The Philippine Statistician, 2004
Vol. 53, Nos. 1-4, pp. 67-76

A Binary Sequence Generator Based on the Thue-Morse Sequence and Multiples of Primes

Joel Reyes Noche*

Received May, 2004; Revised August, 2004

ABSTRACT

An apparently novel binary sequence generator that uses reversible operations and does not use a seed is shown to perform well in the DIEHARD tests, a popular set of tests for randomness of uniformly distributed numbers. The generator inverts every p bits of the Thue-Morse sequence, where p is a prime number, and does this for the first f prime numbers. Samples from the resulting binary sequence can be used in statistical applications requiring random numbers.

Key words: Thue-Morse sequence, randomness, uniform distribution, prime numbers, DIEHARD

1. INTRODUCTION

Large quantities of "random" numbers are currently in demand (Hayes, 2001) for applications in information security and scientific computation, among others. The latter includes statistical random sampling such as those used in the bootstrap method (Diaconis and Efron, 1983) and in the Monte Carlo method (Eckhardt, 1987).

In this paper, a modest definition of randomness is used: the randomness of a binary sequence is described by its performance in commonly accepted statistical tests.

Pseudo-random number generators (PRNGs) usually refer to generators of values having a uniform distribution over the interval (0,1). (Arbitrary distributions are usually generated by applying transformations to these values.) PRNGs differ from (true) random number generators (RNG) in that the former perform deterministic operations on a seed value, while the latter rely on a non-deterministic source of randomness (usually a physical process such as radioactive decay or thermal noise).

Surveys of PRNGs (L'Ecuyer (1994), L'Ecuyer (2004)) classify PRNGs into linear generators and nonlinear generators. Linear generators include multiple recursive (which include linear congruential and lagged-Fibonacci), multiply-with-carry (which include add-with-carry and subtract-with-borrow), linear feedback shift register (LFSR), generalized feedback shift register (GFSR), twisted GFSR, and combinations of these. Examples of nonlinear generators mentioned are multiplicative-inversive, quadratic, and cubic. The binary sequence generator presented here does not seem to fall into any of these classifications.

* Assistant Professor, Department of Electrical and Electronics Engineering, College of Engineering, University of the Philippines, Diliman, 1101, Quezon City, Philippines.
E-mail: joel.noche@ieee.org.ph

The binary sequence generator presented here differs from the usual PRNG in two ways: it outputs a large binary sequence at one time (instead of generating its output a few bits at a time), and it does not require a seed value. Of course, the output values of PRNGs can be concatenated to form a large sequence, and there already exist binary sequence generators that can work with practically no seed (see, for example, Wolfram (2002)).

In some PRNGs, such as those that use the modulo operation, the operations may not be reversible, that is, information may be lost. The binary sequence generator presented here uses reversible operations; given the values at any step in the generation, any past values can be calculated. Even so, the binary sequence it generates performs well in a popular set of randomness tests.

2. MATERIALS AND METHODS

The Thue-Morse binary sequence (also known as the Prouhet-Thue-Morse sequence) $\{x(n)\}_{n=0}^{\infty}$ = 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, ... can be defined or generated in many ways (see, for example, Allouche and Shallit (1999) and Wolfram (2002, pp. 83, 890, 892, 895, 944, 1073, 1092, and 1186)). One algorithm to generate its first 2^m bits is shown in Figure 1 (where ¬ denotes the logical NOT operation).

Figure 1. An Algorithm to Generate the First 2^m Bits of the Thue-Morse Sequence

```
x(0) ← 0
for n = 0 to m − 1
    for i = 0 to 2^n − 1
        x(2^n + i) ← ¬x(i)
```

A sequence $\{a(n)\}$ is said to be *uniformly recurrent* if for each finite block of symbols w occurring in $\{a(n)\}$ there exists an integer j such that for all i, the sub-block $a(i+1)$, ..., $a(i+j)$ contains an occurrence of w. A sequence $\{a(n)\}$ is said to be *ultimately periodic* if there exists integers $k \geq 1$, $N \geq 0$ such that $a(i) = a(i+k)$ for all $i \geq N$. Among the many properties of the Thue-Morse sequence, of main interest here is the fact that it is uniformly recurrent but not ultimately periodic (Allouche and Shallit, 1999). Practically speaking, the regularity of the sequence makes it easy to generate and its non-periodicity invites investigation on its usefulness in PRNGs.

