02443 - Handin of Exercises

s183529 - Jonas Søeborg Nielsen

s194323 - Aleksander Svendstorp

Computer Exercise 1: Generation and Testing of Random Numbers

In [ ]:
```python
# Plotting
import plotly.graph_objects as go
import plotly.express as px
import plotly.subplots as sp
import plotly.io as pio
pio.renderers.default = "notebook+pdf"
pio.templates.default = "plotly_dark"

# Utilities
import numpy as np
import pandas as pd
from scipy.stats import chi2, norm
from scipy.special import kolmogorov
```

Part 1 - Linear Congruential Generator (LCG)

First we implement a Linear Congruential Generator (LCG) to generate random numbers. The implementation can be seen in the python code below.

In [ ]:
```python
def LCG(a:int, c:int, M:int, x0:int, n:int, as_int=False):
    """
    LCG generates a list of random numbers using the Linear Congruent Generation method.
    """
    # Preallocate and initialize array for pseudorandom numbers
    X = np.zeros(n+1, dtype=int)
    X[0] = x0

    # Generate pseudorandom numbers
    for i in range(1, n+1):
        X[i] = (a*X[i-1] + c) % M

    if as_int:
        return X[1:]
    else:
        return X[1:]/M
```

To make sure our implementation works as intended, we compare with an example from the slides and check if the same stream of random numbers is generated.

In [ ]:
```python
M = 16
a = 5
c = 1
x0 = 3

U_true = np.array([0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3])
U_LCG = LCG(a, c, M, x0, 16, as_int=True)

print(f"True sequence:\t{U_true}")
print(f"LCG sequence:\t{U_LCG}")
```

```
True sequence:  [ 0  1  6 15 12 13  2 11  8  9 14  7  4  5 10  3]
LCG sequence:   [ 0  1  6 15 12 13  2 11  8  9 14  7  4  5 10  3]
```

Below we prepare some functions for plotting histograms and scatter plots of the generated random numbers.

In [ ]:
```python
# Histogram
def plot_histogram(U, title="", n_bins=20):
    fig = go.Figure(go.Histogram(x=U, xbins=dict(start=0, end=1, size=1/n_bins), histnorm='probability density', marker=dict(color=
    fig.add_trace(go.Scatter(x=[0, 1], y=[1, 1], mode='lines', line=dict(color='Red', width=2)))
    fig.update_layout(title=title, xaxis_title="Value", yaxis_title="Density", width=600, height=400, bargap=0.1, showlegend=False)
    fig.show()

# Correlation plot
def plot_correlation(U, title="Correlation plot of consequtive numbers."):
    fig = go.Figure(go.Scatter(x=U[:-1], y=U[1:], mode='markers', marker=dict(size=2, color='Blue')))
```

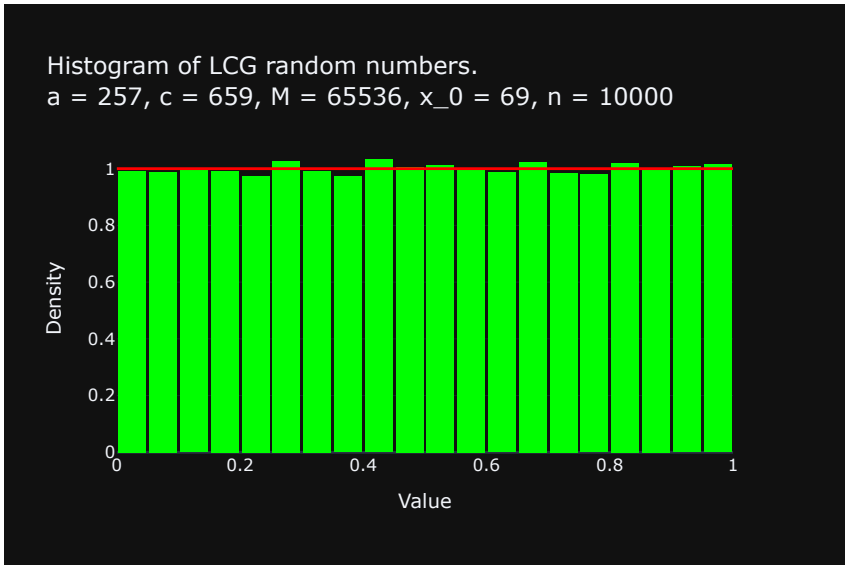
```
    fig.update_layout(title=title, xaxis_title=r"$U_{i-1}$", yaxis_title=r"$U_{i}$", width=600, height=600)
    fig.show()
```

