Ramses

Version

Table of Contents

Ramses Documentation

The *Ramses* code is intended to be a versatile platform to develop applications using Adaptive Mesh Refinement (AMR) for computational astrophysics. The current implementation allows solving the classical and relativistic Euler equations in presence of self-gravity, magnetic field and radiation field. The *ramses* code can be used on massively parallel architectures, if properly linked to the MPI library. It can also be used on single processor machines without MPI. Output data are generated using Fortran unformatted files. A suite of post-processing routines is delivered within the present release, allowing the user to perform a simple analysis of the generated output files. A pdf version of this documentation can be found here.

New users of Ramses are invited to follow the ramses's tutorials, that cover the basics to setup a simulation and help with getting started with some common applications.

Table of Contents

Getting started

In this chapter, we will explain step by step how to get the RAMSES package and install it, then how to perform a simple test to check the installation.

Obtaining the package

The package can be downloaded from the GitHub repository using git:

```
$ git clone https://github.com/ramses-organisation/ramses
```

This will create a new repository called `ramses/` In this directory, you will see:

```
$ ls -F
README          bin/            mhd/            patch/          rt/
amr/            doc/            namelist/       pm/             utils/
aton/           hydro/          pario/          poisson/
```

Each directory contains a set of files with a given common purpose. For example, `amr/` contains all Fortran 90 routines dealing with the AMR grid management and MPI communications, while `hydro/` obviously contains all Fortran 90 routines dealing with hydrodynamics. The first directory you are interested in is the `bin/` directory, in which the code will be compiled.

Compiling the code

You need to go first to the `bin/` directory:

```
$ cd ramses/bin
$ ls -F
Makefile        Makefile.rt
```

We will use the first `Makefile` to compile the code. The first thing to do is to edit the `Makefile` and modify the two variables `F90` and `FFLAGS`. Several examples corresponding to different Fortran compilers are given. The default values are:

```
F90 = gfortran -O3 -frecord-marker=4 -fbacktrace -ffree-line-length-
none
FFLAGS = -x f95-cpp-input -DWITHOUTMPI $(DEFINES)
```

The first variable is obviously the command used to invoke the Fortran compiler. In this case, this is the GNU Fortran compiler. The second variable contains Fortran compilation flags and preprocessor directives. The first directive, `-DWITHOUTMPI`, switches off all MPI routines. On the other hand, if you don't use this directive, the code must be linked to the MPI library. We will discuss this point later.

Other preprocessor directives are defined in variable `DEFINES` in the `Makefile`:

```
# Compilation time parameters
NVECTOR = 64
NDIM = 3
NPRE = 8
NVAR = 8
SOLVER = hydro
PATCH =
EXEC = ramses
DEFINES = -DNVECTOR=$(NVECTOR) -DNDIM=$(NDIM) -DNPRE=$(NPRE) -DNVAR=$
(NVAR) -DSOLVER$(SOLVER)
```

These additional directives are called *Compilation Time Parameters*. They should be defined in the `Makefile` and the code must be recompiled entirely using:

```
$ make clean
$ make
```

We list now the definitions of these parameters.

`NVECTOR=64` : This parameter is used to set the vector size for computation-intensive operations. It must be determined experimentally on each new hardware.

`NPRE=4` : This parameter sets the precision of the floating point operations. `NPRE=4` stands for single precision arithmetics, while `NPRE=8` is for double precision.

`NENER=0` : This parameter sets the number of energy variables used in the hydro or mhd solver.

`NDIM=3` : This parameter sets the dimensionality of the problem. The value `NDIM=1` is for 1D, plan-parallel flows. `NDIM=2` and `NDIM=3` are resp. for 2D and 3D flows.

`SOLVER=hydro` : This parameter selects the type of hyperbolic solver used. Possible values are: `hydro` for the adiabatic Euler equations, `mhd`, the Constrained Transport scheme for the ideal MHD equations. and `rhd` for relativistic hydro.

`NVAR=8` : This parameters defines the number of variables in the hyperbolic solver. For `SOLVER=hydro`, `NVAR>=NDIM+2`. For `SOLVER=mhd`, `NVAR>=8` and for `SOLVER=rhd`, one has `NVAR>=5`.

Our goal is now to compile the code for a simple one-dimensional problem. You need to modify the `Makefile` so that:

```
NDIM=1
SOLVER=hydro
NVAR=3
```

Then type:

```
$ make
```

If everything goes well, all source files will be compiled and linked into an executable called `ramses1d`.

Additional compilation preprocessor flags

`-DTSC` : This parameter sets the triangular shape cloud approximation; only works with `NDIM=3`

`-DOUTPUT_PARTICLE_POTENTIAL` : This parameter forces the code to output particle potentials at snapshots

`-DQUADHILBERT` : This parameter sets longer Hilbert curve necessary if `levelmax>19`

`-DLONGINT` : This parameter switches to long ints (necessary when one has lots of particles)

`-DNOSYSTEM` : This parameter handles operating system commands

Executing the test case

To test the compilation, you need to execute a simple test case. Go up one level and type the following command:

```
$ cd trunk/ramses
$ bin/ramses1d namelist/tube1d.nml
```

The first part of the command is the executable we have just compiled. The second part, the only command line argument, is an input file containing the *run time parameters*. Several examples of such parameter files are stored in the `namelist/` directory. The namelist file we have just used `tube1d.nml` is the Sod test, a simple shock tube simulation in 1D. For comparison, we now show the last 14 lines of the standard output:

```
 Mesh structure
 Level  1 has           1 grids (       1,       1,       1,)
 Level  2 has           2 grids (       2,       2,       2,)
 Level  3 has           4 grids (       4,       4,       4,)
 Level  4 has           8 grids (       8,       8,       8,)
 Level  5 has          16 grids (      16,      16,      16,)
 Level  6 has          27 grids (      27,      27,      27,)
 Level  7 has          37 grids (      37,      37,      37,)
 Level  8 has          17 grids (      17,      17,      17,)
 Level  9 has          16 grids (      16,      16,      16,)
 Level 10 has          13 grids (      13,      13,      13,)
 Main step=    43 mcons=-1.97E-16 econs= 1.61E-16 epot= 0.00E+00
ekin= 1.38E+00
 Fine step=   688 t= 2.45047E-01 dt= 3.561E-04 a= 1.000E+00 mem= 7.6%
 Run completed
```

If your execution looks similar, it means your installation was successfull. Users are encouraged to redirect the standard output into a *log file*. This log file contains all simulation control variables, as well as output variables, but for 1D simulations only.

```
$ bin/ramses1d namelist/tube1d.nml > tube.log
```

Reading the log file

We will now briefly describe the structure and the nature of the information available in the log file. We will use as example the file `tube.log` we have just created. It should contain, starting from the top:

```
 _/_/_/       _/_/     _/    _/    _/_/_/   _/_/_/_/    _/_/_/
 _/    _/    _/  _/    _/_/_/_/   _/    _/  _/         _/    _/
 _/    _/   _/    _/   _/ _/ _/   _/        _/         _/
 _/_/_/     _/_/_/_/   _/    _/     _/_/    _/_/_/       _/_/
 _/    _/   _/    _/   _/    _/         _/  _/              _/
 _/    _/   _/    _/   _/    _/   _/    _/  _/         _/    _/
 _/    _/   _/    _/   _/    _/    _/_/_/   _/_/_/_/    _/_/_/
                        Version 3.0
        written by Romain Teyssier (CEA/DSM/IRFU/SAP)
                      (c) CEA 1999-2007
```

```
 Working with nproc =     1 for ndim = 1
 Using the hydro solver with nvar =  3

 Building initial AMR grid
 Initial mesh structure
 Level  1 has          1 grids (       1,       1,       1,)
 Level  2 has          2 grids (       2,       2,       2,)
 Level  3 has          4 grids (       4,       4,       4,)
 Level  4 has          8 grids (       8,       8,       8,)
 Level  5 has          8 grids (       8,       8,       8,)
 Level  6 has          8 grids (       8,       8,       8,)
 Level  7 has          8 grids (       8,       8,       8,)
 Level  8 has          8 grids (       8,       8,       8,)
 Level  9 has          6 grids (       6,       6,       6,)
 Level 10 has          4 grids (       4,       4,       4,)
 Starting time integration
 Output    58 cells
 ---------------------------------------------
 lev      x           d           u           P
  4  3.12500E-02  1.000E+00  0.000E+00  1.000E+00
  4  9.37500E-02  1.000E+00  0.000E+00  1.000E+00
...
  4  9.06250E-01  1.250E-01  0.000E+00  1.000E-01
  4  9.68750E-01  1.250E-01  0.000E+00  1.000E-01
 ---------------------------------------------
 Fine step=     0 t= 0.00000E+00 dt= 6.603E-04 a= 1.000E+00 mem= 3.2%
 Fine step=     1 t= 6.60250E-04 dt= 4.420E-04 a= 1.000E+00 mem= 3.2%
```

After the code banner and copyrights, the first line indicates that you are currently using 1 processor and 1 space dimension for this run. The second line confirms the solver used and the number of variables defined for this run. The code then reports that it is building the initial AMR grid. The next lines give the resulting mesh structure.

The first level of refinement in *ramses* covers the whole computational domain with 2 (resp. 4 and 8) cells in 1 (resp. 2 and 3) space dimension. The grid is then entirely refined up to `levelmin`, which in this case is defined in the parameter file to be `levelmin=3`. This defines the *coarse grid*. The grid is then adaptively refined up to `levelmax`, which in this case `levelmax=10`. Each line in the log file indicates the number of octs (or grids) at each level of refinement. The maximum number of grids in each level `level` is equal to `2**(level-1)` for `NDIM=1`, to `4**(level-1)` for `NDIM=2` and to `8**(level-1)` for `NDIM=3`.

The numbers inside parentheses give the minimum, maximum and average number of grids per processor. This is obviously only relevant to parallel runs.

The code then indicates that the time integration starts. After outputting the initial conditions to screen, the first *control line* appears, starting with the words `Fine  step=`. The control line gives information on each *fine step*, its current number, its current time coordinate, its current time step. Variable `a` is for cosmology runs only and gives the current expansion factor. The last

variable is the percentage of allocated memory currently used by ramses to store each flow variable on the grid.

In ramses, adaptive time stepping is implemented, which results in defining *coarse steps* and *fine steps*. Coarse steps correspond to the coarse grid, which is defined by variable `levelmin`. Fine steps correspond to finer levels, for which the time step has been recursively subdivided by a factor of 2. Fine levels are sub-cycled, twice as more as their parent coarse level. This explains why, at the end of the log file, only 43 coarse steps are reported (1 through 43), for 689 fine steps (numbered from 0 to 688).

When a coarse step is reached, the code writes in the log file the current mesh structure. A new control line then appears, starting with the words `Main step=`. This control line gives information on each coarse step, namely its current number, the current error in mass conservation within the computational box `mcons=`, the current error in total energy conservation `econs=`, the gravitational potential energy and the fluid total energy (kinetic plus thermal).

This constitutes the basic information contained in the log file. In 1D simulations, output data are also written to standard output, and thus to the log file. For 2D and 3D, output data are stored into unformatted Fortran binary files (named `output_00001`, `output_00002`...). In our example, the fluid variables are listed using 5 columns: level of refinement, position of the cell, density, velocity and pressure:

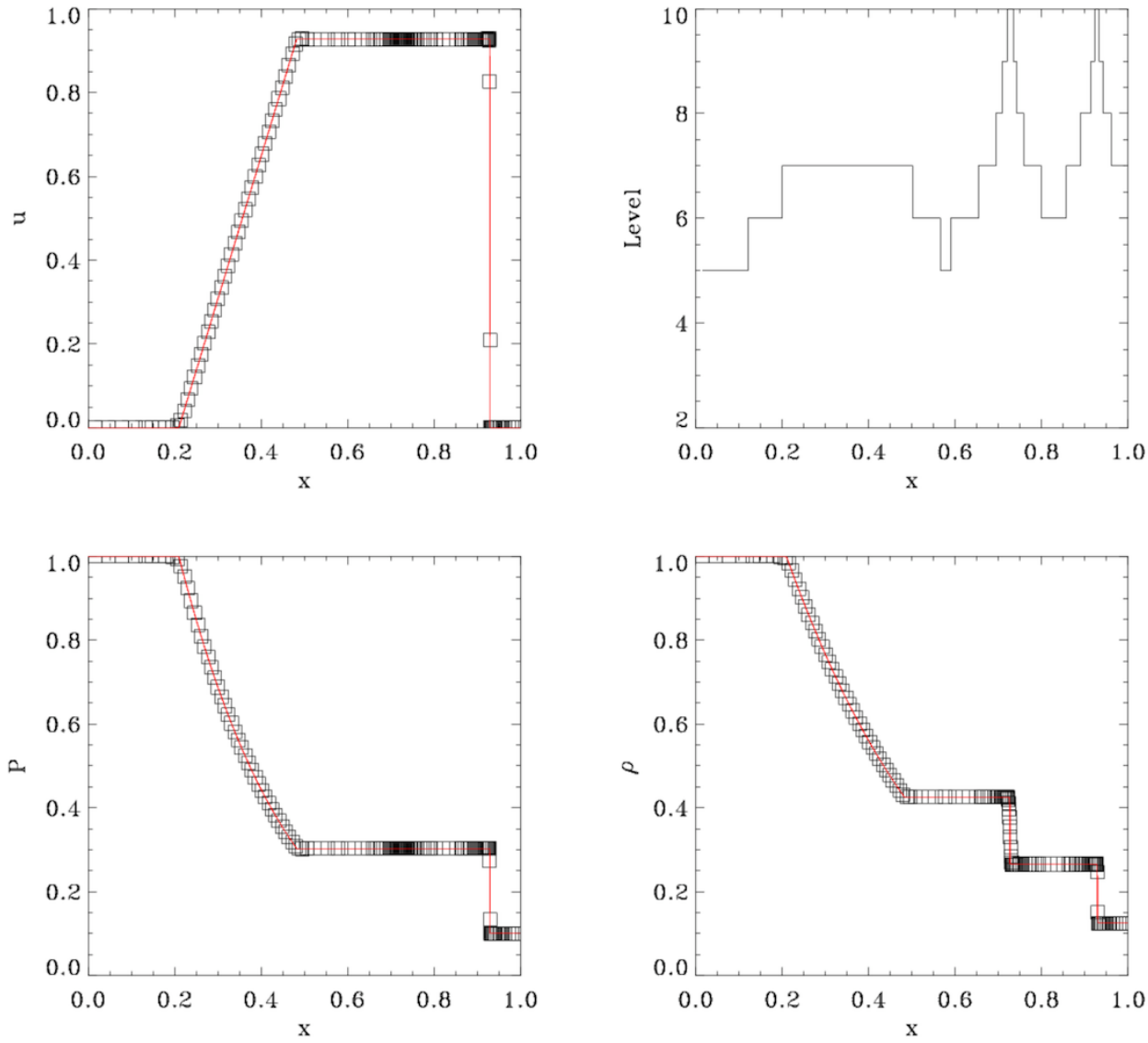
```
Output   142 cells
================================================
lev      x           d           u           P
 5  1.56250E-02  1.000E+00  0.000E+00  1.000E+00
 5  4.68750E-02  1.000E+00  0.000E+00  1.000E+00
 5  7.81250E-02  1.000E+00  0.000E+00  1.000E+00
 5  1.09375E-01  1.000E+00  1.564E-09  1.000E+00
 6  1.32812E-01  1.000E+00  2.112E-08  1.000E+00
```

You can cut and paste the 142 lines into another file and use your favorite data viewer like `xmgrace` or `gnuplot` to visualize the results. These should be compared to the plots shown on the figure below. If you have obtained comparable numerical values and levels of refinements, your installation is likely to be valid. You are encouraged to edit the parameter file `tube1d.nml` and play around with other parameter values, in order to test the code performances. You can also use other parameter files in the `namelist/` directory

If you would like to run a 2D simulation (using file `sedov2d.nml` for example), do not forget to recompile entirely the code using:

```
$ cd trunk/ramses/bin
$ make clean
$ make NDIM=2
```

This last image shows the numerical results obtained with ramses for the Sod shock tube test (symbols) compared to the analytical solution (red line).



Runtime Parameters

The Ramses parameter file is based on the Fortran namelist syntax. The Sod test parameter file is shown below, as it should appear if you edit it.

```
$ more tube1d.nml
This is the RAMSES parameter file for Sod''s shock tube test.

&RUN_PARAMS
hydro=.true.
nsubcycle=3*1,2
```

```
/

&AMR_PARAMS
levelmin=3
levelmax=10
ngridmax=2000
nexpand=1
boxlen=1.0
/

&BOUNDARY_PARAMS
nboundary=2
ibound_min=-1,+1
ibound_max=-1,+1
bound_type= 1, 1
/

&INIT_PARAMS
nregion=2
region_type(1)='square'
region_type(2)='square'
x_center=0.25,0.75
length_x=0.5,0.5
d_region=1.0,0.125
u_region=0.0,0.0
p_region=1.0,0.1
/

&OUTPUT_PARAMS
noutput=1
tout=0.245
/

&HYDRO_PARAMS
gamma=1.4
courant_factor=0.8
slope_type=2
riemann='hllc'
/

&REFINE_PARAMS
err_grad_d=0.05
err_grad_u=0.05
err_grad_p=0.05
interpol_var=0
interpol_type=2
/
```

This parameter file is organized in namelist blocks. Each block starts with &BLOCK_NAME and ends with the character "/". Within each block, you can specify parameter values using standard Fortran namelist syntax. There are currently 11 different parameter blocks implemented in RAMSES.

4 parameters blocks are mandatory and must always be present in the parameter file. These are `&RUN_PARAMS`, `&AMR_PARAMS`, `&OUTPUT_PARAMS` and `&INIT_PARAMS`. The 8 other blocks are optional. They must be present in the file only if they are relevant to the current run. These are `&BOUNDARY_PARAMS`, `&HYDRO_PARAMS`, `&PHYSICS_PARAMS`, `&POISSON_PARAMS`, `&REFINE_PARAMS`, `&CLUMPFIND_PARAMS`, `&SINK_PARAMS` and `&MOVIE_PARAMS`.

The variables you can set or adjust in each namelist block are described in more details in the following sections.

Global parameters

This block, called `&RUN_PARAMS`, contains the run global control parameters. These parameters are now briefly described. More thorough explanations will be given in dedicated sections of the wiki.

Variable name, syntax, default value	Fortran type	Description
`cosmo=.false.`	`logical`	Activate cosmological supercomoving cooordinates and computes the expansion factor
`pic=.false.`	`logical`	Activate Particle-In-Cell solver
`sink=.false.`	`logical`	Activate sink particles
`clumpfind=.false.`	`logical`	Activate the clump finder
`tracer=.false.`	`logical`	Use Monte Carlo tracer particles

Variable name, syntax, default value	Fortran type	Description
`unbind=.false.`	`logical`	Activate the particle unbinding for clumps
`make_mergertree=.false.`	`logical`	Make merger trees
`poisson=.false.`	`logical`	Activate Poisson solver for self-gravity
`hydro=.false.`	`logical`	Activate hydro or MHD solver.
`rt=.false.`	`logical`	Activate radiative transfer using CPU-based M1 solver. This solver works on the AMR grid.
`aton=.false.`	`logical`	Activate radiative transfer using GPU-based M1 solver. This solver works only on unigrid at `levelmin`.
`verbose=.false.`	`logical`	Activate verbose mode.
`cost_weighting=.true.`	`logical`	Load balancing based on computational cost, not memory. This is rather expensive in term of memory usage. For memory limited runs, using `cost_weighting=.false.` is better.
`nrestart=0`	`integer`	

Variable name, syntax, default value	Fortran type	Description
		Output file number from which the code loads backup data and resumes the simulation, The default value, zero, is for a fresh start from the beginning (time=0).
`nrestart_quad=0`	`integer`	Restart with double precision Hilbert keys. Must be equal to `nrestart`. Default value is 0.
`nstepmax=1000000`	`integer`	Maximum number of coarse time step. This can be used to terminate the simulation after a fixed amount of main steps.
`ncontrol=1`	`integer`	Frequency of screen output for control lines (to stdout), usually redirected into a log file.
`nremap=0`	`integer`	Frequency of call, in units of coarse time steps, for the load balancing routine, for MPI runs only. The default value, zero, means never. Load balancing is a slow operation, so use as high a value as possible.
`ordering="hilbert"`	`character(len=128)`	Cell ordering method used in the domain decomposition of the grid,

Variable name, syntax, default value	Fortran type	Description
		for MPI runs only. Possible values are `hilbert`, `planar` and `angular`.
`nsubcycle=2,2,2,2,2,2,`	`integer array`	Number of fine level sub-cycling steps within one coarser level time step. Each value in the array corresponds to a given level of refinement, starting from the coarse grid at `levelmin` up to the finest level at `levelmax`. `nsubcycle(1)=1` means that `levelmin` and `levelmin+1` are synchronized. To enforce single time stepping across the whole AMR hierarchy, you need to set `nsubcycle=1,1,1,1,1,1,1,`
`static=.false.`	`logical`	Activate full static mode (RT post processing)
`static_dm=.false.`	`logical`	Activate static mode for DM particles only (isolated initial conditions relaxation)
`static_stars=.false.`	`logical`	Activate static mode for star particles only (isolated initial conditions relaxation)
`remap_pscalar=ndim+3,ndim+4,...,nvar`	`integer array`	Mapping for the passive scalars and non-thermal

Variable name, syntax, default value	Fortran type	Description
		pressures. Value indicates in which variable the scalar from the restart should be loaded. [0 = ignore this scalar in the restart output, -N = do not read but initialise ivar=N to 0, N = read and initialise ivar=N from the restart output]. For example `remap_pscalar=-6,0,7,8,9` translates to: do not read first restart var but initialise ivar=6 to 0, skip second restart var, read ivar=[8,9,10] from the restart snapshot and store it in the current ivar=[7,8,9].

AMR Parameters

This sets of parameters, contained in the namelist block `&AMR_PARAMS`, controls the AMR grid global properties. Parameters specifying the refinement strategy are described elsewhere, and the corresponding namelist block `&REFINE_PARAMS` is used only if `levelmax>levelmin`.

Variable name, syntax, default value	Fortran type	Description
`levelmin=1`	`integer`	Minimum level of refinement. This parameter sets the size of the coarse (or base) grid to `nx=2**levelmin`.
`levelmax=1`	`integer`	Maximum level of refinement. If `levelmax=levelmin`, the simulation will be executed on a standad Cartesian grid of linear size `nx=2**levelmin`

Variable name, syntax, default value	Fortran type	Description
`ngridmax=1`	`integer`	Maximum number of grids (or octs) that can be allocated during the run within each MPI process.
`ngridtot=1`	`integer`	Maximum number of grids (or octs) that can be allocated during the run for the entire set of MPI processes. One has `ngridmax=ngridtot/ncpu`.
`npartmax=1`	`integer`	Maximum number of particles of all types that can be allocated during the run within each MPI process.
`nparttot=1`	`integer`	Maximum number of particles of all types that can be allocated during the run for the entire set of MPI processes. One has `npartmax=nparttot/ncpu`.
`nsinkmax=1`	`integer`	Maximum number of sink particles during the run. Sink particles are duplicated over all MPI processes.
`nexpand=1`	`integer`	Number of times the mesh expansion is applied to the refinement map (see mesh smoothing).
`boxlen=1.0`	`real`	Box size in user's units
`nlevel_collapse=3`	`integer`	Number of levels above `levelmin` to follow initial halos collapse with a constant comoving resolution (`cosmo=.true.` only)

The parameters `ngridtot` and `nparttot` specify the maximum memory allocated for AMR grids and collisionless particles respectively. These numbers should be greater than or equal to the actual number of AMR grids and particles used during the course of the simulation. `ngridtot` stands for the total number of AMR grids allocated over all MPI processes. The `ngridmax` parameter can be used equivalently, but stands for the local number of AMR grids within each MPI process. Obviously, one has `ngridtot=ngridmax*ncpu`.

⚠ Warning

Recall that, in RAMSES, we call “AMR grid” or “oct” a group of \(2^{\mathrm{ndim}}\) cells. If for some reason, during the course of the execution, the maximum allowed number of grids or particles has been reached, the simulation stops with the message:

```
No more free memory
Increase ngridmax
```

In this case, don’t panic: just increase ngridmax in the Parameter File and resume the execution, starting from the last backup file.

Memory management

The `ngridmax` and `npartmax` parameters (or there `*tot` equivalents) control the memory allocated by RAMSES. It is important to know how much memory in Gb is actually allocated by RAMSES for a given choice of parameters. This can be approximated by:

- \(0.7(\mathtt{ngridmax}/10^6) + 0.7(\mathtt{npartmax}/10^7)\) Gbytes per cpu for pure N -body runs,
- \(1.4(\mathtt{ngridmax}/10^6) + 0.7(\mathtt{npartmax}/10^7)\) Gb per cpu for N -body and hydro runs,
- \(1.0(\mathtt{ngridmax}/10^6)\) Gb per cpu for pure hydro runs.

Because of MPI communications overheads, the actual memory used can be slightly higher. Note that these numbers are valid for double precision arithmetic. For single precision runs, using the preprocessor directive `-DNPRE=4`, you can decrease these figures by 40%.

⚠ Warning

The memory occupation will be higher when using modules that add new variables, such as MHD, Radiative transfer, Chemistry, Passive scalars, etc.

Refinement parameters

The block named `&REFINE_PARAMS` contains the parameters related to grid refinement.

Variable name	Fortran type	Default value	Description
`x_refine`	`real array`	0.0	Geometry-based strategy: center of the refined region at each level of the AMR grid.
`y_refine`	`real array`	0.0	Geometry-based strategy: center of the refined region at each level of the AMR grid.
`z_refine`	`real array`	0.0	Geometry-based strategy: center of the refined region at each level of the AMR grid.
`r_refine`	`real array`	1e10	Geometry-based strategy: **diameter** (yes, not a radius) of the refined region at each level.
`a_refine`	`real array`	1.0	Geometry-based strategy: ratio Y/X of the refined region at each level.
`b_refine`	`real array`	1.0	Geometry-based strategy: ratio Z/X of the refined region at each level.
`exp_refine`	`real array`	2.0	Geometry-based strategy: exponent of the norm.
`jeans_refine`	`real array`	-1.0	Jeans refinement strategy: each level is refined if the cell size exceeds the local Jeans length divided by jeans_refine(ilevel).
`mass_cut_refine`	`real`	-1.0	Mass threshold for particle-based refinement
`m_refine`	`real array`	-1.0	Quasi-Lagrangian strategy: each level is refined if the baryons mass in a cell exceeds `m_refine(ilevel)*mass_sph`, or if the number of dark matter particles exceeds `m_refine(ilevel)`, whatever the mass is.

Variable name	Fortran type	Default value	Description
`mass_sph`	`real`	0.0	Quasi-Lagrangian strategy: `mass_sph` is used to set a typical mass scale. For cosmo runs, its value is set automatically.
`err_grad_d`	`real`	-1.0	Discontinuity-based strategy: density gradient relative variations above which a cell is refined
`err_grad_u`	`real`	-1.0	Discontinuity-based strategy: velocity gradient relative variations above which a cell is refined
`err_grad_p`	`real`	-1.0	Discontinuity-based strategy: pressure gradient relative variations above which a cell is refined
`floor_d`	`real`	1e-10	Density floor below which gradients are ignored
`floor_u`	`real`	1e-10	Velocity floor below which gradients are ignored
`floor_p`	`real`	1e-10	Pressure floor below which gradients are ignored
`ivar_refine`	`int`	-1	Variable index for refinement
`var_cut_refine`	`real`	-1.0	Threshold for variable-based refinement
`interpol_var`	`int`	0	Variables used to perform interpolation (prolongation) and averaging (restriction). `interpol_type=0`: conservatives; `interpol_type=1`: primitives
`interpol_type`	`int`	1	Type of slope limiter used in the interpolation scheme for newly refined cells. `interpol_type=0`: Straight injection (1st order), `interpol_type=1`: MinMod limiter, `interpol_type=2`: MonCen

Variable name	Fortran type	Default value	Description
			limiter, `interpol_type=3`: unlimited central slope, `interpol_type=4`: type 3 for velocity and type 2 for density and internal energy (if `interpol_var=2`)
`sink_refine`	`bool`	`.false.`	Refines grid around sinks to levelmax

Output Parameters

This namelist block, called `&OUTPUT_PARAMS`, is used to set up the frequency and properties of data output to disk.

Variable name, syntax, default value	Fortran type	Description
`noutput=1`	`integer`	Number of specified output times. At least one output time should be given, corresponding to the end of the simulation.
`foutput=1000000`	`integer`	Frequency of additional outputs in units of coarse time steps. `foutput=1` means one output at each time step. Specified outputs (see above) will not be superseded by this parameter.
`tout=0.0,0.0,0.0,`	`real array`	Value of specified output times.
`aout=1.1,1.1,1.1,`	`real array`	Value of specified output expansion factor (for cosmology runs only). `aout=1.0` means “present epoch” or “zero redshift”.

Variable name, syntax, default value	Fortran type	Description
`delta_tout=0`	`real`	Frequency of outputs in user time units.
`delta_aout=0`	`real`	Frequency of outputs in expansion factor (for cosmology runs only).
`tend=10`	`real`	End time of the simulation, in user time units.
`aend=0`	`real`	End time of the simulation, in expansion factor (for cosmology runs only).
`walltime_hrs=-1`	`real`	Wallclock time given in ramses job submission, used for dumping an output at the end. Default value of -1 means this is not used.
`minutes_dump=1`	`real`	Dump an output this many minutes before walltime_hrs
`write_conservative=.false.`	`logical`	Output conservative hydro variables as stored in `uold` instead of primitive ones
`read_conservative=.false.`	`logical`	When conservative variables have been outputted, this flag should be set to `.true.` to match the correct variables on restart.
`exact_output_time=.false.`	`logical`	Enforce outputs at the exact requested times (`tout` or `aout`) by adjusting the timestep.

Warning (R. Teyssier) When using the `exact_output_time` option, having an abrupt change of time step can potentially affect the quality of the solution. Orbital integration is not symplectic anymore and second-order hydro schemes with non-linear slope limiters will produce a different solution.

Restart from previous output

A simulation which has been terminated during run time can be restarted from the last (or any) snapshot output, by setting

```
nrestart=64
```

in the namelist file to the output number. If you don't want to change the namelist file, simply append the restart output number to the command execution, e.g.

```
./ramses3d parameters.nml 64
```

Saving progress before job limit termination

SLURM jobs on computer clusters often have a time limit, after which the process will be terminated. If you don't want to lose the computation progress since your last regular output, you can instruct the SLURM scheduler to send a "warning" signal to the process seconds before killing it with **–signal=10@**. An example sbatch script with set to 120 seconds would look like this:

```
#!/bin/bash
#SBATCH -J simulation
#SBATCH -p normal
#SBATCH -n 128
#SBATCH --time=24:00:00
#SBATCH --output=logfile-%j.txt
#SBATCH --error=error-%j.txt
#SBATCH --signal=10@120

aprun -B ./ramses3d parameters.nml 64
```

RAMSES will catch this signal and dump the current simulation state to a new output, which can be used to restart the simulation from.

⚠ Warning

The signalling does not work on all machines. Sometimes the signal 10 is accompanied by a kill signal and the job is dead before it can perform an output. In this case, there are a couple of useful parameters in the output_params namelist: `walltime_hrs` can be used to specify the walltime given to a job in hours, and `minutes_dump` can then be used to tell RAMSES to dump an output a few minutes before the walltime runs out.

Dump immediate output

The above mechanism can be used to force an output be written to the disk immediately at any time during the simulation by sending signal 10 to the process:

```
scancel --signal=10 <jobid>
```

or, if you run without SLURM:

```
kill --signal=10 <processid>
```

Initial Conditions Parameters

This sets of parameters, contained in the namelist block `&INIT_PARAMS`. This is used to set up the initial conditions.

Variable name, syntax, default value	Fortran type	Description
`condinit_kind=region`	`60*char`	Set which kind of initial condition to use. Default is the region-based initialisation
`nregion=1`	`integer`	Number of independent regions in the computational box used to set up initial flow variables.
`region_type=square`	`10*char`	Geometry defining each region. `square` defines a generalized ellipsoidal shape, while `point` defines a delta function in the flow.
`x_center=0.0`	`real arrays`	X coordinate of the center of each region.
`y_center=0.0`	`real arrays`	Y coordinate of the center of each region.

Variable name, syntax, default value	Fortran type	Description
`z_center=0.0`	`real arrays`	Z coordinate of the center of each region.
`length_x=0.0`	`real arrays`	Size in X direction of each region.
`length_y=0.0`	`real arrays`	Size in Y direction of each region.
`length_z=0.0`	`real arrays`	Size in Z direction of each region.
`exp_region=2.0`	`real arrays`	Exponent defining the norm used to compute distances for the generalized ellipsoid. `exp_region=2` corresponds to a spheroid, `exp_region=1` to a diamond shape, `exp_region>=10` to a perfect square.
`d_region=0.0`	`real arrays`	Density. For `point` regions this is used to define mass.
`u_region=0.0`	`real arrays`	X velocity. For `point` regions this is used to define velocity.
`v_region=0.0`	`real arrays`	Y velocity. For `point` regions this is used to define velocity.
`w_region=0.0`	`real arrays`	Z velocity. For `point` regions this is used to define velocity.
`p_region=0.0`	`real arrays`	Pressure. For `point` regions this is used to define specific pressure.

Variable name, syntax, default value	Fortran type	Description
`A_region=0.0`	`real arrays`	X magnetic field.
`B_region=0.0`	`real arrays`	Y magnetic field.
`C_region=0.0`	`real arrays`	Z magnetic field.
`filetype=ascii`	`20*char`	Type of initial conditions file for particles. Possible choices are `ascii` or `grafic`.
`aexp_ini=10.0`	`real`	This parameter sets the starting expansion factor for cosmology runs only. Default value is read in the IC file.
`multiple=.false.`	`logical`	If `.true.`, each processors reads its own IC file. For parallel runs only.
`initfile=`	`80*char`	Directory where IC files are stored (when relevant).

Advanced initial conditions

The `condinit` routine in `hydro/condinit.f90` can be modified to set custom initial conditions. The calling sequence is `call condinit(x,u,dx,ncell)`, where

- `x` is an input array of cell center positions,
- `u` is an output array containing the volume average of the fluid conservative variables, namely (\(\rho\), \(\rho u\), \(\rho v\), \(\rho w\) and \(E\)), in this exact order. If more variables are defined, then the user should exploit this routine to define them too.
- `dx` is a single real value containing the cell size for all the cells and ncell is the number of cells.

This routine can be used to set the initial conditions directly with Fortran instructions.

