Seminar: Curve Fitting Exercises

```
#| edit: false
#| echo: false
#| execute: true

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

# Set default plotting parameters
plt.rcParams.update({
    'font.size': 10,
    'lines.linewidth': 1.5,
    'lines.markersize': 5,
    'axes.labelsize': 10,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
    'xtick.top': True,
    'xtick.direction': 'in',
    'ytick.right': True,
    'ytick.direction': 'in',
})

def get_size(w, h):
    return (w/2.54, h/2.54)

# ============================================
# DATASETS FOR EXERCISES
# ============================================

# Dataset 1: Linear data (Ohm's law: V = IR)
np.random.seed(42)
current_data = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
```

```
resistance_true = 47.0  # Ohms
voltage_data = resistance_true * current_data + np.random.normal(0, 0.5, len(current_data))
voltage_err = np.ones(len(current_data)) * 0.5

# Dataset 2: Exponential decay (radioactive decay)
np.random.seed(123)
time_decay = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
half_life_true = 3.0  # time units
N0_true = 1000
decay_data = N0_true * np.exp(-np.log(2) * time_decay / half_life_true)
decay_data = decay_data + np.random.normal(0, 20, len(time_decay))
decay_err = np.sqrt(np.abs(decay_data))  # Poisson-like errors

# Dataset 3: Pendulum period vs length (T = 2√(L/g))
np.random.seed(456)
length_data = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
g_true = 9.81
period_data = 2 * np.pi * np.sqrt(length_data / g_true)
period_data = period_data + np.random.normal(0, 0.02, len(length_data))
period_err = np.ones(len(length_data)) * 0.02

# Dataset 4: Gaussian peak (spectroscopy)
np.random.seed(789)
wavelength_data = np.linspace(500, 600, 25)
amplitude_true = 100
center_true = 550
sigma_true = 15
gaussian_data = amplitude_true * np.exp(-(wavelength_data - center_true)**2 / (2 * sigma_tru
gaussian_data = gaussian_data + np.random.normal(0, 5, len(wavelength_data))
gaussian_err = np.ones(len(wavelength_data)) * 5

# Dataset 5: Fluorescence lifetime (TCSPC simulation)
np.random.seed(321)
tau_true = 4.2  # nanoseconds
n_photons = 500
photon_arrivals = np.random.exponential(tau_true, n_photons)
# Remove photons arriving after detection window
photon_arrivals = photon_arrivals[photon_arrivals < 30]
```

Introduction

In this seminar, you will practice the curve fitting concepts from the lecture. Each exercise builds on the previous one, progressively developing your skills in:

1. Implementing fitting formulas
2. Using `scipy.optimize.curve_fit`
3. Evaluating fit quality
4. Handling real-world complications

Instructions: Work through the exercises in order. Each exercise has hints and solutions available. Try to solve each problem yourself before looking at the solution.

```
#| setup: true
#| exercise: ex_1

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

def get_size(w, h):
    return (w/2.54, h/2.54)

# Dataset: Linear data (Ohm's law: V = IR)
np.random.seed(42)
current_data = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
resistance_true = 47.0  # Ohms
voltage_data = resistance_true * current_data + np.random.normal(0, 0.5, len(current_data))
voltage_err = np.ones(len(current_data)) * 0.5
```

Exercise 1: Linear Regression by Hand

Ohm's Law: The voltage across a resistor is proportional to the current: $V = IR$
We measured voltage at different currents to determine the resistance of an unknown resistor. Your task is to:

1. Implement the weighted linear regression formulas from the lecture
2. Calculate the resistance (slope) and its uncertainty
3. Compare your result with `curve_fit`

Recall the formulas (for $y = bx$, forcing intercept through zero):

