead to more repeats of numbers and more values in certain bins.

[48]:
```
question1(18)
```

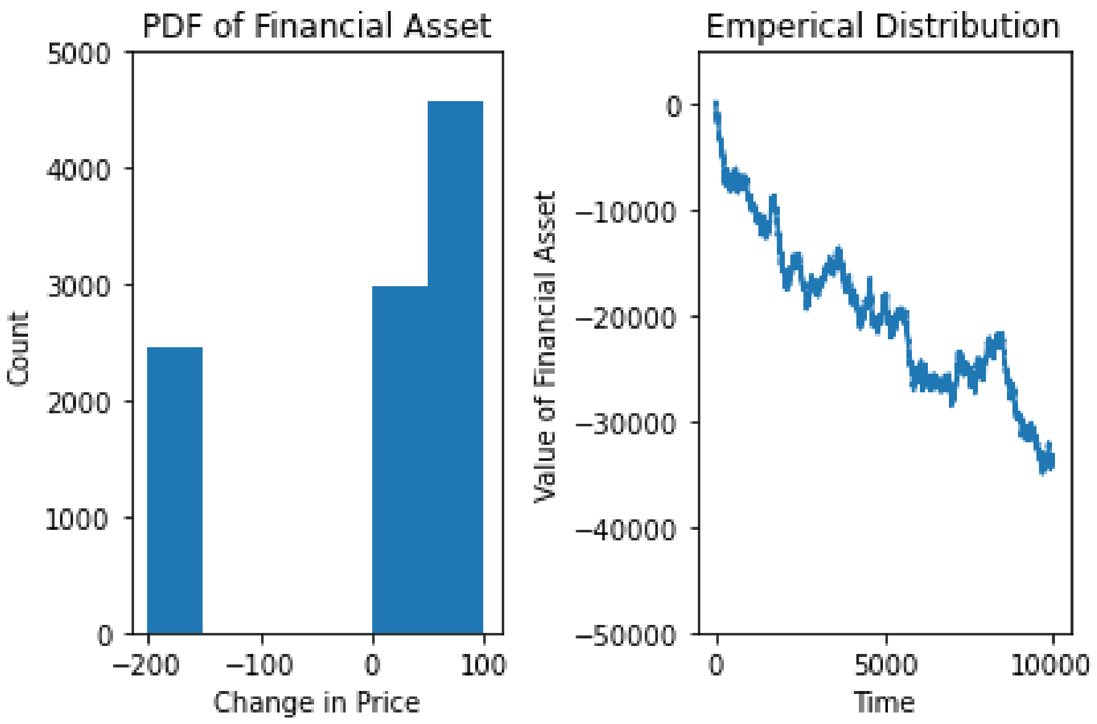


0.2 Question 2

We want to generate 10000 uniform [0,1] values with the MinSTD Lehmer Peusdorandom Number Generator discussed in Question 1 and then imitate the fluctuation of price on a financial asset by setting probabilities to each uniform random value. We create another array that converts each value of the uniform values that we produce into a change in price. Since 30% of the time the price stays the same, we set 0 <= x < 0.3 to 0. 45% of the time the price goes up by 100, so we set values 0.3 <= x < 0.75 to 100. 25% of the time the price goes down by 200, so we set values 0.75 <= x <= 1 to -200. We then graph the Probability Distribution Function or PDF on the left, which shows how often each individual value occurred, as well as the Emperical Distribution or CDF on the left, which is the cumulation of these values as time passes.

```
[49]: question2(19)
```

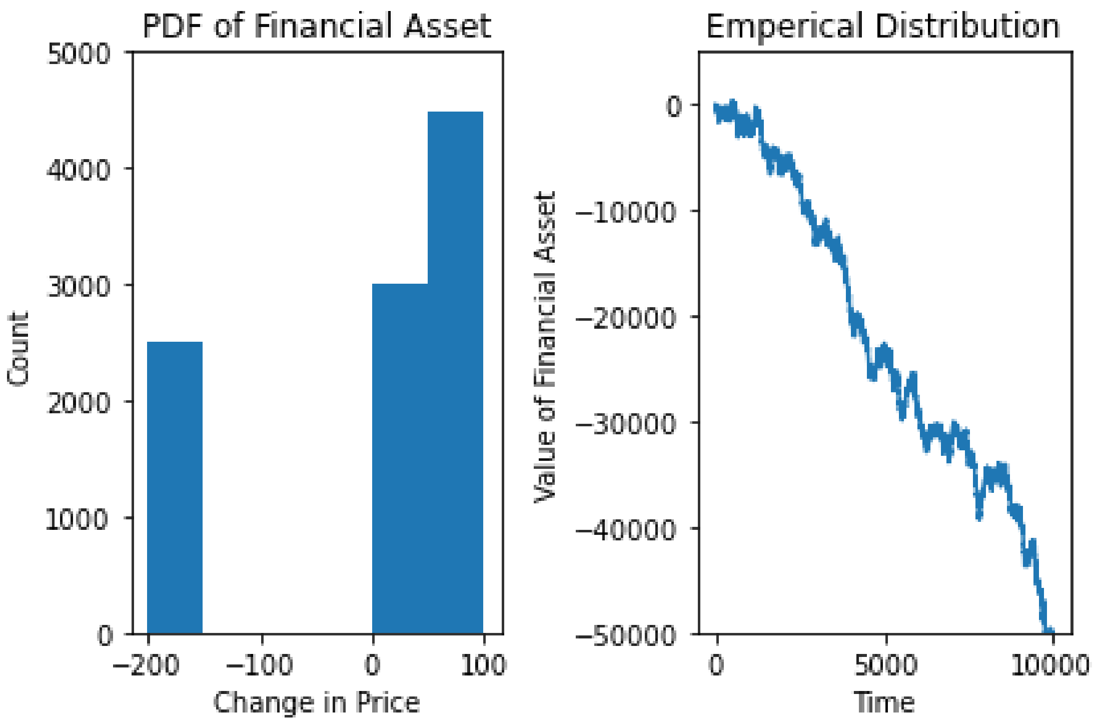
```
(array([2450.,    0.,    0.,    0., 2983., 4566.]), array([-200, -150, -100,
-50,    0,   50,  100]), <BarContainer object of 6 artists>)
```



As we can see from the left, -200 occurs ~24.5% of the time, 0 occurs ~29.8% of the time, and 100 occurs ~45.6% of the time, showing that our pesudorandom number generator from Question 1 is close to uniform. We also can calculate what the expected value of the cdf is after 10000 points by finding the expected value of one point, which is 0.3*(0) + 0.45*(100) + 0.25*(-200) = -5. Thus, with 10000 points, on average we would have around -50000. In this case we got a bit more +100 and a bit less -200, so our emperical distribution's end value is only ~ -33,000. If we pick another coprime as the seed, we can view a different but similar result.

[50]:
```
question2(21)
```

```
(array([2501.,    0.,    0.,    0., 3003., 4495.]), array([-200, -150, -100,
-50,    0,   50,  100]), <BarContainer object of 6 artists>)
```



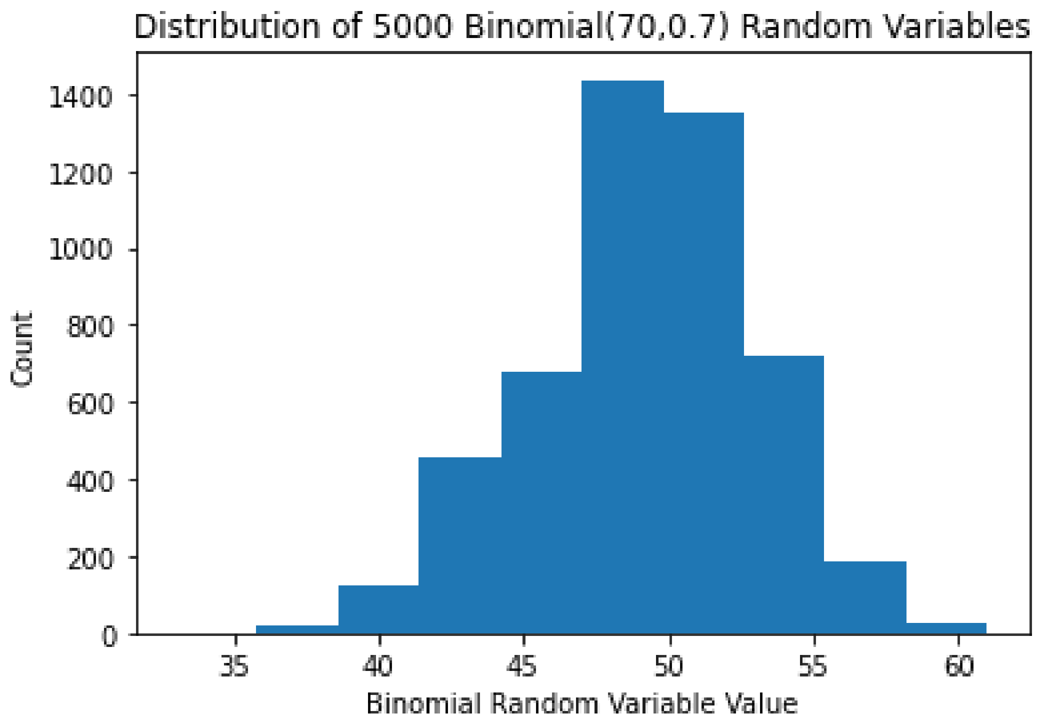
This time we got very close to the distribution we wanted of 25%, 30%, and 45%, so we got the expected value of -50000 on the right graph.

