PHYS 331 – Introduction to Numerical Techniques in Physics

Homework 6: Least-Squares, Triangular Systems

Due Friday, Mar 2, 2018, at 11:59pm.

Problem 1 – Nonlinear Least-Squares Fitting: an example from Optical Spectroscopy (25 points)

Many atomic and molecular energy level transitions result in the emission of photons at specific peak wavelengths λ, where the lineshape (the shape of the optical spectrum) is described by a Lorentzian function:

$$L(\nu) = \frac{1}{\pi} \frac{\frac{1}{2}\Gamma}{(\nu - \nu_0)^2 + \left(\frac{1}{2}\Gamma\right)^2} \tag{1}$$

where ν, the wavenumber, is defined as the reciprocal of the wavelength, $\nu = 1/\lambda$; the peak wavenumber, $\nu_0 = 1/\lambda_0$, is where the lineshape is at the maximum; and Γ is the full-width at half maximum, *i.e.*,

$$L\left(\nu \pm \frac{\Gamma}{2}\right) = \frac{L(\nu_0)}{2} \tag{2}$$

The lineshape function is already normalized such that its integral = 1. The wavenumber is a convention used in optical spectroscopy as it is more convenient than the optical frequency ($\omega = 2\pi c/\lambda$) because it can be computed more simply from $1/\lambda$. Somewhat strangely, it is given in units of cm^{-1}.

In the laboratory, you collect spectral data (optical intensity) from a sample of gases and note that there are two spectral lines. **When there are multiple lines, the lineshape functions from Eq. (2) essentially add together, but each function has an independent peak position, width, and amplitude**. As such, you wish to fit your data, $S(\nu)$ sampled at discrete values of ν in units of cm^{-1}, to a function of the form:

$$S(\nu) = c_1 L_1(\nu) + c_2 L_2(\nu) \tag{3}$$

where c_1 and c_2 are constants (amplitudes) that indicate the relative strengths of the transitions, and the peak wavenumbers ν_{01} and ν_{02}, associated with Lorentzian lineshapes L_1 and L_2, respectively, will give you needed information to determine the energy levels involved in the transitions. L_1 and L_2 also each have an associated value of Γ_1 and Γ_2, respectively. Below you will learn to import your data into Python, and perform a nonlinear least-squares fit in order to extract these 6 parameters of interest.

(a) The spectral data has been provided to you as a comma-separated-values (CSV) file, *HW6p1data.csv*. The CSV file format is a text file** containing a pair of (ν, $S(\nu)$) numbers in each line, separated by a comma. I recommend opening and examining the contents of the file in a program like Excel.

Use the numpy function `loadtxt` to import this data into an array. First, check out the info page on this function: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.loadtxt.html Note that one parameter is required to be passed to this function (*fname*), as well as several optional parameters that can be passed (this includes anything that is already set = to something else, indicating that it has a default value if not used). In particular, you will want to use the "*delimiter*" parameter to tell Python that commas are being used to separate (delimit) each set of data values. **What is the size (height × width) of the resulting array?**

** A "text" file means that the characters are encoded using the ASCII format (see, for example, http://www.asciitable.com/), which is a very common format used by simple, non-formatted editing programs like notepad. Your Python scripts are also stored in this format.

(b) Now that you have imported your data into an array, plot it. Be warned, the plot function requires separate vectors for your "x" and "y" values to be plotted, but the data matrix that you just loaded isn't natively in this format, so you'll need to manipulate the data in order to plot it. Since you will be using these later on, assign your "x" values to a new array called "`v`" to represent your wavenumber values, and your "y" values to a new array called "`Sv`" to represent your optical intensity measurements at each ν.

(c) Now, write a function `ModelSpectrum(c1,c2,v01,v02,g1,g2,v)` which takes the input wavenumber, `v`, and all of the other input parameters needed to define your spectrum S(v) of Eq.(3), and outputs the value of $S(\nu)$. Take an educated guess at the input parameters, and make an overlay plot of the function over the same range in ν as your data (where the data is plotted as individual points). It should look like two peaks, although the heights, widths, and positions will not match your data. Adjust your initial guess as needed to make it look reasonably close to the input data.

(d) The next step is to place all of the unknown parameters into a vector, in preparation to use the nonlinear solver. Do this by creating a new function, `ModelSpectrum2(x,v)`, where `x` is now a **vector** containing all of the parameters `c1, c2`, etc, and produces the same output as in `ModelSpectrum`. Hint: you will need to modify the computations within your function to call individual elements of `x`. For example, the first element of `x`, `x[0],` would replace `c1` in your computation, `x[1]` replaces `c2`, and so on. Plot the output of the function over the same range of `v`, using the same values of the input parameters as in part (c); overlay it with the original data as well. To do this, define a vector `x0` as a numpy array containing your initial guess values `(c1, c2, v01, v02, g1, g2)`. You can also continue to try to make `x0` better (*i.e.*, closer to the initial data) in this step.

