Dept. of Math. University of Oslo
Statistical Research Report No. 10
ISSN 0806–3842 August 2004

EXACT SEQUENTIAL SIMULATION OF BINARY VARIABLES GIVEN THEIR SUM

Arne Bang Huseby

Abstract

The paper considers the problem of simulating a vector, $\boldsymbol{X}$ of n independent binary variables conditioned on their sum, S. For a fixed value of S an exact simulation method is provided in Huseby and Naustdal[4]. In certain situations, however, it is of interest to generate an increasing sequence of binary vectors $\boldsymbol{X}_1 < \cdots < \boldsymbol{X}_n$, such that the s-th vector is distributed as the vector $\boldsymbol{X}$ given $S = s$, $s = 1, \ldots, n$. If all the variables of the vector $\boldsymbol{X}$ are identically distributed, it can be shown that this is equivalent to generating a random permutation, $\{\pi_s\}_{s=1}^n$, of the index set, $\{1, \ldots, n\}$. For more details about this, see Huseby and Naustdal[4]. In the present paper, however, we provide a simulation algorithm for the case when the variables of the vector $\boldsymbol{X}$ do not necessarily have the same distribution. This algorithm utilizes the fact that the distribution of a sum of independent binary variables is always log-concave.

1 Introduction

When using Monte Carlo simulations to estimate stochastic properties of a model, it is often necessary to accelerate the convergence of the simulations. One way of doing this is to condition on certain functions of the input variables. In this paper we will focus on the problem of simulating a vector, $\boldsymbol{X} = (X_1, \ldots, X_n)$ of independent binary variables conditioned on their sum, S. It is well-known (see e.g., Huseby and Naustdal[4]) that the distribution of S can be calculated in $O(n^2)$ time. Thus, if $\phi = \phi(\boldsymbol{X})$ is some function of interest, then $E[\phi]$ can often be estimated more efficiently by conditioning on S through the following formula:

$$E[\phi] = \sum_{s=0}^{n} E[\phi \mid S = s] \Pr(S = s) = \sum_{s=0}^{n} \theta_s \Pr(S = s) \tag{1.1}$$

where we have introduced $\theta_s = E[\phi \mid S = s]$ for $s = 0, 1, \ldots, n$. Instead of estimating $E[\phi]$ directly, we estimate the conditional expectations, $\theta_0, \theta_1, \ldots, \theta_n$. This is done by sampling from the conditional distribution of $\boldsymbol{X}$ given $S = s$ for $s = 0, 1, \ldots, n$. It turns out to be easy to sample $\boldsymbol{X}$ from the conditional distribution given $S = s$. An exact method for doing this is provided in Huseby and Naustdal[4]. We will refer to this approach as the *direct sampling method*. The computational complexity of this method is $O(n)$ per simulation. Since, however, this needs to be repeated for $s = 0, 1, \ldots, n$, the total computational complexity of the direct sampling method becomes $O(n^2)$.

In order to obtain faster methods, it is of interest to find an efficient way to generate an increasing sequence of binary vectors, $\boldsymbol{X}_1 < \cdots < \boldsymbol{X}_n$, such that the s-th vector is distributed as the vector $\boldsymbol{X}$ given $S = s$, $s = 1, \ldots, n$. Such a sampling method will be referred to as a *sequential sampling method*. If this can be done with computational complexity lower than $O(n^2)$ per simulated sequence, we have improved the direct method.

A general approach to a sequential sampling method can be described as follows: Let $C = \{1, \ldots, n\}$ denote the index set of $\boldsymbol{X}$. We then sample a permutation, $(i_1, \ldots, i_n)$ of indices as follows: Assume that we have sampled the s first indices of the permutation, i.e., $i_1, \ldots, i_s$, and denote the corresponding index set $\{i_1, \ldots, i_s\}$ by A_s. Then the next index is sampled from the set of remaining indices, i.e., $C \setminus A_s$, with probability:

$$\begin{aligned} \alpha_{A_s,i} = \Pr\left(\text{The } (s+1)\text{-th sampled index is } i \mid A_s\right), \\ \text{for all } i \in C \setminus A_s. \end{aligned} \tag{1.2}$$

The probability of sampling a given permutation, $(i_1, \ldots, i_n)$ is then:

$$\alpha_{A_0,i_1} \cdot \alpha_{A_1,i_2} \cdots \alpha_{A_{n-1},i_n}, \tag{1.3}$$

where $A_0 = \emptyset$.

From this sequence of index sets, $A_1 \subset \cdots \subset A_n$, we obtain the corresponding increasing sequence of binary vectors, $\boldsymbol{X}_1 < \cdots < \boldsymbol{X}_n$, by letting $\boldsymbol{X}_s = \boldsymbol{x}(A_s)$, $s = 1, \ldots, n$, where $\boldsymbol{x}(A)$ denotes the vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ such that $x_i = 1$ if $i \in A$, and 0 otherwise.

