Network analysis and simulation
Homework 1

Davide Peron

Exercise 1

In Figure 1, Figure 2, Figure 3, Figure 4 and Figure 5, the behavior of a Linear Congruence Generator (LCG) has been studied. In the last two figures, the two *non-standard* distributions have been taken in account. In Figure 6 the distribution is

$$f_Y(y) = K\frac{sin^2(y)}{y^2}\mathbf{1}_{\{-a\leq y\leq a\}}$$

with $a = 10$, while in Figure 7 there are a set of sample of the random vector (X_1, X_2) with a density proportional to $|X_1 - X_2|$.

In the follow the said figures are reported.

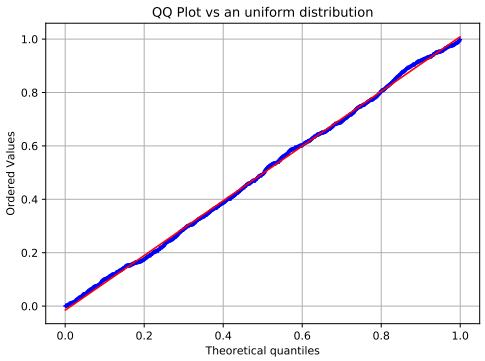

Figure 1: QQPlot of the LCG versus a uniform distribution (Figure 6.5a)

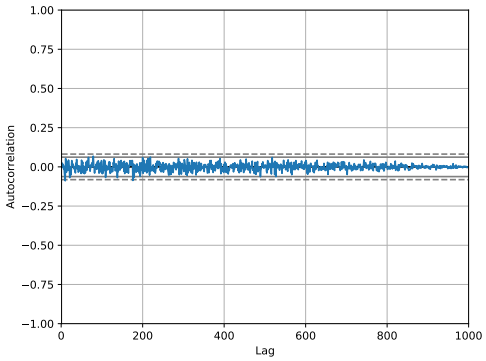

Figure 2: Autocorrelation of the LCG (Figure 6.5b)

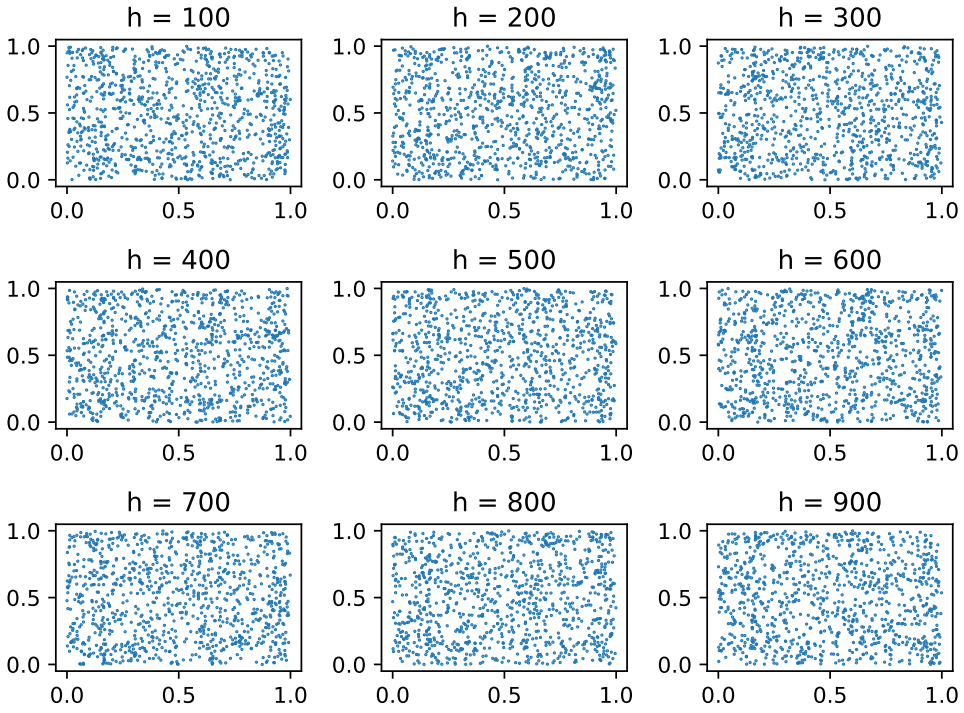

Figure 3: Lag plots (Figure 6.5c)

Exercise 2

Trying to generate a set Binomial random variables $B(n, p)$ with $n = 50$ and $p = 0.5$, we can see that the fastest approach is the CDF inversion technique, although extracting n Bernoulli rvs is much faster than the Geometric method. In the follow, the three different implementations are presented.

The first method inplemented is the one of the CDF inversion.

```
# With CDF inversion

X_CDF = []
F_CDF = []
start = time.time()
c = theta/(1-theta)
for _ in range(0,n_rvs):
    pr = (1-theta)**n
    u = np.random.rand()
    i = 0
    F = pr
    while u >= F:
        pr *= c*(n-i)/(i+1)
        F += pr
        i += 1
    X_CDF.append(i)
    F_CDF.append(F)
    mean_CDF += i
end = time.time()
time_CDF = end - start
```

A Binomial Random Variable can also be created drawing n Bernoulli rvs and counting the successes.

