1 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 $O(N^2)$ problem to (on average) $O(N \lg(N))$. Cf. Fast Fourier Transform (Gauss 1866 (Latin), Runge 1906, Danielson and Lanczos (crystallography) 1942, Cooley-Tukey 1965). See Figure 1.
- Simulation.
- Sampling: unbiased selection of random data in statistical computations (opinion polls, experimental measurements, voting, Monte Carlo integra-
Figure 1: 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 this 1200-year anniversary Soviet Union stamp.

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

The latter is done like this:

Here is an example of a simple, smooth, and exactly integrable function, and the relative error of its Monte Carlo integration.
2 One-time pad encryption

% hoc -q crypto.hoc
*******************************************************************************
*******************************************************************************
** Demonstration of a simple one-time pad symmetric-key encryption algorithm **
*******************************************************************************
*******************************************************************************
------------------------------------------------------------------------
The encryption does not reveal message length, although it DOES reveal common plaintext prefixes:

encrypt(123,"A")

2b04aa0f ef15ce59 654a0dc6 ba409618 daef6924 5729580b af3af319 f579b0bc

encrypt(123,"AB")

2b47315b 22fdc9f1 b90d4fdb 1eb8302a 4944eddb e7dd1bff 8d0d1f10 1e46b93c

encrypt(123,"ABC")

2b47752c 286a4724 40bf188f c08caffa 1007d4cc 2c2495f9 cd999566 abfe0c2d

encrypt(123,"ABCD")

2b477571 f970b4a2 7346ca58 742e8379 e0ce97b3 1d69dc73 c7d921dc 018bc480

------------------------------------------------------------------------
The encryption does not reveal letter repetitions:

encrypt(123,"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA")

2b46736e 3b83cd28 777d88c8 ad1b12dc c28010ef 407d3513 e1ed75bc 5737fd71 6e68fb7d 4ac31248 94f21f9f d009455f 6d299f

------------------------------------------------------------------------
Now encrypt a famous message from American revolutionary history:

ciphertext = encrypt(123, \

"One if by land, two if by sea: Paul Revere’s Ride, 16 April 1775")
println ciphertext

3973974d 63a8ac49 af5cb3e8 da3efdbb f5b63ece 68a21434 19cca7e0 7730dc80
8e9c265c 5be7476c c51605d1 af1a6d82 9114c057 620da15b 0670bb1d 3c95c30b
ed

Attempt to decrypt the ciphertext with a nearby key. Decryption DOES reveal the message length, although that flaw could easily be fixed:
decrypt(122, ciphertext)

Attempt to decrypt the ciphertext with the correct key:
decrypt(123, ciphertext)
One if by land, two if by sea: Paul Revere’s Ride, 16 April 1775

Attempt to decrypt the ciphertext with another nearby key:
decrypt(124, ciphertext)

% cat encrypt.hoc
### -*-hoc-*-
### ====================================================================
### Demonstrate a simple one-time-pad encryption based on a
### pseudo-random number generator.
### [23-Jul-2002]
### ====================================================================
### Usage: encrypt(key,plaintext)
### The returned string is an encrypted text stream: the ciphertext.
func encrypt(key,plaintext) 
{  
    plaintext = (plaintext char(255)) # add message terminator
    while (length(plaintext) < 32) 
        plaintext = (plaintext char(randint(1,255))) # pad to 32*n characters
    setrand(key) # restart the generator
    n = 0
    ciphertext = "\n\t"
for (k = 1; k <= length(plaintext); ++k) {
    # Output 32-character lines in 4 chunks of 8 characters each
    if ((n > 0) && (n % 32 == 0))
        ciphertext = ciphertext "\n\t"
    else if ((n > 0) && (n % 4 == 0))
        ciphertext = ciphertext " 
    ciphertext = sprintf "%s%02x", ciphertext, 
        ((ichar(substr(plaintext,k,1)) + randint(0,255)) % 256)
    n++
}
ciphertext = ciphertext "\n"
return (ciphertext)

% cat decrypt.hoc
### -*-hoc-*- 
### ==============================================================
### Demonstrate a simple one-time-pad decryption based on a
### pseudo-random number generator.
### [23-Jul-2002]
### ==============================================================

