COCONUT user documentation

By KU Leuven, Centre for Mathematical Plasma Astrophysics

,

Contents

1. Introduction

This is a user documentation of COCONUT, the COolfluid COronal uNstrUcTured global coronal modelling code developed at the Centre for mathematical Plasma Astrophysics. COCONUT is a 3D magnetohydrodynamics (MHD) code based on the Finite Volume (FV) method, programmed and available within the COOLFluiD[1] framework. The purpose of the code is to resolve the dynamics of the plasma flow in the solar corona up to ~ 0.1 AU, where it can be coupled to heliospheric software. The code is rapid compared to the state-of-the-art, see Ref. [16], and its results have been validated for a large variety of test cases, see, e.g., Ref. [12] and Ref. [2].

First and foremost, we start with a word of caution. COCONUT is a rather complex computational fluid dynamics (CFD) solver that, through a variety of internal and external libraries, plugins and user inputs, solves large systems. Just like any other CFD code, COCONUT is not a "press a button and solve" software and requires deeper understanding of how the corresponding physics and numerics should be set up to achieve convergence and physically accurate solutions. This is one of the reasons why this document is rather extensive - as it does contain some of the necessary basics to develop this understanding.

To keep this document constrained in length, however, we will not go into the finer details and all of the justification of the physics and numerics that are used by COCONUT. Where justifications and further details might be useful, references will be made to the corresponding publications or to the thesis in Ref. [3], the latter of which describes the code development in more detail (but without the more practical information).

The reader is encouraged to read the following publications as examples of which kinds of simulations can be carried out with the code, see Figure 1.

- basic, polytropic, steady-state runs: in Ref. [16], Ref. [12], Ref. [15]
- polytropic time-accurate runs of evolving coronal mass ejections: in Ref. [13], Ref. [10],
- full-MHD coupled to heliospheric software: in Ref. [2], and
- multi-fluid runs with ion-neutral modeling: in Ref. [6].

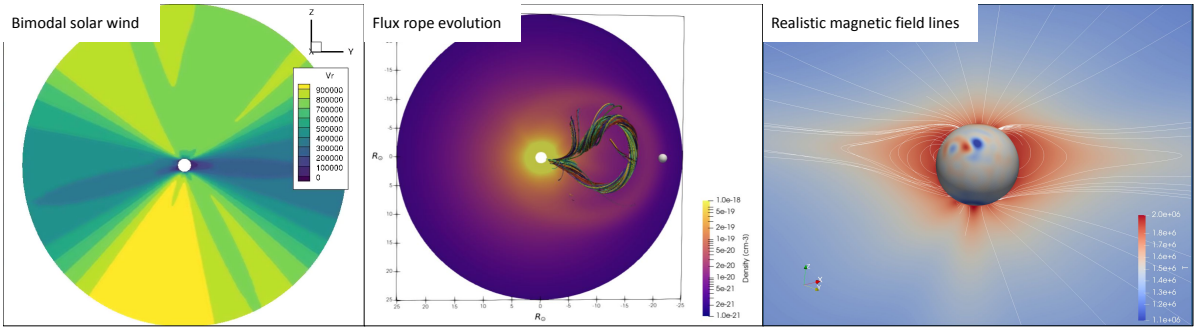

Figure 1: Demonstration of some of the COCONUT results. For details, refer to Ref. [2], Ref. [13] and Ref. [12], respectively.

Note that conducting multi-fluid simulations will not be discussed in this document, as the solver setup for that purpose differs greatly.

[1] `https://github.com/andrealani/COOLFluiD`

2. COCONUT setup

To understand how the solver can be run, we must first understand what it does and its required inputs. Here, a short overview of the solver setup will be given. More information is available in Ref. [3].

2.1. Physical setup

The baseline code solves for a set of MHD equations for the so-called primitive variables ρ (density), V_x, V_y, V_z (Cartesian velocity components), B_x, B_y, B_z (Cartesian magnetic field components), p (pressure) and ψ (a divergence cleaning parameter, which will be explained in Section 2.2). The code can be run in a polytropic mode, where one solves

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\mathbf{V}\\ \mathbf{B}\\ E\\ \phi\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\mathbf{V}\\ \rho\mathbf{V}\mathbf{V}+I\left(p+\frac{1}{2}|\mathbf{B}|^2\right)-\mathbf{B}\mathbf{B}\\ \mathbf{V}\mathbf{B}-\mathbf{B}\mathbf{V}+I\phi\\ \left(E+p+\frac{1}{2}|\mathbf{B}|^2\right)\mathbf{V}-\mathbf{B}(\mathbf{V}\cdot\mathbf{B})\\ V^2_{\text{ref, HDC}}\mathbf{B}\end{pmatrix}=\begin{pmatrix}0\\ \rho\mathbf{g}\\ 0\\ \rho\mathbf{g}\cdot\mathbf{V}\\ 0\end{pmatrix}, \tag{1}$$

or in a full-MHD mode, where we add a source term S,

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho\mathbf{V}\\ \mathbf{B}\\ E\\ \phi\end{pmatrix}+\nabla\cdot\begin{pmatrix}\rho\mathbf{V}\\ \rho\mathbf{V}\mathbf{V}+I\left(p+\frac{1}{2}|\mathbf{B}|^2\right)-\mathbf{B}\mathbf{B}\\ \mathbf{V}\mathbf{B}-\mathbf{B}\mathbf{V}+I\phi\\ \left(E+p+\frac{1}{2}|\mathbf{B}|^2\right)\mathbf{V}-\mathbf{B}(\mathbf{V}\cdot\mathbf{B})\\ V^2_{\text{ref}}\mathbf{B}\end{pmatrix}=\begin{pmatrix}0\\ \rho\mathbf{g}\\ 0\\ \rho\mathbf{g}\cdot\mathbf{V}+S\\ 0\end{pmatrix}, \tag{2}$$

which represents radiation, heat conduction, and coronal heating,

$$S=-\nabla\cdot\mathbf{q}+Q_{\text{rad}}+Q_H, \tag{3}$$

see the work in Ref. [2]. The radiative approximation is determined by assuming the radiative losses to be optically thin,

$$Q_{rad}=-n_e n_p P(T). \tag{4}$$

The heat conduction computation varies throughout the domain as closer to the sun, $r<10\,\mathrm{R}_\odot$, we assume a collisional regime,

$$\mathbf{q}_{<10R_\odot}=-\kappa_{||}\hat{\mathbf{b}}\hat{\mathbf{b}}\cdot\nabla T, \tag{5}$$

where $\kappa_{||}=9\cdot10^7 T^{\frac{2}{5}}$, and further away, we assume a collisionless regime,

$$\mathbf{q}_{>10R_\odot}=\alpha n_e k_B T\mathbf{V}. \tag{6}$$

The coronal heating approximation may take several forms, with some of the options defined in Ref. [2]. Perhaps the most popular choice is the expression that scales the amount of heating with the magnetic field strength and follows a distance envelope function,

$$Q_H=H_0|\mathbf{B}|e^{-\frac{r-R_\odot}{\lambda}}, \tag{7}$$

where both λ and H_0 can be tuned.

In the polytropic case, the ratio of specific heats γ is 1.05, see Ref. [16], whereas, in the full-MHD case, one can use the physical $\gamma = 1.66$.

The solver is formulated adimensionally to improve its convergence and flexibility. The reference values are selected as follows,

- $B_{\text{ref}} = 2.2 \cdot 10^{-4}$ T,
- $\rho_{\text{ref}} = 1.67 \cdot 10^{13}$ kg/m^3,
- $p_{\text{ref}} = 0.03851$ Pa, and
- $l_{\text{ref}} = 6.96 \cdot 10^{8}$ m.

The above values give reference velocity of $\sim 4.8 \cdot 10^5$ m/s.

2.2. Numerical setup

Let us know discuss some of the numerical particularities that are important from the user's perspective.

To ensure that the divergence of the magnetic field is zero in the domain, we use the so-called hyperbolic divergence cleaning method, which is implemented as the last equation shown in the syste,s (1) and (2), see the derivation in Ref. [21] or further explanation in Ref. [3].

The HLL scheme (see Ref. [11]) is used to reconstruct the fluxes at cell interfaces. The reconstructed fluxes are limited via the Venkatakrishnan limiter (see Ref. [19]), the limiter function ϕ_L of which, by default, follows

$$\phi_{C,P} = \begin{cases} \psi\left(\frac{\mathbf{U}_{\max C\&N} - \mathbf{U}_C}{\nabla\mathbf{U}|_C \cdot (\{r\}_P - \{r\}_C)}\right) & \text{if } \nabla\mathbf{U}|_C \cdot (\{r\}_P - \{r\}_C) > 0, \\ \psi\left(\frac{\mathbf{U}_{\min C\&N} - \mathbf{U}_C}{\nabla\mathbf{U}|_C \cdot (\{r\}_P - \{r\}_C)}\right) & \text{if } \nabla\mathbf{U}|_C \cdot (\{r\}_P - \{r\}_C) < 0, , \\ 1 & \text{if } \nabla\mathbf{U}|_C \cdot (\{r\}_P - \{r\}_C) = 0, \end{cases} \tag{8}$$

where P and C correspond to the quadrature points on the cell faces and in the cell centre, see Ref. [3] for more information. The function ψ with the numerator Δ^+ and denominator Δ^- is given as,

$$\psi\left(\frac{\Delta^+}{\Delta^-}\right) = \frac{\left(\Delta^+\right)^2 + 2\Delta^+\Delta^- + \epsilon}{\left(\Delta^+\right)^2 + \Delta^+\Delta^- + 2\left(\Delta^-\right)^2 + \epsilon}. \tag{9}$$

The small parameter ϵ scales with the cell scale Δh and can be tuned by the user via the term K to either increase or decrease the amount of numerical dissipation,

$$\epsilon = (K\Delta h)^3. \tag{10}$$

To improve the control of the limiter, we further include multiplication by the weight w,

$$\psi'_{\rho,\mathbf{v},p} = \psi w = \psi\left(C_s |U_{\min}/U_{\max}| + (1 - C_s)\right), \tag{11}$$

Which can also be tuned for (and modified throughout) the simulation to better and interactively control the level of limiting.

2.3. Simulation grid

COCONUT solves 3D MHD via the Finite Volume method, which means that its solution domain requires discretisation into smaller volumes (cells). The domain of COCONUT, as it is a global coronal modelling code that couples to heliospheric software at 0.1 AU (which is $\sim 21.5\ R_\odot$), spans from what we refer to as the "lower corona" at $\sim 1.01\ R_\odot$ to ≥ 0.1 AU. Note that in the current version of the code, the domain extends to $\sim 25\ R_\odot$ instead of $21.5\ R_\odot$ to limit the outer boundary condition effects at the coupling location, see the discussion in Ref. [5].

Thus, the resulting modelling domain is a spherical shell structure, where the inner boundary represents the lower solar corona and the outer boundary the location at ~ 0.1 AU. The rationale behind the location of the outer boundary condition (and the coupling of the heliospheric software) is the fact that according to the Parker Solar Wind model, by 0.1 AU, the solar wind should typically be both supersonic and super-Alfvénic. This means that the coupling to the heliospheric software can be made one-way as the characteristics cannot propagate back into the coronal domain. Note that due to this assumption, if one simulates a certain case with COCONUT that does not result in the solar wind being supersonic at the outer boundary, the solution does not make sense physically.

A lot of effort has been made to design a suitable grid that would cover this spherical shell domain. COOLFluiD grids are, by default, unstructured, which can be an advantage if one wishes to employ advanced performance enhancement techniques such as higher-order reconstruction or unstructured AMR (adaptive mesh refinement). In addition, solvers with structured spherical grids generally contain singularities in the polar regions, requiring additional boundary conditions and degrading the solution accuracy.

After a trade-off performed in Ref. [8], an icosahedron-based surface discretisation was selected for COCONUT, see the left side of Figure 2. If necessary, an unstructured version of the structured grid, right, can be also generated, but it must be noted that convergence issues may arise in the polar regions (see the discussion in Ref. [8]).

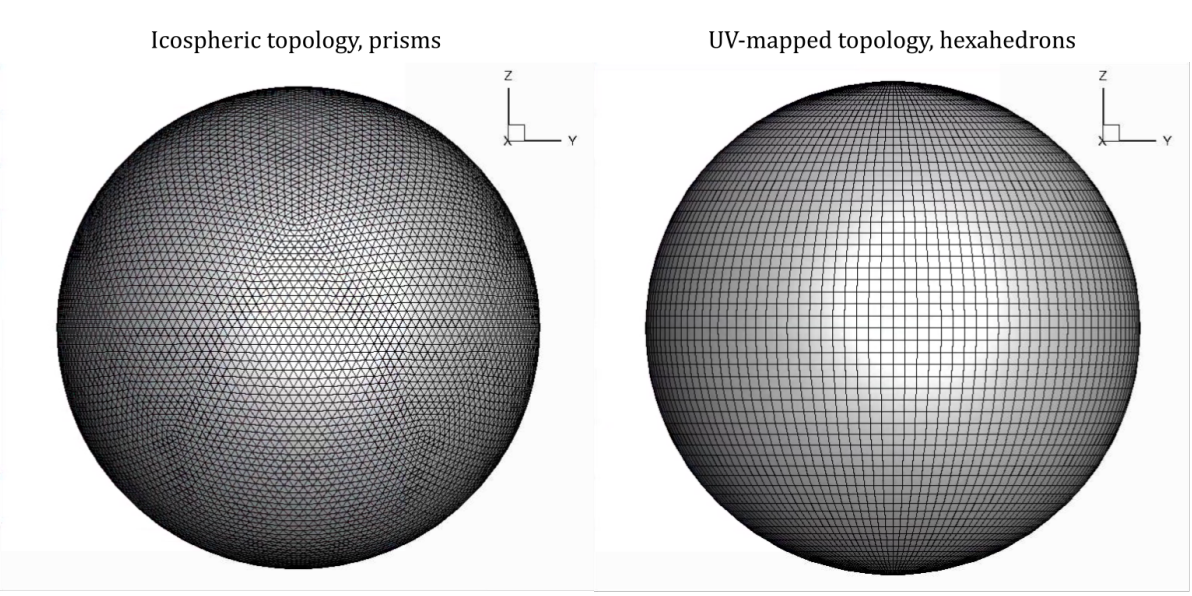

Figure 2: Figure from Ref. [8]. Shown is the default surface grid topology (left) and an available on-demand topology (right).

