Curve fitting

Curve fitting is a fundamental skill in experimental physics that allows us to extract physical parameters from measured data. In this lecture, we'll explore how to apply the least-squares method to fit a quadratic function with three parameters to experimental data. It's worth noting that this approach can be applied to more complex functions or even simpler linear models.

Before diving into the fitting process, it's essential to consider how to best estimate your model parameters. In some cases, you may be able to derive explicit estimators for the parameters, which can simplify the fitting procedure. Therefore, it's advisable to carefully consider your approach before beginning the actual fitting process.

For those who want to delve deeper into this subject, you might find it interesting to explore concepts like maximum likelihood estimation. This method offers an alternative approach to parameter estimation and can provide valuable insights in certain scenarios.

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

import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
plt.rcParams.update({'font.size': 18})
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# default values for plotting
plt.rcParams.update({'font.size': 12,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 11,
                     '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))
```

```
<IPython.core.display.HTML object>
```

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

import pandas as pd

data = {
    'x': [0.000000000000000000e+00, 1.111111111111111049e-01, 2.222222222222222099e-01, 3.333
    'y': [9.916839204057067425e-01, 1.183667840440161712e+00, 1.310862148961057017e+00, 1.193
    'error': [5.332818799198481979e-02, 5.339559646742838422e-02, 5.366783326382672248e-02, 5
}
df = pd.DataFrame(data)

x_data=df.x.values
y_data=df.y.values
err=df.error.values
```

Idea

In experimental physics, we often collect data points to understand the underlying physical phenomena. This process involves fitting a mathematical model to the experimental data.

The data typically comes as a series of N paired points:

x-data	y-data
x_1	y_1
x_2	y_2
…	…

x-data	y-data
x_N	y_N

Each point $\{x_i, y_i\}$ may represent the result of multiple independent measurements. For instance, y_1 could be the mean of n repeated measurements $y_{1,j}$ at position x_1:

$$y_1 = \frac{1}{n} \sum_{j=1}^{n} y_{1,j}$$

When these individual readings have a **true (underlying) standard deviation** σ, the sum of all n measurements has a variance of $n\sigma^2$ and a standard deviation of $\sqrt{n}\sigma$. Consequently, the mean value has an associated error known as the Standard Error of the Mean (SEOM):

$$\sigma_{SEOM} = \frac{\sigma}{\sqrt{n}}$$

This SEOM is crucial in physics measurements—it tells us how precisely we know the mean value.

In practice, we don't know the true σ, so we must **estimate it from our data**. The estimated variance s_1^2 for the measurements at point x_1 is calculated as:

$$s_1^2 = \frac{1}{n-1} \sum_{j=1}^{n} (y_{1,j} - y_1)^2$$

This is the **sample variance**, which uses $n - 1$ in the denominator (Bessel's correction) instead of n. The factor $n - 1$ accounts for the fact that we estimate the mean y_1 from the same data, which reduces our degrees of freedom by one. The estimated standard deviation is then $s_1 = \sqrt{s_1^2}$.

For large n, the estimate s_1 approaches the true value σ. The estimated standard error of the mean becomes:

$$s_{SEOM} = \frac{s_1}{\sqrt{n}}$$

This statistical framework forms the basis for analyzing experimental data and fitting mathematical models to understand the underlying physics.

Least squares

In experimental physics, we often collect data points to understand the underlying physical phenomena. To make sense of this data, we fit a mathematical model to it. One common method for fitting data is the least squares method.

Why use least squares fitting?

The goal of least squares fitting is to find the set of parameters for our model that best describes the data. This is done by minimizing the differences (or residuals) between the observed data points and the model's predictions.

Gaussian uncertainty and probability

When we take measurements, there is always some uncertainty. Often, this uncertainty can be modeled using a Gaussian (normal) distribution. This distribution is characterized by its mean (average value) and standard deviation (a measure of the spread of the data).

If we describe our data with a model function, which delivers a function value $f(x_i, \mathbf{a})$ for a set of parameters $\mathbf{a} = (a_1, a_2, \dots, a_M)$ at the position x_i, the Gaussian uncertainty dictates a probability of finding a data value y_i:

$$p_{y_i} = \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - f(x_i, \mathbf{a}))^2}{2\sigma_i^2}\right) \tag{1}$$

Here, σ_i represents the uncertainty in the measurement y_i.

Combining probabilities for multiple data points

To understand how well our model fits all the data points, we need to consider the combined probability of observing all the data points. This is done by multiplying the individual probabilities:

$$p(y_1, \dots, y_N) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(y_i - f(x_i, \mathbf{a}))^2}{2\sigma_i^2}\right) \tag{2}$$

Maximizing the joint probability

The best fit of the model to the data is achieved when this joint probability is maximized. To simplify the calculations, we take the logarithm of the joint probability:

$$\ln(p(y_1, \dots, y_N)) = -\frac{1}{2}\sum_{i=1}^{N}\left(\frac{y_i - f(x_i, \mathbf{a})}{\sigma_i}\right)^2 - \sum_{i=1}^{N}\ln\left(\sigma_i\sqrt{2\pi}\right) \tag{3}$$

The first term on the right side (except the factor 1/2) is the least squared deviation, also known as χ^2:

$$\chi^2 = \sum_{i=1}^{N}\left(\frac{y_i - f(x_i, \mathbf{a})}{\sigma_i}\right)^2 \tag{4}$$

The second term is just a constant value given by the uncertainties of our experimental data.

