mioni_rng

August 21, 2021

[1]:
```
import numpy as np
import pandas as pd
from mioni_helper import load_mioni_data
from random_helper_functions import get_bitstring
```

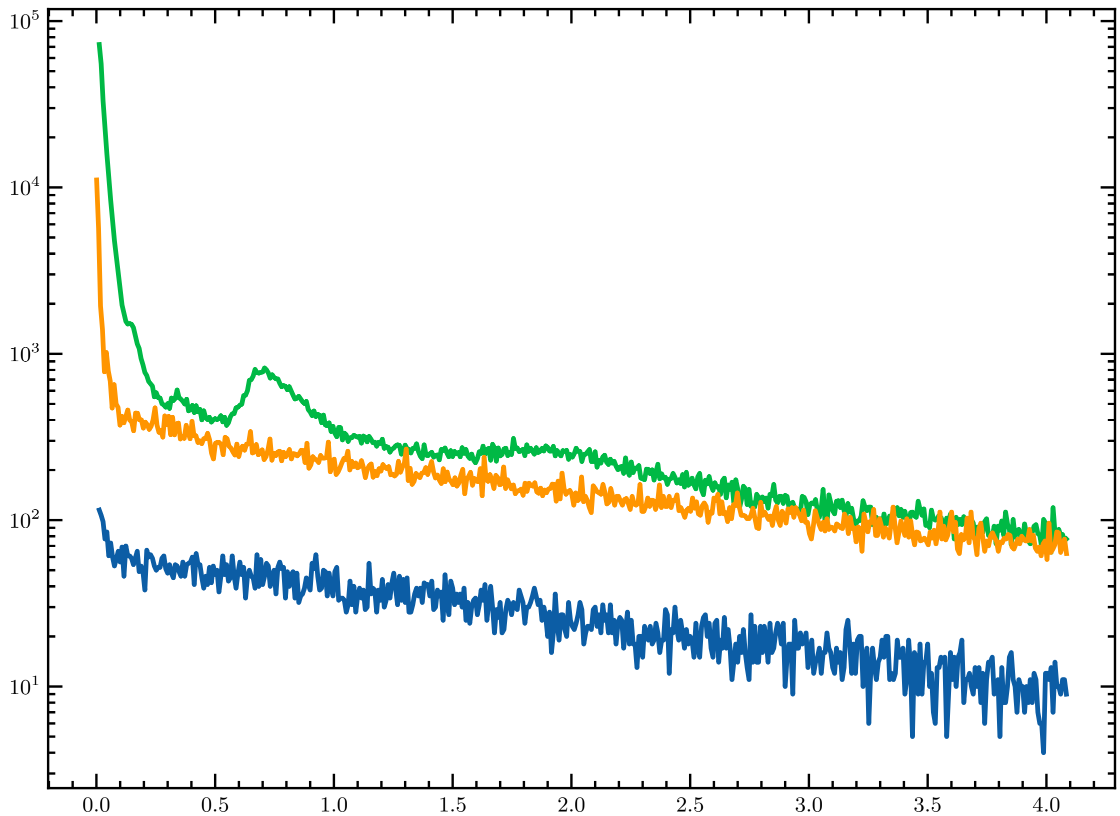
[2]:
```
import sys

import matplotlib.pyplot as plt

sys.path.append('../..')
from plotting.matplotlib_setup import configure_latex, savefig,␣
↪set_size_decorator, savefig, thiner_border

tex_dir, images_dir = 'porocilo/main.tex', 'porocilo/images'

configure_latex(style=['science', 'notebook'], global_save_path=images_dir)

%config InlineBackend.figure_format = 'pdf'
```

1 Meritve

[3]:
```
dfs = load_mioni_data()
```

[4]:
```
for df in dfs:
    ind = df[np.abs(df['N']) < 1].index
    df.drop(ind, inplace=True)

    ind = df[df['us'] < 0].index
    df.drop(ind, inplace=True)
```

[5]:
```
plt.yscale('log')

for df in dfs:
    plt.plot(df['us'], np.abs(df['N']))
```

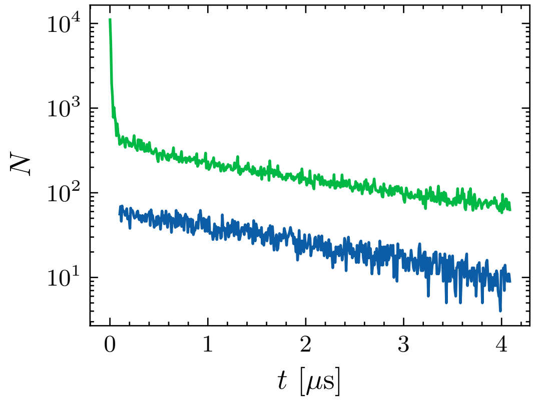


[6]:
```
dfs = [dfs[0], dfs[2]]
```

[7]:
```
dfs[0] = dfs[0][(len(dfs[0]) - len(dfs[1])):]
```

[8]:
```
fig, ax = set_size_decorator(plt.subplots, fraction=0.5, ratio='4:3')(1, 1)

ax.set_yscale('log')

for df in dfs:
    ax.plot(df['us'], np.abs(df['N']), lw=1)

ax.set_xlabel(r'$t$ [$\mu$s]')
ax.set_ylabel('$N$')

# savefig('mioni_meritve')
```

