Lab # 2 - Generation of Random Variates

(Implementation in C – check the lecture slides and read carefully the subject)

1) Find "**Matsumoto Home page**" and the "**last C implementation**" of the original Mersenne Twister (MT)

In the first lab, you implemented various techniques for generating pseudorandom numbers and observed that achieving high-quality randomness is not straightforward. In this lab, we will use one of the best generators proposed for scientific applications in the 21st century (uniformly distributed in 623 dimensions and with a period of 2^{19937} numbers). Though not crypto secure, it will be the generator that you will use for this lab and the next ones.

You can find the current C implementation of the Mersenne Twister (MT) by searching for "Matsumoto Home Page" (Google search). Locate the 2002 version, which includes both explanations and the C source code.

Download the .tar file with the source code + expected output and readme (take the 32 bits version).

Compile and test if you obtain the expected output (for portability & **repeatability**). For the lab exercises, use genrand_int32 or genrand_real (1 or 2) functions. Untar and unzip the archive in a single command (Unix command: "`tar zxvf yourfile.tgz`") and use the example. Compare the result you obtain locally on your computer session with the expected output (reproducibility – see the README file proposed by Matsumoto and the expected output). From now, always use a fine generator like MT or other very good generators.

Once you have tested the bitwise reproducibility of this code, you will test the next functions of this lab by adding your code before the main function of Makoto's code, and you will modify the test functions of Makoto's to test your lab functions. Notice that functions dealing with real numbers use double precision.

2) Generation of uniform pseudorandom numbers between A and B

Implementation : Using the MT function providing numbers between [0..1], propose a C function named "uniform" with 2 parameters 'a' and 'b' (real numbers) and generate pseudorandom numbers between 'a' and 'b'. Test this function for temperatures between -89,2°C and 56,7°C.

3) Reproduction of discrete empirical distributions

Suppose we have field data with 3 classes: 500 observations in class A, 150 in class B and 350 in class C., giving the following probability distribution of 3 species (A, B and C): 50% for A, 15% for B et 35% for C.

To reproduce (simulate) a population of individuals with the same distribution, we can use the following procedure using a uniform pseudorandom number generator between 0 and 1.

When a random number is drawn, if it is strictly less than 0.5, the individual is assigned to category A (class A for species A). If the number drawn is between 0.5 and 0.65 (strictly), the individual is assigned to class B and if the number drawn is above 0.65, the individual is set in the C class. Figure 1 presents the histogram corresponding to this situation.

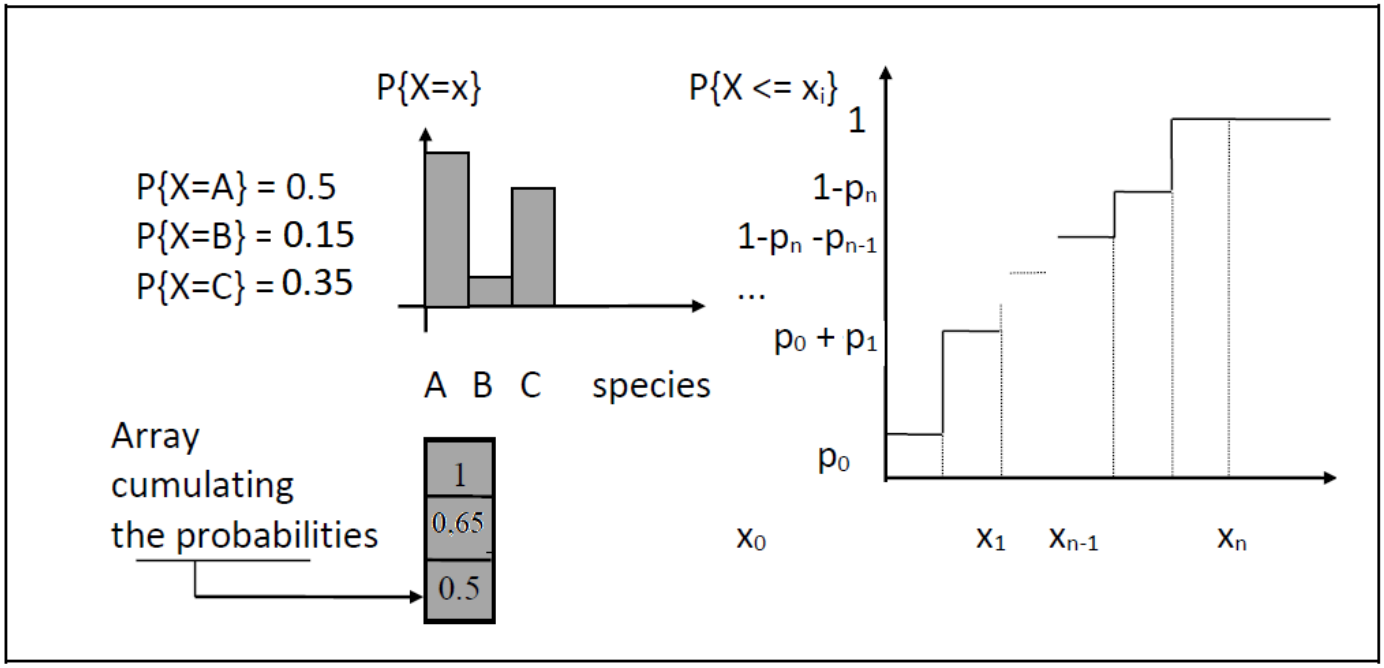

Figure 1. Sample histogram for 3 species and generalization to the CDF (cumulative density function)

