Experimental Physics (II)
Report

Fundamental Python
Basic Usage of Python

Group 2

洪　瑜 B125090009

黃巧涵 B122030003

洪懌平 B102030019

2025/03/25

Abstract

This experiment explores the application of Python in data analysis, emphasizing statistical concepts and practical programming skills. Utilizing core libraries such as NumPy, Matplotlib, Pandas, and SciPy within the Anaconda environment, the team conducted four exercises: (1) analyzing normally distributed data, (2) filtering data based on standard deviation thresholds, (3) fitting noisy linear data using curve fitting techniques, and (4) visualizing data from CSV files. Through these tasks, we examined the law of large numbers, evaluated data consistency with theoretical expectations, and practiced data visualization and model fitting. The experiment highlights Python's effectiveness in processing and interpreting experimental data and reinforces key statistical principles relevant to physical measurement and error analysis.

1 Introduction

1.1 Application of Python Principles in This Experiment

1.1.1 NumPy（Numerical Python）

NumPy is a fundamental and core library for scientific computing in Python. It has several important features:

1. Provides a highly efficient multi-dimensional array mathematical library.
2. Offers convenient and useful capabilities for linear algebra and Fourier transform.
3. Replaces Python lists with NumPy arrays for better performance.
4. Supports vectorized operations, improving computational performance by avoiding the use of "for" loops.

1.1.2 Matplotlib

Matplotlib is a powerful open-source toolkit used for representing and visualizing data. It allows users to define various output functionalities, where the front-end code (plotting code) is passed to the back-end to handle the rendering, creating visually appealing or complex graphics.

1.1.3 Pandas

By combining the features of NumPy with the data manipulation capabilities of spreadsheets and relational databases (SQL), you can perform operations such as restructuring, slicing, aggregating, and selecting subsets of data. With the use of the pandas library, data analysis becomes much easier compared to using pure Python alone.

1.1.4 SciPy

SciPy is a Python library for algorithms and mathematical tools. It includes modules for optimization, linear algebra, integration, interpolation, special functions, fast Fourier transform, signal and image processing, ODE solving, and other computations commonly used in science and engineering.

1.1.5 Anaconda

Anaconda is a Python platform designed for data science and machine learning developers, integrating the most commonly used data analysis tools and libraries. It comes with a rich collection of data science packages, including NumPy, Pandas, and Seaborn, allowing users to skip the complex installation process and start working immediately.

1.2 Mathematical Concepts in This Experiment

In this experiment, we analyze data using Python, focusing on the properties of the normal distribution, the calculation of standard deviation, and data filtering and visualization. Mastering these methods can help us process and interpret measurement results more effectively. The following are the mathematical principles applied in this experiment:

1.2.1 Normal Distribution

The normal distribution is characterized by most data being concentrated around a certain mean, while data points that deviate from the mean appear less frequently. This distribution forms a bell-shaped curve (as shown in Fig. 1), known as the Gaussian distribution.

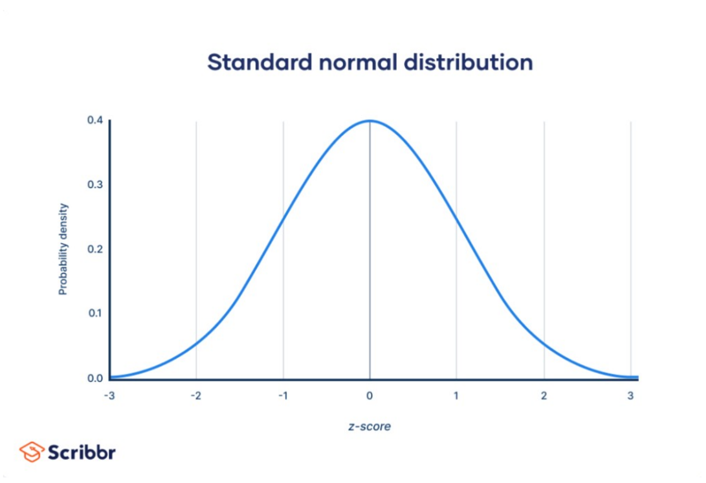


Figure 1: The bell-shaped curve of the normal distribution

The probability density function (PDF) of the normal distribution is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{1}$$

The symbol μ represents the mean, indicating the center of the data, while σ is the standard deviation, describing the dispersion of the data. In the analysis of experimental data, understanding the normal distribution allows us to assess measurement errors, filter out outliers, and further build data models.

1.2.2 Mean

The mean μ represents the central position of a dataset and serves as a typical value of the measurement results. When the same physical quantity is measured multiple times, the results may vary slightly each time, but they generally fluctuate around a certain value, which is the mean. The mean can be calculated by dividing the sum of all data points by the total number of data points:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{2}$$

x_i represents the ith measured value, and N is the number of measurements. The mean helps us determine a representative value for a dataset, and in most cases, it can reflect the true value of the physical quantity.

1.2.3 Standard Deviation

The standard deviation σ is used to measure the spread of data, indicating the degree of dispersion.

It is defined as:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} (x_i - \mu)^2} \tag{3}$$

1.2.4 Standard Deviation Interval

In statistics, the so-called ”68-95-99.7 rule” describes the application of standard deviation in data filtering:

1. About 68% of the data falls within ± 1 standard deviation of the mean.
2. About 95% of the data falls within ± 2 standard deviations of the mean.
3. About 99.7% of the data falls within ± 3 standard deviations of the mean.

Therefore, if a data point significantly exceeds ± 3 standard deviations from the mean, it is highly likely to be an outlier.

1.2.5 Data Filtering

In real-world physics experiments, the measurement process may be affected by environmental factors, equipment issues, or other sources of error, causing some data points to deviate from the normal range. To ensure accuracy in data analysis, data can be filtered based on standard deviation intervals.

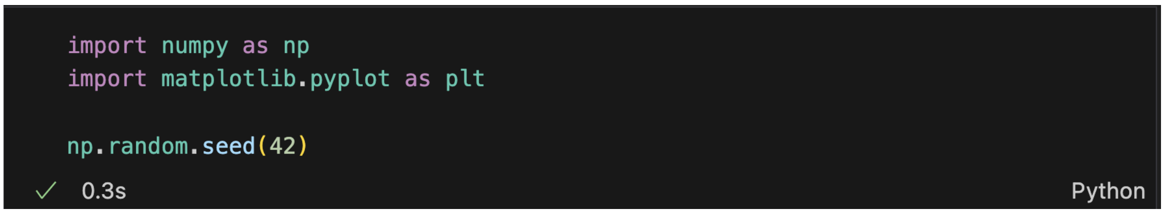
In this experiment, we use Python to generate random data and determine whether each data point falls within ± 1 standard deviation of the mean, filtering out any anomalous values. This filtering method is widely applied in various scientific studies. For example, in astronomical observations, scientists need to remove anomalous data caused by atmospheric interference or other external factors.

1.2.6 Data Visualization

We make the data more visually intuitive by creating charts, allowing us to observe the distribution of the data more clearly and identify trends or outliers. In this experiment, we use Matplotlib to plot a histogram to display the distribution of the data.

2 Experimental steps

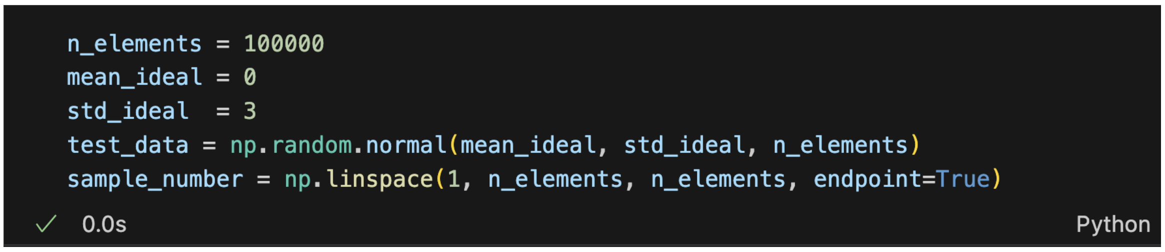
2.1 Practice 1



```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
```
✓ 0.3s Python

Figure 2: Import `numpy` and `matplotlib.pyplot` packages for better vectorized calculation and making plots. `np.random.seed(42)` is to keep the reproducibility.

