CRONOS Manual

Jens Kleimann <jk@tp4.rub.de>
Draft of 17 October 2018

Contents

1 Introduction **3**
- 1.1 Disclaimer . . . 3
- 1.2 Requirements . . . 3
- 1.3 Notable features . . . 3
- 1.4 Grid layout and coordinates . . . 3
- 1.5 Equations . . . 4

2 Installation **5**
- 2.1 Parallel HDF output . . . 6
- 2.2 Working with git . . . 6

3 Setting up a simulation **7**
- 3.1 Files to be modified by the User . . . 7
- 3.2 The Simulation hierarchy . . . 7
- 3.3 Entries of `constants.H` . . . 8
  - 3.3.1 Keywords . . . 8
  - 3.3.2 Flags . . . 9
  - 3.3.3 Variables . . . 10
- 3.4 The energy integration . . . 10
- 3.5 Corotating frame of reference . . . 11
- 3.6 Entropy correction . . . 11
- 3.7 Choice of Riemann solvers . . . 12
- 3.8 Setting up the mod file . . . 12
  - 3.8.1 Read parameters . . . 13
  - 3.8.2 Initial conditions . . . 13
  - 3.8.3 Boundary conditions . . . 14
  - 3.8.4 Source terms . . . 14
  - 3.8.5 Coordinate singularities . . . 15
- 3.9 Extended equations . . . 16
  - 3.9.1 Add new equations . . . 16
  - 3.9.2 Boundary conditions for additional variables . . . 17
  - 3.9.3 Source terms for additional variables . . . 17
  - 3.9.4 Extend existing equations . . . 18
- 3.10 The cat file . . . 18
- 3.11 Non-linearly scaled grids . . . 20
  - 3.11.1 Employing a pre-defined mapping . . . 21

3.11.2 Implementation of a user-defined mapping . . . 21
3.12 How to compile and run . . . 22
3.12.1 Starting a single-CPU run . . . 23
3.12.2 Starting a parallel run . . . 23
3.13 Makefile . . . 23

4 Output **24**
4.1 HDF double . . . 24
4.2 HDF float . . . 24
4.3 HDF grid . . . 24
4.4 HDF mov . . . 24
4.5 ASCII . . . 25

5 Further data processing **25**
5.1 Merging HDF files from MPI runs . . . 25
5.2 CronosPlot (IDL) . . . 25
5.3 Paraview . . . 25

6 Troubleshooting **26**
6.1 When compiling . . . 26
6.2 At runtime . . . 26
6.3 git-related . . . 27

Bibliography **27**

1 Introduction

This text gives a brief introduction to the numerical 3D-MHD simulation code Cronos. For a peer-reviewed reference paper see Kissmann et al. (2018). The code is written in C++ and maintained by Ralf Kissmann <ralf.kissmann@uibk.ac.at>. Just like this document, it is in a state of continuous modification, and comes with no warranties whatsoever.

1.1 Disclaimer

Both the code itself and this manual are not to be distributed without the consent of the respective authors, except to colleagues working on the same project for which this consent has already been granted.

1.2 Requirements

The code has so far only been used on Linux and Linux-like architectures (both 32 and 64bit). Root privileges are *not* required to compile and run. For simulations on multiple cores, a working installation of MPI (e.g. *OpenMPI* or *mpich*) is required.

1.3 Notable features

Cronos employs a semi-discrete finite-volume scheme to integrate the equations of single- or multifluid magnetohydrodynamics forward in time. For the magnetic field, a constrained transport scheme is used to keep the field divergence-free. The global timestep is adaptively chosen such that a pre-defined CFL number is not exceeded. The code can run in parallel on multiple kernels/CPUs on a single machine, as well as on multiple machines over a network connection. Binary output files are generated using HDF5 (hierarchical data format, see `https://www.hdfgroup.org/downloads/hdf5`). Currently supported coordinate systems are Cartesian, cylindrical, and spherical. The grid can be deformed in a user-definable fashion, provided that grid lines remain orthogonal.
The handling of cylindrical and spherical coordinate singularities for MHD runs is included except for the spherical origin ($r = 0$). For pure hydro runs, the latter may be implemented by the inclined user, while for MHD this is not possible because it would require modifications too deep inside the main code.

1.4 Grid layout and coordinates

Cronos uses a logically rectangular, three-dimensional grid. The cell indices i, j, k run from 0 to $N_{x,y,z} - 1$, such that an $[N_x, N_y, N_z]$ grid comprises $(N_x \times N_y \times N_z)$ cells. For constant spacing, this implies a mapping

$$x_i = x_0 + i\ \Delta x \tag{1}$$
$$y_j = y_0 + j\ \Delta y \tag{2}$$
$$z_k = z_0 + k\ \Delta z \tag{3}$$

from index space to coordinate space, where integer indices designate the cells' centers. This implies that the simulation actually covers the x interval

$$[x_{-1/2}, x_{N-1/2}] = [x_0 - \Delta x/2, x_{N-1} + \Delta x/2]$$

of length $(x_{N-1} - x_0) + \Delta x = N\ \Delta x$ in physical space (and similarly for y and z). For reasons of backward compatibility, it is also possible (though not recommended) to clip cells containing either 0 or $N-1$ in their indices, such that the covered space only extends over $x \in [x_0, x_{N-1}]$ (with $N-2$ full cells framed by two half-cells in each direction, such that edge cells only have a 1/4 of their usual volume, and corner cells a mere 1/8).
At present, Cartesian, cylindrical, and spherical coordinates are supported, with user-definable grid spacing (i.e. variable Δ's) along coordinate lines.
Fluid variables $(\rho, u_x, u_y, u_z, e, p)$ are localized at the cell centers $\mathbf{r}_{i,j,k}$, while the magnetic field components are face-centered on (and normal to) the respective cell faces. The vector potential is centered on (and pointing along) the cell edges (see Table 1).
Note that, since C++ syntax requires array elements to have integer indices, the internal index is offset from the physical one by $-1/2$ for the magnetic field and by $+1/2$ for the vector potential. This means that a shifted index $\alpha \in \{x, y, z\}$ runs from -1 (rather than 0) to $N_\alpha - 1$ for $\mathbf{B}$ and from 0 to N_α for $\mathbf{A}$, yielding a total of $N_\alpha + 1$ values in both cases[1].

vector component	physical location	index notation	range
v_x	$\mathbf{r}_{i,j,k}$	`[i,j,k]`	$[0, m]$
v_y	$\mathbf{r}_{i,j,k}$	`[i,j,k]`	
v_z	$\mathbf{r}_{i,j,k}$	`[i,j,k]`	
B_x	$\mathbf{r}_{i\pm1/2,j,k}$	`[i(-1),j,k]`	$[0(-1), m]$
B_y	$\mathbf{r}_{i,j\pm1/2,k}$	`[i,j(-1),k]`	
B_z	$\mathbf{r}_{i,j,k\pm1/2}$	`[i,j,k(-1)]`	
A_x	$\mathbf{r}_{i,j\pm1/2,k\pm1/2}$	`[i,j(+1),k(+1)]`	$[0, m(+1)]$
A_y	$\mathbf{r}_{i\pm1/2,j,k\pm1/2}$	`[i(+1),j,k(+1)]`	
A_z	$\mathbf{r}_{i\pm1/2,j\pm1/2,k}$	`[i(+1),j(+1),k]`	

Table 1: Localization of $\mathbf{A}$ and $\mathbf{B}$ components, using $m := N - 1$.

