Project 1: Stochastic Modeling, Monte Carlo methods, and Variance Reduction

Problem 2: Pareto Distribution

Given a Pareto-distributed random variable X, with PDF

$$p_X(x) = \begin{cases} \frac{\alpha}{x^{(\alpha+1)}}, if\ x \geq 1 \\ 0, \quad if\ x < 1 \end{cases}$$

Where α = 3/2.

To find the mean (expectation) of the given PDF, we must first find the Monte Carlo estimator and run the Monte Carlo algorithm N different times for n samples. Here, I have taken N = 10^4 trials and n = 10^4 total samples.

Running Monte Carlo Algorithm:

As mentioned in lecture notes (Lecture 5, Page 5-1), the Monte Carlo algorithm has three major steps:

1. Sample $x^{(i)}$ from X_i for $i = 1 \dots \dots n$
   To obtain samples, I used the rand () function on Python to obtain 10^4 random numbers from a uniform distribution $U(0,1)$.
   Here I am sampling $x^{(i)}$ from X_i random variable for $i = 1 \dots \dots 10^4$.
   To sample from a uniform distribution, we must convert the PDF to an inverse CDF. I used the first general purpose sampling technique from Lecture 6, page 6-2, the Inverse CDF Transform. For this technique, I first found the CDF by integrating the PDF and then found the inverse of that function to find the inverse CDF, as shown below:

$$P_X(x) = \int p_X(x)dx = \int_1^x \frac{\alpha}{x^{\alpha+1}}$$

$$P_X(x) = \alpha \int_1^x x^{-(\alpha+1)} = \alpha \left[\frac{x^{-(\alpha+1)+1}}{-(\alpha+1)+1}\right]_1^x = \left[-\frac{1}{x^\alpha}\right]_1^x = 1 - \frac{1}{x^\alpha}$$

   To find the inverse CDF,

$$y = 1 - \frac{1}{x^\alpha}$$

$$x = \left(\frac{1}{1-y}\right)^{1/\alpha} = (1-y)^{-\frac{1}{\alpha}}$$

$$x = (1-y)^{-\frac{2}{3}}$$

2. Evaluate $g(x^{(i)})$

   Here $g(x^{(i)}) = (1-x)^{-\frac{2}{3}}$ obtained from the inverse CDF.
   To evaluate $g(x^{(i)})$, I passed the random values obtained in step 1 into the function which generated 10^4 $g(x)$ values.

3. Evaluate $S_n(g) = \sum_{i=1}^{n} g(x^{(i)})$

   To evaluate $S_n(g)$, I summed $g(x^{(i)})$ values obtained in step 2 and divided the summation by total number of samples collected which was 10^4. This gives the value of the estimator for the given run.

Steps 1-3 were repeated for each sequence of $x^{(i)}$. In total I ran 10^4 trials of random sequences to construct a convergence plot as shown in figure 1.

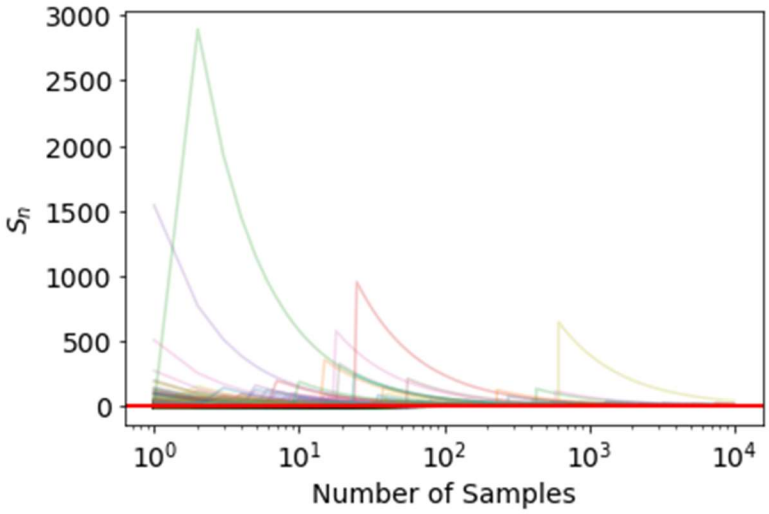


Figure 1: Convergence plot for Pareto distribution

In Figure 1, the x-axis describes the number of samples from 1 to 10^4 and y-axis describes the value of the Monte Carlo estimator. The different colored lines represent each trial of the Monte Carlo run on the obtained inverse CDF function. The thick red line represents the true mean of the function that was calculated analytically as shown below.

Analytical Calculation of True Mean:

Mean can be calculated by integrating the given PDF as shown:

$$\mu_x = \int x.p(x)dx = \int_{-\infty}^{1} x.0\ dx + \int_{1}^{\infty} x\frac{\alpha}{x^{(\alpha+1)}}dx$$

$$\mu_x = \int_{1}^{\infty} x\frac{\alpha}{x^{(\alpha+1)}}dx = \alpha\int_{1}^{\infty} x^{-\alpha}dx = \alpha\left[\frac{x^{-\alpha+1}}{-\alpha+1}\right]_1^{\infty} = \frac{-\alpha}{-\alpha+1} = \frac{-\frac{3}{2}}{-\frac{3}{2}+1} = 3$$

Therefore, true mean has the value 3, which is a finite value. As seen in Figure 1, it is evident that the mean calculated using the Monte Carlo estimator for all 10^4 trials converges to the true mean value, as the number of samples increases. This shows that the Monte Carlo algorithm works for any function that can be sampled uniformly.

Analytical Calculation of Variance:

$$Var(x) = E[(x-\mu_x)^2] = \int_{-\infty}^{\infty} (x-\mu_x)^2.\ p(x)\ dx = \int_{-\infty}^{\infty} (x-\mu_x)^2 . \frac{\alpha}{x^{(\alpha+1)}}\, dx$$

$$Var(x) = \int_{-\infty}^{\infty} (x-3)^2 . \frac{\frac{3}{2}}{x^{\left(\frac{3}{2}+1\right)}}\, dx$$

On evaluation,

$$Var(x) = \left[2x^{\frac{1}{2}} - 6^{-\frac{3}{2}} + 12x^{-\frac{1}{2}}\right]_{-\infty}^{\infty} = \infty$$

Here, we can see that the variance for the given PDF is infinite. The variance of the Monte Carlo estimator was also calculated in the code. The graph in figure 2 shows the variance of the given function and that of a known function $\frac{1}{n}$ for comparison, on a logarithmic scale.

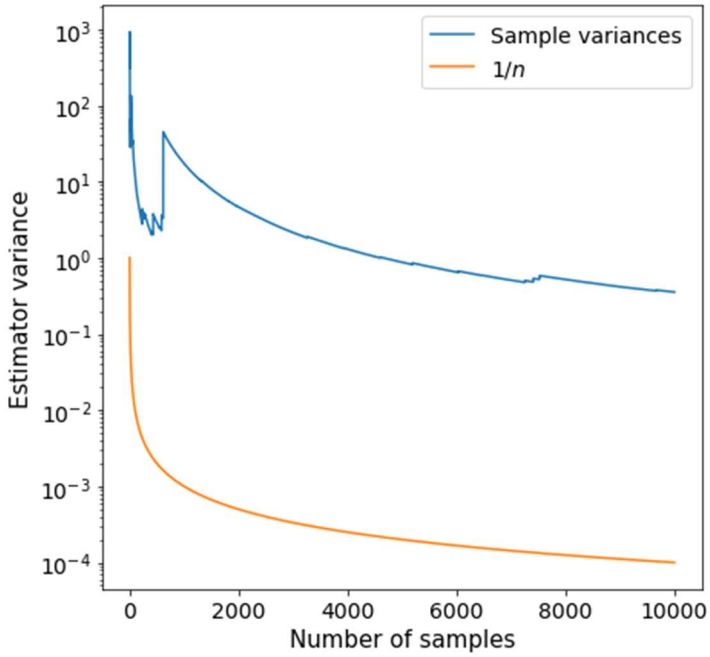


Figure 2: Comparison of Variance of given PDF to Variance of 1/n

It is evident from figure 2 that the variance of the given function in the problem does not converge to 0. The variance converges to a certain value greater than 0 and hence can be assumed to be infinite. The Monte Carlo algorithm is very useful in calculating the expectation of an event. However, it is very expensive and may not always be the most efficient method.

Problem 3: Importance Sampling for Random Walks

3.1. 1-D Bernoulli Random Walk

Here we have considered a standard random walk in one dimension. The number of steps is fixed at $N = 100$. Each step is of length 1 and the direction is randomly chosen. For a 1-D walk, only two directions are considered and in the given question, the probability of the event that that walker goes in either of the two direction is given to be equal.

$$P(X_j = 1) = P(X_j = -1) = 0.5$$

For each j, where $X_1, X_2, \ldots, X_N$ are identical and independent (i.i.d) random variables.

Question 2.1

a. To simulate an N-step 1-D bernoulli random walk, I used the generate_random_walk() and generate_random_walk_steps() functions from the class lecture codes for Brownian Motion and Random Walks. In the generate_random_walk() function, I created an array that stores a sequence of random numbers using the rand function, that samples from a uniform distribution $U(0,1)$. Since the probability of the direction of each step is 0.5, and since each sample is between 0 and 1, we can use the following logic to categorize the samples:

- Case 1: If the sample is between 0 and 0.5, then the direction of the step would be negative.
- Case 2: If the sample is between 0.5 and 1, then the direction of the step would be positive.
- The length for both cases remain the same

Here, depending on which case is satisfied by each sample, a new array is created and each element corresponding to the subject sample is assigned a value of 1 or -1. This creates a 1-D Bernoulli random walk. Mathematically, this can be written as:

$$p_X(x) = \begin{cases} X_i = 1, & p \\ X_i = -1, & 1 - p \end{cases}$$

Where, p = 0.5.

The second function, generate_random_walk() uses the cumsum function to find the cumulative sum of all X_i at every step from 1 to 100. This helps visualize the random walk and plot each walk graphically. Figure 3 shows 20 separate random walks. The x-axis represents the steps from starting at position 0 until position 100. The y-axis represents the cumulative sum of the length of each step, at each position.

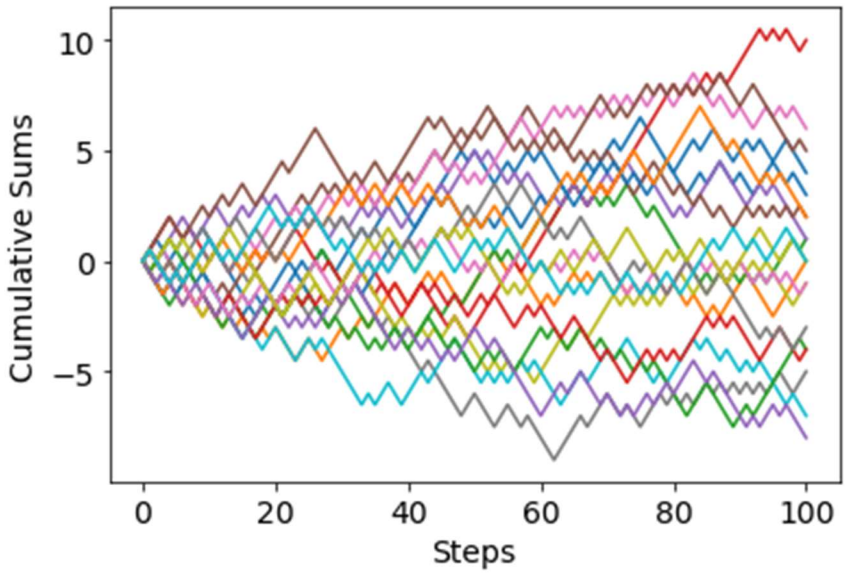


Figure 3: 1-D Bernoulli Random Walk Visualization

b. Given $S_n = \sum_{j=1}^{n} X_j$ which is the sum of the weights assigned to each step of the walk. To run a Monte Carlo on this function, I first repeated every step from 2.1(a) for 10^5 trials. This created 10^5 walks and each walk was stored in a (10^5 x 100) dimension matrix. This represents the sampling part of the Monte Carlo Algorithm.
Next, to evaluate $g(x^{(i)})$, where $i = 1 \ldots \ldots 10^5$, we must check the given condition. The question asks to find the probability such that $S_n > 10$. To find the probability of this condition, we need to check the value of each S_n and count the number of instances where the given condition holds true. The logic to obtain $g(x^{(i)})$ values is as follows:
   - If $S_n > 10$, the $g(x^{(i)})$ is assigned a value of 1
   - If $S_n < 10$, the $g(x^{(i)})$ is assigned a value of 0

   For the last step of the Monte Carlo algorithm, I added all 10^5 elements of $g(x^{(i)})$ and divided it by the number of walks which is 10^5. This gives the probability of the condition that $S_n > 10$.

   Results from code: Probability $P(S_n > 10) = 0.13407$

c. The question asks to find $P(S_n > 55)$ using importance sampling for 10^5 trials. Importance sampling has three steps:
   i. Choosing a proposal distribution function $\pi(x)$: This function is chosen such that when the $g(x^{(i)})$ is evaluated, the instances where the condition $S_n > 55$ is satisfied, must be higher in count i.e., we would be sampling more of the steps with value +1. To do this, we must alter the probability such that the instances where step value of +1 observed must be higher than that of step value -1 observed. Mathematically, this can be written as:

$$p_X(x) = \begin{cases} X_i = 1, & p \\ X_i = -1, & 1-p \end{cases}$$

Where, p = 0.75.

