Project background

Low temperature plasmas underpin some of the most critical technologies of our modern society, including manufacturing processes for microchip fabrication. A limitation on improving these technologies is the lack of cheap and accurate simulations of plasma systems. Plasma simulations are generally expensive for two reasons:

- Plasmas can only be modeled accurately in high dimensions (up to 6-dimensions, we will use 4-dimensions and apply ML to 2-dimensions).
- Plasmas exhibit many different time scales, meaning that the time for a system to reach equilibrium can be much longer than the smallest physical processes which must be resolved for accurate simulations.

This second problem can mean that we require millions or 100s of millions of simulation time steps for the simulation to reach "convergence", which is the state we seek to measure and understand for engineering purposes (such as microchip manufacturing).

While you will likely not need to know the governing equations for plasma physics in this work, it could be helpful to take a look at them (and perhaps it will also make things more motivating). My favourite text for getting started with plasma physics is "Plasma Physics: An Introduction".

The governing equations for plasma physics is the Boltzmann equation acting under the force of an electrostatic field (also known as the Vlasov equation):

$$\frac{\partial f}{\partial t} + v \cdot \frac{\partial f}{\partial x} + \frac{q}{m} E \cdot \frac{\partial f}{\partial v} = \left(\frac{\partial f}{\partial t}\right)_{coll}$$

Where $f = f(x, v, t)$ is the plasma distribution function (state of the plasma over space x, velocity v and time t). q is the species charge, m is the species mass. The electric field E is solved from Gauss's law:

$$\nabla \cdot E = q \int f dv$$

Finally $\left(\frac{\partial f}{\partial t}\right)_{coll}$, is the effect of collisions.

Effectively this system of PDEs acts to evolve the kinetic plasma state f over time.

What you SHOULD know is that this system can simply be lumped into the following form:

$$\frac{\partial f}{\partial t} = g(f, E)$$

Where g is some non-linear function and E is defined as before. Since E is just a function of f we can make things even simpler. We will just consider the system of equations in the following form,

$$\frac{\partial f}{\partial t} = h(f)$$

Where h is, similarly, some (different) non-linear function.

Since we cannot solve a continuous problem using a computer, we have to discretize our problem in space and time. Let us first just consider time. Assume that we can discretized time into event chunks of length Δt, then we have time represented by,

$$t^k = t^0 + \Delta t k$$

Where k is our iteration integer and t^0 is our initial time (usually just zero).

There are many ways to discretize our time derivative, however for illustrative purposes I will just consider the simplest one, Euler's forward method. This gives,

$$\frac{\partial f}{\partial t} \approx \frac{f^{k+1} - f^k}{\Delta t} = h(f^k)$$

Or, rearranging,

$$f^{k+1} = f^k + \Delta t\, h(f^k) \qquad (1)$$

So the algorithm for solving our problem is,

1. Initialize the plasma system with some initial condition f^0.
2. Update the system (many times) using Eq. 1
3. Once the system has converged, stop iterating and record the final output. Let's call this step K, so we are interested in getting measurements of f^K.

How do we measure convergence? The simplest way could be to to define some error,

$$\epsilon = \frac{|f^{k+1} - f^k|}{|f^k|}$$

And the say the system is converged once $\epsilon < \epsilon_{tol}$, where ϵ_{tol} is our convergence tolerance. Or in other words we stop the simulation once f^k stops evolving in time.

To restate the initial problem more succinctly. To reach the point of convergence, current problems require $K = 10^6 - 10^8$, which takes days to weeks on modern supercomputers. This is simply not acceptable, and will never be used by the semiconductor industry. *The goal of this project is to try and reduce the time to convergence of our plasma simulations.*

Project description

You could think of many ways to potentially reduce the time to convergence. However the algorithm we will initially pursue for this project can be described as follows.

1. Run equation (1) for some number of time steps. Let's just say 10,000 for now. So we collect data for all f^k, with $k \in [0, 10^4]$.
2. Using the 10^4 data points we have for f^k, develop a *data-driven (ML) model* which can *predict* the future behaviour of f^k.
3. Evolve this ML model over some long period of time, call this $\Delta T \gg \Delta t$ (for example, perhaps $\Delta T = 10^5 \Delta t$).
4. Restart the full simulation from the new state of f^k.
5. Repeat steps 1-4 as needed until convergence is achieved.

An important consideration will be the cost of training and inference for our ML model. In short, we need to be sure that the training and inference time for our big jump in time (ΔT) does not take longer than just taking many small time steps with the regular code!