1.5 Equations

In its most basic set-up, CRONOS numerically solves the equations of ideal magnetohydrodynamics

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{4}$$

$$\partial_t (\rho \mathbf{u}) + \nabla \cdot \left[\rho \mathbf{u}\mathbf{u} + (p + \|\mathbf{B}\|^2/2)\mathcal{I} - \mathbf{B}\mathbf{B}\right] = \mathbf{0} \tag{5}$$

$$\partial_t e + \nabla \cdot \left[(e + p + \|\mathbf{B}\|^2/2)\mathbf{u} - (\mathbf{u} \cdot \mathbf{B})\mathbf{B}\right] = 0 \tag{6}$$

$$\partial_t \mathbf{B} + \nabla \times \mathbf{E} = \mathbf{0} \tag{7}$$

$$\tag{8}$$

[1] This unfortunate inconsistency merely exists for historical reasons, and will hopefully be corrected in a future revision.

subject to the closure relations

$$e = \frac{p}{\gamma - 1} + \frac{\rho \|\mathbf{u}\|^2}{2} + \frac{\|\mathbf{B}\|^2}{2} \quad (9)$$

$$\mathbf{E} + \mathbf{u} \times \mathbf{B} = \mathbf{0} \quad \text{and} \quad (10)$$

$$\nabla \cdot \mathbf{B} = 0 \, . \quad (11)$$

Here, ρ, p, and e are scalar functions of space $\mathbf{r}$ and time t, and both $\mathbf{u}$ and $\mathbf{B}$ are three-dimensional vector fields (which also depend on $\mathbf{r}$ and t). $\mathcal{I}$ is the unit tensor. Alternatively, the energy equation (6) can be replaced by an isothermal or adiabatic equation of state (see Section 3.4).

If so desired, the set of equations (4) to (6) can be augmented by additional terms, both of divergence form and as source terms on the respective right hand sides. This also includes the possibility to negate any terms by adding them again with a minus sign. In addition, the user is free to provide an arbitrary number of additional equations for additional variable fields which are to be solved simultaneously with the above set.

2 Installation

1. Contact Ralf Kissmann to obtain access to the Cronos repository at `https://git.uibk.ac.at`.

2. In your shell, create your local working directory for Cronos (e.g. `src`) and change into it using `mkdir src && cd src`

3. `git clone git@git.uibk.ac.at:c706138/CronosNumLib.git`
   This will create a directory `CronosNumLib` and several files therein.

4. `cd` into `CronosNumlib` and follow steps listed in the `README` file.

5. Move up into `src` by `cd ..` and check out Cronos code files with
   `git clone git@git.uibk.ac.at:c706138/CronosCode.git` If you have permission to write files to the repository, you should also execute `git push -u origin master`.

6. Follow steps described in `CronosCode/README.md`. In our example case, `'path-to-CronosNumLib'` is `'~/src/CronosNumLib'`. Do not use relative path specifiers!

Note: In order to compile properly on the TP4 cluster, the include and lib entries in file `CronosNumLib/makeinclude/include.mak` should read

```
X_INC = -I$(X_LIB_ROOT_DIR)/include -I/usr/lib/x86_64-linux-gnu/openmpi/include \
                                    -I/usr/include/hdf5/openmpi
X_LIB = -L$(X_LIB_ROOT_DIR)/lib/$(X_OSTYPE) -L/usr/lib/x86_64-linux-gnu/hdf5/openmpi
```

See Section 6 for additional installation troubleshooting.

2.1 Parallel HDF output

To enable parallel output of all involved processors to a single HDF5 file during MPI runs, the following steps have to be taken:

1. Install the respective hdf5 package `libhdf5-openmpi-1.8.4` (for Debian and Ubuntu). At TP4, this is pre-installed on all 8-core machines and Intel clusters.

2. Make the following changes in `src/makeinclude` :

   (a) In `include.mak`, change `X_INC = -I\$(X_ROOT_DIR)/include` to `X_INC = -I\$(X_ROOT_DIR)/include -I/usr/lib/openmpi/include`.

   (b) In the file for the respective operating system (e.g. `Linux-amd64`) use the following setting for CXX: `CXX = g++ -DOMPI_SKIP_MPICXX`
   These changes are necessary to maintain the code's capability to produce output for non-MPI simulations.

3. Create backups of your `lib` and `include` folders located in your home folder. Remove all HDF-related files in `lib` and `include`. If you want to go back to serial output for each processor, you will have to restore the backups of these folders and undo the change to `cronos/generic/Hdf5file_cbase.H` (next step).

4. Set `#define HDF_PARALLEL_IO SWITCH_ON` in `cronos/generic/Hdf5file_cbase.H`.

5. In your call to `mpirun` omit the option for the machine file (`-mf` ...). A bash script that takes care of all this can be found at `/home/home1/tow/bash-script/cronos-mpi.sh`

NOTE: The time it takes to write an HDF file in this manner is slightly higher than in the case of multiple files for each processor. In case of rapid output it can be advantageous to use the serial output. However, the advantage of the parallel output is not to have to take care of merging the serial files into one file, which is time consuming also and necessary for data analysis. Also, the parallel output can be easily analysed already during runtime.

2.2 Working with git

The main `CronosCode` repository comes with three different branches corresponding to different development states of the code. The user can switch between them by using `git checkout <branch-name>`.
Starting from the bottom, the **master-alpha** branch contains the bleeding edge development versions of Cronos. This branch can be updated quite frequently depending on the developer's activity. It should not be used for production, since it can contain work in progress and is certainly prone to contain bugs. After finishing a development step an automated testing of the code will be performed. If the code passes all setups the commit receives a new version tag (e.g. `0.1.x`) and is merged to the **master-beta** branch. This will happen on the timescale of weeks, depending on the development activity. Hotfixes, depending on their severity, can be pushed directly to this branch. Cronos developers are therefore encouraged to use this branch, especially to get a more interactive testing of the code and some degree of feedback before

merging it to the **master** branch. This merging will happen on the timescale of months after performing again automated tests and altering the version tag, e.g. to `0.x.0`. The **master** branch is supposed to contain the stable version of Cronos and should be used for production purposes.

3 Setting up a simulation

3.1 Files to be modified by the User

All the files which have to be edited in order to set up and run a simulation are located in the `cronos/User` directory. Alternatively, it is possible to set up a separate directory (of arbitrary name) for each group of projects in analogy to `cronos/User`. For the remainder of this document, all files will be assumed to reside in this directory unless explicitly specified otherwise.

`constants.H`
: includes very basic parameters and the definitions of some global variables (see Section 3.3).

`machinefile.local`
: For runs using multiple machines, this file contains a list of the hosts and the number of CPUs to be used on that host (see Section 3.12.2). For single-machine runs (and multicore runs on clusters consisting of several cores on a single machine), this file is irrelevant.

`modules.C`
: relates simulation classes to their types (see Section 3.2).

`mod_`*mySim*`.H`
: Routines specifically needed for the simulation project *mySim*, realized as a C++ class of the same name. Initial and boundary conditions, source terms, etc. go here. Each project name that occurs in `modules.C` requires a corresponding `mod_*.H` file to be present.

`Makefile`
: to include external user directories, source files and additional compiler flags (see Section 3.13).