0.3 Question 3

We want to generate 5000 binomial random variables of 70 trials and probability 0.7 by using bernoulli variables. Bernoulli random variables are random variables that simulate a single trial and output 0 or 1 based on the probability. In this case, we'll be using bernoulli variables that have a 30% chance of being 0, and 70% chance of being 1, similar to flipping a weighted coin. Since a binomial random variable is assuming that we are doing a bernoulli trial multiple times, we simulate binomial random variables with n = 70 by doing 70 bernoulli trials and adding their outputs up to get one binomial random variable. We then do this 5000 times to get 5000 binomial random variables.

```
[51]: question3()
```

```
0.5434
```



Here we see the distribution of binomial random variables that we got from doing 70 bernoulli trials with p = 0.7 for each binomial random variable. We also printed the proportion of values that are less than 50, which in this case is 55.04% of the binomial random variables are less than 50. We can calculate the theoretical value, which comes from the formula

$p_x = \binom{n}{x} p^x q^{n-x}$

Where p = 0.7, q = 0.3. Since we want x < 50, we want to add P(0) + P(1) + ... + P(49). We calculate this with the following function to get the following result:

[52]:
```
binomial(50)
```

```
0.5449796328698386
```

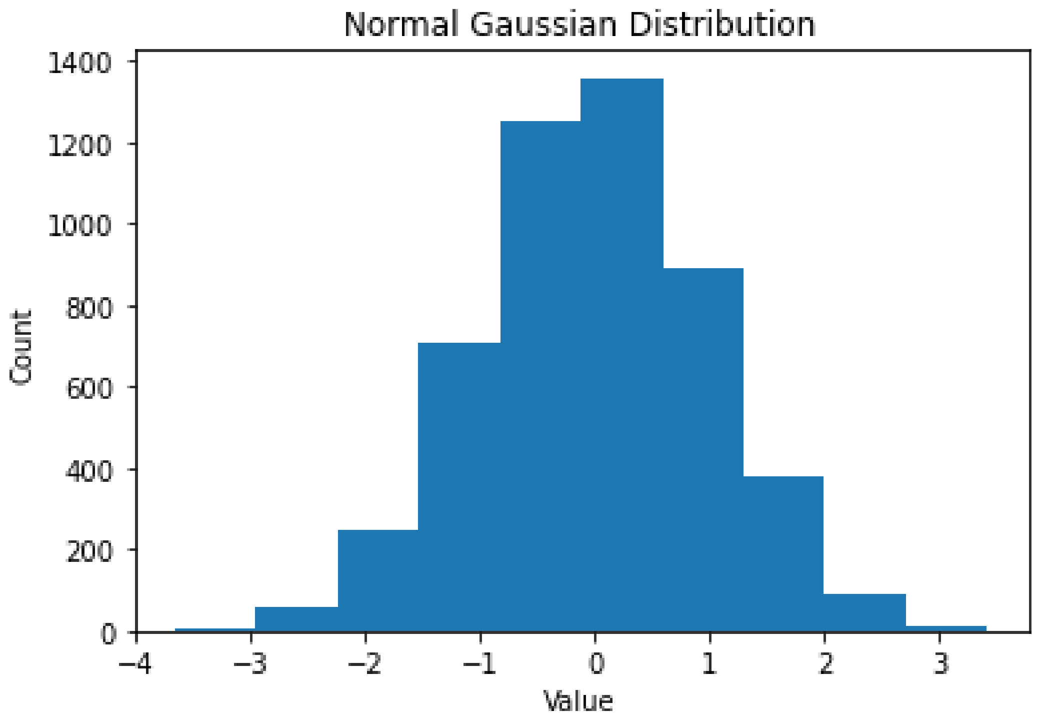
So the theoretical probability of a binomial random value is less than 50 is 54.49%. We got 55.04% for our data, which is close, showing that our estimation is close to the expected probability that the binomial random value is less than 50.

0.4 Question 4

We generate 5000 normal-Gaussian points with python's numpy's random number generator. Python uses PCG as their peusdorandom number generator to generate pesudorandom points, in this case in a normal distribution.

[53]:
```
question4()
```

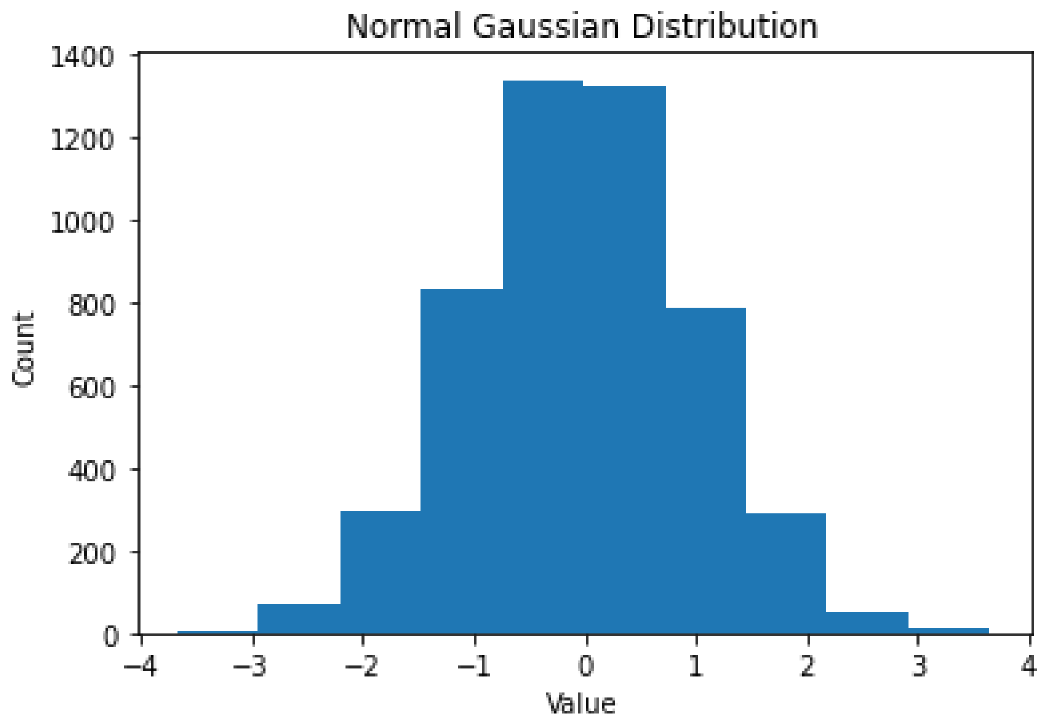
```
[ 0.28894395 -0.8029799   1.00697306 … -0.3107029  -1.10546748
 -0.13737452]
5000
```



As we can see, we have created 5000 points from the size of the array, and when we plot them into histogram bins, we can see that it creates a bell shape, which is standard for the normal distribution. This bell shape occurs because the mean, median, and mode of the data is in the center and the data is close to symmetrical around this central area, creating a bell shape where a lot more points occur in the center compared to the sides. Running it again will show similar results with a different seed.

```
[54]: question4()
```

```
[-0.3347656   0.07441714  1.78925411 …  0.6246024   0.64563704
 -1.21525432]
5000
```

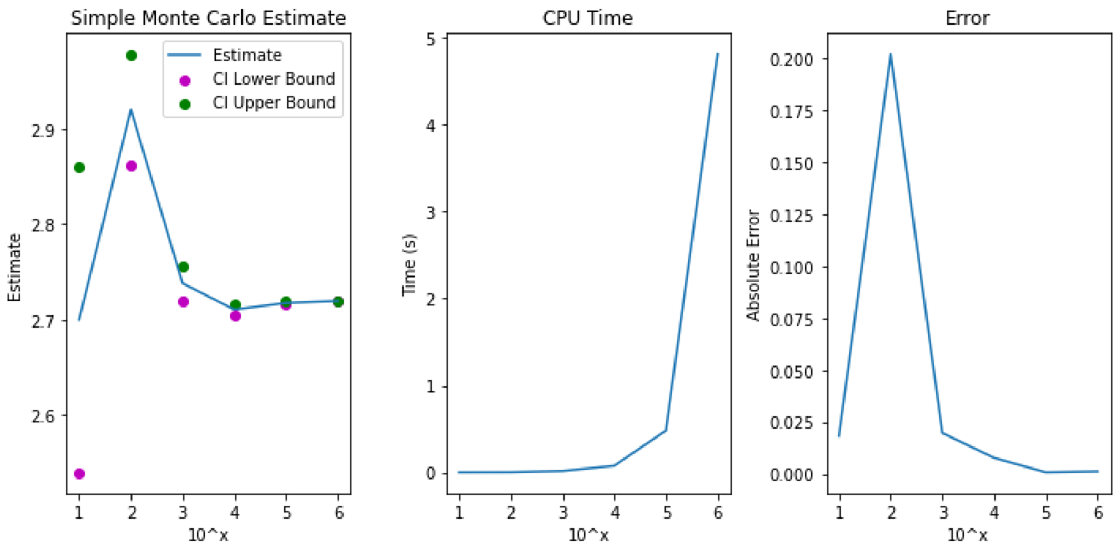


0.5 Question 5

Here we want to calculate the minimum number of random uniform (0,1) variables such that the sum of these variables exceed 1. We will simulate this through Monte Carlo by doing N trials and getting the average, showing the estimate for N = 10, 100, ..., 1000000

