Pitfalls in Computation: Random and not so Random Numbers*

Richard P. Brent
MSI & RSISE
Australian National University

9 October 2006

 NAMS06t

Abstract

Computers often give the wrong answers. This can be caused by human errors ("garbage in, garbage out"), by software errors (programming "bugs" or incorrect program specifications), or even by hardware errors. As an example of a hardware error we describe the Intel Pentium divide bug that is estimated to have cost Intel $US 475 million in 1994. Similarly expensive was the software error that caused the crash of an Ariane 5 rocket in 1996.

In simulation we often want computers to generate "random" numbers that are independent and have a known distribution. This is not so easy as it seems. We give some examples where the "random" numbers turned out to be not so random as expected, with potentially disastrous results.

To summarise, it is easy to produce random numbers when you don't want to, but not so easy when you do.

Part 1 – Getting the Wrong Answer

There are many reasons why computers sometimes give incorrect answers:

- Asked the wrong question.
- Used an unstable numerical algorithm.
- Entered the wrong data.
- Program bug.
- Compiler bug.
- Firmware/hardware bug.
- Transient error.

We'll look at some examples.

Asked the wrong question

If someone asks your advice on a numerical or statistical problem, they usually say

"I want to do XXX."

You can answer

"This is how to do XXX $\cdots$"

but it is often better to answer

"Do you *really* want to do XXX?
Wouldn't YYY be better?"

Example – Runge's phenomenon

For example, you might have some data y_j given at n equally spaced points x_j, and want to fit an approximation $P_n(x)$ such that $P(x_j) = y_j$. It's natural to consider a polynomial $P_n(x)$ of degree $n-1$. However, this is likely to be disastrous as $P_n(x)$ may oscillate violently between the data points (and this only gets worse as n increases).

Runge gave the example

$$f(x) = \frac{1}{1+25x^2}$$

on $[-1,1]$ with equally spaced points (spacing $h = 2/(n-1)$).

Unfortunately

$$\lim_{n\to\infty} ||f - P_n||_\infty = +\infty$$

More on Runge's example

The key point of Runge's example is that

$$f(z) = \frac{1}{1+25z^2}\,,$$

regarded as a function of a complex variable z, has poles in the complex plane at $z = \pm i/5$. These poles are close to the interval $[-1,1]$ on which we wish to approximate f, in fact so close that they make polynomial interpolation (at equally spaced points) very innaccurate.

Polynomial interpolation is not always bad – Weierstrass's theorem says that any continuous function can be approximated arbitrarily closely by a polynomial. Interpolation at Chebyshev points

$$x_j = \cos\left(\frac{2j-1}{2n}\,\pi\right)\,,\quad j = 1,\ldots,n$$

works in this example, However, we might not be free to choose Chebyshev points.

In practice it would almost certainly be better to use piecewise polynomials, i.e. *splines*.

Used an unstable numerical algorithm

Numerical analysts worry about the *numerical stability* of algorithms. That is, they worry about the effect of *rounding errors* on the results. Rounding errors are (usually) unavoidable if we use the computer hardware to perform "real" arithmetic – because it is only an approximation to real arithmetic.

Nowadays most computers implement the *IEEE 754* standard for binary floating-point arithmetic, which uses 32-bit or 64-bit computer words to store floating-point numbers in the form

$$(\text{sign},\ \text{exponent},\ \text{fraction})$$

The fraction is sometimes called a *mantissa* for obscure historical reasons. The fraction has 24 or 53 bits (including an implicit leading 1 bit). Thus the precision is roughly equivalent to 7 or 16 decimal digits.

Numerical stability

The most useful definition of numerical stability is what Wilkinson called "backward stability" – an algorithm is *stable* if it computes the exact solution to a nearby problem.

For example, when solving a system of linear equations

$$Ax = b$$

we don't expect the computed solution $\tilde{x}$ to be close to the exact solution x. If A is *ill-conditioned* this is too much to expect. What we do expect of a (backward) stable algorithm is that the computed solution $\tilde{x}$ satisfies

$$\tilde{A}\tilde{x} = b$$

where $\tilde{A}$ is close to A.

Closeness

"Close" usually means close in the relative sense in some norm, e.g.

$$||\tilde{A} - A|| \leq 2^{-t} f(n) ||A|| \,,$$

where $f(n)$ is a slowly-growing function of the size of the problem (usually a low-degree polynomial), and t is the number of bits in the floating-point fraction.

The symmetric eigenvalue problem

Recall that the *eigenvalues* of a square $n \times n$ matrix A are (possibly complex) numbers λ such that $A - \lambda I$ is singular. Also,

$$f(z) = \det(zI - A)$$

is called the "characteristic polynomial" of A. Thus, the eigenvalues are just the zeros of this characteristic polynomial.

