Pseudo-random numbers: $\underbrace{\text{a}\ \overbrace{\text{line}}^{\text{of code}}}_{\text{mostly}}$ at a time

Nelson H. F. Beebe

Research Professor
University of Utah
Department of Mathematics, 110 LCB
155 S 1400 E RM 233
Salt Lake City, UT 84112-0090
USA

Email: beebe@math.utah.edu, beebe@acm.org,
beebe@computer.org (Internet)
WWW URL: http://www.math.utah.edu/~beebe
Telephone: +1 801 581 5254
FAX: +1 801 581 4148

22 April 2015

What are random numbers good for?

- Decision making (e.g., coin flip).
- Generation of numerical test data.
- Generation of unique cryptographic keys.
- Search and optimization via random walks.
- Selection: `quicksort` (C. A. R. Hoare, *ACM Algorithm 64: Quicksort*, Comm. ACM. **4**(7), 321, July 1961) was the first widely-used divide-and-conquer *algorithm* to reduce an $\mathcal{O}(N^2)$ problem to (on average) $\mathcal{O}(N \lg(N))$. Cf. Fast Fourier Transform (Clairaut (1754), Lagrange (1759), Gauss (1805 unpublished, 1866) [Latin], Runge (1903), Danielson and Lanczos [crystallography] (1942), Cooley and Tukey (1965)).

Historical note: al-Khwarizmi

Abu 'Abd Allah Muhammad ibn Musa al-Khwarizmi (ca. 780–850) is the father of *algorithm* and of *algebra*, from his book *Hisab Al-Jabr wal Mugabalah (Book of Calculations, Restoration and Reduction)*. He is celebrated in a 1200-year anniversary Soviet Union stamp:

ПОЧТА СССР
1983
4к
1200 ЛЕТ
Мухаммед аль-Хорезми

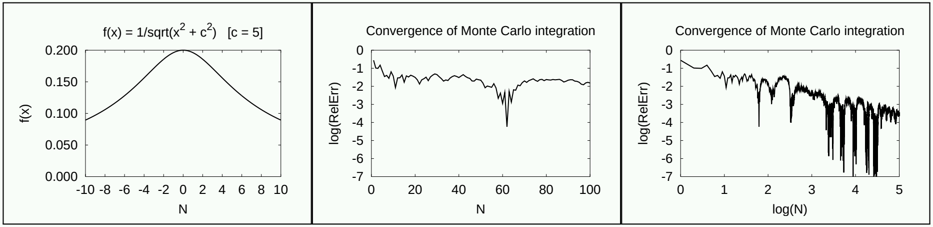
What are random numbers good for? . . .

- Simulation.
- Sampling: unbiased selection of random data in statistical computations (opinion polls, experimental measurements, voting, Monte Carlo integration, . . .). The latter is done like this (x_k is random in (a, b)):

$$\int_a^b f(x)\,dx \approx \left(\frac{(b-a)}{N} \sum_{k=1}^{N} f(x_k) \right) + \mathcal{O}(1/\sqrt{N})$$

Monte Carlo integration

Here is an example of a simple, smooth, and exactly integrable function, and the relative error of its Monte Carlo integration:



When is a sequence of numbers random?

- Computer numbers are rational, with limited precision and range. Irrational and transcendental numbers are not represented.
- Truly random integers would have occasional repetitions, but most pseudo-random number generators produce a long sequence, called the *period*, of distinct integers: these cannot be random.
- It isn't enough to conform to an expected distribution: the *order* that values appear in must be haphazard.
- Mathematical characterization of randomness is possible, but difficult.
- The best that we can usually do is *compute statistical measures of closeness* to particular expected distributions.

Distributions of pseudo-random numbers

- Uniform (most common).
- Exponential.
- Normal (bell-shaped curve).
- Logarithmic: if $\mathrm{ran}()$ is uniformly-distributed in (a, b), define $\mathrm{randl}(x) = \exp(x\,\mathrm{ran}())$. Then $a\,\mathrm{randl}(\ln(b/a))$ is logarithmically distributed in (a, b). [Important use: sampling in floating-point number intervals.]

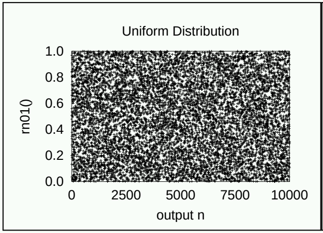
Distributions of pseudo-random numbers ...

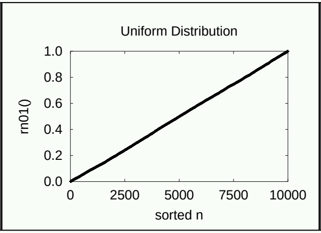
Sample logarithmic distribution:

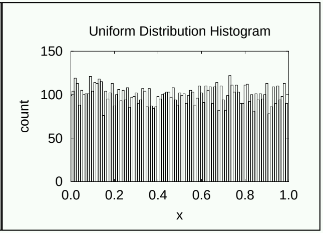
```
% hoc
a = 1
b = 1000000
for (k = 1; k <= 10; ++k) printf "%16.8f\n", a*randl(ln(b/a))
    664.28612484
 199327.86997895
 562773.43156449
  91652.89169494
     34.18748767
    472.74816777
     12.34092778
      2.03900107
  44426.83813202
     28.79498121
```

Uniform distribution

Here are three ways to visualize a pseudo-random number distribution, using the Dyadkin-Hamilton generator function `rn01()`, which produces results uniformly distributed on $(0, 1]$:

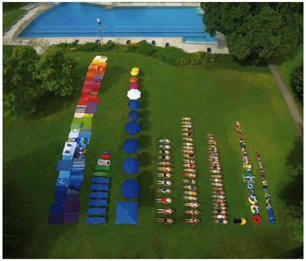
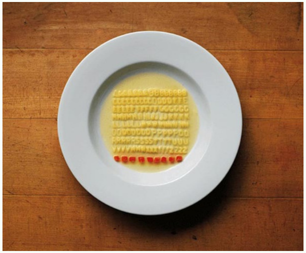
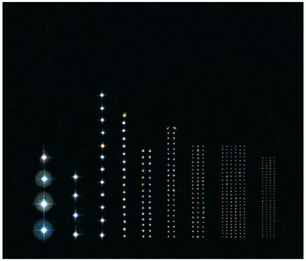