Note that bits whose order in the sequence is a multiple of p, where p is an odd prime number, are not functions of each other when using the algorithm in Figure 1. That is, to generate $x(ap - 1) = x(2^n + i) = \neg x(i)$, where a is a natural number, the values of i that are used are never equal to $ap - 1$. Equivalently, when generating $x(ap) = x(2^n + i + 1) = \neg x(i + 1)$, the values of $i + 1$ used are never multiples of p. This can be seen as follows: if $i + 1$ was a multiple of p, then 2^n should also be a multiple of p. But since 2^n is never a multiple of p, then $i + 1$ is not a multiple of p.

This leads to an interesting method of adding "disorder" to the original sequence: invert the bits whose order in the sequence is a multiple of p, and do this sequentially for

increasing p, for the first f odd prime numbers. (Odd prime numbers are chosen because no "disorder" is introduced by inverting all the even bits, that is, the multiples of the only even prime number 2.)

3. RESULTS

One popular way to test PRNGs and RNGs is the DIEHARD series of tests (Marsaglia, 1995). For example, Intel (1999) has used it to test its hardware RNG. DIEHARD contains fifteen tests for randomness of uniformly distributed numbers. Care should be taken when running these tests. For example, Marsaglia reports that the hardware RNGs he tested failed "spectacularly." Davies (1997) found that the supposed failures were mostly due to a data handling error: each byte with a value of 10 was being recorded as two bytes with values of 13 and 10 (a carriage return followed by a line feed). Another possible problem is the different conventions for the order in which the bits or bytes are stored in a word. The rearrangement of bytes when copying between computers with different conventions can affect the results of correlations tests and other tests (Davies, 2000). In this work, both problems are avoided: the output is saved as a binary file (not a text file), and no conversions between different operating systems or computer systems are done.

Details of the tests and the associated p-values are in Marsaglia (1995). A short list of the fifteen tests as well as their associated p-values is shown in Table 1.

Table 1. List of DIEHARD Tests and Associated p-values

DIEHARD Tests	p-values
1. Birthday Spacings	p_1
2. Overlapping Permutations	p_2, p_3
3. Ranks of 31×31 and 32×32 matrices	p_4, p_5
4. Ranks of 6×8 Matrices	p_6
5. Monkey Tests on 20-bit Words	$p_7 - p_{26}$
6. Monkey Tests OPSO, OQSO, DNA	$p_{27} - p_{49}, p_{50} - p_{77}, p_{78} - p_{108}$
7. Count the 1's in a Stream of Bytes	p_{109}, p_{110}
8. Count the 1's in Specific Bytes	$p_{111} - p_{135}$
9. Parking Lot Test	p_{136}
10. Minimum Distance Test	p_{137}
11. Random Spheres Test	p_{138}
12. The Squeeze Test	p_{139}
13. Overlapping Sums Test	p_{140}
14. Runs Test	$p_{141} - p_{144}$
15. The Craps Test	p_{145}, p_{146}

Five test cases are presented here, each of them 16,777,216 bytes long. (DIEHARD needs at least an 80 million bit long test sequence for all of its tests to run.) Their results in the DIEHARD tests are shown in Tables 2, 3, and 4. In the first case, z2000, the initial sequence is composed of all zero bits, and the first 2000 prime numbers are used. In the next five cases, the initial sequence is the Thue-Morse sequence, and the first 10, 100, 1000, and 2000 prime numbers, respectively, are used.

For comparison, the results of the PRNG KISS (Marsaglia, 1995) are also included. KISS combines three sequences: a linear-congruential sequence, a shift register sequence,

and a multiply-with-carry sequence. It "seems to pass all [the DIEHARD] tests and is highly recommended for speed and simplicity" (Marsaglia, 1995). The KISS sequence tested here is 11,468,800 bytes long and used the seed integers 1, 4, 3, and 44.