Maximum Likelihood Estimation (MLE)

What we have just derived is actually a powerful statistical method called **Maximum Likelihood Estimation**. Let's formalize this connection.

The Likelihood Function

The *likelihood function* $\mathcal{L}(\mathbf{a})$ is defined as the probability of observing our data, given the model parameters $\mathbf{a}$:

$$\mathcal{L}(\mathbf{a}) = p(y_1, y_2, \dots, y_N|\mathbf{a}) = \prod_{i=1}^{N} p(y_i|\mathbf{a})$$

Note the subtle but important shift in perspective: we treat the *data as fixed* and the *parameters as variable*. We ask: "Which parameters make our observed data most probable?"

The Log-Likelihood

In practice, we work with the *log-likelihood* $\ell(\mathbf{a}) = \ln \mathcal{L}(\mathbf{a})$ for two reasons:

1. **Numerical stability**: Products of many small probabilities can underflow; sums of logarithms don't
2. **Mathematical convenience**: Products become sums, making derivatives easier

Connection to Least Squares

For Gaussian-distributed measurements, the log-likelihood is:

$$\ell(\mathbf{a}) = -\frac{1}{2}\chi^2 - \sum_{i=1}^{N}\ln(\sigma_i\sqrt{2\pi})$$

Since the second term is constant (independent of $\mathbf{a}$), maximizing the log-likelihood is equivalent to **minimizing** χ^2:

$$\underset{\mathbf{a}}{\operatorname{argmax}}\, \ell(\mathbf{a}) = \underset{\mathbf{a}}{\operatorname{argmin}}\, \chi^2(\mathbf{a})$$

This is a profound result: **least squares fitting is the maximum likelihood estimator for Gaussian errors**.
Why MLE Matters

1. **Theoretical foundation**: MLE provides rigorous justification for the least squares method
2. **Optimal properties**: Under mild conditions, MLE estimators are:
   - *Consistent*: converge to true values as $N \to \infty$
   - *Efficient*: achieve the lowest possible variance
   - *Asymptotically normal*: uncertainties follow Gaussian statistics for large N
3. **Generalization**: MLE works for any probability distribution, not just Gaussian:
   - **Poisson statistics**: counting experiments (radioactive decay, photon counting)
   - **Binomial statistics**: success/failure experiments
   - **Custom distributions**: any physical situation with known error distribution

Example: Poisson Likelihood
For counting experiments where y_i follows a Poisson distribution with expected value $\mu_i = f(x_i, \mathbf{a})$:

$$\mathcal{L}(\mathbf{a}) = \prod_{i=1}^{N} \frac{\mu_i^{y_i} e^{-\mu_i}}{y_i!}$$

This leads to a different χ^2-like quantity that must be minimized numerically. SciPy and other fitting tools can handle this via custom likelihood functions.
Experimental Example: Fluorescence Lifetime Measurement
A beautiful application of MLE is **Time-Correlated Single Photon Counting (TCSPC)**, used to measure excited state lifetimes of fluorescent molecules.
The Experiment:

1. A short laser pulse excites dye molecules to an excited state
2. Each molecule decays back to the ground state by emitting a photon
3. We record the arrival time t_i of each detected photon relative to the excitation pulse
4. The decay follows an exponential distribution with lifetime τ

Experimental procedure:

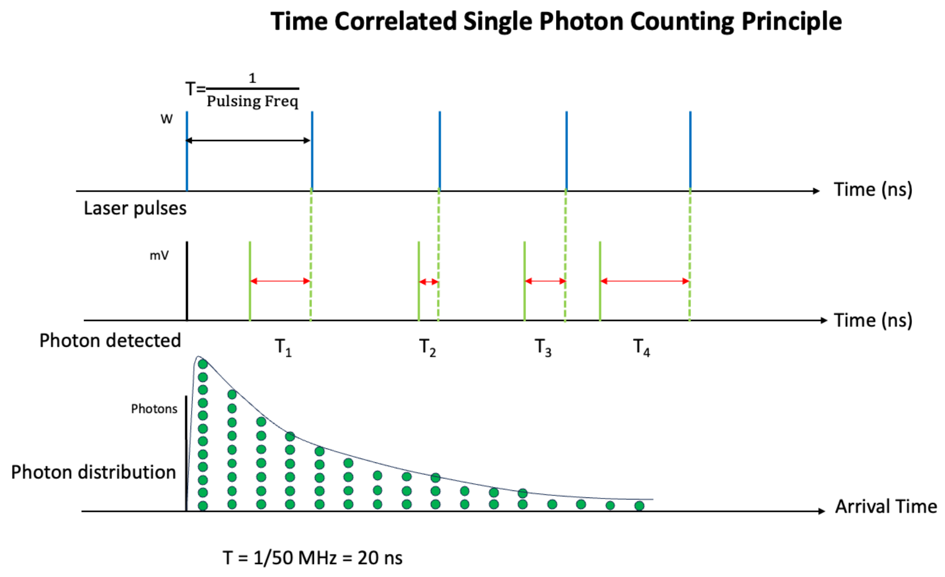

Figure 1: Time-Correlated Single Photon Counting (TCSPC) procedure for measuring fluorescence lifetimes.

