2017

A New Approximation Scheme for Monte Carlo Applications

Bo Jones
Claremont McKenna College

Claremont McKenna College

A New Approximation Scheme for Monte Carlo Applications

submitted to
Professor Mark Huber

by
Bo Jones

for
Senior Thesis
Academic Year 2016/2017
April 21, 2017

A NEW APPROXIMATION SCHEME FOR MONTE CARLO APPLICATIONS

BO JONES

Abstract. Approximation algorithms employing Monte Carlo methods, across application domains, often require as a subroutine the estimation of the mean of a random variable with support on $[0, 1]$. One wishes to estimate this mean to within a user-specified error, using as few samples from the simulated distribution as possible. In the case that the mean being estimated is small, one is then interested in controlling the relative error of the estimate. We introduce a new (ϵ, δ) relative error approximation scheme for $[0, 1]$ random variables and provide a comparison of this algorithm's performance to that of an existing approximation scheme, both establishing theoretical bounds on the expected number of samples required by the two algorithms and empirically comparing the samples used when the algorithms are employed for a particular application.

Contents

Date: April 21, 2017.

1. Introduction

It is common across a wide range of problem areas to encounter Monte Carlo approximation algorithms that require as a subroutine the estimation of a mean of a simulated random variable with support on $[0, 1]$. In fact we will motivate the work presented in this paper with such an approximation algorithm coming from the domain of network science.

For these problems one is interested in employing an approximation scheme for estimating the mean that achieves a user-specified bound on the probability of exceeding a user-specified error, that is an (ϵ, δ) approximation of the mean. In the case that the mean to be estimated is small, a meaningful measure of error is the relative error. In a Monte Carlo setting one is able to determine the number of samples on which to base an estimate, simply generating that many samples by simulation, but sampling from the distribution in question has an inherent computational cost. It is beneficial then, in designing approximation schemes, to minimize the number of samples needed.

We first introduce the problem of (ϵ, δ) approximation, and the question of how many samples are needed to achieve it. In Section 3 we present an existing (ϵ, δ) approximation scheme introduced in [3]. In Section 5 we present a new approximation scheme that makes use of a new estimator, introduced by [1] and described in Section 4, for the mean of $[0, 1]$ random variables. Section 6 explores potential areas of improvement for a particular step of the new approximation scheme.

We compare the performance of the new approximation scheme to that of the existing approximation scheme. In Section 7 we establish a lower bound on the expectation of the number of samples required to generate an estimate with the desired error for the existing algorithm, and prove an upper bound on the expectation of the number of samples required by the new algorithm.

In Section 8 we present an empirical comparison of the number of samples required when the algorithms are used on both test data and data arising from an approximation algorithm introduced in [7].

2. Problem: Mean Approximation to Specified Error

Consider random variables $Z_1, Z_2, \ldots \overset{\text{iid}}{\sim} Z$ for some distribution Z with support on $[0, 1]$. We wish to estimate the mean of Z, denoted μ_Z, using as few samples $Z_1, Z_2, \ldots$ as possible. Having designed an estimator $\hat{\mu}_Z$ for μ_Z we can ask the question of how close our estimator is to the true mean value. We may want to capture this information in the difference $\mu_Z - \hat{\mu}_Z$, called the absolute error of our estimate.

However as μ_Z becomes very small, absolute error has less and less meaning. It then becomes useful to examine the relative error, defined as

$$\epsilon_{\text{rel}} = \frac{\hat{\mu}_Z}{\mu_Z} - 1.$$

Now consider the case where the Z_i are not given, but rather the random variable Z arises in a Monte Carlo application in which the variates Z_i are simulated draws from Z. Then the question becomes not how confident can I be about my estimator given the samples that I have seen, but rather, if I can devise a random process that estimates μ_Z using samples drawn from Z, how many samples must I generate in order to be so confident that the error of my estimator will fall below a certain value. We can formulate this question as the question of how many samples are needed to generate an estimate $\hat{\mu}_Z$ such that $\hat{\mu}_Z$ is an (ϵ, δ) relative error approximation of μ_Z, that is an approximation such that

$$\mathbb{P}\left(\left|\frac{\hat{\mu}_Z}{\mu_Z} - 1\right| > \epsilon\right) \leq \delta.$$

A natural candidate for the estimator $\hat{\mu}_Z$ is the sample mean, given by

$$\hat{\mu}_Z = \frac{Z_1 + Z_2 + \cdots + Z_n}{n}.$$

In [3] Dagum, Karp, Luby and Ross prove the following theorem regarding the sample mean estimator.

Theorem 1 (The Generalized Zero-One Estimator Theorem [3])**.** *Let $Z_1, Z_2, \ldots, Z_n$ denote random variables that are independent and identically distributed according to Z. Let $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$.*
If $\epsilon < 1$ and

$$n = 4(e-2)\ln(2\delta^{-1})\rho_Z/(\epsilon\mu_Z)^2,$$

then

$$\mathbb{P}\left[(1-\epsilon)\mu_Z \leq \frac{\sum_{i=1}^n Z_i}{n} \leq (1+\epsilon)\mu_Z\right] > 1 - \delta.$$

In fact, by an argument similar to that given by Huber in [5] for Bernoulli random variables, the Central Limit Theorem tells us that the *minimum* number of samples required ought to be approximately $2\ln(2\delta^{-1})\epsilon^{-2}\mu_Z^{-1}$. This is the number of samples we would need if we could assume that the data was normally distributed.

However, both the number of samples given by the Generalized Zero-One Estimator Theorem and the Central Limit Theorem heuristic are dependent upon the unknown value μ_Z that we are trying to estimate. We now present an existing algorithm designed to overcome this problem of dependence on actual value by employing a initial estimates of both μ_Z and the variance of Z, denoted σ_Z^2.

3. An Existing Algorithm

In [3] Dagum et al. introduce an algorithm which uses an initial $(\min\{1/2, \sqrt{\epsilon}\}, \delta/3)$ approximation of μ_Z, obtained using their Stopping Rule Algorithm, and an initial estimate of σ_Z^2 in order to determine the number of samples needed in order for the sample mean to be an (ϵ, δ) approximation of μ_Z.

Algorithm 1. ($\mathcal{AA}$ algorithm in [3]).
Inputs: ϵ, error parameter.
δ, error parameter.
Output: $\hat{\mu}$, an estimate for μ_Z.

Let $\{Z_i\}$, $\{Z_i'\}$, and $\{Z_i''\}$ denote three sets of random variables independently and identically distributed according to Z.

Step 1: Use the Stopping Rule Algorithm of [3], drawing random variables $Z_1, Z_2 \ldots$ to obtain a $(\min\{1/2, \sqrt{\epsilon}\}, \delta/3)$ approximation $\hat{\mu}_1$ of μ_Z.
Step 2: Set $N_2 = 8(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\log(3/\delta)(1/(\epsilon\hat{\mu}_1))$
and initialize $S \leftarrow 0$.
For $i = 1, \ldots, N_2$ do: $S \leftarrow S + (Z'_{2i-1} - Z'_{2i})^2/2$.
$\hat{\rho}_Z \leftarrow \max\{S/N_2, \epsilon\hat{\mu}_1\}$.
Step 3: Set $N_3 = 8(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\log(3/\delta)(\hat{\rho}_Z/(\epsilon\hat{\mu}_1)^2)$.
and initialize $S \leftarrow 0$.
For $i = 1, \ldots, N_3$ do: $S \leftarrow S + Z_i''$.
$\hat{\mu}_Z \leftarrow S/N_3$.

We will call Algorithm 1 DKLR. Dagum et al. establish that DKLR provides an (ϵ, δ) approximation. Using the Sequential Probability Ratio Test of Wald [6], they also prove the following result.

Theorem 2. *(Lower Bound Theorem Part 3 in [3]).*
Let $\rho_Z = \max\{\sigma_Z^2, \epsilon\mu_Z\}$. For any randomized approximation scheme that yields an (ϵ, δ) approximation of μ_Z, let T be the number of samples required by the approximation scheme.
There is a universal constant c such that

$$\mathbb{E}[T] \geq c4(e-2)\ln(2/\delta)\frac{\rho_Z}{(\epsilon\mu_Z)^2}.$$

Dagum et al. show that DKLR requires an expected number of samples that is less than some constant multiple of this universal lower bound. Thus improving upon the number of samples needed is a question of reducing the constant coefficients that are present in the number of samples needed for DKLR.

4. A New Estimator for (ϵ, δ) Approximation

We develop an alternative (ϵ, δ) approximation scheme for the mean of random variables with support on $[0, 1]$, using as our estimator, not the sample mean, but the M-estimator introduced by Catoni in [1].

Catoni's M-estimator is an estimate for μ_Z which is given by the parameter value $\hat{\mu}_Z$ that, instead of minimizing the mean square error, minimizes an implicit error function that accounts for deviations. The influence function ψ that Catoni constructs to capture this

implicit error is given by

$$\psi(x) = \begin{cases} \ln(1 + x + x^2/2), & x \geq 0, \\ -\ln(1 - x + x^2/2), & x \leq 0. \end{cases}$$

The estimator $\hat{\mu}_Z$ is then the solution of the equation

$$\sum_{i=1}^{n} \psi[\alpha(Z_i - \hat{\mu}_Z)] = 0,$$

for some number of samples n and parameter α to be specified.

The M-estimator so constructed can be used to obtain an (ϵ, δ) randomized approximation scheme for estimating μ_Z. The key is the following lemma.

Lemma 1 (Lemma 2.3 in [1])**.** *Suppose $\alpha^2\sigma^2 + 2\ln(2/\delta)/n \leq 1$. Then the M-estimator of Catoni $\hat{\mu}$ satisfies*

$$\mathbb{P}\left(|\hat{\mu} - \mu| \geq \eta(\alpha, \sigma^2, n)\right) \leq \delta,$$

where

$$\eta(\alpha, \sigma^2, n) = \left(\frac{\alpha\sigma^2}{2} + \frac{\ln(2/\delta)}{\alpha n}\right)\left(\frac{1}{2} + \frac{1}{2}\sqrt{1 - \alpha^2\sigma^2 - \frac{2\ln(2/\delta)}{n}}\right)^{-1} \tag{1}$$

Note that the η function is increasing in σ^2. So if we replace it with an upper bound on the variance: $\sigma^2 \leq b_1$, then the function becomes larger, giving the following corollary.

Corollary 1. *Suppose $\sigma^2 \leq b_1$, and $\alpha^2 b_1 + 2\ln(2/\delta)/n \leq 1$. Then the M-estimator of Catoni $\hat{\mu}$ satisfies*

$$\mathbb{P}\left(|\hat{\mu} - \mu| > \eta(\alpha, b_1, n)\right) \leq \delta,$$

where η is defined as before.

This gives rise to the following procedure for implementing Catoni's M-estimator and an (ϵ, δ)-ras.

Lemma 2. *Suppose $\sigma^2 \leq b_1$, $\mu \geq b_2$, and $(\epsilon b_2)^2 \leq 1/2$. Then let*

$$n = 2\left(\frac{b_1}{b_2^2}\epsilon^{-2} + 1\right)\ln\left(\frac{2}{\delta}\right)$$

and

$$\alpha = \frac{\epsilon b_2}{b_1 + (\epsilon b_2)^2}.$$

With this (n, α), the Catoni M-estimator $\hat{\mu}$ satisfies

$$\mathbb{P}(|\hat{\mu} - \mu| > \epsilon\mu) \leq \delta.$$

Proof. Let $\epsilon b_2 = b_3$. Then n and α can be written

$$n = 2\left(\frac{b_1 + b_3^2}{b_3^2}\right)\ln\left(\frac{2}{\delta}\right), \quad \alpha = \frac{b_3}{b_1 + b_3^2}$$

Therefore $2\ln(2/\delta)/n = b_3\alpha$ and

$$\eta(\alpha, b_1, n) \leq \left(\frac{\alpha b_1 + b_3}{2}\right)\left(\frac{1}{2} + \frac{1}{2}\sqrt{1 - \alpha^2 b_1 - b_3\alpha}\right)^{-1}$$

Since $\alpha(b_1 + b_3^2) = b_3$, multiplying top and bottom by $2(b_1 + b_3^2)$ gives

$$\begin{aligned}\eta(\alpha, b_1, n) &= \frac{b_3 b_1 + b_3(b_1 + b_3^2))}{b_1 + b_3^2 + \sqrt{(b_1 + b_3^2)^2 - b_3^2 b_1 - b_3(b_3)(b_1 + b_3^2)}} \\ &= b_3 \frac{b_1 + b_1 + b_3^2}{b_1 + b_3^2 + \sqrt{b_1^2 + 2b_1 b_3^2 + b_3^4 - b_3^2 b_1 - b_3^2 b_1 - b_3^4}} \\ &= b_3 = \epsilon b_2 \leq \epsilon\mu.\end{aligned}$$

Finally, note that

$$\alpha^2 b_1 + \frac{2\ln(2/\delta)}{n} = \frac{b_1(\epsilon b_2)^2 + b_1(\epsilon b_2)^2 + (\epsilon b_2)^4}{b_1 + (\epsilon b_2)^2} \leq 2(\epsilon b_2)^2.$$

Therefore, when $(\epsilon b_2)^2 \leq 1/2$, we have $\alpha^2 b_1 + 2\ln(2/\delta)/n \leq 1$, and the proof is complete. □

Thus if we can obtain the bounds, b_1 and b_2, on the variance and mean of Z, we can estimate μ_Z to (ϵ, δ) relative error using $2(b_1/(\epsilon b_2)^2 + 1)\ln(2/\delta)$ samples.

5. A New Approximation Scheme

We now introduce a new approximation scheme that has three steps. In step 1 we produce an initial estimate of μ_Z that will be used to determine the lower bound on μ_Z needed to employ the M-estimator introduced in the previous section. In step 2 we estimate a value that will be used to determine the upper bound on σ_Z^2 needed to employ the M-estimator. In step 3 we use the M-estimator to obtain an (ϵ, δ) approximation.

Algorithm 2. (New Approximation Scheme).

Inputs:	ϵ,	error parameter (must satisfy $\epsilon < 2^{-3/2} \approx .35$).
	δ,	error parameter.
	γ,	error parameter for variance estimate (must satisfy $\gamma > 1 - 1/\sqrt{2} \approx .3$).
	w,	as demonstrated in Section 7, this parameter can be used to weight the contributions of step 2 vs step 3 in the runtime.
Output:	$\hat{\mu}_Z$,	an estimate for μ_Z.

Let $\{Z_i\}$, $\{Z_i'\}$, and $\{Z_i''\}$ denote three sets of random variables independently and identically distributed according to Z. Let the random variables W_i be Bernoulli random variables drawn from distribution given by, for $U \sim \mathsf{Unif}([0,1])$, $W = \mathbb{1}(U \leq X)$. By a well known result, stated as Lemma 8 in [5], $W \sim \mathsf{Bern}(\mu_X)$.

Step 1: Use the biased GBAS algorithm of [4], drawing $W_1, W_2, \ldots \sim \mathsf{Bern}(\mu_Z)$ using $Z_1, Z_2 \ldots$ with

$$c_1 = \frac{2\epsilon^{1/3}}{(1-\epsilon^{2/3})\ln(1+2\epsilon^{1/3}/(1-\epsilon^{1/3}))}$$

and

$$k_1 = \left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)\epsilon^{-2/3}, \left(\epsilon^{-1/3}+\frac{2}{3}\right)^2 \right\} \right\rceil + 1$$

to obtain an approximation $\hat{\mu}_1$ of μ_Z.

Step 2: Let $a = \sigma_Z^2 + w\epsilon\hat{\mu}_1$.
Use the biased GBAS algorithm of [4], drawing $W_1', W_2', \ldots \sim \mathsf{Bern}(\sigma^2 + w\epsilon\hat{\mu}_1)$ using $(Z_1' - Z_2')^2/2, (Z_3' - Z_4')^2/2 \ldots$ with

$$c_2 = \frac{2(1-\gamma)}{(2\gamma-\gamma^2)\ln(1+2(1-\gamma)/\gamma)}$$

and

$$k_2 = \left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)(1-\gamma)^{-2}, \left((1-\gamma)^{-1}+\frac{2}{3}\right)^2 \right\} \right\rceil + 1$$

to obtain an approximation $\hat{a}$ of a.

Step 3: Use the Catoni M-estimator, drawing $Z_1'', Z_2'' \ldots Z_n''$, with
variance upper bound $b_1 = \hat{a}/\gamma - w\epsilon\hat{\mu}_1$,
mean lower bound $b_2 = \hat{\mu}_1/(1+\epsilon^{1/3})$,
to obtain an $(\epsilon, \delta/3)$ approximation $\hat{\mu}_Z$ of μ_Z.