Table 2. Results of DIEHARD Tests (part 1 of 3)

	z2000	t10	t100	t1000	t2000	KISS
p_1	0.3710	0.7122	0.0487	0.3038	0.7763	0.2681
p_2	0.9985	0.9989	0.6564	0.8490	0.0446	0.8745
p_3	1.0000	0.8911	0.7935	0.1323	0.0283	0.1197
p_4	0.3711	0.7855	0.5869	0.8244	0.3859	0.6224
p_5	0.4729	0.3859	0.3630	0.7471	0.3257	0.4290
p_6	1.0000	1.0000	0.3395	0.5196	0.6396	0.6257
p_7	1.0000	0.9696	0.3001	0.9557	1.0000	0.1760
p_8	1.0000	0.9995	0.9288	0.2285	0.9989	0.8175
p_9	1.0000	0.9589	0.2257	0.8290	0.8565	0.5601
p_{10}	1.0000	0.9262	0.6576	0.9061	0.7703	0.2495
p_{11}	1.0000	0.9514	0.4302	0.2292	0.3157	0.3224
p_{12}	1.0000	0.9978	0.9658	0.7856	0.8688	0.2236
p_{13}	1.0000	0.9998	0.1922	0.3428	0.5006	0.6226
p_{14}	1.0000	1.0000	0.3742	0.6011	0.6190	0.2059
p_{15}	1.0000	0.7836	0.7603	0.9723	0.9963	0.6182
p_{16}	1.0000	0.9985	0.3334	0.9943	0.6550	0.3116
p_{17}	1.0000	0.9997	0.8997	0.5360	0.4486	0.8016
p_{18}	1.0000	0.9997	0.9673	0.9815	0.3149	0.7877
p_{19}	1.0000	0.7977	0.9624	0.8863	0.3116	0.3549
p_{20}	1.0000	1.0000	0.1865	0.8799	0.9994	0.8877
p_{21}	1.0000	0.9977	0.5100	0.3419	0.4569	0.3522
p_{22}	1.0000	0.1840	0.3436	0.2600	0.1090	0.9208
p_{23}	1.0000	1.0000	0.8156	0.0406	0.4662	0.8284
p_{24}	1.0000	0.9886	0.2342	0.3514	0.1884	0.9021
p_{25}	1.0000	0.9425	0.8087	0.5793	0.2215	0.0736
p_{26}	1.0000	0.8785	0.9942	0.8804	0.8377	0.5674
p_{27}	1.0000	1.0000	1.0000	0.9979	0.9942	0.4257
p_{28}	1.0000	1.0000	0.9935	0.9838	0.8444	0.7102
p_{29}	1.0000	1.0000	0.9772	0.9944	0.9827	0.4844
p_{30}	1.0000	1.0000	0.8648	0.7055	0.8611	0.5599
p_{31}	1.0000	1.0000	0.9766	0.9800	0.9863	0.8541
p_{32}	1.0000	1.0000	0.9990	0.8564	0.9728	0.7828
p_{33}	1.0000	1.0000	0.7523	0.9997	1.0000	0.5882
p_{34}	1.0000	1.0000	0.9256	0.9285	0.5489	0.6319
p_{35}	1.0000	1.0000	0.9870	0.8772	0.7534	0.8827
p_{36}	1.0000	1.0000	0.9969	0.9747	0.9513	0.2216
p_{37}	1.0000	1.0000	0.9888	0.9725	0.8477	0.5639
p_{38}	1.0000	1.0000	0.9812	0.9651	0.9942	0.0667
p_{39}	1.0000	1.0000	0.9999	0.9640	0.9165	0.2785
p_{40}	1.0000	1.0000	0.9997	0.8265	0.9637	0.5976
p_{41}	1.0000	1.0000	0.9828	0.9922	0.7265	0.2186
p_{42}	1.0000	1.0000	0.9760	0.9979	0.9986	0.6862
p_{43}	1.0000	1.0000	0.9312	1.0000	0.9950	0.0162
p_{44}	1.0000	1.0000	0.9991	0.9967	0.9980	0.1392
p_{45}	1.0000	1.0000	0.9988	0.9957	0.9933	0.2739
p_{46}	1.0000	1.0000	1.0000	1.0000	0.9988	0.0517
p_{47}	1.0000	1.0000	0.9936	0.9294	0.9997	0.9357
p_{48}	1.0000	1.0000	1.0000	0.9908	1.0000	0.3454
p_{49}	1.0000	1.0000	0.9378	1.0000	0.9933	0.0510