[8]:
```
Text(0, 0.5, '$N$')
```



2 Noise

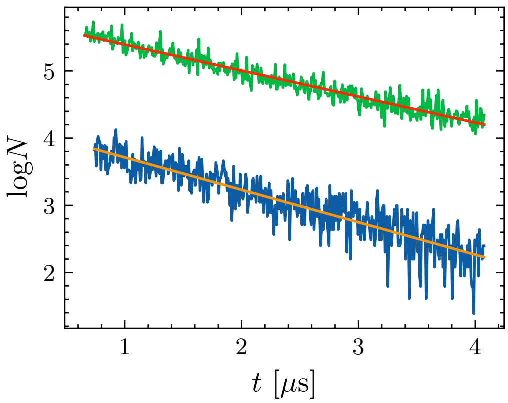
[9]:
```
from scipy.optimize import curve_fit

def lin_func(t, a, b):
    return a * t + b
```

[10]:
```
start_fit, end_fit = 80, -1

ks = []

for df in dfs:
    xdata = df['us'].values[start_fit:end_fit]
    ydata = np.log(df['N'].values[start_fit:end_fit])

    popt, pcov = curve_fit(lin_func, xdata, ydata)

    k, k_err = popt[0], np.sqrt(np.diag(pcov))[0]
    ks.append(popt)
```

[11]:
```
fig, ax = set_size_decorator(plt.subplots, fraction=0.5, ratio='4:3')(1, 1)

cc = 0
cs = [['C0', 'C2'], ['C1', 'C3']]
for k, df in zip(ks, dfs):
    xdata = df['us'].values[start_fit:end_fit]
    ydata = np.log(df['N'].values[start_fit:end_fit])

    ax.plot(xdata, ydata, lw=1, c=cs[cc][0])
    ax.plot(xdata, lin_func(xdata, *k), lw=1, c=cs[cc][1])

    cc += 1
```

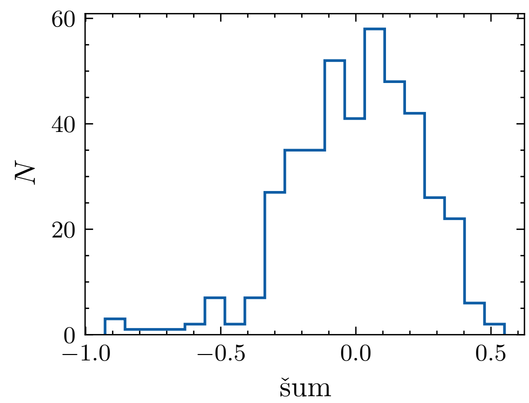
```python
ax.set_xlabel(r'$t$ [$\mu$s]')
ax.set_ylabel('log$N$')

# savefig('mioni_fit')
```

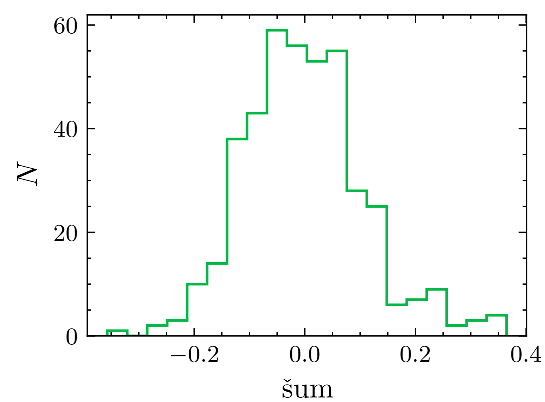
[11]: Text(0, 0.5, 'log$N$')



[12]:
```python
noise = []
ys = []

for k, df in zip(ks, dfs):
    xdata = df['us'].values[start_fit:end_fit]
    ydata = np.log(df['N'].values[start_fit:end_fit])

    n = ydata - lin_func(xdata, *k)

    noise.append(n)
```

[13]:
```python
for i, n in enumerate(noise):
    fig, ax = set_size_decorator(plt.subplots, fraction=0.5, ratio='4:3')(1, 1)
    # plt.plot(range(len(n)), n)
    ax.hist(n, bins=int(np.sqrt(len(n))), histtype='step', lw=1, color=f'C{i}')
    ax.set_xlabel(r'šum')
    ax.set_ylabel('$N$')
    # savefig(f'mioni_sum_{i}')
```





