AEROSP 567
Project 1

Hardik Parwana, hardiksp@umich.edu, UMID: 536 4805

Note: The code is included in the Appendix in sequential order for Questions 2,3,4. The code is well labelled and therefore explicit references in the report with page numbers are not given.

2 Warmup: life without a CLT

A monte carlo estimator of mean of random variable X is written. A custom sampler for Pareto distribution is written using Inverse CDF sampler method. The CDF of Pareto is given by

$$CDF = \int_1^x \frac{\alpha}{x^{\alpha+1}} = 1 - \frac{1}{x^\alpha} \tag{1}$$

Fig.1 shows the MC estimation as function of sample size n. The estimation distribution seems to

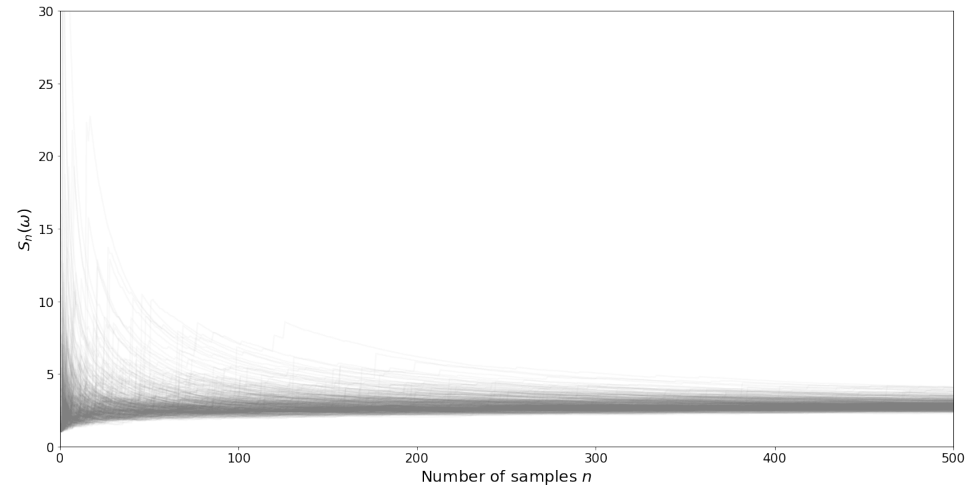


Figure 1: Mean Estimation with MC

converge for large values of n but has high variance (converges not to a point but to set of points). This suggests that there is no asymptotic convergence to the true mean.

In order to discern the convergence rates, strong(asymptotic convergence) and weak(convergence in probability) convergence properties are investigated. Suppose the true mean is μ ($\mu = 3$ for $\alpha = 1.5$), and estimated mean is $\hat{\mu}$. Then defining event A as

$$A = \{\hat{\mu} \mid \; |\mu - \hat{\mu}| < \epsilon\} \tag{2}$$

where ϵ is a user decided threshold. Here the true mean μ is obtained analytically to be $\mu = \alpha(\alpha - 1), \alpha >= 1$ .Numerical estimation is done for above two properties in similar fashion as done during lectures and is plotted as function of number of samples in Fig. 2.

The MC mean is observed to be $\mu = 2.83$.

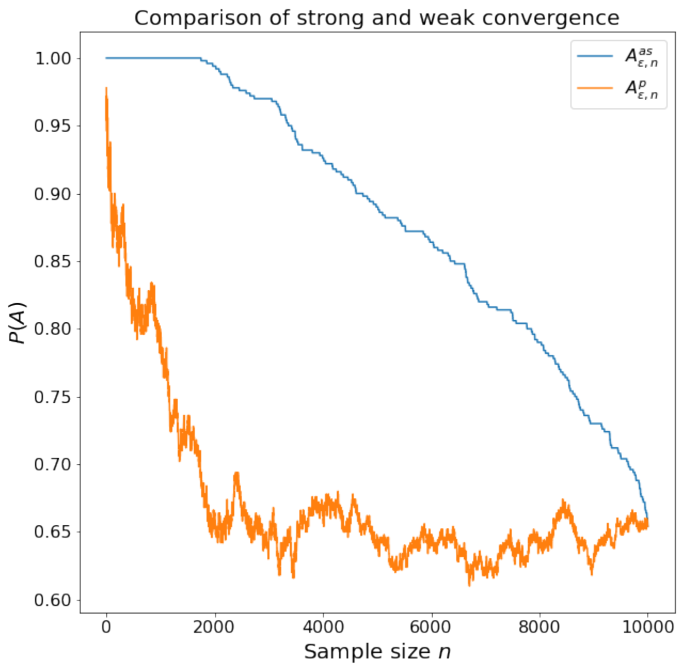


Figure 2: Convergence properties for $\epsilon = 0.1$

Importance sampling for random walks

2.1 (a)

Inverse CDF sampler is written for random walk. Fig.3 shows the visualization for 500 trials as a function of number of steps n upto $n = 100$. The figure is symmetric about X axis which suggests that the mean is always 0 whereas the variance keeps increasing with n.

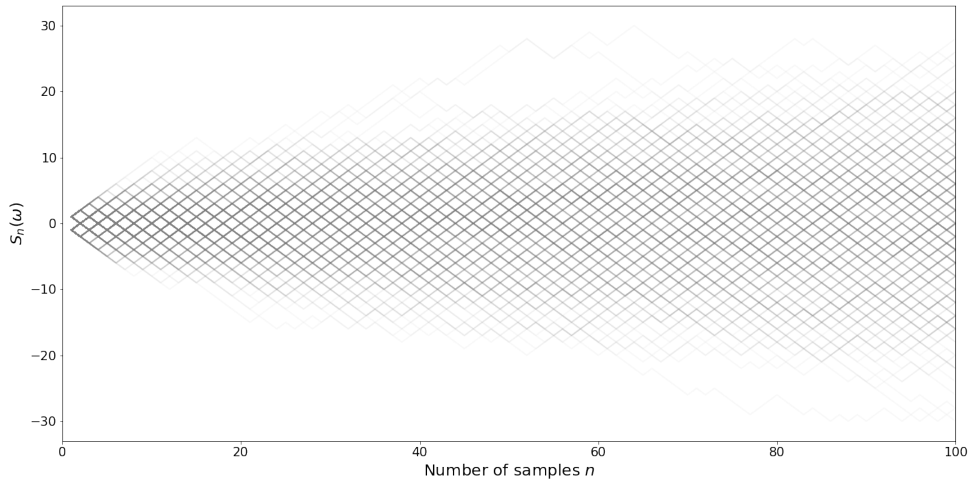


Figure 3: Visualization of N-step random walk

2.1(b)

A simple monte carlo estimator is written to estimate $\mathbb{P}(S > 10)$. This is obtained as fraction of paths out of 10^5 that satisfy the condition that final position is greater than 10.

$$\mathbb{P}(S > 10) = 0.1368$$

Fig.4 shows how this probability varies with threshold with this simple MC estimator. It can be seen that for $threshold > \tilde{2}5$, the probabilities are too small and the estimator is not efficient. In fact the MC estimator gives exactly 0 for $\mathbb{P}(S > 55)$. Therefore, for $\mathbb{P}(S > 55)$, importance sampling will be used in next question to give more weight to samples in region of interest.

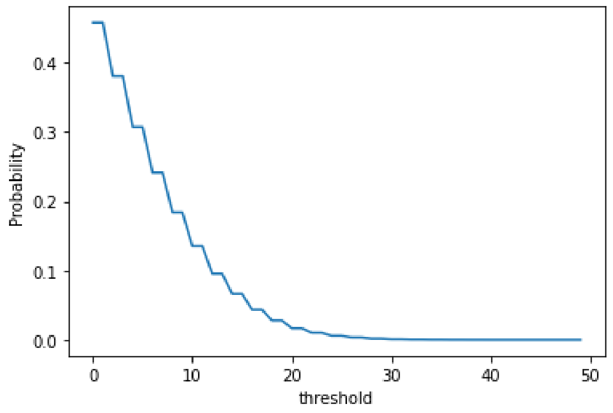


Figure 4: Visualisation of $\mathbb{P}(S > threshold)$

2.1(c)

For importance sampling a proposal distribution is chosen as a biased random walk with

$$\mathbb{P}(1) = 0.9, \mathbb{P}(-1) = 0.1$$

An Inverse CDF sampler is written for this shifted/biased random walk. Note that this bias is tuned based on experience so that we get many samples where the agent goes beyond 55 steps. A MC estimator is then written by sampling from this biased random walk and weighting the samples by the ratio *Unbiased Random walk probability / Biased Random Walk probability.* The final probability is obtained as

$$\mathbb{P}(S > 55) = 4 \times 10^{-5}$$

Fig.7 shows variation of this probability for different threshold (but same bias) and it can be seen that compared to normal MC, importance sampling procedure higher probabilities in the tail of distribution.

2.1(d)

The actual probabilities can be calculated as follows. First we know that $(S > T) = 1 - \mathbb{P}(S <= T)$. Now suppose $+1$ is chosen k times and -1 is chosen $100 - k$ times. This particular choice can be realized in $\binom{100}{k}$(possible combinations of choosing k positions out of n) ways and probability of

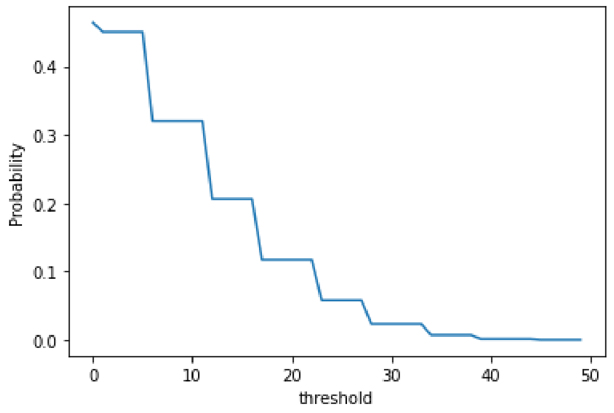


Figure 5: Visualisation of $\mathbb{P}(S > \mathit{threshold})$ with Importance Sampling

each of these realization is $0.5^k 0.5^{100-k}$. Also, for each of these, the position at end is given as $k(1) + (100-k)(-1) = 2k - 100 \leq T$. Therefore,

$$k \leq (T + 100)/2$$

- For $T = 10$, $k \leq 55$ and

$$\mathbb{P}(S \leq 10) = \sum_{k=0}^{55} \binom{n}{k} (0.5)^k (0.5)^{100-k} \tag{3}$$

  and $\mathbb{P}(S > 10) = 0.1356$

- For $T = 55$, $k \leq 78$ and

$$\mathbb{P}(S \leq 55) = \sum_{k=0}^{78} \binom{n}{k} (0.5)^k (0.5)^{100-k} \tag{4}$$

  and $\mathbb{P}(S > 55) = 7.952 \times 10^{-9}$

2.1(e)

Part (b)

1. The MC standard error is evaluated for each estimate. First 1000 replicates of MC algorithm is run to compute probability $\mathbb{P}(S > \mathit{threshold})$. Each of these replicates has 10^5 trials. These 1000 probability estimates are then used to compute standard error as 0.00218.

2. The confidence intervals are obtained as

$$\mathbb{P}(S_n - \frac{z\sigma}{\sqrt{n}} \leq \mu \leq \frac{z\sigma}{\sqrt{n}} + S_n) = 1 - \delta \tag{5}$$

   For $1 - \delta = 0.95$, $z \approx 2$ for a gaussian.

3. The fraction of replicates that contain true mean within the above confidence bound (note that S_n changes for each replicate in above formula) is 95% for submitted code in Appendix. But it was seen to vary between 92-97% if the whole algorithm is run again. This is simply due to low number of replicates used (=1000).

2.2 (a)

The code is given in Appendix. *numpy's* random function is used to give a random array of size 3 (X = np.random.normal(size = (num_dim,num_steps))).

