MoMeDa_1

May 5, 2024

1 Moderne Methoden der Datenanalyse SS2024

2 Practical Exercise 1

2.1 A Short Introduction to Jupyter Notebooks

You can find some detailed information on the jupyter notebook concept in the documentation of the project.

Here are some basic instructions: - Each code block, a so called "cell", can be executed by pressing **shift + enter**. - You can run multiple cells by marking them and then pressing **shift + enter** or via the options in the "Run" menu in the top bar. - The order of execution matters! The order in which the cells have been executed is indicated by the integers in the brackets to the left of the cell. For instance `In [1]` was executed first. This means that code at the end of the notebook can affect the code at the beginning, if the cells at the beginning are executed after the cells at the end. - You can change between three cell types in Jupyter Lab: "Code", "Markdown" or "Raw". * The "Code" cells will be interpreted by Python. * The "Markdown" cells will be rendered with Markdown and can be used for documentation and text, such as this cell. You can use them for your answers and also add LaTeX formulas such as $f(x) = \frac{1}{x}$. If you double click **this** Markdown cell you can see the raw code of this LaTeX equation. By pressing **shift + enter** the cell will be rendered with Markdown again. * The "Raw" cells won't be interpreted at all. - If you want to reset your notebook, to *"forget"* all the defined functions, classes and variables, go to `Kernel -> Restart Kernel` in the top bar. Your code and text will remain untouched when doing this. - If you write or read files and provide only the file name, the notebook will look for the file in the directory it is located itself. Use relative paths or absolute paths to read or write files from or to somewhere else.

For some more information on the JupyterLab interface and some useful shortcuts you can check out: - the JupyterLab interface documentation - an overview of some shortcuts - and another shortcut overview

If you have any issues with the notebook or additional questions, contact your tutors or try google.

2.2 Exercise 1

To complete the exercises, follow the steps described within this notebook and fill in the blank parts of the code.

You can make use of common Python packages such as numpy or pandas for handling of data and matplotlib for plotting. Alternatively, you can use CERN's ROOT to solve the exercises. Hints for both approaches will be provided.

Some of the cells in this template will throw errors, as some code is missing! It is your job to add the code!

You do not have to implement both the Python and the ROOT approach to the exercises! Simply delete the cells containing the templates and hints containing the approach you choose not to use.

2.2.1 Exercise 1.1

Write a code snippet (function, class, etc.) that - creates `N` Gaussian distributed random numbers with mean `m=0` and a standard deviation of sigma `s=1` and - plots these numbers as a histogram.

The parameter `N` should be an argument of the code snippet.

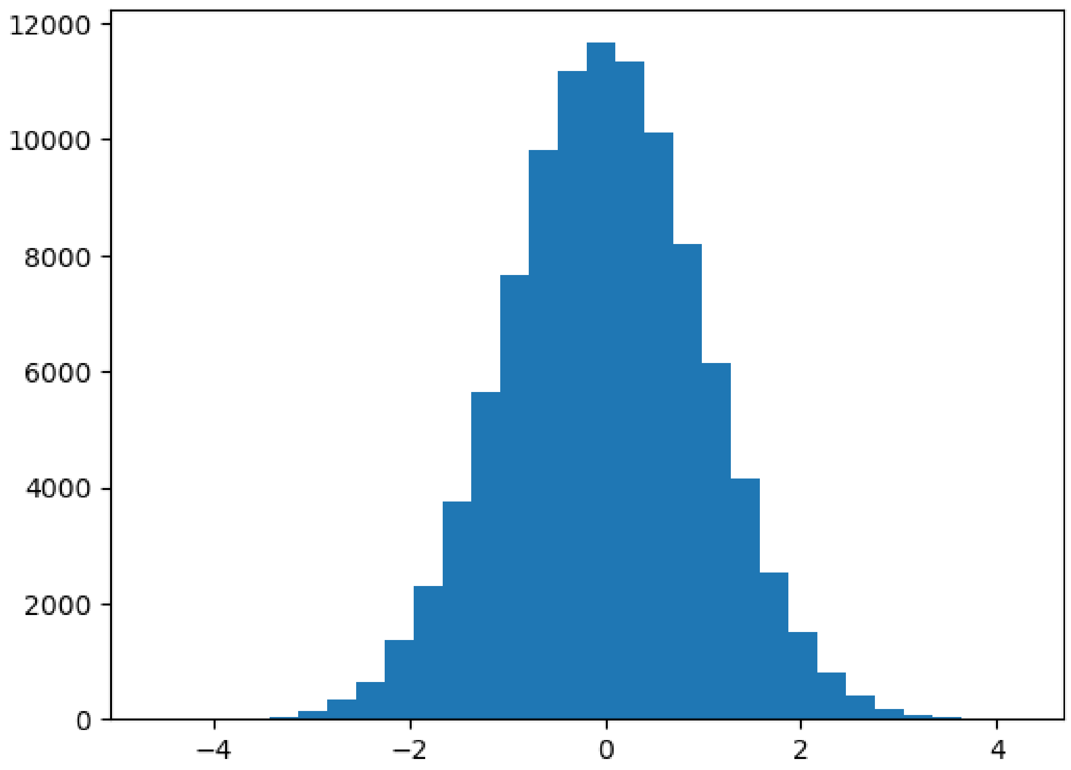
Python Approach: If you want to use the Python approach, you should have a look at the package `numpy` and the therein provided method `numpy.random.normal` in particular. This method can create a numpy array of random numbers.

A simple way to visualize these numbers is the `hist` method provided by the matplotlib sub-package pyplot.

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[4]:
```
def create_gaussian_histogram(N, mean=0., sigma=1.):
    # Create a numpy array with gaussian distributed numbers
    random_numbers = np.random.normal(loc=mean, scale=sigma, size=(N))
   # Visualize the content of the array as histogram with the help of matplotlib.
    plt.hist(random_numbers, bins=30)
    plt.show()

    # Return the numpy array
    return random_numbers
```

[5]:
```
gaussian_numbers = create_gaussian_histogram(N=100000)
```



ROOT Approach: If you want to use `ROOT`, you should check out `gRandom->Gaus()`. You can use this method to fill a `ROOT` `TH1F` histogram one by one with the random numbers.

