Permar - a suite of analysis routines for the Lare3d code

Val Aslanyan – 26/12/22

Summary

LARES PERMARINI: Roman guardian deities of seafarers

This package contains native Python3 routines to read, process and display outputs from the Lare3d code. In particular, the SDF file format of the Centre for Fusion Space and Astrophysics (CFSA) at the University of Warwick (as opposed to a completely different and more widespread scientific SDF file format) can be read using a single Python function without the need for external routines.
There is also an extension to the Lare3d code itself to output the velocity at *e.g.* the bottom boundary every timestep, thereby allowing the "footpoints" of magnetic field lines to be accurately tracked throughout the simulation. Higher level quantities such as the squashing factor Q and the connectivity of magnetic field lines can also be calculated.
The package is written mostly in Python version $\geq$ 3.7, with additional high-performance routines in Fortran 90.

Installing and using Permar

Clone the Permar git repository You will see the main Permar_Functions_Python.py file containing the main processing routines - this should be kept once, pulled from the repository as required and probably not modified. Alongside are smaller scripts which you will probably want to copy and paste into multiple folders with Lare runs as required and modify heavily.

Near the top of every one of the smaller scripts is the following preamble, which will load the prerequisite libraries and the Permar routines themselves if the string '/Change/This/Path' is set to the location of Permar_Functions_Python.py:

```
import sys
sys.path[:0]=['/Change/This/Path']
from Permar_Functions_Python import *
```

The routines and scripts should be easy to follow for anyone familiar with Python. Take a look at ReadLare3d.py as a starting point. There is a function call to read

```
lare3d_file_properties,lare3d_blocks=read_cfsasdf_file(lare3d_filename)
```

Normalizing constants are read from the lare3d.dat file as follows:

```
lare3d_dat=read_lare3d_dat(lare3d_datfilename)
```

Once loaded, a specific quantity can be extracted from the blocks with quantity_from_blocks for a generic grid/quantity combination or regularize_Bfield, which will return the magnetic field on a single cell-centered grid (normally, each component is stored by Lare3d at a different cell edge location).

If memory becomes an issue, for example if a single array is > 10GB, the following function call will return a single quantity as economically as possible,

```
time,grid_x,grid_y,grid_z,data=read_cfsasdf_file_lean(filename,target)
```

where target is a string specifying the quantity.

The following widespread `Python` modules are used in the package:

- **numpy** [mandatory]
- **matplotlib** [2D plotting]
- **mayavi** [3D plotting]

To install `Python` modules, type the following into a terminal:

```
    pip3 install [module] --user
```

What is not included in the package

- **ffmpeg** - required for movies; this is open source and can be installed straightforwardly on Linux/Windows
- Base version of `Lare3d` - this can be cloned from https://github.com/Warwick-Plasma/Lare3d
- Modified version of `Lare3d` to output surface velocity - can be obtained from me on request
- Base version of `QSL Squasher` - this can be cloned from https://bitbucket.org/tassev/qsl_squasher/src/hg/
- Modified version of `QSL Squasher` which writes its result to a structured file - can be obtained from me on request

Running `QSL Squasher` with `Lare3d` outputs

`QSL Squasher` is a tool for computing the squashing factor Q for a magnetic field. It requires a GPU in order to integrate bundles of thousands of field lines. As mentioned above, you will need to obtain a modified version of `QSL Squasher` which writes its results to a self-contained file. Thereafter, you must have `OpenCL` installed to compile it by running

```
    ./complie.sh
```

in the source directory. It will output an executable file called `qslSquasher`. It must be recompiled from scratch for every combination of input/output grid; in particular, the variables `NX`, `NY` and `NZ` in `options.hpp` must match the grid of `Lare3d`. It is therefore advisable to rename the executable to something like `qslSquasher_Lare1` or `qslSquasher_256`, *etc*; remember this choice for later and copy the file into the directory above where your `Lare3d` data is.
In the same directory as your `Lare3d` data, run the script `Lare_out_to_squash_in.py`. This will create a directory called **QSL** by default with further subdirectories numbered after each `SDF` file with the grid and field data required by `qslSquasher`. Copy `Run_QSL.py` to the **QSL** directory and modify the following lists:

```
    steps_to_process=[0]
    binaries_to_run=['qslSquasher_Lare1']
    outfile_names=['qslLare']
```

As the names suggest, the lists in turn specify the `SDF` output step number, the name of the executable from above, and what you want the output corresponding to the above to be called. Once that is set, run `Run_QSL.py` to handle everything with `QSL Squasher` itself. In each of the folders corresponding to `steps_to_process`, you will have files such as `qslLare.qsl`.
In the end, your directory structure and files within it should look as follows:

```
{Top Level Directory} → qslSquasher_Lare1
   |– – – – – – – – – – – {Lare3d Run} → Lare_out_to_squash_in.py
   |                          |– – – – – – {QSL} → Run_QSL.py
   |                          |                |– – – – – – {0000}
   |                          |                |– – – – – – {0001}
   |– – – – – – – – – – – {Another Run} → Lare_out_to_squash_in.py
   |                          |– – – – – – {QSL} → Run_QSL.py
   |                          |                |– – – – – – {0000}
```

You can use `Plot_QSL_2D.py` or `Animate_QSL_2D.py` to display a single or series of Q maps, respectively.

