AST3310 Project 4
Visualisation module user guide

Lars Frogner

August 28, 2016

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.

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.
- 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` and keep it in the same directory as your simulation programme. 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 the only (public) class in the module, 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`. But first there are a few requirements on your implementation that you must 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

```
# 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,
    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 `fps` thus gives a finer time resolution for the saved data. A detailed list of possible arguments to `save_data` can be found here.

By default, the binary files containing the 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>'`.

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 shows 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 here.

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 here for `animate_1D` and here for `animate_2D`.

One thing to be aware of when creating 2D animations is that `animate_2D` assumes that your arrays are indexed like matrices, in other words that the first index refers to the row (height) and the second index refers to the column (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. Also make sure that increasing an index corresponds to increasing the position coordinate, so if your simulation box spans $x \in [0, L_x], z \in [-L_z, 0]$, the index pair [0, 0] refers to the position $(x, z) = (0, -L_z)$.

8 Plotting time evolution

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 here) can also be used with `plot_avg`. See here 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 Method argument descriptions

10.1 `FluidVisualiser`

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

10.2 `save_data`

- 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 updating 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, and that are to be saved. They specify density, horizontal velocity, vertical velocity, internal energy, pressure and temperature, respectively. They must all have the same shape. For 1D, whether `u` or `w` is specified determines the assumed direction (whether the horizontal axis should be labeled with `x` or `z`). `x` is assumed if no velocity (or both) is specified.

- `Lx=None`, `Lz=None`: Types: `int`, `float`.
  Respectively width and height of simulation area, in metres. Used for labeling the animation. The area is assumed to span $x = [0, L_x]$ and/or $z = [-L_z, 0]$. For 2D, if not both are are provided, $L_z = 1$ Mm is assumed and L_x is calculated from the aspect ratio of the array shape. For 1D, if the length of the assumed direction isn't provided it will be assumed to be 1 Mm.

- `sim_params=None`: Type: `dict`
  Dictionary with parameter names (as dict keys) and their corresponding values (as dict values). The key/value pairs will be displayed on top of the simulation figure.

- `t_init=0`: Types: `int`, `float`.
  What time the simulation is considered to start at.

- `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.

- folder='auto': Type: str.
  Name of the folder to save the data in. By default the 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.

10.3 animate_1D

- First argument: Type: str.
  Which quantity to visualise. See list of quantities below.
- 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.
- height=7: Types: int, float.
  The animation figure height.
- fps=1: 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.
- 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.
- save=False: Type: bool.
  Whether to save the animation as an mp4 file rather than showing it.
- video_time='auto': Types: int, float.
  The number of seconds of simulation time that will be included in the saved 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.
- 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.

10.4 animate_2D

- First argument: Type: str.
  Which quantity to visualise. See list of quantities below.

- `folder='default'`: Type: `str`.
  See `animate_1D` description.
- `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.
- `height=7`: Types: `int`, `float`.
  See `animate_1D` description.
- `fps=1`: Types: `int`, `float`.
  See `animate_1D` description.
- `showDeviations=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.
- `save=False`: Type: `bool`.
  See `animate_1D` description.
- `video_time='auto'`: 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.
- `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`.

10.5 `plot_avg`

- First argument: Type: `str`.
  Which quantity to measure average of. See list of quantities below.
- `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 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].