Here I have assumed that the probability of observing a step value of +1 is 0.75 and that of observing a -1 is 0.25. In the code, I created a function sampling_proposal_distribution() that samples from a uniform distribution $U(0,1)$ and checks for the condition described above. This function returns an array of step values of +1s and -1s depending on the altered probability. This was run for 10^5 trials and stored in a matrix of dimension (10^5 x 100).

ii. Finding the likelihood ratio $w(x) = \frac{f_X(x)}{\pi(x)}$: Once the probability is changed, the next step is to find the likelihood ratio or the weight, which is the ratio of nominal distribution to the proposal distribution. Here, $f_X(x)$ is essentially the probability mass function (PMF) of the original function with the original probability, and $\pi(x)$ is the PMF of the proposal distribution function with the altered probability. Here, we are no longer sampling from S_n and are instead sampling from the individual X_i. Since the individual X_i are i.i.d, the likelihood ratio can be calculated as follows:

$$w(X_i) = \frac{f_X(x)}{\pi(x)} = \frac{f_X(x^1).f_X(x^2).f_X(x^3).f_X(x^4) \dots \dots \dots \dots f_X(x^{100})}{\pi(x^1).\pi(x^2).\pi(x^3).\pi(x^4) \dots \dots \dots \dots \pi(x^{100})}$$

Where, $f_X(x^1) = f_X(x^2) = f_X(x^3) = \cdots = f_X(x^{100}) = 0.5$. And, $\pi(x^i)$ is either 0.75 or 0.25 depending on the step value. To find $\pi(x)$,

$$\pi(x) = 0.75^a * 0.25^b$$

Where, $a = frequency\ of\ occurence\ of\ +1\ in\ step\ values\ of\ each\ trial$

$b = frequency\ of\ occurence\ of\ -1\ in\ step\ values\ of\ each\ trial$

The $w(x)$ was calculated 10^5 times for the corresponding 10^5 sequences of step values. The values of $w(x)$ was stored in a 10^5 x 1 vector.

iii. Finding the Importance Sampling Estimator $S_n^{IS} = \frac{1}{n}\sum_{i=1}^{n} g(X_i).w(X_i)$

To find the estimator, I had to first obtain the $g(X_i)$ for each of the 10^5 trials. This was calculated using the step values matrix obtained from 2.1(c)(i). The sum of the step values of all 100 steps for each trial was found. Each of these sums were then checked using the condition $S_n > 55$ and $g(X_i)$ was assigned a value of 1 or 0 depending on whether the condition was satisfied. The $g(X_i)$ was obtained using the following logic:

- If $S_n > 55$, the $g(x^{(i)})$ is assigned a value of 1
- If $S_n < 55$, the $g(x^{(i)})$ is assigned a value of 0

Once the $g(X_i)$ vector was found, S_n^{IS} was found by first finding the dot product of $g(X_i)$ and $w(X_i)$ vectors using the function matmul() on Python which gives the

value of $\sum_{i=1}^{n} g(X_i).w(X_i)$. This value was then divided by the total number of trials (10^5) to find the value of probability such that $S_n > 55$.

Results from code: Probability $P(S_n > 55) = 7.843 * 10^{-9}$

d. Analytically solving for probabilities for (b) and (c)

- For $P(S_n > 10)$,
  Let N_{10} be the minimum number of +1s required in the 100 step values for each trial. Then,

$$N_{10}(1) + (100 - N_{10})(-1) = 10$$
$$N_{10} - 100 + N_{10} = 10$$
$$2N_{10} = 110$$
$$N_{10} = 55$$

  Therefore, $P(S_n > 10) = P(N_{10} > 55)$

$$P(N_{10} > 55) = P(N_{10} = 56) + P(N_{10} = 57) + \cdots + P(N_{10} = 100)$$

$$P(N_{10} > 55) = \binom{100}{56}.0.5^{56}.0.5^{44} + \binom{100}{57}.0.5^{57}.0.5^{43} + \cdots \binom{100}{100}.0.5^{100}.0.5^{0}$$

$$P(N_{10} > 55) = \left[\binom{100}{56} + \binom{100}{57} + \binom{100}{58} + \cdots + \binom{100}{100}\right].0.5^{100}$$

On evaluation, we get

$$P(N_{10} > 55) = 0.1356$$

This result is very close to the value obtained in 2.1(b) for $P(S_n > 10)$, which is 0.13407.

- For $P(S_n > 55)$,
  Let N_{55} be the minimum number of +1s required in the 100 step values for each trial. Then,

$$N_{55}(1) + (100 - N_{55})(-1) = 55$$
$$N_{55} - 100 + N_{55} = 55$$
$$2N_{55} = 155$$
$$N_{55} = 77.5$$

  Therefore, $P(S_n > 55) = P(N_{10} \geq 78)$

$$P(N_{55} \geq 78) = P(N_{55} = 56) + P(N_{55} = 57) + \cdots + P(N_{55} = 100)$$

$$P(N_{55} \geq 78) = \binom{100}{78}.0.5^{78}.0.5^{22} + \binom{100}{79}.0.5^{79}.0.5^{21} + \cdots \binom{100}{100}.0.5^{100}.0.5^{0}$$

$$P(N_{10} > 55) = \left[\binom{100}{78} + \binom{100}{79} + \binom{100}{80} + \cdots + \binom{100}{100}\right].0.5^{100}$$

On evaluation, we get

$$P(N_{55} > 55) = 7.9526 * 10^{-9}$$

This result is very close to the value obtained in 2.1(c) for $P(S_n > 55)$, which is $7.843 * 10^{-9}$. In the code, I used math.comb() function to compute these values.

e. Estimating errors in Monte Carlo estimates for 2.1(b) and (c)

i. The Monte Carlo Standard error is defined as:

$$MCSE = Var(S_n) = \frac{Var\big(g(x)\big)}{n} = \frac{\sigma^2}{n}$$

Where, $g(x)$ is the evaluation function and n is the number of trials.
To find the Monte Carlo error, I found the variance of the evaluations for both cases, $P(S_n > 10)$ and $P(S_n > 55)$ and divided by 10^5 trials.

Result from code: MCSE for $P(S_n > 10) = 1.161 * 10^{-6}$
MCSE for $P(S_n > 55) = 2.042 * 10^{-6}$

Using the Central Limit Theorem, the confidence intervals are found using the following equation:

$$S_n + z.\frac{\sigma}{\sqrt{n}} \geq \mu_x \geq S_n - z.\frac{\sigma}{\sqrt{n}}$$

This can be rewritten as:

$$S_n + z.\sqrt{Var\big(g(x)\big)} \geq \mu_x \geq S_n - z.\sqrt{Var\big(g(x)\big)}$$

For a 95% confidence interval, $z = 2$
Here, S_n is the Monte Carlo Estimator which is also the probability that was calculated from 2.1 (b) and (c). Substituting these values into the above equation, I got the following confidence intervals from the code:

Result from code:

Left Confidence Interval for $P(S_n > 10) = 0.136225$
Right Confidence Interval for $P(S_n > 10) = 0.131915$
Left Confidence Interval for $P(S_n > 10) = 0.0028615$
Right Confidence Interval for $P(S_n > 10) = -0.00286148$

Here the probability that $S_n > 10$ is well within the left and right confidence intervals. The values of both left and right CIs for this case is close to the true probability value. This makes sense since the $P(S_n > 10)$ is significantly higher than $P(S_n > 55)$ for the random walk, as shown in the analytical solution. The probability that $S_n > 55$ is also well within the left and right CIs. However, the values for both CIs is much higher than the true probability value which is very close to zero. This means that for multiple trials of the random walk, it is likely that there would be more instances where the probability value of $P(S_n > 55)$ would lie within the confidence intervals, as compared to the probability value of $P(S_n > 10)$. This is evident from the results in the next section, in 2.1(e)(ii).

ii. As per the question, I ran 1000 independent replicates of the Monte Carlo estimators for $P(S_n > 10)$ and $P(S_n > 55)$. For each of the two estimators for each of the 1000 replicates, I found an array of 1000 Monte Carlo Standard Error, an array of 1000 left CIs, and an array of 1000 right CIs. Using an if statement in the code, I checked for how many of the probability values found were within the confidence intervals. The results are as follows:

Results from code:
Number of confidence intervals that contain the true value of the probability that $S_n > 10 = 956$
Number of confidence intervals that contain the true value of the probability that $S_n > 55 = 1000$

iii. For this question, to extract a sequence of "running mean" estimators, I used the np.cumsum() function to first compute the cumulative summation of the steps for each of the 10^5 trials. This gives a matrix of dimension (10^5 x 101). Then I took the last element of the cumulative sum for each trial and divided it by the cumulative sum of the steps. This gives the running mean of the estimator for 10^5 trials. I found the running mean for 1000 such replicates of the estimator and stored it in an array. This array was then sorted in ascending order to enforce a 95% band.
Unfortunately, I did not have enough time to attempt the entire question and implement it in the code.

3.2. 3-D Gaussian Random Walk

Here we have considered a 3-D gaussian random walk,

$$S = \sum_{j=1}^{N} \mathbf{X_j}$$

Where, $\mathbf{X_j} = (x_1, x_2, x_3)$ is a 3-D vector. Each vector is randomly sampled from a normal distribution $x_i \sim N(0,1)$ for $i = 1, 2, 3$.

The question considers $N = 100$ steps and asks to find the probability such that the distance travelled in N steps is greater than a given value where distance is calculated by

$$|S| = \sqrt{s_1^2 + s_2^2 + s_3^2}$$

Question 2.2

a. Writing an N-step random walk for a 3-D Gaussian is like that of a 1-D Bernoulli. For each of the values of (x_1, x_2, x_3), I used the rand function three times to obtain three

sequences of 100 samples from a gaussian distribution $N(0,1)$. Each value of (x_1, x_2, x_3) represents the lengths of the step in the x-y-z directions for each of the 100 steps.
I also used the cumsum() function on all three elements of $\mathbf{X_j}$ to create a 3-D plot for visualization. Figure 4 shows a 3-D plot of 10 random walks. The x-y-z axes represent the values of (x_1, x_2, x_3) for all 100 steps.

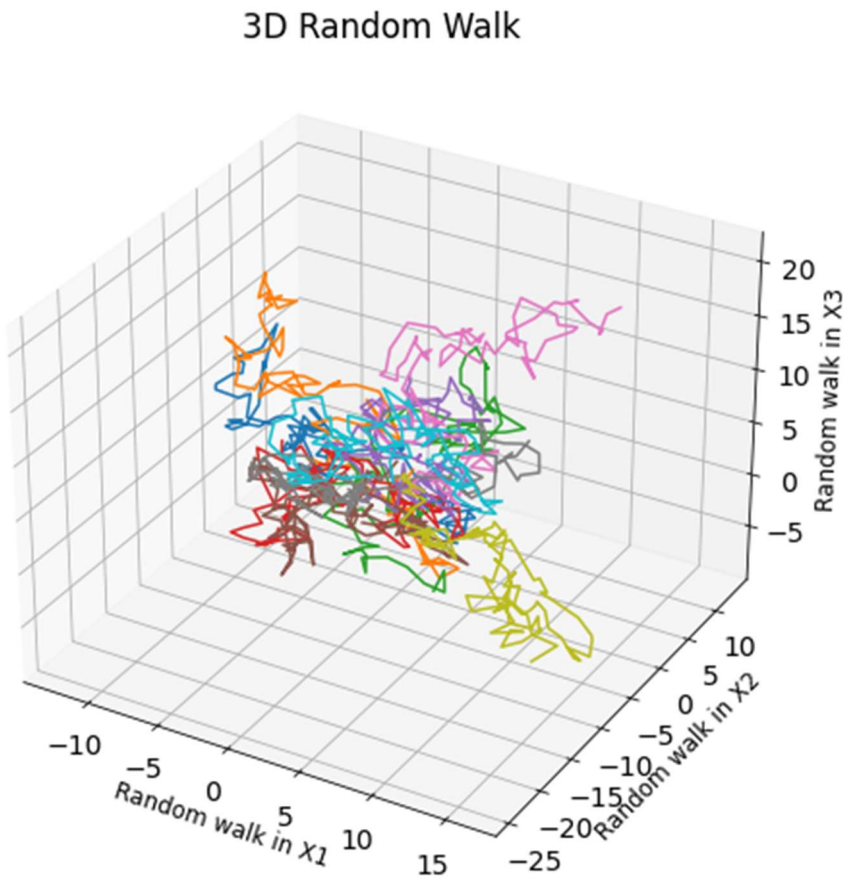


Figure 4: 3-D Gaussian Random Walk

b. Given $S = \sum_{j=1}^{N} \mathbf{X_j}$ which is the sum of the weights assigned to each step of the walk. To run a Monte Carlo on this function, I first repeated every step from 2.2(a) for 10^5 trials. This created 10^5 walks and each x_i walk was stored in three different (x_1 x 100) dimension matrices. This represents the sampling part of the Monte Carlo Algorithm. Next, to evaluate $g(X^{(i)})$, where $i = 1 \ldots \ldots 10^5$, we must check the given condition. The question asks to find the probability such that $|S| > 10$. To find the probability of this condition, we need to first find an array of 10^5 values of $|S|$. From the three matrices constructed for the elements of (x_1, x_2, x_3), I summed all 100 elements of x_1, x_2, x_3 for

each of the 10^5 trials to construct three arrays for each of the elements (s_1, s_2, s_3). Mathematically this can be written as:

$s_1 = \sum_{j=1}^{10^5} x_{1_j} \quad s_2 = \sum_{j=1}^{10^5} x_{2_j} \quad s_3 = \sum_{j=1}^{10^5} x_{3_j}$

Then find $|S|$,

$$|S| = \sqrt{s_1^2 + s_2^2 + s_3^2}$$

Now I have an array of 10^5 elements of $|S|$. To check for the given probability condition $|S| > 10$, I used the following logic:

- I counted the number of instances when each value of the array $|S|$ is larger than the value 10
- Next, I divided this count by the total number of trials which is 10^5 here, to obtain the probability of the required condition.

Results from code: Probability $P(|S| > 10) = 0.80166$

c. To do importance sampling for $P(|S| > 55)$, a new biasing distribution is necessary. For 1-D Bernoulli random walk, it was a simpler solution since there were only two outcomes (+1 or -1) which were assigned equal probabilities. Altering the probability towards the desired outcome was easier in this case because of the nature of a Bernoulli random variable. However, for a gaussian random variable, the outcomes are infinitely many. It would be impossible to define a weightage to each outcome and then alter them to facilitate importance sampling. I did not have enough time to address this question and attempt to find an answer to this.

d. To find the Monte Carlo standard error, I first used an array to store the evaluations $g(X^{(i)})$ for $P(|S| > 10)$. Then using the following equations, I found the error and left and right confidence intervals. This is same as the method used in 2.1. (e)(i).

$$MCSE = Var(S_n) = \frac{Var\big(g(x)\big)}{n} = \frac{\sigma^2}{n}$$

$$S_n + z.\sqrt{Var\big(g(x)\big)} \geq \mu_x \geq S_n - z.\sqrt{Var\big(g(x)\big)}$$

Results from code:

$$MCSE = 1.5819 * 10^{-6}$$

$$\text{Left Confidence Interval for } P(S > 10) = 0.805475$$

$$\text{Right Confidence Interval for } P(S > 10) = 0.800545$$

The value of probability obtained in 2.2(b) which is 0.80166 is well within the confidence interval.

Problem 4: Multilevel Monte Carlo and Control Variates for Stochastic ODEs

Question: What is the distribution of ΔW_n?

Answer: Given that $\Delta W_n = W(t_{n+1}) - W(t_n)$

Where, $W(t_{n+1})$ is the Weiner process at t_{n+1} and $W(t_n)$ is the Weiner process at t_n. We know that $W(t + \Delta t) - W(t) \sim N(0, \Delta t)$ i.e., the difference in two instances of a Weiner processes separated by a Δt has a gaussian distribution.