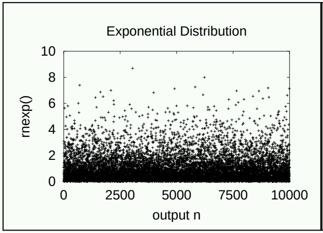
Disorder and order

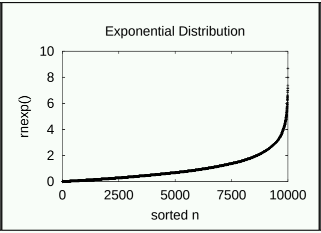
The Swiss artist Ursus Wehrli dislikes randomness:

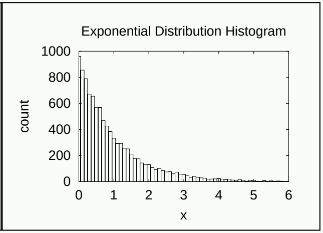




Exponential distribution

Here are visualizations of computations with the Dyadkin-Hamilton generator `rnexp()`, which produces results exponentially distributed on $[0, \infty)$:



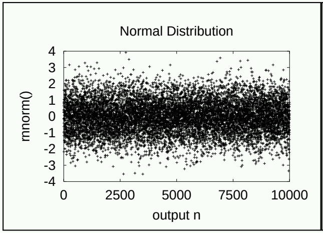


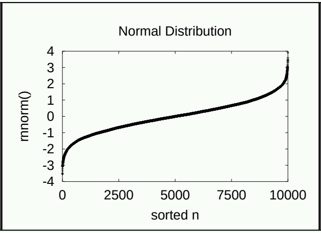


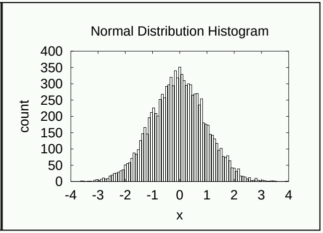
Even though the theoretical range is $[0, \infty)$, the results are practically always modest: the probability of a result as big as 50 is smaller than 2×10^{-22}. At one result per microsecond, it could take 164 million years of computing to encounter such a value!

Normal distribution

Here are visualizations of computations with the Dyadkin-Hamilton generator `rnnorm()`, which produces results normally distributed on $(-\infty, +\infty)$:



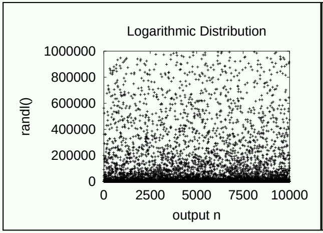


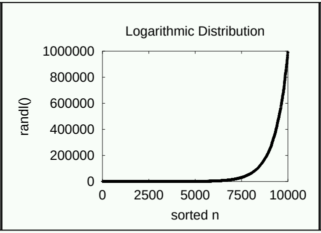


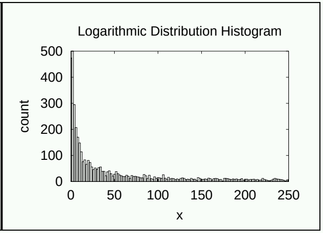
Results are never very large: a result as big as 7 occurs with probability smaller than 5×10^{-23}. At one result per microsecond, it could take 757 million years of computing to encounter such a value.

Logarithmic distribution

Here are visualizations of computations with the `hoc` generator `randl(ln(1000000))`, which produces results logarithmically distributed on $(1, 1000000)$:







The graphs are similar to those for the exponential distribution, but here, the result range is controlled by the argument of `randl()`.

Goodness of fit: the χ^2 measure

Given a set of n *independent observations* with measured values M_k and expected values E_k, then $\sum_{k=1}^{n} |(E_k - M_k)|$ is a measure of goodness of fit. So is $\sum_{k=1}^{n}(E_k - M_k)^2$. Statisticians use instead a measure introduced in 1900 by one of the founders of modern statistics, the English mathematician Karl Pearson (1857–1936):

(1880)

$$\chi^2 \text{ measure} = \sum_{k=1}^{n} \frac{(E_k - M_k)^2}{E_k}$$

Equivalently, if we have s categories expected to occur with probability p_k, and if we take n samples, counting the number Y_k in category k, then

$$\chi^2 \text{ measure} = \sum_{k=1}^{s} \frac{(np_k - Y_k)^2}{np_k}$$

Goodness of fit: the χ^2 measure ...

The theoretical χ^2 distribution depends on the number of degrees of freedom, and table entries look like this (highlighted entries are referred to later):

D.o.f.	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
$\boldsymbol{\nu = 1}$	0.00016	0.00393	0.1015	**0.4549**	1.323	3.841	**6.635**
$\nu = 5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
$\boldsymbol{\nu = 10}$	2.558	3.940	6.737	9.342	12.55	18.31	**23.21**
$\nu = 50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15

For example, this table says:

> **For $\boldsymbol{\nu = 10}$, the probability that the χ^2 measure is no larger than 23.21 is 99%.**
> **In other words, χ^2 measures larger than 23.21 should occur only about 1% of the time.**

Goodness of fit: coin-toss experiments

Coin toss has one degree of freedom, $\boldsymbol{\nu = 1}$, because if it is not heads, then it must be tails.

```
% hoc
for (k = 1; k <= 10; ++k) print randint(0,1), ""
0 1 1 1 0 0 0 0 1 0
```

This gave four 1s and six 0s:

$$
\begin{aligned}
\chi^2 \text{ measure} &= \frac{(10 \times 0.5 - 4)^2 + (10 \times 0.5 - 6)^2}{10 \times 0.5} \\
&= 2/5 \\
&= 0.40
\end{aligned}
$$

Goodness of fit: coin-toss experiments . . .

From the table, for $\boldsymbol{\nu = 1}$, we expect a χ^2 measure no larger than **0.4549** half of the time, so our result is reasonable.
On the other hand, if we got nine 1s and one 0, then we have

$$\begin{aligned}\chi^2 \text{ measure} &= \frac{(10 \times 0.5 - 9)^2 + (10 \times 0.5 - 1)^2}{10 \times 0.5} \\ &= 32/5 \\ &= 6.4\end{aligned}$$

This is close to the tabulated value **6.635** at $p = 99\%$. That is,
we should only expect nine-of-a-kind about once in every 100 experiments.
If we had all 1s or all 0s, the χ^2 measure is 10 (probability $p = 0.998$) [**twice in 1000 experiments**].
If we had equal numbers of 1s and 0s, then the χ^2 measure is 0, indicating an exact fit.