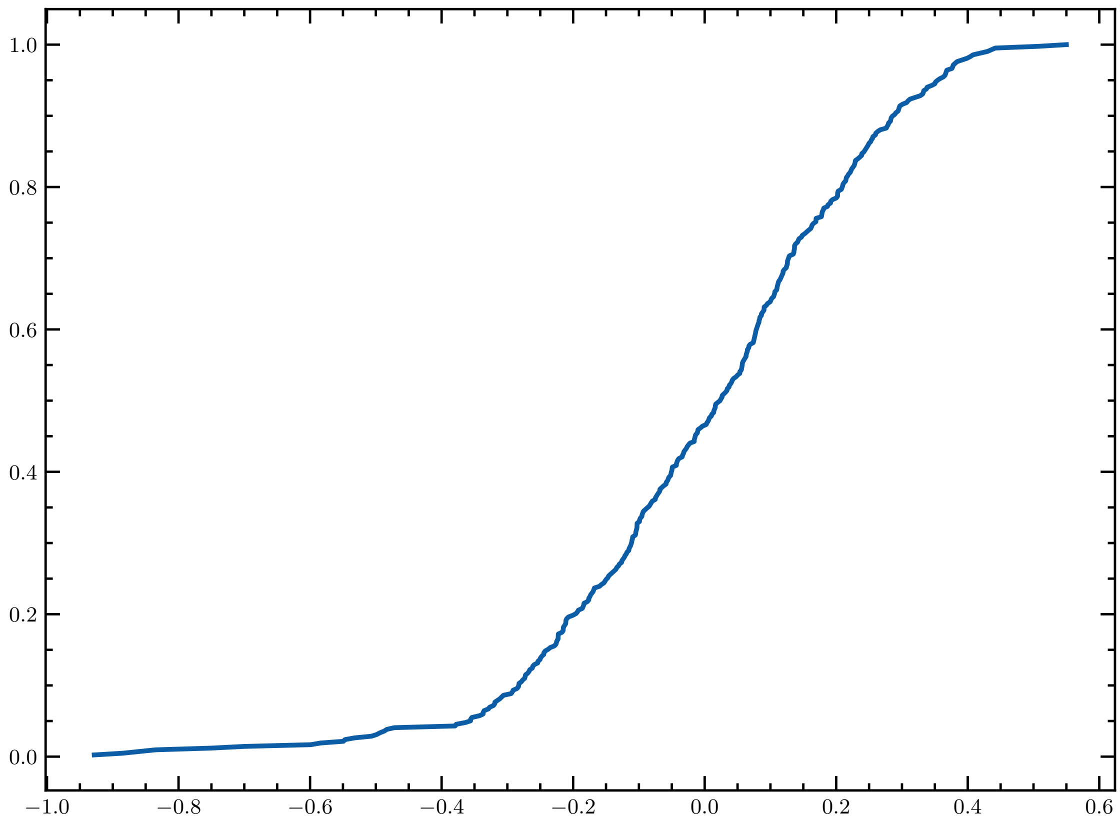
3 ECDF: does not work!!!

https://stackoverflow.com/questions/24788200/calculate-the-cumulative-distribution-function-cdf-in-python
https://en.wikipedia.org/wiki/Probability_integral_transform

[14]:
```
from statsmodels.distributions.empirical_distribution import ECDF

ecdf = ECDF(noise[0])
x, y = ecdf.x, ecdf.y
```

[15]:
```
plt.plot(x, y)
```

[15]:
```
[<matplotlib.lines.Line2D at 0x7fb457806400>]
```



[16]:
```
from scipy.stats import norm
```

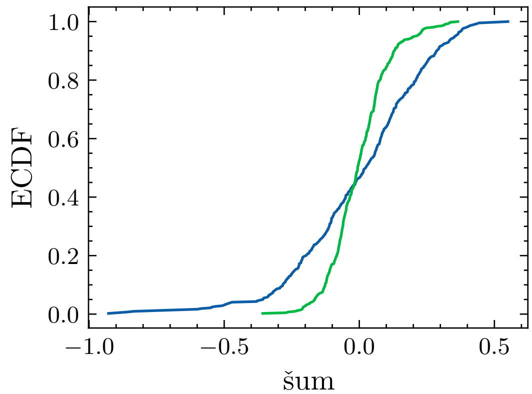
[ ]:
```
def ecdf(a):
    x, counts = np.unique(a, return_counts=True)
    cusum = np.cumsum(counts)

    inv = np.argsort(a)

    cusum = cusum / cusum[-1]

    return x, cusum, inv
```

