Using the FVis module

Lars Frogner

Applies to version 1.1.3

Contents

1 Introduction

This guide describes the usage of the Python module `FVis.py`, which provides a simple way of saving and visualising the results of your fluid simulations. The most important aspects of the module should be covered here, but any questions or comments can be directed to lars.frogner@astro.uio.no.

2 Required packages

The module requires matplotlib and NumPy. Make sure that both are updated to the latest version.

For generating an mp4 file from an animation, FFmpeg is required. Windows users can download it from the linked site. In order for matplotlib to find the FFmpeg executable, you must open the source code (`FVis.py`), and at the top specify the path to the folder that FFmpeg was downloaded to (you'll see where to put it once you open the file). For the latest versions of Ubuntu, FFmpeg should already be installed, and you can leave the path specification commented out.

3 What the module can do

- Run your simulation for a given amount of time and save the resulting data to binary files.
- Read data previously saved by the module.
- Add more simulation data to existing files.
- Show a 1D or 2D animation of a quantity that can be derived from the stored data.
- Save the animation as an mp4 file.
- Plot the time evolution of the average value of a quantity (useful for checking if something is conserved).

4 Getting started

Download `FVis.py` from `https://github.com/lars-frogner/FVis` and save it to your working directory. At the top of your code, import the module by calling

```
import FVis
```

Then, below all your simulation code, create an instance of the visualisation class by writing

```
vis = FVis.FluidVisualiser()
```

`FluidVisualiser` is a class containing all the methods you need for saving simulation data and creating animations, and you will be using the instance `vis` for every action you perform with the module. You are now almost ready to save some data, which is done with the method `save_data`. This method will advance your simulation for you and output the required simulation variables (density, velocity etc.) to files with regular intervals. We will see how to do this in a second, but first there are a few requirements on your solver implementation that you need to know about.

5 Requirements on your solver

Suppose you have written a 1D solver for the Navier-Stokes equations with no energy equation. Your primary variables would be density `rho` and velocity `u`, with the pressure `P` as a secondary variable. All of these must be represented as numpy arrays. You will provide the `save_data` method with references to the arrays that you want to save.

You will also need to have a function or method, let's call it `step`, that can advance the simulation by one time step and update the content of the arrays. `step` can take no arguments; it must have access to the arrays either as class attributes (highly recommended) or as global variables (not recommended). Here are examples for both cases:

```
# *** Class approach ***

def step(self):

        # <Calculate derivatives, find time step, set BCs etc..>

        # Update attributes
        self.rho[:] = ...
        self.u[:] = ...
        self.P[:] = ...

        # Return time step
        return dt
```

```
# *** Global approach ***

def step():

        global rho, u, P # Get access to the global variables

        # <Calculate derivatives, find time step, set BCs etc..>

        # Update global variables
        rho[:] = ...
        u[:] = ...
        P[:] = ...

        # Return time step
        return dt
```

`step` will also be provided to `save_data`, which will call it in a loop to advance the simulation. The length of the time step must be returned, so that `save_data` can keep track of the elapsed simulation time. Notice also the brackets after `rho`, `u` and `P`. They are required for `save_data` to work. Here's why: Consider the two statements

```
arr1 = arr2
```

and

```
arr1[:] = arr2
```

The first statement basically says to take the memory allocated to `arr2` and label it `arr1`. The second one says to take the memory allocated to `arr2` and copy it's content into the memory allocated to `arr1`. See now why we have to use the second approach? Otherwise the references provided to `save_data` initially will no longer point to the correct data after `step` is called. (Note that it doesn't have to be `[:]`, you could for instance have to set the internal values (`[1:-1]`) and the boundary points (`[0]`, `[-1]`) separately.)

6 Saving data

Say we now want to run our hypothetical 1D solver (implemented as a class) for 1000 simulation seconds and save the contents of the arrays to file every other

second. The syntax for this would be something like this:

```
# Create an instance of your solver class
solver = MySolver(<arguments>)

# <Call any methods from solver required for setting initial
    conditions>

# Run simulation for 1000 seconds and save rho, u and P
vis.save_data(1000, solver.step, rho=solver.rho, u=solver.u, P=
    solver.P, sim_fps=0.5)
```

The first argument to `save_data` is the number of seconds to simulate. The second argument is the stepping function/method. Following that are a series of keyword arguments for the references to the arrays that are to be saved. Finally in this call we have specified the number of frames to save each second. A higher value for `sim_fps` thus gives a finer time resolution for the saved data. A detailed list of possible arguments to `save_data` can be found in the save_data section.