The sequential sampling method is characterized by the sampling probability distributions used in each sampling step, i.e., by the $\alpha_{A_s,i}$'s. Using such a sampling method, the probability that after having sampled s indices, we have sampled the set A, where $|A| = s$, is given by:

$$\Pr(A_s = A) = \sum_{(i_1,\ldots,i_s)\in\Pi(A)} \alpha_{\emptyset,i_1} \cdots \alpha_{\{i_1\},i_2} \cdots \alpha_{\{i_1,\ldots,i_{s-1}\},i_s}\ , \tag{1.4}$$

where $\Pi(A)$ denotes the set of all permutations of A.

Now, ideally we want the sequential sampling method to produce a sequence of sets $A_1, \ldots, A_n$ with the "correct" probabilities. That is, we want the $\alpha_{A_s,i}$'s to be chosen such that:

$$\Pr(A_s = A) = \Pr(\boldsymbol{X} = \boldsymbol{x}(A) \mid S = s), \qquad s = 1, \ldots, n, \tag{1.5}$$

In the special case where all the binary variables have the same distribution, this is easily accomplished by using a simple *uniform sampling method*, i.e., by letting:

$$\alpha_{A_s,i} = \frac{1}{n - |A_s|} = \frac{1}{n - s}\ , \qquad \text{for all } i \in C \setminus A_s \tag{1.6}$$

With this choice we get from (1.4) that:

$$\Pr(A_s = A) = \frac{s!}{n(n-1)\cdots(n-s+1)} = \frac{1}{\binom{n}{s}}, \qquad s = 1, \ldots, n, \tag{1.7}$$

which is indeed is the correct probability. In fact for this particular case all the index permutations are equally likely to occur. Thus, all we need to do to

obtain a sequence of binary vectors with the correct conditional distributions, is to generate a random permutation of the indexes. This can be done in $O(n)$ time by using the algorithm of Knuth[7], which of course is significantly better than the direct sampling method. For more details see Huseby and Naustdal[4].

However, when the binary variables have different distributions, it is much more difficult to find the right sampling probability distributions. Several alternative methods, typically involving some kind of importance sampling or weighted sampling, have been suggested for handling this case.

One such approach is to use *weighted sampling without replacement*. This method starts out by assigning weights to each index. Let w_i be the weight assigned to index i, $i = 1, \ldots, n$. The sampling probability distributions are then given by the following:

$$\alpha_{A_s,i} = \frac{w_i}{\sum_{j \notin A_s} w_j} , \qquad \text{for all } i \in C \setminus A_s \tag{1.8}$$

We observe that if all the weights are equal, this method reduces to the uniform sampling method which can be handled by the algorithm of Knuth[7]. The general case, however, can be handled by using a method given in Huseby, Naustdal and Vårli[5]. The computational complexity of this method $O(n \log(n))$ which again is better than the direct approach. The problem with this approach, however, is that it is difficult to find weights that produce the correct sampling distribution. Still the error resulting from using the "wrong" weights can be corrected by using importance sampling.

A different approach is to derive the permutations as follows: Let $U_1, \ldots, U_n$ be n independent absolute continuous random variables. In each simulation we sample $U_1, \ldots, U_n$, and sort the resulting value in decreasing order. Let $U_{i_1} > \cdots > U_{i_1}$ be the ordered sample. The resulting permutation is then $(i_1, \ldots, i_n)$. Note that if the U_i's are identically distributed, then all permutations are equally likely. By choosing the distributions of the U_i's in a suitable way, different permutation distributions can be simulated. The computational complexity of this method is determined by the sorting operation which can be done in $O(n \log(n))$ time. Thus, this approach has the same complexity as weighted sampling without replacement.

Finding distributions for the U_i's such that the resulting permutation distribution is "correct" is again difficult. Still, as with permutation distribution, it is possible to correct the results by using importance sampling. For more details on these approaches see Broström and Nilsson[3], Huseby and Naustdal[4], Huseby, Naustdal and Vårli[5], Lindqvist and Taraldsen[8], and Nilsson[9].

One serious drawback with importance sampling is that the importance factor may affect the statistical properties of the estimates. In some cases such effects can be used to reduce the variance of the estimator, while in other cases the variance may increase significantly. This problem is discussed in Huseby and Naustdal[4]. Thus, finding an efficient exact sequential sampling method is of interest. In the present paper we will show how this can be done using a modified version of the direct sampling method.