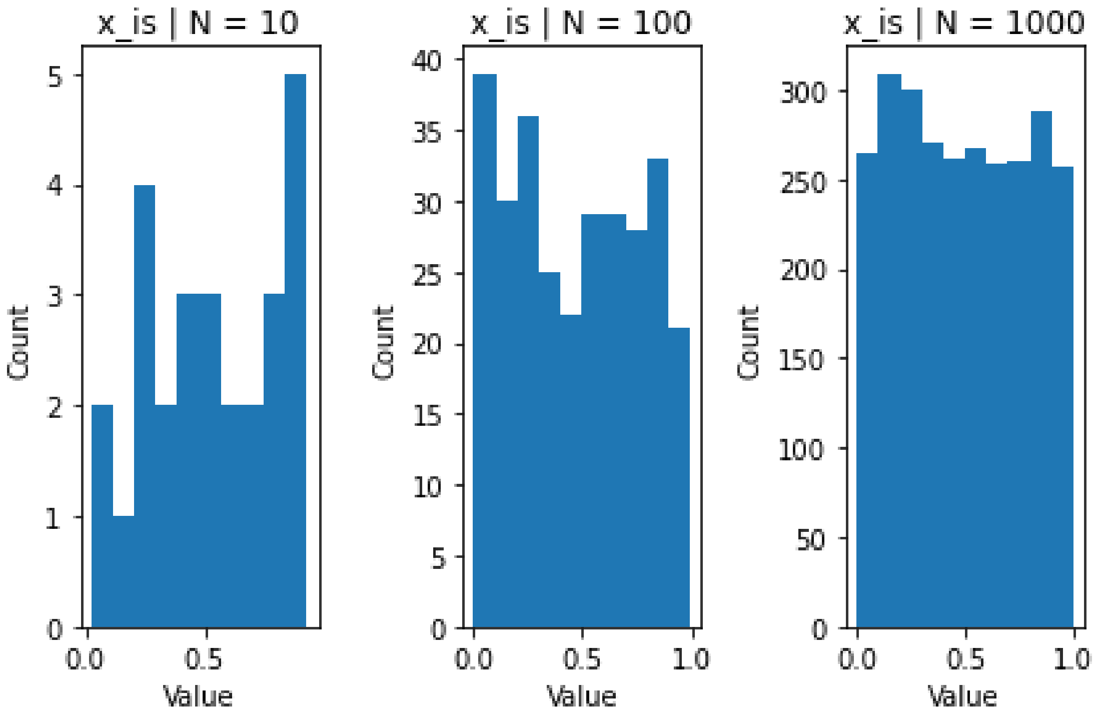
[55]:
```
data, trials, table = question_5()
print(table)
```

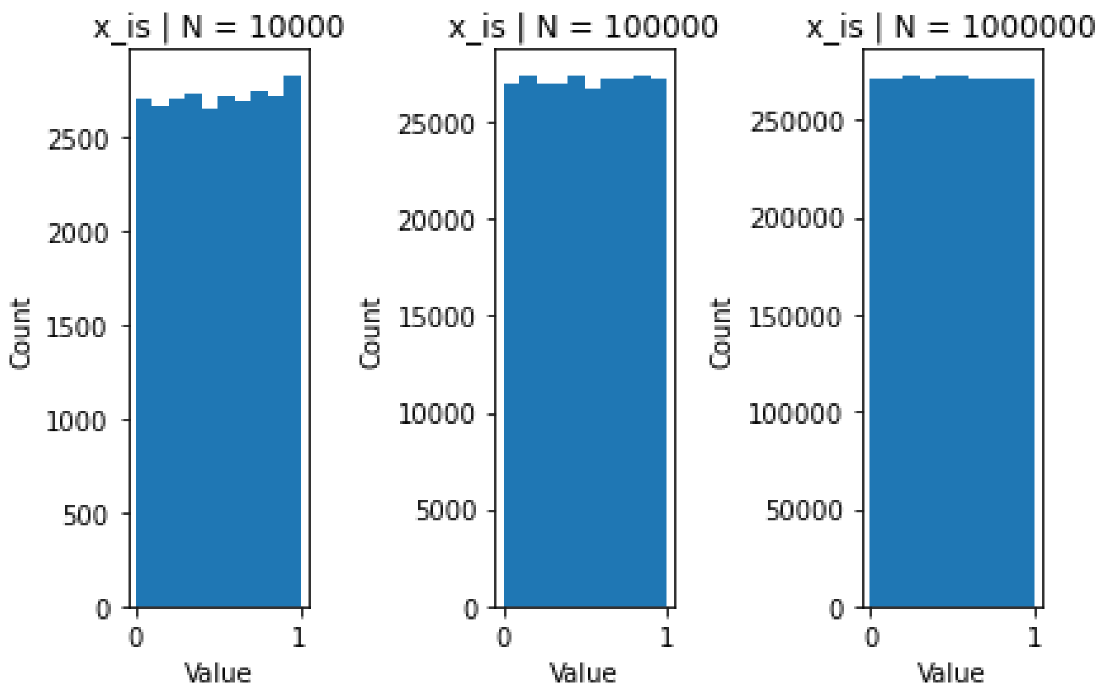
```
         simple_monte_carlo_estimate  cpu_time     error  confidence_intervals
N
10                          2.700000  0.000205  0.018282          (2.54, 2.86)
100                         2.920000  0.001524  0.201718        (2.862, 2.978)
1000                        2.738000  0.015155  0.019718         (2.72, 2.756)
10000                       2.710500  0.075450  0.007782        (2.705, 2.716)
100000                      2.717460  0.480089  0.000822        (2.716, 2.719)
1000000                     2.719525  4.812555  0.001243         (2.719, 2.72)
```



As we can see from the table and the graphs, as N increases in size, the confidence_interval gets smaller in width and the estimate converges to a value close to e, which is approximately ~2.718. This means it takes approximately e amount of uniform distributed random variables for its sum to exceed 1. As N increases, the error gets smaller as the estimate gets closer to the e. However, the cpu time gets much larger as N gets larger, going from 0.47 seconds at 10^5 to 4.76 seconds at 10^6

```
[56]: printData(data, trials)
```



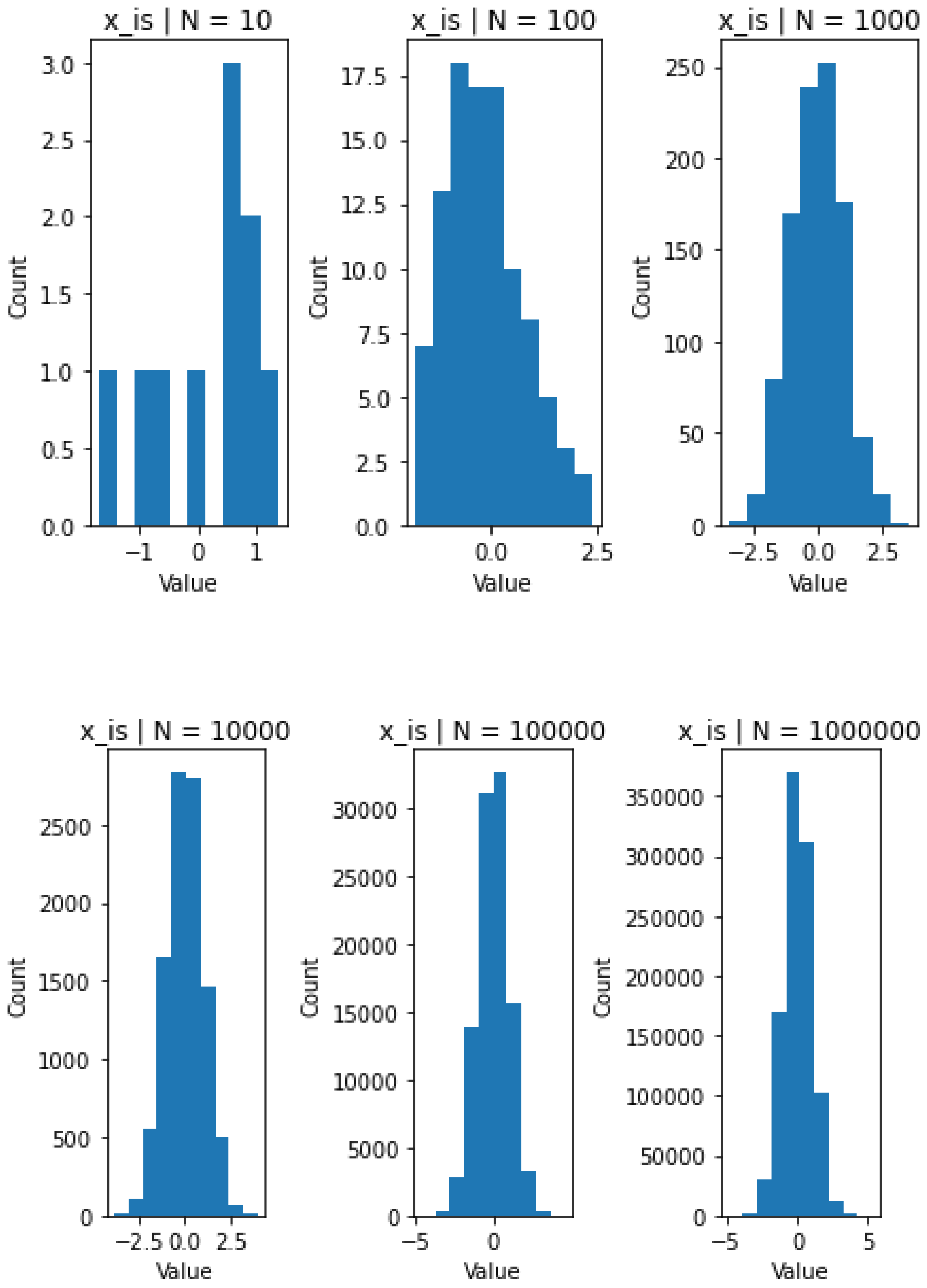


The histograms above show our generated data that we used. The generated data consists of x_is where X is uniform(0, 1). We can notice that as N increases, especially when large, the histograms reflect data that is much more uniformly distributed. Note that N is the number of trials of calculating the minimum number of uniform random variables to exceed the sum of 1, not the number of random variables produced.

