MECH4480: Notes on sphere simulations

From lecture on 8 October 2018

These notes capture the demonstration in class on Monday 8 October, 2018. The demonstration considered the supersonic flow over a sphere that is problem 1 on the compressible flow tutorial sheet. In this demonstration, we discussed how to use some features of the Eilmer simulation code to give us better diagnostics of the flow, aid in post-processing, and some tricks to speed up simulation time. I have placed my "final" version of the `sphere.lua` file with these notes on blackboard.

1 Setting history points to record flow at select locations

When using Eilmer, we can take complete snapshots of the flow field development during the simulation. The frequency of those snapshots is controlled with `config.dt_plot`. However, if we want a fine-grained time resolution of the flow properties at some select points, the whole-of-flow-field snapshot approach is not appropriate: a) it does not solve the problem of getting the data at a specific location; and b) it is wasteful on disk space to record a whole flow field when we are only interested in specific points. This is why Eilmer offers the option to record detailed flow history at some select points in the flow field.

When using the history recording feature, we make two changes to our input script:

1. Set as many history points as one likes using the `setHistoryPoint` function.
2. Configure the `config.dt_history` value to an appropriate time sampling increment.

The `setHistoryPoint` function requires that you pass it the coordinates of the point of interest. At simulation start-up, Eilmer will find the nearest cell-centre to that given coordinate. So note, the located cell may not (most likely will not) have a cell-centre that precisely matches your given x and y coordinates; but it will be very close, depending on your grid.

In our example of the sphere, I added these lines to the input script `sphere.lua`:

```
setHistoryPoint{x=A.x, y=A.y}
setHistoryPoint{x=B.x, y=B.y}
config.dt_history = 1.0e-7
```

Note we would need to re-run the prep stage and the run stage of the simulation in order to have that history data recorded.

After re-running the simulation, we should have two history files recorded in the `hist/` subdirectory. For my grid, those two files are `sphere-blk-0-cell-1199.dat`

and `sphere-blk-0-cell-29.dat`. How do we know which file belongs to which history cell? The easiest approach is by inspection. Just open the history files and look at the x (column 2) and y (column 3) values to find which file associates with which history point. A shortcut to opening only a portion of a file in linux is to use the `head` command. Here I use `head -2` on one of the history files so that I can just see the first two lines of the file:

```
> head -2 hist/sphere-blk-0-cell-29.dat
```

I see that the x and y values are very close to (0,0) and so I deduce that this file is associated with point A in my grid.

The history files are plain text files. They contain columns of data separated by whitespace. The first line of the data file is a comment line that describes what data is in each column. You can import this file into your favourite plotting program and produce plots of key flow field properties to see how they change with time.

1.1 Using history points for flow diagnostics

That's the mechanics of how we set history points. Why is that useful and how does it help with diagnostics of our simulations? For steady-state simulations, the history points are a good way of determing if we marched the solution forward in time for long enough to reach the steady state. We should be careful to pick meaningful locations for the history points: some points in the flow field settle at earlier times than others. We might be misled if we determined steady flow from a history point that settles relatively early in the flow field. It takes some experience and intuition to work out which points in a flow field settle earlier than others. However, you can also apply some physical reasoning. Points in a flow field further downstream usually take longer to settle than points upstream (particularly in supersonic flow because of the directionally-limited influence in those flows). Also, for viscous flows, separated regions typically take a long time to settle — if, in fact, they do settle.

In this particular case, I would expect the shoulder point at B to take a little longer to settle than the nose point at A. I can plot both values and inspect their histories of pressure and temperature to determine if a steady-state has been reached. My determination of steady-state is by inspection: I'm looking for region in the history where the flow value no longer changes with time.

Another use for history points is for comparison to experiments. If an experiment has pressure transducers or temperature gauges mounted at a particular location on the model surface, then we can use history points, set at those same locations, to act as virtual data recorders. This facilitates direct comparison to time-sampled experimental data.

2 Extracting surface properties and loads

History points record history at specific points in the flow field. We can also record the history at complete surfaces at the edge of the domain. Some surfaces have special significance such as walls and bodies of a model. As such, we might like more than just the flow properties: we would also like to know the forces and heat loads at those locations. In Eilmer, we do that by using the `config.compute_loads` feature. To use this feature, we make three modifications:

1. Set the `config.compute_loads` option on.

2. Set a time sampling increment for the loads computation and storage to disk: `config.dt_loads`.
3. Set the `group` tag on the selected boundaries to `''loads''`.

Let's expand on that last point. All boundary conditions in Eilmer can be associated with a `group`. We can put multiple boundaries in the same `group` by setting that parameter for a boundary condition. There is a special string `''loads''`. If the string `''loads''` appears anywhere in the `group` name, then Eilmer will identify that boundary condition as one which will have its surface loads computed and stored. It does that computation and storage at time increments of `config.dt_loads`.