This means ΔW_n has a gaussian distribution.

4.1 Geometric Brownian Motion

For a Geometric Brownian Motion, we are given,

$$b(t, Y_t) = \mu Y_t$$

$$h(t, Y_t) = \sigma Y_t$$

Where $\mu = 0.05$ and $\sigma = 0.2$. Also given are the initial state $Y_0 = 1$ and range $0 < t < 1$.

The Weiner process is defined as:

$$W^n(t) = \frac{1}{\sqrt{n}} \sum_{1 \le k \le \lfloor nt \rfloor} X_k$$

1. Simulating a Geometric Brownian Motion for Δt:
   To simulate a geometric Brownian motion for the given conditions, I used two functions from the lecture notes MLMC_SDE_Ornstein_Uhlenbeck_two_levels. The first function is Brownian_motion_simulate() which takes the final time, T, small increments of time, Δt, and number of samples, num_samples and returns an array of ΔW_n at each time step. First, this function generates samples for the required number given to the function and samples from a normal distribution $N(0,1)$. Each sample is divided by the square root of the number of samples. The cumulative sum of these samples is essentially the ΔW_n, which is returned from the function. The second function that I used is the euler_maruyama(). This function takes b and h which are defined in the question, the initial point x_0, time step Δt, and final time, T. This function also takes the Brownian motion values that were returned from the first function. This function then evaluates the sequential iteration according to the given equation:

$$Y_{n+1} = Y_n + b(t, Y_t)\Delta t + h(t, Y_t)\Delta W_n$$

   The very last value of the states, $Y(1)$, for each step, are returned by the function along with its corresponding time steps. To find the mean of the last state $Y(1)$, I summed all the elements of the array and divided it by the number of time steps. To find the variance, I used the var() function on the same array. The results are as follows:

Results from code: Mean = 1.2817 and Variance = 0.02477

Figure 5 shows the plot of the Brownian motion for the last state, $Y(1)$, against the corresponding time steps.

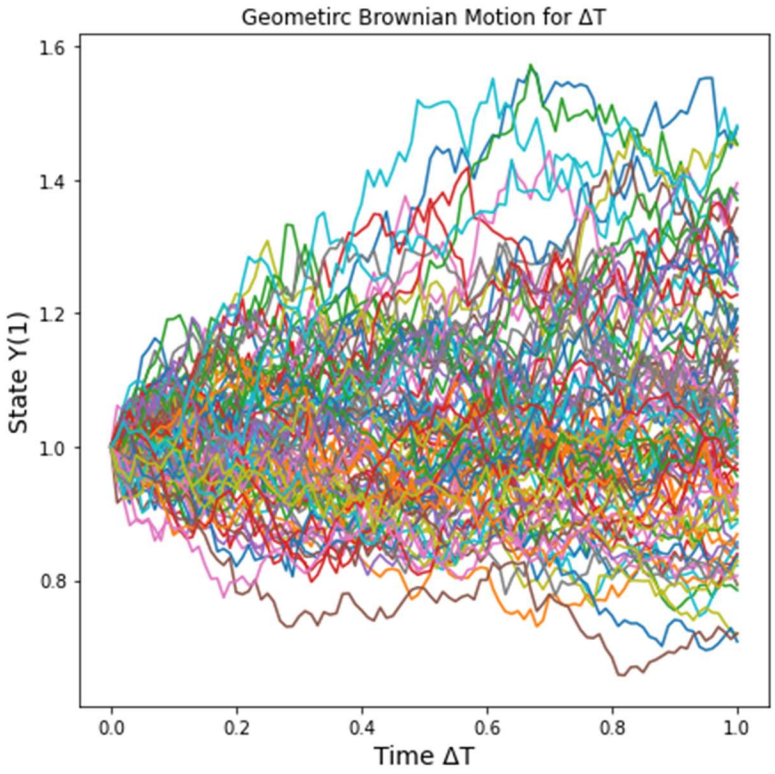


Figure 5: Geometric Brownian Motion for Δt

Analytically calculating the last state $Y(1)$ from the given equation:

$$Y_t = Y_0 \exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t\right)$$

Where Y_0, μ, σ are given. W_t is the array of brownian motion corresponding to the very last state which was obtained from the brownian_motion_simulate() function. On substituting the necessary values, I obtained a mean and variance which is as follows:

Results from code: Mean = 1.29975 and Variance = 0.02229

These values are very close to the values of mean and variance obtained from the first part of the question. Therefore, it is evident the Euler Maruyama algorithm is an efficient stochastic process.

To find the variance of the Monte Carlo Estimator of the mean, I ran the Monte Carlo algorithm on states $Y_i(t)$ for 1000 times. For each run, I took the last state $Y(1)$ and stored it in a separate array. Then I summed all elements of the array of final states and divided by the number of runs, which is 1000 here. This gives me the Monte Carlo estimator which is also the mean of the final

states. I used the same function as the previous part to find the variance of the same array of Monte Carlo estimators. The results obtained are shown below:

Results from code: Variance = $4.54 * 10^{-5}$

2. For this question, I simulated two different processes, one with Δt and the other with $4\Delta t$. The process for both is very similar. The only aspect that was different is the time step that was passed into the two functions that were used. To simulate a $4\Delta t$ process, I sampled every fourth element of the time step array and passed it into the brownian_motion_simulate() function. This resulted in the array of Brownian motion cutting down to 25 x 100 elements. The Brownian motion is then passed into the Euler Maruyama function to simulate a geometric Brownian motion. Figure 6 shows two subplots of the geometric Brownian motion for Δt and $4\Delta t$.

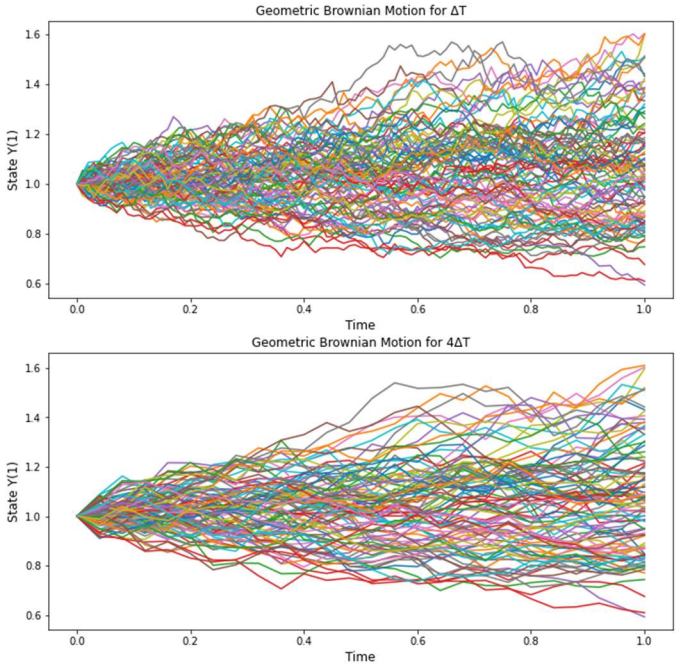


Figure 6: Geometric Brownian Motion for Δt and $4\Delta t$

3. To perform a four-level MLMC, I used the multilevel_montecarlo_sde() and brownian_motion_fine_to_coarse() functions from the same lecture notes as mentioned in 4.1 (1). In the first function I incorporated levels 0 to 4 to simulate a multilevel Monte Carlo, to calculate the mean and variance of $E[Y(1)]$ and $E[P]$. This function takes five variables with different number of samples for each of the levels 0 to 4, which are uniformly decreasing at each level. The number of samples at each level is chosen this way because the time step becomes less fine or coarser at each level. This function also takes the evaluator function, time step at level 4 (the time step at the finest level), and a coarse scalae value that scales the time step at each level.
   Within this function, I calculated the brownian motion using brownian_motion_simulate() function and then passed these values into the brownian_motion_fine_to_coarse() function. This function essentially takes the fine brownian motion values at a particular level and

extracts a sample of coarse brownian motion values for that level. Once I obtained the fine and coarse brownian motion values, I passed them into the evaluator() function. This function uses the euler_maruyama() function to obtain times and states for a given brownian motion and time step. I sliced the matrix of states from the euler_maruyama() function to obtain the very last state at $Y(1)$. This array is then passed back into the multilevel_montecarlo_sde() function which stores separate states for the fine and coarse brownain motions. These are the states used to find mean and variance. This entire process was repeated five times for the levels 0 to 4. A separate mean and variance were calculated for each level. The summation of the means at each level gives the expectation at the last state $E[Y(1)]$. Additionally, the difference between the states at the fine and coarse timesteps were taken to find the states at Δt between each level. Four such sequences were obtained which were also used to find the variance of the estimator for Δt between each level.

This entire process was again repeated using the p_evaluator() function to obtain $E[P]$. This function essentially uses the same logic and functions as the evaluator() function. The only additional aspect is adding a payoff function P which is given in the question as:

$$P = \exp(-0.05)\max(0, Y(1) - 1)$$

After obtaining all the means and variances at each level for 1000 trials, these values were plotted using histograms. These plots are shown in Figure 7 and Figure 8.

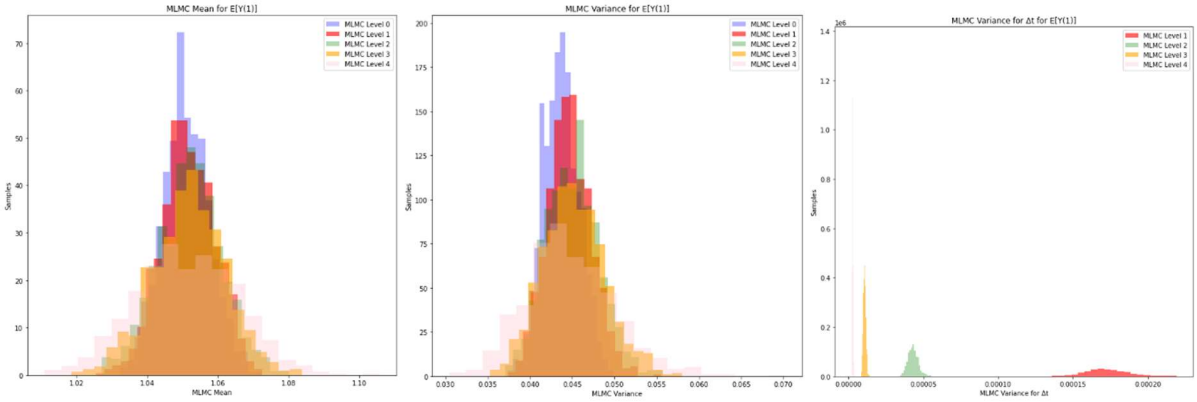


Figure 7: (left to right) MLMC mean, variance and variance for Δt for $E[Y(1)]$

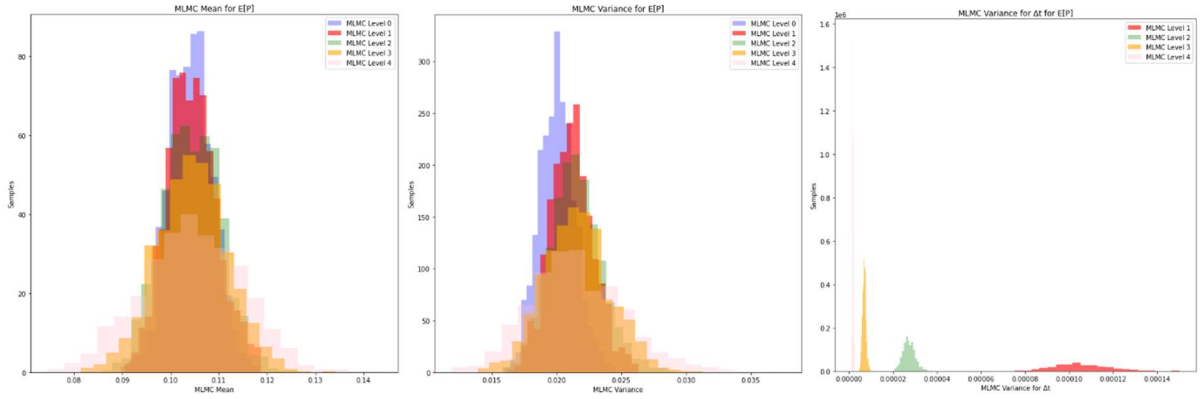


Figure 8: (left to right) MLMC mean, variance and variance for Δt for $E[P]$

One thing that is evident and most definitely wrong about these histograms is the fact that the variance at the finest level (level 4) seems to be the highest and the variance seem to be increasing as the levels are added. The idea of incorporating multiple levels into a Multilevel Monte Carlo is to reduce the variance of the Monte Carlo estimator such that this algorithm gives accurate results for the problem in question. However, these graphs seem to be giving the opposite results such that the variance at each level and between the levels seem to be directly proportional to the number of level when it should in fact be decreasing as each level is introduced. This could be a problem with the logic of my code. I was not able to find exactly what could have gone wrong here.
However, here are the means obtained from the code for the Multilevel Monte Carlo Estimator:
Results from code: $E[Y(1)] = 5.249 \ \ and \ \ E[P] = 0.534$

4. The theoretical cost of a MLMC estimator is defined as follows:

$$C_N^{MLMC} = N_0 C_0 + \sum_{l=1}^{L} N_l (C_l + C_{l-1})$$

Where, l denotes a summation corresponding to a particular level of the Multilevel Monte Carlo.

This equation was taken from the "LectureNotes2019" under the section, "Multilevel Monte Carlo" from page 206.