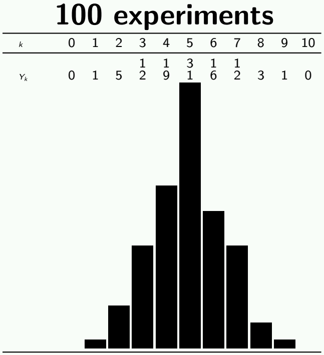
Goodness of fit: coin-toss experiments …

Let's try 100 similar experiments, counting the number of 1s in each experiment:

```
% hoc
for (n = 1; n <= 100; ++n) {
    sum = 0
    for (k = 1; k <= 10; ++k) \
        sum += randint(0,1)
    print sum, ""
}
4 4 7 3 5 5 5 2 5 6 6 6 3 6 6 7 4 5 4 5 5 4
3 6 6 9 5 3 4 5 4 4 4 5 4 5 5 4 6 3 5 5 3 4
4 7 2 6 5 3 6 5 6 7 6 2 5 3 5 5 5 7 8 7 3 7
8 4 2 7 7 3 3 5 4 7 3 6 2 4 5 1 4 5 5 5 6 6
5 6 5 5 4 8 7 7 5 5 4 5
```

Goodness of fit: coin-toss experiments . . .

The measured frequencies of the sums are:



k	0	1	2	3	4	5	6	7	8	9	10
Y_k	0	1	5	12	19	31	16	12	3	1	0

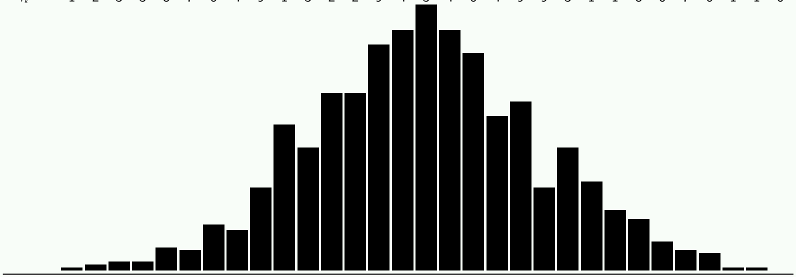
Notice that nine-of-a-kind occurred **once** each for 0s and 1s, as predicted.

Goodness of fit: coin-toss experiments . . .

A simple one-character change on the outer loop limit produces the next experiment:

1000 experiments

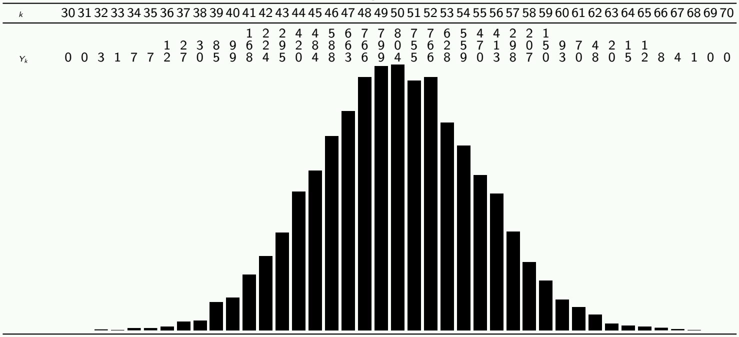
k	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65
Y_k	1	2	3	3	8	7	16	14	29	51	43	62	62	79	84	93	84	76	54	59	29	43	31	21	18	10	7	6	1	1	0



Goodness of fit: coin-toss experiments . . .

Another one-character change gives us this:

10 000 experiments



k	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70
Y_k	0	0	3	1	7	7	12	27	30	85	99	168	224	295	420	484	588	663	766	799	804	755	766	628	559	470	413	298	207	150	93	70	48	20	15	12	8	4	1	0	0

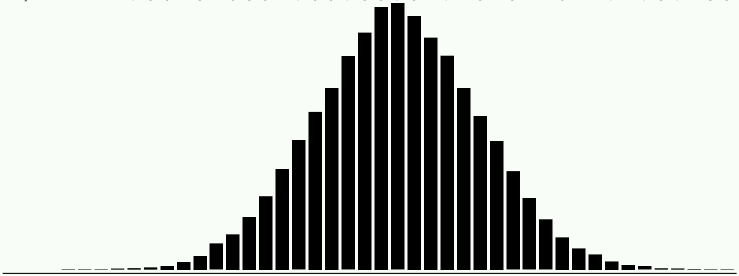
Goodness of fit: coin-toss experiments ...

A final one-character change gives us this result for one million coin tosses:

100 000 experiments

k	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
Y_k	1	4	12	34	47	70	118	242	418	802	1086	1633	2259	3112	3987	4880	5609	6587	7320	8113	8227

k	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70
Y_k	7828	7171	6607	5604	4740	3962	3029	2212	1544	996	654	474	257	141	107	43	37	21	5	5



Are the digits of π random?

Here are χ^2 results for the digits of π from recent computational records ($\boldsymbol{\chi^2(\nu = 9, p = 0.99) \approx 21.67}$):

π

Digits	**Base**	χ^2	**p(χ^2)**
6B	10	9.00	0.56
50B	10	5.60	0.22
200B	10	8.09	0.47
1T	10	14.97	0.91
1T	16	7.94	0.46

$1/\pi$

Digits	**Base**	χ^2	**p(χ^2)**
6B	10	5.44	0.21
50B	10	7.04	0.37
200B	10	4.18	0.10

Whether the fractional digits of π, and most other transcendentals, are *normal* ($\approx$ equally likely to occur) is an outstanding unsolved problem in mathematics.

Are the first 1000 fractional digits of π random?

```
3.14159265358979323846264338327950288419716939937510
  58209749445923078164062862089986280348253421170679
  82148086513282306647093844609550582231725359408128
  48111745028410270193852110555964462294895493038196
  44288109756659334461284756482337867831652712019091
  45648566923460348610454326648213393607260249141273
  72458700660631558817488152092096282925409171536436
  78925903600113305305488204665213841469519415116094
  33057270365759591953092186117381932611793105118548
  07446237996274956735188575272489122793818301194912
  98336733624406566430860213949463952247371907021798
  60943702770539217176293176752384674818467669405132
  00056812714526356082778577134275778960917363717872
  14684409012249534301465495853710507922796892589235
  42019956112129021960864034418159813629774771309960
  51870721134999999837297804995105973173281609631859
  50244594553469083026425223082533446850352619311881
  71010003137838752886587533208381420617177669147303
  59825349042875546873115956286388235378759375195778
  18577805321712268066130019278766111959092164201989
```

Are the first 1000 fractional digits of π random?