Theorem 3. *The estimator $\hat{\mu}_Z$ generated by Algorithm 2 is an (ϵ, δ) approximation of μ_Z.*

Proof. Say that step 1 is successful if $\hat{\mu}_1/(1+\epsilon^{1/3}) < \mu_Z$. Say that step 2 is successful if $\hat{a}/\gamma > a$. Say that step 3 is successful if $\hat{\mu}_Z \in [(1-\epsilon)\mu_Z, (1+\epsilon)\mu_Z]$.

Let S_i denote the event that the ith step is a success. We are interested in the probability that all three steps succeed, i.e. $\mathbb{P}(S_1, S_2, S_3)$. This can be rewritten

$$\mathbb{P}(S_1, S_2, S_3) = \mathbb{P}(S_3|S_2, S_1)\mathbb{P}(S_2|S_1)\mathbb{P}(S_1).$$

In order to establish the values of these conditional probabilities we will make use of the following result of Huber.

Lemma 3. *(Lemma 6 in [4]).*

Let $\hat{\mu}_c$ denote the estimate for μ obtained using the biased GBAS algorithm with bias parameter c. If

$$c = \frac{2\epsilon}{(1-\epsilon^2)\ln(1+2\epsilon/(1-\epsilon))},$$

then, letting $f(t) = te^{1-t}$,

$$\mathbb{P}\left(\left|\frac{\hat{\mu}_c}{\mu} - 1\right| > \epsilon\right) \leq \frac{1}{\sqrt{2\pi(k-1)}} \cdot \frac{1}{1-c(1-\epsilon)} \left(f\left(\frac{1}{c(1-\epsilon)}\right)\right)^{k-1}.$$

The following corollary serves to simplify this bound.

Corollary 2. *If*

$$\frac{1}{\sqrt{(k-1)}} \cdot \frac{1}{1-c(1-\epsilon)} < 1$$

then

$$\mathbb{P}\left(\left|\frac{\hat{\mu}_c}{\mu} - 1\right| > \epsilon\right) \leq \frac{2}{\sqrt{2\pi}} \left(e^{-\epsilon^2/2}\right)^{k-1}.$$

Proof. The condition of the corollary immediately yields the coefficient $2/\sqrt{2\pi}$. The $e^{-\epsilon^2/2}$ term is obtained by taking the Taylor series expansion of $f(1/(c(1-\epsilon)))$ and observing

$$\begin{aligned} \frac{e^{1-\frac{1}{c(1-\epsilon)}}}{c(1-\epsilon)} &= 1 - \frac{\epsilon^2}{2} + x && \text{(where } x < 0\text{)} \\ &\leq e^{-\epsilon^2/2}. \end{aligned}$$

□

We can then establish a second corollary concerning k.

Corollary 3. *For a given error parameter δ. If*

$$k > \max\left\{2\ln(2/(\sqrt{2\pi}\delta))\epsilon^2, \left(\frac{1}{\epsilon} + \frac{2}{3}\right)^2\right\} + 1,$$

then

$$\mathbb{P}\left(\left|\frac{\hat{\mu}_c}{\mu} - 1\right| > \epsilon\right) < \delta.$$

Proof. Consider the condition for Corollary 2,

$$\frac{1}{\sqrt{(k-1)}} \cdot \frac{1}{1-c(1-\epsilon)} < 1.$$

This is equivalent to

$$k > \left(\frac{1}{1-c(1-\epsilon)}\right)^2 + 1.$$

Taking the Taylor series expansion of $1/(1-c(1-\epsilon))$ yields

$$\begin{aligned}\frac{1}{1-c(1-\epsilon)} &= \frac{1}{\epsilon} + \frac{2}{3} + x \qquad \text{(where } x < 0\text{)} \\ &\leq \frac{1}{\epsilon} + \frac{2}{3}.\end{aligned}$$

Thus $k > (1/\epsilon + 2/3)^2 + 1$ ensures that this condition is satisfied. Then, applying Corollary 2 for $k > 2\ln(2/(\sqrt{2\pi}\delta))\epsilon^2 + 1$ yields the desired bound δ.

□

Our choice of c_1 and k_1 then, in step 1, allows us to conclude, by Corollary 3, that $\hat{\mu}_1$ is an $(\epsilon^{1/3}, \delta/3)$ approximation of μ_Z, meaning that $\hat{\mu}_1/(1+\epsilon^{1/3}) < \mu_Z$ with probability at least $1-(\delta/3)$, and thus $\mathbb{P}(S_1) > 1-(\delta/3)$.

Now conditioned on step 1 being a success, our choice of c_2 and k_2 for step 2 ensure, by Corollary 3, that $\hat{a}$ is a $(1-\gamma, \delta/3)$ approximation of a, meaning that $\hat{a}/\gamma > a$ with probability at least $1-(\delta/3)$, and thus $\mathbb{P}(S_2|S_1) > 1-(\delta/3)$.

Conditioned on the success of both step 1 and step 2, the bounds b_1 and b_2 are in fact the bounds on μ_Z and σ_Z^2 required for the Catoni M-estimator. Therefore by Lemma 2, $\hat{\mu}_Z$ is an $(\epsilon, \delta/3)$ approximation of μ_Z, and thus $\mathbb{P}(S_3|S_2, S_1) > 1-(\delta/3)$.

Combining the above yields

$$\mathbb{P}(S_1, S_2, S_3) = \mathbb{P}(S_3|S_2, S_1)\mathbb{P}(S_2|S_1)\mathbb{P}(S_1) > (1-(\delta/3))^3.$$

Then $\delta < 1$ allows us to conclude

$$\mathbb{P}(S_1, S_2, S_3) > 1-\delta.$$

□

6. Alternatives for Obtaining an Upper Bound on Variance

In step 2 of the new approximation scheme, Algorithm 2, we obtain an upper bound on the variance of our random variable Z by estimating the value $a = \sigma_Z^2 + w\epsilon\hat{\mu}_1$. We obtain this estimate by drawing $\{0,1\}$ random variables which have mean a and using the GBAS algorithm of [4] to estimate their mean.

It is conceivable that we could have used random variables drawn from a different distribution, having mean a, and in fact the constant $\epsilon\hat{\mu}_1$ term in a presents us with an opportunity to add some determinism to the way in which we make draws from our distribution, thereby reducing the variance of our draws, and allowing us to construct a mean estimator that will require fewer samples in order to realize our $(1-\gamma, \delta/3)$ error bound.

We explore two potential ways of doing this. The first is to design a distribution with lower variance than a $\mathsf{Bern}(a)$ to which we could then apply Catoni's M estimator. The second is to adapt the Stopping Rule Algorithm of [3], considered as an estimator based on the hitting time of a Markov chain, including in our Markov chain a deterministic drift given by $w\epsilon\hat{\mu}_1$.

6.1. **Using Catoni.** We wish to estimate $a = \sigma_Z^2 + w\epsilon\hat{\mu}_Z$ by introducing a random variable H with mean a and distribution given by

$$\mathbb{P}(H = h) = \begin{cases} \frac{1}{2} - \sigma_Z^2, & \text{for } h = 0 \\ \frac{1}{2}, & \text{for } h = 2w\epsilon\hat{\mu}_Z \\ \sigma_Z^2, & \text{for } h = 1. \end{cases}$$

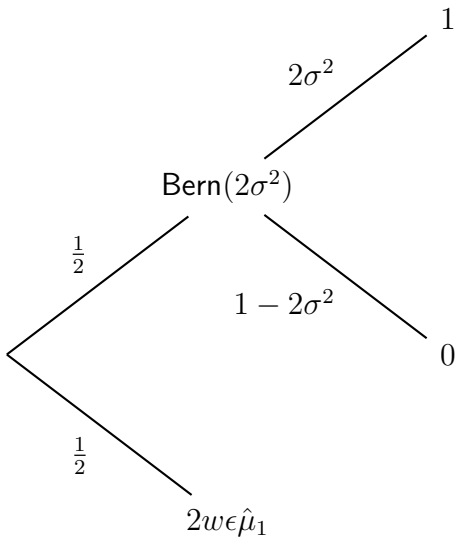

FIGURE 1. Drawing from Distribution H. $\mathbb{E}[H] = \sigma^2 + w\epsilon\hat{\mu}_1$.

The reduction in variance inherent in constructing H this way becomes apparent when we consider drawing from H as a process in which we flip a fair coin, the result of which then determines whether we return the constant value $2w\epsilon\hat{\mu}_1$ or draw a random variable distributed $\mathsf{Bern}(2\sigma^2)$. This process is pictured in Figure 1.

The mean of H can be estimated using Catoni's M-estimator. We are able to bound the ratio appearing in the number, n in Lemma 2, of samples needed to apply this estimator as follows.

Lemma 4. *Letting H be the random variable defined above,*

$$\frac{\mathbb{V}(H)}{\mathbb{E}[H]^2} \leq \frac{1}{4w\epsilon\hat{\mu}_1}.$$

Proof. Let σ_Z^2 be denoted v and let $w\epsilon\hat{\mu}_1$ be denoted m.

$$\begin{aligned}\frac{\mathbb{V}(H)}{\mathbb{E}[H]^2} &= \frac{v(1-2v)+(v-m)^2}{(v+m)^2} \\ &\leq \frac{v+(v-m)^2}{(v+m)^2} \\ &\leq \max_v \left(\frac{v+(v-m)^2}{(v+m)^2}\right).\end{aligned}$$

Let

$$\begin{aligned}g(v) &= \ln\left(\frac{v+(v-m)^2}{(v+m)^2}\right) \\ &= \ln(v+(v-m)^2) - 2\ln(v+m).\end{aligned}$$

Differentiating with respect to the variance of Z yields

$$\begin{aligned}\frac{\partial g}{\partial v} &= \frac{1+2(v-m)}{v+(v-m)^2} - \frac{2}{v+m} \\ &= \frac{(1-4m)(m-v)}{(v+(v-m)^2)(v+m)}.\end{aligned}$$

For $v = m$, $\partial g/\partial v = 0$. In addition $v, m > 0$ ensure that the denominator $(v+(v-m)^2)(v+m)$ is positive. Thus, if $m < .25$, we have

$$\begin{aligned}\partial g/\partial v > 0, \text{ for } 0 < v < m \\ \partial g/\partial v < 0, \text{ for } 0 < m < v.\end{aligned}$$

Thus

$$\max_v \left(\frac{v+(v-m)}{(v+m)^2}\right) = \frac{m+(m-m)}{(m+m)^2} = \frac{1}{4m}.$$

□

Thus were this upper bound on the ratio to be realized for established bounds b_1 and b_2 on the variance and mean of H, we could achieve a $(1/2, \delta/3)$ approximation of a using Catoni's M-estimator using at most

$$2\left(\left(\frac{1}{4w\epsilon\hat{\mu}_Z}\right)\left(\frac{1}{2}\right)^{-2}+1\right)\ln\left(\frac{6}{\delta}\right) = 2\left(\frac{1}{w\epsilon\hat{\mu}_Z}+1\right)\ln\left(\frac{6}{\delta}\right)$$

draws of *of* H. Because, on each draw of H, we have a $1/2$ chance of drawing a Bernoulli that requires 2 draws from Z. This then implies that using this estimate of H, step 2 in the new algorithm would require only at most an expected $2((1/w\epsilon\hat{\mu}_Z)+1)\ln(6/\delta)$ draws from Z.

However, while Catoni's number of samples is concerned with the maximization of this ratio, Catoni also requires b_1 and b_2 which directly bound both the variance and mean of the random variable in order to construct the M-estimator. In this case we are unable to establish bounds b_1 and b_2 that produce this bound on the ratio, or more generally that would produce a number of samples lower than that needed by GBAS in step 2 of our algorithm.

As a matter of future work we are interested in designing an estimator for which knowledge of the $\mathbb{V}(X)/\mathbb{E}[X]^2$ bound established here would be sufficient to bound the number of samples needed to the same degree that Catoni is able to bound the number of samples needed given b_1 and b_2.

6.2. **Using Discrete Markov Chains: Two Alternatives.** For the sake of simplifying notation, for this section we restate the problem of estimating $a = \sigma_Z^2 + w\epsilon\hat{\mu}_1$ as estimating $p+\delta$ where p and δ are nonnegative constants.

We will consider two estimators for $p+\delta$. We construct these estimators by constructing two simple processes. In order to construct the processes, we assume that $p+\delta \leq 1$ and that δ is of the form 1/n for some positive integer n. Note that these are reasonable assumptions for our motivating problem as we are concerned with relative error approximation for random variables with small mean, and we can always approximate small $w\epsilon\hat{\mu}_1$ by 1/n for some n.

For some constant M, the first process, the Markov chain X_t, is given by letting $X_0 = 0$ and constructing transitions as follows.
If $X_{t-1} < M$ then

$$\begin{aligned}\mathbb{P}(X_t = X_{t-1}+1) &= p+\delta\\ \mathbb{P}(X_t = X_{t-1}) &= 1-(p+\delta).\end{aligned}$$

If $X_{t-1} = M$ then

$$\mathbb{P}(X_t = X_{t-1}) = 1.$$

We specify the constant M that is the value of the state at which the chain remains in place.

The second process, the Markov chain Y_t, is given by letting $Y_0 = 0$ and constructing transitions as follows.
If $Y_{t-1} < M$ then

$$\begin{aligned}\mathbb{P}(Y_t = Y_{t-1}+1+\delta) &= p\\ \mathbb{P}(Y_t = Y_{t-1}+\delta) &= 1-p.\end{aligned}$$

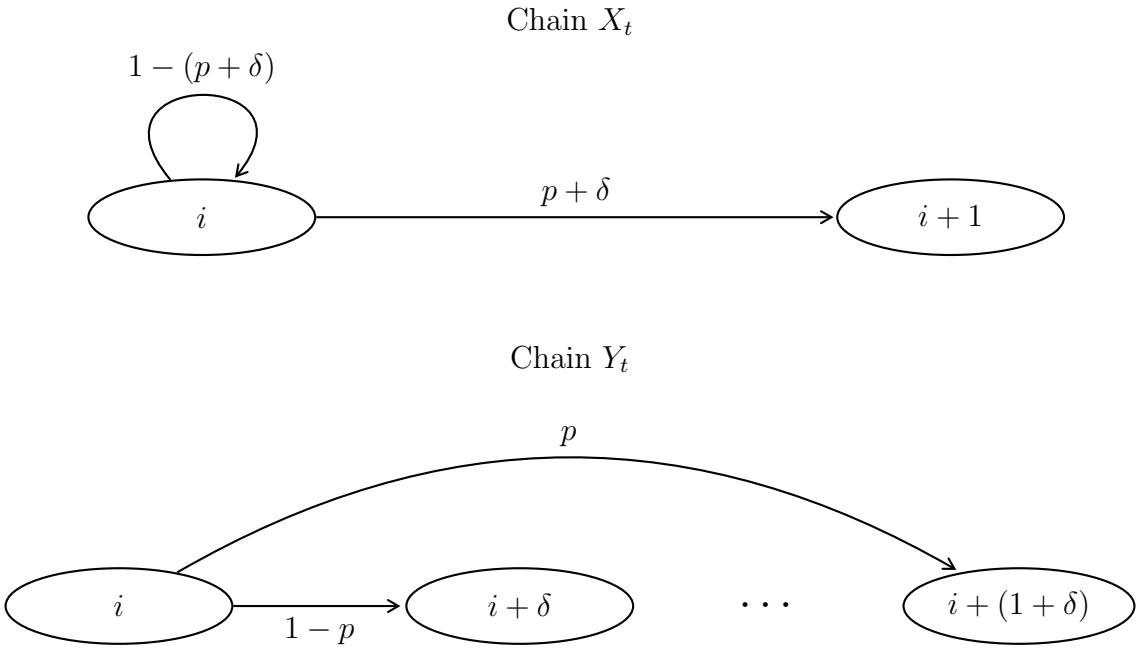

FIGURE 2. Alternative Processes for Estimation of $p+\delta$

If $Y_{t-1} \geq M$ then

$$\mathbb{P}(Y_t = Y_{t-1}) = 1.$$

The key characteristic of these constructions is that, though the chain Y_t uses the δ parameter as a deterministic drift term, for both of the chains we have the expected size of one step is given by

$$\mathbb{E}[X_t - X_{t-1}] = \mathbb{E}[Y_t - Y_{t-1}] = p + \delta.$$

Let

$$T_X = \inf\{t : X_t \geq M\}$$
$$T_Y = \inf\{t : Y_t \geq M\}.$$

We will call T_X and T_Y the hitting times of the chains.

