,

Can the Bits of π, e, $\log_e 2$, ϕ, and $\sqrt{2}$ be used to Generate Random Numbers?

Ronald S. Remmel

Abstract. Randomness can be simply defined as the lack of any discernable pattern. When numbers such as π, e, $\log_e 2$, the golden ratio $\phi = (\sqrt{5}+1)/2$, and $\sqrt{2}$ are computed to millions of digits, they appear random. In this paper we computed each number to 2^{28} bits, and ran exhaustive statistical tests on the bit strings. All these numbers passed all the tests for randomness. Applications to Monte Carlo integration and models of stocastic processes are discussed.

1. INTRODUCTION Many processes in nature are believed to be random. In quantum mechanics, the uncertainty principle states that both the position and the momentum of a particle cannot be measured beyond a limiting uncertainty. The wave function $\psi(\vec{x}, t)$ is a complex number, whose absolute value squared $|\psi|^2$ gives the probability density that a particle is found at point $\vec{x}$ at time t. As far as physicists know, quantum mechanics is random. Randomness occurs as well in thermodynamics and statistical mechanics, turbulance, weather, chaos theory, noise in electronic circuits, chemical and biological reactions, genetics, and evolution. Million-dollar medical trials often involve thousands of subjects so that statistical errors in tests of hypotheses can be minimized.

Such processes are often modeled using random number generators, which usually work by multiplying large integers together and by looking at the middle bits of the product; multiplication is assumed to mix up the bits in a pseudorandom fashion. Here we instead consider whether the infinite series of bits for the numbers π, e, $\log_e 2$, the golden ratio $\phi = (\sqrt{5}+1)/2$, and $\sqrt{2}$ are pseudorandom numbers which can be used as random number generators. The rational number 1/239 was also studied as a control.

Arndt and Haenel in their wonderful book about π [**?**] give a very interesting discussion about whether the decimal digits of π are random. They quote Yasumasa Kanada's 1999 calculation of 206.1 billion decimal digits of π. Kanada showed that each of the 10 digits occur with equal probability within statistics. The frequency of various combinations of digits was also shown to be random, although certain *weird combinations* of digits were discovered–the dubious practice of *numerology* of finding unusual, fluky patterns.

All of the numbers (except 1/239) we studied are irrational (not the ratio of two integers) and have an infinite number of nonrepeating binary digits, except that 1/239 has a repeating but infinite binary representation. $\sqrt{2}$ and ϕ are not transcendental (they are roots of polynomial equations with rational coefficients). The numbers π, e, and $\log_e 2$ are believed to be transcendental.

In this paper we study randomness in the binary representations of π, e, $log_e 2$, ϕ, and $\sqrt{2}$ in order to determine whether these bit streams can be used as random number generators.

2. METHODS For computing π, there are efficient power series methods [**?**] [**?**] [**?**] and iterative formulas. [**?**] [**?**] The Gauss-Legendre iterative algorithm as modified by

Schonhage (Arndt, p. 93 [**?**]) might be the fastest method of all. π has been computed to at least 10^{13} decimal digits. [**?**]

Rather than attempt a record-breaking computation involving specialized programming, we have chosen instead to use the GNU Multiprecision Floating-Point Reliable Library (MPFR). [**?**] These functions can calculate very large numbers which fit into random-access memory, with no use of the disc as virtual memory. Programs were run on a PC with 64-bit Linux Fedora 20 and programmed in C++. MPFR has π and $\log_e 2$ available as defined constants, and has functions for computing square roots and exponentials. For convenience, each number was evaluated to 2^{28} bit accuracy, and then converted to hexidecimal, and then to binary disc files for use by the statistical programs.

