# Experimental Physics (II) Notebook

# Fundamental Python
# Basic Usage of Python


Group 2

洪　瑜 B125090009

黃巧涵 B122030003

洪懌平 B102030019


2025/03/25

# 1 Experimental steps, preliminary results, and preliminary analysis

## 1.1 Practice 1

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

np.random.seed(42)
```
✓ 0.3s Python

Figure 1: 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 2: Initialize our samples (normal distribution of 10000 samples with mean = 0 and standard deviation = 3).

```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}")
```
✓ 0.0s Python

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

Figure 3: The actual mean $\mu$ and standard deviation $\sigma$ of our sample: $\mu \approx 0.0029$, $\sigma \approx 3.0027$

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

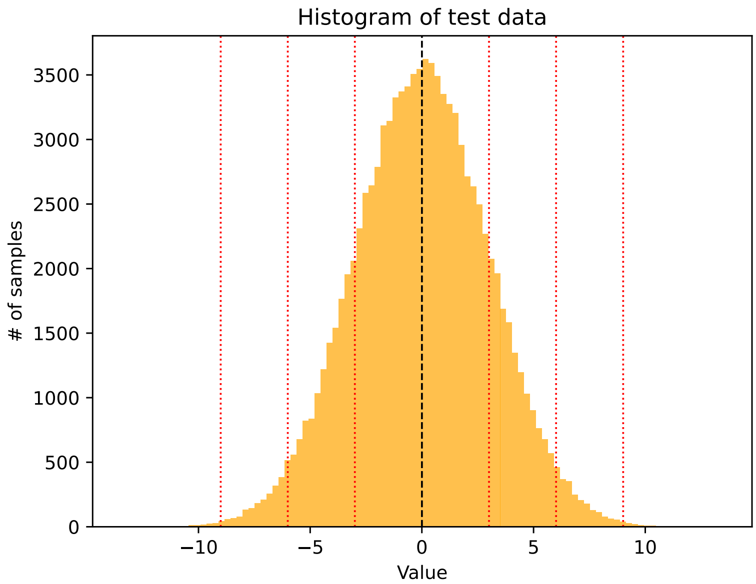


Figure 5: The distribution of our samples (normal distribution of 10000 samples with mean = 0 and standard deviation = 3). The black dashed line labels the mean of the samples, and the red dotted lines from inner to outer represent the range of $\pm 1\sigma$, $\pm 2\sigma$, and $\pm 3\sigma$, respectively.

```python
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()
```
✓ 0.3s Python

Figure 6: Code for making scattered plot of our samples

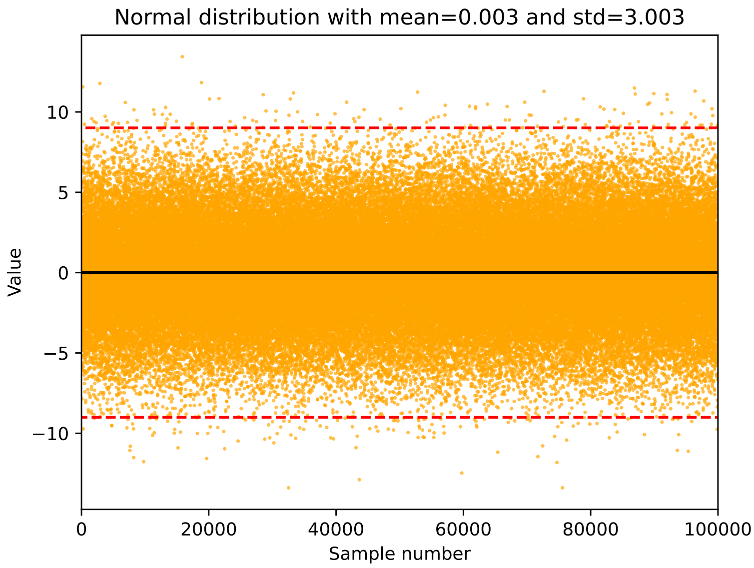


Figure 7: The data points of our samples. The black solid line labels the mean of the samples, and the red dashed lines note the range with $\pm 3\sigma$.

3. Will the mean and standard deviation be the same as the population?
   Comparing the initial mean and deviation to the data above (Fig.3), 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.

$$|\mu_{sample} - \mu_{ideal}| < tolerance \tag{1}$$
$$|\sigma_{sample} - \sigma_{ideal}| < tolerance \tag{2}$$