3.2 The Simulation hierarchy

Different simulation categories (representing, for instance, different physical "target objects"), are distinguished by their *type*, which is a unique (and usually small) integer. The relation between a simulation category and that category's type is specified in the file `modules.C`. Therefore, the first step in the setup of a new simulation project is to modify this file by adding

```
#include "mod_mySim.H"
```

at the very beginning and inserting the code segment

```
else if (type == simtype) {
  Problem = new mySim(gdata);
```

where `mySim` is the unique name of the project, and `simtype` is the project's unique type number. Lines referring to a category which is not to be used can be removed along with the corresponding mod file. The code segment for the minimum case of just one single category would thus be

```
if (type == simtype) {
  Problem = new mySim(gdata);
} else {
  ...
```

Each simulation of a given type (characterized by, e.g. different sets of parameters) requires a so-called *cat file* (see Section 3.10), in which these parameters are specified.

3.3 Entries of `constants.H`

Next, the file `constants.H` should be inspected to see if changes need to be made.

3.3.1 Keywords

`FLUID_TYPE`
: Switch between hydro or MHD, single- or multifluid simulations. Must be either `CRONOS_MULTIFLUID`, `CRONOS_HYDRO` or `CRONOS_MHD`. With the latter option hydro runs are also possible, but in that case the former is more efficient (no memory allocated for magnetic field and several routines are skipped).

`RK_STEPS`
: Order of accuracy to use for the Runge-Kutta time-stepping. Must be either `2` or `3`. 3rd order is generally more accurate, has the same memory requirements, but takes ∼50% longer to run.

`GEOM`
: Coordinate system to use. Currently supported choices are:
  `1` → Cartesian $[x, y, z]$
  `2` → cylindrical $[R, \varphi, Z]$
  `3` → spherical $[r, \vartheta, \varphi]$

`CT_TYPE`
: Variant of constrained transport to use. Recommended choices are: `CONSISTENT` when using the HLL Riemann solver, or `STONE` for HLLD.

`STONE_TYPE`
: Possible choices are `STONE_SIMPLE` or `STONE_CENTRE` (recommended). Only matters if `CT_TYPE` is set to `STONE`.

`DIM`
: Number of dimensions. Currently, the only admissible value is `3`.

ENERGETICS
: Set this to FULL if Equation (6) is to be integrated, or to ISOTHERMAL[2] if an adiabatic closure relation (see Section 3.4) is to be used. (This keyword has replaced the N_OMEGA variable known from previous versions.)

CRSWITCH_DUAL_ENERGY
: Set this to CRONOS_ON to enable entropy correction (see Section3.6), else use CRONOS_OFF.

NON_LINEAR_GRID
: Enable non-linear grid scaling (CRONOS_ON) or not (CRONOS_OFF). This is discussed in Section 3.11.

REAL
: Variable type to be used for floating point data. Admissible types are double and float, the former being the recommended choice. The latter may, however, be useful to decrease the code's memory consumption if necessary.

OMS_USER
: Use additional variable fields (TRUE) or not (FALSE). Has to be TRUE if at least one of the projects listed in modules.C uses additional variables.

USE_COROTATION
: Perform simulations in a corotating frame of reference (CRONOS_ON) or not (CRONOS_OFF). Requires entry omegaZ in the cat file. See Section 3.5 for details.

3.3.2 Flags

PHYSDISS
: Can be used to enable physical dissipation/ viscosity (currently undocumented code feature).

MHD, VEC_POT_BCS
: Reserved for future use, irrelevant as of now.

SOUND
: This entry is obsolete, but should be left untouched until it will be cleanly removed from the code in a forthcoming revision.

SAVEMEM
: For cases in which memory usage is an issue, this can be set to favor re-computation of some auxiliary variable fields over storing them for later use. Users interested in memory/ runtime opimisation are encouraged to experiment with this flag to quantify its performance impact for the given architecture. This flag only has an effect for FLUID_TYPE = CRONOS_MHD.

[2]Note that this term is somewhat misleading, since the fluid need not be isothermal, depending on what adiabatic exponent is used.

3.3.3 Variables

`const int B`
: Number of ghost cell layers to be considered in the computation.

`const int BOUT_FLT`
: Number of ghost cell layers to include in FLOAT output (see Section 4.2). Must be $\leq$ `B`.

`const int BOUT_DBL`
: Number of ghost cell layers to include in DOUBLE output (see Section 4.1). Must be $\leq$ `B`.

`const int N_OMINT_USER`
: Number of additional variable fields if such are taken into account (i.e. if `OMS_USER = TRUE` has been set). See Section 3.9 for details.

`const int N_P`
: Obsolete, and to be removed in a forthcoming revision. Until then, best left untouched.

`const int N_ADD`
: Probably also obsolete, no need to touch it.

All else should be left untouched if possible. If you must use additional global variables and/or keywords, these should be defined in this file (that is, in `constants.H`).

3.4 The energy integration

Depending on how the energy budget is to be computed, different settings are required:

- Isothermal ($p = \rho\ c_{\rm iso}^2$): Set `ENERGETICS` to `ISOTHERMAL` in `constants.H` and edit the cat file (see Section 3.10) to set the adiabatic index γ equal to unity and the isothermal sound speed $c_{\rm iso}$ equal to $\sqrt{T_0}$, where T_0 is the desired constant temperature.

- Adiabatic ($p \propto \rho^\gamma$): Same as above, but with $\gamma \neq 1$. Note that in this case the constant of proportionality is fixed via

$$\frac{p}{p_0} = \left(\frac{\rho}{\rho_0}\right)^\gamma \ \Rightarrow\ p = \frac{c_{\rm iso}^2}{(\rho_0)^{\gamma-1}}\ \rho^\gamma \tag{12}$$

  where the initial density ρ_0 is again specified in the cat file. This is only permissible if initially $\rho = \rho_0$ holds throughout the entire volume! For initial setups involving a spatially varying density, a full energy equation (see next item) is mandatory, even if no energy source terms are being used.

- To solve a full energy equation (6), set `ENERGETICS` to `FULL` and $\gamma \neq 1$. In this case, the $c_{\rm iso}$ and ρ_0 cat entries are not used (but may still be accessed via the `rho0` and `c2_iso` member variable/function).

3.5 Corotating frame of reference

It is often desirable to perform the simulations in a corotating frame of reference to simplify boundary conditions. Implementing the resulting fictitious (Coriolis-, centrifugal-, and Euler) forces as source terms can lead to instabilities and non-conservation of angular momentum. Instead, a conservative treatment was developed, which results in the following set of equations. These are equivalent to the original ideal MHD equations with fictitious forces as source terms, such that the new set

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{13}$$

$$\partial_t (\rho \mathbf{u}) + \nabla \cdot \left[\rho \mathbf{v}\mathbf{u} + (p + \|\mathbf{B}\|^2/2)\mathcal{I} - \mathbf{B}\mathbf{B}\right] = -\rho \boldsymbol{\Omega} \times \mathbf{u} \tag{14}$$

$$\partial_t e + \nabla \cdot \left[e\mathbf{v} + (p + \|\mathbf{B}\|^2/2)\mathbf{u} - (\mathbf{u} \cdot \mathbf{B})\mathbf{B}\right] = 0 \tag{15}$$

replaces Eqs. (4–6), and Ohm's law (10) is now

$$\mathbf{E} + \mathbf{v} \times \mathbf{B} = \mathbf{0} \ . \tag{16}$$

Here, $\mathbf{v}$ denotes the velocity in the corotating frame, which is related to the inertial frame velocity $\mathbf{u}$ via

$$\mathbf{v} = \mathbf{u} + \mathbf{r} \times \boldsymbol{\Omega} \ . \tag{17}$$

The implementation is such that the axis of rotation conincides with the z axis. To enable this feature, set `USE_COROTATION CRONOS_ON` in `constants.H`. The value for the rotation frequency Ω has to be set in the cat file using the `omegaZ` variable. In the mod file, the user deals with the corotating frame velocity $\mathbf{v}$ and has to set initial- and boundary conditions as well as additional source- and flux terms accordingly. The function `init_fields` has to be supplemented at the top by

```
#if (USE_COROTATION == CRONOS_ON)
  gdata.om[q_sx].rename("v_x_Corot");
  gdata.om[q_sy].rename("v_y_Corot");
  gdata.om[q_sz].rename("v_z_Corot");
#endif
```