We also devised the following C++ random bit generator for comparison:

```
bool _ranbit()
        {_seeda = _seeda + 3880979639;    // 3474467 * 1117
        unsigned long long xa = (_seeda * _seeda ) >> 31;
        bool ba = (xa % 2 > 0);
        _seedb = _seedb + 2449993031;    // 227251 * 10781
        unsigned long long xb = (_seedb * _seedb ) >> 31;
        bool bb = (xb % 2 > 0);
        _seedc = _seedc + 2640739957;   // 37277 * 70841
        unsigned long long xc = (_seedc * _seedc ) >> 31;
        bool bc = (xc % 2 > 0);
        _seedd = _seedd + 3064999387;   // 4999999 * 613
        unsigned long long xd = (_seedd * _seedd ) >> 31;
        bool bd = (xd % 2 > 0);
        _seede = _seede + 1977326743;   // 7**11
        unsigned long long xe = (_seede * _seede ) >> 31;
        bool be = (xe % 2 > 0);
        _seedf = _seedf + 2357947691;   // 11**9
        unsigned long long xf = (_seedf * _seedf ) >> 31;
        bool bf = (xf % 2 > 0);
        _seedg = _seedg + 1911109759;   // 43**5 * 13
        unsigned long long xg = (_seedg * _seedg ) >> 31;
        bool bg = (xg % 2 > 0);
        return ( ba ^ bb ^ bc ^ bd ^ be ^ bf ^ bg );}
```

The generator is very simple. The 64-bit integer seeds are set initially to zero. Then to seeda is added 3880979639 (primes 3474467×1117); this number does not have 2 as a factor, implying that the successive values of seeda on subsequent function calls do not repeat for a very long time. Then seeda is squared, which mixes up the bits in a pseudorandom fashion. Next the 31st bit is extracted by the shift $(>> 31)$ and the modulo $(\%)$ instructions, which sets the boolean variable ba to true or false. Boolean variables bb...bg are similarly computed, but with different seeds. Because these boolean variables are each not sufficiently random, they are all exclusive-ored (the ^ instruction) to further randomize ranbit. It was difficult to make a good random number generator.

Next we used the rand() generator in the C stdlib.h library to generate a random 1 bit when rand() was greater than half. Finally as a control, the number 1/239 was calculated to 2^{28} bits; this number is a rational number and thus has a repeating string of bits. Statistical tests were programmed in C++; no statistical packages were used.

3. RESULTS

A. Means test: First, tests of the randomness of the bit strings were done. Hypothesis: one bits occur with $p_1 = 0.5$, and zero bits with $p_2 = 0.5$. The bits were divided into 16 sets of $n = 2^{24}$ bits, and the number of ones was counted. For each set the χ^2 statistic was computed: [**?**]

$$\chi^2 = \sum_{i=1}^{2} \frac{[f_i - np_i]^2}{np_i} \tag{1}$$

f_1 is the number of ones, and f_2 is the number of zeros. Each term is approximately the deviation from the mean–squared, divided by the expected frequency, so that each term is approximately one. In the limit of large n, if the null hypothesis is true, this statistic has the χ^2 distribution with one degree of freedom. The cumulative probability distribution of χ^2 gave the percentile, i.e., the goodness of fit. If the percentile exceeds 95%, the hypothesis is very unlikely to be true.

The bits for π gave the worst results; the number of one bits in each set varied from the expected number 2^{23} = 8388608:

set	one bits	χ^2	percentile
1	8392207	3.09	92
2	8388254	0.03	13
3	8390241	0.63	57
4	8387414	0.34	44
5	8388031	0.08	22
6	8385301	2.61	89
7	8382562	8.72	99.8
8	8390321	0.70	59
9	8389805	0.34	44
10	8389355	0.13	28
11	8388221	0.04	15
12	8387401	0.35	44
13	8388122	0.06	18
14	8390013	0.47	50
15	8388433	0.01	6
16	8389312	0.12	27

The 7th set had $\chi^2 = 8.72$ (99.8%). This value would normally lead us to reject the null hypothesis, except that if we consider many 16-set calculations, sometimes we will get 99.8% by chance. Thus the bit series for π is still considered random. The results for e, $\log_e 2$, ϕ, $\sqrt{2}$, ranbit(), and rand() all gave χ^2 values within statistics of randomness. The test on 1/239 failed spectacularly.