In the first 1000 fractional digits of π:

- 83 digit pairs (81 expected)
- 77-digit sequence without a 4 (probability: $(9/10)^{77} \approx 0.0003$)
- six consecutive 9 digits (probability: 1/1,000,000)
- last five digits are a calendar year (probability: 1/100,000)

Conclusion: for a finite sequence of digits, the answer is no!
See Aaldert Compagner, *Definitions of randomness*, American Journal of Physics **59**(8) 700–705 (1991).
URL `http://m.ajp.aapt.org/resource/1/ajpias/v59/i8/p700_s1`

Is your name in π?

From `http://www.dr-mikes-maths.com/pisearch.html`:
NELSON was not found, but I searched 31 415 929 digits of π, and found BEEBE 4 times. The first occurrence was at position 846 052. What this means is that

$$\begin{aligned}\pi = \quad & 3 + \ldots + \\ & \frac{B}{27^{278246052}} + \frac{E}{27^{278246053}} + \frac{E}{27^{278246054}} + \frac{B}{27^{278246055}} + \frac{E}{27^{278246056}} + \ldots\end{aligned}$$

where $A = 1$, $B = 2$, $C = 3$, and so on.

The Central-Limit Theorem

The famous **Central-Limit Theorem** (de Moivre (1718), Laplace (1810), and Cauchy (1853)), says:

> **A suitably normalized sum of independent random variables is likely to be normally distributed, as the number of variables grows beyond all bounds. It is not necessary that the variables all have the same distribution function or even that they be wholly independent.**
>
> *— I. S. Sokolnikoff and R. M. Redheffer*
> *Mathematics of Physics and Modern Engineering, 2nd ed.*

The Central-Limit Theorem . . .

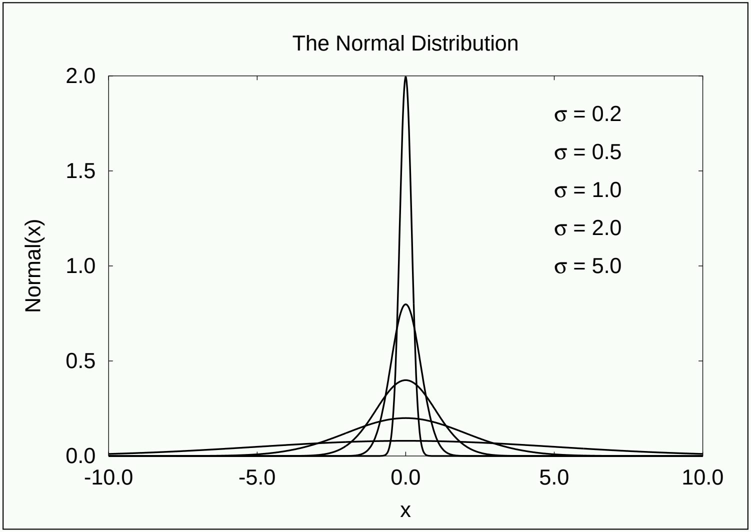
In mathematical terms, this is

$$P(n\mu + r_1\sqrt{n} \le X_1 + X_2 + \cdots + X_n \le n\mu + r_2\sqrt{n})$$
$$\approx \frac{1}{\sigma\sqrt{2\pi}} \int_{r_1}^{r_2} \exp(-t^2/(2\sigma^2))\,dt$$

where the X_k are independent, identically distributed, and bounded random variables, μ is their **mean value**, σ is their **standard deviation**, and σ^2 is their **variance**.

The Central-Limit Theorem . . .

The integrand of this probability function looks like this:



The Central-Limit Theorem …

The normal curve falls off very rapidly. We can compute its area in $[-x, +x]$ with a simple midpoint quadrature rule like this:

```
func f(x) {
    global sigma;
    return (1/(sigma*sqrt(2*PI)))* exp(-x*x/(2*sigma**2))
}

func q(a,b) {
    n = 10240
    h = (b - a)/n
    area = 0
    for (k = 0; k < n; ++k) \
         area += h*f(a + (k + 0.5)*h);
    return area
}
```

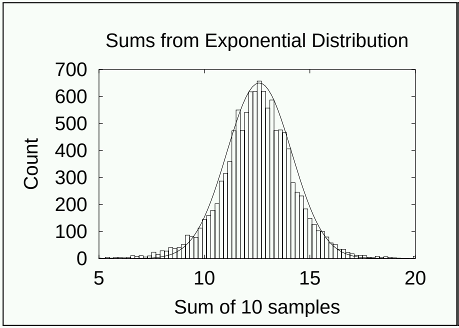
The Central-Limit Theorem ...

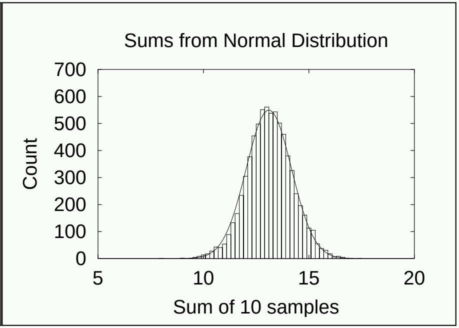
```
sigma = 3
for (k = 1; k < 8; ++k) \
    printf "%d  %.9f\n", k, q(-k*sigma,k*sigma)
1  0.682689493
2  0.954499737
3  0.997300204
4  0.999936658
5  0.999999427
6  0.999999998
7  1.000000000
```

In computer management, **99.999% (five 9's) availability** is **five minutes downtime per year**.
In manufacturing, Motorola's **6σ reliability** with 1.5σ drift is about **three defects per million** (from $q(-(6-1.5)*\sigma, +(6-1.5)*\sigma)/2$).

The Central-Limit Theorem . . .