Let $\hat{a}_X = M/T_X$, and $\hat{a}_Y = M/T_Y$ be estimates for $a = p+\delta$. We consider the variance of each of these estimators. When $p+\delta = 1$, $\delta > 0$, X_t becomes a deterministic walk forward to M, and hence $\mathbb{V}(\hat{a}_X) = 0$, but $\hat{a}_Y$ still has positive variance. However, for $\delta > 0$, as p goes to 0, X_t becomes a deterministic walk and thus $\mathbb{V}(\hat{a}_Y) = 0$ while the variance of $\hat{a}_X$ remains positive.

In order to ensure that we capture variance reduction in employing the Markov chain approach to estimating a it is necessary to determine the parameter values p and δ for which we should employ the first versus the second process.

We wish to determine the parameter values p and δ for which one of the estimators given above has lower variance than the other. As a first step in determining these parameter values we must have a way of calculating the variance of the hitting time used to build our estimator for our two random walk alternatives.

We will see that in the case of the process X_T we have access to a closed-form expression of the variance of T_X in terms of our parameter values. In the case of Y_T, the addition of the deterministic drift to our random walk makes the variance of the hitting time less accessible. In the following subsections we present multiple methods that we devised to calculate this variance.

6.2.1. *Simulation.* A straightforward method of estimating the variance of the hitting times of the two processes is to simply simulate the chains. We use the distribution of the hitting time across simulations to test the correct design and implementation of the more efficient methods for calculating the variance described below.

Note that for the fully random walk described above as X_t, the possibility of remaining in the current state means that the hitting time T_X is an unbounded random variable. Thus, in generating distributions by simulation it is necessary to specify an upper bound time at which to stop running in such a way that we know how much of the hitting time distribution we are losing by stopping the process. For this we employ the Chernoff Bound (see Appendix A).

6.2.2. *Transition Matrix.* Both of the processes considered, as Markov chains, can be described by a transition matrix. For a chain with number of states n the transition matrix P is an $n \times n$ matrix with entries $P_{i,j}$ the probability of transitioning from state i to state j in one step. Powers of the transition matrix have the property that $P^t_{i,j}$ gives the probability that the chain ends in state j having began in state i and taken t steps.

For the fully random walk described above as X_t, the the set of possible states is given by $\{0, 1, \ldots, M\}$. P is then an $(M+1) \times (M+1)$ matrix. In order to compute the distribution of $T_X = \inf\{t : X_t \geq M\}$ we consider the transition matrix entry $P_{1,(M+1)}$. The first row of P corresponds to our start state 0, the $(M+1)$st column corresponds to M. Thus we have

$$\mathbb{P}(X_t = M) = P^t_{1,(M+1)}.$$

Recall that having reached M the chain remains in the state M. Thus the probability that the first time at which the chain reaches M is T, is given by

$$\mathbb{P}(T_X = t) = P^t_{1,(M+1)} - P^{t-1}_{1,(M+1)}.$$

We construct the transition matrix and compute its powers, employing an upper bound (see Appendix A) on the values t for which to compute a probability.

For the walk with a deterministic drift, described above as Y_T, the set of possible states is given by $\{0, 1/\delta, 2/\delta, \ldots, (M+1)/\delta\}$. Let P be the $(M+1)/\delta \times (M+1)/\delta$ transition matrix for this chain. Then the probability that $Y_t \geq M$ is given by

$$\mathbb{P}(Y_t \geq M) = \sum_{j=\frac{M}{\delta}+1}^{\frac{M+1}{\delta}+1} P^t_{1,j}.$$

Once again for chain states greater than or equal to M, having reached a state, the chain remains there. Thus we can calculate the distribution of the hitting time $T_Y = \inf\{t : Y_t \geq M\}$ by

$$\mathbb{P}(T_Y = t) = \sum_{j=\frac{M}{\delta}+1}^{\frac{M+1}{\delta}+1} P^t_{1,j} - \sum_{j=\frac{M}{\delta}+1}^{\frac{M+1}{\delta}+1} P^{t-1}_{1,j}.$$

We construct the transition matrix and compute powers P^t for $0 \leq t \leq \frac{M}{\delta}$.

6.2.3. *Direct Calculation.* Though the transition matrix method provides us with an accurate distribution (up to the error introduced by choosing a bound on t in the fully random case), this method, while not as much so as simulation, is computationally expensive, particularly for the chain with drift when M/δ is large. We devise more direct methods for calculating the variance without the need to calculate the probabilities of each hitting time.

In the case of the fully random chain, X_t, this task is simple, as the distribution of T_X is known.

Lemma 5.

$$\mathbb{V}[T_X] = M(1-(p+\delta))/(p+\delta)^2.$$

Proof. We need only note that the value of the state X_t is given by the sum of t Bernoulli random variables with success probability $(p+\delta)$. The hitting time T_X is then the number of Bernoulli's drawn in the process before reaching the Mth success.

Thus we see that T_X has a negative binomial distribution with parameters M and $(p+\delta)$. The variance of this negative binomially distributed random variable is then simply

$$\mathbb{V}[T_X] = M\frac{1-(p+\delta)}{(p+\delta)^2}.$$

□

For the process with the deterministic drift, Y_t, we calculate the variance by establishing a recurrence relation using the conditional variance formula. For a given M, let $T_M = \inf\{t : Y_t \geq M | Y_0 = 0\}$. Considering the first step in our chain, we see that a recursive formula for T_M is

$$T_M = 1 + p\mathbb{E}[T_{M-1-\delta}] + (1-p)\mathbb{E}[T_{M-\delta}],$$

for $M \geq 0$ and $T_M = 0$ for $M < 0$.

The conditional variance formula can be used to find a recursive formula for $\mathbb{V}[T_M]$.

Lemma 6.

$$\begin{aligned}\mathbb{V}[T_M] = \quad & p\mathbb{E}[T_{M-(1+\delta)}]^2 + (1-p)\mathbb{E}[T_{M-\delta}]^2 \\ & - (p\mathbb{E}[T_{M-(1+\delta)}] + (1-p)\mathbb{E}[T_{M-\delta}])^2 \\ & + \ p\mathbb{V}[T_{M-(1+\delta)}] + (1-p)\mathbb{V}[T_{M-\delta}].\end{aligned}$$

Proof. The conditional variance formula tells us that

$$\mathbb{V}[T_M] = \mathbb{V}[\mathbb{E}[T_M|Y_1]] + \mathbb{E}[\mathbb{V}[T_M|Y_1]].$$

Here Y_1, the variable that we condition on, is the state reached after the first step in our chain. $Y_1 = 1 + \delta$ with probability p and $Y_1 = \delta$ with probability $(1-p)$.

We first consider the $\mathbb{E}[\mathbb{V}[T_M|Y_1]]$ term. If $Y_1 = 1 + \delta$, then $\mathbb{V}[T_M|Y_1] = \mathbb{V}[T_{M-(1+\delta)}]$. Likewise if $Y_1 = \delta$, then $\mathbb{V}[T_M|Y_1] = \mathbb{V}[T_{M-\delta}]$. Thus the expectation of the conditional variance is given recursively by

$$\begin{aligned}\mathbb{E}[\mathbb{V}[T_M|Y_1]] &= \mathbb{E}[\mathbb{1}(Y_1 = 1+\delta)\mathbb{V}[T_{M-(1+\delta)}] + \mathbb{1}(Y_1 = \delta)\mathbb{V}[T_{M-\delta}]] \\ &= p\mathbb{V}[T_{M-(1+\delta)}] + (1-p)\mathbb{V}[T_{M-\delta}].\end{aligned}$$

Now consider $\mathbb{E}[T_M|Y_1]$. If $Y_1 = 1 + \delta$, then $\mathbb{E}[T_M|Y_1] = \mathbb{E}[T_{M-(1+\delta)}]$. Likewise if $Y_1 = \delta$, then $\mathbb{E}[T_M|Y_1] = \mathbb{E}[T_{M-\delta}]$. Thus the conditional expectation is given recursively by

$$\mathbb{E}[T_M|Y_1] = \mathbb{1}(Y_1 = 1+\delta)\mathbb{E}[T_{M-(1+\delta)}] + \mathbb{1}(Y_1 = \delta)\mathbb{E}[T_{M-\delta}].$$

We determine the first and second moments of $\mathbb{E}[T_M|Y_1]$ expressed in this way as follows.

$$\begin{aligned}\mathbb{E}[\mathbb{E}[T_M|Y_1]] &= \mathbb{E}[\mathbb{1}(Y_1 = 1+\delta)\mathbb{E}[T_{M-(1+\delta)}] + \mathbb{1}(Y_1 = \delta)\mathbb{E}[T_{M-\delta}]] \\ &= p\mathbb{E}[T_{M-(1+\delta)}] + (1-p)\mathbb{E}[T_{M-\delta}].\end{aligned}$$

To compute the second moment we make use of two observations about indicator functions. The square of any indicator function is equivalent to the indicator function, and the product $\mathbb{1}(Y_1 = a)\mathbb{1}(Y_1 = b) = 0$ for any $a \neq b$. Thus we have

$$\begin{aligned}\mathbb{E}[\mathbb{E}[T_M|Y_1]^2] &= \mathbb{E}[\mathbb{1}(Y_1 = 1+\delta)\mathbb{E}[T_{M-(1+\delta)}] + \mathbb{1}(Y_1 = \delta)\mathbb{E}[T_{M-\delta}]^2] \\ &= \mathbb{E}[\mathbb{1}(Y_1 = 1+\delta)\mathbb{E}[T_{M-(1+\delta)}]^2 + \mathbb{1}(Y_1 = \delta)\mathbb{E}[T_{M-\delta}]^2] \\ &= p\mathbb{E}[T_{M-(1+\delta)}]^2 + (1-p)\mathbb{E}[T_{M-\delta}]^2.\end{aligned}$$

Combining the above yields the result. □

Letting $\mathbb{E}[T_M] = \mathbb{V}[T_M] = 0$ for $M < 0$, and knowing $\mathbb{E}[T_M] = 0$ for $M = 0$, $\mathbb{E}[T_M] = 1$ for $M = \delta$, and $\mathbb{V}[T_M] = 0$ for both $M = 0$ and $M = \delta$, we use dynamic programming and this recurrence relation to find $\mathbb{V}[T_M]$ for our desired M, thereby calculating $\mathbb{V}[T_Y]$).

6.2.4. *Examining Hitting Time Variance Across Parameter Values.* In order to get an initial visual intuition for the parameter values for which one chain has lower variance than the other we plot the ratio $\mathbb{V}(T_Y)/\mathbb{V}(T_X)$. Figure 3 shows this ratio as it varies across δ values for given p values. Figure 4 captures the behavior of this ratio at different parameter configurations such that $p = \delta$.

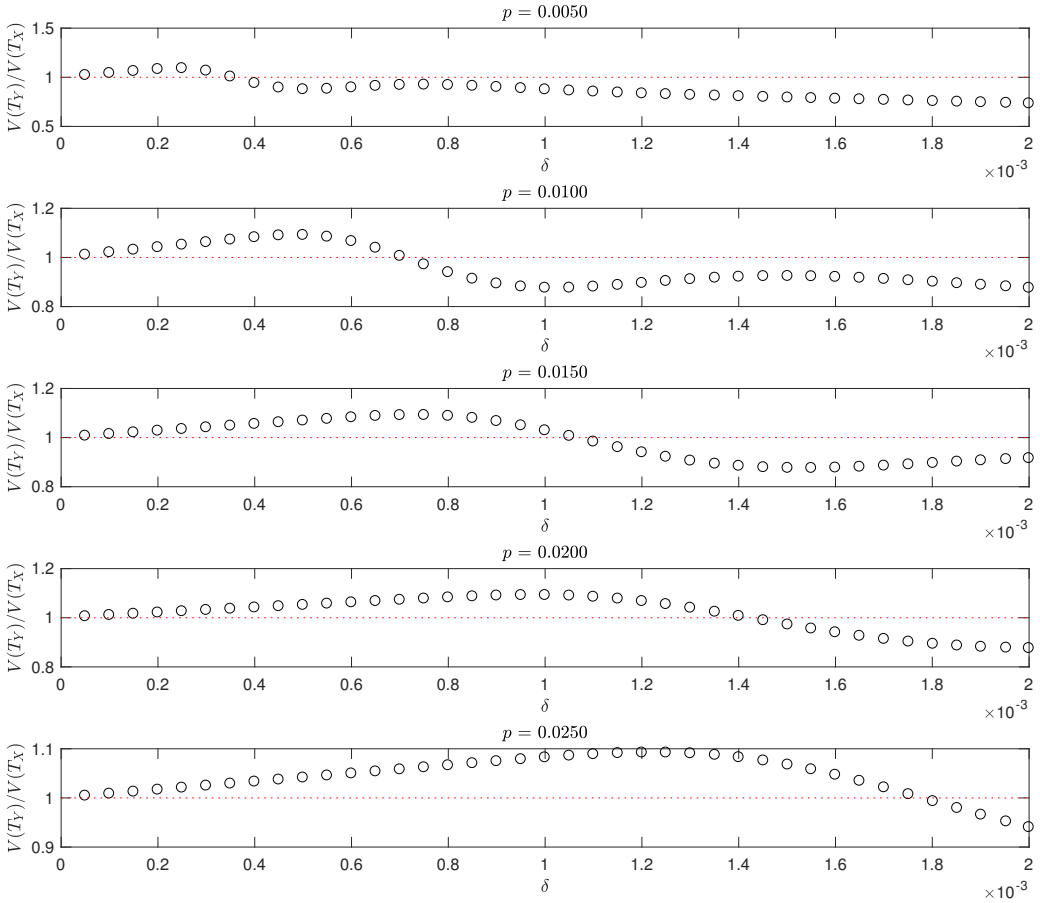

FIGURE 3. Ratio of Variances of Hitting Times for Chains Y_t and X_t. Values are plotted over a range of p and δ with $M = 10$. Calculated using methods provided in 6.2.3.

Without a means of establishing a closed form expression for $\mathbb{V}[T_Y]$, and with these numerical results not yielding any readily discernible transition points, we were unable to determine

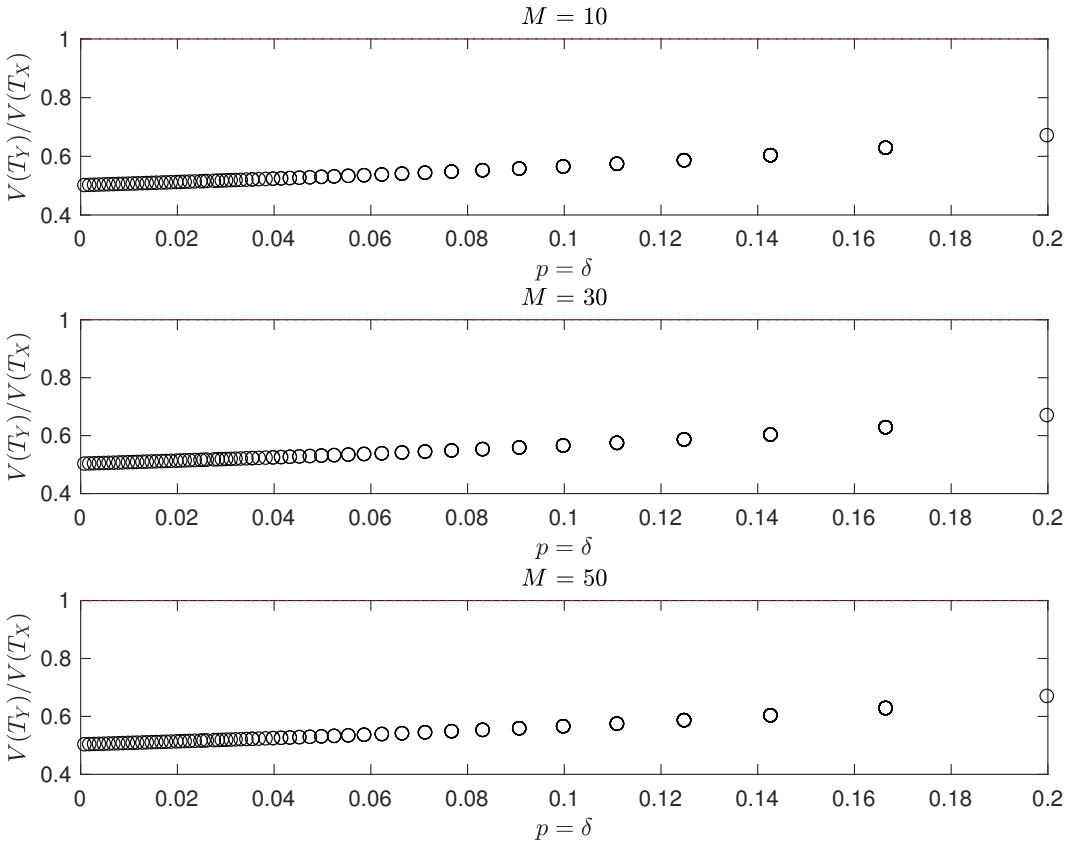

FIGURE 4. Ratio of Variances of Hitting Times for Chains Y_t and X_t.
Plotted for parameter configuration $p = \delta$
over range of M values.
Calculated using methods provided in 6.2.3.

parameter ranges for which we can ensure a given process reduces the variance of the hitting time.