The surface discretisation is extended into a full 3D spherical shell grid by stacking the surface cell layers radially outward, in pre-determined radial levels, from the inner boundary (where the initial cell layer is deposited) in the direction towards the outer boundary, see Figure 3. To generate this type of a grid (e.g., if the user requires a grid with their own surface resolution or radial spacing), the code "YLVIS" is available on Github[2] with is own documentation provided.

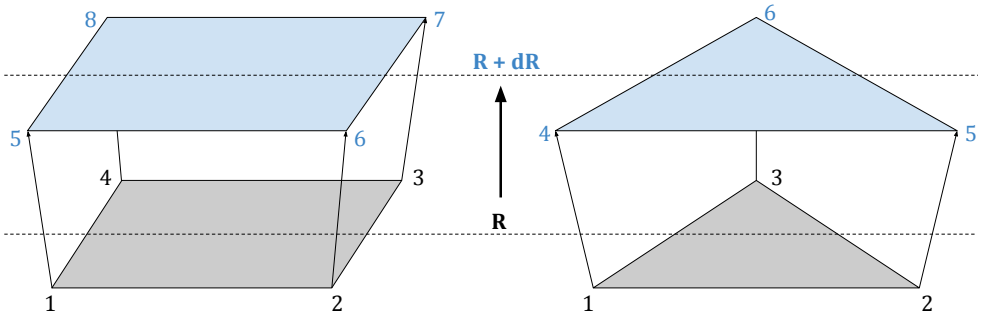

Figure 3: Figure adjusted from Ref. [8]. Shown is the principle through which the radial extension is carried out to transform the surface grid, shown in Figure 2, to a spherical shell grid spanning the entire domain.

The default radial spacing that can be visible if one cuts through the domain is shown in Figure 4. Near the inner boundary, the grid has very thin layers to capture and accommodate the strong gradients. Near the outer boundary, the spacing increases considerably (apart from the last thin cell-layer inserted to minimise extrapolation errors, see Ref. [5]).

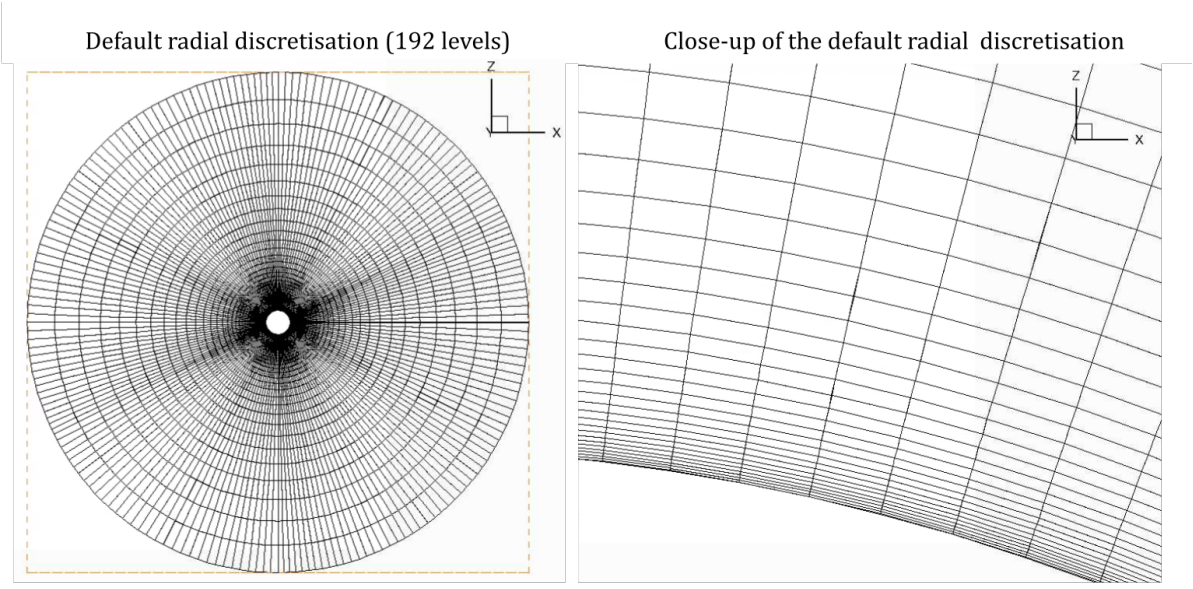

Figure 4: Figure from Ref. [8]. Shown is the default radial discretisation in the domain.

To consult what level of resolution is required, refer to the grid resolution recommendations in Ref. [3]. Generally speaking, for data-driven cases that require capturing of finer features, a 6th level surface division of the basic icosahedron is required with 50 to 200

[2] `https://github.com/Mbrchnelova/YLVIS`

radial layers. This results in grids of 1M to 4M cells. For simpler cases intended for testing and numerical experiments, such as modelling a dipole or a quadrupole, the 5th level icosahedron generally suffices (300k to 1M cells).

2.4. Boundary conditions

As is clear from the discussion above, there are two boundary conditions in the domain; the inner boundary representing the lower solar corona and the outer boundary at ~ 0.1 AU representing the supersonic outflow and the coupling location to the heliospheric domain. The justification and baseline design of the boundary conditions is provided in Ref. [5]. At each boundary, we must prescribe the state of the primitive variables, namely ρ, V_x, V_y, V_z, B_x, B_y, B_z and ψ.

By default, the outer boundary simply extrapolates the states from the last inner cell in the domain to the so-called ghost state located at the other side of the domain boundary (see Ref. [5] for details about this extrapolation). This extrapolation is allowed since the boundary condition is assumed to be supersonic as discussed above. However, should subsonic flow be present near the outer boundary (it has been observed to happen in certain time periods), this boundary condition is no longer accurate. In that case, it might be sufficient to simply extend the domain by providing a larger grid (such that the flow is supersonic at the boundary further away), otherwise background pressure in these boundary points has to be prescribed. Contact the code developers if that is the case.

The inner boundary is a bit more involved. For the magnetic field, we use the prescription for B_r from the provided magnetogram. Thus, the first prescription for B_x, B_y and B_z comes from B_r. B_θ and B_ϕ are unconstrained and may evolve as the simulation progresses. For the other variables, the default prescriptions are

- a constant homogeneous boundary density of $1.67 \cdot 10^{-13}$ kg/m^3,
- a constant homogeneous boundary pressure of 0.00416 Pa, and
- an outflow of ~ 1.3 km/s aligned with the background magnetic field, see Ref. [5].

Naturally, it is expected that the density and pressure would be different in Quiet Sun (QS) regions when compared to locations such as active regions and coronal holes, which is not captured by the homogeneous boundary conditions shown above. For further discussion of this, refer to Ref. [4]. There is a possibility to limit the boundary values of the prescribed plasma β and Alfvén speed, which act as proxies to the plasma pressure and density for a given magnetogram. Constraining the minimum plasma β and the maximum Alfvén speed lead to locally increasing the plasma pressure and density in the regions of stronger magnetic fields, and might prevent the formation of unphysical features as shown in Ref. [4].

Note that the features that one wishes to resolve on the inner boundary must be consistent with the grid resolution chosen for the simulation, see Figure 5. In this case, the smaller parts of the large active region are just resolved with the chosen grid. Features can be only properly resolved if they are at least a few cells wide.

2.5. Initial conditions and restart

The meaning and setup of the initial conditions largely depend on whether the simulation is meant to be steady-state or time-accurate; the two setups will be discussed in Section 2.6

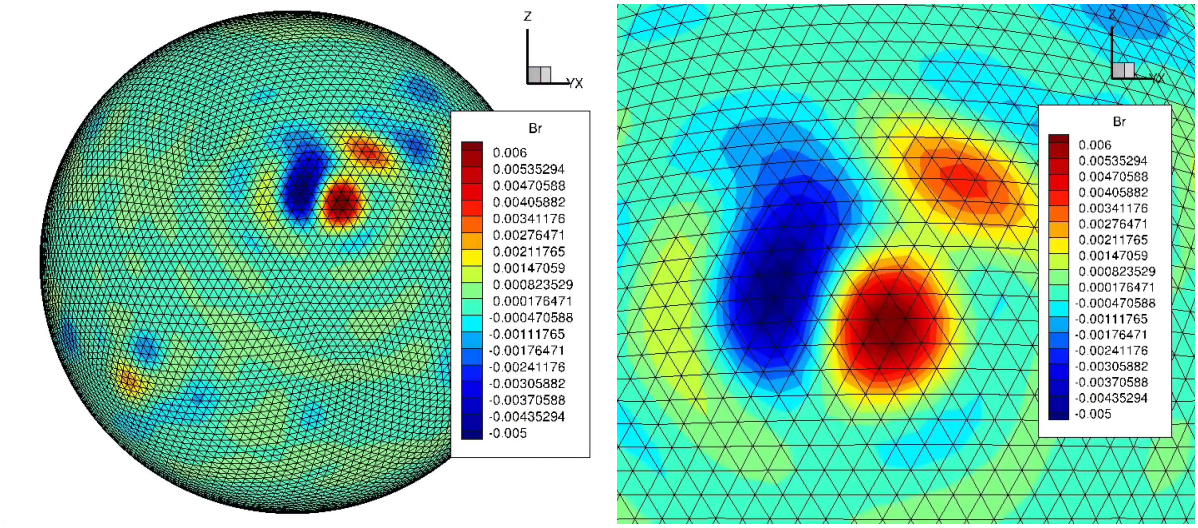

Figure 5: Shown is a resolution of a strong active region with the default 6th level icosahedron grid, demonstrating that to capture the smaller features within it, a finer grid may be required.

and 2.7, respectively.

For steady-state simulations, the initial state should not, in principle, affect the final solution. However, starting from an initial condition that is too far away from the final state, or one which does not conform to boundary conditions (e.g., one which defines a very different thermodynamic state at the boundary compared to what the boundary condition prescribes) might worsen convergence or make convergence impossible altogether. The initial condition for the magnetic field is computed by a Poisson solver. The initial thermodynamic state can be modified or set from scratch by the user such that this field is consistent with the values that the user wishes to prescribe at the inner boundary surface.

For time-accurate simulations, the initial state largely defines the further evolution of the simulation. For the time-accurate simulations carried out with COCONUT (see, e.g., Ref. [13]), carried out to, for instance, capture the eruption and evolution of CMEs, it is the custom to obtain the initial condition from a steady-state solution of the corresponding background solar wind. Then, the solution from the steady-state solver can be used as an initial state (a simulation "restart") that will overwrite any other initial conditions set in the solver.

2.6. Steady-state setup

Steady-state simulations are the "default" mode in which COCONUT is typically run. The mode evolves the solution of the System (1) or (2) until the derivatives are zero; that is, the flowfield is no longer changing between subsequent iterations within a certain tolerance. To evaluate the extent to which the flowfield is changing, one monitors the so-called residual, which is, for a given variable a (being one of ρ, V_x, V_y, V_z, B_x, B_y, B_z, p, or ψ) defined as

$$\mathrm{res}(a) = \log \sqrt{\sum_i \left(a_i^t - a_i^{t+1}\right)^2}, \tag{12}$$

where i and t are the spatial and temporal indices, respectively. Note that the scale is logarithmic. These residuals are continuously (each iteration) updated and output into a

convergence file such that the user can monitor the progress of the simulation. For example, if one reads that at the current iteration, the residual in the (nondimensional) simulation density is -5, this means that the total sum of the changes in the flowfield density between two consecutive iterations is 10^{-5}, which is typically already well converged. The target residual parameter is also set by the user such that the simulation stops automatically once a certain residual value of a certain primitive variable is reached.

To evolve the simulation between iterations, the steady-state setup uses a so-called implicit time-stepping, which means that CFL numbers (that, in principle, define the time step between the subsequent iterations) of more than 1 can be used. This results in a setup that can converge rapidly. It is good practice to keep CFL smaller or even just at 1 in the initial few hundred of iterations when very large changes in the flowfield still occur. Afterwards, however, CFL can be increased in steps up to tens and hundreds to accelerate the convergence process. The CFL number can be set either in a pre-determined fashion, or it can be interactively controlled by the user during the simulation run.

Note that, however, since the time-stepping is implicit with CFL that can be higher than 1, it also means that the solution is evolved unphysically fast. Thus, while the final obtained steady-state solution is independent of the implicit "pseudotime-stepping" process, the solution evolution during the simulation cannot be interpreted physically. For the time-accurate solution marching, intended for the cases in which the user is interested in the physically accurate temporal evolution and not just a final steady-state, we thus have a separate solver, see below.