Alternatively you can use the `FillRandom` method to fill the histogram directly with Gaussian distributed numbers, e.g. `my_hist.FillRandom("gaus", 1000)`. If you want to use a ROOT's gauss function with other values for mean and sigma than the default values, you have to define a one dimensional function `TF1` with the respective parameters first:

```
gaussian = TF1("gaussian","gaus",-3,3)
gaussian.SetParameters(1,0,1)  # last parameter is sigma, second to last is the mean of the ga
```

Plotting with ROOT in jupyter notebooks is a bit tricky, which is why we will give you some detailed hints on how to do this. First of all, the plot will not be shown if it is created in a function. Thus, your function should return the created `TH1F` object and you can then draw it on a canvas:

```
canvas = TCanvas("c1", "c1")

root_histogram = create_gaussian_histogram_with_root(N=1000)

root_histogram.Draw()
canvas.Modified()
canvas.Update()
canvas.Draw()
```

A second problem arises if you try to run the code a second time, because the created ROOT objects are still present in the notebook and jupyter will not be able to create them again unless you delete them first, e.g. with a code snippet such as:

```
try:
    del canvas
except NameError:
    pass
```

This will delete the ROOT object `canvas` if it was already created. If it has not been created, yet, the `try-except` approach will catch the `NameError` that will be thrown, as the object `canvas` does not exist, yet. In this case nothing will be done (look up what the python build-in keyword `pass` does). You can also restart the jupyter kernel to achieve this, but this is rather inconvenient.

[1]:
```
from ROOT import TH1F, gRandom, TFile, TCanvas, TF1, gROOT
from IPython.display import display, HTML
```

```
Welcome to JupyROOT 6.30/04
```

[9]:
```
def create_gaussian_histogram_with_root(N, mean=0., sigma=1.):
    # Create an histogram with 20 bins from -3 to 3
    root_histogram = TH1F("myHisto", "Histogram containing random numbers", 20, -
3, 3)

    # Initialize the random numbers generator
    gRandom.SetSeed(1234)
    gaussian = TF1("gaussian","gaus",-3,3)
    gaussian.SetParameters(N,mean,sigma)

    # Generate N random numbers following a gaussian distribution and fill the␣
  ↪histogram with them
    root_histogram.Fill(gaussian.GetBinContent(bin))

    return root_histogram
```

[10]:
```
try:
    del c1
except NameError:
    pass

try:
    del root_histogram
except NameError:
    pass

c1 = TCanvas("c1", "c1")

root_histogram = create_gaussian_histogram_with_root(N=100000)
```

```
root_histogram.Draw()
c1.Modified()
c1.Update()
c1.Draw()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[10], line 13
      9     pass
     11 c1 = TCanvas("c1", "c1")
---> 13 root_histogram = create_gaussian_histogram_with_root(N=100000)
     15 root_histogram.Draw()
     16 c1.Modified()

Cell In[9], line 11, in create_gaussian_histogram_with_root(N, mean, sigma)
      8 gaussian.SetParameters(N,mean,sigma)
     10 # Generate N random numbers following a gaussian distribution and fill␣
 ↪the histogram with them
---> 11 root_histogram.Fill(gaussian.GetBinContent(bin))
     13 return root_histogram

AttributeError: 'TF1' object has no attribute 'GetBinContent'
```

```
Warning in <TROOT::Append>: Replacing existing TH1: myHisto (Potential memory
leak).
```

2.2.2 Exercise 1.2

Extend your code from Exercise 1.1 so that the histogram data is written to a file.

This step is a bit more tricky if you are following the python approach, so please take a look at the hints. If you are using ROOT, you can write your `TH1F` histogram directly to a ROOT file.

Python Approach: Numpy does not provide a dedicated class for histograms as ROOT does. In Exercise 1.1 we created an array containing the random numbers and visualized these numbers as histogram.

You can use numpy's `numpy.histogram` method to interpret the data as histogram. However, this method will not create or return a "histogram" object, it will simply give you the bin counts and the bin edges of the histogram in form of two numpy arrays. You will have to decide yourself how to handle these return values.

Option 1) Store the data, not the histogram.

You can use `numpy.save` or `pandas.to_hdf` (you have to convert your data into the `pandas.DataFrame` or the `pandas.Series` format for this) to save the data you created directly. You can also write the data to a csv file or any other format you might be familiar with. Using this option, you have to recreate the histogram again, because you stored the raw data and not the histogram interpretation. This has the advantage of being able to reinterpret the data.

Option 2) Define a histogram object that can be stored to a file.

Storing the data in form of a histogram has two advantages: Firstly, it is clear how the data should be interpreted, and secondly, when handling large amounts of data, the histogram is a storage efficient way to save the data. Instead of saving each individual data point, only the bin count and the bin edges are stored. This means, if you consider a histogram with `n` bins, `2*n+1` (1 bin count per bin -> `n` bin counts; and `n+1` bin edge positions) numbers have to be stored. Keep in mind, however, that the amount of information stored in the histogram is also reduced compared to the full raw data set, as you only store one interpretation of the data.

You can also use Python's `pickle` to directly dump the tuple of bin counts and bin edges returned by the numpy histogram method to store these python objects in a pickle file. See also the example given on the python website which shows how a python dictionary is written and loaded again.

The downside of these methods is, that you have to put a bit more effort into defining the object that is stored.

You can try to implement both options, but focus on the **Option 2)**.