0.6 Question 6

Here we want to compute P(V > 5), where V is a standard Gaussian random variable, or a variable sampled from N(0,1). We will first be doing a simple Monte Carlo technique where we sample V N times, where N = 10, 100, ..., 1000000. We then count the number of times this value is larger than 5, and divide by the total number of trials to get the probability.

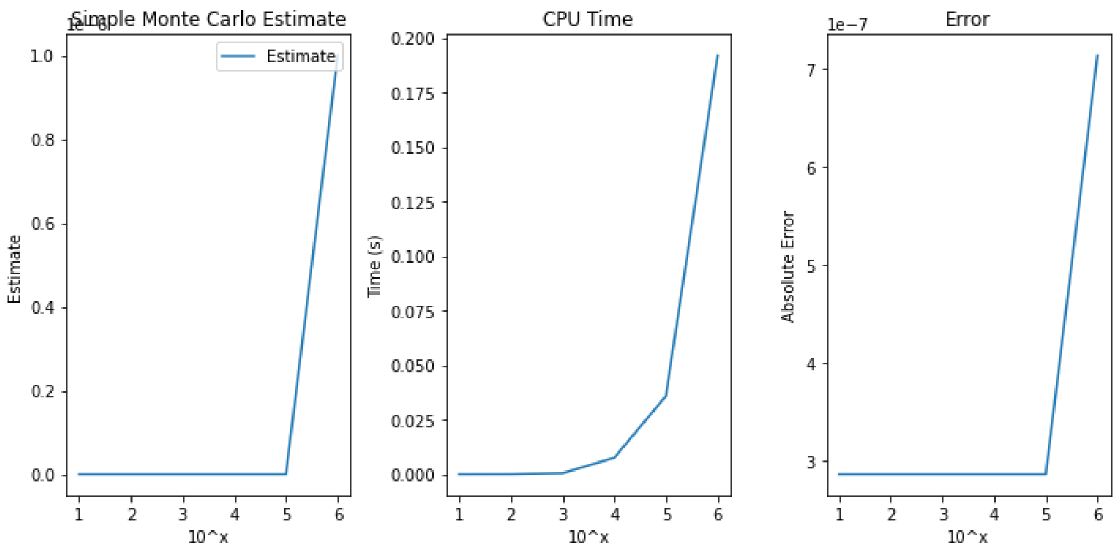
[57]:
```
data = generate_and_plot_question_six_a_data()
```



The histograms above show our generated data that we used. The generated data consists of x_is where X is N(0, 1). We can notice that as N increases, especially when large, the histograms reflect data that is much more normally distributed.

[58]:
```
question_six_a(data)
```

[58]:
```
         simple_monte_carlo_estimate  cpu_time         error
N
10                          0.000000  0.000030  2.866516e-07
100                         0.000000  0.000057  2.866516e-07
1000                        0.000000  0.000510  2.866516e-07
10000                       0.000000  0.007490  2.866516e-07
100000                      0.000000  0.035814  2.866516e-07
1000000                     0.000001  0.192042  7.133484e-07
```

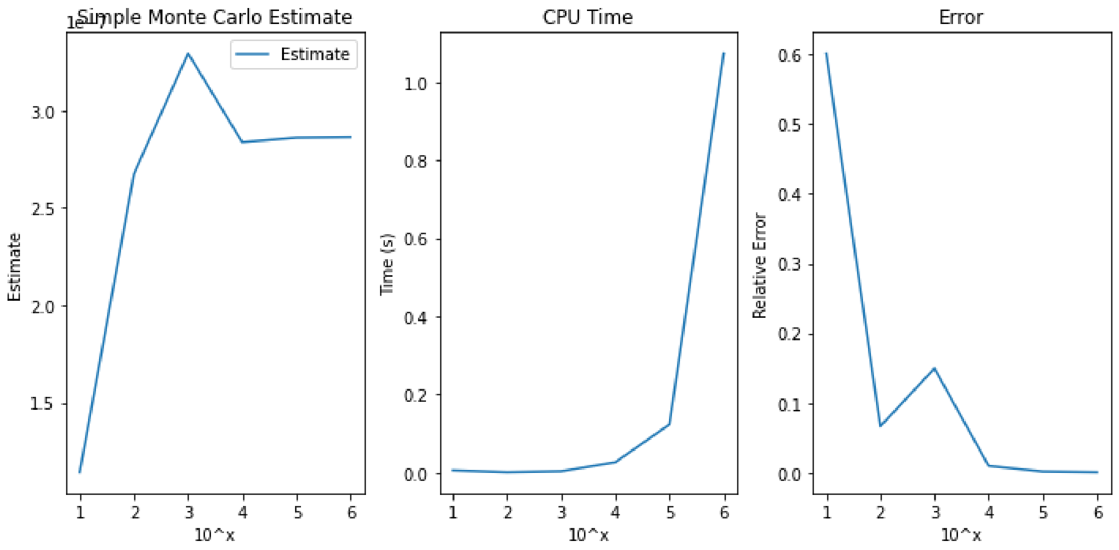


Monte Carlo Sampling Methods are effective when we are sampling often in the region of importance. However, since P(V $>$ 5) is near the tail of the normal distribution, this means that there is a very small region that is important when it comes to sampling. This causes the simple Monte Carlo Sampling Method to not get any samples of data, so even though the actual value of P(V>5) is around $2.87e^{-7}$, the actual estimates that the Monte Carlo Sampling Method give are 0 since it never samples in the important region.

For Importance Sampling Method, we pick a new function q(x) that looks like the importance region that we will be sampling. Since we are looking at the tail of a Normal Gaussian distribution, an exponential function that is shifted will capture the importance region of this sampling. We use q(x) $= e^(- x + 5)$ to have our exponential distribution be centered around the area that we want to observe, which is P(V $>$ 5). As N gets larger, it looks more like an exponential distribution.

[60]:
```
question_six_b(data)
```

```
[60]:          simple_monte_carlo_estimate  cpu_time     error
      N
      10                      1.142684e-07  0.005632  0.601368
      100                     2.675786e-07  0.000956  0.066537
      1000                    3.295048e-07  0.003457  0.149496
      10000                   2.838442e-07  0.026300  0.009794
      100000                  2.861435e-07  0.123488  0.001772
      1000000                 2.864303e-07  1.074037  0.000772
```



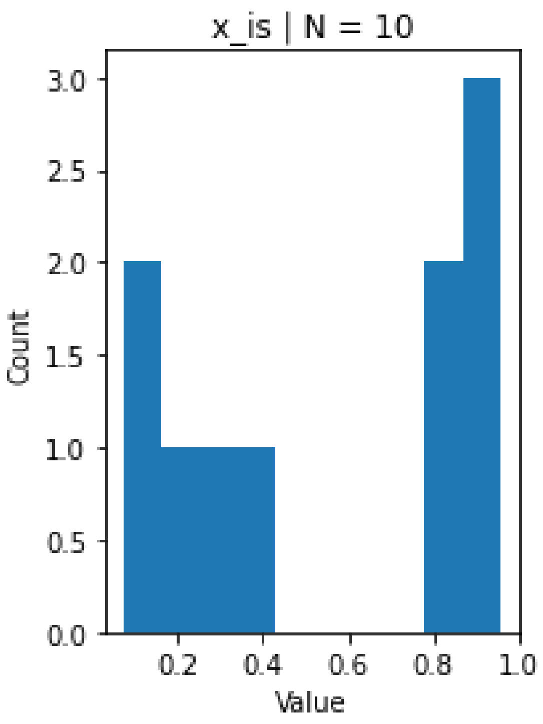
Importance sampling goes as follows: we sample from our new q(x) that looks similar to the important region of our distribution. We then check if it is greater than 5, and then we add p(x)/q(x).

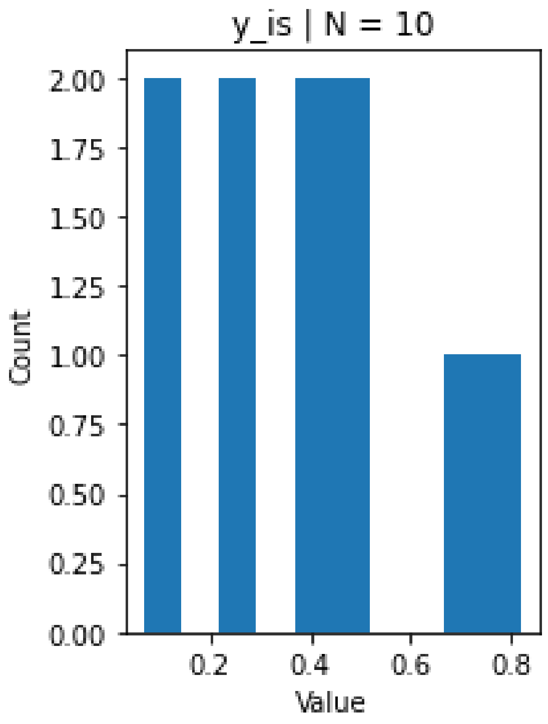
In our case, we can see that we were able to produce results that were non-zero in the Monte Carlo Estimates. We can now get estimates close to the true value, with error dropping and the process converging to ~$2.87e^{-7}$