3.6 Entropy correction

If the ratio of thermal to total energy changes sharply (e.g. at shock fronts), numerical inaccuracies may lead to negative values of pressure and/or temperature, at which times the code has to abort because the sound speed becomes undefined. The suggestion of Balsara & Spicer (1999) is to integrate an additional equation

$$\partial_t s + \nabla \cdot (s \ \mathbf{u}) = 0 \tag{18}$$

for the entropy density $s \equiv p/\rho^{\gamma-1}$ and to compute p from s rather than e whenever the former value appears to be more trustworthy for a given cell (and in particular when e_{therm} has become negative).
For runs which are prone to negative pressures, this entropy correction should be enabled (albeit at the cost of slightly increased numerical expenditure, and less accurate results). Otherwise do not use it.

3.7 Choice of Riemann solvers

Three types of Riemann solvers are available, and can be specified in the cat file:

1. HLL: most simple Riemann solver. Very stable but also more dissipative than the other options.
2. HLLC: recommended solver for HD. For isothermal runs this is equivalent to HLL
3. HLLD: recommended for MHD. Set `EXTRACT_PRESSURE` to `FALSE` in `constants.H` when using this solver. (For the others this does not matter.)

Unsuitable choices will be rejected, and a comment is issued.

3.8 Setting up the mod file

Next, the inital conditions, as well as possible source terms and special boundary conditions (if so desired), need to be specified, all of which occurs in the file `mod_`*mySim*`.H` (whose name should correspond to the one chosen in `modules.C`). Any additional, non-standard procedures can also be implemented here. (For the following, it is advisable to copy an existing mod file and make the needed changes there, rather than writing one from scratch.) In this file, the simulation (again dubbed `mySim` in this example) is set up as a C++ class, the basic skeleton of which looks like this:

```
#ifndef CRONOS_MYSIM_H
#define CRONOS_MYSIM_H = 1

#include "gridgen.H"

class mySim:public ProblemType {
public:
  mySim(Data &);
  void init_fields(Data &, int [], int []);
private:
  REAL my_param;
};

mySim::mySim(Data &gdata) {
  name = " My new simulation ";
  this->my_param = value((char*)"My Parameter");
}

void mySim::init_fields(Data &gdata, int ibeg[3], int iend[3])
{
  for (int k = ibeg[2]; k <= iend[2]; ++k)
    for (int j = ibeg[1]; j <= iend[1]; ++j)
```

```
      for (int i = ibeg[0]; i <= iend[0]; ++i)
        gdata.om[0](i,j,k) = density_function(i,j,k);
        gdata.om[2](i,j,k) =     v_y_function(i,j,k);
}

#endif
```

Three blocks of code can be identified:

1. The *class definition* (starting with the `class` keyword) lists all functions and variables of the problem class.
2. The *constructor* has the same name as the class and lacks a return type. It gets called only once at the beginning. Class-wide variables should be defined here. For clarity of output, a name should (but need not) be set by (re-)defining the `name` string variable.
3. The initialization routine called `init_fields`, see Section 3.8.2.

(Note the leading `#ifndef`/`#define` directives, which exist to prevent the compiler from including the same section multiple times. The keyword can in principle be chosen freely but must be unique, which is easily achieved by using some variant of the project name.)

3.8.1 Read parameters

The class constructor is a good location to read parameters from the cat file via the `value` method. This method's syntax is
`REAL param = value((char*)"`*parameter_name*`");`
It scans the text file *datadir*`/`*pname*`.cat` for the line

parameter_name `=` *value*

and returns the floating point value of *value*. If the search comes up empty, an error message is issued. If more than one whitespace-separated entry follow the equal sign, only the first entry is returned, and the rest ignored[3]. Having been declared in the `private` section, its value can now be accessed from all methods of the class (but not from outside). For more info on cat files see Section 3.10.

3.8.2 Initial conditions

Any `ProblemType` class *must* provide the method `::init_fields`, which defines the initial conditions. The `gdata` object contains the properties of both the grid and the values of the physical fields on the grid. These data can be accessed according to Table 3. `ibeg[]` and `ibeg[]` hold the first and last indices of the part of the grid which the current process works on (including the ghost cell layers). For the single-CPU case, they are thus equal to $[-B, -B, -B]$ and $[N_x - 1 + B, N_y - 1 + B, N_z - 1 + B]$, where B is the `B` value from `constants.H`. Fields which lack an explicit initialization are initialized to zero.
`om[`q`](`i`,`j`,`k`)` denotes the value of of the physical field q at the rid cell (x_i, y_j, z_k), where q is chosen from Table 2. It should be noted that `gdata.om[q_Eges]` denotes just the thermal

[3]This suggests that if you tentatively modify a parameter, you can leave the original value standing in the same line, and later easily revert to it without having to remember it in the meantime.

q_rho	mass density	q_Eges	energy density
q_sx, q_sy, q_sz	velocity components	q_Bx, q_By, q_Bz	magnetic field/ vector potential components

Table 2: Admissible q values to refer to (standard) physical fields.

part $e_{\rm th} \equiv p/(\gamma - 1)$ of the total energy density. The same is true for the boundary conditions and source terms (although this distinction is not of great importance for these two cases). By default, `gdata.om[q_B{x,y,z}]` denote the components of the initial magnetic field. It can, however, be more appropriate to use the vector potential instead, the advantage being that the code will compute the corresponding $\mathbf{B}$ from $\nabla \times \mathbf{A}$, such that $\nabla \cdot \mathbf{B} = 0$ is automatically satisfied. Otherwise, the user himself has to make absolutely sure that this is the case[4]. To use $\mathbf{A}$ rather than $\mathbf{B}$, insert the lines

```
gdata.om[q_Bx].rename("A_x");
gdata.om[q_By].rename("A_y");
gdata.om[q_Bz].rename("A_z");
```

before assigning the components of $\mathbf{A}$ to `gdata.om[q_B{x,y,z}]`, and make sure that the shift of indices (cf. Table 1) is taken into account.

3.8.3 Boundary conditions

Arbitrary boundary conditions other than periodic, extrapolation, outflow, or reflecting (which should be set directly in the cat file, see Section 3.10) can be achieved on any of the six sides of the computational volume by implementing the method

```
void mySim::bc_User(Data &gdata, NumMatrix<REAL,3> &omb,
                    int dir, int top, int q, int rim)
```

which can be used to set any boundary condition for the physical field `omb` (= `gdata.om[`q`]`) of number q (cf. Table 2 in coordinate direction `dir` (`0,1,2` for x,y,z) and side (cube face) `top` (`0` = min, `1` = max). As before, `gdata.om[q_Eges]` denotes thermal (not total) energy density. If nothing is specified for a given q and boundary, an error message will be issued.
To make a quantity simply retain its initial values at that boundary (Dirichlet boundary condition), this value has to be specified explicitly.[5]

3.8.4 Source terms

Physical source terms such as gravity and heating functions, which are to be included on the right hand sides of the equations (4) to (6), can be specified by implementing

[4]It is vital that this constraint be satisfied numerically for the discretized field, not just analytically for the formulas used in the assignment.