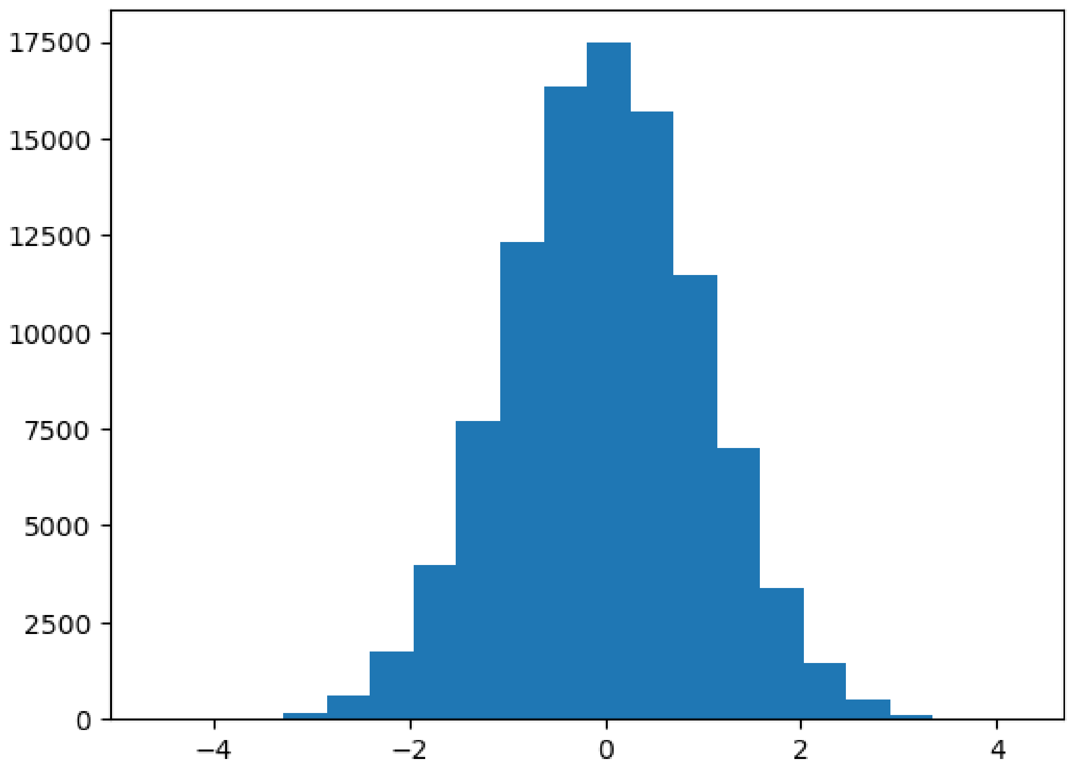
[9]:
```
import pickle

np.save("data", gaussian_numbers)

with open("hist.pickle", "wb") as outfile:
    pickle.dump(np.histogram(gaussian_numbers), outfile)
# Save the data...
```

Have a look at the custom histogram class `HistogramClass` below and try to understand how it works and which features it provides!

You can use this class or try your own method.

[10]:
```
# Define a histogram object
import pandas as pd

class HistogramClass:
    def __init__(self, data, bins=20, bin_range=None):
        if isinstance(data, tuple) and all(isinstance(e, pd.Series) for e in␣
 ↪data):
            self._bins = len(data[0].index)
            self._bin_counts = data[0].values
            self._bin_edges = data[1].values
            self._mean = data[2].values[0]
            self._std = data[2].values[1]
            self._entries = data[2].values[2]
            self._underflow = data[2].values[3]
            self._overflow = data[2].values[4]
        elif isinstance(data, np.ndarray) and isinstance(bins, int):
            self._bins = bins
           bin_counts, bin_edges = np.histogram(data, bins=bins, range=bin_range)
            self._bin_counts = bin_counts
            self._bin_edges = bin_edges

            if bin_range is not None:
                bounds = (data >= bin_range[0]) & (data <= bin_range[1])
            else:
                bounds = np.full(shape=data.shape, fill_value=True)
            self._mean = np.mean(data[bounds])
            self._std = np.std(data[bounds])
            self._entries = len(data[bounds])

            self._underflow = 0 if bin_range is None else len(data[data <␣
↪bin_range[0]])
            self._overflow = 0 if bin_range is None else len(data[data >␣
↪bin_range[1]])
        else:
            raise ValueError("The parameter 'data' must be a 1 dimensional numpy␣
↪array and the parameter 'bins' an integer!")

    @property
    def bins(self):
        return self._bins

    @property
    def bin_edges(self):
        return self._bin_edges

    @property
    def bin_mids(self):
        return (self._bin_edges[1:] + self._bin_edges[:-1]) / 2.

    @property
    def bin_counts(self):
        return self._bin_counts

    @property
    def mean(self):
        return self._mean

    @property
    def std(self):
        return self._std

    @property
    def entries(self):
        return self._entries

    @property
    def underflow(self):
        return self._underflow

    @property
    def overflow(self):
        return self._overflow

    def draw(self, *args, **kwargs):
        plt.hist(x=self.bin_mids, bins=self.bin_edges, weights=self.bin_counts,␣
↪*args, **kwargs)

    def save(self, file_path):
        with pd.HDFStore(path=file_path, mode="w") as hdf5store:
            hdf5store.append(key="bin_edges", value=pd.Series(self.bin_edges))
            hdf5store.append(key="bin_counts", value=pd.Series(self.bin_counts))
            meta_info = pd.Series([self.mean, self.std, self.entries, self.
↪underflow, self.overflow])
            hdf5store.append(key="meta_info", value=meta_info)

    @classmethod
    def load(cls, file_path):
        with pd.HDFStore(path=file_path, mode="r") as hdf5store:
            bin_edges = hdf5store.get(key="bin_edges")
            bin_counts = hdf5store.get(key="bin_counts")
            meta_info = hdf5store.get(key="meta_info")
        assert len(bin_counts.index) + 1 == len(bin_edges.index)

        instance = cls(data=(bin_counts, bin_edges, meta_info))
        return instance
```

[31]:
```
hist = HistogramClass(gaussian_numbers, bins=20)
hist.draw()
hist.save("hist.hist")
# TODO: Initialize a HistogramClass filled with your data
# TODO: Draw and save the histogram.
```



ROOT Approach: With ROOT you can just store the TH1F object to a ROOT file using the build-in methods.