### Usage: isprint(c)
### Return 1 if c is printable, and 0 otherwise.
func isprint(c) {
    return ((c == 9) || (c == 10) || ((32 <= c) && (c < 127)))
}

__hex_decrypt = "0123456789abcdef"

### Usage: decrypt(key,ciphertext)
### Return the decryption of ciphertext, which will be the original
### plaintext message if the key is correct.
func decrypt(key,ciphertext) {
    global __hex_decrypt
    setrand(key)
    plaintext = ""
    for (k = 1; k < length(ciphertext); k++) {
        n = index(__hex_decrypt,substr(ciphertext,k,1))
        if (n > 0) \

{  # have hex digit: decode hex pair
    k++
    c = 16 * (n - 1) + index(\_\_\_hex_decrypt,\_\_\_substr(ciphertext,k,1)) - 1
    n = int((c + 256 - randint(0,255)) % 256)  # recover plaintext char
    if (n == 255) {
        break;
    } else if (!isprint(n)) {
        n = ichar(?)  # mask unprintable characters
        plaintext = plaintext char(n)
    }
}
return (plaintext)

3 When is a sequence of numbers random?

If the numbers are not random, they are at least higgledy-piggledy.
— George Marsaglia (1984)

- Observation: all finite computer numbers (both fixed and floating-point)
  are rational and of limited precision and range: irrational and transcendental numbers are not represented.

- Most pseudo-random number generators produce a long sequence, called the period, of distinct integers: truly random integers would have occasional repetitions. Thus, any computer-generated sequence that has no repetitions is strictly not random.

- It isn’t enough to conform to an expected distribution: the order that values appear in must be haphazard. This means that simple tests of moments (called mean, variance, skewness, kurtosis, ... in statistics) are inadequate, because they examine each value in isolation: tests are needed to examine the sequence itself for chaos.


- The best that we can usually do is compute statistical measures of closeness to particular expected distributions. We examine a particularly-useful measure in Section 5.

4 Distributions of pseudo-random numbers

- Uniform (most common).
- **Uniform Distribution**

- **Exponential**

- **Normal (bell-shaped curve)** (see Section 7).

- **Logarithmic**: if ran() is uniformly-distributed in \((a, b)\), define randl(x) = \(\exp(x \cdot \text{ran})\). Then \(a \cdot \text{randl}(\ln(b/a))\) is logarithmically distributed in \((a, b)\).

```
% hoc
a = 1
b = 1000000
```
5 Goodness of fit: the $\chi^2$ 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 by Pearson (1900):

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

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

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

The theoretical $\chi^2$ distribution depends on the number of degrees of freedom, and table entries look like this (boxed entries are referred to later):
This says that, e.g., for $\nu = 10$, the probability that the $\chi^2$ measure is no larger than 23.21 is 99%.

