Post-processing Pixie3D Data

Jason Hamilton

September 19, 2023

Introduction

This note will explain the post-processing work done for Pixie3D data. This work is predominantly done through Julia scripts. Julia is now installed by default on most HPC machines, so the only prerequisite is the installation of individual 3rd-party packages, which is done through the Julia command line interface, the REPL. Despite being very Python-like in style and being able to be run interpretively, Julia is a high performance compiled language, comparable to C and Fortran. A user will **not** need to be familiar with Julia in order to run these scripts. Note: like Fortran, Julia uses column-major ordering for arrays, and indexes from 1 not 0.

As an alternate to scripts, Julia can be run in a Jupyter notebook. Instructions for setting up a Jupyter notebook with Julia are found in Giannis Keramidas' notes, and will be posted here as well at some point.

Pixplot

Pixplot is the post-processing arm of Pixie3D. It will read the various `record.bin` ADIOS files in the current directory and output various files that are more useful for analysis. The HDF5 file `Pixie3D.h5` is the main file for our purposes. Also generated are `draw*.bin` files that can be read with `xdraw` (not typically included by default on HPC machines). These xdraw files are most useful for keeping an eye on a run's output while it is still ongoing.

The other kind of file generated are `poinc_t=*.bin` files which are to be used with Nemato below.

Which files are generated or not depends on the settings in the `graphics` or `graphdef` namelist of the input deck `Pixie3D.in`. A common run command for Pixplot will look like "`./pixplot.petsc.x -dplot 10.`" where the resulting output timesteps will be (roughly) every 10 Alfven times. The output interval will not be exact since Pixie3D will write data to the `record.bin` ADIOS files at an interval defined by the `ts` namelist variable `dstep` in the input deck. Pixie3D will also write data at the end of a session (i.e. when an allocation ends on a HPC machine), regardless of the last time data was written. So if `dstep` is too large relative to the duration of the session, Pixie3D may be writing data at irregular intervals over the course of many sessions.

Nemato

The Nemato code is used to generate Poincare section plots of the magnetic field lines. A slurm run script called `run_nemato.sh` is used that submits a slurm job for the Nemato code. This batch script uses a python script written by Giannis Keramidas `poinc.py` which generates images and movies of the resulting Poincare plots. In order to plot the boundary geometry along with the Poincare plots, the python modules `efit_plots.py` and `master_read.py` are used. As long as these python modules are located in a directory in python's path (bash variable $PYTHON_PATH), they will be used to read the EQDSK file to find the geometry information. The python files will do a brief attempt to locate such an EQDSK file if it exists, but won't plot boundary info if it can't locate it. Typically this file will be in the same scratch directory as the Nemato files or its parent directory, and there should not be an issue locating it if it exists.

This shell script can be run with "`sbatch run_nemato.sh`" which will process all `poinc_t=*.bin` files in the current directory. I recommend moving all of these files into a separate directory along with `run_nemato.sh`, `poinc.py`, the input deck `nemato.in`, and the Nemato executable `nemato.mpi.x`. This is because the large number of output files from Nemato will clutter your working directory. In order to run this script, the same MPI module must be loaded that was used to compile Nemato. The script will automatically load a python module, but one must exist to generate the images and movie.

The shell script has several options that can be included at run time. `run_nemato.sh --help` will explain these options.

The main product of this script is the generation of an animation of the Poincare section at all output timesteps, nominally called `poincare.mp4`. This movie is generated by `ffmpeg` which is also typically installed by default on all HPC machines. However, each installation of ffmpeg may have different video codecs enabled. If ffmpeg gives a runtime error, or you want to change the video format, you will have to edit the ffmpeg command in `poinc.py` (near the very bottom of the script). The default settings of this command are chosen to be robust, but not necessarily optimal.

Nemato will also generate xdraw files, so that the Poincare plots can be viewed quickly for testing or avoiding reliance on the `poinc.py` script.