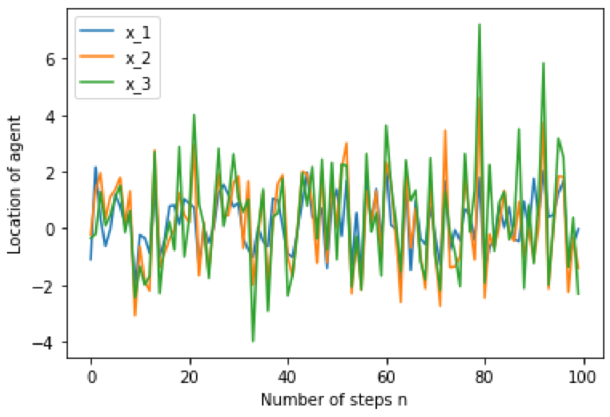


Figure 6: Visualisation of three components of Gaussian Random Walk with time

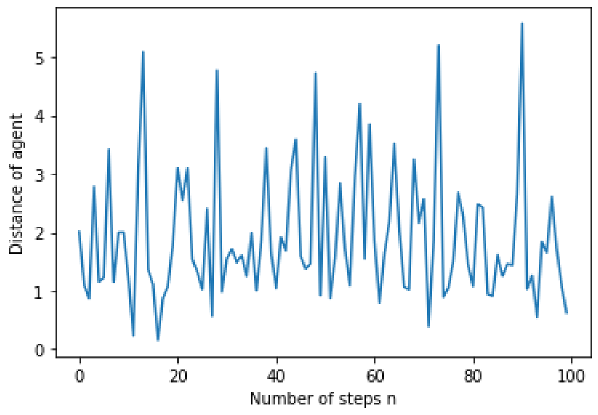


Figure 7: Visualisation of distance of Gaussian Random walking agent with time

2.2 (b)

A simple MC estimator is written to compute probability $\mathbb{P}(|S| > 10)$. Compared to previous code, the condition used to evaluate fraction of paths is based on norm of the 3 component Gaussian random walk. The probability comes out to be

$$\mathbb{P}(|S| > 10) = 10^{-5}$$

It was observed that the variance of this estimate, even with 10^5 trails is high as many times the probability comes out to be 0. Since Gaussian random walk takes much shorter steps over a time horizon compared to normal random walk. Therefore, again there is even more of a need to do importance sampling for $\mathbb{P}(S > 55)$.

2.2 (c)

A shifted Gaussian random walk sampler is written with Gaussian that has mean 2.0 and variance 1.0. Importance sampling is then used with this shifted Gaussian and following observation is made:

$$\mathbb{P}(S > 55) = 0.00272$$

4: Multilevel Monte Carlo and Control Variates for Stochastic ODEs

Question 1: What is distribution of ΔW_n?

With $t_{n+1} = \Delta t(n+1), t_n = \Delta t n$.

$$\Delta W_n = W(t_{n+1}) - W(t_n) \tag{6}$$

$$\tag{7}$$

The central limit theorem and Donsker's theorem can be used to show that Wiener process is given by following expression:

$$W(t) = \frac{1}{\sqrt{2\pi t}} e^{-x^2/2t} \tag{8}$$

and has following properties: $E[W_t] = 0, Var(W_t) = t$. This gives us that $E[\Delta W_n] = 0$ and $Var(\Delta W_n) = Var(W(t_{n+1})) - Var(W(t_n)) = t_{n+1} - t_n = \delta t$. Therefore, $\Delta W_n \sim \mathcal{N}(0, \delta t)$.

References: 1. https://en.wikipedia.org/wiki/Donsker%27s_theorem
2. https://en.wikipedia.org/wiki/Wiener_process

(1)

Given

$$\begin{aligned} Y_t &= Y_0 \exp{(\mu - \sigma^2/2)t + \sigma W_t} && (9) \\ E[Y_t] &= Y_0 e^{\mu - \sigma^2/2} E[e^{\sigma W_t}] && (10) \\ &= Y_0 e^{\mu - \sigma^2/2} \int_{-\infty}^{\infty} e^{\sigma x} \frac{1}{\sqrt{2\pi t}} e^{-x^2/2t} dt && (11) \\ &= 1.05127 && (12) \end{aligned}$$

First, i.i.d Gaussian steps are sampled for ΔW. These are fed to Euler-Maruyama scheme for generating Geometric Brownian Motion paths. The current simulation uses $dt = 0.01$ and number of trials as 1000. Code is written to simulate Geometric Brownian Motion. Fig.8 shows the plot of location vs time.

- The MC mean is calculated to be 1.0479.
- The MC variance is calculated to be 0.0442.

(2)

Code is written to make a coarse realization of fine resolution brownian motion. This coarse brownian motion is then used to generate coarse geometric motion. Code can be found in Appendix. Fig.9 shows the results.

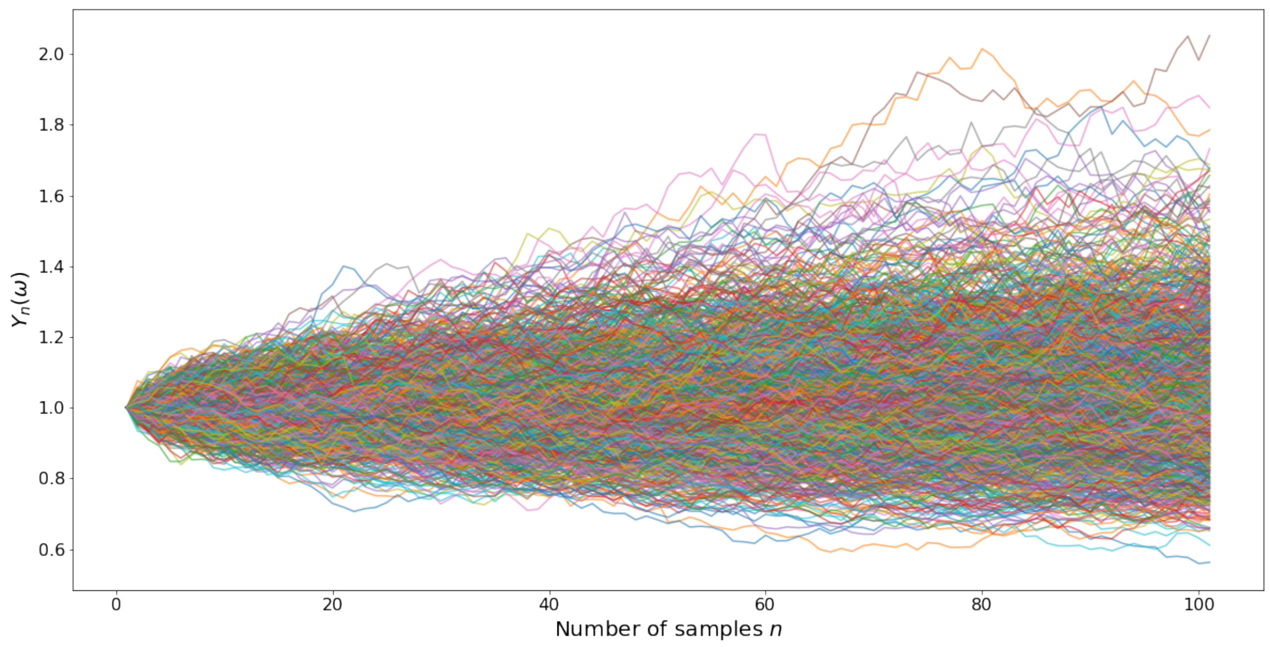


Figure 8: Geometric Brownian Motion

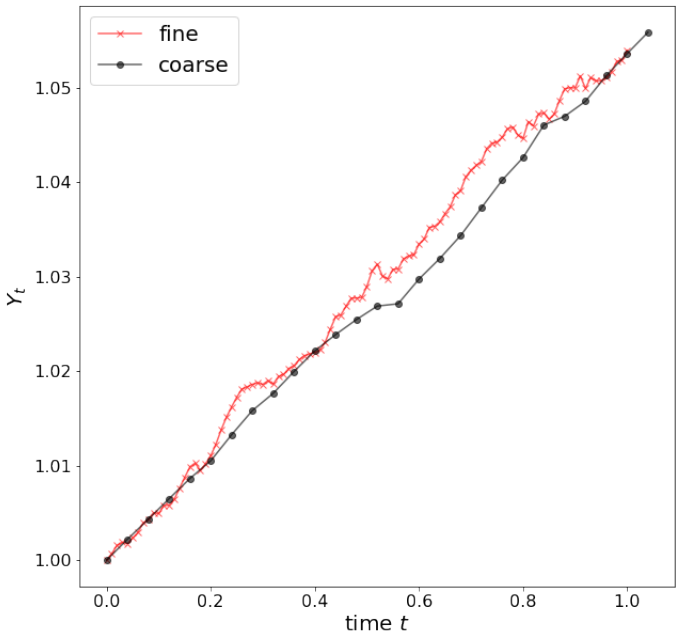


Figure 9: Coarse Realization of Geometric Motion

(3)MLMC

A Multi level Monte Carlo scheme is written to calculate the mean.

$$E[X_L] = \underbrace{E[X_0]}_{level\ 0\ term} + \underbrace{E[X_1 - X_0]}_{level\ 1\ term} + \underbrace{E[X_2 - X_1]}_{level\ 2\ term} + \underbrace{E[X_3 - X_2]}_{level\ 3\ term} \tag{13}$$

X_0 corresponds to $\Delta t = 4^{-2}$ and X_3 corresponds to $\Delta t = 4^{-5}$. We expect variance to reduce with each level. The plot below will verify that.

For MLMC scheme, each of term in above equation are generated with independent samples. However, within each term, say $X_1 - X_0$, samples are generated from same underlying process and hence the lower level is a coarse realization of the higher level. Therefore in this term X_0 is produced using coarsed version of X_1's brownian motion component as done above in part (2).

Fig. 10,11 show the plot of mean and variance of different levels. Level 0 is supposed to be close to the actual mean and is therefore high. Other levels are more like error corrective terms obtained from different of two resolution processes. Therefore, they have small values. The variance of Level 0 is much higher than other levels. This is expected from theory as X_i, X_{i+1} are correlated and so their different should reduce variance compared to individual X_i. The trend between Level 1,2,3 though is not fixed. This might be dependent on the rate at which variance changes with resolution.

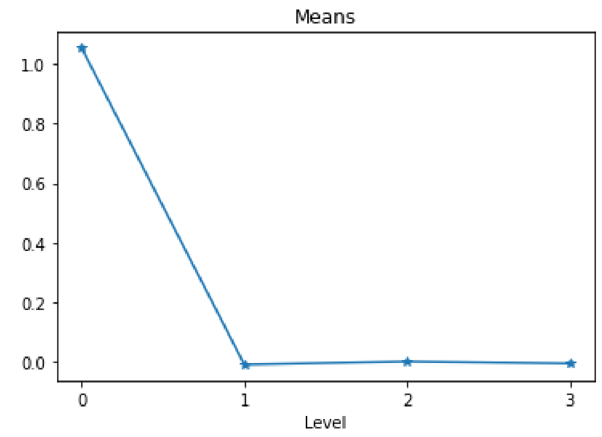


Figure 10: Expectation of each level of the estimator

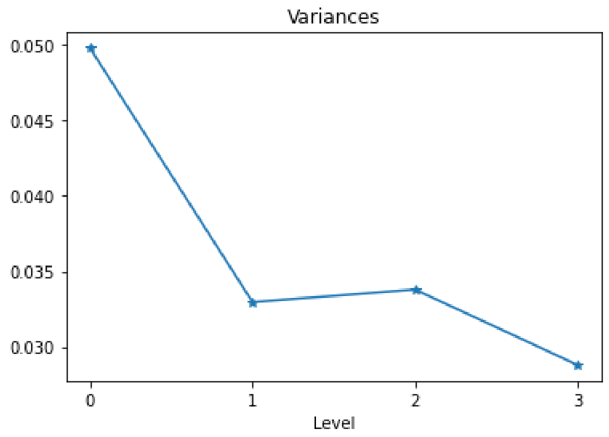


Figure 11: Variance of each level of the estimator

(4) Theoretical Cost and Optimal Cost of MLMC estimator

Note: The solution mostly follows the supplementary material on Multi Level Monte Carlo given in class.