One particular keyword argument to `save_data` is worth mentioning in a little more detail here; the argument `sim_params`. You can use it for storing the values of all kinds of parameters that characterise your simulation run, like the number of grid points, the size of the simulation box, the viscosity etc. You do this by creating a dictionary where the keys are string representations of the parameter names, and the values are the numerical values used for the parameters. This dictionary is then given to the `save_data` method through the `sim_params` argument. There are two uses of this. The first is that you can have your parameters displayed on top of the animation, making it easier to analyse various simulation runs. The other is that you can retrieve the parameters later and use them for continuing the simulation from where it ended (see Adding more data).

For a 2D simulation you would also call `save_data` almost like above, only this time you would have a couple of additional variables in your simulation (like internal energy and one more velocity component), and references to these must be included in your call to `save_data`.

By default, the binary files containing the output data are saved in a time stamped folder inside your working directory, but you can also specify a custom name with the keyword argument `folder='<My custom name>'`.

Should you get tired of waiting and want to end the writing process earlier than planned, just do a keyboard interrupt (ctrl-C), and the program will finish up what it was doing and abort gracefully. Your data will be just fine.

7 Animating

Now that we have saved our precious data, it's time to visualise it. This is done with the method `animate_1D` (or the corresponding 2D version, `animate_2D`). Here is a call that creates an animation of the density.

```
vis.animate_1D('rho', folder='FVis_output_<yyyy_mm_dd_hh_MM>')
```

The first argument is a string describing the quantity to show. This can be any of the quantities that were saved (so `rho`, `u` or `P` for the case discussed above), or a derived quantity like horizontal momentum (`'ru'`) or pressure contrast (`'dP'`). A complete list of the possible quantities can be found in the Quantities that can be visualised section.

The keyword argument `folder` in the above call specifies the name of the folder to read the binary files from. In this example the data is read from some default time stamped folder. If you want to animate data that was saved with the same `FluidVisualiser` instance (in the same running of the program), the folder name doesn't have to be specified since it is already known to the instance.

There are a number of other keyword arguments available, all listed in the `animate_1D` section for `animate_1D` and the `animate_2D` section for `animate_2D`.

One thing to be aware of when creating 2D animations is that `animate_2D` by default assumes that your arrays are indexed like matrices, in other words that the first index specifies the row (and thus denotes height) and the second index specifies the column (and thus denotes width). If you have done it the other way around, add the keyword argument `matrixLike=False` in `animate_2D` so that your arrays can get transposed before they are shown.

8 Plotting time evolution of an average

If we for instance want to check whether mass is conserved in our simulation, we can use the `plot_avg` method. It will calculate the average of some quantity over the entire simulation region for each time step, and produce a plot of the resulting time evolution. A call to it can look like this

```
vis.plot_avg('rho', folder='FVis_output_<yyyy_mm_dd_hh_MM>')
```

All quantities that can be used with one of the animation methods (listed in the Quantities that can be visualised section) can also be used with `plot_avg`. See the `plot_avg` section for additional keyword arguments.

9 Temporary visualisation

If we are doing some quick and dirty testing of parameters and don't want to keep the data that gets produced, the method `delete_current_data` is our friend. As can be guessed from it's name, it deletes the data that was saved with the same instance of `FluidVisualiser` (again, in the same running of the program). Below is an example of it's usage.

```
# <Instantiate classes etc..>

# Save 200 seconds worth of data
vis.save_data(200, solver.step, rho=solver.rho, u=solver.u, P=
    solver.P)

# Animate the pressure
vis.animate_1D('P')

# Delete the data after the animation window is closed
vis.delete_current_data()
```

You will be asked to confirm the deletion in the terminal window, so that you don't accidentally delete an hour's worth of data just because you forgot to comment out the deletion command.

10 Adding more data

It is usually hard to know beforehand how long the simulation must run for in order to get to the interesting bits. If you have to start the simulation from scratch when you just need to see what happens next, a lot of time is wasted calculating the same things all over again.

