Introduction II - Random Number Extraction

Gabriele Pompa
gabriele.pompa@unisi.com

March 23, 2020

Contents

Executive Summary

The following sections are organized as follows:

- In Section 1 we generate N samples from a Standard Normal Distribution (that is, a Normal Distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$) and visualize the array of extracted values.
- In Section 2 we plot a histogram of the extracted values.
- In Section 3 we plot a histogram of the empirical distribution of the extracted values.
- In Section 4, we compare the theoretical Standard Normal distribution with the empirical distribution of the extracted values.
- In Section 5 we do a Normal fit to the empirical distribution and compute empirical Skewness and Kurtosis.

- In Section 6 we do Jarque-Bera Normality test to check whether the extracted values are really normally distributed.

Preliminary Imports

These are our preliminary imports to load functionalities not included in the Python Standard Library

[1]:
```
import numpy as np
```

Plotting functionalities can be imported in the Matplotlib Python plotting library. Importing only the `pylab` module is standard and will be enough for our examples too.

[2]:
```
import matplotlib.pylab as plt
```

To request inline plots to the Jupyter Notebook engine, we use the following directive:

[3]:
```
%matplotlib inline
```

Statistical functions are defined in `scipy` library

[4]:
```
from scipy import stats
```

Notice in particular the syntax `from scipy import stats`. This is a case of a package `scipy` from which we decide to load only the `stats` module. Then we can access functions defined in `stats` as usual: `stats.functionName()`

1 Extraction from a Standard Normal Distribution

The Numpy `random` module contains functionalities to generate random samples. By default it uses a Mersenne Twister pseudo-random number generator (PRNG).

A PRNG is initialized by a *seed*. Once the seed is set, the PRNG will produce always the same random sequence of numbers. For us is thus important to set the seed to have reproducible results (e.g. to check errors in our code).

[5]:
```
seed = np.random.seed(987654321)
```

The particular value used to set the seed of the PRNG (here 987654321) is subject of research because it has impact on the quality of the random sequence generated by the PRNG (e.g. on its *period*, that is the length of the random sequence after which the PRNG starts to repeat itself). Thus, don't change it unless you have a very sound reason to do it.

Now, we set how many pseudo-random numbers we want to generate: $N = 10^5$

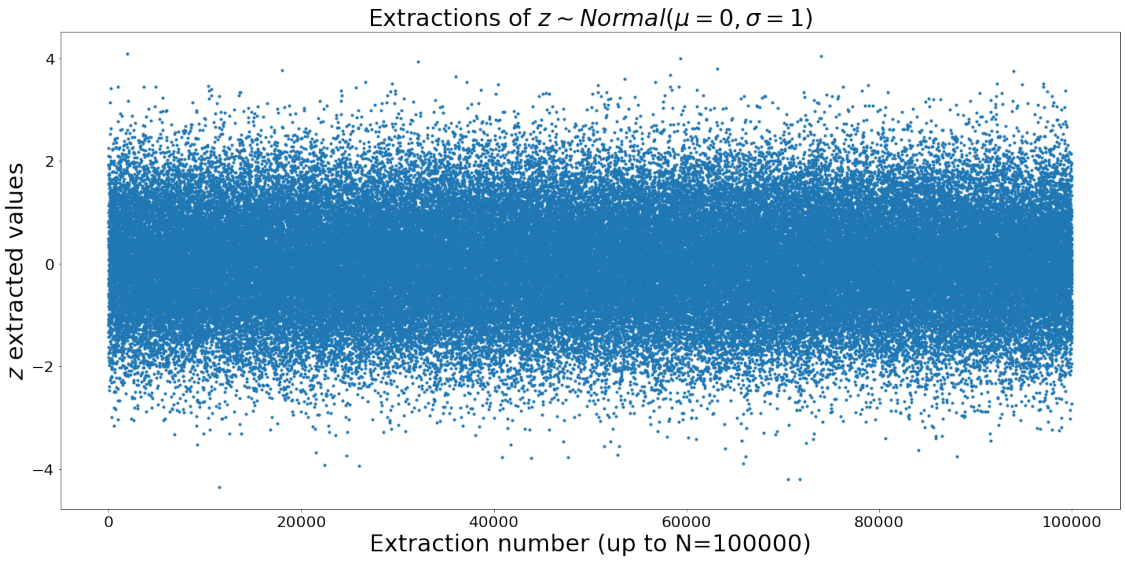
[6]:
```
N = 100000
```

To generate the numbers we simply call the `standard_normal` function defined in the `random` module of `Numpy`

