Design Flaws in the Implementation of the Ziggurat and Monty Python methods (and some remarks on Matlab randn)

Boaz Nadler[*]

boaz.nadler@weizmann.ac.il

February 14, 2006

Abstract

Ziggurat and *Monty Python* are two fast and elegant methods proposed by Marsaglia and Tsang to transform uniform random variables to random variables with normal, exponential and other common probability distributions. While the proposed methods are theoretically correct, we show that there are various design flaws in the uniform pseudo random number generators (PRNG's) of their published implementations for both the normal and Gamma distributions [1, 2, 3]. These flaws lead to non-uniformity of the resulting pseudo-random numbers and consequently to noticeable deviations of their outputs from the required distributions. In addition, we show that the underlying uniform PRNG of the published implementation of Matlab's randn, which is also based on the Ziggurat method, is not uniformly distributed with correlations between consecutive pairs. Also, we show that the simple linear initialization of the registers in matlab's randn may lead to non-trivial correlations between output sequences initialized with different (related or even random unrelated) seeds. These, in turn, may lead to erroneous results for stochastic simulations.

1 Introduction

Pseudo random number generators (PRNG) are a key component of stochastic simulations. Most PRNG's produce sequences of (seemingly) uniformly distributed real numbers in the interval $[0, 1)$, typically by quantizing a sequence of integers in the set $\Omega_k = \{0, 1, 2^k - 1\}$ via division by 2^k. Random variables with normal (Gaussian), Gamma or other distributions, are then typically constructed via transformations of uniform random variables [4, 5].

In [6], Marsaglia and Tsang proposed the Ziggurat method to generate random variables with a decreasing or symmetric unimodal density from uniform random variables. While in the original paper [6] the authors assumed the existence of a suitable PRNG that outputs uniform random numbers in the range $[0, 1)$, in [1] and [2], the same authors refined their original method and suggested a specific underlying fast uniform PRNG along with computer code to produce normal or Gamma distributed numbers. In a different work [3], the same authors proposed the Monty Python method to generate random variables with normal, exponential or other common distributions, and proposed a specific implementation based on a multiply-with-carry (MWC) uniform PRNG. In this paper we show that there are various flaws in the design of these underlying PRNG's, which lead to significant deviations of their outputs from uniformity, and consequently poor distributions of the resulting normal or Gamma distributions.

Statistical problems with the implementation of [1] were recently noted by Leong & al [7], who reported that the resulting sequences of normally distributed numbers fail the simple χ^2 test, and attributed this finding to the (relatively) short period of the underlying PRNG $(2^{32} - 1)$. Our analysis elucidates the design flaws leading to these statistical problems, which are mainly due to non-uniformity and correlations of the outputs of the PRNG. While the short period of the specific suggestion of [1, 2], only magnifies these problems, we show that other PRNG's with much longer periods but same output function lead to the same statistical problems.

Another normal random number generator based on the Ziggurat method is Matlab's built-in function `randn`. We analyze the underlying uniform PRNG of this function, based on the matlab code published in [8]. We show that while individual outputs of this PRNG are uniformly distributed, pairs of consecutive outputs are correlated. Since the output of the Ziggurat method is highly non-linear in its input, it is difficult to detect these correlations in the resulting normal random numbers. However, we show that initializations of the function `randn` with different, either related or even random unrelated seeds, as done in parallel implementations and other stochastic simulations, may lead to non-trivial correlations between the resulting output sequences of random numbers. We give a simple example where a sequence of such initializations yields incorrect results for a stochastic simulation.

The paper is organized as follows. In section 2 we present a probabilistic setting in which we can properly define statistical properties of sequences of random numbers from PRNG's. The design flaws and statistical weaknesses of the PRNG's published in [1, 2] are analyzed in section 3. In Section 4 we analyze the weaknesses in the multiply with carry generator and their consequences on the Monty Python method of [3]. The analysis of Matlab's `randn` is presented in Section 5. We conclude in Section 6 with a summary and discussion.

2   Randomness and statistical requirements from a PRNG

The main goal of a uniform PRNG on a set of possible outputs Ω is to produce long sequences of numbers which imitate, in a statistical sense, realizations of a sequence of corresponding independently identically distributed (i.i.d.) uniform random variables on the same set.

While sequences of random variables are a well defined concept within the framework of probability theory, the notion of "randomness" in a sequence of numbers produced by a deterministic algorithm is unclear and requires proper definition.

Following [9], we consider a uniform random number generator as a finite state machine (F, g, s, P) with an internal state set $\mathcal{S}$ and output set Ω. The current state of the PRNG is denoted by $s \in \mathcal{S}$, $F : \mathcal{S} \rightarrow \mathcal{S}$ is the transition function, $g : \mathcal{S} \rightarrow \Omega$ is the output function, and $P : \mathcal{T} \rightarrow \mathcal{S}$ is the initialization function, where $\mathcal{T}$ is a set of possible seeds. The internal state is updated according to $s_{i+1} = F(s_i)$ while the output at step i is $g(s_i)$. The state machine is initialized by a seed, such that $s_0 = P(seed)$. We denote by $u(s_0, j)$ the j-th output of the generator initialized with state s_0. Note that $\mathcal{S}$ and Ω need not (and in general should not) be of the same size. For example, the internal state could be of size $|\mathcal{S}| = 2^{128}$ (e.g. 128 bits), while the output set Ω could be of size 2^{32}.