To kick things off we will just consider one iteration of steps 1-3 above. I will provide you with a large sample of f^k (say 1 million time steps). Your job will be to use some subset of the initial steps to predict the future of f^k (which you can compare with true simulation data).

Note: I have not gone into much detail here, however don't forget that in general f^k is not just some scalar value. It is a function with multiple dimensions (in the spatial and velocity coordinates mentioned above). In other words we will be evolving thousands or millions of points over time.

Data-driven techniques

Before jumping into fancy ML methods, I want us to start with something “relatively” simple. A technique which is relatively lightweight (cheap to train) and has very fast inference. This is called Dynamic Mode Decomposition (DMD). More specifically we will use Extended Optimized DMD, which is designed for non-linear systems (like the plasma equations), and for systems which have noise (which our simulation data does).

Here are some references to get started with DMD:

- Predictive Science Lab
- Seth Pendergrass
- PyDMD tutorial
- There are a lot of good YouTube tutorials
- Or just talk to ChatGPT!

To begin with I will ask you to write your own simple DMD algorithms, however when we get into the larger datasets we will rely on existing packages. There are some nice Python packages which implement DMD which I recommend we try to work with this through the project:

- PyDMD Github: https://github.com/PyDMD/PyDMD
- PyDMD documentation: https://pydmd.github.io/PyDMD/
- PyDMD paper: https://arxiv.org/abs/2402.07463

Following testing with DMD we can begin to look into more advanced techniques. Recurrent Neural Networks (RNNs) are the standard approach for time-series data. However RNNs often behave poorly during long rollouts. So we will likely need to implement something like Long Short Term Memory (LSTM) or ResNet’s.

In case I didn’t say it out loud already, you’ve also done more official ML classes than I have. So I’d also be open to exploring other ideas which you are interested in!

First steps

To begin I just want you to start familiarizing yourself with the DMD algorithm using the references and packages listed above.

Next we will tackle some test problems leading up to analyzing the full plasma simulation data. My plan (which might change) is to look at the following:

1. A 1D linear problem evolving in time (regular DMD).
2. A 1D linear problem evolving in time with process noise (Optimized DMD).
3. A 1D non-linear problem evolving in time with process noise (Extended Optimized DMD).
4. The full plasma data, which is non-linear 2D evolving in time with simulation noise.

Advection Diffusion Equation

To start our exploration of DMD analysis we'll turn to a canonical PDE called the advection-diffusion equation, which is a simplified equation for modeling fluid flow. The equation is linear, and should be highly amenable to DMD analysis.

$$\frac{\partial u}{\partial t} = -c\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}$$

Where u is the fluid velocity, c is the fluid speed and ν is the dissipation constant. The equation models the motion of fluid (through the advection term $\frac{\partial u}{\partial x}$) and advection (through the $\frac{\partial^2 u}{\partial x^2}$ term) in 1D. As for the plasma system of equations, we can also re-write this as,

$$\frac{\partial u}{\partial t} = h(u)$$

Or in discrete form,

$$u^{k+1} = u^k + \Delta t\, h(u^k)$$

However, as above, you do not need to actually know the equation, just the data for u^k.

I have solved this equation in two ways, (1) using an "exact" FFT approach, (2) using a finite-difference approximation which introduces some numerical errors. I have also output results with added noise, to test see how DMD handles the noise which can result from plasma simulations.

You can find the data here. It is stored as text with each line representing the spatial values of u^k at each time step, with subsequent rows being consecutive time steps.

There are 4 datasets:

- U_fft - Exact solution

- U_fd - Numerical (approximate) solution.
- U_fft_noise - Exact solution with added noise
- U_fd_noise - Numerical (approximate) solution with added noise.

I've written a helper script called *plot_drift_diffusion_linear.py* which unpacks the data and plots the timesteps at a given frequency. You should be able to better understand how the data is structured by investigating this script.

Each dataset contains 2,049 timesteps.

Task 1
I want you to "train" a DMD algorithm on the initial timesteps from the U_fft and U_fd datasets (without noise) and see how well you can predict the future of the data.

For example, train your DMD on the first 1,000 steps and see how accurately it can predict future steps and when this accuracy might begin to decay. I'll let you decide how you want to measure accuracy. But it would be good to have a plot of accuracy vs time.

I encourage you to try and roll your own DMD solver for these simple cases. But if you want to jumpy right into PyDMD go ahead!

Task 2
Do the same for the noisy data.

You can try the standard DMD algorithm for this data. However you will likely need to rely on something called Optimized DMD, which is designed to handle noisy data.

Once you've made progress on this I can share the script I used to create the data and we can play around with the noise level to see how well these different approaches work.