ParaView

The HDF5 file `Pixie3D.h5` should be readable in ParaView if the VisitBridge add on was enabled during the compilation of ParaView. On some versions of ParaView (such as 5.8), VisitBridge is enabled by default. In newer versions, it appears there is a bug present in VisitPixieReader that does not correctly parse strings in the HDF5 file. The HDF5 file by default includes a "Visit Expression" string that tells Visit which variables are vectors and which variables are their corresponding vector components. This bug in VisitBridge causes an error when ParaView loads the HDF5 file, resulting in no data being loaded. To remove the Visit Expression string, a Julia script called

`removeexpressions.jl` can be used. After loading a Julia module in your HPC environment, run "`julia removeexpressions.jl`" and the `Pixie3D.h5` in your current directory will be edited so that it can be opened with ParaView. You may want to edit the simple script `removeexpressions.jl` to change the output file name to something else.

Once loaded in ParaView, all of the vector components will be read as independent variables. You can reform these components into Vectors by using either the ParaView calculator feature (i.e. "myvector = (iHat*X_VELOCITY) + (jHat*Y_VELOCITY) + (kHat*Z_VELOCITY)") or more conveniently by using the `Merge Vector Components` filter, which is included by default in all newer ParaView versions.

Of course, Visit itself can also be used to read `Pixie3D.h5` and do much of the same analysis as ParaView can, although I have personally never used it, and ParaView has a much more active user community and documentation, with new pre-built filters being added regularly.

Plotting in Julia

Some plots are frequently desired, so instead of loading the large HDF5 file in ParaView every time new data is available, it is much faster to instead use a Julia script to load the HDF5 and plot this data. This can be done with the script `diagnosticplots.jl` which loads the `Pixie3D.h5` file in the current directory, and output plots and movies in the same directory. Note that the VisitExpressions string mentioned above does not need to be removed to be loaded in Julia, although it won't be used so its presence is unnecessary. This script can be edited to change the particulars of the desired plots. There are sufficient examples of both scatter/line plots as well as 2D plots that a user should be able to change variables/settings to what they desire. As pointed out above, the local installation of ffmpeg may impact which video codecs and hence what video formats are possible, so the ffmpeg commands may need to be altered.

This script makes use of 4 packages that need to be installed in your local Julia environment before being used (this only needs to be done once per version of Julia). All packages used in my Julia scripts are compatible with Julia 1.0+ (prior versions are not backward-compatible), so the most recent versions of these packages should be suitable. These packages are HDF5, Plots, Statistics, CairoMakie. Install these packages by first running `julia` from the shell, which opens the Julia REPL. Then enter this:

```
using Pkg ; Pkg.add(["HDF5","Plots","Statistics","CairoMakie"])
```

After completion, you can now exit Julia with `exit()` and run the script with `julia diagnosticplots.jl`, which itself may take a couple of minutes depending on the size of the dataset.

Magnetic Coordinates

The most comprehensive post-process script is `magnetic_coordinates.jl` which reads data from the logical Pixie3D grid (r, θ, ϕ) and constructs a new grid (ψ, θ_m, ϕ) wherein

magnetic fields are straight. The script is intended to be robust enough to work for any Pixie3D data, so no micromanaging of the code by a user should be required.

This script is most efficient when run in parallel with MPI. While it can be run in a login or interactive node serially, if the dataset contains many output timesteps and high resolution, it can take up to an hour or more to finish, so submitting it as a slurm job is recommended. Run this script serially with "`julia magnetic_coordinates.jl`" or as a slurm job with "`sbatch -n 8 julia magnetic_coordinates.jl`". Replace slurm with `mpiexec` when running in an environment without slurm. 8 tasks is of course just an example, suitable for a small dataset. However, because the code is extremely parallelizable with the only overhead being the initial loading of the HDF5 file and the writing to the output file/plots at the end of the script, the script is efficient with an increase in processes (until the number of processes equals the number of timesteps!).

As with the plotting script in the previous section, some packages need to be installed first (only once per Julia version):

```
using Pkg ; Pkg.add(["Interpolations","Dierckx","MPI","Statistics"])
Pkg.add(["HDF5","NPZ","EllipsisNotation","Plots","PyPlot"])
```

(split into several lines for presentation purposes and allowing someone to copy-paste into their Julia REPL)

The rest of this section will detail the calculations in this script.