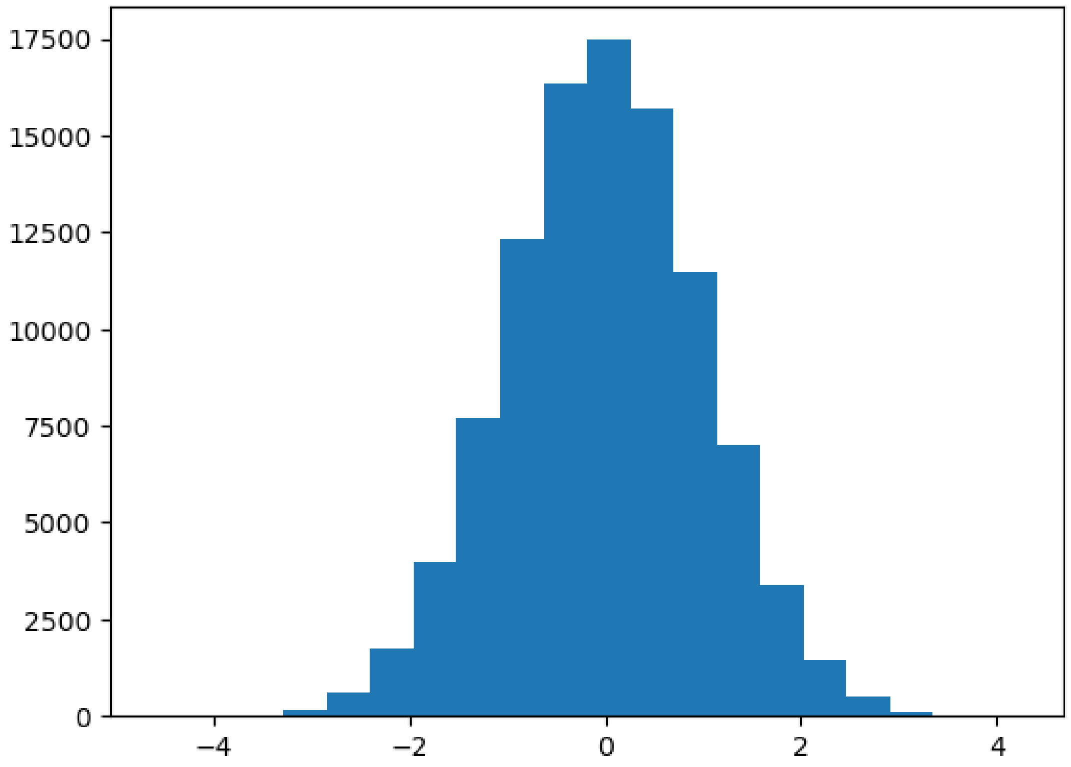
[32]:
```
root_file = TFile("root_histogram_with_gaussian_random_numbers.root",␣
 ↪"recreate")

# TODO: Write the ROOT histogram to the root_file
```

2.2.3 Exercise 1.3

Load the histogram you wrote to disk in Exercise 1.2 again and plot it.

Python Approach: Using the interface of the `HistogramClass` this is now easy. You can initialize a `HistogramClass` instance from a given file by using the class method `HistogramClass.load`.

[33]:
```
hist = HistogramClass.load('hist.hist')
hist.draw()
```



ROOT Approach: Create the `TFile` object for the file you saved earlier and get your ROOT `TH1F` histogram from it. Use a canvas as described in Exercise 1.1 to draw the loaded histogram.

[14]:
```
# TODO: Load and your ROOT histogram again and draw it to a new canvas
```

Compare the sizes of the different files, if you made the effort to implement more than one approach to store a histogram and/or the raw data You can use for instance `os.path.getsize` or `!ls -lh` to do this. Evaluate the file sizes for different amounts of gaussian random numbers `N`.

[15]:
```
import os

# TODO: Try any of the methods to check the file sizes of your histograms or raw␣
 ↪data,
#       if you saved them in different formats
```

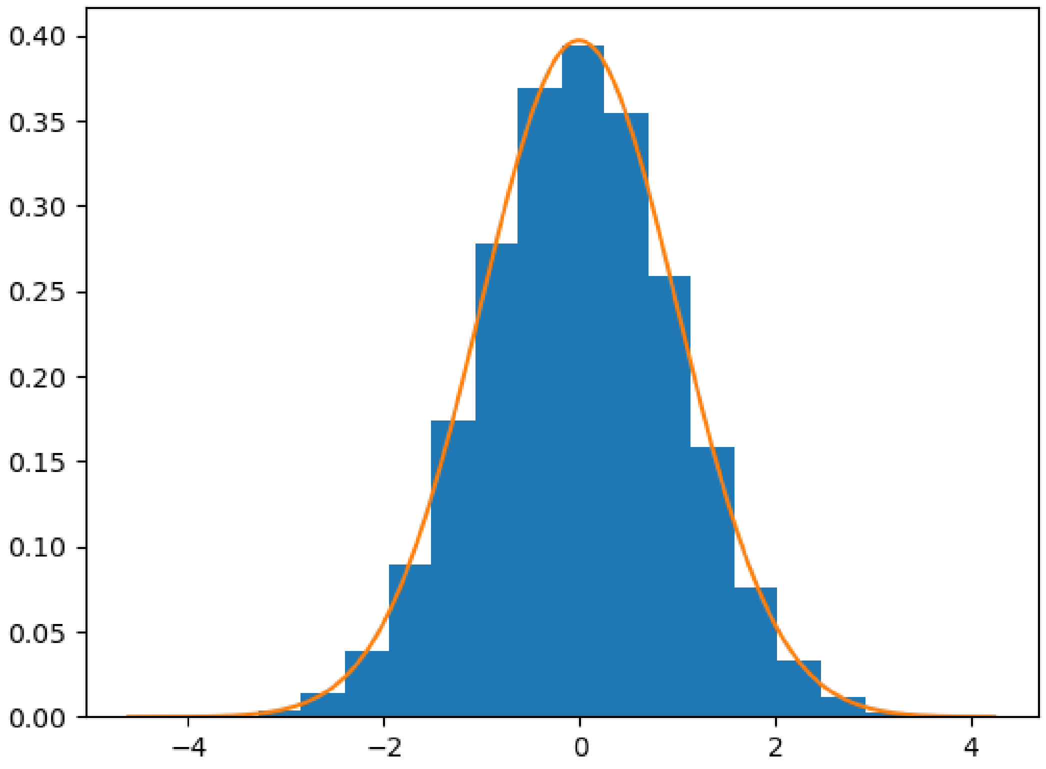
2.2.4 Exercise 1.4

Fit a Gaussian function to the histogram(s) you created in the previous exercises.