It is remarkable that the Central-Limit Theorem applies also to nonuniform distributions. Here is a demonstration with sums from exponential and normal distributions:





Superimposed on the histograms are rough fits by eye of normal distribution curves $650\exp(-(x-12.6)^2/4.7)$ and $550\exp(-(x-13.1)^2/2.3)$.

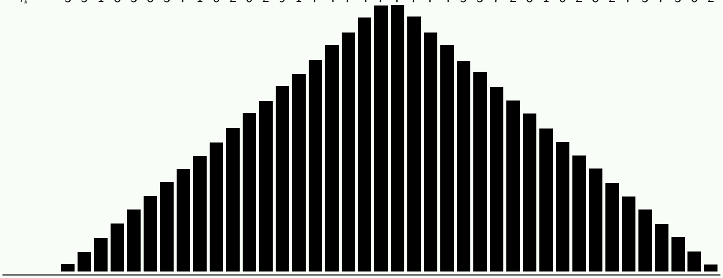
The Central-Limit Theorem . . .

Not everything looks like a normal distribution. Here is a similar experiment, using *differences* of successive pseudo-random numbers, bucketizing them into 40 bins from the range $[-1.0, +1.0]$:

10 000 experiments (counts scaled by 1/100)

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Y_k	13	35	61	88	113	138	163	187	211	236	262	290	312	339	361	387	414	437	464	487

k	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Y_k	487	467	437	414	385	365	337	312	288	261	236	212	188	162	137	113	87	63	36	12



This one is known from theory: it is a *triangular* distribution. A similar result is obtained if one takes pair sums instead of differences.

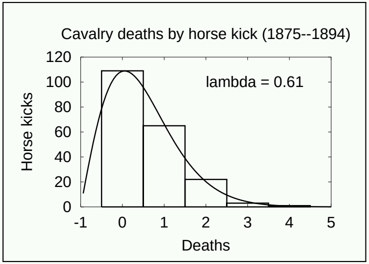
Digression: Poisson distribution

The *Poisson* distribution arises in time series when the probability of an event occurring in an arbitrary interval is proportional to the length of the interval, and independent of other events:

$$P(X = n) = \frac{\lambda^n}{n!} e^{-\lambda}$$

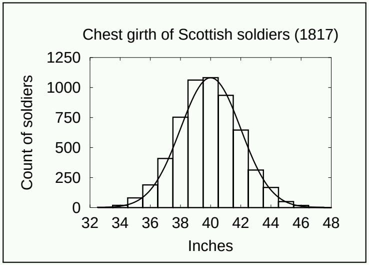
In 1898, Ladislaus von Bortkiewicz collected Prussian army data on the number of soldiers killed by horse kicks in 10 cavalry units over 20 years: 122 deaths, or an average of $122/200 = 0.61$ deaths per unit per year.

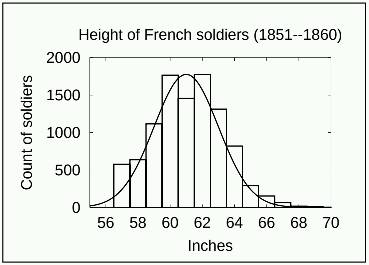
	$\lambda = 0.61$	
Deaths	Kicks (actual)	Kicks (Poisson)
0	109	108.7
1	65	66.3
2	22	20.2
3	3	4.1
4	1	0.6

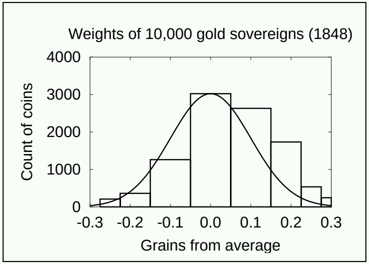


The Central-Limit Theorem . . .

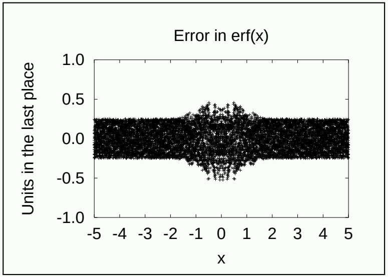
Measurements of physical phenomena often form normal distributions:

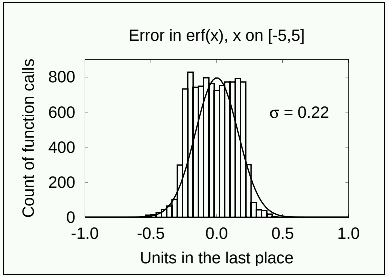


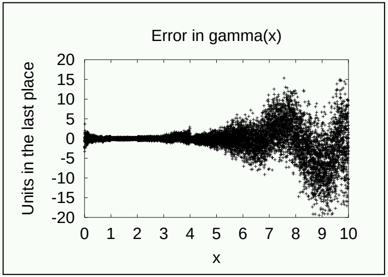


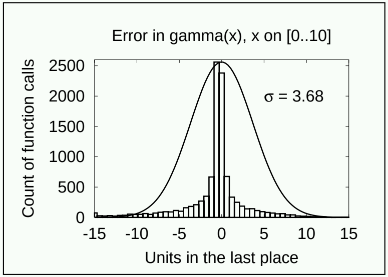


The Central-Limit Theorem . . .

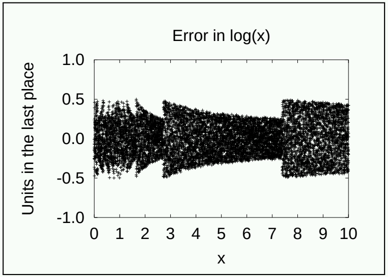


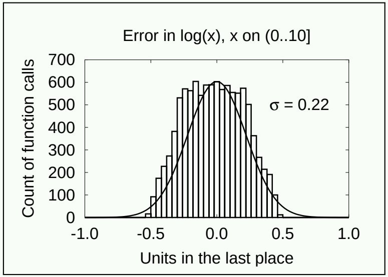


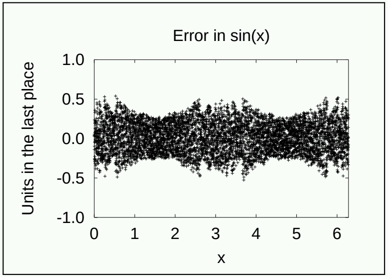


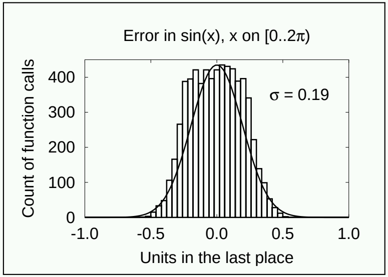


The Central-Limit Theorem . . .