$$b = \frac{S_{xy}}{S_{xx}}, \quad \sigma_b = \sqrt{\frac{1}{S_{xx}}}$$

where $S_{xy} = \sum \frac{x_i y_i}{\sigma_i^2}$ and $S_{xx} = \sum \frac{x_i^2}{\sigma_i^2}$
Time estimate: 25-30 minutes

```
#| exercise: ex_1

# Data: Current (A) and Voltage (V) measurements
I = current_data  # Current in Amperes
V = voltage_data  # Voltage in Volts
sigma = voltage_err  # Uncertainty in Voltage

# Step 1: Plot the data with error bars
plt.figure(figsize=get_size(10, 8))
# Your plotting code here
____

# Step 2: Calculate weighted sums
S_xx = ____
S_xy = ____

# Step 3: Calculate resistance (slope) and its uncertainty
R_manual = ____
R_uncertainty = ____

print(f"Manual calculation: R = {R_manual:.2f} ± {R_uncertainty:.2f} Ω")

# Step 4: Compare with curve_fit
def linear_through_origin(x, R):
    return R * x

popt, pcov = curve_fit(____)
R_curvefit = popt[0]
R_curvefit_err = np.sqrt(pcov[0, 0])

print(f"curve_fit result:   R = {R_curvefit:.2f} ± {R_curvefit_err:.2f} Ω")
print(f"True resistance:    R = {resistance_true:.2f} Ω")
```

Tip

For the weighted sums, loop over all data points and accumulate: - `S_xx += (I[i]**2) / (sigma[i]**2)` - `S_xy += (I[i] * V[i]) / (sigma[i]**2)`

For `curve_fit`, you need to pass: the function, x-data, y-data, and use `sigma=` for uncertainties and `absolute_sigma=True`.

Solution.

Note

```
# Data
I = current_data
V = voltage_data
sigma = voltage_err

# Step 1: Plot the data
plt.figure(figsize=get_size(10, 8))
plt.errorbar(I, V, yerr=sigma, fmt='o', capsize=3, label='Data')
plt.xlabel('Current (A)')
plt.ylabel('Voltage (V)')
plt.title("Ohm's Law: Determining Resistance")
plt.legend()
plt.tight_layout()
plt.show()

# Step 2: Calculate weighted sums
S_xx = np.sum(I**2 / sigma**2)
S_xy = np.sum(I * V / sigma**2)

# Step 3: Calculate resistance and uncertainty
R_manual = S_xy / S_xx
R_uncertainty = np.sqrt(1 / S_xx)

print(f"Manual calculation: R = {R_manual:.2f} ± {R_uncertainty:.2f} Ω")

# Step 4: Compare with curve_fit
def linear_through_origin(x, R):
    return R * x

popt, pcov = curve_fit(linear_through_origin, I, V, sigma=sigma, absolute_sigma=True)
R_curvefit = popt[0]
R_curvefit_err = np.sqrt(pcov[0, 0])

print(f"curve_fit result:   R = {R_curvefit:.2f} ± {R_curvefit_err:.2f} Ω")
print(f"True resistance:    R = {resistance_true:.2f} Ω")

# Plot with fit
plt.figure(figsize=get_size(10, 8))
plt.errorbar(I, V, yerr=sigma, fmt='o', capsize=3, label='Data')
I_fit = np.linspace(0, 1.1, 100)
plt.plot(I_fit, R_manual * I_fit, 'r-', label=f'Fit: R = {R_manual:.1f} Ω')
plt.xlabel('Current (A)')
plt.ylabel('Voltage (V)')
plt.legend()
plt.tight_layout()
plt.show()
```

```
#| setup: true
#| exercise: ex_2

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

def get_size(w, h):
    return (w/2.54, h/2.54)

# Dataset: Exponential decay (radioactive decay)
np.random.seed(123)
time_decay = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
half_life_true = 3.0  # time units
N0_true = 1000
decay_data = N0_true * np.exp(-np.log(2) * time_decay / half_life_true)
decay_data = decay_data + np.random.normal(0, 20, len(time_decay))
decay_err = np.sqrt(np.abs(decay_data))  # Poisson-like errors
```

Exercise 2: The Importance of Initial Guesses

Radioactive Decay: The number of radioactive nuclei decreases exponentially:

$$N(t) = N_0 \cdot e^{-\lambda t} = N_0 \cdot 2^{-t/t_{1/2}}$$

Your task is to:

1. Fit the decay data with good initial guesses
2. Try fitting with bad initial guesses and observe what happens
3. Calculate the half-life and its uncertainty