I made two additions in the `sphere.lua` file to enable the loads calculation. I added lines:

```
config.compute_loads = true
config.dt_loads = 1.0e-5
```

and I altered the `east` boundary condition to associate it with the `''loads''` group. I chose the `east` boundary because it represents the body of the sphere.

```
blk0 = FluidBlock:new{grid=grid0, initialState=inflow,
                      bcList={west=InFlowBC_Supersonic:new{flowState=inflow},
                              north=OutFlowBC_Simple:new{},
                              east=WallBC_WithSlip:new{group="loads"}}
}
```

Just like the history points, we need to re-prep and re-run Eilmer for get this loads output. For larger simulations, you should carefully think in advance about which history and surface loads you would like to record because you cannot get those values after the event if you failed to configure them. This is also why we suggest starting at small scale where you get quick feedback on your simulations to sort out exactly what you would like to record.

After re-running the simulation, you will have some loads files at various timestamps in the `loads/` subdirectory. If you are interested in steady-state, you would typically grab the *last* loads file. That file, like the history file, is plain text: columns of data, separated by whitespace, with some header information marked as comments.

A final note on the recording of loads: when you restart simulations in Eilmer, **the loads files are not overwritten.** Instead, Eilmer will continue to add more timestamps in that loads directory. This behaviour might seem a bit odd. For most other things, Eilmer overwrites the files. This behaviour was deliberately chosen and implemented to help with large simulations that run over a long time. In those cases, we may have to stop and re-start Eilmer several times. It is more convenient that Eilmer **does not** automatically wipe out loads files in those cases. What this means for you is that you might need to clean out the loads directory when you are re-starting from scratch but continue working in the same directory:

```
> rm -rf loads/
```

2.1 Using surface loads

The surface loads file gives values on the surface associated with the cell interfaces lying on the surface. These are small pieces of information on an interface-by-interface basis. You would need to integrate this information if you wanted an

integral quantity for the surface. For example, to compute the pressure drag, you could sum the pressure times area contributions. We would only be interested in the x-component for this sphere simulation, since x is aligned with the direction of flight and drag is defined in the direction opposite to the direction of flight. We can use the component of the normal in the x-direction to account for the orientation of the interface on the surface. In terms of our surface-loads file, we could sum the contributions from columns:

$$D = \sum_{i=0}^{n-ifaces} \sigma_i \times area_i \times n.x_i$$

Note we use the formal notation of a normal stress (σ) in the loads file. The normal stress is effectively the pressure.

3 Restarting a simulation

The third feature we looked at in the class demonstration is how to restart a simulation in Eilmer. We might do this because the simulation was interrupted by a computer shutdown, or we might want to pick up a completed simulation but march it further forward in time (if say, steady-state were not reached). In the latter case, it will be important to set a new `max_time` value. You should edit the `.control` file directly to set a new `max_time`. You will need to be very pedantic in keeping the syntax correct in the `.control` file. This file format is not tolerant to syntax errors.

We used the `--tindx-start` option on the `e4shared` command line to restart a simulation from a given time index. This process is all adequately described in the Eilmer manual and you are referred to Section 4.1.4 in the manual for more details on restarting a simulation.

`http://cfcfd.mechmining.uq.edu.au/eilmer/pdfs/eilmer-user-guide.pdf`

4 Warm-starting: using a previous completed simulation as initial condition for a new simulation

In the simulations so far, we marched forward in time from a somewhat arbitrary (but stable) selection of initial condition. As good practitioners of CFD, we would like to repeat the calculation on successively refined grids to check that our extracted results are converging as we control for numerical error. When starting simulations on refined grids, we need not start again from scratch in terms of the initial condition. In other words, that initial transient of the shock entering the domain, reflecting against the body and finding its steady-state position is not of much interest to us for steady-state results. We can shortcut this by taking the coarse grid solution and using it as a starting point for the finer grid calculations. We can use the `FlowSolution` object in Eilmer to facilitate doing a "warm" start. I call this a warm start because we are starting our calculation with most of the flow features in place. The flow field will evolve a little from this position. Although that was the steady representation on the coarse grid, flow will evolve slightly in response to a refined grid.[1]

When using this technique, it's useful to set up a directory structure of the form: `coarse`, `medium`, `fine`. We'll assume we ran and successfully completed a simulation in the `coarse/` directory. In the `medium` directory, I'm going to copy the input

[1] Think about why the steady-state flow field alters as the grid is changed.

files from the `coarse` directory. I will make three changes to the `sphere.lua` file in the `medium`:

1. Increase the cell count by some factor (hopefully systematic) to get a refined grid.
2. Initialise a `FlowSolution` object that reads in the previously completed coarse grid solution.
3. Set the `initialState` in the block configuration to use that `FlowSolution` object.

Increasing the cell count is straightforward. In this case, I doubled the number of cells in each direction. It's not alway necessary to double the number of cells. It is helpful though to do a systematic refinement of the grid. This makes it a little easier to reason about the spatial convergence behaviour.

The additon I made to use the `FlowSolution` object is shown here:

```
initial = FlowSolution:new{jobName='sphere', dir="../coarse",
                           tindx='last', nBlocks=1}
```

The parameters in the `FlowSolution` object refer to the *old*, previously run solution, that is, the coarse grid solution in this case. Note because of our directory layout, I used the relative path to the coarse grid with respect to the present working directory: `''../coarse''`. You will need remember (or look up) how many blocks you used in the old solution so that you can set the `nBlocks` parameter. Section 4.5 in the manual explains the full details of those parameters.