Changed in version 2025: There is now the possibility to change initial condition without recompiling each time. Just write you initial condition as a new routine and add it to the select/ case structure

```
  select case (condinit_kind)

  case('region')
     call condinit_default(x, u, dx, nn)

  case('your_new_routine')
     call condinit_your_new_routine(x, u, dx, nn)

  case DEFAULT
     if (myid == 1.and. first_call)  write(*,*) "[condinit] Void or
invalid condinit_kind, using default IC"
     call condinit_default(x, u, dx, nn)

  end select
```

Input files

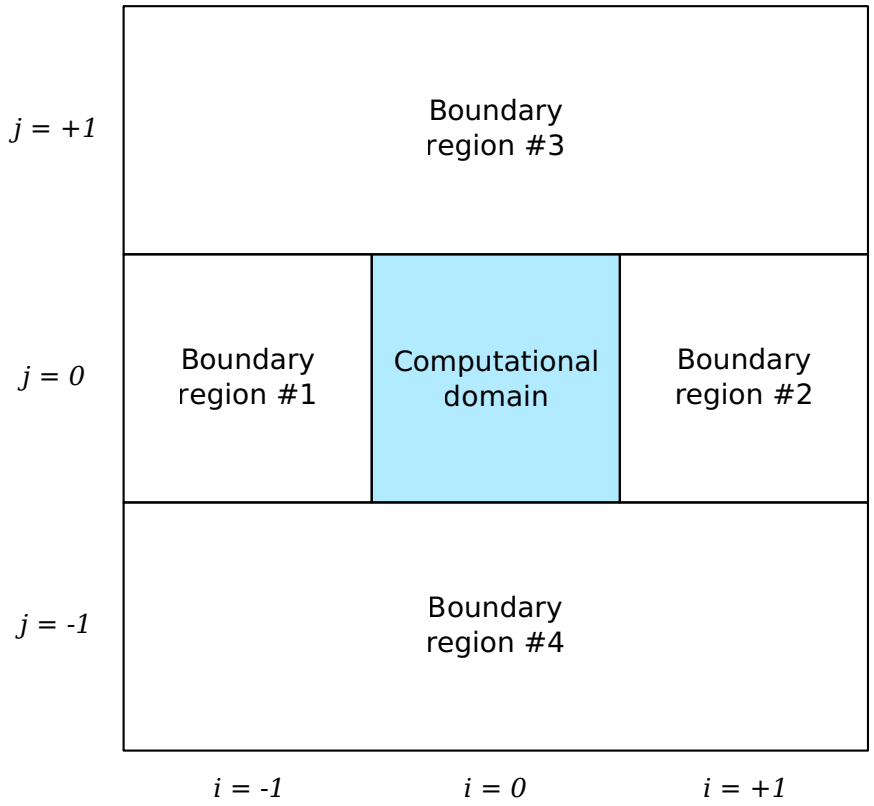
Another way to define initial conditions in RAMSES is by using input files (`initfile` parameter) in the grafic format.

Boundary Parameters

This set of parameters, contained in the namelist block `&BOUNDARY_PARAMS`, is used to set up boundary conditions on the current simulation. If this namelist block is absent, periodic boundary conditions are assumed. Setting up other types of boundary conditions in RAMSES is quite complex. The default setting, corresponding to a periodic box, should be sufficient in most cases. The strategy to set up boundary conditions is based on using “ghost regions” outside the computational domain, where flow variables are carefully specified in order to mimic the effect of the chosen type of boundary. Note that the order in which boundary regions are specified in the namelist is very important, especially for reflexive or zero gradient boundaries. Specific examples can be found in the namelist/ directory of the package.

Variable name, syntax, default value	Fortran type	Description
`nboundary=1`	`integer`	Number of ghost regions used to specify the boundary conditions.
`bound_type=0,0,0,`	`integer array`	Type of boundary conditions to apply in the corresponding ghost region. Possible values are: `bound_type=0` (periodic), `bound_type=1` (reflexive), `bound_type=2` (outflow, zero gradients), `bound_type=3` (inflow, user specified).
`d_bound=0.0` `u_bound=0.0` `v_bound=0.0` `w_bound=0.0` `p_bound=0.0`	`real arrays`	Flow variables in each ghost region (density, velocities and pressure). They are used only for inflow boundary conditions.
`ibound_min=0` `jbound_min=0` `kbound_min=0`	`integer arrays`	Coordinates of the lower, left, bottom corner of each boundary region. Each coordinate lies between -1 and +1 in each direction.
`ibound_max=0` `jbound_max=0` `kbound_max=0`	`integer arrays`	Likewise, for the upper, right and upper corner of each boundary region.

Example



The 4 boundary regions shown in the above figure are defined by the following namelist block:

```
&BOUNDARY_PARAMS
nboundary=4
ibound_min=-1,+1,-1,-1
ibound_max=-1,+1,+1,+1
jbound_min= 0, 0,+1,-1
jbound_max= 0, 0,+1,-1
bound_type= 1, 1, 1, 1
/
```

The first region is located in the rectangle defined by coordinate (i=-1,j=0), while the third region is defined by coordinates (-1≤i≤+1, j=+1). The boundary type for all 4 regions is set to “reflexive” (bound_type=1). The fluid variables within the ghost region are therefore taken equal to the values of their symmetric cells, with respect to the boundary. This is why the order of the ghost regions is so important: regions 1 and 2 are updated first, using only the fluid variables within the computational domain. Regions 3 and 4 are updated afterwards, using the fluid variables within the computational domain, but also within regions 1 and 2.

In this way, all cells within boundary regions are properly defined, especially in the 4 corners of the computational domain. It is possible to define only 2 regions (say region 1 and 2 in the Figure),

the orthogonal direction will be considered as periodic. For gravity runs, the gravitational force is also updated in the ghost regions, following the same rules than for the velocity vector.

Hydro parameters

This namelist is called &HYDRO_PARAMS, and is used to specify runtime parameters for the Godunov solver. These parameters are quite standard in computational fluid dynamics. We briefly describe them now

Variable name, syntax, default value	Fortran type	Description
`gamma=1.4`	`Real`	Adiabatic exponent for the perfect gas EOS
`gamma_rad=1.333`	`Real`	Adiabatic exponent for the non-thermal pressure EOS (if NENER>1)
`courant_factor=0.5`	`Real`	CFL number for time step control (less than 1)
`smallr=1d-10`	`Real`	Minimum density to prevent floating exceptions
`smallc=1d-10`	`Real`	Minimum sound speed to prevent floating exceptions
`riemann='llf'`	`Character LEN=20`	Name of the desired Riemann solver. Possible choices are 'exact', 'acoustic', 'llf', 'hll' or 'hllc' for the hydro solver and 'llf', 'hll', 'roe', 'hlld', 'upwind' and 'hydro' for the MHD solver.
`riemann2d='llf'`	`Characher LEN=20`	Name of the desired 2D Riemann solver for the induction equation (MHD only). Possible choices are 'upwind', 'llf', 'roe', 'hll', and 'hlld'.
`scheme='muscl'`	`Character LEN=20`	Name of the desired Godunov integrator. The hydro solver accepts 'muscl' (MUSCL-HANCOCK

Variable name, syntax, default value	Fortran type	Description
		scheme) or 'pldme' (Collela's PLMDE scheme). The MHD solver accepts 'muscl' or 'induction'.
`niter_riemann=10`	`Integer`	Maximum number of iterations used in the exact Riemann solver
`slope_type=1`	`Integer`	Type of slope limiter used in the Godunov scheme for the piecewise linear reconstruction: slope_type=0: First order scheme, slope_type=1: MinMod limiter, slope_type=2: MonCen limiter, slope_type=3: Multi-dimensional MonCen limiter. In 1D runs only, it is also possible to choose:slope_type=4: Superbee limiter, slope_type=5: Ultrabee limiter
`slope_mag_type=1`	`Integer`	Type of slope limiter used in the Godunov scheme in the MHD solver
`pressure_fix=.false.`	`logical`	Activate hybrid scheme (conservative or primitive) for high-Mach flows. Useful to prevent negative temperatures.
`beta_fix=0d0`	`Real`	With pressure_fix=.true., changes the threshold at which the energy is truncated
`difmag=0d0`	`Real`	Modifies diffusive flux in the PLDME integrator (hydro only).
`eta_mag=0d0`	`Real`	Modifies dtdiff in PLDME integrator (MHD only, warning, divide by zero error if eta_mag=0d0)

Changed in version 2017: In the version of RAMSES after RUM 2017 (PR #284 and after), new blocks were introduced instead of one large `&PHYSICS_PARAMS`:

Physics and Units Parameters

Cooling parameters

The block named `&COOLING_PARAMS` contains the parameters related to cooling / basic chemistry

Variable name	Fortran type	Default value	Description
`cooling`	`boolean`	`.false.`	Enable gas atomic & metal cooling
`metal`	`logical`	`.false.`	Enable metals advection (requires 1 hydro variable)
`isothermal`	`logical`	`.false.`	(deprecated) Enable equation of state for gas (heating and cooling disabled if `.true.`)
`barotropic_eos`	`logical`	`.false.`	Enable barotropic equation of state for gas (heating and cooling disabled if `.true.`). Replaced 'isothermal'
`barotropic_eos_form`	`string`	`legacy`	Type of barotropic EOS. Options: isothermal, polytrope, double_polytrope, custom, legacy
`polytrope_rho`	`real array`	1.0d50	sets normalisation or knee-densities in EOS (see hydro/eos.f90), in g/cm3
`polytrope_index`	`real array`	1.0d0	sets polytropic indices in EOS (see hydro/eos.f90)
`T_eos`	`real`	10	sets T0 in EOS (isothermal temperature or temperature normalisation), in K

Variable name	Fortran type	Default value	Description
`mu_gas`	`real`	1d0	molecular weight
`haardt_madau`	`logical`	`.false.`	Enable UV background
`J21`	`real`	0.0	UV flux at threshold in 10^21 units
`a_spec`	`real`	1.0	Slope of the UV spectrum
`self_shielding`	`logical`	`.false.`	Enable self-shielding for densities above 0.01 g.cm^-3
`z_ave`	`real`	0.0	Initial average metal abundance
`z_reion`	`real`	8.5	Reionization redshift (UV background disabled for higher redshifts)
`ind_rsink`	`real`	4.0	Number of cells defining the radius of the sphere where AGN feedback is active
`T2max`	`real`	HUGE	Temperature ceiling for gas heating (heating ceiling if `isothermal=.false.`)
`neq_chem`	`logical`	`.false.`	Enable non-equilibrium chemistry
`X`	`real`	0.76d0	Hydrogen fraction
`Y`	`real`	0.24d0	Helium fraction

Star formation parameters

The block named `&SF_PARAMS` contains the parameters related to star formation

Variable name	Fortran type	Default value	Description
`m_star`	`real`	-1.0	Star particle mass in units of mass_sph
`n_star`	`real`	0.1	Star formation density threshold in H/cc
`T2_star`	`real`	0.0	Typical ISM polytropic temperature (cooling floor if `isothermal=.false.`)
`g_star`	`real`	1.6	Typical ISM polytropic index (cooling floor if `isothermal=.false.`)
`del_star`	`real`	2D2	Star formation density threshold in critical density (`cosmo=.true.` only)
`eps_star`	`real`	0.0	Star formation efficiency
`jeans_ncells`	`real`	-1.0	Jeans polytropic equation of state (cooling floor if `isothermal=.false.`)
`sf_virial`	`logical`	`.false.`	Enable turbulent star formation prescriptions (requires 1 hydro variable)
`sf_trelax`	`real`	0.0	Relaxation time for star formation (`cosmo=.false.` only)
`sf_tdiss`	`real`	0.0	Dissipation timescale for subgrid turbulence in units of turbulent crossing time (`sf_virial=.true.` only)

Variable name	Fortran type	Default value	Description
`sf_model`	`integer`	3	Turbulent star formation model (`sf_virial=.true.` only)
`sf_log_properties`	`logical`	`.false.`	Output gas properties of cells undergoing a star particle birth or supernova event
`sf_compressive`	`logical`	`.false.`	Store and advect the compressive and solenoidal turbulent field in two different hydro variables (`sf_virial=.true.` only)

Feedback parameters

The block named `&FEEDBACK_PARAMS` contains the parameters related to stellar feedback

Variable name	Fortran type	Default value	Description
`eta_sn`	`real`	0.0	Supernova mass fraction
`eta_ssn`	`real`	0.95	Single supernova ejected mass fraction (`sf_imf=.true.` only)
`t_sne`	`real`	10.0	Supernova blast time in Myr
`delayed_cooling`	`logical`	`.false.`	Enable delayed cooling through passive energy scalar advection (requires 1 hydro variable)
`t_diss`	`real`	20.0	Dissipation timescale for supernova feedback in Myr (`delayed_cooling=.true.` only)

Variable name	Fortran type	Default value	Description
`yield`	`real`	0.0	Supernova metal yield
`mass_gmc`	`real`	0.0	Stochastic exploding GMC mass in solar mass
`kappa_IR`	`real`	0.0	IR dust opacity for supernova feedback
`f_ek`	`real`	0.0	Supernova kinetic energy fraction (only between 0 and 1)
`f_w`	`real`	0.0	Supernova mass loading factor (`f_ek>0` only)
`rbubble`	`real`	0.0	Supernova superbubble radius in pc (`f_ek>0` only)
`ndebris`	`integer`	1	Supernova debris particle number (`f_ek>0` only)
`mass_star_max`	`real`	120.0	Maximum mass of a star to undergo a supernova with `eta_ssn` efficiency in solar mass (`sf_imf=.true.` only)
`mass_sne_min`	`real`	10.0	Minimum mass of a star to undergo a supernova blast in solar mass (`sf_imf=.true.` only)
`ir_feedback`	`logical`	`.false.`	Enable IR feedback from accreting sink particles
`ir_eff`	`real`	0.75	Efficiency of the IR feedback on sink particles (`ir_feedback=.true.` only)

Units parameters

The block named `&UNITS_PARAMS` contains the parameters related to units that are set "by hand" (`cosmo=.false.` only!).

Variable name	Fortran type	Default value	Description
`units_density`	`real`	1.0	Density unit in cgs (only for `cosmo=.false.`, requires G=1)
`units_time`	`real`	1.0	Time unit in cgs (only for `cosmo=.false.`, requires G=1)
`units_length`	`real`	1.0	Length unit in cgs (only for `cosmo=.false.`, requires G=1)

GRACKLE parameters

Additionally, if the code has been compiled with GRACKLE=1 in the Makefile and properly linked against the hdf5 and grackle libraries, it is possible to define a `&GRACKLE_PARAMS` block to control the grackle parameters. Please visit https://grackle.readthedocs.io/en/grackle-3.0/Parameters.html for a more extensive description of the grackle parameters.

Variable name	Fortran type	Default value	Description
`use_grackle`	`integer`	1	Activate Grackle
`grackle_with_radiative_cooling`	`integer`	1	Include radiative cooling
`grackle_primordial_chemistry`	`integer`	0	Primordial chemistry flag

Variable name	Fortran type	Default value	Description
`grackle_metal_cooling`	`integer`	0	Enable metal cooling using Cloudy tables
`grackle_UVbackground`	`integer`	0	Enable UV background
`grackle_cmb_temperature_floor`	`integer`	1	Enable effective CMB temperature floor
`grackle_h2_on_dust`	`integer`	0	Enable H2 formation on dust grains
`grackle_photoelectric_heating`	`integer`	0	Enable a spatially uniform heating term approximating photo-electric heating
`grackle_use_volumetric_heating_rate`	`integer`	0	Signal that an array of volumetric heating rates is being provided
`grackle_use_specific_heating_rate`	`integer`	0	Signal that an array of specific heating rates

Variable name	Fortran type	Default value	Description
			is being provided
grackle_three_body_rate	integer	0	Flag to control which three-body H2 formation rate is used
grackle_cie_cooling	integer	0	Enable H2 collision-induced emission cooling
grackle_h2_optical_depth_approximation	integer	0	Enable H2 cooling attenuation
grackle_ih2co	integer	1	
grackle_ipiht	integer	1	
grackle_NumberOfTemperatureBins	integer	600	
grackle_CaseBRecombination	integer	0	
grackle_Compton_xray_heating	integer	0	Flag to enable Compton heating from an X-ray background

Variable name	Fortran type	Default value	Description
`grackle_LWbackground_sawtooth_suppression`	`integer`	0	Flag to enable suppression of Lyman-Werner flux due to Lyman-series absorption
`grackle_NumberOfDustTemperatureBins`	`integer`	250	
`grackle_use_radiative_transfer`	`integer`	0	Signal that arrays of ionization and heating rates from radiative transfer solutions are being provided
`grackle_radiative_transfer_coupled_rate_solver`	`integer`	0	Flag that must be enabled to couple the passed radiative transfer fields to the chemistry solver
`grackle_radiative_transfer_intermediate_step`	`integer`	0	Flag to enable intermediate stepping in applying radiative transfer fields

Variable name	Fortran type	Default value	Description
			to chemistry solver
`grackle_radiative_transfer_hydrogen_only`	`integer`	0	Flag to only use hydrogen ionization and heating rates from the radiative transfer solutions
`grackle_self_shielding_method`	`integer`	0	Switch to enable approximate self-shielding from the UV background
`grackle_Gamma`	`real`	5./3.	The ratio of specific heats for an ideal gas
`grackle_photoelectric_heating_rate`	`real`	8.5D-26	The heating rate in units of erg cm-3 s-1
`grackle_HydrogenFractionByMass`	`real`	0.76	
`grackle_DeuteriumToHydrogenRatio`	`real`	2.0*3.4e-5	
`grackle_SolarMetalFractionByMass`	`real`	0.01295	

Variable name	Fortran type	Default value	Description
`grackle_TemperatureStart`	`real`	1.0	
`grackle_TemperatureEnd`	`real`	1.0D9	
`grackle_DustTemperatureStart`	`real`	1.0	
`grackle_DustTemperatureEnd`	`real`	1500.	
`grackle_LWbackground_intensity`	`real`	0.0	Intensity of a constant Lyman-Werner H2 photo-dissociating radiation field in units of 10-21 erg s-1 cm-2 Hz-1 sr-1
`grackle_UVbackground_redshift_on`	`real`	7.0	
`grackle_UVbackground_redshift_off`	`real`	0.0	
`grackle_UVbackground_redshift_fullon`	`real`	6.0	
`grackle_UVbackground_redshift_drop`	`real`	0.0	
`grackle_cloudy_electron_fraction_factor`	`real`	9.153959D-3	
`grackle_data_file`	`character`		Path to the data file containing the

Variable name	Fortran type	Default value	Description
			metal cooling and UV background HDF5 tables

Physics parameters (LEGACY ONLY)

Deprecated since version 2017: In the version of RAMSES after RUM 2017 (PR #284 and after), new blocks were introduced instead of one large `&PHYSICS_PARAMS`:

The block named `&PHYSICS_PARAMS` used to contain the parameters related to physical models.

Variable name	Fortran type	Default value	Description
`units_density`	`real`	1.0	Density unit in cgs (only for `cosmo=.false.`, requires G=1)
`units_time`	`real`	1.0	Time unit in cgs (only for `cosmo=.false.`, requires G=1)
`units_length`	`real`	1.0	Length unit in cgs (only for `cosmo=.false.`, requires G=1)
`cooling`	`boolean`	`.false.`	Enable gas atomic & metal cooling
`isothermal`	`logical`	`.false.`	Enable equation of state for gas (heating and cooling disabled if `.true.`)
`metal`	`logical`	`.false.`	Enable metals advection (requires 1 hydro variable)

Variable name	Fortran type	Default value	Description
`haardt_madau`	`logical`	`.false.`	Enable UV background
`self_shielding`	`logical`	`.false.`	Enable self-shielding for densities above 0.01 g.cm^-3
`smbh`	`logical`	`.false.`	Enable super massive black holes for sink particles
`agn`	`logical`	`.false.`	Enable AGN feedback from super massive black holes
`neq_chem`	`logical`	`.false.`	Enable non-equilibrium chemistry
`ir_feedback`	`logical`	`.false.`	Enable IR feedback from accreting sink particles
`sf_virial`	`logical`	`.false.`	Enable turbulent star formation prescriptions (requires 1 hydro variable)
`sf_compressive`	`logical`	`.false.`	Store and advect the compressive and solenoidal turbulent field in two different hydro variables (`sf_virial=.true.` only)
`sf_log_properties`	`logical`	`.false.`	Output gas properties of cells undergoing a star particle birth or supernova event
`delayed_cooling`	`logical`	`.false.`	Enable delayed cooling through passive energy scalar advection (requires 1 hydro variable)

Variable name	Fortran type	Default value	Description
`sf_imf`	`logical`	`.false.`	Model Initial Mass Function during thermal feedback events
`T2_star`	`real`	0.0	Typical ISM polytropic temperature (cooling floor if `isothermal=.false.`)
`g_star`	`real`	1.6	Typical ISM polytropic index (cooling floor if `isothermal=.false.`)
`jeans_ncells`	`real`	-1.0	Jeans polytropic equation of state (cooling floor if `isothermal=.false.`)
`T2max`	`real`	1D50	Temperature ceiling for gas heating (heating ceiling if `isothermal=.false.`)
`n_star`	`real`	0.1	Star formation density threshold in H/cc
`m_star`	`real`	-1.0	Star particle mass in units of mass_sph
`del_star`	`real`	2D2	Star formation density threshold in critical density (`cosmo=.true.` only)
`eps_star`	`real`	0.0	Star formation efficiency
`t_star`	`real`	0.0	Star formation time scale in Gyr (only used if `eps_star>0`)
`sf_trelax`	`real`	0.0	Relaxation time for star formation (`cosmo=.false.` only)
`sf_tdiss`	`real`	0.0	

Variable name	Fortran type	Default value	Description
			Dissipation timescale for subgrid turbulence in units of turbulent crossing time (`sf_virial=.true.` only)
`sf_model`	`integer`	3	Turbulent star formation model (`sf_virial=.true.` only)
`eta_sn`	`real`	0.0	Supernova mass fraction
`eta_ssn`	`real`	0.95	Single supernova ejected mass fraction (`sf_imf=.true.` only)
`t_sne`	`real`	10.0	Supernova blast time in Myr
`t_diss`	`real`	20.0	Dissipation timescale for supernova feedback in Myr (`delayed_cooling=.true.` only)
`yield`	`real`	0.0	Supernova metal yield
`mass_gmc`	`real`	0.0	Stochastic exploding GMC mass in solar mass
`kappa_IR`	`real`	0.0	IR dust opacity for supernova feedback
`f_ek`	`real`	0.0	Supernova kinetic energy fraction (only between 0 and 1)
`f_w`	`real`	0.0	Supernova mass loading factor (`f_ek>0` only)

Variable name	Fortran type	Default value	Description
`rbubble`	`real`	0.0	Supernova superbubble radius in pc (`f_ek>0` only)
`ndebris`	`integer`	1	Supernova debris particle number (`f_ek>0` only)
`mass_star_max`	`real`	120.0	Maximum mass of a star to undergo a supernova with `eta_ssn` efficiency in solar mass (`sf_imf=.true.` only)
`mass_sne_min`	`real`	10.0	Minimum mass of a star to undergo a supernova blast in solar mass (`sf_imf=.true.` only)
`J21`	`real`	0.0	UV flux at threshold in 10^21 units
`a_spec`	`real`	1.0	Slope of the UV spectrum
`z_ave`	`real`	0.0	Initial average metal abundance
`z_reion`	`real`	8.5	Reionization redshift (UV background disabled for higher redshifts)
`ind_rsink`	`real`	4.0	Number of cells defining the radius of the sphere where AGN feedback is active
`ir_eff`	`real`	0.75	Efficiency of the IR feedback on sink particles (`ir_feedback=.true.` only)

Poisson Parameters

This namelist, `&POISSON_PARAMS`, is used to specify runtime parameters for the Poisson solver. It is used only if `poisson=.true.` or `pic=.true.`

Solvers

Two different Poisson solvers are available in RAMSES: conjugate gradient (CG) and multigrid (MG). Unlike the CG solver, MG has an initialization overhead cost (at every call of the solver), but is much more efficient on very big levels with few "holes". The multigrid solver is therefore used for all coarse levels.

In addition, MG can be used on refined levels in conjuction with CG. The parameter `cg_levelmin` selects the Poisson solver as follows:

- Coarse levels are solved with MG
- Refined levels with l < `cg_levelmin` are solved with MG
- Refined levels with l >= `cg_levelmin` are solved with CG

Gravity types

Different sources of gravity can be added to the simulation. To activate self-gravity of the gas and particles, set `self_gravity=.true.`. Additional sources of gravity can be added by

- adding an analytical density distribution to the Poisson source term from which the gravitational potential is determined (see rho_ana.f90)
- adding an analytical force directly (see gravana.f90)

These two methods can be combined by setting both `gravity_rho_ana_type` and `gravity_force_ana_type`.

Variable name	Fortran type	Default value	Description
`gravity_type`	`integer`	0	(deprecated) Type of gravity force. Possible choices are: `gravity_type=0` self-gravity (Poisson solver); `gravity_type>0` analytical gravity vector; `gravity_type<0` self-gravity plus additional analytical density profile
`self_gravity`	`logical`	`.true.`	Switch on self-gravity (Poisson solver)
`gravity_rho_ana_type`	`integer`	0	Add gravity from an analytical density profile. This will add an analytical contribution to the Poisson source term. 0 = no additional contribution, 2 = a point mass, 3 = galactic disk (see rho_ana.f90)
`gravity_force_ana_type`	`integer`	0	Add an analytical gravitational force. 0 = no additional contribution, 1 = constant vector, 2 = a point mass, 3 = galactic disk (see gravana.f90)
`epsilon`	`real`	1e-4	Stopping criterion for the iterative Poisson solver: residual 2-norm should be lower than `epsilon` times the right hand side 2-norm.
`gravity_params`	`real array`	0.0,	(deprecated) Parameters used to define the analytical gravity field (routine `gravana.f90`) or the analytical mass density field (routine `rho_ana.f90`).

Variable name	Fortran type	Default value	Description
`gravity_rho_ana_params`	`real array`	0.0,	Parameters used to define the analytical mass density field (routine `rho_ana.f90`).
`gravity_force_ana_params`	`real array`	0.0,	Parameters used to define the analytical gravity field (routine `gravana.f90`)
`cg_levelmin`	`integer`	999	Minimum level from which the Conjugate Gradient solver is used in place of the Multigrid solver.
`cic_levelmax`	`integer`	0	Maximum level for CIC dark matter interpolation (default `cic_levelmax=nlevelmax`)

Clumpfinder (PHEW)

The block named `&CLUMPFIND_PARAMS` contains the parameters related to the built-in RAMSES clump finder (PHEW). The parameters are described only briefly, for more backround information read the PHEW paper.

Variable name, syntax, default value	Fortran type	Description
`ivar_clump=1`	`integer`	Control which density field is used for clump finding (1: gas density, 0: particle density)
`density_threshold=-1.d0`	`float`	Density threshold for the clump finder (code units)

Variable name, syntax, default value	Fortran type	Description
`rho_clfind=-1.d0`	`float`	Density threshold for the clump finder (g/cc). Not recommended - use `density_threshold` instead.
`n_clfind=-1.d0`	`float`	Density threshold for the clump finder (H/cc). Not recommended -use `density_threshold` instead.
`relevance_threshold=2.d0`	`float`	Relevance (peak-to-saddle ratio) threshold for a clump to be considered real (instead of noise).
`saddle_threshold=-1.d0`	`float`	Saddle density threshold for sub-structure merging (code units). A negative value turns sub-structure merging off (PHEW will only merge noise).
`mass_threshold=0.d0`	`float`	When set to a value > 0, the properties of those clumps/haloes with `mass > mass_threshold * particle_mass` are written to disk. `particle_mass` is the smallest strictly positive particle mass in the simulation. Setting this parameter does NOT affect the merging of noise/ clumps.
`age_cut_clfind=0.d0`	`float`	When set to a value > 0, only the stars with an age lower than `age_cut_clfind` are used by the clump finder. Stellar and DM particles are used otherwise.

Sink particles

The block named `&SINK_PARAMS` contains the parameters related to the sink particle implementation, which can be used for

- star formation (most recently Bleuler & Teyssier 2015)

- supermassive black hole evolution and AGN feedback (most recently Biernacki, Teyssier and Bleuler 2017)

Overview of parameters

Variable name	Fortran type	Default value	Description
`smbh`	`boolean`	`.false.`	Controls if sink behaves as a star or SMBH
`agn`	`boolean`	`.false.`	Controls if SMBH sink produces feedback
`create_sinks`	`boolean`	`.false.`	Specifies if sinks are formed with clump finder
`check_energies`	`boolean`	`.true.`	when flagging clumps for sink formation, check whether their gravitational energy is dominant
`nsinkmax`	`integer`	`2000`	Maximum number of sinks allowed to form
`mass_sink_direct_force`	`float`	`-1`	Mass in Msun above which sinks are treated with direct N-body solver
`ir_cloud`	`integer`	`4`	Radius of cloud region in unit of grid spacing
`ir_cloud_massive`	`integer`	`3`	Radius of massive cloud region in unit of grid spacing
`sink_soft`	`integer`	`2`	

Variable name	Fortran type	Default value	Description
			Gravitational softening length in dx at levelmax for “direct force” sinks
`n_sink`	`float`	`.false.`	Sink (as a star) formation density threshold in H/cc
`rho_sink`	`float`	`.false.`	Sink (as a star) formation density threshold in g/cc
`d_sink`	`float`	`.false.`	Sink (as a star) formation density threshold in code units
`clump_core`	`boolean`	`.false.`	Trims the clump (for sinks as stars only)
`mass_sink_seed`	`float`	`0.0`	Dynamical mass of sink seed in Msun
`mass_smbh_seed`	`float`	`0.0`	Accretion mass of sink seed in Msun, if 0, then dynamical and accretion masses are equivalent
`mass_halo_AGN`	`float`	`1e10`	Mass of a halo in which AGN sinks are seeded
`mass_clump_AGN`	`float`	`1e10`	Mass of a clump in which AGN sinks are seeded
`accretion_scheme`	`string`	`none`	Accretion scheme: none, bondi, threshold

Variable name	Fortran type	Default value	Description
`eddington_limit`	`boolean`	`.false.`	Controls if accretion rate should be limited by Eddington rate
`acc_sink_boost`	`float`	`1`	Value of boost factor in Bondi velocity (-1 to depend on density)
`boost_threshold_density`	`float`	`0.1`	Threshold density for boosting, typically the same as n_star; in H/cc
`bondi_use_vrel`	`boolean`	`.false.`	Excludes relative velocity between gas and sink from the accretion calculations
`mass_merger_vel_check_AGN`	`float`	`-1`	Mass above which the check for two sinks' binding energy is applied, in Msun
`merging_timescale`	`float`	`-1`	Time during which sinks are considered for merging (non-SMBH only)
`verbose_AGN`	`boolean`	`.false.`	Controls verbosity of AGN in the log file
`AGN_fbk_mode_switch_threshold`	`float`	`0.1`	Controls the AGN feedback switching
`AGN_fbk_frac_ener`	`float`	`1.0`	Controls what fraction of energy goes into thermal feedback

Variable name	Fortran type	Default value	Description
`T2_min`	`float`	`1e7`	Minimum temperature to which AGN blast should heat the gas in K
`T2_AGN`	`float`	`1e12`	Temperature of AGN blasts in K - feedback efficiency
`AGN_fbk_frac_mom`	`float`	`1.0`	Controls what fraction of energy goes into kinetic feedback
`epsilon_kin`	`float`	`1.0`	Kinetic feedback coupling efficiency
`kin_mass_loading`	`float`	`100.`	Kinetic feedback mass loading
`cone_opening`	`float`	`180.`	Opening angle of the cone through which momentum feedback proceeds
`nlevelmax_sink`	`integer`	`0`	Allow sinks to form on a level that is lower than what is set by nlevelmax. This is useful in zoom simulations where nlevelmax is set to a high value but in the first phase refinement is limited to only a few levels. nlevelmax_sink should be set to the maximum allowed refinement level.
`cloud_pts_check`	`boolean`	`.false.`	Useful when there is a large number of sinks (>1000) with a large number of cores. When

Variable name	Fortran type	Default value	Description
			true: optimizes the creation of cloud particles.

Example set of parameters for cosmological simulations with AGN feedback

The example listed below by no means can fit everyone's needs, but can serve as a minimum starting example.

```
#!fortran
&SINK_PARAMS
! General switches
smbh=.true.                  ! turns sinks into SMBH
agn=.true.                   ! enables AGN feedback
create_sinks=.true.          ! enables formation of new sink particles
mass_sink_direct_force=1.0   ! minimum mass of sink to treat it with
direct solver, in M_sun

! Seeding masses
mass_sink_seed=1.0d6         ! dynamical mass of sink particle
mass_smbh_seed=1.0d6         ! accretion mass of sink particle
mass_halo_AGN=5.d10          ! minimum mass of PHEW halo in which a
sink is seeded
mass_clump_AGN=1.d9          ! minimum mass of PHEW clump in which a
sink is seeded

! Accretion
accretion_scheme='bondi'     ! selects Bondi accretion as accretion
mode
eddington_limit=.true.       ! enables Eddington limit on accretion
acc_sink_boost=-1            ! boosts accretion according to
Booth&Schaye 2009
boost_threshold_density=0.1 ! threshold density for boosting,
typically the same as n_star; in H/cc
bondi_use_vrel=.false.       ! excludes relative velocity between gas
and sink from the accretion calculations

! Merging
mass_merger_vel_check=1e8    ! sum of sinks' masses for which
velocities are checked upon merging to determine if the system is
bound

! Feedback
```

```
T2_min=0                    ! if feedback can heat the gas to this
temperature then deposit it; here deposits at every fine step
T2_AGN=0.15d12              ! thermal feedback efficiency
AGN_fbk_frac_ener=1.0       ! controls what fraction of energy goes
into thermal feedback, here 100%
AGN_fbk_frac_mom=0.0        ! controls what fraction of energy goes
into momentum feedback, here 0%
AGN_fbk_mode_switch_threshold=1d-2 ! switches between thermal (above)
and momentum modes for this ratio of Bondi-to-Eddington
epsilon_kin=1.              ! momentum feedback efficiency
kin_mass_loading=100.       ! mass loading factor of momentum
feedback
cone_opening=90.            ! opening angle of the cone in which
momentum feedback is deposited (180. means full sphere)
/
```

Notes:

- if `agn=.false.` all feedback settings are disregarded
- in order to seed the sink *both* `mass_halo_AGN` and `mass_clump_AGN` have to be satisfied (as well as condition of only one sink per halo and minimum density of the halo - at least star forming)
- currently the only other accretion mode besides `bondi` is `none`; please feel free to add more
- to not boost accretion, set `acc_sink_boost=0.0`
- it is *highly advised* to use `T2_min=0.0` in order to have all feedback modes on the finest timestep
- if `chi_switch=0.0`, then the initial values of `AGN_fbk_frac_*` will be used throughout the simulation

Movies

The block named `&MOVIE_PARAMS` contains the parameters related to the movies. One can produce up to 5 movies, which center at different parts of the simulation box and have various camera behavior (see example blocks in the bottom).