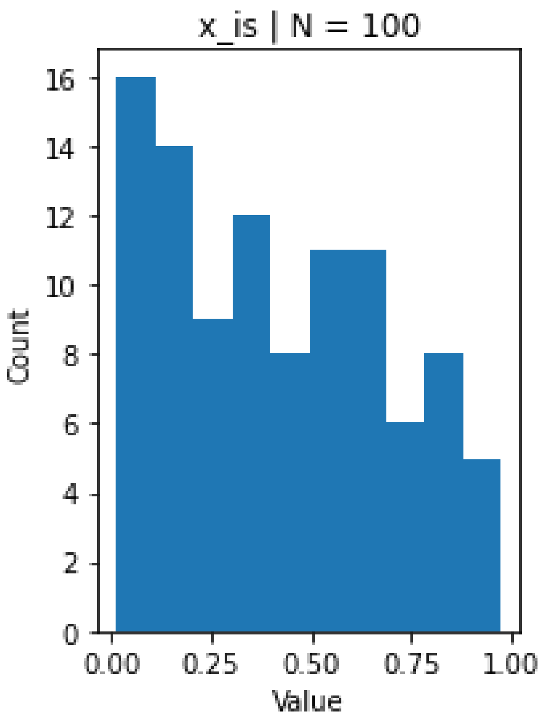
0.7 Question 7

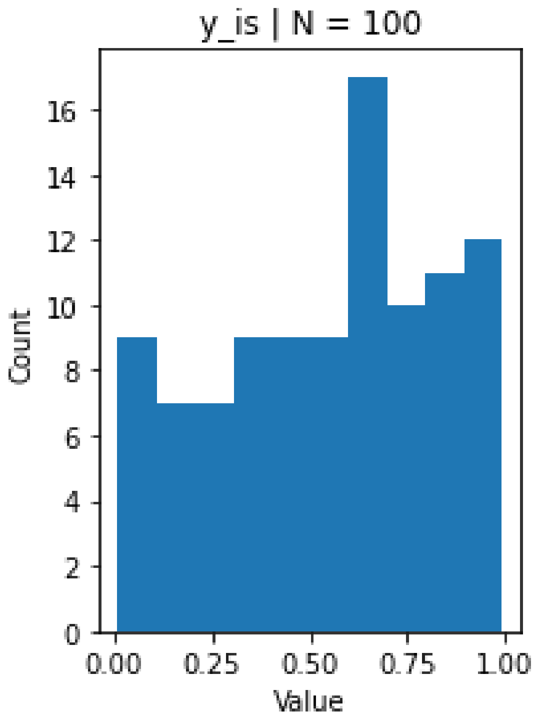
Here, we compute an estimate of the integral I using three different approaches. These methods are helpful in estimating the values of integrals that are difficult to compute analytically.

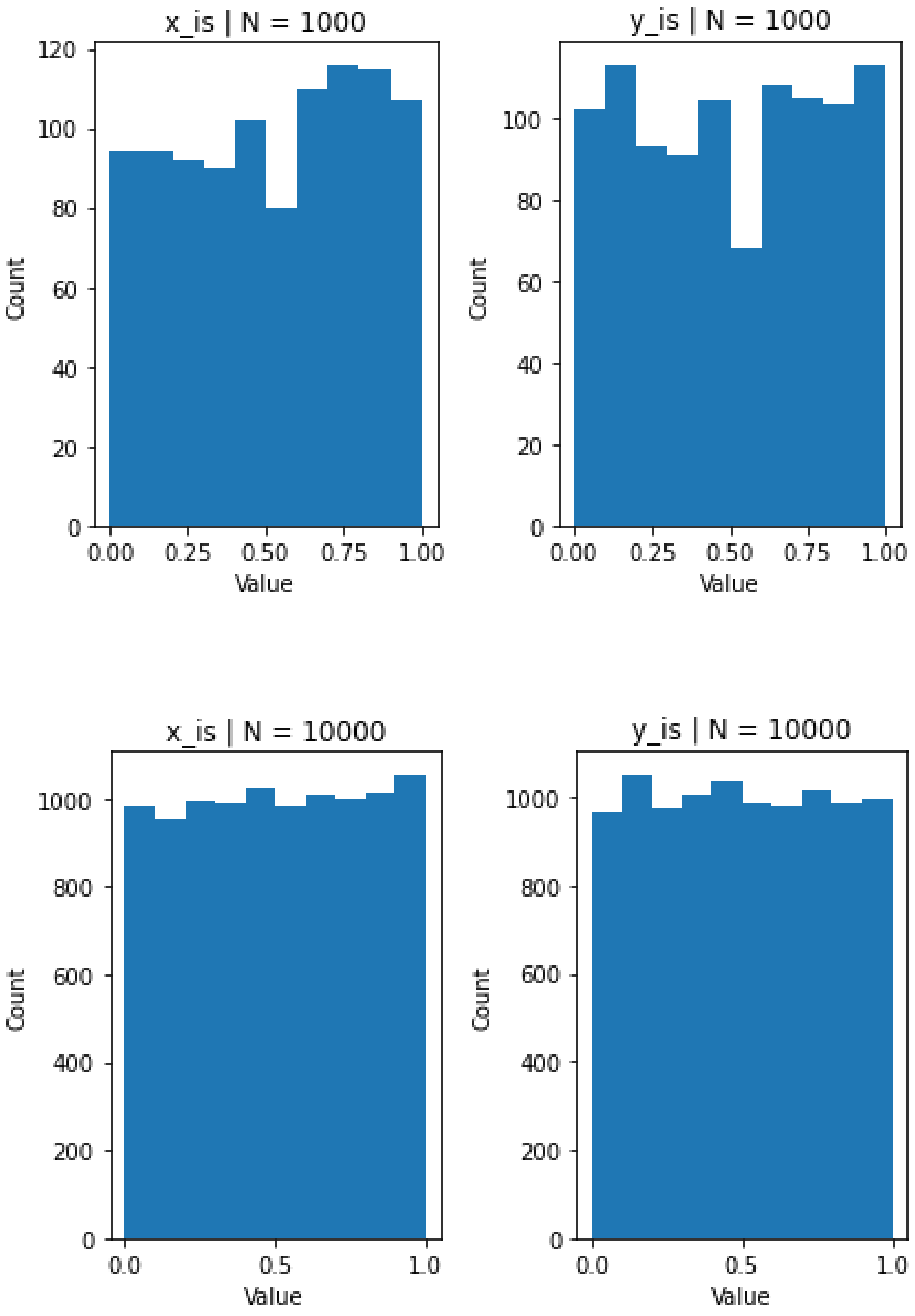
```
[65]: data = generate_and_plot_question_seven_data()
```


