Computational Physics: Assignment 3

Sarah Roberts

October 9, 2018

Part 1. This portion of the assignment was written in Python. An attempt was made to use gradient descent to curve-fit the `hw3_fitting.dat` to both a Lorentzian:

$$\phi_L(v) = \frac{1}{\pi}\frac{\alpha_L}{(v-v_0)^2+\alpha_L^2}$$

and a Gaussian:

$$\phi_G(v) = \frac{1}{\alpha_D}\sqrt{\frac{\ln(2)}{\pi}}e^{\frac{-\ln(2)(v-v_0)^2}{\alpha_D^2}}.$$

The gradient descent method is only one component of the Levenberg-Marquardt method, used when the approximated values are found with low error, because Gradient Descent converges much more slowly than Newton's method, the other component to Levenberg-Marquardt. The initial data is shown below in Figure 1.

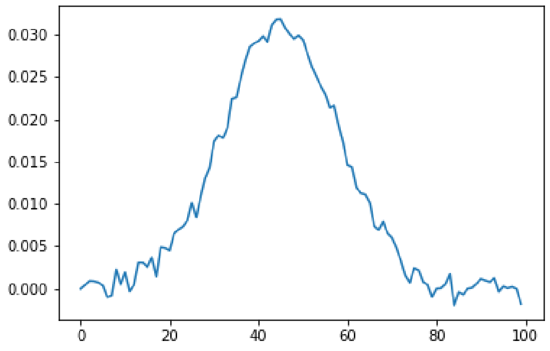

FIGURE 1. `10.png` This is the initial data from `hw3_fitting.dat`.

The Lorentzian fit is shown in Figure 2 and the Gaussian in Figure 3.

From these plots, it is clear that the gradient descent method I implemented is not functioning as intended. In fact, the initial values for α and v_0 are not adjusted by the algorithm. The Lorentzian algorithm is implemented in `gradient_descent()` and the Gaussian algorithm is implemented in `gradient_descent2()`. The error in the calculations of α and v_0 is calculated in `error()`.

Part 2.

2.1. The following plot (Figure 4) of the Runge function was generated in Python for 100 points on $x = [-1, 1]$.

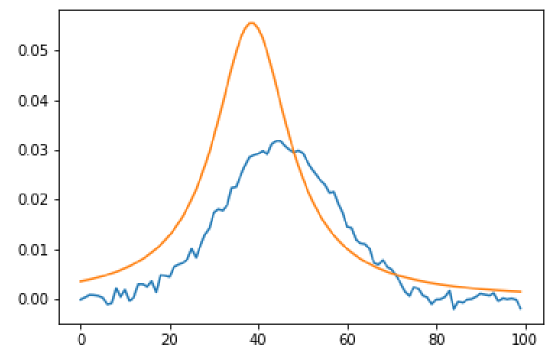

FIGURE 2. `12.png` Lorentzian fit using gradient descent.

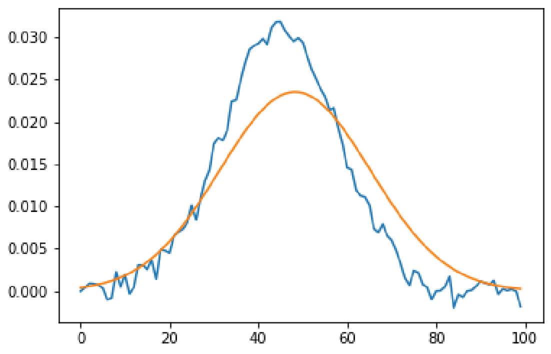

FIGURE 3. `13.png` Gaussian fit using gradient descent.