Let C_i be the cost of evaluating X_0. Then the effective cost $\hat{C}_l$ of each level can be represented as

$$\hat{C}_l = \begin{cases} C_0 & \text{if } l = 0 \\ C_l + C_{l-1} & otherwise \end{cases} \tag{14}$$

and suppose each term(without expectation) in Eq.(15) is represented as $L_l, l = 0, 1, 2, 3$ and variances as $V_l = Var(L_l), l = 0, 1, 2, 3$. Then we would like minimum variance for monte carlo estimate that is done with N_i samples for each of levels L_i, $i = 0, 1, 2, 3$.

$$S_N = \underbrace{S_{N_0}[L_0]}_{level\ 0\ term} + \underbrace{S_{N_1}[L_1]}_{level\ 1\ term} + \underbrace{S_{N_2}[L_2]}_{level\ 2\ term} + \underbrace{S_{N_3}[L_3]}_{level\ 3\ term} \tag{15}$$

and

$$Var(S_N) = \sum_{l=1}^{3} \frac{V_l}{N_l} \tag{16}$$

Therefore, given a user-decided maximum cost C, we can formulate the following optimization problem for for optimal allocation

$$\min_{N_0,N_1,N_2,N_3} \sum_{l=1}^{3} \frac{V_l}{N_l} \text{ subject to} \sum_{l=0}^{3} N_l \hat{C}_l \leq C \tag{17}$$

This is mixed integer optimization problem (since N_i's are integers) and very difficult to solve. Therefore, we can approximate it by continuous variables and then solve it using Lagrangian formulation to take the constraint into account.

$$L(N, \lambda^2) = \sum_{l=1}^{3} \frac{V_l}{N_l} + \lambda^{-2}(\sum_{l=0}^{3} N_l \hat{C}_l - C) \tag{18}$$

with $N = [N_0, N_1, N_2, N_3]$ and we minimize this Lagrangian.

$$\frac{\partial L(N, \lambda^2)}{\partial N_l} = 0 \implies N_l = \lambda \sqrt{V_l \hat{C}_l} \tag{19}$$

The optimal variance then becomes

$$Var(S_N) = \lambda^{-1} \sum_{l=0}^{3} \sqrt{V_l \hat{C}_l} \tag{20}$$

For a user-given accuracy(variance) ϵ^2, we get

$$\lambda = \epsilon^{-2} \sum_{l=0}^{3} \sqrt{V_l \hat{C}_l} \tag{21}$$

Therefore, this gives us the optimal sample allocation as

$$N_l = (\epsilon^{-2} \sum_{l=0}^{3} \sqrt{V_l \hat{C}_l}) \sqrt{V_l \hat{C}_l} \tag{22}$$

(5) Optimal Sample Allocation for target Accuracies

,

- We can equate variance of high fidelity model(based on X_3) alone to the user given accuracy. Suppose number of trials of MC is N_3^{eq}. Then $Var(S_{N_3^{eq}}) = \epsilon^2 = Var(X_3)/N_3^{eq}$.

$$N_3^{eq} = Var(X_3)/\epsilon^2 \tag{23}$$

- The equivalent number of samples are obtained by comparing the cost of MC and MLMC. If MLMC uses a sample allocation corresponding to N, then an equivalent MC estimator will use $N^{eq} = C_N^{MLMC}/C_3$. Here, the cost of MLMC is given as

$$C_N^{MLMC} = \epsilon^{-2}\left(\sum_{l=0}^{3}\sqrt{V_l\hat{C}_l}\right)^2 \tag{24}$$

The variance of each level is taken from the plot in Fig. 11. The cost is assumed to be uniformly 1 for evaluating a single random variable, fine or coarse version. Therefore $C_i = 1$, $\hat{C}_i = 2, i = 1, 2, 3$. Fig.12 shows the number of samples of high-fidelity model required for MC and MLMC. It can be seen that the number of samples of high-fidelity model required is much smaller for MLMC compared to MC.

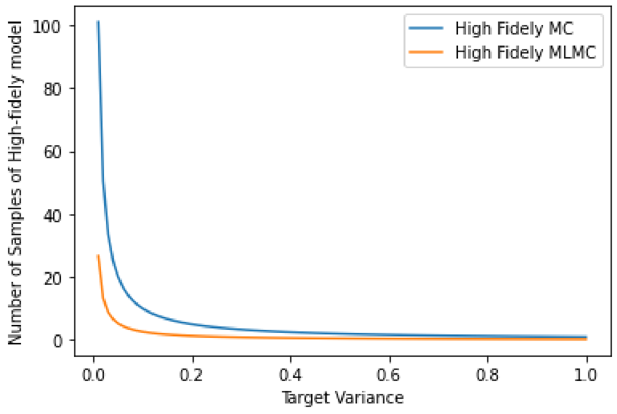


Figure 12: Comparison between MC and MLMC for high fidely model

Appendix

Project 1_2

October 20, 2021

0.0.1 2: Warmup - life without a CLT

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

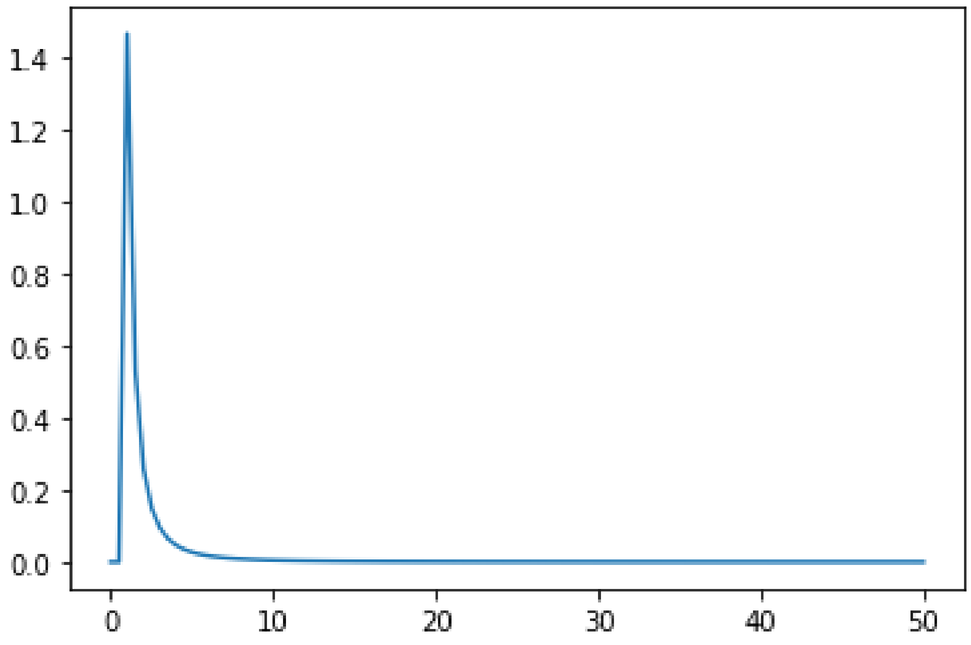
```
[2]: def monte_carlo(alpha,num_samples, sample_generator, g_evaluator, cumsum=False):
         """Perform Monte Carlo sampling

         Inputs
         ------
         num_samples: integer, number of samples
         sample_generator: A function that generates samples with signature␣
      ↪sample_generator(nsamples)
         g_evaluator: a function that takes as inputs the samples and outputs the␣
      ↪evaluations.
                      The outputs can be any dimension, however the first dimension␣
      ↪should have size *num_samples*
         cumsum: Boolean, an option to return estimators of all sample sizes up to␣
      ↪num_samples

         Returns
         -------
         A Monte Carlo estimator of the mean, samples, and evaluations
         """
         samples = sample_generator(alpha,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,␣
      ↪dtype=np.float)

         return estimate, samples, evaluations
```

```
[ ]:
```

```
[3]: # Pareto PDF
     def pareto_rv(alpha,xinput):
         p = [alpha/( x**(alpha+1) ) if x>=1 else 0 for x in xinput]
         return p
```

```
[4]: alpha = 3/2
     x = np.linspace(0,50,100)
     y = pareto_rv(alpha,x)
     plt.plot(x,y)
```

```
[4]: [<matplotlib.lines.Line2D at 0x7f4505b3a940>]
```



CDF of pareto is given by

$$CDF = \int_1^x \frac{\alpha}{x^{\alpha+1}} = 1 - \frac{1}{x^{\alpha}} \tag{1}$$

Now creating a pareto sampler with Inverse CDF sampler

```
[5]: np.random.seed(10)

     def pareto_cdf(alpha,xinput):
         return 1 - 1/xinput**alpha

     def pareto_sampler(alpha,num_samples):
         # Function inefficient since it generates sample_space_x_cdf every time it␣
      ↪is called
```

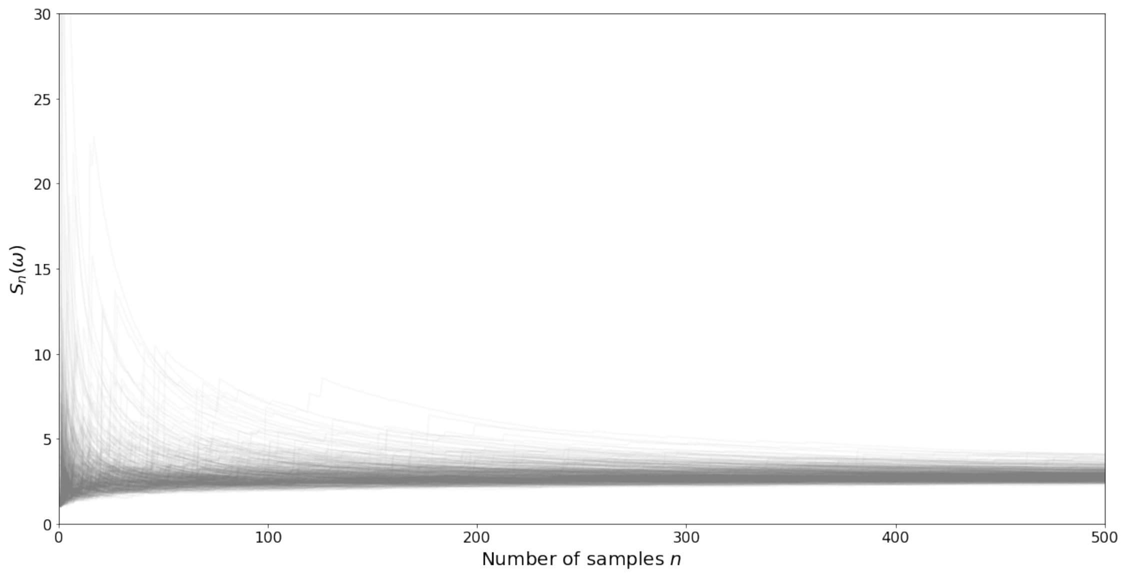
```
    # Following naming convention from lecture notes
    u = np.random.uniform(size = num_samples)
    sample_space_x = np.linspace(1,500,100000 )
    sample_space_x_cdf = pareto_cdf(alpha,sample_space_x)
    x_output = []
    for i in range(num_samples):
        index = np.argmax( sample_space_x_cdf > u[i] )
        x_output.append( sample_space_x[index] )
    return x_output
```

[6]:
```
# Generating Paths
alpha = 3/2
g = lambda x: x # identity function
num_trials = 500
num_samples = 10000
estimator_vals = np.zeros((num_trials, num_samples))
for trial in range(num_trials):
    estimator_vals[trial, :], _, _ = monte_carlo(alpha,num_samples,␣
 ↪pareto_sampler, g, cumsum=True)
```

0.0.2 Plotting

[9]:
```
print("estimates",np.shape(estimator_vals))
plt.figure(figsize=(20,10))
plt.plot(np.arange(1, num_samples+1), estimator_vals.T, color='grey', alpha=0.
 ↪05)
plt.ylabel(r'$S_n(\omega)$', fontsize=20)
plt.xlabel(r'Number of samples $n$', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.xlim([0, 500])
plt.ylim([0, 30])
plt.show()
```

```
estimates (500, 10000)
```



0.0.3 Getting mean from all trials. actual mean = 3.0

```
[8]: # print(estimator_vals[:,-1])
```

```
[7]: print("Mean is ",np.mean(estimator_vals[:,-1]))
```

```
Mean is  2.8735793153731515
```

