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

20 September 2017
What are random numbers good for?

- Decision making (e.g., coin flip).
- Generation of numerical test data and fuzz testing.
- 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 Fast Fourier Transform (Clairaut (1754), Lagrange (1759), Gauss (1805 unpublished, 1866) [Latin], Runge (1903), Danielson and Lanczos [crystallography] (1942), Cooley and Tukey (1965)).
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:
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 $(a, b)$):

$$\int_a^b f(x) \, dx \approx \left( \frac{b - a}{N} \sum_{k=1}^{N} f(x_k) \right) + O\left(\frac{1}{\sqrt{N}}\right)$$
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 \( \text{ran}() \) is uniformly-distributed in \((a, b)\), define \( \text{randl}(x) = \exp(x \cdot \text{ran}()) \). Then \( a \cdot \text{randl}(\ln(b/a)) \) 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
```
Here are three ways to visualize a pseudo-random number distribution, using the Dyadkin-Hamilton generator function \( rn01() \), which produces results uniformly distributed on \((0, 1]\):

![Uniform Distribution](image1)

![Uniform Distribution](image2)

![Uniform Distribution Histogram](image3)
The Swiss artist Ursus Wehrli dislikes randomness:
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 $[0, \infty)$, results are practically always modest: the probability of a result as big as 50 is smaller than $2 \times 10^{-22}$. At one result per microsecond, it could take 164 million years of computing to encounter such a value!
Here are visualizations of computations with the Dyadkin-Hamilton generator \texttt{rnnorm()}, which produces results normally distributed on \((−\infty, +\infty)\):

Results are \textit{almost never} very large: a result as big as 7 occurs with probability smaller than \(5 \times 10^{-23}\). 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 \texttt{hoc} generator \texttt{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 \texttt{randl()}. 

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 in 1900 by one of the founders of modern statistics, the English mathematician Karl Pearson (1857–1936):

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

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

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

(1880)
“There were essentially only two editors of *Biometrika* in the first 65 years and only three in the first 90 years (Karl Pearson [27 March 1857–27 April 1936], Egon Sharpe Pearson [11 August 1895–12 June 1980], and Sir David Roxbee Cox [5 July 1924–]). . . . *Biometrika* was founded for the statistical study of biological problems and it was still the leading journal for biometric anthropology in 1936. Though it established a niche in this specialized branch of biology, it did not realize the hopes of its founders and the real importance of K. P.’s *Biometrika* was in its role in establishing mathematical statistics as a discipline.”

John Aldrich

*Karl Pearson’s Biometrika: 1901–36*

*Biometrika*, **100**(1) 3–15, March 2013

http://dx.doi.org/10.1093/biomet/ass077
Goodness of fit: the $\chi^2$ measure ...

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

<table>
<thead>
<tr>
<th>D.o.f.</th>
<th>$p = 1%$</th>
<th>$p = 5%$</th>
<th>$p = 25%$</th>
<th>$p = 50%$</th>
<th>$p = 75%$</th>
<th>$p = 95%$</th>
<th>$p = 99%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\nu = 1$</td>
<td>0.00016</td>
<td>0.00393</td>
<td>0.1015</td>
<td><strong>0.4549</strong></td>
<td>1.323</td>
<td>3.841</td>
<td><strong>6.635</strong></td>
</tr>
<tr>
<td>$\nu = 5$</td>
<td>0.5543</td>
<td>1.1455</td>
<td>2.675</td>
<td>4.351</td>
<td>6.626</td>
<td>11.07</td>
<td>15.09</td>
</tr>
<tr>
<td>$\nu = 10$</td>
<td>2.558</td>
<td>3.940</td>
<td>6.737</td>
<td>9.342</td>
<td>12.55</td>
<td>18.31</td>
<td><strong>23.21</strong></td>
</tr>
<tr>
<td>$\nu = 50$</td>
<td>29.71</td>
<td>34.76</td>
<td>42.94</td>
<td>49.33</td>
<td>56.33</td>
<td>67.50</td>
<td>76.15</td>
</tr>
</tbody>
</table>

For example, this table says:

For $\nu = 10$, the probability that the $\chi^2$ measure is no larger than **23.21** is 99%.

In other words, $\chi^2$ measures larger than **23.21** should occur only about 1% of the time.
Coin toss has one degree of freedom, $\nu = 1$, because if it is not heads, then it must be tails.

This gave four 1s and six 0s:

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

$$= \frac{2}{5}$$

$$= 0.40$$