Due to the trend in Figure 4, we conjecture that as $p = \delta$ approaches 0, the value of $\mathbb{V}(T_Y)/\mathbb{V}(T_X)$ approaches 1/2, which would suggest that for small and nearly equal σ_Z^2 and $w\epsilon\hat{\mu}_1$ using Y_t rather than X_t would halve the number of samples needed to produce a $(\gamma, \delta/3)$ approximation in step 2. As a matter of future work, we hope that a proof of this conjecture will provide insight into the parameter ranges, allowing us to incorporate these estimators into the new approximation scheme.

7. Bounds on Expected Number of Samples Needed

In order to compare the number of samples used by DKLR to the number of samples used by the new algorithm, we establish a lower bound on the expected number of samples needed for $DKLR$.

Theorem 4. *The DKLR algorithm uses a number of Z_i draws T, where*

$$\begin{aligned}\mathbb{E}[T] \geq \; & 16(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\left(\frac{1}{\epsilon\mu_Z}\right) \\ & + 8(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\max\left\{\frac{1}{\epsilon\mu_Z}, \frac{\sigma_Z^2}{(\epsilon\mu_Z)^2}\right\}.\end{aligned}$$

Proof. We will make use of the following lemma.

Lemma 7. *(Part 2 of Stopping Rule Theorem in [3]).*
For an estimate $\hat{\mu}_Z$ of μ_Z obtained using the Stopping Rule Algorithm,

$$\frac{1}{\mu_Z} \leq \frac{1}{\hat{\mu}_Z}.$$

In Step 1 of DKLR, an initial estimate $\hat{\mu}_Z$ is obtained for μ_Z using the Stopping Rule Algorithm. For the purposes of determining a lower bound on the expected number of samples, we disregard the samples needed for Step 1, by far the fastest step of the approximation scheme.

In Step 2 a number of samples, T_2, given in terms of the initial estimate for μ_Z obtained in Step 1 is used to compute an estimate $\hat{\sigma}_Z^2$ of σ_Z^2 using an unbiased estimator for σ_Z^2. Using Part 2 of the stopping rule theorem, we obtain

$$\begin{aligned}\mathbb{E}[T_2] &= \mathbb{E}\left[16(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\frac{1}{\epsilon\hat{\mu}_Z}\right] \\ &\geq 16(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\frac{1}{\epsilon\mu_Z}.\end{aligned}$$

Step 3 employs the basic sample mean estimator for μ_Z using a number of samples T_3 given in terms of the initial estimates $\hat{\mu}_Z$ and $\hat{\sigma}_Z^2$.

$$\begin{aligned}\mathbb{E}[T_3] &= \mathbb{E}\left[8(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\max\left\{\frac{1}{\epsilon\hat{\mu}_Z}, \frac{\hat{\sigma}_Z^2}{(\epsilon\hat{\mu}_Z)^2}\right\}\right] \\ &= 8(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\max\left\{\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_Z}\right], \mathbb{E}\left[\frac{\hat{\sigma}_Z^2}{(\epsilon\hat{\mu}_Z)^2}\right]\right\}.\end{aligned}$$

Once again by Part 2 of the Stopping Rule Theorem we have

$$\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_Z}\right] \geq \frac{1}{\epsilon\mu_Z}.$$

Consider $\mathbb{E}[\hat{\sigma}^2/(\epsilon\hat{\mu}_Z)^2]$. The initial estimates $\hat{\sigma}^2$ and $\hat{\mu}_Z$ are computed using independent draws from Z. Thus

$$\begin{aligned}
\mathbb{E}\left[\frac{\hat{\sigma}_Z^2}{(\epsilon\hat{\mu}_Z)^2}\right] &= \mathbb{E}[\hat{\sigma}_Z^2]\mathbb{E}\left[\frac{1}{(\epsilon\hat{\mu}_Z)^2}\right] \\
&= \sigma^2\mathbb{E}\left[\frac{1}{(\epsilon\hat{\mu}_Z)^2}\right] \\
&\geq \sigma^2\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_Z}\right]^2 && \text{(by Jensen's inequality)} \\
&\geq \sigma^2\frac{1}{(\epsilon\mu_Z)^2}. && \text{(by Stopping Rule Theorem)}
\end{aligned}$$

We can therefore bound the expectation of T_3 by

$$\mathbb{E}[T_3] \geq 8(e-2)(1+\sqrt{\epsilon})(1+2\sqrt{\epsilon})\ln(3/\delta)\max\left\{\frac{1}{\epsilon\mu_Z}, \frac{\sigma^2}{(\epsilon\mu_Z)^2}\right\}.$$

□

We now establish upper bounds on the samples needed for Algorithm 2. In order to do so we will make use of the following bound on the parameter c that we employ for the biased GBAS algorithm.

Lemma 8. *If* $\epsilon \leq 1/\sqrt{2}$, *and*

$$c = \frac{2\epsilon}{(1-\epsilon^2)\ln(1+2\epsilon/(1-\epsilon))},$$

then

$$c < 1 + \frac{4}{3}\epsilon^2.$$

Proof. We take the Taylor series expansion of c to find that

$$\begin{aligned}
c &= 1 + \frac{2}{3}\epsilon^2 + \lambda_1\epsilon^4 + \lambda_2\epsilon^6 + \cdots && \text{(where } \lambda_i < 2/3 \text{ for all } i\text{)} \\
&\leq 1 + \frac{2}{3}\epsilon^2(1 + \epsilon^2 + \epsilon^4 + \epsilon^6\cdots) \\
&\leq 1 + \frac{2}{3}\epsilon^2(2). && \text{(by } \epsilon \leq 1/\sqrt{2}\text{)}
\end{aligned}$$

□

Note that the errors $\epsilon^{1/3}$ and $1-\gamma$, used for step 1 and step 2 respectively, satisfy the condition of this lemma by input restrictions on ϵ and γ.

Theorem 5. *Let*

$$\tilde{c}_1 = 1 + (4/3)\epsilon^{2/3}$$
$$\tilde{c}_2 = 1 + (4/3)(1-\gamma)^2.$$

The new approximation scheme introduced as Algorithm 2 uses a number of Z_i draws T, where

$$\mathbb{E}[T] \leq \left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)\epsilon^{-2/3}, \left(\epsilon^{-1/3}+\frac{2}{3}\right)^2 \right\} \right\rceil \left(\frac{1}{\mu_Z}\right) \tag{1}$$
$$+ \frac{1}{w}\left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)(1-\gamma)^{-2}, \left((1-\gamma)^{-1}+\frac{2}{3}\right)^2 \right\} \right\rceil \frac{\tilde{c}_1 k_1}{(k_1-1)}\left(\frac{1}{\epsilon\mu_Z}\right) \tag{2}$$
$$+ \frac{2w(\tilde{c}_2 k_2/(k_2-1)-\gamma)\tilde{c}_1 k_1 (1+\epsilon^{1/3})^2}{\gamma(k_1-1)} \ln(6/\delta)\left(\frac{1}{\epsilon\mu_Z}\right) \tag{3}$$
$$+ \frac{2\tilde{c}_1^2\tilde{c}_2 k_1(k_1+1)k_2(1+\epsilon^{1/3})^2}{\gamma(k_1-1)^2(k_2-1)} \ln(6/\delta)\left(\frac{\sigma_Z^2}{(\epsilon\mu_Z)^2}\right) \tag{4}$$
$$+ \frac{1}{\mu_Z} + \frac{\tilde{c}_1 k_1}{w(k_1-1)}\cdot\frac{1}{\epsilon\mu_Z} + 2\ln(6/\delta). \tag{5}$$

Proof. We consider the expected number of samples used in each step of the algorithm.

The expected number of samples needed to obtain an estimate using GBAS is given by k times the inverse of the mean being estimated.

Thus, letting T_1 be the number of samples drawn in step 1, we have

$$\mathbb{E}[T_1] = \left(\left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)\epsilon^{-2/3}, \left(\epsilon^{-1/3}+\frac{2}{3}\right)^2 \right\} \right\rceil + 1 \right)\left(\frac{1}{\mu_Z}\right).$$

Likewise, letting T_2 be the number of samples drawn in step 2 we have

$$\mathbb{E}[T_2] = \left(\left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)(1-\gamma)^{-2}, \left((1-\gamma)^{-1}+\frac{2}{3}\right)^2 \right\} \right\rceil + 1 \right)\mathbb{E}\left[\frac{1}{\sigma_Z^2 + w\epsilon\hat{\mu}_1}\right]$$
$$\leq \left(\left\lceil \max\left\{ 2\ln\left(\frac{6}{\sqrt{2\pi}\delta}\right)(1-\gamma)^{-2}, \left((1-\gamma)^{-1}+\frac{2}{3}\right)^2 \right\} \right\rceil + 1 \right)\frac{1}{w}\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_1}\right]$$

The estimator $\hat{\mu}_1$ generated by the biased GBAS algorithm is given by $(k_1 - 1)/(c_1 R)$ where $R \sim \mathsf{Gamma}(k_1, \mu_Z)$. Thus

$$\begin{aligned}\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_1}\right] &= \frac{c_1 k_1}{(k_1-1)} \cdot \frac{1}{\epsilon\mu_Z} \\ &\leq \frac{\tilde{c}_1 k_1}{(k_1-1)} \cdot \frac{1}{\epsilon\mu_Z}. \qquad \text{(by Lemma 8)}\end{aligned}$$

Let T_3 be the number of samples drawn in step 3. By Lemma 2

$$\begin{aligned}T_3 &= 2\left(\frac{\hat{a}/\gamma - w\epsilon\hat{\mu}_1}{\left(\hat{\mu}_1/(1+\epsilon^{1/3})\right)^2}\epsilon^{-2} + 1\right)\ln(6/\delta) \\ &= 2\left(\frac{(1+\epsilon^{1/3})^2}{\gamma}\cdot\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2} - w(1+\epsilon^{1/3})^2\frac{1}{\epsilon\hat{\mu}_1} + 1\right)\ln(6/\delta).\end{aligned}$$

Taking the expectation,

$$\mathbb{E}[T_3] = 2\left(\frac{(1+\epsilon^{1/3})^2}{\gamma}\mathbb{E}\left[\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2}\right] - w(1+\epsilon^{1/3})^2\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_1}\right] + 1\right)\ln(6/\delta).$$

Having obtained an upper bound on $\mathbb{E}[1/\epsilon\hat{\mu}_1]$ above, we now bound $\mathbb{E}[\hat{\alpha}/(\epsilon\hat{\mu}_1)^2]$ as follows. First note that

$$\mathbb{E}\left[\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2}\right] = \mathbb{E}\left[\mathbb{E}\left[\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2}\middle|\hat{\mu}_1\right]\right].$$

The estimate $\hat{a}$ generated by the biased GBAS algorithm is given by $(k_2-1)/(c_2 R)$ where $R \sim \mathsf{Gamma}(k_2, \sigma_Z^2 - w\epsilon\hat{\mu}_1)$. Thus

$$\mathbb{E}\left[\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2}\middle|\hat{\mu}_1\right] = \frac{c_2 k_2(\sigma_Z^2 + w\epsilon\hat{\mu}_1)}{(k_2-1)(\epsilon\hat{\mu}_1)^2},$$

and we have

$$\begin{aligned}\mathbb{E}\left[\mathbb{E}\left[\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2}\middle|\hat{\mu}_1\right]\right] &= \mathbb{E}\left[\frac{c_2 k_2(\sigma_Z^2 + w\epsilon\hat{\mu}_1)}{(k_2-1)(\epsilon\hat{\mu}_1)^2}\right] \\ &= \frac{c_2 k_2 \sigma_Z^2}{k_2-1}\mathbb{E}\left[\frac{1}{(\epsilon\hat{\mu}_1)^2}\right] + \frac{c_2 k_2 w}{k_2-1}\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_1}\right].\end{aligned}$$

Once again we note that because we know the distribution of $1/\hat{\mu}_1$, we know

$$\mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_1}\right] = \frac{c_1 k_1}{k_1 - 1}\cdot\frac{1}{\epsilon\mu_Z}.$$

The expectation of the square is then given by

$$\begin{aligned}\mathbb{E}\left[\frac{1}{(\epsilon\hat{\mu}_1)^2}\right] &= \mathbb{V}\left[\frac{1}{\epsilon\hat{\mu}_1}\right] + \mathbb{E}\left[\frac{1}{\epsilon\hat{\mu}_1}\right]^2 \\ &= \frac{c_1^2 k_1}{(k_1-1)^2}\cdot\frac{1}{(\epsilon\mu_Z)^2} + \left(\frac{c_1 k_1}{k_1-1}\cdot\frac{1}{\epsilon\mu_Z}\right)^2 \\ &= k_1(k_1+1)\cdot\frac{c_1^2}{(k_1-1)^2}\cdot\frac{1}{(\epsilon\mu_Z)^2}\end{aligned}$$

Then, combining the above yields

$$\begin{aligned}\mathbb{E}\left[\frac{\hat{\alpha}}{(\epsilon\hat{\mu}_1)^2}\right] &= \frac{c_1^2 c_2 k_1(k_1+1)k_2}{(k_1-1)^2(k_2-1)}\cdot\frac{\sigma_Z^2}{(\epsilon\mu_Z)^2} + \frac{c_1 c_2 w k_1 k_2}{(k_1-1)(k_2-1)}\cdot\frac{1}{\epsilon\mu_Z} \\ &\leq \frac{\tilde{c}_1^2\tilde{c}_2 k_1(k_1+1)k_2}{(k_1-1)^2(k_2-1)}\cdot\frac{\sigma_Z^2}{(\epsilon\mu_Z)^2} + \frac{\tilde{c}_1\tilde{c}_2 w k_1 k_2}{(k_1-1)(k_2-1)}\cdot\frac{1}{\epsilon\mu_Z}. \qquad \text{(by Lemma 8)}\end{aligned}$$

□

We can compare this upper bound to the lower bound given for the number of samples for DKLR. We first examine, line 1 of the bound given in Theorem 5. Line 1 corresponds to the number of samples needed for step 1 of the algorithm. If

$$\delta < \frac{6}{\sqrt{2\pi}} e^{-\epsilon^{2/3}(\epsilon^{-1/3}+2/3)^2/2},$$

then this number of samples is a multiple of $1/(\epsilon^{2/3}\mu_Z)$, which is of lower order than $1/(\epsilon\mu_Z)$.

Lines 2 and 3 are readily comparable to the $(1/\epsilon\mu_Z)$ term in the lower bound for DKLR. If we let $\gamma = (1/2)$ and $w = 2$, and reasonably assume that

$$\delta < \frac{6}{\sqrt{2\pi}} e^{-(1-\gamma)^2((1-\gamma)^{-1}+2/3)^2/2} \approx .98,$$

then lines 2 and 3 yield a coefficient for $(1/\epsilon\mu_Z)$ of about $8\ln(6/\delta)$, compared to DKLR's $16(e-2)\ln(3/\delta)$.

We can compare line 4 to the $\max\{1/\epsilon\mu_Z, \sigma_Z^2/(\epsilon\mu_Z)^2\}$ term in the number of samples needed for DKLR. Using $\gamma = (1/2)$ yields a coefficient of about $4\ln(6/\delta)$, compared to DKLR's $8(e-2)\log(3/\delta)$.

8. Empirical Comparison

We provide a comparison of the number of samples required for DKLR and the number of samples required by the new algorithm. We use the two algorithms to approximate the mean of test data in the form of beta-distributed random variables with representative

characteristics. We are particularly concerned with the behavior of the algorithms with respect to the relative size of σ^2 and $\epsilon\mu_Z$.

We then compare the performance of the algorithms for a distribution arising from a network science application, an approximation algorithm used to estimate the reliability of highly unstable networks. As presented, this algorithm relies on DKLR to generate its estimate.