[ ]:
```
x1, y1, inv1 = ecdf(noise[0])
x2, y2, inv2 = ecdf(noise[1])
```

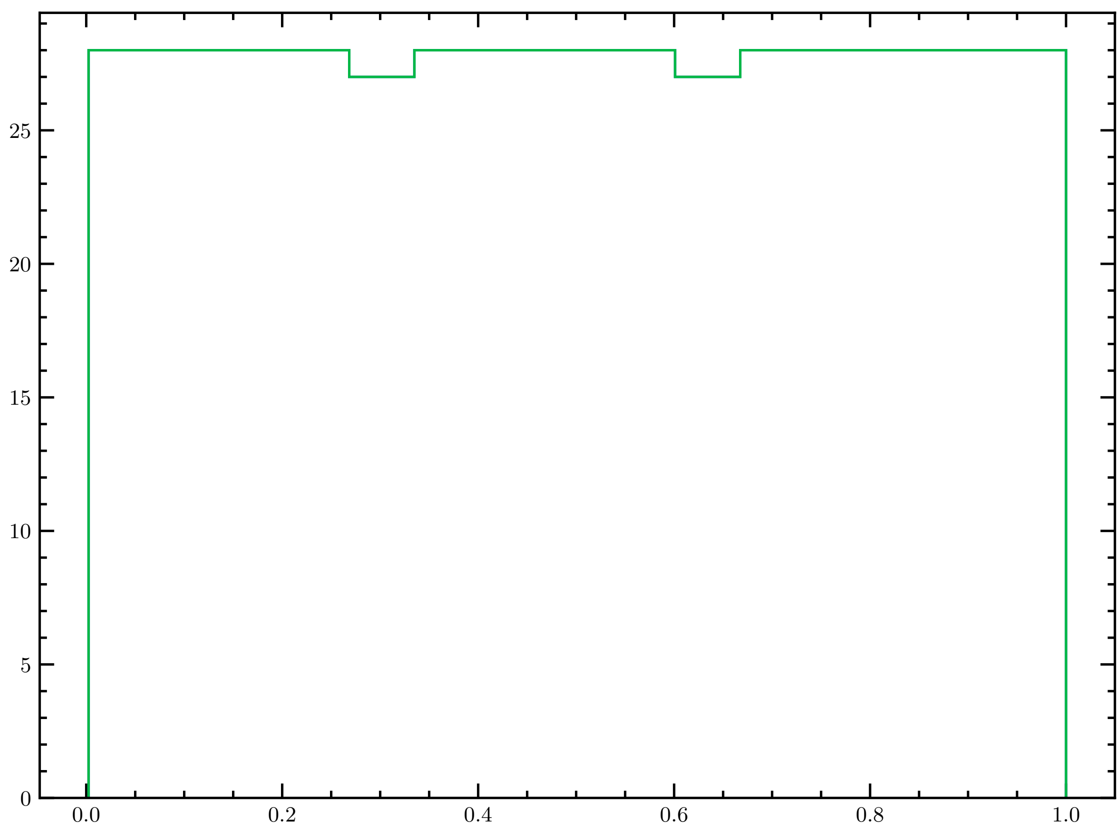
[19]:
```
fig, ax = set_size_decorator(plt.subplots, fraction=0.5, ratio='4:3')(1, 1)

ax.plot(x1, y1, lw=1)
ax.plot(x2, y2, lw=1)
ax.set_xlabel('šum')
ax.set_ylabel('ECDF')
```

```
# savefig('mioni_ecdf')
```

[19]: Text(0, 0.5, 'ECDF')



[20]:
```
plt.hist(y1, bins=15, histtype='step')
plt.hist(y2, bins=15, histtype='step')
plt.show()
```



[21]:
```python
from NIST_tests import RNG_test
from random_helper_functions import float32_to_bin, split_to_arr,␣
↪bin_str_to_matrix
```

[22]:
```python
from numba import njit

@njit
def binary_tree_walk(arr):
    bits = np.zeros(len(arr)).astype(np.int16)
    for i, a in enumerate(arr):
        if a > 0.5:
            bits[i] = 1
    return bits
```

[23]:
```python
rng1 = y1[inv1]
rng2 = y2[inv2]

rng = np.concatenate((rng1, rng2))
rng = rng[rng != 1]
```

[24]:
```python
bits = float32_to_bin(rng2, cut=9)
bits = ''.join(bits)

result1 = RNG_test(bits, short_df=True)
result1
```

```
100%|
                    | 16/16 [00:00<00:00, 29.96it/s]
```

[24]:
```
      0     1     2     3     4     5     6     7    8     9    10    11  \
p  0.73  0.85  0.86  0.19  0.37  0.90  0.15  0.20  nan  0.82  0.00  0.00

     12    13    14
p  0.38  0.06  0.14
```

[25]:
```python
bits = binary_tree_walk(rng2).astype(str)
bits = ''.join(bits)

result2 = RNG_test(bits, short_df=True)
result2
```

```
100%|
                    | 16/16 [00:00<00:00, 326.15it/s]
```

[25]:
```
      0     1     2     3    4     5     6    7    8    9    10    11    12  \
p  1.00  0.20  0.84  0.71  nan  0.25  0.34  nan  nan  nan  0.30  1.00  0.69
```

```
      13    14
p  0.69  0.43
```

CDF ~ U(0, 1) => argsort / max(argsort) ~ U(0, 1) <-> argsort, linspace(0, 1)[argosrt] ~ U(0, 1)

[26]:
```python
rngs = [rng1, rng2]

res = []
b = []
for rng in rngs:
    bits = get_bitstring(rng, length=32, cut=30)
    bits = ''.join(bits)
    b.append(bits)

    test_res = RNG_test(bits, short_df=True)
    res.append(test_res)


for rng in rngs:
    bits = binary_tree_walk(rng).astype(str)
    bits = ''.join(bits)
    b.append(bits)

    test_res = RNG_test(bits, short_df=True)
    res.append(test_res)
```

```
100%|
                    | 16/16 [00:00<00:00, 304.40it/s]
100%|
                    | 16/16 [00:00<00:00, 297.10it/s]
100%|
                    | 16/16 [00:00<00:00, 338.44it/s]
100%|
                    | 16/16 [00:00<00:00, 316.95it/s]
```