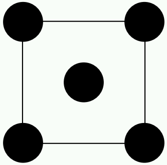






The Normal Curve and Carl-Friedrich Gauß (1777–1855)

The Normal Curve and the Quincunx

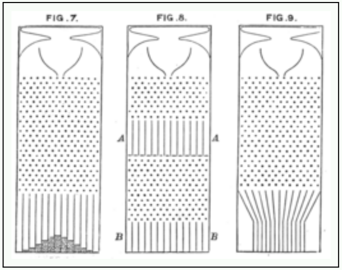


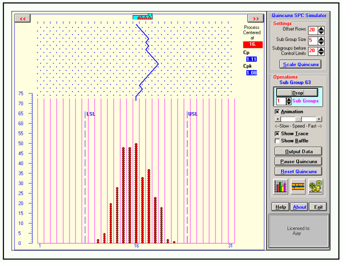
quincunx, *n.*

2. An arrangement or disposition of five objects so placed that four occupy the corners, and the fifth the centre, of a square or other rectangle; a set of five things arranged in this manner.
b. spec. as a basis of arrangement in planting trees, either in a single set of five or in combinations of this; a group of five trees so planted.

Oxford English Dictionary

The Normal Curve and the Quincunx …





Process Centered at 16.
Cp 1.11
Cpk 1.08

Settings
Offset Rows 20
Sub Group Size 5
Subgroups before Control Limits 20

Operations
Sub Group 63
1 Sub Groups
[✓] Animation
<--Slow - Speed - Fast -->
[✓] Show Trace
[ ] Show Baffle

Licensed to: Ajay

For simulations and other material on the quincunx (Galton's *bean machine*), see:

- `http://www.ms.uky.edu/~mai/java/stat/GaltonMachine.html`
- `http://www.rand.org/statistics/applets/clt.html`
- `http://www.stattucino.com/berrie/dsl/Galton.html`
- `http://teacherlink.org/content/math/interactive/flash/quincunx/quincunx.html`
- `http://www.bun.kyoto-u.ac.jp/~suchii/quinc.html`

Remarks on random numbers

Any one who considers arithmetical methods of producing random numbers is, of course, in a state of sin.

— John von Neumann (1951)

[*The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, 3rd ed., p. 1]

He talks at random; sure, the man is mad.

— Queen Margaret

[William Shakespeare's *1 King Henry VI*, Act V, Scene 3 (1591)]

A random number generator chosen at random isn't very random.

— Donald E. Knuth (1997)

[*The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, 3rd ed., p. 384]

How do we generate pseudo-random numbers?

- ❏ Linear-congruential generators (most common):
  $r_{n+1} = (ar_n + c) \bmod m$, for integers a, c, and m, where $0 < m$, $0 \le a < m$, $0 \le c < m$, with starting value $0 \le r_0 < m$.
- ❏ Fibonacci sequence (bad!):
  $r_{n+1} = (r_n + r_{n-1}) \bmod m$.
- ❏ Additive (better): $r_{n+1} = (r_{n-\alpha} + r_{n-\beta}) \bmod m$.
- ❏ Multiplicative (bad):
  $r_{n+1} = (r_{n-\alpha} \times r_{n-\beta}) \bmod m$.
- ❏ Shift register:
  $r_{n+k} = \sum_{i=0}^{k-1} (a_i r_{n+i} \pmod 2) \qquad (a_i = 0, 1)$.

How do we generate pseudo-random numbers? . . .

Given an integer $r \in [A, B)$, $x = (r - A)/(B - A + 1)$ is on $[0, 1)$. However, interval reduction by $A + (r - A) \bmod s$ to get a distribution in (A, C), where $s = (C - A + 1)$, is possible only for certain values of s. Consider reduction of $[0, 4095]$ to $[0, m]$, with $m \in [1, 9]$: we get equal distribution of remainders only for $m = 2^q - 1$:

	m	counts of remainders k mod $(m+1)$, $k \in [0, m]$									
OK	1	2048	2048								
	2	**1366**	1365	1365							
OK	3	1024	1024	1024	1024						
	4	**820**	819	819	819	819					
	5	**683**	**683**	**683**	**683**	682	682				
	6	**586**	585	585	585	585	585	585			
OK	7	512	512	512	512	512	512	512	512		
	8	**456**	455	455	455	455	455	455	455	455	
	9	**410**	**410**	**410**	**410**	**410**	**410**	409	409	409	409

How do we generate pseudo-random numbers? . . .

Samples from other distributions can usually be obtained by some suitable transformation. Here is the simplest generator for the normal distribution, assuming that `randu()` returns uniformly-distributed values on $(0, 1]$:

```
func randpmnd() \
{ ## Polar method for random deviates
  ## Algorithm P, p. 122, from Donald E. Knuth,
  ## The Art of Computer Programming, vol. 2, 3/e, 1998
  while (1) \
  {
    v1 = 2*randu() - 1  # v1 on [-1,+1]
    v2 = 2*randu() - 1  # v2 on [-1,+1]
    s = v1*v1 + v2*v2   # s on [0,2]
    if (s < 1) break    # exit loop if s inside unit circle
  }
  return (v1 * sqrt(-2*ln(s)/s))
}
```

Period of a sequence

All pseudo-random number generators eventually reproduce the starting sequence; the *period* is the number of values generated before this happens.

Widely-used historical generators have periods of a few tens of thousands to a few billion, but good generators are now known with very large periods:

$> 10^{14}$	POSIX `drand48()` LCG (2^{48}) (1982),
$> 10^{57}$	Marsaglia short and fast `xorshift()` (2^{192}) (2003),
$> 10^{18}$	Numerical Recipes `ran2()` (1992),
$> 10^{38}$	NIST Advanced Encryption Standard (AES) (2^{128}) (2003),
$> 10^{449}$	Matlab's `rand()` ($\approx 2^{1492}$ Columbus generator),
$> 10^{2894}$	Marsaglia's Monster-KISS (2000),
$> 10^{6001}$	Matsumoto and Nishimura's Mersenne Twister (1998),
$> 10^{14100}$	Deng and Xu (2003),
$> 10^{16736}$	Berdnikov, Trutia, & Compagner `MathLink` (1996).