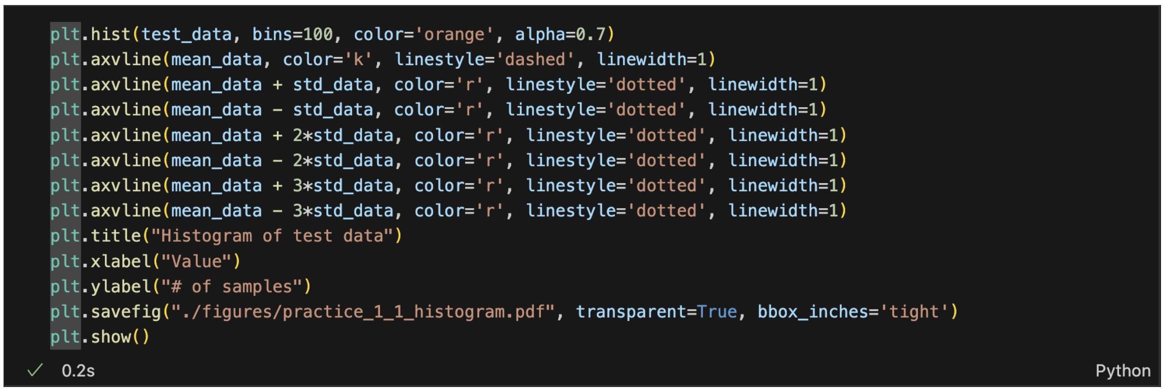
1. Generate a 1-d array with several elements, each element is sampled from a statistical population with a normal distribution with mean = 0 and standard deviation = 3.



```python
n_elements = 100000
mean_ideal = 0
std_ideal  = 3
test_data = np.random.normal(mean_ideal, std_ideal, n_elements)
sample_number = np.linspace(1, n_elements, n_elements, endpoint=True)
```
✓ 0.0s Python

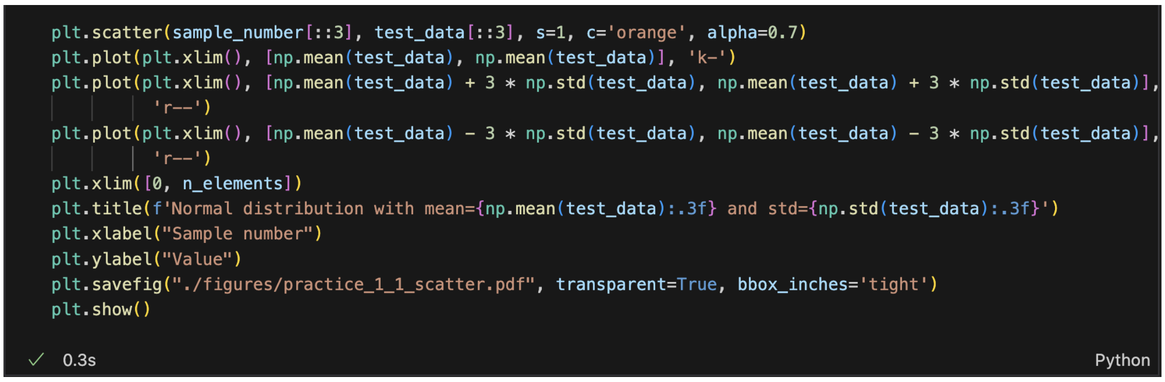
Figure 3: Initialize our samples (normal distribution of 10000 samples with mean = 0 and standard deviation = 3).

2. Plot the data points and mark the mean and standard deviation calculated from the data.



```python
plt.hist(test_data, bins=100, color='orange', alpha=0.7)
plt.axvline(mean_data, color='k', linestyle='dashed', linewidth=1)
plt.axvline(mean_data + std_data, color='r', linestyle='dotted', linewidth=1)
plt.axvline(mean_data - std_data, color='r', linestyle='dotted', linewidth=1)
plt.axvline(mean_data + 2*std_data, color='r', linestyle='dotted', linewidth=1)
plt.axvline(mean_data - 2*std_data, color='r', linestyle='dotted', linewidth=1)
plt.axvline(mean_data + 3*std_data, color='r', linestyle='dotted', linewidth=1)
plt.axvline(mean_data - 3*std_data, color='r', linestyle='dotted', linewidth=1)
plt.title("Histogram of test data")
plt.xlabel("Value")
plt.ylabel("# of samples")
plt.savefig("./figures/practice_1_1_histogram.pdf", transparent=True, bbox_inches='tight')
plt.show()
```
✓ 0.2s Python

Figure 4: Code for making histogram of our samples

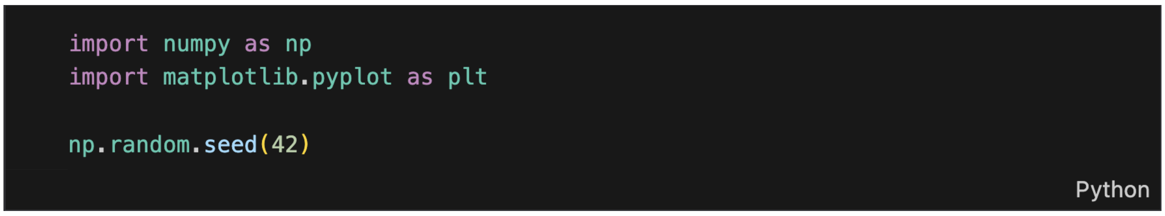


```
plt.scatter(sample_number[::3], test_data[::3], s=1, c='orange', alpha=0.7)
plt.plot(plt.xlim(), [np.mean(test_data), np.mean(test_data)], 'k-')
plt.plot(plt.xlim(), [np.mean(test_data) + 3 * np.std(test_data), np.mean(test_data) + 3 * np.std(test_data)],
            'r--')
plt.plot(plt.xlim(), [np.mean(test_data) - 3 * np.std(test_data), np.mean(test_data) - 3 * np.std(test_data)],
            'r--')
plt.xlim([0, n_elements])
plt.title(f'Normal distribution with mean={np.mean(test_data):.3f} and std={np.std(test_data):.3f}')
plt.xlabel("Sample number")
plt.ylabel("Value")
plt.savefig("./figures/practice_1_1_scatter.pdf", transparent=True, bbox_inches='tight')
plt.show()
```

Figure 5: Code for making scattered plot of our samples

3. Will the mean and standard deviation be the same as the population?

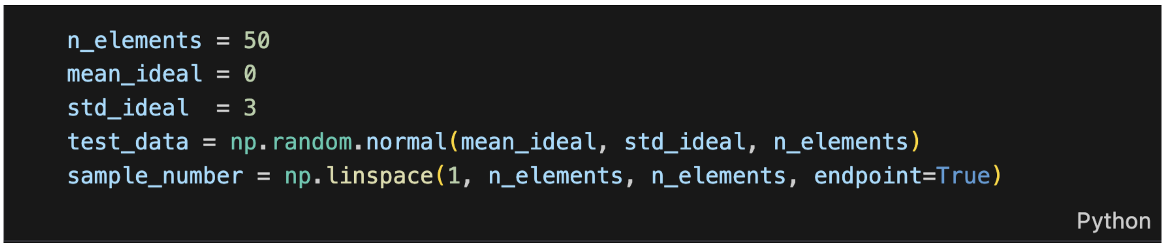
2.2 Practice 2



```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
```