For example, coin toss has $\nu = 1$: if it is not heads, then it must be tails.

```c
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:

$$\chi^2 = \frac{(10 \times 0.5 - 4)^2 + (10 \times 0.5 - 6)^2}{10 \times 0.5} = \frac{2}{5} = 0.40$$

From the table, we expect a $\chi^2$ 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

$$\chi^2 = \frac{(10 \times 0.5 - 9)^2 + (10 \times 0.5 - 1)^2}{10 \times 0.5} = \frac{32}{5} = 6.4$$

This is close to the tabulated value 6.635 at $p = 0.99$. 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 $\chi^2$ measure is 10 (probability $p = 0.998$).

If we had equal numbers of 1s and 0s, then the $\chi^2$ measure is 0, indicating an exact fit.

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

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

The measured frequencies of the sums are:
Notice that nine-of-a-kind occurred once each for 0s and 1s, as predicted. A simple one-character change on the outer loop limit produces the next experiment:

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Y_k)</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Another one-character change gives us this:

| 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 |
| \(Y_k\) | 1 | 2 | 3 | 3 | 8 | 7 | 6 | 4 | 9 | 1 | 3 | 2 | 2 | 9 | 4 | 3 | 4 | 6 | 4 | 9 | 9 | 3 | 1 | 1 | 8 | 0 | 7 | 6 | 1 | 1 | 0 |

Another one-character change gives us this:

| 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 | 2 | 7 | 0 | 5 | 9 | 8 | 4 | 5 | 0 | 4 | 8 | 3 | 6 | 9 | 4 | 5 | 6 | 8 | 9 | 0 | 3 | 8 | 7 | 0 | 3 | 0 | 8 | 0 | 5 | 2 | 8 | 4 | 1 | 0 | 0 |

100 experiments

<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Y_k)</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
A final one-character change gives us this result for one million coin tosses:

100,000 experiments

<table>
<thead>
<tr>
<th>k</th>
<th>1 1 2 3 3 4 5 6 7 8 8 7 7 6 5 4 3 3 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 4 8 0 6 2 1 9 8 6 5 3 1 2 8 1 6 6 7 9 0 2 5 9 6 4 2 1 1</td>
</tr>
<tr>
<td>Y_k</td>
<td>1 3 4 7 1 4 1 0 8 3 5 1 8 8 0 8 2 1 2 2 7 0 0 4 6 2 1 4 9 5 7 5 4 0 4 3 2</td>
</tr>
</tbody>
</table>

In the magazine Science 84 (November 1984), the chi-square test was ranked among the top twenty scientific discoveries of the 20th Century that changed our lives:

| 1900–1919 | 1 Bakelite | 11 Fission |
| 2 IQ test  | 12 Red Shift |
| 3 Non-Newtonian physics | 13 DDT |
| 4 Blood groups | 14 Television |
| 5 Chi-square test | 1940–1959 |
| 6 Vacuum tubes | 15 Birth control pills |
| 7 Hybrid corn | 16 Colossus and Eniac |
| 8 Airfoil theory | 17 Psychoactive drugs |
| 9 Antibiotics | 18 Transistor |
| 10 Taung skull | 19 Double helix |
| 11 | 20 Masers and lasers |

6 Randomness of digits of π

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

<table>
<thead>
<tr>
<th>Digits</th>
<th>Base</th>
<th>$\chi^2$</th>
<th>$P(\chi^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6B</td>
<td>10</td>
<td>9.00</td>
<td>0.56</td>
</tr>
<tr>
<td>50B</td>
<td>10</td>
<td>5.60</td>
<td>0.22</td>
</tr>
<tr>
<td>200B</td>
<td>10</td>
<td>8.09</td>
<td>0.47</td>
</tr>
<tr>
<td>1T</td>
<td>10</td>
<td>14.97</td>
<td>0.91</td>
</tr>
<tr>
<td>1T</td>
<td>16</td>
<td>7.94</td>
<td>0.46</td>
</tr>
</tbody>
</table>
Whether the fractional digits of \( \pi \), and most other transcendentals, are normal \((\approx\) equally likely to occur) is an outstanding unsolved problem in mathematics.

De Morgan suspected that Shanks’ 1872–73 computation of \( \pi \) to 707 decimal digits was wrong because the frequency of the digit 7 was low. De Morgan was right, but it took a computer calculation by Ferguson in 1946 to show the error at Shanks’ digit 528.

7 The Central-Limit Theorem

The normal law of error stands out in the experience of mankind as one of the broadest generalizations of natural philosophy. It serves as the guiding instrument in researches in the physical and social sciences and in medicine agriculture and engineering. It is an indispensable tool for the analysis and the interpretation of the basic data obtained by observation and experiment.

— W. J. Youdon (1956)

(from Stephen M. Stigler, Statistics on the Table (1999), p. 415)

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.

In mathematical terms, this is

<table>
<thead>
<tr>
<th>Digits</th>
<th>Base</th>
<th>( \chi^2 )</th>
<th>( P(\chi^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>6B</td>
<td>10</td>
<td>5.44</td>
<td>0.21</td>
</tr>
<tr>
<td>50B</td>
<td>10</td>
<td>7.04</td>
<td>0.37</td>
</tr>
<tr>
<td>200B</td>
<td>10</td>
<td>4.18</td>
<td>0.10</td>
</tr>
</tbody>
</table>
\[ P(n\mu + r_1\sqrt{n} \leq X_1 + X_2 + \cdots + X_n \leq n\mu + r_2\sqrt{n}) \approx \frac{1}{\sigma\sqrt{2\pi}} \int_{r_1}^{r_2} \exp(-t^2/(2\sigma^2))dt \]

where the \(X_k\) are independent, identically distributed, and bounded random variables, \(\mu\) is their mean value, and \(\sigma^2\) is their variance (not further defined here).

The integrand of this probability function looks like this:

![The Normal Distribution](image)

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

```plaintext
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; s = 0;
    for (k = 0; k < n; ++k) s += h*f(a + (k + 0.5)*h);
    return s}

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
In computers, 99.999% (five 9’s) availability is 5 minutes downtime per year. In manufacturing, Motorola’s $6\sigma$ reliability with $1.5\sigma$ drift is about 3.4 defects per million (from $q(4.5 \times \sigma)/2$).