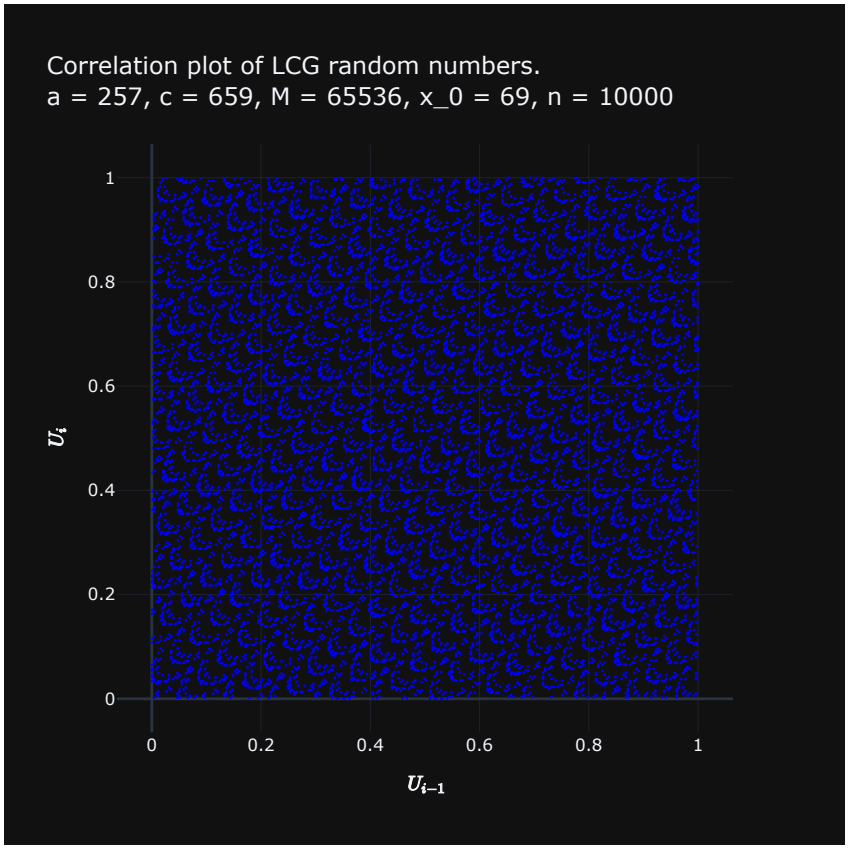
(a) Generating 10000 pseudorandom numbers

Using our implementation of the LCG algorithm, we generate 10000 pseudo random numbers. To ensure full cycle length, the parameters of the RNG is chosen according to the criteria in slide 14 of lecture 2.

We plot the histogram of the numbers and a scatter plot of the numbers in pairs of two.

In [ ]:
```
a = 257     # Prime
c = 659     # Prime
M = 65536   # 2^16
x0 = 69     # 3*23
n = 10000   # Number of samples

# Generate pseudorandom numbers
U = LCG(a, c, M, x0, n)

# Plot histogram
plot_histogram(U, title="Histogram of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0, n))

# Plot correlation
plot_correlation(U, title="Correlation plot of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0,
```





Considering the histogram, the numbers appear to be fairly uniformly distributed. However, the scatter plot clearly shows that there some underlying structure of these pseudo random numbers.