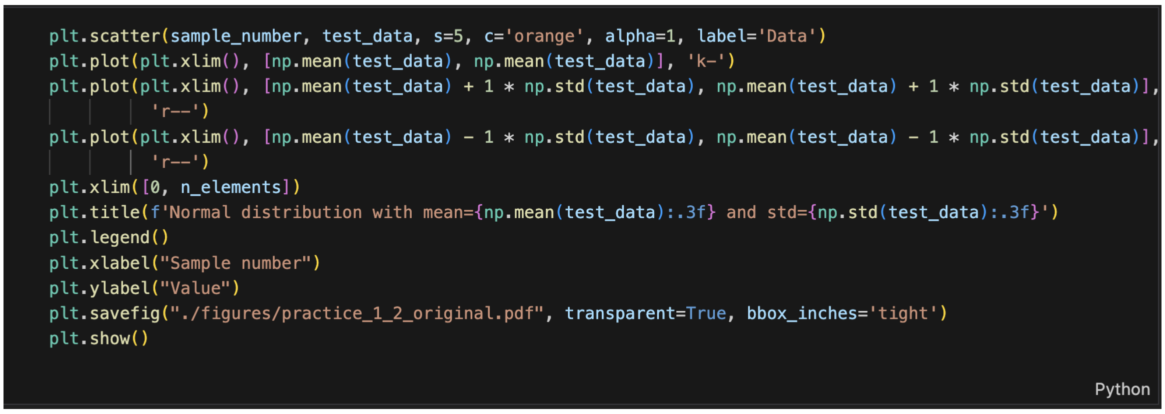
Figure 6: Import `numpy` and `matplotlib.pyplot` packages for better vectorized calculation and making plots

1. Generate a 1-d array with 50 elements, each element is sampled from a statistical population with a normal distribution with mean = 0 and standard deviation = 3.



```
n_elements = 50
mean_ideal = 0
std_ideal  = 3
test_data = np.random.normal(mean_ideal, std_ideal, n_elements)
sample_number = np.linspace(1, n_elements, n_elements, endpoint=True)
```

Figure 7: Initialize our samples (normal distribution of 50 samples with mean = 0 and standard deviation = 3).

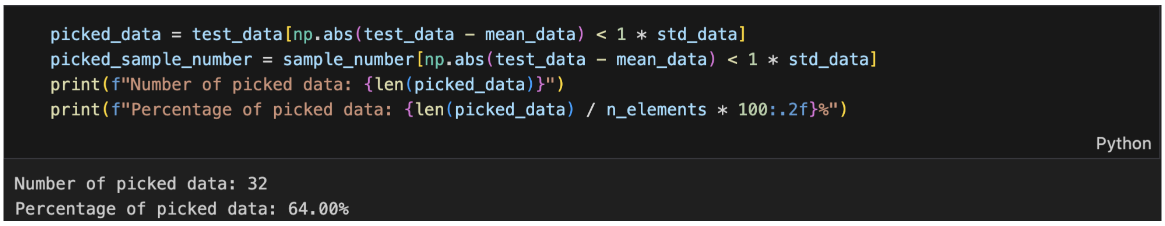


```python
plt.scatter(sample_number, test_data, s=5, c='orange', alpha=1, label='Data')
plt.plot(plt.xlim(), [np.mean(test_data), np.mean(test_data)], 'k-')
plt.plot(plt.xlim(), [np.mean(test_data) + 1 * np.std(test_data), np.mean(test_data) + 1 * np.std(test_data)],
            'r--')
plt.plot(plt.xlim(), [np.mean(test_data) - 1 * np.std(test_data), np.mean(test_data) - 1 * np.std(test_data)],
            'r--')
plt.xlim([0, n_elements])
plt.title(f'Normal distribution with mean={np.mean(test_data):.3f} and std={np.std(test_data):.3f}')
plt.legend()
plt.xlabel("Sample number")
plt.ylabel("Value")
plt.savefig("./figures/practice_1_2_original.pdf", transparent=True, bbox_inches='tight')
plt.show()
```

Python

Figure 8: Code for making scattered plot of our samples

2. Pick up the data points that are within 1 standard deviation from the mean.



```python
picked_data = test_data[np.abs(test_data - mean_data) < 1 * std_data]
picked_sample_number = sample_number[np.abs(test_data - mean_data) < 1 * std_data]
print(f"Number of picked data: {len(picked_data)}")
print(f"Percentage of picked data: {len(picked_data) / n_elements * 100:.2f}%")
```

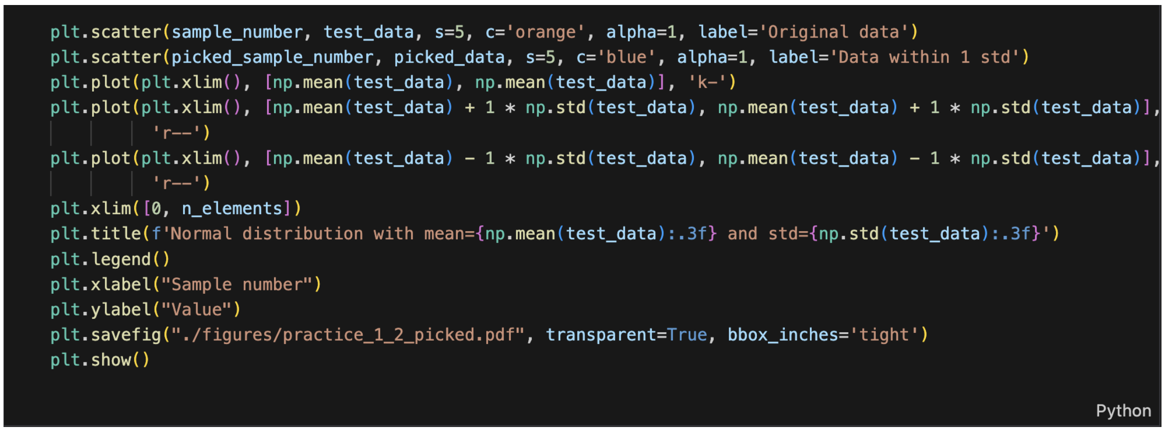
Python

```
Number of picked data: 32
Percentage of picked data: 64.00%
```

Figure 9: Total number of picked data is 32 and the percentage of picked data is 64%

3. Plot the picked data points and original data points to check if the filtering process is correct.

4. Mark the *mean* − *stdev* and *mean* + *stdev* on the plot might be helpful.



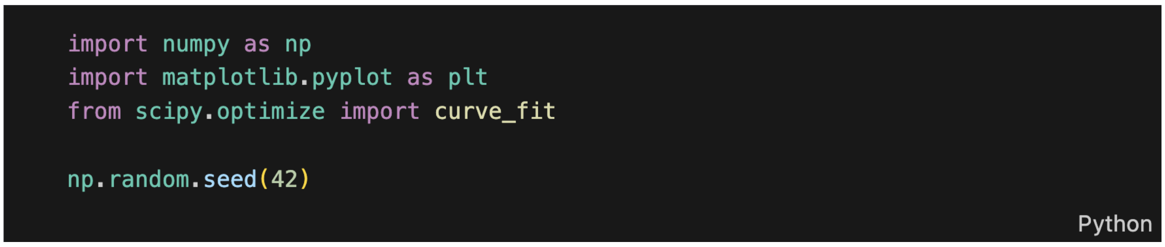
```python
plt.scatter(sample_number, test_data, s=5, c='orange', alpha=1, label='Original data')
plt.scatter(picked_sample_number, picked_data, s=5, c='blue', alpha=1, label='Data within 1 std')
plt.plot(plt.xlim(), [np.mean(test_data), np.mean(test_data)], 'k-')
plt.plot(plt.xlim(), [np.mean(test_data) + 1 * np.std(test_data), np.mean(test_data) + 1 * np.std(test_data)],
            'r--')
plt.plot(plt.xlim(), [np.mean(test_data) - 1 * np.std(test_data), np.mean(test_data) - 1 * np.std(test_data)],
            'r--')
plt.xlim([0, n_elements])
plt.title(f'Normal distribution with mean={np.mean(test_data):.3f} and std={np.std(test_data):.3f}')
plt.legend()
plt.xlabel("Sample number")
plt.ylabel("Value")
plt.savefig("./figures/practice_1_2_picked.pdf", transparent=True, bbox_inches='tight')
plt.show()
```

Python

Figure 10: Code for making scattered plot of our samples

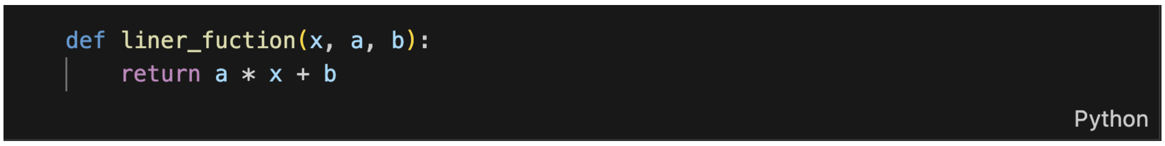
5. Print all the indices of the data points that are picked (These indices might range from 0 to 49).