Given a (real, square) symmetric matrix A, it turns out that the eigenvalues are real and *well-conditioned*. That is, if we make a small (symmetric) change in A, the eigenvalues only change by a small amount. More precisely

$$|\lambda_j - \tilde{\lambda}_j| \leq ||A - \tilde{A}||_F \,.$$

There are well-known stable algorithms for computing the eigenvalues of A. Because the problem is well-conditioned, stable algorithms compute the eigenvalues with small absolute errors (small compared to $||A||$).

Using the characteristic polynomial

The obvious algorithm: computing the characteristic polynomial $f(z)$ of A, then finding the zeros of $f(z)$, is *not* stable.

Wilkinson gives the example

$$A = \mathrm{diag}(1, 2, \ldots, 20)$$

so

$$f(z) = (z-1)\cdots(z-20) = z^{20} - 210z^{19} + \cdots + 20!$$

Suppose we compute $f(z)$ but make just *one* rounding error that changes the coefficient 210 to $210 + 2^{-23}$, that is we compute

$$\tilde{f}(z) = f(z) - 2^{-23} z^{19} \,.$$

Then all hope of solving the orignal problem accurately is lost, because ten of the zeros of $\tilde{f}(z)$ are complex with quite large imaginary parts, e.g. there is a pair

$$16.73 \pm 2.81i \,.$$

Comment

There is a tendency to regard such facts as "well-known" or "ancient history", and hence not worth learning, but if you don't know history you are doomed to repeat it!

Tradeoffs

Sometimes there is a tradeoff between speed and stability. For example, the fastest known *parallel* algorithm to solve an $n \times n$ linear system

$$Ax = b$$

is *Csanky's algorithm*, which requires time $O\left((\log n)^2\right)$ using many (but polynomial in n) processors. This is of theoretical interest, because it shows that the problem is in the complexity class NC.

Unfortunately Csanky's method depends on fast computation of the characteristic polynomial of A (followed by an application of the Cayley-Hamilton theorem) and is numerically unstable. Avoid computing the characteristic polynomial!

Entered the wrong data

Now for something completely different.

In September 1997 the US Navy missile cruiser USS Yorktown was off the coast of Virginia. A sailor accidentally entered a zero into a data field of the *Remote Database Manager Program*, part of the new *Smart Ship* system, which was designed "to automate tasks that sailors have traditionally done themselves".

This caused a divide by zero in the program, since whoever wrote the program had not bothered to check for zero data. The divide by zero caused a buffer overflow, and the "failsafe" system shut down.

Unfortunately the program also controlled the ship's propulsion, so the ship was "dead in the water" for more than two hours before the Navy figured out how to reboot the system (based on Windows NT).

Lucky it wasn't in a war zone!

Program bug

If you are developing your own programs, then a program bug is the most likely reason for wrong answers.

Some suggestions:

- Start with as simple a program as possible and debug that before adding refinements such as optimisations, bells and whistles.
- As you add "improvements" check that the output is still correct.
- If possible, use two different methods and compare results.
- If possible, try to replicate published results before trying to get new results (but be aware that published results are not always correct).

Example of debugging – GMP-ECM

An example of a program that is particularly hard to debug is the *elliptic curve method* (ECM) for integer factorisation. This uses a Monte Carlo algorithm, so you need to be able to rerun the program with the same "random" numbers in order to test changes. Also, the algorithm has two phases and a bug in one phase will not necessarily stop the other phase from finding factors (albeit more slowly).

As a way to help people debug and compare their ECM programs, I introduced a "standard" way of generating a pseudo-random elliptic curve (parameterised by one pseudo-random number σ). Then the developers of GMP-ECM were able to compare the output of their program with the output of my (earlier) program using exactly the same elliptic curves and other parameters. This was extremely useful for debugging some sophisticated enhancements of phase 2.

Ariane 5 rocket crash

In June 1996 the European Space Agency launched its new Ariane 5 rocket, an upgrade of the old Ariane 4. Unfortunately the software had not been (sufficiently) upgraded. The rocket veered off course and exploded 37 seconds after liftoff. $US500 million worth of communications satellites on board were lost (not to mention ESA's reputation).

It turned out that horizontal velocity was stored in a 64-bit floating-point number and then converted to a 16-bit signed integer. This was OK for Ariane 4, but for Ariane 5 the number exceeded $2^{15} - 1$ and integer overflow occurred, with disastrous results.

Was this a program bug? More a bug in the program specification, which was not upgraded at the same time as the rocket.