[7]:
```
z = np.random.standard_normal(N)
```

We can visualize the extracted values z using the plotting functionalities of the imported matplotlib.pylab module

[8]:
```
# fig and ax are instances of plot-like objects.
# These objects have functionalities that allow us to modify/specify features␣
 ↪of plot
# (e.g. the x- and y-axis labels, the title, and so forth...)
fig, ax = plt.subplots(figsize=(20,10))

# The plot fuction actually draws the plot of z values.
# The argument 'marker' set the kind of marker of the plot: '.' is for dots␣
 ↪markers
# linestyle="" simply avoid the dots to be connected by lines (remove it and␣
 ↪re-run and see that the dots become connected)
plt.plot(z, marker='.', linestyle="")

# The ax objects has two sub-objects xaxis and yaxis that give us acces to␣
 ↪functionalities of the x-axis and y-axis respctively.
# Here, we set size of the labels of the ticks of the two axis (e.g. 4, 2, 0,␣
 ↪-2, -4 for the y-axis).
ax.xaxis.set_tick_params(labelsize=20)
ax.yaxis.set_tick_params(labelsize=20)

# Here we set the labels of the two axis, providing a text string and the␣
 ↪desired size of the font
ax.set_xlabel('Extraction number (up to N={})'.format(N), fontsize=30)
ax.set_ylabel('$z$ extracted values', fontsize=30)

# Here we set the title of the plot
ax.set_title(r'Extractions of $z \sim Normal(\mu=0, \sigma=1)$', fontsize=30)

# Here we just improve the apperance of the plot, tweaking the spacing to␣
 ↪prevent clipping of ylabel
fig.tight_layout()
plt.show()
```



By default, `plt.plot(z,...)` plots the values of the z array against its index. That is, against an array of integer values rangin from 0 and ending to `len(z) - 1` (here `len(z) == N`).

Qualitatively, we see a cloud of points, more dense around 0 and less at greater and smaller values. This is something we expect, since we extracted these values from a Standard Normal Distribution.

2 Histogram of the Extracted Values

Now we draw a histogram of the z values extracted.

First we set the number of bins, that is the number of intervals in which we partition the extracted values.

[9]:
```
num_bins = 50
```

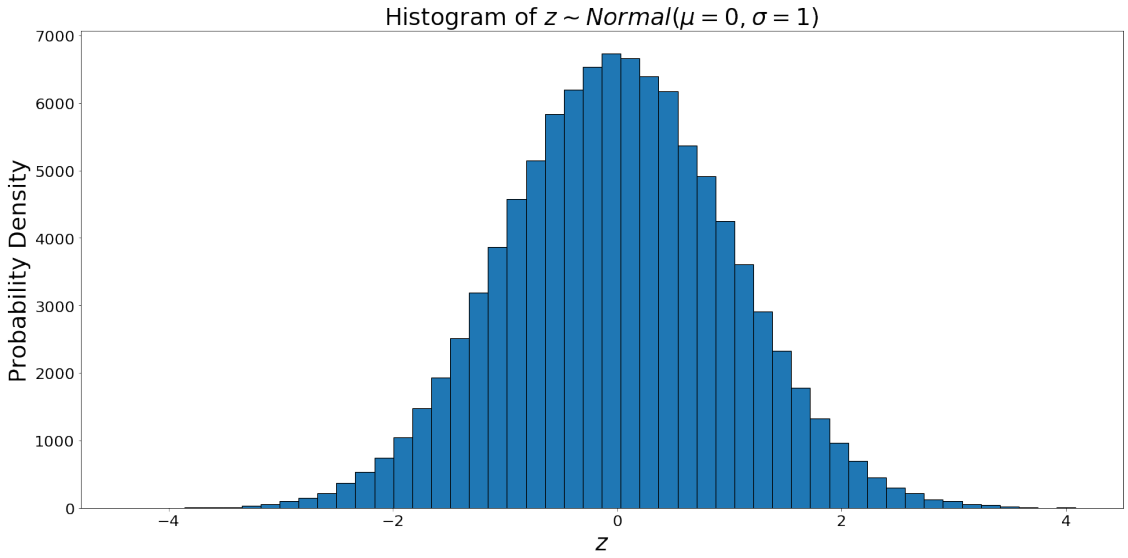
An histogram will then simply be a plot where: - on the x-axis there are the extracted values partitioned in bins; - on the y-axis there are the counts in each bin, that is how many values fall within the range of each single bin

