PHYS 331 – Introduction to Numerical Techniques in Physics

Homework 4: Fixed-Point Methods and Multi-Dimensional Root Finding

Due Friday, Feb. 9, 2018, at 11:59pm.

Problem 1 – Choosing a Root-Finding Algorithm that Converges (15 points)

Consider the real roots of the 5th order polynomial below, which is the same as that which we explored in Homework 3. In the prior homework, you performed Newton-Raphson root-finding to compute the 3 real roots, and examined the basin of convergence (see solutions). In this problem, you will be asked to design and explore other fixed-point methods.

$$f(x) = x^5 - 3x^3 + 15x^2 + 29x + 9$$

(a) On paper, generate 3 distinct fixed point equations for the roots of $f(x)=0$; $g_1(x) = x$, $g_2(x) = x$, and $g_3(x) = x$. Then, calculate the derivative of each. You do not need to simplify the functions any more than needed to program them into Python. (As always, show your steps).

(b) Write the functions as `g1(x)`, `g2(x)` and `g3(x)`, and their derivatives as `dg1(x)`, `dg2(x)` and `dg3(x)`, in your Python script. To analyze their convergence properties, make overlay plots of each function and the absolute value of its corresponding derivative (3 plots total). Also plot the lines y=x, y=0, and y=1, to aid in your analysis. As always, the plots should cover the region of interest and be scaled appropriately to observe relevant features. **Make a prediction for each method of which roots they will be able to converge upon (if any)**. (Hint: the roots should be the same as those from Homework 3; if your roots for g(x) = x are not identical then you have made an error).

(c) Write a fixed-point iteration function, `fixed_pt(g,xstart,tol)` that accepts arguments of a Python function `g`, floating point variable `xstart` that represents the starting point for the iteration, and tolerance in *x* `tol`. The function should return the answer as a single, floating point variable after the tolerance is reached. Estimate the tolerance as the absolute difference between successive iterations, $|x_{n+1}-x_n|$. While you are welcome to explore adding error checking to your function, you can keep your function simple if you wish.

(d) Apply your fixed point iteration method using each of the three fixed-point equations you developed above. Attempt to measure each root to a tolerance of 10^{-20} using starting values within the basin of convergence. Summarize (make a table) whether it was possible to find each of the 3 roots with each of the 3 methods, and whether these matched your predictions from part (b). Were any methods superior to the Newton-Raphson implementation from Homework 3?

Problem 2 – Root-Finding in Two Dimensions (20 points)

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

$$f(z) = 2z^3 - 1 + 3i \tag{1}$$

We will use this system to develop and test a basic two-dimensional Newton-Raphson method using the procedure described below.

(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, $\mathbf{J}^{-1}(x,y)$.

(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 or take matrix inverses in numpy; for the purposes of your learning coding at a deeper level, **you are not allowed to use any functions that manipulate matrices in this problem**. 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. Second, do not go through extraordinary lengths to force all of the initial guesses below to work…

Tangential comments: 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 3 below), which makes the book-keeping much simpler. Furthermore, you probably wouldn't want to manually calculate $\mathbf{J}^{-1}$, but rather either supply $\mathbf{J}$ (and let a built-in matrix inverse function calculate $\mathbf{J}^{-1}$ 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, computing the roots with a tolerance of 10^{-5}:

$$\begin{aligned}
(x_1, y_1) &= (1,0) \\
(x_2, y_2) &= (-1,0) \\
(x_3, y_3) &= (0,1) \\
(x_4, y_4) &= (0,-1) \\
(x_5, y_5) &= (-1,-1) \\
(x_6, y_6) &= (1,1) \\
(x_7, y_7) &= (1,-1) \\
(x_8, y_8) &= (-1,1) \\
(x_9, y_9) &= (0,0)
\end{aligned} \tag{3}$$

How many unique roots should Eqn. (1) have? How many unique roots did you find when using the various starting values – did you find them all? Were there any initial values that didn't converge?

Extra Credit (up to 5 points): Interestingly, the basin of convergence for many types of cubic complex functions, such as $f(z)$, are fractal, and thus one can use root-finding to generate fractals. Use your `rf_newton2d` function above to display the fractal corresponding to the basic of convergence of $f(z)$ as a 3-color image, sampled over the range of (-1.5,1.5) in x and y.

Problem 3 – 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 *HW4p3template.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.