2.7. Time-accurate setup

For processes where the time derivatives are important, such as when studying coronal mass ejection evolution, one requires a setup that does marches physically. In this setup, both the time-step and the initial condition determine the solution at a given time.

The time-accurate solver evolves the solution from the initial condition with a time step profile that is either pre-determined or controlled interactively, just like the CFL number. The time step cannot be too large, otherwise the simulation might evolve unphysically and/or have problems converging.

Ensuring suitable settings of the time-accurate solver is an involved task and apart from just monitoring the CFL number and the residual as in steady-state simulations, here, one has to also ensure sufficient convergence of the solution between subsequent time steps. Between each iteration in time, several of so-called subiterations are carried out to converge the solution, making this system effectively consist of a nested loop.

The user thus has to not only determine the time step, but also the tolerance to which the solution has to be converged between being marched further in time. Insufficient convergence between time steps will lead to an unphysical evolution, even if the time step by itself is sufficiently small.

Note that just like with the other code variables, the current time and time steps are indicated adimensionally, with $t_{\mathrm{ref}} \sim 1450$ s for our l_{ref} and V_{ref}.

2.8. Time-evolving simulation setup

In this module, a series of time-evolving photospheric magnetograms are utilized to drive the evolution of coronal structures [20]. Please read paper [20] in detail before performing the time-evolving coronal simulation.

During the time-evolving coronal simulations, the time-step sizes are set to be several minutes. They are smaller than the time interval between two adjacent magnetograms, thus we perform a cubic Hermit interpolation on these input magnetograms as below to get the required inner-boundary magnetic field at each time step.

$$\begin{aligned} B_{BC,r}\left(t,\theta,\phi\right) = {} & h_{00}\left(t^{'}\right) B_r\left(\theta,\phi\right)_m \\ & + h_{10}\left(t^{'}\right)\left(t_{m+1}-t_m\right)\left(\frac{\partial B_r\left(\theta,\phi\right)}{\partial t}\right)_m \\ & + h_{01}\left(t^{'}\right) B_r\left(\theta,\phi\right)_{m+1} \\ & + h_{11}\left(t^{'}\right)\left(t_{m+1}-t_m\right)\left(\frac{\partial B_r\left(\theta,\phi\right)}{\partial t}\right)_{m+1} \end{aligned} \tag{13}$$

$$\text{with } t^{'} = \frac{t-t_m}{t_{m+1}-t_m}.$$

Here $B_{BC,r}\left(t,\theta,\phi\right)$ is the interpolated magnetic field at the inner-boundary facial centroid located at $\left(R_s,\theta,\phi\right)$, t denotes the physical time, and the subscripts "$_m$" and "$_{m+1}$" refer to the m-th and (m+1)-th magnetograms, two adjacent magnetograms nearby t, with $t_m < t \le t_{m+1}$, $\left(\frac{\partial B_r(\theta,\phi)}{\partial t}\right)_m = \frac{1}{2}\left(\frac{B_r(\theta,\phi)_{m+1}-B_r(\theta,\phi)_m}{t_{m+1}-t_m} + \frac{B_r(\theta,\phi)_m-B_r(\theta,\phi)_{m-1}}{t_m-t_{m-1}}\right)$ and $\left(\frac{\partial B_r(\theta,\phi)}{\partial t}\right)_{m+1} = \frac{1}{2}\left(\frac{B_r(\theta,\phi)_{m+2}-B_r(\theta,\phi)_{m+1}}{t_{m+2}-t_{m+1}} + \frac{B_r(\theta,\phi)_{m+1}-B_r(\theta,\phi)_m}{t_{m+1}-t_m}\right)$. Besides, $h_{00}\left(t^{'}\right) = 2t^{'3} - 3t^{'2} + 1$, $h_{10}\left(t^{'}\right) = t^{'3} - 2t^{'2} + t^{'}$, $h_{01}\left(t^{'}\right) = -2t^{'3} + 3t^{'2}$ and $h_{11}\left(t^{'}\right) = t^{'3} - t^{'2}$ are used to normalize the cubic Hermit interpolation formula.

This interpolation is made in the file titled "NodalStatesExtrapolator.ci". Below are some codes need to be revised according to your specific case.

The code snippets in Figures. 6 and 7 is defined in subroutine:

template <typename DATA>

void NodalStatesExtrapolator<DATA>::setup()

In Figure. 6, "m_magfiles" denote the total number of input magnetogram .dat files, for example, "map_gong_lmax20_2019062911110400.dat"; "Cadance" is the time interval, in unite of hour, between two adjacent magnetograms; "year" corresponds to the year of the first magnetogram .dat file; "data_start" denotes the data of the initial time in the time-evolving coronal simulation, in format of month_day_hour; "current_data" should be one "Cadance" earlier than "data_start".

In Figure. 7, "current_time=(i-2)*2.4876;" transfer the physical time to code time. One hour in physical time corresponds to 2.4876 in code.

Based on these initially defined magnetograms and corresponding times, the following subroutine perform a cubic Hermit interpolation (default) or linear interpolation at each physical time step to get the interpolated magnetogram corresponding to a specific moment in the time-evolving coronal simulation.

template <typename DATA>

void NodalStatesExtrapolator<DATA>::extrapolateVarsFromFileInTime()

```
    if (m_fileNameTw.size() == 1) {m_fileNameTime.resize(1, 0.0);}
    cf_assert(m_fileNameTw.size() == m_fileNameTime.size());
    CFLog(INFO, "m_fileNameTw.size():setup="<<m_fileNameTw.size()<<"\n");
    CFLog(INFO, "m_fileNameTime.size():setup="<<m_fileNameTime.size()<<"\n");
    for(CFuint i=0; i<m_fileNameTw.size(); ++i){
        CFLog(INFO, "m_fileNameTw[i]:setup="<<m_fileNameTw[i]<<"i="<<i<<"\n");
        CFLog(INFO, "m_fileNameTime[i]:setup="<<m_fileNameTime[i]<<"i="<<i<<"\n");
    }

if(m_fileNameTw.size()>3){
 std::string FileName_prefix="./MapData/map_gong_lmax20_2019";
 std::string FileName_suffix="0400.dat";
 CFuint m_magfiles=1285;
 CFuint Cadence=1;
 CFuint year=2019;
 CFuint days_Feb=29;
 CFuint day_M=30;
 CFuint hour=0;
 CFuint hours=0;
 CFuint day=0;
 CFuint days=0;
 CFuint month=0;
 CFuint months=0;
 CFuint data_start=62911;
 CFuint zerosToAdd=0;
 CFuint current_data=data_start;
 std::string current_dataStr = std::to_string(current_data);
 CFuint desiredLength = 6;
 if(current_dataStr.size()<desiredLength){
     zerosToAdd=desiredLength-current_dataStr.size();
     current_dataStr.insert(0,zerosToAdd,'0');
 }
 std::string current_Filename=FileName_prefix+current_dataStr+FileName_suffix;
 m_fileNameTw.resize(m_magfiles, current_Filename);
 m_fileNameTw[0]=current_Filename;
```

Figure 6: Shown is part of the code snippets to read input magnetograms.

Some times, we need restart from a specific moment. We should modify the file titled "SubSystemStatus.cxx" as illustrated in Figure. 8. "restarttime" means the physical time in unit of hour corresponding to the start moment, which can be found in the "slurm-xxx.out" file. Correspondingly, we should modify the CFmesh file name "corona-time_1245_6.CFmesh" to the corresponding CFmesh of the restart moment in the .CFcase file. As illustrated in Figure. 9.

```
  CFreal current_time=0.0;
  m_fileNameTime.resize(m_magfiles, current_time);
  m_fileNameTime[0]=current_time;
  CFLog(INFO, "m_fileNameTime[0]:setup="<<m_fileNameTime[0]<<"\n");
  for (int i = 1; i < m_magfiles; ++i){
     current_time=(i-2)*2.4876;
     //for cadence=6, choose "current_time=(i-2)*14.9254;"
     m_fileNameTime[i]=current_time;
     CFLog(INFO, "m_fileNameTime[i]:setup="<<m_fileNameTime[i]<<" i="<<i<<"\n");
  }
  CFLog(INFO, "m_fileNameTime.size():setup="<<m_fileNameTime.size()<<"\n");
 }
 //>> mark 2024.04.27

    if (m_fileNameTw.size() > 1) {
      // read and store all the surface objects, one for each boundary file
      // these will be used later on for time interpolation
      m_allSurfaces.resize(m_fileNameTw.size());
      for (CFuint is = 0; is < m_fileNameTw.size(); ++is) {
    readSurfaceData(m_allSurfaces[is], m_fileNameTw[is]);
    const CFuint nbSurf = m_allSurfaces[is].size();
    cf_assert(nbSurf >= 1);
    CFLog(INFO, "m_allSurfaces[is].size()="<<m_allSurfaces[is].size()<<", is="<<is
    CFLog(INFO, "m_allSurfaces[is][0]->Tw[100]="<<m_allSurfaces[is][0]->Tw[100]<<'
       ", m_allSurfaces[is][0]->xyz(100,1)="<<m_allSurfaces[is][0]->xyz(100,1)<<",
    CFLog(INFO, "m_allSurfaces[is][0]->Tw[200]="<<m_allSurfaces[is][0]->Tw[200]<<'
       ", m_allSurfaces[is][0]->xyz(200,1)="<<m_allSurfaces[is][0]->xyz(200,1)<<",
      }
    }
    // store SurfaceData for time "0"
    readSurfaceData(m_surfaceAtTime, m_fileNameTw[0]);
    extrapolateVarsFromFile(m surfaceAtTime);
```

Figure 7: Shown is part of the code snippets to write the time variables corresponding to input magnetograms.

```
CFreal SubSystemStatus::getCurrentTimeDim() const
{
  cf_assert(PhysicalModelStack::getActive()->getImplementor()->getRefTime() > 0.);
  CFLog(VERBOSE, "SubSystemStatus::getCurrentTimeDim(): [refTime, currTime] = [" <<
    PhysicalModelStack::getActive()->getImplementor()->getRefTime() << ", "
    << m_currentTime << "]\n");
    CFreal restarttime = 1245.6007;
    //666.667;
    //333.334;
    //583.333;
  return (PhysicalModelStack::getActive()->getImplementor()->getRefTime())*m_currentTime+restarttime*3600.0/1447.2;
}
```

Figure 8: Shown is the code snippets to restart from a specific moment.

```
# First reader
#Simulator.SubSystem.CFmeshFileReader0.Data.FileName = ./BCindependent5.CFmesh
# Restart
#Simulator.SubSystem.CFmeshFileReader0.Data.FileName = ./results-map-res/Level6/eclipse/FullMHD/Lmax10/L6corona.CFmesh #./results-map-res/Level5
# Xusuan-Restart during time-dependent simulation: Correspondingly, the following code should be revised.
# L222  CFreal restarttime = 0.0;
# L223  return (PhysicalModelStack::getActive()->getImplementor()->getRefTime())*m_currentTime+restarttime*1447.2/3600.0;
# in \src\Framework\SubSystemStatus.cxx(216):CFreal SubSystemStatus::getCurrentTimeDim() const
Simulator.SubSystem.CFmeshFileReader0.Data.FileName = ./results-map-res/Level6/eclipse/Timeevolving/FullMHD/Lmax20dt2/corona-time_1245_6.CFmesh

#Simulator.SubSystem.CFmeshFileReader0.Data.ScalingFactor = 1.
Simulator.SubSystem.CFmeshFileReader0.ParReadCFmesh.ParCFmeshFileReader.ParMetis.NCommonNodes = 2 # ??

# Second reader
#Simulator.SubSystem.CFmeshFileReader1.Data.FileName = ./BCindependent5.CFmesh
# Restart
#Simulator.SubSystem.CFmeshFileReader1.Data.FileName = ./results-map-res/Level6/eclipse/FullMHD/Lmax10/L6corona.CFmesh #./results-map-res/Level5
#xusuan-Restart during time-dependent simulation
Simulator.SubSystem.CFmeshFileReader1.Data.FileName = ./results-map-res/Level6/eclipse/Timeevolving/FullMHD/Lmax20dt2/corona-time_1245_6.CFmesh
```

Figure 9: Shown is the text required to be modified in .CFcase file to restart from a specific moment.

3. COOLFluiD structure

With the basic logic of the code introduced, we will now move on to the more practical matters and show how the solver can be set up, run, and the results analysed.

Let us start with what the first "CO" in COCONUT stands for. COCONUT is embedded in the COOLFluiD framework. COCONUT is, in principle, just one of many setups that can be run within COOLFluiD, along with setups for aerodynamics, chemistry and so forth. Thus, one has to download and install COOLFluiD first in order to run COCONUT. General download and install instructions are available on the Wiki page of COOLFluiD Github[3]. One has to follow these and if necessary, adjust the process depending on the architecture of their system (this is system-dependent, so it will not be elaborated on here in more detail).

Let us assume that the home folder that contains COOLFluiD is called `COOLFluiD_base`. Magnetohydrodynamics is one of the many so-called "plugins" in COOLFluiD; basically, a piece of physics/ numerics that is used to create setups. Thus, after installation, one will have the relevant source code that they can work with in `COOLFluiD_base/plugins` where they will find a plugin `MHD` and `FiniteVolumeMHD`. The former contains the general MHD code, such as the formulation of the primitive variables, the conversion between the conservative and primitive variables, details about the available schemes etc., while the latter contains FV specifics, such as boundary conditions and source terms.

