An Universal Random Number Generator

Maurice.Clerc@WriteMe.com

2009-12-14

2010-10-10 Modified source code

1 Why this tool?

For stochastic optimisation, we need to generate pseudo-random numbers on a computer, according to a given distribution law. However, in any language, the list of predefined RNG (random number generators) is very short. In practice, we often have just the uniform one $U(0,1)$, and the standard normal one $\mathcal{N}(0,1)$. Sometimes, we may have a few more RNG: logistic, Cauchy, exponential, cosine, hyperbolic, etc., but it may be not enough. In particular, it is usually difficult to find a RNG when the distribution law is multimodal.

In practice, it means that researchers tend to adapt their models to the available RNG, as it should be the contrary. Here, we show how to define an "universal" RNG, by starting from $U(0,1)$.

2 Theoretical approach

Let f be a probability distribution. We consider it only on its support. It means that it is strictly positive. Theoretically, the method is then a simple two steps one:

- solve the differential equation $\frac{dx}{dh} = f(h)$. It gives a function $x(h)$ strictly increasing
- find the inverse $h(x)$

Then, if x follows an uniform distribution, h follows f. The idea of this "inversion method" is not new (see for example http://cg.scs.carleton.ca/~luc/chapter_two.pdf) but it is here simplified, and we give below a tool (C code) to apply it.

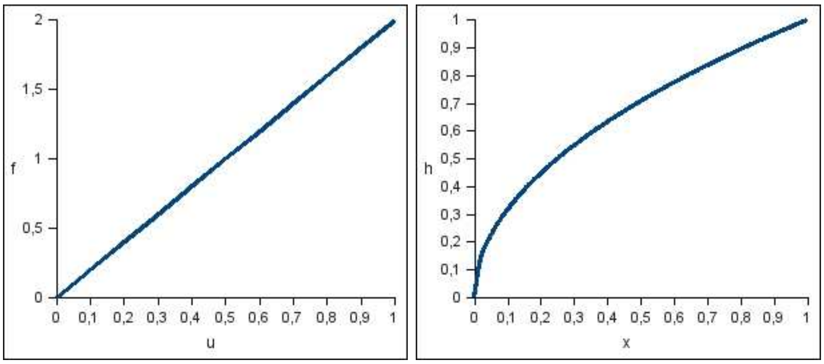

Figure 1: Linear distribution f, and "conversion" Uniform x to f, thanks to h

3 Examples

3.1 Uniform

We have $f(u) = 1$, on $[0,1]$. Then, $\frac{dx}{dh} = 1$. A solution is $x = h$, i.e. $h = x$.

3.2 Linear

For example $f(u) = 2u$, on $]0,1]$. Then $\frac{dx}{dh} = 2h$. A solution is $x = h^2$. The inverse is $h = \sqrt{x}$. On the figure 1 (right), we can intuitively see what happens. An uniform distribution for x indeed implies a non uniform one for h, whose density is increasing on $]0,1]$.

3.3 Gaussian

Here $f(u) = \frac{1}{\sqrt{2\pi}} e^{-\frac{u^2}{2}}$. Then, and in order to have x always in $[0,1]$, a solution of $\frac{dx}{dh} = f(h)$ is

$$x(h) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_0^h e^{-t^2/2} dt$$

sometimes written as $x(h) = \frac{1}{2} erf\left(\frac{h}{\sqrt{2}}\right)$, where erf is the "error function". Unfortunately, it can not be expressed in terms of finite additions, subtractions, multiplications, and root extractions, and so must be either computed numerically or otherwise approximated. And in practice, we have to define a finite interval for u.

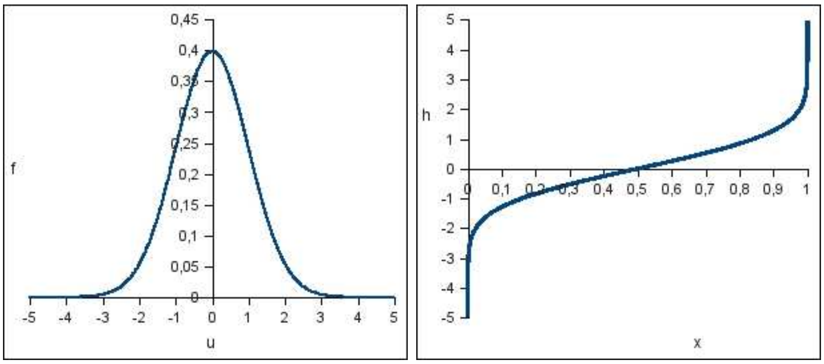

Figure 2: Normal distribution f, and "conversion" Uniform x to f, thanks to h

4 URNG in practice

4.1 Method

We suppose f defined on $[a, b]$. We have

$$x(h) = x(a) + \int_a^h f(u)\, du$$

As we can see below, we do not need to know $x(a)$. Let δ be $(b-a)/(n-1)$. We compute first f on n points $(u_1 = a, u = a + \delta_2, \ldots, u_n = b)$ and then

$$\widetilde{x}(h) = \delta \sum_{i=1}^{k} f(u_i)$$

with $u_{k-1} < h \leq u_k$. In practice, we compute the n values $\widetilde{x}(u_i)$. For the example 3.3, and for $n = 200$, we can plot the curve u_i vs $\widetilde{x}(u_i)$, which is an estimation of $h(x)$. As we can see on figure 3, this estimation is pretty good. Note that, anyway, it does not need to be very precise, for we will just use it to generate pseudo-random numbers. The method is the following:

- generate a random number r according to $U(0,1)$
- find i so that $\widetilde{x}(u_i) \leq r \leq \widetilde{x}(u_{i+1})$
- take $h = a + u_i + \delta(r - \widetilde{x}(u_i)) / (\widetilde{x}(u_{i+1}) - \widetilde{x}(u_i))$ as final result of the random number generation (linear interpolation)

We can check that it works by generating a big number of h and by plotting the histogram of frequencies for a given class size.

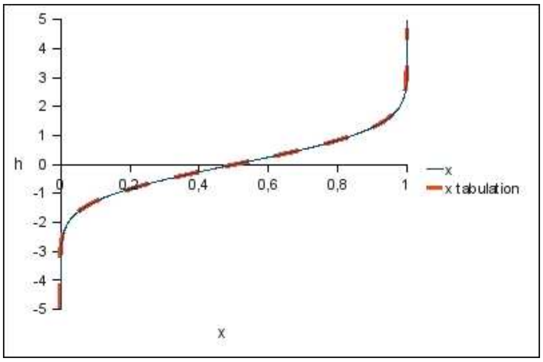

Figure 3: Normal distribution. Comparison between $h(x)$ and the estimation for $n = 200$

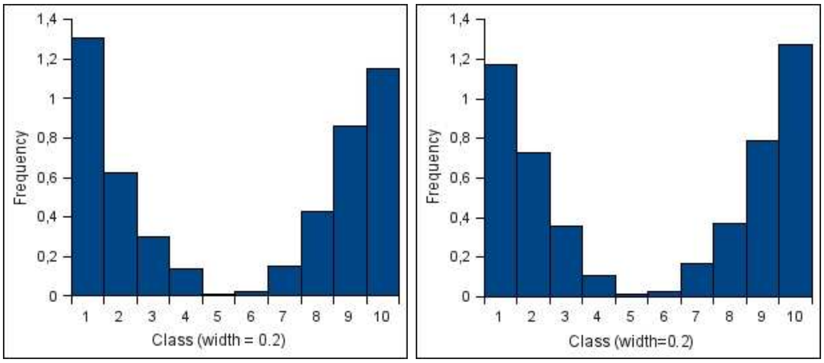

Figure 4: Parabolic distribution. Histogram for 2000 points and class size = 0.2. On the left, with the C function rand(). On the right, with the uniform RNG KISS, which is less biased

4.2 Parabolic example

Let us apply this method for the "parabolic" distribution defined by $f(u) = 3(u-1)^2/2$ on $[0,2]$. We generate 2000 points according to $U(0,1)$, and compute the 2000 corresponding h by using the above method, with $n = 101$. The histogram of these 2000 h values is given on the figure 4. We can see that the probability distribution is indeed very near of what we want. Note that the rand() function in C is biased (the density is not really uniform, and a bit too low near 1). So it may interesting to use a better uniform RNG, like KISS.

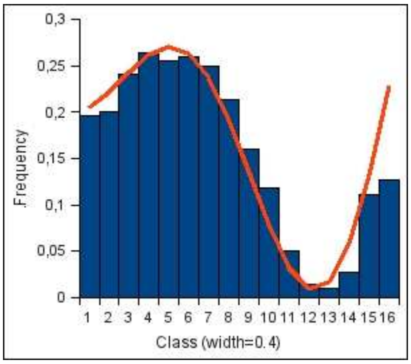

Figure 5: Multimodal distribution. Histogram for 2000 points (class size=0.4), and theoretical curve

4.3 Multimodal example

The wanted probability distribution is given by $f(u) = (5 + u * sin(u))/8\pi$. The histogram on the figure 5 shows that URNG generate random points that indeed follow this probabilistic law.