Python Approach: To perform a fit to your numpy histogram, you can use the fitting tools provided in the `scipy.optimize` package, for instance the `curve_fit` method. You can find an example on how to use it here.

You will have to define the function describing the gaussian distribution you want to fit to the histogram. To plot this function you can use the `matplotlib.pyplot.plot` plot function.

```
[34]: from scipy.optimize import curve_fit
      import scipy.stats as stats

      # TODO: Define your Gaussian fit function
      #stats.norm.pdf(x, mu, sigma)

      # TODO: Implement the fit
      centers = (hist.bin_edges[:-1] + hist.bin_edges[1:])/2
      binwidth = hist.bin_edges[1] - hist.bin_edges[0]
      (mu, sigma), ((Dmu,_),(_,Dsigma)) = curve_fit(stats.norm.pdf, centers, hist.
       ↪bin_counts/np.sum(hist.bin_counts)/binwidth, p0=(0,1))

      # Plot the histogram and the fitted function.
      hist.draw(density=True)
      x = np.linspace(np.min(hist.bin_edges), np.max(hist.bin_edges), 200)
      plt.plot(x, stats.norm.pdf(x, mu, sigma))
      plt.show()
```



ROOT Approach: Define a ROOT `TF1` `gaus` function to be fitted to the ROOT histogram. To perform the fit, use the predefined `Fit` method of the ROOT histogram.

Draw the histogram again onto a new canvas using the same approach as above to visualize the result of the fit.

[17]:
```
# Define a gaussian function
# TODO: Define the TF1 gaussian function to be fitted to the histogram

# Fit the histogram with this function
# TODO: Perform the fit of the gaussian to the ROOT root_histogram

c3 = TCanvas("c3", "c3")
# TODO: Draw the histogram to the canvas c3 to show the fitted function and the␣
 ↪histogram itself
```

2.2.5 Exercise 1.5

Make the plot nicer and save it as vector graphic, e.g. eps or pdf. The latter can be displayed within jupyter lab by clicking on it in your file browser on the left.

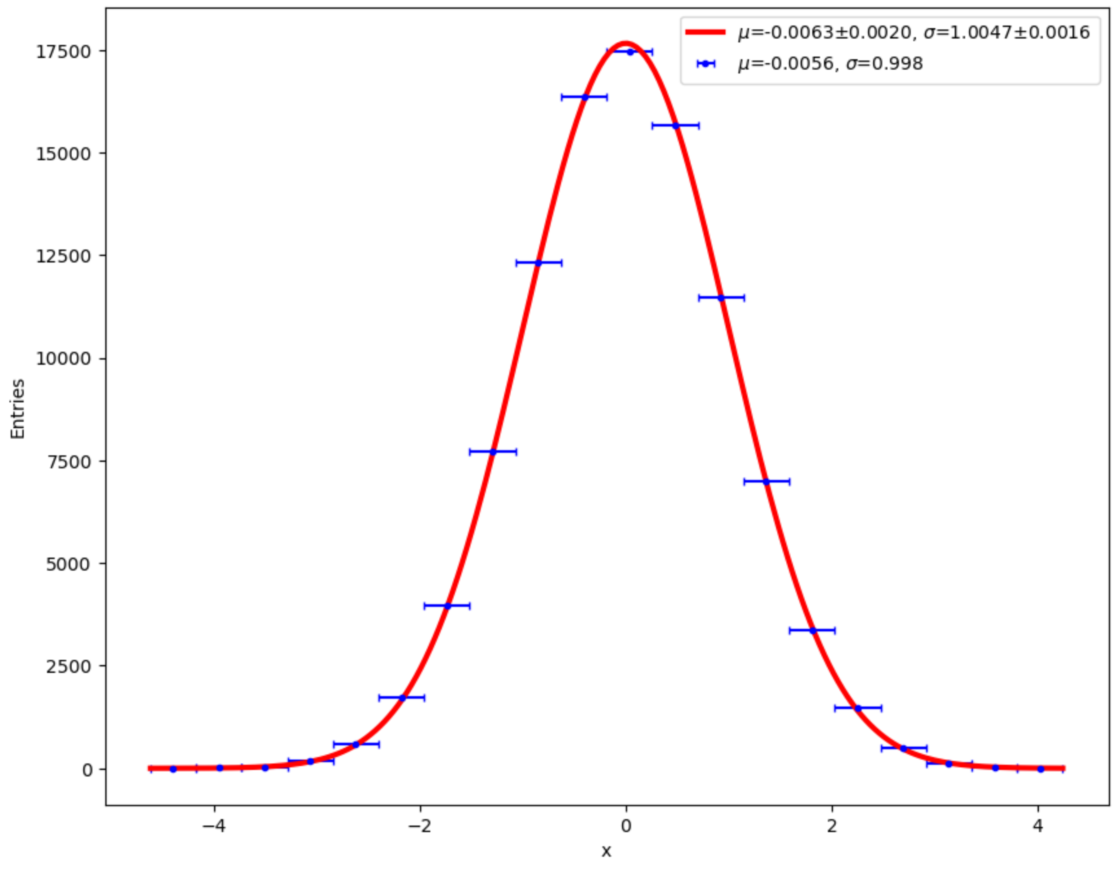
The plot should: - use blue filled boxes for the histogram with horizontal error bars to indicate the bin width - show the fitted gaussian function as red line with a thickness/width of 3 - label the `x` and `y` axes with "x" and "Entries", respectively - display the mean and standard deviation of the histogram in the legend - display the fitted parameters with uncertainties as well as the fit probability in the legend.

Python Approach Matplotlib provides a lot of information about the available plot style options in the documentation of the respective plot functions. You will also find a lot of matplotlib examples when googling for certain key words. To get change the style of your plot more drastically, you might have to change the plot function you are using. Have a look at `matplotlib.pyplot.errorbar` instead of `matplotlib.pyplot.hist`, for instance.

Obtaining the fit probability in python is not as simple as with ROOT, so you can skip it.