Since the PRNG is deterministic, the output sequence is uniquely determined by the initial seed and the functions F, g, P. Moreover, since F is a finite state machine, it is obviously periodic with some period $l \leq |\mathcal{S}|$. We introduce a probability space in this setting by considering both the time of observation and the initial state s_0 as *random variables*, with the initial state uniformly distributed over the set $\mathcal{S}$ and the time of observation uniformly distributed over the integer set $\{1, 2 \ldots l\}$. We denote by $\{U_1, U_2, \ldots\}$ the resulting sequence of random variables.

Ideally, this sequence should have the same probability distribution as a sequence of i.i.d. uniform random variables $\{X_1, X_2, \ldots\}$ over the set Ω. Obviously, this requirement does not hold, since $U_{i+l} = U_i$ for example. Therefore, one of the basic requirements from a PRNG is to have a very long period $l \gg 1$, significantly longer than the total number of outputs used by the simulation. In addition, inside the long period we require that at least the *low order statistics* of $\{U_i\}$ and $\{X_i\}$ coincide. Specifically, we consider a PRNG (F, g, s, P) as statistically sound if it satisfies (at least) the following requirements (see also [10]):

1. *Uniformity of first order statistics on Ω:* We require that

$$\Pr\{U_1 = u \in \Omega\} :\equiv \frac{1}{|\mathcal{S}|l} \sum_{s_0 \in \mathcal{S}} \sum_{t=1}^{l} \delta(u(s_0, t), u) = \frac{1}{|\Omega|} \tag{1}$$

where $\delta(i, j)$ is the Kroneker delta function, equal to one if $i = j$ and zero otherwise.

2. *Uniformity of second order statistics on Ω:* We require that

$$\Pr\{(U_1, U_2) = (u_1, u_2)\} :\equiv \frac{1}{|\mathcal{S}|l} \sum_{s_0 \in \mathcal{S}} \sum_{t=1}^{l} \delta(u(s_0, t), u_1)\delta(u(s_0, t+1), u_2) = \frac{1}{|\Omega|^2} \tag{2}$$

Note that combining requirements (1) and (2) implies that the *conditional* distribution of U_2 given U_1 is also uniform on the set Ω. In other words, observation of a single output does not affect the distribution of the next output.

3. *Insensitivity to initialization with related seeds:* Recall that the initial state is set according to $s_0 = P(seed)$. Let U, U' be the random variables that correspond to initializations with seeds that differ by Δ. We require that for any $\Delta \in \mathcal{T} \setminus \{0\}$,

$$\Pr\{U = u, U' = u'\} = \sum_{seed \in \mathcal{T}} \delta(u(P(seed), j), u)\delta(u(P(seed + \Delta), j), u') = \frac{1}{|\Omega|^2} \quad (3)$$

For many PRNG's, the set of requirements (1)-(3) does not hold exactly. We still consider a PRNG as statistically acceptable if the discrepancy between these distributions and the corresponding uniform ones is extremely small, say below a threshold ε, such that detection of this discrepancy would require more than 2^{100} outputs, for example.

Obviously, PRNG's need to satisfy many more requirements to be considered acceptable from a statistical point of view, and their output sequences are typically required to pass various empirical statistical tests (see, for example [4, 10, 12, 15] and references therein). However, as we shall see below, each one of the requirements (1)-(3) is *essential* in the context of both the Ziggurat and the Monty Python methods, and possibly so in the more general context of stochastic simulations. While requirements (1) and (2) seem obvious, requirement (3) is important when different runs are made with different seeds, as in parallel implementations of stochastic simulations.

3   Design Flaws in the uniform RNG of [1, 2]

For the paper to be reasonably self contained, we briefly describe the basic Ziggurat method. To generate a non-negative random variable with a monotonically decreasing density $f(x)$ from a uniform r.v. $U[0, 1]$, we choose k points $0 = x_0 < x_1 < x_2 < \ldots < x_{k-1}$, such that

$$x_i(f(x_{i-1}) - f(x_i)) = x_{k-1}f(x_{k-1}) + \Pr\{x > x_{k-1}\} \quad 1 \leq i \leq k - 1$$

Given the set $\{x_i\}_{i=0}^{k-1}$, the Ziggurat method works as follows:

1. Choose an index $0 \leq i < k$ at random with uniform probability $1/k$.

2. Draw a random number u from the uniform distribution $U[0, 1]$, and let $x = ux_i$. If $i \geq 1$ and $x < x_{i-1}$ return x.

3. If $i \geq 1$ draw another uniform random number y. If $y(f(x_{i-1}) - f(x_i)) < f(x) - f(x_i)$ return x.

4. If $i = 0$, generate an x from the tail $x > x_{127}$ and return x.

5. Otherwise, return to step 1.

The generation of random numbers from the tail of the distribution is described explicitly below. In most applications k is chosen to be a power of 2, (typically $k = 64, 128, 256$), so

that choosing a random index with probability $1/k$ is easily done by considering the first few bits of a random 32 bit uniformly distributed integer, for example.