In practice, we build an array with cumulative probabilities (corresponding to the cumulative distribution function (CDF) :

Array [1] = 0.5
Array [2] = 0.5 + 0.15 = 0.65
Array [3] = 0.5 + 0.15 + 0.35 = 1

The particular case of a 2 values histogram is similar to a biased coin tossing simulation.

a) **Implement and test** a program simulating this discrete distribution with the 3 classes A, B and C. Test it with 1 000, and 1 000 000 drawings. Cumulate the number of individual of each species in 3 variables and display the percentage obtained.

b) **Implement** a more generic function with the following input parameters: the size of an array of classes, then the array itself with the number of individuals observed in each class (see the lecture slides for an example – the HDL 'good' cholesterol and use these values to check this question).

   a. First compute the corresponding array with the probability of being in each class (distribution function) and test this.
   b. Then compute another array giving the cumulative probabilities (CDF). This function outputs the latter array. Test it.
   c. Test the whole function with data given in the slides (and/or with your own data) and check a simulated distribution with 1000 and 1000 000 drawings.

4) Reproduction of continuous distributions

It is possible to reproduce continuous distributions by inverting the distribution function. When drawing a pseudorandom number between 0 and 1, it is possible to obtain a number distributed according to a given continuous distribution function (F) supposing that the latter is reversible.

$$x = F^{-1} \text{ (Random number drawn)}$$

This technique, named *anamorphosis* is not completely generic, but it can be applied to many distribution laws (Binomial, Weibull, Uniform,…). For instance, the distribution function of an exponential distribution (negative exponential law) is given in equation (8) leading to the inverse law of equation (9) which has to be implemented. We can see it as an analogue of the Poisson distribution. Actually, the time between two events in a Poisson process (intuitively: the time between two rare events) follows an exponential distribution. For instance, the time between two radioactive disintegrations.

$$F(x) = \int_0^x \frac{1}{M} e^{-\frac{1}{M}z} dz = 1 - e^{-\frac{1}{M}x} \qquad (8)$$

$$RandomNumberDrawn = 1 - e^{-\frac{1}{M}x}$$

$$\Rightarrow 1 - RandomNumberDrawn = e^{-\frac{1}{M}x}$$

$$\Rightarrow \ln(1 - RandomNumberDrawn) = -\frac{1}{M}x$$

$$\Rightarrow x = -M \ln(1 - RandomNumberDrawn) \qquad (9)$$

Uniform law between A and B : $x = F^{-1}$ (Random number drawn) = A+(B-A) * Random number drawn

Mean = (B + A) / 2 Variance = 1/12 * $(B - A)^2$

Negative exponential law(Average) : $x = F^{-1}$ (Random number drawn) = - Mean $_x$ Log (1 - Random number drawn)

Mean is the average time between two events (arrivals for instance) – lambda is the rate (λ)

Mean = $1/\lambda$ Variance = $1/\lambda^2$

For instance: if you model the time between two radioactive decays, λ represents the average frequency of decays per unit time, and M is the average waiting time between two decays.

Example: λ = 0.1 per second → on average, one event every 10 seconds.

Therefore, λ is the rate parameter: it indicates how quickly events occur.
And its inverse, M = 1/λ, is the average interval between events.

Figure 2. Inverse function of the uniform and negative exponential law

Here is what you have to code:

a. Implement the negExp function accepting the mean as a parameter.
b. Test this function with a mean of 11. Check that the average obtained after drawing 1000 (then 1000 000), it should come close to 11. This supposes using fine random numbers between 0 and 1 in

equation (9) to obtain the correct distribution. For instance, such numbers could represent the inter-arrival time between two jobs submitted to a computing cluster.

c. Check this discrete distribution (like you would check if a dice is biased, but now the distribution is not uniform). Use an array with 20 bins and test the frequency of numbers between 0. and 1., between 1. and 2.,… and between 19. and 20 ; above 20. keep only one bin. For each number drawn, count in which bin it appears and cumulate this for all your drawings (1 000, 1 000 000)

```
Test20bins[ (int) negExp(10) ] ++;
```

With a Mean set to 10, you will produce many numbers between 0 and 1, a bit less between 1 and 2, etc. If you display an histogram, it can produce something that looks like figure 3 (with a different slope). You should also test that the observed mean is corresponding to the theoretical mean.

