Curve Fitting with Python

For extracting parameters and uncertainties from measurements. Assumes you can load data into arrays and plot it — see Arrays and Plotting.

Fitting data to a model is one of the most common tasks in experimental work: you have measurements, you have a function you think describes them, and you want the best-fit parameters *and their uncertainties*. This page covers `scipy.optimize.curve_fit`, from a quick-start fit through assessing whether the fit is any good.

The quick start is enough for most measurements. The later sections (residuals, χ^2, weighted fits) are what you reach for when you need to defend a result or understand why a fit looks off.

1 Quick Start

Define your model as a function whose first argument is the independent variable and whose remaining arguments are the parameters to fit. Hand it to `curve_fit` with your data and an initial guess; read the best-fit parameters and their uncertainties off the result.

```
import numpy as np
from scipy.optimize import curve_fit

def linear(x, m, b):
    """y = m*x + b"""
    return m * x + b

x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([2.1, 3.9, 6.2, 7.8, 10.1])

popt, pcov = curve_fit(linear, x_data, y_data)   # popt = best-fit params
perr = np.sqrt(np.diag(pcov))                     # perr = 1-sigma uncertainties

m, b = popt
m_err, b_err = perr
print(f"slope     = {m:.3f} ± {m_err:.3f}")
print(f"intercept = {b:.3f} ± {b_err:.3f}")
```

Two things to take away:

- `popt` holds the best-fit parameter values, in the order your function declares them.
- `pcov` is the covariance matrix; the square roots of its diagonal are the 1σ uncertainties on each parameter. **Always report the uncertainty, not just the value.**

1.1 Look at the Fit

`curve_fit` returns numbers whether or not the model describes your data. Plot it before you believe it:

```
import matplotlib.pyplot as plt

x_model = np.linspace(x_data.min(), x_data.max(), 200)

plt.plot(x_data, y_data, "o", label="Data")
```

```
plt.plot(x_model, linear(x_model, *popt), "-", label="Fit")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Two details worth copying. The model is evaluated on a **dense** `x_model` rather than the five data points, so the curve is smooth rather than a join-the-dots polyline — which matters as soon as the model isn't a straight line. And `linear(x_model, *popt)` uses `*` to unpack the fitted parameters into the function's arguments, so the same line works whatever the parameter count.

An eyeball check catches the failures that no summary number will: a model that's the wrong shape, a fit that latched onto the wrong branch, one bad point dragging the line. Residuals make the same check quantitative once the plot looks broadly right.

2 Initial Guesses (and Why Nonlinear Fits Need Them)

`curve_fit` searches for the best parameters starting from an initial guess (`p0`). For a linear fit it almost doesn't matter. For a **nonlinear** model it matters a lot: a guess far from the true values can send the search into a flat or ill-behaved region and it may fail to converge.

```
import numpy as np
from scipy.optimize import curve_fit

def gaussian(x, amplitude, center, width):
    return amplitude * np.exp(-((x - center) ** 2) / (2 * width ** 2))

x_data = np.linspace(-5, 5, 50)
y_data = gaussian(x_data, 10, 0, 1) + np.random.normal(0, 0.5, 50)  # noisy sample

p0 = [8, 0.5, 1.5]   # [amplitude, center, width] — rough estimates from the data
popt, pcov = curve_fit(gaussian, x_data, y_data, p0=p0)
perr = np.sqrt(np.diag(pcov))

for name, val, err in zip(["amplitude", "center", "width"], popt, perr):
    print(f"{name:9s} = {val:.2f} ± {err:.2f}")
```

How to pick `p0` : read rough values straight off a plot of your data. For the Gaussian above: the amplitude is the peak height, the center is where the peak sits, the width is roughly its half-width. You don't need precision — just close enough that the search starts in the right valley.

3 Weighted Fits (Using Known Uncertainties)

If you have an uncertainty on each data point, tell `curve_fit` about it with `sigma` . Points with larger uncertainty then count for less in the fit — which is what you want.