Time estimate: 20-25 minutes

```python
#| exercise: ex_2

# Data
t = time_decay  # Time
N = decay_data  # Count rate
N_err = decay_err  # Uncertainty

# Define the decay model
def decay_model(t, N0, half_life):
    return N0 * 2**(-t / half_life)

# Step 1: Plot the data first to estimate initial guesses
plt.figure(figsize=get_size(10, 8))
____
plt.show()

# Step 2: Estimate initial guesses from the plot
# N0 should be approximately the value at t=0
# half_life is the time when N drops to N0/2
N0_guess = ____
half_life_guess = ____

print(f"Initial guesses: N0 = {N0_guess}, half_life = {half_life_guess}")

# Step 3: Fit with good initial guesses
p0_good = [N0_guess, half_life_guess]
popt_good, pcov_good = curve_fit(____)

print(f"\nGood initial guess result:")
print(f"  N0 = {popt_good[0]:.1f} ± {np.sqrt(pcov_good[0,0]):.1f}")
print(f"  half_life = {popt_good[1]:.2f} ± {np.sqrt(pcov_good[1,1]):.2f}")

# Step 4: Try with bad initial guesses
p0_bad = [10, 100]  # Very wrong!
try:
    popt_bad, pcov_bad = curve_fit(decay_model, t, N, sigma=N_err, p0=p0_bad,
                                    absolute_sigma=True, maxfev=200)
    print(f"\nBad initial guess result:")
    print(f"  N0 = {popt_bad[0]:.1f}, half_life = {popt_bad[1]:.2f}")
except Exception as e:
    print(f"\nBad initial guess failed: {type(e).__name__}")

# Step 5: Calculate reduced chi-squared for the good fit
____
```

Tip

1. From the plot, estimate N0 as the first data point value (~1000)
2. Find where N drops to about half of N0 to estimate the half-life (~3)
3. For `curve_fit`, pass `sigma=N_err` and `p0=p0_good`
4. Calculate $\chi^2 = \sum((N - N_{model})^2 / N_{err}^2)$ and divide by degrees of freedom (N__points - N__params)

Solution.

Note

```
# Data
t = time_decay
N = decay_data
N_err = decay_err

def decay_model(t, N0, half_life):
    return N0 * 2**(-t / half_life)

# Step 1: Plot the data
plt.figure(figsize=get_size(10, 8))
plt.errorbar(t, N, yerr=N_err, fmt='o', capsize=3)
plt.xlabel('Time (arbitrary units)')
plt.ylabel('Count rate')
plt.title('Radioactive Decay')
plt.tight_layout()
plt.show()

# Step 2: Estimate initial guesses
N0_guess = 1000  # From the plot at t=0
half_life_guess = 3  # Where N   500

print(f"Initial guesses: N0 = {N0_guess}, half_life = {half_life_guess}")

# Step 3: Fit with good guesses
p0_good = [N0_guess, half_life_guess]
popt_good, pcov_good = curve_fit(decay_model, t, N, sigma=N_err,
                                  p0=p0_good, absolute_sigma=True)

print(f"\nGood initial guess result:")
print(f"  N0 = {popt_good[0]:.1f} ± {np.sqrt(pcov_good[0,0]):.1f}")
print(f"  half_life = {popt_good[1]:.2f} ± {np.sqrt(pcov_good[1,1]):.2f}")
print(f"  True half_life = {half_life_true:.2f}")

# Step 4: Try bad initial guesses
p0_bad = [10, 100]
try:
    popt_bad, pcov_bad = curve_fit(decay_model, t, N, sigma=N_err, p0=p0_bad,
                                    absolute_sigma=True, maxfev=200)
    print(f"\nBad initial guess result:")
    print(f"  N0 = {popt_bad[0]:.1f}, half_life = {popt_bad[1]:.2f}")
except Exception as e:
    print(f"\nBad initial guess failed: {type(e).__name__}")

# Step 5: Calculate reduced chi-squared
N_model = decay_model(t, *popt_good)
chi2 = np.sum(((N - N_model) / N_err)**2)
dof = len(t) - 2  # 2 parameters
chi2_reduced = chi2 / dof

print(f"\nFit quality:")
print(f"   ² = {chi2:.2f}")
print(f"  Degrees of freedom = {dof}")
print(f"  Reduced  ² = {chi2_reduced:.2f}")
```

```
#| setup: true
#| exercise: ex_3

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

def get_size(w, h):
    return (w/2.54, h/2.54)

# Dataset: Pendulum period vs length (T = 2 √(L/g))
np.random.seed(456)
length_data = np.array([0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
g_true = 9.81
period_data = 2 * np.pi * np.sqrt(length_data / g_true)
period_data = period_data + np.random.normal(0, 0.02, len(length_data))
period_err = np.ones(len(length_data)) * 0.02
```