[35]:
```
fig = plt.figure(figsize=(10,8))
from scipy.optimize import curve_fit
import scipy.stats as stats

# TODO: Define your Gaussian fit function
#stats.norm.pdf(x, mu, sigma)
plt.errorbar(hist.bin_mids,
             hist.bin_counts,
             color="blue",
             fmt=".",
             xerr=binwidth/2,
             capsize=2,
             label=f"$\mu$={hist.mean:.4f}, $\sigma$={hist.std:.3f}")
#hist.draw(rwidth=0.5)
```

```
x = np.linspace(np.min(hist.bin_edges), np.max(hist.bin_edges), 200)
plt.plot(x,
         stats.norm.pdf(x, 0, 1)*np.sum(hist.bin_counts)*binwidth,
         linewidth=3,
         color="red",
         label=f"$\mu$={mu:.4f}$\pm${np.sqrt(Dmu):.4f}, $\sigma$={sigma:.
 ↪4f}$\pm${np.sqrt(Dsigma):.4f}")

plt.legend()

plt.ylabel("Entries")
plt.xlabel("x")
plt.savefig("plt.pdf", format="pdf", bbox_inches="tight")
plt.show()
```



```
[19]: from IPython.display import IFrame
      IFrame("plt.pdf", width=800, height=600)
```

```
[19]: <IPython.lib.display.IFrame at 0x7efc91a147c0>
```

ROOT Approach To improve the histogram plot, you can for instance have a look at the options described in the overview of ROOT's `TStyle` Class

```
[20]: from ROOT import TH1F, TFile, TF1, gStyle

      # TODO: Change the properties of gStyle, the ROOT histogram and the gaussian fit␣
       ↪function
      #       to improve the style of the plot

      c4 = TCanvas("c4", "c4")

      # TODO: Draw the histogram to the canvas c4 and save it as vector graphic (pdf␣
       ↪files can be viewed with JupyterLab)
```

2.2.6 Exercise 1.6 (Obligatory)

Fill a histogram with the quotient $f(x_1, x_2) = x_1/x_2$ of two Gaussian distributed random numbers x_1 and x_2 with the mean $m_1 = 2$ and standard deviation $\sigma_1 = 1.5$ and $m_2 = 3$, $\sigma_2 = 2.2$, respectively.

Assuming standard error propagation without correlations

$$\sigma_f^2 = \sum_i \left(\frac{\partial f}{\partial x_i}\right)^2 \sigma_i^2$$

calculate the propagated uncertainty for this function $f(x_1, x_2)$ (using the mean values of x_1 and x_2).

How does the result compare with the properties of the created histogram?

2.2.7 Theoretical Calculation

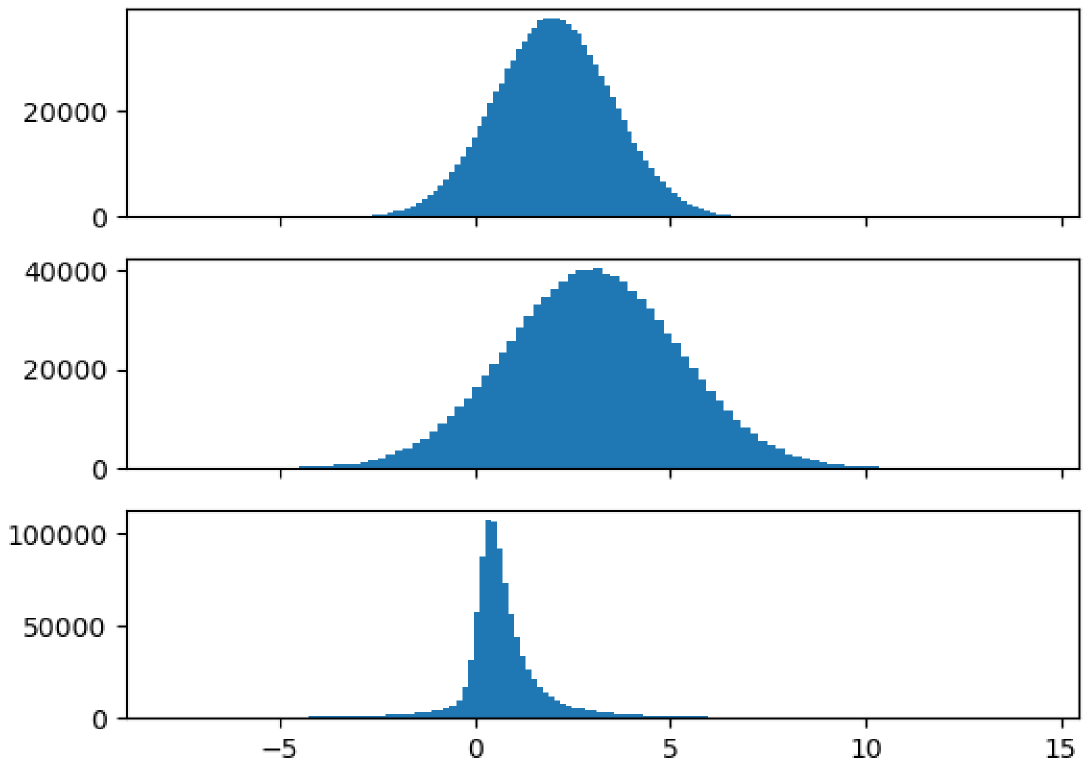
$$\sigma_f^2 = \left(\frac{\partial f}{\partial x_1}\right)^2 \sigma_1^2 + \left(\frac{\partial f}{\partial x_2}\right)^2 \sigma_2^2 = \left(\frac{1}{m_2}\right)^2 \sigma_1^2 + \left(-\frac{m_1}{m_2^2}\right)^2 \sigma_2^2 = \left(\frac{1.5}{3}\right)^2 + \left(\frac{2\cdot 2.2}{3^2}\right)^2 = 3961/8100 \approx 0.4890$$

Python Approach Use the methods you learned in the previous exercises to create three numpy arrays containing the random numbers with the properties of x_1, x_2 and $f(x_1, x_2)$ where the last is simply the quotient of the first two arrays. Plot and evaluate the histograms of these random numbers.

```
[36]: def create_histograms(N, mean1=2., sigma1=1.5, mean2=3., sigma2=2.2, bins=100):
          # Create numpy arrays with gaussian distributed numbers for x_1 and x_2
          # TODO: Create the arrays gauss1 and gauss2 with the random numbers
          gauss1 = np.random.normal(loc=mean1, scale=sigma1, size=(N))
          gauss2 = np.random.normal(loc=mean2, scale=sigma2, size=(N))

          # Calculate the array containing the quotient of x_1 and x_2
```