Feel the need to refresh the concept of histogram? This is a good time to do it and Wikipedia, as always, is there to help.

We use the `hist` function, provided by the `matplotlib` library, to actually make the plot.

[10]:
```
fig, ax = plt.subplots(figsize=(20,10))

# This is the only new piece of code: the histogram of the z values is actually␣
↪drwan here.
# histtype='bar' simply says that we want an histogram made of (vertical) bars.
# Different styles can be chosen, take a look at hist's documentation.
# ec='black' simply display in black the borders of the bars.
```

```
heights, _, _ = ax.hist(z, num_bins, histtype='bar', ec='black')

ax.xaxis.set_tick_params(labelsize=20)
ax.yaxis.set_tick_params(labelsize=20)

ax.set_xlabel('$z$', fontsize=30)
ax.set_ylabel('Probability Density', fontsize=30)

ax.set_title(r'Histogram of $z \sim Normal(\mu=0, \sigma=1)$', fontsize=30)

fig.tight_layout()
plt.show()
```



To understand the syntax of `heights, _, _ = ax.hist(z, ...)` simply consider that the elements of an array in Python can be assigned directly to variables. For example this can be done:

[11]:
```
a, b, c = np.array([10, 20, 30])

print(a)
print(b)
print(c)
```

```
10
20
30
```

Now, function `hist` returns an array of 3 values. We capture the first element only and assign it to the variable `heights` while we skip the capture of the other 2 using the underscores `_` character. This means that the other two outputs of `hist` function will be intentionally lost.

The variable `heights` is itself an array of length `num_bins == 50` representing the *height* of each single bin.

How do I know this? Easy! Take a look at *Returns* section of `hist` function's documentation for the meaning of its output and at *Parameters* section for its arguments.

[12]:
```
heights
```

[12]:
```
array([3.000e+00, 0.000e+00, 3.000e+00, 8.000e+00, 6.000e+00, 1.300e+01,
       3.400e+01, 5.400e+01, 1.080e+02, 1.520e+02, 2.180e+02, 3.660e+02,
       5.330e+02, 7.460e+02, 1.041e+03, 1.478e+03, 1.930e+03, 2.519e+03,
       3.187e+03, 3.870e+03, 4.578e+03, 5.150e+03, 5.832e+03, 6.193e+03,
       6.535e+03, 6.728e+03, 6.667e+03, 6.394e+03, 6.170e+03, 5.374e+03,
       4.913e+03, 4.248e+03, 3.614e+03, 2.913e+03, 2.323e+03, 1.778e+03,
       1.326e+03, 9.650e+02, 7.020e+02, 4.560e+02, 3.020e+02, 2.230e+02,
       1.220e+02, 9.800e+01, 5.500e+01, 4.300e+01, 1.900e+01, 4.000e+00,
       2.000e+00, 4.000e+00])
```

