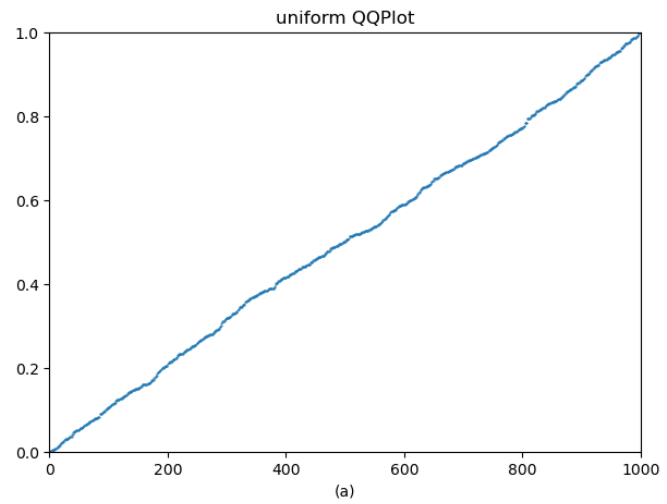
Network Analysis and Simulation - Homework 2

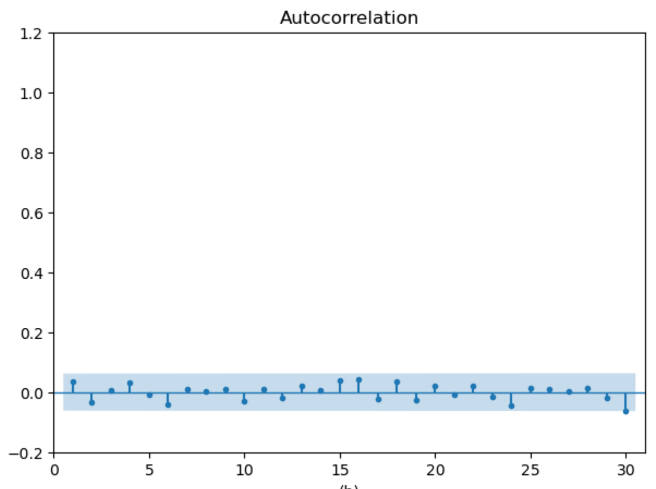
Michele Scapinello

1 Exercise 1

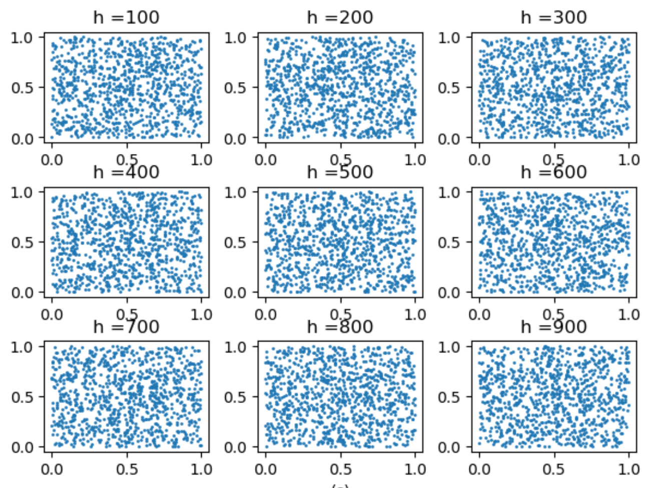
Produce results as in Figs. 6.5, 6.7 and 6.10.



(a)