```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)
```

✓ 44.5s Python

Figure 8: Code to testify law of large numbers

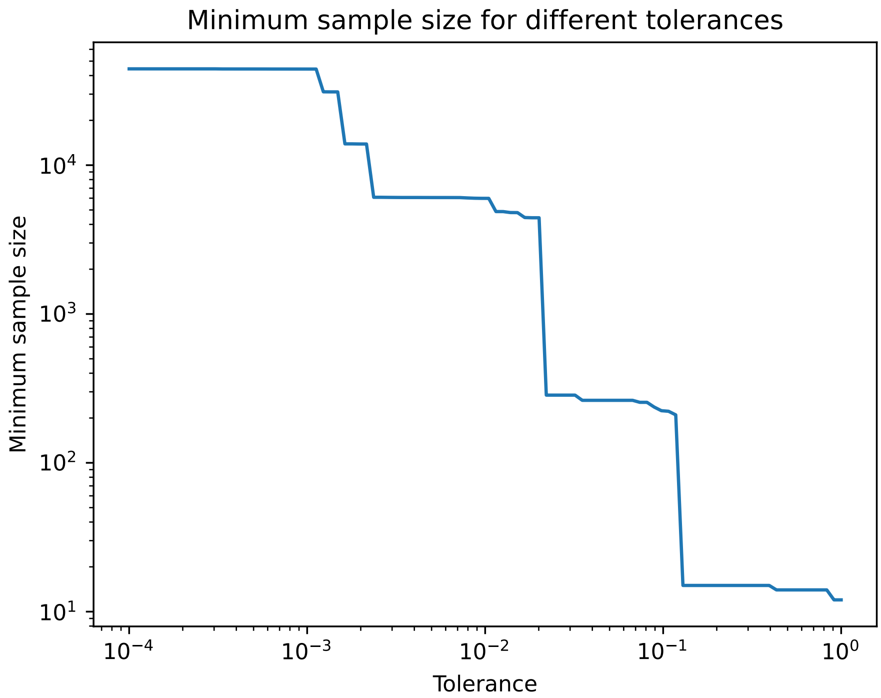


Figure 9: Minimum sample sizes required to match the given tolerances

As we can see in Fig.9, the smaller the given tolerance, the larger the minimum sample sizes needed, confirming the law of large numbers (LLN).

## 1.2 Practice 2

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

np.random.seed(42)
```
Python

Figure 10: 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.

```python
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)
```
Python

Figure 11: Initialize our samples (normal distribution of 50 samples with mean = 0 and standard deviation = 3).

```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}")
```
Python