5 C code

The following C code can of course be improved. Note that, though, we do not need something very precise for anyway the probabilistic models used in stochastic optimisation are largely approximate.

```
/*  Copyright (C) Maurice 2009 <Maurice.Clerc@WriteMe.com>
Starting for the uniform distribution U(0,1),
generate random numbers according to any other
distribution f.
Last update 2010-10-10
Fixed a bug (wrong frequency near u=1)
*/
#define Nmax 201
#define Gmax 2000
#include "stdio.h"
#include "math.h"
#include <stdlib.h>
#include <time.h>
// To generate pseudo-random numbers with KISS
#define ulong unsigned long
```

```
#define RAND_MAX_KISS ((unsigned long) 4294967295)
// Subroutines ulong rand_kiss();
// For the pseudo-random number generator KISS
void seed_rand_kiss(ulong seed);
//
double alea (double a, double b);
double alea_f(double min, double delta);
double f(double u,int distrib);
// Global variables
double infinity = 1.e16;
double pi;
double x[Nmax];
double zero=1.e-16;
// Files
FILE * f_urng;
int main() {
double delta; int distrib; int g; int gener; double h; int i;
double max, min; int n;
pi=acos(-1); f_urng = fopen ("f_urng.txt", "w");
distrib=6;
// 0 = uniform on [a,a+1]
// 1 = linear on [0,1]
// 2 = normal on [-a,a], with a big enough so that
//     sum(f) on [-a,a] almost equal to 1
// 3 = parabolic on [0,2]
// 4 = bi-linear on [0,1]
// 5 = multimodal on [0, 2pi]
// 6 = Decreasing on [0,a], for initialisation in [0,a]^D
// a=max and D are hard coded in f() n=101; min=0;
// Support of f (the target distribution)
max=1; delta=(max-min)/(n-1); gener=1000;
// Number of random numbers to generate
// Estimate once x = sum(f)
x[0]=0; x[n]=1;
for(i=1;i<n;i++)
{ x[i]=x[i-1] + delta*f(min+i*delta,distrib); }
// Use the RNG for(g=0;g<gener;g++) {

    h=alea_f(min,delta);
    // Save for future use (histogram, in particular)
    fprintf(f_urng,"%f\n",h); }

return (0); }
//=========================================================
double alea (double a, double b) {
```

```
// random number (uniform distribution) in [a b]
double r;
//r=a + (double) rand () * (b - a)/RAND_MAX;
// It may be a good idea to use a better uniform RNG, like KISS
// for this one is biased
r=a+(double)rand_kiss()*(b-a)/RAND_MAX_KISS; return r;
}
//=================================================================
double alea_f(double min, double delta) {
double h; int k; double r;
// x is a global list, computed once
// Generate according to U(0,1)
r=alea(0,1);
// Find k so that x[k-1] <= r <=x[k] k=0;
while (r>x[k]) { k=k+1; }
//Linear interpolation
h=min+(k-1)*delta + delta*(r-x[k-1])/(x[k]-x[k-1]);
return h;
}
//=================================================================
double f(double u, int distrib) {
// Probability distribution.
// Integral on [min, max] must be equal to 1
// ADD YOUR OWN DISTRIBUTION
double a; double D; double p0,p1;
switch(distrib)
{ default: // Uniform on [a, a+1] return 1;

    case 1: // Linear on [0,1] return 2*u;
    case 2: // Normal return exp(-u*u/2)/sqrt(2*pi);
    case 3: // Parabolic on [0,2] return 1.5*(u-1)*(u-1);
    case 4: // Bi-linear on [0,1]
    a=0.2; p0=0.5; p1=2*(1-p0*a/2)/(1-a); // So that sum(f)=1
    if(u<=a) return p0*(1-u/a); return p1*(u-a)/(1-a);
    case 5: // Multimodal on [0,2pi] return (5+u*sin(u))/(8*pi);
    case 6: // Decreasing for initialisation in optimisation
    a=1; D=2;
    if(u>zero) return pow(u/a,1./D-1)/D; else return infinity;

}
}
//================================================== KISS
/* A good pseudo-random numbers generator
The idea is to use simple, fast, individually promising
generators to get a composite that will be fast, easy to code
```

```
have a very long period and  pass all the tests put to it.
The three components of KISS are
x(n)=a*x(n-1)+1 mod 2^32 y(n)=y(n-1)(I+L^13)(I+R^17)(I+L^5),
z(n)=2*z(n-1)+z(n-2) +carry mod 2^32 The y's are a shift register
sequence on 32bit binary vectors period 2^32-1; The z's are a simple
multiply-with-carry sequence with period 2^63+2^32-1.
The period of KISS is thus 2^32*(2^32-1)*(2^63+2^32-1) > 2^127
*/
static ulong kiss_x = 1; static ulong kiss_y = 2;
static ulong kiss_z = 4;
static ulong kiss_w = 8; static ulong kiss_carry = 0;
static ulong kiss_k;
static ulong kiss_m;
void seed_rand_kiss(ulong seed) { kiss_x = seed | 1;
kiss_y = seed | 2;
kiss_z = seed | 4; kiss_w = seed | 8; kiss_carry = 0; }
ulong rand_kiss() { kiss_x = kiss_x * 69069 + 1;
kiss_y ^= kiss_y << 13;
kiss_y ^= kiss_y >> 17; kiss_y ^= kiss_y << 5;
kiss_k = (kiss_z >> 2) + (kiss_w >> 3) + (kiss_carry >> 2);
kiss_m = kiss_w + kiss_w + kiss_z + kiss_carry;
kiss_z = kiss_w;
kiss_w = kiss_m; kiss_carry = kiss_k >> 30;
return kiss_x + kiss_y + kiss_w;
}
```