The movie routine is very useful for creating on the fly visualizations of a simulation with a small time step between frames, without having to write and process many huge snapshots for that. For turning the binary images created by this routine into a movie that can be watched on any media player, check out RAM.

Variable name, syntax, default value	Fortran type	Description
`movie`	`boolean`	Turns movies on and off
`imov`	`integer`	Number of starting frame
`imovout`	`integer`	Total number of frames
`tstartmov`	`float`	Start time of the movie
`astartmov`	`float`	Start aexp of the movie
`tendmov`	`float`	End time of the movie
`aendmov`	`float`	End aexp of the movie
`levelmax_frame`	`integer`	Maximum level of the frame
`nw_frame`	`integer`	Width of frame in px
`nh_frame`	`integer`	Height of frame in px
`xcentre_frame`	5* `float, float, float, float`	Four (**for each projection**) coordinates for z position of frame centre [code units] - spline coefficients
`ycentre_frame`	5* `float, float, float, float`	Four (**for each projection**) coordinates for y position of frame centre [code units] - spline coefficients

Variable name, syntax, default value	Fortran type	Description
`zcentre_frame`	5* `float, float, float, float`	Four (**for each projection**) coordinates for z position of frame centre [code units] - spline coefficients
`deltax_frame`	`float, float`	Two coordinates for x delta of frame [code units]
`deltay_frame`	`float, float`	Two coordinates for y delta of frame [code units]
`deltaz_frame`	`float, float`	Two coordinates for z delta of frame [code units]
`zoom_only_frame`	5* `boolean`	Consider only particles in the zoom region of cosmological runs
`proj_axis`	5* `character`	Letter code of projection axis; x, y, z - maximum 5 line of sights
`movie_vars_txt`	`strings`	Determines which variables to save - `dens`, `temp`, `pres`, `vx`, `vy`, `vz`, `varX`, `FpX` (RT-only), `stars` (star particles), `dm` (dark matter particles), `lum` (stellar luminosity)
`theta_camera`	5* `float`	Azimuthal angle of the camera with respect to the line of sight [degrees]
`phi_camera`	5* `float`	Polar angle of the camera with respect to the line of sight [degrees]
`dtheta_camera`	5* `float`	Azimuthal angle rotation completed between `tstartmov` and `tendmov` [degrees]

Variable name, syntax, default value	Fortran type	Description
`dphi_camera`	5* `float`	Polar angle rotation completed between `tstartmov` and `tendmov` [degrees]
`focal_camera`	5* `float`	Distance of the focal plane of the camera [code units]. Camera distance is set to `boxlen` and `focal_camera` is set to `boxlen` by default.
`dist_camera`	5* `float`	Distance of the camera [code units]. Camera distance is set to `boxlen` and `focal_camera` is set to `boxlen` by default.
`ddist_camera`	5* `float`	Motion of the camera between `tstartmov` and `tendmov` [code units]
`perspective_camera`	5* `boolean`	Perspective corrections for the projected cells
`shader_frame`	5* `strings`	Shader applied for the leaf cells projection [`square` , `sphere` , `cube`]
`ivar_frame`	5* `integer`	Index of hydro variable to use for selecting cells to project (default=-1)
`varmin_frame`	5* `integer`	Only project cells within `varmin_frame<uold(*,ivar_frame)<varmax_frame` . Density is expressed in cm^-3, velocities in km/s, temperature in K/mu, all other hydro variables in code units.

Variable name, syntax, default value	Fortran type	Description
`varmax_frame`	5* `integer`	Only project cells within `varmin_frame<uold(*,ivar_frame)<varmax_frame`. Density is expressed in cm^-3, velocities in km/s, temperature in K/mu, all other hydro variables in code units.
`method_frame`	5* `strings`	Available projection methods: `mean_mass` (default), `mean_dens`, `mean_vol`, `sum`, `min`, `max`

Examples:

- cosmological simulation

```
#!fortran
&MOVIE_PARAMS
movie=.true.
imov=0
aendmov=1.
imovout=1000
nw_frame=1920
nh_frame=1080
levelmax_frame=18
xcentre_frame=0.49944825,-0.00391524,0.02661462,-0.01714339
ycentre_frame=0.49512072,0.00498905,-0.01826436,0.01097619
zcentre_frame=0.483284613,0.0246328776,-0.0149075526,-1.06750114e-04
deltax_frame=0.0,0.001
deltay_frame=0.0,0.0005625
deltaz_frame=0.0,0.001
proj_axis='z'
movie_vars_txt='dm','stars','temp','dens','var6'
/
```

This will result in 1000 frames 1920x1080 (Full HD) between starting redshift and z=0. Values for `xcentre_frame`, `ycentre_frame` and `zcentre_frame` come from fitting the spline coefficients with `utils/py/get_spline_coeffs.py` (it requires previous run of the same setup with clump finder; e.g. DM-only). Size of the frame will be reflecting physical units (e.g. delta_x=deltax_frame[0]+deltax_frame[1]/a; where a is the explansion factor). Frames are going to

be projected along the z axis and projections of dark matter density, stellar density, temperature, gas density and var6 (extra variable, here metallicity) will be saved.

- non-cosmological

```
#!fortran
&MOVIE_PARAMS
movie=.true.
imov=0
tendmov=10.
imovout=1000
nw_frame=1920
nh_frame=1080
levelmax_frame=14
xcentre_frame=5.0,0.,0.,0.,5.0,0.,0.,0.
ycentre_frame=5.0,0.,0.,0.,5.0,0.,0.,0.
zcentre_frame=5.0,0.,0.,0.,5.0,0.,0.,0.
deltax_frame=1.0,0.,1.0,0.
deltay_frame=1.0,0.,1.0,0.
deltaz_frame=1.0,0.,1.0,0.
proj_axis='zy'
movie_vars_txt='dm','stars','temp','dens','var6'
/
```

Same variables as for the cosmological run are going to be saved with the same resolution. Here, the camera is always centred at the centre of the box (assuming boxlen=10) and encompasses volume of 10% x 10% x 10% of the box. Tendmovie is set in the code units of time.

Turbulence driving

The block named `&TURB_PARAMS` contains the parameters related to the turbulence driving. Originally implemented by Andrew Mcleod.

General principle[1]

The model used for turbulent driving in Ramses is a generalization of the Ornstein-Uhlenbeck process. The force is computed in Fourier space and then applied to the gas. The evolution of the Fourier modes $\vec{\hat{f}}$ of the force was obtained via the resolution of the following stochastic differential equation:

$$\mathrm{d}\vec{\hat{f}}(\vec{k}, t) = - \vec{\hat{f}}(\vec{k}, t)\dfrac{\mathrm{d}t}{T} + F_0(\vec{k})\vec{P_\chi}\left(\vec{k}\right) \mathrm{d}\vec{W}_t.$$

In this equation, $\mathrm{d}t$ is the timestep for integration and T is the autocorrelation timescale. The perturbation $\mathrm{d}\vec{W_t}$ is a small vector randomly chosen following the Wiener process. The power spectrum F_0 is a way to select the relevant mode.

Example

A parabolic power spectrum between $k=1$ and $k=3$:

$$\begin{split} F_0(\vec{k}) = \begin{cases} 1 - \left(\dfrac{\vec{k}}{2\pi} - 2\right)^2\text{ if } 1 < \dfrac{\vert k \vert}{2\pi} < 3 \\ 0 \text{ if not.} \end{cases} \end{split}$$

The projection operator $\vec{P_\chi}$ is a weighted sum of the components of the Helmholtz decomposition of compressive versus solenoidal modes:

$$ \vec{P_\chi}(\vec{k}) = (1 - \chi) \vec{P}^{\perp}(\vec{k}) + \chi \vec{P}^{\parallel}(\vec{k}) \;, $$
with $\vec{P}^{\perp}$ and $\vec{P}^{\parallel}$ the projection operators respectively perpendicular and parallel to $\vec{k}$ and χ the compressive driving fraction. This compressive driving fraction applies only to the driving and is different from the compressive ratio measured in the velocity field. The forcing field $\vec{f}(\vec{x}, t)$ is then computed from the Fourier transform:

$$ \vec{f}(\vec{x}, t) = g(\chi) f_{\mathrm{rms}} \int\vec{\hat{f}}(\vec{k}, t) e^{i\vec{k}\cdot x} d^3\vec{k}\;. $$
The parameter f_{rms} is directly linked to the power injected by the turbulent force into the simulation. The $g(\chi)$ factor is an empirical correction so that the resulting time-averaged root-mean-square of the power of the Fourier modes is equal to f_{rms}, independently of the compressive fraction χ.

Overview of parameters

Variable name	Fortran type	Default value	Notation	Description
`turb`	`boolean`	`.false.`		Turn on or off driving
`turb_seed`	`integer`	`-1`		Random number generator seed. -1 = random

Variable name	Fortran type	Default value	Notation	Description
`turb_type`	`integer`	`1`		How the driving changes over time. 1=driven evolving, 3=decaying
`instant_turb`	`boolean`	`.true.`		Generate initial turbulence before start
`comp_frac`	`float`	`0.3333`	\(\chi\)	The weight of compressive over solenoidal modes
`turb_T`	`float`	`1`	\(T\)	Turbulent velocity auto-correlation time in code units.
`turb_Ndt`	`integer`	`100`	\(T/dt\)	Number of timesteps per auto-correlation time
`turb_rms`	`float`	`1`	\(f_\mathrm{rms}\)	Root-mean-square turbulent forcing in code units.

Variable name	Fortran type	Default value	Notation	Description
`turb_min_rho`	`float`	`1d-50`		Minimum density for turbulence. Not forcing is added onto cellswith a density less than this value.
`forcing_power_spectrum`	`string`	`parabolic`	\(F_0\)	Power spectrum type of the forcing, which describes the relative strength of individual modes. Options are: power_law, parabolic, konstandin

[1]
adapted from Brucy et al. 2024.

Monte Carlo tracer particles

The block named `&TRACER_PARAMS` contains the parameters related to the Monte Carlo tracer particles, see Cadiou et al 2018

Overview of parameters

Variable name	Fortran type	Default value	Description
`MC_tracer`	`boolean`	`.false.`	Activate MC tracers

Variable name	Fortran type	Default value	Description
`tracer_feed`	`string`	`none`	Filename to read the tracer from
`tracer_feed_fmt`	`string`	`ascii`	Format of the input (ascii,binary or inplace)
`tracer_mass`	`float`	`-1.0`	Mass of the tracers, used for outputs and seed
`tracer_first_balance_levelmin`	`integer`	`-1`	Set to >0 to add more weight on level finer than this
`tracer_first_balance_part_per_cell`	`integer`	`0`	Typical initial number of parts per cell

RHD Params

This sets of parameters, contained in the namelist block `&RT_PARAMS`, sets radiative properties for RAMSES RHD runs. Note that the number of photon groups (`NGROUPS`) is a compilation parameter, to be set in the Makefile.

For detailed descriptions of the concepts described here, see the following papers:

- [1] RAMSES-RT: radiation hydrodynamics in the cosmological context
- [2] A scheme for radiation pressure and photon diffusion with the M1 closure in RAMSES-RT
- [3] Galaxies that Shine
- [4] A simple model for molecular hydrogen chemistry coupled to radiation hydrodynamics

and the documentation page about RHD simulations.

Variable name, syntax, default value	Fortran type	Description
`X=0.76`	`real`	Hydrogen mass fraction.
`Y=0.24`	`real`	Helium mass fraction.
`isH2=.false.`	`logical`	Include molecular hydrogen? See [4]
`isHe=.true.`	`logical`	Include helium ionization?
`rt_flux_scheme='glf'`	`character(len=10)`	Intercell flux function for the advection of radiation (see §3.2 in [1]). Use either 'glf' or 'hll'. Note that only the GLF flux function is compatible with the inclusion of trapped IR radiation (see §2.2 in [2]).
`hll_evals_file=''`	`character(len=128)`	Eigenvalues file, necessary only for the HLL intercell flux. Can also be set by environment variable `RAMSES_HLLFILE`. Such a file can be found with the source code (rt/hll_evals.list).
`rt_c_fraction=1.`	`real`	Reduced light speed fraction, for keeping a manageable timestep-length (see §4.1 in [1]). The default corresponds to a full light speed.
`rt_courant_factor=0.8`	`real`	Courant factor for photon advection between cells.

Variable name, syntax, default value	Fortran type	Description
`rt_nsubcycle=1`	`integer`	Maximum number of RT-steps during one hydro/gravity/etc timestep.
`rt_smooth=.true.`	`logical`	Smooth out the operator splitting of photon advection and thermochemistry by incrementally updating the advected quantities in the chemistry. Usually speeds up the calculation. See §4.4 in [1].
`rt_otsa=.true.`	`logical`	Assume the on-the-spot approximation, such that straight-to-ground recombinations in H and He do not emit ionising radiation (i.e. emitted photons are assumed to be instantly absorbed in the same cell – see §3.3.2 in [1]).

Variable name, syntax, default value	Fortran type	Description
`rt_is_init_xion=.false.`	`logical`	Set to initialize H and He ionization fractions from local photoionisation equilibrium (PIE), using the temperature, density, and radiation in each cell. Note that this is done by default (even if `rt_is_init_xion=.false.`) when starting simulations from scratch – this parameter should only be used for resetting the ionisation fractions in *restarts*, which can be useful when postprocessing hydro simulations with radiative transfer.
`upload_equilibrium_x=.false.`	`logical`	Set PIE ionization fractions when merging cells, instead of taking children averages.
`rt_is_outflow_bound=.false.`	`logical`	Force outflow boundary for RT on all box sides, regardless of how boundaries are defined for hydrodynamics. By default, boundaries are the same for RT and hydrodynamics.
`rt_Tconst=-1`	`real`	Constant temperature, in Kelvin, to assume for all temperature-dependent interaction rates (to run the first Iliev test). The default negative value means the actual cell temperature is used.

Variable name, syntax, default value	Fortran type	Description
`rt_output_coolstats=.false.`	`logical`	Write thermochemistry statistics to the standard output.
`iPEH_group=0`	`int`	Photon group used for photoelectric heating (default no group)
`heat_unresolved_HII=0`	`int`	Subgrid model for unresolved HII regions (1==thermal heating, 2==non-thermal heating with NENER)
==========	==========	**Stellar emission parameters**
`rt_star=.false.`	`logical`	Turn on photon emission from stellar particles. If `rt_star=.true.` and `rt=.true.` (in `RUN_PARAMS`), RT turns on when the first stellar particles are created in a simulation.
`sed_dir=”`	`character(len=128)`	Directory containing spectral energy distribution (SED) model for the stellar emission (see Appendix B in [1]). This can also be set by the environment variable `RAMSES_SED_DIR`.
`sedprops_update=-1`	`integer`	Frequency (per coarse timestep) of photon group updates according to SED model (see Appendix B in [1]). The default

Variable name, syntax, default value	Fortran type	Description
		value of -1 means that the update is never done.
`nSEDgroups=NGROUPS`	`integer`	Number of photon groups dedicated to stellar emission, and relevant only if `rt_star=.true.`. These are the first photon groups: the last `NGROUPS-nSEDgroups` do not carry direct stellar radiation (are e.g. for a propagated UV background).
`SED_isEgy=.false.`	`logical`	Energy-conserving stellar emission when using a SED model. The default is photon number conserving within each photon group. With stellar particles of different ages and metallicities, the particle emission cannot be both energy conserving and number conserving at the same time, because the individual particles' spectral shapes differ, and we must use 'average' shapes for the photon groups. The luminosity of a particle can either be correct (against the SED model) in terms of energy but not exactly by photon number, by taking the energy luminosity, or in terms of photon number but not exactly energy, by taking the photon number luminosity.

Variable name, syntax, default value	Fortran type	Description
`rt_esc_frac=1.`	`real`	'Escape fraction' of photons from stellar particles, essentially just a multiplication factor for the particle emission.
`convert_birth_times=.false.`	`logical`	Convert birth times of stellar particles from conformal to proper time. By default the birth times are stored as proper in RHD simulations, for faster estimation of the stellar luminosity as a function of age. However, when postprocessing non-RHD simulations, this parameter is needed, since non-RHD simulations store the birth time in conformal units.
==========	==========	**UV background parameters**
`rt_UVsrc_nHmax=-1.`	`real`	Hydrogen density threshold (number per cubic cm) for non-homogeneous UV emission and propagation. Default value corresponds to no UV propagation.

Variable name, syntax, default value	Fortran type	Description
`nUVgroups=0`	`integer`	Number of photon groups dedicated to the propagated UV background, and relevant (and set to `NGROUPS`) only if `rt_UVsrc_nHmax>0.`. These are the last photon groups: the first `NGROUPS-nUVgroups` do not carry the UV background (are e.g. for stellar radiation).
`uv_file=''`	`character(len=128)`	File containing UV model, which is needed for a propagated UV background (`rt_UVsrc_nHmax>0`), or a homogeneous one (`haardt_madau=.true.` in `$PHYSICS_PARAMS`). This can also be set by the environment variable RAMSES_UV_FILE.
`==========`	`==========`	**Radiation pressure and IR radiation parameters**
`rt_isIR=.false.`	`logical`	Assume first photon group represents IR photons, in local thermal equilibrium (LTE) with dust (which is assumed to scale linearly in content with the gas metallicity). All other photon groups are 'reprocessed' locally and energy-conservatively into the IR group when interacting with dust via their `kappaAbs` parameter.

Variable name, syntax, default value	Fortran type	Description
`rt_isIRtrap=.false.`	`logical`	Apply trapping of IR photons in optically thick gas, in effect correctly modelling the IR propagation when the mean free path becomes unresolved (see §2.4.2 in [2]). With this set to `.true.`, the code must be compiled with a dedicated non-thermal energy variable, with `NENER>0` and a corresponding increment in `NVAR` (e.g. set `NENER=1` and increase `NVAR` by one, and recompile).
`is_kIR_T=.false.`	`logical`	Use functions for the dust absorption and scattering opacities, such that they scale with the radiation temperature squared (eq. 79 in [2]), with a normalisation set by the `kappaAbs` and `kappaSc` parameters in `$RT_GROUPS`. The default is to use constant opacities (set by the same parameters).
`rt_isoPress=.false.`	`logical`	Use the 'reduced flux approximation' described in [3] (§2.7), where the radiation in each cell is assumed to be fully directional.
`rt_pressBoost=1.`	`real`	Multiplication factor to boost or reduce radiation pressure on gas and dust from the default.

Variable name, syntax, default value	Fortran type	Description
`rt_vc=.false.`	`logical`	Include relativistic corrections for the Lorentz boost and work done by photons on the gas (see Appendix A and B in [2]).
==========	==========	**RT refinement parameters**
`rt_err_grad_cn=-1.0` `rt_err_grad_xHI=-1.0` `rt_err_grad_xHII=-1.0`	`real`	Discontinuity-based strategy: photon flux and ionization fraction gradients above which a cell is refined.
`rt_floor_cn=1d-10` `rt_floor_xHI=1d-10` `rt_floor_xHII=1d-10`	`real`	Discontinuity-based strategy: photon flux and ionization fraction floor below which gradients are ignored.
`rt_refine_aexp=-1.0`	`real`	Cosmological expansion at which to turn on RT refinement strategies.
==========	==========	**RT source regions**
`rt_nsource=0`	`integer`	Number of independent source (photon emission) regions in the computational box.
`rt_source_type='square'`	`character(len=128) array`	Geometry defining each source region. 'square' defines a generalized ellipsoidal shape with photons injected everywhere inside, 'shell' defines a finite width spherical shell into which photons are

Variable name, syntax, default value	Fortran type	Description
		injected, and 'point' defines a point source.

`rt_src_x_center=0.0`
`rt_src_y_center=0.0`
`rt_src_z_center=0.0`
	`real arrays`	Coordinates (0 to boxlen) of the center of each source region.

`rt_src_length_x=0.0`
`rt_src_length_y=0.0`
`rt_src_length_z=0.0`
	`real arrays`	Sizes in all directions of each source region. If a spherical shell is used, rt_src_length_x and rt_src_length_y represent outer and inner radius.

`rt_exp_source=2.0`	`real array`	Exponents defining the norm used to compute distances for the generalized ellipsoid. `rt_exp_source=2` corresponds to a spheroid, `rt_exp_source=1` to a diamond shape, `rt_exp_source=10` to a perfect square.
`rt_src_group=1`	`integer array`	Photon groups into which photons are emitted in each source region (1 .le. rt_src_group .le. M, where M is the number of groups).

`rt_n_source=0.0`
`rt_u_source=0.0`
`rt_v_source=0.0`
`rt_w_source=0.0`
	`real arrays`	Injection rates, in cgs units, into photon densities and fluxes.

`==========`	`==========`	**RT initialisation regions**

Variable name, syntax, default value	Fortran type	Description
`rt_nregion=0`	`integer`	Number of independent initial radiation regions in the computational box.
`rt_region_type='square'`	`character(len=128) array`	Geometry defining each initial radiation region. 'square' defines a generalized ellipsoidal shape with photons initialised everywhere inside, 'shell' defines a finite width spherical shell in which photons are initialised, and 'point' defines a point flash.
`rt_reg_x_center=0.0` `rt_reg_y_center=0.0` `rt_reg_z_center=0.0`	`real arrays`	Coordinates (0 to boxlen) of the center of each initialisation region.
`rt_reg_length_x=0.0` `rt_reg_length_y=0.0` `rt_reg_length_z=0.0`	`real arrays`	Sizes in all directions of each initialisation region. If a spherical shell is used, rt_src_length_x and rt_src_length_y represent outer and inner radius.
`rt_reg_source=2.0`	`real array`	Exponents defining the norm used to compute distances for the generalized ellipsoid. `rt_reg_source=2` corresponds to a spheroid, `rt_reg_source=1` to a diamond shape, `rt_reg_source=10` to a perfect square.
`rt_reg_group=1`	`integer array`	

Variable name, syntax, default value	Fortran type	Description
		Photon groups for which photons are initialised in each region (1 .le. rt_reg_group .le. M, where M is the number of groups).

`rt_n_region=0.0`
`rt_u_region=0.0`
`rt_v_region=0.0`
`rt_w_region=0.0`
	`real arrays`	Values in each region, in cgs units, for photon densities and fluxes.

RHD Groups

This set of parameters, contained in the namelist block `&RT_GROUPS`, sets radiation group properties for RAMSES RHD runs. Note that the number of photon groups (`NGROUPS`) is a compilation parameter, to be set in the Makefile. The default settings are HI, HeI and HeII ionizing photon groups with ionisation cross sections and energies derived from a blackbody spectrum with an effective temperature of 10^5 Kelvin.

For detailed descriptions of the concepts described here, see the following papers:

- [1] RAMSES-RT: radiation hydrodynamics in the cosmological context
- [2] A scheme for radiation pressure and photon diffusion with the M1 closure in RAMSES-RT

and the documentation page about RHD simulations.

Variable name, syntax, default value	Fortran type	Description
`groupL0 = 13.60,24.59,54.42`	`real array`	Lower energy boundaries, in eV, of each photon group (see §2 in [1]). Used for calculating SED model emission from stellar particles.

Variable name, syntax, default value	Fortran type	Description
`groupL1 = 24.59,54.42,0.0`	`real array`	Upper energy boundaries, in eV, of each photon group. A value of 0.0 is used to represent infinity.
`group_egy = 18.85,35.079,65.666`	`real array`	Average photon energies (eV) for each group. These can either be set manually or left to RAMSES to derive from SED models (with `SEDprops_update>0`).
`group_csn(1,:) = 3.0d-18,0.0,0.0` `group_csn(2,:) = 5.7d-19,4.5d-18,0.0` `group_csn(3,:) = 7.9d-20,1.2d-18,1.1d-18`	`real matrix`	2d matrix representing average ionisation cross sections (cm^2) between each group (first index) and species (second index). These can either be set manually or left to RAMSES to derive from SED models (with `SEDprops_update>0`).
`group_cse(1,:) = 2.8d-18,0.0,0.0` `group_cse(2,:) = 5.0d-19,4.1d-18,0.0` `group_cse(3,:) = 7.4d-20,1.1d-18,1.0d-18`	`real matrix`	2d matrix representing energy weighted average ionisation cross sections (cm^2) between each group (first index) and species (second index). These can either be set manually or left to RAMSES to derive from SED models (with `SEDprops_update>0`).
`spec2group = 1,2,3`	`integer array`	Determines, for each recombining species (HII, HeII, HeIII) which photon group the recombination photons are injected into. Note that recombination emission must be

Variable name, syntax, default value	Fortran type	Description
		activated with `rt_otsa=.false.` in `$RT_PARAMS`.
==========	==========	**Radiation pressure and IR radiation parameters**
`kappaAbs = 0.0`	`real array`	Dust absorption (Planck) opacity factor. The real opacity scales with the local metallicity and if `is_kIR_T=.true.`, the IR opacity also varies with the local gas temperature.
`kappaSc = 0.0`	`real array`	Scattering (Rosseland) opacity factor, only used for the IR photon group (which is the first group). The real opacity scales with the local metallicity and if `is_kIR_T=.true.`, it also varies with the local gas temperature.

Advanced simulations

In this section, we describe in more detail how to configure RAMSES to perform more advanced simulations.

Changing the source code

In order to perform more customised simulations, the source code of RAMSES can be changed/extended.

There are two different ways in which this task can be managed:

1. Create an appropriately called git branch and all the development changes will be reflected in the code commits. Great variety of resources can be found on this, but see e.g. this brief introduction. This is the **recommended** way of patching RAMSES.

2. Create a patch directory and store there your changed files. This is presently **disfavoured**, but is discussed here due to historical reasons.

Patch directory

After storing all the changed source files in the directory of choice you have to pass this path to Makefile - adjust the `PATCH` variable (make sure there is no trailing space!).

During the compilation, `ramses/utils/scripts/cr_write_patch.sh` will be invoked. This script prepares a Fortran source code file which contains all the patches as a long string. This is written in each RAMSES output directory for later reference (`patches.txt`).

WARNING: Changes in RAMSES source code that are uncommitted or not in patch directory are not being tracked!

Radiation Hydrodynamics in RAMSES

Radiation hydrodynamics are implemented in RAMSES, as described in those papers:

- RAMSES-RT: radiation hydrodynamics in the cosmological context
- A scheme for radiation pressure and photon diffusion with the M1 closure in RAMSES-RT
- A simple model for molecular hydrogen chemistry coupled to radiation hydrodynamics

Compiling for RHD runs

An example makefile for an RHD compilation, `Makefile.rt`, is included under `ramses/bin`. To activate radiative transfer, you must compile with the flag `-DRT`, and some RHD specific .f90 files must be included in the compilation (all of which is included in the `Makefile` example).

For a run with only atomic hydrogen (i.e. ionisation species HI and HII), you should use `NIONS=1`. For including molecular hydrogen, add one to `NIONS`, and for including helium ionization, add two to `NIONS`. The number of hydro variables needs to increase accordingly, but this is done automatically in `Makefile.rt`. You also set the number of photon groups in the Makefile with the `NGROUPS` parameter. When running with IR radiation trapping (`rt_isIRtrap=.true.`), you must also compile with a dedicated non-thermal energy variable, by setting `NENER=1` in `Makefile.rt` (`NVAR` is incremented automatically in the `Makefile`).

RHD outputs

The radiation field variables are written separately in each output to files named `rt_XXXXX.outYYYYY`, where XXXXX is the output number and YYYYY the cpu number. The naming convention and format of the files is exactly the same as for the hydro variables. For each photon group, there are four cell variables, `c_r*N`, `Fx`, `Fy`, `Fz`, where `c_r` is the reduced light speed, `N` the photon number density, and the rest are the photon number flux in the x, y, and z directions. The factors for converting those to cgs units are stored in a file named `info_rt_XXXXX.txt` in each output directory (`unit_np` and `unit_pf`), along with the reduced light speed and the photon group properties. The non-thermal energy (trapped IR radiation pressure) is stored in runtime next after the thermal pressure, but in the output it comes just before.

RHD Runtime parameters

Radiative transfer is activated by setting `rt=.true.` in `&RUN_PARAMS`. For RT post-processing, set also `static_gas=.true.` in the same namelist. Note that this is not sufficient to turn on the advection of radiation – this is done in the code (with `rt_advect=.true.`) only if sources of radiation (stars, gas, or idealised sources) are detected in the run.

If `rt=.true.`, non-equilibrium thermochemistry of hydrogen (and optionally helium) is used instead of the default equilibrium chemistry in RAMSES. The equilibrium chemistry can also be turned on without RHD, by setting `neq_chem=.true.` in `&PHYSICS_PARAMS` (but you must still compile with `Makefile.rt`, with `NGROUPS=0`).

For RHD runs, there are two additional dedicated namelists:

- RT_PARAMS
- RT_GROUPS

Generation of spectral energy distribution (SED) tables for RHD simulations

The (metallicity and age dependent) radiative luminosity of stellar population particles and photon group properties are calculated on-the-fly in RAMSES runs using SED tables. RAMSES-readable tables can be generated from Starburst99 and BC03 formats using a python utility found in `utils/py/sed_utils.py`. The generated files are linked to the RAMSES run with the `sed_dir` parameter in the `&RT_PARAMS` namelist.

Particle Unbinding

Quickstart Guide

To activate particle unbinding, you need to set two runtime parameters in the `&RUN_PARAMS` namelist (in addition to whatever you have in there):

```
&RUN_PARAMS
clumpfind=.true.
unbind=.true.
/
```

The code will create `output_XXXXX/unbinding_XXXXX.outYYYYY` files following the same logic as the other particle output files `output_XXXXX/part_XXXXX.outYYYYY` files. They will contain the associated clump ID of each particle. In only works when particles, i.e. dark matter, is present in your simulation.

Important notes

- You can't do particle unbinding without doing clump/halo finding.
- Some parts of the code (e.g. binning particles in mass profiles of halos) rely on consistent floating-point operations. The (intel) fortran compiler however doesn't necessarily use value-safe optimisations, which may lead to **errors resulting in warnings**, but the code doesn't crash. The error should be small (~1e-16), and you may choose to ignore it. Otherwise, you might want to compile the code with the `-fp-model precise` flag for intel, or the appropriate flag for the compiler you'd like to use.
- The previously available `unbinding_formatted_output` namelist parameter has been removed. Make sure you remove it from your namelists if you used it!
- For anything else regarding particle unbinding, feel free to contact Mladen Ivkovic (mladen.ivkovic [at] hotmail DOT com)

Documentation

What it does

The purpose of particle unbinding is to identify unbound particles in clumps as identified by the clumpfinder and pass them on to the parent clumps, until the halo-namegiver clumps are reached (where there are no more parent structures to pass the particles on to.)

It will write unformatted output in `output_XXXXX/unbinding_XXXXX.outYYYYY` files the same way it is done for any other backup files in `ramses`, containing the assigned clump IDs of every particle after unbinding. The clump IDs correspond to the clump IDs as used in the `halo_XXXXX.txtYYYYY` and `clump_XXXXX.txtYYYYY` files. If a particle has clump ID 0, it wasn't found to be in any clump.

How it works

First all particles that are in a clump are gathered and assigned the corresponding clump ID. Simultaneously, linked lists of these particles are created for each clump that is not a halo-namegiver, i.e. that is not a clump whose ID will be the halo ID. Such clumps are never merged into another clump, but may have arbitrarily many clumps merged into them. Then clump properties such as centre of mass, the mass profile and bulk velocity are acquired using the linked lists. Starting with the lowest clump level, particles are checked if they are bound to the clump they are assigned to. By default, this unbinding is done iteratively: The clump properties are recomputed using only the remaining particles, and then all particles checked again. Furthermore, by default particles are not allowed to leave the boundaries of the clump they're assigned to in order to be considered bound; For this, the potential at the closest saddle from the clump's centre of mass to a neighbouring clump is subtracted from the particle's energy. (This default behaviour can however be changed with the namelist parameters `saddle_pot` and `iter_properties`). Also note that the bulk velocity and centre of mass of a clump are recovered using only the bound particles per iteration, the mass profile however always uses all included particles, including the substructure particles. The iteration per clump level stops when the bulk density of each clump of that level has converged, i.e. `v_clump_old/v_clump_new < conv_limit` (or when a maximal number of iterations is reached). Particles that are found to be not bound are passed to the parent structure for examination, provided such a structure exists, and the iterations repeated for the next clump level, provided there are clumps of a higher level. There are two possibilities implemented to define the clump's centre. By default, the centre will be the position of the associated density peak. However using the `-DUNBINDINGCOM` preprocessing flag, you can make the centre be the centre of mass, which will also be determined iteratively like the bulk velocity.

More details can be found here.

Namelist Parameters for unbinding

Can be set in the `UNBINDING_PARAMS` block

Name	default	type	function
`particlebased_clump_output=`	`.false.`	logical	write resulting clump properties based on particles after unbinding, not default cell-based properties

Name	default	type	function
`nmassbins=`	`50`	integer	Number of bins for the mass binning of the cumulative mass profile. Any integer > 1.
`logbins=`	`.true.`	logical	use logarithmic binning distances for cumulative mass profiles (and gravitational potential of clumps). If false, the code will use linear binning distances.
`saddle_pot=`	`.true.`	logical	Take neighbouring structures into account; Cut potential off at closest saddle.
`iter_properties=`	`.true.`	logical	whether to unbind multiple times with updated clump properties determined by earlier unbindings
`conv_limit =`	`0.01`	real	convergence limit. If `v_clump_old/v_clump_new < conv_limit`, stop iterating for this clump. (only used when `iter_properties=.true.`)
`repeat_max =`	`100`	integer	maximal number of loops per level for iterative unbinding (in case a clump doesn't converge) (shouldn't happen) (only used when `iter_properties=.true.`)

Making Mergertrees (And Mock Galaxies)

Quickstart Guide

To get your merger trees, you need to set three runtime parameters in the `&RUN_PARAMS` namelist (in addition to whatever you have in there):

```
&RUN_PARAMS
clumpfind=.true.
unbind=.true.
make_mergertree=.true.
/
```

For dark matter only (DMO) simulations, the clump finder recovers good halos and subhalos with these `CLUMPFIND_PARAMS`:

```
&CLUMPFIND_PARAMS
relevance_threshold=3
density_threshold=80
saddle_threshold=200
/
```

By default, mock galaxy catalogues will be created. You can turn this behaviour off by setting

```
&MERGERTREE_PARAMS
make_mock_galaxies=.false.
/
```

in the `&MERGERTREE_PARAMS` block in your namelist.

What output files are created and how to read them is described further below.

Important notes

- To make merger trees, you need to use the clump finder and the particle unbinding routines first. More on clump finding parameters can be found here. More on particle unbinding can be found here.
- Some parts of the code (e.g. binning particles in mass profiles of halos) rely on consistent floating-point operations. The (intel) fortran compiler however doesn't necessarily use value-safe optimisations, which may lead to **errors resulting in warnings**, but the code doesn't crash. The error should be small (~1e-16), and you may choose to ignore it. Otherwise, you might want

to compile the code with the `-fp-model precise` flag for intel, or the appropriate flag for the compiler you'd like to use.