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

Figure 12: The actual mean $\mu$ and standard deviation $\sigma$ of our sample: $\mu \approx -0.6764$, $\sigma \approx 2.7729$

```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 13: Code for making scattered plot of our samples

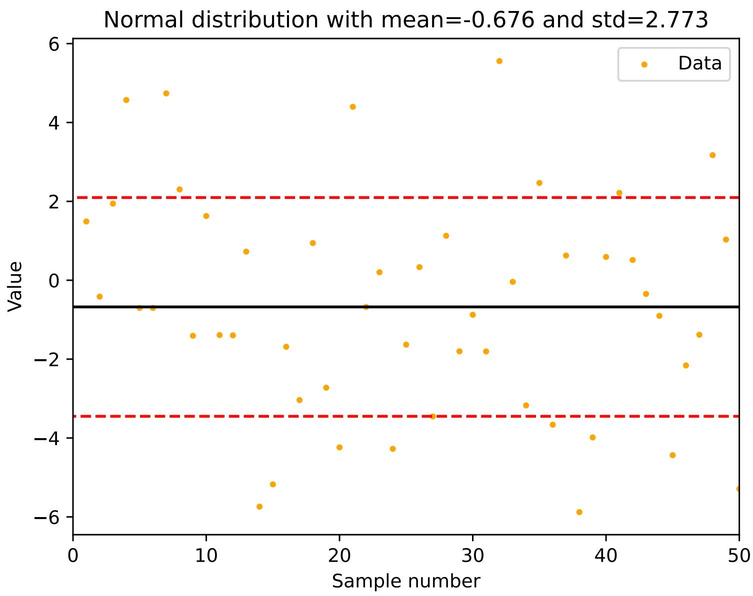


Figure 14: The data points of our samples (normal distribution of 50 samples with mean = 0 and standard deviation = 3). The black solid line labels the mean of the samples, and the red dashed lines note the range with $\pm 1\sigma$.

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 15: 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 16: Code for making scattered plot of our samples

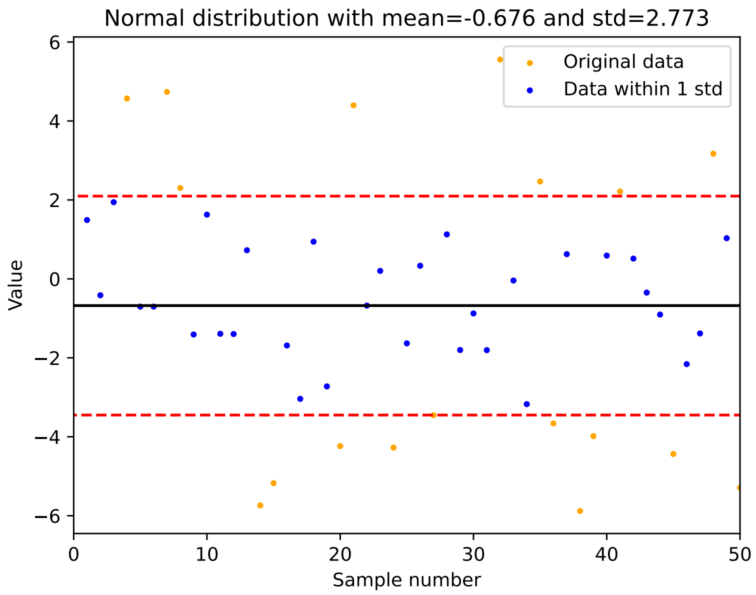


Figure 17: The same plot as Fig.14 but the orange points are the data outside the $\pm 1\sigma$, and the blue points are within $\pm 1\sigma$.

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

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

Python

```
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 18: 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)

## 1.3 Practice 3

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

np.random.seed(42)
```

Python

Figure 19: 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

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

Python

Figure 20: 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 $\sigma_0$ 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
```
Python

Figure 21: 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

```python
plt.scatter(x, y)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Noised Samples')
plt.savefig('./figures/practice_2_1_original_samples.pdf',
            transparent=True, bbox_inches='tight')
plt.show()
```
Python