[13]:
```
len(heights)
```

[13]:
```
50
```

Let's label the bins with the index the index $i = 0, ..., \text{num\_bins} - 1$ (in our case `num_bins == 50`) and let the array h_i represent the height of bin i:

$$h_0, ..., h_{\text{num\_bins}-1}$$

By definition of histogram, the sum of the heigts h_i in each single bin i must be equal to the total numbers extracted (in this case `N == 100000`)

$$\text{SUM OF BIN HEIGHTS} = \sum_{i=0}^{\text{num\_bins}-1} h_i = N$$

which allow us to legitimately interpret the height of each bin as the counts of z values falling in that bin:

$$h_i : \text{count of the number of } z \text{ values extracted that fall in the } i\text{-th bin}$$

Let's check whether this is `True` or `False` using the function `sum`, defined in Numpy:

[14]:
```
np.sum(heights) == N
```

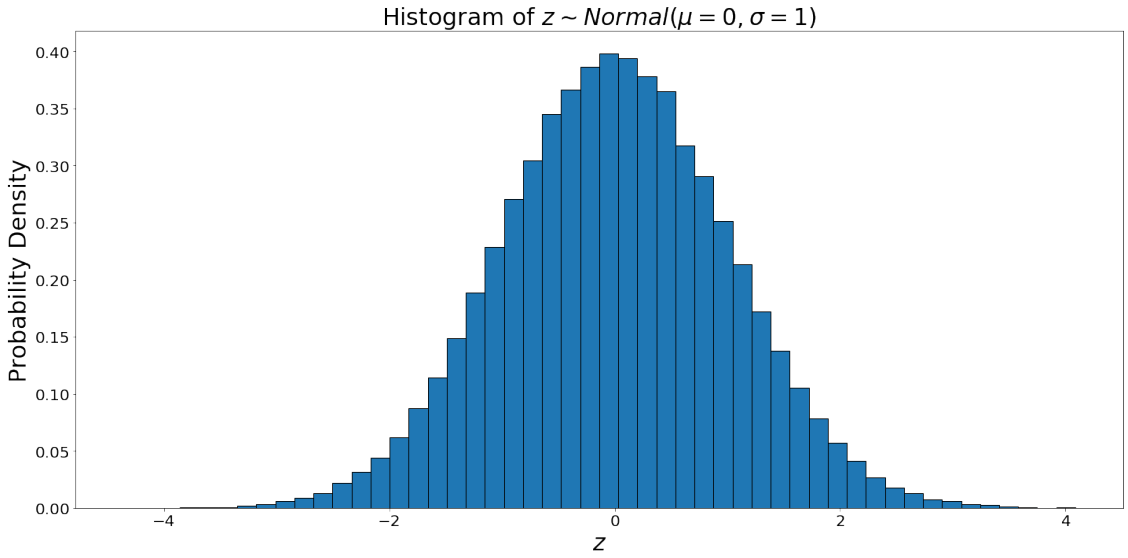
[14]:
```
True
```

Notice the use of the `==` operator which is an operator that check whether the two arguments on its sides are equal. It returns as output a boolean variable `True` or `False`.

3 Histogram of the Distribution of the Extracted Values

Let's now draw a histogram of the empirical distribution of the z values extracted.

[15]:
```
fig, ax = plt.subplots(figsize=(20,10))

# This is the only new piece of code: the normalized histogram of the z values␣
↪is actually drwan here.
# density=True sepcifies that we want a normalized histogram (see below)
# bin_edges is an array of num_bins + 1 values representing the edges (from␣
↪left to right) of the bins
heights, bin_edges, _ = ax.hist(z, num_bins, density=True, histtype='bar',␣
↪ec='black')

ax.xaxis.set_tick_params(labelsize=20)
ax.yaxis.set_tick_params(labelsize=20)

ax.set_xlabel('$z$', fontsize=30)
ax.set_ylabel('Probability Density', fontsize=30)

ax.set_title(r'Histogram of $z \sim Normal(\mu=0, \sigma=1)$', fontsize=30)

fig.tight_layout()
plt.show()
```



We can see that the edges of the bins (wich are automatically chosen by the `hist` function) qualitatively span the range of z values generated (roughly from -4 to 4).