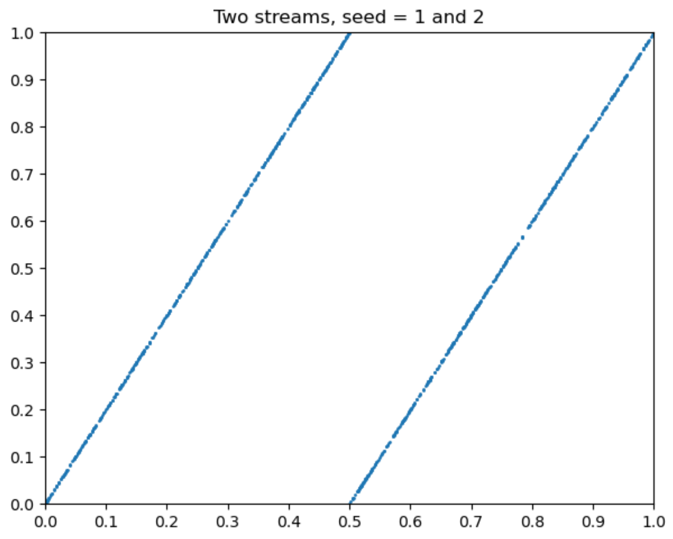


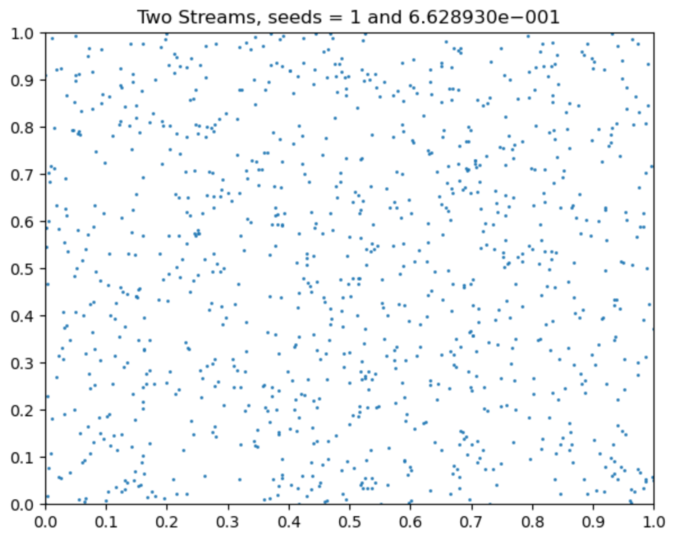
(b)



(c)

Reproduction of Figure 6.5: 1000 successive numbers for the generator in Example 6.8. (a) QQplot against the uniform distribution in (0, 1), showing a perfect match. (b) autocorrelation function, showing no significant correlation at any lag (c) lag plots at various lags, showing independence.





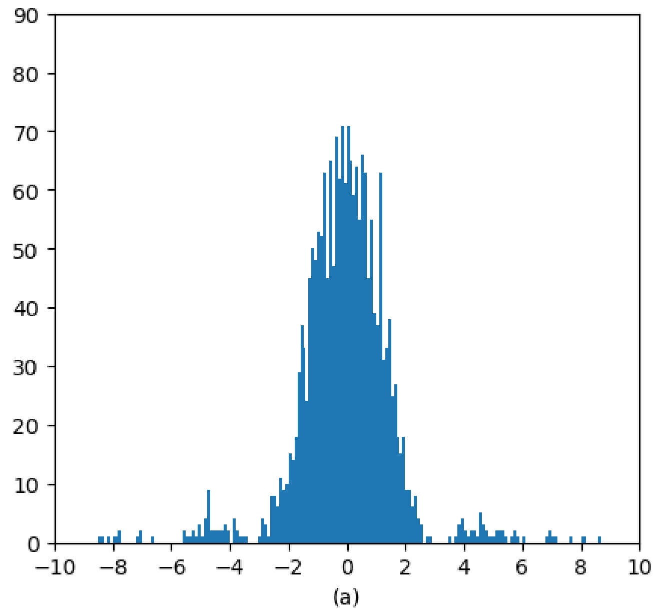
Reproduction of Figure 6.7: x_n versus x'_n for two streams generated with the linear congruential in Example 6.8. (a) seed values are 1 and 2 (b) seed values are (1, last value of first stream).

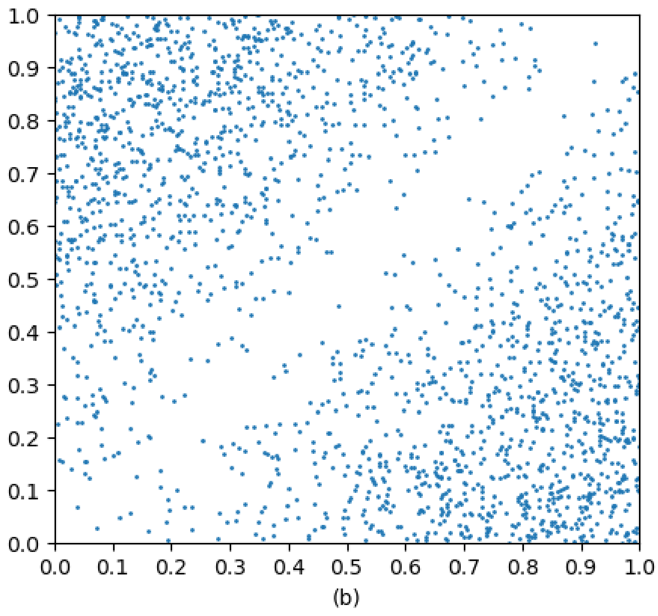
For the reproduction of Figure 6.10 shown below the rejection sampling method has been implemented. More specifically to generate samples for plot (a) the rejection sampling algorithm performs the following steps using a=10 and M=1:

