Fitting functions with Python and the Gauss-Newton Algorithm

The one concept that you probably haven't encountered before is the notion of the Jacobian matrix. The Jacobian is a matrix of first-order partial derivatives, where the partial derivative in respect to a variable is put into separate columns. That is, a Jacobian of a function with 3 variables (a partial derivative can be taken with respect to each one) will have 3 columns. The number of rows is determined by the number of points at which it will be evaluated. If you're evaluating the function at 50 points, the matrix will have 50 rows. Mathematically, if there were m variables and n points at which the partial were to be evaluated, it would look like this:

$$J = \begin{bmatrix} \frac{\partial f}{\partial a_1} & \frac{\partial f}{\partial b_1} & \cdots & \frac{\partial f}{\partial m_1} \\ \frac{\partial f}{\partial a_2} & \frac{\partial f}{\partial b_2} & \cdots & \frac{\partial f}{\partial m_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial a_n} & \frac{\partial f}{\partial b_n} & \cdots & \frac{\partial f}{\partial m_n} \end{bmatrix}$$

Don't fear that matrix! You can think of it as a combination of column vectors, where the number of column vectors is the number of variables that you're trying to fit. The number of rows is the number of points at which you have data - essentially, the x values of any function $y = f(x)$. Each column vector is a vector of partial derivatives with respect to a variable, evaluated at each point where you have data.

Note: some books define the Jacobian to be the transpose of what's given above. It doesn't matter much, although the notation of most optimization algorithms uses the form from above, so we'll stick with it.

The equation for the Gauss-Newton Algorithm is:

$$(J^T J)\delta = -J^T e$$

Where: J^T is the transpose of the Jacobian, J is the Jacobian, δ is the adjustment parameters, and e is the residual (difference between modeled and actual values).

The goal is to solve for δ and add its respective components to the guess parameters from the previous iteration step. At every step, the algorithm adds these values to the previously adjusted values in order to get closer to the actual solution. Essentially, you're adding values to the parameters that you want to fit (starting from the initial guesses) until you obtain a reasonable solution.

So, let's rewrite the equation to look like what you'll be solving:

$$\delta = -(J^T J)^{-1} J^T e$$

Keep in mind that since the algorithm converges iteratively

$$\delta_{i+1} = \delta_{i-1} + \delta$$

and that δ_0 is a vector of the initial guesses for the parameters to be estimated.

Remember, you can't divide by matrices, you must multiply by the inverse instead.

Recall, the Theis equation is:

$$s = \frac{-Q}{4\pi T} W(u), \text{ where: } u = \frac{r^2 S}{4Tt}$$

Differentiating the Theis equation looks daunting. The definition of a derivative to the rescue! As long as the parameter that you add to the variable of which you're taking the partial derivative is really small, the definition of a derivative can be used to easily approximate the partial. Recall:

$$\frac{\partial f(x,y,z)}{\partial x} \approx \frac{f(x+\Delta x, y, z) - f(x,y,z)}{\Delta x}$$

$$\frac{\partial f(x,y,z)}{\partial y} \approx \frac{f(x, y+\Delta y, z) - f(x,y,z)}{\Delta y}$$

$$\frac{\partial f(x,y,z)}{\partial z} \approx \frac{f(x, y, z+\Delta z) - f(x,y,z)}{\Delta z}$$

Do you see a pattern here? The nice thing is that this can be expanded to a function of as many variables as you wish, and, as long as the Δ is very small, you can use this to easily calculate Jacobians.

Now that you've got a way to calculate the Theis equation using some parameters and calculate the partial derivatives, you can move on to writing the script to estimate the best fitting T and S values.

In this example, we have the drawdown at a single monitoring well measured over a time period, which is slightly different from the previous assignment. Specify the pumping rate, which is $500m^3/d$, to be variable **Q**, also specify the radius, which is 300 m, to be variable **r**.

The Gauss-Newton algorithm will need to be initialized with two reasonable initial guess parameters; one for each variable (T and S). I suggest using 10. for T and $1e-9$ for S. If you choose to use this algorithm for other problems, you will have to play around with initial guesses to get the algorithm to converge. You will also need to set some convergence criteria (for example the changes in the estimated values, or a general error value between the actual data and the model). I used the root mean square value as the criterion.

Since you do not know when the algorithm will converge, you have to use a while loop instead of a for loop. With the criterion being an rms value your while loop will look like: while rms¿x: do something. I used an rms value of .5. You can calculate the rms of a data vector (let's call it data) as rms=sqrt(mean(data**2.)).

To transpose a vector in Python, you have to use the `transpose()` command. To multiply matrices A and B, you have to do `dot(A,B)`. To invert a matrix, you have to import a command from Numpy using `from numpy.linalg import inv`.

Our example is to estimate the T and S parameters given drawdown and time at an observation well $300m$ away with the well pumping rate being $500m^3/d$ (from: Schwartz and Zhang, page 228).