The second output of `hist` function, captured by `bin_edges` variable, is an array be_i of `num_bins + 1 == 51` values:

$$be_0, ..., be_{\text{num\_bins}}$$

ordered in ascending order. Array be_i represents the edges of the bins (num_bins left edges and the right edge of last bin). Again, see `hist` function's documentation for details.

[16]:
```
bin_edges
```

[16]:
```
array([-4.36361809, -4.19457925, -4.02554042, -3.85650159, -3.68746276,
       -3.51842393, -3.34938509, -3.18034626, -3.01130743, -2.8422686 ,
       -2.67322977, -2.50419093, -2.3351521 , -2.16611327, -1.99707444,
       -1.82803561, -1.65899677, -1.48995794, -1.32091911, -1.15188028,
       -0.98284145, -0.81380261, -0.64476378, -0.47572495, -0.30668612,
       -0.13764729,  0.03139155,  0.20043038,  0.36946921,  0.53850804,
        0.70754687,  0.87658571,  1.04562454,  1.21466337,  1.3837022 ,
        1.55274103,  1.72177987,  1.8908187 ,  2.05985753,  2.22889636,
        2.39793519,  2.56697402,  2.73601286,  2.90505169,  3.07409052,
        3.24312935,  3.41216818,  3.58120702,  3.75024585,  3.91928468,
        4.08832351])
```

[17]:
```
len(bin_edges)
```

[17]:
```
51
```

Noticed the parameter `density=True` of `hist`? This specifies that we want a *normalized* histogram, that is a histogram in which the area under the histogram sums to 1. We can check whether this is true.

Let's define as bw_i the width of the i-th bin:

$$bw_i = be_{i+1} - be_i$$

therefore, there will be num_bins bind widths (as many as the bins of course)

$$bw_0, ..., bw_{\text{num\_bins}-1}$$

Bin widths array `bin_widths` can be easily computed in Python *slicing* the `bin_edges` array:

[18]:
```
bin_widths = bin_edges[1:] - bin_edges[:-1]
```

where: - `bin_edges[1:]` is a slice of the array `bin_edges` from the 2nd element to the last one - `bin_edges[:-1]` is a slice of the array `bin_edges` from the 1st element to the next to the last

Take a look at Numpy Quickstart Tutorial for more informations on array indexing and slicing. More on this in a future lesson.

[19]:
```
len(bin_widths)
```

[19]:
```
50
```

[20]:
```
bin_widths
```

[20]:
```
array([0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883,
       0.16903883, 0.16903883, 0.16903883, 0.16903883, 0.16903883])
```

We see that the bins, which are chosen for us by `hist` function, have fixed width.

The normalized histogram condition is:

$$\begin{aligned}\text{AREA UNDER HISTOGRAM} &= \sum_{i=0}^{\text{num\_bins}-1} \text{AREA } i\text{-th BIN RECTANGLE} \\ &= \sum_{i=0}^{\text{num\_bins}-1} i\text{-th BIN HEIGHT} \times i\text{-th BIN BASE} \\ &= \sum_{i=0}^{\text{num\_bins}-1} h_i \times bw_i \\ &= 1\end{aligned}$$

that in Python is

[21]:
```
sum(heights * bin_widths)
```

[21]:
```
1.0000000000000002
```

4 Comparing Empirical and Theoretical Distribution of the Extracted Values

We now want to compare qualitatively the empirical distribution of z, as described by the normalized histogram, with the teoretical distribution from which we have sampled z.

We have sampled z from a standard normal distribution, thus the theoretical probability density function $pdf(z)$ of z is the Gaussian bell:

$$pdf(z) = N(z; \mu = 0, \sigma = 1) = \frac{1}{\sqrt{2\pi}} \exp -\frac{z^2}{2\sigma^2}$$

This pdf is implemented in the `stats.norm` module as `pdf` function, which takes in input: - the data points over which $pdf(z)$ has to be evaluated - the location parameter `loc` for the mean μ - the scale parameter `scale` for the standard deviation σ

See `scipy.stat.norm` documentation for details on `pdf` and other properties of o normal random variable already implemented in Python (e.g. the cumulative density function, the moments, …)

For aesthetical reasons only, we evaluate the standard normal `pdf` over a uniformly spaced grid of points ranging between the minimum and maximum extracted values of z. We use Numpy function `linspace` to do this. Check `linspace` documentation for details.