The second part of the question asks to find the optimal sample allocation for a given target accuracy. The target accuracy is the fixed estimator variance which can be denoted by:

$$Var(S_N^{MLMC}) = \epsilon^2$$

To find the optimal sample allocation, we must find the number of samples N for which the variance of the estimator is the least. This requires optimization of the variance of the estimator subject to the maximum allowable cost. Mathematically this can be written as:

$$\min_{\mathbf{N}} \sum_{l=0}^{L} \frac{V_l}{N_l} \text{ subject to: } \sum_{l=0}^{L} N_l \hat{C}_l \leq C$$

Where, $V_l = Var[\Delta_l]$

$$\hat{C}_l = \begin{cases} C_0, & if\ l = 0 \\ C_l + C_{l-1}, & otherwise \end{cases}$$

For simplification of the optimization, a relaxed form of the problem is solved where N_l is assumed to be continuous. Considering the Lagrangian formulation:

$$L(N, \lambda^2) = \sum_{l=0}^{L} \frac{V_l}{N_l} + \lambda^{-2} \left(\sum_{l=0}^{L} N_l \hat{C}_l - C \right)$$

Where the Lagrangian dual function is:

$$g(\lambda^2) = \inf_{\mathbf{N}} L(\mathbf{N}, \lambda^2)$$

To find the optimal sample allocation for a given lambda, we must find the extremum of the Lagrangian by setting its gradient to zero. Writing this mathematically, we get:

$$\frac{\partial L(N, \lambda^2)}{\partial N_l} = -\frac{V_l}{N_l^2} + \lambda^{-2} \hat{C}_l = 0$$

Solving this we get:

$$N_l = \lambda \sqrt{\frac{V_l}{\hat{C}_l}}$$

We can find the variance of the estimator using the allocation N_l, which is given by:

$$Var(S_N^{MLMC}) = \epsilon^2 = \lambda^{-1} \sum_{l=0}^{L} \sqrt{V_l \hat{C}_l}$$

For the given target variance, we can find λ as:

$$\lambda = \epsilon^{-2} \sum_{l=0}^{L} \sqrt{V_l \hat{C}_l}$$

Now, substituting λ into the optimal sample allocation equation gives the expression for N_l:

$$N_l = \epsilon^{-2} \sum_{l=0}^{L} \sqrt{V_l \hat{C}_l} \cdot \sqrt{\frac{V_l}{\hat{C}_l}}$$

This expression provides the number of samples required for least variance of the Multilevel Monte Carlo estimator.

In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

In [2]:
```
import numpy as np
import sympy as sympy # a compute algebra system in python
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib
font = {'size'   : 14}

matplotlib.rc('font', **font)
```

In [3]:
```
def monte_carlo(num_samples, sample_generator, g_evaluator, cumsum=False):

    samples = sample_generator(num_samples)
    evaluations = g_evaluator(samples)
    if cumsum is False:
        estimate =  np.sum(evaluations, axis=0) / float(num_samples)
    else:
        estimate = np.cumsum(evaluations, axis=0) / np.arange(1,num_samples+1, dt

    return estimate, samples, evaluations
```

In [4]:
```
nsamples = 10000
sampler = np.random.rand # uniform distribution
#sampler = np.random.rand
#sampler = np.random.gamma
g = lambda x: ((1 - x)**(-2/3)) # the evaluator
estimate, samples, evaluations = monte_carlo(nsamples, sampler, g)
print("Estimated integral value = {0}".format(estimate))
print(evaluations)
```

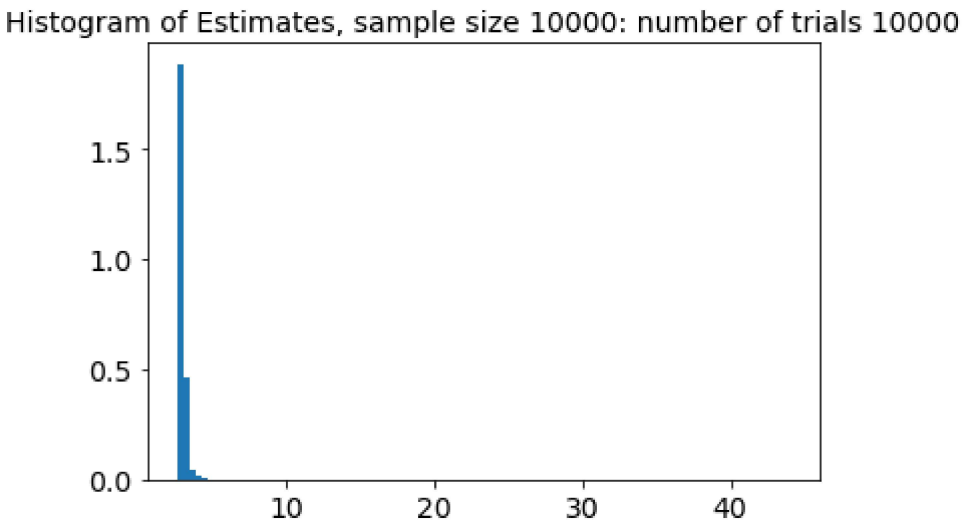
```
Estimated integral value = 2.9850419618918766
[ 1.16508263  1.6636528   1.11911452 ...  1.17449909  1.28581855
 10.01861937]
```

In [5]:
```
print(samples)
```

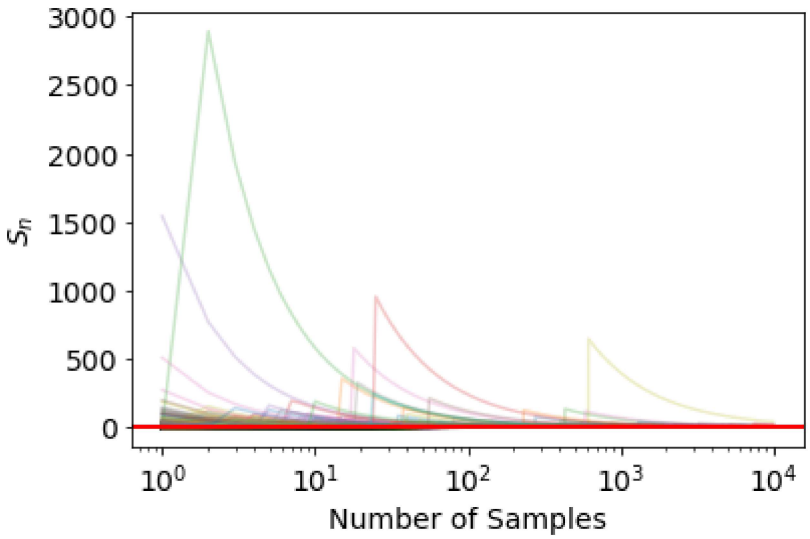
```
[0.20482099 0.5339785  0.15532779 ... 0.21436473 0.31414791 0.96846534]
```

In [6]:
```
nsamples = 10000
ntrials = 10000
estimates = np.zeros((ntrials))

for ii in range(ntrials):
    estimate, samples, evaluations = monte_carlo(nsamples, sampler, g)
    estimates[ii] = estimate
    if ii % 1000 == 0:
        print("Trial {0}: estimated integral {1}".format(ii, estimates[ii]))

fig, axs = plt.subplots(1)
axs.hist(estimates, bins=100, density=True)
#axs.axvline(x=int_numeric, linewidth=6, color='red')
axs.set_title("Histogram of Estimates, sample size 10000: number of trials 10000"
plt.show()
```

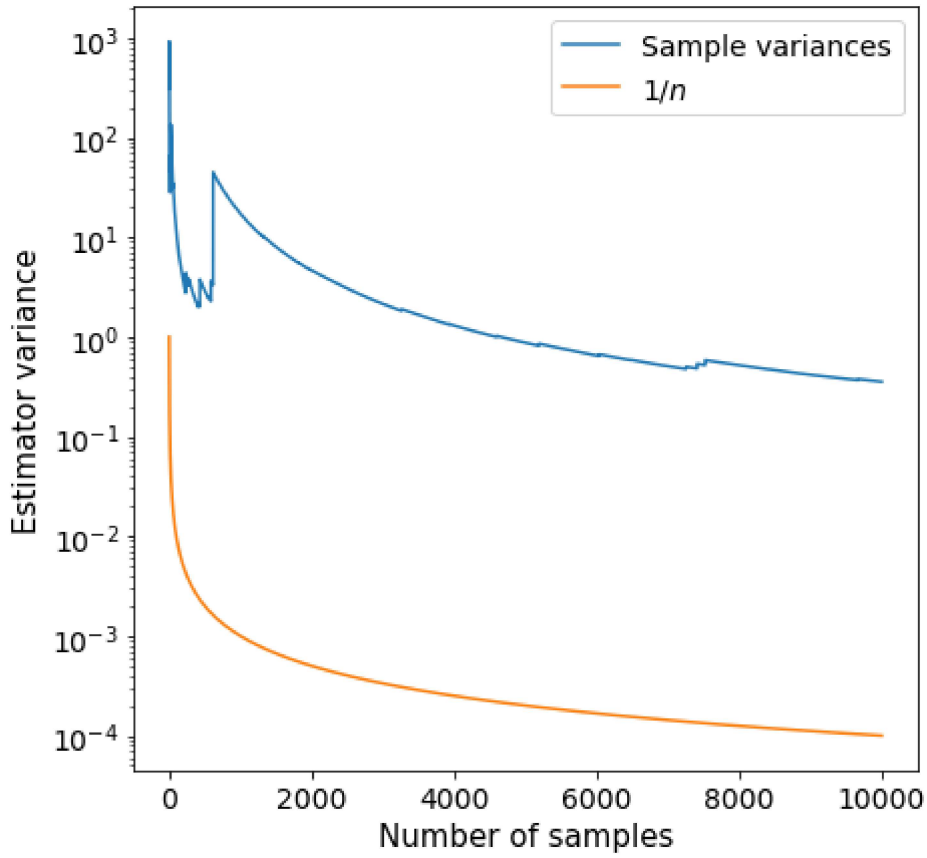
```
Trial 0: estimated integral 3.0516306130269593
Trial 1000: estimated integral 3.003761473486258
Trial 2000: estimated integral 2.899131638680744
Trial 3000: estimated integral 2.9946143513291053
Trial 4000: estimated integral 2.8510954309234267
Trial 5000: estimated integral 2.9941672693812174
Trial 6000: estimated integral 3.0329223424265046
Trial 7000: estimated integral 2.956906095816782
Trial 8000: estimated integral 4.641648526833894
Trial 9000: estimated integral 2.855779322802149
```



In [7]:
```
ntrials = 10000
nsamples = 10000
estimate_vals = np.zeros((ntrials, nsamples))
plt.figure()
for ii in range(ntrials):
    estimate, samples, evaluations = monte_carlo(nsamples, sampler, g, cumsum=Tru
    estimate_vals[ii, :] = estimate
    plt.semilogx(np.arange(1, nsamples+1), estimate_vals[ii, :], '-', alpha=0.25)
plt.axhline(y=3, color='red', linewidth=2)
plt.xlabel('Number of Samples')
plt.ylabel(r'$S_n$')
plt.show()
```



In [8]:
```
plt.figure(figsize=(7,7))
variances = np.var(estimate_vals, axis=0) # variance of the histogram at each $n$
plt.semilogy(np.arange(1, variances.shape[0]+1), variances, label='Sample varianc
plt.semilogy(np.arange(1, variances.shape[0]+1), 1/(np.arange(1, variances.shape[
plt.ylabel('Estimator variance', fontsize=15)
plt.xlabel('Number of samples', fontsize=15)
plt.legend(fontsize=14)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
plt.show()
```



In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

In [2]:
```
import numpy as np
import sympy as sympy # a compute algebra system in python
import math
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib
font = {'size'   : 14}

matplotlib.rc('font', **font)
```

In [3]:
```
def generate_random_walk_steps(num_steps):
    """ Generate a set of steps for a random walk"""
    X = np.random.rand(num_steps) # samples from a uniform
    # Inverse CDF Trick
    X[X > 0.5] = 1.0
    X[X < 0.5] = -1.0
    return X
```

In [4]:
```
def generate_random_walk(steps, n):
    Y = np.concatenate((np.array([0]), np.cumsum(steps)/np.sqrt(n)))
    return Y
```

In [5]:
```
n = 1
nwalks = 10**5
nsamples = 100
steps_vals = np.zeros((nwalks, nsamples))
S = np.zeros(nwalks)
rand_walk_for_plot = np.zeros((nwalks, nsamples+1))

for ii in range(nwalks):
    steps_vals[ii,:] = generate_random_walk_steps(nsamples)
    S[ii] = sum(steps_vals[ii,:])
    rand_walk_for_plot[ii] = generate_random_walk(generate_random_walk_steps(nsam
print(steps_vals)
print(S)

print(rand_walk_for_plot)
tplot = np.linspace(0, nsamples, nsamples+1)
plt.plot(tplot, rand_walk_for_plot.T, n)
plt.xlabel('Steps', fontsize=14)
plt.ylabel('Cumulative Sums', fontsize=14)
```

```
[[-1.  1. -1. ... -1. -1. -1.]
 [-1. -1.  1. ...  1. -1.  1.]
 [ 1.  1.  1. ... -1. -1.  1.]
 ...
 [ 1.  1. -1. ... -1. -1.  1.]
 [ 1.  1.  1. ... -1.  1.  1.]
 [-1.  1.  1. ...  1.  1. -1.]]
[ -2.  -8.   6. ...   8. -24. -16.]
```

In [6]:
```
ntrials = 10**5
g_x = np.zeros(ntrials)
cumsum_step_vals = np.cumsum(steps_vals, 1)
for jj in range(ntrials):
    if cumsum_step_vals[jj,-1] > 10:
        g_x[jj] = 1
    else:
        g_x[jj] = 0
#plt.figure(cumsum_step_vals)
prob_greater_10 = np.sum(g_x)/ntrials
print(prob_greater_10)
print(g_x)
print(g_x.shape)
```

```
0.1365
[0. 0. 0. ... 0. 0. 0.]
(100000,)
```

In [7]:
```
def sampling_proposal_distribution(num_steps):
    X_prop_dist = np.random.rand(num_steps) # samples from a uniform
    # Inverse CDF Trick
    X_prop_dist[X_prop_dist > 0.25] = 1.0
    X_prop_dist[X_prop_dist < 0.25] = -1.0
    return X_prop_dist
```

In [8]:
```
ntrials = 10**5
nsamples = 100
steps_vals_prop_dist = np.zeros((ntrials, nsamples))
S_pd = np.zeros(ntrials)
plt.figure()

for kk in range(ntrials):
    steps_vals_prop_dist[kk,:] = sampling_proposal_distribution(nsamples)
    S_pd[kk] = sum(steps_vals_prop_dist[kk,:])
print(steps_vals_prop_dist)
print(S_pd)
```

```
[[ 1.  1.  1. ...  1.  1. -1.]
 [-1.  1. -1. ...  1.  1. -1.]
 [ 1. -1.  1. ... -1.  1.  1.]
 ...
 [ 1.  1. -1. ... -1. -1.  1.]
 [-1.  1.  1. ...  1.  1.  1.]
 [-1.  1.  1. ...  1. -1.  1.]]
[54. 54. 46. ... 44. 46. 48.]

<Figure size 432x288 with 0 Axes>
```

In [9]:
```
f_X = 0.5**nsamples
ntrials = 100000
pi_X = np.zeros((ntrials,1))

for jj in range(ntrials):
    c_plus1 = np.count_nonzero(steps_vals_prop_dist[jj,:] == 1)
    c_minus1 = np.count_nonzero(steps_vals_prop_dist[jj,:] == -1)
    pi_X[jj] = 0.75**c_plus1 * 0.25**c_minus1

likelihood_ratio = f_X/pi_X
print(likelihood_ratio)
```

```
[[2.31559687e-07]
 [2.31559687e-07]
 [1.87563346e-05]
 ...
 [5.62690038e-05]
 [1.87563346e-05]
 [6.25211154e-06]]
```