1 Initializing + user options

The script loads all packages, then initializes MPI (ranks, communicators, etc.). Default settings are then defined, before overwriting them with any user provided options when running the script. User options include:

`filepath` : specify the path to the .h5 file. Default is `pwd`

`filename` : specify the name of the .h5 file. Default is `/Pixie3D.h5`

`tstart` : specify which timestep to start processing. Default is 0

`tend` : specify which timestep to stop processing. Default is last timestep in the .h5 file

`q_only` : set to true to stop the script once q is calculated. Default is false

`plot_surfaces` : set to true to plot flux surfaces and then stop the script. Default is false

`closed_boundary` : set to true to tell the normalization routine to use the boundary value of ψ instead of an x-point value

2 Loading data

The HDF5 file is loaded. The timesteps and their time values in Alfven times are loaded from the file with `getTimesteps`. The script then uses the number of timesteps to determine how to partition the dataset for parallelization. Each process will be allocated a number of timesteps to load. Variable data is then read from the HDF5 file using `loadh5`. Coordinate data is read from the first timestep as the Pixie3D grid is static (until/if AMR is added – then this script will need to be updated to contain coordinate data at each timestep). The HDF5 file is then closed.

3 Data format + cell/grid distinctions

First check is determining if the sign of the poloidal flux ψ needs to be flipped (align sign of the toroidal magnetic field on the axis to the positive $\hat{\phi}$ direction). The script checks for the sign of ψ at the origin, and flips $\psi(r, \theta, \phi)$ correspondingly. The second check is if the data is in 2D (i.e. `nzd = 1` in the input deck), and if so, extrude to a second toroidal point so that the interpolation methods don't throw a fit. If the `graphdef` namelist in the input deck contains `car_diag_plots = f`, then ψ will be cell-based, if not then ψ will be node-based. We want all data to be node-based, so we convert ψ from cells to nodes if needed using `ConvertCell2Node`. The magnetic field components will be cell-based, so all of those will need to be converted.

The cell and node locations are made into arrays based on the dimensionality of the dataset. Cylindrical R values of each node are also computed. Once everything is in the correct node-based format, toroidal averages are taken corresponding to the n=0 modes, e.g. $\psi_0(r, \theta, t)$.

4 Construction of the integrand

In order to perform the magnetic coordinate transformation, the magnetic angle θ_m (defined as the angle such that flux surfaces are perpendicular to lines of constant θ_m) must be calculated. Derivations exist in textbooks, or the appendix of Bonfiglio et al (2017), which gives (in a general coordinate system):

$$\theta_m(s) = \frac{1}{q} \int \frac{B_0^\phi}{\sqrt{g_{rr}(B_0^r)^2 + 2g_{r\theta}B_0^r B_0^\theta + g_{\theta\theta}(B_0^\theta)^2}} ds \tag{1}$$

(Bonfiglio's derivation is in the cylindrical case where $g_{rr} = 1$, $g_{r\theta} = 0$ and $g_{\theta\theta} = r^2$) where s is the path defining the flux surface at a fixed ϕ. To determine the safety factor q, if the path s is taken to be exactly 1 complete circuit around the flux surface, then $\theta_m = 2\pi$ by definition, and we have:

$$q = \frac{1}{2\pi} \oint \frac{B_0^\phi}{\sqrt{g_{rr}(B_0^r)^2 + 2g_{r\theta}B_0^r B_0^\theta + g_{\theta\theta}(B_0^\theta)^2}} ds = \frac{1}{2\pi} \oint I ds \tag{2}$$

where we will refer to the integrand as I for brevity.

Since Pixie3D stores contravariant vectors with a Jacobian factor, $B^{(1,2,3)} = \mathcal{J} B^{(r,\theta,\phi)}$ and covariant vectors as $B_{(1,2,3)} = B_{(r,\theta,\phi)}$, we can tranform the integrand I into a form using the Pixie3D vectors:

$$B_0^\phi = B_0^3/\mathcal{J} \tag{3}$$

$$g_{rr}(B_0^r)^2 = g_{rr}(B_0^1)^2/\mathcal{J}^2 \tag{4}$$

$$2g_{r\theta}B_0^r B_0^\theta = 2g_{r\theta}B_0^1 B_0^2/\mathcal{J}^2 \tag{5}$$

$$g_{\theta\theta}(B_0^\theta)^2 = g_{\theta\theta}(B_0^2)^2/\mathcal{J}^2 \tag{6}$$

$$I = \frac{B_0^3}{\sqrt{g_{rr}(B_0^1)^2 + 2g_{r\theta}B_0^1 B_0^2 + g_{\theta\theta}(B_0^2)^2}} \tag{7}$$