One of the most handy features of this module is the ability to extend an existing simulation by adding more data to it. The tool that allows us to do this is the `get_last_data` method. A call to it looks like this:

```
arrs, params = vis.get_last_data('FVis_output_<yyyy_mm_dd_hh_MM>')
```

It takes as the only argument the name of the folder for the simulation we want to extend. The first return value, `arrs`, is a dictionary with an entry for each variable there is data for. The dictionary keys are just the string representations of their names (`'rho'`, `'e'`, `'P'` etc.), and the values are the corresponding arrays that were last added to the files. So to retrieve e.g. the last temperature array that was written in that simulation, we could write `T_last = arrs['T']`.

Provided that all the necessary variables were saved initially (at least as many as the number of primary variables), you can use the content of `arrs` as initial conditions for your solver to start from the point where the original simulation ended.

The second return value is also a dictionary, in fact the same dictionary that was provided to `save_data` as the `sim_params` argument when the data was first written. As long as you included all the relevant parameters in the `sim_params` dictionary when you called `save_data`, this information lets you make sure that the solver uses the same set of parameter values as was used originally.

Now that your solver is all set, the final step is to call `save_data` just like before, only this time with the keyword argument `appendMode` set to `True`. This lets the method know that you don't want to create new files for the upcoming simulation data, but rather append it to the end of some old files. Since it also needs to know which files to append the data to, you must specify this with the `folder` argument.

You can more or less automate the process of resetting your solver to the correct state by having it take all of it's parameters in the form of a dictionary in the first place. Then, when you create a new set of simulation data, use that dictionary as the `sim_params` argument to `save_data`. If you later want to extend that simulation, you can get the same dictionary back with `get_last_data`. Just feed it directly into your solver, and all parameters will be set to the correct values. Then it is just a question of setting the arrays in `arrs` as initial conditions, and you are good to go.

11 Argument descriptions

11.1 `FluidVisualiser`

Constructor.

- `printInfo=True`: Type: `bool`.
  Whether to print info about execution to the terminal.

11.2 `save_data`

Advances the simulation by a given amount of time and saves the relevant data to binary files.

- First argument: Types: `int`, `float`.
  Number of seconds to simulate.
- Second argument: Type: `callable`.
  Function/method for advancing the simulation by a time step and updat-

ing arrays of the primary and secondary variables. Must take no input, and return the time step length.

- `rho=None`, `u=None`, `w=None`, `e=None`, `P=None`, `T=None`: Type: `ndarray`.
  Arrays that get updated by the function/method given in the second argument. Their content will be saved to binary files regularly during the simulation run. They must all have the same shape. They specify density, horizontal velocity, vertical velocity, internal energy, pressure and temperature, respectively.

- `sim_fps=1`: Types: `int`, `float`.
  The number of times per simulation second to add the array content to file.

- `useDblPrec=False`: Type: `bool`.
  Whether to save the data as 64 bit float values rather than 32 bit.

- `sim_params=None`: Type: `dict`
  Dictionary with names (as dict keys) and the corresponding values of any parameters you use in your simulation. The key/value pairs will be stored inside the output folder, and can be displayed with the animation. The parameters can be retrieved later with the `get_last_data` method, so that you can use them to reconstruct the particular settings of that simulation should you decide to add more data to it.

- `appendMode=False`: Type: `bool`.
  Whether to add the new simulation data to the end of the binary files in the specified folder (the automatic folder name option (see below) will thus not be accepted in append mode). The set of inputted arrays must correspond exactly to the set of arrays in the specified folder, and the shapes of course have to match. The specified precision and the dictionary of simulation parameters will be ignored in append mode.

- `folder='auto'`: Type: `str`.
  Name of the folder to save the data in (or to append data in when append mode is activated). By default the output folder will be named `'FVis_output_<yyyy_mm_dd_hh_MM>'`, where the bracketed letters describe the date and time when the folder was created.

11.3 `animate_1D`

Creates an animation of the time evolution of a 1D simulation.

- First argument: Type: `str`.
  Which quantity to visualise. See list of quantities in the Quantities that can be visualised section.

- `folder='default'`: Type: `str`.
  Name of the folder to save read the data from. The default value can only

be used when the same instance has already been used to save data. Then the folder that the data was saved to will automatically be used.

- `extent=[0, 1]`: Type: `list`.
  List specifying the spatial coordinate of the left and right edge of the simulation area. Used to label the horizontal axis of the animation. A third element in the list can also be included, which must be a string specifying the unit that the coordinates are given in (e.g. `'Mm'` if the numbers are given in megametres).

- `anim_fps='auto'`: Types: `int`, `float`.
  The number of times per simulation second to show an animation frame. Will not in practice be larger than the fps used to save the data, but any value is accepted. Default uses the same fps as the data was saved with.