The Physics:
The probability density for a photon arriving at time t is:

$$p(t|\tau) = \frac{1}{\tau} e^{-t/\tau}$$

Two Approaches to Extract τ:
Approach 1: Histogram Fitting (Least Squares)

- Bin the arrival times into a histogram
- Fit the histogram to an exponential using least squares
- Problem: Binning loses information, and bin counts follow Poisson statistics (not Gaussian)

Approach 2: Direct MLE
Write the likelihood for N photon arrival times $\{t_1, t_2, \ldots, t_N\}$:

$$\mathcal{L}(\tau) = \prod_{i=1}^{N} \frac{1}{\tau} e^{-t_i/\tau} = \frac{1}{\tau^N} \exp\left(-\frac{1}{\tau}\sum_{i=1}^{N} t_i\right)$$

The log-likelihood is:

$$\ell(\tau) = -N\ln(\tau) - \frac{1}{\tau}\sum_{i=1}^{N} t_i$$

To find the maximum, take the derivative and set it to zero:

$$\frac{d\ell}{d\tau} = -\frac{N}{\tau} + \frac{1}{\tau^2}\sum_{i=1}^{N} t_i = 0$$

Solving for τ:

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

The MLE for the lifetime is simply the mean arrival time!
Why MLE Wins Here:

1. **No binning required**: Uses every photon's exact arrival time
2. **Statistically optimal**: Achieves the lowest possible uncertainty
3. **Simple formula**: Just compute the mean—no iterative fitting needed
4. **Works with few photons**: Histogram fitting needs many counts per bin; MLE works even with sparse data

The uncertainty on the MLE estimate is:

$$\sigma_{\hat{\tau}} = \frac{\tau}{\sqrt{N}}$$

This example shows the power of MLE: by using the correct probability distribution, we obtain a simple, optimal estimator without any curve fitting at all!
Summary
The least squares method you're learning is not just a convenient recipe—it's the statistically optimal approach for Gaussian measurement errors. Understanding its foundation in MLE will help you extend these ideas to more complex situations in your future physics career.

Data

Let's have a look at the meaning of this equation. Let's assume we measure the trajectory of a ball that has been thrown at an angle α with an initial velocity v_0. We have collected data points by measuring the height of the ball above the ground at equally spaced distances from the throwing person.

The table above shows the measured data points y_i at the position x_i with the associated uncertainties σ_i.

We can plot the data and expect, of course, a parabola. Therefore, we model our experimental data with a parabola like

$$y = ax^2 + bx + c \tag{5}$$

where the parameter a must be negative since the parabola is inverted.

I have created an interactive plot with an interact widget, as this allows you to play around with the parameters. The value of χ^2 is also included in the legend, so you can get an impression of how good your fit of the data is.

```
//| echo: false
viewof aSlider = Inputs.range([-4, 0], { label: "a", step: 0.01, value: -1.7 });
viewof bSlider = Inputs.range([-2, 2], { label: "b", step: 0.01, value: 1.3 });
viewof cSlider = Inputs.range([-2, 2], { label: "c", step: 0.01, value: 1.0 });
```

```
//| echo: false
//| fig-align: center
filtered = transpose(data);
// Create the plot

xValues = Array.from({ length: 100 }, (_, i) => i / 100);
parabolaData = xValues.map(x => ({ x, y: parabola(x, aSlider, bSlider, cSlider) }));


parabola = (x, a, b, c) => a * x**2 + b * x + c

calculateChiSquared = (data, a, b, c) => {
  let chisq = 0
  let x= data.map(d => d.x)
  let y= data.map(d => d.y)
  let err= data.map(d => d.error)
  for (let i = 0; i < x.length; i++) {
    let y_model = parabola(x[i], a, b, c)
    chisq += ((y[i] - y_model) / err[i])**2
  }
  return chisq
}

chisq = calculateChiSquared(filtered, aSlider, bSlider, cSlider)
```

```
Plot.plot({
  marks: [
    Plot.dot(filtered, { x: "x", y: "y" }),
    Plot.ruleY(filtered, { x: "x", y1: d => d.y - d.error, y2: d => d.y + d.error }),
    Plot.line(parabolaData, { x: "x", y: "y" }),
    Plot.text([{ x: 0.8, y: 1.5, label: ` ²: ${chisq.toFixed(2)}` }], {
          x: "x",
          y: "y",
          text: "label",
          dy: -10, // Adjust vertical position if needed
          fill: "black", // Set text color
          fontSize: 16
        }),
    Plot.frame()
  ],
  x: {
    label: "X Axis",
    labelAnchor: "center",
    labelOffset: 35,
    grid: true,
    tickFormat: ".2f", // Format ticks to 2 decimal places
    domain: [0, 1]
  },
  y: {
    label: "Y Axis",
    grid: true,
    tickFormat: ".2f", // Format ticks to 2 decimal places
    labelAnchor: "center",  // Center the label on its axis
    labelAngle: -90,
    labelOffset: 60,
    domain: [0, 2],
  },
  width: 400,
  height: 400,
  marginLeft: 100,
  marginBottom: 40,
  style: {
    fontSize: "14px",           // This sets the base font size
    "axis.label": {
      fontSize: "18px",         // This sets the font size for axis labels
      fontWeight: "bold"        // Optionally make it bold
    },
    "axis.tick": {
      fontSize: "14px"         // This sets the font size for tick labels
    }
  },
})
```

```
#| autorun: false
#| fig-align: center
import numpy as np
import io
import pandas as pd
import matplotlib.pyplot as plt
plt.figure(figsize=get_size(8,8))
plt.errorbar(df.x,df.y,yerr=df.error,marker='o',fmt="none",color='k')

plt.scatter(df.x,df.y,marker='o',color='k')
plt.xlabel('x- position')
plt.ylabel('y- position')
plt.tight_layout()
plt.show()
```