If one wishes to add/ remove code, this can be done through `CMakeLists.txt` text files in each folder that define which code is compiled during the compilation process. Note that each time one makes changes to the source code in `COOLFluiD_base/plugins`, the code needs to be recompiled.

The compiled code can be found in `COOLFluiD_base/OPENMPI/{BUILD}`, where the name of `{BUILD}` represents the type of build, e.g., the production runs (`DEBUG`) or the development runs (`OPTIM`), with or without CUDA. The difference between these two is in some of the used compilation flags that allow either faster runs with fewer warnings and checks (ideal for users) or somewhat slower runs that are easier to debug if something fails (ideal for developers).

Note that in the current setting, even the `DEBUG` configuration produces an `OPTIM` folder.

Let us assume that we decide to build `OPTIM_NOCUDA`. Then, we can find our compiled plugins, including `MHD`, in `COOLFluiD_base/OPENMPI/optim/plugins`. For COCONUT, specifically, we can also find example setups in `COOLFluiD_base/OPENMPI/optim/plugins/MHD/testcases`. In these directories, we can find the files necessary to run a simulation. These will be, one-by-one, discussed in the following section.

[3] `https://github.com/andrealani/COOLFluiD/wiki/HOWTO`

4. COCONUT files

The COOLFluiD files required to run a simulation consist of several types. Below, we will discuss the purpose and contents of each of them. Example cases along with the corresponding files can be found on the code Github page[4].

4.1. Source files for full MHD

In this section, each additional source term implemented in the S term in Equations **??** is considered in detail. These terms are explicitly given in Eq. 3.

4.1.1 Thermal conduction

The first term represents the contribution from thermal conduction (also mentioned as heat conduction). The term is defined in two limits: $r < 10\,\mathrm{R}_\odot$ is considered to be a collision-dominated region and defined as follows:

$$\mathbf{q}_{<10R_\odot} = -\kappa_{||}\hat{\mathbf{b}}\hat{\mathbf{b}}\cdot\nabla T, \tag{14}$$

where $\kappa_{||} = 9\cdot 10^7 T^{\frac{5}{2}}$, and further away, we assume a collisionless regime,

$$\mathbf{q}_{>10R_\odot} = \alpha n_e k_B T\mathbf{V}. \tag{15}$$

The contribution from the thermal conduction is implemented as the flux $\mathbf{q}$, and the divergence of these two vectors is taken to obtain the energy term introduced in the energy equation. The implementation is done in `~/plugins/MHD/MHD3DProjectionDiffVarSet.cxx` file.

The flux terms are defined for both regimes, below and beyond 10 $R_\odot$, and the total flux is passed to the solver to calculate the energy term generated with $-\nabla\cdot\mathbf{q}$.

4.1.2 Radiative loss

The second term in Eq 3 corresponds to the radiative loss function in the optically thin limit, which is given by the following definition:

$$Q_{rad} = -n_e n_p P(T). \tag{16}$$

Here the $n_e = n_p$, and the $P(T)$ is the cooling curve given in [18]. Another implementation of the radiative loss function is implemented in COCONUT according to [1]. The term is introduced by

$$Q_{rad} = -n_e n_p f_n(T), \tag{17}$$

where

$$f_p(T) = f_n(T)T^{-2}, \tag{18}$$

and $f_p(T)$ is a cooling curve defined in [1]. The implemented terms are given in Figure 10. These two terms are implemented in

[4] `https://github.com/andrealani/COOLFluiD/tree/master/plugins/MHD/testcases/COCONUT_Dipole`

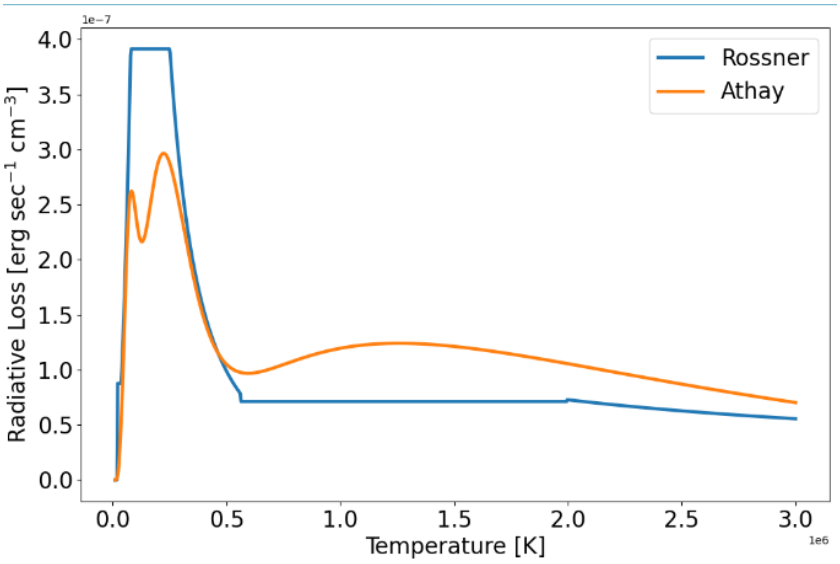

Figure 10: The radiative loss functions defined according to [18] and [1].

`~/plugins/FiniteVolumeMHD/MHDConsACASourceTerm.cxx` file.

The radiative loss function according to [18] is activated when the following term is added to the equation:

```
if (_RadiativeLossTerm == 1) {
source[7] -= Q_rad*nondimconst*volumes[elementID];
socket_radiativeloss.getDataHandle()[elementID] = Q_rad;
}
```

The radiative loss function according to [1] is activated when the following term is added to the equation:

```
if (_RadiativeLossTerm == 1) {
source[7] -= Q_rad_athay*nondimconst*volumes[elementID];
socket_radiativeloss.getDataHandle()[elementID] = Q_rad_athay;
}
```

4.1.3 Coronal Heating

Currently, the approximated heating functions are implemented in COCONUT. These functions are also implemented in

`~/plugins/FiniteVolumeMHD/MHDConsACASourceTerm.cxx` file.

In our model, we have used the three most common approximations found in the literature. As the first approximation, we used an exponential envelope function similar to [14] and [9],

$$Q_H = H_0 e^{-\frac{r-R_s}{\lambda}}, \tag{19}$$

where R_s is the solar radius, H_0 is the local heating rate at $r = R_s$ and λ is the scale height.

According to [9] we use $H_0 = 7.28 \cdot 10^{-5}$ erg cm^{-3} s^{-1} and $\lambda = 40$ Mm.
[17] established that there is a linear dependence of the magnetic field strength and X-ray radiance. Therefore, a function approximating this law is tested in COCONUT with a slight modification. [9] suggests to also use the radial damping of the heating term when using the coronal heating model based on the magnetic field configuration. Thus, the second model considered here reads:

$$Q_H = H_0 \cdot |\mathbf{B}| \cdot e^{-\frac{r-R_S}{\lambda}}. \tag{20}$$

where $H_0 = 4 \cdot 10^{-5}$ erg cm^{-3} s^{-1} G^{-1} and $\lambda = 0.7 R_\odot$.

Finally, we also considered the most complex heating function approximation that considers exponential heating, the contribution describing the quiet Sun, and active region heating. The last approximated heating function is taken from [14]. The function depends on the magnetic field, approximating the heating for the quiet Sun and the active regions

$$Q_H = Q_H^{exp} + Q_H^{QS} + Q_H^{AR}, \tag{21}$$

$$Q_H^{exp} = H_0 e^{\frac{-(r-R_\odot)}{\lambda_0}}, \tag{22}$$

where $H_0 = 4.9128 * 10^{-7}$ erg cm^{-3} s^{-1} and $\lambda_0 = 0.7 R_\odot$.

$$Q_h^{QS} = H_0^{QS} f(r) \frac{B_t^2}{B(|B_r| + B_r^c)}, \tag{23}$$

where $B_t = \sqrt{B_\theta^2 + B_\phi^2}$ is the tangential magnetic field.

$$Q_H^{AR} = H_0^{AR} g(B) \left(\frac{B}{B0}\right)^{1.2}. \tag{24}$$

In the performed simulations, the following constants are fixed according to [14], $H_0^{QS} = 1.18 * 10^{-5}$ erg cm^{-3} s^{-1}, $B_r^c = 0.55G$, $H_0^{AR} = 1.87 * 10^{-5}$ erg cm^{-3} s^{-1}, $B_0 = 1G$. The functions f(r) and g(B) are defined as follows:

$$f(r) = \frac{1}{2}\left(1 + \tanh \frac{1.7 - r/R_\odot}{0.1}\right) \exp\left(-\frac{r/R_\odot - 1}{0.2}\right), \tag{25}$$

$$g(B) = \frac{1}{2}\left(1 + \tanh \frac{B - 18.1}{3.97}\right). \tag{26}$$

The thresholds for the $g(B)$ and $f(r)$ functions can be modified according to the case and the magnetic field strength.

In `MHDConsACASourceTerm.cxx` file Eq. 19 is implemented with the formula in `Qh3` term. The heating profile in Eq. 20 is implemented with the formula in `Qh4` term. The complex heating function in Eq. 21 is implemented in `H_CH` term. Which heating profile is activated is controlled from the CFcase file which will be explained later.

4.2. CFcase files

The `.CFcase` file contains all the details regarding the physical and numerical setup and is the one that is perhaps the most important - and the most challenging - to understand. There are hundreds of lines in the file that link the available plugins and libraries within

COOLFluiD that together create COCONUT. Note that in the CFcase files, if the line starts with #, it denotes a comment or an unused setting. The example `.CFcase` file from Github[5] will be dissected here. We start with the path lines

```
Simulator.Paths.WorkingDir = ./
Simulator.Paths.ResultsDir = ./results
Simulator.SubSystem.InteractiveParamReader.FileName = ./map.inter
```

into which one sets the current directory path for running the code, the path to the directory to which the results should be saved (this directory will be created if it does not exist prior to the run) and finally, the path to the interactive `.inter` file (see its importance later in this document).

Next up, the lines

```
Simulator.SubSystem.Flow.Data.DistanceBased.FileNameTw = ./dipole.dat
Simulator.SubSystem.EM.Data.DistanceBased.FileNameTw = ./dipole.dat
```

contain the paths to the radial magnetic field datafile, `.dat`. This file contains the magnetogram B_r information for a set of coordinate points x, y, and z, that is to be interpolated onto the inner boundary. In some setup files, only the second, `EM`, line is present, which is also sufficient.

Here, we also notice that in our setup, we have two systems working together: a Poisson solver for the initial condition (the `EM` solver) and a full magnetohydrodynamic flow solver (the `Flow` solver). Hence, in many cases, input files and settings must be defined for both of them separately.

Both solvers also require a grid, the path to which is given in

```
Simulator.SubSystem.CFmeshFileReader0.Data.FileName =
./BCindependent5.CFmesh
Simulator.SubSystem.CFmeshFileReader1.Data.FileName =
./BCindependent5.CFmesh
```

where the grid files to which these lines should point are the `.CFmesh` files that will be described below in further detail. The lines underneath denoting the Parmetis settings do not usually need adjustments.

Next, in the lines

```
Simulator.SubSystem.MHD3DProjection.ConvTerm.gamma = 1.05
Simulator.SubSystem.Flow.Jet1.rotation = 1
```

one may set the ratio of specific heats γ (1.05 for polytropic runs, 1.66 for full-MHD runs) and whether solar rotation should be taken into account or not.

The inner boundary conditions can be set in

[5] `https://github.com/andrealani/COOLFluiD/blob/master/plugins/MHD/testcases/COCONUT_Dipole/map.CFcase`

```
Simulator.SubSystem.Flow.Jet1.rhoBC = 1.0
Simulator.SubSystem.Flow.Jet1.pBC = 0.108
```

where the former prescribes a homogeneous density of 1 and the latter a homogeneous pressure of 0.108, both in code units (see Section 2.1 for normalisations). One may also set the velocity boundary condition if they wish to prescribe a different outflow that the default outflow of ~ 2 km/s via changing the line

```
Simulator.SubSystem.Flow.Jet1.VrBC = 1935.07
```

Where in this case, the input is given in physical SI units (m/s).

To prescribe a nonhomogeneous boundary condition, as described in Section 2.4 and in [4] so as to accommodate stronger regions by locally increasing the density and pressure, one may do so by adding the following lines,

```
Simulator.SubSystem.Flow.Jet1.Nonhomogeneous = true
Simulator.SubSystem.Flow.Jet1.vAmax = 1e6
Simulator.SubSystem.Flow.Jet1.betamin = 1e-3
```

Where, e.g., in this case, the maximum prescribed Alfvén speed on the boundary is set to 10^6 (m/s) and the minimum prescribed plasma β on the boundary is 10^{-3}; and the values of the density and pressure are adjusted accordingly. This setting should help with more accurately resolving solar maxima especially.

The initial conditions for the primitive variables are defined in

```
Simulator.SubSystem.Flow.InField.Def =
1.0*((1/r)**5)
1000/480363.085276*(-52.1+106*log(r+0.78))*(x/r)
1000/480363.085276*(-52.1+106*log(r+0.78))*(y/r)
1000/480363.085276*(-52.1+106*log(r+0.78))*(z/r)
1.
2.
3.
0.108*((1/r)**5)
0.
```

where `r` represents the radial distance from the Sun ($r = \sqrt{x^2 + y^2 + z^2}$, all nondimensional). The functions follow the basic ordering of the primitive variables, i.e., ρ, V_x, V_y, V_z, B_x, B_y, B_z, p, ψ, all in code units. The conditions for B_x, B_y, B_z (1,2 and 3) do not have any physical meaning as they are instantly replaced by the Poisson solution after the simulation initiates.