1. Draw $U \sim (-a, a)$ and $X \sim (0, M)$;
2. If $U \leq f_X^n(x)$ return X else go to 1.

To generate samples for plot (b) the rejection sampling algorithm performs the following steps:

1. Draw X_1, X_2 and $U \sim (0, 1)$;
2. If $U \leq |X_1 - X_2|$ return (X_1, X_2) else go to 1.





Reproduction of Figure 6.10: (a) Empirical histogram (bin size = 10) of 2000 samples of the distribution with density $f_X(x)$ proportional to $\frac{sin^2(x)}{x^2} 1_{\{-a \leq y \leq a\}}$ with a=10. (b) 2000 independent samples of the distribution on the rectangle with density $f_{X_1,X_2}(x_1, x_2)$ proportional to $|x_1 - x_2|$.

2 Exercise 2

Implement the generation of a Binomial rv in the three ways we saw in class:

a. CDF inversion;

b. Using a sequence of n Bernoulli variables;

c. Using geometric strings of zeros.

Compare the time it takes to produce a large number of iid variables for different values of n and p.

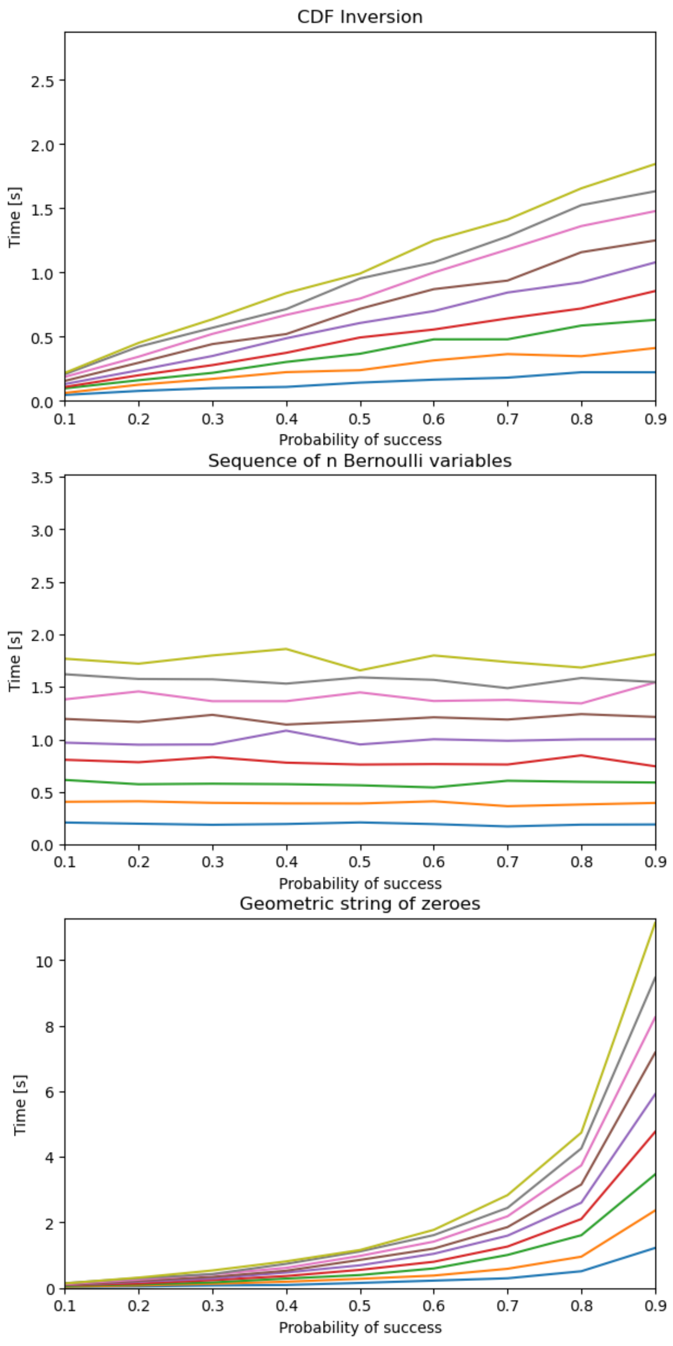


Figure 2.1: Time taken to each generation method to generate 10000 samples with an increasing value of n. Specifically the values of n are contained in the range [10, 90] with an interval between values of 10