Patriot missile problem

The Patriot missile provides another example where the software met the original specification, but the specification proved inadequate when the system was used in a way that was not anticipated by the designers.

During the first Iraq war (Feb 1991) an Iraqi Scud missile was fired towards Dhahran, Saudi Arabia. A US Patriot missile should have intercepted it but failed to do so. 28 soldiers died as a result, and 97 were injured.

What went wrong?

It turned out that the Patriot system had an internal clock that incremented every 0.1 seconds, and the time (in seconds) was determined by multiplying the counter value by a 24-bit approximation to 1/10. Note that $1/n$ is a non-terminating fraction in binary for any n that is not a power of 2, in particular for $n = 10$. Effectively the Patriot was multiplying by a number close to 0.0999999 instead of 0.1000000

The Patriot was intended to be a mobile system that would run for only a few hours at one site, and in that case the rounding error would not be serious. However, in Dhahran it ran for 100 hours and the rounding error was 0.34 seconds (greater than the clock cycle of 0.1 sec). The Patriot became confused and could not track the Scud missile, so treated it as a false alarm.

It is sometimes said that the Patriot "missed" the incoming Scud but this is misleading because the Patriot never left the ground!

Mars Climate Orbiter

In Dec 1998 the Mars Climate Orbiter was launched from Earth. It arrived at Mars in Sept 1999. However, a navigation error caused it to burn up in the Martian atmosphere.

A review board found that some data was calculated on the ground in Imperial units (pound-seconds) and reported that way to the navigation system, which was expecting the data in metric units (newton-seconds).

Unfortunately most programming languages deal only with dimensionless quantities – the responsibility for conversion of units rests with the programmer!

The Climate Orbiter was intended to relay signals from the Mars Polar Lander, which was launched in Jan 1999. Communication with the Polar Lander was lost during an attempted landing near the South pole of Mars in Dec 1999.

Compiler bugs

If your program does not work as expected, it is tempting to blame the compiler. In nearly all cases the compiler is not to blame. It is usually a typo, logic bug, or a misunderstanding of the syntax/semantics of the programming language (e.g. beware implicit type conversions, the operators "=" and "==" in C, etc).

Turn on compiler warnings and don't ignore them unless you are sure it is safe to do so. Debug with optimisation turned off then see if turning it on changes anything. (If it does, check for uninitialised variables and array bound violations.)

Compiler bugs do exist, especially for "exotic" or rarely used features (e.g. extended precision). Avoid such features if possible.

The Pentium FDIV bug

Firmware/hardware bugs are the least likely, but also the most spectacular and expensive (for the computer manufacturer)!

Perhaps the best-known is the 1994 Intel Pentium "FDIV" bug. Eventually Intel offered to replace all faulty Pentium processors at an estimated cost of $US475 million. Many users did not bother to get their processors replaced, so it cost Intel less than their estimate. However, no one at Intel got a Christmas bonus in 1994!

What was the FDIV bug?

When designing their Pentium processor to replace the 80486, Intel aimed to speed up floating-point scalar code by a factor of three compared to the 486DX chip,

The 486 used a traditional shift-and-subtract algorithm for division, generating one quotient bit per clock cycle. For the Pentium Intel decided to use the SRT algorithm that can generate two quotient bits per clock cycle. The SRT algorithm uses a lookup table to calculate intermediate quotients.

Intel's lookup table should have had 1066 table entries, but due to a faulty optimisation five of these were not downloaded into the programmable logic array (PLA). When any of these five entries is accessed by the floating point unit (FPU), the FPU fetches zero instead of the correct value. This results in an incorrect answer for FDIV (double-precision division) and related operations.

Example

The error is usually in the 9th or 10th decimal digit, but in rare cases it can be much worse.

FDIV is supposed to give a 53-bit result, i.e. about 16 decimal digits.

An example found by Tim Coe is

$$c = \frac{4195835}{3145727}.$$

The correct value is

$$c = 1.333\underline{8}20449136241\cdots$$

but a faulty Pentium gives

$$c = 1.333\underline{7}3906\cdots$$

which is an error of about one part in 2^{14}.

Summary of Intel's reaction

1. There is no problem.
2. There is a small problem, but it is not serious.
3. The problem might be serious for some users; people who can prove that they are affected by the problem can have their Pentium processor replaced by Intel.
4. OK, we'll replace any flawed processor free of charge. [Note: the word "flaw" is used, not "bug".]