Note that these initial conditions must be consistent with the boundary conditions prescribed higher up. For example, if one wants to increase the boundary density to 10 from 1, here, the initial condition function for density should change from `1.0*((1/r)**5)` to `10.0*((1/r)**5)` such that the value at the boundary agrees. The velocity field defined above is increasing radially in a spherically symmetric fashion, but it also can be changed

as long as it is more or less consistent with the boundary outflow velocity.

The solver stop condition is defined by

```
Simulator.SubSystem.Flow.Data.LinearLS3D.limitRes = -10
Simulator.SubSystem.Flow.Data.LinearLS3D.limitIter = 10000
```

Which here means that the simulation stops if the target residual reaches -10 or if 10000 iterations are exceeded. In the given example, these lines are not shown in the `.CFcase` file, but instead, they are located in the `.inter` file. Both options are possible (depending on whether the user wants to modify these settings interactively or not). Typically, simple cases such as the magnetic dipole take a few hundred to a few thousand of iterations to converge; for large, complex, data-driven cases, this can be between a few thousand to a few tens of thousands of iterations.

In between the iterations, sub-iterations are carried out to converge the solution for each iteration, and the stop condition of this sub-iteration process is defined through

```
Simulator.SubSystem.StopCondition = Norm
Simulator.SubSystem.Norm.valueNorm = -4.0
```

if the norm is used to determine convergence or through

```
   Simulator.SubSystem.StopCondition = MaxNumberSteps
Simulator.SubSystem.MaxNumberSteps.nbSteps = 50
```

If the maximum number of sub-iterations is defined as a stop condition instead.

For steady-state simulations, the CFL can be either manipulated interactively during the run via the use of the interactive `.inter` file or its profile can be a predefined function set here in the `.CFcase` file. The lines

```
Simulator.SubSystem.FlowIterator.Data.CFL.ComputeCFL = Function
Simulator.SubSystem.FlowIterator.Data.CFL.Function.Def =
if(i<20,2,if(i<70,8,if(i<100,16,if(i<120,32,min(1000,cfl*1.05**2)))))
```

show an example of such a predefined function, where the CFL value depends on the current iteration number `i` and the current CFL value `cfl`. If one wishes to, instead, modify the CFL value during the run, they have to set

```
Simulator.SubSystem.FlowIterator.Data.CFL.ComputeCFL = Interactive
```

and then use the interactive file to access and modify the value during the run (see below in this document).

Next up are the lines that define Poisson solver on B_x, B_y and B_z (the 4th, 5th and 6th primitive variables),

```
Simulator.SubSystem.Flow.Restart = false
Simulator.SubSystem.Flow.PreProcessCom = ComputeFieldFromPotential
```

```
Simulator.SubSystem.Flow.PreProcessNames = PreProcess1
Simulator.SubSystem.Flow.PreProcess1.VariableIDs = 4 5 6
Simulator.SubSystem.Flow.PreProcess1.OtherNamespace = EMNamespace
Simulator.SubSystem.Flow.PreProcess1.InterRadius = -1.
Simulator.SubSystem.Flow.PreProcess1.DeltaSelection = 1000.
Simulator.SubSystem.Flow.PreProcess1.ProcessRate = 100000000
```

which should all be active (uncommented) if the simulation starts from the initial Poisson solution, i.e., from scratch. Otherwise, nothing in them needs to be modified. However, if one wants to restart the simulation from an existing solution (which means that the Poisson solver should not be re-engaged to overwrite the initial state), the first line must be set to `true` and the follow up lines must be commented out, giving

```
Simulator.SubSystem.Flow.Restart = true
#Simulator.SubSystem.Flow.PreProcessCom = ComputeFieldFromPotential
#Simulator.SubSystem.Flow.PreProcessNames = PreProcess1
#Simulator.SubSystem.Flow.PreProcess1.VariableIDs = 4 5 6
#Simulator.SubSystem.Flow.PreProcess1.OtherNamespace = EMNamespace
#Simulator.SubSystem.Flow.PreProcess1.InterRadius = -1.
#Simulator.SubSystem.Flow.PreProcess1.DeltaSelection = 1000.
#Simulator.SubSystem.Flow.PreProcess1.ProcessRate = 100000000
```

in which case also the grid path lines defined above

```
Simulator.SubSystem.CFmeshFileReader0.Data.FileName = ./solution.CFmesh
Simulator.SubSystem.CFmeshFileReader1.Data.FileName = ./solution.CFmesh
```

must point to the solution `.CFmesh` file from which the restart should happen instead of just an empty grid file.

Next, the line

```
Simulator.SubSystem.MHD3DProjection.ConvTerm.refSpeed = 1.
```

defines the value of the reference speed for divergence cleaning presented in systems (1) and (2) in code units. This value should be higher than any physical value in present the flowfield. E.g., if one is using full MHD to resolve fast solar wind of ~ 900 km/s (1.85 in code units), this reference speed value should be set to at least 2, ideally 3. The value of 1 is only sufficient for polytropic runs, where only the slow solar wind is resolved.

Finally, the lines

```
Simulator.SubSystem.Flow.Data.Limiter = Venktn3DStrict
Simulator.SubSystem.Flow.Data.Venktn3DStrict.coeffEps = 5.
Simulator.SubSystem.Flow.Data.Venktn3DStrict.isMFMHD = true
Simulator.SubSystem.Flow.Data.Venktn3DStrict.strictCoeff = 0.5
```

define the limiter settings. By default, the Venkatakrishnan limiter is used; see Section

2.2. The `coeffEps` is the K term in Equation (10) and the `strictCoeff` parameter defines the weight w introduced in Equation (11). Note that the higher the `coeffEps` term, the lower the limiting; and the opposite applies to `strictCoeff` (0 for no additional limiting, 1.0 for full limiting).

4.2.1 Time-accurate setup

The setup shown above applies to the default steady-state runs. To transform it into a time-accurate setup (see Sections 2.6 and 2.7), the implicit pseudo time-stepping method has to be exchanged for a physical time-stepping. This can be done by exchanging the line

```
Simulator.Modules.Libs = libShapeFunctions libCFmeshFileReader
libCFmeshFileWriter libParaViewWriter libTecplotWriter libNavierStokes
libPoisson libMHD libFiniteVolume libFiniteVolumePoisson libNewtonMethod
libFiniteVolumeMHD libGmsh2CFmesh libGambit2CFmesh libForwardEuler
libPetscI
```

for

```
Simulator.Modules.Libs = libShapeFunctions libCFmeshFileReader
libCFmeshFileWriter libParaViewWriter libTecplotWriter libNavierStokes
libPoisson libMHD libFiniteVolume libFiniteVolumePoisson libNewtonMethod
libFiniteVolumeMHD libGmsh2CFmesh libGambit2CFmesh libForwardEuler
libPetscI libBackwardEuler libNewtonMethodMHD,
```

to load a different set of plugins,

```
Simulator.SubSystem.ConvergenceMethod = NewtonIterator NewtonIterator
```

for

```
Simulator.SubSystem.ConvergenceMethod = NewtonIterator BDF2
```

and

```
Simulator.SubSystem.Flow.ComputeTimeRHS = PseudoSteadyTimeRhs
Simulator.SubSystem.Flow.PseudoSteadyTimeRhs.zeroDiagValue = 0 0 0 0 0 0 0
0 1
Simulator.SubSystem.Flow.PseudoSteadyTimeRhs.useGlobalDT = true
```

for

```
Simulator.SubSystem.Flow.ComputeTimeRHS = BDF2TimeRhsLimited
Simulator.SubSystem.Flow.BDF2TimeRhsLimited.TimeLimiter = MinMod
Simulator.SubSystem.Flow.BDF2TimeRhsLimited.MinMod.SlopeRatio = 3.
Simulator.SubSystem.Flow.BDF2TimeRhsLimited.zeroDiagValue = 0 0 0 0 0 0 0
0 1.
```

Finally, one needs to add lines through which it is possible to control the time-step,
These lines :

```
Simulator.SubSystem.FlowSubSystemStatus.ComputeDT = FunctionDT
Simulator.SubSystem.FlowSubSystemStatus.FunctionDT.Vars = i
Simulator.SubSystem.FlowSubSystemStatus.FunctionDT.Def =
if(i<30,1e-4,if(i<60,1e-3,1e-2))
```

or this one :

```
Simulator.SubSystem.FlowSubSystemStatus.TimeStep = 1e-4.
```

Tests have been conducted, and to achieve a balance between accuracy and computational time, the value of dt can range between 1e-3 and 1e-2.

Additionally, the ends condition needs to be modified. It will be determined by the max time reached in code units during the simulation. Be mindful that, for instance, if you select a save rate of 10, a timestep of 1e-2, and a maximum time of 60, this results in a total of 600 * the number of processes files saved, which is significant.

```
Simulator.SubSystem.StopCondition = Norm
Simulator.SubSystem.Norm.valueNorm = -10.0
```

for

```
Simulator.SubSystem.StopCondition = MaxTime
Simulator.SubSystem.MaxTime.maxTime = 60
```

Be cautious not to start from zero, as doing so could result in a potential field replacing our implemented CME. Therefore, it is essential to have the following conditions:

```
Simulator.SubSystem.Flow.Restart = true
```

and comment

```
Simulator.SubSystem.Flow.PreProcessCom = ComputeFieldFromPotential
Simulator.SubSystem.Flow.PreProcessNames = PreProcess1
Simulator.SubSystem.Flow.PreProcess1.VariableIDs = 4 5 6
Simulator.SubSystem.Flow.PreProcess1.OtherNamespace = EMNamespace
Simulator.SubSystem.Flow.PreProcess1.InterRadius = -1.    15e9 2.5  -1
Simulator.SubSystem.Flow.PreProcess1.DeltaSelection = 1000.  1e-4
Simulator.SubSystem.Flow.PreProcess1.ProcessRate = 100000000
```

The presence of a CME (Coronal Mass Ejection) may lead to negative pressures, so it is necessary to address this issue:

```
Simulator.SubSystem.FlowIterator.UpdateSol = StdUpdateSol
Simulator.SubSystem.FlowIterator.StdUpdateSol.Relaxation= 1.
```

for

```
Simulator.SubSystem.FlowIterator.UpdateSol = UpdateSolMHD
Simulator.SubSystem.FlowIterator.UpdateSolMHD.pressureCorrectionValue = 0.00000000000
```

It is also necessary to add the following lines:

```
Simulator.SubSystem.FlowIterator.Data.L2.MonitoredVarID = 7
Simulator.SubSystem.FlowIterator.Data.MaxSteps = 5
Simulator.SubSystem.FlowIterator.Data.Norm= -4.
Simulator.SubSystem.FlowIterator.Data.PrintHistory = true.
```

Here, the value Norm=-4 can be changed depending on the desired accuracy.

The final step involves specifying which file should be saved and at what frequency. For example:

```
Simulator.SubSystem.CFmesh1.FileName = corona.CFmesh
Simulator.SubSystem.CFmesh1.SaveRate = 20
Simulator.SubSystem.CFmesh1.AppendTime = false
Simulator.SubSystem.CFmesh1.AppendIter = true
```

4.2.2 Full-MHD setup

In the full MHD setup, a few settings are changed compared to the polytropic case CFcase files. The correct values for the given settings will be listed below.

```
Simulator.SubSystem.MHD3DProjectionDiff.ConvTerm.gamma = 1.67
Simulator.SubSystem.MHD3DProjectionDiff.ConvTerm.refSpeed = 3.
Simulator.SubSystem.FlowNamespace.PhysicalModelType = MHD3DProjectionDiff
Simulator.SubSystem.FlowNamespace.PhysicalModelName = MHD3DProjectionDiff
Simulator.SubSystem.Tecplot1.Data.DataHandleOutput.CCSocketNames = gravity
Manchester RadiativeLossTerm wavepressure
Simulator.SubSystem.Tecplot1.Data.DataHandleOutput.CCVariableNames = g@4 heating
radloss pw
Simulator.SubSystem.Tecplot1.Data.DataHandleOutput.CCBlockSize = 1 1 1 1
Simulator.SubSystem.Flow.Data.DiffusiveFlux = NavierStokesMHD
```

```
Simulator.SubSystem.Flow.Data.DerivativeStrategy = Corrected3D
Simulator.SubSystem.Flow.Data.NavierStokesMHD.isRadiusNeeded = true
Simulator.SubSystem.Flow.Data.DiffusiveVar = Prim
Simulator.SubSystem.Flow.Data.MHDConsACAST.PevtsovHeating = 1
Simulator.SubSystem.Flow.Data.MHDConsACAST.Manchester = 1
Simulator.SubSystem.Flow.Data.MHDConsACAST.ManchesterHeatingAmplitude = 100.
Simulator.SubSystem.Flow.Data.MHDConsACAST.Qh4H_const = 4.0e-5
Simulator.SubSystem.Flow.Data.MHDConsACAST.Qlio_AR = 5.  # This is to control
active region strength in Qhlio
Simulator.SubSystem.Flow.Data.MHDConsACAST.P0_W=0.01
Simulator.SubSystem.Flow.Data.MHDConsACAST.alfven_pressure = 0
Simulator.SubSystem.Flow.Data.MHDConsACAST.dpdxalfven = 0
Simulator.SubSystem.Flow.Data.MHDConsACAST.Qh2_activate = 0
Simulator.SubSystem.Flow.Data.MHDConsACAST.Qh3_activate = 0 # Activates heating
in Eq. 19
Simulator.SubSystem.Flow.Data.MHDConsACAST.Qh4_activate = 1 # Activates heating
in Eq. 20
Simulator.SubSystem.Flow.Data.MHDConsACAST.Qh_lio_activate = 0 # Activates
heating in Eq. 21
Simulator.SubSystem.Flow.Data.MHDConsACAST.RadiativeLossTerm = 1
```

These settings must be checked in the CFcase file so that the correct values are set and the full MHD module of COCONUT is activated. `Qh2_activate` activates the heating in the following form $Q_h = H_0|\mathbf{B}|$ with `H_0`$\bar{Q}$`h4H_const`. The parameter `Qlio_AR` controls the strength of the active region in the boundary magnetic map in Gauss.