- For accurate merger trees, consider running your simulation for a handful (I used 3) snapshots more than you actually need to make sure past merging events are actually mergers, not just two clumps too close to each other to be recognized as distinct clumps. You'll also need to check in these "extra snapshots" later whether any clump re-emerged later.
- I'm not really able to predict how much memory the patch will need, because it will accumulate orphan galaxies over the simulation. It shouldn't be much, but obviously will depend on how big of a simulation you are trying to run. It never was a significant amount of memory (`< 10 Mb`) when I tried it with `512^3` particles, but I'd keep that in mind if you have memory issues.
- For anything else regarding the merger trees, feel free to contact Mladen Ivkovic (mladen.ivkovic [at] hotmail DOT com)

Documentation

What it does

This functionality creates dark matter halo merger trees. Essentially, clumps as identified by the clumpfinder PHEW between two snapshots are linked as progenitors and descendants, if possible. Preferrably clumps between two adjacent snapshots are linked, but if a descendant has no direct progenitor in the adjacent snapshot, the program will try to find progenitors in older snapshots.

Optionally, it can create mock galaxy catalogues. Using a parametrised SHAM stellar-mass-halo-mass relation, the most bound particle of each clump is assigned a stellar mass. Once a subhalo is merged, the galaxy, now an orphan, is still being tracked until the end of the simulation.

Mergertree Output

The merger trees are stored in `output_XXXXX/mergertree_XXXXX.txtYYYYY` files. Each file contains 11 columns:

- `clump`: clump ID of a clump at this output number
- `progenitor`: the progenitor clump ID in output number "prog_outputnr"
- `prog_outputnr`: the output number of when the progenitor was an alive clump
- `desc_mass`: mass of the current clump.
- `desc_npart`: number of particles of the current clump.

- `desc_x,_y,_z`: x, y, z position of current clump.
- `desc_vx,_vy,_vz`: x, y, z velocities of current clump.

desc_mass and desc_npart will be either inclusive or exclusive, depending on how you set the `use_exclusive_mass` parameter. (See below for details)

How to read the output:

- A clump > 0 has progenitor > 0: Standard case. A direct progenitor from the adjacent previous snapshot was identified for this clump.
- A clump > 0 has progenitor = 0: no progenitor could be established and the clump is treated as newly formed.
- A clump > 0 has progenitor < 0: it means that no direct progenitor could be found in the adjacent previous snapshot, but a progenitor was identified from an earlier, non-adjacent snapshot.
- A clump < 0 has progenitor > 0: this progenitor merged into this clump, but is not this clump's main progenitor.
- A clump < 0 has progenitor < 0: this shouldn't happen.

Mock Galaxy Output

The mock galaxy output is stored in `output_XXXXX/galaxies_XXXXX.txtYYYYY` files. Every file contains 5 columns:

- `Associated_clump`: Provided this is not an orphan galaxy, the clump ID in which this galaxy is. Orphan galaxies have associated clump = 0
- `Stellar_Mass`: The galaxy's stellar mass in units of solar mass.
- `x, y, z`: position of the galaxy.
- `Galaxy_Particle_ID`: The ID of the particle this galaxy is attributed to.

How it works

After every clumpfinding and unbinding step in the simulation, the merger tree code is called. For every clump, the `nmost_bound` number of most bound (= with lowest total energy) particles are found and written to file, as they will be treated as tracers for this clump. In the next output step, those files will be read in and sorted out: The clumps of the previous output will be progenitors of this output. Based on in which descendant clump each progenitor's particles ended up in, progenitors and descendants are linked, i.e. possible candidates are indentified this way. Next, the main progenitor of each descendant and the main descendant of each progenitor need to be

found. A descendant may have multiple progenitors, but only one main progenitor. Progenitors however are only allowed to have one descendant, their main descendant.

The tree-making is performed iteratively. A main progenitor-descendant pair is established when the main progenitor of a descendant is the main descendant of said progenitor. At every iteration, all descendant candidates of all progenitors that haven't found their match yet are checked; The descendants however only move along one progenitor candidate. The iteration is repeated until every descendant has checked all of its candidates or found its match. Progenitors that haven't found a main descendant that isn't taken yet will be considered to have merged into their best fitting descendant candidate.

After the iteration, any progenitor that is considered as merged into its descendant will be recorded as a "past merged progenitor". Then descendants that haven't got a progenitor will try to find a progenitor in non-adjacent snapshots, which are stored as "past merged progenitors". (Obviously not in one of the newly added past merged progenitors.) As the past merged progenitors are traced via 1 particle ("galaxy particle"), the past merged progenitor of the most bound "galaxy particle" that is also assigned as a particle of a descendant will be considered the main progenitor of the descendant under consideration.

Mock galaxy catalogues are created using a parametrised SHAM relation between (sub)halo mass and stellar mass adapted from Behroozi, Wechsler and Conroy 2013. Once a subhalo merges into another clump, its (orphan) galaxy is still being tracked for a user-specified number of snapshots (`max_past_snapshots` paarameter below) by tracking what was the last identifiable most bound particle of that subhalo.

For more details on how it works, some tests and results, you can have a look here.

New namelist parameters for this pach

Can be set in the `MERGERTREE_PARAMS` block.

Name	default	type	function
`nmost_bound =`	`200`	integer	Up to how many particles per clump to track between two snapshots.
`max_past_snapshots =`	`0`	integer	maximal number of past snapshots to store. If = 0, all past merged progenitors will be stored. If `make_mock_galaxies=.true.`, it will also limit the number of snapshots for which orphan galaxies are tracked.

Name	default	type	function
`use_exclusive_mass =`	`.true.`	logical	how to define clump mass: If false, all substructure of a clump is considered part of its mass. Otherwise, use only particles that are bound to the clump itself (excluding main haloes: main haloes always consist of all the particles within them). Note that this mass definition is only used for creating the merger trees, not for the clump/halo output!
`make_mock_galaxies =`	`.true.`	logical	whether to also create mock galaxy catalogues on the fly.

Visualisation and Postprocessing

`ramses/utils/py/mergertreeplot.py` is a python 2 script to plot the merger trees as found by this patch. `ramses/utils/py/mergertree-extract.py` is a python3 script to extract the mass evolution of a single clump, a halo with all its subhaloes, or all haloes in the simulation. Details on options and usage are at the start of the scripts as a comment, or can be called using the `--help` flags.

Crashing on MPI writing routines?

Apparently some MPI implementations have issues with collective writing routines, which are used by default in the merger tree patch. To circumvent this problem, the `-DMTREE_INDIVIDUAL_FILES` preprocessing directive can be set in the Makefile. Just add it to the `DEFINES=` line at the top of the file. With this flag in use, instead of collective files every MPI task will write an individual unformatted Fortran file, and then read it back in at the later snapshot and communicate the data appropriately.

However, be advised: Using this flag creates a lot of small files (`4* #MPI tasks` number of extra files in addition to `2-3 * #MPI tasks` to result files for particle unbinding, merger trees, and, if chosen, galaxy files per snapshot). This might become an issue if the machine you're working on applies file number quotas.

Cosmological Zoom Simulations

It might be desirable to run a simulation to analyze the properties of an individual halo within a certain mass scale (e.g. the one of a galaxy cluster, a galaxy group or an individual galaxy). For this purpose it is of course essential to maximize the resolution on this object. However, within a cosmological framework it is necessary to simulate a sufficiently large region of the universe, in other words to choose a sufficiently large Boxsize (e.g. 100 Mpc/h), to obtain an accurate result. Because of the limitations of computational resources (RAM memory and computation time) a larger box will necessarily need to be simulated with lower resolution than a smaller one, or in turn a higher resolved astrophysical object will allow for a smaller simulation volume, than a lower resolved object. So in principle there is a trade-off between resolution and size of the simulated volume. A way out of this offers the concept of zoom simulations. The word zoom hereby means, that within the chosen simulation volume, the specific region of interest is simulated with a higher resolution, than the rest of the box (the zoom volume might for example be 0.001 times the box size).

To make use of this concept, two major steps are required: First, to run a simulation with uniform resolution (unigrid run) of a box of the desired volume (as described above), and then to determine the region of interest. Given this information the second step, a RAMSES run with a higher resolution in the volume of interest, can be undertaken. This is called the zoom run. Before it can be carried out it is also required to create a new set of initial conditions, which contain the desired resolution in the zoom region.

1. Run a zoom with RAMSES

1. Zoom initial conditions with grafic files

3. Generate zoom initial conditions with MUSIC

Testing

1. Running the automatic test suite

To run the automatic tests, navigate to the tests directory, and run the `run_test_suite.sh` script:

```
>$ cd tests
>$ ./run_test_suite.sh
```

The tests will begin and the output should look like:

```
###############################################
#   Running RAMSES automatic test suite       #
###############################################
Will perform the following tests:
 [ 1] hydro/implosion
 [ 2] hydro/sod-tube
 [ 3] mhd/imhd-tube
 [ 4] mhd/orszag-tang
 [ 5] rt/stromgren2d
 [ 6] sink/smbh-bondi
-----------------------------------------------
Test 1/6: hydro/implosion
Cleanup
Compiling source
```

and so on. Once the tests have completed, a report is generated in a `.pdf` file named `test_results.pdf`, alongside a log file `test_suite.log`.

Options

- Run the suite in parallel (on 4 cpus):

```
./run_test_suite.sh -p 4
```

- Do not delete results data:

```
./run_test_suite.sh -d
```

- Run in verbose mode:

```
./run_test_suite.sh -v
```

- Select individual tests (for tests 3 to 5, and 10):

```
./run_test_suite.sh -t 3-5,10
```

- Run all tests in `mhd` directory:

```
./run_test_suite.sh -t mhd
```

- Run test suite with coverage:

```
./run_test_suite.sh -s
```

- Run tests with restart:

```
./run_test_suite.sh -r
```

This will add an intermediate output in the middle of the test, and restart from it.

2. Creating a new test

The following steps describe how to add a new test to the test suite. In this example, the test will be named `sedov-3d`.

The first step is to create a new directory `sedov-3d` in one of the `hydro`, `mhd`, `rt`, or `sinks` directories. No need to modify the `run_test_suite.sh` script, the new test will automatically be picked up and added to the list. We will choose to place it inside the `hydro` directory. Please use hyphens (`-`) in your test names instead of underscores (`_`) as `latex` does not like underscores.

```
>$ cd hydro
>$ mkdir sedov-3d
```

Note: use one directory per test. If you want to run a 2D and a 3D sedov test, create separate `sedov-2d` and `sedov-3d` directories.

In that directory, you will need:

- A `config.txt` file: usually just contains the Makefile flags, e.g. `FLAGS: NDIM=3 PATCH=SOLVER=hydro`
- A namelist: `sedov-3d.nml` (the name needs to be the same as the test directory)

⚠ Warning

For the restart system to work, there is some limitations on the output parameters you can use. They have to be on the form

```
&OUTPUT_PARAMS
noutput=1 ! should be 1
tout=0.620
/
```

or

```
&OUTPUT_PARAMS
noutput=1 ! should be 1
aout=1.355E-01
/
/
```

for cosmo runs or

```
&OUTPUT_PARAMS
foutput=1 ! should be 1
tend=0.05
/
```

- A file for plotting and checking the solution against a reference: `plot-sedov-3d.py`. It is advised to copy a file from the other directories to see how to write this. **Note that this file needs to contain at least one call to `visu_ramses.check_solution(data["data"], 'sedov-3d')`**.
- A reference solution: `sedov-3d-ref.txt`. To create it, run your test and once the final output (number 2 in this case) has been created, do the following:

```
import visu_ramses
data = visu_ramses.load_snapshot(2)
visu_ramses.check_solution(data["data"], 'sedov-3d', overwrite=True)
```

- A `Readme.md` containing a short description of the test

Optional files:

- `condinit.f90`: you can have your own initial setup if it's not entirely definable in a namelist. **REMEMBER** to set the correct `PATCH` in the `config.txt` file! (e.g. `PATCH=../tests/hydro/sedov-3d`). An other option is to add a custom condinit routine option in the existing condinit.f90 file.
- `before-test.sh`: if this file is present in the test directory, it will be run before the test begins (useful for e.g. creating symbolic links to libraries...)
- `after-test.sh`: if this file is present in the test directory, it will be run after the test begins (useful for e.g. cleaning up symbolic links to libraries...)

Tuning tolerances for solution verification

By default, relative differences between the sums of all the variables inside all leaf cells in the domain and the reference solution cannot exceed `3.0e-13`. Sometimes, some variables are more volatile than others when running simulations on different numbers of CPUs, and this limit is too low, leading to false failed tests. The `check_solution` method in the `visu/visu_ramses.py` module can be tuned to work for your test using the following options:

- `tolerance`: a dictionary listing the allowed relative difference between the sum over all leaf cells and reference value. The default is `{"all":3.0e-13}`. To make the check on `density` less restrictive, use for instance `tolerance={"density":1.0e-10}`.
- `threshold`: relative value below which a vector component is set to zero. Default is `2.0e-14`.
- `norm_min`: minimum value for the norm of a vector, to protect against null vectors. Default is `1.0e-30`.
- `min_variance`: if the data differs by less than this value from the average value, it is set to the average. Default is `1.0e-14`.

3. Creating a new group of tests

If your test does not fall under the categories already present in the `tests` directory (`hydro`, `mhd`, `rt`, `sinks`), you can create a new directory and put your tests in there. You will then have to edit the `run_test_suite.sh` file to ensure your new tests will be picked up.

Say you want to create 3 new tests, `sedov-1d`, `sedov-2d`, and `sedov-3d` inside a new `sedov` directory, you have to find the line describing the list of directories to be scanned at the top of the `run_test_suite.sh` file:

```
# List of directories to scan
testlist="hydro,mhd,rt,sink";
```

and add your new directory separated from the previous one by a comma, i.e.

```
# List of directories to scan
testlist="hydro,mhd,rt,sink,sedov";
```

External tools

This page is designed to promote tools, datasets or other useful things that may be interesting to other RAMSES users but don't belong on the main RAMSES repository.

Feel free to edit this page and add your own tools with details on how to obtain them.

Initial Conditions

DICE

DICE is designed to set up one or more galaxies in isolation. It is included in RAMSES.

- Code repository.

MUSIC

MUSIC generates cosmological initial conditions and can be used to set up RAMSES cosmo simulations.

- Code repository,
- Doc page.

MPgrafic

MPgrafic generates (large) cosmological initial conditions and can be used to set up RAMSES cosmo simulations.

- Code repository,
- Reference.

Analysis and Post-Processing

RAMSES tools

RAMSES comes with a decent number of Fortran routines and programs to extract information from RAMSES outputs.

You can find some documentation on the relevant doc page.

OSYRIS

OSIRIS is a simple python interface developed by Neil Vaytet to visualise RAMSES outputs

- Code repository

YT

YT is a large community code supporting a number of simulation codes, including RAMSES.

- Website

PYNBODY

Pynbody is a python interface developed by Andrew Pontzen to visualise particle data. It also works for RAMSES outputs by turning cells into particles.

- Code repository

Pymses

Pymses is an analysis library written in Python for RAMSES outputs.

- Webpage (outdated)

MERA

Mera is a Julia package developed by Manuel Behrendt to efficiently read/store/analyse RAMSES outputs.

- Code repository

Visualisation

GLnemo2

GLnemo2 is an interactive visualisation 3D program using OpenGL. This software, developed by Jean-Charles Lambert, can help you visualise particles from many different codes, including RAMSES.

It works on your laptop or on larger servers.

- Code repository

Implementation details

This section of the documentation discusses the structure of the code and the implementation of the different modules. It aims to provide developers with a better understanding of the inner workings of RAMSES.

This documentation is written in the style of lectures adressing specified topics and includes exercises. Its first version was written in 2025 for the first RAMSES developer school organised by the RAMSES SNO (see credits below).

Structure of the code

Contents

1. The program RAMSES

The root of the program RAMSES is found in *amr/ramses.f90*:

```
program ramses
 call read_params      ! Read run parameters
 call adaptive_loop    ! Start time integration
end program ramses
```

First, the routine `read_params` will load the parameters from the namelist that was given as input by the user. Then, the routine `adaptive_loop` is called. It is found in *amr/adaptive_loop.f90* and structured as follows:

```
subroutine adaptive_loop
  ! Initialize the simulation
```

```
  call init_amr
  call init_time
  if(hydro)call init_hydro
  ...
  ! Main time loop
  do
    ! Make new refinements level 1 to levelmin
    ...
    ! Call base level
    call amr_step(levelmin,1)
    ! Do some other stuff
    ...
    ! Print some info
    ...
  end do
end subroutine adaptive_loop
```

The simulation starts with the initialization: arrays are allocated and set to appropriate initial values, initial conditions are calculated or read from file,... After that, the main time loop is started, which will evolve the simulation in time.

The core of RAMSES is the recursive routine `amr_step` found in the file `amr/amr_step.f90`. In this routine, all individual physics components are called in a specific order.

Exercise

Look into the file `amr/amr_step.f90`. Can you make a list of which physical processes are modelled in ramses? Take a few minutes to explore the different directories of the code. Can you find out where the code of each physical process is?

2. An overview of `amr_step`

A simplified schematic version of the core routine `amr_step` shows the structure (see `amr/amr_step.f90`):

```
recursive subroutine amr_step(ilevel,icount)

   call refine
   call load_balance

   ... ! Some sink and particle stuff

   if(time_to_output) call dump_all

   if (conditions are met)
      call kinetic_feedback              ! feedback from stars
```

```
      OR
      call make_stellar_from_sinks
      call make_sn_stellar              ! feedback from sinks
   end if

   if(poisson) call rho_fine   ! calc density field for Poisson
source term

   ... ! Some particle stuff

   ! Gravity update: compute grav potential and acceleration
   if(poisson)then
      ...
      call phi_fine_cg(ilevel,icount) OR
multigrid_fine(ilevel,icount)
      call force_fine(ilevel,icount)
      ...
  end if

  if(rt .and. rt_star/sink) call update_star/sink_RT_feedback(ilevel)

  call calc_turb_forcing(ilevel)   ! turbulence forcing

  call newdt_fine(ilevel)          ! Compute new time step

  if(hydro)call set_unew(ilevel)   ! set unew = uold
  if(rt)call rt_set_unew(ilevel)

  ! --- Recursive call to amr_step ---
  ...
  !-----------------------------------

  if(conditions met) call thermal_feedback(ilevel)  ! feedback from
stars

  if(sink.and.hydro) call grow_sink(ilevel,.false.)  ! sink accretion

  ! Hydro step: solve hydro and add source terms
  if((hydro).and.(.not.static_gas))then
     call godunov_fine(ilevel)
     ...
  endif

 ! Do RT/Chemistry step -> works on uold
  if(rt .and. rt_advect) then
     call rt_step(ilevel)
  else
     call cooling_fine(ilevel)
  endif

  if(pic) call move_fine(ilevel)  ! Move particles
```

```
  if(conditions met)call star_formation(ilevel)

  ...  ! Update physical and virtual boundaries

  if(MHD) call diffusion  ! Magnetic diffusion step

  if(conditions met) call flag_fine  ! Compute refinement map

  ... ! particle stuff

  if(conditions met)call create_sink  ! Sink production

end subroutine amr_step
```

Things are done in a specific order. The reasons for this will become more clear over the course of these lectures.

3. Recursivity of `amr_step`

```
recursive subroutine amr_step(ilevel,icount)
  ...
  ! do things in the beginning
  ...
  !-------------------------
  ! Recursive call to amr_step
  !-------------------------
  if(ilevel<nlevelmax)then
     if(numbtot(1,ilevel+1)>0)then  !if there is stuff in next level
        if(nsubcycle(ilevel)==2)then
           call amr_step(ilevel+1,1)
           call amr_step(ilevel+1,2)
        else
           call amr_step(ilevel+1,1)
        endif
     else
        ! Otherwise, update time and finer level time-step
        dtold(ilevel+1)=dtnew(ilevel)/dble(nsubcycle(ilevel))
        dtnew(ilevel+1)=dtnew(ilevel)/dble(nsubcycle(ilevel))
        call update_time(ilevel)
     end if
  else
     call update_time(ilevel)
  end if
  ...
  ! do things at the end
  ...
end subroutine amr_step
```

Exercise

Write down the calls to `amr_step` assuming there are 3 refinement levels.

1. First assume there is no subcycling (nsubcycle(ilevel)==1 for all levels).

2. Now assume you have subcycling for all levels.

Solution

If we split the computations done in amr_step into two parts: stuff done before the recursive call to amr_step and stuff done after

```
recursive subroutine amr_step(ilevel,icount)

   ! calc phi(ilevel), set unew(ilevel)=uold(ilevel), ..., calc
dt(ilevel)
   stuff_before(ilevel)

   ! recursive call
   if(ilevel<nlevelmax)then
      if(nsubcycle(ilevel)==2)then
         call amr_step(ilevel+1,1)
         call amr_step(ilevel+1,2)
      else
         call amr_step(ilevel+1,1)
      endif
   else
   call update_time(ilevel)
   end if

   ! solve hydro, set uold(ilevel)=unew(ilevel), ...
   stuff_after(ilevel)

end subroutine amr_step
```

1. Without subcycling

```
call amr_step(l-1)
      stuff_before(l-1)
   call amr_step(l)
      stuff_before(l)
      call amr_step(l+1)
            stuff_before(l+1)
            t = t+dt(l+1)
            stuff_after(l+1)
```

```
      stuff_after(l)
   stuff_after(l-1)
```

At the end we have advanced by dt(l+1)

1. With subcycling

```
call amr_step(l-1,1)
      stuff_before(l-1)
   call amr_step(l,1)
      stuff_before(l)
      call amr_step(l+1,1)
            stuff_before(l+1)
            t = t+dt(l+1)
            stuff_after(l+1)
      call amr_step(l+1,2)
            stuff_before(l+1)
            t = t+dt(l+1)
            stuff_after(l+1)
      stuff_after(l)
   call amr_step(l,2)
      stuff_before(l)
      call amr_step(l+1,1)
            stuff_before(l+1)
            t = t+dt(l+1)
            stuff_after(l+1)
      call amr_step(l+1,2)
            stuff_before(l+1)
            t = t+dt(l+1)
            stuff_after(l+1)
      stuff_after(l)
   stuff_after(l-1)
```

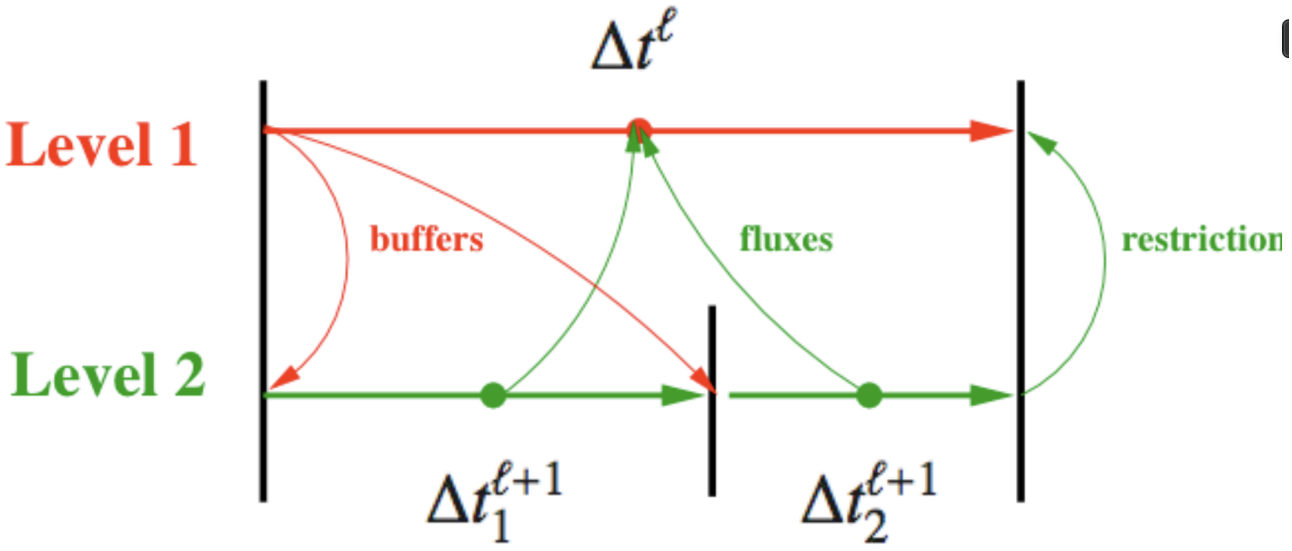
At the end we have advanced by 4 level l+1 timesteps dt(l+1)

4. Time stepping

RAMSES enables adaptive time stepping where each AMR level evolves with individual timesteps. Though, the following rule always applies:

\(\Delta t^{\ell}=\Delta t^{\ell+1}_1+\Delta t^{\ell+1}_2\)

An example of time stepping with two levels in the figure below (Credits: Romain Teyssier)



Level 2 is updated first with first with a time step of size \(\Delta t^{\ell+1}_1\) and second with \(\Delta t^{\ell+1}_2\).The coarse level \(\ell=1\) is frozen during fine level solves (one order of accuracy down !). The fine flux are averaged in time at coarse fine boundaries. Then level \(\ell\) is updated.

\(\bf{F}^{n+1/2,\ell}_{i+1/2,j}=\frac{1}{\Delta t_1^{\ell+1}+\Delta t_2^{\ell+1}}\left(\Delta t_1^{\ell+1}\frac{\bf{F}^{n+1/4,\ell+1}_{i+1/2,j-1/4}+\bf{F}^{n+1/4,\ell+1}_{i+1/2,j+1/4}}{2} + \Delta t_2^{\ell+1}\frac{\bf{F}^{n+3/4,\ell+1}_{i+1/2,j-1/4}+\bf{F}^{n+3/4,\ell+1}_{i+1/2,j+1/4}}{2} \right)\)

The timestep is computed in `pm/newdt_fine.f90`. For more information, see Section 2.4 in the RAMSES paper (Teyssier 2002).

Hydrodynamics

Contents

Hydrodynamics 105

1. The Euler equations of hydrodynamics

RAMSES uses conservative variables that are updated with the conservative equations (mass, momentum, total energy)

$$\frac{\partial \rho}{\partial t} + \nabla\cdot \left[\rho\textbf{u} \right] = 0 \\ \frac{\partial \rho\textbf{u}}{\partial t} + \nabla \cdot\left[\rho \textbf{u}\otimes \textbf{u} + P \mathbb{I} \right] = 0\\ \frac{\partial E_\mathrm{T}}{\partial t} + \nabla\cdot \left[\textbf{u} \left(E_\mathrm{T} + P \right)\right] = 0,$$

with ρ the density, $\textbf{u}$ the velocity, $E_\mathrm{T}=e + 1/2\rho \textbf{u}^2$ the total energy, e the gas thermal energy, and $P=(\gamma-1)e$ the gas pressure. γ is the adiabatic index.

The system can be written in the canonical form

$$\frac{\partial \mathbb{U}}{\partial t} + \nabla\cdot\mathbb{F}(\mathbb{U}) = \mathbb{S},$$

where $\mathbb{U}$ is the vector of conservative variables

$$\mathbb{U}=\left[\begin{array}{c} \rho \\ \rho\textbf{u} \\ E_\mathrm{T} \\ \end{array} \right],$$

$\mathbb{F}$ is the flux, and $\mathbb{S}$ the source terms. $\mathbb{U}$ are the fundamentals variables in RAMSES which are stored in `uold` and `unew` (see below). For practical reasons, RAMSES often switches to primitive variables

$$\mathbb{Q}=\left[\begin{array}{c} \rho \\ \textbf{u} \\ P \\ \end{array} \right].$$

Indeed, it is easier to use the dual space of the primitive variables in the integration of the hyperbolic solver. In addition, when for instance feedback is activates, mass, momentum or thermal energy are added, which requires to use primitive variables.

Additionnal variables can also be handled in RAMSES. These are non-thermal energies E_NT (cosmic rays, radiative energy) and passive scalars X (metals, chemical species, tracers, etc...). Passive scalars are variables that are passively advected with the flow. These variables are integrated with the following evolution equations

$$\frac{\partial \rho X}{\partial t} + \nabla\cdot \left[\rho X\textbf{u} \right] = 0$$

$$\frac{\partial E_\mathrm{NT}}{\partial t} + \nabla\cdot \left[E_\mathrm{NT} \textbf{u} \right] = -P_\mathrm{NT}\nabla.\textbf{u}$$

with $P_\mathrm{NT}=(\gamma_\mathrm{rad}-1)E_\mathrm{NT}$.

Magnetic fields \(\textbf{B}\) are stored at each cell faces. They are updated using the induction equation in the ideal MHD limit.

\(\frac{\partial \textbf{B}}{\partial t} - \nabla\times \left[\textbf{u} \times \textbf{B}\right] = 0.\)

2. Hydro variables in RAMSES

The arrays `uold` and `unew`

In RAMSES, all hydro variables are stored together in the two-dimensional arrays `uold` and `unew`, which contain the value of each variable in each cell of the AMR grid. They are defined in `hydro_commons`

```
real(dp),allocatable,dimension(:,:)::uold,unew
```

and allocated in `init_hydro`

```
allocate(uold(1:ncell,1:nvar))
allocate(unew(1:ncell,1:nvar))
```

The arrays `uold` and `unew` are used for different purposes. `uold` stores the state that is used as input to integrate forward in time. `unew` is a temporary array that is used to sum up flux contributions in all directions, contributions for source terms, or feedback. Note that `unew` can be desynchronized in time when AMR and adaptive timesteps are used. In that case, `unew` stores the contribution from the fine level to the coarse level. Importantly, this means that `unew` should never be used to access the state variables, or to communicate state variables of level \(\ell+1\) or \(\ell-1\) between MPI domains.

Number of hydro variables `nvar`

The total amount of independent variables stored on the AMR grid is set at compile time in the Makefile by the parameter `nvar`. It includes

- the Euler variables: density, velocity and pressure
- the magnetic field vector (on the left cell face), when compiled with MHD
- an optional number of non-thermal energies (`NENER`)
- an optional number of passive scalars (`NMETALS` and `NPSCAL`)

Remark that `NMETALS` and `NPSCAL` are not passed to the source code. For the hydro-solver there is no distinction between these types of variables and both are treated as passive scalars.

> **⚠ Warning**
>
> Remark that `uold` and `unew` actually contain the **conservative** Euler variables \(\mathbb{U}\): density, momentum and total energy. For **primitive** variables (density, velocity and pressure) a conversion needs to be made first. See subroutine `ctoprim`.

Accessing variables in `uold` and `unew`

Several parameters are used to keep track of the number of different types of variables and their indices in the arrays `uold` and `unew`.

The number of **Euler variables** is indicated by `neul` in the code. We always have density and pressure, but the amount of velocities to keep track of depends on the number of dimensions of the simulation and whether the code is compile with the HYDRO or MHD solver. For HYDRO, we have `neul = ndim+2` with `ndim` the amount of spatial dimensions of the simulation. For MHD simulations, we always need to keep track of the three velocities and so `neul = 5`. They are defined in `hydro_parameters.f90`.

When using the MHD solver, we need additional variables to keep track of the magnetic field in the three dimensions. Contrary to the hydro variables which are defined in the center of each cell, the magnetic field is defined in the cell faces. The magnetic field (on the left cell face) for the three spatial directions is added after the Euler variables. The number of Euler variables with addition of the magnetic field is indicated by `nhydro` in the code. This makes it easy to loop over them:

```
do i=1,nhydro
```

We have

- for HYDRO: `nhydro=neul`
- for MHD: `nhydro=neul+3=8`

Additionally, the magnetic field on the right cell face is added at the very end of the variable array. This means that when following the evolution of magnetic fields, we store 6 additional variables. For convenience, the code defines the parameter `nvar_all` which, in addition to the independent `nvar` variables, includes the right magentic field. So we have

- for HYDRO: `nvar_all = nvar`
- for MHD: `nvar_all = nvar+3`

To compute the cell-center magnetic field, for example for outputting, we do

```
Bx(i) = 0.5*(uold(i,neul+1) + uold(i,nvar+1))
By(i) = 0.5*(uold(i,neul+2) + uold(i,nvar+2))
Bz(i) = 0.5*(uold(i,neul+3) + uold(i,nvar+3))
```

RAMSES allow to include additional variables in the simulation: passive scalars and non-thermal energies. Passive scalars can be used to keep track of metals and star formation recipe variables. These specific scalars are accessed through the indices `imetal`, `idelay`, `ivirial1`, `ivirial2`, `ixion`, `ichem`. These are used in different star formation recipes. In the hydro-solver, these are evolved as regular passive scalars.

Summary

To access the Euler variables in `uold` and `unew`, use the indices:

- density: `1`
- total energy or pressure: `neul`
- momentum or velocities (HYDRO case): `2` up to `1+ndim`
- momentum or velocities (MHD case): `2, 3, 4`

To access the magnetic field:

- on the left side of the cells: `neul + 1`, `neul+2`, `neul+3`
- on the right side of the cells: `nvar+1`, `nvar+2`, `nvar+3`

Additional variables are stored after the left magnetic field in the following order:

- NENER: `inener=nhydro+1` up to `nhydro+nener`
- passive scalars: `nhydro+nener+1` up to `nvar`

Exercise

How to add a field variable to `uold` and `unew`? List all the things that need changing (allocation?, initialisation?)

Solution

- nvar in makefile

- how to output metadata info, input (don't forget conversion to primitive vars)
- a recipe to convert conservative to primitive variable in the routine `ctoprim`

3. The hydro step in `amr_step`

Exercise

Where are 'uold' and 'unew' altered? Ignore MPI communications for now. Because RAMSES makes use of common arrays which are globally defined and accessible by all parts of the code, it can be tricky to see which routines use these arrays as input or alter their state. Rewrite `amr_step` in pseudo-code, indicating where the arrays `uold` and `unew` are updated.

Solution

```
...
! Compute new time step
call newdt_fine(ilevel)

! Set unew = uold
if(hydro)call set_unew(ilevel)

!--------------------------
! Recursive call to amr_step
!--------------------------
...

!-----------
! Hydro step
!-----------
if(hydro)then
   ! Hyperbolic solver - add flux to unew
   call godunov_fine(ilevel)

   ...
   ! Add gravity source terms to unew
   if(poisson) call add_gravity_source_terms(ilevel)

   ! Add non conservative pdV terms to unew
   ! for thermal and/or non-thermal energies
   if(pressure_fix.OR.nener>0) call add_pdv_source_terms(ilevel)

   ! Set uold = unew
```

```
    call set_uold(ilevel)

    ...

endif
...
```

Gravity

Contents

1. The Poisson equation for gravity

Poisson's equation for gravity is written as follows

$$\nabla^2 \phi = -4 \pi G \rho$$

where ϕ is the gravitational potention and ρ the total density field, taking into account the gas and particles. The gravitational force is given by

$$\bf{f}= -\nabla \phi$$

The steps for solving gravity are as followed:

- determine the Poisson source term
- compute the gravitational potential by solving the Poisson equation
- calculate the gravitational force (or acceleration) by taking the gradient of the potential
- apply the force to the gas and particles.