Using geometric strings method is the slowest of those applied, and certainly the way the algorithm was implemented affected the performances. As shown in Figure 2.1 time taken to generate 10000 samples grows exponentially as the values of p increases and also depends on the value of n. To generate a geometric random variable the following formula is used:

$$X \equiv \left\lfloor \frac{ln(U)}{ln(1-p)} \right\rfloor \tag{1}$$

which outputs smaller values of X as p grows. The implementation of the algorithm is the following:

```
def binomial_from_geometric(n, p):
    count = 0
    sum = 0
    while sum < n:
        x = generate_geometric(p)
        sum += x
        count += 1
    return count
```

where the method `generate_geometric(p)` uses (1) to generate a geometric random variable of parameter `p`. The CDF inversion method instead has a generation time that grows linearly, as the value of p and n increases. For smaller values of p the CDF inversion method is faster than the Bernoulli sequence method however, with larger values of p it becomes slower, most likely because method Bernouillu sequence has a time complexity that depends only on the value of n. In fact, Figure 2.1 shows how the time taken to generate 10000 samples is constant for each value of p and obviously grows with the value of n. The algorithms for CDF inversion and Bernoulli sequence are the following:

```
def binomial_from_bernoulli(n, p):
    x = 0
    for _ in range(n):
        x += bernoulli(p)
    return x
```

where the method `bernoulli(p)` generates a Bernoulli variable with parameter `p`.

```
def binomial_from_cdf_inversion(n, p):
    u = random()
    c = p / (1 - p)
    i = 0
    f = pr = pow((1 - p), n)
    while True:
        if u < f:
            return i
        pr = ((c*(n-i))/(i+1))*pr
        f += pr
        i += 1
```

3 Exercise 3

Implement the generation of a Poisson rv in the three ways we saw in class

a. CDF inversion;

b. Using exponential until sum > 1;

c. Using uniforms as long as product is $> e^{-\lambda}$.

Compare the time it takes to produce a large number of iid variables for different values of lambda.

As shown in Figure 3.1 method uniform product method is the slowest among those applied. However for small values of lambda all the algorithms are fast but as the value increase the uniform product method becomes the slower. One of the reasons may be the fact that the uniform product method requires to generate a large number of uniform variables. Moreover, as the value of lambda increases the probability that the product of the uniform variables will be greater than $e^{-\lambda}$ decreases, thus requiring more uniform variables to generate a Poisson rv. The algorithm used to generate a Poisson rv using the product of uniform rv is the following:

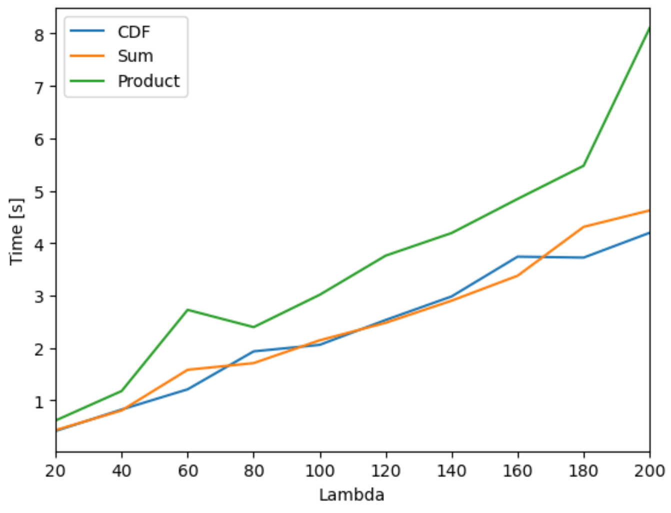


Figure 3.1: Time taken to each RNG method to generate 10000 samples with an increasing value of λ

```
def poisson_from_product(lam):
    i = 0
    prod = 1
    while prod > np.exp(-lam):
        prod *= np.random.uniform(0, 1)
        i += 1
    return i - 1
```

CDF inversion is faster than uniform product probably because it only needs to generate one uniform variable for each Poisson rv and compute some mathematical formulas. The exponential sum method instead, requires to generate a varying number of exponential variables that most likely turns out to be a relatively small number since the execution time of this algorithm align those of the CDF inversion. The implementation of the CDF inversion method and the exponential sum method is shown below.

```
def poisson_from_sum(lam):
    i = 0
    s = 0
    while s <= 1:
        s += np.random.exponential(1/lam)
        i += 1
    return i - 1


def poisson_from_cdf(lam):
    u = random()
    i = 0
    f = p = np.exp(-lam)
    while u >= f:
        p = (lam * p) / (i + 1)
        f += p
        i += 1
    return i
```

4 Exercise 4

Consider the two LCGs: LCG1: a = 18, m = 101; LCG2: a = 2, m = 101

a. Check whether they are full period;

b. Plot all pairs (U_i, U_{i+1}) in a unit square and comment the results.