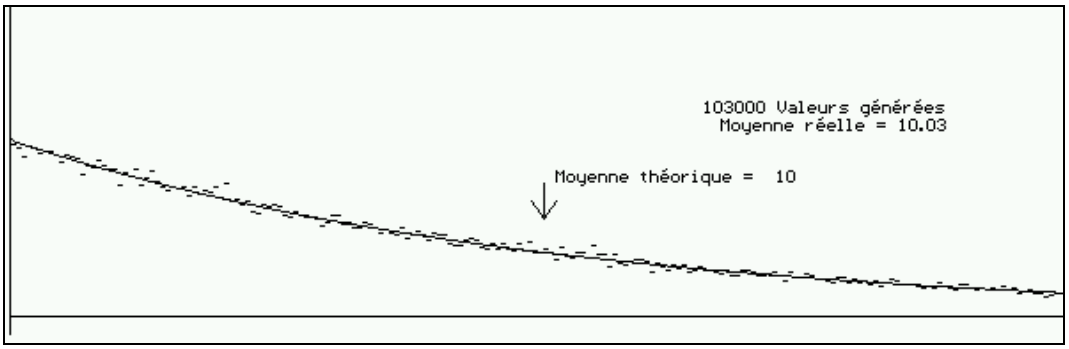

Figure 3. Simulation of the Inverse function of the uniform and negative exponential law.

5) Simulating non reversible distribution laws

In the case of non-reversible distribution laws, we can use the rejection technique, which is a Monte Carlo inspired technique. Below is a standard rejection algorithm for generating a number according to a probability distribution f(x) between 2 values MinX and MaxX (+ Min Y and MaxY which are the values providing a box around the probability distribution (density) function (PDF).

```
(1)  Generate 2 random numbers Na1 and Na2
(2)  Compute  X = MinX + Na1 * (MaxX - MinX)
(3)  Compute Y = MaxY * Na2
(4)  If Y is <= f(X)
     Then  X is considered as distributed according
           a law with f(x) as density function
     Else  reject X and goto (1) ie : draw again 2 pseudorandom numbers
           between 0 and 1, etc..
     EndIf
```

Figure 4. Generic rejection algorithm for any distributions

The Special case of the Gaussian distribution:

The density of a normal law (reduced and centred: average = 0, standard deviation = 1) is noted *N*(0,1) and given by equation (10) hereafter.

$$p(x) = \frac{1}{\sqrt{(2\pi)}} e^{-\frac{x^2}{2}} \qquad (10)$$

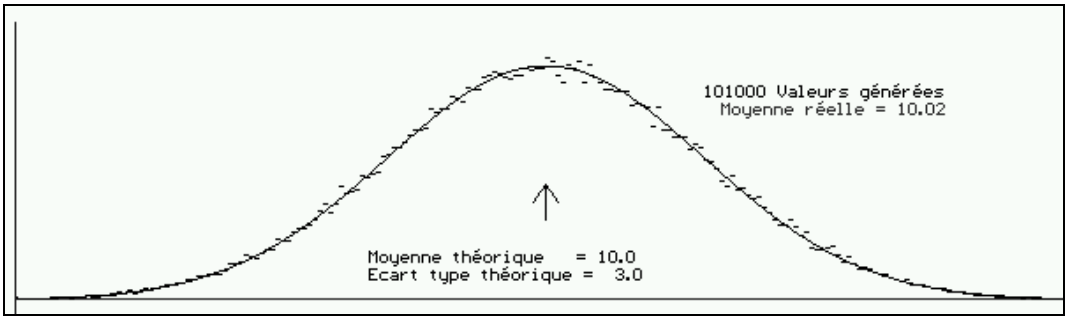

Figure 5. Generation of values following a Gaussian distribution (average = 10, std. Dev. = 3)

Consider an experiment drawing 20 times a common dice. Sum the obtained results. The expected result is between 20 (as a minimum : 20 x face and a potential maximum of 120 (20 x face 6) with a very low probability ($1 / 6^{20}$)

Implementation: Simulate this experiment 'many' times to obtain an approximation of the average (and of the standard deviation). You can then define statistical bins around the mean to see the (expected) bell curve.

In 1958, Box and Muller presented an exact method without using the Central Limit theorem and using two pseudo random numbers. Equation (14) uses two random numbers Rn1 and Rn2 and produces two numbers distributed on both sides of the centred and reduced Gaussian law - *N*(0,1). Many variants exist to approximate a Gaussian distribution, some are faster, some more precise…

$$x_1 = \cos(2\pi Rn_2)(-2\ln(Rn_1))^{\frac{1}{2}}$$
$$x_2 = \sin(2\pi Rn_2)(-2\ln(Rn_1))^{\frac{1}{2}} \quad (14)$$

Implementation: Test the Box and Muller functions to generate numbers around 0 following *N*(0,1). Two pseudorandom numbers give 2 numbers. Check for 1000 and 1000000 drawings how many numbers are distributed in 20 bins around -1 & 1 (between [-0.3..-0.2[, [-0.2.. -0.1[, [-0.1.. 0[, [0..0.1[, [0.1..0.2[, [0.2..0.3[. Print your result and see Does it fit with the known statistics for the Gaussian distribution? Think about a Gaussian distribution with an average of 10 and with a standard deviation at 3. Find on the internet how you could simulate this.

Testing the rejection method can be done as a bonus question.

6) Find libraries in C/C++ and Java that generate random variates like you did in the previous exercices.