8.1. **Test Data.** We compare the number of samples needed for DKLR to the number of samples needed for the new algorithm. We are interested in three cases.

1) $\sigma_Z^2 \approx \epsilon\mu_Z$
   Table 1 provides a comparison of the number of samples required by DKLR and Algorithm 2, using parameters $\gamma = (1/2)$ and $w = 2$, to obtain a an ϵ, δ approximation of the mean μ for Beta random variables with $\sigma_Z^2 = \epsilon\mu_Z$.

2) $\sigma_Z^2 \gg \epsilon\mu_Z$
   Table 2 provides a comparison of the number of samples required by DKLR and Algorithm 2, using parameters $\gamma = (1/2)$ and $w = 2$, to obtain a an ϵ, δ approximation of the mean μ for Beta random variables with $\sigma_Z^2 \approx \mu_Z(1 - \mu_Z)$.

3) $\sigma_Z^2 \ll \epsilon\mu_Z$
   Table 3 provides a comparison of the number of samples required by DKLR and Algorithm 2, using parameters $\gamma = (1/2)$ and $w = 2$, to obtain a an ϵ, δ approximation of the mean μ for Beta random variables with $\sigma_Z^2 = \epsilon\mu_Z \cdot 10^{-4}$.

(ϵ, δ)	μ_Z	σ_Z^2	DKLR	New	**New/DKLR**
$(0.1000, 0.0100)$	0.01	0.001	$4.5159 \cdot 10^5$	$9.3649 \cdot 10^4$	$\mathbf{2.0738 \cdot 10^{-1}}$
$(0.1000, 0.0100)$	0.05	0.005	$8.9702 \cdot 10^4$	$2.6444 \cdot 10^4$	$\mathbf{2.9479 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	0.01	0.001	$6.3353 \cdot 10^5$	$1.273 \cdot 10^5$	$\mathbf{2.0094 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	0.05	0.005	$1.2739 \cdot 10^5$	$3.1703 \cdot 10^4$	$\mathbf{2.4887 \cdot 10^{-1}}$

TABLE 1. Samples Needed Comparison: Beta Random Variables $\sigma_Z^2 \approx \epsilon\mu_Z$. Values for the new algorithm obtained using Algorithm 2 with $\gamma = 1/2$, $w = 2$. Number of samples averaged over 5 runs.

For these beta random variables we observe that the new algorithm required fewer than half the number of samples required by DKLR in the high-variance case, and as little as fewer than a fifth of the samples required by DKLR in the low-variance case, for $\epsilon = .1, \delta \in \{.01, .001\}$.

(ϵ, δ)	μ_Z	σ_Z^2	DKLR	New	**New/DKLR**
$(0.1000, 0.0100)$	0.01	$9.899 \cdot 10^{-3}$	$1.6685 \cdot 10^{6}$	$5.712 \cdot 10^{5}$	$\mathbf{3.4235 \cdot 10^{-1}}$
$(0.1000, 0.0100)$	0.05	$4.7495 \cdot 10^{-2}$	$3.3022 \cdot 10^{5}$	$9.2411 \cdot 10^{4}$	$\mathbf{2.7985 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	0.01	$9.899 \cdot 10^{-3}$	$2.423 \cdot 10^{6}$	$8.4063 \cdot 10^{5}$	$\mathbf{3.4694 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	0.05	$4.7495 \cdot 10^{-2}$	$4.4943 \cdot 10^{5}$	$1.9384 \cdot 10^{5}$	$\mathbf{4.313 \cdot 10^{-1}}$

TABLE 2. Samples Needed Comparison: Beta Random Variables $\sigma_Z^2 \gg \epsilon\mu_Z$. Values for the new algorithm obtained using Algorithm 2 with $\gamma = 1/2$, $w = 2$. Number of samples averaged over 5 runs.

(ϵ, δ)	μ_Z	σ_Z^2	DKLR	New	**New/DKLR**
$(0.1000, 0.0100)$	0.01	$1 \cdot 10^{-7}$	$3.7881 \cdot 10^{5}$	$6.8047 \cdot 10^{4}$	$\mathbf{1.7963 \cdot 10^{-1}}$
$(0.1000, 0.0100)$	0.05	$5 \cdot 10^{-7}$	$7.5765 \cdot 10^{4}$	$1.4342 \cdot 10^{4}$	$\mathbf{1.8929 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	0.01	$1 \cdot 10^{-7}$	$5.3687 \cdot 10^{5}$	$8.9777 \cdot 10^{4}$	$\mathbf{1.6722 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	0.05	$5 \cdot 10^{-7}$	$1.0738 \cdot 10^{5}$	$1.7513 \cdot 10^{4}$	$\mathbf{1.631 \cdot 10^{-1}}$

TABLE 3. Samples Needed Comparison: Beta Random Variables $\sigma_Z^2 \ll \epsilon\mu_Z$. Values for the new algorithm obtained using Algorithm 2 with $\gamma = 1/2$, $w = 2$. Number of samples averaged over 5 runs.

8.2. **Application: Network Reliability.** We test the two approximation schemes' performance for the critical estimation step of the algorithm, introduced by Zenklusen and Laumanns in [7], for approximating the *st*-reliability of a network. The problem of *st*-reliability is as follows.

We have a directed, acyclic, network. Each arc in the network, considered independently, will be operational with some small probability p. We will refer to an instantiation of the network with a certain set of operating arcs as an operating state of the network. For given nodes s and t in the network, we define the *st*-reliability of the network to be the probability that there exists a path from s to t in the network such that every arc on the path is operational.

We can intuitively see that this notion would be useful for thinking about networks such as telecommunications networks, which one would design to be highly reliable. One might then want to know the likelihood that, in the rare event of arc failures, two nodes s and t remain connected.

This algorithm is primarily concerned however, with the case in which the network is highly unreliable, that is the probability p assigned to each edge is very small and hence the likelihood of an operating path is also small. As is noted in [7], this case is applicable

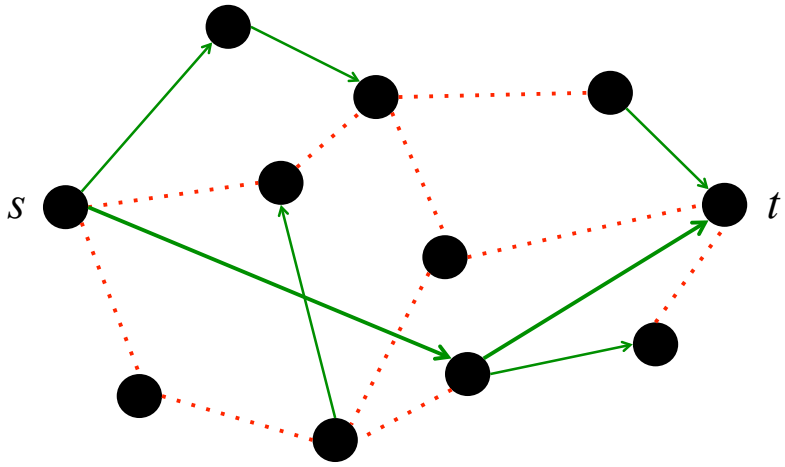

FIGURE 5. The st-reliability of a network is the probability of there being an operating path from node s to node t given that arcs will only be operational with a certain probability. In this operating state of the network pictured above, we see one operating st-path.

to questions of network dynamics such as disease spreading, where one is interested in the unlikely event that a disease can spread from one node to another node in the network given a low likelihood of disease transmission occurring along any particular arc.

The computational complexity of determining exact st-reliabilities requires that they be estimated, and naturally, in this small probability case, the algorithm is concerned with obtaining an (ϵ, δ) relative error approximation for the reliability.

Zenklusen and Laumanns devise a method for drawing $[0, 1]$ random variables with mean proportional to the st-reliability and then uses DKLR to obtain an (ϵ, δ) approximation of this mean. We compare the number of samples needed to perform this estimation using DKLR to the number needed using Algorithm 2, when the algorithm is applied to random networks.

The random networks to which we apply the algorithm are Delaunay networks. Delaunay networks are formed by first taking a Delaunay triangulation of n points dropped uniformly at random in a unit square to determine the nodes. The nodes s and t are then chosen to be the two points with the largest pairwise Euclidean distance. To determine the arcs, all of the edges of the triangulation are then oriented so as to have nonnegative scalar product with the vector from s to t. This random construction is convenient for this problem as it ensures that we have constructed a network consisting of only arcs that lie on paths from s to t and hence arcs that are relevant to the reliability approximation.

The results of the algorithm comparison are given in Table 4. We see improvement using the new approximation scheme. For the network parameters used and for $\epsilon = .1, \delta \in$

(ϵ, δ)	nodes	p arcs op	DKLR	New	**New/DKLR**
$(0.1000, 0.0100)$	100	0.01	$3.7928 \cdot 10^3$	$7.264 \cdot 10^2$	$\mathbf{1.9152 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	100	0.01	$5.3736 \cdot 10^3$	$1.0338 \cdot 10^3$	$\mathbf{1.9238 \cdot 10^{-1}}$
$(0.0100, 0.0100)$	100	0.01	$1.6682 \cdot 10^5$	$4.6878 \cdot 10^3$	$\mathbf{2.8102 \cdot 10^{-2}}$
$(0.0100, 0.0010)$	100	0.01	$2.3892 \cdot 10^5$	$7.0216 \cdot 10^3$	$\mathbf{2.939 \cdot 10^{-2}}$
$(0.1000, 0.0100)$	100	0.05	$3.8408 \cdot 10^3$	$7.522 \cdot 10^2$	$\mathbf{1.9584 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	100	0.05	$5.4512 \cdot 10^3$	$1.056 \cdot 10^3$	$\mathbf{1.9372 \cdot 10^{-1}}$
$(0.0100, 0.0100)$	100	0.05	$1.7089 \cdot 10^5$	$5.6818 \cdot 10^3$	$\mathbf{3.3249 \cdot 10^{-2}}$
$(0.0100, 0.0010)$	100	0.05	$2.448 \cdot 10^5$	$8.4346 \cdot 10^3$	$\mathbf{3.4456 \cdot 10^{-2}}$
$(0.1000, 0.0100)$	100	0.1	$4.0558 \cdot 10^3$	$9.43 \cdot 10^2$	$\mathbf{2.3251 \cdot 10^{-1}}$
$(0.1000, 0.0010)$	100	0.1	$5.745 \cdot 10^3$	$1.1194 \cdot 10^3$	$\mathbf{1.9485 \cdot 10^{-1}}$
$(0.0100, 0.0100)$	100	0.1	$2.0455 \cdot 10^5$	$1.2898 \cdot 10^4$	$\mathbf{6.3054 \cdot 10^{-2}}$
$(0.0100, 0.0010)$	100	0.1	$2.9209 \cdot 10^5$	$1.7337 \cdot 10^4$	$\mathbf{5.9353 \cdot 10^{-2}}$

TABLE 4. Number of Samples Needed Comparison
st-Reliability Estimation on 100 Node Delaunay Networks.
Values for the new algorithm obtained using Algorithm 2 with $\gamma = 1/2$,
$w = 2$. Number of samples averaged over 5 runs.

$\{.01, .001\}$, Algorithm 2 required approximately a fifth of the number of samples required by DKLR. For the smaller ϵ of .01 the ratio of the number of samples required by the new approximation scheme to the number required by DKLR improved further, to order 10^{-2}.

9. Conclusions

Adapting a new estimator for the mean of $[0, 1]$ random variables, we have introduced an (ϵ, δ) relative error approximation scheme to be employed in Monte Carlo approximation algorithms for which it is necessary to estimate the mean of a simulated distribution to user-specified error and this mean is known to be small. Having established an upper bound on the expected number of samples required by this new algorithm, we see improvement over a lower bound on the expected number of samples required by an existing approximation scheme. Corroborating this theoretical result, the computational comparison performed showed that the new algorithm required significantly fewer samples both when used on test data and on data arising from an algorithm for approximating network reliabilities.

References

[1] O. Catoni. Challenging the empirical mean and empirical variance: A deviation study. *Ann. Inst. Henri Poincare Probab. Stat.*, 48:1148–1185, 2012.

[2] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. of Math. Stat.*, 23:493–509, 1952.
[3] P. Dagum, R. Karp, M. Luby, and S. Ross. An optimal algorithm for Monte Carlo estimation. *Siam. J. Comput.*, 29(5):1484–1496, 2000.
[4] J. Feng, M. Huber, and Y. Ruan. Monte carlo with user-specified relative error. 2017. Submitted.
[5] M. Huber. A Bernoulli mean estimate with known relative error distribution. *Random Structures Algorithms*, 2016. arXiv:1309.5413. To appear.
[6] A. Wald. *Sequential Analysis*. Dover Publications, Inc., 1947.
[7] R. Zenklusen and M. Laumanns. High-confidence estimation of small s-t reliabilities in directed acyclic networks. *Networks*, 57(4):376–388, 2011.

Appendix A. Chernoff Bound on Tails of the Binomial Distribution

Because of the possibility of remaining in the same state in the fully random chain, there is no upper bound on the chain's hitting time. For this reason, in order to simulate the chain and make calculations based on powers of the chain's transition matrix, we are interested in determining an upper bound which will capture the density of the hitting time distribution to within a specified error.

In the random chain, the chain state at time t, X_t, is binomially distributed with parameters $(p+\delta)$ and t. We wish to establish an upper bound time at which the probability that the state is still lower than M is less than our desired error. We employ the Chernoff Bound introduced in [2]. For the binomial distribution the Chernoff Bound gives

$$\mathbb{P}(X_t \leq (1-\epsilon)t(p+\delta)) \leq \left(\frac{(1-\epsilon)^{1-\epsilon}}{e^{-\epsilon}}\right)^{-t(p+\delta)} \leq e^{-\epsilon^2 t(p+\delta)/2}.$$

Letting our desired error be denoted γ, we wish to turn this into a bound of the form

$$\mathbb{P}(X_t \leq M) \leq \gamma.$$

We let

$$t = \frac{M}{(1-\epsilon)(p+\delta)},$$

and determine the value of ϵ that yields the desired bound.

$$\begin{aligned} \gamma = e^{-\epsilon^2(\frac{M}{(1-\epsilon)}/2)} &\implies \epsilon^2 - \frac{2\ln(\gamma)}{M}\epsilon + \frac{2\ln(\gamma)}{M} = 0 \\ &\implies \epsilon = \frac{\ln(\gamma)}{M} + \left[\left(\frac{\ln(\gamma)}{M}\right)^2 - \frac{2\ln(\gamma)}{M}\right]^{\frac{1}{2}}. \end{aligned}$$