[22]:
```
zmin, zmax = min(z), max(z)
z_unif_grid = np.linspace(zmin, zmax, N)
z_unif_grid
```

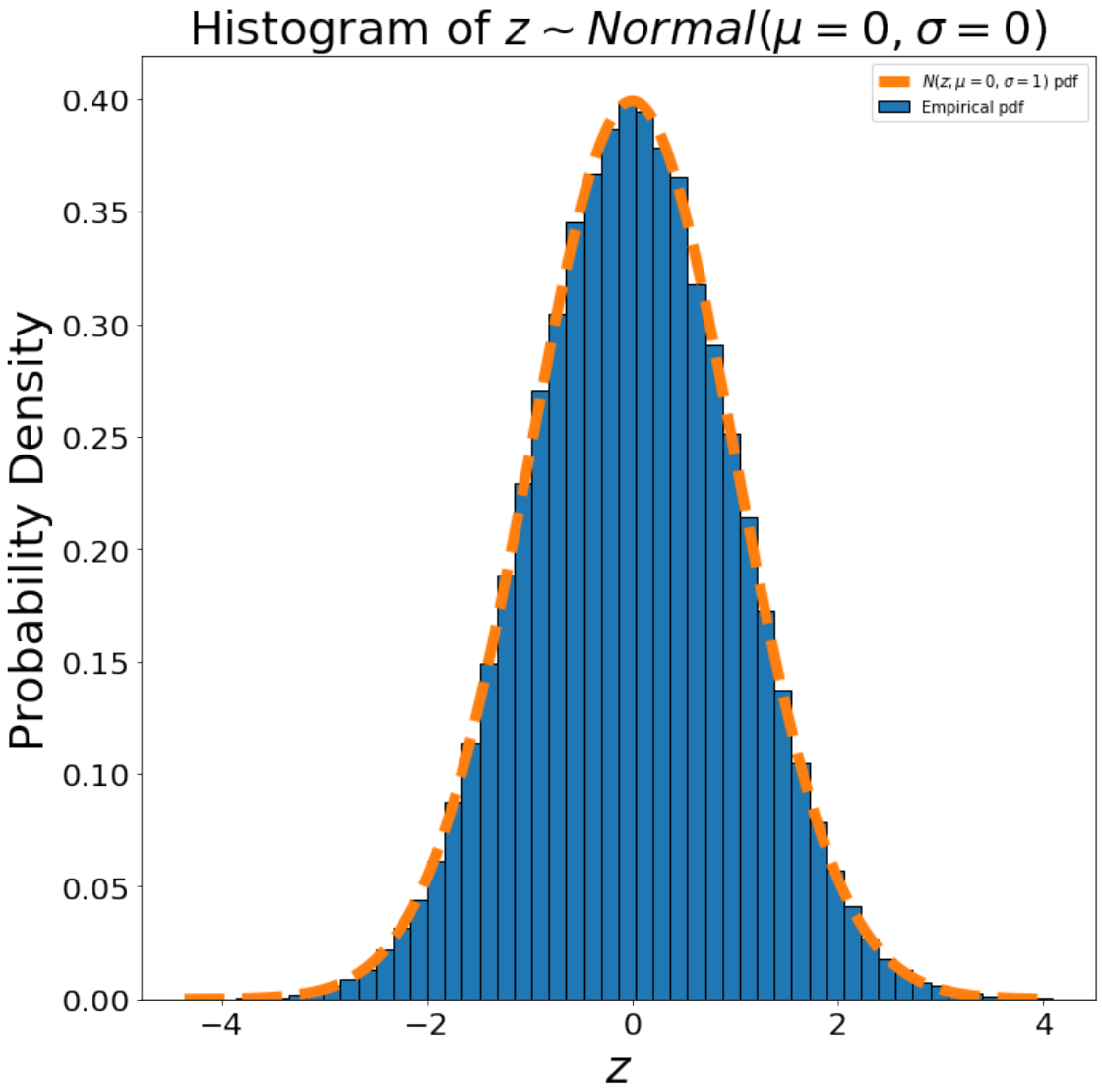
[22]:
```
array([-4.36361809, -4.36353357, -4.36344905, …,  4.08815447,
        4.08823899,  4.08832351])
```