We have that troubling point at the right edge with a large uncertainty. This could represent a situation where the measurement conditions were worse (e.g., the ball was harder to track at larger distances, or there was more environmental interference). However, since the value of χ^2 divides the deviation by the uncertainty σ_i, the weight for this point overall in the χ^2 is smaller than for the other points. This is exactly how weighted fitting should work: less reliable measurements contribute less to the fit.

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i, \mathbf{a})}{\sigma_i} \right)^2 \tag{6}$$

You may simply check the effect by changing the uncertainty of the last data points in the error array.

Least square fitting

To find the best fit of the model to the experimental data, we use the least squares method. This method minimizes the sum of the squared differences between the observed data points and the model's predictions.

Mathematically, we achieve this by minimizing the least squares, i.e., finding the parameters **a** that minimize the following expression:

$$\frac{\partial \chi^2}{\partial a_k} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \frac{\partial f(x_i, \mathbf{a})}{\partial a_k} [y_i - f(x_i, \mathbf{a})] = 0 \tag{7}$$

This equation must be satisfied for each parameter a_k in our model.

This kind of least squares minimization is done by fitting software using different types of algorithms.

Derivation: Linear Regression Parameters

For simple cases, we can derive explicit formulas for the fit parameters. Let's work through the derivation for **weighted linear regression** with the model:

$$f(x) = a + bx$$

where a is the intercept and b is the slope.

Step 1: Write out the χ^2 expression

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - a - bx_i)^2}{\sigma_i^2}$$

Step 2: Take partial derivatives and set to zero

For parameter a:

$$\frac{\partial \chi^2}{\partial a} = -2 \sum_{i=1}^{N} \frac{y_i - a - bx_i}{\sigma_i^2} = 0$$

For parameter b:

$$\frac{\partial \chi^2}{\partial b} = -2 \sum_{i=1}^{N} \frac{x_i(y_i - a - bx_i)}{\sigma_i^2} = 0$$

Step 3: Define convenient sums

To simplify notation, we define weighted sums:

$$S = \sum_{i=1}^{N} \frac{1}{\sigma_i^2}, \quad S_x = \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2}, \quad S_y = \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2}$$

$$S_{xx} = \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2}, \quad S_{xy} = \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2}$$

Step 4: Rewrite as a system of linear equations (normal equations)

$$aS + bS_x = S_y$$

$$aS_x + bS_{xx} = S_{xy}$$

Step 5: Solve for a and b
Define the determinant: $\Delta = S \cdot S_{xx} - S_x^2$
The solutions are:

$$a = \frac{S_{xx} \cdot S_y - S_x \cdot S_{xy}}{\Delta}$$

$$b = \frac{S \cdot S_{xy} - S_x \cdot S_y}{\Delta}$$

Step 6: Parameter uncertainties
The covariance matrix elements can also be derived analytically:

$$\sigma_a^2 = \frac{S_{xx}}{\Delta}, \quad \sigma_b^2 = \frac{S}{\Delta}, \quad \mathrm{cov}(a,b) = -\frac{S_x}{\Delta}$$

Special case: Unweighted regression ($\sigma_i = \sigma$ for all i)
When all uncertainties are equal, the σ^2 cancels out and we get the familiar formulas:

$$b = \frac{N \sum x_i y_i - \sum x_i \sum y_i}{N \sum x_i^2 - (\sum x_i)^2}$$

$$a = \bar{y} - b\bar{x}$$

where $\bar{x}$ and $\bar{y}$ are the sample means.
Why this matters
This derivation shows that for linear models, we can obtain **closed-form solutions** without iterative optimization. For more complex models (like our parabola), numerical algorithms are needed, which is why we use `curve_fit`.