(b) RNG evaluation

Now we want to run different tests to evaluate the quality of the RNG. Below we implement the tests before running and comparing them.

Functions for testing distribution of random numbers

In [ ]:
```python
def chi_sq_test(U, n_classes=10):

    # Compute expected number of observations in each class
    n_expected = len(U) / n_classes

    # Count number of observations in each class
    n_obs, _ = np.histogram(U, bins=n_classes)

    # Compute test statistic
    T_obs = np.abs(np.sum((n_obs - n_expected)**2 / n_expected))

    # Compute p-value
    df = n_classes-1 # when number of estimated parameters is m=1
    p = 1 - chi2.cdf(T_obs, df)

    return T_obs, p
```

In [ ]:
```python
def kolmogorov_smirnov_test(U):

    # Get number of observations
    n = len(U)

    # Setup expected values of F
    F_exp = np.linspace(0, 1, n+1)[1:]

    # Compute test statistic
    Dn = max(abs(F_exp-np.sort(U)))

    # Compute p-value
    p = kolmogorov(Dn)

    return Dn, p
```

Runtests for testing independence of random numbers

In [ ]:
```
def above_below_runtest1(U):

    median = np.median(U)
    n1 = np.sum(U < median)
    n2 = np.sum(median < U)

    # Compute total number of observed runs
    temp = U > median
    T_obs = sum(temp[1:] ^ temp[:-1])

    # Compute p-value
    mean = 2*n1*n2/(n1 + n2) + 1
    log_expr = np.log(2) + np.log(n1) + np.log(n2) + np.log(2*n1*n2 - n1 - n2) - 2*np.log(n1 + n2) - np.log(n1 + n2 - 1)
    var = np.exp(log_expr)
    Z_obs = (T_obs - mean) / np.sqrt(var)
    p = 2 * (1 - norm.cdf(np.abs(Z_obs)))

    return T_obs, p
```

In [ ]:
```
def up_down_runtest2(U):

    n = len(U)

    # Get indeces where runs change (Append -1 and n-1 at ends to handle first and last run)
    idx = np.concatenate(([-1], np.where(U[1:]-U[:-1] < 0)[0], [len(U)-1]))

    # Compute run lengths and count them (clamp to 6)
    run_lengths = np.clip(idx[1:] - idx[:-1], 1, 6)
    R = np.array([np.count_nonzero(run_lengths == i) for i in range(1, 7)])

    # Compute test statistic
    A = np.array([
        [4529.4, 9044.9, 13568, 18091, 22615, 27892],
        [9044.9, 18097, 27139, 36187, 45234, 55789],
        [13568, 27139, 40721, 54281, 67852, 83685],
        [18091, 36187, 54281, 72414, 90470, 111580],
        [22615, 45234, 67852, 90470, 113262, 139476],
        [27892, 55789, 83685, 111580, 139476, 172860]
    ])
    B = np.array([1/6, 5/24, 11/120, 19/720, 29/5040, 1/840])
    Z_obs = (1/(n - 6)) * (R - n*B).T @ A @ (R - n*B)

    # Compute p-value
    p = 1 - chi2.cdf(Z_obs, 6)

    return Z_obs, p
```

In [ ]:
```
def up_and_down_runtest3(U):

    n = len(U)

    # Find runs (Append 0 at ends to handle first and last run)
    seq = np.concatenate(([0], np.sign(U[1:] - U[:-1]), [0]))

    # Get indeces where runs change
    idx = np.flatnonzero(seq[:-1] != seq[1:])

    # Compute run lengths
    run_lengths = idx[1:] - idx[:-1]
    X_obs = len(run_lengths)

    # Compute test statistic
    Z_obs = (X_obs - (2*n-1)/3) / np.sqrt((16*n - 29) / 90)

    # Compute p-value
    p = 2*(1 - norm.cdf(np.abs(Z_obs)))

    return Z_obs, p
```