[5]In the future, it will be possible to implement a Dirichlet BC using another type in the cat file; this is scheduled to be realized in a forthcoming edition.

```
void mySim::src_User(Data &gdata,
                     NumMatrix<REAL,3> nom     [N_OMINT     ],
                     NumMatrix<REAL,3> nom_user[N_OMINT_USER])
```

Source terms for the standard equations for variable q at cell $[i, j, k]$ are assigned using

```
nom[q](i,j,k) -= some_function(i,j,k);
```

(Note the minus sign, and be sure *not* to use `=` instead of `-=`.) In the case of additional variables, use

```
nom_user[q](i,j,k) -= some_other_function(i,j,k);
```

instead.

3.8.5 Coordinate singularities

In many cases, the desired computational domain may contain one or more coordinate singularities, which are defined as regions where at least one scale factor tends to zero (implying that several vertices of a cell coincide, leading to wedge-shaped cell geometries). For cylindrical coordinates, this can only occur at the axis $R = 0$, while for spherical coordinates, the three possible cases are $\vartheta = 0$ ("north pole"), $\vartheta = \pi$ ("south pole"), and the origin at $r = 0$. If these locations are to be included, the grid should be specified such that the singularity exactly coincides with a cell face. Furthermore, it is necessary to include the keyword `Include_CoordinateAxis` (previously `IncludeAxis`) into the cat file to activate the singularity treatment. The effect of this is two-fold:

`gdata.`name	meaning
`REAL om[`q`](`i`,`j`,`k`)`	value of variable $q \in \{$`q_rho`$, \ldots\}$ at (i, j, k)
`REAL xb[`d`]`	$d \in [0, 1, 2] \to [x_0, y_0, z_0]$
`REAL dx[`d`]`	$d \in [0, 1, 2] \to [\Delta x, \Delta y, \Delta z]$
`int mx[`d`]`	$d \in [0, 1, 2] \to [N_x - 1, N_y - 1, N_z - 1]$
`REAL h`c`(`i`,`j`,`k`)`	$c \in [1, 2, 3] \to$ scale factor h_c for direction c
`REAL time`	current simulation time
`REAL dt`	current timestep
`REAL t_end`	time after which simulation run will halt
`getCen_x(`i`)`	returns cell center x_i as a function of index i, and analogously for `getCen_y` and `getCen_z`.
`get_x(`i`,`s`)`	corresponds to `getCen_x(`$i + s/2$`)`, $s \in \{0, \pm 1\}$ and similarly for `get_y(`j`,`s`)` and `get_z(`k`,`s`)`
`get_pos(`d`,`i`,`s`)`	$d \in [0, 1, 2] \to$ [`get_x(`i`,`s`)`, …, `get_z(`i`,`s`)`]
`get_x_global(`i`,`s`)`	grid-wide version of `get_x(`i`,`s`)`
`get_pos_global(`d`,`i`,`s`)`	grid-wide version of `get_pos(`d`,`i`,`s`)`
`get_CellVolume(`i`,`j`,`k`)`	geometric volume of cell

Table 3: Some important members of `gdata`. All of $\{i, j, k, d, s\}$ are integers. All `get_*` functions return `REAL`. For routines without "`global`" in their names, positional indices (i, j, k) refer to the subgrid that the given processor works on, i.e., i runs from 0 to N_x/`nprocx` and similarly for j and k. For a more complete (but largely uncommented) list consult the respective implementations in `grid.C`.

coord. system	boundary	cells to copy	minus sign for
cylindrical	$R = 0$	$[R, \varphi, z] \leftarrow [-R, \varphi \pm \pi, z]$	R, φ components
spherical	$\vartheta \in \{0, \pi\}$	$[r, \vartheta, \varphi] \leftarrow [r, \vartheta, \pi \pm \varphi]$	r, φ components
spherical	$r = 0$	$[r, \vartheta, \varphi] \leftarrow [r, \pi - \vartheta, \pi \pm \varphi]$	r, ϑ, φ components

Table 4: Boundary cell prescription at coordinate singularities. The $\pm$ signs are to be chosen such that the resulting cell exists and is located within the domain.

1. Since boundary cells at $R_{\min}$ (cylindrical) or $\vartheta_{\min/\max}$ and $r_{\min}$ (spherical) are physically equivalent to internal cells on the other side of the singularity, the cell contents is copied from there, possibly amended with a minus sign that follows from symmetry considerations (see Table 4).

2. Runs involving a magnetic field pose the additional difficulty that the inward-pointing B component (B_R for $R = 0$, B_ϑ for $\vartheta \in \{0, \pi\}$, and B_r for $r = 0$) is localized exactly at the singularity, at which the field integration diverges. Therefore, this component has to be replaced by a finite value computed from the magnetic field at the surrounding cells, while the electric field receives a corresponding treatment.

This will keep the axis fields finite, but has not yet been extended to the spherical origin case. Therefore, MHD simulations involving the spherical origin are not currently possible at all, while for HD runs, the user has to implement the proper boundary conditions according to Table. 4 himself. For boundaries at which a singularity is present, the specified boundary condition (unless equal to `4`, see Section 3.10) will be overridden by the code, and is therefore irrelevant.

3.9 Extended equations

3.9.1 Add new equations

The set of variable fields $\{\rho, \mathbf{u}, \mathbf{B}, e\}$ can be extended by a arbitrary number of additional fields if so desired. To do so, you need to

1. edit `constants.H` by setting `OMS_USER = TRUE` and `N_OMINT_USER` equal to the number of additional variable fields,

2. supply formulas for the physical fluxes (the divergence part of the equations, without the divergence operator itself), and

3. implement initial conditions, boundary conditions, and source terms if applicable (see above).

For example, to add a passively advected scalar field ψ (i.e. a field which obeys

$$\partial\psi + \nabla \cdot (\psi \mathbf{u}) = 0$$

but does not exert pressure or any other back-reaction upon the usual fluid), set `N_OMINT_USER = 1` and implement

```
void mySim::get_PhysFluxUser(Data &gdata,
                             cronos::vector<REAL> &iPos,
                             phys_fields_1D &pf,
                             phys_fields_1D &pfUser,
                             int dir, REAL shift) {

  pfUser.flux[0].clear();
  for (int i = -1; i <= gdata.mx[dir]+1; ++i)
    pfUser.flux[0](i) = pfUser.uPri[0](i)*pf.uPri[1+dir](i);
}
```

within the mod file. This provides the physical fluxes (the term inside the divergence operator) for direction `dir` (0,1,2 for x, y, z) and user variable number `0` at grid cell `i` as a function of that variable (`pfUser.uPri`) and the standard variable (`pf.uPri`) number $q = 1 +$ `dir`. In other words, the above is equivalent to

$$\text{flux}(\psi)_i = \begin{cases} \psi\ (u_x)_i & : \ \texttt{dir} = 0 \\ \psi\ (u_y)_i & : \ \texttt{dir} = 1 \\ \psi\ (u_z)_i & : \ \texttt{dir} = 2 \end{cases} \tag{19}$$

3.9.2 Boundary conditions for additional variables

In this example, boundary conditions for ψ could be chosen in analogy to those for ρ. One small complication may arise from the fact that the argument `q` is used to refer to both `om[q]` and `om_user[q]`, while different implementations will be generally be desired for both cases. To discriminate the two cases,

```
if (q == 0) { ... <code for om[0] / om_user[0]> }
```

could be replaced by something like

```
if (omb.getName() == "rho")
  { ... <code for om[0]>  }
else if (omb.getName() == "om_user0")
  { ... <code for om_user[0]> }
```

Note that the default name `om_user`$\langle q \rangle$ for the additional variable q can be changed to a more descriptive string using
`gdata.om[`q`].rename("new name")`
during the initialization. Applying this method to also customize the names for the standard fields is possible but strongly advised against, except when switching from **B** to **A** initialization (see Section 3.8.2).