(e) There is still one more step before you can to the fitting – you need a function that defines the residuals, *i.e.*, the difference between the data and your model. For the residuals, we will use this definition instead of the usual definition:

$$\text{Residuals}(x_i) = y_i - f(x_i;\text{parameters}) \qquad (4)$$

Where y_i is the spectral intensity data at x_i, and $f(x_i;\text{parameters})$ is your model computed at x_i. Normally, we'd take the sum of the differences squared, but the Python function you are going to use wants the above type of input (as it will automatically sum the squares for you). Write a function `Residuals(x,v,Sv)` where `x` is the array containing the input parameters as above, and `v` and `Sv` are arrays containing the raw data such that (ν[i], S_ν[i]) are each data pair, and the output is that from Eq.(4). Make a plot of the residuals give the same guess of the parameters, `x0`, you used in part (d). Now, adjust your parameters `x0` as needed to try to eyeball it closer to zero; **before you proceed to the next step, the residuals should be at least a little smaller than the max amplitude of the raw data peaks plotted in the above steps**.

(f) Finally, look up the scipy.org help page on the function `scipy.optimize.leastsq()`. It's not terribly helpful, although there is an implementation example at the bottom. With the guidance above, you are now pretty much ready to use this! First, `import scipy.optimize`. You have already defined the function to be minimized, `Residuals`. You have already established a reasonable starting point for your parameters, `x0`. And you already have your data, `args = (v, Sv)`. When you run the function, what does it return?

(g) Let's extract the fitted parameters and make sure they work. Store the output of the `leastsq` function in part (f) into the variable `res`. Note that it is a tuple of size 2. Assign the first element

of the tuple to a new variable, `x1`. What is the data type and size of `x1`? Now plot `Residulas(x1,v,Sv)`. Is it close to zero?

(h) Finally, overlay your original data (plotted without any interpolating lines) with the best-fit curve using `ModelSpectrum2(x1,vmesh)`. Note that it makes more sense to use a `vmesh` with much smaller spacing than the data `v`, because your best-fit curve is now a well-defined function at all v. Do they match reasonably well? Does the best-fit curve go exactly through the data points, or only approximately? What are the best-fit peak wavenumbers, `v01` and `v02`?

Problem 2 – Triangular System Solver (25 points). Now that you learned via our in-class worksheet how to solve triangular linear systems, you will write your own solver.

(a) Write a function `triSolve(M, b, upperOrLower)` that solves the system of equations $\mathbf{M}\vec{x} = \vec{b}$ (where $\mathbf{M}$ is upper or lower triangular) via forward or back substitution. The input variable upperOrLower will be used to inform the function whether the input system is upper or lower triangular, and the function may otherwise ignore components in the lower or upper triangular sector of the matrix, respectively. Implement your function such that `upperTriangle = 1` corresponds to an upper triangular, and `upperTriangle = 0` corresponds to a lower triangular. You may assume that the input `M` is square (n × n) and that `b` is a column vector of length n (matched to `M`). Your function should return the solution column vector $\vec{x}$ as a 1D numpy array – be sure it is a column vector and not a row vector!

The spirit of this problem is that you will write the method from scratch, **not using any built-in functions for manipulating or solving matrices** or triangular matrices. That said, **for function `checksolve` in part b only**, you may use built-in matrix multiplication functions (such as `np.dot`) if they are useful to you.

The most difficult aspect of this problem is to determine how to manage a triangular matrix of arbitrary size *n*. Plan out your strategy first (try abstracting the math method in terms of an arbitrary row *i* within the triangular matrix and how you will solve for the unknown x_i corresponding to that row). Start with either upper or lower triangular first to get that working. You also may want to define additional functions within `triSolve` for parts of code that need to be called multiple times. Defining functions within functions is a little odd to me, but I found this webpage which explains when and why it can be useful:
https://realpython.com/blog/python/inner-functions-what-are-they-good-for/
Alternately, you can create additional functions outside of `triSolve` as needed to perform this task. However, as is our general rule in this course, do NOT use global variables.

(b) Write a second function `checkSolve(M, x, b)` that checks whether the system of equations $\mathbf{M}\vec{x} = \vec{b}$ is satisfied. The formats of `M`, `x`, and `b` are as above. Have the function output the residual, that is, the operation $\mathbf{M}\vec{x} - \vec{b}$, which should be a column vector of zeros if `x` is the solution. This will be handy for troubleshooting and for the two steps to follow.

(c) Check your `triSolve` function by solving $\mathbf{M}\vec{x} = \vec{b}$ with:

$$\mathbf{M} = \begin{pmatrix} 9 & 0 & 0 \\ -4 & 2 & 0 \\ 1 & 0 & 5 \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} 8 \\ 1 \\ 4 \end{pmatrix}$$

Run the output of `triSolve` through your `checkSolve` function and show that the residuals are effectively zero.

(d) To really test the robustness of your code, you should be generating larger matrices filled with random numbers. Write code to define a random triangular matrix, $\mathbf{M}$, of size n × n (either upper or lower), where `n` is a variable that you can change easily. You can explore using random number generators such as `np.random.randn` for values that are Gaussian (normal) distribution, or `np.random.random` for values that are evenly distributed. You might also amplify the output by a multiplicative factor so the numbers vary over a range greater than (-1,1). Also define a random column matrix $\vec{b}$. When you pass these random matrices to your `triSolve` code, how much resulting error do you find with `checkSolve`? How does it change as you increase n from 3, to 10, to 30, to 100?