Table 3. Results of DIEHARD Tests (part 2 of 3)

	z2000	t10	t100	T1000	t2000	KISS
p_{50}	1.0000	1.0000	0.9447	0.9392	0.6259	0.2761
p_{51}	1.0000	1.0000	0.9999	0.9847	0.0507	0.7603
p_{52}	1.0000	1.0000	0.9833	0.9010	0.8745	0.1684
p_{53}	1.0000	1.0000	0.9945	0.9683	0.5655	0.9598
p_{54}	1.0000	1.0000	0.9772	0.5828	0.9297	0.6612
p_{55}	1.0000	1.0000	0.8188	0.7956	0.3823	0.4874
p_{56}	1.0000	1.0000	0.8293	0.9346	0.9788	0.7035
p_{57}	1.0000	1.0000	0.6525	0.9404	0.4349	0.9512
p_{58}	1.0000	1.0000	0.8600	0.5788	0.5414	0.2086
p_{59}	1.0000	1.0000	0.9439	0.7809	0.3823	0.7093
p_{60}	1.0000	1.0000	0.8961	0.6424	0.2235	0.8841
p_{61}	1.0000	1.0000	0.9998	0.5947	0.6181	0.9502
p_{62}	1.0000	1.0000	0.9819	0.5655	0.9462	0.8460
p_{63}	1.0000	1.0000	0.6374	0.1736	0.5548	0.6012
p_{64}	1.0000	1.0000	0.9443	0.6857	0.5907	0.2605
p_{65}	1.0000	1.0000	0.9894	0.9795	0.6917	0.0890
p_{66}	1.0000	1.0000	0.9990	0.9936	0.8905	0.5548
p_{67}	1.0000	1.0000	1.0000	0.7508	0.8943	0.3269
p_{68}	1.0000	1.0000	0.9586	0.9974	0.5881	0.9439
p_{69}	1.0000	1.0000	0.6246	0.8759	0.8079	0.8088
p_{70}	1.0000	1.0000	0.9039	0.9995	0.9826	0.6550
p_{71}	1.0000	1.0000	0.9883	1.0000	1.0000	0.4617
p_{72}	1.0000	1.0000	0.9574	0.9908	1.0000	0.5629
p_{73}	1.0000	1.0000	0.9539	0.8667	1.0000	0.6563
p_{74}	1.0000	1.0000	0.6869	0.9615	0.4887	0.1576
p_{75}	1.0000	1.0000	0.9341	0.5802	0.9301	0.6893
p_{76}	1.0000	1.0000	0.9737	0.9062	0.9814	0.4283
p_{77}	1.0000	1.0000	0.9994	0.8484	0.6064	0.9117
p_{78}	1.0000	1.0000	0.3481	0.1103	0.9606	0.6663
p_{79}	1.0000	1.0000	0.4690	0.7755	0.8831	0.8180
p_{80}	1.0000	1.0000	0.5940	0.8140	0.2627	0.3848
p_{81}	1.0000	1.0000	0.9032	0.0572	0.4282	0.5102
p_{82}	1.0000	1.0000	0.9752	0.8603	0.9081	0.0555
p_{83}	1.0000	1.0000	0.9863	0.9990	0.5849	0.4398
p_{84}	1.0000	1.0000	0.2050	0.0610	0.2550	0.2598
p_{85}	1.0000	1.0000	0.9933	0.9949	0.6280	0.5986
p_{86}	1.0000	1.0000	0.2239	0.6801	0.8272	0.4761
p_{87}	1.0000	1.0000	0.7386	0.5814	0.5664	0.0724
p_{88}	1.0000	1.0000	0.2401	0.4808	0.8905	0.0138
p_{89}	1.0000	1.0000	0.9942	0.5501	0.1206	0.2812
p_{90}	1.0000	1.0000	0.2109	0.4098	0.9052	0.0437
p_{91}	1.0000	1.0000	0.8796	0.4006	0.8044	0.3713
p_{92}	1.0000	1.0000	0.7666	0.1493	0.9370	0.5952
p_{93}	1.0000	1.0000	0.8877	0.8256	0.9990	0.1171
p_{94}	1.0000	1.0000	0.1125	0.7711	0.0498	0.7358
p_{95}	1.0000	1.0000	0.9880	0.8124	0.2292	0.7538
p_{96}	1.0000	1.0000	0.9134	0.5114	0.3635	0.9644
p_{97}	1.0000	1.0000	0.9129	0.9022	0.9096	0.3949
p_{98}	1.0000	1.0000	0.9637	0.9426	0.1278	0.4029