[27]:
```python
df = pd.concat([i for i in res])
df.index = [f'$p_{i}$' for i in range(1, len(df)+1)]
df.columns = [i + 1 for i in range(len(df.columns))]
```

[28]:
```python
df
```

[28]:
```
         1     2     3     4    5     6     7    8    9   10    11    12  \
$p_1$  0.94  0.12  0.21  0.71  nan  0.33  0.02  nan  nan  nan  0.10  1.00
$p_2$  0.94  0.12  0.37  0.84  nan  0.12  0.45  nan  nan  nan  0.74  1.00
$p_3$  1.00  0.45  0.01  0.28  nan  0.81  0.00  nan  nan  nan  0.00  1.00
$p_4$  1.00  0.20  0.84  0.71  nan  0.25  0.34  nan  nan  nan  0.30  1.00
```

```
         13    14    15
$p_1$  0.54  0.21  0.26
$p_2$  0.31  0.01  0.76
$p_3$  0.34  0.93  0.40
$p_4$  0.69  0.69  0.43
```

[29]:
```
fig, ax = set_size_decorator(plt.subplots, fraction=0.5, ratio='4:3')(1, 1)

bits_arr = split_to_arr(b[0])
m = bin_str_to_matrix(bits_arr)
ax.matshow(m, cmap='Greys_r')
ax.axis('off')
# savefig('mioni_matrika_dobra', save_format='png', dpi=1000)
```

[29]:
```
(-0.5, 27.5, 27.5, -0.5)
```

4 Reverse Box–Muller transform

[30]:
```
from scipy.optimize import curve_fit

def gauss(x, a, b, c):
    """c...sigma, b...mu, a...normalization"""
    return a * np.exp(-((x - b)**2) / (2 * c**2))


def make_hist_and_fit(dist, func):
    n_bins = int(np.sqrt(len(dist)))
    y, x = np.histogram(dist, bins=n_bins)
    x = x[1:]

    popt, pcov = curve_fit(gauss, x, y)
    return popt, x, y
```

```python
def get_Z(dist):
    popt, _, _ = make_hist_and_fit(dist, gauss)

    sigma, mu = popt[2], popt[1]
    Z = (dist - mu) / sigma

    # Z_popt, _, _ = make_hist_and_fit(Z, gauss) # debug

    return Z
```

[31]:
```python
import sympy as smp
```

[32]:
```python
X1, X2, U1, U2 = smp.symbols('X1, X2, U_1, U_2', real=True)
R, theta = smp.symbols(r'R, \theta', real=True)
```

[33]:
```python
x1 = R * smp.cos(theta)
x1
```

[33]: $R\cos(\theta)$

[34]:
```python
x2 = R * smp.sin(theta)
x2
```

[34]: $R\sin(\theta)$

[35]:
```python
eq1 = smp.Eq(X1, x1)
eq1
```

[35]: $X_1 = R\cos(\theta)$

[36]:
```python
eq2 = smp.Eq(X2, x2)
eq2
```

[36]: $X_2 = R\sin(\theta)$

[37]:
```python
res = smp.solve([eq1, eq2], [R, theta], dict=True)
res
```

[37]:
```
[{R: (-X1**2 + X1*sqrt(X1**2 + X2**2) - X2**2)/(X1 - sqrt(X1**2 + X2**2)),
  \theta: -2*atan((X1 - sqrt(X1**2 + X2**2))/X2)},
 {R: -(X1**2 + X1*sqrt(X1**2 + X2**2) + X2**2)/(X1 + sqrt(X1**2 + X2**2)),
  \theta: -2*atan((X1 + sqrt(X1**2 + X2**2))/X2)}]
```

[38]:
```python
R_, theta_ = res[1][R], res[1][theta]
```

[39]:
```python
R_
```

[39]:

$$-\frac{X_1^2 + X_1\sqrt{X_1^2 + X_2^2} + X_2^2}{X_1 + \sqrt{X_1^2 + X_2^2}}$$

[40]:
```
theta_
```

[40]:

$$-2\operatorname{atan}\left(\frac{X_1 + \sqrt{X_1^2 + X_2^2}}{X_2}\right)$$

[41]:
```
eq1 = smp.Eq(R, R_)
eq1
```

[41]:

$$R = -\frac{X_1^2 + X_1\sqrt{X_1^2 + X_2^2} + X_2^2}{X_1 + \sqrt{X_1^2 + X_2^2}}$$

[42]:
```
eq2 = smp.Eq(theta, theta_)
eq2
```

[42]:

$$\theta = -2\operatorname{atan}\left(\frac{X_1 + \sqrt{X_1^2 + X_2^2}}{X_2}\right)$$

[43]:
```
eq1 = eq1.subs(R, smp.sqrt(-2 * smp.log(U1)))
eq1
```

[43]:

$$\sqrt{2}\sqrt{-\log\left(U_1\right)} = -\frac{X_1^2 + X_1\sqrt{X_1^2 + X_2^2} + X_2^2}{X_1 + \sqrt{X_1^2 + X_2^2}}$$