The key point and the beauty of the Ziggurat method is that if the two numbers x and y in steps 2 and 3 are indeed (statistically) random, independent and uniformly distributed over $[0, 1)$, then the output will be a truly normal distributed random variable. The original publication [6] described the method with an unspecified underlying uniform PRNG. However, in [1, 2] the following code for steps 1-3 was suggested by Marsaglia and Tsang, using $k = 128$,

```
unsigned long jsr,jz;
long hz,iz;
#define SHR3 (jz=jsr, jsr^=(jsr<<13), jsr^=(jsr>>17),jsr^=(jsr<<5),jz+jsr)
#define RNOR (hz=SHR3, iz=hz&127, (fabs(hz)<kn[iz])? hz*wn[iz] : nfix())
#define UNI (.5 + (signed) SHR3*.2328306e-9)

float nfix()
{   float x,y;
    for(;;){
        x = hz * wn[iz];
        if(iz==0){ // generate an output from the tail
            do{ x=-log(UNI)/x[k-1]; y=-log(UNI);}
            while(y+y<x*x);
            return (hz>0)? r+x : -r-x;
        }
        if(fn[iz]+UNI*(fn[iz-1]-fn[iz]) < exp(-.5*x*x) ) return x;

        hz=SHR3; iz=hz&127; if(fabs(hz)<kn[iz]) return hz*wn[iz];
    }
}
```

First, a few explanatory words on the code above: The inline code RNOR produces a normal random number. It first calls SHR3, which both updates the 32-bit register jsr and outputs a 32-bit integer, which should be uniformly distributed in the set $0 \ldots 2^{32} - 1$. To produce both the positive and negative parts of the normal distribution, the output of SHR3 is assigned to the (signed) variable hz of type long, The two tables kn and wn are initialized to store the following values: $\texttt{kn[i]} = 2^{-31} x_{i-1}/x_i$ and $\texttt{wn[i]} = x_i/2^{31}$ for $i \geq 1$ and special values for $i = 0$. The procedure nfix() takes care of steps 3-5, whenever step 2 fails.

The register jsr is updated via a linear transformation made up of three shifts, hence the name SHR3 (shift register 3). For future use we denote this linear transformation by T, so that $\texttt{jsr}^{(t+1)} = \texttt{T}(\texttt{jsr}^{(t)})$. According to [16] this transformation has maximal period, e.g. $2^{32} - 1$. Note, however, that the output of SHR3 is $\texttt{jsr}+\texttt{T}(\texttt{jsr})$ (mod 2^{32}), and not the value of jsr itself. We will come back to this point later on.

In our analysis we will need estimates on the number of outputs required to distinguish between a discrete distribution over k values with probabilities $(p_1, \ldots, p_k)$ and one with

probabilities $(q_1, \ldots, q_k)$ with $q_i = p_i(1 + \varepsilon_i)$. As shown in the appendix, for a distinguisher based on the χ^2 statistic, the number of required outputs is of the order of

$$N = \frac{\sqrt{2k}}{\displaystyle\sum_{j=1}^{k} p_j \varepsilon_j^2} = O(1/\varepsilon^2) \tag{4}$$

`Design Flaw # 1:` The first problem we observe, as also recently noted by Doornik [11], is that the *same* 7 least significant bits of `hz` are used both for choosing the random index $0 \leq i \leq 127$ in step 1, and for the uniform random number u in step 2 of the algorithm. This introduces statistical dependencies into the algorithm in two different locations: i) All random numbers from the i-th index all end with the same 7 least significant bits, and ii) computation of rejection probabilities at step 2. Let us roughly estimate this second deviation and its shortcomings: For a 32 bit uniform random number and a table with $k = 128$ (e.g. 7 bits), the fact that the last 7 bits are fixed induces errors in the rejection question at step 2 (whether $ux_i < x_{i-1}$) of the order of $\varepsilon = 1/2^{32-7} = 2^{-25}$ (instead of the quantization error of 2^{-32}). Therefore, according to (4), to detect such a deviation one would need $O(1/\varepsilon^2) = 2^{50}$ outputs. While this design flaw certainly produces a deviation from the normal distribution, it is quite negligible as compared to the next design flaw that we now describe.

`Design Flaw # 2:` The output of `SHR3`, of the form `x+Tx` is highly non-uniform and fails to satisfy the basic requirement (1). Due to the specific structure of `T`, the output `x+Tx` is not one-to-one, but rather a contractive mapping with 1543756180 outputs (about $2^{30.5}$) not possible at all. Thus the output range of `SHR3` is restricted to only about 64% of the possible 2^{32} outcomes, with some values 10 times more probable than expected in a uniform distribution. Table 1 shows the distribution of the number of sources of a possible output value y, e.g. the number of values x such that $x + Tx = y$.

This non-uniform distribution of `SHR3` yields non-negligible deviations from normality for the resulting normal random numbers. Due to the structure of `T` one can prove that the lowest seven bits of `x+Tx` are uniformly distributed. Therefore, the probability of choosing a specific index i in step 1 of the algorithm is still $1/128$ as should be. However, the resulting variables ux_i are not uniform over $[0, x_i]$ and in the expected rejection probabilities at step 2 of the algorithm deviate from the correct ones. We now describe the effects of these deviations and estimate the number of outputs needed to detect them in a simple χ^2 test on the resulting normal numbers.

