Generating Random Numbers

Harchol-Balter & Sutner

Carnegie Mellon University

Spring 2012

Outline

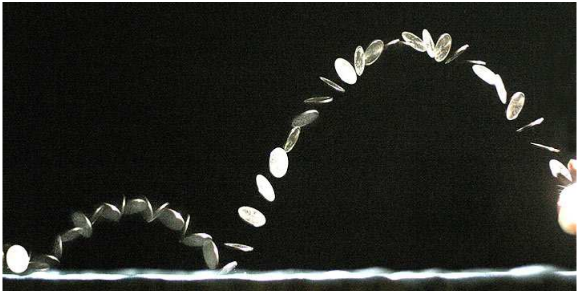
Generating Random Numbers

As we have seen, random numbers (or random bits) are a crucial ingredient in many algorithms.

Since computers are deterministic devices (more or less), it is actually not all that easy to produce random bits using a computer: whatever program we run will produce the same bits if we run it again.

To make matters worse, it is rather difficult to even say exactly what is meant by a sequence of random bits. Of course, intuitively we all know what randomness means, right?

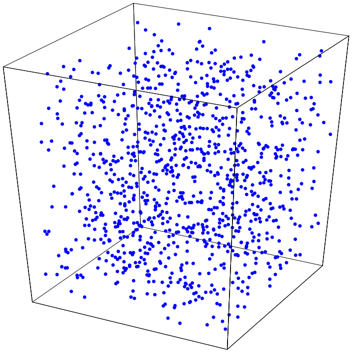
Flipping Coins



Lava Lamps

Krypton-85

Radioactivity is another great source of randomness – except that no one likes to keep a lump of radioactive material and a Geiger-Müller counter on their desk. Solution: keep the radioactive stuff someplace else and get the random bits over the web.



True random bits from `www.fourmilab.ch/hotbits`.

Pre-History

In the olden days, the RAND Corporation used a kind of electronic roulette wheel to generate a million random digits (rate: one per second).

In 1955 the data were published under the title:

A Million Random Digits With 100,000 Normal Deviates

"Normal deviates" simply means that the distribution of the random numbers is bell-shaped rather than uniform. But the New York Public Library shelved the book in the psychology section.

The RAND guys were surprised to find that their original sequence had several defects and required quite a bit of post-processing before it could pass muster as a random sequence. This took years to do.

Available at `http://www.rand.org/publications/classics/randomdigits`.

Fiat Lux

Incidentally, Noll and Cooper at Silicon Graphics discovered one day that the pretty lava lamps were completely irrelevant: they could get even better random bits with the lens cap on (there is enough noise in the circuits to get good randomness).

Another way to use light, very much unlike the original lava lamp system, is to exploit an elementary quantum optical process: a photon hitting a semi-transparent mirror either passes or is reflected.

The Quantis systems was developed at the University of Geneva, the first practical model was released in 1998.

Note that quantum physics is the only part of physics that claims that the outcome of certain processes is fundamentally random (which is why Einstein was never very fond of quantum physics).

`http://www.idquantique.com/`

The Magic Device

Quantis RNG

Features

- True quantum randomness
- High bit rate of 4Mbits/sec (up to 16Mbits/sec for PCI card)
- Low-cost device (1000+ Euros)
- Compact and reliable
- USB or PCI, drivers for Windows and Linux

Applications

- Numerical Simulations
- Statistical Research
- Lotteries and gambling
- Cryptography

Unpredictability

So far, all the examples we have seen are based on the notion of unpredictability in physics: the behavior of some physical systems is so complicated that the only way to determine the state at time t is "run" the system till time t.

Now suppose you use random bits in some algorithm and you want to **prove** your algorithm to be correct. Without a formal, mathematical definition you cannot even start with the proof.

And try to convince a proof checker that it's correct when the first line reads "flip a coin 10000 times."

Obstructions to Randomness

It is somewhat easier to consider infinite binary sequences $\boldsymbol{\alpha} \in \mathbf{2}^{\omega}$ rather than finite ones. Infinity is often an excellent approximation for finiteness.

Let's turn around for the moment and look at properties that would disqualify a sequence from being random in any intuitive sense of the word. Here are two obvious potential problems.

- Bias : the probability of a 0 is not $1/2$.
- Correlation: the $i+1$st bit is not independent from the i th bit.

Fortunately, we don't need to achieve perfection in either category: there are algorithms that can turn a slightly biased and/or correlated sequence into an unbiased and uncorrelated one.

Here is a simple though not entirely satisfactory method to eliminate bias due to von Neumann. Dealing with correlated bits is harder, we won't get involved.

Removing Bias

Suppose we have an imperfect source of random bits (real world sources typically fall into this category) that are already independent but that the probability of a 0 is $1/2 + \varepsilon$. To eliminate this bias, John von Neumann suggested to following algorithm:

- Read the bits, two at a time.
- Skip 00 and 11.
- For 01 and 10 output the first bit.

The probabilities of all 2-blocks are easily computed since we assume independence:

$$\begin{array}{lll}
00 & (1/2+\varepsilon)^2 & = 1/4 + \varepsilon + \varepsilon^2 \\
01 & (1/2+\varepsilon)(1/2-\varepsilon) & = 1/4 - \varepsilon^2 \\
10 & (1/2-\varepsilon)(1/2+\varepsilon) & = 1/4 - \varepsilon^2 \\
11 & (1/2-\varepsilon)^2 & = 1/4 - \varepsilon + \varepsilon^2
\end{array}$$

The resulting sequence has no bias, as needed. Of course, it's a bit shorter.

Density

It is easy to define the density of a finite binary word x of length n:

$$D(x) = 1/n \sum_i x_i$$

But how about infinite sequences?

Let $\boldsymbol{\alpha} \in \mathbf{2}^\omega$ and define the density of $\boldsymbol{\alpha}$ up to n to be $D(\boldsymbol{\alpha}, n) = \frac{1}{n} \sum_{i<n} \alpha_i$.
The limiting density of $\boldsymbol{\alpha}$ is

$$D(\boldsymbol{\alpha}) = \lim_n D(\boldsymbol{\alpha}, n),$$

Note that there is a huge problem with this definition: limits are precisely defined in analysis, and there is not much reason to assume that this particular limit should exist.

The Law of Large Numbers

The LoLN says that if we repeat an experiment often, the observed average does in fact converge to the expected value; almost certainly.

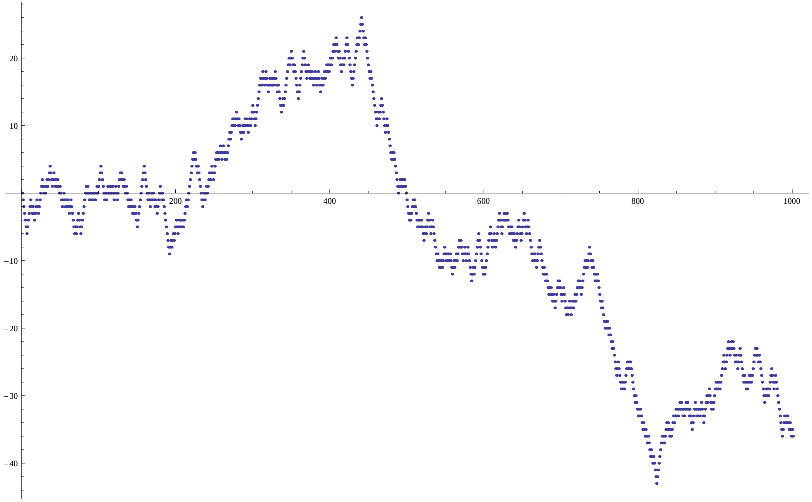
For example, for an unbiased coin we should approach the the limiting density of $1/2$. Don't ask what an unbiased coin is.

OK, since you asked ...

Also note that we should not expect the averages to be exactly equal to the expectation.

For example, performing a one-dimensional random walk with steps ± 1 we should expect to be up to $O(\sqrt{n})$ from the origin after n steps (so that $\sqrt{n}/n \to n$ for $n \to \infty$).

A Random Walk



Counting Blocks

We can look at density not just for single bits but for arbitrary finite blocks $w \in \mathbf{2}^m$: the number of occurrences of a particular block w in $x_1 x_2 \ldots x_n$ should approach $n/2^m$.

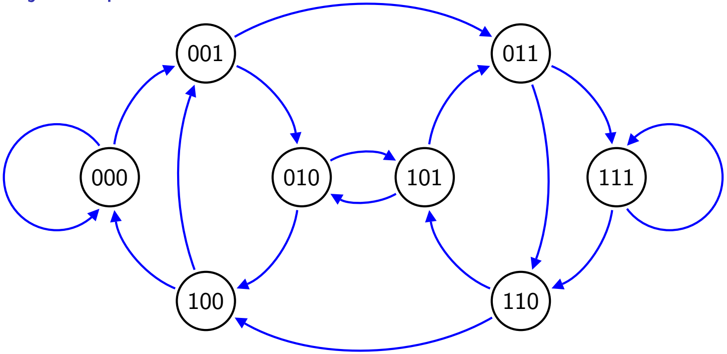
The limiting density of block w should be 2^{-k}.

A good way of thinking about this is to have $\boldsymbol{\alpha}$ trace a path in a de Bruijn graph. Recall that the de Bruijn graph of order k is defined as

$$\mathbb{B}_k = \left\langle \mathbf{2}^k, E \right\rangle$$
$$E = \{ (au, ub) \mid a, b \in \mathbf{2}, u \in \mathbf{2}^{k-1} \}$$

If we slide a window of width k over $\boldsymbol{\alpha}$ we obtain a sequence of nodes in $\mathbb{B}_k$, and the sequence is in fact a path in the graph.

De Bruijn Graph of Order 3

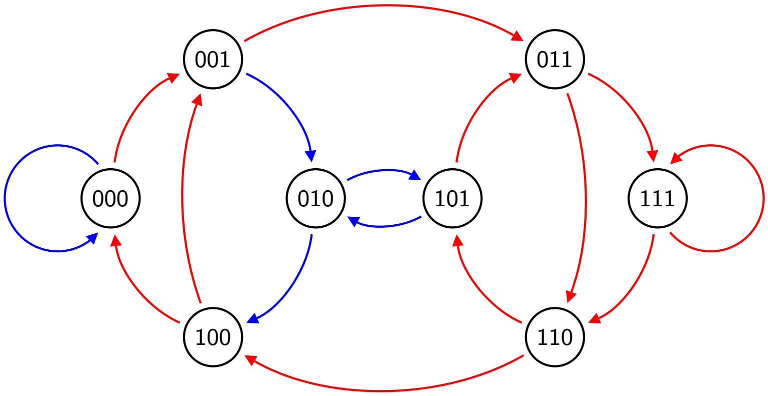


De Bruijn graphs are Hamiltonian, and the Hamiltonian cycles are called de Bruijn sequences: they contain every block of length k exactly once and have length $2^k + k - 1$.

Any finite or infinite sequence of bits (of length at least k) traces a path in the graph. If the sequence is unbiased the path hits each node equally often in the limit.

Example

The first 30 bits of the binary expansion of $\sqrt{5}$ from above produces



After 59 bits all edges lie on the path.

de Bruijn Sequences

Exercise

Suppose we have a de Bruijn sequence X of order k and let $\boldsymbol{\alpha} = X^\omega$. That that $\boldsymbol{\alpha}$ has limiting density $1/2^m$ for all blocks of length $m \leq k$ but not for larger blocks.

Martin-Löf Randomness

There is an elegant definition of randomness due to Martin-Löf: a sequence is random if it passes a certain test.

The definition of such a so-called universal sequential test used computability theory, so we'll pass.

But note that this definition is very strongly supported by the empirical fact that any practical test of randomness in ordinary probability theory can be translated into a sequential test. So, we are just dealing will all of these tests at once (plus all conceivable others).

Another good indication for the correctness of randomness in the sense of Martin-Löf random is that it is equivalent to other approaches (such as Kolmogorov-Chaitin randomness).

How Bad is Randomness?

The definition may be elegant and right, but it does not yield any methods to construct a random sequence. Aux contraire:

Theorem

Any Martin-Löf random sequence fails to be computable.

If you are familiar with computability theory: there are random sequences at level Δ_2 in the arithmetical hierarchy. Close, but not computable.

- Randomness and Physics
- (Not) Formalizing Randomness

3. Generating Random Numbers

No Such Thing

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.

John von Neumann

Arithmetical methods here should be read as “deterministic algorithm.”

Pseudo-Randomness

In the real world, one often makes do with a pseudo-random number generator (PRNG) based on iteration: the pseudo-random sequence is the orbit of a some initial element under some function.

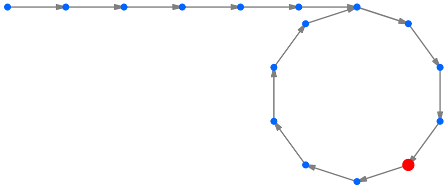
Find some nice discrete domain D (such as the integers) and a nice function $f : D \to D$; set

$$\begin{aligned} x_0 & \qquad \text{chosen somehow, at random :-)} \\ x_{n+1} &= f(x_n) \end{aligned}$$

Here f must be easily computable, typically using arithmetic and some bit-plumbing. One may use some simple filter h to produce $h(x_n)$ as the actual pseudo-random bits.

The Lasso

Of course, we are taking a huge step away from real randomness here, this sequence would perish miserably when exposed to a Martin-Löf test. The function f typically operates on some finite domain such as 64-bit words. Every orbit necessarily looks like so:



So we can only hope to make f fast and guarantee long periods.

Still, there are many applications where this type of pseudo-randomness is sufficient.

One has to be very careful with cryptography, though!

The Seed

Needless to say, running a PRNG twice with the same seed x_0 is going to produce exactly the same “random” sequence.

This can be a huge advantage, because it makes computations that are base on the random numbers reproducible (important for debugging and verification).

More generally, if we are willing to pay for a truly random seed we would hope that the iterative PRNG would amplify the randomness: we provide m truly random bits and get back n high-quality pseudo-random bits where $n \gg m$.

Hence PRNGs reduces the need for truly random bits but does not entirely eliminate them.

Linear Congruential Generator

A typical example: a simple affine map modulo m.

$$x_{n+1} = a\,x_n + b \bmod m$$

The trick here is to choose the proper values for the coefficients. Can be found in papers and on the web.

A choice that works reasonably well is

$$a = 1664525, b = 1013904223, m = 2^{32}$$

Note that a modulus of 2^{32} amounts to unsigned integer arithmetic on a 32-bit architecture, so this is implementation-friendly.

Multiplicative Congruential Generator

Omit the additive offset and use multiplicative constants only.

If need be, use a higher order recurrence.

$$x_n = a_1 x_{n-1} + a_2 x_{n-2} + \ldots a_k x_{n-k} \bmod m$$

For prime moduli one can achieve period length $m^k - 1$.

This is almost as fast and easy to implement as LCG (though there is of course more work involved in calculating modulo a prime).

Note, though, that there is more state: we need to store all of $x_{n-1}, x_{n-2}, \ldots, x_{n-k}$.

Inverse Congruential Generator

Choose the modulus m to be a prime number and write

$$\overline{x} = \begin{cases} 0 & \text{if } x = 0, \\ x^{-1} & \text{otherwise.} \end{cases}$$

Then we can define a pseudo-random sequence by

$$x_{n+1} = a\,\overline{x_n} + b \bmod m$$

Computing the inverse can be handled by the extended Euclidean algorithm.

Again, it is crucial to choose the proper values for the coefficients.

Blum-Blum-Shub

Pick two sufficiently large primes p and q, both congruent 3 modulo 4, and let $n = pq$.

Choose a coprime with n at random and set

$$\begin{aligned} a_0 &= a^2 \bmod n \\ a_{i+1} &= a_i^2 \bmod n. \end{aligned}$$

The initial a can be generated very cheaply.

Masking out the last few bits of a_i provides a very good source of pseudo-randomness.

Quality

The arithmetic used in BBS makes this generator expensive.

However, it is useful for cryptographic applications.

As usual, unless factorization is easy this should be a secure method.

Exercise

Show how to compute a_t directly, without first computing all a_i, $i < t$.

Wild Hacks

Stephen Wolfram suggested the use of a particular elementary cellular automaton known as rule 30 as a RNG.

An elementary cellular automaton is given by a local map or rule, a function

$$\rho : \mathbf{2} \times \mathbf{2} \times \mathbf{2} \longrightarrow \mathbf{2}$$

So, a local map is just a ternary boolean function. and can be specified by 8 bits, one for each possible input.

We can apply this local to map to a sequence of bits of arbitrary length by chopping the sequence into overlapping blocks of length 3 and applying f in parallel to all the blocks.

Global Maps

Let $\mathbf{2}^\infty = \mathbb{Z} \to \mathbf{2}$ denote the collection of all (bi-infinite) configurations.

We can extend any local map ρ to a global map

$$G_\rho : \mathbf{2}^\infty \longrightarrow \mathbf{2}^\infty$$

by setting

$$G_\rho(X)(i) = \rho(X(i-1), X(i), X(i+1))$$

Real World

In the pesky Real World we have to make do with a finite sequence of bits of length n; think of n as being around 2^8.

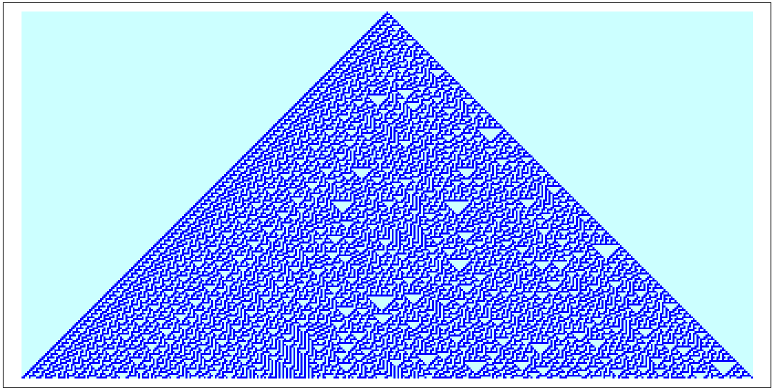
To deal with boundary problems assume the sequence wraps around at the end. So, we get a map

$$G_\rho : \mathbf{2}^n \to \mathbf{2}^n$$

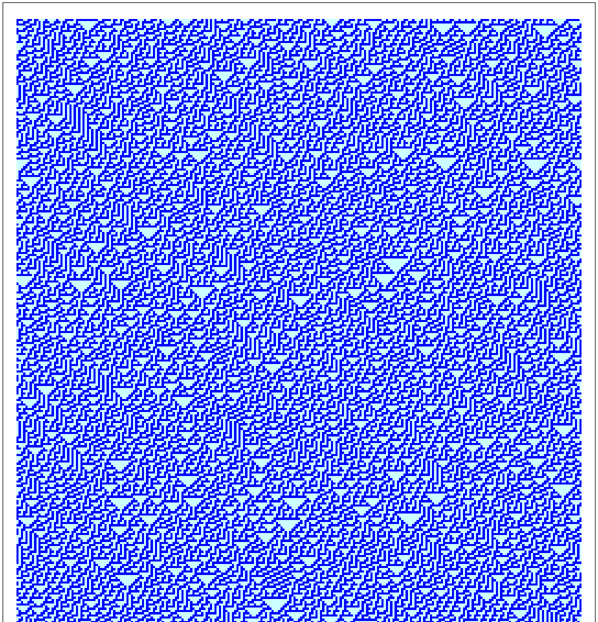
For rule 30, the local map looks like this:

$$\begin{array}{ll} 000 \to 0 & 100 \to 1 \\ 001 \to 1 & 101 \to 0 \\ 010 \to 1 & 110 \to 0 \\ 011 \to 1 & 111 \to 0 \end{array}$$

One Point Seed



A Little Later



Mersenne Twister

Fairly recent (1998) method by Matsumoto and Nishimura, seems to be the tool of choice at this point.

- Has huge period of $2^{19937} - 1$, a Mersenne prime.
- Is statistically random in all the bits of its output (after a bit of post-processing).
- Has negligible serial correlation between successive values.
- Only statistically unsound generators are much faster.

The method is very clever and not exactly obvious.

The algorithm works on bit-vectors of length w (typically 32 or 64).

Let k be the degree of the recursion, and choose $1 \leq m < k$ and $0 \leq r < w$.

The MT Recurrence

So we are trying to generate a sequence of bit-vectors $x_i \in \mathbf{2}^w$.

Define the join join(x, y) of $x, y \in \mathbf{2}^w$ to be the the first $w - r$ bits of x followed by the last r bits of y.

$$\mathrm{join}(x, y) = (x_1, x_2, \ldots, x_{w-r}, y_{w-r+1}, \ldots, y_w)$$

We can use the join operation to define the following recurrence:

$$x_{n+k} = x_{n+m} + \mathrm{join}(x_n, x_{n+1}) \cdot A$$

Here A is a sparse companion-type matrix that makes it easy to perform the vector-matrix multiplication.

Companion Matrix

The $w \times w$ matrix A has the following form:

$$A = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & \dots & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \dots & a_0 \end{pmatrix}$$

Note that $z \cdot A$ is not really a vector-matrix operation and can be handled in $O(w)$ steps.

This is just a convenient way to describe the necessary manipulations.

Good Parameters

Here is one excellent choice for the parameters:

$$w = 32, k = 624, m = 397, r = 31$$

and the A matrix is given by

$$\boldsymbol{a} = 0\text{x}9908\text{B}0\text{DF}$$

which hex-number represents the entries in the last row of A.

Recall that the recurrence determines x_{n+k} in terms of x_{n+m}, x_n and x_{n+1}, so we need to store a bit of state: $x_n, \ldots, x_{n+k-1} = x_{n+623}$.

So we need to store 624 words, not too bad.

A Miracle

This particular choice of parameters achieves the theoretical upper bound for the period:

$$2^{wk-r} = 2^{19937} \approx 4.32 \times 10^{6001}.$$

After a little bit of post-processing of the sequence (x_i) this method produces very high quality pseudo-random numbers, and is not overly costly.