Outputting surface velocity from `Lare3d`

A modified version of `Lare3d` is available which will output the plasma velocity at every timestep on a single plane of constant z, typically at the z_{min} boundary. This allows the motion of parcels of plasma, and in particular frozen-in field lines, to be accurately tracked. Additions have been made to `control.F90`, `boundary.F90` and `core/shared_data.F90`; these can be incorporated into an existing setup.
Compile such a modified `Lare3d` program with at least one of the following variables inside `control.F90` set to `.true.`:

```
output_Vx=.false.
output_Vy=.false.
output_Vz=.false.
```

You can optionally change `output_V_frequency` to output the velocity once every that many timesteps and `output_V_layer` to output the velocity on a given plane of z; warning - at present, it is up to the user to ensure that the layer must lie within the set of processors which control the bottom boundary. When you run `Lare3d`, you will see potentially many files such as the following in your **Data** directory:

```
Surft_000000000.bin
SurfVx_0000_000000000.bin
SurfVx_0001_000000000.bin
...
```

Once the simulation run(s) are complete (the velocity outputs are robust to multiple restarts), run `Convert_SurfaceV_Files.py` to merge this mess of surface velocity files into between 4 and 6 reasonably neat files containing the x and y grids, the simulation time and the velocity fields themselves:

```
Surft.bin
```

```
Surfx.bin
Surfy.bin
SurfVx.bin
...
```

Note that surface velocities from subsequent `Lare3d` restart runs will be appended to the end of these if you run `Convert_SurfaceV_Files.py` again with `overwrite_outputs=False`. You can set the variable `remove_split_files=True` to delete the raw files afterwards.

Using the surface velocity to move fieldlines

Note: all calculations in this section use normalized units directly from `Lare3d`.

You may wish to use the saved surface velocity to advance a number of "footpoints" of magnetic field lines from one simulation time to another. The set of x, y coordinates of the footpoints is referred to as a "cage" (remember that they will be moved in the plane of constant z in which the surface velocity was saved). The initial cage can be structured into a grid, or chosen for aesthetic reasons.

Specify two pairwise lists in `Cage_Manual.py` and run it to produce `SurfaceCage0.bin` in your **Data** directory. Alternatively, run `Cage_FromQSL.py` to produce a cage from the grid written by `QSL Squasher`. To move the cage, you will need to modify the top of `Cage_Advance.F90`. You must specify which of the velocity components has been saved and at which times you would like the cage to be subsequently output. You will likely wish these times to coincide with the actual `SDF` files, in which case you should run `Times_From_dat.py` and copy in the result.

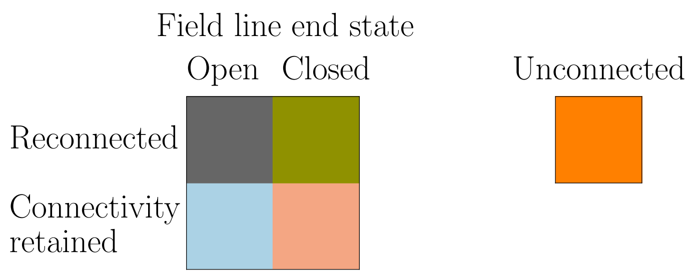
Now compile and run the program as follows (the compilation command is given at the top of the relevant file):

```
gfortran Permar_Functions_Fortran.F90 Cage_Advance.F90 -o Cage_Advance
./Cage_Advance
```

The cage will be written to files named such as `SurfaceCage1.bin`, where the index ≥ 1 corresponds to successive times specified (0 being the index of the original cage). Since any subsequent cage file run will overwrite these files, you should rename them manually, or automatically using `Rename_Cage_Files.py`. The cage can then be read, *e.g.* to be used as start points for fieldline integration, using the following function:

```
x_cage,y_cage=read_surface_cage_file(filename)
```

A more involved use of the routines which move fieldlines is calculating the "connectivity" of the latter. We classify the footpoints of magnetic field lines into one of five categories shown below in the color scheme provided with `Permar`. Blue and red correspond to field lines which have been open (connected at only one end to the lower simulation boundary) and closed (both ends connected to the lower boundary) since the beginning of the simulation. Orange field lines are unconnected to the lower boundary. More interestingly, the grey and brass-colored footpoints correspond to those fieldlines which are at the given moment in time open or closed, respectively, but which have undergone magnetic reconnection since some reference time.



The two categories of reconnected field lines are determined by moving a cage of fieldlines and comparing the start and end connectivity at the cage location. For such a connectivity map to be constructed, `QSL Squasher` must be used to generate a minimum of two Q maps for a comparison to be made. Once complete, modify the file `Connectivity_Map_Create.F90`, specifying `output_times` using the method above, specify paths to `QSL_files` and output paths to `connectivity_files`. Once compiled, the program can be executed with

```
./Connectivity_Map_Create
```

This will take quite a while to run (currently not multithreaded). The connectivity files can then be displayed using `Plot_Connectivity_Map.py` or `Animate_Connectivity_Map.py`.