[44]:
```
U1_sol = smp.solve(eq1, U1)[0] # U1
U1_sol
```

[44]:

$$e^{-\frac{\left(X_1^2 + X_1\sqrt{X_1^2 + X_2^2} + X_2^2\right)^2}{2\left(X_1 + \sqrt{X_1^2 + X_2^2}\right)^2}}$$

[45]:
```
eq2 = eq2.subs(theta, 2 * smp.pi * U2)
eq2
```

[45]:

$$2\pi U_2 = -2\operatorname{atan}\left(\frac{X_1 + \sqrt{X_1^2 + X_2^2}}{X_2}\right)$$

[46]:
```
U2_sol = smp.solve(eq2, U2)[0] # U2
U2_sol
```

[46]:

$$-\frac{\operatorname{atan}\left(\frac{X_1 + \sqrt{X_1^2 + X_2^2}}{X_2}\right)}{\pi}$$

[47]:
```
U1_f = smp.lambdify([X1, X2], U1_sol)
U2_f = smp.lambdify([X1, X2], U2_sol)
```

[48]:
```python
Z1, Z2 = get_Z(noise[0]), get_Z(noise[1])

n1 = U1_f(Z1, Z2)
n2 = U2_f(Z1, Z2)

from scipy import signal

n2 = n2 + 0.5
```

[49]:
```python
from stat_tests import chi2_test, ks_test
```

[50]:
```python
n = [n1, n2]

tests = []
for i in n:
    tests.append([chi2_test(i, n_bins=20), ks_test(i)])
```

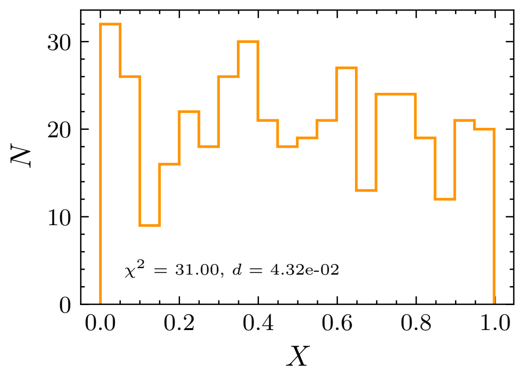
[51]:
```python
tests
```

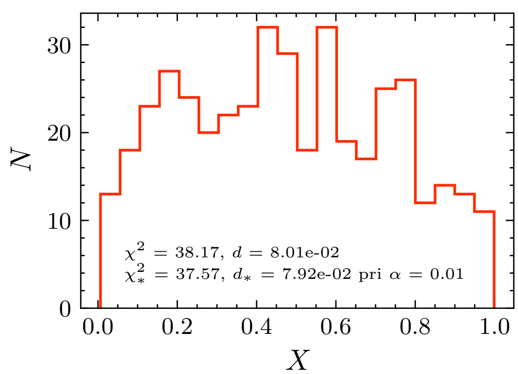
[51]:
```
[[(array([[30.99521531,  0.04042166]]), 37.56623478662507),
  (array([[0.04322233, 0.40424711]]), [0.07917726627266543])],
 [(array([[3.81722488e+01, 5.64228228e-03]]), 37.56623478662507),
  (array([[0.08006591, 0.00887653]]), [0.07917726627266543])]]
```

[52]:
```python
alpha = 0.01

for c, i in enumerate(n):
    fig, ax = set_size_decorator(plt.subplots, fraction=0.5, ratio='4:3')(1, 1)

    ax.hist(i, histtype='step', color=f'C{c+2}', bins=20)

    chi2 = tests[c][0][0][0][0]
    ks = tests[c][1][0][0][0]

    critical_value_chi2 = tests[c][0][1]
    critical_value_ks = tests[c][1][1][0]

    print(critical_value_ks, critical_value_chi2)

    an1 = f'\n$\chi^2$ = {chi2:.2f}, $d$ = {ks:.2e}'
    an2 = f'\n$\chi^2_*$ = {critical_value_chi2:.2f}, $d_*$ =␣
 ↪{critical_value_ks:.2e}'
    an3 = r' pri $\alpha$ = {}'.format(alpha)

    if c == 1:
        an = an1 + an2 + an3
    else:
```

```
        an = an1

    ax.annotate(an, xy=(0.1, 0.1), xycoords='axes fraction', fontsize=6)

    ax.set_xlabel('$X$')
    ax.set_ylabel('$N$')
    savefig(f'box_muller_uniform_{c}')
```

```
0.07917726627266543 37.56623478662507
0.07917726627266543 37.56623478662507
```





[53]:
```
# bits = get_bitstring(n1, length=32)
# bits = ''.join(bits)
bits = binary_tree_walk(n1).astype(str)
bits = ''.join(bits)

res1 = RNG_test(bits, short_df=True)
```

```
100%|
                    | 16/16 [00:00<00:00, 334.50it/s]
```

