Network Analysis and Simulation - AY 2020/2021
Homework 2

Amedeo Giuliani

April 13, 2021

Exercise 1

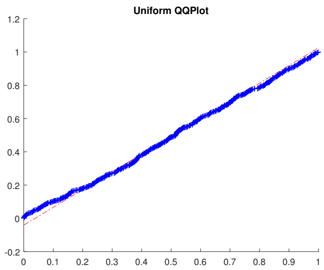

Figure 1: Figure 6.5a

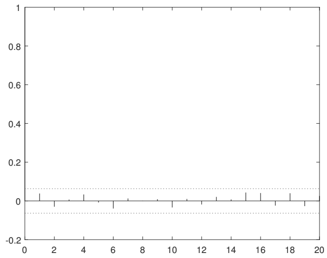

Figure 2: Figure 6.5b

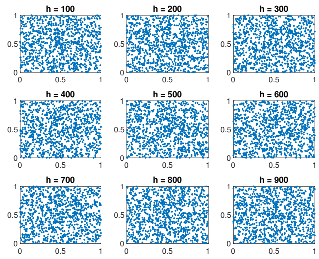

Figure 3: Figure 6.5c

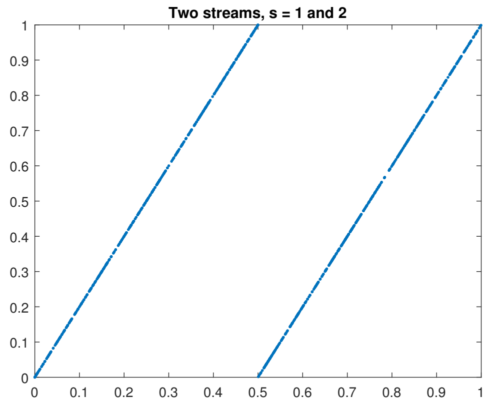

Figure 4: Figure 6.7a

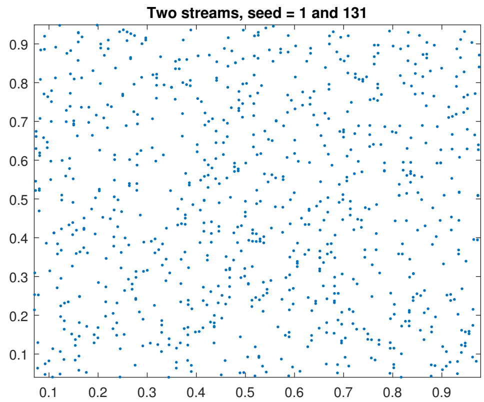

Figure 5: Figure 6.7b

Exercise 2

We want to generate a vector of $\boldsymbol{x} = (x_1, ..., x_k) \sim \mathcal{B}(n, p)$ using: (1) the CDF inversion method, (2) a sequence of n Bernoulli RVs and (3) geometric strings of zeros. For each of them we will count the time it takes to generate $\boldsymbol{x}$ with the MATLAB functions "*tic*" and "*toc*". The initial parameters of the Binomial vector are $n = 30,\ p = 0.2$.

CDF inversion

Here we have simply implemented the algorithm in [1, p.57] which took a time $t_1 = 0.013293$ s to complete.

Sequence of n Bernoulli RVs

For this method, we implemented the following algorithm, which took a time $t_2 = 0.020782$ s to complete.

Algorithm 1: sequence of n Bernoulli RVs

```
initialize empty array x;
k = 1000;
i = 1;
j = 1;
while j ≤ k do
    initialize empty array b;
    while i ≤ n do
        generate a uniform random number r;
        if r ≤ p then
            append to b the value 1;
        end
        else
            append to b the value 0;
        end
    end
    append to x the sum of the values in b;
end
```

Geometric string of zeros

For this last method, we implemented the following algorithm, which took a time $t_3 = 0.078115$ s to complete.

Algorithm 2: geometric string of zeros

```
initialize empty array x;
k = 1000;
j = 1;
while j ≤ k do
    initialize empty array seq;
    generate a geomtric random number g;
    while g+length(seq) ≤ n do
        append to seq a sequence of g zeros followed by a one;
        generate a geomtric random number g;
    end
    fill the remaining space with n-g-length(seq) zeros;
    append to x the sum of the values in seq;
end
```

Considerations

If we increase the parameter p we notice that: with the first method the elapsed time slightly increases; the second method is independent of p as we always generate n Bernoulli RVs and so on average the elapsed time remains the same; the third method is the most sensible to the change as for increasing p the probability of sampling a zero from the geometric distribution becomes larger and larger and so the inner loop in 2 is executed for a greater number of times.
If instead we increase the parameter n we observe that: with the first method the time elapsed is, on average, the same; in the second method we generate n Bernoulli RVs and of course the time increases with n; for the third method, also in this case the elapsed time varies the most among the three because the inner loop is executed for a greater number of times if we increase n.
In general, the third method takes always much more time than the other two because it has three inner loops: one is the while loop, then we have a for loop inside the while to fill the seq array with g zeros followed by a one and then we have another for to fill the remaining spaces in seq with zeros.

Exercise 3

We want to generate a vector of $\boldsymbol{x} = (x_1, ..., x_k) \sim \mathcal{P}(\lambda)$ using: (1) the CDF inversion method; (2) exponential until sum is less than 1; (3) uniform numbers until their product is greater than $e^{-\lambda}$. The initial value of the parameter of the Poisson distribution is $\lambda = 1$.