Figure 22: Code for making scattered plot of $x$ and $y$

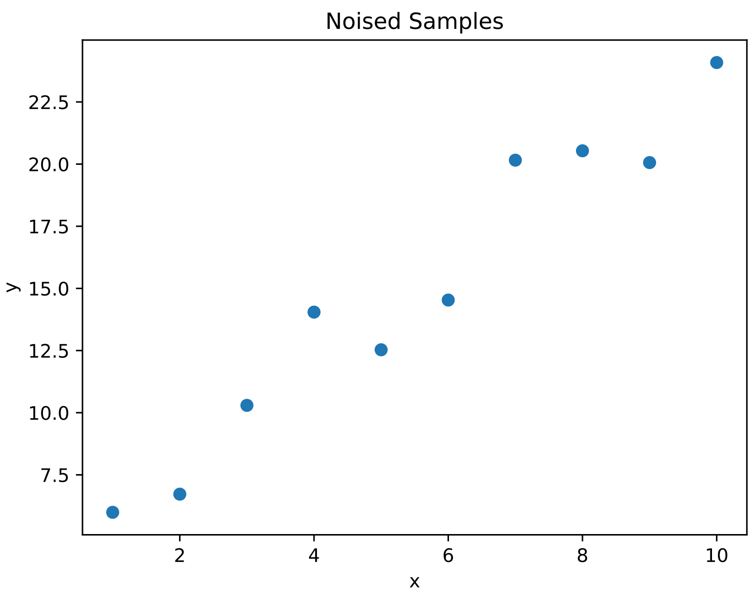


Figure 23: The scattered plot of of $x$ (`np.linspace(1, 10, 10)`) and $y = a*x+b+noise$ where the noise is the normal distribution with mean =0 and standard deviation = 2

3. Use the `curve_fit` function in `scipy.optimize` to fit the data points with the function $y = a * x + b$.

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

Python

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

Figure 24: Fit the noisy $y$ with the linear function (Fig.20) with `curve_fit`. The result is $y_{fit} \approx 1.9861x + 3.9720$.

4. Plot the data points and the fitting curve.

```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()
```

Python

Figure 25: Code to plot the data points and the fitting curve

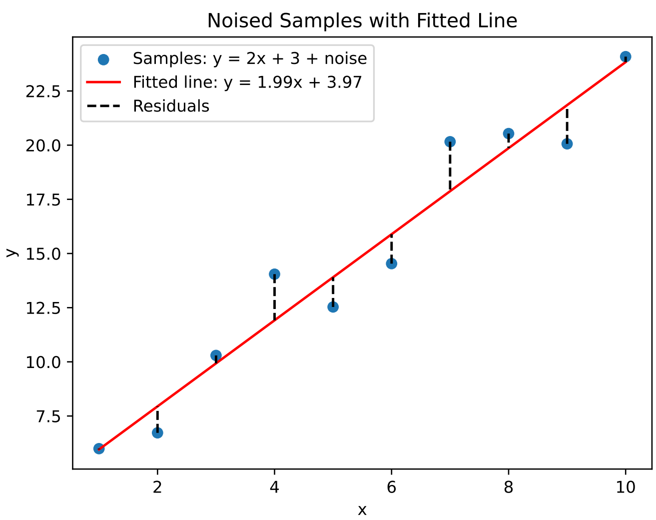


Figure 26: The data point $(x, y)$ (blue dots) and the fitted curve $(x, y_{fit})$ (red line). The residuals of each point are also labeled with black dashed lines on blue dots.

5. Calculate the residuals and plot the residuals.

```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()
```

Python

Figure 27: Code to plot the residuals.