[54]:
```
# bits = get_bitstring(n2, length=32)
# bits = ''.join(bits)
bits = binary_tree_walk(n2).astype(str)
bits = ''.join(bits)

res2 = RNG_test(bits, short_df=True)
```

```
100%|
                    | 16/16 [00:00<00:00, 348.55it/s]
```

[55]:
```
r = np.concatenate((n1, n2))

# bits = get_bitstring(r, length=32)
# bits = ''.join(bits)
bits = binary_tree_walk(r).astype(str)
bits = ''.join(bits)

res3 = RNG_test(bits, short_df=True)
```

```
100%|
                    | 16/16 [00:00<00:00, 312.06it/s]
```

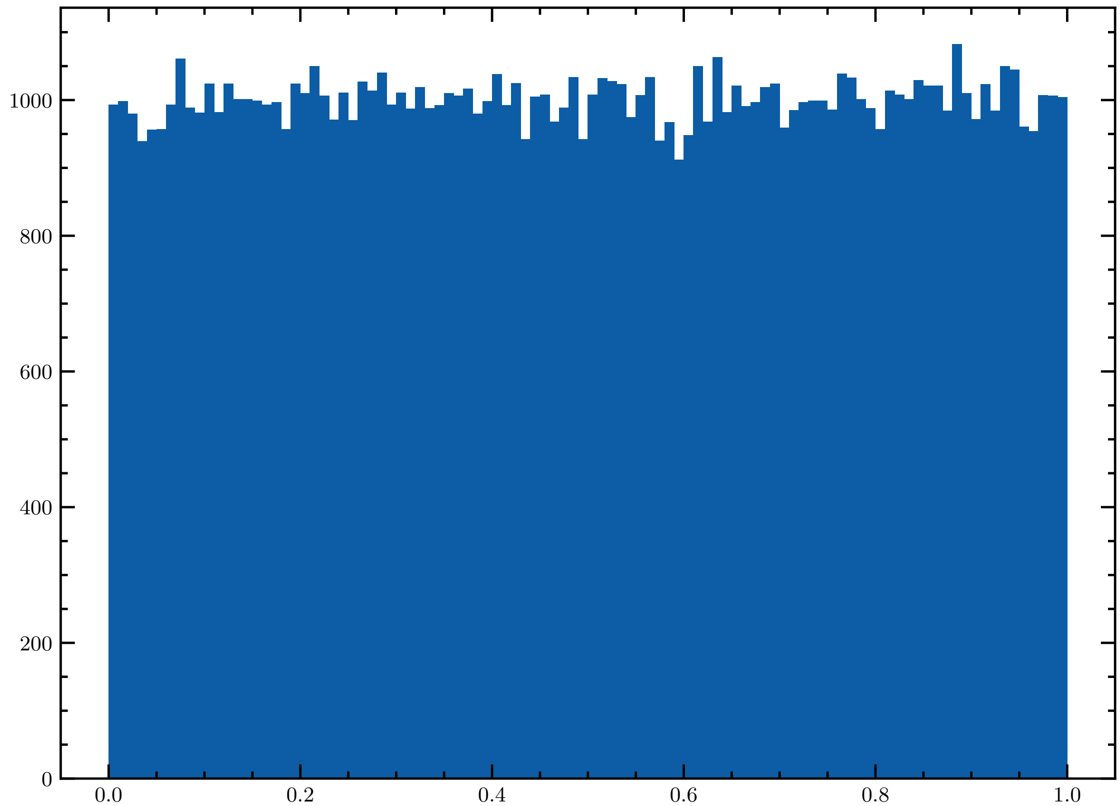
[56]:
```
res = [res1, res2, res3]

df = pd.concat([i for i in res])
df.index = [f'$p_{i}$' for i in range(1, len(df)+1)]
df.columns = [i + 1 for i in range(len(df.columns))]
```

[57]:
```
df
```

[57]:
```
         1     2     3     4    5     6     7    8    9   10    11    12  \
$p_1$  0.33  0.00  0.22  1.00  nan  0.27  0.34  nan  nan  nan  0.22  1.00
$p_2$  0.04  0.00  0.05  0.22  nan  0.49  0.00  nan  nan  nan  0.70  1.00
$p_3$  0.03  0.00  0.02  0.93  nan  0.36  0.00  nan  nan  nan  0.01  1.00

        13    14    15
$p_1$  0.19  0.00  0.54
$p_2$  0.03  0.69  0.84
$p_3$  0.03  0.01  0.36
```

[58]:
```
# test
a = np.random.normal(size=100000)
b = np.random.normal(size=100000)

Z1, Z2 = get_Z(a), get_Z(b)

n1 = U1_f(Z1, Z2)
```