Exercise 3: Model Comparison and Residual Analysis

Pendulum Period: The period of a simple pendulum depends on its length:

$$T = 2\pi\sqrt{\frac{L}{g}}$$

We can rewrite this as $T = A \cdot L^n$ where theoretically $A = 2\pi/\sqrt{g}$ and $n = 0.5$. Your task is to:

1. Fit the data with a power law model
2. Extract the gravitational acceleration g
3. Analyze residuals to verify the model
4. Compare with a linear model to see why it fails

Time estimate: 25-30 minutes

```
#| exercise: ex_3

# Data
L = length_data  # Length in meters
T = period_data  # Period in seconds
T_err = period_err  # Uncertainty

# Define models
def power_law(L, A, n):
    return A * L**n

def linear_model(L, a, b):
    return a * L + b

# Step 1: Fit with power law
p0_power = [2.0, 0.5]  # Initial guess
popt_power, pcov_power = curve_fit(____)

A_fit = popt_power[0]
n_fit = popt_power[1]
print(f"Power law fit: T = {A_fit:.3f} * L^{n_fit:.3f}")

# Step 2: Extract g from A = 2 /√g
g_extracted = ____
print(f"Extracted g = {g_extracted:.2f} m/s² (true: {g_true:.2f} m/s²)")

# Step 3: Calculate residuals for power law
T_model_power = power_law(L, *popt_power)
residuals_power = ____

# Step 4: Fit with linear model
popt_linear, pcov_linear = curve_fit(____)
T_model_linear = linear_model(L, *popt_linear)
residuals_linear = ____

# Step 5: Calculate chi-squared for both models
chi2_power = ____
chi2_linear = ____
dof = len(L) - 2

print(f"\nReduced  ² (power law): {chi2_power/dof:.2f}")
print(f"Reduced  ² (linear):    {chi2_linear/dof:.2f}")

# Step 6: Plot data, fits, and residuals
fig, axes = plt.subplots(2, 2, figsize=get_size(16, 14))
```

```
# Top left: Data with power law fit
# Top right: Data with linear fit
# Bottom left: Power law residuals
# Bottom right: Linear residuals

----

plt.tight_layout()
```

Tip

1. Use `curve_fit(power_law, L, T, sigma=T_err, p0=p0_power, absolute_sigma=True)`
2. From $A = 2\pi/\sqrt{g}$, we get $g = (2\pi/A)^2$
3. Residuals: `T - T_model`
4. Chi-squared: `np.sum(((T - T_model) / T_err)**2)`
5. Look for systematic patterns in residuals — random scatter means good model

Solution.

Note

```
# Data
L = length_data
T = period_data
T_err = period_err

def power_law(L, A, n):
    return A * L**n

def linear_model(L, a, b):
    return a * L + b

# Step 1: Fit with power law
p0_power = [2.0, 0.5]
popt_power, pcov_power = curve_fit(power_law, L, T, sigma=T_err,
                                    p0=p0_power, absolute_sigma=True)
A_fit, n_fit = popt_power
A_err, n_err = np.sqrt(np.diag(pcov_power))
print(f"Power law fit: T = ({A_fit:.3f}±{A_err:.3f}) * L^({n_fit:.3f}±{n_err:.3f})")

# Step 2: Extract g
g_extracted = (2 * np.pi / A_fit)**2
print(f"Extracted g = {g_extracted:.2f} m/s² (true: {g_true:.2f} m/s²)")

# Step 3: Residuals for power law
T_model_power = power_law(L, *popt_power)
residuals_power = T - T_model_power

# Step 4: Fit with linear model
popt_linear, pcov_linear = curve_fit(linear_model, L, T, sigma=T_err, absolute_sigma=T
T_model_linear = linear_model(L, *popt_linear)
residuals_linear = T - T_model_linear

# Step 5: Chi-squared comparison
chi2_power = np.sum(((T - T_model_power) / T_err)**2)
chi2_linear = np.sum(((T - T_model_linear) / T_err)**2)
dof = len(L) - 2

print(f"\nReduced  ² (power law): {chi2_power/dof:.2f}")
print(f"Reduced  ² (linear):    {chi2_linear/dof:.2f}")

# Step 6: Plotting
fig, axes = plt.subplots(2, 2, figsize=get_size(16, 14))

L_fine = np.linspace(0.05, 1.05, 100)
```

```
# Top left: Power law fit
axes[0, 0].errorbar(L, T, yerr=T_err, fmt='o', capsize=3, label='Data')
axes[0, 0].plot(L_fine, power_law(L_fine, *popt_power), 'r-', label='Power law fit')
axes[0, 0].set_xlabel('Length (m)')
axes[0, 0].set_ylabel('Period (s)')
axes[0, 0].set_title(f'Power Law:  ²/dof = {chi2_power/dof:.2f}')
axes[0, 0].legend()
```

```
#| setup: true
#| exercise: ex_4

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

def get_size(w, h):
    return (w/2.54, h/2.54)

# Dataset: Gaussian peak (spectroscopy)
np.random.seed(789)
wavelength_data = np.linspace(500, 600, 25)
amplitude_true = 100
center_true = 550
sigma_true = 15
gaussian_data = amplitude_true * np.exp(-(wavelength_data - center_true)**2 / (2 * sigma_tru
gaussian_data = gaussian_data + np.random.normal(0, 5, len(wavelength_data))
gaussian_err = np.ones(len(wavelength_data)) * 5
```