Reproducible sequences

In computational applications with pseudo-random numbers, it is *essential* to be able to reproduce a previous calculation. Thus, generators are required that can be set to a given **initial seed**:

```
% hoc
for (k = 0; k < 3; ++k) \
{
    setrand(12345)
    for (n = 0; n < 10; ++n) print int(rand()*100000),""
    println ""
}
88185 5927 13313 23165 64063 90785 24066 37277 55587 62319
88185 5927 13313 23165 64063 90785 24066 37277 55587 62319
88185 5927 13313 23165 64063 90785 24066 37277 55587 62319
```

Reproducible sequences . . .

If the seed is not reset, different sequences are obtained for each test run. Here is the same code as before, with the `setrand()` call disabled:

```
for (k = 0; k < 3; ++k) \
{
    ## setrand(12345)
    for (n = 0; n < 10; ++n) print int(rand()*100000),""
    println ""
}
36751 37971 98416 59977 49189 85225 43973 93578 61366 54404
70725 83952 53720 77094 2835 5058 39102 73613 5408 190
83957 30833 75531 85236 26699 79005 65317 90466 43540 14295
```

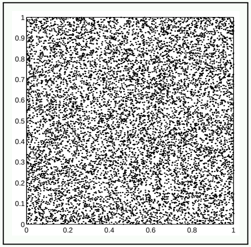
In practice, **software must have its own source-code implementation of the generators**: vendor-provided ones do *not* suffice.

The correlation problem

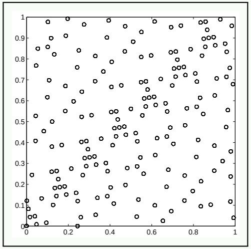
Random numbers fall mainly in the planes
— *George Marsaglia (1968)*

Linear-congruential generators are known to have correlation of successive numbers: if these are used as coordinates in a graph, one gets patterns, instead of uniform grey:

Good



Bad



The number of points plotted is the same in each graph.

The correlation problem . . .

The good generator is Matlab's `rand()`. Here is the bad generator:

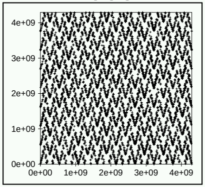
```
% hoc
func badran() {
    global A, C, M, r;
    r = int(A*r + C) % M;
    return r }
M = 2^15 - 1; A = 2^7 - 1 ; C = 2^5 - 1
r = 0 ; r0 = r ; s = -1 ; period = 0

while (s != r0) {period++; s = badran(); print s, "" }
    31 3968 12462 9889 10788 26660 ... 22258 8835 7998 0

# Show the sequence period
println period
    175

# Show that the sequence repeats
for (k = 1; k <= 5; ++k) print badran(),""
    31 3968 12462 9889 10788
```

The correlation problem . . .

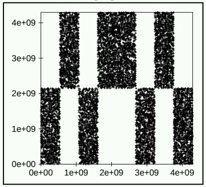
Marsaglia's (2003) family of xor-shift generators:

```
y ^= y << a; y ^= y >> b; y ^= y << c;
```

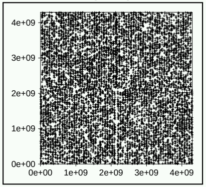
I-003



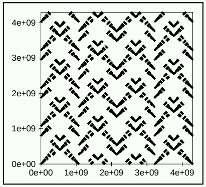
I-007



I-028



I-077



Generating random integers

When the endpoints of a floating-point uniform pseudo-random number generator are uncertain, generate random integers in [low,high] like this:

```
func irand(low, high) \
{
    # Ensure integer endpoints
    low = int(low)
    high = int(high)

    # Sanity check on argument order
    if (low >= high) return (low)

    # Find a value in the required range
    n = low - 1
    while ((n < low) || (high < n)) \
        n = low + int(rand() * (high + 1 - low))

    return (n)
}


for (k = 1; k <= 20; ++k) print irand(-9,9), ""
-9 -2 -2 -7 7 9 -3 0 4 8 -3 -9 4 7 -7 8 -3 -4 8 -4

for (k = 1; k <= 20; ++k) print irand(0, 10^6), ""
986598 580968 627992 379949 700143 734615 361237
322631 116247 369376 509615 734421 321400 876989
940425 139472 255449 394759 113286 95688
```

Generating random integers in order

```
% hoc
func bigrand() { return int(2^31 * rand()) }

# select(m,n): select m pseudo-random integers from (0,n) in order
proc select(m,n) \
{
    mleft = m
    remaining = n
    for (i = 0; i < n; ++i) \
    {
        if (int(bigrand() % remaining) < mleft) \
        {
            print i, ""
            mleft--
        }
        remaining--
    }
    println ""
}
```

See Chapter 12 of Jon Bentley, *Programming Pearls*, 2nd ed., Addison-Wesley (2000), ISBN 0-201-65788-0. [ACM TOMS **6**(3), 359–364, September 1980].

Generating random integers in order . . .

Here is how the `select()` function works:

```
select(3,10)
5 6 7

select(3,10)
0 7 8

select(3,10)
2 5 6

select(3,10)
1 5 7

select(10,100000)
7355 20672 23457 29273 33145 37562 72316 84442 88329 97929

select(10,100000)
401 8336 41917 43487 44793 56923 61443 90474 92112 92799
```

Improving a bad generator

The *n*-element shuffle generator (Bays & Durham, 1976) increases the period dramatically ($P_{\mathrm{shuffle}} \approx \sqrt{\pi n!/(2P_{\mathrm{gen}})}$), and further randomizes results:

```
unsigned long int shuffle_gen(void)
{
    int k; static int do_init = 1; static unsigned long int s;
    static unsigned long int buffer[MAXBUF + 1];
    if (do_init)
    {
        for (k = 0; k <= MAXBUF; ++k)
            buffer[k] = RAND();
        s = RAND(); do_init = 0;
    }
    k = (s >> 20) & 0xff;       /* NB: assumes MAXBUF == 256 */
    s = buffer[k];
    buffer[k] = RAND();
    return (s);
}
```

Testing pseudo-random number generators

Most tests are based on computing a χ^2 measure of computed and theoretical values.

If one gets values $p < 1\%$ or $p > 99\%$ for several tests, the generator is suspect.