We will address each of these steps in more detail below.

The routines to compute the gravitational potential and force can be found in the subdirectory *poisson/*. Since source term and force application deal with either gas or particles, the routines related to these steps are either found in the directory *pm/* or *hydro/*.

2. Gravity variables in RAMSES

The variables for the gravity module are defined in `poisson_commons`:

```
real(dp),allocatable,dimension(:)  ::phi,phi_old       ! Potential
real(dp),allocatable,dimension(:)  ::rho               ! Density
real(dp),allocatable,dimension(:,:)::f                 ! 3-force
```

and allocated in `init_poisson`:

```
allocate(rho (1:ncell))
allocate(phi (1:ncell))
allocate(phi_old (1:ncell))
allocate(f   (1:ncell,1:3))
rho=0; phi=0; f=0
```

They consist of

- the gravitational force `f`, a vector with `ndim` dimensions,
- the gravitational potential `phi` and a copy of the old state `phi_old`,
- the total density distribution `rho`, including gas and particles.

3. Computing the Poisson source term

The first step is to compute the Poisson source term on the grid. We need to take into account both the contribution from the gas and the particles. This implies converting the collection of free moving particle masses to a density field on the grid. There are several numerical schemes to do this. In RAMSES, we use the cloud-in-cell (CIC) scheme, which will be detailed in the chapter on

Particles, section XX. If the simulation has both gas and paticles, the contributions of both will be added together, resulting in a total density field (stored in `rho`) which will be used as the Poisson source term.

The routine responsible for this step is `rho_fine`, which is called in `amr_step`:

```
! Compute poisson source term (i.e. the density field)
call rho_fine(ilevel,icount)
```

4. Solving for the gravitational potential and force

Once we have the density field, we can feed it to our Poisson solver of choice, which can be either Multigrid (`multigrid_fine`) or conjugent gradient (CG - `phi_fine_cg`). These routines calculate the gravitational potential and store it in `phi`. The inner workings of these solver is quite complex, so we will not go into the details here.

The gravitational force can then be determined by computing the gradient of `phi`. This is done by the routine `force_fine`. The resulting 3-force is stored in the array `f`.

We can find these two step in `amr_step`:

```
! Compute gravitational potential
if(ilevel>levelmin)then
   if(ilevel .ge. cg_levelmin) then
      call phi_fine_cg(ilevel,icount)
   else
      call multigrid_fine(ilevel,icount)
   end if
else
   call multigrid_fine(levelmin,icount)
end if
!when there is no old potential...
if (nstep==0)call save_phi_old(ilevel)

! Compute gravitational acceleration
call force_fine(ilevel,icount)
```

5. Applying the gravitational force on the gas

When there is gravity, a source term $\mathbb{S}$ is added to the Euler equations to account for the gravitational acceleration:

$\frac{\partial \mathbb{U}}{\partial t} + \nabla\cdot\mathbb{F}(\mathbb{U}) = \mathbb{S}$.

In ramses, the gravitational source term is calculated as

$$\mathbb{S}^{n+\frac{1}{2}}_{i}= \left[\begin{array}{l} 0 \\[6pt] \dfrac{\rho_i^{n}\nabla \phi_i^{n} + \rho_i^{n+1}\nabla \phi_i^{\,n+1}}{2} \\[10pt] \dfrac{(\rho \textbf{u})_i^{n}\nabla \phi_i^{n} + (\rho \textbf{u})_i^{n+1}\nabla \phi_i^{\,n+1}}{2} \end{array} \right] \;\cdot$$

In the code, $\mathbb{S}$ is referred to as the gravity source term (not to be confused with the Poisson source term). For the gas, gravitional acceleration is taken into account using a Velvet scheme (time centered).

Exercise

Where in `amr_step` is the gravitationnal acceleration source term integrated when you have hydro? Write the corresponding pseudo-code (assuming there are no particles).

Solution

The acceleration is added in four places, but with a subtile change of sign in one of the calls. Equation 13 in Teyssier (2002) is done with `add_gravity_source_terms(ilevel)` (index n) and `line 19` for n+1.

The parts in `amr_step` relevant for the gravity calculation in the case of hydro (ignoring particles) can be summarized as follows:

```
! GRAVITY UPDATE
if(poisson)then
   ! Save old potential for time-extrapolation at level
boundaries
   call save_phi_old(ilevel)

   ! Compute poisson source term (i.e. the density field)
   call rho_fine(ilevel,icount)

   ! Remove gravity source term with half time step and old
force (u+0.5*f*dt)
   if(hydro) call synchro_hydro_fine(ilevel,-0.5*dtnew(ilevel),
1)

   ! Compute gravitational potential using multigrid of CG
method
   call multigrid_fine(ilevel,icount) OR
phi_fine_cg(ilevel,icount)

   ! Compute gravitational acceleration
   call force_fine(ilevel,icount)

   ! Add gravity source term with half time step and new force
   if(hydro) call synchro_hydro_fine(ilevel,+0.5*dtnew(ilevel),
```

```
1)
end if
...

! Compute new time step
call newdt_fine(ilevel)

! Set unew equal to uold
if(hydro)call set_unew(ilevel)

! RECURSIVE STEP TO AMR_STEP
...

! HYDRO STEP
if(hydro)then
   ! Solve hydro
   ...
   ! Add gravity source terms with half a time step to unew
   if(poisson)call add_gravity_source_terms(ilevel)

   ! Set uold equal to unew
   call set_uold(ilevel)

   ! Add gravity source term with half time step and old force
   ! in order to complete the time step
   if(poisson)call synchro_hydro_fine(ilevel,+0.5*dtnew(ilevel),
1)
   ...
end if
```

Remark that the routine `synchro_hydro_fine()` alters `uold`, while `add_gravity_source_terms()` alters `unew`.

Half a timestep is added at the end of the global time step to syncrhonize all levels and to make outputs at the beginning of the next timestep. This contribution is then removed ater the dump.

6. Applying the gravitational force on the particles

For the particles, the gravitational acceleration is taken into account using a leap-frog integrator (see Section 2.2.5 in Teyssier (2002). The particle positions and velocities are first updated by a predictor step (“kick-drift”):

\(v_{n+1/2}=v_n + \frac{1}{2} f_n \Delta t\)

$x_{n+1}=x_n + v_{n+1/2} \Delta t$

where f is the gravitational acceleration. This is then followed by a corrector step ("kick"):

$v_{n+1} = v_{n+1/2} + \frac{1}{2} f_{n+1} \Delta t$

Finding where these updates are performed can be a little tricky and counter-intuitive. The predictor step is done by `move_fine`, while the corrector step is done by `synchro_fine` (it *synchronises* the velocities to the current time). For the corrector step, the gravitational acceleration at time t^{n+1} is needed. For this reason, it is postponed until the *next* time step, right after the new gravitational force has been calculated using the updated particle positions. In `amr_step`, we thus find `synchro_fine` *before* `move_fine`:

```
! Gravity update
if(poisson)then
   ! Compute gravitational potential using multigrid of CG method
   call multigrid_fine(ilevel,icount) OR phi_fine_cg(ilevel,icount)

   ! Compute gravitational acceleration
   call force_fine(ilevel,icount)

   ! Synchronize remaining particles for gravity
   if(pic) call synchro_fine(ilevel)

end if
...
! Compute new time step
call newdt_fine(ilevel)

! RECURSIVE STEP TO AMR_STEP
...

! Move particles
if(pic) call move_fine(ilevel) ! Only remaining particles
```

Because the gravitational force is known on the grid, not at the particle positions, we need to apply the inverse CIC scheme (see chapter on Particles) when updating particle velocities. In summary, the force acting on the particle will be interpolated from the cells with which the particle "overlaps". Once this is done, the particle velocities can be updated using the time-step of the level on which the particle lives.

Mesh data structures

Several data structures are used to keep track of the cells in the grid. We go into the purpose of each of them.

Contents

1. Octree representation of the computational domain

To represent the grid, RAMSES uses an octree structure which can be refined locally. The computational domain is divided into **grids** (classically called octs): collections of 2\(^\texttt{ndim}\) **cells** (8 in 3D) where `ndim` is the number of dimensions. On level 1, we start with 1 grid containing 2, 4 or 8 cells which span the entire computational domain. The maximum number of cells in each dimension for a refinement level \(L\) is 2\(^L\).

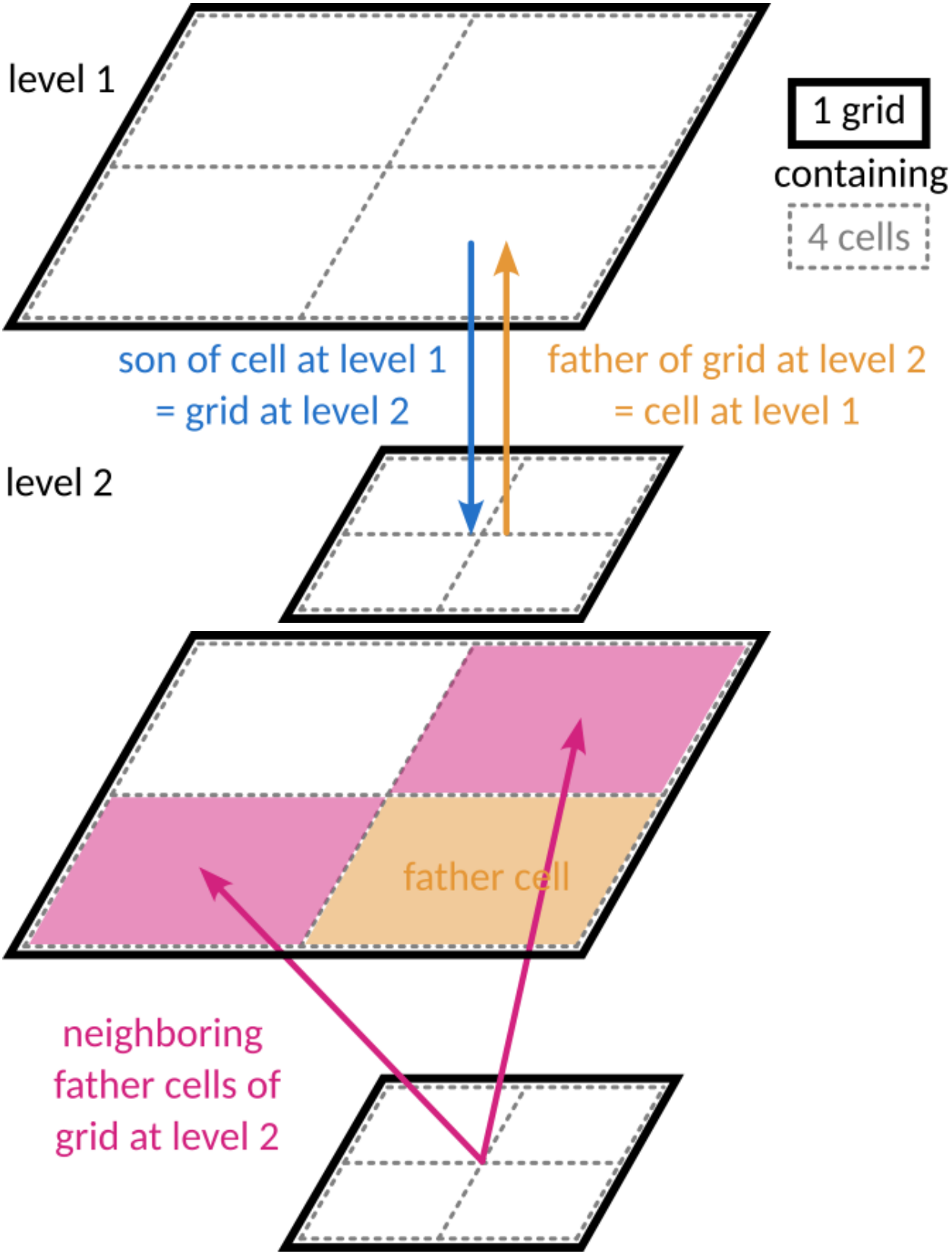
Remarkt that the number of cells in a grid is a quantity that is often used in the code, and so it has its own decided variable: 2\(^\texttt{ndim}=\) `twondim`.

To keep track of the relation between cells and grids of different refinement levels, ramses stores three arrays. They defined in the module `amr_commons` and allocated in the subroutine `init_amr`:

```
integer ,allocatable,dimension(:)  ::son     ! son grid
integer ,allocatable,dimension(:)  ::father  ! father cell
integer ,allocatable,dimension(:,:)::nbor    ! neighboring father
cells
```

```
allocate(son   (1:ncell))
allocate(father(1:ngridmax))
allocate(nbor  (1:ngridmax,1:twondim))
```

A schematic example for two refinement levels in a 2D grid:



- If a cell is refined, the array `son` contains the index of the child grid of the cell, from which the 2\(^\texttt{ndim}\) children cells can be accessed. If the cell is not refined, its son is set to 0.

- A grid at level L has a parent cell at level L-1. The array `father` stores for each grid the index of their parent cell.

- The father cell of a grid has two directly neighoring cells in each spatial direction (2 in 1D, 4 in 2D and 6 in 3D). The refinement rules dictate that these neighbors exist on the level (but they may be refined themselved). The indices of these neighbors are stored in the array `nbor`, which is a 2D array of size [ngridmax, 2 x ndim].

Exercise

Using the arrays defined above, how can you get the \(2 \times \texttt{ndim}\) directly neighboring grids of a grid at index `igrid`?

Solution

```
do j=1,twondim
   neighbor_grid(j) = son(nbor(igrid, j))
end do
```

This is implemented in the routine `getnborgrids` in *amr/nbors_utils.f90*.

2. Grid versus cell based arrays

In general, AMR grid related arrays are defined in *amr_commons.f90*, and allocated in *init_amr.f90*. Arrays can be either grid-based or cell-based, which determines their size at allocation:

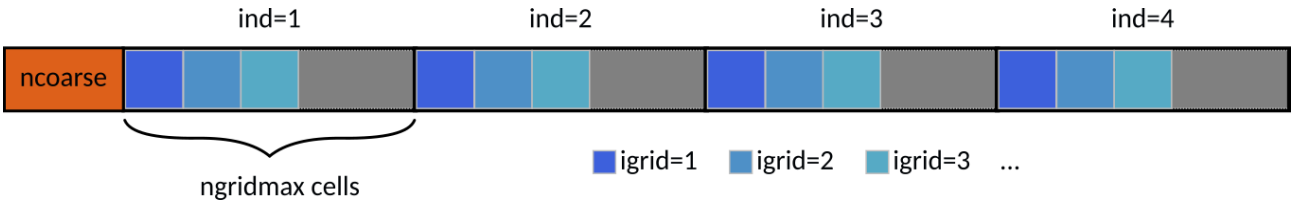
```
allocate(grid_based_array(1:ngridmax))
allocate(cell_based_array(1:ncell))
```

When running a simulation, the user specifies the number of grids to be allocating for working memory through the parameter `ngridmax`. Remark that this is the number of grids or cells allocated for each MPI process. Alternatively, the user can specify the total number of grids over the entire simulation domain using `ngridtot=ngridmax*ncpu`. This is then converted to `ngridmax` to be used in the remainder of the source code. Do **not** use `ngridtot` inside your code, other than to convert to `ngridmax`.

The corresponding number of cells is then `ngridmax` multiplied by 2\(^\texttt{ndim}\), plus the number of cells at the coarses level that takes to take into account the physical bounderies:

```
ncell=ncoarse+twotondim*ngridmax
```

The cells inside a grid are stored in a specific order. Their position is typically indicated by the variable `ind` which goes from 1 to 2\(^\text{ndim}\) (`twotondim`). In a cell-based array, all cells with `ind=1` are stored first, followed by all those with `ind=2`, etc. A schematic example in 2D:



Added at the beginning of the list is the root cell and the coarse cells for the physical boundaries, a total of `ncoarse` cells, with `ncoarse=nx*ny*nz`. In the case of periodic boundary conditions `nx=ny=nz=1`, meaning the only coarse cell is the root cell of the octree at level 0.

How exactly the positions `ind` are defined in space is a matter of arbitrary convention. Variables that define the spatial relations between neighboring grids and cells can be found in `amr_constants`, for example `lll` and `mmm`, and in the routines in *amr/nbors_utils.f90*. They are used, for example, in the neighbor-searching routines (see further). simply want to access all neighbors, and the spatial order is irrelevant.

Exercise

Given the index of a cell `icell`, how can we obtain the index of the grid to which it belongs?

Solution

```
! Get the cell's position in its grid, that is the
! index ind, between 1 and twotondim.
ind=(icell-ncoarse-1)/ngridmax+1  !integer devision

! Convert the cell's index to the index of its grid
iskip=ncoarse+(ind-1)*ngridmax
igrid=icell-iskip
```

3. Linked list variables

The grids making up the computational domain are stored in a linked list using an ordering recipe chosen by the user. For more on ordering, see the chapter on Domain decomposition.

A linked list is a chain of individual elements where each element knows about the next (index array `next`) and the previous element (index array `prev`) in the list. In RAMSES, these variables are defined in the module `amr_commons` and allocated in the subroutine `init_amr`:

```
integer ,allocatable,dimension(:)  ::next    ! next grid in list
integer ,allocatable,dimension(:)  ::prev    ! previous grid in list
```

```
allocate(next  (1:ngridmax))
allocate(prev  (1:ngridmax))
```

This makes it easy to insert new elements at any location, when a grid is created as a result of AMR refinement. Grids can also easily be removed when de-refinement occurs. A grid can be removed by connecting its previous element to the next element. Remark that these are *grid*-based arrays.

A linked list is stored for each MPI process (cpu) and AMR level. RAMSES keeps track of

- `headl`: the index of the head, i.e. the first element in the list
- `taill`: the index of the tail, i.e the last element in the list
- `numbl`: the number of elements in the list

```
! Pointers for each level linked list
integer,allocatable,dimension(:,:)::headl ! Head grid in the level
integer,allocatable,dimension(:,:)::taill ! Tail grid in the level
integer,allocatable,dimension(:,:)::numbl ! Number of grids in the
level
```

```
! Allocate linked list for each level
allocate(headl(1:ncpu,1:nlevelmax))
allocate(taill(1:ncpu,1:nlevelmax))
allocate(numbl(1:ncpu,1:nlevelmax))
```

At the start of the simulation, `ngridmax` grids are allocated. As mentioned before, this parameter needs to be set in the namelist. These `ngridmax` grids are divided amongst the following three linked lists:

- headl, taill and numbl: the main computational domain inside an mpi process
- headb, tailb, numbb: physical boundary grids around the computational domain
- headf,tailf,numbf: unused grids (free memory)

A grid can only be part of **one** list at a time.

Exercise:

Given how the linked list is stored, write some code to remove a grid `igrid` from the linked list of refinement level `ilevel` in the MPI domain `icpu`. Think about all the possible positions in the linked list the grid could be.

Solution

```
if(prev(igrid).ne.0) then
   if(next(igrid).ne.0)then
      ! grid is somewhere in the middle of the list
      next(prev(igrid))=next(igrid)
      prev(next(igrid))=prev(igrid)
   else
      ! grid is at the tail of the list
      next(prev(igrid))=0
      taill(icpu,ilevel)=prev(igrid)
   end if
else
   if(next(igrid).ne.0)then
      ! grid is at the head of the list
      prev(next(igrid))=0
      headl(icpu,ilevel)=next(igrid)
   else
      ! grid is the only item in the list
      headl(icpu,ilevel)=0
      taill(icpu,ilevel)=0
   end if
end if
numbl(icpu,ilevel)=numbl(icpu,ilevel)-1
```

The code for removing grids from the linked list can be found in the subroutine `kill_grid`, in the file `amr/refine_utils.f90`. In this routine, a specified number of grids are removed and their corresponding variables reset to zero. We go into this further in the section on refinement.

Exercise

Analogously to the previous exercise, write code to add a grid `igrid` to the end of the linked list of refinement level `ilevel` in the MPI domain `icpu`.

Solution

```
if(numbl(icpu,ilevel)>0)then
   next(igrid)=0
   prev(igrid)=taill(icpu,ilevel)
   next(taill(icpu,ilevel))=igrid
   taill(icpu,ilevel)=igrid
   numbl(icpu,ilevel)=numbl(icpu,ilevel)+1
else
   next(igrid)=0
   prev(igrid)=0
   headl(icpu,ilevel)=igrid
   taill(icpu,ilevel)=igrid
   numbl(icpu,ilevel)=1
end if
```

The subroutine `make_grid_fine` in the file `amr/refine_utils.f90` handles adding new grids. This routine does various things (see section refinement). The part that updates the linked list can be found under the comment `!Connect news grids to level ilevel linked list`. Remark that a seperate routine exists for adding cells at level 1: `make_grid_coarse`. The last code block of this routine handles the linked list update.

4. The variables `active` and `boundary`

One way to access the grids for processing them, is to simply iterate throught the linked list by starting at the `headl` and following `next`. This is however not always practical. Instead, the grid indices are gathered in advance, by iterating through the linked lists and storing them in the variable `active`. An equivalant exists for the grids in the physical boundary, named `boundary`. These variables are defined in `amr_commons`:

```
type(communicator),allocatable,dimension(:)  ::active    !
1:nlevelmax
type(communicator),allocatable,dimension(:,:)::boundary  !
1:MAXBOUND,1:nlevelmax
```

They are arrays of a costume defined data structure called `communicator`. Derived types in fortran are analogous to structs in C/C++. To access its members, the synthax `%` is used. A communicator has various fields, but for keeping track of the grids only two are used:

```
type communicator
  integer                              ::ngrid   ! number of grids
  integer     ,dimension(:)  ,pointer::igrid   ! list of grid indices
  ...
end type communicator
```

(More on communicators in the Chapter on MPI communication). This data structure contain a list of the index of each grid, as found by iterating over the linked grid list (`head`, `next`).

The array `active` holds a communicator for each AMR level of the current MPI domain with ID `myid`. Each communicator the contain a list of the index of each grid, as found by iterating over the linked grid list using `headl` and `next`. The array `boundary` is similar, but instead keeps track of the different physical boundary regions for each level. Its contents is gathered by starting at `headb`.

Exercise

Write code to fill `active(ilevel)` of the current MPI domain by iterating through the corresponding linked list.

Solution

```
ncache=numbl(myid,ilevel)
! Reset old communicator
if(active(ilevel)%ngrid>0)then
   active(ilevel)%ngrid=0
   deallocate(active(ilevel)%igrid)
end if
if(ncache>0)then
   ! Allocate grid index to new communicator
   active(ilevel)%ngrid=ncache
   allocate(active(ilevel)%igrid(1:ncache))
   ! Gather all grids
   igrid=headl(myid,ilevel)
   do jgrid=1,numbl(myid,ilevel)
      active(ilevel)%igrid(jgrid)=igrid
      igrid=next(igrid)
   end do
end if
```

The arrays `active` and `boundary` are synchronised with the linked list in the subroutine `build_comm` in `virtual_boundaries.f90`, which builds the communication structure for a specified AMR level.

5. Processing grids and cells

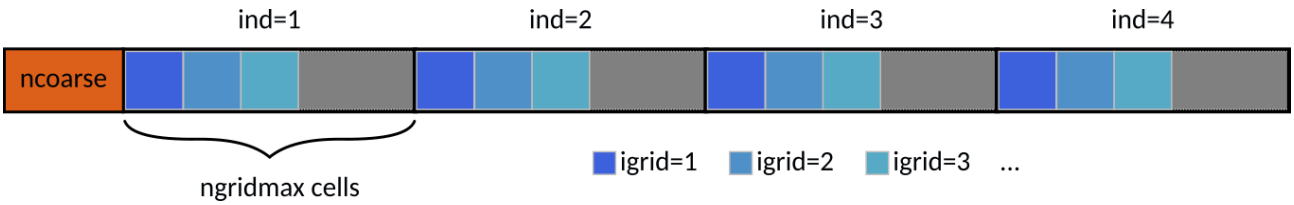
Iterating through grids

To process grids, we can now simply use `active` which contains the grids that are actually in use. To iterate through the grids, use the following straighforward code pattern:

```
do i=1,active(ilevel)%ngrid
   some_grid_array(active(ilevel)%igrid(i))=something
end do
```

Iterating through cells

Most of the arrays we want to process are however cell-based instead of grid-based. Remember that for each grid, there are 2\(^\mathrm{ndim}\) cells stored according to the pattern:



A variable `iskip` gives the number of elements to skip when looping over the grids and processing all cells at position `ind`:

```
iskip=ncoarse+(ind-1)*ngridmax
```

A commonly found loop pattern in the code is then:

```
do ind=1,twotondim
   iskip=ncoarse+(ind-1)*ngridmax
   do i=1,active(ilevel)%ngrid
      some_cell_array(active(ilevel)%igrid(i)+iskip)=something
   end do
end do
```

where the outer loop goes over the position of the cells inside the grids, while the inner loops go over the grids. When accessing multiple arrays, the cell indices are typically calculated in advance (for the current vector sweep, see next section). An example:

```
do ind=1,twotondim
  ! gather cell indices
  iskip=ncoarse+(ind-1)*ngridmax
  do i=1,ngrid
    ind_cell(i)=iskip+ind_grid(i)
  end do
  ! Compute pressure from temperature and density
  do i=1,ngrid
    uold(ind_cell(i),neul)=uold(ind_cell(i),1)*uold(ind_cell(i),neul)
  end do
end do
```

Nvector sweeps

Nowadays, CPUs are able to operate on multiple values at once that are located in neighboring memory locations. This only works if the exact same operation is to be executed, that is without if-else branching. The memory layout of the AMR-tree can however be complex, possibly leading to

unfavorable memory access-patterns. To circumvent this problem, explicit vectorization using the compile-time parameter `NVECTOR` is build-in in RAMSES.

An standard loop structure in RAMSES then looks like this:

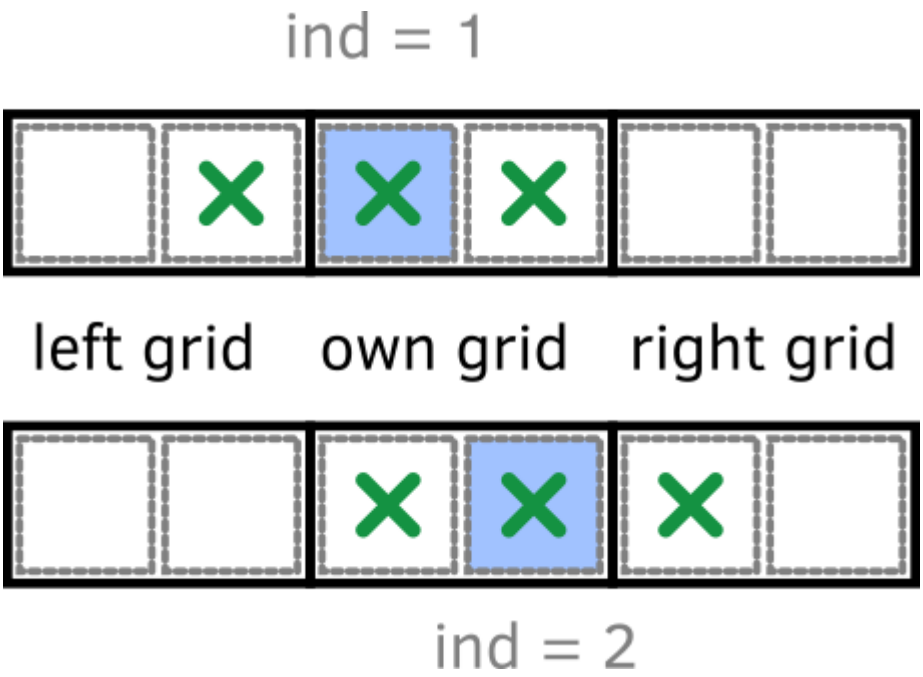
```
! Loop over active grids by vector sweeps
ncache=active(ilevel)%ngrid
do igrid=1,ncache,nvector
   ngrid=MIN(nvector,ncache-igrid+1)
   do i=1,ngrid
      ind_grid(i)=active(ilevel)%igrid(igrid+i-1)
   end do
   call calculation_routine1(ind_grid,ngrid,ilevel)
end do
```

This is going to gather the indices of `nvector` grids before sending them off to a calculation routine. An example can be found for the hydro solver, where in the routine `godunov_fine` grids are send to the routine `godfine1`, which is the entry point for the actual calculation.

The size of nvector has a strong impact on code performance. A large nvector may lead to a better vectorization rate, but small nvector values are better for hardware cache performance. The best value is dependent in the machine and the type of simulation.

6. Finding neighbors

Routines to find neighboring cells and grids are implemented in the file *amr/nbor_utils.f90*. While in most cases, we can simply make use of these routines, it is insightfull to understand how they work. As an exercise, we will go through the process of finding neighbors in 1D, where two configurations are possible:



Exercise (part 1):

In the case of 1D, what are the steps to find the neighboring cells of a given cell? We want the center cell to be included in the list of neighbors. The input of the routine will be the index of the cell.

Solution

- Step 1: Determine the position of the cell in its grid.
- Step 2: Determine the index of the grid to which the cell belongs.
- Step 3: Get the neighboring grids of the grid found in step 2.
- Step 4: Using the position found in step 1, determine which of the grids found in step 3 actually contain the neighbor cells.
- Step 5: Using the position found in step 1, find for each neighbor cell the position in its grid (the grids found in step 4)

Exercise (part 2):

Write (pseudo-)code to execute each of the steps found in part 1 of the exercise.

Exercise (part 3):

Take a look at the different routines that are implemented in the file *nbor_utils.f90*. Which routine serves which purpose? Which routine could be used to get the neighboring cells in 1D? Compare your (pseudo-)code to what is implemented in *nbor_utils.f90*

Solution

The 1D case is a bit of a special case, because there is no distinction between getting the direct neighbors or also including diagonal neighbors (since there are no diagonals...). You could either use the implementation the routine `getnborgrids` or `get3cubefather`.

Exercise (part 4):

What would change in the case of 2D? Make a distinction between retrieving the 4 direct neighbors and including also the diagonal neighbors.

Particles

In this chapter, we will cover the implementation of particles in RAMSES, namely:

- How particles are represented in the code
- How we can we navigate the list of particles in the code
- How particles interact with the grid

Contents

1. Particles implementation in RAMSES

1.1 Particle arrays

The core of the particle-related code is stored in the `pm` folder, where PM stands for "particle-mesh" (i.e., the part of the code responsible for interactions between the particles and the mesh). As for other parts of the code, the majority of the important variables are stored in a *commons* module. For the particles, the module is `pm_commons`, found in `pm/pm_commons.f90`. Additional variables are stored in the corresponding `pm_parameters` module, located in `pm/pm_parameters.f90`.

Most quantities related to particles are stored in large **global arrays**, with in general one variable per array. The size of these arrays is fixed by the maximum number of particles allowed for the run, set by the parameter `npartmax` defined in the `pm_parameters` module. We will cover how to set this parameter in the section on the namelist. Just like `ngridmax` for the grids, the number of particles can either be set directly through `npartmax` (which sets the maximum number of particles *per MPI process*), or through the *total* number of particles across all MPI processes, `nparttot`. These two are related through `nparttot = npartmax*ncpu`.

⚠ Warning

It is usually more convenient to work with `npartmax` when developing code, as it is the actual size of the arrays available for each MPI process. However, when running a simulation with a fixed number of particles (e.g., a dark matter-only simulation), `nparttot` might be more convenient.

The particle masses are stored in the `mp` variable, which is a one-dimensional array of size `npartmax`. The positions and velocity are stored in `xp` and `vp`, respectively: these are two-dimensional arrays, with size `(npartmax, ndim)` where `ndim` is the number of dimensions (usually `ndim=3`). Similar arrays exist for the birth time, the metallicity, or the AMR level at which the particles are living.

All these arrays are defined in `pm/pm_commons.f90`:

```
! Particles related arrays
real(dp),allocatable,dimension(:,:)  ::xp       ! Positions
real(dp),allocatable,dimension(:,:)  ::vp       ! Velocities
real(dp),allocatable,dimension(:)    ::mp       ! Masses
...
real(dp),allocatable,dimension(:)    ::tp       ! Birth epoch
real(dp),allocatable,dimension(:)    ::zp       ! Birth metallicity
...
integer ,allocatable,dimension(:)    ::levelp   ! Current level of
particle
```

1.2 Different types of particles

While RAMSES was originally (Teyssier, 2002) designed to work with only DM particles, the code was quickly extended to star formation, galactic winds, black holes, etc. All these developments have relied, to some extent, on "particles": for example, stars are modelled as particles representing a stellar population with a unique age and metallicity.

In practice, the choice in RAMSES is that all these "particles" are implemented using the same global arrays: `xp` for positions, `mp` for mass, etc. However, this requires some adjustments to identify which particle correspond to what type of thing. For example, some physical models such as supernova explosions only apply to star particles, and we therefore need a simple way to identify which particles corresponds to stars, and which don't.

Since 2017, RAMSES implements these as *particle types* (similar, for example, to Gadget/AREPO particle *types*), stored in the `typep` array. Contrary to other arrays we have seen so far, `typep` is not an *simple* array with real/integer values, instead it is an array of `part_t`, a *derived type* that contains two variables, defined at the end of `pm/pm_parameters.f90`:

- a `family` variable
- a `tag` variable

The `typep` variable is defined in `pm/pm_commons.f90`:

```
type(part_t), allocatable, dimension(:) :: typep  ! Particle type
array
```

and the type itself is defined in `pm/pm_parameters.f90`:

```
type part_t
   ! We store these two things contiguously in memory
   ! because they are fetched at similar times
   integer(1) :: family
   integer(1) :: tag
end type part_t
```

The `family` variable stores what the particle represent:

Family	Value	Physical meaning
`FAM_DM`	1	DM particle
`FAM_STAR`	2	Star particle
`FAM_CLOUD`	3	“Cloud” particle, used for some black-hole implementations
`FAM_DEBRIS`	4	“Debris” particle, used for some supernovae implementations
`FAM_OTHER`	5	“Other” type of particle, defined as a catch-all value
`FAM_UNDEF`	127	Used for “undefined” type, internally

In addition, six other values are used for *tracer* particles, mirroring values in the table above (e.g., `-1` for DM tracers, `-2` for star tracers, etc), as well as a `FAM_TRACER_GAS=0` value for gas tracer particles.

The particle tags are currently not commonly used, but allow flexibility to have different types of *the same kind* of particles. For example, one could decide to have Pop III stars and normal Pop II stars implemented at the same time: both would be considered as *stars* by the code (`FAM_STAR` family), but still identified independently with their own tag. This can be useful either for post-processing (e.g., “make a map of all Pop III stars”), or even for specific physical models (e.g., Pop III stars could have a different radiative output or stellar evolution model).