Exercise 4: Gaussian Peak Fitting and Covariance

Spectroscopy: A spectral line can be modeled as a Gaussian peak:

$$I(\lambda) = A \cdot \exp\left(-\frac{(\lambda - \lambda_0)^2}{2\sigma^2}\right)$$

where A is the amplitude, λ_0 is the center wavelength, and σ is the width.
Your task is to:

1. Fit the spectral data to extract peak parameters
2. Report parameters with uncertainties
3. Examine the correlation matrix
4. Calculate the Full Width at Half Maximum (FWHM $= 2\sqrt{2\ln 2}\sigma \approx 2.355\sigma$)

Time estimate: 20-25 minutes

```
#| exercise: ex_4

# Data
wavelength = wavelength_data  # nm
intensity = gaussian_data  # arbitrary units
intensity_err = gaussian_err

# Define Gaussian model
def gaussian(x, A, x0, sigma):
    return A * np.exp(-(x - x0)**2 / (2 * sigma**2))

# Step 1: Estimate initial parameters from the data
A_guess = ____  # Maximum intensity
x0_guess = ____  # Wavelength at maximum
sigma_guess = ____  # Estimate from peak width

p0 = [A_guess, x0_guess, sigma_guess]

# Step 2: Perform the fit
popt, pcov = curve_fit(____)

# Step 3: Extract parameters and uncertainties
A_fit, x0_fit, sigma_fit = popt
uncertainties = np.sqrt(np.diag(pcov))
A_err, x0_err, sigma_err = uncertainties

print("Fitted parameters:")
print(f"  Amplitude: {A_fit:.2f} ± {A_err:.2f}")
print(f"  Center:    {x0_fit:.2f} ± {x0_err:.2f} nm")
print(f"  Sigma:     {sigma_fit:.2f} ± {sigma_err:.2f} nm")

# Step 4: Calculate FWHM
FWHM = ____
FWHM_err = ____  # Error propagation: FWHM_err = 2.355 * sigma_err
print(f"  FWHM:      {FWHM:.2f} ± {FWHM_err:.2f} nm")

# Step 5: Calculate correlation matrix
correlation = np.zeros((3, 3))
for i in range(3):
    for j in range(3):
        correlation[i, j] = pcov[i, j] / np.sqrt(pcov[i, i] * pcov[j, j])

print("\nCorrelation matrix:")
print(correlation)

# Step 6: Plot the fit and residuals
____
```

Tip

1. `A_guess = np.max(intensity)`
2. `x0_guess = wavelength[np.argmax(intensity)]`
3. Estimate `sigma_guess` as roughly 1/4 of the visible peak width (~15)
4. FWHM = 2.355 * sigma
5. The correlation matrix shows how parameters depend on each other

Solution.