2.3 Practice 3



```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

np.random.seed(42)
```

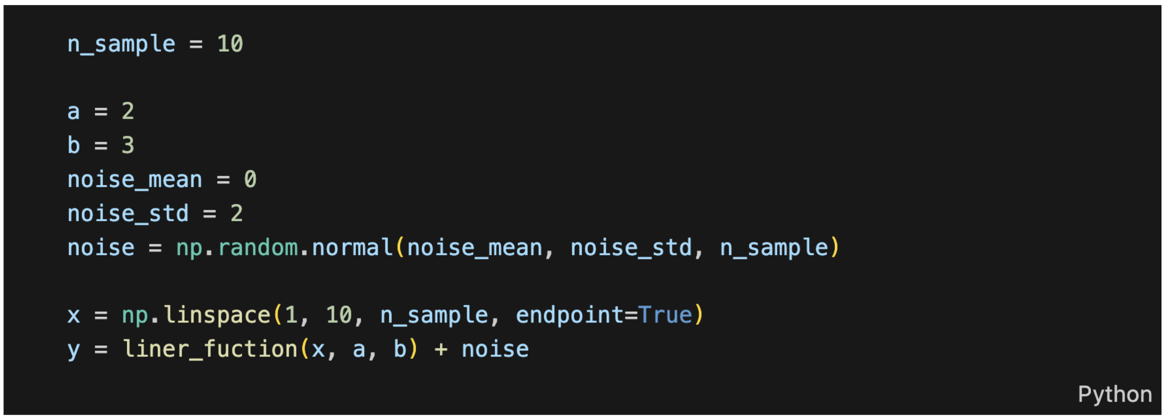
Figure 11: Import `numpy` and `matplotlib.pyplot` packages for better vectorized calculation and making plots. Also, from `scipy.optimize` package to import `curve_fit` function for fitting



```python
def liner_fuction(x, a, b):
    return a * x + b
```

Figure 12: Define a liner function $y = a * x + b$.

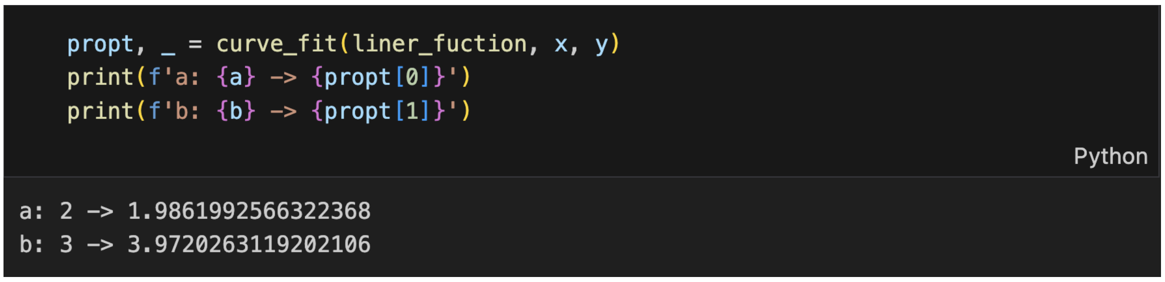
1. Generate a set of data points defined by a function $y = a * x + b + noise$ with `x = np.linspace(1, 10, 10)`.
2. The noise is sampled from a normal distribution with mean = 0 and standard deviation $= \sigma_0$. (define σ_0 whatever you like)



```python
n_sample = 10

a = 2
b = 3
noise_mean = 0
noise_std = 2
noise = np.random.normal(noise_mean, noise_std, n_sample)

x = np.linspace(1, 10, n_sample, endpoint=True)
y = liner_fuction(x, a, b) + noise
```

Figure 13: Initialize x (`np.linspace(1, 10, 10)`) and $y = a * x + b + noise$ where the noise is the normal distribution with mean =0 and standard deviation = 2

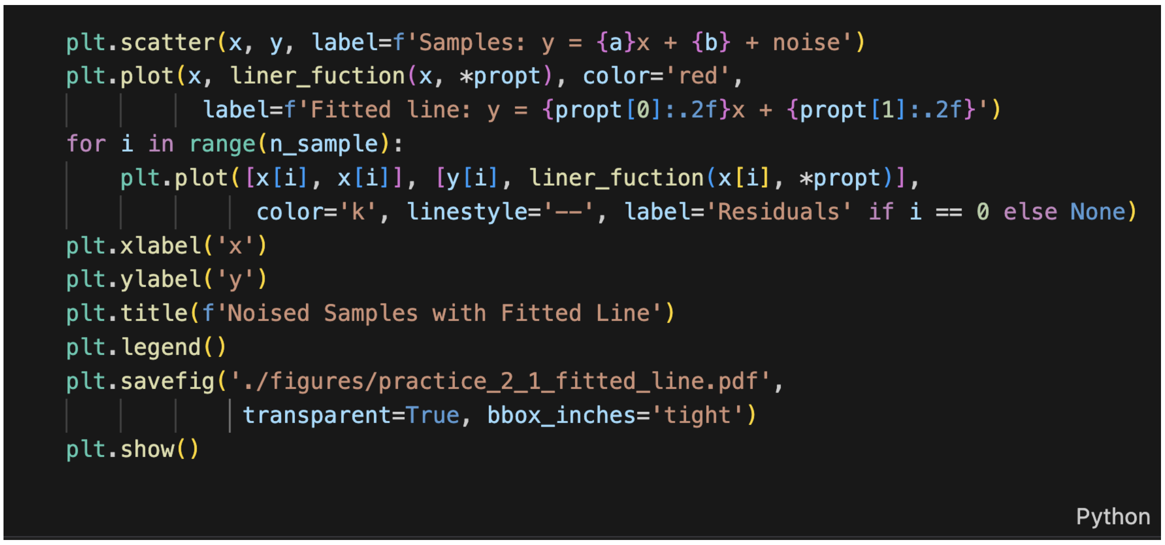
3. Use the `curve_fit` function in `scipy.optimize` to fit the data points with the function $y = a * x + b$.
4. Plot the data points and the fitting curve.
5. Calculate the residuals and plot the residuals.



```python
propt, _ = curve_fit(liner_fuction, x, y)
print(f'a: {a} -> {propt[0]}')
print(f'b: {b} -> {propt[1]}')
```

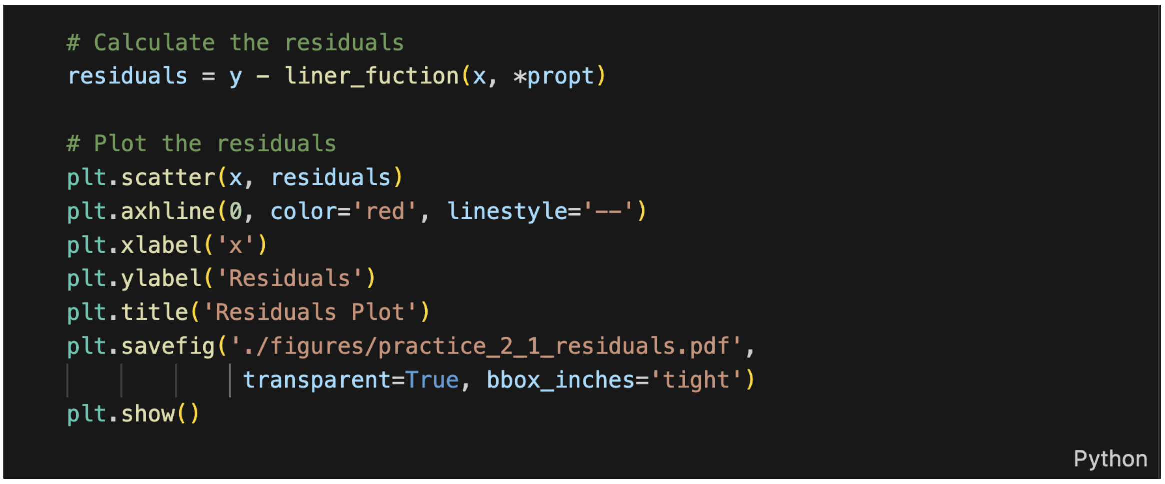
```
a: 2 -> 1.9861992566322368
b: 3 -> 3.9720263119202106
```