In [ ]:
```
def corr_coef(U, h=2):
    n = len(U)
    ch = np.sum(U[:n-h]*U[h:])/(n-h)
    Z = (ch - 0.25)/(7/(144*n))
    p = 2*(1 - norm.cdf(np.abs(ch)))
    return ch, p
```

In [ ]:
```
def test_random_numbers(U):
```

```
    tests = [chi_sq_test, kolmogorov_smirnov_test, above_below_runtest1, up_down_runtest2, up_and_down_runtest3, corr_coef]
    table = np.array([test(U) for test in tests])

    df = pd.DataFrame(np.round(table, 2),
                      index=["Chi squared", "Kol-Smi", "Above/Below (I)", "Up/Down (II)", "Up and Down (III)", "Correlation"],
                      columns=["Test statistic", "p-value"]
    )
    print(df)
```

In [ ]:
```
test_random_numbers(U)
```

```
                  Test statistic  p-value
Chi squared                 1.38     1.00
Kol-Smi                     0.00     1.00
Above/Below (I)          4974.00     0.59
Up/Down (II)              368.35     0.00
Up and Down (III)          18.52     0.00
Correlation                 0.25     0.80
```

In the table above we have plotted the test statistics and the corresponding p-values for each of the tests. We see that the chi squared test and the Kolmogorov-Smirnov test have p-values above 0.05, which means we cannot reject the null hypothesis. For these two tests, the null hypothesis is that the random numbers are uniformly distributed. The next four tests are to test if the numbers we have generated are independent. We get mixed results from these tests. Two of the tests have p value above 0.05 meaning we cannot reject the null hypothesis, while the other two have p value below 0.05 meaning we can reject the null hypothesis. However by looking at the scatter plot of our generated points from part (a) we can see that the points are not independent. This is because the points are not scattered randomly, but rather in a pattern.

So in conclusion, the random numbers we have generated are not independent, but they are uniformly distributed. This is most likely since our choice of parameters can be further tuned to get better results.