- `showDeviations=True`: Type: `bool`.
  Whether to show labels with the relative difference between the current total mass and/or energy (whatever is available) and the initial total mass and/or energy.

- `showParams=True`: Type: `bool`.
  Whether to display the simulation parameters that were given with the `sim_params` argument when the data was saved.

- `height=7`: Types: `int`, `float`.
  The animation figure height.

- `aspect=1.1`: Types: `int`, `float`.
  The aspect ratio of the animation window (width/height).

- `title='auto'`: Type: `str`.
  Figure title to use. Default value will use the name of the quantity that is being visualised.

- `save=False`: Type: `bool`.
  Whether to save the animation as an mp4 file rather than showing it.

- `anim_time='auto'`: Types: `int`, `float`.
  The number of simulation seconds the animation will run for when saving to an mp4 file. Default value uses the number of seconds spanned by the saved data. If set to larger than this, the animation will restart from the initial time.

- `video_fps=30`: Types: `int`, `float`.
  The fps to use for the mp4 video. The difference between this and the animation fps option is that the latter describes the number of frames to show per simulation time, while the former describes the number of frames to show per real time. The length of the video in real seconds will be `anim_time*anim_fps/video_fps`.

- `video_name='auto'`: Type: `str`.
  Name to use for saved mp4 file. Don't include the .mp4 extension. Default uses the name of the data folder.

11.4 `animate_2D`

Creates an animation of the time evolution of a 2D simulation.

- First argument: Type: `str`.
  Which quantity to visualise. See list of quantities in the Quantities that can be visualised section.

- `matrixLike=True`: Type: `bool`.
  Whether the 2D arrays are indexed like matrices or not. If `[i, k]` refers to column `i` and row `k` in your arrays, set to false.

- `backgrounds=None`: Type: `dict`.
  Used for specifying alternative initial states $\phi(\mathbf{r}, 0)$ to use when visualising any of the contrast quantities in the Quantities that can be visualised section. If set to `None`, the earliest state in the current set of visualisation data will be used. Otherwise, the argument must be a dictionary where the keys are the names of the relevant quantities (like `'rho'`, `'T'` etc.), and the values are 2D arrays representing the corresponding initial states.

- `folder='default'`: Type: `str`.
  See `animate_1D` description.

- `extent=[0, 1, 0, 1]`: Type: `list`.
  List specifying the spatial coordinate of the left, right, lower and upper edge of the simulation area. Used to label the axes of the animation. A fifth element in the list can also be included, which must be a string specifying the unit that the coordinates are given in (e.g. `'Mm'` if the numbers are given in megametres).

- `anim_fps='auto'`: Types: `int`, `float`.
  See `animate_1D` description.

- `showDeviations=True`: Type: `bool`.
  See `animate_1D` description.

- `showParams=True`: Type: `bool`.
  See `animate_1D` description.

- `showQuiver=True`: Type: `bool`.
  Whether to show a quiver plot of the velocity field on top of the animation. Nothing will happen if not both velocity arrays are available.

- `quiverscale=1`: Types: `int`, `float`.
  Scaling factor for the quiver arrows. An initial scaling is done based on an

estimation of the sound speed. This is a modifier for that scaling, so using e.g. 2 will double the length of the arrows. The arrow scale is displayed to the top right in the animation window. It shows the speed that an arrow represents if it has a length of 1 Mm.

- `N_arrows=20`: Type: `int`.
  The number of quiver arrows to use in the vertical direction. The number in the horizontal direction is then found from the aspect ratio of the array shape. Warning: Using a lot of arrows can have a significant impact on performance.

- `interpolation='none'`: Type: `str`.
  The interpolation type to use for the animation frames. Examples are `bicubic` and `spline16`. By default no interpolation will take place. Warning: Interpolating can significantly impact performance, and can also hide useful information.

- `cmap='jet'`: Type: `str`.
  The color map to use for the animation frames. Examples are `viridis` and `gray`.

- `height=7`: Types: `int`, `float`.
  See `animate_1D` description.

- `aspect='equal'`: Types: `int`, `float`.
  The aspect ratio of the animation window (width/height). Default value uses the same aspect ratio as the array shape.

- `title='auto'`: Type: `str`.
  See `animate_1D` description.

- `save=False`: Type: `bool`.
  See `animate_1D` description.