0.0.4 Evaluate convergence properties

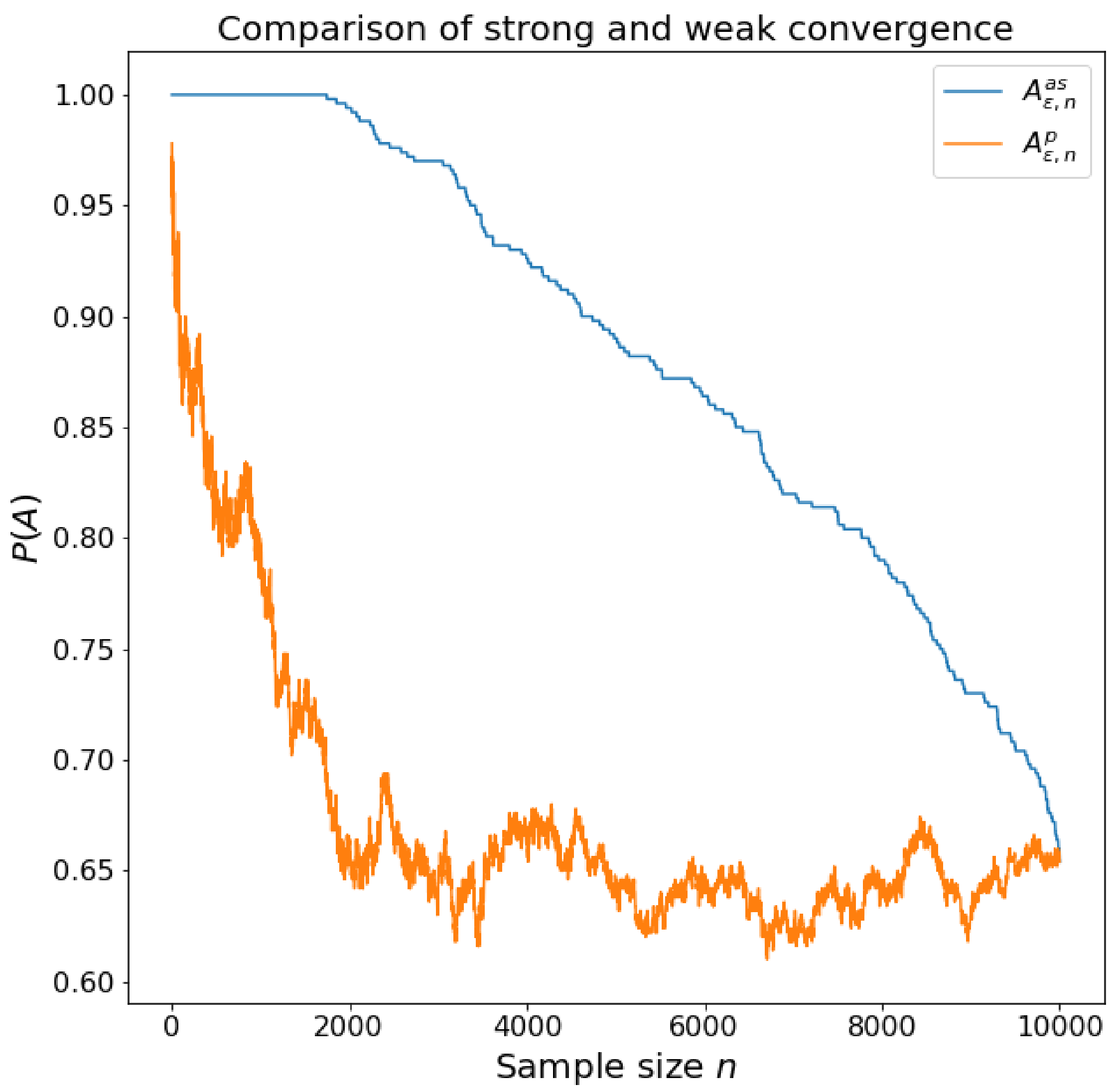
Copying functions from lecture notebooks on weak_vs_strong_convergence

```
[108]: def estimate_probability_prob(sample_path_errs, epsilon, n):
           """ Estimate the probability of the event related to convergence in␣
        ↪probability

           sample_path_errs: (Npaths, Nsamples_per_path) array of errors of each path
           epsilon: float, target error region
           n: positive integer
           """

           Npaths, Nsamples_per_path = sample_path_errs.shape
           estimate = np.sum(sample_path_errs[:, n-1] > epsilon) / float(Npaths) #n-1␣
        ↪because indexing by zero
           return estimate

       def estimate_probability_as(sample_path_errs, epsilon, n):
```

```
    """ Estimate the probability of the event related to convergence almost␣
↪surely

    sample_path_errs: (Npaths, Nsamples_per_path) array of errors of each path
    epsilon: float, target error region
    n: positive integer

    Note
    ----
    This function is a bit inefficient, it would be better if n could be a list␣
↪of variables so that we can reuse
    the calculations
    """

    Npaths, Nsamples_per_path = sample_path_errs.shape
    # Note the difference from in probability ---- we are looking into the␣
↪future
    # We are looking if any value in the path satisfies the error condition
    paths_satisfy_condition = np.any(sample_path_errs[:, n-1:] > epsilon,␣
↪axis=1)
    estimate = np.sum(paths_satisfy_condition) / float(Npaths)
    return estimate
```

[114]:
```
epsilon = 0.1
n = np.arange(1, num_samples+1)
prob_as = np.zeros((n.shape[0])) # probability almost surely
prob_p = np.zeros((n.shape[0]))
for ii, nn in enumerate(n):
    # we take absolute values because the true answer is 0
    prob_as[ii] = estimate_probability_as(np.abs(estimator_vals-3.0), epsilon,␣
↪nn)
    prob_p[ii] = estimate_probability_prob(np.abs(estimator_vals-3.0), epsilon,␣
↪nn)
```

[115]:
```
plt.figure(figsize=(10,10))
plt.plot(n, prob_as, label=r'$A_{\epsilon, n}^{as}$')
plt.plot(n, prob_p, label=r'$A_{\epsilon, n}^{p}$')
plt.title('Comparison of strong and weak convergence', fontsize=20)
plt.ylabel(r'$P(A)$', fontsize=20)
plt.xlabel(r'Sample size $n$', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.legend(fontsize=16)
plt.show()
```


Comparison of strong and weak convergence
$P(A)$
Sample size n
$A^{as}_{\varepsilon,n}$
$A^{p}_{\varepsilon,n}$
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0
2000
4000
6000
8000
10000

```
[ ]:
```

Project 1_3 (copy)

October 20, 2021

0.1 3: Importance sampling for random walks

```
[2]: import numpy as np
     import matplotlib.pyplot as plt
```

0.1.1 3.1 1-D Bernoulli random walk

```
[3]: x = 0.5
     N = 100
     length = 1
     # left or right: 0 or 1
```

0.1.2 2.1 (a) Below:

- define random walk sampler to pass to random walk simulation

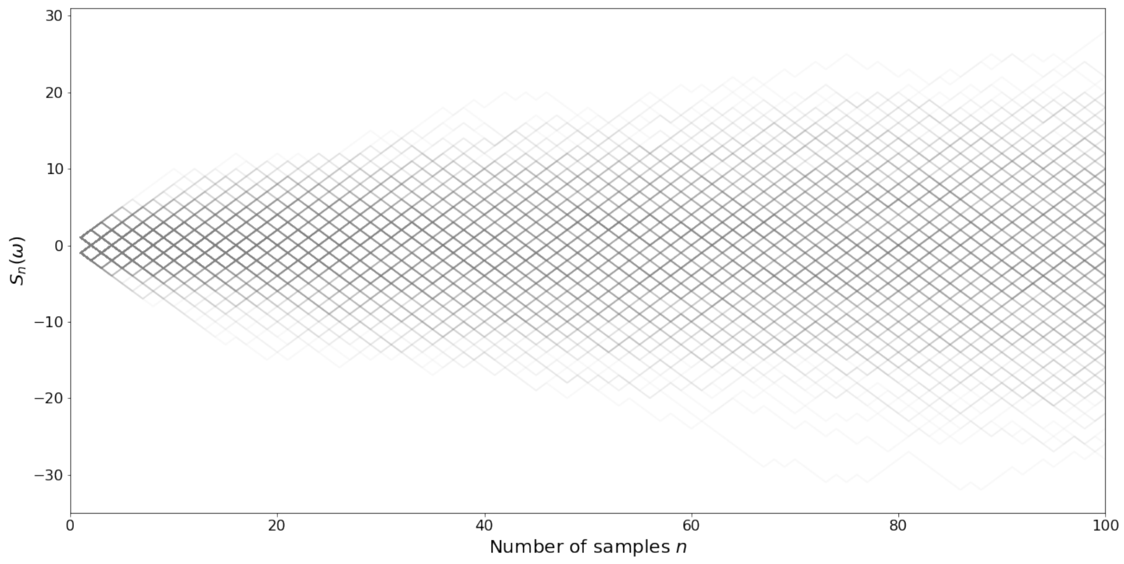
```
[4]: def random_walk_sampler(num_steps):
         """ Generate a set of steps for a random walk"""
         X = np.random.rand(num_steps) # samples from a uniform
         # Inverse CDF Trick
         X[X > 0.5] = 1.0
         X[X < 0.5] = -1.0
         return X
```

```
[5]: def random_walk(num_samples, sample_generator, cumsum=False):

         samples = sample_generator(num_samples)
         if cumsum is False:
             estimate =  np.sum(samples, axis=0)
         else:
             estimate = np.cumsum(samples, axis=0)

         return estimate, samples
```

Generating random walk paths

```
[6]: g = lambda x: x # identity function
     num_trials = 500
     num_samples = N
     estimator_vals = np.zeros((num_trials, num_samples))
     for trial in range(num_trials):
         estimator_vals[trial, :], _ = random_walk(num_samples, random_walk_sampler,␣
      ↪cumsum=True)
     #     print("one trial: ", estimator_vals[trial, :])
```

```
[7]: plt.figure(figsize=(20,10))
     plt.plot(np.arange(1, num_samples+1), estimator_vals.T, color='grey', alpha=0.
      ↪05)
     plt.ylabel(r'$S_n(\omega)$', fontsize=20)
     plt.xlabel(r'Number of samples $n$', fontsize=20)
     plt.xticks(fontsize=16)
     plt.yticks(fontsize=16)
     plt.xlim([0, 100])
     # plt.ylim([0, 30])
     plt.show()
```



0.1.3 2.1(b)

- define Monte carlo function below
- then define another function that computes cases when S>10

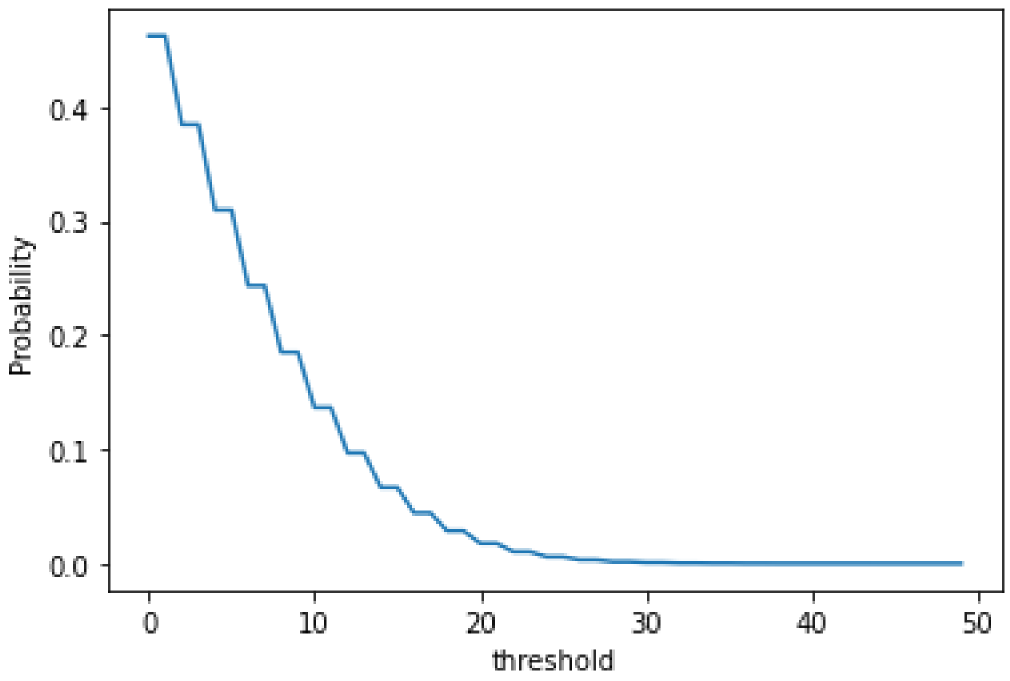
```
[8]: num_trials = 10**5
     def monte_carlo_probability(sample_paths_vs_n, threshold):
         N_paths, _ = sample_paths_vs_n.shape
```

```
    s_estimates_satisfy_condition = np.any( sample_paths_vs_n[:,-1:] >␣
↪threshold, axis = 1 )  # Choose last element which corresponds to N = 100
    s_estimate = np.sum(s_estimates_satisfy_condition) / float(N_paths)
    return s_estimate
```

[9]:
```
estimator_vals = np.zeros((num_trials, num_samples))
for trial in range(num_trials):
    estimator_vals[trial, :], _ = random_walk(num_samples, random_walk_sampler,␣
↪cumsum=True)

threshold = 10
print("P(S>10) = ", monte_carlo_probability(estimator_vals, threshold))

threshold = 55
print("P(S>55) = ", monte_carlo_probability(estimator_vals, threshold))

probs = []
for threshold in range(50):
    probs.append(monte_carlo_probability(estimator_vals, threshold))

plt.plot(probs)
plt.xlabel('threshold')
plt.ylabel('Probability')
```

```
P(S>10) =  0.13676
P(S>55) =  0.0
```

[9]:
```
Text(0, 0.5, 'Probability')
```



Therefore, probability for steps > about 25 is too low and cannot be computed with standard monte carlo easily. We need importance sampling.