Appendix B. Code for DKLR and New Approximation Scheme

```
# Script for comparing DKLR runtime to runtime of new approximation scheme

# DKLR ------------------------------------------------------------------------
DKLR <- function(one_sample_fun, epsilon, delta) {
  # Provides an epsilon, delta approximation for the mean of the variable from which input one_sample\\
      \\_fun generates
  # independent draws using the AA algorithm of Dagum, Karp, Luby and Ross 2000
  # Outputs:
  # A list with keys
  # estimate      = an estimate of the mean
  # num_samples   = the number of samples drawn
  # Inputs:
  # one_sample_fun       = function taking no parameters that generates one sample
  # epsilon              = desired error factor
```

```
  # delta                 = desired probability that estimate is not within factor 1+epsilon of \\
      \\reliability

  Ups <- 4 * (exp(1) - 2) * log(2/delta) * epsilon^(-2)

  ######################################## STEP 1 ########################################
  Ups_one <- 1 + (1 + epsilon) * Ups
  i <- 0
  S <- 0
  samples_one <- vector()
  repeat {
    i <- i + 1
    sample <- one_sample_fun()
    S <- S + sample
    samples_one <- append(samples_one, sample)
    if (S >= Ups_one) {
      break
    }
  }
  mu_hat <- Ups_one/i

  ######################################## STEP 2 ########################################
  Ups_two <- Ups * 2 * (1 + sqrt(epsilon)) * (1 + 2*sqrt(epsilon)) * (1 + log(3/2) / log(2/delta))
  N_two <- ceiling(Ups_two * epsilon / mu_hat)
  S <- 0
  samples_two <- replicate(2*N_two, one_sample_fun())
  S <- sum(mapply(function(sample_a, sample_b) (sample_a - sample_b)^2, samples_two[1:N_two], samples\\
      \\_two[(N_two+1):(2*N_two)]))
  rho <- max(S / N_two, epsilon * mu_hat)

  ######################################## STEP 3 ########################################
  N_three = ceiling(Ups_two * rho / mu_hat^2)

  samples_three <- replicate(N_three, one_sample_fun())

  S <- sum(samples_three)

  estimate <- S/N_three

  num_samples <- length(samples_one) + length(samples_two) + length(samples_three)

  DKLR_output <- list("estimate" = estimate, "num_samples" = num_samples)
  return(DKLR_output)
}

# GBAS ----------------------------------------------------------------------------------------

determine_k_for_GBAS <- function(epsilon, delta) {
  # Determines value of k which ensures desired error bound using GBAS algorithm
  k<-1
```

```
  while (pgamma(1+epsilon/2,k,k-1) - pgamma((1+epsilon/2)^(-1),k,k-1) < 1-delta)
    k <- k+1
  return(k)
}

GBAS <- function(one_sample_fun, epsilon = NULL, delta = NULL, k = NULL, biased = FALSE, c = NULL) {
  # Provides an epsilon, delta approximation for the mean of the variable from which input one_sample\\
      \\_fun generates
  # independent draws using a the Gamma Bernoulli Approximation Scheme of Huber
  # Outputs:
  # A list with keys
  # estimate       = an estimate of the mean
  # num_samples    = the number of samples drawn
  # Inputs:
  # one_sample_fun        = function taking no parameters that generates one sample from a Bernoulli
  #                         distributed random variable
  # EITHER
  #   epsilon             = desired error
  #   delta               = desired probability that estimate is not within factor 1+epsilon of true \\
      \\mean
  # OR
  #   k                   = parameter of GBAS determined by error parameters, can be
  #                         specified in place of epsilon and delta
  # biased                = indicator to use biased k-1/cR estimator
  # c                     = bias parameter for biased estimator
  if (is.null(k)) {
    k <- determine_k_for_GBAS(epsilon, delta)
  }
  S <- 0
  R <- 0
  num_samples = 0
  repeat {
    S <- S + one_sample_fun()
    num_samples = num_samples + 1
    R <- R + rexp(1)
    if (S == k) {
      break
    }
  }

  if (biased) {
    estimate = (k-1)/(c*R)
  } else {
    estimate = (k-1)/R
  }

  GBAS_output <- list("estimate" = estimate, "num_samples" = num_samples)

  return(GBAS_output)
}
```

```
# Catoni \\
    \\-----------------------------------------------------------------------------
root_by_bisection_zero_one = function(fun, interval_bound_one, interval_bound_two) {
  # returns a root of function fun
  # Inputs:
  # fun                = a function taking a single parameter defined on on [0,1]
  # interval_bound_one = bound on interval, for which sign of function value at this bound is
  #                      opposite of that at interval_bound_two
  # interval_bound_two = bound on interval, for which sign of function value at this bound is
  #                      opposite of that at interval_bound_one

  fun_val_one = fun(interval_bound_one)
  fun_val_two = fun(interval_bound_two)

  repeat {
    root = (interval_bound_one + interval_bound_two)/2
    fun_val_root = fun(root)
    if (abs(fun_val_root) < 10^(-9)){
      break
    } else if (sign(fun_val_root) == sign(fun_val_one)) {
      interval_bound_one = root
      fun_val_one = fun_val_root
    } else {
      interval_bound_two = root
      fun_val_two = fun_val_root
    }
  }
  return(root)
}


Catoni <- function(one_sample_fun, epsilon, delta, mean_lower_bound, var_upper_bound) {

  num_samples <- ceiling(2 * (var_upper_bound / (epsilon * mean_lower_bound)^2 + 1) * log(2/delta))
  alpha <- epsilon * mean_lower_bound / (var_upper_bound + (epsilon * mean_lower_bound)^2)

  # samples <- replicate(num_samples, one_sample_fun())
  #
  # psi <- function(x) ifelse(x >= 0, log(1 + x + x^2/2), -log(1 - x + x^2/2))
  # influence_function <- function(estimate) sum(sapply(samples, function(sample) psi(alpha * (sample\\
    \\ - estimate))))
  #
  # estimate = root_by_bisection_zero_one(influence_function, min(samples), max(samples))

  estimate <- 0

  Catoni_output = list("estimate" = estimate, "num_samples" = num_samples)

  return(Catoni_output)
```

```
}


# New Approximation Scheme \\
    \\-----------------------------------------------------------------------------
new_RAS <- function(one_sample_fun, epsilon, delta, c, gamma, step_two_alg = c('GBAS', 'Catoni')) {
  # Provides an epsilon, delta approximation for the mean of the variable from which input one_sample\\
      \\_fun generates
  # independent draws using a the Gamma Bernoulli Approximation Scheme of Huber and the M estimator \\
      \\of Catoni 2012
  # Outputs:
  # A list with keys
  # estimate       = an estimate of the mean
  # num_samples    = the number of samples drawn
  # Inputs:
  # one_sample_fun        = function taking no parameters that generates one sample
  # epsilon               = desired error factor
  # delta                 = desired probability that estimate is not within factor 1+epsilon of \\
      \\reliability
  # c                     = parameter weighting number of samples in terms of variance versus number \\
      \\of samples
  #                         in terms of inverse of mean
  # gamma                 = desired error factor for lower bound on variance in step 2
  # step_two_alg          = specifies algorithm to use for variance bound step
  #                         Options: 'GBAS' and 'Catoni'

  ################################### STEP 1 #####################################

  epsilon_one = epsilon^(1/3)

  B_one = function() as.integer(runif(1) < one_sample_fun())
  c_one = 2 * epsilon^(1/3) / ((1-epsilon^(2/3)) * log(1 + 2 * epsilon^(1/3)/(1-epsilon^(1/3))))
  k_one = ceiling(max(2 * log(6/(sqrt(2*pi)*delta)) * epsilon^(-2/3), (epsilon^(-1/3) + 2/3)^2)) + 1

  GBAS_one_output = GBAS(B_one, k = k_one, biased = TRUE, c = c_one)
  mu_one = GBAS_one_output$estimate
  num_samples_one = GBAS_one_output$num_samples

  ################################### STEP 2 #####################################

  switch(step_two_alg,

         GBAS  =  {B_two = function() as.integer(runif(1) < ((one_sample_fun() - one_sample_fun())^2)\\
             \\/2 + c*epsilon*mu_one)
                   c_two = 2*(1-gamma) / ((2*gamma-gamma^2) * log(1 + 2*(1-gamma) / gamma))
                   k_two = ceiling(max(2 * log(6/(sqrt(2*pi)*delta)) * (1-gamma)^(-2), ((1-gamma)\\
                       \\^(-1) + 2/3)^2)) + 1

                   GBAS_two_output = GBAS(B_two, k = k_two, biased = TRUE, c = c_two)
                   a_hat = GBAS_two_output$estimate
```

```
                num_samples_two = GBAS_two_output$num_samples})

  ################################## STEP 3 ######################################

  mean_lower_bound = mu_one / (1 + epsilon_one)
  var_upper_bound = (a_hat / (1 - gamma)) - c * epsilon * mu_one

  Catoni_output = Catoni(one_sample_fun, epsilon, delta, mean_lower_bound, var_upper_bound)
  Catoni_estimate = Catoni_output$estimate
  num_samples_three = Catoni_output$num_samples

  estimate = Catoni_estimate
  num_samples <- num_samples_one + num_samples_two + num_samples_three

  new_RAS_output <- list("estimate" = estimate, "num_samples" = num_samples)
  return(new_RAS_output)
}

# Testing Estimators ------------------------------------------------------------------
test_estimator <- function(estimator = c("DKLR", "new_RAS_GBAS", "new_RAS_Catoni")) {
  # Generates data on how often estimator is coming out within desired error bounds
  # Outputs:
  # error_data  = data_frame giving error rate by parameter values
  # Inputs:
  # estimator   = specifies estimator to test
  #               DKLR              = Approximation Algorithm of Dagum Karp Luby Ross 2000
  #               new_RAS_GBAS      = New approximation algorithm using GBAS for step 2
  #               new_RAS_Catoni    = New approximation algorithm using Catoni for step 2

  num_estimates <- 100
  mu_list       <- c(.1)
  epsilon_list  <- c(.1)
  delta_list    <- c(.1)

  switch(estimator,
         DKLR               = {estimate_fun <- function(one_sample_fun, epsilon, delta)  DKLR(one_\\
             \\sample_fun, epsilon, delta)},
         new_RAS_GBAS       = {estimate_fun <- function(one_sample_fun, epsilon, delta)  new_RAS(one_\\
             \\sample_fun, epsilon, delta, 2, 1/2, 'GBAS') },
         new_RAS_Catoni     = {estimate_fun <- function(one_sample_fun, epsilon, delta)  new_RAS(one_\\
             \\sample_fun, epsilon, delta, 2, 1/2, 'Catoni')})

  error_data <- data.frame(matrix(nrow = prod(sapply(list(epsilon_list, delta_list, mu_list), length)\\
      \\), ncol = 3))
  names(error_data) <- list("(eps,_del)", "mu", "error_rate")

  run <- 1
  for (epsilon in epsilon_list) {
    for (delta in delta_list) {
      for (mu in mu_list) {
```

```
        one_sample_fun <- function() ifelse(runif(1) < mu, 1, 0)

        error_data[run, "(eps, del)"] <- sprintf("(%.4f, %.4f)", epsilon, delta)
        error_data[run, "mu"] <- mu

        is_in_bound <- function(estimate) abs(estimate/mu - 1) < epsilon
        error_data[run, "error_rate"] <- mean(replicate(num_estimates, !is_in_bound(estimate_fun(one_\\
            \\sample_fun, epsilon, delta)$estimate)))

        run <- run + 1
      }
    }
  }
  return(error_data)
}

# Algorithm Comparison ----------------------------------------------------------------
generate_num_samples_table <- function(mu_list, epsilon_list, delta_list, regime = c('eq', 'var_\\
    \\greater', 'eps_mu_greater'), num_runs) {
  # Generates table with number of samples needed to approximate the mean of beta random variables \\
      \\having
  # propeerty specified by regime using the different algorithm alternatives
  # Inputs:
  # mu_list       = list of mean values to consider
  # epsilon_list  = list of epsilon values to consider
  # delta_list    = list of delta values to consider
  # regime        = specifies property of random beta variable
  #   eq             = variance \approx epsilon mu
  #   var_greater    = variance >> epsilon mu
  #   eps_mu_greater = variance << epsilon mu
  # num_runs      = number of runs over which to average the number of samples used
  # Outputs: a data frame

  data <- data.frame(matrix(nrow = prod(sapply(list(epsilon_list, delta_list, mu_list), length)), \\
      \\ncol = 6))
  names(data) <- list("(eps, del)", "mu", "var", "DKLR", "New", "New/DKLR")

  row = 1
  for (epsilon in epsilon_list) {
    for (delta in delta_list) {
      for (mu in mu_list) {

        switch(regime,
               eq             = {alpha <- mu * (1-mu) / (epsilon) - mu
                                 beta <- alpha*(1/mu - 1)},

               var_greater    = {alpha <- .0001 * mu
                                 beta <- alpha*(1/mu - 1)},

               eps_mu_greater = {alpha <- mu * (1-mu) / (.0001 * epsilon) - mu
```

```
                                    beta <- alpha*(1/mu - 1)})

        one_sample_fun <- function() rbeta(1, alpha, beta)

        data[row, "(eps, del)"] <- sprintf("(%.4f, %.4f)", epsilon, delta)
        data[row, "mu"] <- mu

        variance <- alpha * beta / ((alpha + beta)^2 * (alpha + beta + 1))
        data[row, "var"] <- variance

        DKLR_num_samples <- mean(replicate(num_runs, DKLR(one_sample_fun, epsilon, delta)$num_samples\\
            \\))
        data[row, "DKLR"] <- DKLR_num_samples

        new_Ras_num_samples <- mean(replicate(num_runs, new_RAS(one_sample_fun, epsilon, delta, 2, 1/\\
            \\2, 'GBAS')$num_samples))
        data[row, "New"] <- new_Ras_num_samples

        data[row, "New/DKLR"] <- new_Ras_num_samples / DKLR_num_samples

        row <- row + 1
      }
    }
  }
  return(data)
}



## Generates three data frames storing tables giving comparison of number of samples needed for each \\
    \\algorithm
## for different parameter values. Writes these data frames to excel file "comparison_by_regime.xlsx"
## data_eq             = number of samples required under regime variance \approx epsilon mu
## data_var_greater    = number of samples required under regime variance >> epsilon mu
## data_eps_mu_greater = number of samples required under regime variance << epsilon mu

require('xlsx')

mu_list        <- c(.01, .05)
epsilon_list   <- c(.01)
delta_list     <- c(.01, .001)
num_runs       <- 5

data_eq             <- generate_num_samples_table(mu_list, epsilon_list, delta_list, 'eq', num_runs)
data_var_greater    <- generate_num_samples_table(mu_list, epsilon_list, delta_list, 'var_greater', \\
    \\num_runs)
data_eps_mu_greater <- generate_num_samples_table(mu_list, epsilon_list, delta_list, 'eps_mu_greater'\\
    \\, num_runs)

write.xlsx(data_eq, file = "comparison_by_regime.xlsx", sheetName = "approx_equal")
```

```
write.xlsx(data_var_greater, file = "comparison_by_regime.xlsx", sheetName = "variance_greater", \\
    \\append = TRUE)
write.xlsx(data_eps_mu_greater, file = "comparison_by_regime.xlsx", sheetName = "epsmu_greater", \\
    \\append = TRUE)

print("finished_algorithm_comparison,_wrote_result_to_comparison_by_regime.xlsx")
```