Figure 14: Fit the noisy y with the linear function (Fig.12) with `curve_fit`.



```python
plt.scatter(x, y, label=f'Samples: y = {a}x + {b} + noise')
plt.plot(x, liner_fuction(x, *propt), color='red',
            label=f'Fitted line: y = {propt[0]:.2f}x + {propt[1]:.2f}')
for i in range(n_sample):
    plt.plot([x[i], x[i]], [y[i], liner_fuction(x[i], *propt)],
                color='k', linestyle='--', label='Residuals' if i == 0 else None)
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Noised Samples with Fitted Line')
plt.legend()
plt.savefig('./figures/practice_2_1_fitted_line.pdf',
               transparent=True, bbox_inches='tight')
plt.show()
```

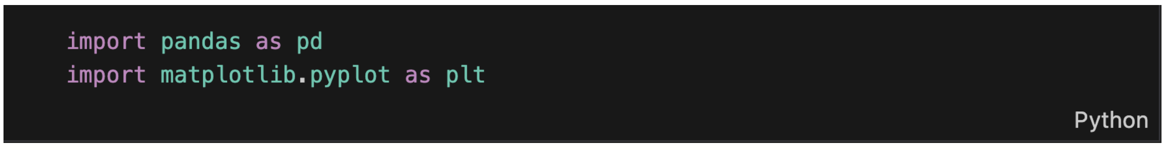
Figure 15: Code to plot the data points and the fitting curve



```python
# Calculate the residuals
residuals = y - liner_fuction(x, *propt)

# Plot the residuals
plt.scatter(x, residuals)
plt.axhline(0, color='red', linestyle='--')
plt.xlabel('x')
plt.ylabel('Residuals')
plt.title('Residuals Plot')
plt.savefig('./figures/practice_2_1_residuals.pdf',
               transparent=True, bbox_inches='tight')
plt.show()
```

Figure 16: Code to plot the residuals.

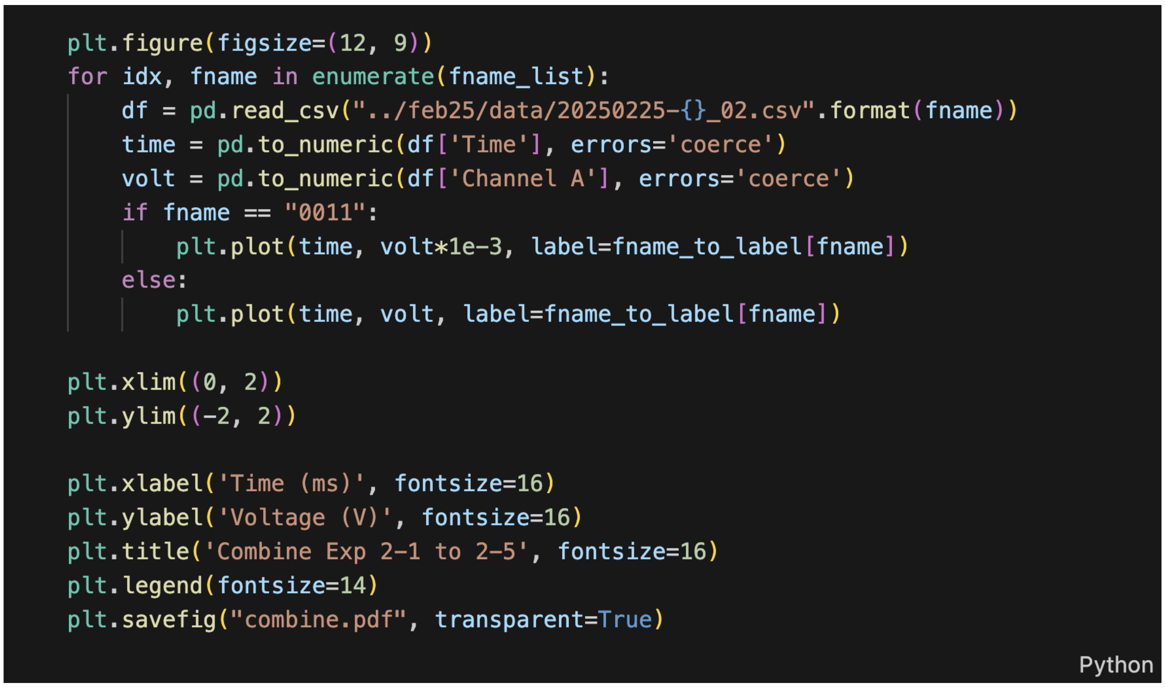
2.4 Practice 4



```python
import pandas as pd
import matplotlib.pyplot as plt
```

Figure 17: Import `panda` and `matplotlib.pyplot` packages to read csv files and make plots, respectively

1. Read out the data from a csv file.
2. Plot the data points.
3. The following is an example of how to read out the data from a csv file.
4. You can modify the code to read out the data from the file you have

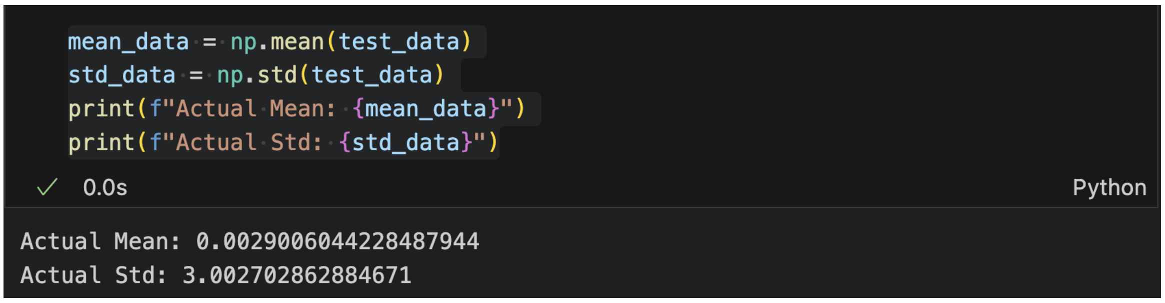


```python
plt.figure(figsize=(12, 9))
for idx, fname in enumerate(fname_list):
    df = pd.read_csv("../feb25/data/20250225-{}_02.csv".format(fname))
    time = pd.to_numeric(df['Time'], errors='coerce')
    volt = pd.to_numeric(df['Channel A'], errors='coerce')
    if fname == "0011":
        plt.plot(time, volt*1e-3, label=fname_to_label[fname])
    else:
        plt.plot(time, volt, label=fname_to_label[fname])

plt.xlim((0, 2))
plt.ylim((-2, 2))

plt.xlabel('Time (ms)', fontsize=16)
plt.ylabel('Voltage (V)', fontsize=16)
plt.title('Combine Exp 2-1 to 2-5', fontsize=16)
plt.legend(fontsize=14)
plt.savefig("combine.pdf", transparent=True)
```

Figure 18: Using ”for loop” to iteratively read files and plot the data.