Table 4. Results of DIEHARD Tests (part 3 of 3)

	z2000	t10	t100	t1000	t2000	KISS
p_{99}	1.0000	1.0000	0.3881	0.5940	0.0980	0.8502
p_{100}	1.0000	1.0000	0.7151	0.5860	0.6937	0.4098
p_{101}	1.0000	1.0000	0.8910	1.0000	0.9991	0.6032
p_{102}	1.0000	1.0000	0.8596	0.9787	0.9809	0.6978
p_{103}	1.0000	1.0000	0.6999	0.8550	0.9976	0.1142
p_{104}	1.0000	1.0000	0.7060	0.9859	0.6313	0.9850
p_{105}	1.0000	1.0000	0.5396	0.7593	0.7191	0.0582
p_{106}	1.0000	1.0000	0.6446	0.7231	0.5466	0.4784
p_{107}	1.0000	1.0000	0.7657	0.8590	0.5998	0.1459
p_{108}	1.0000	1.0000	0.9370	0.3341	0.9366	0.7764
p_{109}	1.0000	1.0000	0.9788	0.9626	0.9972	0.9189
p_{110}	1.0000	1.0000	1.0000	1.0000	0.8349	0.3365
p_{111}	1.0000	1.0000	0.0057	0.3466	0.5955	0.1668
p_{112}	1.0000	1.0000	0.0456	0.8933	0.5467	0.2732
p_{113}	1.0000	0.9999	0.9848	0.9142	0.3247	0.8841
p_{114}	1.0000	1.0000	0.9857	0.4724	0.3456	0.5963
p_{115}	1.0000	1.0000	0.2575	0.8042	0.5010	0.2929
p_{116}	1.0000	1.0000	0.9572	0.7441	0.9975	0.0918
p_{117}	1.0000	1.0000	0.9428	0.5476	0.6420	0.7820
p_{118}	1.0000	1.0000	0.8531	0.8603	0.4037	0.3705
p_{119}	1.0000	1.0000	0.4538	0.1093	0.9995	0.0389
p_{120}	1.0000	0.8673	0.6045	0.8819	0.8266	0.1373
p_{121}	1.0000	0.9562	0.7910	0.4054	0.1517	0.9848
p_{122}	1.0000	0.7658	0.0721	0.6072	0.9603	0.9921
p_{123}	1.0000	0.9790	0.1441	0.7787	0.9770	0.6217
p_{124}	1.0000	0.9133	0.2970	0.5766	0.8444	0.2718
p_{125}	1.0000	0.9946	0.8697	0.5250	0.5326	0.6614
p_{126}	1.0000	0.8347	0.3911	0.6134	0.5468	0.8014
p_{127}	1.0000	1.0000	0.9551	0.2841	0.9996	0.8939
p_{128}	1.0000	1.0000	0.6116	0.6001	0.9288	0.9398
p_{129}	1.0000	1.0000	0.4390	0.7111	0.3382	0.2047
p_{130}	1.0000	1.0000	0.4960	0.9820	0.8006	0.7576
p_{131}	1.0000	1.0000	0.7989	0.8115	0.9100	0.2818
p_{132}	1.0000	1.0000	0.0715	0.6744	0.8113	0.5977
p_{133}	1.0000	1.0000	0.0190	0.0914	0.1225	0.9842
p_{134}	1.0000	1.0000	0.2324	0.6989	0.2342	0.8054
p_{135}	1.0000	1.0000	0.1598	0.5256	0.9767	0.8171
p_{136}	1.0000	0.9651	0.6269	0.9902	0.9989	0.8107
p_{137}	0.6670	0.9949	0.4967	0.8966	0.2325	0.8361
p_{138}	0.3425	0.4647	0.4085	0.7463	0.4653	0.1521
p_{139}	1.0000	1.0000	1.0000	0.4420	0.9990	0.7997
p_{140}	0.9983	0.9656	0.7400	0.1966	0.7124	0.5315
p_{141}	0.5112	0.1506	0.6646	0.6512	0.8336	0.8084
p_{142}	0.9959	0.5854	0.5057	0.6346	0.7836	0.5250
p_{143}	0.7896	0.2180	0.2605	0.3904	0.1105	0.9071
p_{144}	0.8130	0.6413	0.8517	0.1351	0.6417	0.9782
p_{145}	0.9603	1.0000	0.6383	0.8354	0.5905	0.5957
p_{146}	0.0124	1.0000	0.9842	0.5108	0.3202	0.5659