B. Skipping bits test: Next we tested whether half the bits were one, but it skipped 0...20 bits, i.e., when skipping one bit, only every other bit was counted. This test was to determine whether the bits had some kind of periodic pattern. The bit series for $\log_e 2$ gave the most deviations:

skips	ones/total	χ^2	percentile
0	0.500038	1.575	79
1	0.500081	3.518	94
2	0.499991	0.028	13
3	0.500139	5.205	98
4	0.499992	0.012	8
5	0.500006	0.006	6
6	0.500093	1.332	75
7	0.500112	1.677	80
8	0.500132	2.063	85
9	0.500045	0.213	35
10	0.500035	0.117	27
11	0.500107	1.021	69
12	0.500011	0.010	8
13	0.500034	0.088	23
14	0.499968	0.075	21
15	0.500280	5.258	98
16	0.499946	0.182	33
17	0.500136	1.102	70
18	0.499892	0.654	58
19	0.500138	1.022	69
20	0.500213	2.316	87

All these results are random, except that the skip-3 and skip-15 results are at 98%, which can occur by chance. The bit sequence for 1/239 was way off.

C. Strings of ones test: We consider that the test of counting strings of ones–like strings of heads when flipping coins–is our most significant test. The probabilities are $p_1 = 1/2$ to get a one followed by a zero, $p_2 = 1/4$ to get two ones followed by a zero, $p_3 = 1/8$ to get three ones followed by a zero, and so forth; the probabilities add up to 1. For $\log_e 2$ for n = 67,111,729 strings of ones, the following is the number f_i of each string length; this example was the worst fit to theory:

ones	f_i	np_i	numstddev
1	33,550,796	33,555,900	-1.24
2	16,785,424	16,777,900	2.11
3	8,386,374	8,388,970	-0.96
4	4,192,439	4,194,480	-1.03
5	2,098,601	2,097,240	0.95
6	1,049,817	1,048,620	1.18
7	524,710	524,310	0.55
8	261,263	262,155	-1.75
9	131,280	131,078	0.56
10	65,406	65,539	-0.52
11	32,911	32,769	0.78
12	16,471	16,385	0.67
13	8,148	8,192	-0.49
14	4,015	4,096	-1.27
15	2,063	2,048	0.33
16	963	1,024	-1.91
17	566	512	2.39
18	255	256	-0.06
19	119	128	-0.80
20	48	64	-2.00
21	31	32	insig.
22	10	16	insig.
23	9	8.0	insig.
24	5	4.0	insig.
25	3	2.0	insig.
26	1	1.0	insig.
27	0	0.5	insig.
28	0	0.25	insig.
29	1	0.125	insig.

A remarkable string of 29 ones occurred once. In order to evaluate the deviation from theory, the following was computed:

$$numstddev = (f_i - np_i)/\sqrt{np_i(1-p_i)} \tag{2}$$

f_i is the number of strings of i heads in a row. We have normalized by dividing by the standard deviation. For strings of 2, 17, and 20, the deviations were 2 or more standard deviations, which is about the expected number. A χ^2 test done on strings 1-20 gave χ^2 = 29.1 (94%), which is consistent with randomness. The results for π, e, $\log_e 2$, ϕ, $\sqrt{2}$, ranbit(), and rand() were all consistent with randomness for this very stringent test. The series for 1/239 failed miserably.