In practice, rather than testing directly the `typep` value of a particle, the best practice is to use the *functions* defined at the end of the `pm_commons` module to test whether a particle is of a given type. For example, the `is_star` function returns `True` if the particle is a star, and `False`

otherwise. This allows more concise code, as we can write multiple tests in one function. For example, `is_tracer` returns `True` if a particle is a tracer particle, *irrespective of what kind of tracer*.

NB: when implementing new types of particles, make sure to use the right `family` (if one already exists), use or define the appropriate `tag`, and (if necessary) add the right tester function.

1.3 Initialising particles

In the code, the global particle arrays are *defined* in the `pm_commons` module, but are not *allocated* there: indeed, `npartmax` is a runtime parameter, so it cannot be fixed once and for all in the code.

Instead, all the particle-related arrays are allocated when the simulation is initialised, in the `init_part` routine, defined in `pm/init_part.f90`. This is done through the following code:

```
! Allocate particle variables
allocate(xp    (npartmax,ndim))
allocate(vp    (npartmax,ndim))
allocate(mp    (npartmax))
```

By default, all these arrays are set to 0 prior to proper initialisation. Then, depending on whether the code is starting from initial conditions or re-starting from a previous output, the variables are set to the right values.

When **starting from initial conditions**, three formats are supported by default: ASCII files, GRAFIC files, and GADGET files. We will not cover the inner details of each of these formats, but the corresponding subroutines (`load_ascii`, `load_grafic`, and `load_gadget`) can all be found in `pm/init_part.f90`. The easiest to understand is `load_ascii`, but they all work in similar ways:

1. we read the input file header to deal with units, box size, etc

2. we read all arrays in the input file and count the number of particles

3. (we communicate information across MPI processes)

4. while iterating over particles, we fill the global arrays (`xp`, `vp`, `mp`, etc).

Exercise

Can you explain how the `load_ascii` and `load_grafic` subroutines work? For example, can you identify where the positions are read, and where they are set in the `xp` array?

In the case of a **restart**, the process is roughly the same, except that each MPI process opens the corresponding file from the last output. For example, if we restart from output 42, the MPI process 37 will open the file `output_00042/part_00042.out00037`.

⚠ Warning

This strategy to initialise the data at restart is part of the reason why RAMSES enforces that the number of MPI processes stays identical when restarting a simulation.

Exercise

Identify where, in `init_part.f90`, the particle output files are opened.

Solution

Look around `if(nrestart>0) then`:

```
call title(nrestart,nchar)
fileloc='output_'//TRIM(nchar)//'/part_'//TRIM(nchar)//'.out'
...
call title(myid,nchar)
fileloc=TRIM(fileloc)//TRIM(nchar)
...
open(unit=ilun,file=fileloc,form='unformatted')
```

The `title` function transforms the output number `nrestart` or the MPI process ID `myid` to a string padded with five `0`.

Once the file is open, we first read a header and then, each array stored in the output will be read. For example, the positions are read as:

```
! Read position
allocate(xdp(1:npart2))
do idim=1,ndim
   read(ilun)xdp
   xp(1:npart2,idim)=xdp
end do
```

Let's explain how these lines work. First, we allocate a temporary array (`xdp`) with size `npart2` corresponding to the number of particles *in the file*. Then, for each dimension, we read the position from the file to the array `xdp`, and we fill the position array `xp` with the array we just read.

Note that we use and re-use several temporary arrays (`xdp`, `isp8`, `isp`, `ii1`), each corresponding to a data type.

> **⚠ Warning**
>
> **Careful:** if you change the output format for the particles, you will need to change the section of `init_part.f90` that deals with restarts.

2. Navigating particle arrays with linked lists

Because of the AMR structure of RAMSES, we need to store particles in a structure that can efficiently be connected to the grid. As the grid evolves with time, this structure also needs to evolve with time. The "natural" solution for this is to use a linked-list for the particles, just like what is done for the grid.

The key idea here is that particles living in the same grid are linked together with a linked-list that is defined *per-grid*, and we use the grid structure with its own linked list to connect all the particles together.

As this leads to a tree structure, the bulk of the code dealing with the particle list is in the aptly named `pm/particle_tree.f90` file.

2.1 Particle linked list structure

In each grid, we have a particle linked list defined using the following variables, defined in `pm/pm_commons.f90`:

```
integer ,allocatable,dimension(:)    ::headp    ! Head particle in
grid
integer ,allocatable,dimension(:)    ::tailp    ! Tail particle in
grid
integer ,allocatable,dimension(:)    ::numbp    ! Number of particles
in grid
```

They are allocated in the `init_amr` subroutine (in `amr/init_amr.f90`), if the run parameter flag `pic` (particle in cell) is set:

```
if(pic)then
   allocate(headp(1:ngridmax))
   allocate(tailp(1:ngridmax))
   allocate(numbp(1:ngridmax))
   headp=0; tailp=0; numbp=0
endif
```

Note that if `pic` is *not* set, there are no particles, so we don't have to do any of this.

As required for a linked list, for each particle the next and previous particle in the list is stored. These arrays are defined in `pm/pm_commons.f90`:

```
integer ,allocatable,dimension(:)    ::nextp    ! Next particle in
list
integer ,allocatable,dimension(:)    ::prevp    ! Previous particle
in list
```

They are allocated in `pm/init_part.f90`:

```
allocate(nextp (npartmax))
allocate(prevp (npartmax))
```

One specificity of particles is that they can move freely through the AMR grid: they can enter and exit cells during a timestep, and (because of the timestepping strategy), they can move to grids that live on different timesteps.

Dealing with this requires careful consideration, and the subroutines dealing with this are found in `pm/particle_tree.f90`, with the most important one being `make_tree_fine`. It essentially detaches particles that have moved from their original parent grid, and reattaches them to the grid in which they are now located.

Similarly, when a grid is refined or derefined, special care needs to be taken of the particles: the routines doing this are, respectively, `kill_tree_fine` and `merge_tree_fine`.

Finally, when particles move across processor boundaries (see the section on MPI communications), the routine `virtual_tree_fine` deals with the linked list structure.

2.2 Iterating over particles

The *particle* and *grid* linked lists structures are used to iterate over particles.

The following code block represents the usual way of looping over all particles in a given MPI process:

```
do icpu=1,ncpu
! Loop over cpus
   igrid=headl(icpu,ilevel)
   ! Loop over grids
   do jgrid=1,numbl(icpu,ilevel)
      npart1=numbp(igrid)  ! Number of particles in the grid
      if(npart1>0)then
         ipart=headp(igrid)
         ! Loop over particles
         do jpart=1,npart1
            ! Save next particle   <--- Very important !!!
            next_part=nextp(ipart)

            !----
            ! Do something with particle ipart
            !----

            ipart=next_part  ! Go to next particle
         end do
      endif
      igrid=next(igrid)   ! Go to next grid
   end do
end do
```

Let's unravel this and start from the `igrid = headl(icpu, ilevel)` line. This initialises the variable `igrid` to the index of the first element of the list of *grids* at the current level `ilevel`.

Then, the section

```
do jgrid=1, numbl(icpu, ilevel)
    npart1=numbp(igrid)
    ...
    igrid=next(igrid)
end do
```

loops over all grids at the current level. The variable `npart1=numbp(igrid)` stores the number of particles that exist in grid indexed by `igrid`. Note here that when we first enter the loop, `igrid` is defined from the `headl` access, but at the end of the loop, we go to the next grid in the list, determined by `next(igrid)`.

If there is at least one particle in the grid (i.e., if `npart1>0`), we can loop over the particles in the grid with the particle linked-list:

```
ipart=headp(igrid)
do jpart=1,npart1
    next_part=nextp(ipart)
    ...
```

```
    ipart=next_part
end do
```

This is done in many places in the code, and looking for the `Very important !!!` comment is likely to return many of these places. You can guess how people usually go about writing this loop.

2.3 Adding and deleting new particles

Multiple routines in RAMSES need to create or destroy particles. Once again, the most typical example is the star formation routine: when new stars are formed, we need to "spawn" new particles and update the linked lists accordingly. Conversely, it can happen that particles are destroyed (for example, the "cloud particles" surrounding black holes when two black holes merge).

This is handled by routines found in `pm/add_list.f90` and `pm/remove_list.f90`. Let's quickly review how they work together.

When spawning new particles, we need to insert them somewhere in the particle linked list. This is done relatively easily thanks to the linked list structure: we just need to add the particle index after the current last particle in the `nextp` array. This is essentially what the `add_list` subroutine does in `pm/add_list.f90`:

```
nextp(tailp(ind_grid(j))) = ind_part(j)
```

Here, `tailp(ind_grid(j))` indicates the tail of the particle linked list in the grid `ind_grid(j)`, and we define the `nextp` particle as the one with index `ind_part(j)`.

To make sure that there is enough space in the particle arrays, we need to use the `remove_free` subroutine of `pm/remove_list.f90`, which essentially reserves a block of the particle arrays for the new particles.

As a result, the correct way to update the linked lists when adding new particles is **always** to *first* call `remove_free`, then add the particles with `add_list`. This is for example done in the star formation routine in `pm/star_formation.f90`:

```
! Update linked list for stars
call remove_free(ind_part,nnew)
call add_list(ind_part,ind_grid_new,ok_new,nnew)
```

3. Cloud-in-Cell scheme

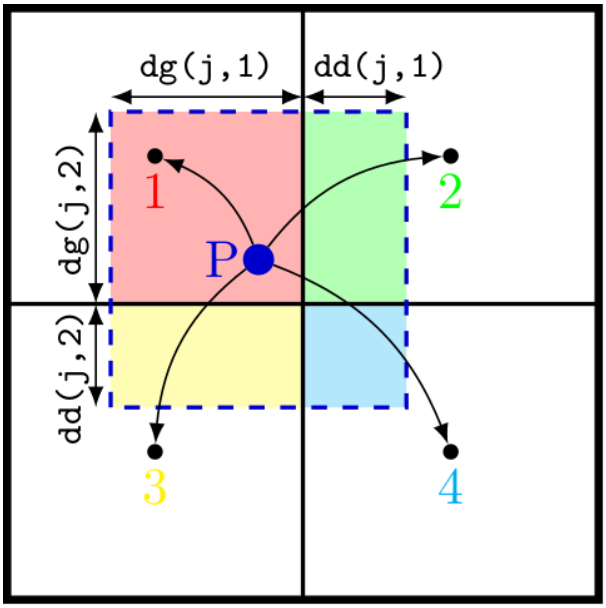
Particles interact with the rest of the simulation in essentially two ways: either through direct exchange of mass, energy, momentum, radiation (...) with the grid, or through their gravitational influence. The former will be discussed in the section on subgrid modelling, and we will now focus on the way particles interact with gravity.

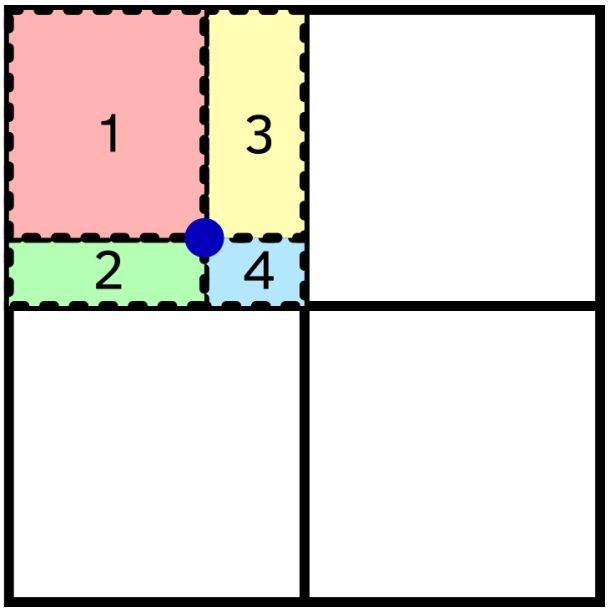
CIC is a general mapping between grids and particles: it can be used beyond gravity (e.g., for sink particles accretion). As discussed in the section on subgrid modelling, other assignment schemes are used for feedback, for example.

3.1 Overview

As already discussed in the general review of the code, (massive) particles act as a source term for gravity: they contribute to the density field \(\rho\). This field is represented in the code by the `rho` array, which is defined *for each cell*. In order for particles to contribute to the density field, we need a method to distribute their mass on the grid.

RAMSES does this through the **cloud-in-cell** scheme, or **CIC**. The idea is to distribute the mass of each particle in a cloud with cubic shape (in 3D): unless the particle is perfectly at the centre of a cell, its cloud will overlap with other cells, and the mass of the particle will be spread over those cells depending on the volume fraction of the overlapping region.





In more details: a particle that "lives" at a level `ilevel` will be represented as a cube with side length \(\Delta x = L/2^{\texttt{ilevel}}\), and its volume will therefore be \(\Delta x^{\texttt{ndim}}\), usually \(\Delta x^3\). We can show that the cloud will overlap with \(2^{\texttt{ndim}}\) grid cells. In order to compute the contribution of a particle to each of the overlapping cells, we need:

- the indices of the cells with which the particle overlaps
- the overlapping sub-volume of the particle cloud with these cells.

In practice, the second part is done before the first: the sub-volumes are calculated as rectangular cuboids, using the distances to the cell boundaries `dd` and `dg` (respectively for *distance droite* and *distance gauche*).

3.2 Walking through `cic_amr`

To better understand how the CIC scheme is applied, let's look at the `cic_amr` routine that can be found in the `pm/rho_fine.f90` file.

After some book-keeping, we recover the neighbouring father cells with

```
call get3cubefather(ind_cell,nbors_father_cells,ng,ilevel)
```

This will be needed to get the grid index of all the neighbours.

We then rescale all the positions at the current level to get the position of the 3×3×3 neighbour grids (in 3D), rescaled to be between 0 and 6 in each dimension.

```
! Rescale particle position at level ilevel
do idim=1,ndim
   do j=1,np
      x(j,idim)=xp(ind_part(j),idim)/scale+skip_loc(idim)
      x(j,idim)=x(j,idim)-x0(ind_grid_part(j),idim)
      x(j,idim)=x(j,idim)/dx
   end do
end do
```

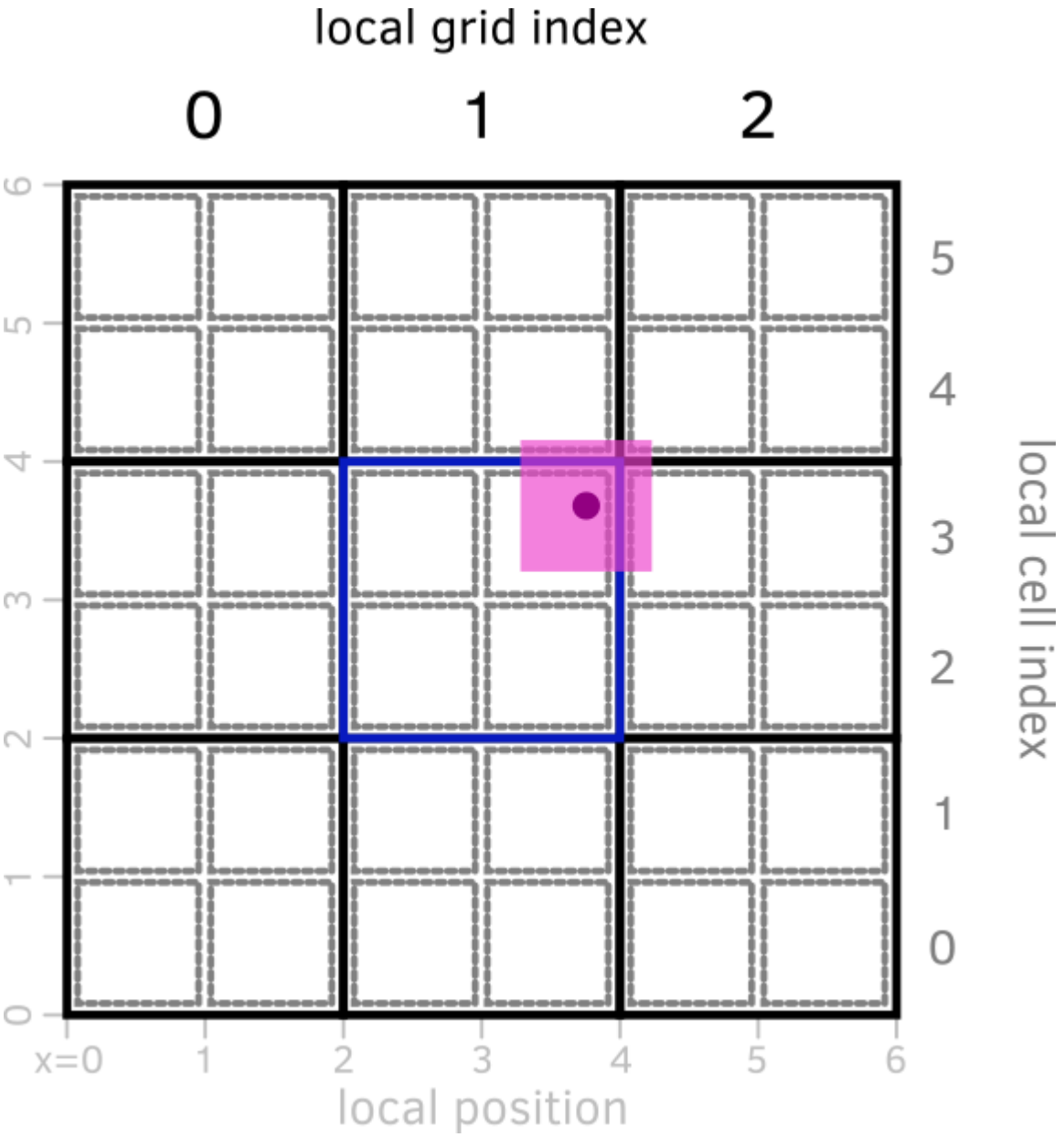
This is then the time to get the particle properties that we may want to dump on the grid, for example the mass of non-tracer particles. This is done by reading the particle mass `mp` in the `cic_amr` routine, but we could do the same thing for other extensive quantities (e.g., metal mass).

After some extra checks, we then compute `dd` and `dg` along each dimension. The volume of each of the sub-volumes is computed from the values of `dg` and `dd`. For example, in two dimensions:

```
#if NDIM==2
  do j=1,np
     vol(j,1)=dg(j,1)*dg(j,2)
     vol(j,2)=dd(j,1)*dg(j,2)
     vol(j,3)=dg(j,1)*dd(j,2)
     vol(j,4)=dd(j,1)*dd(j,2)
  end do
#endif
```

Particle `j` will have a fraction of its cloud volume in cell #1 given by `dg(j,1)*dg(j,2)`, the fraction in cell #2 given by `dd(j,1)*dg(j,2)`, and so on. All these are stored in the array `vol(:,:)`, where the first dimension corresponds to the particle, and the second to the sub-volume.

Once this is all computed, we need to assign these sub-volumes to the relevant cells.



First, we need to identify the local index of the parent grid. During the `dd` and `dg` computation, we get the index of the left and right boundaries in `ig` and `id`, respectively. As the parent grid lives on level `ilevel-1`, the corresponding local) grid indices (`igg` and `igd`, for index grid *gauche* and index grid *droite*) will be halved:

```
do idim=1,ndim
   do j=1,np
      igg(j,idim)=ig(j,idim)/2
      igd(j,idim)=id(j,idim)/2
   end do
end do
```

On the figure above, this corresponds to computing the local grid index from the local position.

From there, we compute an index `kg` for each of the 8 sub-volumes, which is used to determine the *global* grid index `igrid` of that parent grid:

```
do ind=1,twotondim
   do j=1,np

igrid(j,ind)=son(nbors_father_cells(ind_grid_part(j),kg(j,ind)))
   end do
end do
```

We then need to determine which of the 8 cells belonging to the parent grid is relevant, i.e., which is the local `ind` (as defined in the section on AMR structure). This is done again by index arithmetics, depending on the values of `ig`, `id`, `igg`, and `igd`, and yields the array `icell(:,:)` where the first dimension corresponds to the particle on which we are working, and the second to the 8 sub-volumes.

We *finally* can compute the global cell index as

```
! Compute parent cell adress
do ind=1,twotondim
   do j=1,np
      indp(j,ind)=ncoarse+(icell(j,ind)-1)*ngridmax+igrid(j,ind)
   end do
end do
```

Once all of this is done, we can loop over the particles to do something with the CIC scheme. For example, in `cic_amr`, we compute the contribution of the particles to the density array `rho`: we first define the “mass fraction” `vol2`:

```
do j=1,np
   ok(j)=(igrid(j,ind)>0).and.is_not_tracer(fam(j))
   vol2(j)=mmm(j)*vol(j,ind)/vol_loc
end do
```

and then increment `rho` with it:

```
do j=1,np
   if(ok(j))then
      rho(indp(j,ind))=rho(indp(j,ind))+vol2(j)
   end if
end do
```

3.3 Details of the CIC density calculation

If we look more carefully at the main loop where the density field is computed, we can note several things.

```
if(cic_levelmax==0.or.ilevel<=cic_levelmax)then
   do j=1,np
      if(ok(j))then
         rho(indp(j,ind))=rho(indp(j,ind))+vol2(j)
      end if
   end do
else if(ilevel>cic_levelmax)then
   do j=1,np
      ! check for non-DM (and non-tracer)
      if ( ok(j) .and. is_not_DM(fam(j)) ) then
         rho(indp(j,ind))=rho(indp(j,ind))+vol2(j)
      end if
   end do
endif

if(ilevel==cic_levelmax)then
   do j=1,np
      ! check for DM
      if ( ok(j) .and. is_DM(fam(j)) ) then
         rho_top(indp(j,ind))=rho_top(indp(j,ind))+vol2(j)
      end if
   end do
endif
```

First of all, for DM particles, CIC assignment is only done if the level is below `cic_levelmax`. Otherwise, a different array, `rho_top` is updated. This is a way of dealing with the fact that DM particles are usually much more massive than the baryons in the simulation. In practice, this "smooths" the DM particles to the level `cic_levelmax`, which is a free namelist parameter.

⚠ Warning

A good way of choosing `cic_levelmax` is to run a DM-only simulation, and see what is the maximum level of refinement that is reached. This will be the natural smoothing level for the DM particles, and avoids dumping all the DM particle mass in a small cell (whose size is determined by its baryonic content rather than DM content).

This is note done for non-DM particles: indeed, they are expected to "live" at the finest grid level.

Second, we can also note that CIC is used in several places.

- `cic_amr` called from `rho_from_current_level`, itself called in `rho_fine`, which calculates the density field of the particles.
- `cic_cell` called from `cic_from_multipole` which represents the gas cells as *pseudo-particles* to calculate the gas density field (used as input for the gravity calculation) in the same way as the particles. This has some advantages.
- `cic_only` called from `rho_only_level` in the clumpfinder, to calculate the density field on which to perform the clump finding.

While all these routines are pretty similar, this leads to a lot of code duplication, which can *sometimes* be hard to maintain. There is work being done on this, but it takes time.

Also, RAMSES has an alternative to the CIC scheme that is in principle more accurate: the TSC scheme, for *triangular-shaped cloud*. This is a smoother, more expensive, assignment scheme.

Exercise

Go through the code for the TSC scheme. What is different about it?

3.4 Inverse CIC: grid affecting the particles

As we have seen in the gravity section, once the Poisson equation has been solved, we need to update the particle positions and velocities. As the acceleration (from the gravitational field) is only known at the grid position, we need to interpolate it at the particle positions.

This is done using the same type of CIC scheme as for the density calculation, except that instead of changing a quantity on the grid, we read the updated force at the particle location. Just like previously, we need:

- the identity of each neighbouring cell
- the fraction of the particle cloud in each neighbouring cell
- the contribution of each of these cell to the acceleration

Note that special care must be taken when the particle overlaps with cells that are at a different refinement level.

As an exercise, you can go through the `sync` routine called by `synchro_fine` in `pm/synchro_fine.f90` and `move1` called by `move_fine` in `pm/move_fine.f90` to go through the logic.

MPI communications

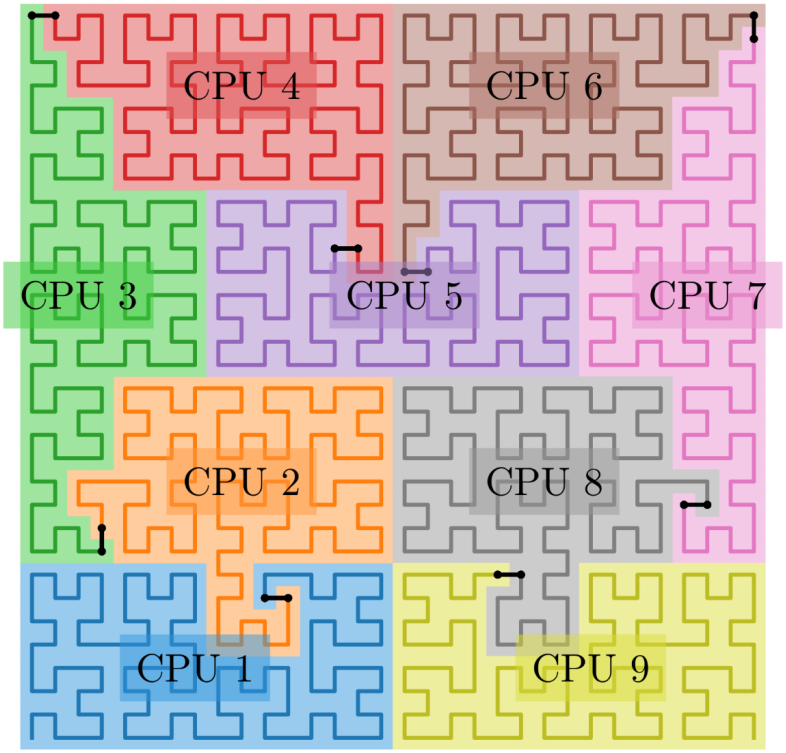
Contents

Domain decomposition over MPI processes

Hilbert curve

Several recipes exist to divide the cells making up the computational box into individual domains to be assigned to MPI processes. The default, and most commonly used in RAMSES, is Hilbert domain decomposition. The decomposition curve will assign to each location in a multi-dimensional space a one-dimensional order. By setting the namelist parameter `ordering` in the `RUN_PARAMS` block, users could choose options other than Hilbert.

An example of the Hilbert curve in a 2D computational domain and the resulting domain decomposition is shown below:



Credits: Cadiou (2019).

The index indicating a cell's position in the 1D Hilbert curve (or any other ordering recipe) is stored in the global array `hilbert_key`, defined in `amr_commons`

```
real(qdp),allocatable,dimension(:)::hilbert_key
```

and allocated in `init_amr`:

```
allocate(hilbert_key(1:ncell)) ! Ordering key
hilbert_key=0.0d0
```

Note that the Hilbert curve is always defined on `levelmax`. This means that when using deep levels of refinement (`levelmax` >19), the value of the index on the curve may be so large that it does not fit into a regular double precision real number. In this case, one should compile with `QUADHILBERT`, which increases the size of the elements of the array `hilbert_key`:

```
#ifdef QUADHILBERT
  integer,parameter::qdp=kind(1.0_16) ! real*16
#else
  integer,parameter::qdp=kind(1.0_8) ! real*8
#endif
```

CPU map

To distribute the cells over the MPI domains, the Hilbert curve is then cut into “equal” part according to some recipe (see further).

The variables keeping track of which cell is in which MPI domain are defined in `amr_commons` and allocated in `init_amr`

```
integer ,allocatable,dimension(:)  ::cpu_map  ! domain decomposition
integer ,allocatable,dimension(:)  ::cpu_map2 ! new domain
decomposition
```

```
! Allocate MPI cell-based arrays
allocate(cpu_map    (1:ncell)) ! Cpu map
allocate(cpu_map2   (1:ncell)) ! New cpu map for load balance
cpu_map=0; cpu_map2=0
```

They are used in the load balancing.

Load balancing

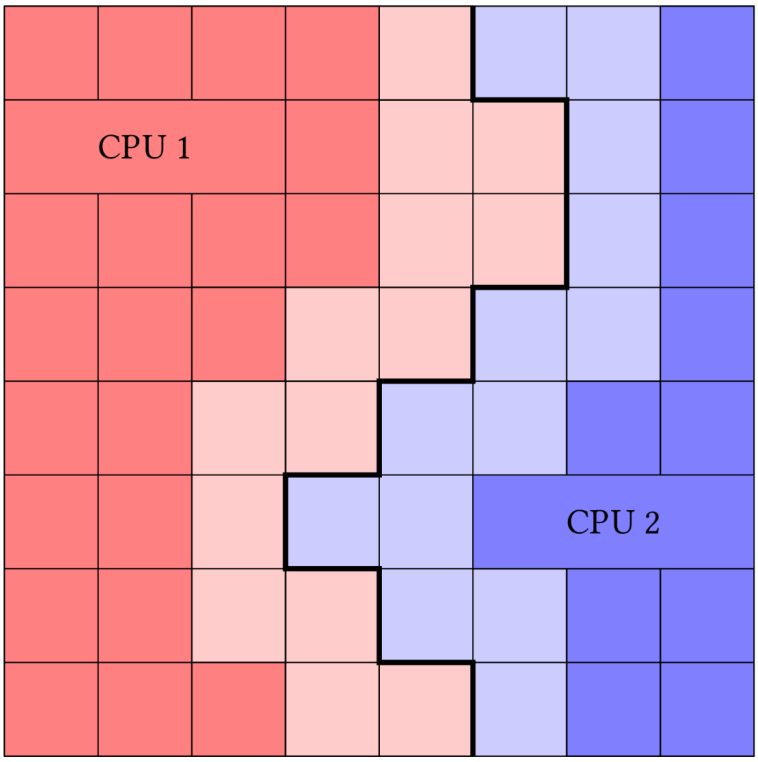
Load balancing is the operation of redistributing the computational load between the MPI domains. The main routine `load_balance` can be found in the file `amr/load_balance.f90`. It goes through the following steps. * Compute new cpu map using the chosen ordering (usually Hilbert) * Expand boundaries to account for new mesh partition * Update physical boundary conditions * Rearrange octs between cpus * Compute new grid number statistics * Set old cpu map to new cpu map * Shrink boundaries around new mesh partition

The computational cost is estimated with a simple heuristic, assuming (by default) the cost to be level-dependent

\[\sum_{\ell=\ell_\mathrm{min}}^{\ell_\mathrm{max}} f_\ell (a * N_{\mathrm{grid},\ell} + b * N_{\mathrm{part},\ell}),\]
where \(f_\ell = \prod_{\ell'=\ell_\mathrm{min}}^{l} N_{\mathrm{subcycle},\ell'}\) is the product of the number of subcycles compared to the base grid (related to adaptive time stepping). \(a\) and \(b\) are *magic numbers*. By default, \(a=80\) and \(b=1\) (i.e., an oct costs 80 more than a particle).

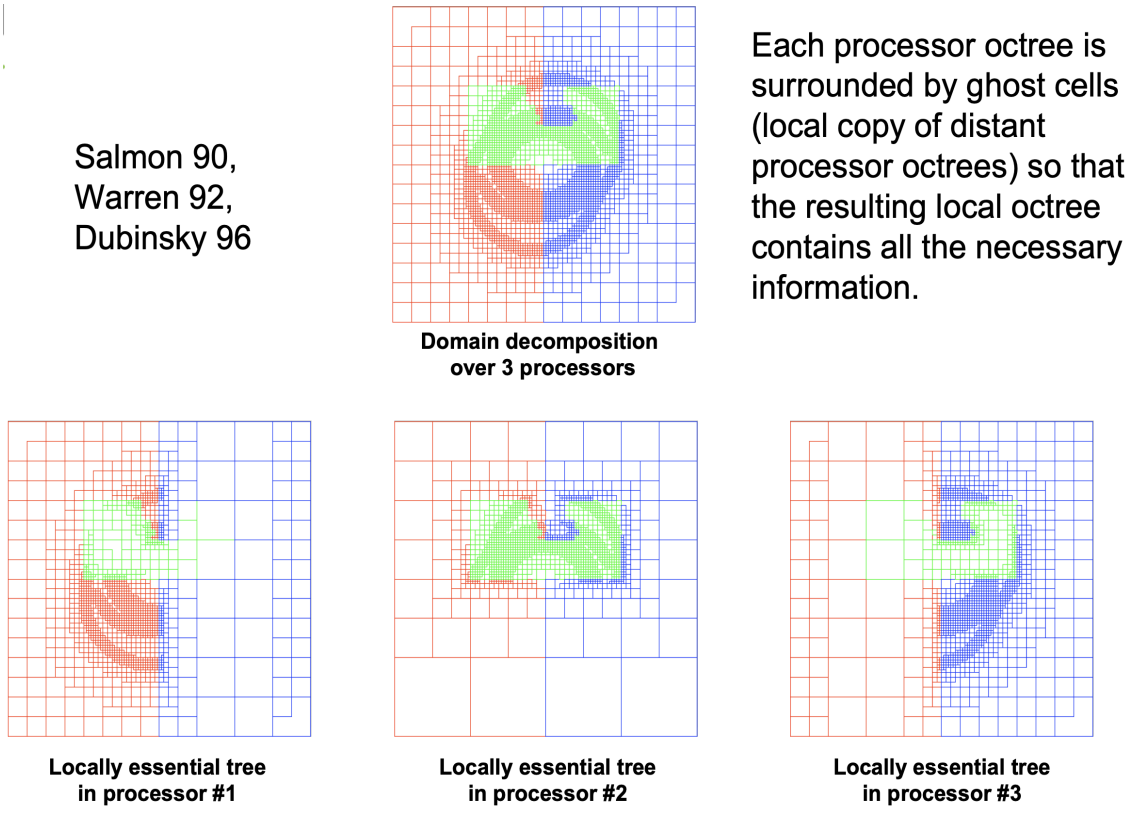
Ghost zones

Example of ghost zones: the light-shaded cells are “known” by both CPU1 and CPU2 in this example, and represent shared *ghost zones*.



Credits: Cadiou (2019).

The ghost zones are replicated on the relevant processors to construct *locally essential trees*, such that each local octree contains all the necessary information for the computations (until communications are needed). The figure below shows an example of ghost zones and corresponding tree stored on different MPI process (from Romain Teyssier lecture):



Credits: Romain Teyssier's lectures.

Communication between MPI domains

Communicator variables

RAMSES defines the **derived type** `communicator`. Derived types in fortran are analogous to structs in C/C++. To access its members, use `%`.

```
! Communication structure
type communicator
  integer                              ::ngrid   ! number of grids
  integer                              ::npart   ! number of particles
  integer     ,dimension(:)  ,pointer::igrid   ! list of grid indices
  integer     ,dimension(:,:),pointer::f       !
  real(kind=8),dimension(:,:),pointer::u       !
  integer(i8b),dimension(:,:),pointer::fp      !
  real(kind=8),dimension(:,:),pointer::up      !
end type communicator
```

In classic RAMSES, four global communicator arrays are defined in `amr_commons`:

```
! Active grid, emission and reception communicators
type(communicator),allocatable,dimension(:)  ::active     !
1:nlevelmax
type(communicator),allocatable,dimension(:,:)::boundary   !
1:MAXBOUND,1:nlevelmax
type(communicator),allocatable,dimension(:,:)::emission   ! 1:ncpu,
1:nlevelmax
type(communicator),allocatable,dimension(:,:)::reception  ! 1:ncpu,
1:nlevelmax
```

Remark that `active` and `boundary` are not for communication, they are simply used to easy book-keeping of the grids. Only the fields ngrid and igrid are used for these (see the chapter on grid data structures).