Let Z denote a standard Gaussian variable with zero mean and unit variance. For $j = 1, \ldots, k-1$ we define $p_j = \Pr\{x_{j-1} < |Z| < x_j\}$ and $p_k = \Pr\{|Z| > x_{k-1}\}$. Let q_j denote the corresponding probabilities in the Ziggurat algorithm with `SHR3` as its underlying PRNG. Then, by definition

$$q_j = \Pr\{x_{j-1} < |x| < x_j\} = \sum_{i=0}^{k-1} \Pr\{\text{index chosen is } i\} \Pr\{x_{j-1} < |x| < x_j| \text{ index } i\} \tag{5}$$

Since the seven least significant bits of `SHR3` are uniformly distributed $\Pr\{\text{index chosen is } i\} = 1/128$. However, the probabilities $\Pr\{x_{j-1} < |x| < x_j | \text{index } i\}$, deviate from the theoretically expected ones. To estimate q_j we performed the following calculation: We passed over

# sources	# of outputs
0	1543756180
1	1616832933
2	808153149
3	256471123
4	58117590
5	10068341
6	1391608
7	159565
8	15358
9	1334
10	109
11	5
12	1
13	0

Table 1: Distribution of the number of sources of the transformation $x + Tx$.

all $2^{32} - 1$ possible initial values for the register `jsr`, computed the first output of `RNOR`, and created a histogram of hits into the 128 bins $[x_{i-1}, x_i]$ and $[x_{127}, \infty)$. In table 2 we present the eight bins with the largest relative deviations (measured as $p_i \varepsilon_i^2$, where $q_i = p_i(1 + \varepsilon_i)$). Applying formula (4), we estimate that after an order of 2^{30} outputs, these deviations from the normal distribution can be detected with a χ^2 test on these 128 bins.

This result is not due to the relatively short period of the register `jsr`. In figure 1 we present numerical results of the χ^2 test done on 200 bins, evenly spaced in the interval $[-7.0, 7.0]$ as done by Leong et al [7], for two other underlying PRNG's with much longer periods, but whose output is computed via `x+Tx`. The two generators are either a combination of `SHR` with `CONG`, a multiplicative congruential generator, which is the underlying generator of matlab's `randn`, and the `KISS` generator [7], which combines also a multiply with carry register. We stress that in both cases, the output is `x+Tx` instead of the original `x`, and as expected the χ^2 statistics start to significantly deviate from the expected value after roughly 2^{32} outputs.

3.1   A quick fix ? Wrong Tail Probabilities !

Since `x+Tx` is non-uniform, a possible and natural "quick fix" is to replace the output of `SHR3` by the state of the 32-bit register `jsr`, via the following inline code `SHR0`,

```
#define SHR0 (jsr^=jsr<<13, jsr^= jsr>>17, jsr^=jsr<<5, jsr)
```

Since `jsr` is a maximal length 32-bit shift register, when averaged over its period, its state is a uniformly distributed 32-bit integer number in the range $1 \dots (2^{32} - 1)$, and thus approximately satisfies requirement (1). One may thus be tempted to conclude that the output sequence of the Zigggurat method with `SHR0` should easily pass the χ^2 statistical test of [7].

Interval i	$[x_{i-1}, x_i]$	p_i	q_i	$\varepsilon_i = (q_i - p_i)/p_i$	$p_i\varepsilon_i^2$
103	[2.1443, 2.1659]	0.0016945	0.0016955	0.00060306	6.1625e-010
82	[1.7748, 1.7900]	0.0024789	0.0024778	-0.00041684	4.3072e-010
109	[2.2843, 2.3104]	0.0014902	0.0014894	-0.00051947	4.0212e-010
92	[1.9353, 1.9525]	0.0020838	0.0020829	-0.00043891	4.0143e-010
108	[2.2591, 2.2843]	0.0015242	0.001525	0.00047076	3.3779e-010
16	[0.7981, 0.8189]	0.0119470	0.011945	-0.00015320	2.8041e-010
104	[2.1659, 2.1882]	0.0016603	0.001661	0.00040957	2.7852e-010
112	[2.3659, 2.3954]	0.001387	0.0013864	-0.00043609	2.6378e-010

Table 2: The eight intervals with the largest relative discrepancies from the normal distribution, measured as $p_i\varepsilon_i^2$.
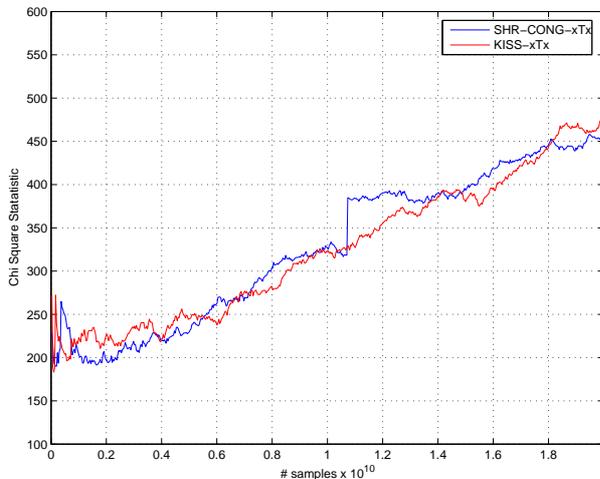

Figure 1: χ^2 values vs. number of samples for generators with output x+Tx.