2 An Exact Sequential Sampling Method

In order to explain our approach we start out by reviewing the direct sampling method presented in Huseby and Naustdal[4]. As before we let $\boldsymbol{X} = (X_1, \ldots, X_n)$ be a vector of binary variables, and let S denote the sum of these variables. Moreover, we assume that $\Pr(X_i = 1) = p_i,\ i = 1, \ldots, n$. We also introduce the following partial sums:

$$S_m = \sum_{i=m}^{n} X_i\ , \qquad m = 1, \ldots, n\ . \tag{2.1}$$

For convenience we include an empty partial sum as well, and denote this by S_{n+1}. Thus, $S_{n+1} \equiv 0$. The distributions of $S_1, \ldots, S_n$ can be calculated recursively using the following formula:

$$\Pr(S_m = s) = p_m \Pr(S_{m+1} = s - 1) + (1 - p_m) \Pr(S_{m+1} = s), \tag{2.2}$$

where s ranges over the set of possible values of S_m, i.e., $\{0, 1, \ldots, (n-m+1)\}$. Note that (2.2) is valid in the limiting cases where $s = 0$ and $s = (n - m + 1)$, since $\Pr(S_{m+1} = s-1) = 0$ in the first case, while $\Pr(S_{m+1} = s) = 0$ in the last case. Moreover, since we have defined S_{n+1} as the empty partial sum, (2.2) is valid if $m = n$ as well.

In order to calculate the distributions of the partial sums, we start out by determining the distribution of S_n. We then proceed recursively by calculating the distribution of S_{n-1}, etc. A table containing all these distributions (including the distribution of the sum of all the binary variables, i.e., $S = S_1$) can thus be derived in $O(n^2)$-time. Note that all the calculations in this stage are done prior to the actual simulations. When discussing the computational complexity of the sampling procedure, we typically ignore this stage.

The algorithm for sampling from the conditional distribution of $\boldsymbol{X}$ given $S = s$ can now be described as follows. We start out by sampling X_1 from the conditional distribution of $X_1 \mid S = s$. We then continue by sampling X_2 from $X_2 \mid S = s,\ X_1 = x_1$, where x_1 denotes the sampled outcome of X_1, and so on. This turns out to be easy noting that:

$$\begin{aligned}
&\Pr(X_m = x_m \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1}, S = s) \\
&\quad = \frac{\Pr(X_m = x_m, S = s \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1})}{\Pr(S = s \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1})} \\
&\quad = \frac{\Pr(X_m = x_m, S_{m+1} = s - \sum_{j=1}^{m} x_j)}{\Pr(S_m = s - \sum_{j=1}^{m-1} x_j)} \\
&\quad = \frac{p_m^{x_m} (1 - p_m)^{1 - x_m} \Pr(S_{m+1} = s - \sum_{j=1}^{m} x_j)}{\Pr(S_m = s - \sum_{j=1}^{m-1} x_j)}.
\end{aligned} \tag{2.3}$$

We observe that the term $\sum_{j=1}^{m-1} x_j$ keeps track of how many 1's we have generated. As soon as this sum reaches s, the formula (2.3) will prevent us from

generating more 1's since we in this case get that:

$$\begin{aligned}\Pr(X_m = 1 \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1}, S = s) \qquad (2.4)\\ = p_m \frac{\Pr(S_{m+1} = s - \sum_{j=1}^{m-1} x_j - 1)}{\Pr(S_m = s - \sum_{j=1}^{m-1} x_j)}\\ = p_m \frac{\Pr(S_{m+1} = -1)}{\Pr(S_m = 0)} = 0.\end{aligned}$$

A similar property of the formula prevents us from generating too many zeros. In order to see this we rewrite (2.3) in the following equivalent form:

$$\begin{aligned}\Pr(X_m = x_m \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1}, S = s) \qquad (2.5)\\ = \frac{p_m^{x_m}(1-p_m)^{1-x_m} \Pr(S_{m+1} = (s-m) + \sum_{j=1}^{m}(1-x_j))}{\Pr(S_m = (s-m+1) + \sum_{j=1}^{m-1}(1-x_j))}.\end{aligned}$$

In this form the term $\sum_{j=1}^{m-1}(1-x_j)$ keeps track of how many zeros we have generated. As soon as this sum reaches $n-s$, the formula will prevent us from generating more zeros since we in this case get that:

$$\begin{aligned}\Pr(X_m = 1 \mid X_1 = x_1, \ldots, X_{m-1} = x_{m-1}, S = s) \qquad (2.6)\\ = p_m \frac{\Pr(S_{m+1} = s - \sum_{j=1}^{m-1} x_j - 1)}{\Pr(S_m = s - \sum_{j=1}^{m-1} x_j)}\\ = p_m \frac{\Pr(S_{m+1} = n - m)}{\Pr(S_m = n - m + 1)} = 1.\end{aligned}$$

Finally we observe that (2.3) is valid when $m = n$ since we have defined S_{n+1} as the empty partial sum.

Assuming that the distributions of $S_1, \ldots, S_n$ are calculated before running the simulations, we see that all the necessary conditional probabilities can be calculated when needed during the simulations without imposing additional computational complexity. In each simulation run we calculate n probabilities, one for each X_j. Moreover, each of these probabilities is calculated using a fixed number of operations (independent of n). Hence, it follows that sampling from the conditional distribution of $\boldsymbol{X}$ given $S = s$, can be done in $O(n)$ time. In fact, as we shall see, we can calculate all the conditional probabilities we need before running the simulations. This saves additional time, although the computational complexity is still $O(n)$.