0.1.4 Actual probability can be calculated as follows:

P(S>10) = 1 - P(S<=10)

Suppose +1 is chosen k times and -1 is chosen 100-k times. Then position at end = $k(1) + (100 - k)(-1) = 2k - 100 \leq 10$. Therefore, $k \leq 55$.

$$P(S \leq 10) = \sum_{k=0}^{55} \binom{n}{k} (0.5)^k (0.5)^{100-k} \tag{1}$$

[58]:
```
# Actual Probability
from math import factorial
def comb(n, k):
    return factorial(n) / factorial(k) / factorial(n - k)

prob = 0
for k in range(56):
    prob = prob + comb(100,k)
print(prob)
prob = prob * 0.5**100
print("Actual P(S>10) = ", 1 - prob)
```

```
1.09572357083777e+30
Actual P(S>10) =  0.13562651203691733
```

0.1.5 2.1 (c)

- Importance sampling with 10^5 trials for P(S>55)
- Next we try importance sampling with a proposal DISCRETE distribution that puts more probability on getting $X_i = 1$.

: P(1) = 0.8, P(-1) = 0.2

This is because, we want to use importance sampling on X_i and therefore the only way we can do this is by shidting probability towards 1.0

[11]:
```
shift = 0.1

def shifted_random_walk_sampler(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 > shift] = 1.0
    X[X < shift] = -1.0
    return X

def prob_normal_walk(sample):
    return 0.5

def prob_shifted_walk(sample):
    if sample>0:
        return 1 - shift
    else:
        return shift
```

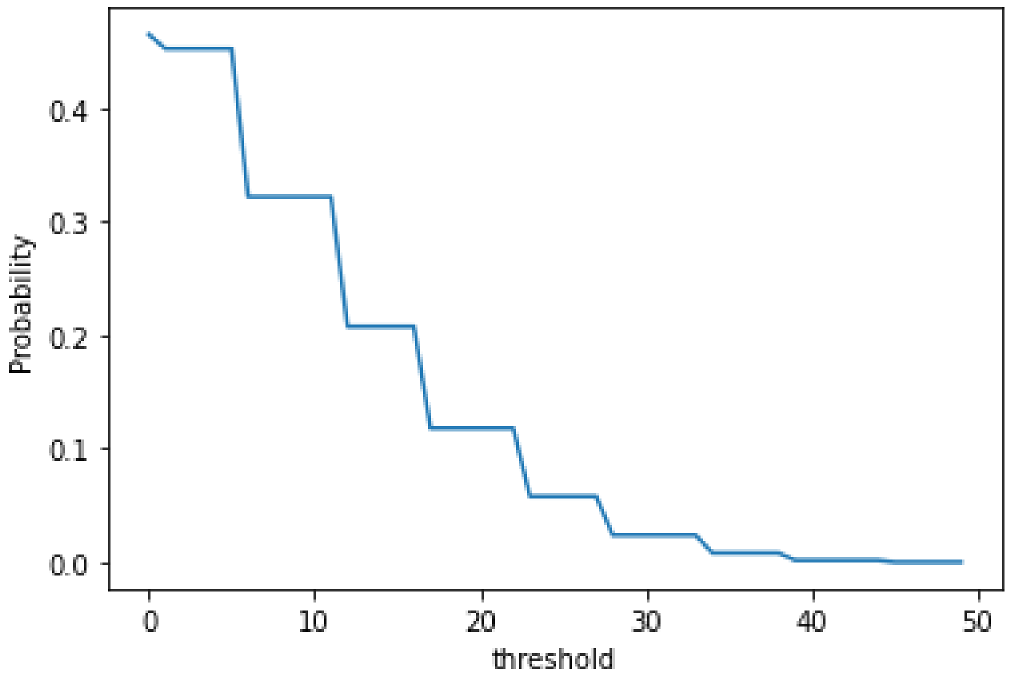
[12]:
```
def random_walk_weighted(num_samples, sample_generator, cumsum=False):

    samples = sample_generator(num_samples)
    weighted_samples = np.asarray([ sample * prob_normal_walk(sample) /␣
↪prob_shifted_walk(sample) for sample in samples])
    if cumsum is False:
        estimate =  np.sum(weighted_samples, axis=0)
    else:
        estimate = np.cumsum(weighted_samples, axis=0)

    return estimate, samples

def monte_carlo_importance(sample_paths_vs_n, threshold):
    N_paths, _ = sample_paths_vs_n.shape
    s_estimates_satisfy_condition = np.any( sample_paths_vs_n[:,-1:] >␣
↪threshold, axis = 1 )  # Choose last element which corresponds to N = 100
```

```
    s_estimate = np.sum(s_estimates_satisfy_condition) / float(N_paths)
    return s_estimate
```

[13]:
```
num_trials = 10**5
estimator_vals_importance = np.zeros((num_trials, num_samples))
for trial in range(num_trials):
    estimator_vals_importance[trial, :], _ = random_walk_weighted(num_samples,␣
 ↪shifted_random_walk_sampler, cumsum=True)

threshold = 55
print("P(S>55) = ", monte_carlo_importance(estimator_vals_importance,␣
 ↪threshold))

probs = []
for threshold in range(50):
    probs.append(monte_carlo_importance(estimator_vals_importance, threshold))

plt.plot(probs)
plt.xlabel('threshold')
plt.ylabel('Probability')
```

```
P(S>55) =  5e-05
```

[13]:
```
Text(0, 0.5, 'Probability')
```



Actual probability can be calculated as follows: P(S>55) = 1 - P(S<=55)

Suppose +1 is chosen k times and -1 is chosen 100-k times. Then position at end = $k(1) + (100 - k)(-1) = 2k - 100 \leq 55$. Therefore, $k \leq 77.5$.

$$P(S \leq 55) = \sum_{k=0}^{77} \binom{n}{k} (0.5)^k (0.5)^{100-k} \tag{2}$$

[14]:
```
# Actual Probability
from math import factorial
def comb(n, k):
    return factorial(n) / factorial(k) / factorial(n - k)

prob = 0
for k in range(78):
    prob = prob + comb(100,k)
print(prob)
prob = prob * 0.5**100
print("Actual P(S>55) = ", 1 - prob)
```

```
1.26765059014703e+30
Actual P(S>55) =  7.952664082822025e-09
```

0.1.6 2.1 (e) (i) Monte Carlo errors

[15]:
```
def mc_estimate_of_probability(num_trials, num_samples, threshold=10):
    estimator_vals = np.zeros((num_trials, num_samples))
    for trial in range(num_trials):
        estimator_vals[trial, :], _ = random_walk(num_samples,␣
 ↪random_walk_sampler, cumsum=True)
    return monte_carlo_probability(estimator_vals, threshold)

def replicates(num_replicate, num_samples, true_value, threshold=10):
    # One replicate
    probability_estimates = np.zeros(num_replicate)
    for run in range(num_replicate):
        probability_estimates[run] = mc_estimate_of_probability(100000,␣
 ↪num_samples, threshold)

    print(probability_estimates)
    z = 2
    std_error =np.std(probability_estimates)#/np.sqrt(num_replicate) sigma/
 ↪sqrt(n) is the std deviation of Sn. Therefore no need to divide by sqrt(n)

    print("std error", z*std_error)

    success = 0
    for run in range(num_replicate):
        Sn = probability_estimates[run]
        bound_lower = Sn - z*std_error
        bound_upper = Sn + z*std_error
        if true_value<=bound_upper and true_value>=bound_lower:
            success = success + 1

    return success/num_replicate

replicates(100,num_samples, 0.135626, threshold=10)
```

```
[0.13598 0.13477 0.13452 0.13491 0.1336  0.13649 0.13559 0.1361  0.13313
 0.13535 0.13596 0.13527 0.13486 0.13466 0.13426 0.13501 0.13645 0.13719
 0.13551 0.13588 0.13477 0.13633 0.13546 0.13496 0.13495 0.13589 0.13564
 0.1377  0.13507 0.13565 0.1364  0.13749 0.13763 0.13528 0.13582 0.13562
 0.13514 0.13512 0.13609 0.13777 0.1339  0.13546 0.13396 0.13404 0.13496
 0.13661 0.13458 0.13584 0.13555 0.1365  0.1365  0.13624 0.1366  0.13745
 0.13442 0.13601 0.13623 0.13391 0.1351  0.13464 0.13334 0.13458 0.13615
 0.13449 0.13491 0.13586 0.13686 0.13809 0.13721 0.13521 0.1352  0.13522
 0.13545 0.13684 0.13497 0.13573 0.13709 0.13572 0.13508 0.1348  0.13637
 0.134   0.1341  0.13814 0.13536 0.13784 0.13764 0.13433 0.13411 0.13517
 0.1358  0.13552 0.13497 0.13544 0.13641 0.13454 0.13621 0.13539 0.13651
 0.13427]
std error 0.002182282648971026
```

[15]:
```
0.95
```

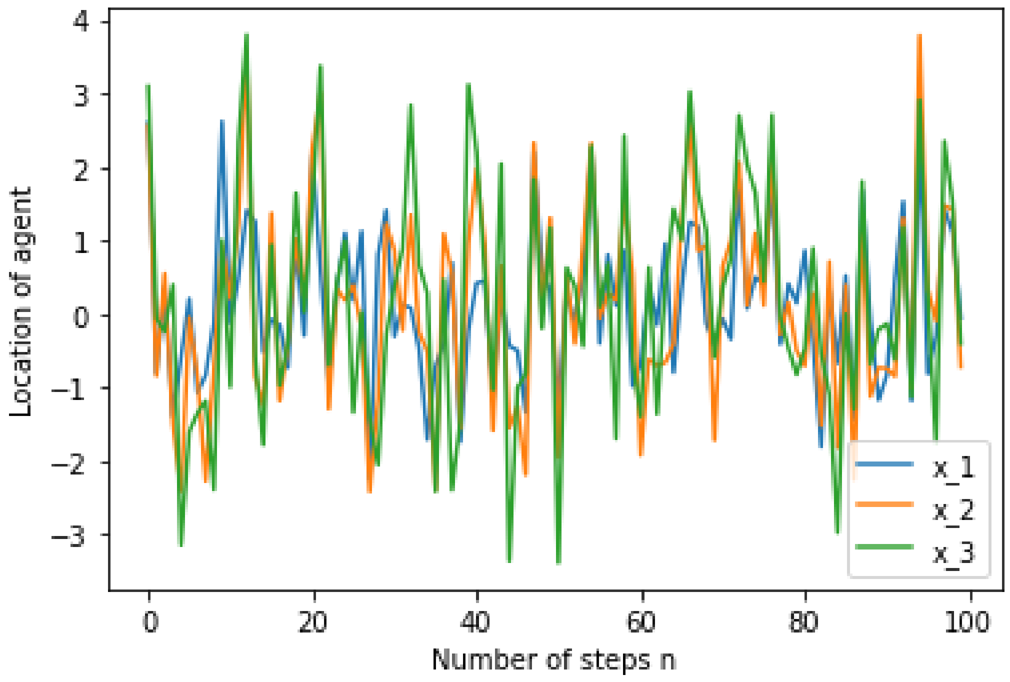
0.2 2.2 3-D Gaussian Random Walk

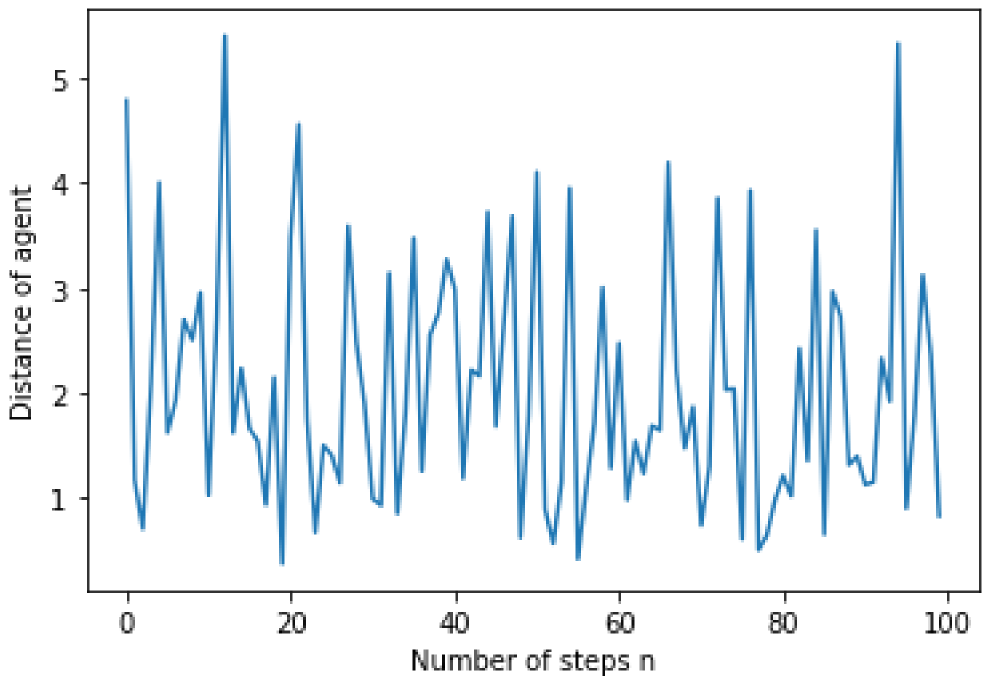
2.2(a)