In [10]:
```
g_x_55 = np.zeros(ntrials)
evals_55 = np.zeros(ntrials)
cumsum_step_vals_prop_dist = np.cumsum(steps_vals_prop_dist, 1)
for jj in range(ntrials):
    if cumsum_step_vals_prop_dist[jj,-1] > 55:
        g_x_55[jj] = 1
    else:
        g_x_55[jj] = 0

g_x_55 = g_x_55.reshape(1,-1)
evals_55 = np.matmul(g_x_55,likelihood_ratio)
prob_greater_55 = evals_55/ntrials

print(g_x_55.shape)
print(likelihood_ratio.shape)
print(evals_55)
print(prob_greater_55)
```

```
(1, 100000)
(100000, 1)
[[0.00079271]]
[[7.92707011e-09]]
```

In [11]:
```
#Analytical expressions
Anex_count_10 = 0
for ii in range(56, 100):
    Anex_count_10 = Anex_count_10 + math.comb(100,ii)
Anex_prob_10 = (0.5**(100))*Anex_count_10
print(Anex_count_10)
print(Anex_prob_10)
```

```
1719270293904594780080413627 99
0.13562651203691736
```

In [12]:
```
Anex_count_55 = 0
for ii in range(78, 100):
    Anex_count_55 = Anex_count_55 + math.comb(100,ii)
Anex_prob_55 = (0.5**(100))*Anex_count_55
print(Anex_count_55)
print(Anex_prob_55)
```

```
1008119959331107357 9495
7.95266423689307e-09
```

Problem 2.1 e (i)

In [13]:
```
# Estimate Monte Carlo Error in estimate for P(S>10)
mc_error_10 = np.var(g_x)/ntrials
# Estimate Monte Carlo Error in estimate for P(S>10)
mc_error_55 = np.var(g_x_55)/ntrials
# z = 2 for 95% confidence interval
left_ci_10 = prob_greater_10 + 2*((mc_error_10)**(1/2))
right_ci_10 = prob_greater_10 - 2*((mc_error_10)**(1/2))

left_ci_55 = prob_greater_55 + 2*((mc_error_55)**(1/2))
right_ci_55 = prob_greater_55 - 2*((mc_error_55)**(1/2))
print(mc_error_10)
print(left_ci_10)
print(right_ci_10)
print(mc_error_55)
print(left_ci_55)
print(right_ci_55)
```

```
1.1786774999999997e-06
0.13867133829699566
0.13432866170300437
2.032878231e-06
[[0.00285159]]
[[-0.00285157]]
```

Problem 2.1 e(ii)

In [14]:
```
ntrials = 10**5
nsamples = 100
replicate = 1000
count_2_1b = 0
count_2_1c = 0
steps_vals_2_1b = np.zeros((ntrials, nsamples))
g_x_2_1_b = np.zeros((ntrials,1))

for ii in range(replicate):
    # For 2.1b
    for jj in range(ntrials):
        steps_vals_2_1b[jj,:] = generate_random_walk_steps(nsamples)

    cumsum_step_vals_2_1b = np.cumsum(steps_vals_2_1b, 1)
    for jj in range(ntrials):
            if cumsum_step_vals_2_1b[jj,-1] > 10:
                g_x_2_1_b[jj] = 1
            else:
                g_x_2_1_b[jj] = 0

    prob_2_1_b = np.sum(g_x_2_1_b)/ntrials
    mc_err_2_1b = np.var(g_x_2_1_b)/ntrials
    left_ci_2_1b = prob_2_1_b - 2*((mc_err_2_1b)**(1/2))
    right_ci_2_1b = prob_2_1_b + 2*((mc_err_2_1b)**(1/2))

    if Anex_prob_10 >= left_ci_2_1b and Anex_prob_10 <= right_ci_2_1b:
        count_2_1b += 1
    else:
        count_2_1b += 0

    # For 2.1c
    steps_vals_2_1c = np.zeros((ntrials, nsamples))
    for jj in range(ntrials):
        steps_vals_2_1c[jj,:] = sampling_proposal_distribution(nsamples)

    f_X_2e = 0.5**nsamples
    pi_X_2e = np.zeros((ntrials,1))

    for jj in range(ntrials):
        c_plus1_2e = np.count_nonzero(steps_vals_prop_dist[jj,:] == 1)
        c_minus1_2e = np.count_nonzero(steps_vals_prop_dist[jj,:] == -1)
        pi_X_2e[jj] = 0.75**c_plus1_2e * 0.25**c_minus1_2e
    likelihood_ratio_2e = f_X_2e/pi_X_2e

    g_x_2_1_c = np.zeros(ntrials)
    evals_2_1_c = np.zeros(ntrials)
    cumsum_step_vals_prop_dist = np.cumsum(steps_vals_prop_dist, 1)
    for jj in range(ntrials):
        if cumsum_step_vals_prop_dist[jj,-1] > 55:
            g_x_2_1_c[jj] = 1
        else:
            g_x_2_1_c[jj] = 0

    g_x_2_1_c = g_x_2_1_c.reshape(1,-1)
    evals_2_1_c = np.matmul(g_x_2_1_c,likelihood_ratio_2e)
    prob_2_1_c = evals_2_1_c/ntrials
```

```
    mc_err_2_1c = np.var(g_x_2_1_c)/ntrials
    left_ci_2_1c = prob_2_1_c - 2*((mc_err_2_1c)**(1/2))
    right_ci_2_1c = prob_2_1_c + 2*((mc_err_2_1c)**(1/2))

    if Anex_prob_55 >= left_ci_2_1c and Anex_prob_55 <= right_ci_2_1c:
        count_2_1c += 1
    else:
        count_2_1c += 0

print(count_2_1b)
print(count_2_1c)
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [14], in <cell line: 9>()
     32 steps_vals_2_1c = np.zeros((ntrials, nsamples))
     33 for jj in range(ntrials):
---> 34     steps_vals_2_1c[jj,:] = sampling_proposal_distribution(nsamples)
     36 f_X_2e = 0.5**nsamples
     37 pi_X_2e = np.zeros((ntrials,1))

KeyboardInterrupt:
```

Problem 2.1 e(iii)

In [ ]:
```
ntrials = 10**5
nsamples = 100
replicate = 1000
count_21b = 0
count_21c = 0
steps_vals_21b = np.zeros((ntrials, nsamples))
g_x_21b = np.zeros((ntrials,1))
den_S_run_21b = np.arange((1, ntrials+1))

for ii in range(replicate):
    # For 2.1b
    for jj in range(ntrials):
        steps_vals_21b[jj,:] = generate_random_walk_steps(nsamples)

    cumsum_step_vals_21b = np.cumsum(steps_vals_21b, 1)
    g_x_21b = cumsum_step_vals_21b[jj,-1] > 10
    prob_21b = np.sum(g_x_21b)/ntrials
    Sn_running_21b[ii,:] = np.cumsum(g_x_21b)/den_S_run_21b

    mc_err_2_1b = np.var(g_x_21b)/ntrials
    left_ci_2_1b = prob_21b - 2*((mc_err_2_1b)**(1/2))
    right_ci_2_1b = prob_21b + 2*((mc_err_2_1b)**(1/2))

    if Anex_prob_10 >= left_ci_2_1b and Anex_prob_10 <= right_ci_2_1b:
        count_21b += 1
    else:
        count_21b += 0

Sn_running_mean_21b = np.sum(Sn_running_21b[:,-1])/replicates
Sn_running_mean_21b_sorted = np.sort(Sn_running_21b[:,-1])
right_i = Sn_running_mean_21b_sorted[975] - Sn_running_mean_21b
left_i = Sn_running_mean_21b - Sn_running_mean_21b_sorted[25]
print(right_i)
print(left_i)
```

Problem 2.2 a

In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

In [2]:
```
import numpy as np
import sympy as sympy # a compute algebra system in python
import math
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import matplotlib
font = {'size'   : 14}

matplotlib.rc('font', **font)
```

Problem 2.2 b

In [6]:
```
nsamples = 100
mu = 0
sigma = 1
n_mc_trials = 10**5
x1 = np.zeros(nsamples)
cumsum_x1 = np.zeros((n_mc_trials, nsamples))
x2 = np.zeros(nsamples)
cumsum_x2 = np.zeros((n_mc_trials, nsamples))
x3 = np.zeros(nsamples)
cumsum_x3 = np.zeros((n_mc_trials, nsamples))
s1 = np.zeros(n_mc_trials)
s2 = np.zeros(n_mc_trials)
s3 = np.zeros(n_mc_trials)
g_x = np.zeros(n_mc_trials)

for ii in range(n_mc_trials):

    x1 = np.random.normal(mu, sigma, nsamples)
    x2 = np.random.normal(mu, sigma, nsamples)
    x3 = np.random.normal(mu, sigma, nsamples)

    cumsum_x1[ii,:] = np.cumsum(x1)
    cumsum_x2[ii,:] = np.cumsum(x2)
    cumsum_x3[ii,:] = np.cumsum(x3)

    s1[ii] = cumsum_x1[ii,-1]
    s2[ii] = cumsum_x2[ii,-1]
    s3[ii] = cumsum_x3[ii,-1]

    s = (s1**2 + s2**2 + s3**2)**(1/2)
    g_x[ii] = s[ii] > 10

count_2_2_b = np.count_nonzero(s > 10)
prob_2_2_b = count_2_2_b/n_mc_trials
print(prob_2_2_b)
print(g_x)
```

```
0.80301
[1. 1. 1. ... 0. 0. 1.]
```

Problem 2.2 d

In [7]:
```
# Estimate Monte Carlo Error in estimate for P(S>10)
mc_error = np.var(g_x)/n_mc_trials
z = 1.96 #for 95% confidence interval
left_ci = prob_2_2_b + z*((mc_error)**(1/2))
right_ci = prob_2_2_b - z*((mc_error)**(1/2))
print(mc_error)
print(left_ci)
print(right_ci)
```

```
1.5818493990000002e-06
0.8054751232527398
0.8005448767472602
```

In [ ]:
```
```

Problem 4.1 (1a)

In [1]:
```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt
```

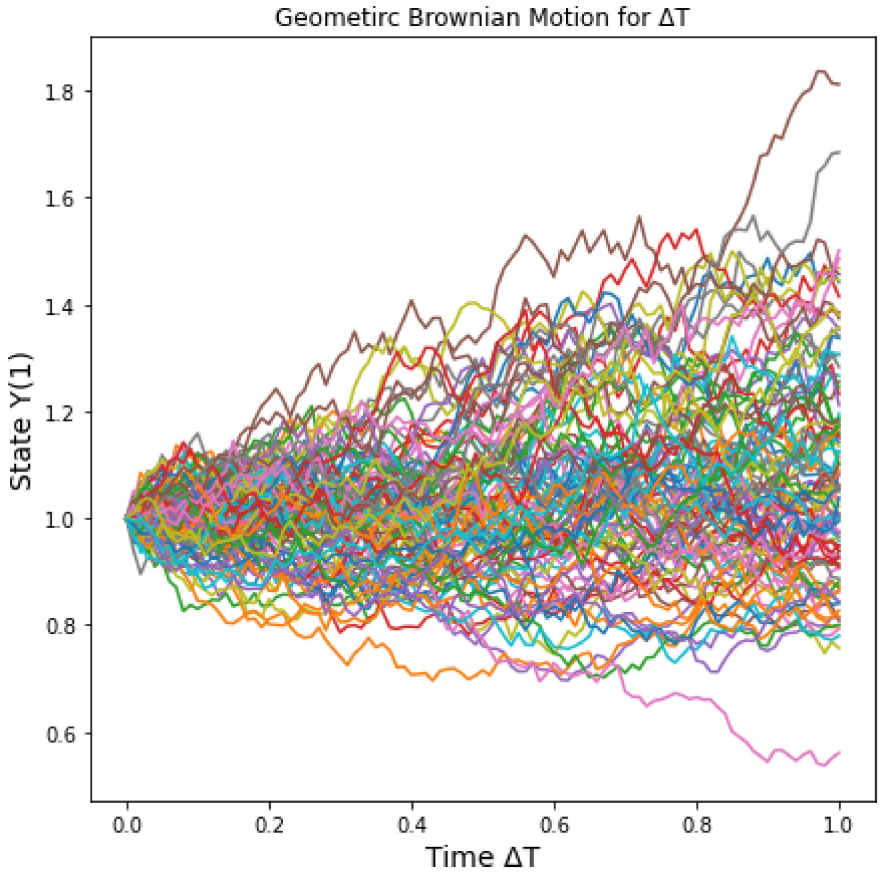
In [2]:
```
def brownian_motion_simulate(T, dt, num_paths):
    nsamples = int(np.floor(T / dt)) + 1
    samples = np.random.randn(nsamples, num_paths) * np.sqrt(dt)
    samples[0] = 0.0
    bmotion = np.cumsum(samples, axis=0)
    return bmotion
```

In [3]:
```
def brownian_motion_fine_to_coarse(T, dt, brownian_fine, M):
    nsamples = int(np.ceil(T/dt)) + 1
    npaths = brownian_fine.shape[1]
    brownian_coarse = np.zeros((nsamples, npaths))
    brownian_coarse[0, :] = brownian_fine[0, :]
    for ii in range(1, nsamples):
        delta = brownian_fine[ii * M, :] - brownian_fine[(ii-1)*M, :]
        brownian_coarse[ii, :] = brownian_coarse[ii-1, :] + delta
    return brownian_coarse
```

In [4]:
```
_maruyama(b, h, x0, dt, T, bmotion):

aths = x0.shape[0]
t bmotion.shape[1] >= num_paths, "Not enough brownian motions simulated"
teps = int(np.ceil(T / dt)) + 1
 = np.zeros((num_steps, num_paths))
[0, :] = x0
 = np.zeros(num_steps)

i in range(1, num_steps):
aths[ii, :] = paths[ii-1, :] + b(t, paths[ii-1, :]) * dt + h(t, paths[ii-1, :]) *
 += dt
imes[ii] = t;
n times, paths
```

In [5]:
```
mu = 0.05
sigma = 0.2
b = lambda t, y: mu * y
h = lambda t, y: sigma * y
```

