Hand-out for FARGO3D code

Sareh Ataiee sarehataiee@um.ac.ir

In this session, we will employ the grid-based MHD code `FARGO3D`, that is basically written to simulate planet-disc interactions, and plot the outputs using the provided script. You are supposed to have a working C compiler and a Python3.10 (or higher) installation along with the standard scientific python packages such as `numpy` and `matplotlib`. You can use your favourite `python` environment, however `Anaconda` is highly recommended.

1 Before coming to the session

1.1 Instal and test `FARGO3D`

First of all, you need to get the code from its git repository:

```
git clone https://github.com/FARGO3D/fargo3d.git
```

Now you should see the code directory as `$HOME/fargo3d/`. Here is the code's documentation. Please check it out to understand the structure of the code. Then, try to compile and do the first run as is explained here.

1.2 Using `fargo3dplot`

There are different ways to visualise the outputs of the code. You can use the python scripts that comes with `FARGO3D` and is in the `utils` directory, write your own script in any language you prefer, or download the tiny `fargo3dplot` package which will be used in the remaining of this manuscript. Preferring the later, please download the package from here, or clone via git as

```
https://github.com/sarehataiee/fargo3dplot.git
```

Then, add the address of where the `fargo3dplot` directory is stored to your `$PYTHONPATH` to ease importing the package. For example, in case of using a bash

shell and assuming that the `fargo3dplot` directory is located in your home inside the `fargo3d` directory, you need to add the below line to your `$HOME/.bashrc` file:

```
export PYTHONPATH="$HOME/fargo3d:PYTHONPATH"
```

After that, either open a new terminal or type `source $HOME/.bashrc`[1]. Now, if you type `from fargo3dplot import *`, in your favourite python environment (such as `ipython console`, `spyder`, or `Jupyter-notebook`), the package should be imported without any problem.

The plotting script, that would be used in the following exercises, are included in the workshop git repository.

2 During the session

After you get your successful first run, we can go through three below examples: a) A 1D shock problem named as Sod shock tube in Sec. 2.1, b) making the fluid unstable as in Sec. 2.2, and c) a growing Jupiter in a protoplanetary disc in Sec. 2.3.

2.1 Sock tube (Sod problem)

Inside the directory `setups/sod1d`, there are several files that describe the initial condition for the code. Interestingly, some of them are empty meaning that those parameters are as default and we do not need to modify them. The initial condition for the fields are set in `condinit.c`, the boundary condition is given in `sod1d.bound`, and the grid properties and some other parameters are in `sod1d.par`. Have a look at them to see how much you understand!

Now, follow the below steps to make your 1D shock problem:

1. Edit the `sod1d.par` file and change `DT` to 0.02 and `Ntot` to 10 in order to have more outputs and finer temporal resolution.
2. Make the setup using `make SETUP=sod1d`
3. Run the code by `./fargo3d setups/sod1d.par`. It should be finished in a blink of eye.
4. Make the animated plots using the scrip named sod1dplot.py (it is included in the scripts you downloaded from the workshop repository) in your favourite python environment. The dashed lines are the exact analytical solution and the solid lines are the code outputs. What do you see?

[1] If you are using `spyder`, you need to add this path via `Tools-> PYTHONPATH manager`.

5. Search in the summary files in `outputs/sod1d` to find the used CFL number. How much is it?
6. Change the CFL number by adding the line `CFL 1` to the `sod1d.par`, and rerun the code. What did happen? What if you set it to 0.01?
7. Return the CFL number to its default value by removing the previously imported line and now decrease your spacial resolution by setting `NZ` to 100 and rerun the code. How is the output different from the first test?
8. With `NZ` equals to 1000, increase the `Ntot` to 100, run the code again, and plot the results. What is happening in later times? What if you add `PeriodicZ YES` in your par file? It forces the z boundary condition to be periodic.
9. Remove the periodic boundary condition from the `.par` file and now modify the `sod1d.bound` and `sod1d.bound.0` file such that your velocity along the z axis has `SYMMETRIC` boundary. Make the code again and run the test. What has changed? Check the boundary section of the manual to understand what happens in the `STMMETRIC` boundary.

Caution: The boundary condition in x direction is always `periodic` as the code is basically designed for modelling protoplanetary discs.

2.2 Kelvin-Helmholtz instability

In this example, you are going to make Kelvin-Helmholtz instability. The setup does not exist in the code default setups but you can download it from the workshop repository. Extract and move the `KH` directory into your `setup` directory. Read and try to understand the files inside the directory. Note that all of the setup file names (except `condition.c`), the name of the setup and output directory in the `.par` file, and name of the setup directory itself should be the same.

Make the setup, run the code, and make the animation from the outputs using the provided script.

Now, it is your turn to explore! You can, for example, change the perturbation amplitude, phase, and/or the velocity gradient between the two layers and see how the outputs change.

2.3 A growing Jupiter

Since `FARGO3D` is written with the purpose of simulating protoplanetary discs, as the last example, let's try to make a growing Jupiter in a disc. If you have more than a free core on your computer, it would be faster to run the code in parallel (it would need openMPI though), otherwise you can go on with the sequential run.

To make the setup directory, it is easiest to copy one of the default setup direc-

tories and then change it as you need. Type the below lines in your terminal to make a setup named `gjup`.

```
cd ~/fargo3d/setups/          # Navigate to the setup directory
cp -r fargo/ gjup             # Copy the fargo directory as gjup
cd gjup                       # Enter gjup
# Rename the files names to the corresponding setup name
for i in fargo.*; do mv $i gjup."${i#*.}"; done
```

Now, we need to modify some of the files inside the `gjup` directory.

- In the `.par` file, change the name of the setup and the output directory to `gjup`. So, we would have

  ```
  Setup                   gjup
  OutputDir               @outputs/gjup
  ```

- To have a mass growing planets, add the below line to your `.par` file. It tells the code to increase the mass of the planet from 0 to $1 M_{\rm Jup}$ in 20 orbits of the planet.

  ```
  MassTaper               62.83
  ```

  Firstly, clean the code, and then make the setup, this time in parallel if you have enough core, with

  ```
  make clean
  make SETUP=gjup para
  ```

  and run the simulation using

  ```
  mpirun -np 4 ./fargo3d setups/gjup/gjup.par
  ```

  I used 4 processors to run my simulation in parallel but you can use any number that your computer is capable of. After the simulation is over, use the provided script and make the output animation. What do you see?

  For the next step, we want to to mess up with the simulation to learn more. So, either make another setup directory, or simply, copy your outputs somewhere else for the future comparison. Comment the line
  `FARGO_OPT += -DSTOCKHOLM`
  in your `.opt` file by adding a `#` to the beginning of the line. Make the code, run the simulation again, and plot the outputs. Do you see any changes?

  To better see the reflections from the boundaries, replace `jupiter.cfg` in your `.par` file with `earth.cfg`, and run the code. When plotting the results, change `vmin` and `vmax` to -0.01 and 0.01 respectively. Could you follow the

reflected waves? Uncomment the line `FARGO_OPT += -DSTOCKHOLM` in your `.opt` file, make the code, run the simulation, and plot the results. Can you guess what this line does?

As your last exercise, double the resolution both in X and Y in the `.par` file and run the code again. What does happen? Can you say why?

Have fun!