The arrays `emission` and `reception` are for actual MPI communication. These arrays are altered by the light MPI implementation to have a smaller memory imprint. They are used both for ghostzone and particle communication.

Ghost zone communication

To communicate the values of the different global arrays in the ghost zones, RAMSES uses the routines in `amr/virtual_boundaries.f90`.

- `make_virtual_fine` (**forward** exchange) Fill the ghost zones (virtual grids) of neighboring MPI domains with up-to-date values from inside current MPI domain. Steps:
  - launch asynchronous receives from the other CPUs (send to `reception` communicator)
  - gather the local data that is to be send to the neighboring domains into the `emission` communicator
  - launch asynchronous sends to the other CPUs
  - wait for all receives to complete
  - copy the received data from the reception buffer to the correct index in the global array
  - wait for sends to complete
- `make_virtual_reverse` (**reverse** exchange) Send data from the current MPI domain's virtual grids to real grids of neighboring domains. Contributions are **added** to the field: `xx(igrid) = xx(igrid) + received_value`. This needs to be done after an update that affect neighboring cells has been performed, i.e., the computation of hydro fluxes and computation of the density field using CIC. For the coarsest levels, sychronous SEND/RECV are used. For the finer levels, a simimal asynchronous send/recv pattern is uses as in the forward exchange.

There are two versions of the `make_virtual_fine` and `make_virtual_reverse` subroutines: one for communicating floating point numbers (`_dp`) and one for integers (`_int`). They are nearly identical. The only differences are:

- the field accessed in the emission/reception communicators (`u` for doubles, `f` for integers),
- the MPI types in the SEND/RECV calls (MPI_DOUBLE_PRECISION and MPI_INTEGER respectively).

Exercise

For hydro, which array should be used `uold` and `unew`? It depends on the exchange(reverse versus forward).

Solution

It is linked to `set_unew`

Particle communication

Particles move freely in the computation domain, which means they can travel to other MPI domains. This is handeled by `virtual_tree_fine` in `pm/particle_tree.f90`.

When to communicate?

Communications must be done each time a global array, e.g., `uold`, is modified (i.e., after cooling or feedback), to inform the neighbor process about the change, using `make _virtual_fine`.

Exercise

Find other arrays that are communicated. Why is `make _virtual_reverse` called only for `unew`, `phi` and `rho`?

Solution

`make_virtual_reverse` in only needed when a flux in applied at a MPI domain boundary, or when a particle belonging to on MPI process deposit mass on a neighboring MPI domain. `phi` is used as temporary array in `rho_fine.f90`.

\[\newcommand{\CH} {{\Lambda}} % Cooling/heating function \newcommand{\CHp} {{\Lambda^\prime}} % Time derivative of coolheat function \newcommand{\Cool} {{\mathcal{L}}} % Cooling function \newcommand{\coolU} {\rm{erg} \, \rm{cm}^{3} \, \rm{s}^{-1}} % erg cm3 s-1 \newcommand{\CoolZ} {{\mathcal{L}_Z}}% Metal cooling function \newcommand{\dt} {\Delta t} % Discrete time interval \newcommand{\dtcool} {\Delta t_{\rm{cool}}}% Discrete time interval \newcommand{\dthyd} {\Delta t_{\rm{hydro}}}% Discrete time interval \newcommand{\ele} {{\rm{e}}} \newcommand{\eps} {{\varepsilon}} % Energy density symbol \newcommand{\ergs} {\rm{erg} \, \rm{s}^{-1}} % erg s-1 \newcommand{\eTconv} {\frac{(\gamma-1)\mh}{\rho \kb}} \newcommand{\Heat} {{\mathcal{H}}} % Heating function \newcommand{\heisub} {\rm{He \scriptscriptstyle I}} \newcommand{\heiisub} {\rm{He \scriptscriptstyle II}} \newcommand{\heiiisub} {\rm{He \scriptscriptstyle III}} \newcommand{\hisub} {\rm{H \scriptscriptstyle I}} \newcommand{\hiisub} {\rm{H \scriptscriptstyle II}} \newcommand{\kb} {k_{\rm{B}}} % Bolzmann constant \newcommand{\mh} {m_{\rm{H}}} % Proton mass \newcommand{\nh} {n_{\rm{H}}} \newcommand{\nel} {n_{\rm{e}}} \newcommand{\nhe} {n_{\rm{He}}} \newcommand{\nhei} {n_{\heisub}} \newcommand{\nheii} {n_{\heiisub}} \newcommand{\nheiii} {n_{\heiiisub}} \newcommand{\nhi} {n_{\hisub}} \newcommand{\nhii} {n_{\hiisub}} \newcommand{\sm} {\rm{s}^{-1}} % s-1 \newcommand{\Tmu} {T_{\mu}} % T_m \newcommand{\xhi} {x_{\rm{H \scriptscriptstyle I}}} \newcommand{\xhii} {x_{\rm{H \scriptscriptstyle II}}} \newcommand{\xhei} {x_{\rm{He \scriptscriptstyle I}}}

\newcommand{\xheii} {x_{\rm{He \scriptscriptstyle II}}} \newcommand{\xheiii} {x_{\rm{He \scriptscriptstyle III}}} \newcommand{\Zsun} {Z_{\odot}} % Solar metallicity\]

Radiative cooling and heating

Contents

Further reading

More information on cooling and heating in RAMSES can be found in Rosdahl's PhD thesis, or in the RAMSES-RT paper. A classical paper on the topic is Katz, Weinberg, & Hernquist (1996).

In most astrophysical contexts, gas dissipates thermal energy through collisional processes, and this energy is carried away by radiation. Gas may also gain energy from radiation, for example through photo-ionisations. In RAMSES, these radiative cooling and heating terms are accounted for in the energy conservation equation with the net cooling rate \(\Lambda\):

\[\frac{\partial E}{\partial t} + \nabla \cdot \left((E + P) \boldsymbol{u} \right) = -\rho \boldsymbol{u} \cdot \nabla \phi + \Lambda(\rho, e).\]
Here, \(t\) is time, \(\boldsymbol{u}\) is the gas bulk velocity, \(\phi\) is the gravitational potential, \(E = \frac{1}{2}\rho u^2 + e\) is the gas total energy, \(e\) is the internal energy, and \(P = (\gamma -1)e\), with \(\gamma\) the ratio of specific heats.

The net cooling rate \(\Lambda\) is a sum over many microscopic processes, involving collisions between ions and electrons, photo-ionisations, etc. In principle, it is thus a function of the temperature and density of the gas, of its detailed chemical composition and ionisation state, and of the local radiation field. The different cooling models implemented in RAMSES mostly differ on how they approximate this function.

The following approximations are available in RAMSES:

1. **ISM cooling**: a simplified prescription for cold, dense interstellar medium conditions.
2. **Equilibrium cooling with a UVB** (default): assumes photo-ionisation equilibrium with a redshift-dependent UV background.
3. **Equilibrium cooling with Grackle**: same physical approximation using the external Grackle library.
4. **Non-equilibrium cooling without RHD**: tracks ion fractions explicitly with a homogeneous UVB.
5. **Non-equilibrium cooling with RHD**: full radiative transfer coupled to chemistry of H and He.

Below, we will detail the implementation of the default model (equilibrium cooling) and of the non-equilibrium cooling (with a UVB) as illustrative examples of all cooling models implementations.

Step by step *default cooling model*

initialisations

Some models require initialisations, and these are done in `init_time` (in `amr/init_time.f90`) which is called at the beginning of a simulation (see code snippet below). For the default cooling model, this involves a call to `set_model` in `hydro/cooling_module.f90`. The `set_model` routine does two things: a) it initialises UV background parameters, and b), for cosmological simulations, it computes the (homogeneous) temperature of the Universe at the starting redshift of the simulation (typically read from the cosmological initial conditions, but possible to override with the aexp_ini namelist parameter).

```
subroutine adaptive_loop
   ...
   call init_time
   ...
   if(cooling) call set_table(dble(aexp))
   ...
   do ! Main time loop
      ...
      call amr_step(levelmin,1)
      ...
```

```
    end do
    ...
end subroutine adaptive_loop
```

A second initialisation here is a call to `set_table()` (from `hydro/cooling_module.f90`) before entering the main time-evolution loop of RAMSES (see above). This is a first computation of the cooling and heating rates tables.

Indeed, the default cooling module assumes photoionization equilibrium (PIE) with a redshift-dependent UVB. In such conditions, the abundances of H and He ions are direct functions of temperature, density, and redshift only. In turn, the cooling and heating rates are also reduced to being functions only of temperature, density, redshift and metallicity, so that the cooling function becomes \(\Lambda \simeq \Lambda(T,n_H, Z, z)\). The default cooling method simplifies things further by assuming a linear dependence with metallicity so that the cooling function becomes \(\CH = (\Heat + \Cool + Z/\Zsun \, \CoolZ)\, \nh^2\), where

- \(\Heat(\Tmu,\nh)\) is the heating rate from the UV background at the current redshift.
- \(\Cool(\Tmu,\nh)\) is the cooling rate due to H and He (and electrons) assuming PIE with the UVB at the current redshift.
- \(\CoolZ(\Tmu,\nh)\) is the cooling rate due to metals, at solar metallicity, and assuming PIE with the UVB at the current redshift.

With only two dimensions to the problem, it is much more efficient to pre-compute these rates and use linear interpolations than to calculate them on-the-fly (exponents, powers). The call to `set_table` computes and saves into tables these rates. As we'll see below, this is done at each coarse timestep to track the evolution of the UVB.

Filling in the rate tables with `set_table`

`set_table` first computes the PIE ionisation state for H and He, with a simple iterative process that involves equating the rates of photo-ionization, collisional ionization and recombination (this is done in the `cmp_chem_eq` routine in `hydro/cooling_module.f90`). Then the cooling and heating rates are computed and stored in tables for a range of \((\Tmu,\nh)\)-bins, where \(T_\mu=T/\mu\) and \(\mu\) is the mean particle mass in units of \(m_H\), \(\nh=X\rho/m_H\) is hydrogen number density, and \(X\) is the hydrogen mass fraction in the gas (a global constant, typically set to the value of \(0.76\)).

The abundances table:

The abundances \(n_i\), of each of the 6 primordial species (*e*, HI, HII, HeI, HeII, HeIII) are computed and stored for a range of (\(\Tmu, \nh\)) values. Here, rates are used for all the possible interactions involving these species - these are functions of photoionization rates \(\Gamma_i\), \(T\) and \(n_i\), and amount to a closed set of equations that are converged iteratively to an equilibrium solution, such that the creation rate equals the destruction rate for

each species. The species abundances also give the value of μ per ($\Tmu, \nh$))-bin, which can be retrieved by

$$\mu = \left[\, X(1+\xhii) + Y/4 (1+\xheii+2\xheiii) \, \right]^{-1},$$
where $Y=1-X$ is the helium mass fraction, $\xhii \equiv \nhii/\nh$, $\xheii \equiv \nheii/\nhe$ and $\xheiii \equiv \nheiii/\nhe$. The tabulation of the abundances also provides a direct mapping between $\Tmu$ and T which is useful in generating the rest of the tables.

Note that these tables are written to each ramses output directory, allowing the user to easily extract the equilibrium ionisation fractions and μ in postprocessing (by performing the same kind of interpolation for every cell as is done in RAMSES).

The cooling rates table:

Given the abundances, it is then a straightforward matter to calculate and tabulate $\Cool(\Tmu, \nh)$ – the cooling rate is a sum of bremsstrahlung, collisional excitation, collisional ionization, recombination, dielectric recombination and Compton cooling rates, all fitted analytic functions of temperature and abundances, and fetched from various sources in the literature (e.g. Cen, 1992).

The heating rates table:

It is also straightforward to tabulate $\Heat(\Tmu,\nh)$. Each bin contains

$$\Heat=\sum_i \Heat_i n_i,$$
where the sum is over the primordial ion species, and $\Heat_i$ are photoheating rates for the individual species ($\nhi$), $\nhei$), $\nheii$), and *e*).

The metal-cooling-contributions table:

RAMSES keeps a hard-coded table of a precomputed metal-cooling rate contribution, $\Cool_{Z}^{CIE}(T)$, which is the difference between zero metallicity and solar metallicity cooling rates calculated assuming collisional ionization equilibrium (CIE, i.e. chemical equilibrium in zero ionizing radiation), with the CLOUDY software package (Ferland et al., 1998). That is,

$$\Cool_{Z}^{CIE}(T) \; = \; \Cool_{Z}^{Cloudy}(T, \, Z=Z_{\odot}, \, UV=0) \; - \; \Cool_{Z}^{Cloudy}(T, \, Z=0, \, UV=0).$$
These rates are computed for a photoionization-free environment so they don't depend on gas density. Using this, the photoionization equilibrium (PIE) metal-cooling rates are approximated and tabulated as

$$\CoolZ(\Tmu,\nh) \; = \; \Cool_{Z}^{CIE}(T) \; \times \; f(T, \, \nh, \, z),$$
where f is a dimensionless analytic function that corrects for density $\nh$ and UV background photoionization at redshift z.

Implementation: For the default cooling model, the implementation of these computations is all in `hydro/cooling_module.f90`, with the following structure.

```
subroutine set_table(aexp)
   ... ! define boundaries of tables in terms of nH and T
   call cmp_table(nH_min,nH_max,T2_min,T2_max,nbin_n,nbin_T,aexp) !
compute the tables.
end subroutine set_table

subroutine cmp_table(nH_min,nH_max,T2_min,T2_max,nbin_n,nbin_T,aexp)
   ! Compute radiative ionization and heating rates
   call set_rates(t_rad_spec,h_rad_spec,aexp)
   ! Create the table
   do i_n = myid,nbin_n,ncpu
      call iterate(i_n,t_rad_spec,h_rad_spec,nbin_T,aexp) ! compute
cooling/heating for a range of temperatures
   end do
end subroutine cmp_table

subroutine iterate(i_n,t_rad_spec,h_rad_spec,nbin_T,aexp)
   nH = ... ! define value of nH from i_n
   do i_T = 1,nbin_T
      ! Compute cooling, heating and mean molecular weight

T2=10d0**table%T2(i_T)      ! define the temperature of current bin
      call cmp_cooling(T2,nH ...) ! compute equilibrium cooling
solution
      ...
      ! Compute cooling and heating derivatives
      T2_eps=10d0**(table%T2(i_T)+0.01d0)  ! define slightly larger
temperature ("slightly = 0.01")
      call cmp_cooling(T2_eps,nH, ...)
      table%cool_prime(i_n,i_T)=(log10(cool_tot_eps)-
log10(cool_tot))/0.01d0   ! evaluate the derivative and store in
_prime tables.
      ...
      ! Compute metal contribution for solar metallicity
      call cmp_metals(T2,nH,mu,metal_tot,metal_prime,aexp)
      ...
   end do

end subroutine iterate

subroutine
cmp_cooling(T2,nH,t_rad_spec,h_rad_spec,cool_tot,heat_tot,cool_com,heat_com,mu_o

   ! Iteration to find mu (rates depend on T, but we only know T/mu)
   do while (error is too large)
      T = ... ! get T from T/mu and a guessed value of mu
      call cmp_chem_eq(T,nH,t_rad_spec,n_spec,n_TOT,mu)   ! get
equilibrium solution at fixed T, including mu.
      ...
   end do
```

```
    ! Get equilibrium abundances
    n_E     = n_spec(1) ! electrons
    n_HI    = n_spec(2) ! H
    n_HII   = n_spec(3) ! H+
    ...
    ! Bremstrahlung
    cb1 = cool_bre(HI  ,T)*n_E*n_HII  /nH**2
    ...
    ! Ionization cooling
    ci1 = cool_ion(HI  ,T)*n_E*n_HI   /nH**2
    ... etc ...

end subroutine cmp_cooling
```

Operator-splitting and temperature update

For all cooling models, RAMSES uses **operator splitting** to solve the Euler equations in two steps. In a first step, gravity is computed, and the gas is advected, basically solving the Euler equations with $\Lambda = 0$. In a second **thermo-chemistry** step, the heating and cooling terms are computed, and the internal energy is updated with $\dot{e} = \Lambda$. This strategy is shown in the code snippet below taken from the main code loop `amr_step`. RAMSES first computes the hydrodynamics and updates `uold` with a call to `godunov_fine`. Then, RAMSES computes cooling on the updated state `uold` with a call to `cooling_fine` (more details below).

```
subroutine amr_step
   ...
   call godunov_fine ! do the hydro and update uold.
   ...
   call cooling_fine ! do the cooling and update uold.
   ...
end subroutine amr_step
```

The thermo-chemistry involves the interaction of radiation and matter, and the implementation differs if the code is used in radiation-hydrodynamics (RHD) mode or in hydro (HD) mode illustrated above. With RHD, the radiation is transfered accross the grid and the thermo-chemistry is computed on the fly.

In the general case, RAMSES implements a **semi-implicit scheme to evolve the temperature** due to cooling during a timestep. With ρ the mass density, γ the ratio of specific heats (usually given the value of $5/3$ in RAMSES, corresponding to monatomic gas), m_H the proton mass, $\kb$ the Boltzmann constant, and μm_H the average particle mass, one can write the internal energy as

$$e = \frac{T}{\mu} \times \frac{\rho \kb}{(\gamma-1) m_H} ,$$

It is clear here that after advection, when the density (and metallicity) is fixed, evolving the internal energy is equivalent to evolving the ratio \(\Tmu \equiv T/\mu\). This is what is done in RAMSES, by solving the equation

\[\frac{\partial \Tmu}{\partial t} = \frac{(\gamma-1) \mh}{\rho \kb} \ \CH.\]
Starting at time \(t\) with temperature \(\Tmu^{t}\), we compute the evolved temperature \(\Tmu^{t+\Delta t}\) with a semi-implicit Euler formulation (See Press et al., 1992):

\[\Tmu^{t+\dt}= \Tmu^{t} + \frac{\CH K \dt}{1-\CHp K \dt},\]
where \(K=\eTconv\) is the conversion factor between \(e\) and \(\Tmu\) and \(\CHp \equiv \frac{\partial \Lambda}{\partial \Tmu}\) can be estimated by finite-differencing the rate tables.

The thermochemistry is called in amr_step with a call to `cooling_fine` (in `hydro/cooling_fine.f90`), after the gravity source term and hydro advection. This evolves the temperature of all cells at the given level, over the current hydrodynamical timestep length, \(\dthyd\).

Details of the `cooling_fine` subroutine.

In `cooling_fine()`, cells at the given level are collected into vectors of size `NVECTOR` and then each vector of cells is processed in `coolfine1()`. This is a generic operation which is useful for vectorisation efficiency, and we illustrate it in the snippet below. Notice that at the end of `cooling_fine`, we again call `set_table` if we're at a coarse level (`ilevel==levelmin`) to prepare the next timestep.

```
subroutine cooling_fine(ilevel)

   ...

   ! Operator splitting step for cooling source term
   ! by vector sweeps
   ncache=active(ilevel)%ngrid
   do igrid=1,ncache,nvector
      ngrid=MIN(nvector,ncache-igrid+1)
      do i=1,ngrid
         ind_grid(i)=active(ilevel)%igrid(igrid+i-1)
      end do
      call coolfine1(ind_grid,ngrid,ilevel)
   end do


if((cooling.and..not.neq_chem.and..not.cooling_ism).and.ilevel==levelmin.and.cos
      call set_table(dble(aexp))
   endif

end subroutine cooling_fine
```

The `coolfine1` subroutine basically collects the density, temperature, and metallity for a batch of cells, in CGS units, and then calls `solve_cooling()` for those cells, which returns the (positive or negative) change in the temperature, in Kelvin, over \(\dthyd\). This is then converted back a change in internal energy densities, in code units, and then `uold` is updated accordingly for each of the cells.

`coolfine1` follows these steps in order:

1. **Unit conversion** — calls `units()` to obtain scale factors from code units to CGS.
2. **loop over cells from ind_grid list**

   2.1. **Select leaf cells** — only cells with no children (`son == 0`) are processed. An index array `ind_leaf` is built; the routine returns immediately if `nleaf == 0`.

   2.2. **Extract physical quantities** — reads `nH`, metallicity `Zsolar`, and total energy from `uold`, using one loop per quantity.

> **⚠ Warning**
>
> Do **not** combine multiple field extractions into a single loop:
>
> ```
> ! BAD: hurts vectorisation
> do i = 1, nleaf
>    nH(i)     = MAX(uold(ind_leaf(i), 1), smallr)
>    T2(i)     = uold(ind_leaf(i), neul)
>    Zsolar(i) = uold(ind_leaf(i), imetal) / nH(i) /
> 0.02d0
> end do
> ```
>
> Use a separate loop for each quantity so the compiler can vectorise the inner loop efficiently.

   2.3. **Compute thermal energy** — subtracts kinetic energy, radiation energy (if `NENER > 0`), and magnetic energy from the total to isolate the thermal part. Converts to \(\Tmu\) in Kelvin via `scale_T2`.

   2.4. **Self-shielding** — if `self_shielding` is enabled, the UV background factor is attenuated exponentially at high densities with a boost factor:

\[\mathrm{boost}(i) = \max\!\left(\exp\!\left(-\frac{\nh}{0.01\,\mathrm{cm}^{-3}} \right), 10^{-20} \right).\]
This is a simple “poor man’s” model: dense gas that would be self-shielded sees a suppressed UVB. Note that since cooling/heating tables are already computed as a

function of \(\Tmu\) and \(n_H\), the boost is in fact applied to density. This is a good approximation, as the dependence of cooling and heating on the radiation intensity is really through a term \(\Gamma/n_H\), so decreasing \(\Gamma\) or increasing \(n_H\) by the same factor has the same effect.

2.5. **Polytrope and temperature floor** — A minimum thermal energy is in general included. This may be because `barotropic_eos` is true, or because `jeans_ncells>0`, or because `T2_star >0`. This energy is expressed as a temperature floor (`T2min`), and needs to be computed and subtracted from the thermal energy before computing cooling. It is then inserted back in step 2.8. below.

2.6. **Cooling integration** — calls `solve_cooling`, which returns `delta_T2`, the change in \(\Tmu\) over the hydro timestep.

2.7. **Delayed cooling** — if `delayed_cooling` is active, cooling is suppressed in blast-wave regions. When the delay tracer scalar (`idelay`) exceeds a threshold, `delta_T2` is clamped to zero (no cooling applied).

2.8. Energy update — adds the polytrope floor energy back, as well as other energy terms (kinetic etc.) and

writes the updated total energy to `uold`:

```
uold(...) = T2(i) + T2min(i) + ekk(i) + err(i) +
emag(i)
```

The call to `solve_cooling` (in `hydro/cooling_module.f90`) is really the heart of the thermochemistry. Here the temperature-change of every cell in the vector is evolved, sub-cycling if needed with timestep lengths \(\dtcool\propto\frac{T}{\Lambda}\). Within each sub-step:

1. Obtain \(\CH\) and \(\CHp\) by bilinear interpolation in the pre-computed 2D tables at the current \((\Tmu, \nh)\).
2. Compute the sub-step size \(\delta t\) from the local cooling time \(\dtcool(\Tmu)\).
3. Apply the semi-implicit update:

   \[\Tmu^{t+\delta t} = \Tmu^{t} + \frac{\CH K \delta t}{1 - \CHp K \delta t}.\]
4. Advance the cell clock: \(t_{\mathrm{cell}} \leftarrow t_{\mathrm{cell}} + \delta t\). When \(t_{\mathrm{cell}} \geq \dthyd\), the cell is finished.

At the end, return \(\Delta\Tmu = \Tmu^{\mathrm{final}} - \Tmu^{\mathrm{init}}\).

Non-equilibrium cooling

This works quite similarly as the default equilibrium cooling, except, here, the non-equilibrium fraction of ionised hydrogen, and, optionally, HeII, HeIII, and neutral hydrogen (implicitly evolving molecular hydrogen) is evolved and tracked in every cell. These ionisation fractions are stored in passive scalars, usually right after the metallicity scalar. Here, the ionization fractions are evolved along with the temperature in each cooling sub-cycling step in a quasi-implicit fashion (see the RAMSES-RT paper by Rosdahl et al. 2013).

The non-equilibrium cooling module was written specifically for radiative transfer (see again the RAMSES-RT paper) and is found in `rt/rt_cooling_module.f90`. It contains the routine `rt_solve_cooling`, called instead of the default `solve_cooling` from `cooling_fine` if the non-equilibrium cooling is activated. In this case, cooling fine updates not only the pressure variable in `uold`, but also the passive scalars corresponding to the ionization fractions of hydrogen and helium (and, if radiative transfer is also activated, the momentum of the gas from radiation-gas interactions and the photon fluxes and densities). The non-equilibrium cooling can only be activated if RAMSES is compiled with the `-RT` flag. However, if the flag is used, non-equilibrium cooling can still be activated without radiative transfer, simply by compiling with zero radiation groups, and using `cooling=.true.` and `neq_chem=.true.` in the `COOLING_PARAMS` namelist.

As for the equilibrium cooling module, cooling and heating rates are tabulated, though in this case only against temperature (in Kelvin), whereas the equilibrium cooling rates can be tabulated against both temperature and gas density. The reason for this is that the cooling rates depend on the ionisation fractions of the gas species, which are direct functions of temperature in equilibrium, but not in non-equilibrium (and tabulating against density is preferrable if possible, since it reduces the computational cost compared to multiplying the cooling rates with densities).

The non-equilibrium heating and cooling rates tables are initialised in `rt_set_table` (called during `init_time`) and updated every coarse timestep from `amr_step`. For the cooling rates, only Compton cooling actually needs to be updated, since the others are not redshift-dependent (with equilibrium cooling all the cooling rates are redshift-dependent trough the PIE ionization fractions).

With non-equilibrium cooling, the homogeneous UV background is more flexible than the hardcoded variants in equilibrium cooling. It can be read from files.

ISM-cooling

Exercise: figure out how it works.

Subgrid modelling utilities

Contents

Introduction

Many processes happen on unresolved scales in any simulation: star formation, supernova explosions, accretion onto black-holes are typical examples of that in cosmological or galaxy-scale simulations. The effects of these unresolved processes on the surrounding, resolved, medium need to be modeled explicitly, as they do not result from the set of equations (Euler+...) solved by RAMSES. This is where **subgrid models** come in.

From a practical viewpoint, many subgrid models deal with the formation or growth of particles, and with the injection of matter, momentum, energy, etc. from these particles into the surrounding

medium. This means **updating particle properties based on grid properties**, and conversely **updating grid quantities based on particle properties**.

Here, we discuss basic algorithms that are commonly used to perform these operations in RAMSES. We will base the explanation on the examples of *star formation* and *SN feedback* subgrid models as a way to illustrate the different operations we need to perform on the grid and on particles.

A **star formation subgrid model** operates on leaf cells in typically in 3 steps:

1. Decide whether star formation may occur in the current cell. This depends on a set of conditions (defined by the modeller). These conditions may be local (e.g., a density threshold), or non-local, i.e., depend on the surrounding gas properties (e.g., a converging flow). In the latter case we need to collect information beyond the current leaf cell.

2. Decide how much gas mass will turn into stars. This depends on the star formation model and may again be a function of local properties (e.g., with a SF efficiency set by the leaf cell's free-fall time) or non-local properties (e.g., with the multi-free-fall models).

3. Remove gas from the mesh and spawn new star particles.

In turn, **SN feedback subgrid models** need to:

1. Decide whether a stellar particle releases matter and/or momentum and energy. This may happen as a one-shot event (typically 5-10 Myr after the stellar population formed), or may be modelled as a rate of events.

2. Figure out in which cell(s) the star particle will inject stuff. This may be a single cell (for metal release, thermal energy) or a number of cells (e.g., for mechanical feedback).

3. Update the star particle and the cell(s) according to the feedback model. Again this can be done in several ways (including through the use of *debris* particles...).

Exercise

These models apply to star formation and supernovae. Can you think of other examples where similar operations can apply?

Solution

- Injection of radiation from stars: this depends on the age/metallicity/mass of the star particle, and adds photons to the cells where star particles are located. Have a look at `star_RT_feedback` in `rt/rt_spectra.f90`.

- Most of AGN physics:
  - BH growth through gas accretion requires reading properties in a region around the BH particles, removing some of the gas in that region, and adding it to the BH
  - AGN feedback requires depositing mass, momentum, and energy in cells surrounding the BH particles
- Other “sink” modules (e.g., sink particles to model stars in high-resolution simulations)

Generic operations

For the models described above, can you define a few **generic operations** that we need to perform:

- Find the host cell of a particle
- Find the 26 neighbours of a cell (\(3^3\) cells minus the central cell)
- Identify all particles (e.g., stars) within a cell.
- Collect particles / cells within some distance of a point
- Remove stuff from cells/particles and add it to particles/cells (without breaking conservation laws)

We will now look at places in the code where these operations are done.

Finding the host cell of a particle

A typical example of this operation is for supernova feedback: we need to find the cell in which energy can be dumped.

Let’s look at the `thermal_feedback` routine in `feedback.f90`. As we have seen in the lecture on particles, the way to loop over particles *already* requires iterating over grids. This is the loop that looks like this:

```
do icpu=1,ncpu
! Loop over cpus
   igrid=headl(icpu,ilevel)
   ! Loop over grids
   do jgrid=1,numbl(icpu,ilevel)
      npart1=numbp(igrid)  ! Number of particles in the grid
      if(npart1>0)then
```

```
          ipart=headp(igrid)
          ! Loop over particles
          do jpart=1,npart1
             ! Save next particle   <--- Very important !!!
             next_part=nextp(ipart)

             !----
             ! Do something with particle ipart
             !----

             ipart=next_part  ! Go to next particle
          end do
       endif
       igrid=next(igrid)   ! Go to next grid
    end do
end do
```

In the `thermal_feedback` routine, this is performed in two steps: first, we count how many particles are stars:

```
if ( is_star(typep(ipart)) ) then
    npart2=npart2+1
endif
```

Then, in a second step, we do something with the star particles by repeating the loop in a *vectorized* way (see the vector sweep for the AMR structure, for example). This calls an *inner* function, `feedbk`, where the actual feedback is done.

It is only in this `feedbk` routine that the actual **cell** is identified, using a simpler version of the CIC scheme we have seen previously. Indeed, for this, RAMSES uses the NGP scheme (for Nearest Grid Point).

Exercise

Go through the `feedbk` routine and explain how NGP differs from CIC.

Finding the neighbours of a cell

In the `star_formation` routine of `pm/star_formation.f90`, some star formation models require to identify the neighbouring cells of star-forming sites in order to compute the velocity dispersion around a given cell.

For this, we use the `getnbor` routine defined in `pm/star_formation.f90`, and that heavily relies on routines developed in `amr/nbors_utils.f90`. The routine returns the (global) index, called `ind_nbor` of all `2*ndim` cells around a cell indexed by `ind_cell`.

This index can then be used to access the properties of the neighbours. For example, the density in the six cells around the target one are defined through

```
d1 = uold(ind_nbor(1,1),1)
d2 = uold(ind_nbor(1,2),1)
d3 = uold(ind_nbor(1,3),1)
d4 = uold(ind_nbor(1,4),1)
d5 = uold(ind_nbor(1,5),1)
d6 = uold(ind_nbor(1,6),1)
```

Note that this `ind_nbor` hides sometimes complex situations. For example:

- the target cell and its neighbours can be at different levels
- in some cases, a neighbour can be non-existent

The `getnbor` routine itself follows the logic described in the section on the AMR structure.

Exercise

Go through the `pm/move_tracer.f90` file to identify how tracer particles are identified and moved to neighbouring cells.

Identifying particles within a cell

As we have seen several times now, the particle linked list is defined using the *grid* structure, through the `headp` and `nextp` arrays: `headp(igrid)` points to the index of the first particle in a given grid `igrid`, and `nextp(ipart)` returns the next particle in the linked list.

To identify the particles within a given *cell*, the easiest way is then to simply identify the grid corresponding to a cell, and loop over the particles in that grid. Functions like `is_tracer(typep(ipart))`, `is_star(typep(ipart))`, etc. can then be used to select specifically all the particles of a given type.

Collect particles and cells in a sphere

The simplest way to identify particles or cells in a given region is to use the particle/grid iteration mechanism, and add a criterion based on the position.

For example, if we want to select all the particles with a radius `rmax` of a position `xtarget`, we can just add

```
dist2 = (xp(ipart, 1)-xtarget(1))**2 + (xp(ipart, 2)-xtarget(2))**2
+ (xp(ipart, 3)-xtarget(3))**2

if (dist2 .le. rmax**2) then
    ...
endif
```

in the particle loop.

Let's now look at the *kinetic feedback* implementation for a practical example of selecting *cells* in a given region. The idea of the model is to add momentum to cells within a given radius `rmax` around supernovae sites. This is implemented in `pm/feedback.f90` using three routines: - `kinetic_feedback`, the main driver, which loops over "debris particles" (you can think of these as "old stars with wind particles attached to them") and fills `xSN`, `vSN`, `mSN`, and other "supernova" arrays, - `average_SN`, which accounts for the grid discretization effects in the way momentum is scaled in the SN region, - `Sedov_blast`, which actually performs the feedback, and injects momentum and energy to the grid.

In `Sedov_blast`, the main loop is the following:

```
do i=1,ngrid
    if(ok(i))then
        ! Get gas cell position
        x=(xg(ind_grid(i),1)+xc(ind,1)-skip_loc(1))*scale
        y=(xg(ind_grid(i),2)+xc(ind,2)-skip_loc(2))*scale
        z=(xg(ind_grid(i),3)+xc(ind,3)-skip_loc(3))*scale
        do iSN=1,nSN
           ! Check if the cell lies within the SN radius
           dxx=x-xSN(iSN,1)
           dyy=y-xSN(iSN,2)
           dzz=z-xSN(iSN,3)
           dr_SN=dxx**2+dyy**2+dzz**2
           if(dr_SN.lt.rmax2)then
              ! Do things with uold(ind_cell(i), :) and SN variables
           endif
        end do
    endif
end do
```

Once again, the main idea is to select cells based on their position.

Updating cells and particles

The final, and perhaps most important, operation that is required for subgrid models is the ability to update the properties of cells and particles.

For grid quantities (e.g., gas density or momentum), we just need to update `uold` or `unew` with the right values.

> ⚠ **Warning**
>
> **Careful:** The choice of `uold` vs `unew` depends on where the subgrid model is called in the main `amr_step` loop.
>
> For example, `thermal_feedback` affects `unew` while `kinetic_feedback` affects `uold`. This is because `thermal_feedback` is called *after* the `set_unew` call, while `kinetic_feedback` is called before.