We can also use:

$$B_{pol} = \sqrt{\frac{B_0^1 B_{10} + B_0^2 B_{20}}{\mathcal{J}}} = \frac{1}{\mathcal{J}} \sqrt{g_{rr}(B_0^1)^2 + 2g_{r\theta} B_0^1 B_0^2 + g_{\theta\theta}(B_0^2)^2} \tag{8}$$

to have in general geometry:

$$I = \frac{B_0^3}{\mathcal{J} B_{pol}} = \frac{B_0^3}{\sqrt{B_0^1 B_{10} + B_0^2 B_{20}}} \frac{1}{\sqrt{\mathcal{J}}} \tag{9}$$

The Jacobian factor itself is readily obtained by evaluating quantities already computed and contained in the `pixie3d.h5` file:

$$\mathcal{J}(r, \theta, \phi) = \frac{B^1 B_1 + B^2 B_2 + B^3 B_3}{B_x^2 + B_y^2 + B_z^2} \tag{10}$$

The script will calculate B_{pol} and then interpolate all field arrays as well as ψ over the entire domain using `NodeInterpolation`, to obtain $I(r, \theta, \phi)$.

5 Normalizing ψ

The other necessary step to creating the magnetic grid is the normalization of the poloidal flux such that $\psi = 0$ at the magnetic axis and $\psi = 1$ at the x-point of the separatrix. This is all done in `NormalizePsi`.

First, a rough approximation of the axis location is made by determining where on the Pixie3D grid is ψ minimized. To avoid cases where ψ has a global minimum beyond the separatrix, only locations relatively close to the origin are considered. The true location of the axis is determined by creating a high resolution interpolation of ψ in the neighbourhood of the grid minimum, then this minimum value is recorded.

Next, a rough approximation of the x-point location is made by determining where in the grid is there a saddle point in ψ. This is done by looping through all grid points and checking the signs of $d\psi/dr$ and $d\psi/d\theta$. Note that such loops are optimal in Julia. Locations neighbouring the boundary and the origin are not considered. Like for the axis, the true x-point location is then determined by creating a high resolution interpolation of now B_{pol} in the neighbourhood of the grid saddle point, then the value of ψ at the minimum location is recorded. The poloidal flux is then normalized via:

$$\psi_N = \frac{\psi - \psi_{axis}}{\psi_{xpnt} - \psi_{axis}} \tag{11}$$

This normalized flux will be the radial magnetic coordinate, with the field lines perpendicular to $\nabla\psi_N$. This ψ grid is created as a collection of 101 points corresponding to $\psi_N = [0., 0.01, 0.02, ..., 0.99, 1.00]$. The next step is to find the flux surfaces at each of these values. Because there is a singularity in I at the axis where $B_{pol} = 0$, we set a ψ_{min} as the lowest value that we will attempt to calculate the flux surface. Similarly we set a ψ_{max} because we want to avoid integrating along a path too close to the

separatrix where we risk following an open field line. So we seek the flux surfaces for $\psi_N = [\psi_{min} : \psi_{max}]$.

Two special cases present themselves. The first is when the boundary condition of the simulation is itself a closed flux surface. In this case, there is typically no x-point found in the domain, and ψ should instead be normalized by the (minimum) value at the boundary $r = 1$. The minimum is taken to avoid small deviations in ψ on this boundary surface that may lead to a field line intercepting the wall. To detect if this boundary condition normalization should be used, there is a Boolean user runtime option `closed_boundary` that should be set to `true` (default is `false`). Even if not set by the user, the script will look for an x-point at the first timestep, and if not found, will force `closed_boundary=true` and broadcast this result to all processes. However, it is recommended that the user set this option themselves as the script will only detect a dataset's first timestep which will not be loaded if the user had set `tstart` to some non-default value.

The second special case is when the simulation has far exceeded the resistive decay time and the closed flux surfaces have completely dissipated. Obviously there are no magnetic coordinates to be computed for these timesteps. The script detects if this has occurred by seeing if the x-point can no longer be found after one had been located at the first timestep. Assuming then that the separatrix has dissipated to nothing, the script marks these timesteps and no further calculations will be done for them. Note that if `closed_boundary=true` has been set, then the script will continue to use the boundary values and likely unintelligible results will follow for the q profile and the coordinate system.