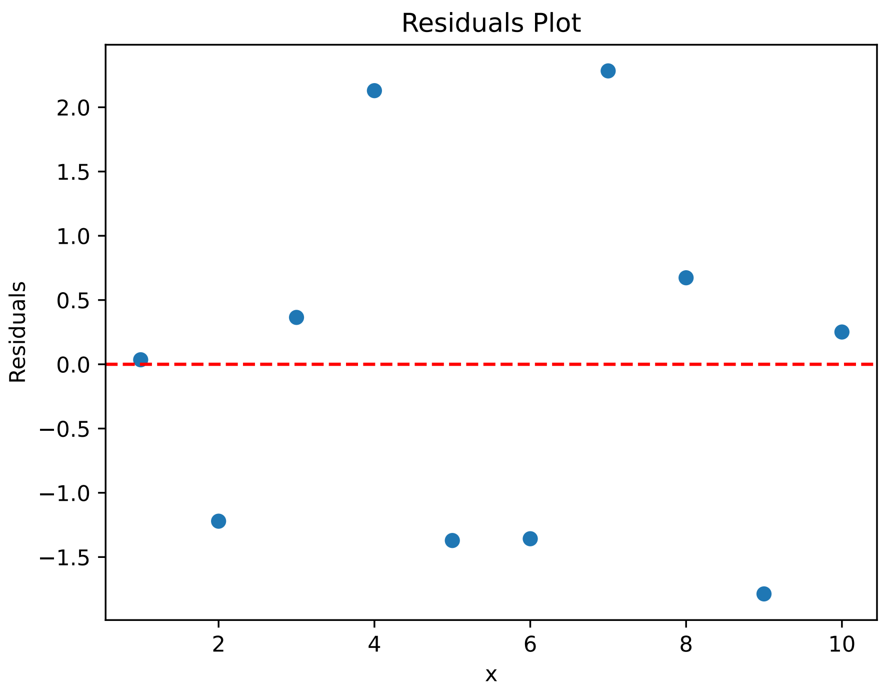


Figure 28: The extracted residuals from Fig.23. The horizontal red line labels residual=0

## 1.4 Practice 4

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

Python

Figure 29: Import `panda` and `matplotlib.pyplot` packages to read csv files and make plots, respectively

The data used in this practice are from PicoScope in the first week (Experiment 1; data can be found at https://github.com/hyp0515/exp_phy_ii/tree/main/feb25/data).

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
fname_list = ["0011", "0008", "0007", "0009", "0010"]
fname_to_label = {
    "0011": "Exp 2-1",
    "0008": "Exp 2-2",
    "0007": "Exp 2-3",
    "0009": "Exp 2-4",
    "0010": "Exp 2-5",
}
```

Python

Figure 30: File names to be read.

```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)
```

Python

Figure 31: Using "for loop" to iteratively read files and plot the data.

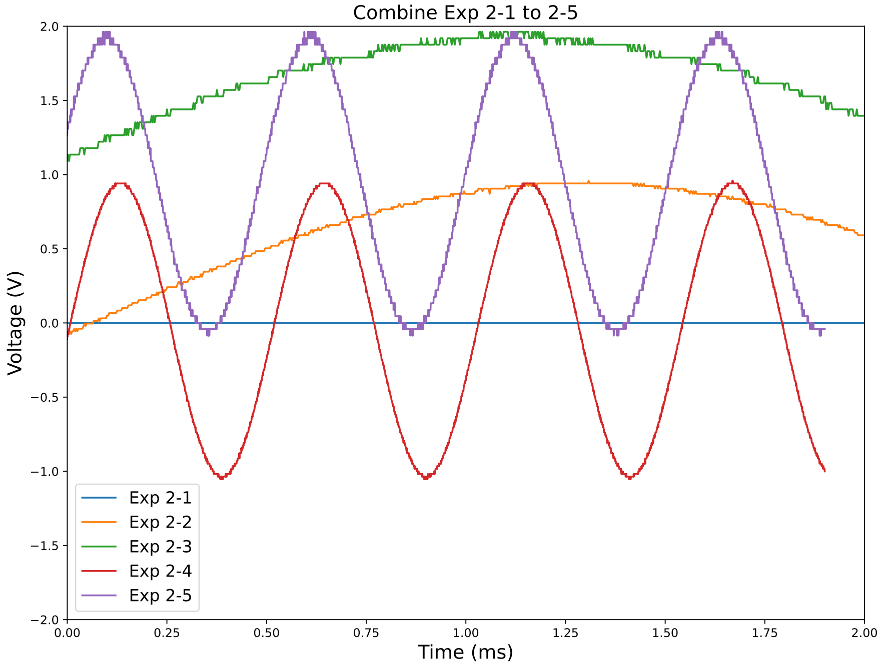


Figure 32: The results

# 2 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