When changing between the heating profiles in the simulation from the CFcase file the compilation of the code is unnecessary, as the values are passed from the CFcase file. When the settings are changed in `MHDConsACASourceTerm.cxx` file, then the code needs to be re-compiled.

4.3. CFmesh files

The CFmesh files are files in which COOLFluiD stores information regarding the geometry of the case and/ or the full solution. In the case of the former, the CFmesh file acts as a grid file that defines the geometry of the domain and the location and orientation of the domain boundaries. In the latter case, it still contains all the geometry and boundary information but also an additional array with resolved cell states.

The CFmesh file that acts as a grid file contains, in the specific order,

- cell node connectivity,
- boundary face node connectivity, and
- list of node coordinates,

whereas the solution CFmesh file contains

- cell node connectivity,

- boundary face node connectivity,
- list of node coordinates and
- cell states.

4.4. Interactive files

Interactive files are small files containing settings that can be changed while the simulation runs - by modifying and saving these values in the file and saving it. For example, the interactive file available in the example test case on Github[6] contains

```
Simulator.SubSystem.FlowIterator.Data.CFL.Interactive.CFL = 2.0
Simulator.SubSystem.Flow.Data.LinearLS3D.limitRes = -10
Simulator.SubSystem.Flow.Data.LinearLS3D.limitIter = 10000
Simulator.SubSystem.Flow.Data.LinearLS3D.gradientFactor = 1.0
Simulator.SubSystem.Flow.Data.LinearLS3D.StopLimiting = 0
```

where the first line is the interactive CFL values, the second is the limit residual value (after which the simulation terminates), the third one is the number of iterations after which the simulation terminates, the fourth makes the scheme second-order accurate in space (first-order accurate would be indicated by a 0) and the last line allows the user to stop limiting if they set the value to 1.

4.5. Data files

The data files, with an extension `.dat`, contain the magnetic field information for the simulation. This data is organized in four columns, the coordinates and the radial magnetic field, `x, y, z,` and `Br`. All of these values are nondimensional (normalized by $R_\odot$ for the coordinate information and by $2.2 \cdot 10^{-4}$ T for B_r). The magnetic field in the file is then interpolated on the inner boundary of the simulation. A python script is available to extract the specific magnetogram data and post-process it to smooth out the values (see Ref. [15], Ref. [12] and Ref. [7] for details regarding this post-processing). This can be provided on request.

4.6. Job submission files

The job submission files are the files required to submit the job to a queue. These are generally required when the software is run on a supercomputer center, and the task must be sent to a queuing system instead of just running it on a core/node directly. For the group at CmPA, the submission script can look like the following

```
#!/bin/bash -l
#SBATCH --account=""
#SBATCH --job-name=""
#SBATCH --mail-type="END,FAIL"
#SBATCH --mail-user=""
```

[6] https://github.com/andrealani/COOLFluiD/blob/master/plugins/MHD/testcases/COCONUT_Dipole/map.inter

```
#SBATCH --nodes="4"
#SBATCH --ntasks-per-node="36"
#SBATCH --ntasks="144"
#SBATCH --time="10:00:00"
module --force purge
module load cluster/genius/batch
module load CMake/3.20.1-GCCcore-10.3.0
module load Boost/1.76.0-GCC-10.3.0
module load ParMETIS/4.0.3-gompi-2021a
module load PETSc/3.15.1-foss-2021a
export OMP_NUM_THREADS=1
mpirun ./coolfluid-solver --scase /COOLFluiD_Base/.../testcase.CFcase
```

when using to the Genius cluster of the Flemish Supercomputer Center, VSC, with the slurm job management system. How the submission script should be formatted is, however, directly dependent on the system in use; and thus each user must set it up according to the environment and architecture of their own machine.

4.7. Output files

COCONUT runs produce a variety of output files, namely

- `convergence.plt-P0.EMNamespace`: that documents the convergence of the Poisson solver at the beginning of the simulation (to create the initial condition for the magnetic field),
- `convergence.plt-P0.FlowNamespace`: that documents the convergence of the flow solver, where for each iteration, the residual values are given in the order of ρ, V_x, V_y, V_z, B_x, B_y, B_z, p, ψ and followed up by the current CFL number (which is only relevant for steady-state simulations), physical time and time-step (which is only relevant for time-accurate simulations), wall time and memory use,
- `.vtu` solution files for Paraview: where the `corona-poisson` files contain the Poisson solution and the `corona-flow0` files the flowfield (note that each file contains the partition information from only one core, and so all of them must be loaded to Paraview to draw the complete flowfield),
- `.plt` solution files for Tecplot: where the `corona_poisson` file contains the Poisson solution and the `corona` file the flowfield solution, and
- `corona.CFmesh` solution file: formatted in the native COOLFluiD format, with the complete solution information and the geometry, which is also the file that can be used for restarting simulations.

How frequently the solution and graphical files are saved can be controlled in the `.CFcase` file. For example, in the line

```
Simulator.SubSystem.CFmesh1.SaveRate = 500
```

one can change the output frequency of the `.CFmesh` solution file (here, by default, the output is written every 500 iterations). The frequency of the `.plt` file can be adjusted in

```
imulator.SubSystem.Tecplot1.SaveRate = 500
```

and the frequency of the `.vtu` files in

```
Simulator.SubSystem.ParaView1.SaveRate = 1000.
```

If one wants to save the files separately each time, such that they do not overwrite each other, it is also possible to append the current iteration (or the time for time-accurate runs) to the file name by setting the corresponding lines

```
Simulator.SubSystem.CFmesh1.AppendTime = false
Simulator.SubSystem.CFmesh1.AppendIter = false

Simulator.SubSystem.Tecplot1.AppendTime = false
Simulator.SubSystem.Tecplot1.AppendIter = false

Simulator.SubSystem.ParaView1.AppendTime = false
Simulator.SubSystem.ParaView1.AppendIter = false
```

to `true` where appropriate.

5. Running a simulation

In the job submission script above, the line

```
mpirun ./coolfluid-solver -scase /COOLFluiD_Base/.../testcase.CFcase
```

is the actual simulation execution line. To be able to execute this line, one needs to have the `coolfluid-solver` executable in the current directory. In practice, we typically create a symbolic link, which, for the structure defined here, would look like

```
ln -sf COOLFluiD_base/OPENMPI/optim/apps/Solver/coolfluid-solver* .
```

see the Github documentation[7] for more information.

In principle, COCONUT can run on any number of cores. In practice, however, one should note that there are certain simulation memory requirements, and as a result, just one or two cores with limited memory might not be sufficient to support the run. How many cores are available and how much memory they have available depends on the properties of the machine of the user. Small simulations, such as the dipole on a coarser grid, may require around ~ 200 GB.

The user should be aware that the performance of the code decreases to a certain extent if the number of cores is too high. Especially for simpler cases on a coarser grid, employing too many cores becomes inefficient due to the fact that the communication between the cores would take up more resources than the computation itself. As an indication of this, Figures 11 and 12 show the scaling of the coding efficiency with an increasing number of cores for a simple case and for a large, data-driven case, respectively, as computed on the Hortense cluster of the Flemish supercomputer center. Naturally, these profiles might behave differently on other systems, and so the optimal number of cores one should select would also vary. The trends are, however, expected to look similar (a higher number of cores being more optimal for more complex cases).

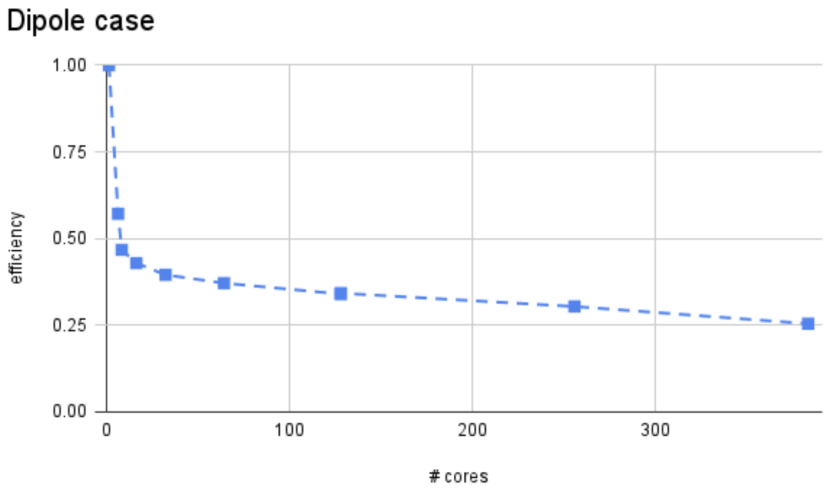

Figure 11: Scaling of COCONUT running a simple dipole case on the Hortense cluster of VSC.

[7] `https://github.com/andrealani/COOLFluiD/wiki/HOWTO-run-a-testcase`

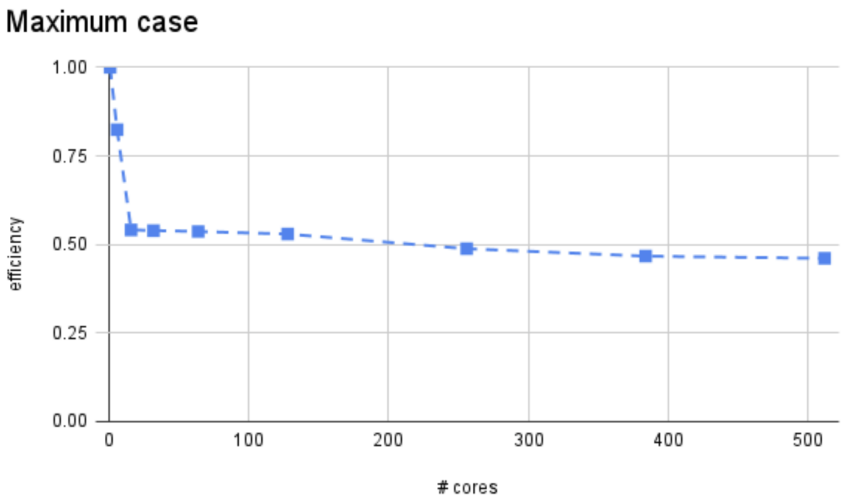

Figure 12: Scaling of COCONUT running a large data-driven case on the Hortense cluster of VSC.

Once the simulation starts, the parameters that were defined in the interactive `.inter` file can be changed according to the user's preference; e.g., after a few cpu-minutes, the interactive CFL number can be increased for steady-state simulations or the time step for time-accurate simulations. The user can simply open the interactive file, modify the value desired, save it and close it again.

6. Monitoring simulation progress

Perhaps the most straightforward method to assess the simulation progress is to monitor the direct code output, which is either on the screen for interactive runs or in a job output file for runs submitted through a queuing system (the name of the job output file depends directly on the queuing system used).

The code output will show how the simulation is set up, i.e., which files are read, which settings are used (and unused if there are any), how the mesh is built, and so on. Thus, if the solver fails in the early stages of the run, going through the direct output can help identify what might be the cause. For example, the direct code output will indicate if the mesh file was not found if a certain path does not exist, or if a certain function was incorrectly formatted.

Once the initial solution (iteration zero) is written, the code output will show the convergence of the Poisson solver to create the initial condition if the Poisson solver was selected to generate it (not the case of simulations that are restarted). These lines would look as follows (the values serve just as examples),

```
KSP convergence reached at iteration:  683
Newton Step:  1 L2 dU: 2.13769 CFL: 1
KSP convergence reached at iteration:  1030
Newton Step:  2 L2 dU: -6.55279 CFL: 1
KSP convergence reached at iteration:  586
Newton Step:  3 L2 dU: -13.3397 CFL: 1
```

and indicate the progression of the Newton solver at converging to the Poisson solution. Note that the `dU` value should decrease between the subsequent iteration, indicating convergence. If this solver does not converge sufficiently or diverges, the flow solver will initiate itself from a highly unphysical initial condition and might, as a result, also have difficulty converging or diverging right away.