From the table, for $\nu = 1$, 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

$$\chi^2 \text{ measure} = \frac{(10 \times 0.5 - 9)^2 + (10 \times 0.5 - 1)^2}{10 \times 0.5}$$

$$= \frac{32}{5}$$

$$= 6.4$$

This is close to the tabulated value 6.635 at $p = 99\%$. 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 $p = 0.998$) [twice in 1000 experiments].

If we had equal numbers of 1s and 0s, then the $\chi^2$ 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
The measured frequencies of the sums are:

<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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

| $k$ | 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  | 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  |
Goodness of fit: coin-toss experiments . . .

Another one-character change gives us this:

<table>
<thead>
<tr>
<th>$k$</th>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_k$</td>
<td>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</td>
</tr>
</tbody>
</table>

![Bar chart showing the distribution of $Y_k$ values for 10,000 experiments.](image)
A final one-character change gives us this result for one million coin tosses:

<table>
<thead>
<tr>
<th>$k$</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>40</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
<th>46</th>
<th>47</th>
<th>48</th>
<th>49</th>
<th>50</th>
<th>51</th>
<th>52</th>
<th>53</th>
<th>54</th>
<th>55</th>
<th>56</th>
<th>57</th>
<th>58</th>
<th>59</th>
<th>60</th>
<th>61</th>
<th>62</th>
<th>63</th>
<th>64</th>
<th>65</th>
<th>66</th>
<th>67</th>
<th>68</th>
<th>69</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Y_k$</td>
<td>1 1 2 3 4 5 6 7 8 8 7 7 6 5 4 3 2 1 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 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>
<td>1 4 2 4 7 0 8 2 8 2 6 3 9 2 7 0 9 7 0 3 7 8 1 7 4 0 2 9 2 4 6 4 4 4 7 1 7 3 7 1 5 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Are the digits of \( \pi \) random?

Here are \( \chi^2 \) results for the digits of \( \pi \) from recent computational records

\[
\chi^2(v = 9, p = 0.99) \approx 21.67
\]

<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>

<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>

Whether the fractional digits of \( \pi \), 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 $\pi$ random?

3.14159265358979323846264338327950288419716939937510
58209749445923078164062862089986280348253421170679
82148086513282306647093844609550582231725359408128
48111745028410270193852110555964462294895493038196
44288109756659334461284756482337867831652712019091
45648566923460348610454326648213393607260249141273
72458700660631558817488152092096282925409171536436
78925903600113305305488204665213841469519415116094
33057270365759591953092186117381932611793105118548
07446237996274956735188575272489122793818301194912
98336733624406566430860213949463952247371907021798
60943702770539217176293176752384674818467669405132
00056812714526356082778577134275778960917363717872
14684409012249534301465495853710507922796892589235
42019956112129021960864034418159813629774771309960
51870721134999999837297804995105973173281609631859
50244594553469083026425223082533446850352619311881
71010003137838752886587533208381420617177669147303
59825349042875546873115956286388235378759375195778
1857780532171226806613001927876611959092164201989
Are the first 1000 fractional digits of $\pi$ random?

In the first 1000 fractional digits of $\pi$:

- 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!**


URL http://m.ajp.aapt.org/resource/1/ajpias/v59/i8/p700_s1
From http://www.dr-mikes-maths.com/pisearch.html: NELSON was not found, but I searched 31 415 929 digits of \(\pi\), and found BEEBE 4 times. The first occurrence was at position 846 052. What this means is that

\[
\pi = 3 + \ldots + \frac{B}{27278246052} + \frac{E}{27278246053} + \frac{E}{27278246054} + \frac{B}{27278246055} + \frac{E}{27278246056} + \ldots
\]

where \(A = 1\), \(B = 2\), \(C = 3\), and so on.
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

\[ P\left(n\mu + r_1\sqrt{n} \leq X_1 + X_2 + \cdots + X_n \leq n\mu + r_2\sqrt{n}\right) \approx \frac{1}{\sigma \sqrt{2\pi}} \int_{r_1}^{r_2} \exp\left(-\frac{t^2}{2\sigma^2}\right)dt \]

where the \(X_k\) are independent, identically distributed, and bounded random variables, \(\mu\) is their mean value, \(\sigma\) is their standard deviation, and \(\sigma^2\) is their variance.
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:

```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
    area = 0
    for (k = 0; k < n; ++k) {
        area += h * f(a + (k + 0.5) * h);
    }
    return area
}
```
The Central-Limit Theorem ...

\[ \sigma = 3 \]

\[
\text{for (k = 1; k < 8; ++k) \n \quad \text{printf "\%d \ %.9f\\n", k, q(-k*sigma,k*sigma) } }
\]

<table>
<thead>
<tr>
<th>k</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.682689493</td>
</tr>
<tr>
<td>2</td>
<td>0.954499737</td>
</tr>
<tr>
<td>3</td>
<td>0.997300204</td>
</tr>
<tr>
<td>4</td>
<td>0.999936658</td>
</tr>
<tr>
<td>5</td>
<td>0.999999427</td>
</tr>
<tr>
<td>6</td>
<td>0.999999998</td>
</tr>
<tr>
<td>7</td>
<td>1.000000000</td>
</tr>
</tbody>
</table>

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 \( q(-(6 - 1.5) \times \sigma, +(6 - 1.5) \times \sigma)/2 \)).
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 $650 \exp\left(-\left(x - 12.6\right)^2 / 4.7\right)$ and $550 \exp\left(-\left(x - 13.1\right)^2 / 2.3\right)$.
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 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| $Y_k$ | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
|      | 1 | 3 | 6 | 8 | 1 | 3 | 6 | 8 | 1 | 3 | 6 | 8 | 1 | 3 | 6 | 8 | 1 | 3 | 6 | 8 | 1 | 3 | 6 | 8 | 3 | 1 | 8 | 6 | 3 | 1 | 8 | 6 | 3 | 1 | 8 | 6 | 3 | 1 |
|      | 3 | 5 | 1 | 8 | 3 | 8 | 3 | 7 | 1 | 6 | 2 | 0 | 2 | 9 | 1 | 7 | 4 | 7 | 4 | 7 | 7 | 7 | 7 | 4 | 5 | 5 | 7 | 2 | 8 | 1 | 6 | 2 | 8 | 2 | 7 | 3 | 7 | 3 |

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 \( \frac{122}{200} = 0.61 \) 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>
The Central-Limit Theorem . . .

Measurements of physical phenomena often form normal distributions:

Chest girth of Scottish soldiers (1817)

Count of soldiers

Inches

Height of French soldiers (1851--1860)

Count of soldiers

Inches

Weights of 10,000 gold sovereigns (1848)

Count of coins

Grains from average

Nelson H. F. Beebe (University of Utah)
The Central-Limit Theorem...

- **Error in erf(x)**
  - Units in the last place: -1.0 to 1.0
  - x values: -5 to 5

- **Error in erf(x), x on [-5,5]**
  - σ = 0.22
  - Count of function calls: 800
  - Units in the last place: -1.0 to 1.0

- **Error in gamma(x)**
  - Units in the last place: -20 to 20
  - x values: 0 to 10

- **Error in gamma(x), x on [0..10]**
  - σ = 3.68
  - Count of function calls: 2500
  - Units in the last place: -15 to 15
The Central-Limit Theorem . . .

![Graph of Error in log(x)](image1)

![Graph of Error in sin(x)](image2)

![Graph of Error in log(x), x on (0..10)](image3)

![Graph of Error in sin(x), x on [0..2π)](image4)
The Normal Curve and Carl-Friedrich Gauß (1777–1855)
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
For simulations and other material on the quincunx (Galton’s *bean machine*), see:

- [http://teacherlink.org/content/math/interactive/flash/quincunx/quincunx.html](http://teacherlink.org/content/math/interactive/flash/quincunx/quincunx.html)
- [http://www.bun.kyoto-u.ac.jp/~suchii/quinc.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)  

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)  
How do we generate pseudo-random numbers?

- Linear-congruential generators (most common):
  \[ r_{n+1} = (ar_n + c) \mod m, \text{ for integers } a, c, \text{ and } m, \text{ where } 0 < m, 0 \leq a < m, 0 \leq c < m, \text{ with starting value } 0 \leq r_0 < m. \]

- 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:
  \[ r_{n+k} = \sum_{i=0}^{k-1} (a_i r_{n+i} \mod 2) \quad (a_i = 0, 1). \]
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 \( A + (r - A) \mod s \) 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>820 819 819 819 819</td>
</tr>
<tr>
<td>5</td>
<td>683 683 683 683 682 682</td>
</tr>
<tr>
<td>6</td>
<td>586 585 585 585 585 585 585</td>
</tr>
<tr>
<td>OK 7</td>
<td>512 512 512 512 512 512 512 512</td>
</tr>
<tr>
<td>8</td>
<td>456 455 455 455 455 455 455 455 455</td>
</tr>
<tr>
<td>9</td>
<td>410 410 410 410 410 410 409 409 409 409</td>
</tr>
</tbody>
</table>
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 \texttt{randu()} returns uniformly-distributed values on \((0, 1]\):