However, the Ziggurat method with this underlying PRNG also fails the χ^2 test. The reason is that even though requirement (1) is satisfied, the output of SHR0 fails requirement (2), and this leads to non-negligible deviations in the tail probabilities (when $|x| > x_{127}$). As seen from the code RNOR and the function nfix(), a number from the tail is produced only when the seven least significant bits of jsr are all zero and in addition, the resulting number satisfies the condition fabs(hz)< kn[0]. By enumeration over all $2^{25} - 1$ possible values of jsr with 7 least significant bits all zero, only 2,444,151 values pass this test. For each of these numbers we calculated the resulting normal number and produced a histogram according to the following eight intervals defined by $X^t = \{x_{127}, 3.75, 4.0, 4.25, 4.5, 4.75, 5.0, 5.5\}$, where $x_{127} = 3.44262$. In table 3 we show the resulting conditional probabilities $\Pr\{X_j^t < |x| < X_{j+1}^t \,\big|\, |x| > x_{127}\}$ as computed from the Ziggurat method with SHR0, vs. the correct probabilities from the normal distribution. Applying (4) with $k = 8$ we obtain that these deviations can be detected after observation of $O(2^{21})$ samples with $|x| > x_{127}$ or roughly $N = O(2^{31.5})$ outputs overall.

8

Interval	$\Pr\{x_j^t < |x| < x_{j+1}^t\}$	$\Pr\{x_j^t < |Z| < x_{j+1}^t\}$
1	6.9298e-1	6.9305e-1
2	1.9705e-1	1.9700e-1
3	7.2872e-2	7.2843e-2
4	2.5334e-2	2.5311e-2
5	8.2683e-3	8.2644e-3
6	2.5105e-3	2.5357e-3
7	9.3857e-4	9.2921e-4
8	5.4825e-5	6.5924e-5

Table 3: Tail probabilities from the Ziggurat method vs. the correct probabilities from the normal distribution.

4 Design Flaws in the MWC generator of [3]

In contrast to the Ziggurat method, which requires pre-computation and storage of tables of size k, the Monty Python method [3] can produce random variables with normal and other common distributions without the aid of auxiliary tables. In [3], the authors presented the method and suggested the following multiply-with-carry (MWC) generator as a source of uniformly distributed numbers in the set Ω_{32},

```
unsigned long jsr_z, jsr_w;

#define ZNEW (jsr_z=36969*(jsr_z&65535)+(jsr_z>>16))
#define WNEW (jsr_w=18000*(jsr_w&65535)+(jsr_w>>16))
#define MWC  ((ZNEW<<16) + (WNEW&65535))
```

In this specific suggestion, each call to `MWC` outputs a 32-bit integer, based on two independent 32-bit registers `jsr_z` and `jsr_w` . Some of the properties of the transition function of these registers, which is of the form

$$(c, w) = ((c + aw)\mathrm{div}\, b, (c + aw)\mathrm{mod}\, b) \tag{6}$$

where c is the carry and w is the residual upon division by b have been analyzed by Marsaglia [12], Couture and L'Ecuyer [13]. For generalizations to recursions with a similar form, see also Goresky and Klapper[14]. The main properties depend on the number $m = ab - 1$. As proven in [12], when m is a safe prime (both m and $(m-1)/2$ are primes), the period of any starting value (c, w), other than the trivial fixed points $(0, 0)$ and $(a - 1, b - 1)$ is $(m - 1)/2$. Since there are $m + 1 = ab$ possible values for (c, w), it follows that there are two disjoint orbits with this period. For the specific values of `MWC` above, the period of each non-trivial orbit of `jsr_w` is 589823999, while that of `jsr_z` is 1211400191. Since both periods are prime numbers the combined period of `MWC` is their product, or roughly $2^{59.3}$.

Despite the long period, the main design flaw in `MWC` is the *non-uniformity* of its output. Consider for example the 16 most significant bits of `MWC`, e.g., the lowest 16 bits of the

7 MSB's i	$\Pr\{(\texttt{z\_new} >> 9) = i\}$	ε_i
22	0.007818141	7.2200e-3
107	0.007806859	-7.2199e-3
50	0.007817052	5.8263e-3
79	0.007807948	-5.8262e-3
11	0.007816705	5.3825e-3
118	0.007808295	-5.3825e-3
115	0.007816347	4.9250e-3
14	0.007808652	-4.9249e-3

Table 4: Values of 7 MSB's with largest deviations from uniformity for one orbit of $\texttt{jsr\_z}$.