x_is | N = 10
Count
Value

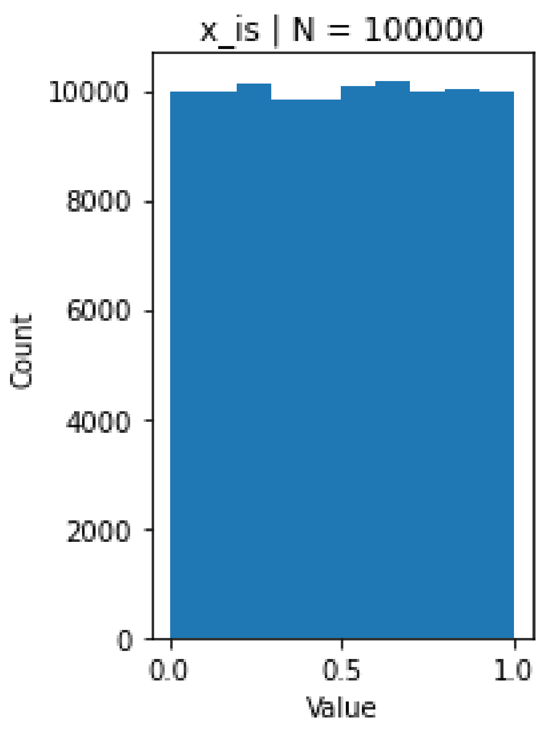

y_is | N = 10
Count
Value

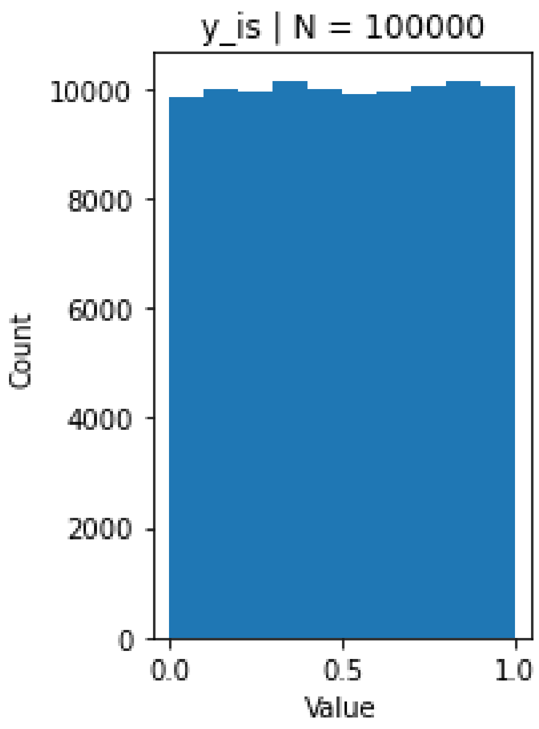

x_is | N = 100
Count
Value

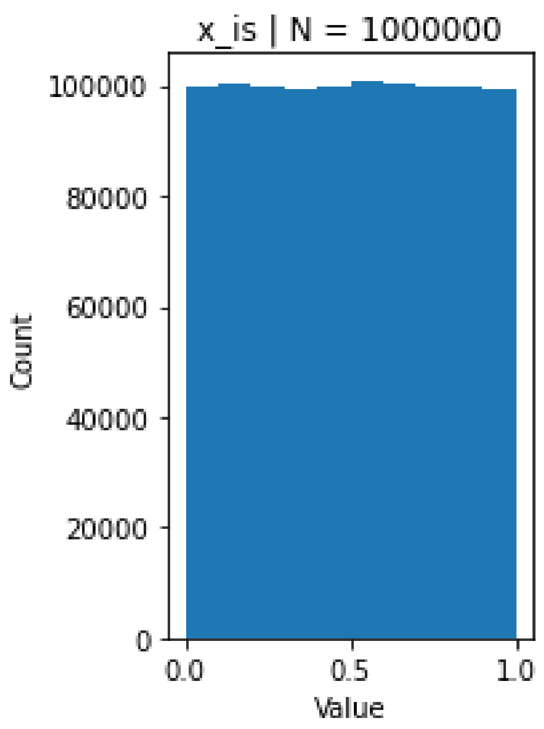

y_is | N = 100
Count
Value

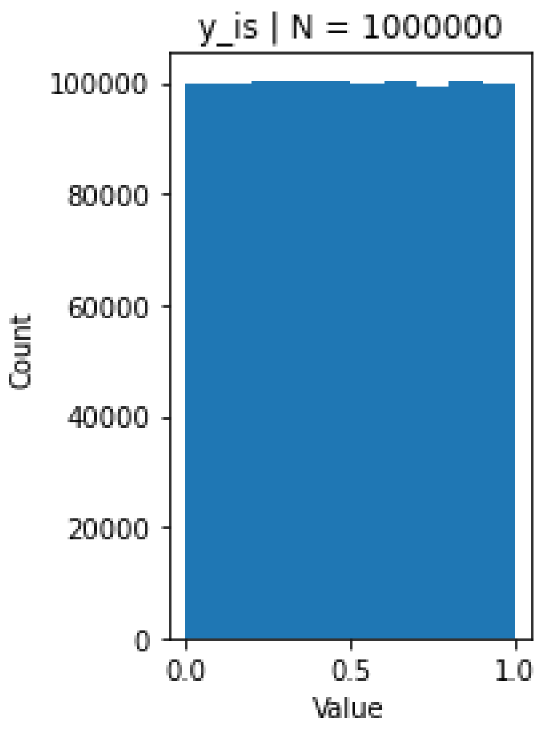


x_is | N = 1000
y_is | N = 1000
x_is | N = 10000
y_is | N = 10000
Count
Value
0
20
40
60
80
100
120
200
400
600
800
1000
0.00
0.25
0.50
0.75
1.00
0.0
0.5
1.0


x_is | N = 100000
Count
0
2000
4000
6000
8000
10000
0.0
0.5
1.0
Value


y_is | N = 100000
Count
0
2000
4000
6000
8000
10000
0.0
0.5
1.0
Value


x_is | N = 1000000
Count
0
20000
40000
60000
80000
100000
0.0
0.5
1.0
Value


y_is | N = 1000000
Count
0
20000
40000
60000
80000
100000
0.0
0.5
1.0
Value

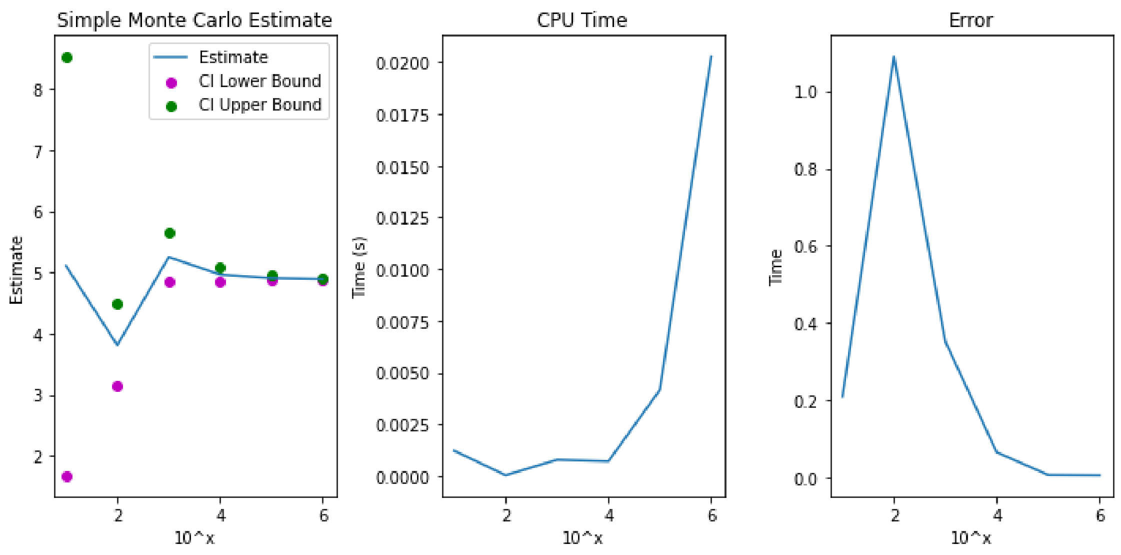
The histograms above show our generated data that we will use in parts a, b, and c. The generated data consists of x_is and y_is where X and Y are uniform(0, 1). We can notice that as N increases, especially when large, the histograms reflect data that is much more uniformly distributed. We will use this data to compute expected values that will give us estimates of I.

```
[66]: true_intergral_estimate = 4.89916
```

We were unable to find an analytical solution to the integral of a way to calculate the error of monte carlo methods. So to approximate our error, we will use the above value computed by Wolfram Alpha. ### a)

```
[67]: question_seven_part_a(data)
```

```
[67]:          simple_monte_carlo_estimate  cpu_time     error confidence_intervals
      N
      10                          5.109409  0.001218  0.210249       (1.681, 8.538)
      100                         3.809418  0.000041  1.089742       (3.132, 4.487)
      1000                        5.251592  0.000784  0.352432       (4.856, 5.648)
      10000                       4.965257  0.000712  0.066097       (4.847, 5.083)
      100000                      4.907095  0.004157  0.007935        (4.87, 4.944)
      1000000                     4.892012  0.020277  0.007148        (4.88, 4.904)
```

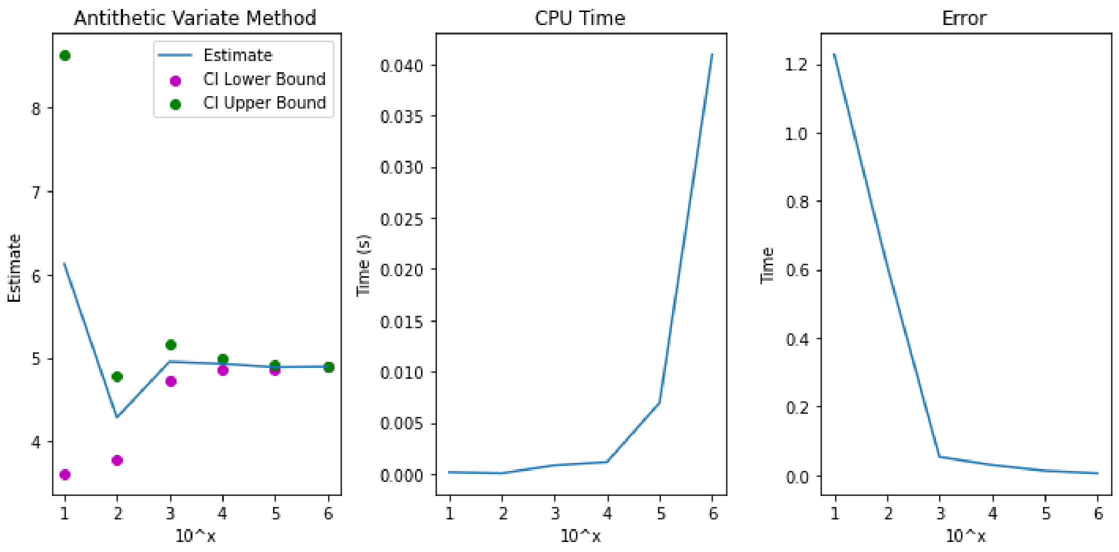


0.7.1 b)

```
[68]: question_seven_part_b(data)
```

```
[68]:          antithetic_estimates  cpu_time     error confidence_intervals
      N
      10                   6.126451  0.000173  1.227291       (3.616, 8.637)
```

```
100                   4.283695  0.000072  0.615465        (3.78, 4.787)
1000                  4.952124  0.000838  0.052964        (4.735, 5.17)
10000                 4.928564  0.001161  0.029404       (4.861, 4.996)
100000                4.886478  0.006949  0.012682       (4.866, 4.907)
1000000               4.893607  0.040945  0.005553         (4.887, 4.9)
```



0.7.2 c)

U and V are uniform(0, 1)

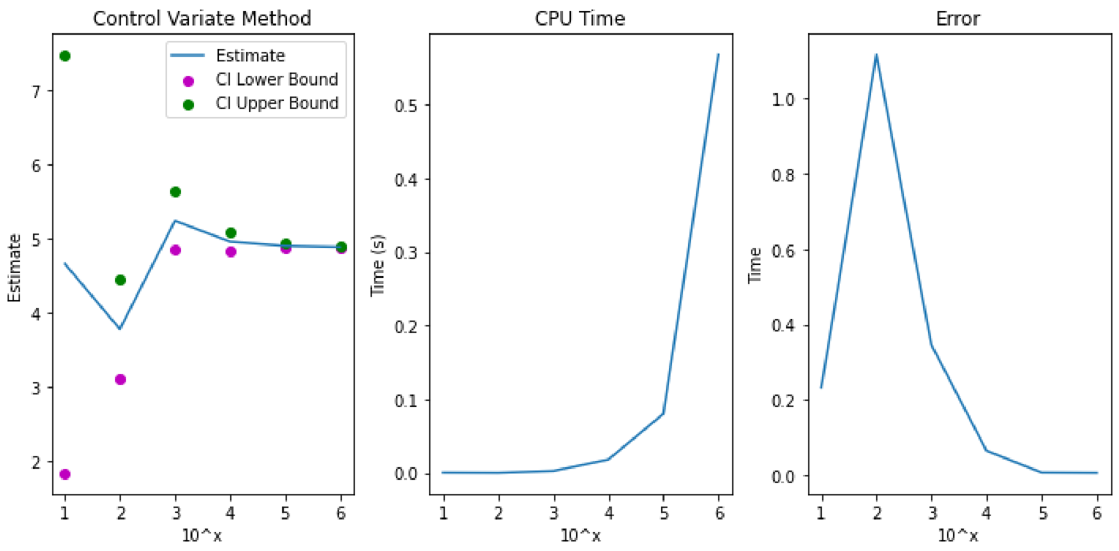
- E[U] = (1 + 0) / 2 = 0.5

- E[V] = (1 + 0) / 2 = 0.5
- E[Y1] = E[U] + E[V] = 1
- Var[U] = $(1-0)^2/12$ = 1/12
- E[Y2] = E[$(U+V)^2$] = E[$U^2+2UV+V^2$] = E[U^2] + E[2UV] + E[V^2]
- E[U^2] = Var[U] + E[U]^2 = 1/12 + 0.25 = 1/3
- E[2UV] = 2E[U]E[V] = 0.5
- E[Y2] = 4/3