Note

```
# Data
wavelength = wavelength_data
intensity = gaussian_data
intensity_err = gaussian_err

def gaussian(x, A, x0, sigma):
    return A * np.exp(-(x - x0)**2 / (2 * sigma**2))

# Step 1: Initial guesses
A_guess = np.max(intensity)
x0_guess = wavelength[np.argmax(intensity)]
sigma_guess = 15  # Estimated from peak width

p0 = [A_guess, x0_guess, sigma_guess]
print(f"Initial guesses: A={A_guess:.1f}, x0={x0_guess:.1f}, sigma={sigma_guess}")

# Step 2: Fit
popt, pcov = curve_fit(gaussian, wavelength, intensity, sigma=intensity_err,
                        p0=p0, absolute_sigma=True)

# Step 3: Parameters with uncertainties
A_fit, x0_fit, sigma_fit = popt
uncertainties = np.sqrt(np.diag(pcov))
A_err, x0_err, sigma_err = uncertainties

print("\nFitted parameters:")
print(f"  Amplitude: {A_fit:.2f} ± {A_err:.2f}")
print(f"  Center:    {x0_fit:.2f} ± {x0_err:.2f} nm")
print(f"  Sigma:     {sigma_fit:.2f} ± {sigma_err:.2f} nm")

print(f"\nTrue values: A={amplitude_true}, x0={center_true}, sigma={sigma_true}")

# Step 4: FWHM
FWHM = 2.355 * sigma_fit
FWHM_err = 2.355 * sigma_err
print(f"\nFWHM: {FWHM:.2f} ± {FWHM_err:.2f} nm")

# Step 5: Correlation matrix
correlation = np.zeros((3, 3))
for i in range(3):
    for j in range(3):
        correlation[i, j] = pcov[i, j] / np.sqrt(pcov[i, i] * pcov[j, j])

print("\nCorrelation matrix (A, x0, sigma):")
print("       A      x0    sigma")
labels = ['A    ', 'x0   ', 'sigma']
for i, label in enumerate(labels):
    print(f"{label} {correlation[i, 0]:6.3f} {correlation[i, 1]:6.3f} {correlation[i,

# Step 6: Plot
fig, axes = plt.subplots(1, 2, figsize=get_size(16, 7))

# Left: Data and fit
```

```
#| setup: true
#| exercise: ex_5

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

def get_size(w, h):
    return (w/2.54, h/2.54)

# Dataset: Fluorescence lifetime (TCSPC simulation)
np.random.seed(321)
tau_true = 4.2  # nanoseconds
n_photons = 500
photon_arrivals = np.random.exponential(tau_true, n_photons)
# Remove photons arriving after detection window
photon_arrivals = photon_arrivals[photon_arrivals < 30]
```

Exercise 5: Fluorescence Lifetime with MLE (Capstone)

Time-Correlated Single Photon Counting (TCSPC): In fluorescence lifetime measurements, we record the arrival time of individual photons after an excitation pulse. The arrival times follow an exponential distribution.
As discussed in the lecture, the MLE for the lifetime is simply the mean arrival time:

$$\hat{\tau}_{MLE} = \frac{1}{N}\sum_{i=1}^{N} t_i = \bar{t}$$

Your task is to:

1. Analyze the simulated photon arrival data
2. Calculate the lifetime using MLE (mean)
3. Compare with histogram fitting
4. Discuss which method is better and why

Time estimate: 25-30 minutes

```
#| exercise: ex_5

# Simulated photon arrival times
arrival_times = photon_arrivals

print(f"Number of detected photons: {len(arrival_times)}")
print(f"First 10 arrival times: {arrival_times[:10]}")

# Step 1: Calculate lifetime using MLE (just the mean!)
tau_mle = ____
tau_mle_err = ____  # Standard error: tau / sqrt(N)

print(f"\nMLE estimate:   = {tau_mle:.3f} ± {tau_mle_err:.3f} ns")
print(f"True lifetime:   = {tau_true} ns")

# Step 2: Create histogram of arrival times
n_bins = 20
hist_counts, bin_edges = np.histogram(arrival_times, bins=n_bins, range=(0, 25))
bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2
bin_width = bin_edges[1] - bin_edges[0]

# Uncertainties for histogram (Poisson: sqrt(N))
hist_err = np.sqrt(hist_counts)
hist_err[hist_err == 0] = 1  # Avoid division by zero

# Step 3: Fit histogram with exponential
def exponential_decay(t, A, tau):
    return A * np.exp(-t / tau)

# Only fit bins with counts > 0
mask = hist_counts > 0
p0_hist = [np.max(hist_counts), 4.0]

popt_hist, pcov_hist = curve_fit(____)

tau_hist = popt_hist[1]
tau_hist_err = np.sqrt(pcov_hist[1, 1])

print(f"Histogram fit:   = {tau_hist:.3f} ± {tau_hist_err:.3f} ns")

# Step 4: Plot comparison
fig, axes = plt.subplots(1, 2, figsize=get_size(16, 7))

# Left: Histogram with fit
----

# Right: Comparison of methods
----

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

Tip

1. `tau_mle = np.mean(arrival_times)`
2. `tau_mle_err = tau_mle / np.sqrt(len(arrival_times))`
3. For histogram fit: `curve_fit(exponential_decay, bin_centers[mask], hist_counts[mask], sigma=hist_err[mask], p0=p0_hist, absolute_sigma=True)`
4. The MLE method typically gives smaller uncertainties because it uses every photon's exact arrival time, not binned data

Solution.

Note