Fitting with SciPy

Let's do some fitting using the `SciPy` library, which is a powerful tool for scientific computing in Python. We will use the `curve_fit` method from the `optimize` sub-module of `SciPy`.

First, we need to define the model function we would like to fit to the data. In this case, we will use our parabola function:

```
#| autorun: false
def parabola(x, a, b, c):
    return a * x**2 + b * x + c
```

Next, we need to provide initial guesses for the parameters. These initial guesses help the fitting algorithm start the search for the optimal parameters:

```
#| autorun: false
init_guess = [-1, 1, 1]
```

We then call the `curve_fit` function to perform the fitting:

```
#| autorun: false
fit = curve_fit(parabola, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
```

`curve_fit` Function

The `curve_fit` function is used to fit a model function to data. It finds the optimal parameters for the model function that minimize the sum of the squared residuals between the observed data and the model's predictions.

Parameters

1. **`parabola`**:
   - This is the model function that you want to fit to the data. In this case, `parabola` is a function that represents a quadratic equation of the form (y = ax^2 + bx + c).
2. **`x_data`**:
   - This is the array of independent variable data points (the x-values).
3. **`y_data`**:
   - This is the array of dependent variable data points (the y-values).
4. **`sigma=err`**:
   - This parameter specifies the uncertainties (standard deviations) of the y-data points. The `err` array contains the uncertainties for each y-data point. These uncertainties are used to weight the residuals in the least squares optimization.
5. **`p0=init_guess`**:
   - This parameter provides the initial guesses for the parameters of the model function. The `init_guess` array contains the initial guesses for the parameters (a), (b), and (c). Providing good initial guesses can help the optimization algorithm converge more quickly and accurately.
6. **`absolute_sigma=True`**:

- This parameter indicates whether the provided sigma values are absolute uncertainties. If absolute_sigma is set to True, the sigma values are treated as absolute uncertainties. If absolute_sigma is set to False, the sigma values are treated as relative uncertainties, and the covariance matrix of the parameters will be scaled accordingly.

When to use each setting:

- Use absolute_sigma=True when your uncertainties are well-known from calibration, repeated measurements, or manufacturer specifications.
- Use absolute_sigma=False when your uncertainties are rough estimates. In this case, the covariance matrix will be scaled by the reduced χ^2, which can compensate for over- or under-estimated uncertainties.

Return Value

The curve_fit function returns two values:

1. popt:
   - An array containing the optimal values for the parameters of the model function that minimize the sum of the squared residuals.
2. pcov:
   - The covariance matrix of the optimal parameters. The diagonal elements of this matrix provide the variances of the parameter estimates, and the off-diagonal elements provide the covariances between the parameter estimates.

The fit variable contains the results of the fitting process. It is composed of various results, which we can split into the fitted parameters and the covariance matrix:

```
#| autorun: false
ans, cov = fit
fit_a, fit_b, fit_c = ans
```

The ans variable contains the fitted parameters fit_a, fit_b, and fit_c, while the cov variable contains the covariance matrix. Let's have a look at the fit and the χ^2 value first:

```
#| autorun: false
fit_a, fit_b, fit_c
```

We can then plot the fitted curve along with the original data points and the χ^2 value:

```
#| autorun: false
plt.figure(figsize=get_size(8,8))
chisq = (((y_data - parabola(x_data, fit_a, fit_b, fit_c)) / err)**2).sum()
plt.plot(x_data, parabola(x_data, fit_a, fit_b, fit_c), label=r'$\chi^2$={0:6.3f}'.format(chi
plt.errorbar(x_data, y_data, yerr=err, marker='o', fmt="none", color='k')
plt.scatter(x_data, y_data, marker='.', color='k')
plt.xlabel('x-position')
plt.ylabel('y-position')
plt.legend()
plt.tight_layout()
plt.show()
```

χ^2 Value and Reduced χ^2

The value of χ^2 gives you a measure of the quality of the fit. We can judge the quality by calculating the expectation value of χ^2:

$$\langle\chi^2\rangle = \sum_{i=1}^{N} \frac{\langle(y_i - f(x_i, \mathbf{a}))^2\rangle}{\sigma_i^2} = \sum_{i=1}^{N} \frac{\sigma_i^2}{\sigma_i^2} = N \tag{8}$$

However, this is only approximately correct. More precisely, we need to account for the **degrees of freedom** $\nu = N - M$, where N is the number of data points and M is the number of fitted parameters. In our case, with 10 data points and 3 parameters (a, b, c), we have $\nu = 10 - 3 = 7$ degrees of freedom.