```plaintext
func randpmnd() {
  ## Polar method for random deviates
  ## Algorithm P, p. 122, from Donald E. Knuth,
  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))
}
```
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),
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 ""
}
```

```plaintext
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:

```c
for (k = 0; k < 3; ++k) {
    ## setrand(12345)
    for (n = 0; n < 10; ++n) print int(rand() * 100000), ""
    println ""
}
36751 37971 98416 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:

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:

\[ y \leftarrow y \ll a; \ y \leftarrow y \gg b; \ y \leftarrow y \ll c; \]

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:

```plaintext
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)
}
```

```plaintext
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
```

```plaintext
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 ""
}

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 ($P_{\text{shuffle}} \approx \sqrt{\pi n!/(2P_{\text{gen}})}$), and further randomizes results:

```c
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);
}
```
PERIOD = 2**32 - 1  # 4.29e+09
for (n = 100; n <= 1000; n += 100) \
    printf("%4d\t%.2e\n", n, sqrt(PI * factorial(n) / (2 * PERIOD)))

100 1.85e+74
200 5.37e+182
300 3.35e+302
400 4.84e+429
500 2.11e+562
600 2.15e+699
700 9.41e+839
800 5.31e+983
900 1.57e+1130
1000 1.21e+1279
Testing pseudo-random number generators

Most tests 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.**

Several test packages are publicly available:

- Marsaglia/Tsang tuftest suite (2002): 3 tests.
- 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.
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 \((r, \theta)\): \(\theta\) 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).
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 $\frac{365}{2} = 182.5$.

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

\[
P(1) = 1 \\
P(n) = P(n-1) \times \frac{365 - (n - 1)}{365}
\]

The $n$-th person has a choice of $365 - (n-1)$ days to not share a birthday with any of the previous ones. Thus, $\frac{365 - (n - 1)}{365}$ 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., \( 1 - P(n) \)):

\[
\begin{align*}
\% \ hoc128 \\
\text{PREC} &= 3 \\
p &= 1 \\
\text{for (n = 1;n <= 365;++n) } \{ \\
& \quad \{p *= (365-(n-1))/365; \text{println n,1-p}\} \\
\end{align*}
\]

1 0
2 0.00274
3 0.00820
4 0.0164
...
22 0.476
23 0.507
24 0.538
...
100 0.999999693
...

\[ P(365) \approx 1.45 \times 10^{-157} \text{ [cf. } 10^{80} \text{ particles in universe}] \]
Digression: Euclid’s algorithm (ca. 300BC)

This is the oldest surviving nontrivial algorithm in mathematics.

```plaintext
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) )
}
```
Complete rigorous analysis of Euclid’s algorithm was not achieved until 1970–1990!

The average number of steps 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

$$M(\gcd(x, y)) = \lceil \log_{\phi} ((3 - \phi)y) \rceil$$
$$\approx 4.785 \log_{10} y + 0.6723$$

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, ...)**
  
  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× slower than integer addition [IEEE Spectrum September 2011]**
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:

```bash
$ 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
c2 6b c4 f0 b0 5a b9 ec 39 45 eb ab 13 8c 7a 8d
c7 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
```
Debian/OpenSSL Fiasco

In 2006, in order to eliminate a Purify warning, a developer at Debian removed a line of code from the randomness-initializer of OpenSSL as shipped with Debian. This reduced the amount of true entropy for initializing SSH keys to approximately 15 bits, or (in some realistic scenarios) 0 bits.

– CCS 17 conference paper by Katherine Ye and others
https://doi.org/10.1145/3133956.3133974
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