On a computer with a Pentium IV processor and a Windows XP operating system, the time to generate the output file (using the source code in Appendix A) ranges from around 14 seconds (for case t10) to around 28 seconds (for case t2000). The KISS output file takes less than a second to generate after the seed values are given.

4. DISCUSSION

It is implied that if all the p-values listed in Table 1 of a sequence tested by DIEHARD lie in the range (0.025, 0.975) then the sequence is said to have passed all the DIEHARD tests. But Marsaglia (1995) notes that even "good" PRNGs (like KISS) have a few p-values that lie outside this range. (For example, $p_{88} = 0.0138$ and $p_{122} = 0.992144$ for the KISS sequence tested here.)

For the method presented here, cases where the initial sequence is composed of all zero bits perform extremely poorly in the tests, even when the number of primes f used is quite large ($f = 2000$ for the case z2000).

Using the initial binary sequence presented here improves the test scores. The case t10 performs noticeably better than the case z2000 in tests 2, 5, 8, and 9, but it still completely fails tests 4, 6, 7, and 12, and performs worse for tests 10 and 15. Increasing f to 100 takes care of tests 4, 5, 6, 8, 10, 15, and part of 7, but it still completely fails test 12 and the other part of 7. With $f = 1000$, test 12 is passed, and with $f = 2000$, performance in test 7 improves. Note that performance in test 6 does not seem to improve much from $f = 100$ to $f = 2000$, and that KISS performs better in this test than the method presented here.

Considering the improvement in test results from case z2000 to the other cases, one might think that the (unmodified) Thue-Morse sequence would also yield good test results. This is not the case. Testing just the Thue-Morse sequence yields p-values that are either 1.0000 or 0.0000. In fact, test 15 for this case does not even finish due to an overflow error.

CONCLUSIONS

The binary sequence generator presented in this paper is shown to perform well in a popular set of tests for randomness. It may be used in statistical applications that require very large amounts of random numbers. As the sequence it generates is perfectly predictable, it would not be ideal for security applications such as cryptography.

Some PRNGs have been found to yield erroneous results when used in certain Monte Carlo simulations (for example, subtract-with-borrow (Peterson, 2004), LFSR (Bauke and Mertens, 2004) and GFSR (Schmid and Wilding, 1995)). Further study on the Thue-Morse sequence or other similar sequences may lead to more effective pseudo-random binary sequence generators.

ACKNOWLEDGMENT

This work was supported by a teaching and research grant from the UP EEE Foundation, Inc. Thanks to the anonymous referee for the helpful suggestions.

References

ALLOUCHE, J.-P. and SHALLIT, J. (1999). "The ubiquitous Prouhet-Thue-Morse sequence," in *Sequences and Their Applications: Proceedings of SETA '98*, Springer-Verlag, 1–16.

BAUKE, H. and MERTENS, S. (2004). "Pseudo random coins show more heads than tails," *Journal of Statistical Physics* 114: 1149–1169.

DAVIES, R. (1997). True random number generators, http://www.robertnz.net/true_mg.html (Accessed April 2004).

DAVIES, R. (2000). Hardware random number generators, http://www.robertnz.net/hwrng.html (Accessed April 2004).

DIACONIS, P. and EFRON B. (1983). "Computer-intensive methods in statistics," *Scientific American* 248(5): 96–108.

ECKHARDT, R. (1987). "Stanislaw Ulam, John Von Neumann, and the Monte Carlo method," *Los Alamos Science* special issue: 131–137.

HAYES, B. (2001). Computing Science: Randomness as a Resource, *American Scientist* 89(4): 300–304.

INTEL (1999). *The Intel Random Number Generator*, http://www.intel.com/design/chipsets/rng/techbrief.pdf (Accessed March 2004).

L'ECUYER, P. (1994). "Uniform random number generation," *Annals of Operations Research*, 53: 77–120.

