PHYS 331 – Introduction to Numerical Techniques in Physics

Homework 4: Multi-Dimensional Root Finding, Spline Interpolation

Due Friday, Sept. 22, 2017, at 11:59pm.

Problem 1 – Root-Finding in Two Dimensions (25 points)

Consider the roots (which may be complex) of the following equation:

$$f(z) = z^3 - 1 \tag{1}$$

We will use this system to develop and test a basic two-dimensional Newton-Raphson method using the procedure described below. (As an aside, the basin of convergence for each of the roots of this function is fractal – as such, this complex function is often used as a generating function for the fractal.)

(a) Rewrite Eq. (1) as a set of two equations of two variables, namely

$$g_1(x,y) = \mathrm{Re}\left[f(z) \right]$$
$$g_2(x,y) = \mathrm{Im}\left[f(z) \right] \tag{2}$$
$$\textit{where} \qquad z = x + iy$$

Now, write a function `f(x,y)` that takes real-valued float inputs `x` and `y`, and returns a numpy array of floats corresponding to the output of each of these functions (*i.e.*, $g_1(x,y)$, $g_2(x,y)$). (Note that for larger numbers of dimensions, your array would be the same length as the number of functions in your system.) Please think carefully about what is being asked for in this function – we will be more aggressive about marking down functions that do not meet the specifications above.

(b) Write down the Jacobian for your two equations, and its inverse. Then, write a function, `Jinv(x,y)` that takes real-valued float inputs `x` and `y`, and returns a 2 × 2 numpy array of floats corresponding to the inverse Jacobian, $J^{-1}(x,y)$.

(c) Write a simple two-dimensional Newton-Raphson method as a function named `rf_newton2d(f_system,Jinv_system,x0,y0,tol,maxiter)`. The inputs are the name of your system of functions, `f_system` (one of which you wrote in part (a)), the name of your inverse Jacobian function, `Jinv_system` (one of which you wrote in part (b)), real-valued starting points `x0` and `y0` that are floats, tolerance `tol` that is a float, and maximum allowed iterations `maxiter` that is integer. The output should be a numpy array of floats corresponding to the position of the root, (x,y). In this case, we define the tolerance in terms of the distance between successive iterations in 2D, *i.e.*, the distance between (x_{n+1},y_{n+1}) and (x_n,y_n).

A couple of comments. First, be aware that there do exist built-in functions to multiply matrices in numpy. However, I'd like you to write `rf_newton2d` without using them. Instead, you will need to think about how to call each element of the arrays output from `f` and `Jinv` appropriately to calculate the steps in x and y. If we were to develop this method for a larger number of dimensions, you would definitely wish to use the built-in matrix multiplication. I also note that, you would also want to treat your position vector $(x_1,x_2,\ldots,x_n)$ as an array as well (which you will do in problem 2 below), which makes the book-keeping much simpler. Furthermore, you probably wouldn't want to manually calculate J^{-1}, but rather either supply J (and let a built-in matrix inverse function calculate J^{-1} for you), or use the finite difference approximation to numerically estimate each partial derivative (see the textbook's example on pages 162-163.)

Now, test the following initial guesses for your root-finder:

$$(x_1, y_1) = (1.01, 0.01)$$
$$(x_2, y_2) = (-0.51, 0.866) \quad (3)$$
$$(x_3, y_3) = (-0.51, -0.866)$$

Discuss for each initial guess whether it converges, upon which root it converges, and how many iterations are needed until a tolerance of 10^{-3} is reached.

Extra Credit (up to 5 points): Use your `rf_newton2d` function above to display the aforementioned fractal as a 3-color image, sampled over the range of (-1,1) in *x* and *y*. Warning: the instructor hasn't tried to do this yet, so it may pose un-anticipated challenges.

Problem 2 – Determining orbital parameters from observed satellite trajectory (15 points)
See problem 27 of problem set 4.1 in the text, then follow the steps below.

(a) First, write this problem as a system of equations for which you want to find the roots. Write down the system of equations, clearly identifying which parameters are part of the vector `x`.

(b) To make life easier, your instructor lumped together a bunch of modules needed to use the textbook's version of multi-dimensional Newton-Raphson root-finding, `newtonRaphson2`. These are located in *HW4p2template.py*. All you need to do is to define your system of equations in a function `f(x)`, where `x` is now an np array of floats containing your vector `x`, and the output is a another numpy vector of the function values. Do not edit the code in the template, only adding your function to the bottom. Hint: Does the numpy `sin` function take angles in degrees or radians?

(c) Remembering that Newton-Raphson requires good starting values, critically assess the data and the orbit equation. Make an educated guess about what values for the parameters are likely to fit your data, and use these as your initial guess into `newtonRaphson2`. It is very possible for the function to complain that the "Matrix is singular" with bad starting values; if you receive this error, revise your starting values and try again. Have your code automatically report the root to the screen.

(d) From the root that you found, finally, you can answer the question in the original problem: What is the smallest *R* and corresponding value of θ? Congratulations, you have successfully predicted the future behavior of a satellite from a small amount of trajectory data. I hear NASA is hiring.

Problem 3 – Getting our feet wet with Interpolation (10 points)
Consider a function of the form:

$$f(x) = \left|\sin(x)\right|. \quad (4)$$

A template file *HW4p3.template.py* has been provided to you. Do not edit the functions; add your code to the bottom only. For now, do not concern yourself with the "LUdecomp" module that appears at the top – we will be getting to this topic in the course shortly. Notably, the cubic spline package from your textbook is provided.

(a) At what values of *x* do you think interpolation of *f*(*x*) may be particularly difficult, and why?

(b) Now, create a sampling of this data (a vector *y*=*f*(*x*)) for *x* in the range (-10,10) with a step size of 0.1. You will interpolate this artificial "data" set, in comparison to the original function. Plot the resulting set of (*x*,*y*) values. What do you notice about how close the data appears to get to *y*=0 for each period?

(c) Now you will use the cubic spline package to interpolate $f(x)$ on a finer mesh than the "data" you created in part (b). First, note that computation of the curvatures must first be computed on your data by a function call of the form `k = curvatures(x,y)`. Once you have computed k for this data, it need not be computed again.

Let's focus on values of x in the smaller range (-0.5,0.5) near zero. Using your "data", interpolate values of the function in this range on a fine enough mesh such that they look continuous when plotted. Overlay your data points on top of this continuous function. In addition, you should calculate the values of the original function on the fine mesh, and overlay that on the same plot for comparison. Label your plot appropriately (axes, legend) and limit the x and y ranges to view the results within the requested range.

What do you notice about the error of the cubic spline as a function of x?