Now, sampling a binary variable Z with $Pr(Z = 1) = p$, is usually done by sampling a uniform variable U and letting $Z = I(U \leq p)$, where $I(\cdot)$ denotes the standard indicator function of an event. By using this method, sampling from the conditional distribution of $\boldsymbol{X}$ given $S = s$ can be done as follows: We start out by generating n independent uniformly distributed variables, $U_1, \ldots, U_n$. We then define the X_m's recursively as follows:

$$\begin{aligned}X_m = I(U_m \leq p_m \frac{\Pr(S_{m+1} = s - \sum_{j=1}^{m-1} X_j - 1)}{\Pr(S_m = s - \sum_{j=1}^{m-1} X_j)}) \qquad (2.7)\\ = I(U_m \leq \mu_m(s - \sum_{j=1}^{m-1} X_j)), \qquad m = 1, \ldots, n,\end{aligned}$$

where we have introduced the following functions representing the conditional probabilities used in the right-hand sides of the inequalities above:

$$\mu_m(r) = p_m \frac{\Pr(S_{m+1} = r-1)}{\Pr(S_m = r)} , \qquad m = 1, \ldots, n , \tag{2.8}$$

defined for $r \in \{0, 1, \ldots, (n-m+1)\}$.

By using this procedure we observe that if $s = 1$, exactly *one* of the binary variables will get the value 1, while the other variables will be zero. More generally, for any given value of s, the sampling probabilities (2.7) will produce a binary vector where exactly s of the coordinates will get the value 1, while the other will be zero. This means that by varying s while using the same set of uniformly distributed variables, $U_1, \ldots, U_n$ we may generate a sequence of binary vectors $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$, such that $\boldsymbol{X}_s$ contains exactly s coordinates equal to 1, $s = 1, \ldots, n$. By using this sequence we actually have an exact sequential sampling method. However, in order to show this, we need to prove that the generated sequence actually is increasing. That is, we must show that as soon as a coordinate gets the value 1 somewhere in the sequence, this coordinate will keep this value throughout the sequence. In order to show this, we must prove that the following fraction:

$$\frac{\Pr(S_{m+1} = s - \sum_{j=1}^{m-1} X_j - 1)}{\Pr(S_m = s - \sum_{j=1}^{m-1} X_j)} , \tag{2.9}$$

is nondecreasing in s for all m. This is of course equivalent to proving that the inverse fraction:

$$\frac{\Pr(S_m = r)}{\Pr(S_{m+1} = r-1)} \tag{2.10}$$

is nonincreasing in r for all m, where we have simplified the notation by replacing $s - \sum_{j=1}^{m-1} X_j$ by r.

The numerator of (2.10) can be rewritten by conditioning on X_m as:

$$p_m Pr(S_{m+1} = r-1) + (1-p_m) Pr(S_{m+1} = r). \tag{2.11}$$

By inserting (2.11) into (2.10) and simplifying the resulting expression, it follows that (2.10) is nondecreasing in r for all m if and only if:

$$\frac{\Pr(S_{m+1} = r)}{\Pr(S_{m+1} = r-1)} \tag{2.12}$$

is nonincreasing in r for all m.

It is easy to see that (2.12) has the desired monotonicity property if the following condition holds for all m and r:

$$\Pr(S_{m+1} = r-1) \Pr(S_{m+1} = r+1) \leq [\Pr(S_{m+1} = r)]^2 \tag{2.13}$$

If an integer valued random variable satisfies a condition like (2.13), its distribution is said to be *log-concave*. Thus, we have established that our sampling method will produce an increasing sequence of binary vectors provided that the partial sums (2.1) have log-concave distributions. The proof of this result is given in the appendix.

We close this section by demonstrating the sampling algorithm on a simple example. In this case we consider a situation where $n = 4$, and assume that we want to sample an increasing sequence $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_4$ of vectors of binary variables. Furthermore, let $p_1 = 0.2$, $p_2 = 0.4$, $p_3 = 0.6$, and $p_3 = 0.8$. The resulting distributions of the partial sums, including the empty sum S_5, are given in Table 1. All the conditional probabilities used in the right-hand sides of the inequalities, i.e., the μ_m-functions, can be calculated by combining the numbers from Table 1 and the p_i's. The resulting values are given in Table 2.

$\cdot$	$s = 0$	$s = 1$	$s = 2$	$s = 3$	$s = 4$
S_1	0.0384	0.2464	0.4304	0.2464	0.0384
S_2	0.0480	0.2960	0.4640	0.1920	0.0000
S_3	0.0800	0.4400	0.4800	0.0000	0.0000
S_4	0.2000	0.8000	0.0000	0.0000	0.0000
S_5	1.0000	0.0000	0.0000	0.0000	0.0000

Table 1: The probability distributions of the partial sums.