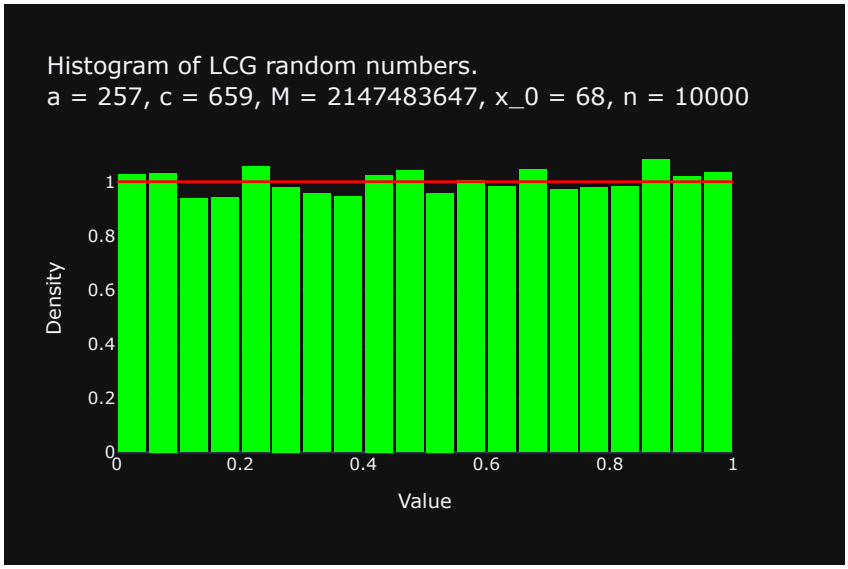
(c) Experimenting with the RNG

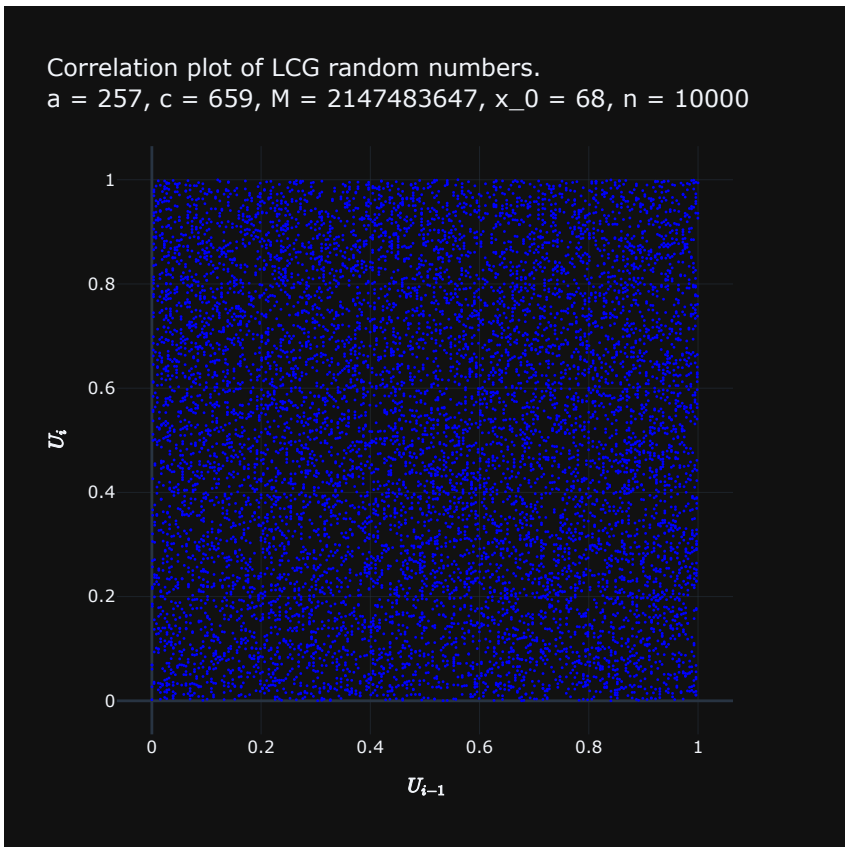
It is clear that our initial choice of parameters for the RNG are not optimal. While the histogram looks fairly uniform, the scatterplot clearly shows a repeating pattern in these number. Additionally, the RNG fail runtests I and II. We suspect that this is due to M being small compared to the number of generated numbers.

To rectify this and hopefully generate better pseudo random numbers, we will increase M to $2^{31} - 1$, a large Mersenne prime.

In [ ]:
```
a = 257     # Prime
c = 659     # Prime
M = 2**31-1 # Large Mersenne Prime
x0 = 69-1   # 3*23 -1
n = 10000   # Number of samples

# Generate pseudorandom numbers
U = LCG(a, c, M, x0, n)

# Plot histogram
plot_histogram(U, title="Histogram of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0, n))

# Plot correlation
plot_correlation(U, title="Correlation plot of LCG random numbers.<br>a = {}, c = {}, M = {}, x_0 = {}, n = {}".format(a, c, M, x0,

test_random_numbers(U)
```

```
C:\Users\jonas\AppData\Local\Temp\ipykernel_7040\2080446664.py:11: RuntimeWarning:

overflow encountered in scalar multiply
```





	Test statistic	p-value
Chi squared	11.23	0.26
Kol-Smi	0.01	1.00
Above/Below (I)	5090.00	0.08
Up/Down (II)	8.94	0.18
Up and Down (III)	0.28	0.78
Correlation	0.25	0.80