Appendix C. Code for *st*-Reliability Application

```
# This script implements the s-t reliability approximation algorithm
# for highly unreliable directed acyclic networks introduced in (Zenklusen and Laumanns 2010)

require(network)
require(geometry)

script_dir <- dirname(sys.frame(1)$ofile)
if(!exists("DKLR", mode="function")) source(sprintf("%s/thesis_comparison_script.R", script_dir))

# ADDITIONAL FUNCTIONS FOR WORKING WITH NETWORK OBJECTS ----------------------------------------

get_nodes_from_arc <- function(network, arc_edgeID) {
  # Returns nodes that compose arc with edgeID input edgeID in input network
  # Outputs:
  # A list with keys
  # out_of          = the node from which the arc comes
  # into            = the node to which the arc is incident
  # Inputs:
  # network       = a network object
  # arc_edgeID    = the edgeID of the desired arc
  # Note:
  # This is just a general function to get from edgeIDs to their nodes which was
  # not implemented in networks package. It is used in functions add_prob_operational_tc
  # and sample_path.
  out_node <- network$mel[arc_edgeID][[1]]$outl
  in_node <- network$mel[arc_edgeID][[1]]$inl
  output_list <- list("out_of" = out_node, "into" = in_node)
}

# RANDOM NETWORK GENERATORS ---------------------------------------------------

max_pair_over_antipodal <- function(points) {
  # Uses rotating calipers algorithm of (Preparata and Shamos 1985) to find all antipodal pairs
  # in set of d-dimensional points input points and returns the pair with the largest distance
  # Returns the antipodal pair with the greatest distance between them
  # Outputs:
  # max_distance_pair      = a list, the pair of vectors giving the max distance points
  # Inputs:
  # points                 = a number of points by d matrix, with rows representing points
  #                          in counterclockwise order, none collinear (e.g. output of
  #                          convhulln(points, options = "Fx"))
```

```
num_points <- nrow(points)

move_one <- function(x) ifelse(x != num_points, x+1, 1)
area <- function(x,y,z) abs(.5 * det(cbind(rbind(x,y,z),c(1,1,1))))
distance <- function(x,y) sqrt((y-x)[1]^2 + (y-x)[2]^2)

p_o <- 1
p_n <- num_points

p <- p_n
q <- move_one(p)

max_pair <- vector("list", length = 2)
max_distance <- 0

repeat {
  q <- move_one(q)
  if (area(points[p,], points[move_one(p),], points[move_one(q),]) <= area(points[p,], points[move_\\
      \\one(p),], points[q,])) {
    break
  }
}
q_o <- q
repeat {
  p <- move_one(p)
  dist_pair <- distance(points[p,], points[q,])
  if (dist_pair > max_distance) {
    max_distance <- dist_pair
    max_pair <- list(points[p,], points[q,])
  }
  repeat {
    q <- move_one(q)
    if (!(p == q_o && q == p_o)) {
      dist_pair <- distance(points[p,], points[q,])
      if (dist_pair > max_distance) {
        max_distance <- dist_pair
        max_pair <- list(points[p,], points[q,])
      }
    }
    if (area(points[p,], points[move_one(p),], points[move_one(q),]) <= area(points[p,], points[\\
        \\move_one(p),], points[q,])) {
      break
    }
  }

  if (area(points[p,], points[move_one(p),], points[move_one(q),]) == area(points[p,], points[move_\\
      \\one(p),], points[q,])) {
    if (!(p == q_o && q == p_n)) {
      dist_pair <- distance(points[p,], points[move_one(q),])
```

```
        if (dist_pair > max_distance) {
          max_distance <- dist_pair
          max_pair <- list(points[p,], points[move_one(q),])
        }
      }
    }
    if (q == p_o) {
      break
    }
  }

  return(max_pair)
}

generate_delaunay <- function(num_nodes) {
  # Generates a random directed acyclic delaunay graph as described in section 8.1.1
  # Inputs:
  # num_nodes                = the number of nodes desired
  # Outputs:
  # a list with keys
  # network       = a network object having the property that all edges
  #                   in the network lie on a path from s to t
  # s             = the source of the network
  # t             = the terminal of the network
  uniforms <- runif(2*num_nodes)
  points <- matrix(uniforms, ncol= 2)
  conv_hull_boundary <- chull(points[,1],points[,2])
  conv_hull_boundary <- points[conv_hull_boundary,]
  max_distance_pair <- max_pair_over_antipodal(conv_hull_boundary)
  s <- max_distance_pair[[1]]
  t <- max_distance_pair[[2]]

  delaunay_triangulation <- delaunayn(points)
  del_sides <- vector("list")
  for (triangle in 1:nrow(delaunay_triangulation)) {
    del_sides <- append(del_sides, lapply(list(c(1,2),c(1,3),c(2,3)), function(side) sort(c(delaunay_\\
        \\triangulation[triangle,][side[1]], delaunay_triangulation[triangle,][side[2]]))))
  }
  del_sides <- unique(del_sides)

  edges <- matrix(ncol = 2, nrow = length(del_sides))
  for (i in 1:length(del_sides)) {
    if (dot(points[del_sides[[i]][2],] - points[del_sides[[i]][1],], (t-s)) > 0) {
      edges[i,] = del_sides[[i]]
    } else {
      edges[i,] = rev(del_sides[[i]])
    }
  }

  network <- network(edges)
```

```
  s <- which(apply(points, 1, function(row) identical(row, s)))
  t <- which(apply(points, 1, function(row) identical(row, t)))

  output_list <- list("network" = network, "s" = s, "t" = t)
  return(output_list)
}

add_prob_operational_del <- function(network, uniform_prob_operational) {
  # Adds edge attribute prob_operational to input network, assigning value
  # input uniform_prob_operational to each edge
  # Outputs:
  # network                    = the network input network with attribute prob_operational
  # Inputs:
  # network                    = a network object, output of generate_delaunay
  # uniform_prob_operational   = probability of being operational to be assigned to each edge
  set.edge.attribute(network, "prob_operational", uniform_prob_operational, e = seq_along(network$mel\\
      \\))
  return(network)
}

generate_topological_construction <- function(num_nodes, arc_density) {
  # Generates a random directed acyclic graph using the topological construction described in section\\
      \\ 8.1.2
  # Inputs:
  # num_nodes                   = the number of nodes desired
  # arc_density                 = probability with which to add edges beyond initial path from s to t
  # Outputs:
  # network       = a network object having the property that all edges
  #                   in the network lie on a path from s to t
  # Note:
  # The source s will have vertex index 1 in the network
  # The terminal t will have vertex index given by num_nodes in the network
  adj_mat <- matrix(0, nrow = num_nodes, ncol = num_nodes)
  network <- network(adj_mat)

  for (i in 1:(num_nodes-1)) {
    add.edge(network, i, i+1)
  }
  for (i in 1:(num_nodes-2)) {
    for (j in (i+2):num_nodes) {
      if (runif(1) < arc_density) {
        add.edge(network, i, j)
      }
    }
  }
  return(network)
}

add_prob_operational_tc <- function(network, prob_operational_param) {
  # Adds edge attribute prob_operational to input network, assigning value
```

```
  # determined by input prob_operational_param to each edge
  # Outputs:
  # network                    = the network input network with attribute prob_operational
  # Inputs:
  # network                    = a network object, output of generate_topological_construction
  # prob_operational_param     = typically in [0,1], smaller values result in less reliable networks
  get_prob_operational <- function(i,j)  (j-i)^(prob_operational_parameter - 1) * runif(1)
  for (arc in unlist(network$iel)) {
    nodes <- get_nodes_from_arc(network, arc)
    prob_operational <- get_prob_operational(nodes$out_of, nodes$into)
    set.edge.attribute(network, "prob_operational", prob_operational, arc)
  }
  return(network)
}

# ALGORITHMS IN SECTION 5 ------------------------------------------------------

get_topological_ordering <- function(network, s, t) {
  # Determines a topological ordering of nodes not s and t in the graph input network using Kahn's \\
    \\algorithm
  # Outputs:
  # ordering  = a length n-2 vector of node ids for the network, a topological ordering of nodes \\
    \\excluding s and t
  # Inputs:
  # network   = a network object representing a directed acyclic graph
  # s         = vertex ID of source for which we are estimating reliability
  # t         = vertex ID of terminal for which we are estimating reliability
  num_nodes <- network.size(network)
  node_vertexID_list <- 1:num_nodes

  ordering <- vector(length = num_nodes)
  order_idx <- 1
  current_sources <- Filter(function(node) length(get.edgeIDs(network, node, neighborhood = "in")) ==\\
    \\ 0, node_vertexID_list)

  while (length(current_sources) != 0) {
    chosen_source_node <- current_sources[1]
    current_sources <- current_sources[-1]

    ordering[order_idx] <- chosen_source_node
    order_idx <- order_idx + 1

    chosen_node_neighborhood <- get.neighborhood(network, chosen_source_node, "out")

    for (edge in get.edgeIDs(network, chosen_source_node, neighborhood = "out")) {
      delete.edges(network,edge)
    }

    for (neighbor_of_chosen_node in chosen_node_neighborhood) {
      if (length(get.neighborhood(network, neighbor_of_chosen_node, "in")) == 0) {
```

```
        current_sources <- append(current_sources, neighbor_of_chosen_node)
      }
    }
  }

  s_index <- seq_along(ordering)[ordering == s]
  ordering <- ordering[-s_index]
  t_index <- seq_along(ordering)[ordering == t]
  ordering <- ordering[-t_index]

  return(ordering)
}

assign_arc_weights <- function(network, s, t, topological_ordering) {
  # Implements Algorithm 2 to assign weights to each arc in directed graph input network
  # which allow for sampling of paths with proabibility proportional to the the probability
  # that all arcs on the path are operational
  # Outputs:
  # network         = a copy of input network with added edge attribute weight
  # Inputs:
  # network              = a network object representing a directed acyclic graph with edge attribute
  #                        prob_operational assigned to each edge, preprocessed so that every
  #                        arc lies on a path from node input s to node input t
  # s                    = vertex ID of source for which we are estimating reliability
  # t                    = vertex ID of terminal for which we are estimating reliability
  # topological_ordering = a vector of vertex IDs giving a topological ordering of vertices
  #                        not s and t, output of get_topological_ordering
  num_nodes <- network.size(network)

  list_of_arcs_into_t <- mapply(function(edge, edgeID) list("edge" = list(edge), "edgeID" = edgeID),
                                get.edges(network, t, neighborhood = "in"), get.edgeIDs(network, t, \\
                                    \\neighborhood = "in"))

  for (i in 1:length(list_of_arcs_into_t["edge",])) {
    arc_into_t <- list_of_arcs_into_t[,i]
    arc_into_t_prob_operational <- get.edge.attribute(arc_into_t$edge, "prob_operational")
    arc_into_t_weight <- arc_into_t_prob_operational
    set.edge.attribute(network, "weight", arc_into_t_weight, arc_into_t$edgeID)
  }

  for (node in rev(topological_ordering)) {
    weight_leaving_node <- sum(unlist(lapply(get.edges(network, node, neighborhood = "out"), function\\
        \\(edge) get.edge.attribute(list(edge),"weight"))))

    list_of_arcs_into_node <- mapply(function(edge, edgeID) list("edge" = list(edge), "edgeID" = \\
        \\edgeID),
                                     get.edges(network, node, neighborhood = "in"), get.edgeIDs(\\
                                         \\network, node, neighborhood = "in"))

    for (i in 1:length(list_of_arcs_into_node["edge",])) {
```

```
      arc_into_node <- list_of_arcs_into_node[,i]
      arc_into_node_prob_operational <- get.edge.attribute(arc_into_node$edge, "prob_operational")
      arc_into_node_weight <- arc_into_node_prob_operational * weight_leaving_node
      set.edge.attribute(network, "weight", arc_into_node_weight, arc_into_node$edgeID)
    }
  }

  return(network)
}

calc_sample_space_weight <- function(network,s) {
  # Calculates the weight of the sample space (line 16 of algorithm 1)
  # Outputs:
  # sample_space_weight    = weight of the sample space, equivalent to mean number of paths from s to\\
    \\ t
  # Inputs:
  # network        = network object representing network for which we are estimating reliability,
  #                  with edge attribute "weight" assigned by function assign_arc_weights
  # s              = vertex ID of source for which we are estimating reliability
  sample_space_weight <- sum(get.edge.attribute(get.edges(network,s),"weight"))
  return(sample_space_weight)
}

determine_num_paths <- function(network, s, t, topological_ordering) {
  # Determines the number of paths from s to t on the network
  # Used to determine number of operating paths on network realizations (line 12 of algorithm 1)
  # Output:
  # num_paths_from_s_to_t   = the number of paths from node input s to node input t on network input \\
    \\network
  # Input:
  # network              = a network object
  # s                    = vertex ID of source for which we are estimating reliability
  # t                    = vertex ID of terminal for which we are estimating reliability
  # topological_ordering = a vector of vertex IDs giving a topological ordering of vertices
  #                        not s and t, output of get_topological_ordering

  for (arc_into_t in get.edgeIDs(network, t, neighborhood = "in")) {
    set.edge.attribute(network, "num_paths_to_t", 1, arc_into_t)
  }

  for (node in rev(topological_ordering)) {
    num_paths_to_t_leaving_node <- sum(unlist(lapply(get.edges(network, node, neighborhood = "out"), \\
      \\function(edge) get.edge.attribute(list(edge), "num_paths_to_t"))))

    for (arc_into_node in get.edgeIDs(network, node, neighborhood = "in")) {
      num_paths_to_t <- num_paths_to_t_leaving_node
      set.edge.attribute(network, "num_paths_to_t", num_paths_to_t, arc_into_node)
    }
  }
```

```
  num_paths_from_s_to_t <-  sum(get.edge.attribute(get.edges(network, s, neighborhood = "out"), "num_\\
      \\paths_to_t"))
  return(num_paths_from_s_to_t)
}

sample_path <- function(network, s, t) {
  # Samples a random path with propability a path is selected proportional to the probability
  # that all arcs on the path are operating (line 4 of algorithm 1)
  # Outputs:
  # path      = a list of edge IDs for arcs in sampled path
  # Inputs:
  # network   = a network object with edge attributes prob_operational and weight
  # s         = vertex ID of source for which we are estimating reliability
  # t         = vertex ID of terminal for which we are estimating reliability
  path <- vector()
  current_node <- s
  while (current_node != t) {
    arcs_edgeIDs_out_of_node <- get.edgeIDs(network, current_node, neighborhood = "out")

    if (length(arcs_edgeIDs_out_of_node) == 1) {
      chosen_arc_edgeID <- arcs_edgeIDs_out_of_node[1]
    } else {
      arcs_edgeLists_out_of_node <- get.edges(network, current_node, neighborhood = "out")
      arc_weights <- get.edge.attribute(arcs_edgeLists_out_of_node, "weight")
      chosen_arc_edgeID <- sample(arcs_edgeIDs_out_of_node, 1, prob = arc_weights)
    }

    path <- append(path, chosen_arc_edgeID)
    chosen_neighbor <- get_nodes_from_arc(network, chosen_arc_edgeID)$into
    current_node <- chosen_neighbor
  }
  return(path)
}

# GENERATING ESTIMATORS ---------------------------------------------------------

estimate_ratio_low_prob <- function(network, s, t, topological_ordering, num_samples) {
  # Estimates the ratio of the s-t reliability to the expected number of operating paths
  # (algorithm 1 modified to return psi)
  # Outputs:
  # estimate   = an estimate of the s-t reliability
  # Inputs:
  # network             = a network object representing a directed acyclic graph composed of
  #                       paths from s to t (i.e. generated using one of the random network \\
      \\generator functions),
  #                       with edge attributes prob_operational and weight (weight assigned using
  #                       function assign_arc_weights)
  # s                   = vertex ID of source for which we are estimating reliability
  # t                   = vertex ID of terminal for which we are estimating reliability
  # num_samples         = number of samples (samples from which to determine estimate)
```

```
  # topological_ordering = a vector of vertex IDs giving a topological ordering of vertices
  #                        not s and t, output of get_topological_ordering
  z <- 0
  for (sample in 1:num_samples) {
    path <- sample_path(network, s, t)
    network_realization <- network

    failed_edges <- Filter(function(edgeID) !(edgeID %in% path) && (runif(1) > get.edge.attribute(\\
        \\network$mel[edgeID], "prob_operational")), unlist(network_realization$iel))
    delete.edges(network_realization, failed_edges)

    num_paths_operational <- determine_num_paths(network_realization, s, t, topological_ordering)
    z <- z + num_paths_operational^(-1)
  }
  ratio_estimate <- z/num_samples
  return(ratio_estimate)
}

estimate_reliability_basic <- function(network, s, t, topological_ordering, num_samples) {
  # Uses basic acceptance-rejection to provide an estimate of the reliability
  # Outputs:
  # reliability_estimate     = an estimate of the probability of an operating path from input s to \\
      \\input t
  #                            in input network
  # Inputs:
  # network                = a network object representing a directed acyclic graph composed of
  #                          paths from s to t (i.e. generated using one of the random network \\
      \\generator functions),
  #                          with edge attributes prob_operational and weight (weight assigned using
  #                          function assign_arc_weights)
  # s                      = vertex ID of source for which we are estimating reliability
  # t                      = vertex ID of terminal for which we are estimating reliability
  # topological_ordering = a vector of vertex IDs giving a topological ordering of vertices
  #                          not s and t, output of get_topological_ordering
  # num_samples            = number of samples (samples from which to determine estimate)
  num_realizations_with_operating_path <- 0
  for (sample in 1:num_samples) {
    network_realization <- network

    failed_edges <- Filter(function(edgeID) (runif(1) > get.edge.attribute(network$mel[edgeID], "prob\\
        \\_operational")), unlist(network_realization$iel))
    delete.edges(network_realization, failed_edges)

    num_paths_operational <- determine_num_paths(network_realization, s, t, topological_ordering)
    indicator_path_operational <- (num_paths_operational > 0)
    num_realizations_with_operating_path <- num_realizations_with_operating_path + indicator_path_\\
        \\operational
  }
  reliability_estimate <- num_realizations_with_operating_path / num_samples
  return(reliability_estimate)
```

```
}

# ESTIMATING RELIABILITY ------------------------------------------------------

estimate_reliability <- function(network, s, t, topological_ordering, epsilon, delta, estimator = c("\\
    \\low_prob", "basic"), algorithm = c("DKLR", "new_RAS_GBAS", "new_RAS_Catoni")) {
  # Provides an epsilon, delta approximation for the s-t reliability of a network
  # using the estimator specified by input estimator and the algorithm specified by
  # input algorithm
  # Outputs:
  # A list with keys
  # estimate      = an estimate of the s-t reliability
  # num_samples   = the number of network instances sampled
  # Inputs:
  # network               = a network object representing a directed acyclic graph composed of
  #                         paths from s to t (i.e. generated using one of the random network \\
      \\generator functions),
  #                         with edge attributes prob_operational and weight (weight assigned using
  #                         function assign_arc_weights)
  # s                     = vertex ID of source for which we are estimating reliability
  # t                     = vertex ID of terminal for which we are estimating reliability
  # topological_ordering  = a vector of vertex IDs giving a topological ordering of vertices
  #                         not s and t, output of get_topological_ordering
  # epsilon               = desired error factor
  # delta                 = desired probability that estimate is not within factor 1+epsilon of \\
      \\reliability
  # estimator             = specifies estimator
  #                         low_prob = the estimate of the ratio of the s-t reliability of the
  #                                    network to the expected number of operating paths, output
  #                                    of estimate_ratio_low_prob
  #                         basic    = estimate of the reliability given by the mean of
  #                                    the Bernoulli random variable that is the indicator
  #                                    that a randomly sampled network realization contains
  #                                    an operating path, output of estimate_reliability_basic
  # algorithm             = specifies algorithm
  #   DKLR                = Approximation Algorithm of Dagum Karp Luby Ross 2000
  #   new_RAS_GBAS        = New approximation algorithm using GBAS for step 2
  #   new_RAS_Catoni      = New approximation algorithm using Catoni for step 2

  switch(estimator,
         low_prob = {one_sample_fun <- function() estimate_ratio_low_prob(network, s, t, topological_\\
             \\ordering, num_samples = 1)},
         basic    = {one_sample_fun <- function() estimate_reliability_basic(network, s, t, \\
             \\topological_ordering, num_samples = 1)})

  switch(algorithm,
         DKLR               = {estimate_fun <- function(one_sample_fun, epsilon, delta)  DKLR(one_\\
             \\sample_fun, epsilon, delta)},
         new_RAS_GBAS       = {estimate_fun <- function(one_sample_fun, epsilon, delta)  new_RAS(one_\\
             \\sample_fun, epsilon, delta, 1, 1/2, 'GBAS') },
```

```
        new_RAS_Catoni      = {estimate_fun <- function(one_sample_fun, epsilon, delta)  new_RAS(one_\\
            \\sample_fun, epsilon, delta, 1, 1/2, 'Catoni')})

  estimate_output <- estimate_fun(one_sample_fun, epsilon, delta)
  estimate <- estimate_output$estimate
  num_samples <- estimate_output$num_samples

  switch(estimator,
         low_prob = {ratio_estimate <- estimate
                     sample_space_weight <- calc_sample_space_weight(network, s)
                     reliability_estimate <- ratio_estimate * sample_space_weight},

         basic    = {reliability_estimate <- estimate})

  reliability_output <- list("estimate" = reliability_estimate, "num_samples" = num_samples)
  return(reliability_output)
}

# RUNNING THROUGH PARAMETERS ------------------------------------------------

epsilon_list     <- c(.1, .01)
delta_list       <- c(.001, .0001)
num_runs         <- 20

num_nodes_list <- c(100)
del_uniform_prob_operational_list   <- .05 * 1:5

data <- data.frame(matrix(nrow = prod(sapply(list(epsilon_list, delta_list, num_nodes_list, del_\\
    \\uniform_prob_operational_list), length)), ncol = 6))
names(data) <- list("(eps, del)", "size", "P(operational)", "DKLR", "New", "New/DKLR")


row <- 1
for (num_nodes in num_nodes_list) {
  del_generator_output <- generate_delaunay(num_nodes)
  network <- del_generator_output$network
  s <- del_generator_output$s
  t <- del_generator_output$t
  topological_ordering <- get_topological_ordering(network, s, t)

  for (uniform_prob_operational in del_uniform_prob_operational_list) {
    network <- add_prob_operational_del(network, uniform_prob_operational)
    network <- assign_arc_weights(network, s, t, topological_ordering)
    total_weight <- calc_sample_space_weight(network, s)

    for (epsilon in epsilon_list) {
      for (delta in delta_list) {

        data[row, "(eps, del)"] <- sprintf("(%.4f, %.4f)", epsilon, delta)
        data[row, "size"] <- num_nodes
```

```
        data[row, "P(operational)"] <- uniform_prob_operational

        DKLR_reliability_output <- estimate_reliability(network, s, t, topological_ordering, epsilon,\\
            \\ delta, "low_prob", "DKLR")
        DKLR_num_samples <- mean(replicate(num_runs, estimate_reliability(network, s, t, topological_\\
            \\ordering, epsilon, delta, "low_prob", "DKLR")$num_samples))

        data[row, "DKLR"] <- DKLR_num_samples

        new_GBAS_num_samples <- mean(replicate(num_runs, estimate_reliability(network, s, t, \\
            \\topological_ordering, epsilon, delta, "low_prob", "new_RAS_GBAS")$num_samples))

        data[row, "New"] <- new_GBAS_num_samples

        data[row, "New/DKLR"] <- new_GBAS_num_samples / DKLR_num_samples

        # new_Catoni_reliability_output <- estimate_reliability(network, s, t, topological_ordering, \\
            \\epsilon, delta, "low_prob", "new_Catoni")
        # new_Catoni_num_samples <- new_Catoni_reliability_output$num_samples
        # data[row, "new_Catoni"] <- new_Catoni_num_samples

        row <- row + 1
      }
    }
  }
  print(sprintf("Completed Size %i Networks", num_nodes))
}

write.csv(data, "delaunay_reliability_alg_comparision.csv")
print("Finished algorithm comparison for Delaunay networks, wrote results to delaunay_reliability_alg\\
    \\_comparision.csv")
```