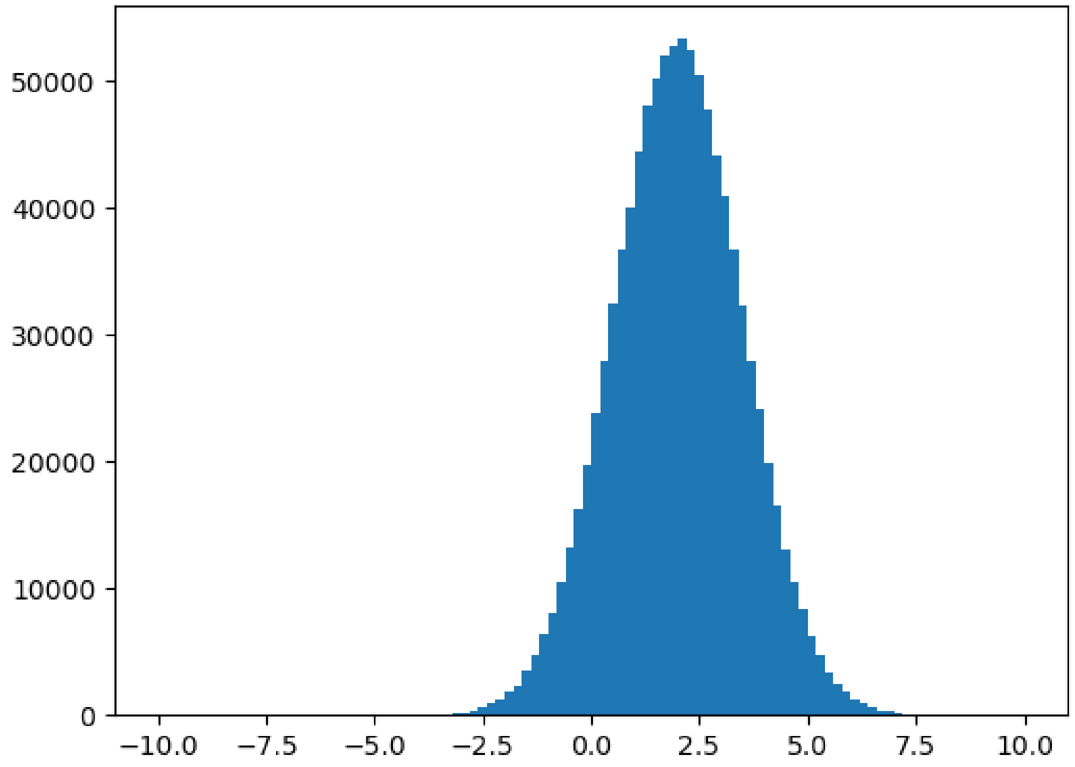
```
    f = gauss1/gauss2

    fig, ax = plt.subplots(nrows=3, ncols=1, sharex=True)
    print
    ax[0].hist(gauss1, bins=bins)
    ax[1].hist(gauss2, bins=bins)
    ax[2].hist(f, bins=bins, range=(-5,10))

    plt.show()
    # Return the numpy arrays
    return gauss1, gauss2, f
```

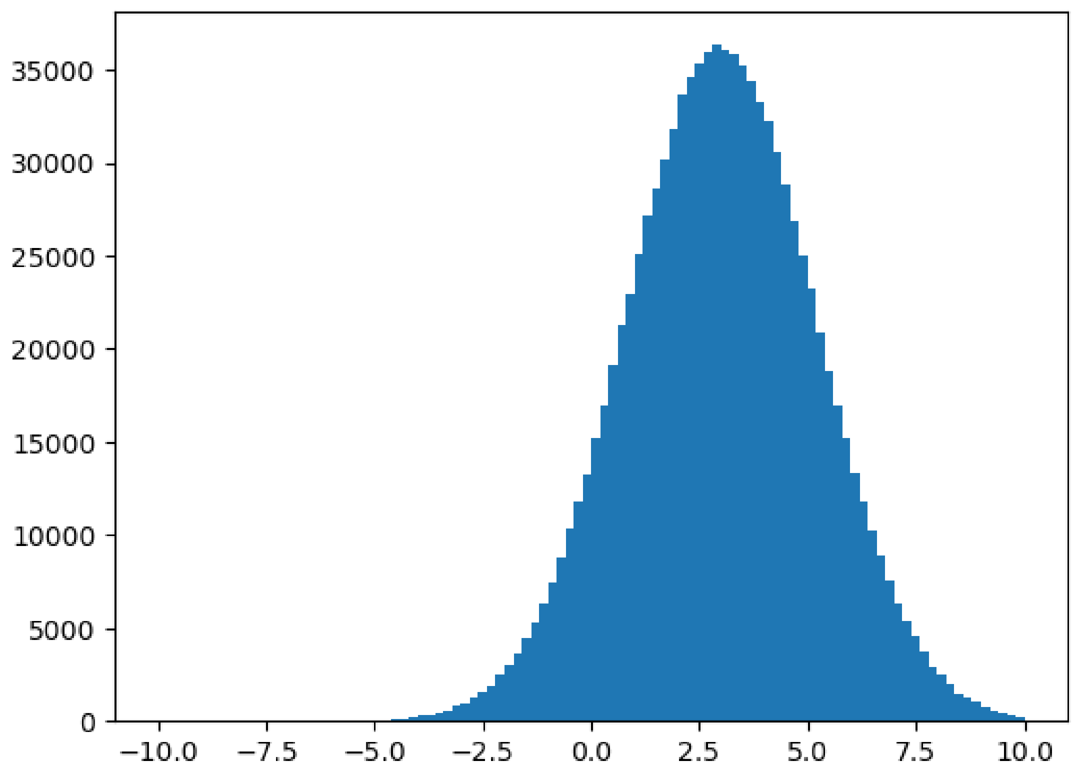
[37]:
```
x1, x2, quotient = create_histograms(N=1000000)
```