[23]:
```
stdn_pdf_z = stats.norm.pdf(z_unif_grid, loc=0.0, scale=1.0)
stdn_pdf_z
```

[23]:
```
array([2.92532485e-05, 2.92640394e-05, 2.92748340e-05, …,
       9.36972210e-05, 9.36648509e-05, 9.36324913e-05])
```

[24]:
```
fig, ax = plt.subplots(figsize=(10,10))

# Here we don't capture any output from hist function
# label parameter is the label displayed in legend
ax.hist(z, num_bins, density=True, histtype='bar', ec='black', label="Empirical␣
 ↪pdf")

# This is the only new piece of code: bell-shaped standard normal pdf-z is␣
 ↪drawn here.
# On the x-axis we plot the uniform grid of z values
# On the y-axis we plot the corresponding values of the Standard Normal density␣
 ↪function
# the dashed line is selected adding argument '--'
# lw=7 sets the width of the line
# label parameter is the label displayed in legend
ax.plot(z_unif_grid, stdn_pdf_z, '--', lw=7, label="$N(z;\mu=0, \sigma=1)$ pdf")

ax.xaxis.set_tick_params(labelsize=20)
ax.yaxis.set_tick_params(labelsize=20)

ax.set_xlabel('$z$', fontsize=30)
ax.set_ylabel('Probability Density', fontsize=30)

ax.set_title(r'Histogram of $z \sim Normal(\mu=0, \sigma=0)$', fontsize=30)
```

```
# Here we add the legend
ax.legend(loc='upper right', ncol=1)

fig.tight_layout()
plt.show()
```



5 Normal Fit of the Distribution of the Extracted Values

5.1 Normal fit

We first do a Normal fit to z. This means the following:

- we assume a normal density function as theoretical model for the probability density function $pdf(z)$ of z

$$pdf(z) = N(z; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{z-\mu}{\sigma}\right)^2}$$

- we find the mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$ such that $N(z; \hat{\mu}, \hat{\sigma})$ best matches the empirical distribution of z (i.e. the normalized histogram)

To get $\hat{\mu}$ and $\hat{\sigma}$ we simply need to call the `fit` function defined in the `norm` class (a class for Normal random variables) of `scipy.stats`. More on classes and objects in a future lesson.

[25]:
```
mu_hat, sigma_hat = stats.norm.fit(z) # get mean and standard deviation
```

[26]:
```
mu_hat, sigma_hat
```

[26]:
```
(0.0022515048620656945, 1.001399256630773)
```

As the $\hat{\mu}$ and $\hat{\sigma}$ fitted values suggest, empirical z values roughly have zero mean and unit standard deviation.

5.2 Higher moments: Skewness and Kurtosis

We now go one step further. We want to test whether, not only the first two moments, but the empirical distribution itself is compatible with the hypothesis that z values are sampled from a Normal distribution. To do this, we consider higher moments: skewness and kurtosis.

For any random variable Z, if we denote the theoretical mean and standard deviation as $\mu = E[Z]$ and $\sigma = E[(Z-\mu)^2]$, respectively:

- Skewness S is the third standardized moment of Z

$$S(Z) = E\left[\left(\frac{Z-\mu}{\sigma}\right)^3\right]$$

  and measures the *asymmetry* of the distribution of Z.