Appendix D. Code for Markov Chain Approach to Variance Estimation

```
function upperBound = calcUpperBound(p, alpha, M, desiredError)
% Uses the Chernoff Bound on a Binomial Distribution to calculate the upper
% bound on the hitting times to be considered for the random chain given by
%       At any state i less than M
%            With probablity p+1/alpha we transition to state i+1
%            With probability p+1/alpha we stay at state i
%       At any state i greater than or equal to M
%            With probability 1 we remain in state i
% With probability 1-desiredError the hitting time is less than or equal to
% upperBound
% Outputs:
% upperBound  = an upper bound on the hitting times that need be considered
%               given the error specified by desiredError
% Inputs:
% p             = chain parameter
% alpha         = chain parameter (must be a positive integer)
```

```
% M              = the value for which we are interested in the time it takes
%                  for the chain state to exceed this value
% desiredError   = specify probability that the hitting time is a time
%                  greater than upperBound

chainTerm = 2 * log(desiredError) / M;

epsilon = (chainTerm + (chainTerm^2 - 4 * chainTerm)^(.5)) / 2;

upperBound = ceil(M/((1-epsilon) * (p + (1/alpha))));


function hittingTimeDistribution = getSimHittingTimeDistribution(p, alpha, M, chain, numSamples, \\
    \\varargin)
% Returns a vector giving the sample distribution of the first time
% at which the chain state exceeds M, sampling numSamples
% simulations of the markov chain
% given by input chain
% Ouputs:
% hittingTimeDistribution    = For a given index i, this vector
%                              gives the probability that the chain first
%                              reaches a state greater than M on the ith
%                              step
% Inputs:
% p           = chain parameter (see chains)
% alpha       = chain parameter (see chains) (must be a positive integer)
% M           = the value for which we are interested in the time it takes
%               for the chain state to exceed this value
% chain       = a string specifying which chain to use
%               'r' = random
%                At any state i less than M
%                     With probablity p+1/alpha we transition to state i+1
%                     With probability p+1/alpha we stay at state i
%                At any state i greater than or equal to M
%                     With probability 1 we remain in state i
%               NOTE: when using random chain p + 1/alpha must be <= 1
%               's' = semideterministic
%                At any state i less than M
%                     With probablity p we transition to state i+(1+1/alpha)
%                     With probability 1-p we transition to state i+1/alpha
%                At any state i greater than or equal to M
%                     With probability 1 we remain in state i
% numSamples = number of samples to draw
% Additional Arguments:
% If using random chain must specify name value pair
% 'upperBound'  = upper bound on hitting times to be considered, output
%                 of calcUpperBound

switch chain

    case 'r'
```

```
        parser = inputParser;
        addParameter(parser, 'upperBound', -1);
        parse(parser,varargin{:})
        upperBound = parser.Results.upperBound;

        simHittingTimeCounts = zeros(1,upperBound);
        for sample = 1:numSamples
            time = 0;
            state = 0;
            while(state < M)
                time = time + 1;
                if rand(1) < (p + (1/alpha))
                    state = state + 1;
                end
            end
            if time <= upperBound
                simHittingTimeCounts(time) = simHittingTimeCounts(time) + 1;
            end
        end

    case 's'
        simHittingTimeCounts = zeros(1, (M * alpha));
        for sample = 1:numSamples
            time = 0;
            state = 0;
            while(state < (M-(10^(-9))))
                time = time + 1;
                if rand(1) < p
                    state = state + 1 + (1/alpha);
                else
                    state = state + (1/alpha);
                end
            end
            simHittingTimeCounts(time) = simHittingTimeCounts(time) + 1;
        end

end

hittingTimeDistribution = simHittingTimeCounts / numSamples;


function hittingTimeDistribution = getTransMatHittingTimeDistribution(p, alpha, M, chain, varargin)
% Calculates the distribution of the time needed to progress from state 1
% to the set of states greater than or equal to M in the markov chain
% given by input chain
% Outputs:
% hittingTimeDistribution   = For a given index i, this vector
%                             gives the probability that the chain first
%                             reaches a state greater than M on the ith
%                             step
% Inputs:
```

```
% p          = chain parameter (see chains)
% alpha      = chain parameter (see chains) (must be a positive integer)
% M          = the value for which we are interested in the time it takes
%              for the chain state to exceed this value
% chain      = a string specifying which chain to use
%              'r' = random
%               At any state i less than M
%                    With probablity p+1/alpha we transition to state i+1
%                    With probability p+1/alpha we stay at state i
%               At any state i greater than or equal to M
%                    With probability 1 we remain in state i
%              NOTE: when using random chain p + 1/alpha must be <= 1
%              's' = semideterministic
%               At any state i less than M
%                    With probablity p we transition to state i+(1+1/alpha)
%                    With probability 1-p we transition to state i+1/alpha
%               At any state i greater than or equal to M
%                    With probability 1 we remain in state i
% Additional Arguments:
% If using random chain must specify name value pair
% 'upperBound'  = upper bound on hitting times to be considered, output
%                  of calcUpperBound

switch chain

    case 'r'
        parser = inputParser;
        addParameter(parser, 'upperBound', -1);
        parse(parser,varargin{:})
        upperBound = parser.Results.upperBound;

         transitionMatrix = vertcat(horzcat(diag((1-(p+(1/alpha))) * ones(1,M)), ...
                                            zeros(M,1)) + ...
                                    horzcat(zeros(M,1), ...
                                            diag((p+(1/alpha)) * ones(1,M))), ...
                                    horzcat(zeros(1,M), 1));

        hittingTimeDistribution = zeros(1, upperBound);
        lastStepMatrix = diag(ones(M+1,1));
        lastStepProbability = 0;
        for hittingTime = 1:length(hittingTimeDistribution)
            thisStepMatrix = lastStepMatrix * transitionMatrix;
            thisStepProbability = thisStepMatrix(1,M+1);
            hittingTimeDistribution(hittingTime) = thisStepProbability - lastStepProbability;
            lastStepMatrix = thisStepMatrix;
            lastStepProbability = thisStepProbability;
        end

    case 's'
        transitionMatrix = vertcat(horzcat(zeros(M * alpha, 1), ...
```

```
                                        diag((1-p)*ones(M * alpha, 1)), ...
                                        zeros(M * alpha, alpha)) + ...
                                    horzcat(zeros(M * alpha,alpha + 1), ...
                                        diag(p*ones(M * alpha, 1))), ...
                                    horzcat(zeros(1 + alpha, M * alpha), diag(ones(1 + alpha, 1))));

        hittingTimeDistribution = zeros(1, M * alpha);
        lastStepMatrix = diag(ones((M+1)*alpha+1, 1));
        lastStepProbability = 0;
        for hittingTime = 1:length(hittingTimeDistribution)
            thisStepMatrix = lastStepMatrix * transitionMatrix;
            thisStepProbability = sum(thisStepMatrix(1, M*alpha + (1:alpha+1)));
            hittingTimeDistribution(hittingTime) = thisStepProbability - lastStepProbability;
            lastStepMatrix = thisStepMatrix;
            lastStepProbability = thisStepProbability;
        end

end


function semiDetDPTable = generateSemiDetDPTable(p, alpha, max_M)
% Uses dynamic programming to generate a table of vectors
% that give parameter values associated with the hitting time, i.e.
% the time at which the chain reaches state M less than input max_M
% for the following markov chain:
%   At any state i less than M
%       With probablity p we transition to state i + (1 + 1/alpha)
%       With probability 1-p we transition to state i + 1/alpha
%   At any state i greater than or equal to M
%       With probability 1 we remain in state i
% Outputs:
% symbolicSemiDetParameterTable = a structure with 4 fields
%   M_vals      = a vector of values of M, indices correspond to those of
%                 parameter values in other fields
%   exp         = a vector with values the expected hitting time for M given
%                 by the value at the corresponding index in the M_list
%   varCondExp  = a vector with values the variance of the expectation,
%                 conditioned on the initial transition, of hitting time
%                 for M given by the value at the corresponding index in
%                 the M_list
%   var         = a vector with values the variance of the hitting time for M
%                 given by the value at the corresponding index in the
%                 M_list
% inputs:
% p       = chain parameter
% alpha   = chain parameter, must be a positive integer
% max_M   = M value up to which to generate table

semiDetDPTable.M_vals       = ((1/alpha) * (0:(max_M * alpha)))';
semiDetDPTable.exp          = zeros(size(semiDetDPTable.M_vals));
semiDetDPTable.varCondExp   = zeros(size(semiDetDPTable.M_vals));
```

```
semiDetDPTable.var           = zeros(size(semiDetDPTable.M_vals));

semiDetDPTable.exp(1) = 0;
semiDetDPTable.exp(2) = 1;

semiDetDPTable.varCondExp(1:2) = 0;

semiDetDPTable.var(1:2) = 0;

for MIdx = 3:(alpha+1)
    semiDetDPTable.exp(MIdx) = 1 + (1-p) * semiDetDPTable.exp(MIdx - 1);
end

for MIdx = (alpha+2):length(semiDetDPTable.M_vals)
    semiDetDPTable.exp(MIdx) = 1 + p * semiDetDPTable.exp(MIdx - (alpha+1)) + (1-p) * semiDetDPTable.\\
        \\exp(MIdx - 1);
end

for MIdx = 3:(alpha+1)
    semiDetDPTable.varCondExp(MIdx) = (1-p) * semiDetDPTable.exp(MIdx - 1)^2 - ((1-p) * \\
        \\semiDetDPTable.exp(MIdx - 1))^2;
end

for MIdx = (alpha+2):length(semiDetDPTable.M_vals)
    semiDetDPTable.varCondExp(MIdx) = p * semiDetDPTable.exp(MIdx - (alpha+1))^2 + (1-p) * \\
        \\semiDetDPTable.exp(MIdx - 1)^2 - (p * semiDetDPTable.exp(MIdx - (alpha+1)) + (1-p) * \\
        \\semiDetDPTable.exp(MIdx - 1))^2;
end

for MIdx = 3:(alpha+1)
    semiDetDPTable.var(MIdx) = (1-p) * semiDetDPTable.var(MIdx - 1) + semiDetDPTable.varCondExp(MIdx)\\
        \\;
end

for MIdx = (alpha+2):length(semiDetDPTable.M_vals)
    semiDetDPTable.var(MIdx) =  p * semiDetDPTable.var(MIdx - (alpha+1)) + (1-p) * semiDetDPTable.var\\
        \\(MIdx - 1) + semiDetDPTable.varCondExp(MIdx);
end


function hittingTimeVar = calcHittingTimeVar(p, alpha, M, chain)
% Calculates the variance of the time needed to progress from state 1
% to the set of states greater than or equal to M in the markov chain
% given by input chain
% Outputs:
% hittingTimeVar     = The variance of the hitting time
% Inputs:
% p         = chain parameter (see chains)
% alpha     = chain parameter (see chains) (must be a positive integer)
% M         = the value for which we are interested in the time it takes
%             for the chain state to exceed this value
```

```
% chain      = a string specifying which chain to use
%               'r' = random
%                At any state i less than M
%                     With probablity p+1/alpha we transition to state i+1
%                     With probability p+1/alpha we stay at state i
%                At any state i greater than or equal to M
%                     With probability 1 we remain in state i
%              NOTE: when using random chain p + 1/alpha must be <= 1
%               's' = semideterministic
%                At any state i less than M
%                     With probablity p we transition to state i+(1+1/alpha)
%                     With probability 1-p we transition to state i+1/alpha
%                At any state i greater than or equal to M
%                     With probability 1 we remain in state i

switch chain

    case 'r'

        hittingTimeVar = M * (1-(p + (1/alpha)))/((p + (1/alpha))^2);

    case 's'

        semiDetDPTable = generateSemiDetDPTable(p, alpha, M);
        DPTableMIdx = M*alpha + 1;
        hittingTimeVar = semiDetDPTable.var(DPTableMIdx);

end
```