$\cdot$	$r = 0$	$r = 1$	$r = 2$	$r = 3$	$r = 4$
μ_1	0.0000	0.0390	0.1375	0.3766	1.0000
μ_2	0.0000	0.1081	0.3793	1.0000	n/a
μ_3	0.0000	0.2727	1.0000	n/a	n/a
μ_4	0.0000	1.0000	n/a	n/a	n/a

Table 2: The μ_m-functions.

In order to generate the sequence, we start out by generating four uniformly distributed variables, $U_1, \ldots, U_4$. Assume e.g., that the outcomes are $U_1 = 0.358$, $U_2 = 0.291$, $U_3 = 0.516$, $U_4 = 0.891$. The resulting increasing sequence $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_4$ of vectors is then calculated by using (2.7). We have divided this process into four steps corresponding to the four values of s and the corresponding four binary vectors. In each step each U_m is compared to the corresponding value of $\mu_m(r)$, where r is the number of remaining 1's to be generated in that step. If $U_m > \mu_m(r)$, the resulting binary variable is zero, otherwise the variable is 1.

Step 1. In this step $s = 1$, and we carry out the following comparisons:

$$
\begin{aligned}
U_1 = 0.358 > \mu_1(1) = 0.0390 &\Rightarrow X_1 = 0.\\
U_2 = 0.291 > \mu_2(1) = 0.1081 &\Rightarrow X_2 = 0.\\
U_3 = 0.516 > \mu_3(1) = 0.2727 &\Rightarrow X_3 = 0.\\
U_4 = 0.891 < \mu_4(1) = 1.0000 &\Rightarrow X_4 = 1.
\end{aligned}
$$

That is, $\boldsymbol{X}_1 = (0, 0, 0, 1)$.

Step 2. In this step $s = 2$, and we carry out the following comparisons:

$$\begin{aligned}
U_1 = 0.358 > \mu_1(2) = 0.1375 &\Rightarrow X_1 = 0.\\
U_2 = 0.291 < \mu_2(2) = 0.3793 &\Rightarrow X_2 = 1.\\
U_3 = 0.516 > \mu_3(1) = 0.2727 &\Rightarrow X_3 = 0.\\
U_4 = 0.891 < \mu_4(1) = 1.0000 &\Rightarrow X_4 = 1.
\end{aligned}$$

That is, $\boldsymbol{X}_2 = (0, 1, 0, 1)$.

Step 3. In this step $s = 3$, and we carry out the following comparisons:

$$\begin{aligned}
U_1 = 0.358 < \mu_1(3) = 0.3766 &\Rightarrow X_1 = 1.\\
U_2 = 0.291 < \mu_2(2) = 0.3793 &\Rightarrow X_2 = 1.\\
U_3 = 0.516 > \mu_3(1) = 0.2727 &\Rightarrow X_3 = 0.\\
U_4 = 0.891 < \mu_4(1) = 1.0000 &\Rightarrow X_4 = 1.
\end{aligned}$$

That is, $\boldsymbol{X}_3 = (1, 1, 0, 1)$.

Step 4. In this step $s = 4$, and we carry out the following comparisons:

$$\begin{aligned}
U_1 = 0.358 < \mu_1(4) = 1.0000 &\Rightarrow X_1 = 1.\\
U_2 = 0.291 < \mu_2(3) = 1.0000 &\Rightarrow X_2 = 1.\\
U_3 = 0.516 < \mu_3(2) = 1.0000 &\Rightarrow X_3 = 1.\\
U_4 = 0.891 < \mu_4(1) = 1.0000 &\Rightarrow X_4 = 1.
\end{aligned}$$

That is, $\boldsymbol{X}_4 = (1, 1, 0, 1)$.

3 Computational Complexity Considerations

In order to determine the computational complexity of the sampling algorithm we consider the example from the previous section one more time. Firstly, we see that the algorithm consists of two main stages. In the first stage we calculate the μ_m-functions. It is easy to see that this can be done in $O(n^2)$ time. As for the direct sampling method we ignore this stage when discussing the computational complexity of the method. The second stage consists of the actual simulations. In each simulation run, we start out by generating the U_m-s. This is done in $O(n)$ time. We then proceed by calculating the resulting sequence of binary vectors. As in the above example, this involves going through n steps, one for each vector in the sequence. In each step the binary vector is determined by evaluating a set of inequalities, one inequality for each coordinate of the vector. Thus, if the dimension of the binary vector is n, then for each of the n steps, we need to evaluate (at most) n inequalities. This implies that all the n steps can be completed in $O(n^2)$ time. The total computational complexity of the algorithm is then $O(n^2)$ per simulation, which is the same as for the direct sampling method.