Linear Diffusion Equation with Boundaries (Fick's Law)

Now we're going to consider a more physically realistic system, with actual boundaries (walls) which absorb our plasma mass/density (still denoted as u below). This will be modeled via a diffusion equation (without the advection part), but also now with a source term. The result is a system which converges (over time) to a non-zero state, much like the plasma systems we will investigate.

This equation is based on Fick's law of diffusion, and is given as,

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + S(x)$$

This should look familiar, except for the addition of the source term. But now we also have Dirichlet boundary conditions.

$$u(x) = u(L) = 0$$

Where L is the length of the domain.

Since in this example our solution has a mean far from zero ($u \geq 0$ everywhere), we need to modify our DMD algorithm a little. Specifically, instead of using u^k as our state vector at each time step, we use,

$$U^k = \begin{pmatrix} u^k \\ S \\ 1 \end{pmatrix}$$

Where S is the values of our source vector (which is constant in time) at each position on the grid. So U^k is literally just a single column vector with the values of u^k, S and then the single scalar 1 stacked on top of each other. In other words, if our grid spans 64 points, the size of U^k should be 2x64+1 = 129 points.

Essentially what this is doing is providing our DMD algorithm with knowledge of the source term (which we know exactly), so that it doesn't have to "learn" that itself. Furthermore the constant 1, effectively acts as a scalar, which can shift the mean of u appropriately.

You then apply your same DMD algorithm on this larger space U^k.

When rolling out the learned operator, you also need to make some minor changes to your algorithm. During each update you would have the new system stage given by,

$$U^{k+1} = AU^k$$

Expanding this slightly, we have,

$$\begin{pmatrix} u^{k+1} \\ S \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} u^k \\ S \\ 1 \end{pmatrix}$$

However, since our DMD matrix does not *perfectly* replicate our system, the S and the 1 on the left hand side may not be perfectly reproducted by the matrix-vector product. Therefore, during each iteration, you also need to replace the S and $\approx$1 computed from the previous steps with the exact value. This will hep reduce drift in your solution!

OK, let's get started.

You can find the data here. It is formatted in the same way as before.

There are 3 datasets:

- U.txt - Solution without noise.
- U_source.txt - The exact source term (S).
- U_noisy.txt - Solution with noise.
- U_steady.txt - The analytical steady state solution (just one state).

Similarly, I've written a helper script called *plot_ficks.py* to plot the timesteps at a given frequency. This also plots the analytical steady solution (black dashed line).

Each dataset contains 7,169 timesteps. The tasks below are similar to the first tasks I had you do for the advection diffusion equation.

Task 1

I want you to train your linear DMD algorithm on the smooth data. See how well you can predict the future of the data.

Check if at long rollout times the DMD solution goes to the true steady state!

Task 2

Now things get more interesting here with noise. Since our solution does not go to zero, the signal to noise ratio stays at some reasonable level (in the advection-diffusion case it goes to infinity as the signal goes to zero). Therefore we might actually see better predictive capability here with the noisy data than what we saw previously. Use the smooth values for S since we assume we know these perfectly.

So I want you to try your Optimized DMD on the noisy dataset. How well does it do? What does the L2 loss landscape look like and how does it compare to the advection diffusion equation?

If you have more time, continue to look into better ways to improve our Optimized DMD algorithm, or try PyDMD.

Non-Linear Diffusion Equation with Boundaries

OK, I promise this will be the last toy problem before we move to plasma equations. This will be as close as we get to the plasma problem without actually doing it.

Essentially I've modified the linear diffusion equation from the previous problem to be non-linear. The equation is now,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\left(\nu_{min} + \nu_0\left|\frac{\partial u}{\partial x}\right|\right)\frac{\partial u}{\partial x}\right) + S(x)$$
$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(u\frac{\partial u}{\partial x}\right) + S(x)$$

This is still a diffusion equation of the form,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D(x)\frac{\partial u}{\partial x}\right) + S(x)$$

Where $D(x)$ introduces a term which makes the whole system non-linear. We also have our same boundary conditions.

$$u(x) = u(L) = 0$$

Below, I'll have you apply DMD in the same way you did before, constructing the expanded state vector (with u^k, S, 1), but you'll likely see that it's not going to work very well. What we need to model this equation is extended physics informed DMD.