3.9.3 Source terms for additional variables

A source term is not needed in this example, but could for instance be useful if the tracer particles are to be inserted continuously during the simulation, rather than being present right from the start: If $Q_\mathrm{s}(t)$ is the output strength of a ball-shaped source of radius r_s (per cell

volume) that moves through the computational volume along an $[x_s(t), 0, z_s(t)]$ trajectory, one could use

```
REAL t = gdata.time;
for (int k = 0; k <= gdata.mx[2]; ++k) {
  REAL z = get_z(k);
  for (int j = 0; j <= gdata.mx[1]; ++j) {
    REAL y = get_y(j);
    for (int i = 0; i <= gdata.mx[0]; ++i) {
      REAL x = get_x(i);
      if (sqr(x-x_s(t))+sqr(y)+sqr(z-z_s(t)) <= sqr(r_s))
        nom_user[0](i,j,k) -= Q_s(t);
    }
  }
}
```

to implement that into `mySim::src_User`. (Note that for non-Cartesian coordinates, $Q(t)$ would need to be normalized to the cell volume, which can be computed using the relevant scale factors $[h_1, h_2, h_3]$, or, more, conveniently, using the `get_CellVolume` member function of `gdata`, see Table 3.
`REAL sqr (REAL arg)` is a member function from the `util` class, which returns `arg*arg`, i.e. the square of the argument.

3.9.4 Extend existing equations

Additional terms of divergence form for the standard equations should *not* be implemented as source terms but as flux terms to take advantage of the conservative properties of the scheme. This is achieved in close analogy to the case for new variable fields, except that

1. the procedure is now called `void mySim::get_Physflux(...)` (but uses the same list of arguments),
2. `pf.flux[`q`](i)` gets modified rather than `pfUser.flux[`q`](i)`,
3. the `clear()` command is omitted, and the new terms are added (via `+=`) rather than assigned (`=`) in order to preserve the terms already introduced by the standard fluxes.

3.10 The cat file

At init, the parameters to be used in a given simulation are read in from the cat file *pname*`.cat` residing in directory *datadir*. This is a text file with lines of pattern

`keyword =` *value*`.`

Table 5 lists the keywords which have to be present.
Some comments:

- For parallel runs, the product of all `nproc`s has to be equal to the total number of processors used, i.e. equal to the product of the `nproc{x,y,z}` cat entries. Also, each `N{x,y,z}` (or 1+`m{x,y,z}`) has to be an integer multiple of `nproc{x,y,z}`.

nproc{x,y,z}	number of processors in x,y,z direction
reorder	allow code to change space partition ($\neq$ 0)
	or enforce usage of nproc{x,y,z} (0)
cfl_threshold	max. value for global CFL number
cfl_min	min. value for global CFL number
runtime	seconds of real time after which run terminates
t_end	final simulation time
dt	initial timestep
dt_{bin,float,ascii, info,mov}	time interval for various outputs
num_{bin,float,ascii, info}	write output every n (integer) timesteps
N{x,y,z}	spatial resolution $N_{x,y,z}$
m{x,y,z}	spatial resolution $N_{x,y,z} - 1$ (see comments in Section 3.10)
{x,y,z}b	lower bounds of physical volume ($\{x, y, z\}_0$)
{x,y,z}e	upper bounds of physical volume ($\{x, y, z\}_N$)
ScaleGrid	0 disables scale_{x,y,z}, $\neq$ 0 will enable it.
scale_{x,y,z}	physical bounds are in units of π (pi, Pi),
	exp(1) (e), or 1 (1)
grid_{x,y,z}	type of grid mapping to use (see Section 3.11)
type	type number of problem (cf. modules.C)
restart	continue existing simulation (1) or start anew (0)
restart_step	timestep from which to restart (if applicable)
Limiter	Use minmod (1) or van Leer limiter (2)
RiemannSolver	Riemann solver to use. Possible choices are
	hll, hllc, or hlld. (See Section 3.7.)
bc_{x,y,z}_bot	boundary conditions at $\{x, y, z\} = \{x_0, y_0, z_0\}$ (see above)
bc_{x,y,z}_top	boundary conditions at $\{x, y, z\} = \{x_N, y_N, z_N\}$
mag	magnetic field present (1) or not (0)
Adiabatic_exponent	γ value to be used
Initial_density	REAL rho0 member of Problem class (see Section 3.4)
Isothermal_Soundspeed	square root of REAL cs2 member variable
rhomin	lowest admissible density during run
pmin	lowest admissible pressure during run
vmax	highest admissible velocity during run

Table 5: Mandatory keywords in the cat file.

- To prevent a run from finishing prematurely, `runtime` can be given extremely high values.
- `dt` will only be used for the first timestep, while the stepsize will then be CFL-controlled (except to match the intervals for data output).
- Possible values for `bc_{x,y,z}_{top,bot}` are:

  0 : Periodic
  1 : Periodic (with data at mx to be overwritten)
  2 : Extrapolation
  3 : Outflow
  4 : User-defined local bcs (from `bc_User` routine)
  5 : Reflecting boundaries

- The recommended way to specify the grid size and extension is by providing the number of grid cells in each direction via `N{x,y,z}`. If all three of them are set, `{x,y,z}b` and `{x,y,z}e` designate the physical coordinates of the first and last cell's outer face in the respective direction, and the contents of `m{x,y,z}` is ignored. (For instance, to cover the complete `y` interval $[0, 2\pi]$ with ten cells, use `yb=0`, `ye=2`, `scale_y=pi`, and `Ny=10`.) Alternatively, the outer cells can be clipped, as described in Section 1.4, by omitting (at least one of) the `N{x,y,z}` and giving the highest cell-centered index (i.e. $N_{x,y,z} - 1$) in `m{x,y,z}`. In this case, `{x,y,z}b` and `{x,y,z}e` are the centers of the outermost cells (prior to clipping). To achieve the same coverage of $[0, 2\pi]$, one would use `yb=0.1`, `ye=1.9`, `scale_y=pi`, and `my=9`. If both `N{x,y,z}` and `m{x,y,z}` are given, the former takes precedence.

3.11 Non-linearly scaled grids

Cronos also allows for a non-linear grid to be used in the simulations. So far this grid still has to be an orthogonal one. This allows the coordinates values along coordinate axes to be stored in a 1D array for each direction after the user has specified the transformation to be applied for a given coordinate direction: While for linear grids the relation between cell index and physical location for a given coordinate direction $u \in \{x, y, z\}$ is given by Eqs. (1 – 3) with $\Delta u \equiv (u_\mathrm{e} - u_\mathrm{b})/N_u$, the user may in addition specify a strictly monotonous function $f_u := [0, 1] \mapsto [0, 1]$ that defines a modified mapping

$$u_i = u_\mathrm{b} + (u_\mathrm{e} - u_\mathrm{b})\ f_u\left(\frac{i + 1/2}{N_u}\right) \tag{20}$$

of cell index i to location u_i while satisfying $u_{-1/2} = u_\mathrm{b}$ and $u_{N_u+1/2} = u_\mathrm{e}$, such that the grid extent is left unchanged. Here, u_e and u_b denote the domain boundaries as given in the cat file (see Section 3.10). Note that the identity mapping $f_\mathrm{lin}(\zeta) = \zeta$ recovers the linear case. It is important that f be strictly monotonous also in the boundary cells beyond $\zeta \in [0, 1]$, since these would otherwise get mapped into $[0, 1]$, i. e. the actual volume.