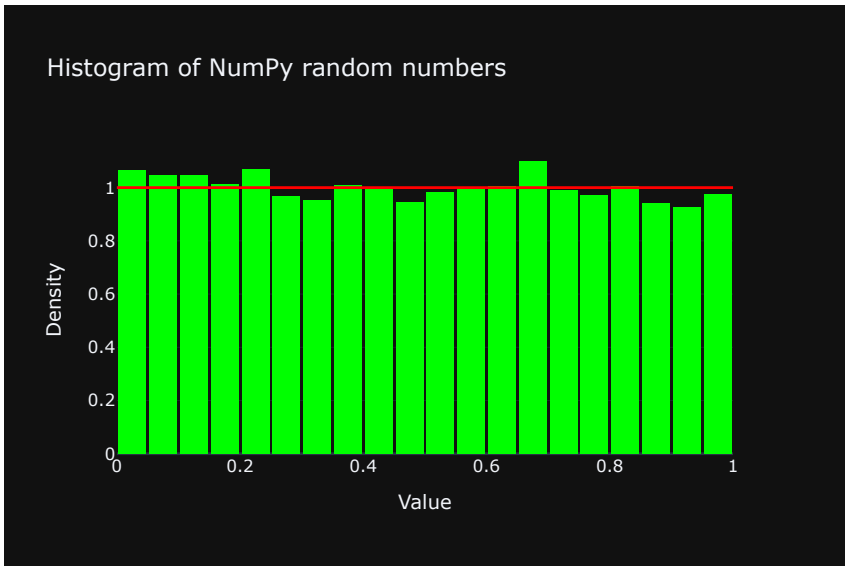
These new pseudorandom appears fairly uniformly distributed in the histogram, and the scatterplot shows no clear pattern within the numbers. All of the tests yield high p-values, thus the hypothesis of these numbers being independent and uniformly distributed cannot be rejected.

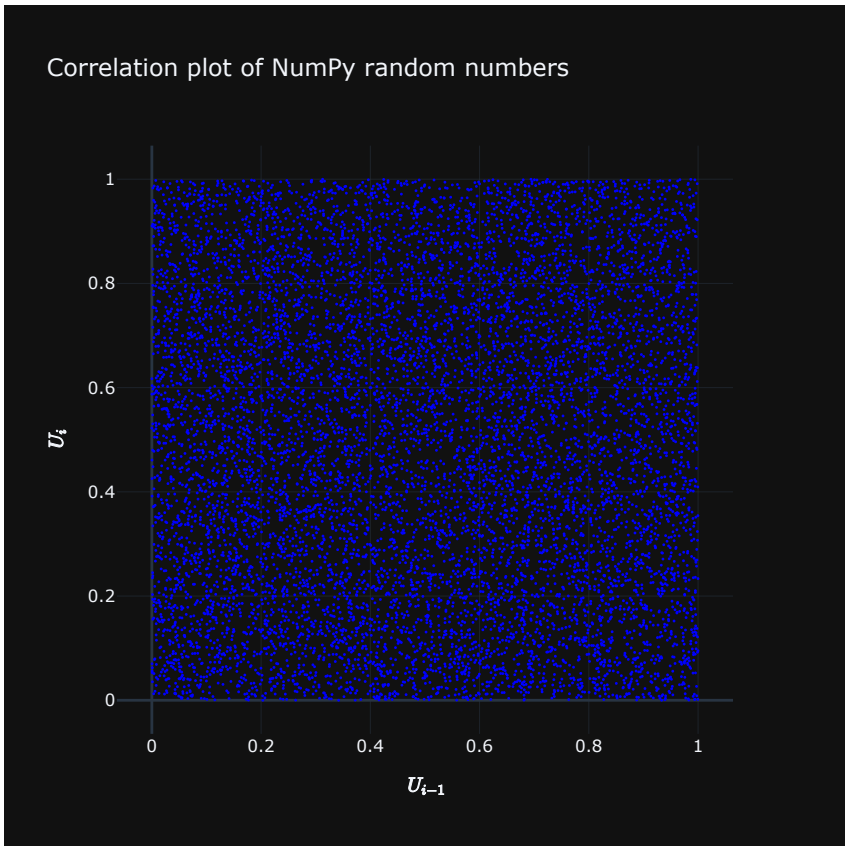
Part 2 - System Available Generator (NumPy)

NumPy uses a Mersenne Twister generator, called MT19937, from the standard C++ library (https://cplusplus.com/reference/random/mt19937/).