3 Result

3.1 Practice 1



```
mean_data = np.mean(test_data)
std_data = np.std(test_data)
print(f"Actual Mean: {mean_data}")
print(f"Actual Std: {std_data}")
```

```
Actual Mean: 0.0029006044228487944
Actual Std: 3.002702862884671
```

Figure 19: The actual mean μ and standard deviation σ of our sample: $\mu \approx 0.0029$, $\sigma \approx 3.0027$

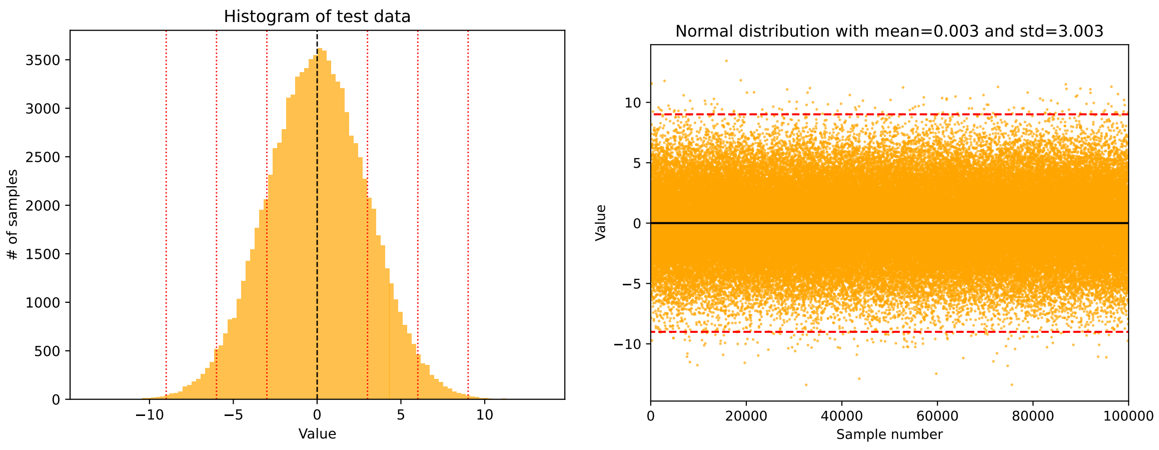
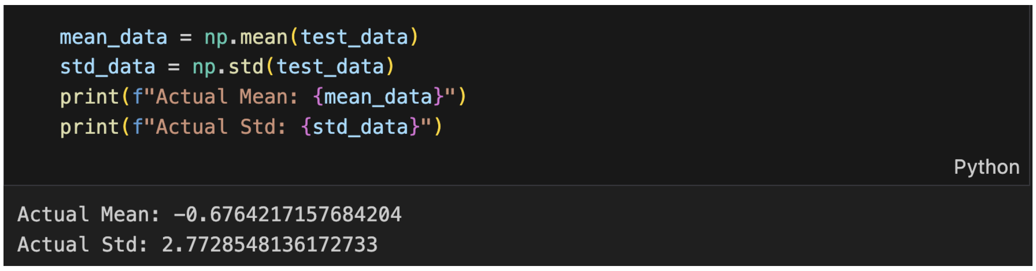


Figure 20: *left*: The distribution of our samples (normal distribution of 10000 samples with mean = 0 and standard deviation = 3). The black dashed line labels the mean of the samples, and the red dotted lines from inner to outer represent the range of $\pm 1\sigma$, $\pm 2\sigma$, and $\pm 3\sigma$, respectively. *right*: The data points of our samples. The black solid line labels the mean of the samples, and the red dashed lines note the range with $\pm 3\sigma$.

3.2 Practice 2



```python
mean_data = np.mean(test_data)
std_data = np.std(test_data)
print(f"Actual Mean: {mean_data}")
print(f"Actual Std: {std_data}")
```

```
Actual Mean: -0.6764217157684204
Actual Std: 2.7728548136172733
```

Figure 21: The actual mean μ and standard deviation σ of our sample: $\mu \approx -0.6764$, $\sigma \approx 2.7729$

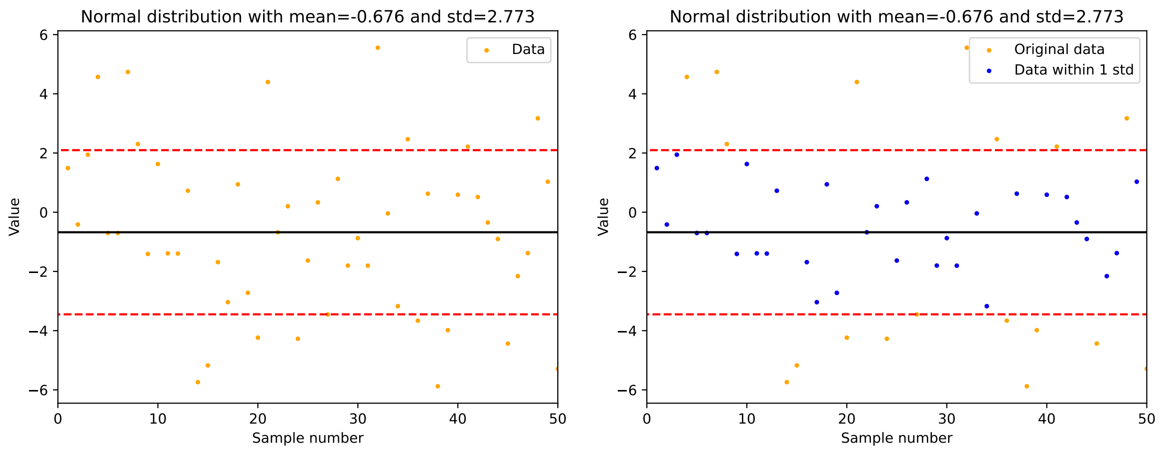
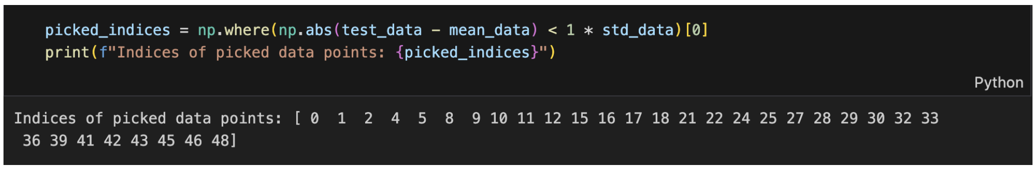


Figure 22: *left*: The data points of our samples (normal distribution of 50 samples with mean = 0 and standard deviation = 3). The black solid line labels the mean of the samples, and the red dashed lines note the range with $\pm 1\sigma$. *right*: The same plot but the blue points are within $\pm 1\sigma$.



```python
picked_indices = np.where(np.abs(test_data - mean_data) < 1 * std_data)[0]
print(f"Indices of picked data points: {picked_indices}")
```

```
Indices of picked data points: [ 0  1  2  4  5  8  9 10 11 12 15 16 17 18 21 22 24 25 27 28 29 30 32 33
 36 39 41 42 43 45 46 48]
```

Figure 23: Print all the indices of the data points that are picked (The indices of the data points that are picked: 0, 1, 2, 4, 5, 8, 9, 10, 11, 12, 15, 16, 17, 18, 21, 22, 24, 25, 27, 28, 29, 30, 32, 33, 36, 39, 41, 42, 43, 45, 46, 48)

3.3 Practice 3

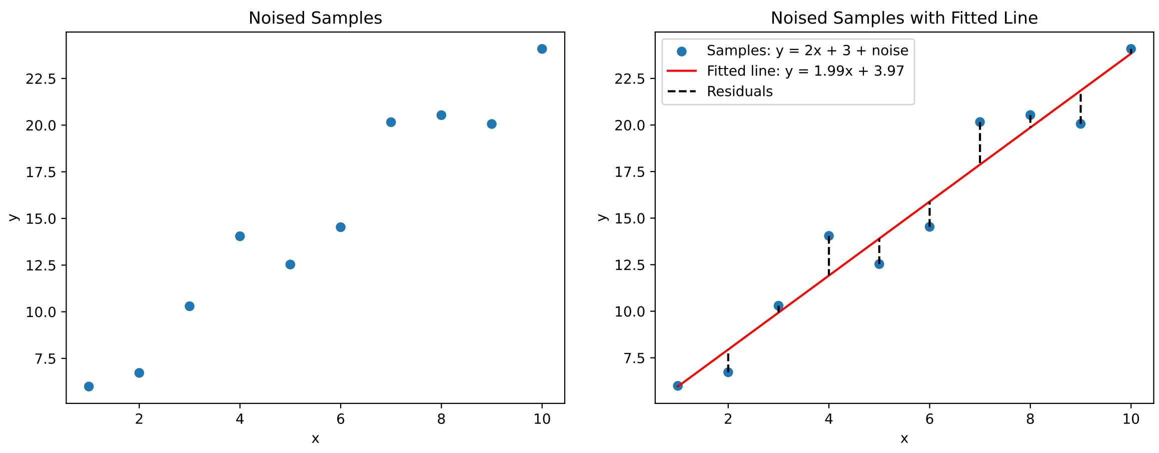


Figure 24: *left*: The scattered plot of of x (`np.linspace(1, 10, 10)`) and $y = a * x + b + noise$ where the noise is the normal distribution with mean =0 and standard deviation = 2. *right*: The data point (x, y) (blue dots) and the fitted curve (x, y_{fit}) (red line). The residuals of each point are also labeled with black dashed lines on blue dots.

