Curve Fitting with Python

Polynomials

There are multiple ways to fit models to data, depending on the model. The simplest is fitting a polynomial. For our purposes, a polynomial is an equation of the form `y = a0 + a1*x + a2*x**2 + ...`, and the degree of the polynomial corresponds to the highest exponent of `x`; so order 1 is a linear equation, order 2 is quadratic etc. Polynomials are really nice because fitting them is fast, usually a few orders of magnitude faster than other models. The reason for this is twofold: Polynomials are simple functions and fitting them only involved computing more polynomials. The math behind this fitting can all be expressed succintly using matrices; but all we care about here is how to do it. `numpy` has us covered, with `np.polyfit` and `np.polyval`. For example, with some `xdata` and `ydata` we can compute a model fit using a 4th order polynomial like so:

```
coefficients = np.polyfit(xdata, ydata, 4)
model_fit = np.polyval(coefficients, xdata)
```

The weakness of polynomials is that they very often deliver no predictive power. Unless there is a well-founded physical model behind the coefficients and/or the data is nearly linear, the model fit usually varies wildly beyond the x values covered by the data. Extrapolation is a challenging problem, but polynomials do you no favors. They are however very useful for a connect-the-dots problem which do crop up from time to time.

Generic Curve Fitting

Polynomial fitting is a particular case of curve fitting such that we can (with pencil, paper, and patience) write down expressions for all the parameters of the model. Usually this is not the case, for a variety of reasons. Enter the iterative solvers.

The Most Generic Case

The best tool for fitting a generic function to data is `scipy.optimize.curve_fit`. This function takes as arguments a function, some `xdata`, some `ydata`, optional initial parameter guesses, and some other options I won't explain here. The technique proceeds in general by repeatedly calling the function with `xdata` as the first argument, and the parameter guesses as the following arguments. If no initial guesses are provided, it will start off with `1` as each parameter (if the function takes an arbitrary number of arguments guesses must be supplied). When the function is evaluated, the output is compared to `ydata`. The `ydata` and function output must be 1-d arrays of floating point numbers, but `xdata` can be anything, even something exotic like a tuple of arrays.

This technique has its own weaknesses, the most apparent of which is that it uses a local optimizer, using least-squares. This means that the function supplied is repeatedly called with parameter adjustments trying to minimize `chisq = sum((ydata - model(xdata, *parameters))**2)`. If there is no path away from the initial guesses that keeps minimizing `chisq`, the optimizer will mostly likely just return the initial guesses. It is also possible that the optimizer gets stuck in some local minimum; in this case the optimizer will return a result that seems nonsensical. Both of these problems are mitigated by providing good initial guesses. Initial parameter guesses will also dramatically reduce the amount of time for the parameter guesses to converge to the "true" value.

Global Optimization

`scipy` does provide some resources for global optimization, but in almost every case I strongly advise against using them. For the few cases in which global optimization is the best solution to a problem, consult `scipy.optimize.brute` for a very simple brute-force evaluation of a function. This does however have an extreme weakness in that it produces a grid of all possible function parameters before optimizing. This means that functions with a large number of parameters will produce enormous memory use. The other options are `scipy.optimize.basinhopping` and `scipy.optimize.differential_evolution`. Both of these are stochastic methods; they use random number generation to randomly sample a parameter space. In general, global optimization of any sort is horribly slow and should be avoided when possible.

Examples

It is possible to fit a polynomial with `scipy.optimize.curve_fit`. Try it for yourself to see the speed difference. The only application of global optimization I have so far found is aligning images. Images taken on a telescope with questionable tracking will often need to be aligned to keep track of stars. Think about how this could be done. Keep in mind that a local optimizer here will fail unless the images are very nearly aligned (the star images overlap).

Summary

To move on you should be comfortable with * Fitting polynomials to data with `np.polyfit` * Fitting a generic function to data with `scipy.optimize.curve_fit`