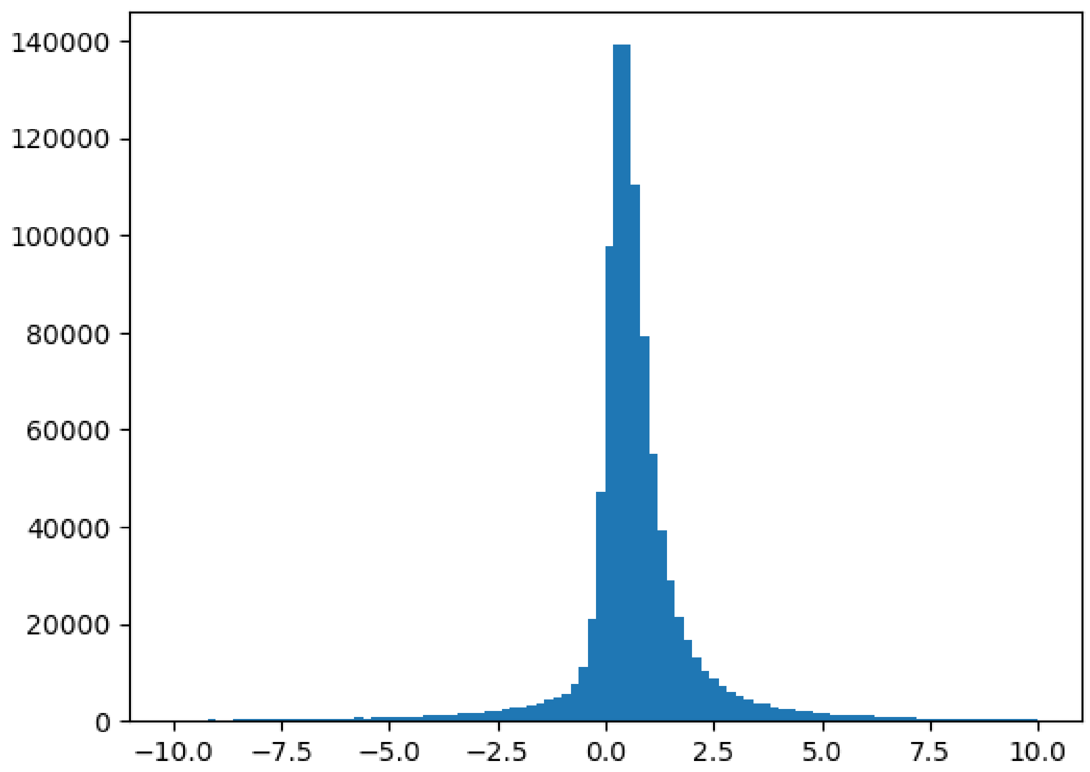


[23]:
```
# TODO: Create three HistogramClass instances from the three arrays you created.
#       Use 100 bins and a range from -10 to 10.
hist1 = HistogramClass(x1, bins=100, bin_range=(-10,10))
hist1.draw()
plt.show()
hist2 = HistogramClass(x2, bins=100, bin_range=(-10,10))
hist2.draw()
plt.show()
```

```
histf = HistogramClass(quotient, bins=100, bin_range=(-10,10))
histf.draw()
plt.show()

print(f"hist1: mean={hist1.mean:.4f}(Theo: 2), std={hist1.std:.4f}(Theo: 1.5)")
print(f"hist1: mean={hist2.mean:.4f}(Theo: 3), std={hist2.std:.4f}(Theo: 2.2)")
print(f"hist1: mean={histf.mean:.4f}(Theo: 0.6666), std={histf.std:.4f}(Theo: 0.
 ↪4890)")


# TODO: Plot the histograms and determine their mean and standard deviation to be
#        able to compare them with the original values and the theoretical␣
 ↪calculation.
```




35000
30000
25000
20000
15000
10000
5000
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


140000
120000
100000
80000
60000
40000
20000
0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

```
hist1: mean=2.0021(Theo: 2), std=1.4986(Theo: 1.5)
hist1: mean=2.9957(Theo: 3), std=2.1904(Theo: 2.2)
hist1: mean=0.6628(Theo: 0.6666), std=1.6497(Theo: 0.4890)
```

Root Approach Similar to the methods used in Exercise 1.1, use the ROOT method `gRandom.Gaus` to generate the random numbers x_1 and x_2 and fill `TH1F` histograms with them. Fill also a histogram with the quotient `f = x_1/x_2`. Draw and evaluate the three histograms with the methods you learned in the previous exercises.

[24]:
```
def create_root_histograms(N, bins=100, bin_range=(-10., 10.)):
    # Create histogram for the distribution of x, y and f(x,y) = x/y with 100␣
 ↪bins from -10 to 10
    min_bin, max_bin = bin_range
    # Initialize three TH1F ROOT histograms for x_1, x_2 and f

    # Initialize the random numbers generator
    gRandom.SetSeed()

    # Generate 2 x N random numbers following a gaussian distribution
    # one with mean = 2 and sigma = 1.5 and
    # one with mean = 3 and sigma = 2.2 and

    for i in range(N):
        pass # TODO: Fill the three histograms with the TH1F.Fill method

    # TODO: Return the three ROOT histograms
```

[25]:
```
x1_hist, x2_hist, f_hist = create_root_histograms(N=100000)

# Draw the three histograms onto the canvases c5, c6 and c7

c5 = TCanvas("c5", "c5")

c6 = TCanvas("c6", "c6")

c7 = TCanvas("c7", "c7")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 1
----> 1 x1_hist, x2_hist, f_hist = create_root_histograms(N=100000)
      3 # Draw the three histograms onto the canvases c5, c6 and c7
      5 c5 = TCanvas("c5", "c5")
```

```
TypeError: cannot unpack non-iterable NoneType object
```

Write down what you observe when you compare the result of the theoretical calculation with what you obtained using the random numbers.

Die Standardabweichung für den Quotienten der gaussverteilten Zufallszahlen entspricht nicht dem nach gausscher Fehlerfortpflanzung errechnetem Wert. Dies liegt daran, dass die gaussche Fehlerfortpflanzung in dieser Art nur für lineare berechnungen funktioniert. Da hier durch eine der größen Geteilt wird liefert die Fehlerfortpflanzung falsche ergebnisse.

```
[ ]:
```