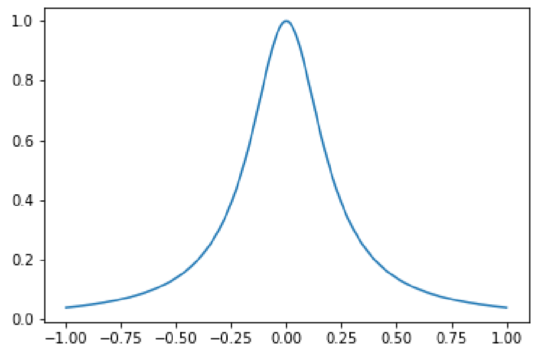

FIGURE 4. `21.png` This is the Runge function, plotted for 200 points on $x = [-1, 1]$

2.2. Lagrange interpolation is implemented in the method `Lagrange(n,x2)` the input n is the order of the Lagrange polynomial constructed, and $x2$ is the set of points over which the polynomial is evaluated. The implementation follows the algorithm outlined in *Numerical Analysis* by Richard L. Burden and J. Douglas Faires, which is Neville's algorithm. The following plot shows interpolation for $n = 6, 8, 10$. Note that the

the edges of this plot were truncated so that the behavior around $x = 0.5$ is more visible. There are lenghty tails on the end of the plot, both above and below the x axis.

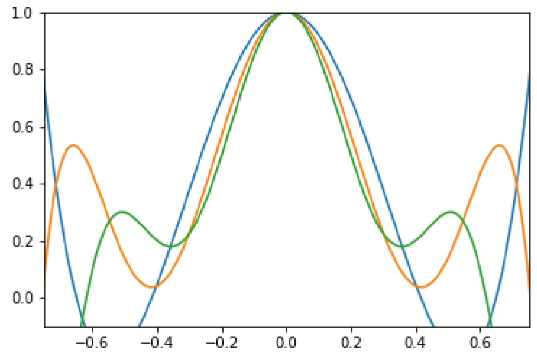

FIGURE 5. `22.png` This figures shows the Lagrange interpolating polynomials of order 6, 8, and 10 with the Runge function on $x = [-1, 1]$. The Runge function is shown in blue.

Part 3. Like the Lagrange algorithm, the cubic splines algorithm, called with `splines(n)` and accepts the parameter n, which is the number of intervals over which the cubic splines are calculated. The algorithm also follows the implementation outlined in *Numerical Analysis*. First, the a coefficients are calculated, and then the tri-diagnoal matrix is solved using back substitution. From there, the higher-order coefficients are solved using the values from the tri-diagonal matrix and the a coefficients. Finally, the formula $s(x) = a_j + b_j(dx) + c_j(dx)^2 + d_j(dx)^3$ for $dx = x - x_j$ where x is the value selected from one of the 10 'bins' along the x axis which make up the different splines and x_j is the corresponding x value from which the spline was generated.

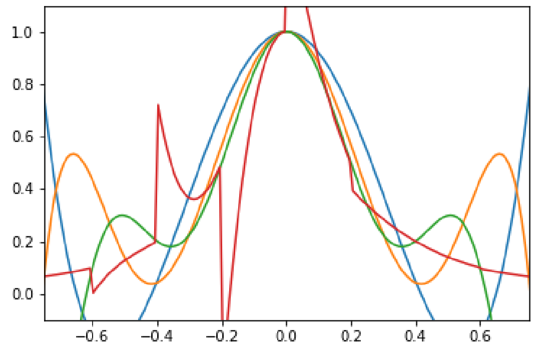

FIGURE 6. `24.png` This figures shows the Lagrange polynomials, cubic spline interpolants, and the Runge function.

Part 4. The higher-order Lagrange interpolants more closely follow the Runge function near $x = 0$ than the cubuc splines. However, the Lagrange interpolants are plagued by large deviations from the "true" function value near $x = -1$ and $x = 1$. This suggests that they are a better choice when concerned about fitting near the center of a range, while the cubic splines are a better fit in general, since their behaviour is less "unruly" in general, when compared to the Lagrange interpolants. Additionally, the Lagrange interpolants are slightly less difficult to calculate than the splines for small order, n.