Validating a Measurement Chain

For deciding whether to believe the numbers your setup produces. Assumes you can acquire data and analyze it — see Arrays and Plotting and Curve Fitting.

A measurement chain — sensor, amplifier, digitizer, analysis code — always produces a number. That is the problem. A wrong scale factor, a mislabeled channel, or a units slip does not raise an exception; it returns something plausible, and plausible numbers get plotted, fitted, and written up.

The defense is to test the chain in stages, starting with a signal whose answer you already know, and only trusting real data once each stage has been checked against something independent. This page is that procedure.

Prerequisites

An experiment that produces numbers, and something you can use as a known input — a function generator, a calibrated source, or just NumPy.

1 The Ladder

Work upward. Each stage assumes the ones below it already passed, so a failure tells you *where* the problem is rather than only that one exists.

Stage	Input	What passing proves	Hardware needed
1	Synthetic data you generated	Your analysis code is correct	None
2	A known physical signal	Your acquisition and scaling are correct	Source + your setup
3	The real signal	Nothing new — but now the number means something	Everything

Most people start at stage 3, because that is where the physics is. When the result looks wrong there, every stage is suspect at once and there is no way to narrow it down. Half an hour spent on stages 1 and 2 buys you a chain where a surprising result is interesting rather than ambiguous.

2 Stage 1: Test the Analysis on Data You Made Up

Generate a signal whose answer you know, run it through the *same* analysis function the real data will use, and check that the known answer comes back.

The point is to call the real function — not a simplified version of it. If your analysis lives in a script that also opens instruments, that is a reason to move the analysis into its own function, not a reason to skip this test.

Fitting an exponential decay is the most common analysis task in the lab, so it makes a good example:

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

```python
def decay(t, amplitude, tau, offset):
    return amplitude * np.exp(-t / tau) + offset


def fit_decay(t, v):
    """The function under test — the same one the real data will use."""
    guess = (v[0] - v[-1], t[-1] / 3, v[-1])
    popt, _ = curve_fit(decay, t, v, p0=guess)
    return {"amplitude": popt[0], "tau": popt[1], "offset": popt[2]}


# --- the test ----------------------------------------------------------
rng = np.random.default_rng(seed=0)      # seed it so the test is repeatable
true_tau = 200e-6                        # 200 µs — the answer we should get back

t = np.linspace(0, 1e-3, 1000)
v = decay(t, amplitude=5.0, tau=true_tau, offset=0.0)
v += rng.normal(0, 0.02, len(t))         # realistic noise, not a perfect curve

result = fit_decay(t, v)
error = abs(result["tau"] - true_tau) / true_tau
print(f"recovered tau = {result['tau']*1e6:.2f} µs   (true {true_tau*1e6:.0f} µs)")
print(f"error = {100*error:.2f}%")
```

Add noise of the size you actually expect. A fit that only works on a perfect curve will not survive contact with the experiment, and you want to find that out now.

2.1 Check across a range, not at one point

One test value can pass by luck — a factor-of-two error still returns *a* number, and if you only ever test one amplitude you have no way to see that the number scales wrongly. Sweep the true value instead and check the recovered values track it:

```python
import numpy as np

true_taus = np.logspace(-5, -3, 20)      # 10 µs to 1 ms
recovered = []

for tau in true_taus:
    t = np.linspace(0, 5 * tau, 1000)
    v = decay(t, 5.0, tau, 0.0) + rng.normal(0, 0.02, 1000)
    recovered.append(fit_decay(t, v)["tau"])

recovered = np.array(recovered)
slope = np.polyfit(true_taus, recovered, 1)[0]
residuals = recovered - true_taus
r_squared = 1 - np.sum(residuals**2) / np.sum((true_taus - true_taus.mean())**2)

print(f"slope = {slope:.4f}   (should be 1.0)")
print(f"R^2   = {r_squared:.6f}")
```

A slope of 0.5 or 2.0 is a scale-factor bug. A slope of 1.0 with poor R^2 means the analysis works somewhere and not elsewhere — usually a fit that fails at one end of the range, which is worth knowing before it happens to real data.

This is also where **units** get caught. A slope of exactly 1000 is not a subtle numerical problem; it is milliseconds meeting seconds.

3 Stage 2: Test the Acquisition With a Known Physical Signal

Stage 1 proves the maths. It says nothing about whether the number reaching your analysis is the voltage you think it is — the wrong channel, an unexpected gain, a range setting, or a probe attenuation all live between the signal and the array.

So put a signal you know into the real chain and measure it.

The simplest version needs no extra equipment: **loop the output back to the input.** Write a known voltage, read it back, compare. The DAQ page describes this for an NI device under Analog Output, and the same idea works with a function generator into an oscilloscope, or any source you trust into any input you don't.

For a stage with a gain — an amplifier, a divider, a current-to-voltage converter — drive a known input and check the output is what the gain says it should be:

```
import numpy as np

# Source a set of known inputs across the range you will actually use,
# measure the output of the stage, and compare against the nominal gain.
nominal_gain = 1e6                      # V/A for a 1 MΩ transimpedance stage

known_inputs = np.array([1e-7, 3e-7, 1e-6, 3e-6, 1e-5])    # amps
measured = np.array([source_and_measure(i) for i in known_inputs])  # volts

gains = measured / known_inputs
fit_gain = np.polyfit(known_inputs, measured, 1)[0]

print(f"gain per point : {gains}")
print(f"fitted gain    : {fit_gain:.4g} V/A   (nominal {nominal_gain:.4g})")
print(f"error          : {100*(fit_gain - nominal_gain)/nominal_gain:+.2f}%")
```

Two things this catches that stage 1 cannot. A **constant offset** shows up as a fitted line that does not pass through the origin, which usually means a bias somewhere you did not account for. And a **gain that drifts with input level** means part of your range is saturating — the measurement is fine in the middle and wrong at the edges, which is the failure mode most likely to survive into a final plot.