[110]:
```
question_seven_part_c(data)
```

[110]:
```
          control_variate_estimates  cpu_time     error confidence_intervals
N
10                         4.665667  0.000572  0.233493       (1.847, 7.484)
100                        3.784566  0.000302  1.114594       (3.121, 4.448)
1000                       5.244984  0.002794  0.345824         (4.85, 5.64)
10000                      4.964681  0.018222  0.065521       (4.847, 5.083)
100000                     4.907039  0.080277  0.007879        (4.87, 4.944)
```

```
1000000                    4.892006  0.567643  0.007154        (4.88, 4.904)
```



0.7.3 d)

When comparing the variance reduction tecniques, the antithetic method worked the best. We can observe that the error is the smallest and the confidence intervals are the tightest for the antithetic method by a slight amount. This is unexpected because the control variates method is better for variance reduction. This likely happened due to randomnes and if we were to continue increasing N, we would likely see the control variates method work the best. We can also note the our error values don't even different by a factor of 10 when N = 10ˆ6. However, both methods are better than the simple monte carlo method as expected, since the variance is reduced.

The CPU time follows the same trend for all methods, with a large spike when by a factor of 10 when N = 10ˆ6. The simple monte carlo is the fastest, followed by the antithetic approach, and then the control variates approach. This is expected since we are doing more computations in the anthetic than the simple simulation, and the control variates than the antithetic.

0.8 Question 8

Here, we want to compute the price of a European call option use a Monte Carlo simulation and the Black-Scholes formula, and compare the results of both

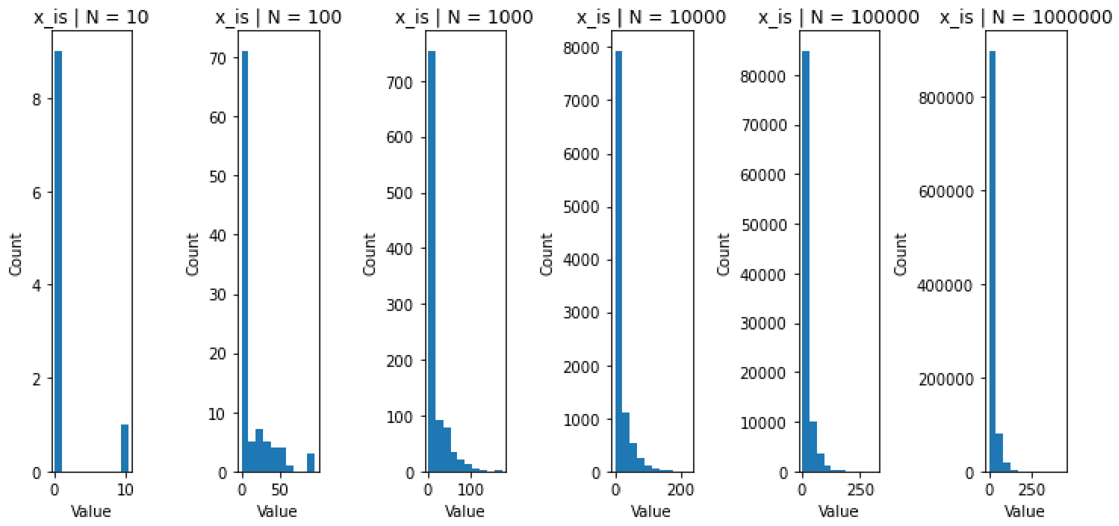
[70]:
```
r = 0.04
sigma = 0.25
s_0 = 90
T = 2
K = 100
```

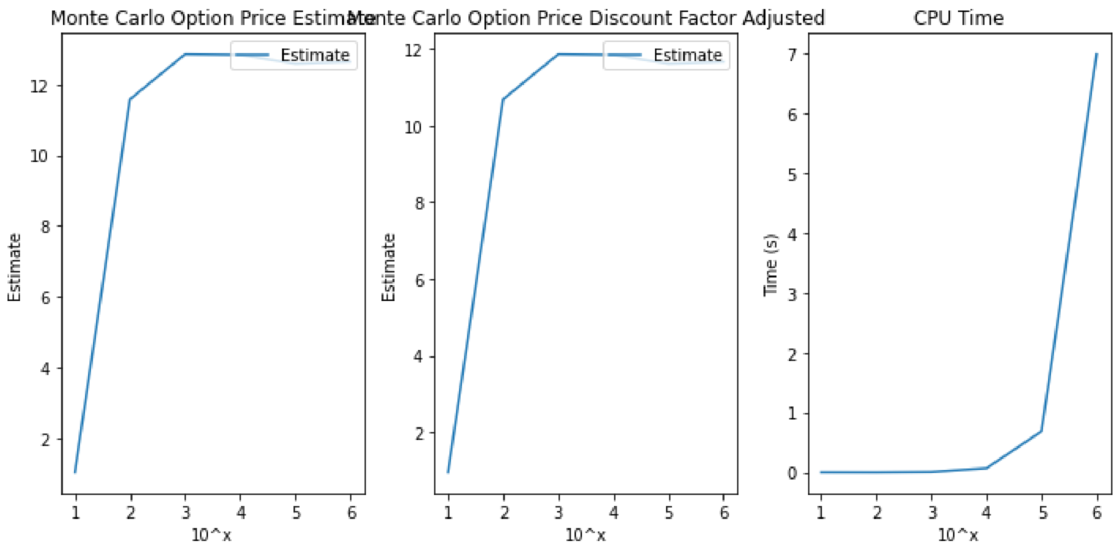
0.8.1 a)

In part a, we want to esimate the price of the European call option by Monte Carlo simulation. To do this, we first need to compute the price of the stock at time T using the Geometric Brownian Motion process. This process has the stock prices follow a series of steps, where each step is a drift plus or minus a random shock. The Weiner process incorporates an element of randomness through the form of a standard normal random variable with variance T. Once we compute the stock price using the Geometric Brownian motion process, we then subract the strike price to compute the price of the option. If the strike price is greater than the computed stock price, we set the price of the option to 0. Additionally, we include an adjusted estimate where we multiply this value by a discount factor to account for the time value of money to see if it makes our estimate more accurate.

[71]:
```
question_eight_part_a(r, sigma, s_0, T, K)
```

[71]:
```
         option_price_estimates  option_price_estimates_adjusted  cpu_time
N
10                     1.030967                         0.951702  0.001494
100                   11.577324                        10.687217  0.000718
1000                  12.850293                        11.862315  0.006865
10000                 12.832645                        11.846025  0.068496
100000                12.580935                        11.613667  0.687282
1000000               12.634084                        11.662730  6.991537
```





The results above indicates that as N increases, the Monte Carlo simulation is kind of converging to an option price of around \$12.65 and an adjusted price of around 11.68. Additionally, we see a dramatic spike in CPU time required when N = 10^6 at 7.84 seconds, taking almost 10 times longer than when N = 10^5. The histograms above indicate that many of the option prices are being evaluated to 0 due to the Geometric Brownian Motion model computing a stock price that is less than the strike price of the option.

0.8.2 b)

Part b asks us to compute the price of the option using the Black Scholes formula, which is a deterministic model under a set of assumptions.

```
[72]: bs_call(s_0, K, T, r, sigma)
```

```
[72]: 11.667369027772402
```

The computed Black Scholes options price is \$11.67 ### c)

Here, we will compare the results from parts a and b. Our monte carlo simulations in part a with large sample sizes resulted in values very close to what we compute using the Black Scholes formula, specifically when accounting for the discount factor. The discount factor is important because it accounts for time value of money. It would be very important to make note of this when using these models in practice because slight increases in accuracy can result in large increases in profit due to the fierce competition and tightening spreads in financial markets. It makes sense that the monte carlo simulation with a large enough sample size computes an options price that matches the Black Scholes formula because the formula uses Geometric Brownian motion to model stock prices under the assumption that underlying price follows the lognormal distribution and the log-returns of price is normally distributed.

0.9 Question 9

Here, we want to compute the price of a European call option for AMZN using 4 different methods. We assume that this option is expiry at the end of December, so T = 2. We are also using 0.25 for volatility, which we looked up.