register $\texttt{jsr\_z}$. When we consider all possible initializations (c, w), these 16 bits are obviously randomly distributed. However, *within each disjoint orbit*, these 16 bits are not uniformly distributed. To illustrate this, we computed the exact probabilities for the seven most significant bits in a given orbit. In a uniform distribution, the probability to obtain each one of the 128 possible outcomes is $p_i = 1/128 = 0.0078125$. However, as shown in table 4, some outcomes have non-negligible deviations from this value. Similarly, the 16 output bits of $\texttt{jsr\_w}$ are also not uniformly distributed within each orbit. Since the union of the two disjoint orbits covers almost all possible ab values for (c, w), if deviations in one orbit are of the form $q_i = p_i(1 + \varepsilon_i)$, then in the other orbit the corresponding deviations are approximately $q_i' = p_i(1 - \varepsilon_i)$. Therefore, a union of output sequences belonging to disjoint orbits is approximately uniformly distributed. This might explain why this generator was reported to pass the DIEHARD tests of Marsaglia. However, within each orbit, this deviation from uniformity is easily detected with simple statistical tests. Using formula (4), we obtain that after 2^{28} outputs, it can be detected by a χ^2 test with 128 bins. We remark that on modern computers, 2^{28} outputs are typically generated in less than 10 seconds.

To conclude, while the period of $\texttt{MWC}$ is larger than 2^{59}, its 32-bit output fails the basic requirement (1). These non-negligible deviations from the uniform distribution also lead to non-negligible deviations from the normal distribution when applying the Monti Python method. Without discussing the exact details, these can be detected by a χ^2 test on 16 bins after 2^{30} outputs, and with even fewer outputs if the number of bins is increased.

We note that the $\texttt{MWC}$ generator is also not suitable for use with the Ziggurat method. The reason is that within each orbit, the 7 lsb's of $\texttt{jsr\_w}$ are not uniformly distributed. Therefore this PRNG would not choose each of the 128 intervals at the required uniform probability of 1/128. Indeed numerical experiments show that the χ^2 statistics on outputs of the Ziggurat method based on this PRNG start to significantly deviate from the expected value after about 2^{32} outputs.

Finally we note that $\texttt{MWC}$ is one of the standard uniform random number generators in the statistical software $\texttt{R}$. Given the above non-uniformity of this generator, we caution against its use in simulations.

5 A statistical analysis of Matlab's randn

5.1 The underlying uniform PRNG of randn

The Matlab software has a built-in function `randn` to produce normally distributed random numbers, which is also based on the Ziggurat method. A matlab code compatible with the pre-compiled built-in function appears in [8]. In contrast to [1], Matlab's `randn` is based on a combination of two different 32-bit registers, `jsr` and `icng`. Here is a pseudo-C code corresponding to matlab's `randn`:

```
unsigned long jsr,icng;
long hz,iz;

#define SHR0 (jsr^=jsr<<13, jsr^= jsr>>17, jsr^=jsr<<5, jsr)
#define CNG  (icng = 69069*icng+1234567)
#define RNOR (hz=CNG+SHR0,iz=hz&63,(fabs(hz)<kn[iz])?hz*wn[iz]:matlab_nfix())
```

The first register `jsr` is updated by `SHR0`, as a linear shift register with maximal period of $2^{32} - 1$. The second register `icng` is updated as a multiplicative congruential generator, with maximal period 2^{32}. The output which serves as a uniform random number is their sum (`jsr + icng`) mod 2^{32}. We denote the transition function of `jsr` by `T`, and that of `icng` by `R`. We also denote its multiplicative part by R_0, e.g. `R`$_0$(`x`)=69069*`x` mod 2^{32}. Since the periods of the two registers are relatively prime, the combined period of `randn` is their product, a number close to 2^{64}. Matlab uses a table of size 64, and since it is based on the original Ziggurat publication [6], both the points x_i, the tables `kn,wn` and the function `matlab_nfix()` are different from the ones described in section 3.

Since `icng` is uniformly distributed over Ω_{32}, individual outputs `jsr+icng` are also uniform over Ω_{32} and satisfy requirement (1). However, pairs of consecutive outputs are highly correlated, and fail requirement (2). Let y_1 and y_2 denote two consecutive outputs of this uniform random number generator. Let a, b denote the unknown initial states at time 1 of the two registers `jsr` and `icng`, that is $(a + b) = y_1$. After a single update of the two registers, the next output is given by $y_2 = T(a) + R(b)$. However, since $b = y_1 - a$, we have that $y_2 = T(a) - R_0(a) + R(y_1)$. Similar to the analysis of section 3, the transformation $T(a) - R_0(a)$ is highly contractive and not one-to-one. Therefore, the pair of outputs (y_1, y_2) is not uniformly distributed over $\Omega_{32} \times \Omega_{32}$ and thus fails to satisfy the requirement (2). Table 5 shows the distribution of $T(a) - R_0(a)$. As shown in the table, some $2^{30.5}$ values are not possible, while other values are 10 times more probable than expected in a uniform distribution. Note that $(y_2 - R(y_1))$ mod 2^{32} is therefore highly non-smooth, and would not pass a χ^2 test for uniformity.

We now consider the implications of these findings on the resulting normal numbers as computed by `randn`. Consider, for example, the rejection probabilities at step 3. Since y depends on x, the rejection probabilities deviate slightly from the correct ones. However, when computing these rejection probabilities over large enough intervals, these x-dependent deviations almost cancel out (they are positive for some x and negative for others). Similarly,

11

# sources	# of outputs
0	1590591029
1	1569484236
2	784774346
3	265026908
4	68022535
5	14147755
6	2484729
7	377496
8	51341
9	6136
10	713
11	65
12	6
13	1

Table 5: Distribution of the number of sources of $Tx - R_0 x$.

tail probabilities at individual x-values also deviate from their correct values, but when averaged over large enough intervals these deviations cancel out. Therefore, even though the underlying generator is not uniform in pairs, its effects on the resulting normal random numbers is difficult to detect by standard tests.