The **reduced chi-squared** is defined as:

$$\chi_\nu^2 = \frac{\chi^2}{\nu} = \frac{\chi^2}{N - M} \tag{9}$$

The expectation value of the reduced chi-squared is:

$$\langle\chi_\nu^2\rangle = 1 \tag{10}$$

This gives us clear criteria for judging fit quality:

- $\chi_\nu^2 \approx 1$ means the fit is good and uncertainties are correctly estimated.
- $\chi_\nu^2 \gg 1$ means the fit is bad (wrong model or underestimated uncertainties).
- $\chi_\nu^2 \ll 1$ means the uncertainties are overestimated.

```
#| autorun: false
# Calculate reduced chi-squared
n_params = 3  # number of parameters (a, b, c)
dof = len(x_data) - n_params  # degrees of freedom
chi2_reduced = chisq / dof
print(f" ² = {chisq:.2f}")
print(f"Degrees of freedom = {dof}")
print(f"Reduced  ² = {chi2_reduced:.2f}")
```

Interpreting Reduced χ^2 in Practice

In real experiments, you will rarely obtain $\chi^2_\nu = 1.00$ exactly. Here's a practical guide for interpretation:

χ^2_ν	Interpretation
< 0.5	Uncertainties likely overestimated
$0.5 - 2.0$	Generally acceptable
≈ 1	Ideal — good fit with correct uncertainties
$2.0 - 3.0$	Borderline — investigate further
> 3.0	Poor fit — wrong model or underestimated uncertainties

Example: What does $\chi^2_\nu = 2$ mean?
A value of $\chi^2_\nu = 2$ is borderline but not necessarily bad. With $\nu = 7$ degrees of freedom, the probability of obtaining $\chi^2 \geq 14$ by chance is about 5%. This could indicate:

1. **Slightly underestimated uncertainties** — error bars might be ~30% too small (since $\sqrt{2} \approx 1.4$)
2. **Model imperfections** — the parabola might not capture all the physics
3. **Outliers** — one or two data points pulling the fit off
4. **Statistical fluctuation** — 5% of experiments will show this even with correct model and uncertainties

For a first-semester lab course, $\chi^2_\nu = 2$ is acceptable. You wouldn't reject the model based on this value alone, but you should examine the residuals for systematic patterns.

It is really important to have a good estimate of the uncertainties and to include them in your fit. If you include the uncertainties in your fit, it is called a **weighted fit**. If you don't include the uncertainties (meaning you keep them constant), it is called an **unweighted fit**.

For our fit above, we obtain a reduced χ^2 around 2, which is borderline acceptable. The model fits reasonably well, though there may be slight underestimation of uncertainties or minor model imperfections.

Residuals

Another way to assess the quality of the fit is by looking at the residuals. There are several types of residuals, each useful in different contexts:

1. Absolute residuals — the raw deviation of data from the model:

$$r_i = y_i - f(x_i, \mathbf{a}) \tag{11}$$

2. Normalized (weighted) residuals — scaled by the measurement uncertainty:

$$r_i^{\text{norm}} = \frac{y_i - f(x_i, \mathbf{a})}{\sigma_i} \tag{12}$$

For a good fit with correct uncertainties, these should be approximately standard normal distributed (mean 0, standard deviation 1).

3. Relative (percentage) residuals — useful when comparing across different scales:

$$r_i^{\text{rel}} = 100 \times \frac{y_i - f(x_i, \mathbf{a})}{y_i} \quad [\%] \tag{13}$$

Importance of Residuals

Residuals are important because they provide insight into how well the model fits the data. If the residuals show only statistical fluctuations around zero, then the fit and likely also the model are good. However, if there are systematic patterns in the residuals, it may indicate that the model is not adequately capturing the underlying relationship in the data.