3.11.1 Employing a pre-defined mapping

The use of non-linearly scaled orthogonal grids is achieved by

1. setting `NON_LINEAR_GRID` to `CRONOS_ON` in `constants.H` and
2. setting `grid_`c `=` m in the cat file, where m is an integer denoting the mapping type for coordinate direction $c \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, e. g. `grid_y = 2`.

Currently, available choices for m are

0: linear mapping $f_{\rm lin}(\zeta) = \zeta$ (which only makes sense for debugging purposes),

1: exponential

$$f_{\rm exp}(\zeta) = \frac{(u_{\rm e}/u_{\rm b})^{\zeta} - 1}{(u_{\rm e}/u_{\rm b}) - 1} \tag{21}$$

which can be shown to result in

$$u_i = u_{\rm b}\ (u_{\rm e}/u_{\rm b})^{(i/N_u)}\ , \tag{22}$$

2: logarithmic $f_{\log}(\zeta) = u_{\rm b}(u_{\rm e}/u_{\rm b})^{\zeta}$

4: user-defined (see Section 3.11.2)

5: percent-wise increase of cell size. This requires cat entries `cellChange_`c

6: slightly different version of (5) This requires cat entries `refSize_`c

3.11.2 Implementation of a user-defined mapping

Internally, each grid mapping is handled by a grid function class which has to be derived from the abstract base class `Interface_GridFunction`. This is done via the method `set_UserGridFunction` in `modules.C`, which gets called whenever the cat entry `grid_`c is set to `4` for a given direction u. In this method a new mapping is established by instantiating a grid class via

```
myNonlinGrid[dir] = new DerivedUserClass();
```

where the class `DerivedUserClass` has to be supplied by the user, e.g., in one of the usual module files. Here `myNonlinGrid` is the pointer to the derived class used by the internal routines generating the grid. Via the class constructor the user should obtain all variables needed to define the grid. The routine to provide the grid mapping has to be of the form

```
virtual REAL get_gridFunc(REAL);
```

As an example, suppose one wants the cell size to increase linearly by a factor of m over the entire range $[x_{\rm e}, x_{\rm b}]$. The cell size at x is proportional to $f'(\zeta)$, and the most general second-order polynomial satisfying $f(0) = 0$ and $f(1) = 1$ is $f(\zeta) = b\zeta + (1-b)\zeta^2$. Hence m is fixed from $m = f'(1)/f'(0) = (2-b)/b$, resulting in $f(\zeta) = \zeta[(m-1)\zeta + 2]/(m+1)$. This is implemented by first adding

```
   myNonlinGrid[dir] = new GridFunction_Incease();
```

under `set_UserGridFunction` in `modules.C` and then implementing the class and its two methods as

```
class GridFunction_Increase: public Interface_GridFunction {
public:
   GridFunction_Increase();                 // constructor
   virtual REAL get_gridFunc (REAL zeta);  // return f_inc
   REAL _ratio;      // 'm' parameter
};

GridFunction_Increase::GridFunction_Increase() {
   _ratio = value((char*)"cellsize_ratio");
}

REAL GridFunction_Increase::get_gridFunc(REAL zeta) {
   return zeta*((_ratio-1.)*zeta+2.)/(_ratio+1.);
}
```

in the module file (or possibly in a separate file like `mygrid.H` and then `#include`-ing it therein. The desired ratio m of cell sizes is read in from the cat file only once, and the second function then uses it to build the grid.

Note that parameters `dir,beg,end,len` are available in `Interface_Gridfunction`, and may thus be passed to the grid class for more specialized grid layouts.

3.12 How to compile and run

After changes have been made to any of the source files listed in Section 3.1, a re-compile is needed for the changes to take effect:

- When `constants.H` has been altered, a complete rebuild is needed. Type `make clean && make` to do that.
- After modifying `modules.C` or *any* `mod_*.H` file, do `touch modules.C && make` .

Note: `make` will build both the parallel and non-parallel version of the executable. If you do not need the parallel version, using `make proj` instead will suffice (and be faster). To build the parallel version only, use `make proj_MPI`.
Changing a cat file does not require a re-build; you can just run again. The same applies to the case of a modified `machinefile.local`. Remember that if your machine has more than one CPU, you can speed up compilation by using the `-j` option to `make`. For instance, to employ two CPUs for compilation, use `make -j2 proj` instead of `make proj`.
After the compiler has exited with no errors, the compiled executables `proj` (for single-CPU runs) and/or `make proj_MPI` are found in a directory whose name is machine-dependent, e.g. `Linux-i386` or `Linux-amd64`.

3.12.1 Starting a single-CPU run

The syntax to run a simulation on one CPU at your local machine (assuming a 64bit architecture) would be
`Linux-amd64/proj` *datadir pname* .
This will read the simulation parameters from the cat file *datadir*`/`*pname*`.cat` and write all binary output into the *datadir* directory. If *datadir* and/or *pname* are omitted, they will be read from the environment variables `poub` (*datadir*) and `pname` (*pname*), which have to have been defined previously in this case using

`export poub=`*datadir* and/or `export pname=`*pname* .

3.12.2 Starting a parallel run

This requires the call of the executable to be preceded by
`mpirun -np` n `-machinefile machinefile.local` where n is the number of processors to be used. Note that the version of mpirun to evoke must match the version of the MPI compiler. `machinefile.local` is a text file containing a list of the available machines, where each line consists of a hostname, optionally followed by a `:` and the number of cores to be used on that machine. If just one hostname is involved (such as for SKYNET/FUJI), this keyword is not needed. Usually the full path to the data directory must be given. Make sure you can login to each host (via `ssh` *hostname*) without having to enter your password. For time-consuming computations, it has proven advantageous to be able to close the window in mid-run (and to keep the simulation running even if the terminal gets closed accidentally). To do this (and to alternatively pipe the text output into a file *protocol.txt*), use
`nohup` [above command line] `>` *protocol*`.txt &`
after which you can use `tail -f` *protocol*`.txt` to continuously view the output that is currently being generated.

3.13 Makefile

The application's `Makefile` allows the user to specify additional compilation flags and/or to include external directories or source code by setting the following variables at the beginning of the `Makefile`.

`APPLICATION_OFLAGS`
: specifies additional compilation flags. Default: `-DNDEBUG` (disables assertions)

`APPLICATION_INCLUDE`
: Additional directories that should be included in the compilation (no `-I` prefix required). Default: *empty*

`APPLICATION_CXX`
: Additional source files that should be compiled. Default: *empty*

4 Output

All output gets written into the *datadir* directory, i.e. the same directory in which the cat file resides. CRONOS will create the following four types of output files at the respective time intervals specified in the cat file:

4.1 HDF double

Name pattern: *pname*_double/*pname*_dbl_step⟨n⟩.h5
where ⟨n⟩ is the integer running number of the timestep at which the output occurred. HDF double files contain the complete data for all variables on the entire grid with double precision. Purpose: provide data to restart a previously aborted or finished run at some intermediate time.
If more than one processor is used and the CRONOS_OUTPUT_COMPATIBILTY flag is enabled, each task will either generate its own output file, all of which then have their filenames extended by _coordc_x-c_y-c_z, where the integer $c_{x,y,z} \in \{0,$ nproc{x, y, z}$\}$ designate the position of the task's sub-grid.