5.2   Initialization Issues

We now consider the initialization of `randn` and its possible consequences. Matlab provides two different initialization options,

$$\mathtt{randn}('\mathtt{state}', a); \qquad \text{OR} \qquad \mathtt{randn}('\mathtt{state}', [a \; b]');$$

The first sets the initial value of `jsr` to `a`, with the initial value of `icng` set to a fixed value 362436069. The second option allows to set also the initial value of `icng` to `b`.

In many applications, such as parallel computations and stochastic simulations, there is a need to create many independent sequences of normal random numbers. In the case of parallel computer systems, it is quite common to initialize the seed of processor number `id` with a seed of the form $\mathtt{seed}_0 + \mathtt{id}$. Various papers describe the dangers and possible pitfalls in using sequences of random numbers produced by different initializations of the same generator (see [10, 15, 17] and references therein). We now present a simple example of such a pitfall for matlab's `randn`.

Suppose we wish to simulate 2^{16} different paths of a stochastic system that requires normal random numbers on a parallel computer with 256 processors. A possible code can be for example

```
for i=1:256
    for j=1:256
        send to processor i the following:
                    randn('state',[i j]');
                    simulate_random_path();
    end
end
```

This code ensures that each simulation thread obtains a different seed. However, consider the output sequences resulting from two initializations [i j] and [i j+64]. These two initializations have the same initial value for jsr and differ only by the initialization of the register icng. Due to the structure of the transition function R of this register, it follows that for all subsequent times, both of these sequences will have the same six low significant bits. Therefore, neglecting the possible misses in the Ziggurat method, which require a call to matlab_nfix(), both sequences will choose the same indices (!), and the resulting normal numbers will be highly correlated.

These correlations between output sequences initialized with different but related seeds are due to the failure of randn to satisfy requirement (3). However, we might be tempted to conclude that if instead we perform initializations with random unrelated seeds the resulting sequences will be uncorrelated. However, even in such cases there can be non-trivial correlations between the first few values of different output sequences. Consider the output sequences from n initializations of the form randn('state',v[i]), where $\{v_i\}_{i=1}^n$ is a set of n random non-zero 32 bit integers. Suppose that there are four distinct indices, i, j, k, l, such that $v_i \oplus v_j = v_k \oplus v_l = \alpha$, where $\oplus$ denotes bitwise exclusive or. Then, for the first few outputs, the resulting 4-tuples of random numbers are not independent. To see this, denote x^i, x^j, x^k, x^l the resulting first output of the uniform random number generator initialized with v_i, v_j, v_k, v_l, respectively. Since all these initializations have the same initial value for the register icng, the first output is given by

$$x^i = T(v_i) + R(icng) \quad x^j = T(v_j) + R(icng) = T(v_i) \oplus T(\alpha) + R(icng)$$

with similar expressions for x^k and x^l. As an example, assume that the most significant bit of $T(\alpha)$ is zero. Then, with high probability the most significant bit of x^i and x^j (and of x^k and x^l) will be the same. Therefore, the resulting normal numbers will have the same sign. For the second output the sign will be determined by the most significant bit of $T^2(\alpha)$, etc. (assuming that no intermediate calls to matlab_nfix() occurred). Another example of dependency occurs if we consider the 7 least significant bits of $T(\alpha)$. For simplicity, if these are all zeros, then the numbers x^i and x^j (and x^k and x^l) point to the same index for the Ziggurat method. This again leads to correlations between the 4-tuples of normal outputs. Needless to say, such correlations between different streams may bias a stochastic simulation in unexpected ways. Given n initializations, the average number of such 4-tuples is of the order of $\binom{n}{4}2^{-32}$. Therefore, after only $n = O(512)$ random sequences there will be on average one such 4-tuple.

6 Summary and Discussion

In this paper we presented various statistical weaknesses in the published implementation of the Ziggurat and Monty Python methods and in the underlying uniform PRNG of matlab's built-in function `randn`. As also noted in other works, the main take home lessons from our analysis are: i) The set $\mathcal{S}$ of internal states of the generator should be much larger than the output set Ω. This is necessary to satisfy requirement (2) and its generalization to independence of d-tupples of outputs with $d > 2$. ii) Correlations between consecutive outputs of a uniform RNG can have detrimental effects on the results of a stochastic simulation. iii) The creation of many different sequences of random numbers via initializations of the same generator with different seeds must be done with great care.

Regarding the initialization of random number generators, we note that most implementations use $P = I$ or some other relatively simple scheme, in which the seed is typically entered into the inner state in a linear fashion. However, since initialization is done only rarely it is possible to spend many more CPU cycles on this stage, and make the inner state be dependent on the initial seed in a much more complicated and non-linear manner.

We remark that there is an interesting connection between our analysis of PRNG's and cryptanalysis of stream and block cyphers. For example, our analysis of the statistical correlations of PRNG's initialized with different but related seeds is similar to the 'related key attacks' introduced by Biham in [18], and used to crack the WEP wireless encryption protocol [19]. This serves as yet another justification for making the initialization stage (the function P in the notation of section 2) a complicated non-linear function.