- `anim_time='auto'`: Types: `int`, `float`.
  See `animate_1D` description.

- `video_fps=30`: Types: `int`, `float`.
  See `animate_1D` description.

- `video_name='auto'`: Type: `str`.
  See `animate_1D` description.

11.5 `animate_energyflux`

Creates an animation of the horizontally averaged vertical energy flux.

- `folder='default'`: Type: `str`.
  See `animate_1D` description.

- extent=[0, 1, 0, 1]: Type: list.
  See animate_2D description.
- anim_fps='auto': Types: int, float.
  See animate_1D description.
- showParams=True: Type: bool.
  See animate_1D description.
- height=7: Types: int, float.
  See animate_1D description.
- aspect=1.1: Types: int, float.
  See animate_1D description.
- title='auto': Type: str.
  See animate_1D description.
- save=False: Type: bool.
  See animate_1D description.
- anim_time='auto': Types: int, float.
  See animate_1D description.
- video_fps=30: Types: int, float.
  See animate_1D description.
- video_name='auto': Type: str.
  See animate_1D description.

11.6 plot_avg

Plots the time evolution of the average of a given quantity.

- First argument: Type: str.
  Which quantity to measure average of. See list of quantities in the Quantities that can be visualised section.
- folder='default': Type: str.
  See animate_1D description.
- measure_time='auto': Types: int, float.
  The number of seconds of simulation time that will be included in the plot. Default value uses the number of seconds spanned by the saved data. If set to larger than this, the measuring will stop automatically.
- showTrendline=False: Type: bool.
  Whether to show a simple, linear trend line for the time evolution.

11.7 `delete_current_data`

Deletes the data saved by ***`save_data`*** *in this instance.*

11.8 `get_last_data`

Reads the last arrays in the files of the given folder and returns them in a dictionary.

- First argument: Type: `str`.
  Name of the folder to return data from.

12 Quantities that can be visualised

- `'rho'`: Mass density, $\rho(\mathbf{r}, t)$, [kg/m^3].
- `'drho'`: Mass density contrast, $(\rho(\mathbf{r}, t) - \rho(\mathbf{r}, 0))/\rho(\mathbf{r}, 0)$, [unitless].
- `'u'`: Horizontal velocity, $u(\mathbf{r}, t)$, [m/s].
- `'w'`: Vertical velocity, $w(\mathbf{r}, t)$, [m/s].
- `'e'`: Internal energy density, $e(\mathbf{r}, t)$, [J/m^3].
- `'de'`: Internal energy density contrast, $(e(\mathbf{r}, t) - e(\mathbf{r}, 0))/e(\mathbf{r}, 0)$, [unitless].
- `'es'`: Specific internal energy, $e(\mathbf{r}, t)/\rho(\mathbf{r}, t)$, [J/kg].
- `'P'`: Pressure, $P(\mathbf{r}, t)$, [Pa].
- `'dP'`: Pressure contrast, $(P(\mathbf{r}, t) - P(\mathbf{r}, 0))/P(\mathbf{r}, 0)$, [unitless].
- `'T'`: Temperature, $T(\mathbf{r}, t)$, [K].
- `'dT'`: Temperature contrast, $(T(\mathbf{r}, t) - T(\mathbf{r}, 0))/T(\mathbf{r}, 0)$, [unitless].
- `'v'`: Speed, $\sqrt{u(\mathbf{r}, t)^2 + w(\mathbf{r}, t)^2}$, [m/s].
- `'ru'`: Horizontal momentum density, $\rho(\mathbf{r}, t)u(\mathbf{r}, t)$, [kg/sm^2].
- `'rw'`: Vertical momentum density, $\rho(\mathbf{r}, t)w(\mathbf{r}, t)$, [kg/sm^2].
- `'rv'`: Momentum density, $\rho(\mathbf{r}, t)\sqrt{u(\mathbf{r}, t)^2 + w(\mathbf{r}, t)^2}$, [kg/sm^2].
- `'eu'`: Horizontal energy flux, $e(\mathbf{r}, t)u(\mathbf{r}, t)$, [W/m^2].
- `'ew'`: Vertical energy flux, $e(\mathbf{r}, t)w(\mathbf{r}, t)$, [W/m^2].
- `'ev'`: Energy flux, $e(\mathbf{r}, t)\sqrt{u(\mathbf{r}, t)^2 + w(\mathbf{r}, t)^2}$, [W/m^2].