LM3331 INDIVIDUAL PROJECT I: Non-linear equations • REPORT

Raquel Romão, 5629608

The goal of this project was to build a routine to calculate the pH of a mixture of components and to do so I divided my solution of the project in three phases: first, I defined my own numerical methods, then I implemented a function whose output is the sum of the charges in the solution and finally, applied the numerical methods to the implemented function and compared them to the built-in python ones.

To define the function, I used a set of matrixes to record the needed constants: one to record the concentrations and Ka's for the species that deprotonate, other for the respective charges in their different protonated forms and other for the concentrations and charges of components that contribute to the total solution charge with extra ions and protons. Considering how each component would dissociate in water, it was possible to obtain the equations used and calculate the sum of all the charges in the solution, which will be equal to 0 when the equilibrium is reached, i.e when the final pH is reached. It was based on this that it was possible to apply the numerical methods and get the solution. It is to notice that, to avoid obtaining a negative root for the proton's concentration and, therefore, unable to apply the logarithm and calculate the pH, I defined the function with one pH value as an input.

I plotted the function for each case to get a sense where the root was, in order to define the initial conditions. For case 1 I set the tolerance to 10^{-15} and for case 2 to 10^{-14}, in accordance with the range of concentrations of protons we can get ($[10^{-14},10^{-1}]$). In general, after applying the different methods to the sum of charges function, they all gave the same solution, up to at least the 7^{th} decimal place: **6.9570139** for case 1 and **8.9135047** for case 2.

Comparing the number of iterations for each method, on one hand, as it was to expect, Regula-Falsi method is the one with the most iterations, since it's the one that implies more evaluations of the function and therefore more computational effort. Bisection method comes after with the second higher number of iterations. It was to expect this since they both have a linear convergence. I chose to use them since, even though they're slow, they are robust and will always converge, even if it takes more iterations. On the other hand, the Secant and Newton-Raphson methods had the smaller number of iterations. For the secant method I wasn't expecting such low iterations since it's a method that behaves similarly to Regula-Falsi. However, the iterations for Newton-Raphson are in accordance with its order of convergence of 2. It's a faster method and its accuracy increases with the number of iterations.

When comparing the built-in python functions with my own, for Newton-Raphson I obtained less iterations with the built-in functions, to solve it I could have given as an additional input the function's derivative, since a method converges faster if we have more information about the function we want to approximate. For the secant method I got similar iterations using both and for the bisection method I got less using my own (but less accurate, by comparing the decimal places with the roots obtained from the other methods). Finally, I thought it would be interesting to do another comparison with the number of iterations using the scipy.optimize.brentq built-in python function. I was expecting for them to be smaller since it combines root bracketing, interval bisection and inverse quadratic interpolation. Indeed it was but not with the impact I was expecting, obtaining still a smaller number of iterations using Newton-Raphson built-in function.