6 The integration path

To determine the path of integration s we use MatPlotLib's contour package to obtain the isosurfaces of $\psi_{min} : \psi_{max}$, extract the polygons that the plotting routines obtained from the python API, and then find the coordinates that define these polygons.

The routine `PsiContour` first creates a much higher resolution interpolation of ψ_N (high resolution is needed to ensure smooth contours at high radius where the grid is less dense).

To avoid segments of the isosurfaces that are beyond the separatrix and hence belong to non-closed surfaces, only the locations inside the separatrix are considered. This is done by using values of r inside where $\psi_N < 1$. Values of ψ_N beyond these r values are set to $\psi_N = 1$ such that all isosurfaces for $\psi_N < 1$ are inside the separatrix.

The Julia package `PyPlot` imports MatPlotLib from Python, which is used to find the isosurfaces. Since the poloidal flux is constant over a flux surface, its isosurfaces are synonymous with the flux surface itself. The Python API allows extraction of the polygons that make up each contour, and these are used to fill the `surfaces`$[\psi, t]$ array with all coordinates defining each flux surface. These coordinates will be our integration path.

The script allows plotting of these flux surfaces with `PlotSurfaces` for testing purposes by using the runtime option `plot_surfaces=true`. Once again, ffmpeg settings may vary depending on your HPC environment, so editing of the `SurfaceAnimation`

function may be required. All tested cases thus far result in very smooth and accurate flux surfaces.

Next, the integration is performed in `flux_integrator` which evaluates the integrand I at all coordinate points in `surfaces`$[\psi, t]$ for the range $\psi_{min} : \psi_{max}$ and all times t.

In the case where the path crosses very close to $r = 0$, then the Jacobian factor will be very small, and hence there will be a large error in the integrand. When this occurs, the script will instead average neighbouring points on the path vs dividing by such a small number. Also note that due to the interpolation of $\mathcal{J}$, it is possible that $\mathcal{J}(r = 0, \theta, \phi) < 0$ by a small amount. The described procedure also avoids this scenario.

The evaluation of Eqn (2) is then simply:

$$q(\psi_N, t) = \frac{1}{2\pi} \sum_k I_k \Delta s_k \tag{12}$$

where the sum runs over all coordinate pairs `surfaces`$[\psi, t](r_k, \theta_k)$ that define s. $I_k \Delta s_k$ can be evaluated using the Midpoint or Simpson's methods, with both results being similar.

7 Defining the magnetic grid

Now that q has been calculated and all flux surfaces are known, we can define the magnetic grid (ψ, θ_m, ϕ) where by definition, the averaged magnetic field lines $\mathbf{B}_0$ are straight. The toroidal angle is chosen to be equal to the logical ϕ coordinate for simplicity in the range $[0, 2\pi)$. The radial coordinate is taken to be ψ_N in the range $[0, 1]$. The poloidal coordinate θ_m traverses the flux surfaces in the range $[0, 2\pi)$ and is obtained from Eq. (1).

8 Evaluating perturbations on the new grid

The script lastly evaluates perturbations of the field lines from their n=0 counterparts. Using the derivation given by Bonfiglio et al (2017),

$$\hat{b}^\rho = q \frac{(B^1 - B_0^1)B_0^2 - (B^2 - B_0^2)B_0^1}{B_0^3} \tag{13}$$

$$\hat{b}^\phi = q \frac{(B^3 - B_0^3)}{B_0^3} \tag{14}$$

which is then interpolated over the new magnetic grid (ψ_N, θ_m, ϕ). The perturbed variables can be analyzed interactively using the output files described in the next section, or the user can specify the desired Fourier mode (using a FFT package) results at the very bottom of the script. The script by default will not evaluate all possible Fourier modes to avoid bloat. The script as is performs an FFT for $\hat{b}^\rho$ and $\hat{b}^\phi$ on the new grid at a specified time, and the results are plotted. If either q is incorrect or noisy or the field lines are not straight on (ψ_N, θ_m, ϕ), then the FFT results will be unintelligible.

9 Output

Output files include `NumPy` data files which can be loaded in Python via:

```
>>> import numpy as np
>>> q = np.load("q_0_989.npy")
>>> np.shape(q)
(101, 989)
```

where the timesteps in the `.npy` file names will reflect the range of timesteps processed with the current run of `magnetic_coordinates.jl`.