There is yet another connection between PRNG's and cyphers. In cryptography, the designer would like the security of a cypher system not be dependent on its specific initialization by the end user (e.g., with say counters or initial values (IV's) increasing by one). We submit that a similar requirement should hold in the design of a PRNG. The output of a PRNG (and correlations between different output runs) should not be highly dependent on simple and natural initializations by the user, who typically does not know nor wishes to fully understand the inner workings of the random number generator at his disposal.

Appendix

Let X and Y be two discrete random variable over k possible values, with probability distributions $(p_1, \ldots, p_k)$ and $(q_1, \ldots, q_k)$, respectively. Our aim is to estimate the number of outputs needed from i.i.d. realizations of Y to check the hypothesis if Y has the same probability distribution as X, using the χ^2 test with k bins. Let $(y_1, \ldots, y_N)$ be N random samples from the distribution of Y, and let $(z_1, \ldots, z_k)$ denote the number of occurrences of the values $(1, \ldots, k)$ in the sequence $\{y_i\}_{i=1}^N$. Then the χ^2 statistic is given by

$$T = \sum_{i=1}^{k} \frac{(z_i - Np_i)^2}{Np_i} = \sum_{i=1}^{k} \frac{z_i^2}{Np_i} - N$$

Its mean (expected) value is

$$\mathbb{E}T = \sum_{i=1}^{k} \frac{\mathbb{E}z_i^2}{Np_i} - N \tag{7}$$

If $Y \sim (q_1, \ldots, q_k)$ then each z_i follows a Binomial distribution $Bin(N, q_i)$. Therefore,

$$\mathbb{E}T = \sum_{i=1}^{k} \frac{N^2 q_i^2 + N q_i (1 - q_i)}{Np_i} - N \tag{8}$$

Writing $q_i = p_i(1 + \varepsilon_i)$ gives

$$\mathbb{E}T = (k - 1) + (N - 1) \sum_{i=1}^{k} p_i \varepsilon_i^2 + \sum_{i=1}^{k} \varepsilon_i \tag{9}$$

Since a χ^2 distribution with k degrees of freedom has a variance of $2k$, it follows that to distinguish between the distribution of X and Y, the χ^2 statistic must significantly deviate from $k - 1 + \sqrt{2k}$. Thus, we require that

$$N = O\left(\frac{\sqrt{2k}}{\sum_i p_i \varepsilon_i^2} \right) \tag{10}$$

References

[1] G. Marsaglia and W.W. Tsang, The Ziggurat method for generating random variables, *J. Stat. Soft.*, 5:1-7 (2000).

[2] G. Marsaglia and W.W. Tsang, A simple method for generating gamma variables, *ACM Trans. Math. Soft.* 26(3):363-72 (2000).

[3] G. Marsaglia and W.W. Tsang, The Monty Python method for generating random variables, *ACM Trans. Math. Soft.* 24(3):341-350 (1998).

[4] D. Knuth, *The art of computer programming, vol 2: seminumerical algorithms*, 3rd edition, Addison-Wesley, Reading, MA, 1998.

[5] L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

[6] G. Marsaglia and W.W. Tsang, A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions, *SIAM J. Sci. Stat. Comput.* 5(2):349-359 (1984).

[7] PHW Leong et. al., A comment on the implementation of the Ziggurat method,*J. Stat. Soft.*, 12(7), (2005).

[8] http://www.mathworks.com/moler/ncm/randntx.m

[9] P. L'Ecuyer, Uniform random number generation, *Ann. Oper. Res.*, 53:77–120 (1994).

[10] R.P. Brent Fast and reliable random number generators for scientific computing, Lecture Notes in Computer Science, Vol. 3732, Springer (2005), 1-10, to appear.

[11] J.A. Doornik, An improved Ziggurat method to generate normal random samples, *working paper.*

[12] G. Marsaglia, The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness, `http://stat.fsu.edu/pub/diehard`.

[13] R. Couture and P. L'Ecuyer, Distribution Properties of Multiply-with-Carry Random Number Generators, *Math. of Comp.*, 66:591–607 (1997).

[14] A. Klapper and M. Goresky, Feedback shift registers, 2-adic span, and combiners with memory, *J. Crypt*, 10:111-147 (1997).

[15] P. Hellekalek, Don't trust parallel Monte Carlo, Proceedings of the twelfth workshop on Parallel and distributed simulation pp:82-89, IEEE, 1998.

[16] G. Marsaglia, Xorshift RNG's, *J. Stat. Soft.*, 8(14):1-6, 2003.

[17] A. De Matteis, S. Pagnutti, Parallelization of random number generators and long-range correlations, Numerische Mathematik, 53(5):595 - 608 (1988).

[18] E. Biham, New type of cryptanalytic attacks using related key, EUROCRYPT 93', Lecture notes in computer science, vol 765:229-246, (1994).

[19] SR. Fluhrer, I. Mantin, Adi Shamir, Weaknesses in the Key Scheduling Algorithm of RC4", Selected Areas in Cryptography, pp124, 2001.