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 $550 \exp(-(x-13.1)^2/2.3)$.

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 $[-1.0, +1.0]$:

| $k$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |

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

Here is another type, the Poisson distribution, which arises in time series when the probability of an event occurring in an arbitrary interval is propor-
tional 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.

<table>
<thead>
<tr>
<th>Deaths</th>
<th>Kicks (actual)</th>
<th>Kicks (Poisson)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>109</td>
<td>108.7</td>
</tr>
<tr>
<td>1</td>
<td>65</td>
<td>66.3</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>20.2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4.1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Measurements of physical phenomena often form normal distributions:
8 How do we generate pseudo-random numbers?

Any one who considers arithmetical methods of producing random numbers is, of course, in a state of sin.
— John von Neumann (1951)

He talks at random; sure, the man is mad.
— Margaret, daughter to Reignier, afterwards married to King Henry in 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
[The Art of Computer Programming, Vol. 2,
Seminumerical Algorithms, 3rd ed., p. 384

- Linear-congruential generators (most common): \( r_{n+1} = (ar_n + c) \mod m \), for integers \( a, c \), and \( m \), where \( 0 < m, 0 \leq a < m, 0 \leq c < m \), with starting value \( 0 \leq r_0 < m \). Under certain known conditions, the period can be as large as \( m \), unless \( c = 0 \), when it is limited to \( m/4 \).

- Fibonacci sequence (bad!): \( r_{n+1} = (r_n + r_{n-1}) \mod m \).

- Additive (better): \( r_{n+1} = (r_n - \alpha + r_{n-\beta}) \mod m \).

- Multiplicative (bad): \( r_{n+1} = (r_n - \alpha \times r_{n-\beta}) \mod m \).

- Shift register: \( r_{n+k} = \sum_{i=0}^{k-1} (a_ir_{n+i} \mod 2) \) \( (a_i = 0, 1) \).

Given an integer \( r \in [A, B) \), \( x = (r - A) / (B - A + 1) \) is on \([0, 1)\). However, interval reduction by \( A + (r - A) \mod s \) 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 \( m = 2^q - 1 \):

<table>
<thead>
<tr>
<th>( m )</th>
<th>counts of remainders ( k \mod (m + 1), \ k \in [0, m] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK 1</td>
<td>2048 2048</td>
</tr>
<tr>
<td>2</td>
<td>1366 1365 1365</td>
</tr>
<tr>
<td>OK 3</td>
<td>1024 1024 1024 1024</td>
</tr>
<tr>
<td>4</td>
<td>820 819 819 819</td>
</tr>
<tr>
<td>5</td>
<td>683 683 683 683 683 682 682</td>
</tr>
<tr>
<td>6</td>
<td>586 585 585 585 585 585 585 585</td>
</tr>
<tr>
<td>OK 7</td>
<td>512 512 512 512 512 512 512 512 512</td>
</tr>
<tr>
<td>8</td>
<td>456 455 455 455 455 455 455 455 455 455 455 455 455 455</td>
</tr>
<tr>
<td>9</td>
<td>410 410 410 410 410 410 410 410 410 410 410 410 410 410 409 409 409 409 409</td>
</tr>
</tbody>
</table>

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

```c
func randpmnd() {
    ## Polar method for random deviates
    ## Algorithm P, p. 122, from Donald E. Knuth, The Art

    while (1) {
        v1 = 2*randu() - 1
        v2 = 2*randu() - 1
        s = v1*v1 + v2*v2
        if (s < 1) break
    }
}```
\}
return (v1 * sqrt(-2*ln(s)/s))

## (v2 * sqrt(-2*ln(s)/s)) is also normally distributed,
## but is wasted, since we only need one return value
\}

9 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. Good generators are now known with periods > $10^{449}$ (e.g., Matlab’s `rand()`).

10 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:

```haskell
for (k = 0; k < 3; ++k) \{
  setrand(12345)
  for (n = 0; n < 10; ++n) print int(rand()*100000), ""
  println ""
}\n
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

for (k = 0; k < 3; ++k) \{
  # setrand(12345)
  for (n = 0; n < 10; ++n) print int(rand()*100000), ""
  println ""
}\n
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, this means that software must have its own source-code implementation of the generators: vendor-provided ones do not suffice.
11 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. The number of points plotted in each is the same in both graphs:

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

12 The correlation problem [cont.]

Marsaglia’s (Xorshift RNGs, J. Stat. Software 8(14) 1–6, 2003) family of generators:
y ^= y << a; y ^= y >> b; y ^= y << c;

13 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

14 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--
    }
}
15 Testing pseudo-random number generators

Most of the tests of pseudo-random number distributions are based on computing a $\chi^2$ measure of computed and theoretical values. If one gets values $p < 1\%$ or $p > 99\%$ for several tests, the generator is suspect.

Knuth devotes about 100 pages to the problem of testing pseudo-random number generators. Unfortunately, most of the easily-implemented tests do not distinguish good generators from bad ones. The better tests are much harder to implement.

The Marsaglia Diehard Battery test suite (1985) has 15 tests that can be applied to files containing binary streams of pseudo-random numbers. The Marsaglia/Tsang tuftest suite (2002) has only three, and requires only functions, not files, but a pass is believed (empirically) to imply a pass of the Diehard suite. All of these tests produce $p$ values that can be checked for reasonableness.

These tests all expect uniformly-distributed pseudo-random numbers. 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 succes-
sive pairs \((x, y)\) as a 2-dimensional vector, and express in polar form \((r, \theta)\): \(\theta\) is then uniformly distributed in \([0, 2\pi)\), and \(\theta / (2\pi)\) is in \([0, 1)\).

16 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 surprises most people: it is just 23, not \(365/2 = 182.5\).

The probability that none of \(n\) people are born on the same day is

\[
P(1) = 1 \\
P(n) = P(n-1) \times (365 - (n - 1)) / 365
\]

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.

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) \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
... 30 0.706
... 40 0.891
... 50 0.970
... 70 0.999
... 80 0.9999
... 90 0.999994
...```
The Marsaglia/Tsang tuftest tests

The first tuftest test is the b'day test, a generalization of the Birthday Paradox to a much longer year. Here are two reports for it:

Good generator

Birthday spacings test: 4096 birthdays, 2^32 days in year

<table>
<thead>
<tr>
<th>Duplicates</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>&gt;=10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>91.6</td>
<td>366.3</td>
<td>732.6</td>
<td>976.8</td>
<td>976.8</td>
<td>781.5</td>
<td>521.0</td>
<td>297.7</td>
<td>148.9</td>
<td>66.2</td>
<td>40.7</td>
</tr>
<tr>
<td>Observed</td>
<td>87</td>
<td>385</td>
<td>748</td>
<td>962</td>
<td>975</td>
<td>813</td>
<td>472</td>
<td>308</td>
<td>159</td>
<td>61</td>
<td>30</td>
</tr>
</tbody>
</table>

(0-E)^2/E 0.2 1.0 0.3 0.2 0.0 1.3 4.6 0.4 0.7 0.4 2.8

Birthday Spacings: Sum((0-E)^2/E)= 11.856, p= 0.705

[Last two results taken from 300-digit computation in Maple.]
Bad generator

Birthday spacings test: 4096 birthdays, 2^32 days in year
Table of Expected vs. Observed counts:

<table>
<thead>
<tr>
<th>Duplicates</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>&gt;=10</th>
</tr>
</thead>
</table>

| Expected   | 91.6 | 366.3 | 732.6 | 976.8 | 976.8 | 781.5 | 521.0 | 297.7 | 148.9 | 66.2 | 40.7  |
| Observed   | 0    | 0    | 0    | 0    | 1    | 3    | 18    | 53    | 82    | 144   | 4699  |