```
# Simulated photon arrival times
arrival_times = photon_arrivals

print(f"Number of detected photons: {len(arrival_times)}")

# Step 1: MLE estimate (just the mean!)
tau_mle = np.mean(arrival_times)
tau_mle_err = tau_mle / np.sqrt(len(arrival_times))

print(f"\nMLE estimate:   = {tau_mle:.3f} ± {tau_mle_err:.3f} ns")
print(f"True lifetime:   = {tau_true} ns")

# Step 2: Create histogram
n_bins = 20
hist_counts, bin_edges = np.histogram(arrival_times, bins=n_bins, range=(0, 25))
bin_centers = (bin_edges[:-1] + bin_edges[1:]) / 2
bin_width = bin_edges[1] - bin_edges[0]

hist_err = np.sqrt(hist_counts)
hist_err[hist_err == 0] = 1

# Step 3: Fit histogram
def exponential_decay(t, A, tau):
    return A * np.exp(-t / tau)

mask = hist_counts > 0
p0_hist = [np.max(hist_counts), 4.0]

popt_hist, pcov_hist = curve_fit(exponential_decay, bin_centers[mask], hist_counts[mas
                                  sigma=hist_err[mask], p0=p0_hist, absolute_sigma=Tru

tau_hist = popt_hist[1]
tau_hist_err = np.sqrt(pcov_hist[1, 1])

print(f"Histogram fit:   = {tau_hist:.3f} ± {tau_hist_err:.3f} ns")

# Step 4: Plotting
fig, axes = plt.subplots(1, 2, figsize=get_size(16, 7))

# Left: Histogram with fit
axes[0].bar(bin_centers, hist_counts, width=bin_width*0.8, alpha=0.7, label='Data')
axes[0].errorbar(bin_centers, hist_counts, yerr=hist_err, fmt='none', color='k', capsi
t_fine = np.linspace(0, 25, 200)
axes[0].plot(t_fine, exponential_decay(t_fine, *popt_hist), 'r-', lw=2,
             label=f'Fit:   = {tau_hist:.2f} ns')
axes[0].axvline(tau_mle, color='blue', linestyle='--', label=f'MLE:   = {tau_mle:.2f} r
axes[0].set_xlabel('Arrival time (ns)')
axes[0].set_ylabel('Counts')
axes[0].set_title('TCSPC Histogram')
axes[0].legend()

# Right: Comparison bar chart
methods = ['True', 'MLE\n(mean)', 'Histogram\nfit']
```

```
#| setup: true
#| exercise: ex_6

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

def get_size(w, h):
    return (w/2.54, h/2.54)
```

Exercise 6: Challenge - Weighted vs Unweighted Fitting

Challenge: Investigate how varying uncertainties affect the fit results.
Create a dataset where uncertainties increase with x (heteroscedasticity), then compare:
1. Unweighted fit (ignoring uncertainties) 2. Weighted fit (using correct uncertainties) 3. Weighted fit with wrong uncertainties (constant)
Which approach gives the best estimate of the true parameters?
Time estimate: 20-25 minutes (optional challenge)

```
#| exercise: ex_6

# Create data with heteroscedastic errors (uncertainty increases with x)
np.random.seed(999)
x_hetero = np.linspace(1, 10, 15)
slope_true = 2.5
intercept_true = 5.0

# True uncertainties increase with x
sigma_true = 0.5 + 0.3 * x_hetero

# Generate data
y_true = slope_true * x_hetero + intercept_true
y_hetero = y_true + np.random.normal(0, sigma_true)

def linear(x, a, b):
    return a * x + b

# Fit 1: Unweighted
popt1, pcov1 = curve_fit(____)

# Fit 2: Weighted with correct uncertainties
popt2, pcov2 = curve_fit(____)

# Fit 3: Weighted with wrong (constant) uncertainties
sigma_wrong = np.ones_like(x_hetero) * np.mean(sigma_true)
popt3, pcov3 = curve_fit(____)