D. Tests of various bases: Next we tabulated the occurrences of the digits for base 4, 8, 16, 32, and 64. For base 4, for instance, two bits were combined to give a digit 0, 1, 2, or 3. The frequency for each digit was counted. The null hypothesis was that each digit was equally probable. The quantity numstddev (eq. 2) was calculated and the digits examined for random deviations; large deviations exceeding ± 2 were circled to see whether there were any extreme patterns. Then the χ^2 for the set of digits for each base was computed, and the cumulative χ^2 distribution calculated to give a percentile. If all the digits occur with equal frequency, then $\chi^2 = 0$ (0%). If the digits are not

random, the percentile should exceed 95%. The percentiles for all bases for all numbers are listed:

number	base 4	base 8	base 16	base 32	base 64
π	17	96	71	32	24
e	23	65	11	80	43
$\log_e 2$	68	53	61	99.1	26
ϕ	97.7	85	95.8	86	98.2
$\sqrt{2}$	3	1.0	51	44	79
ranbit()	72	43	22	86	56
rand()	79	97.2	82	31	82
1/239	bad	bad	bad	bad	bad

Two results exceeded 98%, but this could occur by chance. The other percentages range all up and down the scale, indicating randomness. The number 1/239 failed the test.

E. Flatness test: Next we produced a series of random numbers by taking sets of 16 bits and dividing by $2^{16} = 65,536$ to produce floating-point numbers in the range 0...1. In order to test uniformity, each number was put into one of 64 bins each of width 1/64, and histogrammed. A χ^2 test was done assuming p = 1/64 for each bin (63 degrees of freedom). Results:

number	χ^2	percentile
π	65.7	62
e	63.9	56
$\log_e 2$	46.4	6
ϕ	66.7	65
$\sqrt{2}$	66.7	65
ranbit()	59.4	39
rand()	63.0	52
1/239	bad	bad

The χ^2 values are all consistent with randomness, except for 1/239. The deviations for each of the 64 bins were examined and were consistent with a flat distribution.

F. Correlation test: Correlations between successive random numbers are very bad because they can skew Monte Carlo calculations involving several random numbers. In order to test for correlation, the Pearson's r-coefficient (Pearson's product-moment correlation coefficient) [**?**] is defined:

$$r = \frac{1}{n-1} \sum_{i=1}^{n} \frac{x_i - m_x}{\sigma_x} \times \frac{y_i - m_y}{\sigma_y} \tag{3}$$

x_i and y_i are two successive random numbers on the range 0...1. The means m_x and m_y of course are both 1/2. Because the distribution is assumed known, the standard deviations σ_x and σ_y can both be calculated to be $\sqrt{1/12}$. If the successive x_i and y_i are positively correlated, both will tend to be positive together, or both negative together, which makes r positive. If they are inversely correlated, r will tend to be negative. We need an error estimate for r; the standard deviation of r can be calculated to be $1/\sqrt{n}$. Results:

number	r	error
π	0.00014	0.00035
e	0.00010	0.00035
$\log_e 2$	-0.00044	0.00035
ϕ	0.00029	0.00035
$\sqrt{2}$	0.00002	0.00035
ranbit()	0.00087	0.00035
rand()	-0.00039	0.00035
1/239	0.00432	0.00035

All correlations are within statistics of zero, except that ranbit() is 2.5 standard deviations from zero–still okay. 1/239 is bad.

G. Runs above and below the mean: Next we considered Gaussian-distributed random numbers. Two Gaussian numbers n1 and n2 (mean = 0 and variance = 1) can be produced from two numbers x and y (uniform on 0...1) by the following C++ trick:

```
double theta = 6.2831852*x;
double r = sqrt( -2.0*log(y) );
double n1 = r*cos(theta);
double n2 = r*sin(theta);
```

The next test is for runs above and below the mean (=0); [**?**] numbers above zero are denoted by *a* and below zero by *b*. For example:

```
a b a a a b b a b a b b
```

These are grouped into runs:

```
a b aaa bb a b a bb
```

This series has 8 runs. If the series is random, the number of runs r has:

$$E(r) = \frac{n+2}{2}; \quad var(r) = \frac{n}{4}(1 - 1/(n-1)) \tag{4}$$

We generated n = 16,777,216 events. To test goodness of fit, we defined: numstddev = (r - E(r))/$\sqrt{var(r)}$:

number	r	numstddev
π	8,388,493	-0.06
e	8,387,303	-0.64
$\log_e 2$	8,390,620	0.98
ϕ	8,388,537	-0.03
$\sqrt{2}$	8,388,117	-0.24
ranbit()	8,388,219	-0.19
rand()	8,388,302	-0.15
1/239	8,318,116	-34.4

All the runs were within one standard deviation, except for 1/239.

