Error Propagation with Python

When you compute a result from measured quantities, the uncertainties in those measurements carry through to the result. This page covers doing that propagation two ways: by hand (so you understand what's happening) and with the `uncertainties` package (so you don't have to do it by hand every time).

1 The Formula

For a function $f(x, y, ...)$ of independent quantities with uncertainties $\sigma_x, \sigma_y, ...$:

$$\sigma_f = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 \sigma_x^2 + \left(\frac{\partial f}{\partial y}\right)^2 \sigma_y^2 + \cdots}$$

Each term is a partial derivative (how much f changes when that variable changes) times that variable's uncertainty. The terms add in quadrature. This assumes the uncertainties are **independent** — if two inputs are correlated, this formula needs an extra cross term.

1.1 Common Cases

Working the partials out for the combinations that come up most often gives the rules below. Each one is written to give σ_f directly, so they all combine the same way; a, b, and n are exact constants.

f	σ_f
ax	$\lvert a \rvert \sigma_x$
$ax \pm by$	$\sqrt{a^2\sigma_x^2 + b^2\sigma_y^2}$
xy or x/y	$\lvert f \rvert \sqrt{(\sigma_x/x)^2 + (\sigma_y/y)^2}$
x^n	$\lvert f \rvert \cdot \lvert n \rvert \sigma_x/x$
$\ln x$	σ_x/x
e^x	$\lvert f \rvert \sigma_x$
$g(x)$, any single variable	$\lvert dg/dx \rvert \sigma_x$

Products, quotients, and powers are easiest to remember in **relative** form: for xy and x/y the relative uncertainties add in quadrature, and for x^n the relative uncertainty is multiplied by $\lvert n \rvert$. Squaring a quantity doubles its relative uncertainty; taking a square root halves it.

Two places these rules mislead:

- **Subtracting two close numbers.** The absolute uncertainty of $x - y$ is unremarkable, but the *result* is small, so the relative uncertainty can be enormous. A difference of two 1% measurements that nearly cancel can easily carry 100% uncertainty. If a measurement plan depends on a small difference of large numbers, that is worth catching before taking data rather than after.
- **A variable appearing more than once.** Every rule above assumes the inputs are independent. $x \cdot x$ is not the product rule — x is perfectly correlated with itself, and the correct answer is the power rule for x^2. Rewrite the expression so each measured quantity appears once, or use the package below, which tracks this automatically.

2 By Hand

For a product $f = x \cdot y$, the partials are $\partial f/\partial x = y$ and $\partial f/\partial y = x$:

```python
import numpy as np

x, sigma_x = 5.0, 0.1
y, sigma_y = 3.0, 0.2

f = x * y
sigma_f = np.sqrt((y * sigma_x) ** 2 + (x * sigma_y) ** 2)

print(f"f = {f:.2f} ± {sigma_f:.2f}")
```

Doing it by hand is worth it once or twice to build intuition — for instance, that for a product the *relative* uncertainties add in quadrature. For routine work, let the library do it.

3 With the `uncertainties` Package

The `uncertainties` package attaches an uncertainty to a number and propagates it automatically through arithmetic and math functions.

```python
from uncertainties import ufloat

x = ufloat(5.0, 0.1)   # 5.0 ± 0.1
y = ufloat(3.0, 0.2)   # 3.0 ± 0.2

print(f"x * y = {x * y}")   # 15.0+/-1.1
print(f"x + y = {x + y}")   # 8.0+/-0.22
print(f"x / y = {x / y}")   # 1.67+/-0.12
```

Access the pieces directly with `.nominal_value` and `.std_dev`:

```python
result = x * y
print(result.nominal_value, result.std_dev)
```

3.1 Correlated Variables

The package tracks *which* measurement each uncertainty came from, so a quantity used twice is handled correctly rather than being treated as two independent inputs:

```python
from uncertainties import ufloat

x = ufloat(5.0, 0.1)

print(f"x * x = {x * x}")   # 25.0+/-1.0  — correct, 2x the relative uncertainty
print(f"x - x = {x - x}")   # 0.0+/-0     — exactly zero, not 0.0+/-0.14
```

Applying the product rule by hand to `x * x` gives $\sqrt{2}\times$ too small an uncertainty, and the difference rule gives a nonzero uncertainty for a quantity that is identically zero. This is the main reason to prefer the package for anything beyond a couple of terms: real analyses reuse measured quantities, and the hand formulas silently assume you didn't.

3.2 Math Functions

Use `uncertainties.umath` in place of `math` for functions like `sin`, `cos`, `exp`, `log`:

```
from uncertainties import ufloat
import uncertainties.umath as umath

angle = ufloat(0.5, 0.01)   # radians
print(f"sin = {umath.sin(angle)}")
print(f"cos = {umath.cos(angle)}")

value = ufloat(2.0, 0.1)
print(f"exp = {umath.exp(value)}")
print(f"log = {umath.log(value)}")
```

3.3 Arrays of Uncertain Values

`unumpy` extends this to arrays, so you can propagate uncertainties element-wise:

```
import numpy as np
from uncertainties import unumpy

nominal = np.array([1.0, 2.0, 3.0])
errors  = np.array([0.1, 0.15, 0.2])
values = unumpy.uarray(nominal, errors)

squared = values ** 2
print("nominal:", unumpy.nominal_values(squared))
print("std dev:", unumpy.std_devs(squared))
```

4 Worked Example

Combining several measured quantities, each with its own uncertainty, into one derived result — the propagation happens automatically:

```
from uncertainties import ufloat
import uncertainties.umath as umath

a = ufloat(0.50, 0.02)
b = ufloat(0.75, 0.03)
c = ufloat(100.0, 1.0)

result = umath.sqrt(a * b) / c
print(f"result = {result}")   # value with propagated uncertainty
```

Compared to doing this by hand — three partial derivatives, squared, summed, square-rooted — the package is both faster and less error-prone. Reach for the by-hand formula when you need to *understand* which input dominates the uncertainty; reach for the package when you just need the number.

5 Statistical vs. Systematic Uncertainty

Propagation handles the uncertainties you feed it — but make sure you're feeding it the right ones. Measurement uncertainty comes in two flavors:

- **Statistical (random):** fluctuates from measurement to measurement and averages down with more measurements (σ of the mean shrinks like $1/\sqrt{N}$). Example: voltage noise on a reading.
- **Systematic:** a consistent bias that affects every measurement the same way and does **not** average out. Example: a position axis that reads 0.1 mm high on every point.

Curve-fit uncertainties and repeated-measurement statistics capture the statistical part. Systematic uncertainties have to be identified and folded in separately, then propagated alongside the statistical ones. A complete uncertainty budget accounts for both.

Back to Python resources