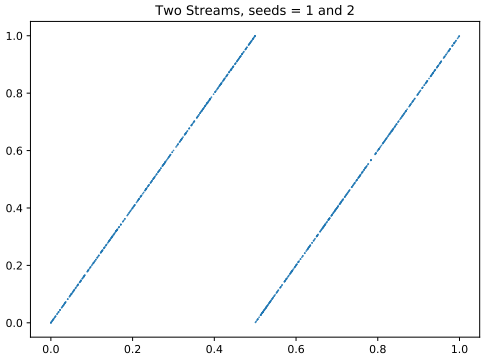

Figure 4: Behavior of two parallel stream using seed $s = 1$ and $s = 2$ (Figure 6.7a)

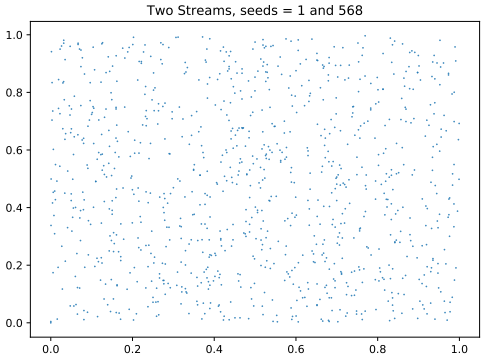

Figure 5: Behavior of two parallel stream using seed $s = 1$ and $s = 568$ (Figure 6.7b)

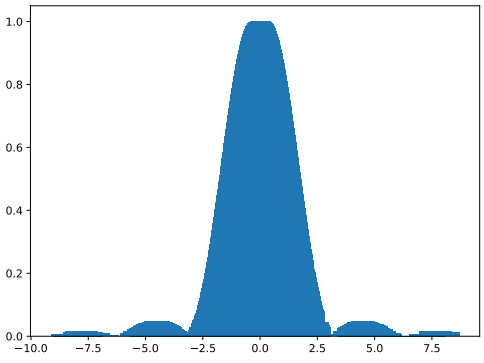

Figure 6: Histograms of a set of sample from the first *weird* distribution (Figure 6.10a)

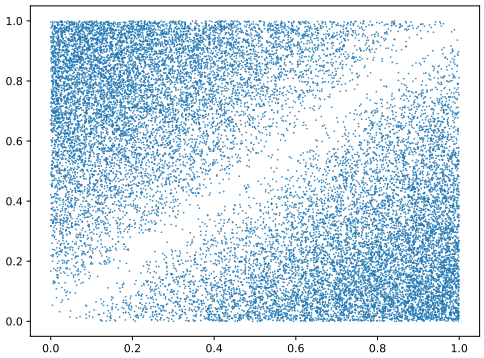

Figure 7: Set of sample from the second *weird* distribution (Figure 6.10b)

```
  # Drawing n Bernoulli rvs
start = time.time()

for _ in range(0,n_rvs):
    X_bernoulli = 0
    for i in range(0,50):
        u = np.random.rand()
        if u > 0.5: #Success
            X_bernoulli += 1
    mean_bernoulli += X_bernoulli
end = time.time()
time_bernoulli = end - start
```

A method to avoid to draw n Bernoulli rvs is to count how much geometric fits into n trials **(DP says: Qui è da rivedere)**, although this approach is the one that requires less operations, is the slower one, probably due to the computational complexity of the *log* function.

```
  # Using Geometric Distribution

start = time.time()
for _ in range(0,n_rvs):
    isFinished = False
```

```
    X_geometric = 0
    count = 0
    while not isFinished:
        u = np.random.rand()
        X_geometric += np.floor(np.log(u)/np.log(1-theta)) + 1
        if X_geometric >= n :
            isFinished = True
        if X_geometric <= n:
            count += 1
    mean_geometric += count
end = time.time()
time_geometric = end - start
```

Exercise 3

Trying to generate a set Poisson random variables $P(\lambda)$ with $\lambda = 5$, we can see that, also in this case, the fastest method is the CDF inversion technique, immediately followed by the Uniform rvs method and the Exponential one. In the follow, the three different implementations are presented.

The first method inplemented is the one of the CDF inversion.

```
#CDF Inversion
mean_CDF = 0
start = time.time()
for k in range(0,n_vars):
    u = np.random.rand()
    i = 0
    p = math.exp(-lambda_par)
    F = p
    while u >= F:
        p *= lambda_par/(i+1)
        F += p
        i += 1
    mean_CDF += i

mean_CDF /= n_vars
end = time.time()
time_CDF = end - start
```

A Poisson Random Variable can also be created counting the number of exponential rvs that you can draw until their sum is less than 1.

```
#Sum of exponentials
mean_exp = 0
start = time.time()
for i in range(0,n_vars):
    n_exp = 0
    sum = 0
    while sum <= 1:
        sum += -math.log(np.random.rand())/lambda_par
        n_exp += 1
    mean_exp += n_exp - 1

mean_exp /= n_vars
```

```
end = time.time()
time_exp = end - start
```

The last method is using uniforms as long as their product is greater than $e^{-\lambda}$.

```
  #Product of uniforms
mean_unif = 0
start = time.time()
for i in range(0,n_vars):
    rv = 1
    count = 0
    while rv > math.exp(-lambda_par):
        rv *= np.random.rand()
        count += 1
    mean_unif += count - 1

mean_unif /= n_vars
end = time.time()
time_unif = end - start
```