# Compare results
print("True parameters: slope = {:.2f}, intercept = {:.2f}".format(slope_true, intercept_t
print("\nFit results:")
print("Unweighted:        slope = {:.3f} ± {:.3f}, intercept = {:.3f} ± {:.3f}".format(
    popt1[0], np.sqrt(pcov1[0,0]), popt1[1], np.sqrt(pcov1[1,1])))
print("Weighted (correct): slope = {:.3f} ± {:.3f}, intercept = {:.3f} ± {:.3f}".format(
    popt2[0], np.sqrt(pcov2[0,0]), popt2[1], np.sqrt(pcov2[1,1])))
print("Weighted (wrong):   slope = {:.3f} ± {:.3f}, intercept = {:.3f} ± {:.3f}".format(
    popt3[0], np.sqrt(pcov3[0,0]), popt3[1], np.sqrt(pcov3[1,1])))

# Plot comparison
____
```

Solution.

Note

```
# Create heteroscedastic data
np.random.seed(999)
x_hetero = np.linspace(1, 10, 15)
slope_true = 2.5
intercept_true = 5.0

sigma_true = 0.5 + 0.3 * x_hetero
y_true = slope_true * x_hetero + intercept_true
y_hetero = y_true + np.random.normal(0, sigma_true)

def linear(x, a, b):
    return a * x + b

# Three fits
popt1, pcov1 = curve_fit(linear, x_hetero, y_hetero)
popt2, pcov2 = curve_fit(linear, x_hetero, y_hetero, sigma=sigma_true, absolute_sigma=
sigma_wrong = np.ones_like(x_hetero) * np.mean(sigma_true)
popt3, pcov3 = curve_fit(linear, x_hetero, y_hetero, sigma=sigma_wrong, absolute_sigma

# Results
print("True parameters: slope = {:.2f}, intercept = {:.2f}".format(slope_true, interce
print("\nFit results:")
print("Unweighted:        slope = {:.3f} ± {:.3f}, intercept = {:.3f} ± {:.3f}".format
    popt1[0], np.sqrt(pcov1[0,0]), popt1[1], np.sqrt(pcov1[1,1])))
print("Weighted (correct): slope = {:.3f} ± {:.3f}, intercept = {:.3f} ± {:.3f}".forma
    popt2[0], np.sqrt(pcov2[0,0]), popt2[1], np.sqrt(pcov2[1,1])))
print("Weighted (wrong):   slope = {:.3f} ± {:.3f}, intercept = {:.3f} ± {:.3f}".forma
    popt3[0], np.sqrt(pcov3[0,0]), popt3[1], np.sqrt(pcov3[1,1])))

# Plotting
fig, axes = plt.subplots(1, 2, figsize=get_size(16, 7))

x_fit = np.linspace(0, 11, 100)

# Left: Data and fits
axes[0].errorbar(x_hetero, y_hetero, yerr=sigma_true, fmt='o', capsize=3,
                  label='Data', color='black')
axes[0].plot(x_fit, linear(x_fit, *popt1), 'b--', label='Unweighted', alpha=0.7)
axes[0].plot(x_fit, linear(x_fit, *popt2), 'g-', label='Weighted (correct)', lw=2)
axes[0].plot(x_fit, linear(x_fit, *popt3), 'r:', label='Weighted (wrong)', alpha=0.7)
axes[0].plot(x_fit, linear(x_fit, slope_true, intercept_true), 'k--',
             label='True', alpha=0.5)
axes[0].set_xlabel('x')
axes[0].set_ylabel('y')
axes[0].legend()
axes[0].set_title('Comparison of Fitting Methods')

# Right: Error bars visualization
axes[1].errorbar(x_hetero, sigma_true, fmt='o-', label='True uncertainties')
axes[1].axhline(np.mean(sigma_true), color='r', linestyle='--',
                label=f'Mean  = {np.mean(sigma_true):.2f}')
axes[1].set_xlabel('x')
axes[1].set_ylabel('Uncertainty  ')
```

Summary

1.
2.
3.
4.
5.
6.

Key Takeaways

-
-
-
-
-
-