4.2 HDF float

Name pattern: *pname*_float/*pname*_flt_step⟨n⟩.h5
Same contents as HDF double files, except using single precision.
Purpose: provide data for scientific analysis and visualisation. By converting the double precision data to single precision the memory demand is nearly cutting in half, while still maintaining sufficiently high accuracy for data analysis. For multi-processor runs, the same naming convention as for HDF double files applies. Separate files from each process may be merged into one single file (per timestep) using the deprecated *Converter* tool (see Sect. 5.1) if so desired.

4.3 HDF grid

Name pattern: *pname*_grid/*pname*_grid.h5
HDF file containing the Cartesian location of every grid point (cell center), along with the transformation matrix with which the curvilinear coordinates can be reconstructed.
Purpose: Needed for data processing using e.g. *Paraview* (see Sect. 5.3). This type of output occurs only once at the very beginning of a run, and has to be explicitly requested by providing the keyword WriteGrid (without arguments) in the cat file. Even for MPI runs, only one single file containing the entire grid will be generated.

The directory *pname*_grid also holds files describing the general grid setup for all subprocesses of a simulation. For a single-core run this information is stored in *pname*_grid.info. For MPI runs the naming scheme is *pname*_grid.info*Rank*, where *Rank* is the number of the individual sub-process.

4.4 HDF mov

Name pattern: *pname*_mov/*pname*_mov_*qname*_*plane*.h5
where *qname* enumerates all integrated quantities (e.g. rho, v_x, ..., one separate file for each

of them), while *plane* denotes the coordinate plane, which is one of `xy`, `xz`, or `yz`. This results in a total of $3 \times [8(7) + \texttt{N\_OMINT\_USER}]$ mov files for the entire simulation. For each variable, its HDF mov files combine plane slices for all output times into a single file, i.e. the spatial coordinate not mentioned in the filename is replaced by a time axis.
Purpose: useful to analyse the temporal evolution of a quantity, provided the planes of interest are known beforehand. Due to the much smaller size (2D slices instead of full 3D fields) movie files can be written more often than HDF double or HDF float files. The standard setup writes cuts through the middle of the numerical domain. A direct access to the position of this cut is planned, but not implemented yet. Currently the only option is to change the routine `InitMovie` in the file `generic/gridgen.C`.

4.5 ASCII

Name pattern: *pname*`_efluct`. Simple text file which contains a table of some statistical values for each desired time step. Can be read with any text editor, and processed using e.g. `gnuplot`.

5 Further data processing

Two more tools are available in the `src/cronos/DataReader` directory, and can be compiled and made available by `cd`'ing into that directory and executing `make` .

5.1 Merging HDF files from MPI runs

HDF float output files from MPI runs can be joined into one large file for the whole grid using the *Converter* tool. After compilation, one then needs to set the environment variables `pname` and `poub` appropriately (relative to the directory from which the converter will be called), and envoke `DataReader/`*machinetype*`/Converter` . Then type `1` for hdf5 input, and select what timestep to convert.

5.2 CronosPlot (IDL)

(check in `src/Visualization` ...)

5.3 Paraview

Paraview is a generic visualization package for large three-dimensional datasets, and is freely available for various computer platforms at `www.paraview.org`. In oder to load and view HDF float files written by Cronos for a given project, an additional XML text file must be created for that project to include additional information mandatory for the correct interpretation of the heavy HDF data. This is done via the XdmfGen tool, which is also created with a simple `make` in that directory. Typing *machinetype*`/XdmfGen` (without arguments) prints information about the usage, which is just

```
XdmfGen <poub> <pname> <itime_1> [<itime_2> [...] ]
```

where the *itime*s are numbers from the filenames of timesteps to include. Alternatively, the script `batch_XdmfGen.sh` can be used to refer to the complete set of all available timesteps.

As a second requirement, a *pname*_grid.h5 file must be present in the *pname*_grid directory (see Sect. 4.3). After successful completion, the file *poub*/*pname*.xmf can be read directly into Paraview.

6 Troubleshooting

List of error messages and possible causes.

6.1 When compiling

- Message: `fatal error:` *something*`.h: File not found`
  Execute `locate` *something*`.h` Execute `locate hdf5.h` and add (one of) the found paths as `-I`*path* behind the `X_INC` line of file `CronosNumLib/makeinclude/include.mak`, e.g. `-I/usr/include/hdf5/openmpi` in the case of `hdf5.h` on the TP4 cluster.
  Alternatively, execute `export CPLUS_INCLUDE_PATH=`*path*, e.g.
  `export CPLUS_INCLUDE_PATH=/usr/include/hdf5/openmpi` if this path was returned.
  If compiling succeeds, add the `export` line to your `.bashrc` file.

- Message: `/usr/bin/ld: cannot find -lhdf5`
  The HDF5 library is not found by the linker. Do `locate libhdf5.a` and take note of one of the *path*s in the output. Edit the file `makeinclude/include.mak` by inserting `-L`*path* at the end of the line defining `X_LIB`. Repeat installation steps.

- Message
  `../modules.C:22: error: cannot convert 'int*' to 'ProblemType*' in assignment`
  Possible cause: You did not `#include` your new mod file into the header of `modules.C` (see Sect. 3.2).

6.2 At runtime

- `Illegal instruction`
  Possible cause: Might be related to a known bug in the grid output routine. Disable the `WriteGrid` keyword in the cat file, then try again.

- `proj: <home>/include/matrix_3d_inc.H:4: T&`
  `Matrix<T,rank>::operator()(int, int, int) [with T = double,`
  `int rank = 3]: rank> Assertion '(i >= lo[0]) && (i <= hi[0])'`
  `failed.`

  Possible cause: Trying to access a matrix element outside the matrix' bounds. Check indices, remember different extension of **A** and **B** fields. Also comes up when trying to access `om[qB{x,y,z}]` with `FLUID_TYPE` set to `CRONOS_HYDRO`.

- Message: `env:`*datadir*`: Permission denied`
  Possible cause: If the message includes
  `Unrecognized argument` *machinedir*`/proj_MPI ignored.`

`Program binary is:` *datadir*
then your `proj_MPI` executable might be missing. Recompile using `make proj_MPI`.

- Message: `Unable to open a GM port`
  Possible cause: You are already running the maximum number of clients on a machine, possibly including leftover orphaned processes after a non-clean exit. Kill all unwanted processes and retry.

- Message: `Unrecognized argument ... ignored.`
  Possible cause: Wrong ordering of arguments. Use
  `<mpirun> -np` *n* `-machinefile` machinefile *machinename*`/proj_MPI` *poub pname* .

- An error message similar to
  `Linux-amd64/proj_MPI: error while loading shared libraries: libhdf5.so.6: cannot open shared object file: No such file or directory` is issued.
  Possible fix: Find path of library via `locate libhdf5.so.6`, then issue
  `declare -x LD_LIBRARY_PATH=`*path-to-library* .

6.3 git-related

- You messed with a file *myfile*, and want to undo your changes by reverting to the repository version of that file. This is achieved by typing `git reset --hard origin/master` in that directory.

- A push is rejected because the repository file has changed since you last pulled it. To discard(!) your changes and revert to the repository version, first fetch all changes using `git fetch --all`, then reset the master using `git reset --hard origin/master`, and finally `pull` again.

Bibliography

Balsara & Spicer, *J. Comp. Phys.* **148**, 133 (1999)

Kissmann et al., *Astrophys. J. Suppl.* **236/2**, article id. 53, 26 (2018)