In [6]:
```
T = 1
dt = 1e-2
num_paths = 100
bmotion_411a = brownian_motion_simulate(T, dt, num_paths)
#print(bmotion)
y0 = np.ones((num_paths)) # start at 1
t_fin_411a, y_states_411a = euler_maruyama(b, h, y0, dt, T, bmotion_411a)

plt.figure(figsize=(7, 7))
plt.plot(t_fin_411a, y_states_411a)
plt.xlabel('Time \u0394T', fontsize=14)
plt.ylabel('State Y(1)', fontsize=14)
plt.title('Geometirc Brownian Motion for \u0394T')
plt.show()

y_f_411a = y_states_411a[:,-1]
bmotion_f_411a = bmotion_411a[:,-1]
mean_y_411a = sum(y_f_411a)/num_paths
var_y_411a = np.var(y_f_411a)
print(mean_y_411a)
print(var_y_411a)
#print(t_fin_411a)
#print(y_states_411a)
```




```
1.0238661751968923
```

```
0.009709714340272223
```

In [7]:
```
# Analytical solution
Y0_411a = 1
t_f = 1
Y_an_f_411a = Y0_411a * np.exp((mu - (sigma**2/2))*t_f + (sigma*bmotion_f_411a))
mean_y_an_411a = sum(Y_an_f_411a)/num_paths
var_y_an_411a = np.var(Y_an_f_411a)
print(mean_y_an_411a)
print(var_y_an_411a)
```

```
1.0372263407547058
0.008558243015729479
```

Problem 4.1 (1b)

In [8]:
```
T = 1
dt = 1e-2
num_paths = 1000
mc_runs = 1000
b = lambda t, y: mu * y
h = lambda t, y: sigma * y
y0 = np.ones((num_paths))
#bmotion_411b_matrix = np.zeros((mc_runs,num_paths))
S_f_states_411b = np.zeros(mc_runs)

for ii in range(mc_runs):
    bmotion_411b = brownian_motion_simulate(T, dt, num_paths)
    t_fin_411b, y_states_411b = euler_maruyama(b, h, y0, dt, T, bmotion_411b)
    y_f_states_411b = y_states_411b[-1,:]
    S_f_states_411b[ii] = np.sum(y_f_states_411b)/num_paths
# mean_411b = np.sum(S_f_states_411b)/mc_runs
var_411b = np.var(S_f_states_411b)

#print(mean_411b)
#print(var_411b)
#print(y_states_411b.shape)
#print(y_f_states_411b.shape)
#print(S_f_states_411b)
print(var_411b)
print(S_f_states_411b)
```

```
4.303662703232498e-05
[1.04391562 1.05154072 1.06146682 1.05506002 1.04947188 1.04520891
 1.05215792 1.0560015  1.04177639 1.04365951 1.03556214 1.04554734
 1.05805716 1.05822806 1.04471034 1.05586706 1.05067621 1.04734315
 1.06231439 1.07192899 1.04532831 1.04565521 1.06142277 1.06139101
 1.0512472  1.05017918 1.04242922 1.05963215 1.05260205 1.05329639
 1.05698214 1.05051854 1.04630787 1.05456202 1.06251699 1.05428613
 1.06113019 1.05638916 1.06005352 1.05843631 1.04753028 1.04641131
 1.04950299 1.04501033 1.04864731 1.05665419 1.05308791 1.04890238
 1.05066302 1.06033806 1.04793104 1.06145698 1.05832504 1.05776129
 1.04557382 1.05122159 1.04364154 1.05224498 1.05007527 1.05833618
 1.05457046 1.0519642  1.05554165 1.05524832 1.05354972 1.04268741
 1.05521439 1.04750168 1.04453175 1.06864331 1.07228702 1.04988047
 1.05157892 1.0420977  1.04626384 1.05189659 1.04844983 1.0575299
 1.04758715 1.06000514 1.06220501 1.04385241 1.0600253  1.04958578
 1.04858257 1.05353656 1.04066095 1.04922905 1.0601525  1.05287723
 1.04165309 1.05519368 1.0421505  1.0511801  1.06001233 1.05021836
 1.05411525 1.0418946  1.05265323 1.04479496 1.05129553 1.04771539
 1.06704339 1.04458255 1.04558781 1.05266146 1.06826688 1.0420083
```

Problem 4.1 (2)

In [9]:
```
T = 1
dt = 1e-2
num_paths = 100
b = lambda t, y: mu * y
h = lambda t, y: sigma * y
y0 = np.ones((num_paths))
bmotion_dt_412 = brownian_motion_simulate(T, dt, num_paths)
bmotion_4dt_412 = bmotion_dt_412[::4,:]
t_fin_412a, y_states_412a = euler_maruyama(b, h, y0, dt, T, bmotion_dt_412)
# t_fin_412a_last = t_fin_412a[-1,:]
bmotion_dt_412_last = bmotion_dt_412[-1,:]
t_fin_412b, y_states_412b = euler_maruyama(b, h, y0, 4*dt, T, bmotion_4dt_412)
# t_fin_412b_last = t_fin_412b[-1,:]
bmotion_4dt_412_last = bmotion_dt_412[-1,:]
# fig, ax = plt.subplots(2,figsize=(11, 11))
# ax[0].plot(t_fin_412a, y_states_412a)
# ax[1].plot(t_fin_412b, y_states_412b)
# ax[0].set_xlabel('Time', fontsize=12)
# ax[0].set_ylabel('State Y(1)', fontsize=12)
# ax[0].set_title("Geometric Brownian Motion for \u0394T")
# ax[1].set_xlabel('Time', fontsize=12)
# ax[1].set_ylabel('State Y(1)', fontsize=12)
# ax[1].set_title("Geometric Brownian Motion for 4\u0394T")
# plt.show()
# print(t_fin_412a)
# print(bmotion_dt_412[-1,:])
# print(t_fin_412b.shape)
# print(bmotion_4dt_412_last.shape)
# np.reshape(t_fin_412b, (100,1))
```

In [10]:
```
# M = 4
# DT = 0.025  # fine scale time step
# DTM = DT * M # coarse scale time step
# TFINAL = 1.0 # final time
# TSPAN_COARSE = np.arange(0, TFINAL + DTM, DTM)
# TSPAN_FINE = np.arange(0, TFINAL+DT, DT)
# brownian_fine = brownian_motion_simulate(TFINAL, DT)
# brownian_coarse = brownian_motion_fine_to_coarse(TFINAL, DTM, brownian_fine, M)

# plt.figure()
# plt.plot(t_fin_412a[0:100], bmotion_dt_412_last, '-ko', label='Fine-scale simul
# plt.plot(t_fin_412b[0:100], bmotion_4dt_412_last, '--rx', label='Coarse-scale s

# plt.xlabel('Time', fontsize=14)
# plt.ylabel(r'$W_t$', fontsize=14)
# plt.legend(fontsize=14)
# plt.show()
```

Problem 4.1 (3a)

In [11]:
```
ou_bfunc = lambda t, y: mu * y
ou_hfunc = lambda t, y: sigma * y

def evaluator(bmotion, dt, T):

    num_paths = bmotion.shape[1]
    x0 = np.ones((num_paths)) # start at 1
    times, states = euler_maruyama(ou_bfunc, ou_hfunc, x0, dt, T, bmotion)
    res = states[-1,:]
    res = res[:, np.newaxis] # make this an N x 1 array, which is what we need fo
    return res
```

In [12]:
```
def multilevel_montecarlo_sde(func, nsamples_L0, nsamples_L1, nsamples_L2, nsampl
    """Multilevel Monte Carlo for Stochastic differential equations"""

    dt_L3_fine = dt_L4_fine * M
    dt_L2_fine = dt_L3_fine * M
    dt_L1_fine = dt_L2_fine * M
    dt_coarse = dt_L1_fine * M

    ############################
    # level 0 Only coarse grid evaluations are used
    ############################
    bmotion_L0 = brownian_motion_simulate(T, dt_coarse, nsamples_L0)
    samples_L0 = func(bmotion_L0, dt_coarse, T)

    ######################
    # level 1
    ######################
    # First generate fine and coarse bmotion samples
    bmotion_L1_fine = brownian_motion_simulate(T, dt_L1_fine, nsamples_L1)
    bmotion_L1_coarse = brownian_motion_fine_to_coarse(T, dt_coarse, bmotion_L1_f

    # Next evaluate the samples
    samples_L1_fine = func(bmotion_L1_fine, dt_L1_fine, T)
    samples_L1_coarse = func(bmotion_L1_coarse, dt_coarse, T)

    # Now subtract
    samples_L1_del = samples_L1_fine - samples_L1_coarse

    ######################
    # level 2
    ######################
    # First generate fine and coarse bmotion samples
    bmotion_L2_fine = brownian_motion_simulate(T, dt_L2_fine, nsamples_L2)
    bmotion_L2_coarse = brownian_motion_fine_to_coarse(T, dt_L1_fine, bmotion_L2_

    # Next evaluate the samples
    samples_L2_fine = func(bmotion_L2_fine, dt_L2_fine, T)
    samples_L2_coarse = func(bmotion_L2_coarse, dt_L1_fine, T)

    # Now subtract
    samples_L2_del = samples_L2_fine - samples_L2_coarse
    ######################
    # level 3
    ######################
    # First generate fine and coarse bmotion samples
    bmotion_L3_fine = brownian_motion_simulate(T, dt_L3_fine, nsamples_L3)
    bmotion_L3_coarse = brownian_motion_fine_to_coarse(T, dt_L2_fine, bmotion_L3_

    # Next evaluate the samples
    samples_L3_fine = func(bmotion_L3_fine, dt_L3_fine, T)
    samples_L3_coarse = func(bmotion_L3_coarse, dt_L2_fine, T)

    # Now subtract
    samples_L3_del = samples_L3_fine - samples_L3_coarse
        ######################
    # level 4
```

```
    ######################
    # First generate fine and coarse bmotion samples
    bmotion_L4_fine = brownian_motion_simulate(T, dt_L4_fine, nsamples_L4)
    bmotion_L4_coarse = brownian_motion_fine_to_coarse(T, dt_L3_fine, bmotion_L4_

    # Next evaluate the samples
    samples_L4_fine = func(bmotion_L4_fine, dt_L4_fine, T)
    samples_L4_coarse = func(bmotion_L4_coarse, dt_L3_fine, T)

    # Now subtract
    samples_L4_del = samples_L4_fine - samples_L4_coarse

    ######################
    ## Combine levels
    ######################
    est_mean_L0 = np.mean(samples_L0)
    est_mean_L1 = np.mean(samples_L1_fine)
    est_mean_L2 = np.mean(samples_L2_fine)
    est_mean_L3 = np.mean(samples_L3_fine)
    est_mean_L4 = np.mean(samples_L4_fine)

    est_mean_Y_1 = est_mean_L0 + est_mean_L1 + est_mean_L2 + est_mean_L3 + est_me

    est_var_L0 = np.var(samples_L0)
    est_var_L1 = np.var(samples_L1_fine)
    est_var_L2 = np.var(samples_L2_fine)
    est_var_L3 = np.var(samples_L3_fine)
    est_var_L4 = np.var(samples_L4_fine)

    est_var_del_1 = np.var(samples_L1_del)
    est_var_del_2 = np.var(samples_L2_del)
    est_var_del_3 = np.var(samples_L3_del)
    est_var_del_4 = np.var(samples_L4_del)

    return est_mean_Y_1, est_mean_L0, est_mean_L1, est_mean_L2, est_mean_L3, est_
```

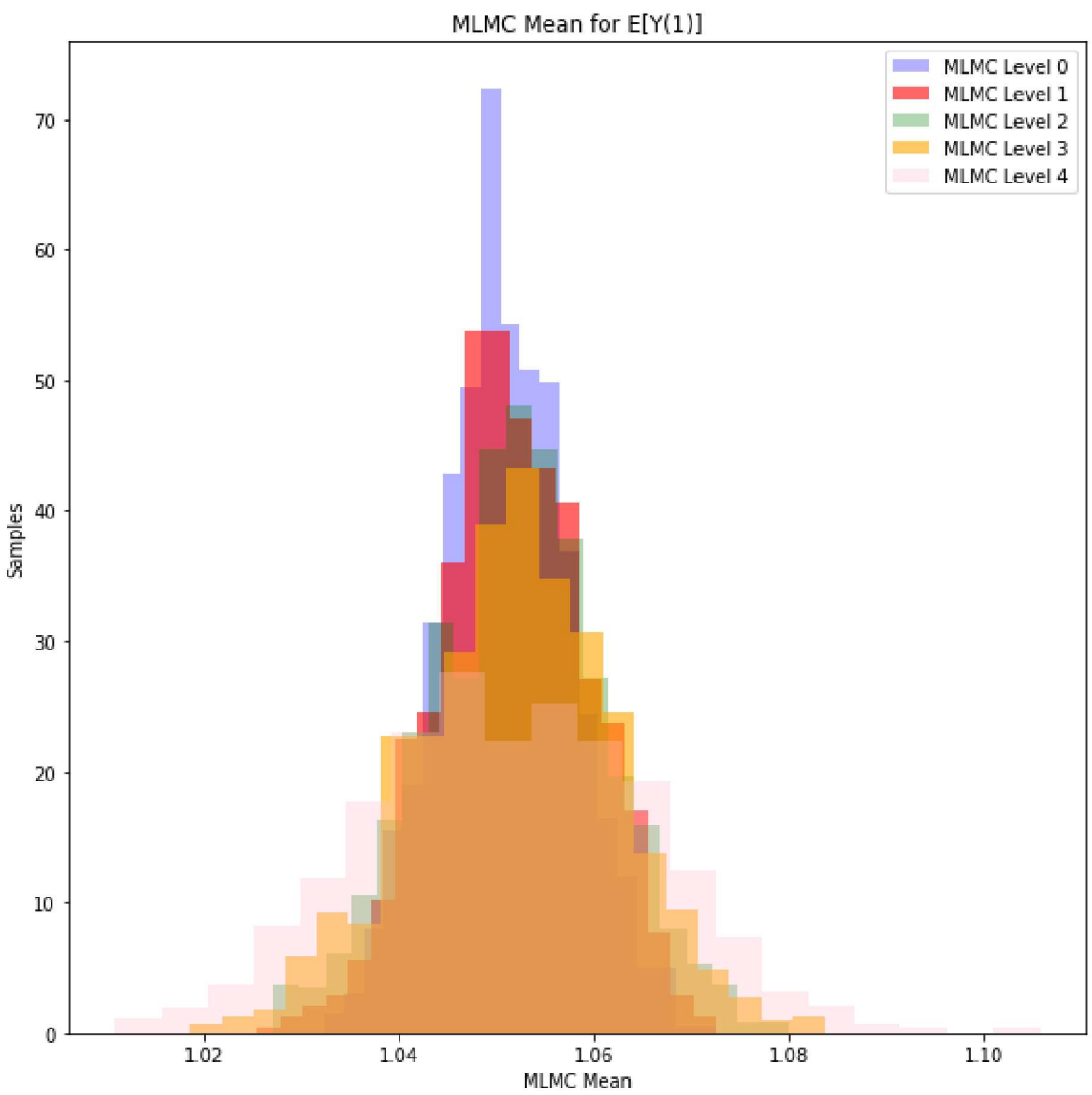
Problem 4.1 (3b)