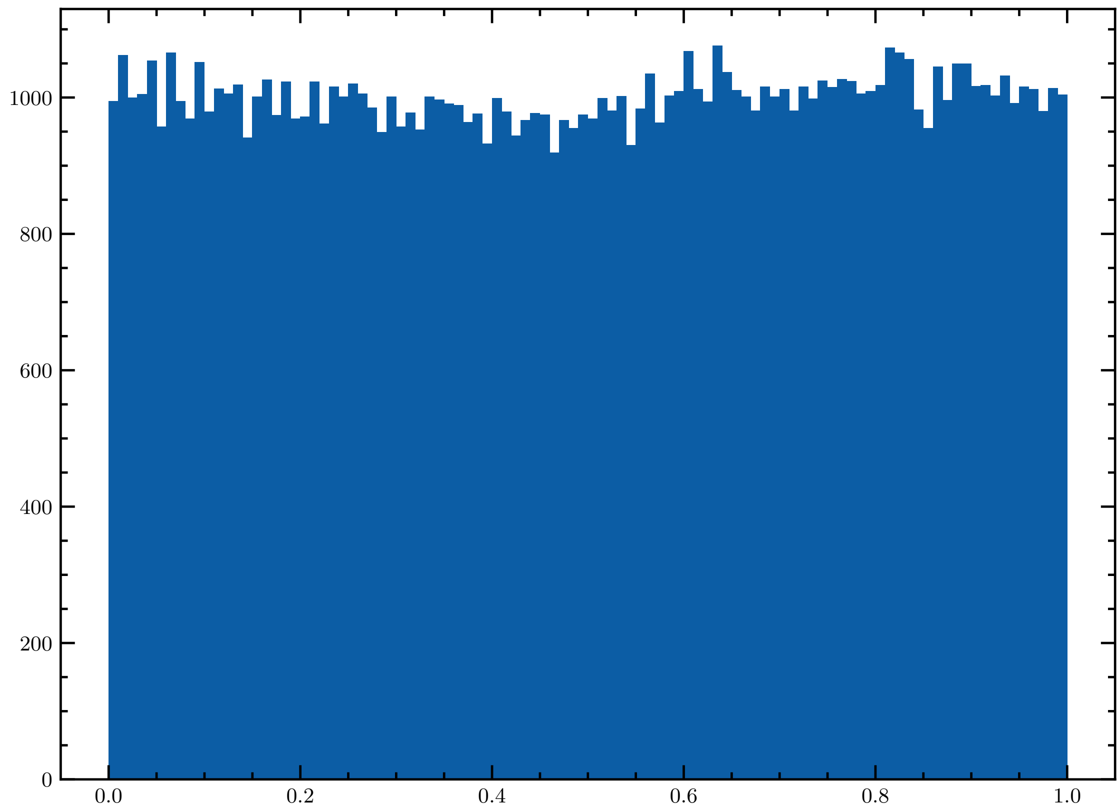
```
n2 = U2_f(Z1, Z2)

n2 = n2 + 0.5

plt.hist(n1, bins=100)
plt.show()
```



[59]:
```
print(ks_test(n1))
print(chi2_test(n1, n_bins=100))
```

```
(array([[0.0023873 , 0.61799027]]), [0.005145321961471824])
(array([[90.45      ,  0.71846975]]), 135.80672317102676)
```

[60]:
```
plt.hist(n2, bins=100)
plt.show()
```



[62]:
```
print(ks_test(n2))
print(chi2_test(n2, n_bins=100))
```

```
(array([[7.13637944e-03, 7.50552966e-05]]), [0.005145321961471824])
(array([[107.834     ,   0.25555649]]), 135.80672317102676)
```

[63]:
```
bits = get_bitstring(n1, length=32)
bits = ''.join(bits)

RNG_test(bits)
```

```
100%|
                    | 16/16 [00:16<00:00,  1.06s/it]
```

[63]:

	test	p
0	Frequency Test (Monobit)	0.05
1	Frequency Test within a Block	0.42
2	Run Test	0.81
3	Longest Run of Ones in a Block	0.41
4	Binary Matrix Rank Test	0.59
5	Discrete Fourier Transform (Spectral) Test	0.19
6	Non-Overlapping Template Matching Test	0.83

```
7           Overlapping Template Matching Test  0.41
8          Maurer's Universal Statistical test  0.74
9                       Linear Complexity Test  0.52
10                                 Serial test  0.73
11                    Approximate Entropy Test  0.02
12              Cummulative Sums (Forward) Test  0.08
13                      Random Excursions Test  0.79
14              Random Excursions Variant Test  0.57
```

[64]:
```python
bits = get_bitstring(n2, length=32)
bits = ''.join(bits)

RNG_test(bits)
```

```
100%|
                   | 16/16 [00:17<00:00,  1.07s/it]
```

[64]:
```
                                          test     p
0                     Frequency Test (Monobit)  0.80
1                Frequency Test within a Block  0.54
2                                     Run Test  0.94
3               Longest Run of Ones in a Block  0.77
4                      Binary Matrix Rank Test  0.65
5    Discrete Fourier Transform (Spectral) Test  0.75
6       Non-Overlapping Template Matching Test  0.60
7           Overlapping Template Matching Test  0.88
8          Maurer's Universal Statistical test  0.87
9                       Linear Complexity Test  0.90
10                                 Serial test  0.21
11                    Approximate Entropy Test  0.75
12              Cummulative Sums (Forward) Test  0.91
13                      Random Excursions Test  0.61
14              Random Excursions Variant Test  0.25
```

[ ]:
```python

```