The Poisson solution is transferred afterward,

```
FVMCC_BaseBC<BASE>::preProcess() => TRS[ Inlet ] => START
FVMCC_BaseBC<BASE>::preProcess() => TRS[ Inlet ] => END
```

and the flow solver starts,

```
KSP convergence reached at iteration:  20
[ FlowSubSystemStatus ] Iter:  1 Res:  [ 0.64339456 -0.56274774 -0.59309443
-0.62177474 -0.015717174 -0.12452016 -0.31613439 0.99829793 1.7340697 ] CFL:
1 CPUTime:  6.540045 Mem:  366.125 MB
```

These output lines indicate, for each iteration, how many sub-iterations were required and what the current residuals are for the primitive variables, in the order of ρ, V_x, V_y, V_z, B_x, B_y, B_z, p, ψ (see Section 2.6 for the description of the residual value)

The flow solver will keep computing until,

- it converges: the target residual in the prescribed variable or the maximum number of iterations has been reached, or

- it diverges: the simulation crashes (see Section 9 for a discussion of the possible causes and how to solve them), or
- the code either runs out of memory or the allocated job time or
- the code is terminated by the user.

If the simulation crashes due to the divergence of the solver, the flow solver output lines will indicate very large residuals, e.g., `KSP convergence reached at iteration: 0`

```
[ FlowSubSystemStatus ] Iter: 7425 Res: [ -1.7976931e+308 -1.7976931e+308
-1.7976931e+308 -1.7976931e+308 -1.7976931e+308 -1.7976931e+308 -1.7976931e+308
-1.7976931e+308 -1.7976931e+308 ] CFL: 2 CPUTime: 14049.378 Mem: 589.344
MB
```

after which the termination occurs.

Example residual curves are shown in Figure 13. Here, the V_x residuals are plotted through `gnuplot` for two simulations. One can see that after a few thousands of iterations, the residuals drop from roughly -1 to -4. One can also see discontinuities in the residual curves, which are caused by increasing the CFL number (here, always by a factor of two). Since the CFL number effectively defines the time-step, if it is increased, it also means that larger changes occur in the flowfield between consecutive iterations, hence the residual jump.

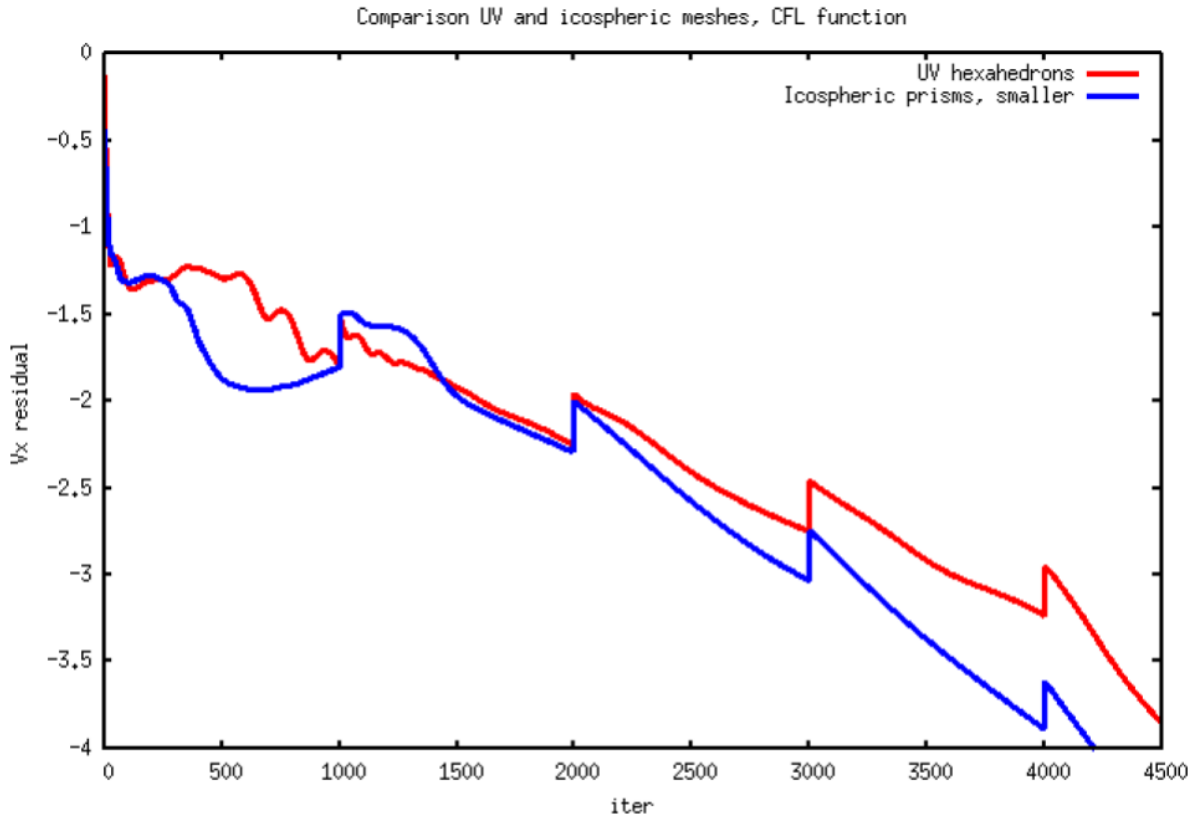

Figure 13: Figure from Ref. [8]. Shown are example residual curves of V_x throughout two simulations.

7. Plotting results and diagnostics

As outlined in Section 4.7, there is a variety of files that can be used to inspect the final solution. Naturally, the convergence progress (residual evolution) from the convergence file can be plotted with software such as gnuplot, as seen above. For the graphical files, generally, we make use of either Tecplot, which can read the `.plt` files, or Paraview, which reads the `.vtu` files. Here, example diagnostic procedures with Tecplot are very briefly demonstrated.

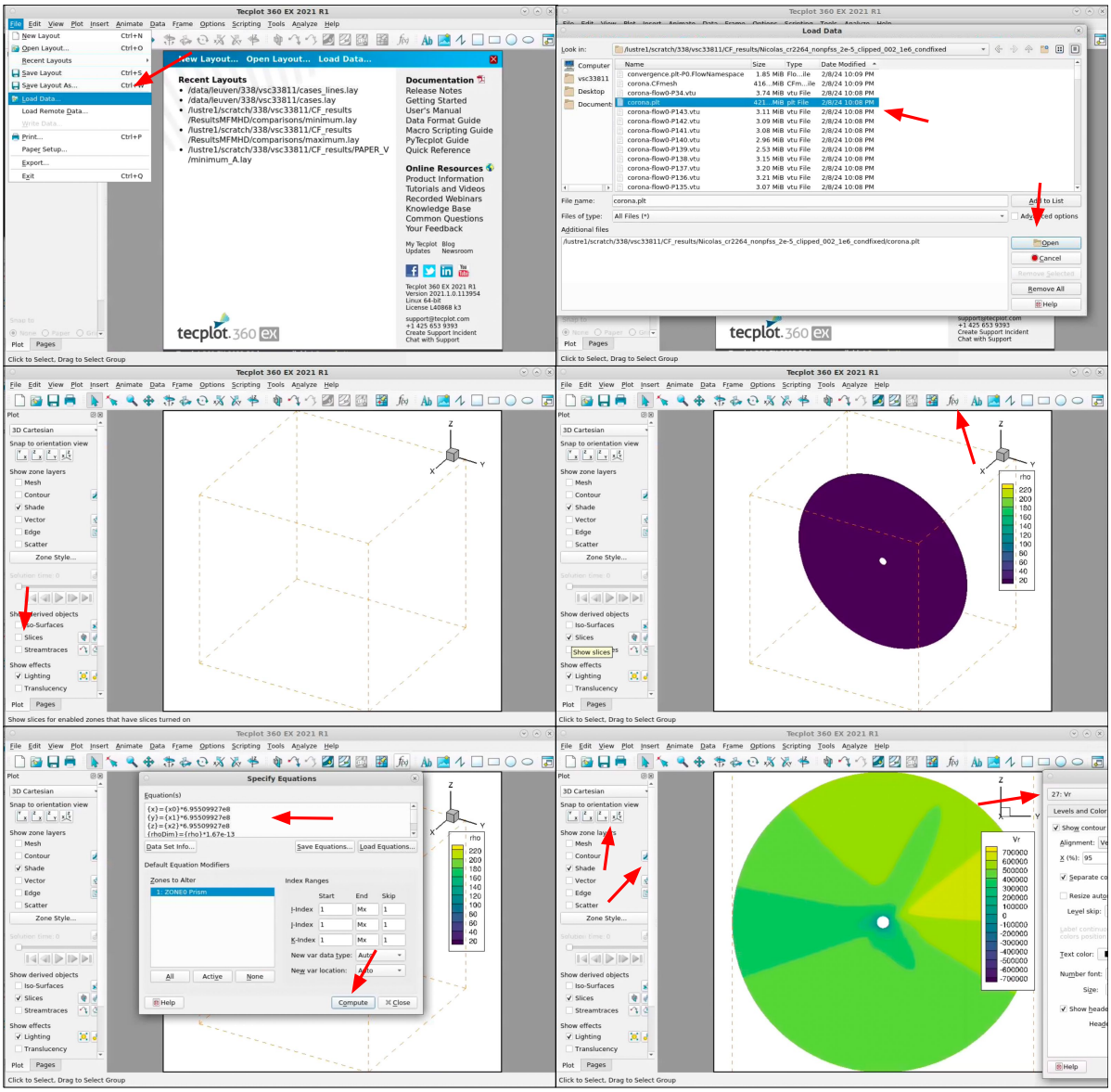

Figure 14: Examples of useful procedures in Tecplot: loading the correct output files, plotting a slide of the domain, adding formulas to convert adimensional Cartesian primitive variables to dimensional spherical variables, and visualising these.

Some of the basic procedures to load data and visualize them are shown in Figure 14. The easiest method to diagnose the data is to plot domain slices, since a full "contour" plot would only show the outer boundary. In the given coordinate system, the "z" axis points to the north pole of the Sun.

In addition, the following set of equations can be imported to Tecplot (in the corresponding formatting) to transform the COCONUT output variables (nondimensional, Cartesian) to dimensional and spherical values:

```
{x}={x0}*6.955e8
{y}={x1}*6.955e8
{z}={x2}*6.955e8
{rhoDim}={rho}*1.67e-13
{uDim}={u}*480248.4
{vDim}={v}*480248.4
{wDim}={w}*480248.4
{BxDim}={Bx}*2.2e-4
{ByDim}={By}*2.2e-4
{BzDim}={Bz}*2.2e-4
{pDim}={p}*0.03851
{r}=sqrt({x}*{x}+{y}*{y}+{z}*{z})
{rnondim} = {r}/6.9551e8
{rXY}=sqrt({x}*{x}+{y}*{y})+1.0e-20
{Vr}={x}/{r}*{uDim}+{y}/{r}*{vDim}+{z}/{r}*{wDim}
{VTheta}={x}*{z}/({rXY}*{r})*{uDim}+{y}*{z}/({rXY}*{r})*{vDim}-
{rXY}/{r}*{wDim}
{VPhi}=-{y}/{rXY}*{uDim}+{x}/{rXY}*{vDim}
{Br}={x}/{r}*{BxDim}+{y}/{r}*{ByDim}+{z}/{r}*{BzDim}
{BTheta}={x}*{z}/({rXY}*{r})*{BxDim}+{y}*{z}/({rXY}*{r})*{ByDim}-
{rXY}/{r}*{BzDim}
{BPhi}=-{y}/{rXY}*{BxDim} + {x}/{rXY}*{ByDim}
{T} = {pDim}/(2*{rhoDim}/1.6e-27)/1.38e-23
```

To plot magnetic field lines, one can make use of the "Streamtraces" diagnostics available in Tecplot in the left panel. After selecting the "Streamtraces" option, one is asked to indicate the names of the velocity field component variables in order to compute the streamlines; so if we provide the magnetic field components instead ("Bx", "By" and "Bz"), the "Streamtraces" diagnostics will plot the magnetic field lines.

The Tecplot user manual is available online[8].

[8] https://www.scc.kit.edu/downloads/sca/tpum.pdf

8. Checking your results numerically

COCONUT can produce a wide variety of results, but not all of them will be reasonable. The code, after all, can only solve what it is prescribed to solve by the user. There are a variety of convergence checks one should perform to determine whether the settings used are reasonable and lead to (numerically) accurate results. These include the following.

8.1. Grid convergence and grid rotation tests

In principle, to determine whether the observed phenomena are accurate and whether important features might be missing/ false features might appear, the following two tests should be performed.

First, the solution should be resolved on a finer grid, both longitudinally/ latitudinally and radially. In principle, it is expected that on a finer grid, more structures with better-pronounced shapes will be present (whether proper resolution of these additional features is required depends on the intended use of the code). However, if the mesh refinement leads to significant changes in the already existing structures, most likely a finer resolution level is necessary, and even a level finer resolution might be needed to perform further convergence checks. This might happen if there are small but very strong active regions that determine the shape of the flowfield and that are not resolved sufficiently on the coarser grid.

Second, as discussed both in Ref. [3] and in Ref. [8], spurious fluxes might occur in the domain in non-dominant dynamics as a result of grid deformities. Thus, if a pronounced feature is visible in the regions of the mesh knots or knot lines, one should re-run the simulation with the grid/ magnetogram rotated (e.g., around the Z-axis by a few degrees) to evaluate whether the shape of this feature might be affected or caused by spurious fluxes.

8.2. Steady-state residual convergence checks

As discussed in the sections above, one may terminate a steady-state simulation once a certain target residual is reached by one of the primitive variables, say, -3 in ρ (see Section 2.6). However, it is impossible to exactly predict which residual will be sufficient to claim convergence, especially for a new test case.