(O-E)^2/E 91.6 366.3 732.6 976.8 974.8 775.5 485.6 201.1 30.0 91.6 533681.1
Birthday Spacings: Sum(O-E)^2/E=538407.147, p= 1.000

The second test is based on the number of steps to find the greatest common denominator by Euclid’s (ca. 330–225BC) algorithm (the world’s oldest surviving nontrivial algorithm in mathematics), and on the expected distribution of the partial quotients.

```plaintext
func gcd(x,y) 
{ 
    rem = abs(x) % abs(y)
    if (rem == 0) return abs(y) else return gcd(y, rem)
}

proc gcdshow(x,y) 
{ 
    rem = abs(x) % abs(y)
    println x, "=", int(x/y), "\times", y, "\times", rem
    if (rem == 0) return
    gcdshow(y, rem)
}

gcd(366,297)
3

gcdshow(366,297)
366 = 1 \times 297 + 69
297 = 4 \times 69 + 21
69 = 3 \times 21 + 6
21 = 3 \times 6 + 3
6 = 2 \times 3 + 0

This took k = 5 iterations, and found partial quotients (1, 4, 3, 3, 2).

Interestingly, the complete rigorous analysis of the number of steps required in Euclid’s algorithm was not achieved until 1970–1990! The average number is

\[ A(gcd(x,y)) \approx \left( \frac{(12 \ln 2)}{\pi^2} \right) \ln y \]
\[ \approx 1.9405 \log_{10} y \]

and the maximum number is
\[ M(\gcd(x, y)) = \lfloor \log_\phi ((3 - \phi)y) \rfloor \approx 4.785 \log_{10} y + 0.6723 \]
where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.6180 \) is the golden ratio. For our example above, we find
\[ A(\gcd(366, 297)) \approx 4.798 \]
\[ M(\gcd(366, 297)) \approx 12.50 \]

Here are two tuftest reports:

**Good generator**

Euclid's algorithm:
- p-value, steps to gcd: 0.452886
- p-value, dist. of gcd's: 0.751558

**Bad generator**

Euclid's algorithm:
- p-value, steps to gcd: 1.000000
- p-value, dist. of gcd's: 1.000000

The third tuftest test is a generalization of the monkey test: a monkey typing randomly produces a stream of characters, some of which eventually form words, sentences, paragraphs, ....

**Good generator**

Gorilla test for 2^{26} bits, positions 0 to 31:
Note: lengthy test---for example, ~20 minutes for 850MHz PC
Bits 0 to 7----> 0.797 0.480 0.096 0.660 0.102 0.071 0.811 0.831
Bits 8 to 15----> 0.731 0.110 0.713 0.624 0.019 0.405 0.664 0.892
Bits 16 to 23----> 0.311 0.463 0.251 0.670 0.854 0.414 0.221 0.563
Bits 24 to 31----> 0.613 0.562 0.191 0.830 0.284 0.752 0.739 0.356
KS test for the above 32 p values: 0.289

**Bad generator**

Gorilla test for 2^{26} bits, positions 0 to 31:
Note: lengthy test---for example, ~20 minutes for 850MHz PC
Bits 0 to 7----> 0.000 0.000 0.000 0.000 0.000 1.000 1.000 1.000
Bits 8 to 15----> 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
Bits 16 to 23----> 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
Bits 24 to 31----> 1.000 1.000 1.000 1.000 1.000 1.000 1.000 1.000
KS test for the above 32 p values: 1.000
18 Further reading


Douglas Lehmer and George Marsaglia have probably written more technical papers on the subject than anyone else: look for them with bibsearch and in the MathSciNet database.

Marsaglia’s *Diehard Battery* test suite is available at:

http://www.stat.fsu.edu/pub/diehard/


http://www.jstatsoft.org/v07/i03/tufests.pdf
http://www.jstatsoft.org/v07/i03/tufests.c

For a history of the Central-Limit Theorem, see


For a real-time demonstration of the Central-Limit Theorem based on balls threading through a grid of pins, visit

http://www.rand.org/methodology/stat/applets/clt.html

For another live demonstration based on dice throws, visit

http://www.math.csusb.edu/faculty/stanton/probstat/clt.html


http://www.cryptoengines.com/~peter/06_random.pdf