In [ ]:
```
# Using numpy to generate random numbers
U = np.random.rand(10000)
```

In [ ]:
```
# Histogram
plot_histogram(U, title="Histogram of NumPy random numbers")

# Plot correlation
plot_correlation(U, title="Correlation plot of NumPy random numbers")

test_random_numbers(U)
```





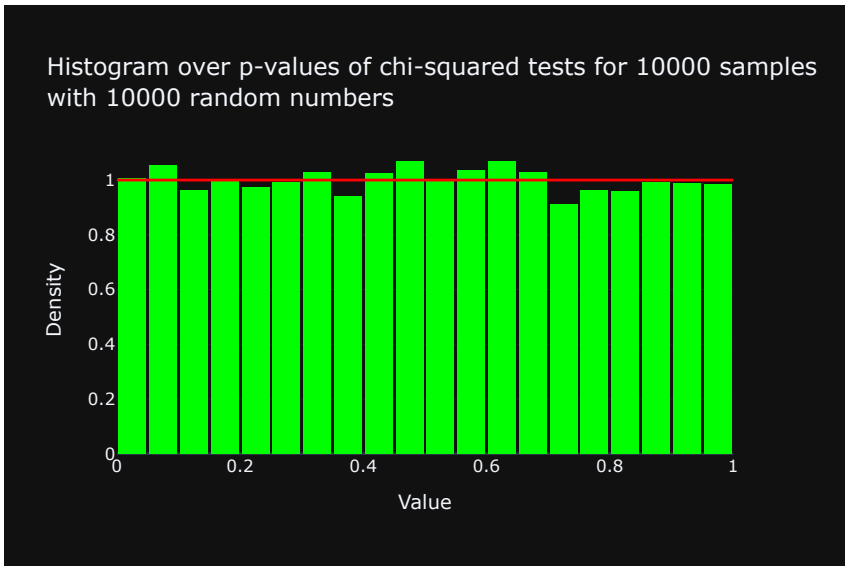
	Test statistic	p-value
Chi squared	11.99	0.21
Kol-Smi	0.01	1.00
Above/Below (I)	5005.00	0.94
Up/Down (II)	6.30	0.39
Up and Down (III)	1.01	0.31
Correlation	0.24	0.81

The numbers generated by NumPy appear to random, independent and uniformly distributed according to our current tests.

Part 3 - Discussion of One Sample Approach

To effectively evaluate a Random Number Generator (RNG), generating only one sample of random numbers is insufficient. Testing a single sample can occasionally produce misleading results, falsely rejecting the hypothesis that the numbers are uniformly distributed or independent. Given the hypothesis is true, the p-values obtained from these tests are expected to be uniformly distributed. Consequently, some samples may exhibit significantly low p-values by chance. Relying on just one sample increases the risk of incorrectly rejecting the hypothesis based on these results. By generating multiple samples, we can determine if the p-values consistently indicate significance or follow a uniform distribution, providing a more robust assessment of the RNG's performance.

In [ ]:
```python
# Run chi-squared test 10000 times
n_samples = 10000
n_random_numbers = 10000
p_values = np.zeros(n_samples)
for i in range(n_samples):
    U = np.random.rand(n_random_numbers)
    _, p_values[i] = chi_sq_test(U, n_classes=10)

# Plot histogram of p-values
plot_histogram(p_values, title=f"Histogram over p-values of chi-squared tests for {n_samples} samples<br>with {n_random_numbers} ra
```



The histogram above shows that the p-values of the chi squared test are uniformly distributed. This indicates that the RNG is working as intended, generating uniformly distributed random numbers.