In [13]:
```
ou_bfunc = lambda t, y: mu * y
ou_hfunc = lambda t, y: sigma * y
num_paths = 1000
def p_evaluator(bmotion, dt, T):

    num_paths = bmotion.shape[1]
    p = np.zeros((num_paths))
    x0 = np.ones((num_paths)) # start at 1
    times, states = euler_maruyama(ou_bfunc, ou_hfunc, x0, dt, T, bmotion)
    for ii in range((num_paths)):
        p[ii] = np.exp(-0.05)*np.maximum(0,states[-1,ii]-1)
    res = p[:, np.newaxis] # make this an N x 1 array, which is what we need for
    return res
```

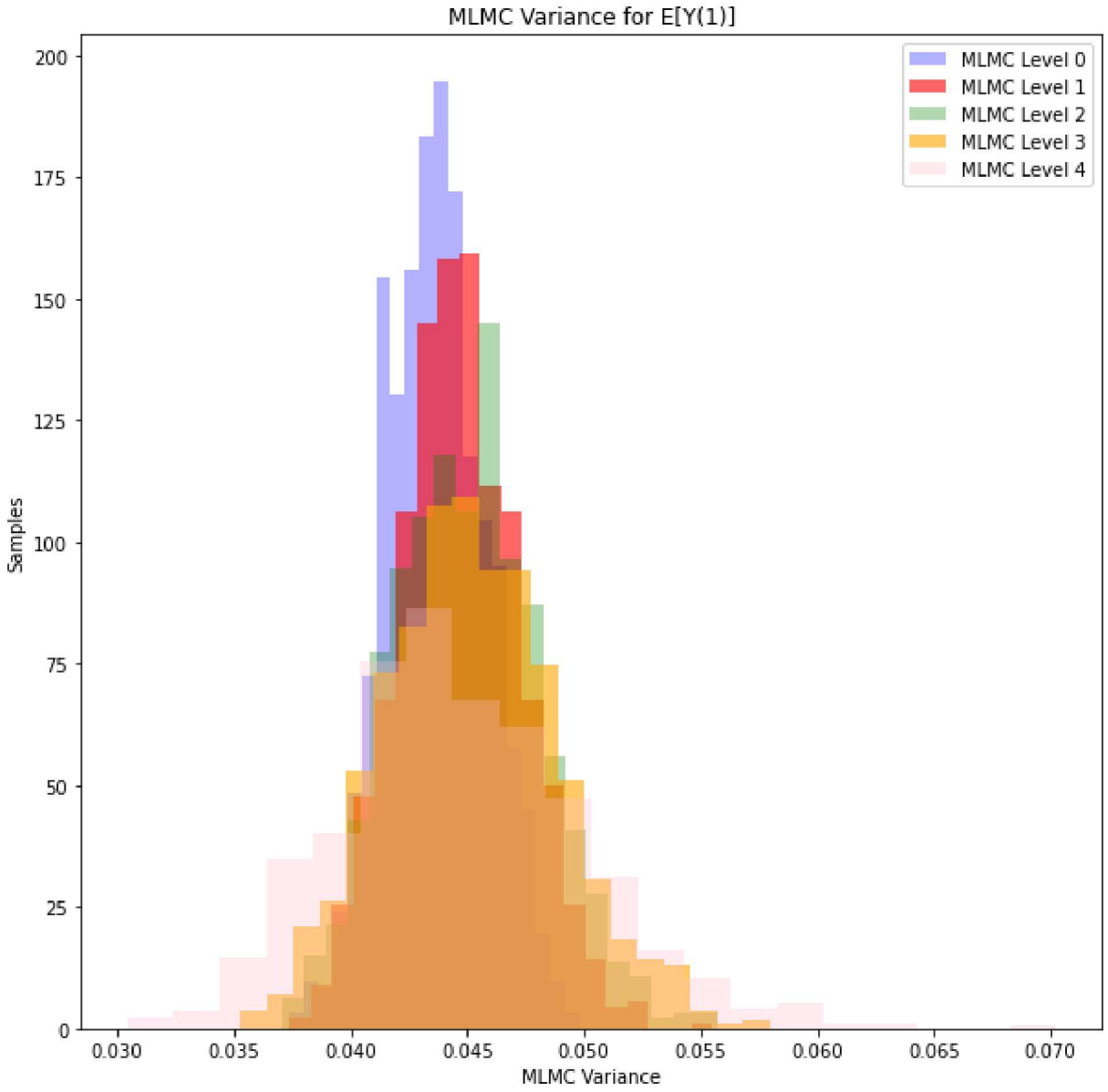
In [14]:
```
dt_fine_L4 = 1/4**(5)
M = 4
T = 1
nsamples_0 = 1000 #level 0
nsamples_1 = 800 #level 1
nsamples_2 = 600 #level 2
nsamples_3 = 400 #level 3
nsamples_4 = 200 #level 4
est_mean_of_peval,_,_,_,_,_,_,_,_,_,_,_,_,_,_ = multilevel_montecarlo_sde(evaluat
est_mean_of_eval,_,_,_,_,_,_,_,_,_,_,_,_,_,_ = multilevel_montecarlo_sde(p_evalua
print(est_mean_of_peval)
print(est_mean_of_eval)
```

```
5.249000743350909
0.5341834985356338
```

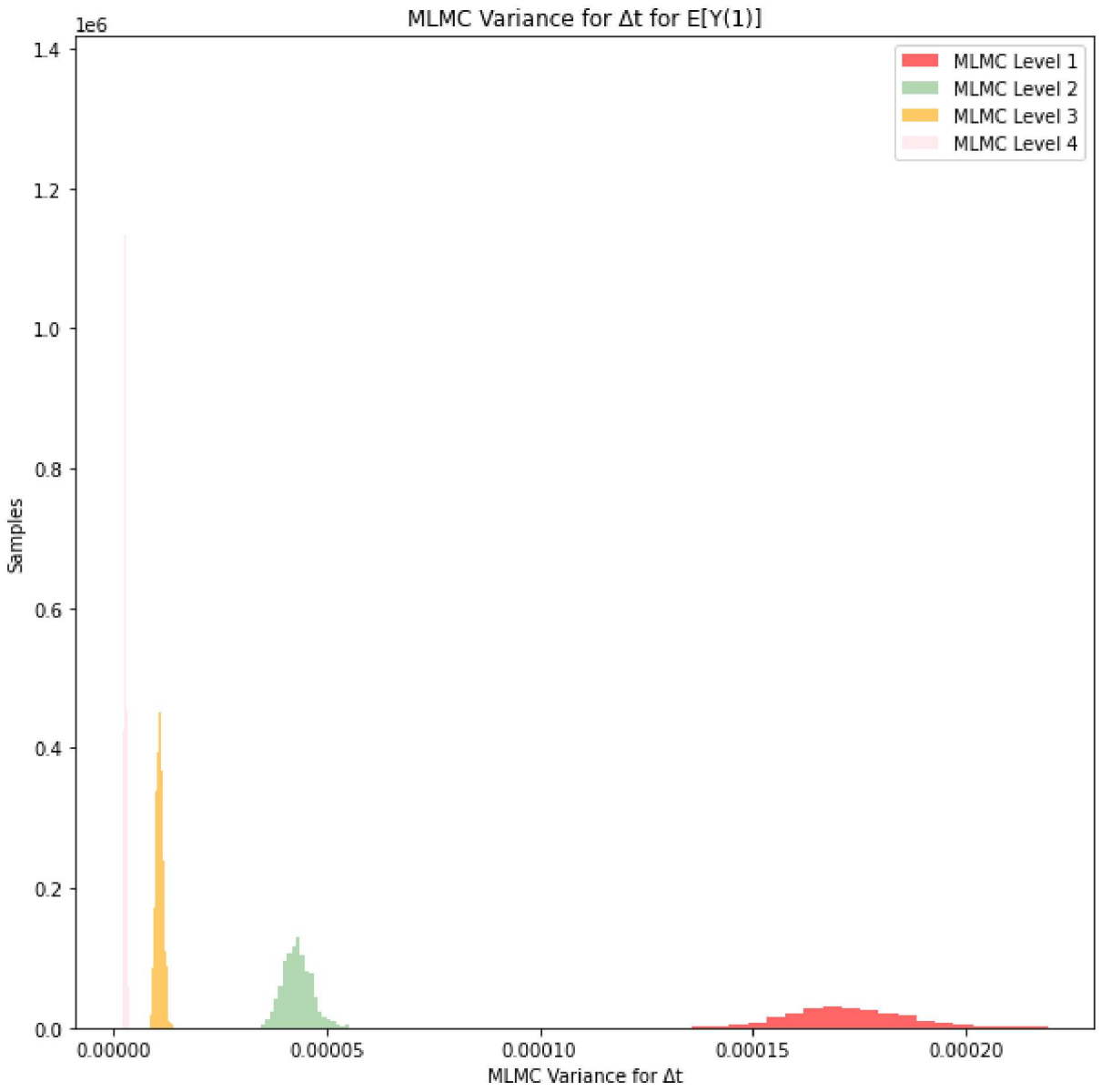
In [15]:
```
ntrials = 1000

L0_mean_Y = np.zeros((ntrials))
L1_mean_Y = np.zeros((ntrials))
L2_mean_Y = np.zeros((ntrials))
L3_mean_Y = np.zeros((ntrials))
L4_mean_Y = np.zeros((ntrials))

L0_var_Y = np.zeros((ntrials))
L1_var_Y = np.zeros((ntrials))
L2_var_Y = np.zeros((ntrials))
L3_var_Y = np.zeros((ntrials))
L4_var_Y = np.zeros((ntrials))

L1_var_del_Y = np.zeros((ntrials))
L2_var_del_Y = np.zeros((ntrials))
L3_var_del_Y = np.zeros((ntrials))
L4_var_del_Y = np.zeros((ntrials))

for ii in range(ntrials):
    mean_p_y, mean_L0_y, mean_L1_y, mean_L2_y, mean_L3_y, mean_L4_y, var_L0_y, va
    var_L2_y, var_L3_y, var_L4_y, var_del_L1_y, var_del_L2_y, var_del_L3_y, var_d
    = multilevel_montecarlo_sde(evaluator,nsamples_0, nsamples_1, nsamples_2,\
    nsamples_3, nsamples_4, dt_fine_L4, T, M)

    L0_mean_Y[ii] = mean_L0_y
    L1_mean_Y[ii] = mean_L1_y
    L2_mean_Y[ii] = mean_L2_y
    L3_mean_Y[ii] = mean_L3_y
    L4_mean_Y[ii] = mean_L4_y

    L0_var_Y[ii] = var_L0_y
    L1_var_Y[ii] = var_L1_y
    L2_var_Y[ii] = var_L2_y
    L3_var_Y[ii] = var_L3_y
    L4_var_Y[ii] = var_L4_y

    L1_var_del_Y[ii] = var_del_L1_y
    L2_var_del_Y[ii] = var_del_L2_y
    L3_var_del_Y[ii] = var_del_L3_y
    L4_var_del_Y[ii] = var_del_L4_y

plt.figure(figsize=(10,10))
plt.hist(L0_mean_Y, density=True, color='blue', label='MLMC Level 0',alpha=0.3, b
plt.hist(L1_mean_Y, density=True, color='red', label='MLMC Level 1',alpha=0.6, bi
plt.hist(L2_mean_Y, density=True, color='green', label='MLMC Level 2',alpha=0.3,
plt.hist(L3_mean_Y, density=True, color='orange', label='MLMC Level 3',alpha=0.6,
plt.hist(L4_mean_Y, density=True, color='pink', label='MLMC Level 4',alpha=0.3, b
plt.xlabel('MLMC Mean')
plt.ylabel('Samples')
plt.title('MLMC Mean for E[Y(1)]')
plt.legend()
plt.show()
```


MLMC Mean for E[Y(1)]
MLMC Level 0
MLMC Level 1
MLMC Level 2
MLMC Level 3
MLMC Level 4
Samples
0
10
20
30
40
50
60
70
1.02
1.04
1.06
1.08
1.10
MLMC Mean

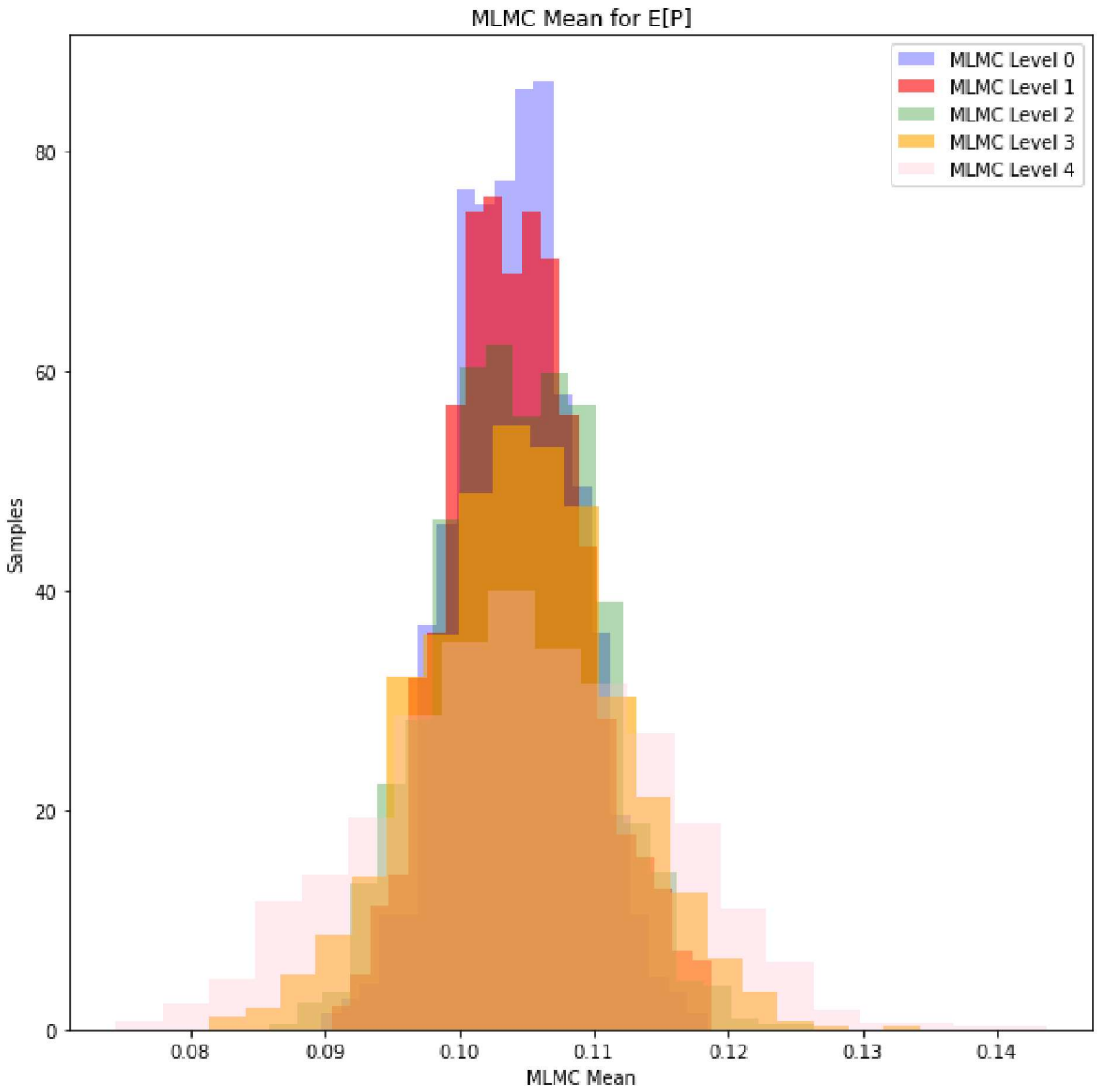
In [16]:
```
plt.figure(figsize=(10,10))
plt.hist(L0_var_Y, density=True, color='blue', label='MLMC Level 0',alpha=0.3, bi
plt.hist(L1_var_Y, density=True, color='red', label='MLMC Level 1',alpha=0.6, bir
plt.hist(L2_var_Y, density=True, color='green', label='MLMC Level 2',alpha=0.3, b
plt.hist(L3_var_Y, density=True, color='orange', label='MLMC Level 3',alpha=0.6,
plt.hist(L4_var_Y, density=True, color='pink', label='MLMC Level 4',alpha=0.3, bi
plt.xlabel('MLMC Variance')
plt.ylabel('Samples')
plt.title('MLMC Variance for E[Y(1)]')
plt.legend()
plt.show()
```