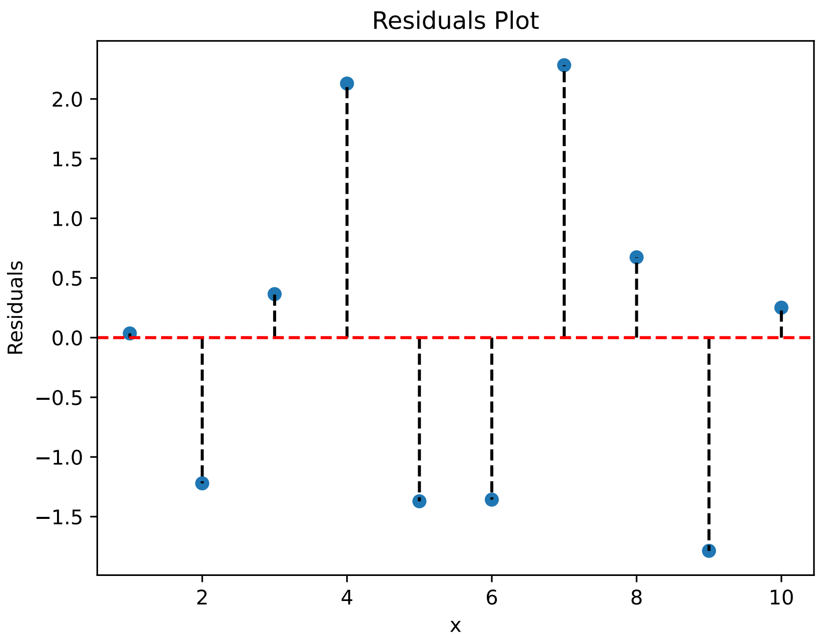


Figure 25: The extracted residuals from Fig.24. The horizontal red line labels residual=0

3.4 Practice 4

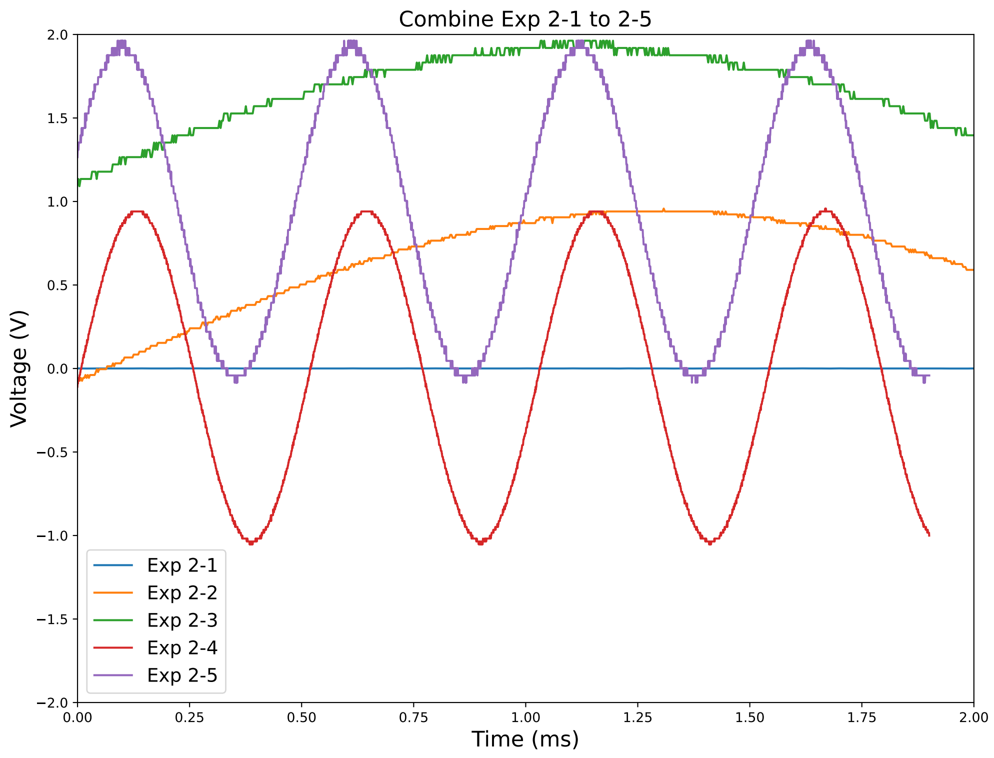


Figure 26: The results

4 Analysis and Discussion

4.1 Practice 1

Will the mean and standard deviation be the same as the population?

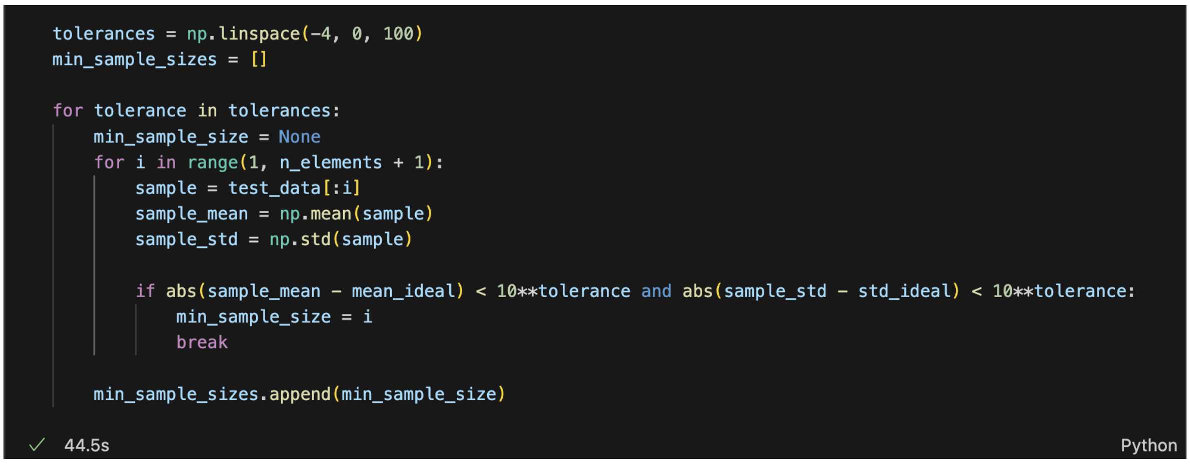
Comparing the initial mean and deviation to the data above (Fig.19), the two values are not equal.

The primary reason for this difference is sampling variability, which means that different samples might lead to different results. Since our samples are random samples based on Gaussian distribution, the number of samples might not be enough to make the two values equal. According to the law of large numbers (LLN), the average of the results obtained from a large number of independent random samples converges to the true value, if it exists. Thus, if we want to make these two values equal, we should raise the sample size.