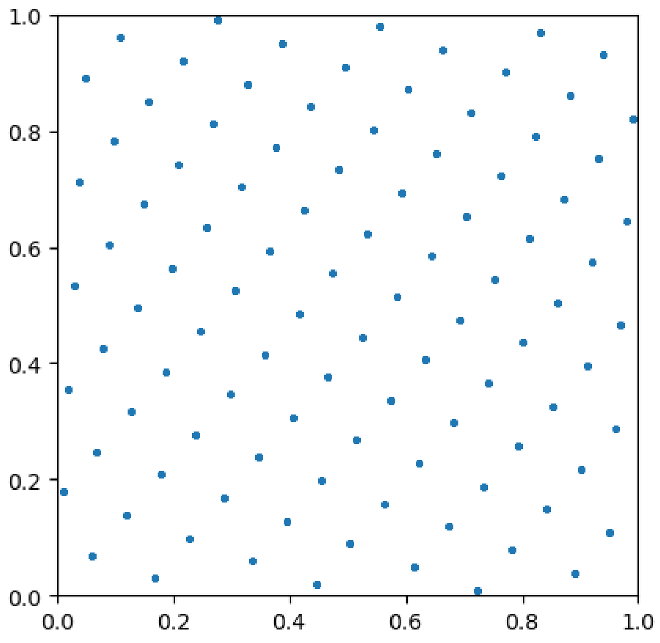


Figure 4.1: (U_i, U_{i+1}) plot for LCG1

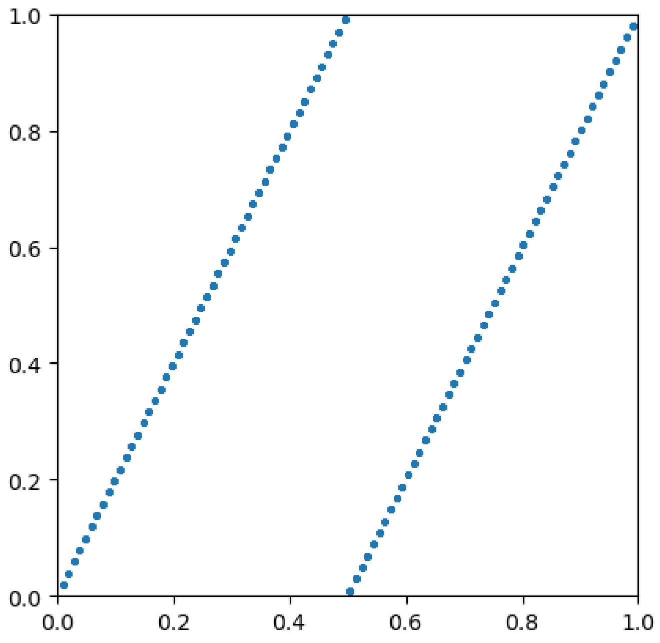


Figure 4.2: (U_i, U_{i+1}) plot for LCG2

To verify that the LCGs are full period, it is possible to run a simple algorithm that, starting from the first generated element, searches within the set of generated elements for a match and thus a value equal to that of the first element. At the end of the search, a counter indicating the period of the LCG will be printed. For both LCGs it turns out that the period has a value equal to $m - 1$, that is 100, so both generators will start repeating themselves after generating 100 numbers. Considering that the value 0 can be omitted from the numbers generated, it is possible to assume that the two generators are full period because they generate all the values in the range $[1, 101)$.

The plot of all the pairs (U_i, U_{i+1}) gives us information about the randomness of the generators. By comparing Figure 4.1 and Figure 4.2 it is possible to notice how the LCG1 is more random than LCG2. The points plotted in a unit square for LCG1 (Figure 4.1) appear to be more uniformly distributed over the area. In any case, there are diagonal patterns that occur regularly, suggesting that although the distribution is apparently random, the data generated are correlated with each other and therefore not as random as expected. For LCG2, on the other hand, the graph shows how the numbers generated are strongly correlated with each other, given the presence of two main diagonals on which all the points lie. This allows us to state that LCG1 is more random than LCG2 but in any case neither generator could be reliable in any application scenarios given the correlation between the numbers generated.