Work in the range you will actually use. A gain verified at 1 µA tells you nothing useful if the experiment runs at 100 pA.

4 The Trap: Validate With a Signal Shaped Like Your Real One

This is the failure this whole procedure exists to prevent, and it is worth stating on its own because a validation test that passes is very convincing.

A stage-2 test is only as good as its resemblance to the real configuration. If the test signal differs from the real signal in a way that matters to the analysis, the test will confirm a calibration that is wrong — and it will do so with excellent numbers.

A documented case from this program: a phase-sensitive measurement was validated by feeding one generator's output into *both* the signal and reference inputs through a BNC tee. The test passed, and it fixed a scale factor that made the generator's amplitude come back correctly. But in the real experiment the signal and the reference are different waveforms with different shapes, and the scale factor that suited identical inputs was wrong by a factor of two for the real pair. Every measurement was half what it should have been. It survived until someone compared against a separately calibrated instrument.

Nothing about the test was sloppy. It was a careful measurement of the wrong configuration.

Before trusting a stage-2 result, ask what differs between the test and the real thing:

- **Shape.** Square, sine, and pulse signals do not carry the same power at the same amplitude. Any analysis that averages, integrates, or mixes is sensitive to this.
- **Independence.** If the test drives two inputs from one source, they are perfectly correlated in a way real inputs never are.
- **Level.** A signal three decades above the real one exercises different ranging and different noise behavior.
- **Source impedance and loading.** Instruments assume terminations. A function generator driving a high-impedance input delivers twice the amplitude it displays unless told about the load — see the warning under Sweeping Frequency — which is the same class of mistake wearing different clothes.

When you cannot make the test resemble the real case, the fallback is an independent check: measure the same quantity a second way, ideally with an instrument that was calibrated by someone else.

5 Stage 3: Real Signals, Honestly Reported

Once stages 1 and 2 pass, the number means something. Report it with its variability rather than on its own.

Take several readings of the same quantity and quote the spread. The **coefficient of variation** — the standard deviation as a percentage of the mean — is the useful summary because it is dimensionless and comparable across a sweep:

```
import numpy as np

readings = np.array([take_one_measurement() for _ in range(10)])

mean = readings.mean()
std = readings.std(ddof=1)                # ddof=1: sample standard deviation
cv_percent = 100 * std / abs(mean)

print(f"{mean:.4g} ± {std:.2g}   (CV {cv_percent:.2f}%)")
```

Don't discard outliers to make this look better. A high CV is information: it tells you the signal-to-noise ratio is poor at that setting, which is exactly what you need to know at the edges of a sweep where the signal is weakest. Removing the points that disagree produces a tidier plot and a dishonest one, and it hides the measurement problem you would otherwise have gone and fixed.

If the CV is worse than you can live with, the fixes are physical or statistical — average for longer, improve shielding and grounding, warm the source up — not editorial.

6 Check What You Assume

Some quantities get typed into a script once and then believed forever. A reference frequency, a gain, a sample rate, a conversion constant: each is a number you asserted rather than measured, and any of them can be quietly wrong.

Measure them at least once. For a frequency, an FFT gives you a value limited by the bin spacing, which for a short record can be coarse — interpolating around the peak does better:

```
import numpy as np

def measured_frequency(signal, sample_rate):
    """Estimate the dominant frequency, interpolating between FFT bins."""
    windowed = (signal - signal.mean()) * np.hanning(len(signal))
    spectrum = np.abs(np.fft.rfft(windowed))
    freqs = np.fft.rfftfreq(len(signal), 1 / sample_rate)

    k = int(np.argmax(spectrum))
    if 0 < k < len(spectrum) - 1:
        # Parabolic interpolation on the three points around the peak
        alpha, beta, gamma = spectrum[k - 1], spectrum[k], spectrum[k + 1]
        offset = 0.5 * (alpha - gamma) / (alpha - 2 * beta + gamma)
    else:
        offset = 0.0
    return freqs[k] + offset * (freqs[1] - freqs[0])


print(f"nominal  81.0 Hz")
print(f"measured {measured_frequency(reference, sample_rate):.3f} Hz")
```

If the nominal and measured values disagree, find out why before going further. A mechanical chopper or a free-running oscillator is under no obligation to match the number on its dial, and an analysis that assumes the dial will drift with it.

7 What a Validation Record Looks Like

Write the results down. A validation you cannot point to is one you will repeat under pressure, six months later, when a result looks wrong at an inconvenient moment.

A real record from a working setup in this program, as an example of the form and of what “good” looks like for a well-behaved chain:

Check	Result
Analysis on synthetic signals	Amplitude recovered to ±0.7%
Linearity across the working range	$R^2 = 0.999998$
Noise floor	0.36 mV
Amplifier gain, from known currents	1.004 MΩ measured against 1.000 MΩ nominal (0.4%)
Repeatability on the real signal	CV 0.66% over 20 readings

Numbers like these are not the goal in themselves — the goal is knowing which number to distrust when something changes. A record turns “the results look odd this week” into a short list of things to re-check.

8 When Something Fails

Work down the ladder, not up.

Stage 1 fails — the problem is in your code, and you have a repeatable test case that does not need the lab. Check for scale factors, units, and sign errors first; they account for most of it.

Stage 1 passes, stage 2 fails — the problem is between the signal and the array. Check the channel, the range and any attenuation, the terminal configuration, and whether something is saturating.

Stages 1 and 2 pass, stage 3 looks wrong — the chain is doing what you asked. Either the physics is more interesting than you expected, or your test signal did not resemble the real one. Re-read the trap above before believing the first option.

Back to Python resources