[21]:
```
def Gaussian_random_walk_sampler(num_steps):
    """ Generate a set of steps for a random walk"""
    num_dim = 3
    X = np.random.normal(size = (num_dim,num_steps)) # samples from a uniform
    return X

def gaussian_random_walk(num_samples, sample_generator, cumsum=False):

    samples = sample_generator(num_samples)
    if cumsum is False:
        estimate =  np.sum(samples, axis=0)
    else:
        estimate = np.cumsum(samples, axis=0)
    return estimate, samples
```

[22]:
```
N = 100
path, samples = gaussian_random_walk(N, Gaussian_random_walk_sampler, cumsum =␣
 ↪True)

plt.figure()
plt.plot(path[0,:], label = r'x_1')
plt.plot(path[1,:], label = r'x_2')
plt.plot(path[2,:], label = r'x_3')
plt.xlabel('Number of steps n')
plt.ylabel('Location of agent')
plt.legend()

plt.figure()
plt.plot(np.linalg.norm(path, axis=0))
plt.xlabel('Number of steps n')
plt.ylabel('Distance of agent')
```

[22]:
```
Text(0, 0.5, 'Distance of agent')
```





0.2.1 2.2(b)

```
[23]: num_trials = 10**5
num_samples = N
estimator_gaussian_vals = np.zeros((num_trials, num_samples))
for trial in range(num_trials):
        path, _  = gaussian_random_walk(num_samples,␣
 ↪Gaussian_random_walk_sampler, cumsum=True)
#         print("path shape", path.shape)
#         print("|S| shape", np.linalg.norm(path, axis=0).shape)
        estimator_gaussian_vals[trial, :] = np.linalg.norm(path, axis=0)
print(estimator_gaussian_vals.shape )
```

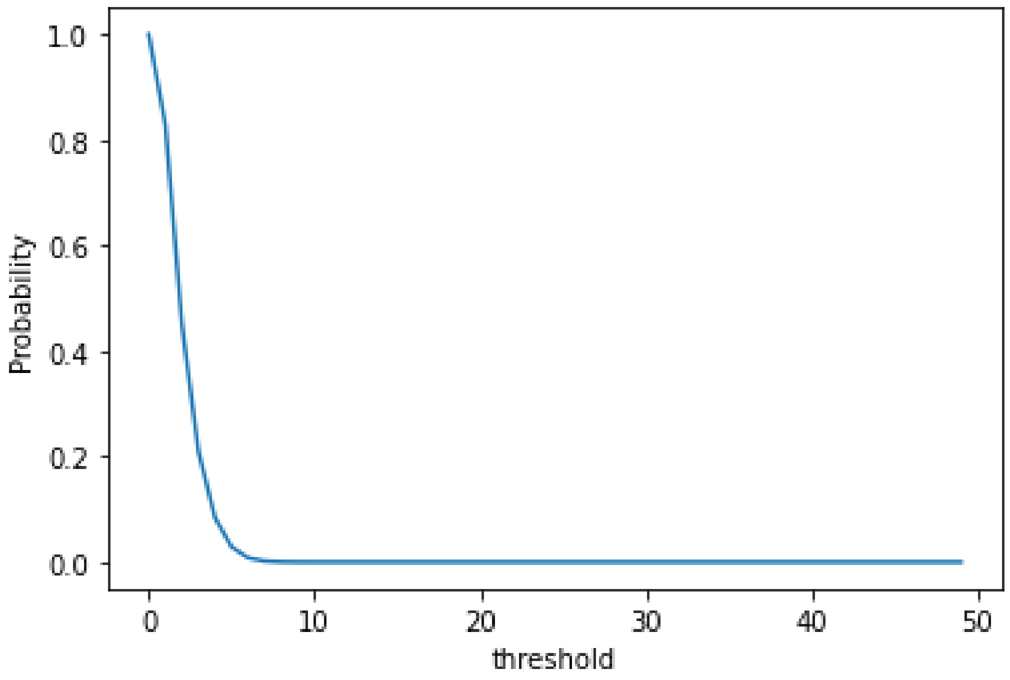
```
(100000, 100)
```

0.2.2 Calculating probabilities

```
[27]: threshold = 10
print("P(|S|>10) = ", monte_carlo_probability(estimator_gaussian_vals,␣
 ↪threshold))

probs = []
for threshold in range(50):
    probs.append(monte_carlo_probability(estimator_gaussian_vals, threshold))
```

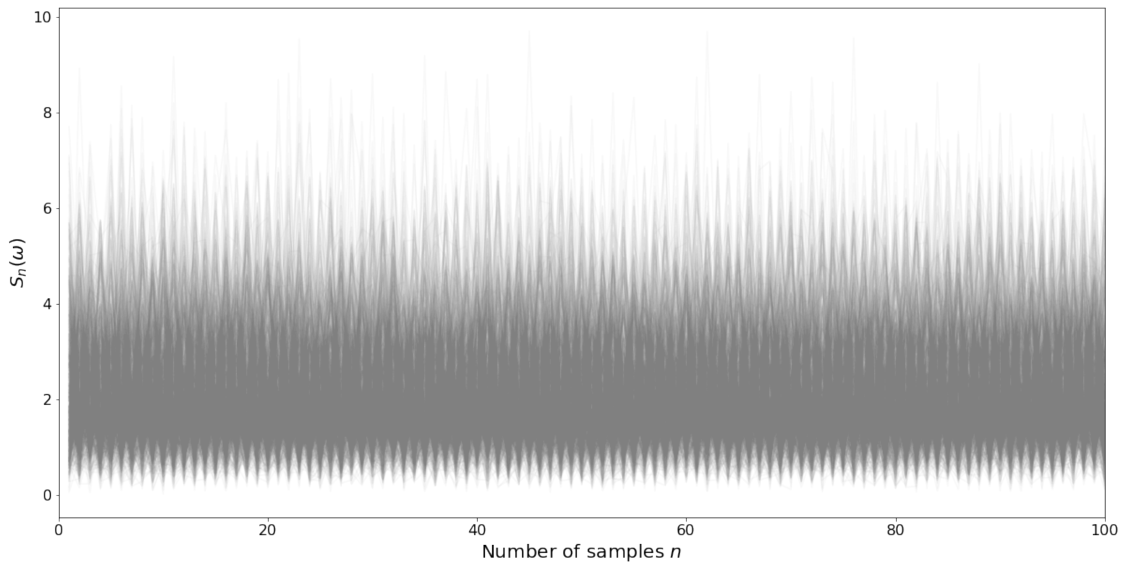
```
plt.plot(probs)
plt.xlabel('threshold')
plt.ylabel('Probability')

threshold = 55
print("P(|S|>55) = ", monte_carlo_probability(estimator_gaussian_vals,␣
 ↪threshold))
```

```
P(|S|>10) =  1e-05
P(|S|>55) =  0.0
```



Note here that the above probability variance is very high even with num_trials = 10^5. Most of the times, the probability will come out to be 0. Other times it is 10^{-5}

```
[159]: # plt.figure(figsize=(20,10))
       # plt.plot(np.arange(1, num_samples+1), estimator_gaussian_vals.T,␣
        ↪color='grey', alpha=0.05)
       # plt.ylabel(r'$S_n(\omega)$', fontsize=20)
       # plt.xlabel(r'Number of samples $n$', fontsize=20)
# plt.xticks(fontsize=16)                                                          ␣
 ↪                                                                                ␣
 ↪                                                                                ␣
 ↪                                                                                ␣
 ↪                                                                                ␣
 ↪                                                                                ␣
 ↪                                                                                ␣
 ↪                                                                                ␣
 ↪
# plt.yticks(fontsize=16)
# plt.xlim([0, 100])
# # plt.ylim([0, 30])
# plt.show()
```



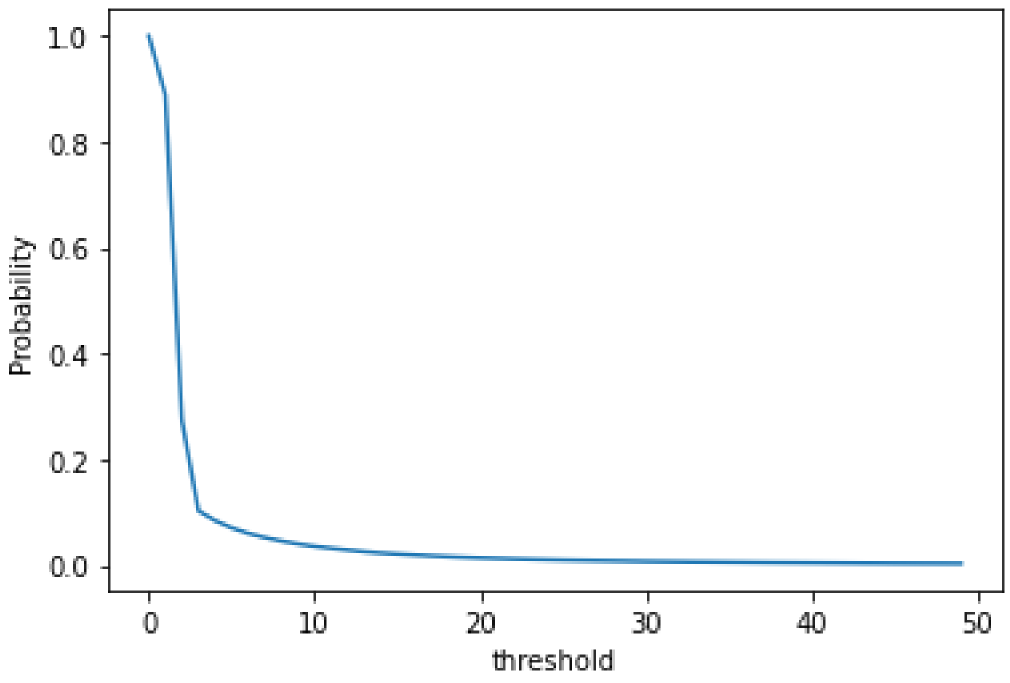
0.2.3 2.2 (c)

[28]:
```
def shifted_Gaussian_random_walk_sampler(num_steps):
    """ Generate a set of steps for a random walk"""
    num_dim = 3
    X = np.random.normal(loc = (2/np.sqrt(3)), size = (num_dim,num_steps)) #␣
 ↪samples from a uniform
    return X

def prob_normal_gaussian_walk(sample):
    return 1/np.sqrt(2*np.pi)*np.exp(-(sample)**2/2)

def prob_shifted_gaussian_walk(sample):
    shift_gaussian = 2/np.sqrt(3)
    return 1/np.sqrt(2*np.pi)*np.exp(-(sample-shift_gaussian)**2/2)

def gaussian_random_walk_weighted(num_samples, sample_generator, cumsum=False):

    samples = sample_generator(num_samples)
    weighted_samples = np.asarray([ sample * prob_normal_gaussian_walk(sample) /
 ↪ prob_shifted_gaussian_walk(sample) for sample in samples])
    if cumsum is False:
        estimate =  np.sum(weighted_samples, axis=0)
    else:
        estimate = np.cumsum(weighted_samples, axis=0)

    return estimate, samples
```

[29]:
```
num_trials = 10**5
estimator_vals_importance = np.zeros((num_trials, num_samples))
for trial in range(num_trials):
    path, _  = gaussian_random_walk_weighted(num_samples,␣
 ↪shifted_Gaussian_random_walk_sampler, cumsum=True)
    estimator_vals_importance[trial, :] = np.linalg.norm(path, axis=0)


threshold = 55
print("P(S>55) = ", monte_carlo_importance(estimator_vals_importance,␣
 ↪threshold))

probs = []
for threshold in range(50):
    probs.append(monte_carlo_importance(estimator_vals_importance, threshold))

plt.plot(probs)
plt.xlabel('threshold')
plt.ylabel('Probability')
```

```
P(S>55) =  0.00272
```