To testify to the law of large numbers (LLN), we examine how many samples are required to decrease the difference in means and standard deviations, i.e., to check the minimum sample size to make the differences lower than assigned tolerances.

$$|\mu_{sample} - \mu_{ideal}| < tolerance \tag{4}$$

$$|\sigma_{sample} - \sigma_{ideal}| < tolerance \tag{5}$$



```python
tolerances = np.linspace(-4, 0, 100)
min_sample_sizes = []

for tolerance in tolerances:
    min_sample_size = None
    for i in range(1, n_elements + 1):
        sample = test_data[:i]
        sample_mean = np.mean(sample)
        sample_std = np.std(sample)

        if abs(sample_mean - mean_ideal) < 10**tolerance and abs(sample_std - std_ideal) < 10**tolerance:
            min_sample_size = i
            break

    min_sample_sizes.append(min_sample_size)
```

Figure 27: Code to testify law of large numbers

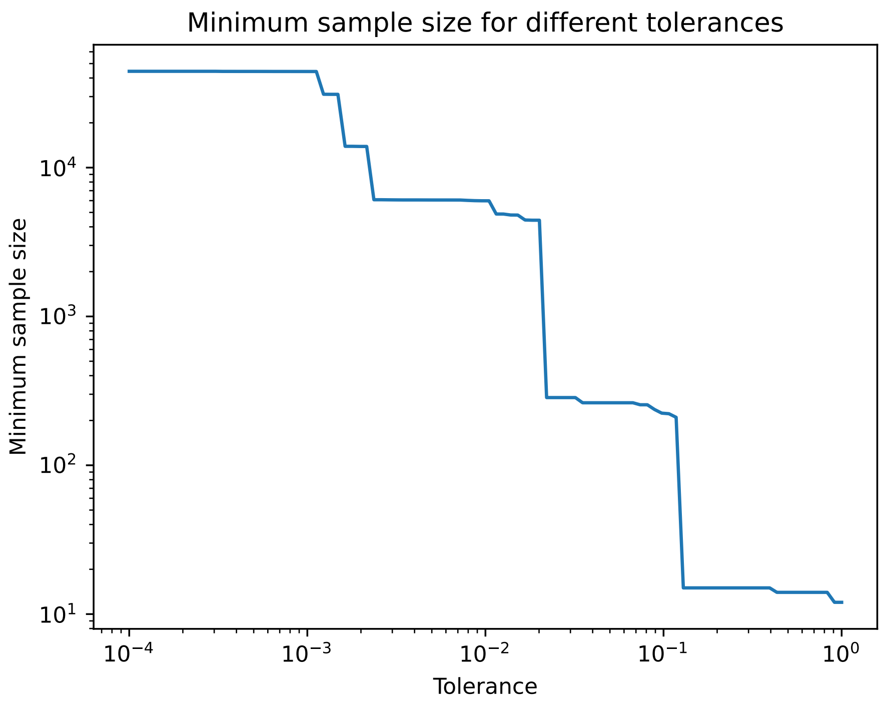


Figure 28: Minimum sample sizes required to match the given tolerances

As we can see in Fig. 28, the smaller the given tolerance, the larger the minimum sample sizes needed, confirming the law of large numbers (LLN).

4.2 Practice 2

This experiment analyzes the statistical characteristics of 50 randomly generated data points with a mean of 0 and a standard deviation of 3. The obtained sample mean is approximately -0.6764, and the standard deviation is around 2.7729.

From the analysis in Fig. 22, 33 data points fall within ± 1 standard deviation of the sample mean, accounting for about 66% of the total data. This is close to the theoretical expectation that approximately 68% of the data should lie within ± 1 standard deviation of the mean. Since the data is randomly generated, experimental results may deviate from theoretical values. To obtain results that more closely align with theoretical expectations, increasing the sample size can help reduce the effect of random errors.

4.3 Practice 3

We add noise to a linear equation and fit the data, where the noise follows a normal distribution with a mean of 0 and a standard deviation of 2. This setting ensures that no outliers appear in the dataset.

Observing the left plot in Fig. 24, the linear trend remains evident.

Analysis of Fig. 25:

1. Definition of residuals:
   The residuals r_i are defined as the difference between the observed value y_i and the predicted value $\hat{y}_i$:
   $$r_i = y_i - \hat{y}_i \tag{6}$$
2. The residuals are approximately distributed around 0, indicating that the linear model has a reasonable fitting capability.
3. Since the noise has a standard deviation of 2, most residuals fall within the range $[-4, 4]$, which aligns with our assumptions.

4.4 Practice 4

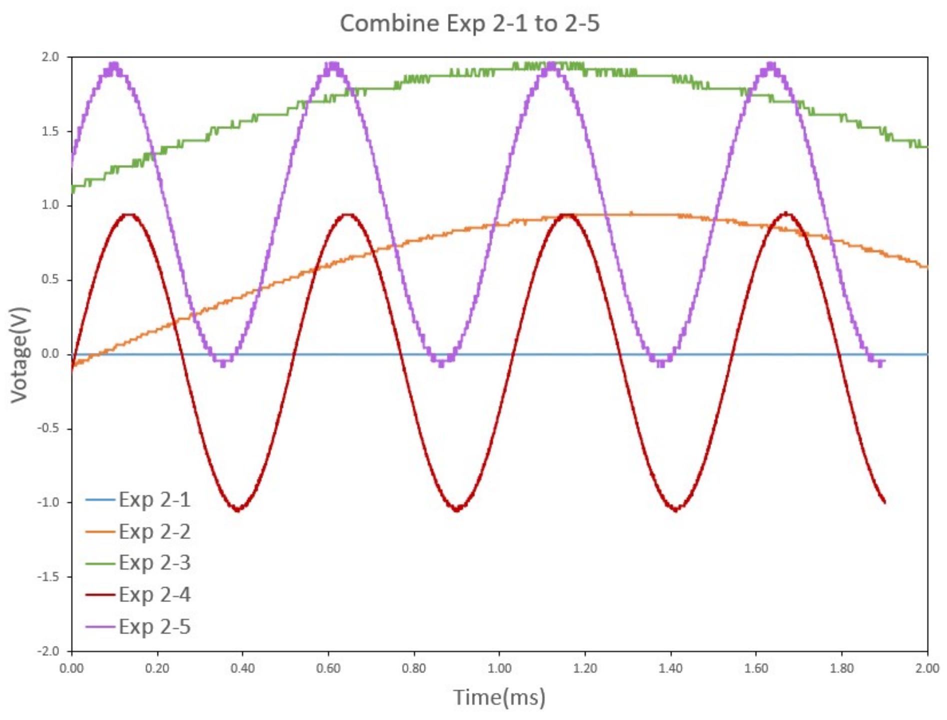


Figure 29: Plot Results from Excel

The above results were obtained by plotting the same dataset using Excel. Comparing them with the results generated using Python, we can see that they are completely consistent.

5 Conclusion

In this course, we learned how to use Python and explored various data processing tools, including NumPy, Matplotlib, Pandas, SciPy, and Anaconda. With these tools, we were able to efficiently obtain the necessary data and create various types of charts that meet specific requirements using Python.

Throughout the course, we completed several hands-on exercises. Practice 1 focused on analyzing the properties of normal distribution using Python. We imported the numpy and matplotlib.pyplot packages and discussed the differences in mean and standard deviation values. Practice 2 emphasized data visualization, also utilizing numpy and matplotlib.pyplot.

Practice 3 introduced the application of the curve_fit function, where we learned to plot fitted points and then calculate and plot the residuals. Practice 4 involved reading data from a CSV file and generating a combined chart for Exp 2-1 to 2-5.

In addition to practicing basic Python syntax and applying it to exercises, we also analyzed the results of our data and visualizations, reflecting on how to improve our methods to achieve more accurate outcomes.

6 Work division

- 洪瑜： Analysis, Report
- 黄巧涵： Analysis, Report
- 洪懌平： Coding, Report

Figure 30: 洪瑜(*left*), 黄巧涵(*middle*), and 洪懌平(*right*)

7 Reference

1 : The Standard Normal Distribution:
https://www.scribbr.co.uk/stats/the-standard-normal-distribution/

8 Appendix

The source codes of these practices can be found at:

- **Practice 1:**
  https://github.com/hyp0515/exp_phy_ii/blob/main/mar25/practice_1_1.ipynb
- **Practice 2:**
  https://github.com/hyp0515/exp_phy_ii/blob/main/mar25/practice_1_2.ipynb
- **Practice 3:**
  https://github.com/hyp0515/exp_phy_ii/blob/main/mar25/practice_2_1.ipynb
- **Practice 4:**
  https://github.com/hyp0515/exp_phy_ii/blob/main/mar25/practice_2_2.ipynb