5 Exercise 5

Consider the LCG with a = 65539, m = 2^{31};

a. Plot all pairs (U_i, U_{i+1}) in a unit square and comment the results;

b. Plot all triples (U_i, U_{i+1}, U_{i+2}) in a unit cube and comment the results.

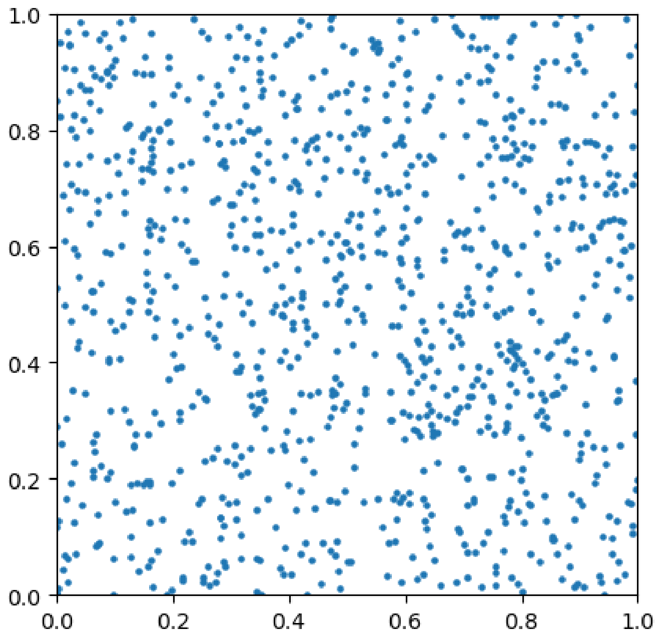


Figure 5.1: (U_i, U_{i+1}) plot for the LCG

The plot of all the pairs (U_i, U_{i+1}) shown in Figure 5.1 suggests that the random numbers generated using the LCG with the given parameters might be truly random because of the uniform distribution of the point across the unit square. By looking at Figure 5.1 it might be possible however to notice some small clusters, suggesting some correlation between the points. In any case the points in the unit square are quite randomly distributed and by looking at this plot it could be possible to say that the generator might be a good LCG, and thus usable in real world problems. However, by plotting in a unit cube all the triplets formed by (U_i, U_{i+1}, U_{i+2}), the points appear to fall on 15 different hyper planes, as shown in figure 5.2, suggesting that the generator is not as random as shown in Figure 5.1. This demonstrates that the numbers generated using $a = 65539$ and $m = 2^{31}$ are correlated between each other and then it is possible to conclude that the LCG with parameters $a = 65539$ and $m = 2^{31}$ is a bad random number generator and so it should be better to avoid using it in real world problems.

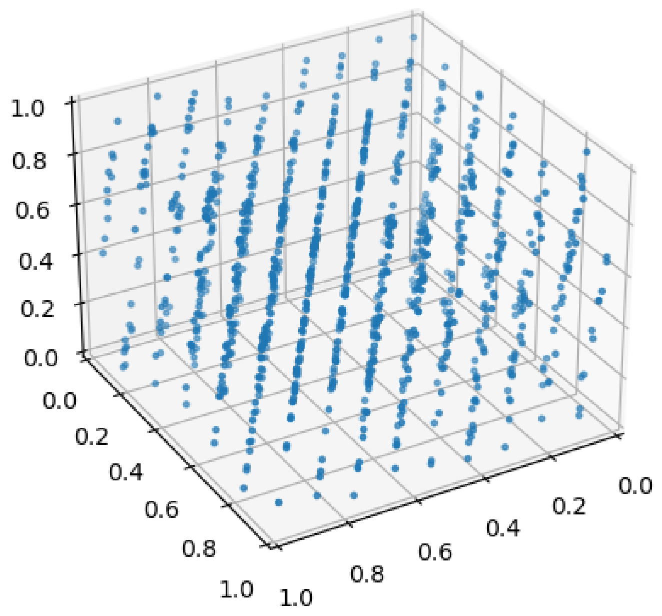


Figure 5.2: (U_i, U_{i+1}, U_{i+2}) plot for the LCG