H. Up and down runs: Up and down runs is our next test of the Gaussian-distributed random numbers. [**?**] For each sample we note whether it is above the previous sample by *a*, or below by *b*, and group them into runs. For n samples, count the total number of runs r. It can be shown that:

$$E(r) = \frac{1}{3}(2n-1); \quad var(r) = \frac{1}{90}(16n-29) \tag{5}$$

As previously, we calculated r and numstddev for n = 16,777,216 samples:

number	r	numstddev
π	11,185,015	0.12
e	11,185,061	0.15
$\log_e 2$	11,187,017	1.28
ϕ	11,183,611	-0.69
$\sqrt{2}$	11,184,345	-0.27
ranbit()	11,183,713	-0.64
rand()	11,185,401	0.35
1/239	10,996,827	-108

All the numbers are within statistics of randomness, except for 1/239.

I. Monte Carlo integration test: A good test for randomness is when the generator gives the correct answer for Monte Carlo integrations. Monte Carlo integrations are easy to program. Even though Monte Carlo calculations converge only as $n^{-1/2}$, on modern computers it takes only a few minutes to generate 10^8 events, which gives 10^{-4} accuracy. We compute our favorite numbers $\pi/4$ and $\log_e 2$, and then a pathological integral z as follows:

$$\frac{\pi}{4} = \int_0^1 \sqrt{1-x^2}dx; \quad \log_e 2 = \int_1^2 \frac{dx}{x}; \quad z = \int_0^1 (\sin(1/x)+1)dx \tag{6}$$

The left integral calculates the area of the upper right quadrant of a unit circle. We generate two uniformly-distributed random numbers: $0 < x < 1$ and $0 < y < 1$. An event is under the curve if $y < \sqrt{1-x^2}$, which is equivalent to $x^2 + y^2 < 1$.

For the middle integral, generate $1 < x < 2$ and $0 < y < 1$, and test $y < 1/x$.

The right integral has a finite result because the function is always between 0 and 2. However, it oscillates faster and faster as $x \to 0$. I could not find the answer in any table of integrals. Attempting numerical integration using Simpson's rule or Gaussian quadrature will not work well because those methods require evaluating the function at several points, which might be located wildly between 0 and 2. However, Monte Carlo integration works well because essentially all values of x are generated randomly.

Error estimate: This is a Bernoulli distribution with p, the probability to be under the curve, and q = 1 - p the probability to be above the curve. For n = 2^{24} events, Bernoulli distributions have E = np, and var = npq. Let b be the number of events below the curve. We again use the goodness-of-fit statistic $numstddev = (b-E)/\sqrt{var}$. The correct answers are $\pi/4 = 0.785398...$ and $\log_e 2 = 0.693147...$

number	calc $\pi/4$	numstddev		calc $\log_e 2$	numstddev		z	error
π	0.785415	0.12		0.693179	0.20		1.50409	$\pm$0.00030
e	0.785309	-0.63		0.693022	-0.79		1.50398	$\pm$0.00030
$\log_e 2$	0.785090	-2.17		0.692851	-1.86		1.50412	$\pm$0.00030
ϕ	0.785414	0.11		0.692977	-1.07		1.50403	$\pm$0.00030
$\sqrt{2}$	0.785493	0.67		0.693112	-0.22		1.50414	$\pm$0.00030
ranbit()	0.785225	-1.22		0.693070	-0.48		1.50408	$\pm$0.00030
rand()	0.785409	0.08		0.693215	0.43		1.50464	$\pm$0.00030
1/239	0.882353	683		0.773110	502		1.54622	$\pm$0.00030

All the results agree within 1-2 standard deviations, except for 1/239.