Jokes

There are lots of Pentium bug jokes. For example:

Intel's new motto:

United We Stand,
Divided We Fall

Time-line

The history is interesting:

- July 1994: Intel discovered the bug, but did not make this information public.
- Sept 1994: Thomas Nicely suspected the bug because he obtained different answers on Pentiums and 80486s (more on this later).
- 30 Oct 1994: Nicely, unable to convince Intel technical support, publicised the bug.
- 7 Nov 1994: Front page story in *Electronic Engineering Times*.

Time-line continued

- 22 Nov 1994: Intel press release "$\cdots$ can make errors in the 9th digit. $\cdots$ [Only] theoretical mathematicians should be concerned."
- 5 Dec 1994: Intel claimed the flaw would occur "once in $27,000$ years" for a typical spreadsheet user.
- 12 Dec 1994: IBM Research said that the error could occur as often as "once every 24 days". IBM stopped shipping PCs based on the Pentium.
- 21 Dec 1994: Intel said "We at Intel sincerely apologize for our recent handling of the recently publicized Pentium processor flaw" and offered to replace faulty processors.

Brun's constant

A *twin prime pair* is a pair $(p, p+2)$ where both p and $p+2$ are prime numbers, e.g. $(11, 13)$.

Brun's constant is the sum of reciprocals of twin primes, i.e.

$$B = \sum_{\text{twins } (p,p+2)} \left(\frac{1}{p} + \frac{1}{p+2} \right)$$

and it is known that B is finite. In contrast, the sum of reciprocals of all primes is divergent,

$$\sum_{\text{prime } p<x} \frac{1}{p} \sim \ln \ln x \to +\infty \,.$$

It is not known if there are infinitely many twin primes, although this would follow from the *Hardy-Littlewood conjecture* which is believed by most number theorists.

In 1975 I computed the sum of reciprocals of twin primes up to 10^{11} and estimated

$$B \approx 1.9021604$$

Nicely's contribution

In 1994 Nicely decided to extend my 1975 computation. He started with some 80486 processors and compared his results with those that I had obtained on a Univac 1108. This revealed various bugs in his program and also a bug in the Borland C++ compiler. By September Nicely had verified my results and was confident that his program was correct.

To be ultra-cautious he computed sums in blocks of size 10^{12} and *performed each run in duplicate on two machines*. He started to use one Pentium as well as several 80486 processors. It soon became clear that the Pentium was producing different results from the other machines! After several days of checking every other possible cause, the problem was narrowed down to the Pentium floating-point hardware unit. However, Nicely was unable to convince Intel technical support of this.

On 30 October Nicely publicised the problem in an email message to several "interested parties". Soon "all hell broke loose".

Other discrepancies

Nicely continued his computations, and occasionally other discrepancies appeared between duplicated runs.

Two discrepancies were traced to defective memory (SIMM) chips; parity checking had failed to report the errors. Once a disk subsystem failure generated many incorrect (but plausible) results. Other discrepancies were probably caused by "soft" memory errors.

Soft memory errors

A *soft memory error* occurs when a cosmic ray or alpha particle flips one or more bits in memory. A single bit error should be detected by a parity check and can be corrected if the memory has "error checking and correction" (ECC) hardware. Usually ECC can detect (but not always correct) a double bit error – this is called SEC/DED. Some systems write an entry in an error log whenever an error is detected.

Soft memory errors are a known but not well-advertised feature of modern memory chips. According to Sun, a system with 10 GB of memory might get an "ECC event" every 100 to 1000 hours, though this depends on the solar sunspot cycle, the latitude, altitude, amount of shielding, degree of miniaturisation, whether the memory is interleaved, etc, etc.

Soft memory errors are probably the most common sort of transient errors (errors that can not be replicated).

Other experiences of transient errors

Apart from Nicely's twin primes computation, several other large computations have been checked and occasional transient errors found.

GIMPS

GIMPS is the "Great Internet Mersenne Prime Search". This is a project to search for primes of the form $2^n - 1$. On 4 Sept 2006 GIMPS announced the Mersenne prime

$$2^{32582657} - 1\,.$$

A number $N = 2^n - 1 > 3$ can be proved prime by performing a *Lucas-Lehmer* test: if $s_0 = 4$ and

$$s_{k+1} = s_k^2 - 2 \bmod N$$

then N is prime if and only if $s_{n-2} = 0$.

For example, if $n = 7$, we get the sequence $(4, 14, 67, 42, 111, 0)$, so 127 is prime.

The GIMPS organisers are careful, and all results are checked before they are announced. This has avoided at least one embarrassment.

The GIMPS experience

Occasionally two computers testing the same N find different values of s_{n-2}. Thus at least one is incorrect! So far about $400,000$ Lucas-Lehmer tests have been checked, and the observed error rate is about 1.1%.

A "P-90 CPU-year" is the work done by a 90 Mhz Pentium in one year. According to George Woltman:

> The average LL test now takes about 6.5 P-90 CPU-years. So my rough calculations are 0.011 errors per 6.5 CPU-years or 1 error every 600 P-90 CPU-years.

Nowadays most machines are much faster than 90 MHz. If you have a cluster of $64 \times 1\,\text{Ghz}$ machines, you can expect an error about once every ten months!

Irreducible and primitive trinomials

A *trinomial* is a polynomial with three nonzero terms, e.g.

$$x^7 + x^3 + 1\,.$$

We consider polynomials over the finite field GF(2). A polynomial is *reducible* if it has nontrivial factors, otherwise it is *irreducible*. For example, $x^3 + 1$ is reducible over $GF(2)$ because it has (irreducible) factors $x + 1$ and $x^2 + x + 1$.

There is an interest in finding irreducible trinomials $x^r + x^s + 1$ of high degree r. We always assume $r > s > 0$. Also, without loss of generality $r \geq 2s$, as otherwise we can consider the "reciprocal trinomial" $x^r + x^{r-s} + 1$.

A *primitive* trinomial is an irreducible trinomial that satisfies another rather technical condition. This condition is always satisfied if $2^r - 1$ is a Mersenne prime. Thus, whenever a new Mersenne prime is discovered, there is a possibility of searching for new primitive trinomials.

Primitive trinomial search

A few years ago Samuli Larvala, Paul Zimmermann and I found a new algorithm for testing irreducibility of trinomials and embarked on a search for large primitive trinomials. Following my own advice, I checked the published results and found an error! The error was not in our program but in a paper by Kumada *et al* in *Math. Comp.*

For the case $r = 859433$, Kumada *et al* claimed there was only one primitive trinomial, with $s = 288477$, but we found another one with $s = 170340$. It turned out that there was a bug in Kumada's program and this trinomial was incorrectly discarded by the sieving step (which looks for small factors).

We then looked at the cases $r = 3021377$ and $r = 6972593$, and found three new primitive trinomials.

$$x^{6972593} + x^{3037958} + 1\,,$$

which was found in August 2002, is the largest known primitive trinomial.

Checking our results

Knowing Nicely's story, I decided to check the new results with a slightly different program running on different machines. So far we have checked 80% of the cases and found four errors! They are all transient errors that can not be replicated, and were probably caused by soft memory errors. Fortunately they do not change the results that Larvala, Zimmermann and I published.

Our error rate is about one error per 1000 P-90 CPU-years, not too much different from the rate observed for the GIMPS project.

Conclusion

If the result of a long computation is important, then it should be verified. Otherwise the result can not be trusted.

Case study – Primality testing

In 2002 Agrawal, Kayal and Saxena (AKS) surprised number theorists by finding a *deterministic polynomial time* primality test. This was certainly a great theoretical result. However, from a *practical* point of view, nothing changed.

Before AKS, the best practical algorithm was the Rabin-Miller *probabilistic* algorithm. If you run this algorithm once, it has a probability $< 1/4$ of giving the wrong answer (an error can only occur if the number being tested is composite). Thus, if you run Rabin-Miller 100 times and it gives the same answer each time, the probability of error is

$$< 4^{-100} < 10^{-60}\,,$$

which is close enough to certainty for practical purposes.

If you really want certainly, there is another probabilistic algorithm, ECPP, which always gives the right answer; only the running time is nondeterministic.

Comparison of algorithms

Let's consider testing a 100-decimal digit number on a 1 GHz machine,

- 10 runs of Rabin-Miller takes 0.03 seconds (using our Magma implementation) with error probability < 0.000001
- 100 runs of Rabin-Miller takes 0.3 seconds with error probability $< 10^{-60}$
- ECPP takes about 2 seconds
- AKS takes 37 weeks and, using the GIMPS statistics, the result has a probability of error exceeding 0.01 even if the program is correct!

Which would you choose ?

Part 2 – Random Number Generators

> *Random numbers should not be generated with a method chosen at random. Some theory should be used.*
>
> Don Knuth (Vol. 2, §3.1).

Pseudo-random number generators (RNGs) are critical in simulation and cryptography.

Note the qualification "pseudo". There is no way to generate random numbers on a deterministic computer (unless we allow random inputs, e.g. from a cosmic ray counter or some quantum device). All we can hope for is a deterministic sequence of numbers (x_n) that *appear* random in the sense that they pass all our favourite statistical tests, so are statistically indistinguishable from a random sequence.

Cryptographic applications

In cryptographic applications (x_n) is usually a sequence of *bits* and it is important that the sequence is *unpredictable* which is a stronger requirement than passing statistical tests.

Formally, for any $\varepsilon > 0$, given $(x_0, x_1, \ldots, x_{n-1})$ it should not be possible to predict x_n [in polynomial time?] with probability greater than $0.5 + \varepsilon$. It is not known if this is possible.

Public-key cryptography is built on the stronger assumption that *one-way* functions exist (this assumption implies $P \neq NP$).

I won't consider cryptographic applications today.

Quasi-random numbers

Quasi-random sequences are designed for specific applications such as numerical quadrature. The numbers need to be "well-distributed" but they do not need to be independent (there can be a strong correlation between x_n and x_{n+1}).

For example, if you want to approximate the integral of an analytic function $f(x)$ on the unit circle using function evaluations at N points, you probably can't do better than use the N-th roots of unity

$$\exp(2\pi ij/N),\ j = 1, \ldots, N\,.$$

These points are certainly not random!

Life gets more interesting on other domains, e.g. a sphere or a high-dimensional cube, but in any case the requirements for *quasi*-random numbers are quite different from those for *pseudo*-random numbers.

We won't consider quasi-random numbers any further today. From now on, "random" will mean "pseudo-random".

Pseudo-random number generators

Often we want random numbers that are uniformly distributed on some bounded interval. By a linear transformation we can assume the interval is $[0, 1]$.

If the computer wordlength is w, it is often convenient to generate pseudo-random *integers* $n \in \{0, 1, \ldots, 2^w - 1\}$ and then scale them appropriately (e.g. return $x = n/2^w$).

There is a certain *discretisation error* here: we have restricted ourselves to a finite set of 2^w rational numbers in $[0, 1)$ instead of the uncountable set of real numbers $[0, 1]$. However, this is probably acceptable if w is not too small.

Nowadays most computers have $w = 32$ or $w = 64$, though in conversion to single/double IEEE "real" format some low-order bits are lost ($32 \to 24$ and $64 \to 53$).

The state

The *state* of a random number generator is the information that uniquely defines all future numbers in the sequence.

Often the state is just the last number in the sequence, so

$$x_{n+1} = f(x_n)$$

for some (deterministic but pseudo-random) function f, Thus, if x_n is represented in a w-bit word, the state is at most w bits.

Sometimes the state includes other numbers in the sequence, e.g.

$$x_{n+1} = f(x_n, x_{n-1})$$

or other bits that are "hidden" from the user:

$$x_{n+1} = f(s_n), \quad s_{n+1} = g(s_n)\,.$$

All practical RNGs have a *finite* (and not too large) state.

The period

The sequence (x_j) is *periodic* with period N if

$$x_{j+N} = x_j \ \text{ for all sufficiently large } \ j$$

and N is the smallest positive integer with this property.

For example,

$$(1, 7, 0, 3, 0, 3, 0, 3, 0, 3, \ldots)$$

is periodic with period 2. Here $(1, 7)$ is the "non-periodic part" or "tail"; often it is empty.

It is important to note that any RNG with n state bits must be periodic with period

$$N \le 2^n\,.$$

This is because there are only 2^n possibilities for the state, and the state uniquely defines the future.

Period should be large

When using a random number generator we don't want the numbers to repeat. Thus the period N should be larger than the number of random numbers that we will ever use.

Early RNGs often had only 32 bits of state, so $N \le 2^{32}$. If we make the reasonable assumption that a random number can be generated and used in 1 μsec, it takes only 72 minutes to run through 2^{32} numbers. Thus, 32 bits of state is *too small*.

In fact, computers are getting faster and we use clusters with many processors, sometimes running programs for days or weeks, so the state should be *much* larger than 32 bits.

More subtle is the fact that deterministic sequences have structure, and many RNGs fail certain statistical tests if we use more than about $\sqrt{N}$ consecutive numbers. Thus, even a state of 64 bits may be too small.

I recommend a state of at least 256 bits.

Linear congruential generators

The most common RNGs used to be *linear congruential* generators of the form

$$x_{n+1} = (ax_n + c) \bmod m\,,$$

where each x_n is a nonnegative integer.
a, c and m are suitably chosen constants, and the sequence is defined once x_0 is given.

x_0 is the *seed*
a is the *multiplier*
m is the *modulus*.

Usually $m \le 2^w$ so each x_n fits in a single computer word (in C, an "unsigned int" or "unsigned long"). Of course x_n can be scaled to give a real number in $[0, 1)$.

The choice of multiplier is important and there is a lot of theory concerning this (see Knuth, Vol. 2).

Mainly in the planes

In a famous paper entitled *"Random numbers fall mainly in the planes"*, Marsaglia showed that linear congruential generators all suffer from a defect (though some more than others).

Suppose we use $(x_{3n}, x_{3n+1}, x_{3n+2})$ to get a point P_n in the unit cube (or similarly in higher dimensions). Then these points all fall on a relatively small number of parallel planes (or hyperplanes). No matter how many points we generate, the space between these planes will never be sampled.

For example, the generator RANDU with $a = 65539$, $c = 1$, $m = 2^{31}$ was once widely used because it was included in an IBM library (and later copied by DEC and Fujitsu). However, the points generated by RANDU lie on just 15 planes!

Analysis of RANDU

The main problem with RANDU is that $a = 2^{16} + 3$ (IBM chose this value so multiplication by a would be fast).
It's easy to see that

$$a^2 - 6a + 9 = 0 \bmod m$$

and deduce that

$$x_{n+2} - 6x_{n+1} + 9x_n = 2^{16} - 2 \bmod m\,.$$

Thus (x_{n+2}, x_{n+1}, x_n) lies on one of the planes

$$x - 6y + 9z = km + 2^{16} - 2$$

and we must have $-5 \le k \le 9$. The distance between the planes is

$$\frac{1}{\sqrt{1^2 + 6^2 + 9^2}} > 0.092\,,$$

so we can fit a sphere of diameter 0.092 between the planes and no points will fall in it!

Short period

A generic problem with linear congruential generators is that the period is at most $m \le 2^w$, which is much too small.

Summary

Linear congruential generators are obsolete and should not be used, except for "toy" examples or applications such as initialising better generators.

Other generators

Many classes of RNGs have been proposed to replace linear congruential generators. For example:

- "xorshift"
- "add with carry" (or "subtract with borrow")
- Generalised Fibonacci
- Linear feedback shift register (LFSR)
- Combinations of the above using shuffling or rejection methods

To give just one example, if we have a primitive trinomial

$$x^r + x^s + 1$$

then there is an associated LFSR (single-bit) generator

$$x_n = x_{n-r} \oplus x_{n-s}$$

and this has period $2^r - 1$ provided $(x_0, x_1, \ldots, x_{r-1})$ are not all zero. The parameters r and s are called *lags*.

Memory requirements

If the state is too large, the generator uses a lot of memory and tends to be slow because the state bits will not all fit in the machine's cache. Also, the state has to be initialised – more on that later. A state of more than 1KB $(8,192$ bits) may be "overkill".

Random number generators based on primitive trinomials of degree r need at least r state bits. For example, the *Mersenne Twister* uses a primitive trinomial of degree $19,937$ and has a state of (slightly more than) $19,937$ bits.

It is dangerous to use generators based on trinomials of small degree because trinomials have low *weight* (each number depends on only two previous numbers in the sequence).

Knuth's random number generator

A few years ago, in collaboration with Pedro Gimeno (Spain) and Don Knuth (Stanford), I fixed a serious flaw in the RNG published in Volume 2 (third edition) of Knuth's *The Art of Computer Programming*.

The flaw is not unique to Knuth's generator: it applies to most commonly-used RNGs!

It's interesting that Knuth recommends a different RNG in each edition of his Volume 2 (and some experts think there is still plenty of room for improvement).

Testing RNGs – traditional approach

Random number generators typically generate a sequence

$$X^{(s)} = (x_0^{(s)}, x_1^{(s)}, x_2^{(s)}, \ldots)$$

which depends on a *seed* s provided by the user. Different seeds should provide different, preferably uncorrelated, sequences.

When testing a random number generator we traditionally select a seed s, generate a long initial segment of the sequence $X^{(s)}$, and apply various empirical tests to see if it behaves "like" a genuine random sequence. For example, Marsaglia's `Diehard` program tests about 3×10^6 numbers using assorted statistical tests.

The traditional method of testing is reasonable if the random numbers are to be used in a long simulation.

Testing RNGs – Pedro's approach

Pedro Gimeno was looking for a good random number generator for a computer game application. He wanted to use many different seeds (say $s = 1, 2, 3, ..., 30000$) and only a short segment of $X^{(s)}$ for each seed (say 100 numbers per seed). Each seed corresponded to a different game and the set of random numbers was used to initialise the game. Think of each seed generating a different game of poker or bridge, and the random numbers being used to shuffle the deck of cards.

Pedro wanted different seeds to generate different games. He also wanted the short segments of random numbers to be uncorrelated so each game was independent of the others. An obvious way to test this is to concatenate the different segments and feed the combined list into `Diehard`.

Testing with Diehard

If S is the number of different games required, and each game uses n random numbers, the list tested by `Diehard` is:

$$\begin{array}{c} x_0^{(1)}, x_1^{(1)}, \quad \ldots \quad , x_{n-1}^{(1)}, \\ x_0^{(2)}, x_1^{(2)}, \quad \ldots \quad , x_{n-1}^{(2)}, \\ \ldots \\ x_0^{(S)}, x_1^{(S)}, \quad \ldots \quad , x_{n-1}^{(S)} \end{array}$$

There is nothing special about Pedro's choice of $S = 30000$, $n = 100$. The traditional case is the extreme $S = 1$, n large.

Another extreme case is S large, $n = 1$. In this case we want $x_0^{(s)}$ to behave like a random function of the seed s.

A good, general-purpose RNG such as Knuth's should pass *all* the tests we care to throw at it. Thus, it should pass *both* the traditional test and Pedro's test.

The problem

Pedro and I found that Knuth's RNG conclusively *failed* several of `Diehard`'s tests when tested with $S = 30000$, $n = 100$.

We tried several other well-known generators. They nearly all failed just as spectacularly as Knuth's (some even more spectacularly – for example it's easy to see why all linear congruential generators fail).

One generator which passed was my `RANU4` – surprising since Knuth's generator is based on the same theory as `RANU4`, but the initialisations differ in a subtle way that turns out to be crucially important.

The solution – outline

> *When writing out a correct proof,*
> *the proof gets shorter.*
> Andrew Wiles, May 2000.

The idea is to ensure that the *least significant bits* of the RNG behave randomly, and that there is enough time for this randomness to propagate to the high order bits (by carry propagation during addition).

Knuth's original method accidentally injected non-randomness into the high-order bits faster than they were randomised by carries from the low order bits!

The improved version passes `Diehard`'s tests with flying colours. As a bonus, it is *simpler* and easier to understand than the earlier version.

However, there are still some doubts about this generator because it is based on a primitive trinomial with quite small lags. I prefer my generator `xorgens`.

Other distributions

Sometimes we need random numbers with a given non-uniform distribution, e.g. exponential or normal (Gaussian). This is a big topic and I don't have time to go into it. However, I can tell you one amusing story about the normal random number generator used at a certain university some years ago.

The software library at the time used the formula

$$x = u_1 + u_2 + \cdots + u_{12}$$

the get an approximation x to a normal $N(0,1)$ random variable. Here $u_1, \ldots, u_{12}$ are independent uniform random numbers in the interval $[-0.5, +0.5]$. It's easy to see that x has the correct mean (0) and variance (1), and by the central limit theorem we expect x to be approximately normally distributed. Clearly the approximation is poor in the tails, since $|x| \leq 6$.

Improvement

When the University's main computer was upgraded to a faster one with a longer wordlength, it was decided to improve the approximation by taking 15 terms instead of 12. Thus the software library was modified to use the formula

$$x = u_1 + u_2 + \cdots + u_{15}\,.$$

This was used for several months, and contributed to quite a few papers and at least one Ph.D. thesis, before someone noticed that the variance of x had changed to $15/12 = 1.25$.

References

[1] R. P. Brent, Some long-period random number generators using shifts and xors, CTAC06, Townsville, July 2006.

[2] R. P. Brent, S. Larvala and P. Zimmermann, A fast algorithm for testing reducibility of trinomials mod 2 and some new primitive trinomials of degree 3021377, *Math. Comp.* 72 (2003), 1443–1452.

[3] G. E. Forsythe, Pitfalls in computation, or why a math book isn't enough, *Amer. Math. Monthly* 77 (1970), 931–956.

[4] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third ed.), Addison-Wesley, Menlo Park, CA, 1998, §3.6. Problems with `ran_start` and `ran_array` fixed in the ninth printing, January 2002: `http://www-cs-faculty.stanford.edu/~knuth/news.html`.

[5] P. L'Ecuyer, Random number generation, in *Handbook of Computational Statistics*, Springer-Verlag, 2004, 35–70.

[6] G. Marsaglia, Random numbers fall mainly in the planes, *Proc. Acad. Sci. USA* 60 (1968), 25–28.

[7] G. Marsaglia, A current view of random number generators, in *Computer Science and Statistics: The Interface*, Elsevier Science Publishers B. V.,1985, 3–10.

[8] Patriot Missile: see *Report to US House of Representatives Committee on Science, Space, and Technology* at http://www.fas.org/spp/starwars/gao/im92026.htm

[9] Terrazon Semiconductor, *Soft errors in electronic memory – a white paper*, 2004. http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf

[10] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, HMSO, 1963.