As discussed in Section 2.6, the residual expresses the logarithm of the sum of the changes of the given variable in the flowfield between two subsequent iterations. However, most of the variables in the flowfield have large variations in them, by even several orders of magnitude. Thus, a residual of ≤ -3 in density might be sufficient to resolve the density structures near the inner boundary, where $\rho \sim 0.1$ to 1 in code units (so the residual would indicate the sum of changes equivalent to 0.1% to 1%), but insufficient to resolve the structures near the outer boundary, where $\rho \sim 10^{-4}$ to 10^{-5}.

If one sets up a new case, perhaps the most straightforward method to determine which residual is sufficient is to run the simulation at least once to almost complete convergence, e.g., residuals of ~ -10. One then has to make sure to separately save the graphical output files for each few hundred of iterations that correspond to the different residual levels, e.g., by appending the iteration number to the names of the graphical files through setting

```
Simulator.SubSystem.Tecplot1.AppendIter = true
```

for Tecplot (or, the same line can be set for Paraview files) as shown above. Then,

one may compare the final solution to the solutions at different convergence levels (e.g., by computing the relative differences and plotting these in the flowfield) to determine at which iteration - and thus, at which residual levels - the flowfield was acceptably close (considering the purposes of the simulation) to the final solution.

In some cases, a complete convergence down to the level of numerical accuracy is impossible. This may be due to the fact that e.g., the simulation is very large and such a run would require resources that are simply not available, or due to the fact that a completely steady-state solution does not even exist. The latter might occur for some cases, where there may exist regions in the flowfield that exhibit oscillations and other transient phenomena (e.g., reconnection in current sheets). If that happens, the minimum residual that the simulation may reach will be constrained by the changes in this unsteady region. Convergence, however, may still be reached in the rest of the flowfield. To ascertain if this is the case, it is possible to plot consecutive solutions, e.g., every few hundred of iterations, in order to confirm that there are no longer changes to the rest of the flowfield (apart from the unsteady region) by evaluating the respective relative differences between the solutions.

Alternatively, in both cases, one may also examine the residuals themselves in the graphical output files instead of separately computing and plotting the relative differences. To do that, however, one must add the appropriate socket into the `.CFcase` setup file with the name `rhs` containing the variable names `rhs0, rhs1, rhs2, rhs3, rhs4, rhs5, rhs6, rhs7, rhs8`. For Tecplot, for example, one would add,

```
Simulator.SubSystem.Tecplot.Data.DataHandleOutput.CCSocketNames = rhs
Simulator.SubSystem.Tecplot.Data.DataHandleOutput.CCVariableNames =
rhs0 rhs1 rhs2 rhs3 rhs4 rhs5 rhs6 rhs7 rhs8
Simulator.SubSystem.Tecplot.Data.DataHandleOutput.CCBlockSize = 1
```

after which the residuals for each primitive variable (the same default ordering, so `rhs0` corresponding to the density residual) will be available next to all the primitive variables in the list of the output arrays.

8.3. Time-accurate convergence checks

Simulator.SubSystem.FlowSubSystemStatus.TimeStep = 1e-2 **Luis finish?**

In time-accurate simulations, the two parameters that largely define time-stepping are the time step and the relative tolerance to be reached in the sub-iterations between consecutive time steps. A time step that is too large will march the solution unphysically fast and lead to altered dynamics of the structures in the domain. A relative tolerance setting that is too high will lead to insufficiently converged solutions between the time steps and, thus, also lead to an unphysical development.

Thus, in time-accurate simulations, after obtaining baseline results, one should re-run the same simulation at least twice, once with a smaller time step (e.g., half of the size) and once with a smaller relative tolerance (e.g., an order of magnitude lower). Keep in mind that both of these settings will require more computational time for running, and the later might also require a higher value of the maximum number of sub iterations

9. Convergence issues

Since COCONUT is a CFD software, there is no guarantee that the solver will converge - and that it will converge to a physical solution - and it is up to the user to provide a suitable physical and numerical setup to achieve convergence.

The first step towards understanding why the code did not run was plotting the solution before the simulation crashed. This is typically the easiest way to determine in which regions the solution evolves improperly and which parts of the numerical and physical setup may be responsible for that. For example, if one sees oscillations or unphysical evolution near the inner or outer boundary, the boundary prescription might be to blame.

Generally, the following are some of the frequent reasons why the code crashes.

9.1. Too steep CFL and time-step profiles

If the initial flowfield is very different from the flowfield we are marching towards, whether this marching is done in a steady-state or a time-accurate fashion, large changes are bound to occur in the domain between the subsequent iterations. Especially in the beginning of the simulations, when changes between these iterations are large (one can judge that from the residual levels), it is important to keep the CFL number (steady-state runs) or the time-step (time-accurate runs) modest. Once the residuals drop, these parameters can be increased. For steady-state runs, it is good practice to monitor the residual level throughout the simulation, and once they have been consistently decreasing for a few hundred iterations, the CFL number can be increased (see, e.g., Figure 13 and the corresponding discussion).

9.2. Gradients that cannot be accommodated by the grid

Especially for data-driven runs, it may happen that one wishes to simulate cases that contain very strong, localized gradients, such as those corresponding to solar maxima with strong localized active regions. If there are not enough cells to capture these gradients properly, naturally, the code may diverge.

If it is not necessary to capture the gradients completely, one may then opt to increase the strength of the limiter or to simply prescribe a more filtered/ smoothed magnetic field (e.g., if smoothing is done via the spherical harmonics projection, one simply includes fewer harmonics; see Ref. [12] for the discussion of magnetic map smoothing). If capturing these gradients is necessary, it may be useful to increase the resolution of the mesh (both surface and radial) to be able to capture all the features; see Figure 5 and the corresponding discussion.

9.3. Boundary and initial conditions inconsistency

If the code is prescribed to start from an initial state that is not at all consistent with the prescribed boundary conditions (especially the inner one in our case), very large changes will take place in the flowfield close the the boundary in the first iterations. For example, if the initial flowfield contains a nondimensional density of 1 near the inner boundary, but the boundary condition prescribes the value of 20, a very large gradient will develop in the flowfield initially, which may lead to the solver diverging. One must thus ensure that the initial state conforms to the boundary prescriptions, at least to a certain extent.

If this is not possible due to the nature of the simulation, one may then start the simulation with a much smaller value of CFL, e.g., 0.1 or even less (or equivalent to the time step).

This will result in very small time steps with which the code may be able to handle the large changes in the flowfield. After a tens or few hundreds of iterations, when the residuals near the boundaries drop as the flowfield adjusts itself to the boundary, one may then set the CFL back to 1 and then progressively higher.

9.4. Exceeding the divergence cleaning reference speed

The systems (1) and (2) contain the V_{ref} term to denote the divergence cleaning reference speed. This value is typically set to 1 (for polytropic) or 3 (for full MHD), which translates into ~ 480 km/s and ~ 1440 km/s, respectively. These values are generally sufficient for the cases that we typically run.

However, it might happen in some simulations that the plasma speed exceeds these values or that these values are assigned improperly. For example, if a full MHD case is run with a reference velocity of 480 km/s instead of 1440 km/s with the intention to resolve the solar wind of speeds of > 700 km/s, the divergence cleaning technique is not "fast enough" to clear $\nabla \cdot \mathbf{B}$ from the flowfield. Thus, if the simulation crashes and one sees very high speeds developing in the flowfield, even if locally, it may be useful to check whether these have not exceeded the value of V_{ref}, and if so, adjust the value of V_{ref} accordingly.

9.5. Unphysical setup

While having an unphysical setup might not always result in the divergence of the solver, it may lead to unphysical features in the solution, see, e.g., the unphysical streams described in Ref. [4]. Obviously, there are many ways in which the setup can be made unphysical, but some of the common ones include:

- prescribing a magnetic field that is too strong without the corresponding adjustments to the boundary pressure and density, see Section 2.4,
- not adjusting the coronal heating approximation for the specific case (large prescribed magnetic fields will, for Equation (7), lead to a very high heating),
- marching time-accurate simulations with an unphysically large time step, and
- having a domain that is too small for the flow to become supersonic and super-Alfvénic, see Section 2.4.

10. Frequently asked questions

11. Further questions and issues

If you have further issues or questions not addressed in this documentation, please turn to andrea.lani@kuleuven.be.

References

[1] R. G. Athay. Radiation Loss Rates in Lyman Alpha for Solar Conditions. , 308:975, Sept. 1986.

[2] T. Baratashvili, M. Brchnelova, A. Lani, and S. Poedts. The full 3D MHD model from Sun to Earth: COCONUT + Icarus. Astronomy and Astrophysics, submitted, Jan. 2024.

[3] M. Brchnelova. Towards faster and more physical MHD and multi-fluid global coronal models. PhD thesis, KU Leuven, 2024.

[4] M. Brchnelova, B. Gudiksen, M. Carlsson, A. Lani, and S. Poedts. Constraining the inner boundaries of COCONUT through plasma β and Alfvén speed. planned for submission, Feb. 2024.

[5] M. Brchnelova, B. Kuźma, B. Perri, A. Lani, and S. Poedts. To E or Not to E : Numerical Nuances of Global Coronal Models. The Astrophysical Journal Supplements Series, 263(1):18, Nov. 2022.

[6] M. Brchnelova, B. Kuźma, F. Zhang, A. Lani, and S. Poedts. COCONUT-MF: Two-fluid ion-neutral global coronal modelling. Astronomy and Astrophysics, 678:A117, Oct. 2023.

[7] M. Brchnelova, B. Kuźma, F. Zhang, A. Lani, and S. Poedts. The role of plasma β in global coronal models: Bringing balance back to the force. Astronomy and Astrophysics, 676:A83, Aug. 2023.

[8] M. Brchnelova, F. Zhang, P. Leitner, B. Perri, A. Lani, and S. Poedts. Effects of mesh topology on MHD solution features in coronal simulations. Journal of Plasma Physics, 88(2):905880205, Apr. 2022.

[9] C. Downs, I. I. Roussev, B. van der Holst, N. Lugaz, I. V. Sokolov, and T. I. Gombosi. Toward a Realistic Thermodynamic Magnetohydrodynamic Model of the Global Solar Corona. , 712(2):1219–1231, Apr. 2010.

[10] J. Guo, L. Linan, S. Poedts, Y. Guo, A. Lani, B. Schmieder, M. Brchnelova, B. Perri, T. Baratashvili, Y. W. Ni, and P. F. Chen. Modelling the propagation of coronal mass ejections with COCONUT: implementation of the Regularized Biot-Savart Laws flux rope model. arXiv e-prints, page arXiv:2311.13432, Nov. 2023.

[11] A. Harten. High resolution schemes for hyperbolic conservation laws. Journal of Computational Physics, 49(3):357–393, 1983.

[12] B. Kuźma, M. Brchnelova, B. Perri, T. Baratashvili, F. Zhang, A. Lani, and S. Poedts. COCONUT, a Novel Fast-converging MHD Model for Solar Corona Simulations. III. Impact of the Preprocessing of the Magnetic Map on the Modeling of the Solar Cycle Activity and Comparison with Observations. The Astrophysical Journal, 942(1):31, Jan. 2023.

[13] L. Linan, F. Regnault, B. Perri, M. Brchnelova, B. Kuzma, A. Lani, S. Poedts, and B. Schmieder. Self-consistent propagation of flux ropes in realistic coronal simulations. Astronomy and Astrophysics, 675:A101, July 2023.

[14] R. Lionello, J. A. Linker, and Z. Mikić. Multispectral Emission of the Sun During the First Whole Sun Month: Magnetohydrodynamic Simulations. , 690(1):902–912, Jan. 2009.

[15] B. Perri, B. Kuźma, M. Brchnelova, T. Baratashvili, F. Zhang, P. Leitner, A. Lani, and S. Poedts. Coconut, a novel fast-converging mhd model for solar corona simulations. ii. assessing the impact of the input magnetic map on space-weather forecasting at minimum of activity. The Astrophysical Journal, 943(2):124, feb 2023.

[16] B. Perri, P. Leitner, M. Brchnelova, T. Baratishvili, B. Kuźma, F. Zhang, A. Lani, and S. Poedts. COCONUT, a novel fast-converging MHD model for solar corona simulations: I. Benchmarking and optimization of polytropic solutions. The Astrophysical Journal, 936(1), 2022.

[17] A. A. Pevtsov, G. H. Fisher, L. W. Acton, D. W. Longcope, C. M. Johns-Krull, C. C. Kankelborg, and T. R. Metcalf. The Relationship Between X-Ray Radiance and Magnetic Flux. , 598(2):1387–1391, Dec. 2003.

[18] R. Rosner, W. H. Tucker, and G. S. Vaiana. Dynamics of the quiescent solar corona. , 220:643–645, Mar. 1978.

[19] V. Venkatakrishnan. On the accuracy of limiters and convergence to steady state solutions. 1993.

[20] H. P. Wang, S. Poedts, A. Lani, M. Brchnelova, T. Baratashvili, L. Linan, F. Zhang, D. W. Hou, and Y. H. Zhou. An efficient, time-evolving, global MHD coronal model based on COCONUT. Astronomy and Astrophysics, submitted, 2024.

[21] M. S. Yalim, D. Vanden Abeele, A. Lani, T. Quintino, and H. Deconinck. A finite volume implicit time integration method for solving the equations of ideal magnetohydrodynamics for the hyperbolic divergence cleaning approach. Journal of Computational Physics, 230(15):6136–6154, July 2011.