4. DISCUSSION We have proven that the bit streams for π, e, $\log_e 2$, ϕ, and $\sqrt{2}$ pass all the tests for randomness. Furthermore, the computer functions ranbit() and rand() pass as well.

The statistician now has available powerful tools for simulating random processes. Strategies for games of chance such as poker and blackjack can be tested. Monte Carlo methods can easily be used to evaluate multidimensional integrals for which no analytical solutions are available. Chemical reactions in which the molecules interact randomly can be modeled. Even evolution with random matings, DNA combinations, and various chances of survival can be programmed.

Quantum-mechanical experiments can be simulated, such as particle collisions and radioactive decays. For instance, we once studied 3,000,000 elementary particle decays of k-mesons into three charged pi-mesons, looking at both the particle and its antiparticle: [**?**]

$$\kappa^+ \to \pi^+\pi^+\pi^- \quad and \quad \kappa^- \to \pi^-\pi^-\pi^+ \tag{7}$$

K-mesons are produced in particle accelerators, and decay randomly with a lifetime of 12.3 ns into lighter particles. 5.6% of the time they randomly decay into three charged pi-mesons, as shown above. The pi-mesons fly off randomly in all directions, subject to conservation of energy and momentum. Because our detectors missed some of the particles, we had to use random numbers to simulate the decays. The physics result was consistent with randomness.

This paper shows that the bits of π, e, $\log_e 2$, ϕ, and $\sqrt{2}$ pass all tests for randomness, and can be used as random number generators.

REFERENCES

1. J. Arndt and C. Haenel, π Unleashed, Springer, Berlin, 2001, chap. 2 and p. 29. This book can be downloaded from: http://bookzz.org/book/1167868/05939f
2. D.V. Chudnovsky and G.V. Chudnovsky, Approximations and complex multiplication according to Ramanajan, *Ramanujan Revisited. Proceedings of the Centenary Conference, University of Illinois at Urbana-Champaign, June 1-5, 1987*, Andrews et al., ed., Academic Press, Boston, 1988, pp. 375-472
3. R.S. Remmel, A more efficient algorithm to compute π, submitted for publication
4. Gauss-Legendre algorithm http://en.wikipedia.org/wiki/Gauss-Legendre_algorithm
5. Borwein's algorithm, http://en.wikipedia.org/wiki/Borwein's_algorithm
6. A.J. Yee and S. Kondo, Round 2...10 trillion digits of pi, 2013, http://numberworld.org/misc_runs/pi-10t/details.html
7. GNU MPFR Multiprecision floating-point reliable library, http://mpfr.org
8. Pearson's chi-squared test, http://en.wikipedia.org/wiki/Pearson's_chi_square_test
9. Pearson's product-moment correlation coefficient, http://en.wikipedia.org/wiki/Pearson_product_moment_correlation_coefficient

10. B.W. Lindgren and G.W. McElrath, Introduction to probability and statistics, Macmillan, New York, 1959, p. 159
11. Ibid., p. 161
12. W.T. Ford et al., Experimental study of the tau+- decay matrix element, Physics Letters 388 (1972) 335-338

RONALD S. REMMEL (remmellabs@juno.com www.remmellabs.com) received his BS in physics from Caltech, and his PhD in physics from Princeton. He was on the faculties of the Univ. of Arkansas for Medical Science and Boston Univ. He has published 26 scientific papers, 1 computer book, 4 medical course manuals, and 8 science fiction novels. In 1991 he founded Remmel Labs, which manufactures eye movement monitors for visual and oculomotor experiments.
Remmel Labs, 1811 Parkfair Ct., Katy, TX 77450 USA
remmellabs@juno.com