Visualizing Residuals

Let's visualize the different types of residuals to better understand their distribution.

```
#| autorun: false
fig, axes = plt.subplots(1, 3, figsize=get_size(24, 8))

# Calculate residuals
model_values = parabola(x_data, fit_a, fit_b, fit_c)
abs_residuals = y_data - model_values
norm_residuals = abs_residuals / err
rel_residuals = 100 * abs_residuals / y_data

# Absolute residuals
```

```python
axes[0].scatter(x_data, abs_residuals, marker='o', color='k')
axes[0].axhline(y=0, color='r', linestyle='--', alpha=0.5)
axes[0].set_xlabel('x-position')
axes[0].set_ylabel('Absolute residuals')
axes[0].set_title('Absolute Residuals')

# Normalized residuals
axes[1].scatter(x_data, norm_residuals, marker='o', color='k')
axes[1].axhline(y=0, color='r', linestyle='--', alpha=0.5)
axes[1].axhline(y=1, color='gray', linestyle=':', alpha=0.5)
axes[1].axhline(y=-1, color='gray', linestyle=':', alpha=0.5)
axes[1].set_xlabel('x-position')
axes[1].set_ylabel(r'Normalized residuals ($r_i / \sigma_i$)')
axes[1].set_title('Normalized Residuals')

# Relative residuals
axes[2].scatter(x_data, rel_residuals, marker='o', color='k')
axes[2].axhline(y=0, color='r', linestyle='--', alpha=0.5)
axes[2].set_xlabel('x-position')
axes[2].set_ylabel('Relative residuals [%]')
axes[2].set_title('Relative Residuals')

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

The **normalized residuals** are particularly useful: for a good fit, most points should fall within ±1 (gray dashed lines), and approximately 95% should fall within ±2.

Common Patterns in Residuals

Random Fluctuations Around Zero:

- If the residuals are randomly scattered around zero, it suggests that the model is a good fit for the data.

Systematic Patterns:

- If the residuals show a systematic pattern (e.g., a trend or periodicity), it may indicate that the model is not capturing some aspect of the data. This could suggest the need for a more complex model.

Increasing or Decreasing Trends:

- If the residuals increase or decrease with x, it may indicate heteroscedasticity (non-constant variance) or that a different functional form is needed.

Covariance Matrix

In the previous sections, we discussed how to fit a model to experimental data and assess the quality of the fit using residuals. Now, let's take a closer look at the uncertainties in the fit parameters and how they are related to each other. This is where the covariance matrix comes into play.

Purpose of the Covariance Matrix

The covariance matrix provides important information about the uncertainties in the fit parameters and how these uncertainties are related to each other. It helps us understand the precision of the parameter estimates and whether the parameters are independent or correlated.

Understanding Covariance

Covariance is a measure of how much two random variables change together. If the covariance between two variables is positive, it means that they tend to increase or decrease together. If the covariance is negative, it means that one variable tends to increase when the other decreases. If the covariance is zero, it means that the variables are independent.

Covariance Matrix in Curve Fitting

When we fit a model to data, we obtain estimates for the parameters of the model. These estimates have uncertainties due to the measurement errors in the data. The covariance matrix quantifies these uncertainties and the relationships between them.

For a model with three parameters (a, b, c), the covariance matrix is a 3×3 matrix that looks like this:

$$\text{cov}(p_i, p_j) = \begin{bmatrix} \sigma_{aa}^2 & \sigma_{ab}^2 & \sigma_{ac}^2 \\ \sigma_{ba}^2 & \sigma_{bb}^2 & \sigma_{bc}^2 \\ \sigma_{ca}^2 & \sigma_{cb}^2 & \sigma_{cc}^2 \end{bmatrix} \tag{14}$$

The diagonal elements provide the variances (squared uncertainties) of the fit parameters, while the off-diagonal elements describe the covariances between the parameters.

Example

Let's calculate the covariance matrix for our fitted model and interpret the results.

```
#| autorun: false
# Calculate the covariance matrix
fit = curve_fit(parabola, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
ans, cov = fit

# Print the covariance matrix
print("Covariance matrix:")
print(cov)
```

Extracting Parameter Uncertainties

The most practical use of the covariance matrix is to extract the uncertainties of the fitted parameters. The diagonal elements give us the variances, so we take the square root to get the standard deviations:

```
#| autorun: false
# Extract uncertainties from the covariance matrix
uncertainties = np.sqrt(np.diag(cov))

# Print parameters with their uncertainties
print("Fitted parameters with uncertainties:")
print(f"  a = {ans[0]:.4f} ± {uncertainties[0]:.4f}")
print(f"  b = {ans[1]:.4f} ± {uncertainties[1]:.4f}")
print(f"  c = {ans[2]:.4f} ± {uncertainties[2]:.4f}")
```

Interpreting the Covariance Matrix

The covariance matrix provides valuable information about the uncertainties in the fit parameters:

- **Diagonal Elements**: The diagonal elements represent the variances of the parameters. The square root of these values gives the standard deviations (uncertainties) of the parameters.
- **Off-Diagonal Elements**: The off-diagonal elements represent the covariances between the parameters. If these values are large, it indicates that the parameters are correlated.

Generating Synthetic Data

To better understand the covariance matrix, let's generate synthetic data and fit the model to each dataset. This will help us visualize the uncertainties in the parameters.

```
#| autorun: false
def generate_synthetic_data(x, a, b, c, uncertainties):
    """
    Generate synthetic data with Gaussian noise.

    Parameters:
    - x: array of x values
    - a, b, c: true model parameters
    - uncertainties: array of measurement uncertainties for each point

    Returns:
    - y values with added Gaussian noise according to uncertainties
    """
    y_true = a * x**2 + b * x + c
    noise = np.array([np.random.normal(0, sigma) for sigma in uncertainties])
    return y_true + noise
```

```
#| autorun: false
x = np.linspace(0, 1, 10)
ym = np.zeros(10)
plt.figure(figsize=get_size(8, 8))

# Use the same uncertainties as our original data
synthetic_err = err

for _ in range(10):
    y = generate_synthetic_data(x, -2.52, 1.6971755, 1, synthetic_err)
    ym += y
    p, cov_synth = curve_fit(parabola, x, y, sigma=synthetic_err, p0=init_guess, absolute_si
    plt.scatter(x, y, color='k', alpha=0.1)
    xf = np.linspace(0, 1, 100)
    plt.plot(xf, parabola(xf, p[0], p[1], p[2]), alpha=0.2)

plt.scatter(x, ym / 10, color='b', label='Mean of synthetic data')
plt.xlabel('x-position')
plt.ylabel('y-position')
plt.legend()
plt.tight_layout()
plt.show()
```

Each gray curve represents a fit to one synthetic dataset. The spread of these curves visualizes the uncertainty in our model parameters.

Correlation Matrix

To better understand the relationships between the parameters, we can normalize the covariance matrix to obtain the correlation matrix. The correlation matrix has values between -1 and 1, where 1 indicates perfect positive correlation, -1 indicates perfect negative correlation, and 0 indicates no correlation.

