Pseudo-, Quasi-, and Real Random Numbers

on Linux

Oleg Goldshmidt

olegg@il.ibm.com

IBM Haifa Research Laboratories

Agenda

- "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." [J. von Neumann, 1951]
  - Sinful pleasures.
- "If the numbers are not random, they are at least higgledy-piggledy." [G. Marsaglia, 1984]
  - Does it look random enough to you?
- "Random numbers should not be generated with a method chosen at random." [D. Knuth, 1998]
  - Pseudo-random and quasi-random.
- "Computers are very predictable devices." [T. Ts'o, probably circa 1994, but maybe as late as 1999]
  - Random tricks with the Linux kernel.

Life in Sin According to Knuth (I)

- Simulation
  - sciences: just about everywhere
  - operations research: workloads
- Sampling
- Numerical analysis
- Computer programming
  - random inputs
  - randomized algorithms
- Decision making
  - strategic executive decisions
  - Technion paper grades
  - optimal strategies in game theory

Life in Sin According to Knuth (II)

- Aesthetics
- Recreation (where do you think Monte Carlo comes from?)

Sins Knuth didn't consider:

- Financial markets
  - How random is IBM share price?
- Cryptography and cryptanalysis
  - Is "random" equivalent to "cryptographycally secure"?
  - RFC 1750, "Randomness Recommendations for Security"

Early (Biblical?) Virtues and Sins

- Intuition: dice, cards, lottery urn, census reports
- Physics: resistance noise (A. Turing, Mark I, 1951)

Disadvantage: irreproducible, difficult to debug

- A CD-ROM of random bytes (G. Marsaglia, 1995)
  - output of noise-diode circuit with scrambled rap music — “white and black noise”

Early attempts, while virtuous, were cumbersome and inadequate. A radical new approach was needed.

Von Neumann's Original Sin

- Can random numbers be produced by ordinary arithmetic?
- Von Neumann (circa 1946): take a long number (e.g., 10 digits), square it, extract the middle digits:
  - $X_n = 3756369827$
  - ${X_n}^2 = 14110314277196009929$
  - $X_{n+1} = 3142771960$
- Are these numbers random?
- No, but who cares? They appear random
- Obvious problem: 0 is stationary
- Another obvious problem: cycles
- 38 bit numbers: period of 750,000 (N. Metropolis, 1956)

Does π Look Random To You?

Are there any patterns in the decimal representation of π?

3.14159265358979323846264338327950

Does π Look Random To You?

Can this be considered a pattern?

3.14159**26**53589793238462**6**4338327950

Does π Look Random To You?

Can this be considered a pattern?

3.14159265358979323846264338327950

“The entire history of human race.” [Dr. I. J. Matrix, 1965]

What Are (Pseudo-)Random Numbers?

- Working definition of computer-generated random sequence:
  - a program that generates random sequences should be different and *statistically independent* from every program that *uses* its output.
- Interpretation of the definition:
  - two different generators ought to produce statistically indistinguishable results when coupled to your application.
  - if they don’t, at least one of them is not a good generator.

What Are Good PRNGs?

- Pragmatic point of view: there are statistical tests that are good at filtering out correlations that are likely to be felt by applications.
  - follows (to a certain extent) from the working definition coupled with insight and experience
  - good generators need to pass all the tests
  - or at least the user should be aware of failures to judge their impact
- How?

χ^2 Test

Consider a set of n independent observations of a random variable with a finite number of possible values
Rolling "true" dice:

s	2	3	4	5	6	7	8	9	10	11	12
p_s	$\frac{1}{36}$	$\frac{1}{18}$	$\frac{1}{12}$	$\frac{1}{9}$	$\frac{5}{36}$	$\frac{1}{6}$	$\frac{5}{36}$	$\frac{1}{9}$	$\frac{1}{12}$	$\frac{1}{18}$	$\frac{1}{36}$

Roll the dice $n = 144$ times, expect to see s on average np_s times:

s	2	3	4	5	6	7	8	9	10	11	12
E_s	4	8	12	16	20	24	20	16	12	8	4
O_s	2	4	10	12	22	29	21	15	14	9	6

What is the probability that the dice are "loaded" (i.e., the results are not random)?

χ^2 test (cont.)

- Compute the statistic

$$\chi^2 = \sum_{s=1}^{k} \frac{(O_s - E_s)^2}{E_s} = \frac{1}{n} \sum_{s=1}^{k} \left(\frac{O_s^2}{p_s} \right) - n$$

- For our experiment, $\chi^2 = 7\frac{7}{48}$ — is it improbable?
- Use χ^2 distribution with $\nu = k - 1 = 10$ degrees of freedom
  - NB: E_s, O_s are not completely independent: given $k - 1$ the k-th value can be computed.
- the probability that the sum of the squares of ν random normal variables of zero mean and unit variance will be greater than χ^2

Properties of χ^2 Distribution

- NB: independent experiments are assumed
  - exercise: combine a set of n experiments with itself, consider it a single set of size $2n$: how will χ^2 be affected?
- depends only on ν, not on n or p_s
- if $\nu >> 1$ and $n >> 1$ the χ^2 distribution is a good approximation
  - n should be large enough that all np_s are large (rule of thumb: more than 5)
  - large n will smooth out locally nonrandom behaviour: not a problem with dice but may be a problem with computer-generated numbers

χ^2 Test Criteria

- χ^2 should not be too high — we do not expect too much of a deviation from “true” dice!
- χ^2 should not be too low — if it is we cannot consider the numbers to be random!
- rules of thumb usually expressed in terms of χ^2 probability:
  - less than 1% or greater than 99% — reject
  - less than 5% or greater than 95% — suspect
  - 5% to 10% or 90% to 95% — somewhat suspect
  - between 10% and 90% — acceptable
- do the test several times - e.g. 2 out of 3

Kolmogorov-Smirnov (KS) Test

- χ^2 test applies when there is a finite number of degrees of freedom
- what about, e.g., random real numbers on [0,1)?
  - yeah, in the computer representation that is finite, but really large, and we want behaviour close to "real" anyway
- KS test: compare cumulative probability distribution functions (CDF)

$$F(x) = Probability(X < x)$$

- empirical CDF: given a sequence $X_0, X_1, X_2, ..., X_n$

$$F_n(x) = \frac{1}{n}\sum_{j=1}^{n} 1(X_j < x)$$

KS Test Algorithm

- theoretical criterion:

$$K_n^+ = \sqrt{n} \max_{-\infty<x<+\infty} (F_n(x) - F(x)),$$
$$K_n^- = \sqrt{n} \max_{-\infty<x<+\infty} (F(x) - F_n(x)).$$

  - what is $\sqrt{n}$ doing there? std. dev. of $F_n(x)$ is proportional to $1/\sqrt{n}$ for fixed x, so the factor makes the statistics (largely) independent of n

- practical criterion (assume sorting, but can do without)

$$K_n^+ = \sqrt{n} \max_{1<j<n} (\frac{j}{n} - F(X_j)),$$
$$K_n^- = \sqrt{n} \max_{1<j<n} (F(X_j) - \frac{j-1}{n}).$$

Practical Application of KS Test

- n should be large enough so that the empirical and the theoretical CDFs are observably different
- n should be small enough not to wipe out significant locally nonrandom behaviour
- apply KS to chunks of a long sequence of medium size ($n \approx 1000$)
  - obtain a sequence of $K_n^+(m)$, $K_n^-(m)$, $m = 1...r$
- apply KS test again to the sequence of $K_n^+(m)$ (and $K_n^-(m)$)
- for large n (≈ 1000) the distribution of $K_n^+(m)$ (and of $K_n^-(m)$) is closely approximated by

$$F_\infty(x) = 1 - \exp(-2x^2), \quad x \geq 0.$$

KS Test Criteria

- Again, K_n^+ and K_n^- should be neither too high nor too low
- There is a probability distribution associated with them, and we reject or suspect too low or too high probabilities
- Can be used in conjunction with the χ^2 test for discreet random variables
  - do χ^2 on chunks of the sequence
  - not a good policy to simply count how many χ^2 values are too large or too small
  - instead, obtain the empirical CDF of χ^2
  - use KS test to compare the empitrical CDF with the theoretical one

KS vs χ^2

- KS applies to CDFs without jumps
- χ^2 applies to CDFs with nothing but jumps
  - can be applied to continuous CDFs by binning
- sometimes KS is better, somethimes χ^2 wins
  - divide [0,1) into 100 bins
  - if deviations for bins 0...49 are positive, and for bins 50...99 — negative, KS will indicate a bigger difference than χ^2
  - if even deviations are positive and odd ones are negative, KS will indicate a closer match than χ^2

Empirical Tests

- outlines of algorithms of some common tests
- theoretical basis (TAOCP)
- uniform real numbers on [0,1):

$$\langle U_n \rangle = U_0, U_1, U_2, \ldots$$

- auxiliary integer sequence on [0,d-1]

$$\langle V_n \rangle = V_0, V_1, V_2, \ldots$$

  where

$$V_n = \lfloor dU_n \rfloor$$

- d is typically a power of 2, large enough for a meaningful test, but not too large to be practical

Empirical Tests (I)

- Frequency test (tests uniformity)
  - use KS test with $F(x) = x$ for $0 \leq x \leq 1$
  - use $\langle V_n \rangle$, for each $r < d$ count $V_j = r$, apply χ^2 test with $\nu = d - 1$ and $p_s = 1/d$.
- Serial test: we want pairs of successive numbers to be uniformly distributed, too: “The sun comes up just as often as it goes down, in the long run, but that does not make its motion random.” [D. Knuth]
  - count $(V_{2j}, V_{2j+1}) = (q, r)$, apply χ^2 test with $\nu = d^2 - 1$ and $p_s = 1/d^2$
  - generalize to triples, quadruples, etc.

Empirical Tests (II)

- Gap test: examine the length of gaps between occurrences of U_n in a given range
  - count number of gaps of different lengths, for lengths of 0,1,...t, and lengths $> t$, until n gaps are tabulated.
  - apply χ^2 test to the counts — details in TAOCP
- Poker test:
  - split $\langle V_n \rangle$ into “hands” (quintiples), apply χ^2 test to “pair”, “two pairs”, “three”, “full house”, “four”, “poker”
  - apply χ^2 test according to the number of distinct values in each “hand” — details in TAOCP.

Empirical Tests (III)

- Coupon collector's test
  - observe the lengths of segments of $\langle V_n \rangle$ that are required to collect a "complete set" of integers from 0 to $d-1$, apply χ^2 — details in TAOCP
- Permutation test
  - divide $\langle U_n \rangle$ into segments of length t
  - each segment can have $t!$ different orderings
  - count occurrences of each ordering, apply χ^2 test with $\nu = t! - 1$ and $p_s = 1/t!$.

Empirical Tests (IV)

- Run test: observe lengths of monotonic segments
  - do not apply χ^2 test: adjacent runs are not independent, a long runs will tend to be followed by a short one, and vice versa
  - throw away the element that immediately follows a run to make runs independent — details in TAOCP
- Collision test: what to do if number of degrees of freedom is much larger than the number of observations?
  - hashing: count the number of collisions
  - a generator will pass the test if it does not generate too few or too many collisions — details in TAOCP

`DIEHARD` I — General Description

- obtainable from `http://stat.fsu.edu/~geo/diehard.html`
- source code available in C, but it it obfuscated: it is "patched and jumbled" Fortran passed through `f2c`
  - you need `f2c` to link it (`-lf2c -lm`)
  - the original Fortran code is 30 years' worth of patches
  - very uncomfortable to alter, so don't — it is not advisable anyway unless you **really** know what you are doing
  - "seems to suit my purposes" [G. Marsaglia]
- there are also executables for DOS, Linux, and Sun

`DIEHARD` II — Components

- `makewhat` — creates test files
- `asc2bin` — converts ascii (hex) to binary
- `diehard` — runs the tests
- `diequick` — a shorter version
- a number of built-in random generators to test
  - `makewhat` prompts with a list
- a battery of tests
  - `diehard` allows you to choose from 15 tests
  - really more than 15, since a few tests are compound
  - some familiar: run test, permutations
  - some custom: DNA test, parking lot test

`DIEHARD` III — Procedure

- write a `main()` that does one of two things:
  - open a binary file and write your random integers to it
  - open a text file and write your random integers to it in hex
    - 8 hex digits per integer, 10 integers per line, no spaces
    - then run `asc2bin` on the text file
    - the ascii file will be twice the size of the binary one
- your PRNG should produce 32-bit integers
  - if 31-bit, then left-justify by left-shift: some tests favour leading bits

(Possible) Problems with `rand(3)`

- often linear congruential generators
  $I_{j+1} = aI_j + c \pmod{m}$ — period no greater than m
- can provide quite decent random numbers with proper choice of a, c, and m, fast
- ANSI C specifies that `rand(3)` return an `int`
  - `RAND_MAX` is no larger than `INT_MAX`
  - ANSI C requires only that `INT_MAX` be greater or equal 32767 — a simulation of 10^6 realizations will repeat the sequence about 30 times
  - usually not a problem on 32-bit machines
- ANSI C reference implementation
  - LC with a sub-optimal choice of a, c, and m
  - botched by implementors who try to “improve”

More Problems with `rand(3)`

- LC PRNGs are not free of sequential correlation on successive calls
  - problem when generating random numbers in many dimensions:
    - plot points in k-dimensional space (between 0 and 1 in each dimension)
    - "random numbers fall mainly in the planes" [G. Marsaglia], i.e., they will lie on less than $m^{1/k}$ $(k-1)$-dimensional planes
    - $m = 32768$ (bad), $k = 3 \Rightarrow$ less than 32 planes
    - $m \approx 2^{32}$ (good) , $k = 3 \Rightarrow$ about 1600 planes
- "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random." [NR]

So how is `glibc rand(3)` doing?

- not an LC generator (cf. `man random`)
  - do read the `man` pages for `rand(3)` and `random(3)`!
- “The period of this random number generator is very large, approximately 16*((2**31)-1)” [`man random`]
  - not really very large
- `DIEHARD` tests on `rand(3)`: so-so

−	−	+	−	+	−	+	+	+
−	−	+	+	+	−	+	+	+

- many (most?) other generators are no better, e.g. `ran2` from NR is only a little bit better, Sun `f77` (old?) is really lousy.

Are there Any Good PRNGs?

- yes, some pass all the tests with flying colors:
  - KISS
  - The "Mother of all random number generators"
  - Multiply-With-Carry $x_n = ax_{n-1} + c(\mathrm{mod} 2^{32})$
  - Mersenne Twister
- do check!
- check by yourself
  - generators improve
  - tests get tougher
- if you are really serious, develop your own tests

Implementation of Good PRNGs

- from
  http://www.cs.yorku.ca/~oz/marsaglia-rng.htm

```
#define znew (z=36969*(z&65535)+(z>>16))
#define wnew (w=18000*(w&65535)+(w>>16))
#define MWC ((znew<<16)+wnew )
#define SHR3 (jsr^=(jsr<<17), \
              jsr^=(jsr>>13), \
              jsr^=(jsr<<5))
#define CONG (jcong=69069*jcong+1234567)
#define FIB ((b=a+b),(a=b-a))
#define KISS ((MWC^CONG)+SHR3)
```

- not the last word in software engineering, one can do better with little effort

Seeding PRNGs

- fixed seed very useful for debugging
- `srand(time(NULL))`
  - get a MOSIX cluster, run

    `for i in ‘seq 1 10‘; do (./sim &); done`
- `srand(clock())`
  - will be surprisingly similar from run to run: many runs will only use a few seeds
- call `gettimeofday(2)` , use low-order bits of microseconds, mix with `pid` , etc.
  - good, but note that `gettimeofday(2)` is not `POSIX` , not guaranteed to work
- entropy — also non-portable!

Monte Carlo Simulations I

- Applications
  - throwing dice or spinning wheels (if you are into getting rich, quickly)
  - modelling price fluctuations in various markets (if you are hired to help the rich keep their money)
  - studying Brownian motion, diffusion, cosmic ray propagation, etc. (if getting rich is not the objective)
  - designing new computers and/or algorithms for efficient management of resources under uncertain workloads (strictly for common good, of course)

Monte Carlo Simulations II

- Technique
  - generate "realizations" based on random sequences
  - compute the expectation value of the result (payoff, displacement, etc.) and the "likely" deviation as an error estimate
  - equivalent to integration over realizations
- example: $\int f(x)dx$ — generate random points (x, y), count those for which $y < f(x)$
- example: $\int f dV$ over a complicated shape V — enclose V into a simple shape W that can be easily sampled, compute $\int g dW$ where $g = f$ in V, $g = 0$ outside of V

Non-Uniform RNG: Transformations

- fundamental transformation law of probability:

$$|p_y(y)dy| = |p_x(x)dx| \;\text{ or }\; p_y(y) = p_x(x)|\frac{dx}{dy}|$$

- for x uniform on [0,1) $p_x(x) = 1$ for $0 \leq x < 1$, zero otherwise
- for $p_y(y) = f(y)$ we must solve $dx/dy = f(y)$ to obtain, with $F(y)$ being the CDF of y

$$y(x) = F^{-1}(x)$$

- in multiple dimensions, with Jacobian $J_{ij} = \partial x_i/\partial y_j$:

$$p_y(\vec{y})d\vec{y} = p_x(\vec{x})d\vec{x} = p_x(\vec{x})|J_{ij}|d\vec{y}$$

Normal Deviates: Box-Muller

Generate normal deviates $y_{1,2}$ from uniform (on [0,1)) $x_{1,2}$:

- the transformation:

$$
\begin{aligned}
y_1 &= \sqrt{-2\ln x_1}\cos 2\pi x_2, \\
y_2 &= \sqrt{-2\ln x_1}\sin 2\pi x_2,
\end{aligned}
$$

  or, equivalently

$$
\begin{aligned}
x_1 &= \exp\left[-\frac{1}{2}\left(y_1^2 + y_2^2\right)\right], \\
x_2 &= \frac{1}{2\pi}\arctan\frac{y_2}{y_1}
\end{aligned}
$$

Normal Deviates: Box-Muller (cont.)

- the Jacobian:

$$|J| = -\left[\frac{1}{\sqrt{2\pi}} \mathrm{e}^{-y_1^2/2}\right]\left[\frac{1}{\sqrt{2\pi}} \mathrm{e}^{-y_2^2/2}\right]$$

- a further trick: pick (v_1, v_2) inside the unit circle (using rejection)
  - $x_1 = R^2 = v_1^2 + v_2^2$, $x_2 = \frac{1}{2\pi} \arctan \frac{v_2}{v_1}$
  - $\cos 2\pi x_2 = v_1/R$, $\sin 2\pi x_2 = v_2/R$
  - get two normal deviates

$$\begin{aligned} y_1 &= 2\sqrt{-\ln R}(v_1/R), \\ y_2 &= 2\sqrt{-\ln R}(v_2/R) \end{aligned}$$

  - no need to compute $\sin$ or $\cos$!

Non-Uniform RNG: Rejection

- What if we do not know the inverse CDF?
- pick $f(x)$ such that
  - $p(x) < f(x)$
  - $F(x) = \int_0^x f(x)dx$ is known and analytically invertible
  - $\int_0^\infty f(x)dx = A$
- generate y_1 uniform in [0,A)
- compute $x = F^{-1}(y_1)$
- generate y_2 uniform on $[0, f(x))$
- accept x if $y_2 \leq p(x)$, reject if $p(x) < y_2$

Variance Reduction Techniques

- complexity analysis for integration using N uniformly distributed random points in an n-dimensional space:
  - each point adds linearly to the accumulate sum that will become the function average
  - it also adds linearly to the accumulated sum of squares that will become the variance
  - the estimates error comes from the square root of the variance, hence $N^{-1/2}$ — slow convergence!
- antithetic variables
- non-uniform sampling (importance, stratification)

Can We beat the Square Root?

- $N^{-1/2}$ is not inevitable
  - choose points on a Cartesian grid, sample each grid point exactly once in whatever order: N^{-1} or better convergence
  - problem: must decide in advance how fine the grid has to be, commit to sample all the points
- can we pick sample points “at random” yet spread them out in some self-avoiding way, eliminating the “local clustering” of uniformly random points?
- another context: search an n-dimensional volume for a point where some locally computable condition holds
  - we want to move smoothly to finer scales, and do better than random

Quasi-Random Sequences

- sequences of n-tuples that fill n-dimensional space more uniformly than uncorrelated random points
- not random at all, “maximally avoiding” each other
- Halton sequence: algorithm
  - for H_j write j as a number in base b, where b is prime
  - reverse the digits and put a radix point in front
  - example: $j = 17$, $b = 3$: 17 base 3 is 122, $H_j = 0.221$ base 3
- Halton sequence: intuition
  - every time the number of digits in j increases, H_j becomes finer-meshed
  - the fastest-changing digit in j controls the most significant digit of H_j

Quasi-Monte Carlo

- many quasi-random (a.k.a. low-discrepancy) sequences: Halton, Faure, Sobol, Niederreiter, Antonov-Saleev (efficient variant of Sobol) — details in NR and references therein
  - non-trivial mathematics involved
- complexity: $(\ln N)^n/N$, i.e., almost as N^{-1}, with a bit of curvature
- can be orders of magnitude better convergence
- tricky to implement
- beware of USPTO!

`drivers/char/random.c`

- Gathering “environmental noise”
  - inter-keypress timings from the keyboard
  - mouse interrupt timings and position as reported by hardware
  - inter-interrupt timings
    - not all interrupts are suitable (consider timer)
  - finishing time of block requests
- maintain an “entropy pool” mixed with a CRC-like function (fast enough to do on every interrupt)
  - random bytes are obtained by taking SHA
    - message of length $< 2^{64}$ bits $\Rightarrow$ 160-bit “digest”
  - keep an estimate of “true randomness” in the pool, if zero an attacker has a chance if he cracks SHA

`/dev/random` and `/dev/urandom`

- `/dev/random`
  - will only return a maximum of the number of bits of randomness contained in the entropy pool
- `/dev/urandom`
  - will return as many bytes as are requested, without giving the kernel time to replenish the pool
  - acceptable for many applications
  - very random, passes `DIEHARD` (Ts'o: suitable for one-time pads)
- `void get_random_bytes(void *buf, int n);`
  - for use within the kernel
- not very fast, good for seeding, non-portable

Unpredictable over Reboots

on shutdown:

```
seed=/var/run/random-seed
touch $seed; chmod 600 $seed
pool=/proc/sys/kernel/random/poolsize
[ -r $pool ] && bytes=‘cat $pool‘ || bytes=512
dd if=/dev/urandom of=$seed count=1 bs=bytes
```

on boot:

```
seed=/var/run/random-seed
[ -f $seed ] && cat $seed > /dev/urandom
touch $seed; chmod 600 $seed
pool=/proc/sys/kernel/random/poolsize
[ -r $pool ] && bytes=‘cat $pool‘ || bytes=512
dd if=/dev/urandom of=$seed count=1 bs=bytes
```

Other Interfaces

- sysctl(8)

```
# /sbin/sysctl -A 2>/dev/null | grep rand
kernel.random.uuid = e005e4a2-fde0-423d-8734-0a
kernel.random.boot_id = 68eedc43-9a54-4a42-80de
kernel.random.write_wakeup_threshold = 128
kernel.random.read_wakeup_threshold = 8
kernel.random.entropy_avail = 4096
kernel.random.poolsize = 512
```

Literature

- D. E. Knuth, “The Art of Computer Programming,” v. 2, Ch. 3, 3rd ed., Addison Wesley
- W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, “Numerical Recipes”, Cambridge University Press
- G. Marsaglia, A. Zaman, “Some Portable Very-Long-Period Random Number Generators,” Computers in Physics, v. 8, p. 117 (1994)

Formal Definition of “Random”

Assume we generated a random sequence

$$\langle U_n \rangle = U_0,\ \ U_1,\ \ U_2,\ \ \ldots$$

of real numbers, $0 \leq U_n < 1$. The sequence is deteministic, but “behaves randomly”. What does that mean? Quantitative definition of “random behaviour” shall list a relatively small number of mathematical properties:

- each property shall satisfy our intuitive notion of random sequence
- the list shall be complete enough for us to agree that *any* sequence with these properties is “random”.

(TAOCP:3.5).

Equidistributed Sequences

- if u and v are real numbers, $0 \le u < v \le 1$, and U is uniformly distributed on [0,1), then $P(u \le U < v) = v - u$.
- for sequence $\langle U_j \rangle$, $0 \le j < n$ let $\nu(n)$ be the number of U_j such that $u \le U_j < v$; if for all u and v, $0 \le u < v \le 1$ we have

  $$\lim_{n\to\infty} \frac{\nu(n)}{n} = v - u$$

  then the sequence $\langle U_j \rangle$ is *equidistributed*
- if $\nu(n)$ is the number of cases when statement $S(j)$ is true, we say that $S(n)$ is true with probability λ, $Pr(S(n)) = \lambda$, if

  $$\lim_{n\to\infty} \frac{\nu(n)}{n} = \lambda$$

Is Equidistributed Random?

No. For instance, it is easy to construct an equidistributed sequence where a number less than $1/2$ will always be followed by a number that is greater than $1/2$. Such a sequence can be constructed from two other equidistributed sequences, $\langle U_n \rangle$ and $\langle V_n \rangle$:

$$\langle W_n \rangle = \frac{1}{2}U_0,\ \ \frac{1}{2}(1 + V_0),\ \ \frac{1}{2}U_1,\ \ \frac{1}{2}(1 + V_1),\ \ ...$$

The resulting sequence is not random by any reasonable definition (but keep your patience...)

k-distributed Sequences

- 2-distributed sequence:

$$Pr(u_1 \le U_n < v_1 \bigcap u_2 \le U_{n+1} < v_2) = (v_1 - u_1)(v_2 - u_2)$$

- k-distributed sequence

$$Pr(u_1 \le U_n < v_1 \bigcap ... \bigcap u_k \le U_{n+k-1} < v_k) = \prod_{j=1}^{k} (v_k - u_k)$$

- an equidistributed sequence is 1-distributed
- a k-distributed sequence is $(k-1)$-distributed
  - proof: set $u_k = 0$, $v_k = 1$
- a sequence is ∞-distributed if it is k-distributed for all positive integers k

Integer (b-ary) Sequences

- $\langle X_n \rangle = X_0, \;\; X_1, \;\; ...$ is a b-ary sequence if each X_n is one of the integers 0, 1, ... , $b-1$.
- a k-digit b-ary number is $x_1 x_2 ... x_k$ where $0 \le x_j < b$ for $1 \le j \le k$.
- a b-ary sequence is k-distributed if *for all* b-ary $x_1 x_2 ... x_k$

$$Pr(X_n X_{n+1} ... X_{n+k-1} = x_1 x_2 ... x_k) = 1/b^k$$

- if $\langle U_n \rangle$ is k-distributed then $\langle \lfloor bU_n \rfloor \rangle$ is also k-distributed (b-ary)
  - k-distributed (b-ary) is $(k-1)$-distributed (b-ary)
- π is a 10-ary sequence: is it ∞-distributed? 1-distributed? No one knows...

Properties of b-ary Sequences

- a 1000000-distributed binary sequence will have runs of a million zeroes in a row!
- a 1000000-distributed [0,1) sequence will have runs of a million consecutive values less than 1/2!
- this will happen only $(1/2)^{1000000}$ of the time, but it **will** happen
- intuitively, this will also occur in any “truly random” sequence
- effect for simulations:
  - if it happens, one would complain about the RNG
  - if it doesn’t, then the sequence is not random and will not be suitable for some other applications
- truly random sequences exhibit local non-randomness!

Further Generalizations

A [0,1) sequence $\langle U_n \rangle$ is (m,k)-distributed if

$$Pr(u_1 \leq U_{mn+j} < v_1 \bigcap ... \bigcap u_k \leq U_{mn+j+k-1} < v_k) = \prod_{j=1}^{k} (v_k - u_k)$$

for all choices of real numbers u_r, v_r, such that $0 \leq u_r < v_r \leq 1$ for $1 \leq r \leq k$, and for all integers j such that $0 \leq j < m$.

- $m = 1$: a k-distributed sequence
- $m = 2$: k-tuples starting in even positions have the same density as k-tuples starting in odd positions

Further Generalizations (cont.)

- an (m, k)-distributed sequence is (m, l)-distributed for $1 \leq l \leq k$
- an (m, k)-distributed sequence is (d, k)-distributed for all divisors d of m
- **Theorem:** An ∞-distributed sequence is (m, k)-distributed for all positive integers m and k [proof in TAOCP].
- **Corollary:** An ∞-distributed sequence will pass all the interesting statistical tests.
- **Generation:** Yes, there are algorithms (including at least one by Knuth, cf. TAOCP).

Is ∞-distributed Random?

- **Simple definition D1:** A [0,1) sequence is defined to be "random" if it is ∞-distributed
  - uncountably many realizations are not even 1-distributed
  - but a random sequence is ∞-distributed with probabilty one
- **Formal definition D2:** A [0,1) sequence $\langle U_n \rangle$ is random if, whenever property $P(\langle V_n \rangle)$ holds true with probability one for a sequence $\langle V_n \rangle$ of independent samples of a uniform random variable, then $P(\langle U_n \rangle)$ is also true.
- Are the two definitions equivalent?

Analysis of D1 and D2

- **D1** only deals with $n \to \infty$, the first million numbers in a ∞-distributed series may be zero — can such a sequence be considered random?
- With probability one, a truly random sequence contains infinitely many runs of numbers less than ϵ, for any $\epsilon > 0$. This can also happen at the beginning...
- Subsequence $\langle U_{n^2} \rangle$ should also be random. "It ain't necessarily so": if $U_{n^2} = 0$ for all n, the counts $\nu(n)$ are changed by $\sqrt{n}$ at most, and $\lim_{n\to\infty} \nu(n)/n$ does not change.

Modifications of D1

- **D3:** A [0,1) sequence is said to be “random” if each of its infinite subsequences is ∞-distributed.
  - does not work either: any equidistributed sequence has a monotonic subsequence $U_{s_0} < U_{s_1} < U_{s_2} < \ldots$
- restrict the subsequences so that they could be defined by a person who does not look at U_n before deciding whether to include it in the subsequence
- **D4:** A [0,1) sequence is said to be “random” if, for every effective algorithm that specifies an infinite sequence of distinct nonnegative integers $\langle s_n \rangle$, the corresponding subsequence $\langle U_{s_n} \rangle$ is ∞-distributed.

Computability Issues

- Computable rules: $\langle f_n(x_1, ..., x_n)\rangle$, can be 0 or 1. X_n is in subsequence if and only if $f_n(X_0, ..., X_{n-1}) = 1$.
- Let's restrict ourselves to computable rules for generating subsequences. Computable rules don't deal well with arbitrary real inputs though — better switch to integer sequences:
- **D5:** A b-ary sequence is said to be "random" if every infinite subsequence defined by a computable rule is 1-distributed. A [0,1) sequence $\langle U_n\rangle$ is said to be "random" if $\langle\lfloor bU_n\rfloor\rangle$ is "random" for all integers $b \geq 2$.
  - NB: D5 says, 1-distributed, not ∞-distributed!
  - ∞-distributivity can be derived from D5.
- Remaining problem: $\langle s_n\rangle$ is monotonic for computable rules — does not correspond to D4

Mathematical Randomness — At Last

- **D6:** A b-ary sequence is said to be "random" if, for every effective algorithm that specifies an infinite sequence of distinct nonnegative integers $\langle s_n \rangle$ and the values $X_{s_0}, X_{s_1}, ..., X_{s_{n-1}}$, the corresponding subsequence $\langle X_{s_n} \rangle$ is "random" in the sense of D5. A [0,1) sequence $\langle U_n \rangle$ is said to be "random" if $\langle \lfloor bU_n \rfloor \rangle$ is "random" for all integers $b \geq 2$.
- Knuth's conjecture: D6 "meets all reasonable philosophical requirements for randomness".
- TAOCP for more...