Note, however, that if $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ is the sequence of vectors, then $\boldsymbol{X}_{i-1}$ and $\boldsymbol{X}_i$ differ only in a single coordinate which is changed from zero to 1 when

going from $\boldsymbol{X}_{i-1}$ to $\boldsymbol{X}_i$. Thus, assume that we have computed $\boldsymbol{X}_{i-1}$ in Step $(i-1)$. Then in Step i we only need to evaluate inequalities for indices where the corresponding coordinate in $\boldsymbol{X}_{i-1}$ is zero. As soon as we find an index m such that the corresponding coordinate in $\boldsymbol{X}_{i-1}$ is zero, and such that the generated value of U_m is below the relevant μ_m-value, we have identified the coordinate which should be changed from zero to 1. Hence, we do not have to evaluate the remaining inequalities.

The number of inequalities we need to evaluate, depends on the the generated values of the U_m's. Thus, this number will change from simulation to simulation. In the best cases the generated values are such that the resulting sequence is $\boldsymbol{X}_1 = (1, 0, \ldots, 0), \boldsymbol{X}_2 = (1, 1, 0, \ldots, 0), \ldots, \boldsymbol{X}_n = (1, 1, \ldots, 1)$. In such cases only one new inequality needs to be evaluated in each step, making the total number of evaluations for all the steps together equal to n. On the other hand in the worst cases we may end up with the sequence $\boldsymbol{X}_1 = (0, \ldots, 0, 1), \boldsymbol{X}_2 = (0, \ldots, 0, 1, 1), \ldots, \boldsymbol{X}_n = (1, 1, \ldots, 1)$. For such cases the total number of evaluations is $n(n+1)/2$.

By expanding (2.8) it follows that $\mu_m(r)$ can be written as:

$$\mu_m(r) = \frac{p_m \Pr(S_{m+1} = r-1)}{p_m \Pr(S_{m+1} = r-1) + (1-p_m) \Pr(S_{m+1} = r)}, \tag{3.1}$$

for $m = 1, \ldots, (n-1)$ and $r = 0, 1, \ldots, (n-m+1)$. From this it is easy to see that $\mu_m(r)$ is increasing in p_m for $m = 1, \ldots, (n-1)$.

In order to minimize the number of inequality evaluations, we want the lower coordinates of $\boldsymbol{X}$ to have the larger probabilities of being 1 and the higher coordinates of $\boldsymbol{X}$ to have the smaller probabilities of being 1. Thus, we should rearrange the indices so that $p_1 \geq p_2 \cdots \geq p_n$. If this is done, and the p_m's are far away from each other, the number of inequality evaluations may be reduced significantly.

To investigate more closely how much the ordering of the indices influences the running time of the algorithm, we consider the following four cases.

Case 1. Linearly increasing probabilities:

$$p_m = \frac{m}{n+1}\,, \qquad m = 1, \ldots, n\,, \tag{3.2}$$

Case 2. Equal probabilities:

$$p_m = 0.5\,, \qquad m = 1, \ldots, n\,, \tag{3.3}$$

Case 3. Linearly decreasing probabilities:

$$p_m = 1 - \frac{m}{n+1}\,, \qquad m = 1, \ldots, n\,, \tag{3.4}$$

Case 4. Exponentially decreasing probabilities:

$$p_m = 0.9^m\,, \qquad m = 1, \ldots, n\,, \tag{3.5}$$

For each of these cases we let n vary in steps of 50 from 50 to 250. For each combination of the four above cases and the five different n-values, we run 10000 simulations and calculate the mean number of inequality evaluations. The results are presented in Table 3 and in Figure 1. The four curves in the plot

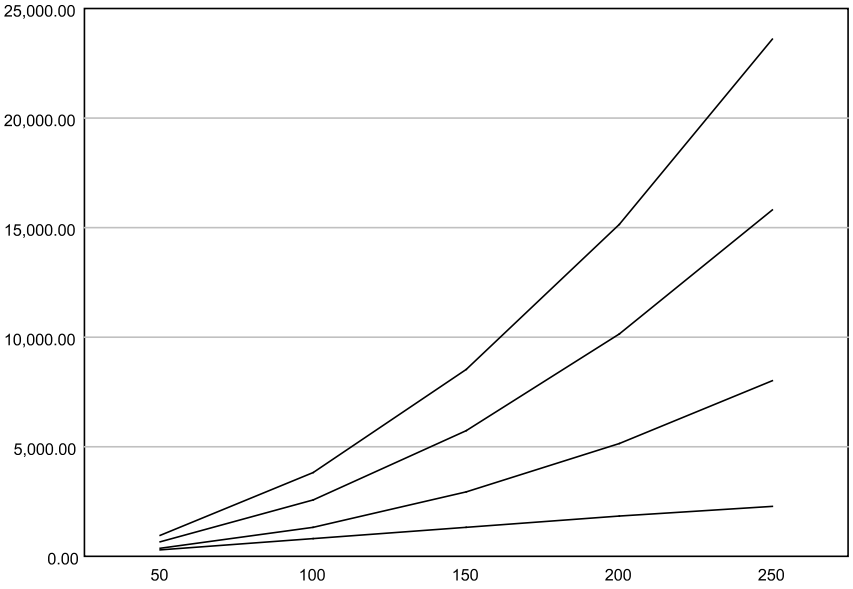

Figure 1: Mean number of inequality evaluations.

represent the four above cases, and show how the mean number of inequality evaluations varies with n. The steepest curve represents Case 1, the second steepest represents Case 2 and so on.