- Kurtosis K is the fourth standardized moment of Z

$$K(Z) = E\left[\left(\frac{Z-\mu}{\sigma}\right)^4\right]$$

  and measures the *tailedness* of the distribution of Z. Notice how we distinguish between Z random variable (capital letter) and z values (lower case) extracted from the $pdf(z)$ of Z.

For Normal random variables (that is, if $Z \sim pdf(z) = N(z; \mu, \sigma)$), it can be shown: - $S(Z) = 0$. This is intuitive, since the Gaussian bell is symmetrical around μ; - $K(Z) = 3$. This is a result.

We can compute the empirical Skewness $\hat{S}$ and excess Kurtosis $\hat{K} - 3$ from our sample of z values as `stats.skew` and `stats.kurtosis` of `scipy`, respectively:

[27]:
```
skew_hat = stats.skew(z)
skew_hat
```

[27]:
```
0.01682179911154367
```

[28]:
```
kurt_hat = stats.kurtosis(z) # this returns the sample excess kurtosis, that is␣
↪sample kurtosis - 3
kurt_hat
```

[28]:
```
0.0002920451221517517
```

Both values are reasonably in line with those expected for a normal distribution.

6 Jarque-Bera test for Normality

6.1 Introduction

We test for normality using a Jarque-Bera test: a goodness-of-fit test of whether sample data have the skewness and kurtosis matching a normal distribution. That is the null-hypothesis (aka H_0) of a Jarque Bera test is

H_0: joint hypothesis of the skewness being zero ($S = 0$) and the excess kurtosis being zero ($K - 3 = 0$).

Jarque-Bera test statistics JB is defined as:

$$JB = \frac{N}{6}\left(\hat{S}^2 + \frac{1}{4}(\hat{K} - 3)^2\right)$$

where N is the number of observations, $\hat{S}$ and $\hat{K}$ are sample estimates of skewness and kurtosis, respectively.

The test statistic JB is always nonnegative. If it is far from zero, it signals the data do not have a normal distribution.

6.2 Python implementation

Jarque-Bera test is implemented in `stats.jarque_bera` function of `scipy`, returning: - `JB`: the JB test statistics; - `JB_p_value`: the p-value p for the H_0 hypothesis of normality.

[29]:
```
JB, JB_p_value = stats.jarque_bera(z)
```

[30]:
```
JB
```

[30]:
```
4.716570798957913
```

This value is reasonably small so that we can expect our z samples to be roughly normal.

6.3 JB replication

We can replicate the `JB` value returned by `stats.jarque_bera`:

[31]:
```
JB_replicated = (N/6) * (skew_hat**2 + 0.25*kurt_hat**2)
JB_replicated
```

[31]:
```
4.716570798957909
```

Due to rounding errors, the comparison cannot be exact…

[32]:
```
JB_replicated == JB
```

[32]:
```
False
```

…but only exact within an error epsilon at most $\epsilon = 10^{-14}$, which is btw surprisingly small anyway (`np.abs` returns the absolute value of its argument):

[33]:
```
best_epsilon = 1e-14
np.abs(JB_replicated - JB) < best_epsilon
```

[33]:
```
True
```

[34]:
```
fail_epsilon = 1e-15 # fail_epsilon < best_epsilon
np.abs(JB_replicated - JB) < fail_epsilon
```

[34]:
```
False
```

6.4 p-value

To provide statistical significance to the conclusion that z samples are normally distributed, we look at the p-value of the test.

[35]:
```
JB_p_value
```

[35]:
```
0.0945822550304295
```

Since $p \approx 0.09 > \alpha = 0.05$, we conclude that the Normality hypothesis H_0 is not rejected with $\alpha = 5\%$ significance level. In other words this means that the probability of rejecting the null hypothesis H_0 given that it is true is at most 5% (this probability is conventionally called Type I error rate).

Observe that: having fixed the significance level α (typically $\alpha = 5\%$), looking at the p-value p of the test, you: - reject H_0 if $p < \alpha$ - fail to reject H_0 if $p > \alpha$ (notice that you do not *accept* H_0)

Usage of p-value of statistical tests can be found here.