Since we know where the non-linearity comes from in our equation, we can just compute this and *also* stack it in our state vector.

$$z^k = \frac{\partial}{\partial x}\left(\left(\nu_{min} + \nu_0\left|\frac{\partial u^k}{\partial x}\right|\right)\frac{\partial u^k}{\partial x}\right)$$

And then we define a new state for our system U^k,

$$U^k = \begin{pmatrix} u^k \\ z^k \\ S \\ 1 \end{pmatrix}$$

But how do we compute z^k numerically? Well, I am already doing this in the code I am using to generate the data, and it therefore makes sense to use that exact same discretization to compute it in your algorithm. I will provide the relevant function for computing z^k in the plotting script *plot_ficks_nonlinear.py*.

You now proceed with DMD in exactly the same way, but with this further expanded U^k. The non-linear behaviour of the system is now captured within the state vector, so it can be "learned" by the procedure.

Similarly, the rollout is performed slightly differently. Consider the expanded formula for updating U^k.

$$\begin{pmatrix} u^{k+1} \\ z^{k+1} \\ S \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} u^{k} \\ z^{k} \\ S \\ 1 \end{pmatrix}$$

Essentially, the only thing we are really going to update at each step is u^k (lower case u). This means we actually throw out the new z^{k+1}, S and ≈1. Every time we update the system and compute u^{k+1}, we want to compute z^{k+1} directly from u^{k+1} (using the code in the script I shard) and as before, replace S and ≈1 with the exact values. This keeps our model consistent and reduces drift over time.

You can find the data here. It is formatted in the same way as before.

There are 3 datasets:
- U.txt - Solution without noise.
- U_source.txt - The exact source term (S).
- U_noisy.txt - Solution with noise

In this case the analytical solution to the steady state is non-trivial (or I am too lazy to work it out - feel free to do so if you want). But you could also treat the last datapoint in the dataset as being the steady state, since it is effectively converged at long times.

Similarly, I've written a helper script called *plot_ficks.py* to plot the timesteps at a given frequency. This script computes (and plots) the non-linear term in exactly the same way as I do in the code which created this data (see line XX). So grab this snipped of code for computing z^k from u^k.

Each dataset contains 20,066 timesteps.

Task 1
First, let's try what we know probably won't work. Apply your regular DMD algorithm to $U^k = [u^k, S, 1]$ only and see how the error evolves with time. Let's stick to the smooth data for now.

Task 2
Now let's try the "physics informed" approach. Where you augment the state vector by including z^k as well. See how your DMD algorithm trains on the combined vector.

Compare the performance to what you did in Task 1!

Task 3
Noise makes things trickier here… First try working with the noisy data using the same approach. But the problem is that numerical gradients used to compute z^k only act to increase the noise in the data! So this simple formula might not be best.

You might want to try different formulas which can better smooth the data and calculate z^k. LLMs are your friend here. They can quickly recommend a wide range of approaches for obtaining smoother estimates of gradients.

Good luck!

Non-Linear Advection Diffusion Equation (Burger’s Equation) - WORK IN PROGRESS

Continuing our exploration of DMD, we turn to the non-linear equivalent of the advection diffusion equation, known as Burger’s equation (guess where Burger was from). This is nearly identical to above, except that the advection speed (c) is now dependent on the velocity itself,

$$\frac{\partial u}{\partial t} = - u\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}$$

The non-linearity is in the first term on the RHS.

Writing numerical solvers for this kind of non-linear equation is less trivial, and there is no easy way to apply the “exact” FFT approach. So I solve it in two ways, using an “upwind” method and then also the Rusanov/Lax–Friedrichs method. As before, you don’t need to know about how this is implemented, simply work with the data to try and predict the future.

Similarly I have also added noise to some of the datasets.

You can find the data here. It is formatted in the same way,with each line representing the spatial values of u^k at each time step, with subsequent rows being consecutive time steps.

There are 4 datasets:

- U_upwind - Solution with the upwind method.
- U_rusanov - Solution with the Rusanov/Lax–Friedrichs method.
- U_upwind_noise - Solution with the upwind method with added noise.
- U_rusanov_noise - Solution with the Rusanov/Lax–Friedrichs method with added noise.

Similarly, I've written a helper script called *plot_burger_diffusion.py* to plot the timesteps at a given frequency.

Each dataset contains 5,567 timesteps.

Task 1

Part a

I want you to train your linear DMD algorithm on U_upwind and U_rusanov. See how well you can predict the future of the data, and compare the prediction quality to the previous, linear, drift-diffusion equation.

Part b

Also try out your current version of Optimized DMD on the noisy datasets.

Part c

Also I'd like you to play around more with the number of training steps and rollout steps. See how this influences the accuracy at some future point in time.

Task 2

Now we will start looking into flavours of DMD which can better handle nonlinearities. We'll try exploring non-linear DMD and physics informed DMD.

…