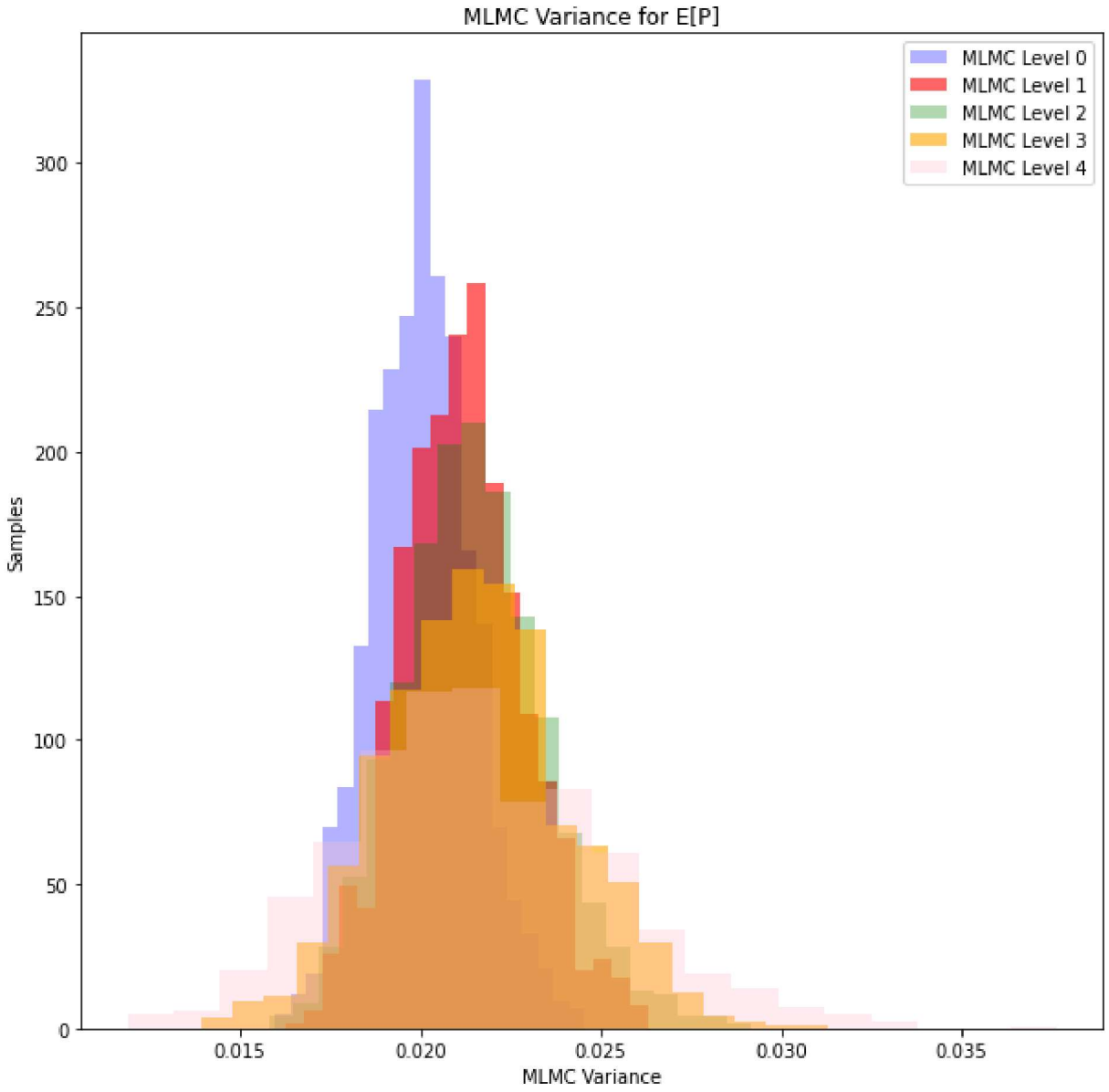
In [17]:
```
plt.figure(figsize=(10,10))
plt.hist(L1_var_del_Y, density=True, color='red', label='MLMC Level 1',alpha=0.6,
plt.hist(L2_var_del_Y, density=True, color='green', label='MLMC Level 2',alpha=0.
plt.hist(L3_var_del_Y, density=True, color='orange', label='MLMC Level 3',alpha=0
plt.hist(L4_var_del_Y, density=True, color='pink', label='MLMC Level 4',alpha=0.3
plt.xlabel('MLMC Variance for \u0394t')
plt.ylabel('Samples')
plt.title('MLMC Variance for \u0394t for E[Y(1)]')
plt.legend()
plt.show()
```



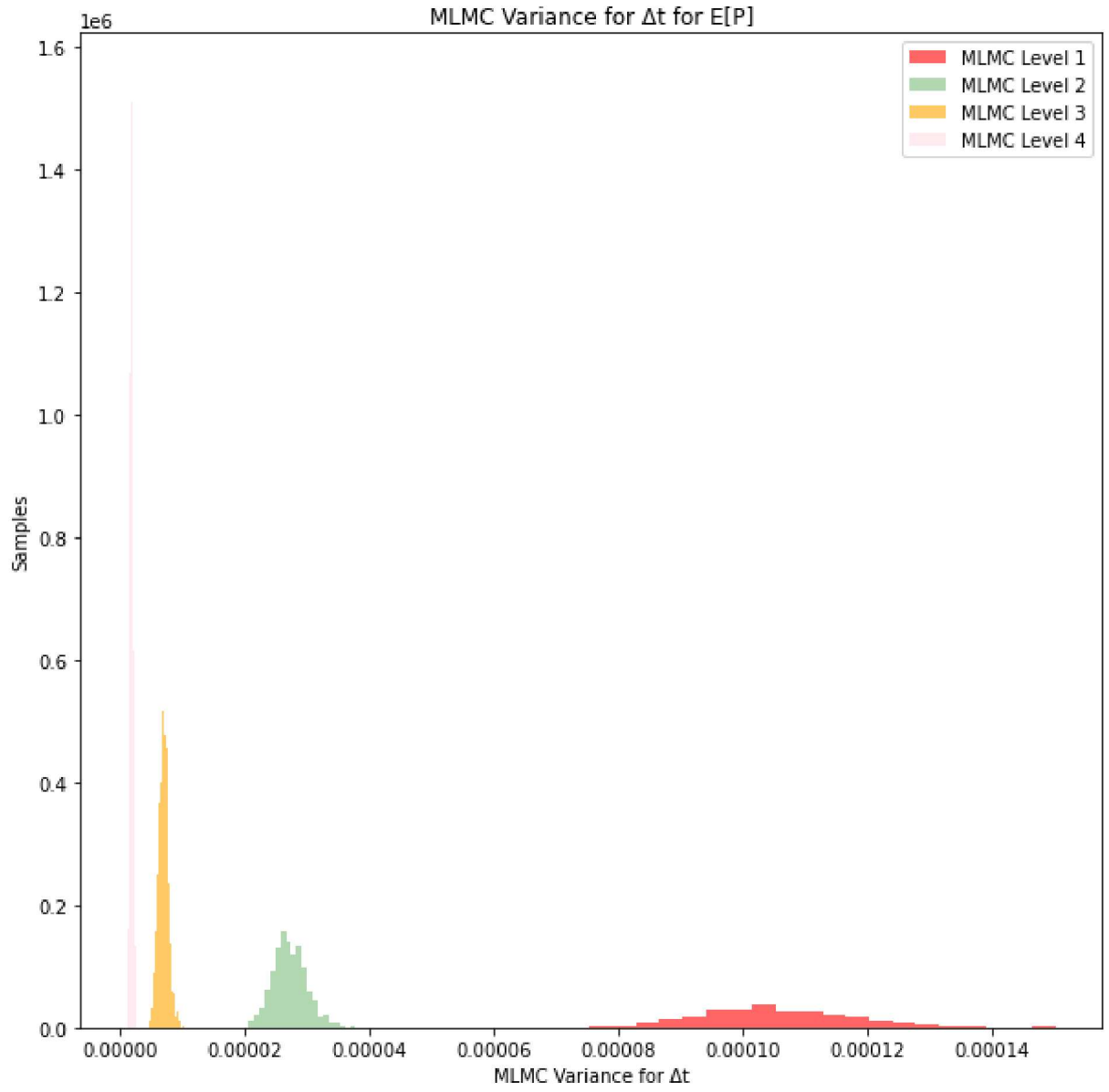
In [18]:
```
ntrials = 1000

L0_mean_p = np.zeros((ntrials))
L1_mean_p = np.zeros((ntrials))
L2_mean_p = np.zeros((ntrials))
L3_mean_p = np.zeros((ntrials))
L4_mean_p = np.zeros((ntrials))

L0_var_p = np.zeros((ntrials))
L1_var_p = np.zeros((ntrials))
L2_var_p = np.zeros((ntrials))
L3_var_p = np.zeros((ntrials))
L4_var_p = np.zeros((ntrials))

L1_var_del_p = np.zeros((ntrials))
L2_var_del_p = np.zeros((ntrials))
L3_var_del_p = np.zeros((ntrials))
L4_var_del_p = np.zeros((ntrials))

for ii in range(ntrials):
    mean_p_p, mean_L0_p, mean_L1_p, mean_L2_p, mean_L3_p, mean_L4_p, var_L0_p, va
    var_L2_p, var_L3_p, var_L4_p, var_del_L1_p, var_del_L2_p, var_del_L3_p, var_d
    = multilevel_montecarlo_sde(p_evaluator,nsamples_0, nsamples_1, nsamples_2,\
    nsamples_3, nsamples_4, dt_fine_L4, T, M)

    L0_mean_p[ii] = mean_L0_p
    L1_mean_p[ii] = mean_L1_p
    L2_mean_p[ii] = mean_L2_p
    L3_mean_p[ii] = mean_L3_p
    L4_mean_p[ii] = mean_L4_p

    L0_var_p[ii] = var_L0_p
    L1_var_p[ii] = var_L1_p
    L2_var_p[ii] = var_L2_p
    L3_var_p[ii] = var_L3_p
    L4_var_p[ii] = var_L4_p

    L1_var_del_p[ii] = var_del_L1_p
    L2_var_del_p[ii] = var_del_L2_p
    L3_var_del_p[ii] = var_del_L3_p
    L4_var_del_p[ii] = var_del_L4_p

plt.figure(figsize=(10,10))
plt.hist(L0_mean_p, density=True, color='blue', label='MLMC Level 0',alpha=0.3, b
plt.hist(L1_mean_p, density=True, color='red', label='MLMC Level 1',alpha=0.6, bi
plt.hist(L2_mean_p, density=True, color='green', label='MLMC Level 2',alpha=0.3,
plt.hist(L3_mean_p, density=True, color='orange', label='MLMC Level 3',alpha=0.6,
plt.hist(L4_mean_p, density=True, color='pink', label='MLMC Level 4',alpha=0.3, b
plt.xlabel('MLMC Mean')
plt.ylabel('Samples')
plt.title('MLMC Mean for E[P]')
plt.legend()
plt.show()
```


MLMC Mean for E[P]
MLMC Level 0
MLMC Level 1
MLMC Level 2
MLMC Level 3
MLMC Level 4
Samples
0
20
40
60
80
0.08
0.09
0.10
0.11
0.12
0.13
0.14
MLMC Mean

In [19]:
```
plt.figure(figsize=(10,10))
plt.hist(L0_var_p, density=True, color='blue', label='MLMC Level 0',alpha=0.3, bi
plt.hist(L1_var_p, density=True, color='red', label='MLMC Level 1',alpha=0.6, bir
plt.hist(L2_var_p, density=True, color='green', label='MLMC Level 2',alpha=0.3, b
plt.hist(L3_var_p, density=True, color='orange', label='MLMC Level 3',alpha=0.6,
plt.hist(L4_var_p, density=True, color='pink', label='MLMC Level 4',alpha=0.3, bi
plt.xlabel('MLMC Variance')
plt.ylabel('Samples')
plt.title('MLMC Variance for E[P]')
plt.legend()
plt.show()
```



In [20]:
```
plt.figure(figsize=(10,10))
plt.hist(L1_var_del_p, density=True, color='red', label='MLMC Level 1',alpha=0.6,
plt.hist(L2_var_del_p, density=True, color='green', label='MLMC Level 2',alpha=0.
plt.hist(L3_var_del_p, density=True, color='orange', label='MLMC Level 3',alpha=0
plt.hist(L4_var_del_p, density=True, color='pink', label='MLMC Level 4',alpha=0.3
plt.xlabel('MLMC Variance for \u0394t')
plt.ylabel('Samples')
plt.title('MLMC Variance for \u0394t for E[P]')
plt.legend()
plt.show()
```



In [ ]: