

Random Number Generators

Random Sequences

There is no such thing as a random number! A *random sequence* is a sequence of values x_0, x_1, \dots chosen from a *set* $\{s_1, s_2, \dots, s_M\}$ of size M with a given *probability distribution* p_1, p_2, \dots, p_M of the set members.

The set can be finite, countably infinite, or continuous: for a continuous set the probability distribution is a continuous function.

Successive coin tosses: $M = 2$, $s_1 = \text{Heads}$, $s_2 = \text{Tails}$, and $p_1 = p_2 = \frac{1}{2}$. Random sequence: H, H, T, H, T, H, H, H, H, H, T, H, ...

Radioactive decay times of a collection of unstable nuclei at time $t = 0$: continuous set $0 \leq t < \infty$ with probability distribution

$$p(t) = \frac{1}{\tau} e^{-t/\tau}, \quad \int_0^{\infty} p(t) dt = 1. \quad (1)$$

Random sequences have the following properties:

- *Non-periodic* — the sequence never repeats.
- *Uncorrelated* — x_n is *independent* of x_{n-1} and preceding values, and cannot be predicted in advance.
- *Good Statistics* — the sequence can be treated like experimental data and compared with the underlying probability distribution (the theoretical model). For example, one can apply the chi square goodness of fit test.

Random sequences have many uses:

- *Simulation* of natural phenomena — random numbers make things realistic.
- *Sampling* of huge sets to provide insight on *typical* cases.
- *Numerical solution* of complex problems using random number algorithms.
- *Computer programming*: Programs can be tested on random inputs. Computer graphics. Cryptography.
- *Decision making* in finance, politics, lotteries, warfare.
- *Recreation*: computer games use random number extensively.

Unfortunately, *true random sequences* are hard to come by! Example: 32-bit unsigned integers. $M = 2^{32}$, $\{s\} = 0, 1, 2, \dots, 2^{32}-1$. $p_i = 2^{-32}$.

Can you spot the bottleneck in this algorithm?

Pseudo-random sequences

A true random sequence is non periodic. Successive numbers in the sequence cannot be predicted a priori. A deterministic computer program cannot generate a true random sequence, because successive numbers are determined by an algorithm on a finite set.

Algorithm 1Random 32-bit Integer Sequence using Coin Tosses

- Repeat as many times as desired:
 - Set $i \leftarrow 0$.
 - Repeat 32 times:
 - Toss a coin.
 - If Heads, set i -th bit to 1, else set i -th bit to 0.
 - Let $i \leftarrow i + 1$.
 - Use random integer.
-

A deterministic computer algorithm can generate random sequences that behave like truly random sequences in particular applications. A simple example is the Quick and Dirty algorithm from Numerical Recipes[1].

Algorithm 2Quick and Dirty Generator

- Choose any *seed* x_0 in $[0, 2^{32}-1]$ for the pseudo-random sequence.
 - `unsigned long idum = x0;`
 - Repeat as many times as desired:
 - `idum = 1664525L * idum + 1013904223L; // will overflow like crazy - don't worry!`
 - Use `idum` as next pseudo-random integer.
-

This takes care of the bottleneck, but it does *not* give a random sequence:

- *Strictly Periodic* with period 2^{32} .
- *Correlated* — x_n can be predicted precisely from x_{n-1} — *Deterministic*, not random!
- *Reasonably Good Statistics* but *only* for applications which require fewer than $2^{32} = 4294967296$ random integers.

Pseudo-random sequences are *fast*, and *if carefully used*, can mimic many desirable statistical properties of real random sequences in particular applications.

Finding good random number generator algorithms is not easy. A famous example of a bad generator is IBM's RANDU <http://en.wikipedia.org/wiki/RANDU>, which was widely used for many years. There is a vast literature on algorithms tests of randomness, see Numerical Recipes[1] and L'Ecuyer's review[2].

Linear congruential generators

The simplest and most widely used algorithms are based on *linear congruent* schemes

$$x_{n+1} = (ax_n + b) \bmod c, \tag{2}$$

where the *multiplier* a , the *increment* b , and the *modulus* c , are carefully chosen positive constants which have been shown by extensive testing to give sequences with good statistical properties.

If we start with a positive *seed* $0 \leq x_0 < c$, this algorithm will generate positive numbers $0 \leq x_n < c$.

Uniform Deviates

A *sequence of uniform deviates* is a sequence of real numbers in the unit interval $[0, 1]$ with probability distribution

$$p(x) = 1, \quad \int_0^1 p(x) dx = 1. \quad (3)$$

On a digital computer, uniform deviates are approximated by real numbers, usually `doubles` in scientific applications. The end point numbers 0 and 1 may or may not be generated by a particular algorithm.

Using a linear congruential generator for example, and dividing each number by the modulus yields real numbers that are uniformly distributed in the unit interval $0 \leq (x_n/c) < 1$.

C++ Program to test various random deviate generators

The following program codes various algorithms and performs simple tests on the quality of random sequences generated by them.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

Seeding the random number generator

It is critically important to use a different seed number when using a pseudo-random number generator. If you use the same seed you will get the same sequence!

A popular strategy for seeding a random number generator is to use the current time in seconds according to the system clock. The C++ `<ctime>` header defines an object `tm` that stores time information in the following data members:

```
int tm_sec    // seconds after the minute [0, 60]
int tm_min    // minutes after the hour [0, 59]
int tm_hour   // hours since midnight [0, 23]
int tm_mday   // day of the month [1, 31]
int tm_mon    // months since January [0, 11]
int tm_year   // years since 1900
```

The number of seconds elapsed since midnight January 1, 1900 is approximately

$$\text{sec} + 60[\text{min} + 60[\text{hour} + 24[(\text{mday} - 1) + 30.42[\text{mon} + 12[\text{year}]]]]]] . \quad (4)$$

This formula does not take anomalies such as leap years and leap seconds into account. Using the number of seconds elapsed is not a good idea because this number increases steadily at the rate of one per second. Anderson[3] suggests using the integer

$$t = (\text{year mod } 100) + 100[\text{mon} + 12[(\text{mday} - 1) + 31[\text{hour} + 24[\text{min} + 60[\text{sec}]]]]] \quad (5)$$

which falls in the range $0 \leq t \leq 32,214,079,999 = 2^{31.58}$. Although t increases monotonically with time, it does at the rate of at least $100 \times 12 \times 24 \times 60 = 53,568,000$ per second, with bigger jumps when the other quantities change.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```
unsigned timeSeed() {
    time_t t;                // usually an unsigned long int
    time(&t);                // get seconds since Jan 1, 1900
    tm* pt = gmtime(&t);     // Universal (Greenwich Mean) Time

    return pt->tm_year % 100 + 100 * (pt->tm_mon + 12 * ((pt->tm_mday - 1)
        + 31 * (pt->tm_hour + 24 * (pt->tm_min + 59 * pt->tm_sec))));
}
```

Linear congruential algorithm

The following are two of the many linear congruential generators listed in Numerical Recipes[1], one with a rather short period and one with a somewhat longer period.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```
double poor(unsigned& seed, bool set) {
    const unsigned IM = 6075, IA = 106, IC = 1283;
    if (set) {
        if (seed >= IM) seed %= IM;    // make it in range 0..IM-1
    } else {
        seed = (IA * seed + IC) % IM; // linear congruential algorithm
    }
    return seed / double(IM);        // make range 0.0 ... 1.0
}
```

```
double good(unsigned& seed, bool set) {
    const unsigned IM = 714025, IA = 1366, IC = 150889;
    if (set) {
        if (seed >= IM) seed %= IM;
    } else {
        seed = (IA * seed + IC) % IM;
    }
    return seed / double(IM);
}
```

C/C++ standard library generator

The following is the standard C/C++ generator defined in `<cstdlib>`. Beware: on some machines it has an unacceptably short period!!

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```
double rand(unsigned& seed, bool set) {
    if (set)
        srand(seed);
    else
        seed = rand();
    return seed / (RAND_MAX + 1.0);
}
```

Park-Miller algorithm

The following Park-Miller generator^[4] is very popular and widely used. It produces sequences of high quality and has a long period $2^{31} - 1 = 2147483647$. The following implementation uses Schrage's algorithm, which works on old machines with 16-bit `int`'s.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```
double ranf(unsigned& seed, bool set) {
    const int IA = 16807, IC = 2147483647, IQ = 127773, IR = 2836;
    int iseed = seed;
    if (iseed <= 0)
        iseed += IC;
    if (!set) {
        int h = iseed / IQ;
        int l = iseed % IQ;
        iseed = IA * l - IR * h;
    }
    if (iseed <= 0)
        iseed += IC;
    return (seed = iseed) / double(IC);
}
```

Quick and dirty algorithm

The following implements the Quick and Dirty algorithm from Numerical Recipes^[1].

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```

double ranq(unsigned& seed, bool set) {
    static unsigned long idum;
    const double TWO_POWER_32 = 4294967296.0;
    if (set) {
        idum = (unsigned long) seed;
        return idum / TWO_POWER_32;
    }
    idum = 1664525L * idum + 1013904223L;
    seed = int(idum);
    return idum / TWO_POWER_32;
}

```

Marsaglia Xorshift algorithm

The new Third Edition of Numerical Recipes proposes a high-quality generator based on the Xorshift algorithms of Marsaglia[5]. It uses 64-bit unsigned integer arithmetic that is standard on all modern machines. It has an extremely long period $\approx 3.138^{57}$. Note the use of the C/C++ bitwise xor (^) and right shift (>>) operators.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```

#ifdef _MSC_VER    /* Microsoft C++ */
    typedef unsigned __int64 Ullong;    // 64-bit unsigned integer
#else
    /* Macintosh, Linux */
    typedef unsigned long long Ullong;
#endif

inline Ullong int64(Ullong& u, Ullong& v, Ullong& w) {
    u = u * 2862933555777941757LL + 7046029254386353087LL;
    v ^= v >> 17; v ^= v << 31; v ^= v >> 8;
    w = 4294957665U*(w & 0xffffffff) + (w >> 32);
    Ullong x = u ^ (u << 21); x ^= x >> 35; x ^= x << 4;
    return (x + v) ^ w;
}

double ranx(unsigned& seed, bool set) {
    static Ullong u = 0, v = 4101842887655102017LL, w = 1;
    if (set) {
        Ullong j = seed;
        v = 4101842887655102017LL;
        w = 1;
        u = j ^ v;
        int64(u, v, w);
        v = u;
        int64(u, v, w);
        w = v;
        seed = int(int64(u, v, w));
    } else {

```

```

    Ullong x = int64(u, v, w);
    seed = unsigned(x);
    return 5.42101086242752217E-20 * x;
}
}

```

Testing generator quality

The following function performs three simple tests:

- Check whether the period is less than 10^7 .
- Compute the χ^2 per degree of freedom by binning a 10^7 element sequence in 10^4 bins.
- Generating data for a two-dimensional eyeball quality test.

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic2/generators.cpp> _____

```

void test(double gen(unsigned&, bool), string name, string remark) {
    cout << " " << name << "(unsigned&, bool) -- " << remark << endl;
    unsigned seed = timeSeed();
    gen(seed, true);
    cout << "    using system-clock-time-based seed " << seed << endl;

    cout << "    checking period ... " << flush;
    int i = 0, start;
    while (true) {
        gen(seed, false);
        if (i++ == 0)
            start = seed;
        else {
            if (seed == start) {
                cout << "= " << i << endl;
                break;
            } else if (i > 10000000) {
                cout << "larger than 10,000,000" << endl;
                break;
            }
        }
    }
}

int tries = 10000000;
const int BINS = 10000;
cout << "    binning " << tries << " tries in "
    << BINS << " bins ..." << flush;
int bin[BINS];
for (int b = 0; b < BINS; b++)
    bin[b] = 0;

```

```

for (int i = 0; i < tries; i++) {
    int b = int(gen(seed, false) * BINS);
    if (b == BINS)
        --b;
    ++bin[b];
}
double chiSqr = 0;
double expect = tries / double(BINS);
for (int b = 0; b < BINS; b++) {
    double diff = bin[b] - expect;
    chiSqr += diff * diff;
}
chiSqr /= expect;
cout << " chi-square/d.o.f. = " << chiSqr/(BINS-1) << endl;

string fileName = name + ".data";
cout << "   writing eyeball test data to " << fileName << endl;
ofstream dataFile(fileName.c_str());
for (int i = 0; i < 10000; i++)
    dataFile << gen(seed, false) << '\t' << gen(seed, false) << '\n';
dataFile.close();
cout << endl;
}

int main() {
    cout << " Testing random number generators\n"
         << " -----\n\n";
    test(poor, "poor", "from Numerical Recipes period 6,075");
    test(good, "good", "from Numerical Recipes period 714,026");
    test(rand, "rand", "rand() from C++ Standard Library");
    test(ranf, "ranf", "Park-Miller generator");
    test(ranq, "ranq", "Quick and Dirty generator");
    test(ranx, "ranx", "Marsaglia Xorshift generator");
}

```

Computing the Value of π

The following program estimates the value of π by generating a number N of points uniformly and randomly distributed inside the unit square. Suppose that N_{inside} of these points fall inside the circle inscribed in the square. If the points are uniformly distributed

$$\pi = \frac{4 \times \text{area of circle}}{\text{area of square}} = \lim_{N \rightarrow \infty} \frac{4N_{\text{inside}}}{N}. \quad (6)$$

Program 2: <http://www.physics.buffalo.edu/phy410-505/topic2/pi.cpp>

```

#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

double pi_estimate(
    const unsigned long points
) {
    unsigned long hits = 0;
    for (unsigned long i = 0; i < points; i++) {
        double x = rand() / (RAND_MAX + 1.0);
        double y = rand() / (RAND_MAX + 1.0);
        if ( (x - 0.5)*(x - 0.5) + (y - 0.5)*(y - 0.5) < 0.25 )
            ++hits;
    }
    return 4 * double(hits) / double(points);
}

void measure_pi(
    const unsigned long trials,
    const unsigned long points,
    double& average,
    double& std_dev
) {
    double sum = 0;
    double squared_sum = 0;
    for (unsigned long t = 0; t < trials; t++) {
        double pi = pi_estimate(points);
        sum += pi;
        squared_sum += pi * pi;
    }
    average = sum / trials;
    std_dev = squared_sum / trials - average * average;
    std_dev = sqrt(std_dev / (trials - 1));
}

int main() {

    cout << " Computing the value of pi using std::rand()" << endl;

    unsigned long trials, points;
    cout << " Enter number of trials: ";
    cin >> trials;
    cout << " Enter number of random (x,y) points per trial: ";
    cin >> points;

    clock_t start_time = clock();
    double average, std_dev;
    measure_pi(trials, points, average, std_dev);
    double secs = (clock() - start_time) / double(CLOCKS_PER_SEC);
}

```

```
cout << " pi = " << average << " +- " << std_dev << '\n'
    << " average - exact = " << average - 4*atan(1.0) << '\n'
    << " CPU time = " << secs << " secs" << endl;
}
```

Homework Problem

Modify the program `pi.cpp` to compute the volume of a sphere of radius $R = 1$ in three dimensions. Plot the standard deviation versus number of random points and fit this data to a model function. Repeat for a different random number generator and comment on any differences you observe.

Note: Determine the standard deviation just like you would in lab by repeating the measurement some fixed number of times!

References

- [1] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, “Numerical Recipes in C” (Cambridge University Press 1992), §7.1 Uniform Deviates, <http://www.nrbook.com/a/bookcpdf/c7-1.pdf>.
- [2] P. L’Ecuyer, “Random Number Generation”, Chapter 4 of *Handbook on Simulation* (Wiley 1998), <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/handsim.ps>
- [3] S.L. Anderson, “Random Number Generators on Vector Supercomputers and Other Advanced Architectures”, *SIAM Review*, **32**, 221 (1990), <http://dx.doi.org/10.1137/1032044>
- [4] S.K. Park and K.W. Miller, “Random number generators: good ones are hard to find”, *Communications of the ACM*, **31**, 1192 (1988), <http://doi.acm.org/10.1145/63039.63042>.
- [5] G. Marsaglia, “Xorshift RNGs”, *J. Stat. Software*, **8**, 1 (2003), <http://www.jstatsoft.org/v08/i14/>