It is not surprising that Case 1 has the worst results as this represents a situation where the indices are sorted in the worst possible way. In Case 2 all the probabilities are equal, so there is no point in rearranging the indices. While the algorithm shows better performance for this case compared to Case 1, the results are not satisfactory. For this particular case it is much better to use the random permutation method by Knuth[7] instead, as the computational complexity of this specialized algorithm is just $O(n)$ per simulation. In the two last cases, however, we see that the performance is improved significantly. In Case 3 the probabilities are spread out evenly on the interval [0, 1]. For small n this results in very good performance. As n grows, however, the intervals between probabilities becomes shorter. As a result the sorting the indices has less effect. In Case 4 the numbers are distributed such that the probability associated with any given index is kept constant as n grows. As a result the performance is more stable.

In order to get a better impression of the performance, we consider the simulation results given in Table 3. Column 4 of this table contains the ratios between the mean number of inequality evaluations and n. In all cases these ratios grow as n grows, indicating that none of cases result in linear computational time. In Column 5 we present the ratio between the mean and $n\log(n)$. In Case 4 we see that this ratio is fairly stable indicating that for this case the mean computational complexity is approximately $O(n\log(n))$. The last column

contains ratios between the mean and the maximum number of inequality evaluations, i.e., $n(n+1)/2$. We observe that these ratios are approximately 0.75, 0.5, and 0.25 for Case 1, 2 and 3 respectively. While a ratio of 0.25 represents a significant reduction compared to the maximum number of inequality evaluations, the average computational complexity of the algorithm is still approximately $O(n^2)$. For Case 4 we see that the ratio is decreasing, indicating once again a lower computational complexity.

$Case$	n	$mean$	$mean/n$	$mean/[n\log(n)]$	$mean/[n(n+1)/2]$
1	50	971	19.421	4.964	0.762
1	100	3818	38.180	8.291	0.756
1	150	8540	56.932	11.362	0.754
1	200	15132	75.659	14.280	0.753
1	250	23612	94.448	17.106	0.753
2	50	663	13.251	3.387	0.520
2	100	2577	25.766	5.595	0.510
2	150	5738	38.253	7.634	0.507
2	200	10158	50.792	9.586	0.505
2	250	15805	63.218	11.450	0.504
3	50	352	7.037	1.799	0.276
3	100	1331	13.314	2.891	0.264
3	150	2934	19.561	3.904	0.259
3	200	5167	25.835	4.876	0.257
3	250	8013	32.051	5.805	0.255
4	50	322	6.442	1.647	0.253
4	100	814	8.139	1.767	0.161
4	150	1314	8.760	1.748	0.116
4	200	1815	9.076	1.713	0.090
4	250	2311	9.245	1.674	0.074

Table 3: Mean number of inequality evaluations

4 Conclusions and further Work

The main objective of this paper has been to provide an exact method for sampling an increasing sequence of vectors of binary variables which is faster than the direct method. In general we cannot guarantee that the computational complexity of this method is better than for the direct method. The actual improvement will typically depend on the success probabilities of the coordinate variables. If these probabilities are far away from each other, and the coordinate variables are sorted so that the corresponding sequence of success probabilities is decreasing, then the proposed method is significantly faster than the direct method.

It should be noted that by using the proposed method, the resulting increasing sequence of vectors will be strongly dependent of each other. In contrast the direct method generates a sequence of independent vectors. How this will affect the actual use of the method, remains to be seen. Sometimes, however, this kind of dependence can be an advantage. See e.g. Huseby and Naustdal[4].

In a future study we will investigate this further, and compare the method to methods based on importance sampling.

5 Appendix

In this section we shall prove that the distribution of a sum of independent binary variables is always log-concave. For an introduction to the theory of log-concave probability distributions see e.g., An[1] or An[2]. An[1] includes the result that log-concavity is preserved under convolutions. From this our result follows more or less immediately. However, we shall provide a more direct argument. We start out by giving the definition of log-concavity for integer valued random variables.

Definition 5.1 *Let S be an integer valued random variable. S is said to have a log-concave distribution if the following property holds for all s:*

$$\Pr(S = s-1)\Pr(S = s+1) \leq [\Pr(S = s)]^2 \tag{5.1}$$

We observe that if S has a log-concave distribution, then (5.1) implies that:

$$\frac{\Pr(S = s)}{\Pr(S = s-1)} \geq \frac{\Pr(S = s+1)}{\Pr(S = s)} \tag{5.2}$$

whenever both the denominators are nonzero. Moreover, by taking logarithms we get that (5.1) also implies that the difference:

$$\log(\Pr(S = s)) - \log(\Pr(S = s-1)) \tag{5.3}$$

is nonincreasing in s whenever the logarithms exist. Thus, we see that the logarithm of a log-concave distribution indeed has a concave shape. Still the condition (5.1) is easier to use as we do not have to assume that the probabilities are positive. In fact (5.1) is trivially satisfied for those values of s where either $\Pr(S = s-1)$, $\Pr(S = s+1)$, or both are zero. From this it follows e.g., that if S is binary, then the distribution of S is trivially log-concave, since for this very simple type of distribution any choice of s will imply that either $\Pr(S = s-1)$, $\Pr(S = s+1)$, or both are zero.