```
import numpy as np
from scipy.optimize import curve_fit
```

```
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([2.1, 3.9, 6.2, 7.8, 10.1])
y_err  = np.array([0.2, 0.3, 0.2, 0.4, 0.3])

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

popt, pcov = curve_fit(
    linear, x_data, y_data,
    sigma=y_err,
    absolute_sigma=True,   # y_err are real standard deviations, not just relative
weights
)
```

Use `absolute_sigma=True` when your `sigma` values are genuine standard deviations (in the same units as the data). Leave it `False` if they only encode *relative* weighting.

4 Assessing the Fit

A fit always returns numbers. Whether those numbers mean anything is a separate question — these are the standard checks.

4.1 Residuals

Residuals are the differences between data and fit, $r_i = y_i - y_{\text{fit}}(x_i)$. Plotting them is the fastest way to see trouble: a good fit leaves residuals scattered randomly around zero, while a systematic curve or fanning pattern signals a wrong model or a missing effect.

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

y_fit = linear(x_data, *popt)
residuals = y_data - y_fit

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8),
                               gridspec_kw={'height_ratios': [3, 1]})
ax1.scatter(x_data, y_data, label='Data')
ax1.plot(x_data, y_fit, 'r-', label='Fit')
ax1.set_ylabel('y'); ax1.legend(); ax1.grid(True, alpha=0.3)

ax2.scatter(x_data, residuals)
ax2.axhline(0, color='r', linestyle='--')
ax2.set_xlabel('x'); ax2.set_ylabel('Residuals'); ax2.grid(True, alpha=0.3)

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

Ask: are the residuals centered on zero? Random, or is there structure? Are they larger in some region of x than others (which would argue for a weighted fit or point to a systematic effect there)?

4.2 Chi-Squared and Reduced Chi-Squared

When you have uncertainties on your points, χ² quantifies goodness of fit:

$$\chi^2 = \sum_i \frac{\left(y_i - y_{\text{fit}}(x_i)\right)^2}{\sigma_i^2}, \qquad \chi^2_{\text{red}} = \frac{\chi^2}{N - n}$$

where N is the number of data points and n the number of fit parameters ($N - n$ is the degrees of freedom).

```
import numpy as np

def reduced_chi_squared(y_data, y_fit, y_err, num_params):
    residuals = y_data - y_fit
    chi2 = np.sum((residuals / y_err) ** 2)
    dof = len(y_data) - num_params
    return chi2, chi2 / dof

chi2, chi2_red = reduced_chi_squared(y_data, linear(x_data, *popt), y_err, num_params=2)
print(f"chi-squared = {chi2:.2f},  reduced = {chi2_red:.2f}")
```

Interpreting reduced χ²:

- **≈ 1** — the fit is consistent with the data given your uncertainties. Good.
- **≫ 1** — the model fits poorly, *or* you underestimated the uncertainties, *or* there's a systematic effect the model doesn't capture.
- **≪ 1** — usually a sign the uncertainties are **over**estimated (an honest, smaller error bar would push χ² up toward 1).

A χ² test only means something if your `sigma` values are honest. Overstated uncertainties make almost any fit "look good."

4.3 When `curve_fit` Underestimates the Uncertainty

The uncertainties from `pcov` assume the only error is independent, random scatter in your y-values and that the model is exactly correct. Real experiments break those assumptions, and when they do the reported uncertainty is too small:

- **Systematic offsets** (e.g., a miscalibrated position axis) shift all points the same way; the fit can't see this, so it isn't in the parameter uncertainty.
- **Model limitations** — if the true behavior departs slightly from your model, the fit uncertainty (which assumes the model is exact) understates reality.
- **Correlated noise** (e.g., 60 Hz pickup affecting neighboring points together) violates the independence assumption, again shrinking the reported error.

The takeaway: `curve_fit`'s uncertainty is a *lower bound* that captures statistical scatter. A complete uncertainty also accounts for systematics — see Error Propagation.

5 Troubleshooting

`OptimizeWarning: Covariance of the parameters could not be estimated` or a `RuntimeError` that optimal parameters weren't found:

1. Provide better initial guesses (`p0`) read off a plot — this fixes most failures.
2. Confirm the model function is actually appropriate for the data.
3. Check for `NaN` or `Inf` in your arrays.

4. Constrain the search with `bounds=(...)` if parameters have physical limits.

Back to Python resources