CDF inversion

For this method we have simply implemented the algorithm described in [1, p.56], which took a time $t_1 = 0.006587$ s to complete.

Exponential sum

Here we used the following algorithm, which took a time $t_2 = 0.022446$ s to complete.

Algorithm 3: exponential sum

```
k = 1000;
initialize empty array x;
for j≤k do
    n = 0;
    initialize empty array e;
    while true do
        generate a random number sampled from the exp distribution with μ = 1/λ;
        append this number to e;
        s = sum(e);
        if s > 1 then
            break;
        end
        n = length(e);
    end
    append n to x;
end
```

Product of uniform random numbers

For this method, the following algorithm was used, which took a time $t_3 = 0.010542$ s.

Algorithm 4: product of uniform random numbers

```
k = 1000;
initialize empty array x;
el = exp(-λ);
for j≤k do
    n = 0;
    initialize empty array u;
    while true do
        generate a uniform random number between 0 and 1;
        append this number to u;
        p = prod(u);
        if p < el then
            break;
        end
        n = length(u);
    end
    append n to x;
end
```

Considerations

If we let λ increase we will see that the second method is the most susceptible to the changes, due to the exit condition on the inner loop in 3: the larger λ the smaller is μ and therefore the smaller will be the exponential random number. Thus, it takes more time to met the condition $s > 1$ and the inner loop is executed more times. With the third method we experience only a slight increase in the elapsed time. The same reasoning conversely applies when we decrease lambda. Instead, for the first method, the elapsed time remains more or less constant.

Exercise 4

We have a *Linear-Congruential Generator* (LCG) with $a = 18, m = 101$. An LCG is said to be "full period" if the generated sequence repeats itself after m numbers, i.e., its period is m. In this case, upon the generation of a sequence long five times m, we indeed have that x_0, the seed, appears in position 1, 101, 201, 301, 401 and 501. From these points onwards, the sequence starts all over again, and they are spaced precisely by m position. Hence, this LCG is "full period".
In Fig. 6 we plotted in a unit square the pairs (Ui,Ui+1) where Ui is the i-th number and Ui+1 is the (i+1)-th number of the generated sequence, respectively. We can see that, even though the points fill up the square, they follow a kind of stripes pattern, which is a flag of some (positive) correlation between the sequence and its shifted version.
Now we take a different LCG with $a = 2$ and same m. Also this one is "full period" for the reason above, but it is worse than the first because now in Fig. 7 the pairs (Ui,Ui+1) appear along two straight lines, which means that the sequences x_n and x_{n+1} are heavily (and positively) correlated. In fact, in Fig. 8, the autocorrelation function for a lag $h = 1$ shows a value of ~ 0.5 that just cannot be negligible.

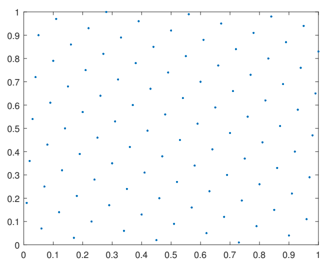

Figure 6: Scatter plot for LCG1.

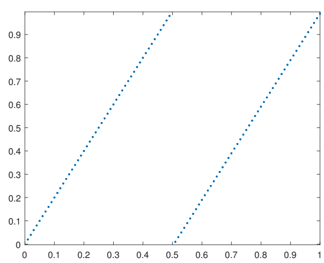

Figure 7: Scatter plot for LCG2.

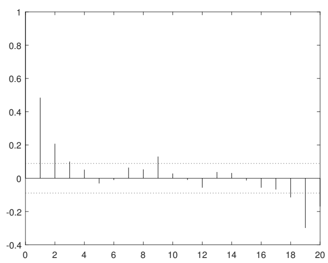

Figure 8: ACF of x_n for LCG2.

Exercise 5

Like in the previous exercise, we analyse an LCG with parameters $a = 65539, m = 2^{31}$. This time, if we plot the pairs (Ui,Ui+1) as in Fig. 9, there is no preferential direction in which the dots align. However, if we consider the triple (Ui,Ui+1,Ui+2) and we perform a 3D plot, as in Fig. 10, it seems that the points are lying on multiple planes, even though from Fig. 11 we can tell that there is almost no correlation between subsequent numbers in the generated sequence x_n.

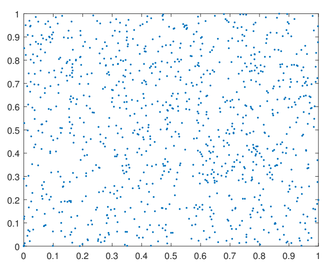

Figure 9: Scatter plot for LCG3.

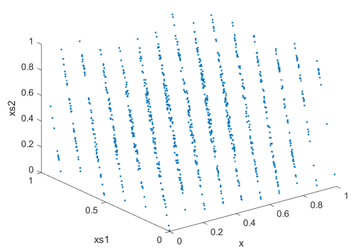

Figure 10: 3D scatter plot for LCG3.

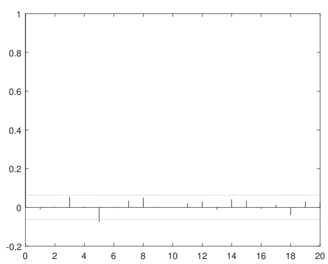

Figure 11: ACF of x_n for LCG3.

References

[1] Sheldon M. Ross, *Simulation, fourth edition*, Academic Press, Inc., USA, 2006.