Several test packages are publicly available:

- ❏ Marsaglia Diehard Battery test suite (1985): 15 tests.
- ❏ Marsaglia/Tsang `tuftest` suite (2002): 3 tests.
- ❏ Brown Dieharder suite (2004). 75+ tests.
- ❏ L'Ecuyer/Simard `TestU01` suite (2007).
- ❏ NIST special publication 800-22rev1a (2010).

All produce p values that can be checked for reasonableness.

Those tests all expect *uniformly-distributed* pseudo-random numbers.

Testing nonuniform pseudo-random number generators

How do you test a generator that produces pseudo-random numbers in some other distribution? You have to figure out a way to use those values to produce an expected uniform distribution that can be fed into the standard test programs.

For example, take the negative log of exponentially-distributed values, since $-\log(\exp(-\text{random})) = \text{random}$.

For normal distributions, consider successive pairs (x, y) as a 2-dimensional vector, and express in polar form (r, θ): θ is then uniformly distributed in $[0, 2\pi)$, and $\theta/(2\pi)$ is in $[0, 1)$.

The Marsaglia/Tsang `tuftest` tests

Just three tests instead of the fifteen of the Diehard suite:

- ❏ b'day test (generalization of Birthday Paradox).
- ❏ Euclid's (ca. 330–225BC) gcd test.
- ❏ Gorilla test (generalization of monkey's typing random streams of characters).

Digression: The Birthday Paradox

The *birthday paradox* arises from the question **How many people do you need in a room before the probability is at least half that two of them share a birthday?**

The answer is just 23, not $365/2 = 182.5$.

The probability that *none* of n people is born on the same day is

$$\begin{aligned} P(1) &= 1 \\ P(n) &= P(n-1) \times (365 - (n-1))/365 \end{aligned}$$

The n-th person has a choice of $365 - (n-1)$ days to not share a birthday with any of the previous ones. Thus, $(365 - (n-1))/365$ is the probability that the n-th person is not born on the same day as any of the previous ones, assuming that they are born on different days.

Digression: The Birthday Paradox …

Here are the probabilities that n people *share* a birthday (i.e., $1 - P(n)$):

```
% hoc128
PREC = 3
p = 1
for (n = 1;n <= 365;++n) \
    {p *= (365-(n-1))/365; println n,1-p}
1 0
2 0.00274
3 0.00820
4 0.0164
...
22 0.476
23 0.507
24 0.538
...
100 0.999999693
...
```

$P(365) \approx 1.45 \times 10^{-157}$ [cf. 10^{80} particles in universe].

Digression: Euclid's algorithm (ca. 300BC)

This is the oldest surviving nontrivial algorithm in mathematics.

```
func gcd(x,y) \
{ ## greatest common denominator of integer x, y
  r = abs(x) % abs(y)
  if (r == 0) return abs(y) else return gcd(y, r)
}

func lcm(x,y) \
{ ## least common multiple of integer x,y
  x = int(x)
  y = int(y)
  if ((x == 0) || (y == 0)) return (0)
  return ((x * y)/gcd(x,y))
}
```

Digression: Euclid's algorithm . . .

Complete rigorous analysis of Euclid's algorithm was not achieved until 1970–1990!

The average number of steps is

$$\begin{aligned} A\left(\gcd(x, y)\right) &\approx \left((12 \ln 2) / \pi^2\right) \ln y \\ &\approx 1.9405 \log_{10} y \end{aligned}$$

and the maximum number is

$$\begin{aligned} M\left(\gcd(x, y)\right) &= \left\lfloor \log_\phi \left((3 - \phi) y\right) \right\rfloor \\ &\approx 4.785 \log_{10} y + 0.6723 \end{aligned}$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.6180$ is the golden ratio.

Hardware generators

When reproducibility is not needed (e.g., for random cryptographic keys), fast hardware generators are of interest:

- Cryptographic accelerator boards (IBM, Sun/Oracle, . . .)
- Quantis PCI card (2004) (`http://www.randomnumbers.info/`)
  863 328 405 985 310 188 300 795 5 886
  84 210 411 664 264 438 221 561 756 152
  617 652 112 316 551 102 682 2 851 425
- New Intel hardware: `RdRand` for 16-, 32-, and 64-bit random values, at a rate only about 15$\times$ slower than integer addition [IEEE Spectrum September 2011]

Operating system random-number generators

Many flavors of Unix provide two pseudo-hardware devices to generate unreproducible random-byte sequences from a kernel data pool that mixes various sources of randomness (disk activity, machine load, network load, temperature, voltage, ...):

- `/dev/random`: best choice for maximal randomness, but blocks output until sufficient entropy is available (seconds, minutes, hours, ...)
- `/dev/urandom`: nonblocking, but with possible degradation of randomness

The two devices produce random *8-bit bytes*, not characters, so their output is not directly printable without further processing.

Operating system random-number generators . . .

Here is a Unix shell session producing random bytes in hexadecimal:

```
$ alias rng="dd ibs=1 count=16 if=/dev/random 2>/dev/null |
             od -t x1 |
             cut  -d ' ' -f 2- -s"
$ for f in `seq 1 10` ; do rng ; done
  01 d8 c1 62 ff 31 d3 96 d5 31 6c 9c ed d6 79 4d
  c2 7e 56 38 bc 96 16 08 df 0c 29 bb 2c 25 7e 29
  7b 6b c4 f0 b0 5a b9 ec 39 45 eb ab 13 8c 7a 8d
  47 b3 49 34 57 09 4c 90 e2 b1 9a 9c 9b b5 c9 e8
  8a 71 9a 7c 3e c6 ce 4e 3b 2c 04 32 04 4f 35 f8
  3b e7 42 ce 5f 05 79 2f 12 db 6b e5 fd 52 0a 13
  bf f5 fe c3 43 8f 15 a4 a6 2f d7 63 58 ab 00 80
  fb 5f 37 95 00 d7 7e 5c e8 43 b4 1a e4 80 e8 04
  47 62 9d fa 60 31 23 d0 4d d7 76 7b b5 44 56 05
  29 10 03 bf 2b ba b9 3a 43 57 45 94 e7 14 c2 5e
```

Key to the PRNG literature

There is a large bibliography about the generation and testing of pseudo-random numbers here:

`http://www.math.utah.edu/pub/tex/bib/prng.bib`
`http://www.math.utah.edu/pub/tex/bib/prng.html`