AM160 HW1 Problem2

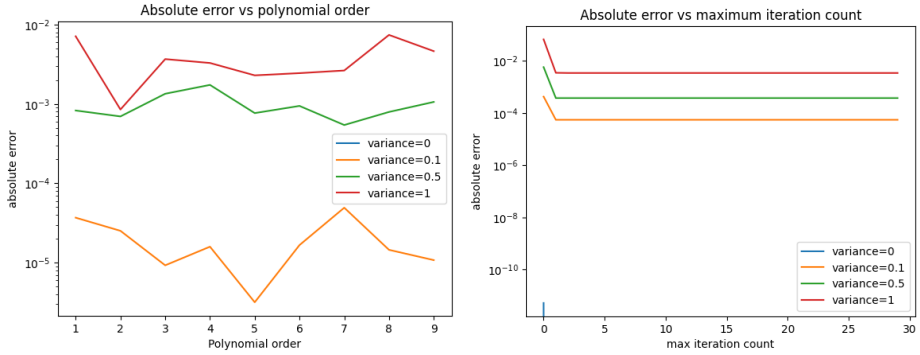
Arjun Dhamrait

February 2025

My Colab containing code is here.

A Noisy Derivatives

When trying to learn from noisy derivative data, unfortunately there is some error that we cannot learn past. We can see from the gollowing that as we increase the variance of the noise, the error of the learned physics (l2 norm of the difference between the learned coefficients of the polynomial and the actual coefficients), increases. This is regardless of the maximum polynomial term of the derivative and the maximum number of iterations.



Interestingly, most commonly when the derivatives aren't exact we seem to learn that there are linear terms in the z-components. (The model learns $\frac{\partial z}{\partial t} = xy + \beta z - 1$ instead of $\frac{\partial z}{\partial t} = xy + \beta z$) I have no clue why this is... maybe the error is linearly added has something to do with it? Honestly that doesn't seem like a correct answer, so I am not sure.

B Calculating Derivatives

When we have noiseless data, we can calculate derivatives instead of computing them from our functions. This would be useful if we did not actually know the underlying physics of our simulations. I tried 2 different methods to compute derivatives.

B.1 Finite Difference

Computing the derivatives with finite difference methods actually proved to be pretty OK! Using forward differences, we can see that the physics is just about perfectly learned, with the addition of that +1 term on the z component just like the noisy derivative section above. Again, I have no clue why this happens... Everything else, however, looks just about correct within a small range of error. Something to note here is that both forward differences, reverse differences, and midpoint differences produced about the same learned physics, which makes sense because in the grand scheme of things with 10000 data points the derivatives calculated through these three methods are just about the same (dt is so small).

B.2 Cubic Spline

I also fit a cubic spline to the data points and computed the derivatives through that. To do this, I used Scipy's handy Cubic Spline interpolation function, which has a built-in derivative calculation. I'm not sure the complexity of this, but if I'm not mistaken it should be somewhere around $O(4^2n)$ because at each point we have to interpolate a cubic spline, which is an $O(4^2)$ operation. So really this is also pretty efficient, though it did take considerably longer than the finite difference methods used in the previous subsections (showing algorithmic complexity doesn't really paint the whole picture of how long an algorithm is going to take, even when we have very large n). This method of computing derivatives gave us perfectly learned physics! That's pretty impressive from an $O(n)$ algorithm for computing derivatives.

B.3 Failed attempt: Gaussian process interpolation

In addition to those two methods, I tried to use a Gaussian Process to fit the entire data set smoothly and give me a derivative. I've left the code in my Colab, however unfortunately with this problem size I was running out of memory! I'm thinking just the idea of inverting a 10000x10000 matrix made the Colab servers cry... It might be useful to try and stencil the Gaussian Process, that way we don't have huge matrices trying to invert, however I have a feeling that would be just about the same as using polynomial splines in the long run. Because the cubic spline technique worked so well, I'm not inclined to try this out because at best it would be higher time complexity.