```
#| autorun: false
# Calculate the correlation matrix
s = np.diag(cov)
R = np.zeros([3, 3])
for i in range(3):
    for j in range(3):
        R[i, j] = cov[i, j] / np.sqrt(s[i] * s[j])

# Print the correlation matrix
print("Correlation matrix:")
print(R)
```

Visualizing the Covariance and Correlation

Let's visualize the covariance and correlation between the parameters using scatter plots.

```
#| autorun: false
# Generate synthetic data and fit the model
a_vals = []
b_vals = []
c_vals = []
x = np.linspace(0, 1, 10)
for _ in range(100):
    y = generate_synthetic_data(x, -2.52, 1.6971755, 1, err)
    p, cov_iter = curve_fit(parabola, x, y, sigma=err, p0=init_guess, absolute_sigma=True)
    a_vals.append(p[0])
    b_vals.append(p[1])
    c_vals.append(p[2])

# Plot the correlation between parameters a and b
plt.figure(figsize=get_size(8, 8))
plt.scatter(a_vals, b_vals, alpha=0.2)
plt.xlabel('Parameter a')
plt.ylabel('Parameter b')
plt.title('Correlation between Parameters a and b')
plt.tight_layout()
plt.show()
```

By examining the covariance and correlation matrices, we can gain a deeper understanding of the uncertainties in the fit parameters and how they are related to each other.

Improving the Model

If we find that the parameters are highly correlated, we might want to find a better model containing more independent parameters. For example, we can write down a different model:

$$y = a(x - b)^2 + c \tag{15}$$

This model also contains three parameters, but the parameter b directly refers to the maximum of our parabola, while the parameter a denotes its curvature.

```
#| autorun: false
def newmodel(x, a, b, c):
    return a * (x - b)**2 + c
```

```
#| autorun: false
fit = curve_fit(newmodel, x_data, y_data, sigma=err, p0=init_guess, absolute_sigma=True)
```

```
#| autorun: false
ans, cov = fit
```

```
#| autorun: false
s = np.diag(cov)
R = np.zeros([3, 3])
for i in range(3):
    for j in range(3):
        R[i, j] = cov[i, j] / np.sqrt(s[i] * s[j])
```

We see from the covariance matrix that the new model has a smaller correlation of the parameters with each other.

```
#| autorun: false
print(cov)
```

This is also expressed by our correlation matrix.

```
#| autorun: false
print(R)
```

By examining the covariance and correlation matrices, we can gain valuable insights into the uncertainties in the fit parameters and how to improve our model.

Summary

In this lecture, we covered the essential concepts of curve fitting for experimental physics:

Key Concepts

1. **Least Squares Method**: We minimize the sum of squared deviations between data and model, weighted by measurement uncertainties:

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i, \mathbf{a})}{\sigma_i} \right)^2$$

2. **Reduced Chi-Squared**: The proper metric for fit quality is $\chi^2_\nu = \chi^2/(N-M)$, where N is the number of data points and M is the number of parameters. A good fit has $\chi^2_\nu \approx 1$.
3. **Residuals**: Always examine residuals after fitting. Look for:
   - Random scatter around zero (good)
   - Systematic patterns (suggests wrong model)
   - Trends with x (suggests missing physics)
4. **Covariance Matrix**: Provides uncertainties on fitted parameters (diagonal elements) and correlations between parameters (off-diagonal elements).
5. **Correlation Matrix**: Normalized covariance showing how parameters depend on each other. High correlations may suggest a better parameterization exists.

Practical Checklist for Curve Fitting

When fitting experimental data, follow these steps:

1. **Define your model** based on physical understanding
2. **Provide reasonable initial guesses** for parameters
3. **Include measurement uncertainties** (weighted fit)
4. **Check** χ^2_ν — should be close to 1
5. **Examine residuals** — should show no systematic patterns
6. **Extract parameter uncertainties** from the covariance matrix
7. **Check parameter correlations** — consider reparameterization if correlations are high

Common Pitfalls to Avoid

- **Ignoring uncertainties**: Always use weighted fits when uncertainties are known
- **Wrong model**: A bad χ^2_ν often indicates the model doesn't capture the physics
- **Overestimated uncertainties**: If $\chi^2_\nu \ll 1$, your error bars may be too large
- **Correlated parameters**: High correlations make individual parameter values less meaningful
- **Poor initial guesses**: Can lead to convergence to local minima instead of the global minimum