[365]:
```
r = 0.02
sigma = 0.25
s_0 = 1765
T = 2
K = 1800
```

0.9.1 a)

The equation for put call parity is as follows:

C - P = U - PV where

- C = Call Option Price
- P = Put Option Price
- U = Spot Price of Underlying Asset
- PV = Present Value Strike Price = K / (1 + r)^T

[370]:
```
bs_call(s_0, K, T, r, sigma)
```

[370]:
```
263.3537716303654
```

[367]:
```
put_call_parity(263.35, s_0, K, r, T)
```

[367]:
```
228.45380622837365
```

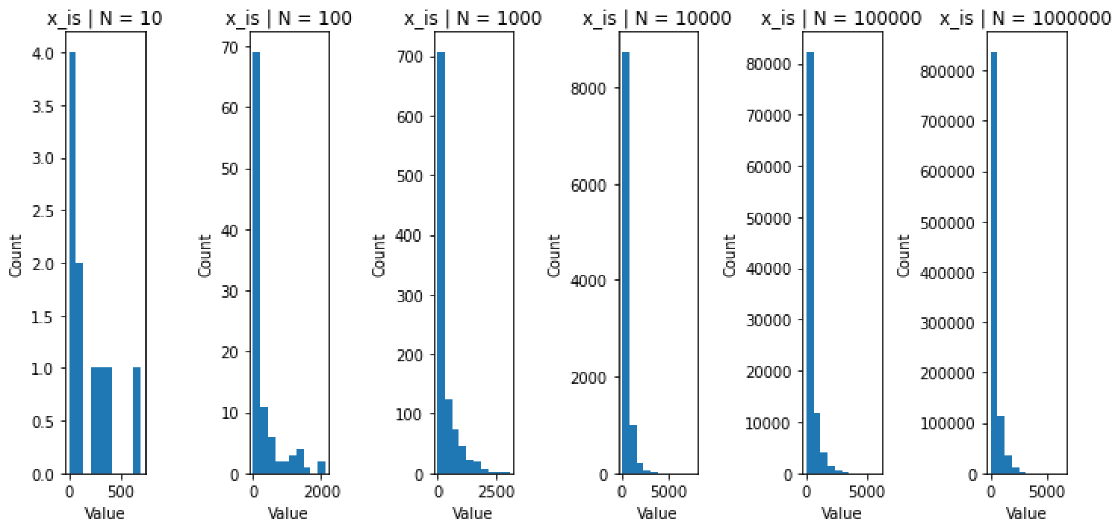
We can notice here that the price of the put and call options diverge, so an arbitrage oppurtunity exists.

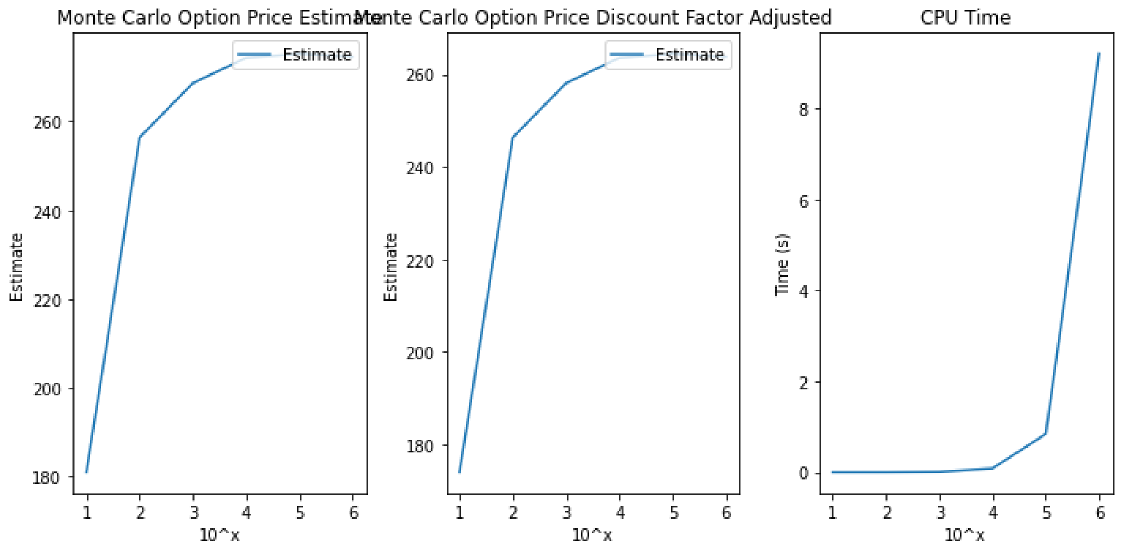
0.9.2 b)

Here, we can use the same function from 8a to estimate the price of the option by Monte Carlo with Geometric Brownian Motion

[368]:
```
question_eight_part_a(r, sigma, s_0, T, K)
```

[368]:
```
         option_price_estimates  option_price_estimates_adjusted  cpu_time
N
10                   181.025827                       173.927703  0.000171
100                  256.384420                       246.331443  0.001169
1000                 268.625446                       258.092492  0.008682
10000                274.330486                       263.573834  0.084806
100000               275.235622                       264.443479  0.842058
1000000              274.574729                       263.808500  9.202171
```





Here, we notice that the estimated options price starts to converge to around \$274.

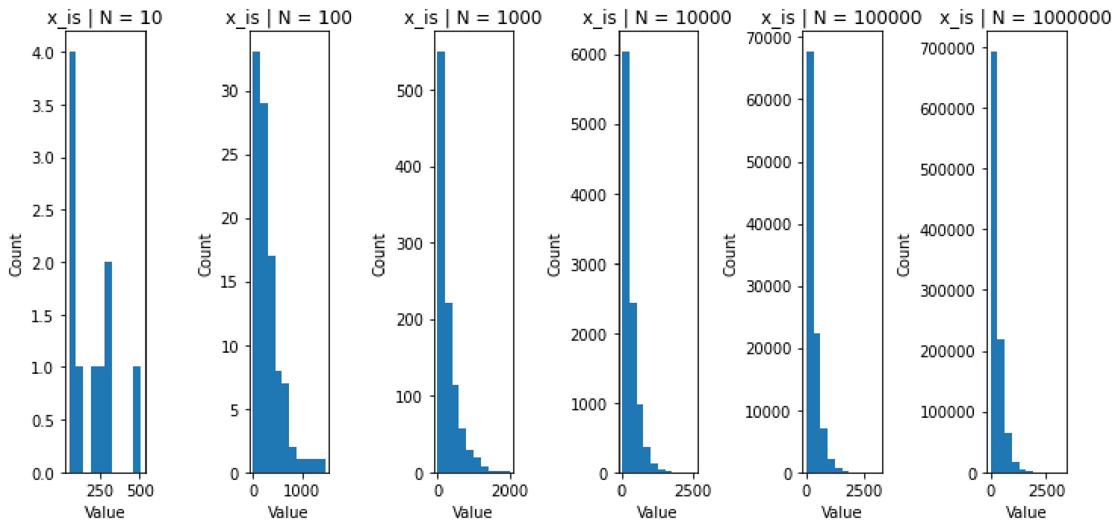
It converges to around \$263.80 when accounting for the discount factor, which is very close to the computed price when using black scholes.

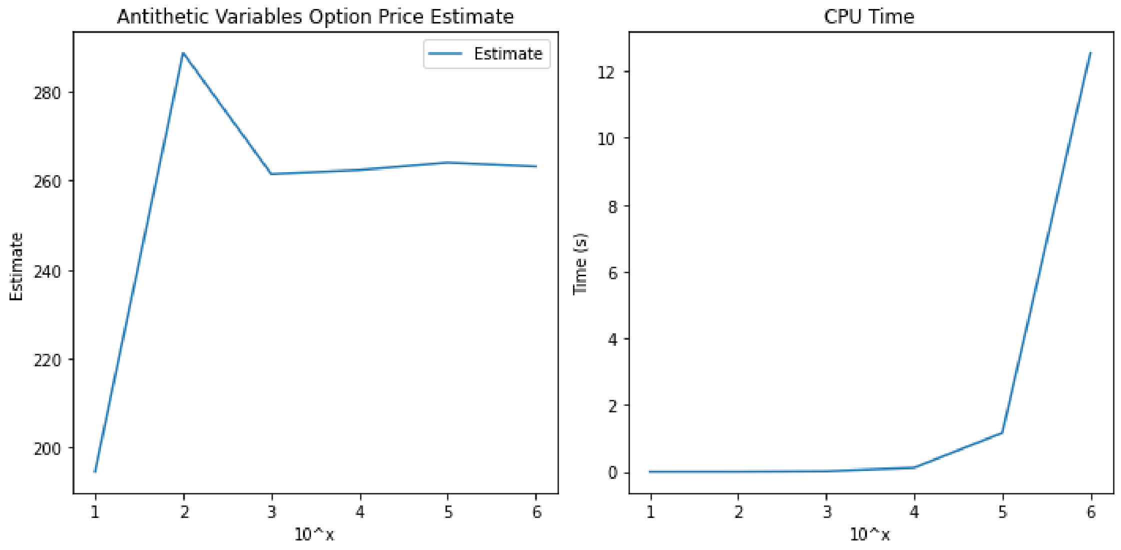
0.9.3 c)

Here we are using the antithetic variance reduction monte carlo method to get our estimate of the option's price.

[350]:
```
question_nine_part_c(r, sigma, s_0, T, K)
```

```
[350]:          option_price_estimates   cpu_time
       N
       10                   194.656142   0.000211
       100                  288.658997   0.001455
       1000                 261.464351   0.012852
       10000                262.375009   0.118926
       100000               264.052247   1.162572
       1000000              263.168708  12.551559
```





We converge to around $263.16 (accouting for the discount rate).

0.9.4 d)

Comparison: all 3 of the monte carlo methods converge to an estimate that is very close to what is computed from the black scholes formula for large values of N, specifically when we account for the discount factor. This is expected since the monte carlo methods are using the geometric brownian motion process, which is used in the black scholes computation. We can also make observations about the CPU time. the black scholes computation is much faster than the monte carlo simulations, which is expected since it is not sampling. The antithetic method takes slightly longer than the simple monte carlo method due to requiring more calculations. The control variates approach takes the longest by a full factor of 10 for multiple values of N. This is due to the fact that it calls Black Scholes for every single data point. The overall trend in CPU usage where N = 10^6 takes significanly longer than other values for N is still consistent across all methods.