[29]:
```
Text(0, 0.5, 'Probability')
```



```
[62]: a = np.array([ [1,2],[3,4] ])
      np.shape(a[0:0,:])
```

```
[62]: (0, 2)
```

```
[ ]:
```

Project 1_4

October 20, 2021

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

```
[3]: def generate_gaussian_steps(num_steps):
         """Generate X_1 to X_n that are iid gaussian """
         X = np.random.normal(size=num_steps)
         return X

     def generate_gaussian_steps_dt(num_steps, dt):
         """Generate X_1 to X_n that are iid gaussian """
         X = np.random.normal(size=num_steps)
         return X*np.sqrt(dt)

     def generate_random_walk_path(steps, n):
         """Generate a random walk that is shrunk in space and sped up in time from␣
      ↪a sequence of steps
             steps size: n*t
             return size: t
         """
         Y = np.array([0])
         for k in range(n):
             Y = np.concatenate( (Y, np.cumsum(steps[0:k+1])/np.sqrt(n) ) )
         return Y
```

0.0.1 Question 1

$$\Delta W_n = W(t_{n+1}) - W(t_n) \tag{1}$$
$$Y_t = Y_0 \exp{(\mu - \sigma^2/2)t + \sigma W_t} \tag{2}$$

```
[ ]:
```

0.1 MLMC Questions

0.1.1 1. Simulate Geometric Brownian Motion

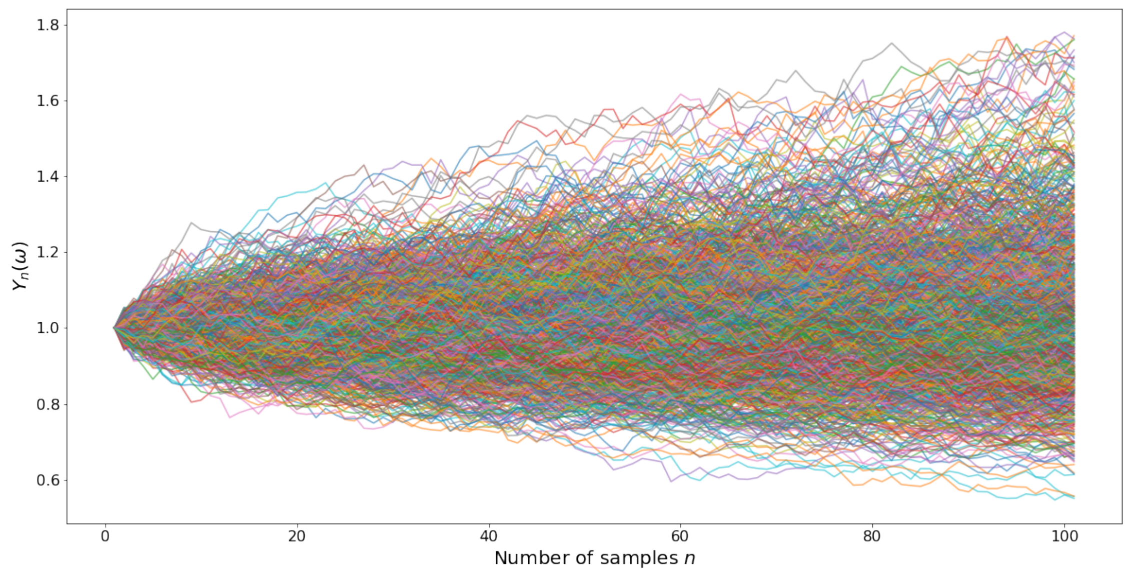
[4]:
```
def generate_geometric_brownian_motion(dW, N, dt):
    '''
    Euler Maruyama IntEgration
    '''
    time = [0]
    Y = [1.0]
    mu = 0.05
    sigma = 0.2
    for n in range(N):
        Y_n = Y[-1]
        bt = mu*Y_n
        ht = sigma*Y_n
        delta_W = dW[n]
        Y_next = Y_n + bt*dt + ht*delta_W
        Y.append(Y_next)
        t = time[-1] + dt
        time.append(t)
    return np.asarray(Y), time
```

0.1.2 4.1 (1)

[5]:
```
TF = 1
N_DIVISIONS = 100
dt = TF/N_DIVISIONS
num_trials = 1000
num_samples = int(np.ceil(TF/dt))
estimator_vals = np.zeros((num_trials, num_samples+1))
for trial in range(num_trials):
    delta_W = generate_gaussian_steps_dt(num_samples,dt)  # Use these as Delta␣
 ↪W_n
    estimator_vals[trial, :], _ = generate_geometric_brownian_motion(delta_W,␣
 ↪num_samples, dt)
```

[6]:
```
print("shape",np.shape(estimator_vals))
plt.figure(figsize=(20,10))
plt.plot(np.arange(1, num_samples+2), estimator_vals.T, alpha=0.6)
plt.ylabel(r'$Y_n(\omega)$', fontsize=20)
plt.xlabel(r'Number of samples $n$', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.show()
```

```
shape (1000, 101)
```



[7]:
```
index_1 = int(1.0/dt)
print("Mean at 1 = ", np.mean(estimator_vals[:,index_1]))
print("Variance at 1 = ", np.var(estimator_vals[:,index_1]))
```

```
Mean at 1 =  1.0365666689542596
Variance at 1 =  0.0441650346300319
```

Analytic Mean of Y(1)

$$Y_t = Y_0 e^{(\mu-\sigma^2/2)t+\sigma W_t} \tag{3}$$
$$E[Y_t] = Y_0 e^{\mu-\sigma^2/2} E[e^{\sigma W_t}] \tag{4}$$

0.1.3 4.1 (2)

[8]:
```
def brownian_fine_to_coarse(T, dt, brownian_fine, M):
    num_samples =  int(np.ceil(T/dt)) + 1
    num_paths = brownian_fine.shape[0]
    brownian_coarse = np.zeros((num_paths, num_samples))
    brownian_coarse[:,0] = brownian_fine[:,0]
    for ii in range(1, num_samples):
        delta = brownian_fine[:, ii * M] - brownian_fine[:, (ii-1)*M]
        brownian_coarse[:, ii] = brownian_coarse[:, ii-1] + delta
    return brownian_coarse
```

0.1.4 (2) Now simulate 2 processes with different dts

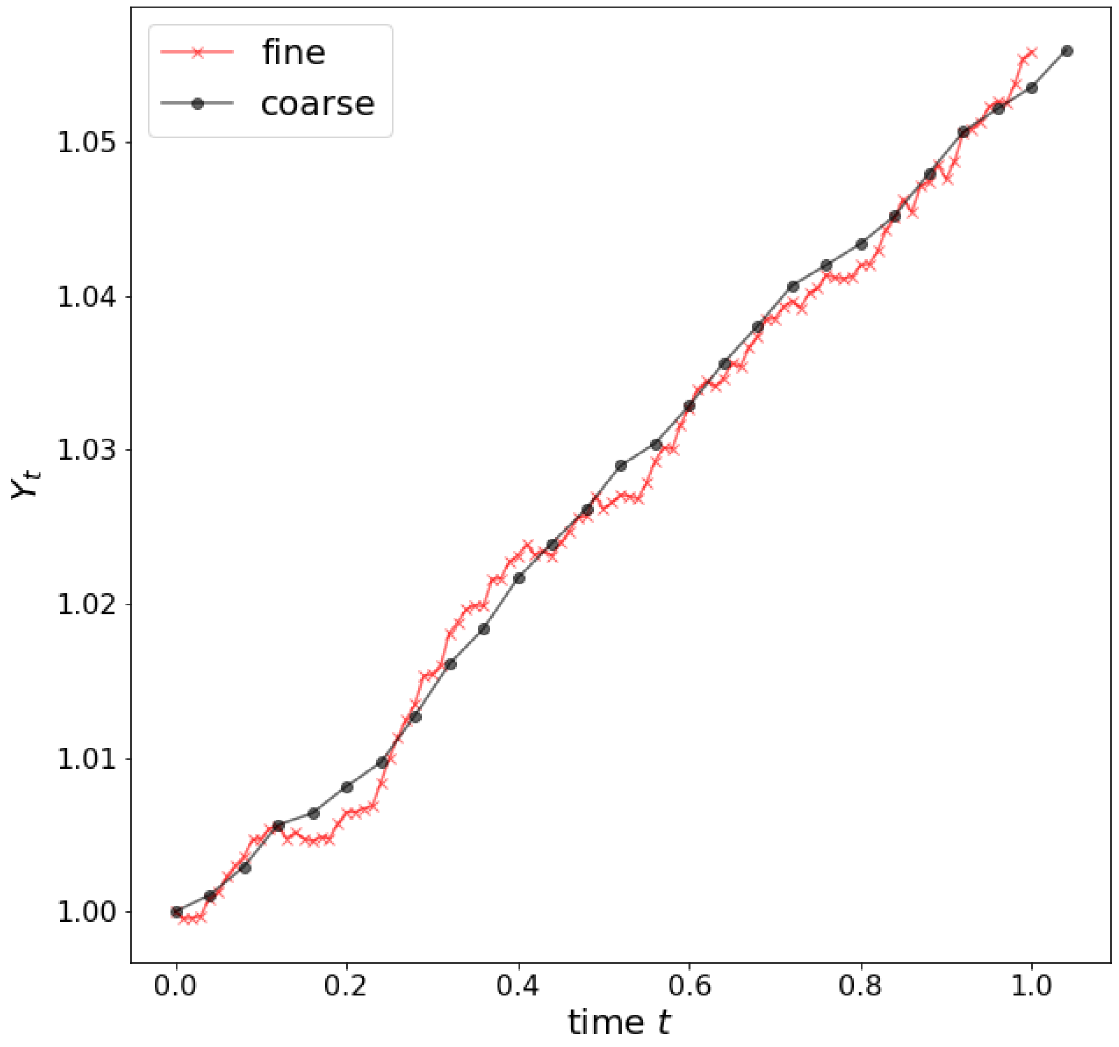
[9]:
```
M = 4
DT =  0.01  # fine scale time step
DTM = DT * M # coarse scale time step
TFINAL = 1.0 # final time
TSPAN_COARSE = np.arange(0, TFINAL + DTM, DTM)
TSPAN_FINE = np.arange(0, TFINAL+DT, DT)
# delta_W = generate_gaussian_steps_dt(num_samples,DT)  # Use these as Delta W_n
# brownian_fine = generate_geometric_brownian_motion(delta_W, num_samples)

num_trials = 1000
brownian_fine = np.zeros((num_trials, num_samples+1))
geometric_fine = np.zeros((num_trials, num_samples+1))
for trial in range(num_trials):
    brownian_fine[trial, :] = generate_gaussian_steps_dt(num_samples+1,DT)  #␣
 ↪Use these as Delta W_n
    geometric_fine[trial, :], times_fine =␣
 ↪generate_geometric_brownian_motion(brownian_fine[trial,:], num_samples, DT)

print("shapes")
print(np.shape(geometric_fine),len(times_fine))
brownian_coarse = brownian_fine_to_coarse(TFINAL, DTM, brownian_fine, M)

num_samples_coarse = int(np.ceil(TFINAL/DTM)) + 1
geometric_coarse = np.zeros((num_trials, num_samples_coarse+1))
for trial in range(num_trials):
    geometric_coarse[trial, :], times_coarse =␣
 ↪generate_geometric_brownian_motion(brownian_coarse[trial,:],␣
 ↪num_samples_coarse, DTM)

# Geometric motion
```

```
shapes
(1000, 101) 101
```

0.1.5 Plotting means of fine and coarse motions

[10]:
```
print("shape",np.shape(estimator_vals))
plt.figure(figsize=(10,10))
plt.plot(times_fine, np.mean(geometric_fine,axis=0), 'r-x', alpha=0.6,␣
 ↪label='fine')
plt.plot(times_coarse, np.mean(geometric_coarse,axis=0), 'k-o', alpha=0.6,␣
 ↪label='coarse')
plt.legend(fontsize=20)
plt.ylabel(r'$Y_t$', fontsize=20)
plt.xlabel(r'time $t$', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.show()
```

```
shape (1000, 101)
```



0.1.6 (3) Multi Level Monte Carlo

Level 0 is lowest fidelity model Level 3 is high fidelity model

$$E[X_L] = \underbrace{E[X_0]}_{level\ 0\ term} + \underbrace{E[X_1 - X_0]}_{level\ 1\ term} + \underbrace{E[X_2 - X_1]}_{level\ 2\ term} + \underbrace{E[X_3 - X_2]}_{level\ 3\ term} \tag{5}$$