L'ECUYER, P. (2004). "Random number generation," in *Handbook of Computational Statistics*, eds. J. Gentle, W. Haerdle, and Y. Mori. Springer-Verlag (draft).

MARSAGLIA, G. (1995). *The Marsaglia Random Number CDROM including the DIEHARD Battery of Tests of Randomness*, http://stat.fsu.edu/pub/diehard/ (Accessed March 2004).

PETERSON, I. (2003). "The bias of random-number generators," http://www.sciencenews.org/articles/20030927/mathtrek.asp (Accessed August 2004).

SCHMID, F. and WILDING, N. (1995). "Errors in Monte Carlo simulations using shift register random number generators," *International Journal of Modern Physics C*, 6(6): 781–787.

WOLFRAM, S. (2002). *A New Kind of Science*. Champaign, Ill.: Wolfram Media, Inc.

Appendix A

```
/* source code for Microsoft Visual C++ 6.0 */

#include <stdio.h>
#include <stdlib.h>

#define M 27
#define L 134217728
#define B 16777216
#define F 2000
#define MAXPRIME 17393

/* L is the length of the sequence in bits; L = 2^M */
/* B is the size of the sequence in bytes */
/* F is the number of (odd) primes to use */
/* MAXPRIME is the Fth odd prime (the (F+1)th prime) */

unsigned long raise2to(unsigned long i);
void invert_prime_multiples(unsigned short prime);

unsigned char *x;

void main(void)
{
      unsigned long i, j = 0, k;
      FILE *fp;

/* get the first F odd primes by getting the first F+1 primes */
/* and ignoring the first prime, which is even */
/* prime[0]=2; prime[1] = 3; the Fth odd prime is prime[F] */

      unsigned short *prime;
      prime = (unsigned short *) malloc((F+1) * sizeof(unsigned short));
      unsigned char *c;
      c = (unsigned char *) malloc((MAXPRIME+1) * sizeof(unsigned char));

/* data types might need to be changed if F is increased */

      for (i = 1; i <= MAXPRIME; i++)
            c[i] = 1;
      c[1] = 0;
      prime[0] = 2;
      do
      {
            for (i = prime[j]; i <= MAXPRIME; i += prime[j])
                  c[i] = 0;
            i = prime[j] + 1;
            while (c[i] == 0)
                  i++;
            prime[++j] = i;
      }
      while (j < F);
      free(c);

/* initialize the sequence by bytes instead of by bits */
/* to reduce memory requirements and processing time */

      x = (unsigned char *) malloc(B * sizeof(unsigned char));
```

```
      x[0] = 0x69;
      x[1] = ~x[0];
      for (i = 1; i < (M-3); i++)
      {
            k = raise2to(i);
            for (j = 0; j < k; j++)
                  x[k+j] = ~x[j];
      }

/* if the sequence is to be initially set to all zero bits, use */
/*    for (i = 0; i < B; i++) */
/*          x[i]=0;          */

/* invert the bits which are multiples of the first F odd primes */

      for (i = 1; i <= F; i++)
            invert_prime_multiples(prime[i]);

/* save the file */
/* it is very important to open the file in binary mode */
/* if the file is opened in text mode, bytes with a value of */
/* 10 will be replaced with two bytes with values of 13 and 10 */
/* and the sequence will "fail" a lot of the tests */

      fp = fopen("binseq.dat", "wb");
      fwrite(x, B, 1, fp);
      fclose(fp);
}

unsigned long raise2to(unsigned long i)
{
      unsigned long j, y=1;
      for (j = 1; j <= i; j++)
            y = y*2;
      return y;
}

void invert_prime_multiples(unsigned short prime)
{
      unsigned long i, k;
      for (i = prime-1; i < L; i += prime)
      {
            k = i/8;
            switch (i%8)
            {
                  case 0: {x[k] ^= 0x80; break;}
                  case 1: {x[k] ^= 0x40; break;}
                  case 2: {x[k] ^= 0x20; break;}
                  case 3: {x[k] ^= 0x10; break;}
                  case 4: {x[k] ^= 0x08; break;}
                  case 5: {x[k] ^= 0x04; break;}
                  case 6: {x[k] ^= 0x02; break;}
                  case 7: x[k] ^= 0x01;
            }
      }
}
```