We are now ready to prove our main result:

Theorem 5.2 *Let $X_1, \ldots, X_n$ be independent binary variables, and let S be the sum of these variables. Then the distribution of S is log-concave.*

Proof: We let $\Pr(X_i = 1) = p_i$ and $\Pr(X_i = 0) = q_i$, $i = 1, \ldots, n$. The result is proved by induction on the number of binary variables. As already pointed out, the result is trivial if $n = 1$. We then consider the general case and assume that the result has already been proved to hold for sums of $(n-1)$ or fewer binary variables. We then introduce T as the sum of the first $(n-1)$ of the X_i's, and let S be the sum of all the n X_i's. Thus, $S = T + X_n$. We also introduce the following simplified notation for the distributions of T and S:

$$\begin{aligned} &\Pr(T = i) = \pi_i,\ i = 0, 1, \ldots, (n-1), \\ &\Pr(S = i) = \tilde{\pi}_i,\ i = 0, 1, \ldots, n. \end{aligned} \tag{5.4}$$

By the induction hypothesis we know that T has a log-concave distribution. That is, the following inequality holds true for all i:

$$\pi_{i-1}\pi_{i+1} \leq \pi_i^2. \tag{5.5}$$

By applying the inequality (5.5) twice it is easy to see that we also have the following inequality for all i:

$$\pi_{i-2}\pi_{i+1} \leq \pi_{i-1}\pi_i. \tag{5.6}$$

We now consider the distribution of S. By conditioning on X_n we get that:

$$\begin{aligned} \tilde{\pi}_{i-1}\tilde{\pi}_{i+1} &= (\pi_{i-1}q_n + \pi_{i-2}p_n)(\pi_{i+1}q_n + \pi_i p_n) \\ &= \pi_{i-1}\pi_{i+1}q_n^2 + (\pi_{i-1}\pi_i + \pi_{i-2}\pi_{i+1})p_n q_n + \pi_{i-2}\pi_i p_n^2 \end{aligned} \tag{5.7}$$

Hence, it follows by applying (5.5) and (5.6) that:

$$\begin{aligned} \tilde{\pi}_{i-1}\tilde{\pi}_{i+1} &\leq \pi_i^2 q_n^2 + 2\pi_{i-1}\pi_i p_n q_n + \pi_{i-1}^2 p_n^2 \\ &= (\pi_i q_n + \pi_{i-1}p_n)^2 \\ &= \tilde{\pi}_i^2. \end{aligned} \tag{5.8}$$

Thus, it follows that the distribution of S is log-concave as well, and so the result is proved by induction. ■

Acknowledgments

The author is grateful to Professor Bo Lindqvist for helpful discussions.

References

[1] M. Y. An, *Log-concave Probability Distributions: Theory and Statistical Testing.* Economics Working Paper Archive at WUSTL, Game Theory and Information, **9611002** (1996).

[2] M. Y. An, Logconcavity versus Logconvexity: A Complete Characterization. *Journal of Economic Theory* **80** (1998).

[3] G. Broström, and L. Nilsson, Acceptance/rejection sampling from the conditional distribution of independent discrete random variables, given their sum, *Statistics* **34**, 247 (2000).

[4] A. B. Huseby, and M. Naustdal, Improved Simulation Methods for System Reliability Evaluation, in *Mathematical and Statistical Methods in Reliability*, World Scientific Publishing Co. Pte. Ltd., 105-121 (2003).

[5] A. B. Huseby, M. Naustdal, and I. D. Vårli, *System Reliability Evaluation Using Conditional Monte Carlo Methods*, Statistical Research Report **2**, University of Oslo (2004).

[6] A. B. Huseby, *System reliabiliy estimation using conditional Monte Carlo simulation*, Fourth International Conference On Mathematical Methods In Reliability, Santa Fe, New Mexico (2004).

[7] D. E. Knuth, *The art of computer programming, Vol. 2, Seminumerical algorithms*, Addison-Wesley, Reading, Mass. second edition, (1981).

[8] B. H. Lindqvist and G. Taraldsen, *Monte Carlo Conditioning on a Sufficient Statistics*, Preprint Statistics **9**, Norwegian University of Science and Technology (2001).

[9] L. Nilsson, *On the simulation of conditional distributions in the Bernoulli case*, Research report **14**, Department of Mathematical Statistics, Umeå University, Sweden (1997).