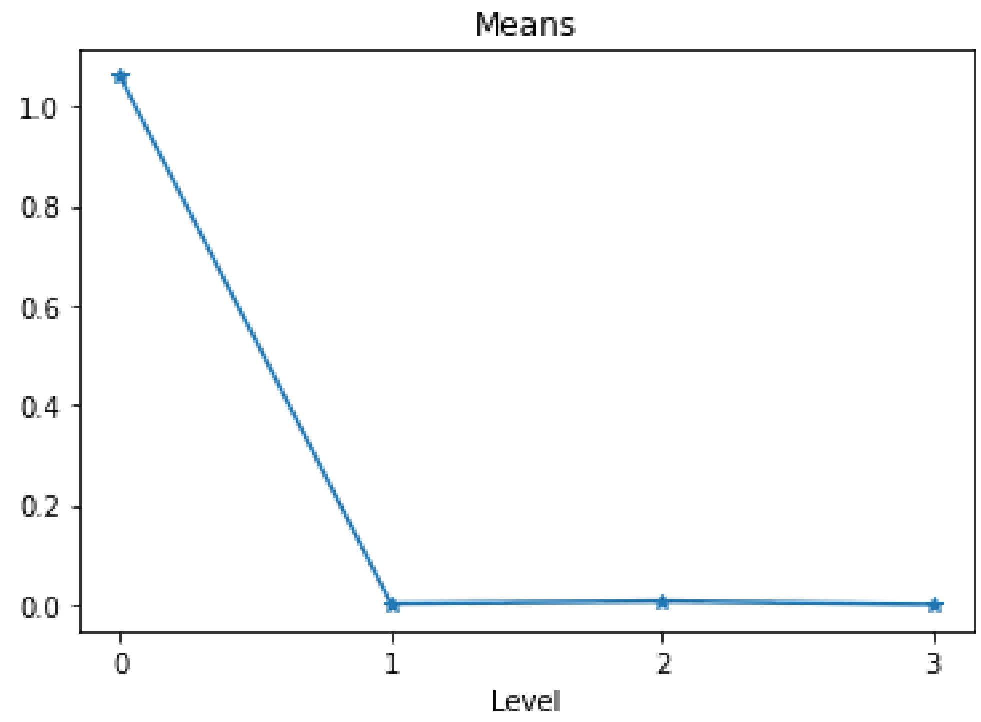
We expect variance to reduce with each level. The plot below will verify that. Each of term above should have independent samples. However, within each term, say $X_1 - X_0$, they should be generated from same underlying process. Therefore in this term X_0 is produced using coarsed version of X_1's brownian motion component as done above in part (2)

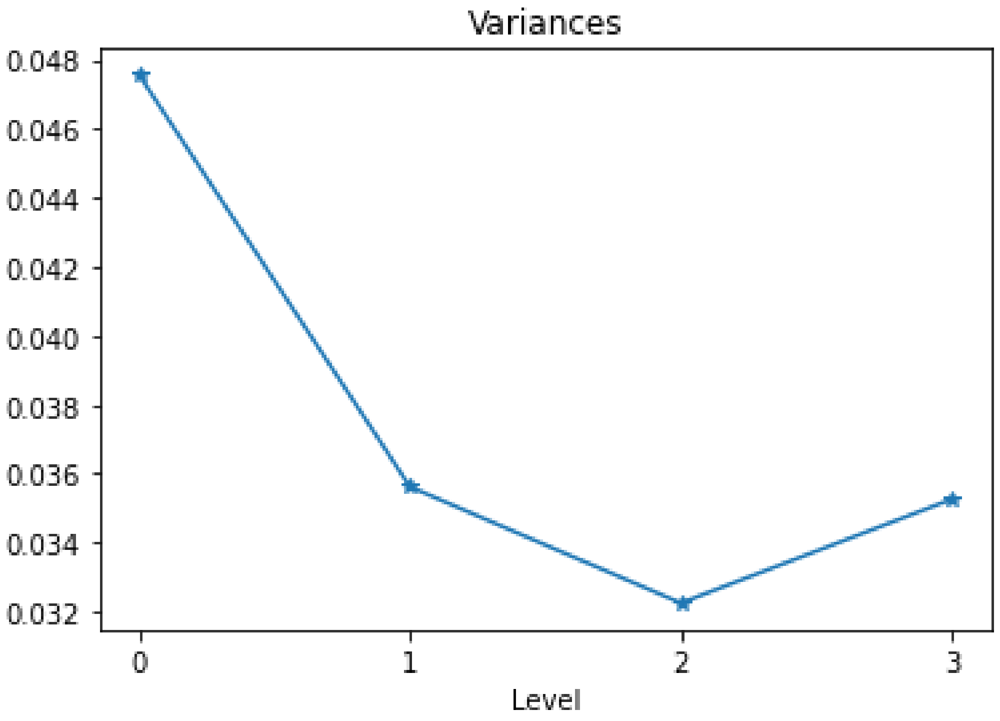
```
[15]: def generate_geometric_motion(DT, TFINAL, num_trials):
          num_samples = int(np.ceil(TFINAL/DT)) + 1
          brownian_ = np.zeros((num_trials, num_samples+1))
          geometric_ = np.zeros((num_trials, num_samples+1))
          for trial in range(num_trials):
              brownian_[trial, :] = generate_gaussian_steps_dt(num_samples+1,DT)  #␣
       ↪Use these as Delta W_n
              geometric_[trial, :], times_fine =␣
       ↪generate_geometric_brownian_motion(brownian_[trial,:], num_samples, DT)
          return geometric_, brownian_

      def generate_coarse_geometric_from_brownian(brownian_fine, TFINAL, DT1, DT2,␣
       ↪num_trials):
          brownian_coarse = brownian_fine_to_coarse(TFINAL, DT2, brownian_fine,␣
       ↪int(DT2/DT1))
          num_samples_coarse = int(np.ceil(TFINAL/DT2)) + 1
          geometric_coarse = np.zeros((num_trials, num_samples_coarse+1))
          for trial in range(num_trials):
              geometric_coarse[trial, :], times_coarse =␣
       ↪generate_geometric_brownian_motion(brownian_coarse[trial,:],␣
       ↪num_samples_coarse, DT2)
          return geometric_coarse


      TFINAL = 1.0 # final time

      DT1 = 4**(-2)  # time step 1
      DT2 = 4**(-3)  # time step 2
      DT3 = 4**(-4)  # time step 3
      DT4 = 4**(-5)  # time step 4

      num_trials = 1000

      # Level 0
      geometric_fine, _ = generate_geometric_motion(DT1, TFINAL, num_trials)
      level_0 = geometric_fine[:,-1] # data only for last (T=1) element

      # Level 1
      geometric_fine, brownian_fine = generate_geometric_motion(DT2, TFINAL,␣
       ↪num_trials)
      geometric_coarse = generate_coarse_geometric_from_brownian(brownian_fine,␣
       ↪TFINAL, DT2, DT1, num_trials)
      level_1 = geometric_fine[:,-1] - geometric_coarse[:,-1]
```

```
# Level 2
geometric_fine, brownian_fine = generate_geometric_motion(DT3, TFINAL,␣
 ↪num_trials)
geometric_coarse = generate_coarse_geometric_from_brownian(brownian_fine,␣
 ↪TFINAL, DT3, DT2, num_trials)
level_2 = geometric_fine[:,-1] - geometric_coarse[:,-1]

# Level 3
geometric_fine, brownian_fine = generate_geometric_motion(DT4, TFINAL,␣
 ↪num_trials)
geometric_coarse = generate_coarse_geometric_from_brownian(brownian_fine,␣
 ↪TFINAL, DT4, DT3, num_trials)
level_3 = geometric_fine[:,-1] - geometric_coarse[:,-1]

# Expectation at Y(1) with MLMC
Y1 = np.mean(level_0) + np.mean(level_1) + np.mean(level_2) + np.mean(level_3)
print("Y1", Y1)

# Expectations of each level across
means = [np.mean(level_0), np.mean(level_1), np.mean(level_2), np.mean(level_3)]
print("Means", means)
plt.figure()
x = [0,1,2,3]
plt.plot(x,means,'*-')
plt.xticks(np.arange(min(x), max(x)+1, 1.0))
plt.title("Means")
plt.xlabel('Level')

# Variances of each level
variances = [np.std(level_0)**2, np.std(level_1)**2, np.std(level_2)**2, np.
 ↪std(level_3)**2]
print("Variances", variances)
plt.figure()
x = [0,1,2,3]
plt.plot(x,variances,'*-')
plt.xticks(np.arange(min(x), max(x)+1, 1.0))
plt.title("Variances")
plt.xlabel('Level')
```

```
Y1 1.0662281309844244
Means [1.0592454180713105, 0.0011965761126632632, 0.006380212350202724,
-0.0005940755497526932]
Variances [0.047567150886924425, 0.035619675745645826, 0.032245239950681195,
0.035260899868914]
```

```
[15]: Text(0.5, 0, 'Level')
```


Means
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
Level


Variances
0.048
0.046
0.044
0.042
0.040
0.038
0.036
0.034
0.032
0
1
2
3
Level

The variance with each level reduces. That's good! The level_0 mean is high and is representative of true values. The remaining means are correction terms and are therefore close to 0

0.1.7 (4) Theoretical Cost of MLMC estimator

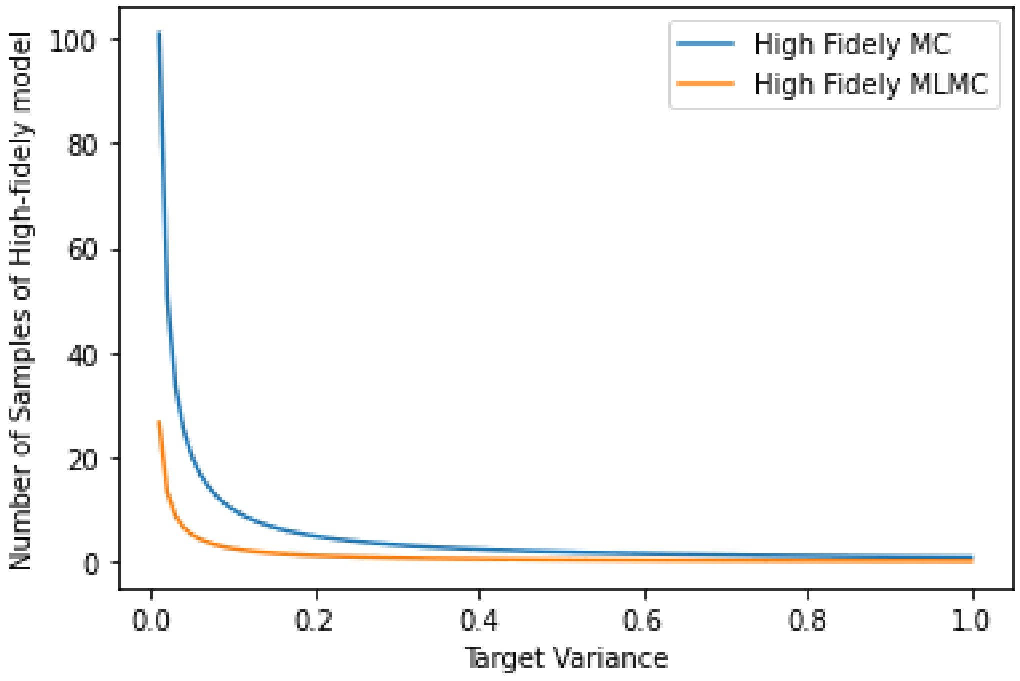
Copy from supplementary material given in course on MLMC

0.1.8 (5) Plot equalent samples of MC and MLMC

[35]:
```
print( np.sqrt(1*variances[0]) + np.sqrt(2*variances[1]) + np.
 ↪sqrt(2*variances[2]) + np.sqrt(2*variances[3]) )
epsilon_sq = np.linspace(0.01,1,100)
sum_roots =  np.sqrt(1*variances[0]) + np.sqrt(2*variances[1]) + np.
 ↪sqrt(2*variances[2]) + np.sqrt(2*variances[3])
N3_mc = 1/epsilon_sq*sum_roots**2
N3_mlmc = 1/epsilon_sq * sum_roots*  np.sqrt( variances[3]*2 )
plt.plot(epsilon_sq,N3_mc, label='High Fidely MC')
plt.plot(epsilon_sq,N3_mlmc, label='High Fidely MLMC')
plt.legend()
plt.xlabel('Target Variance')
plt.ylabel('Number of Samples of High-fidely model')
```

```
1.0045151242115722
```

[35]:
```
Text(0, 0.5, 'Number of Samples of High-fidely model')
```


High Fidely MC
High Fidely MLMC
Number of Samples of High-fidely model
Target Variance