Having set that `initial` object, we need to make use of it. We do that in the block configuration where we instruct Eilmer to use that `initial` object as a means to populate the initial state of the block (or blocks):

```
blk0 = FluidBlock:new{grid=grid0, initialState=initial,
                      bcList={west=InFlowBC_Supersonic:new{flowState=inflow},
                              north=OutFlowBC_Simple:new{},
                              east=WallBC_WithSlip:new{group="loads"}}
}
```

We can check that's working by doing a prep-stage immediately followed by a post-processing step to produce a VTK file for Paraview. When we inspect the *initial* flow field, we should see flow already in place, ready to start our simulation marching forward in time.

You might like to make a final change to your input file. Given that we have this flow field mostly in place, we can cut down the length of simulated time required. In my example, I dropped the number of flow times from 10 down to 3. I defined a flow time as the sphere radius divided by the inflow speed.

5 Running simulations in parallel

As our simulations get larger, running on a single CPU is no longer feasible: it would take far too long to get an answer. The solution in CFD (and in many many other domains) is to do our simulation across multiple CPUs in a coordinated and parallel manner. We can butcher the addage "many hands make light work" and think instead that "many hands make fast work".

At the start of semester, I did not deploy the cloud VMs with parallel execution capability for Eilmer. We can fix that by installing a package and compiling Eilmer for parallel execution. First, to install the required software package:

```
> sudo apt install libopenmpi-dev
```

Secondly, building Eilmer to use the MPI library requires a special compile directive. That is done so:

```
> cd
> cd dgd/src/eilmer
> make WITH_MPI=1 install
```

If that completes successfully, you will have a new executable in your `dgdinst/` directory: `e4mpi`.

So that sets up our execution environment for running simulations in parallel. We now have an MPI environment and the Eilmer parallel executable `e4mpi`. The next step is to consider how we will split up our simulation to make use of parallel processing. It is easiest in Eilmer to think of our units of processing corresponding to the blocks in our domain. Presently, our simulation only has one block. It is not a good candidate for running in parallel. In fact, we can't get any parallel benefit on a single block.

We could go back to our sphere setup and break it into a number of blocks. It wouldn't be so hard to split this simulation up into four blocks. Then we would be ready to simulate using 4 CPUs. However, if tomorrow, someone gives you 8 CPUs to use, then it's annoying to go back to your script and re-work it into 8 blocks. To overcome this annoyance and to give you greater flexibility, Eilmer provides a function to automatically slice a large block into a number of smaller blocks. This facilitates use of parallel processing. It also decouples the concerns of your blocking based on geometry and the concerns of blocking for parallel processing. The function to help us with this is `FluidBlockArray`.

Here is how I use it in the script, and then I'll discuss it below. We will replace the `FluidBlock:new` portion of the block definition with the `FluidBlockArray`:

```
blk0 = FluidBlockArray{grid=grid0, initialState=initial, nib=1, njb=4,
                       bcList={west=InFlowBC_Supersonic:new{flowState=inflow},
                               north=OutFlowBC_Simple:new{},
                               east=WallBC_WithSlip:new{group="loads"}}
}
```

What differs between a single `FluidBlock` and using the `FluidBlockArray` is that we now pass it some extra parameters `nib` and `njb`. These represent the number of blocks we would like to carve in the i logical direction and the j logical direction. In this case, I set `nib=1`. Between the shock and the body, I would like layer of blocks only one block wide. For `njb`, I set the value to 4. I have asked for 4 blocks to be stacked in the direction tangent to the sphere surface.

There is another subtle but important difference between `FluidBlock` and `FluidBlockArray`: the former is an *object* the latter is a *function*. So you will note that `':new'` bit got dropped. All of our objects have the `':new'` pattern; our functions do *not*.

Our next steps are to prep this simulation and run. If things have been done correctly, you will see the prep stage report the construction of 4 blocks. The prep stage is identical to earlier: we use `e4shared`.

```
> e4shared --job=sphere --prep
```

The run stage is different now for a parallel simulation. We will need to use the `mpirun` launcher and `e4mpi` instead of `e4shared`. The example for a 4-block simulation is:

```
> mpirun -np 4 e4mpi --job=sphere --run
```

Note the first part of this command is generic instructions for MPI. By first part, I am referring to `mpirun -np 4`. The second part of the command starts with `e4mpi`. Everything in that second part follows the usual Eilmer options that one can pass at the command line.

The post-processing stage, like the pre-processing stage, works exactly like you are used to: you will use `e4shared` to do any desired post-processing.

So the only difference was at run-stage. We needed to use the `mpirun` launcher and the `e4mpi` parallel version of the eilmer executable.

There should be some performance benefit to running this simple simulation in parallel. You can view the parallel execution of the program by opening another terminal and using `top` or `htop` process monitoring tools. You would do that *while* your simulation is running.