Either way, the logic is similar: once the indices of the cells affected by feedback are identified (in the `indp` array for thermal feedback and in the `indSN` for kinetic feedback), we can directly change their value. For example, at the end of the `Sedov_blast` routine, we have the following loop:

```
do iSN=1,nSN
   if(vol_gas(iSN)==0d0)then
      u=vSN(iSN,1)
      v=vSN(iSN,2)
      w=vSN(iSN,3)
      if(indSN(iSN)>0)then
         uold(indSN(iSN),1)=uold(indSN(iSN),1)+d_gas(iSN)
         uold(indSN(iSN),2)=uold(indSN(iSN),2)+d_gas(iSN)*u
         uold(indSN(iSN),3)=uold(indSN(iSN),3)+d_gas(iSN)*v
         uold(indSN(iSN),4)=uold(indSN(iSN),4)+d_gas(iSN)*w
         uold(indSN(iSN),5)=uold(indSN(iSN),5)
+d_gas(iSN)*0.5d0*(u*u+v*v+w*w)+p_gas(iSN)
         if(metal)uold(indSN(iSN),imetal)=uold(indSN(iSN),imetal)
+d_metal(iSN)
      endif
   endif
end do
```

> ⚠ **Warning**

These updates usually must be done for the **conservative** variables: this is because they describe extensive quantities, which can be added up (e.g., momentum, energy, etc). When developing a module, make sure that you are using the right variable set.

Implementation pitfalls

When writing a new subgrid model (or any extra module that deals with cells and particles), the main task is to connect all the generic operations discussed above in a single, consistent, piece of code.

Beyond just accessing properties, updating cells, and looping over grids, this requires extra care to deal with the fact that the physical model lives inside the rest of the code.

Units

One of the *most* bug-inducing part of writing subgrid models is dealing with units. In RAMSES, the units can be accessed through the `units` subroutine in `amr/units.f90`: this short routine defines a series of `scale_X` conversion factors, and most physical routines in RAMSES start by calling it

```
! Conversion factor from user units to cgs units
call units(scale_l,scale_t,scale_d,scale_v,scale_nH,scale_T2)
```

This defines for example the length conversion factor, `scale_l`, which is fixed for cosmological runs, and a namelist parameter otherwise.

While this presents no fundamental difficulty, special care must be taken to actually *use* these conversion factors.

For example, the `sf_trelax` namelist parameter is converted to **code units** using

```
trel=sf_trelax*Myr2sec/scale_t ! relaxation timescale
```

Here, `sf_trelax` is set in the namelist in Myr, `Myr2sec` converts it to seconds, and the `scale_t` division converts everything to code units.

Empty cells

RAMSES reacts badly when cells are completely devoid of gas. Indeed, when converting from conservative to primitive variable, we operate a division by the density: schematically speaking, `velocity = momentum / density`.

To avoid this issue, RAMSES introduces two main strategies:

- using a floor values when dividing by the density (usually called `smallr` in the code),
- making sure that subgrid models that remove gas from cells **never** remove all of the gas.

The `star_formation` routine, for example, uses this strategy:

```
! Security to prevent more than 90% of gas depletion
if (mgas > 0.9d0*mcell) then
    nstar_corrected=int(0.9d0*mcell/mstar)
    mstar_lost=mstar_lost+(nstar(i)-nstar_corrected)*mstar
    nstar(i)=nstar_corrected
endif
```

Supplementary exercises

We are now equipped to deal with *most* subgrid model requirements. *You* can then try apply this in a series of problems. We will explore this in more practical details in the problem session later in the week.

Star formation

Complete the following metacode to describe the implementation of a star formation model that uses only local criteria, where stars are allowed to form in cells with a density exceeding `density_star` and temperature below `temperature_star`, which are expected to be in CGS.

```
subroutine star_formation()

    ! TO BE COMPLETED ...

    ! getting physical info ... (from uold or unew?, etc.)
    ! units, time, cell sizes, position, density etc
    ! decide how many stars you form
    target_mstar = dt * density / tff * epsilon
    ! Poisson sampling
    ! create particles and add to lists
    ! remove matter from cell
```

In a second step, modify the code to implement a star formation model that forms stars where the local density is above a threshold value and the flow of gas is converging.

Refinement schemes & implementation

Please make sure to read the lecture on the AMR structure before following this lecture on the refinement criterions

Contents

1. What is refinement about?

1.1 Introduction

Refinement is the process of adapting the grid to the physical state of the simulation, by increasing or decreasing the resolution according to user-refinement criteria. In RAMSES, which uses a cell-based adaptive mesh, this consists in creating or destroying grids (i.e. a set of 2^{ndim} cells) when needed. Contrary to other codes, RAMSES does not have a derefinement criteria: a refined grid is marked for derefinement as soon as it no longer satisfies the criterion for refinement. In addition to the user-defined criteria, RAMSES ensures that the refinement map follows some geometrical rules that simplify the implementation of the Godunov solver and help with the stability of the numerical scheme.

1.2 General description

Let first review the description of the refinement algorithm in Teyssier 2002.

Click to read

The first step consists of marking cells for refinement according to user-defined refinement criteria, within the constraint given by a strict refinement rule: any oct in the tree structure has to be surrounded by $3^{ndim} - 1$ neighboring parent cells. Thanks to this rule, a smooth transition in spatial resolution between levels is enforced, even in the diagonal directions. Practically, this step consists in three passes through each level, starting from the finer level `levelmax` down to the coarse grid $l = 0$). | 1. If a split cell contains a children cell that is marked or already refined, then mark it for refinement; 2. Mark the $3^{ndim} - 1$ neighboring cells 3. If any cell satisfies the user-defined refinement criteria, then mark it for refinement.

One key ingredient still missing in this procedure is the so-called “mesh smoothing”. Usually, refinement are activated when gradients (or second derivatives) in the flow variables exceed a given threshold. The resulting refinement map tends to be “noisy”, especially in smooth part of the flow where gradients fluctuates around the threshold. [...] [To avoid this, in Ramses], a cubic buffer is expanded *several times* around marked cells. The number of times one applies the smoothing operator on the refinement map is obviously a free parameter. This parameter is noted `nexpand`. [...] Note that the exact method implemented here [...] leads to a convex structure for the resulting mesh, that is likely to increase the overall stability of the algorithm. Note also that only refinement criteria are necessary in RAMSES: no de-refinement criteria need to be specified by the user. Since the refinement map has been carefully built during [this] step, the refinement rule should be satisfied by construction. This is however not the case if one uses the adaptive time step method [...] . In this case, a final check is performed before splitting leaf cells. If the refinement rule is about to be violated, leaf cells are not refined.

The next step consists in splitting or destroying children cells according to the refinement map. RAMSES performs two passes through each level, starting from the coarse grid up to the finer grid 1. If a leaf cell is marked for refinement, then create its child oct; 2. If a split cell is not marked for refinement, then destroy its child oct.

Creating or destroying a child oct is a time-consuming step, since it implies reorganizing the tree structure. Thanks to the double linked list associated to the fully-threaded tree structure, this is done very efficiently by first disconnecting the child oct from the list, and then reconnecting the list in between the previous and next octs. Note however that this operation can not be vectorized. It is important to stress that this operation is applied at each time step, but for a very small number of octs. In other word, at each time step, the mesh structure is not rebuilt from scratch, but it is slightly modified, in order to follow the evolution of the flow.

In short, the refinement proceeds in two phases:

1. First go through all levels, starting from the finest, and marking (or flagging) cells that need to be refined according to the user-defined criteria. Doing so, the algorithm makes sures that the following rules are respected:
    - **The 2:1 rule**: If your neighbor is at level \(l\), your level should be \(l\) or \(l\pm1\). This ensures that at any boundary of a cell, you have either one neighbor (of level \(l\) or \(l - 1\)) or two (of level \(l + 1\)).
    - **The smoothing rule**: Every “physically” refined cell should be surrounded by at least `nexpand` refined cells in every directions.

- **“Not too fast” derefinement rule**: At each step, in any position, the refinement can only be reduced by one level

2. Split or merge the cells, according to the map built in first phase.

2. Flagging cells for refinement

Building the refinement map is done by the routine `flag` in `amr/flag_utils.f90`. It consists in filling the array `flag1`, with information on which cells should be refined. The actual refinement (if needed), is done in the next phase by the routine `refine` (see Section TBD).

The array `flag1` is defined in `amr/amr_commons.f90` by

```
integer ,allocatable,dimension(:)  ::flag1   ! flag for refine
```

and allocated in `amr/init_amr.f90` by

```
! Allocate AMR cell-based arrays
allocate(flag1(0:ncell)) ! Note: starting from 0
```

where `ncell` is the maximal number of cells for one MPI process. It is a cell-based array, meaning that `flag1(icell)` contains the flag information for the cell `icell` of the current process. If `flag1(icell) == 1` it means that the cells should be refined at during this time step, while `flag1(icell) == 0` means it should not be in a refined state (so derefined if it is currently refined).

In Ramses jargon, to *flag* or to *flag1* a cell `icell` means setting `flag1(icell)` to 1.

⚠ Warning

The array `flag1` is also used in other part of the code for various purposes (including load balance). The refinement information is thus only stored and valid in the routine `refine`, which should be called just after `flag`.

The routine `flag` calls the routine `flag_fine` over each level, starting from the finest one, and then the routine `flag_coarse` on the coarse grid (i.e. \(l = 0\)). For each level, `flag_fine` proceeds in 5 steps:

Show code

```
subroutine flag_fine(ilevel,icount)
use amr_commons
implicit none
integer::ilevel,icount
!----------------------------------------------------------
! This routine builds the refinement map at level ilevel.
!----------------------------------------------------------
integer::iexpand

if(ilevel==nlevelmax)return
if(numbtot(1,ilevel)==0)return

! Step 1: initialize refinement map to minimal refinement rules
call init_flag(ilevel)

! If ilevel < levelmin, exit routine
if(ilevel<levelmin)return
if(balance)return

! Step 2: make one cubic buffer around flagged cells,
! in order to enforce numerical rule.
call smooth_fine(ilevel)

! Step 3: if cell satisfies user-defined physical citeria,
! then flag cell for refinement.
call userflag_fine(ilevel)

! Step 4: make nexpand cubic buffers around flagged cells.
do iexpand=1,nexpand(ilevel)
   call smooth_fine(ilevel)
end do

! In case of adaptive time step ONLY, check for refinement rules.
if(ilevel>levelmin)then
   if(icount<nsubcycle(ilevel-1))then
      call ensure_ref_rules(ilevel)
   end if
end if

end subroutine flag_fine
```

Step 1: Initialisation

The subroutine `init_flag(ilevel)` builds the initial state at each level. It first initializes the `flag1` array at zero, as this array has been altered in other part of the code for other purposes. Please pay attention on how we go through all the active cells of a given level since this is used quite a bit here.

```
! Initialize flag1 to 0
nflag=0
do ind=1,twotondim
   iskip=ncoarse+(ind-1)*ngridmax
   do i=1,active(ilevel)%ngrid
      flag1(active(ilevel)%igrid(i)+iskip)=0
   end do
end do
```

Then, cells are flagged depending on the state of their childeren (if they have any). This is done in the `test_flag` subroutine. If `son(icell)>0` the cell is currenlty refined. We then loop over the children cells to see if any of them * a) has been already flagged for refinement, i.e. has `flag1(i_child_cell)=1`, which is known because the flagging operation is done level by level, starting from the finer ones, * b) is currently refined.

If any of these conditions is true, the cell is flagged. Three example situations are show in the figure below.

By flagging the cells that contain refined cells, we keep them in their refined state, ensuring the **"Not too fast" derefinement rule** is respected (see the middle example).

Of course, in levels below `levelmin`, all cells are marked for refinement.

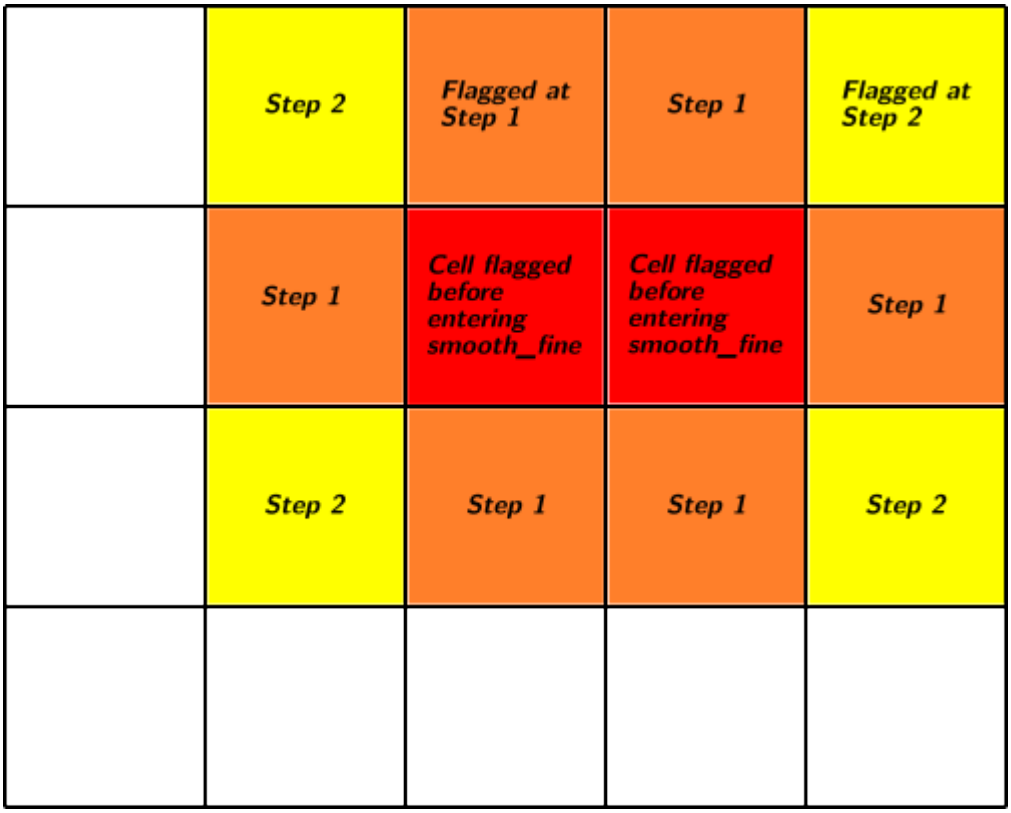
Step 2: Cubic buffer

The next step consist in marking the \(3^\mathrm{ndim} - 1\) neighboring cells of each one of the cells marked at step 1. This is performed by the `smooth_fine` subroutine. This ensures that **2:1 rule** is respected (as long as all the levels are flagged in this timestep, ie no adaptive timestepping is used - see Step 5).

The `smooth_fine` routine proceeds in `ndim` iterations: - iteration 1: flag1 cells with at least 1 flag1 neighbors (if `ndim` > 0) - iteration 2: flag1 cells with at least 2 flag1 neighbors (if `ndim` > 1) - iteration 3: flag1 cells with at least 2 flag1 neighbors (if `ndim` > 2)

In each step, it uses the `flag2` array as a temporary array, because we want to distinguish the cells that were flagged before entering the step from the ones flagged during the step. `flag2` is built the same way as `flag1`.

By doing so, it ensures that each previously flagged cells will be surrounded by flagged cells (see illustration below)



Step 3: Apply user-defined criterion

This is where you may want to change things

Cells are flagged according to a user specified criterion in the routine `userflag_fine`. The different criteria are detailled in the next section.

In cosmo runs, the whole level is prevented to refine in certain conditions.

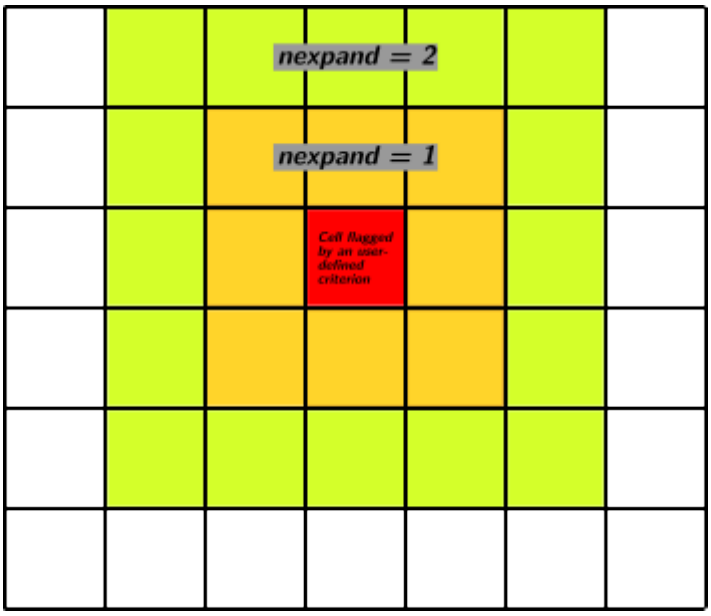
There are several subroutines for refinement criteria: - subroutine `flag_utils.f90/poisson_refine()`: local Lagrangian and variable refinement criteria, based on the mass (or another variable) in each cell, using the parameters `m_refine`, `ivar_refine`, `var_cut_refine`: - subroutine `godunov_utils.f90/hydro_refine()`: error on gradients (parameters `err_grad_d`, `err_grad_p`, `err_grad_u`, `rt_err_grad_xHII`, `rt_err_grad_xHI`, as well as MHD variables) - subroutine `godunov_utils.f90/jeans_length_refine()`: refinement over the jeans length (parameter `jeans_refine`) - subroutine `rt_hydro_refine()`: refinement over `rt_err_grad_cn`.

After each of these routine, a bunch of new cell are flagged as to be refinable using the temporary array `ok` array. There is then an additional filter implementing the geometrical refinement (subroutine `geometry_refine`), that will **unmark** all the cells that do not fulffill it by updating the `ok` array. By using a temporary array instead of `flag1`, there is no risk of breaking the refinement rules)

[1]. Finally all the cells marked with `ok` are marked with `flag1`.

Step 4: Mesh smoothing

Enforce the smoothing rule. This simply applies the `smooth_fine()` subroutine, that was used in step 2, `nexpand` times.



Step 5: Ensure the 2:1 refinement rule (if we are in an adaptive step)

Step 1-4 ensure that the refinement rules are always respected, as long as they are applied to all levels.

However, if adaptive timestepping is used, it may well be that the `flag_fine` routine applied at level `ilevel` but not at level `ilevel - 1` [2], and thus that the planned refinement may break the 2:1 rule. If we are within such a timestep, that is if `icount` (the current subcycling step at this level) is not the last one (`nsubcycle(ilevel-1)`), we call the `ensure_ref_rules` subroutine. This subroutine goes through each grid \(i\) at level `ilevel` and makes sure that all the \(2^\mathrm{ndim}\) neighboring grids exists. If this is not the case, the cells within the grid `i` cannot be refined without potentially breaking the 2:1 rule and are unflagged.

> **⚠ Warning**
>
> Note that because of this, the mesh smoothing rule with `nexpand` > 2 is actually not ensured.

3. Existing user defined criteria

In this section, we review how the existing user-defined criteria are implemented. They are controlled by the `REFINE_PARAMS` namelist block (see the doc here).

3.1. Mass (“Quasi Lagrangian strategy”)

This corresponds to the parameters `m_refine`, `mass_sph` and `mass_cut_refine` and is applied in the `poisson_refine` routine.

The idea is to compute the total mass in each cell (including the contrinution for particles), and comparing it to a reference mass `mass_sph`. If the computed mass exceed `m_refine(ilevel) * mass_sph`, the cell is refined. The mimics the typical refinement of SPH codes.

☢ Danger

In this part of the code, the arrays `cpu_map` and `phi` are used as temporary array to flag the cells that should be refined and the total “Lagrangian” mass in each cell. This allows to save memory and allocation time and works because the information that was stored in these arrays is no longer needed in the `flag` routine.

The routine works differently depending on whether gravity is turned on or not.

If `poisson == .true.`, prior entering the `flag_fine` routine, the array `cpu_map` is filled with a pre-refinement map, containing 1 for the cells that should be refined according to quasi lagragian strategy, and 0 for the others. This is done in the `rho_fine` routine in `pm/rho_fine.f90` where the total density field for the gravity is calculated (the mass-refine information comes almost for free there). The routine `rho_fine` firsts initialize the `phi` array (strange name again, because this is also a reused array) with the mass of gas in each cell divided by `mass_sph`. The contribution of the particles is then added using the cloud-in-cell scheme. The map is then computed (stored in `cpu_map`) by comparing the array `phi` with `m_refine(ilevel)`. In `poisson_refine`, we just use the content of the array `cpu_map` to set the array `ok`, which is then used to update the array `flag1`.

If `poisson == .false.`, the mass of the gas in the cell is simply compared to `m_refine(ilevel) * mass_sph`.

3.2 Variables and passive scalar strategy

The user can also refine **the initial grid** by inputing a threshold on the value of a variable. This is controlled by the `ivar_refine` and `var_cut_refine` parameters. Cells where the hydro variable with the index `ivar_refine` has a value larger than `var_cut_refine` should be refined.

⚠ Warning

This refinement strategy is only working when gravity is on (`poisson == .true.`)

This is also carried on within the `poisson_refine` and the `rho_fine` routines. It acts as a filter on the quasi-lagrangian strateg by preventing the refinement on the mass whenever `uold(ivar_refine)/rho` < `var_cut_refine`.

3.3. Gradients (hydro and rt) strategy

This refines on the gradients of user defined variables. The logic is simpler (compute the gradients and then apply the criterion), but the routines are spread among several files.

Exercise

Follow the different calls of `hydro_flag` and then `rt_hydro_flag` to see when the array `ok` is updated. Did you notice something strange with `rt_hydro_flag` ?

3.4. Jeans length strategy

This strategy is used a lot in simulations that care about the Truelove criterion, which says that to avoid artifical fragmentation of the gas, its Jeans length should be resolved by at least 4 cells.

It is also applied in the `hydro_flag` routine, which calls `jeans_length_refine` which is computing the Jeans length and applying the criterion.

3.5. Geometrical strategy

This is actually a filter on the other kinds of refinement, which is ran after all the previous criteria filled the `ok` array. Only the cells that have `ok(icell) == 1` and satisfy the geometrical criterion are actually flagged to be refined (in `flag1`). Namely, the user has to define a zone at each level in which the refinement can happen.

Exercise 1: Refine on a variable or a combination of variables

Level 1

Modify the “Variables and passive scalar” strategy to make it more useful: 1. Make the “Variables and passive scalar” strategy independent from the “Quasi Lagrangian strategy” (you can code it as a new strategy, with its own parameters). 2. Make it possible to have a different threshold at each level (ie make `var_cut_refine` an array)

Level 2

If you have time, you can also decide to refine on a combination of variables. You will read and parse a formula from the namelist and refine on the result.

Exercise 2: Improve the geometrical criterion

Modify `geometry_refine` to add more than one region of refinement, allowing for more complex shapes.

Clue 1

You can take inspiration from the initial condition "regions" parameters, which allows up to `MAXREGION` regions to be initialized with different parameters.

Clue 2

Helping TODO List:

- ☐ In `amr_parameters`, add a `MAX_REGION_REFINE` parameter
- ☐ Add a new parameter `refine_params/nregions_refine` (look at the appropriate lecture to see how to do that)
- ☐ Change how `r_refine`, `x_refine`, etc .. (up to `b_refine`) are defined. They could look like `real(dp),dimension(1:MAXLEVEL,1:MAXREGION)` Look at clue 3 to see how to deal with multidimensional arrays in the namelist. Ideally we want to make sure of not introducing any breaking changes.
- ☐ Now the hard work is in `geometry_refine`. You need to introduce the appropriate loops. Beware of the logic: `geometry_refine` is a filter: if we have two regions we want to refine cells that are in region 1 OR region 2 [3]. You may want to use the `flag2` array as a temporary array.
- ☐ Test your new implementation! First run the test suite to make sure you didn't break anything, and then add a test that uses your improved geometrical criterion.

Clue 3

Deal with multidimensional arrays in the namelist: there's some help in StackOverflow You can play with the following program (eg. with Online Fortran Compiler)

```
program test

   implicit none

   integer :: ierr, unit, i, ilevel, iregion
   integer,parameter :: MAX_REGION  = 4
   integer, parameter :: MAX_LEVEL = 3
   real(kind=kind(0.0d0)), allocatable :: p(:, :)

   namelist /VAR_p/ p


   allocate(p(MAX_LEVEL,MAX_REGION))
   open(newunit=unit, file='namelist.nml', status='old',
iostat=ierr)
   read(unit, nml=VAR_p, iostat=ierr)
   close(unit)

   write(*,*) "REGIONS"

   do  iregion = 1,MAX_REGION
      write(*,*) p(:,iregion)
   end do

   write(*,*) "LEVELS"

   do ilevel = 1,MAX_LEVEL
      write(*,*) p(ilevel,:)
   end do
end program test
```

```
&VAR_p
!p = 1.30, 0.8, 3.1 ! Only one region, backwards compatible syntax
p(:,1) = 1.30, 0.8, 3.1 ! Region 1
p(:,2) = 1.50, 3.2 ! Region 2
/
```

4. Modify the tree structure

In the previous step, we built a refinement map, stored in `flag1`, whether each cell should be refined or not. Now it's time to actually apply the change, which consist adding a son grid in the leaf cells that were marked for refinement (ie, splitting them) and were not refined before, or destroy the son grid in refined cells that are no longer flagged.

The is done is by the `refine` routine in `amr/refine.f90`. This routine applies the refinement map on all levels, but this time going from the coarsest level to the finest. The creation and destruction of grid was discussed in the lecture on the mesh structure.

An important question that arises when creating a cell is how to interpolate the value of the variables. For the hydro variable, this is done automatically by the code and controlled by the user with the `interpol_var` and `interpol_type` parameters.

> **⚠ Warning**
>
> If you add a new variable, you may have to tell the code how to interpolate it in the `refine_utils/make_new_grid` routine.

[1]
except after the `rt_hydro_refine`, which is a probable bug.

[2]
because all cells in level `ilevel - 1` won't be active and thus left untouched.

[3]
or do an user specified logical operation

Namelist parameters

In this chapter, we cover how users control simulations via namelist parameters.

Contents

1. What is a namelist

In RAMSES, namelist parameters form the primary user interface for configuring a simulation. They allow users to specify runtime settings without recompiling the code — such as solver options, output frequency, cosmological constants, or model flags. At runtime, the user provides an input file (typically *simulation_name.nml*) containing these parameters. An example namelist file and overview of all runtime parameters can be found in the User Documentation. For example, the namelist block setting the AMR-related parameters can look like this:

```
&AMR_PARAMS
levelmin=3
levelmax=10
ngridmax=2000
nexpand=1
boxlen=1.0
/
```

Namelists are Fortran constructs that group configuration variables. They are defined using the keyword `namelist`, followed by a chosen name and a list of the associated variables. For example the `amr_params` namelist in RAMSES is defined as follows:

```
namelist/amr_params/levelmin,levelmax,ngridmax,ngridtot &
& ,npartmax,nparttot,nexpand,boxlen,nlevel_collapse
```

In RAMSES, the variables associated to namelist parameters are usually declared in the one of the parameter or common modules (found in the *module_parameter.f90* or *module_commons.f90* files). They are globally accessible by importing the module, for example:

```
use amr_parameters
```

2. Reading and processing namelist parameters

Namelists are read once during program initialization. The keywords in the namelist file provided by the user are mapped to the variables names in the code. Their value is altered from the default value specified in the code to the value provided by the user. In RAMSES, this is handled by the `read_params` subroutine (found in *amr/read_params.f90*), which is called at the very beginning of the simulation. The code to process a namelist will go through the following steps:

```
! Definition of the namelist
namelist/something_params/param1,param2,param3
```

```
! Go to the beginning of the namelist file
rewind(namelist_unit)

! Read namelist
read(namelist_unit,NML=something_params,IOSTAT=nml_err)

! Checks in whether the namelist was present in the file
if(nml_err<0)then
   ! EOF reached before namelist was found
elseif(nml_err>0)then
   ! Problem with formatting in the file
endif

! Do some check on the read values
if(param1<0) nml_ok=.false.
...
```

Older parts of the code may use more unorthodox syntax, but effectively do the same thing. For example:

```
  ! Read namelist file
  rewind(1)
  read(1,NML=init_params,END=121)
  goto 122
121
write(*,*)' You need to set up namelist &INIT_PARAMS in parameter
file'
  call clean_stop
122 rewind(1)
```

The identifier number for the namelist file is 1.

Remark that each MPI process will read the namelist file, and so each process has access to all parameters.

3. Where are namelists defined in RAMSES

Each module or subsystem (e.g. AMR, hydrodynamics, gravity) is controlled by its own namelist definition and associated variables. The naming convention for namelists is `<module>_params`. The namelist definitions can be found in one of the following places:

- *amr/read_params.f90*, which contains:
    - `run_params`
    - `amr_params`

    - `output_params`
    - `movie_params`
    - `lightcone_params`
    - `tracer_params`
    - `poisson_params`

- *hydro/read_hydro_params.f90*, which contains:
    - `init_params`
    - `hydro_params`
    - `refine_params`
    - `boundary_params`
    - `feedback_params`
    - `cooling_params`
    - `sf_params`
    - `units_params`
    - `grackle_params`
    - `physics_params` (legacy)

- the corresponding code module:
    - `clumpfind_params` in *pm/clump_finder.f90*
    - `mergertree_params` in *pm/merger_tree.f90*
    - `stellar_params` in *pm/read_sink_feedback_params.f90*
    - `sink_params` in *pm/sink_particle.f90*
    - `unbinding_params` in *pm/unbinding.f90*
    - `rt_params` and `rt_groups` in *rt/rt_init.f90*
    - `turb_params` in *turb/read_turb_params.f90*

Remark that recently dedicated subroutines have been created in *amr/read_params.f90* to handle the namelist defined in this file.

4. Adding a new namelist parameter

When introducing a new algorithm or making small modification to the code, you may need to add a new parameter to an existing namelist. The procedure is straightforward:

- Step 1: Determine to which namelist the new variable belongs and add it to the namelist declaration.
- Step 2: Identify the module to which to add the new variable and declare it. A default value should be set, which will be used in case the parameter is not specified in the namelist file. This also ensures backward compatibility with existing namelist files. A comment should be added to describe what the variable represents.
- Step 3: Since the variable is added to an existing namelist, it will be automatically read when the corresponding namelist is parsed. Optionally, you can add checks to verify whether the user provided a sensible value.
- Step 4: Document the new parameter. Add an entry to the markdown file listing the parameters of the altered namelist, which can be found in the folder

doc/wiki.

Exercise

Add a new fictional parameter called `tuto_heating_model`. Imagine that setting this parameter enables an additional heating source for which three models exist in the literature. By default, we want this source to be turned off. Test your code by printing the value of `tuto_heating_model`.

Solution

Step 1 Since the new parameter deals adds a heating source, it should be added to the `cooling_params` namelist, which deals with cooling/heating and chemistry. This namelist is defined in *hydro/read_hydro_params.f90*. We add it at the end, since the order in which the parameters are listed is irrelevant:

```
! Cooling / basic chemistry parameters
namelist/cooling_params/
cooling,metal,isothermal,haardt_madau,J21 &

& ,barotropic_eos,barotropic_eos_form,polytrope_rho,polytrope_index,T_eos,mu_
&

& ,a_spec,self_shielding,z_ave,z_reion,ind_rsink,T2max,neq_chem
```

```
&
      & ,cooling_ism,tuto_heating_model
```

Step 2 Searching for the other variables which are associated with this namelist, we find that they are defined in *amr/amr_parameters.f90*, rather than in *hydro/hydro_parameters.f90*. Legacy codes are often plagued by this sort of inconsistencies. We declare the new variable as an integer, since it will take the values 0,1,2 or 3 and we set its default value to 0, which turns the feature off:

```
logical ::cooling_ism = .false.      ! Use cooling module from
Audit & Hennebelle 2005 (non-RT)
                                       ! instead of ramses
classical cooling
integer ::tuto_heating_model=0       ! Model for the tutorial
heating (1=Cleopatra+1996, 2=Thutmose+1998, 3=Seti+2004)
                                       ! 0 turns off tutorial
heating
```

Step 3 We add some a check to verify that the user chose an existing model:

```
!--------------------------------------------------
! Check tutorial heating model
!--------------------------------------------------
if(tuto_heating_model<0.or.tuto_heating_model>3)then
   if(myid==1)write(*,*)'Error: unknown tuto_heating_model.
Choose 1,2,3 or 0.'
   nml_ok=.false.
endif
if(myid==1)write(*,*)'TUTORIAL: tuto_heating_model is set
to',tuto_heating_model
```

Several other parameters in the cooling namelist have checks. We can place our block of code after the checks on `barotropic_eos` for example.

Step 4 We find the description of the `cooling_params` namelist in the file *doc/wiki/Physics.md*. We add a line to the Cooling parameters table:

```
| `tuto_heating_model`     | `integer` | `0`  | Model for the
tutorial heating. 1=Cleopatra+1996, 2=Thutmose+1998,
3=Seti+2004. 0 turns off tutorial heating.
```

Don't forget to test your code. This can be done, for example, by adding the new parameter to one of the namelists in the test suite.

5. Defining a new namelist block

When adding major new features, it is appropriate to define a new namelist to group the parameters that control the new part of the code. When developing bigger modules, we encourage to follow the example of the turbulence module. In short:

- Define module-specific variables in a *module/module_parameters.f90* file
- Write a routine `read_module_params` in which you define the namelist, read it and process its parameters. Place it in a file *read_module_params.f90* file included in the module's directory.
- Add the call to `read_module_params` to `read_params`.
- Update the documentation of your module.

Remember that when creating new directories and files, the Makefile needs to be updated.

Exercise

In *amr/read_params.f90*, add a subroutine that will read and process a new fictional namelist called `tuto_params`, which contains two parameters: `tuto_efficiency` and `tuto_timescale`. Both of these parameters have to be positive and cannot be zero. For simplicity, you can define the variables inside the new subroutine.

Solution

At the bottom of *amr/read_params.f90*, we add:

```
subroutine read_tuto_params(namelist_unit,nml_ok)
   use amr_parameters, only:dp
   use amr_commons, only:myid
   implicit none
   integer,intent(in)::namelist_unit
   logical,intent(inout)::nml_ok
   integer::nml_err
   real(dp) :: tuto_efficiency=1
   real(dp) :: tuto_timescale=1

   namelist/tuto_params/tuto_efficiency,tuto_timescale

   ! Go to the beginning of the file
   rewind(namelist_unit)

   ! Read namelist
   read(namelist_unit,NML=tuto_params,IOSTAT=nml_err)

   if(nml_err>0)then
```

```
   if(myid==1)write(*,*)'Error reading namelist &TUTO_PARAMS. Check
   formatting.'
         nml_ok=.false.
      endif

      if(tuto_efficiency<=0)then
         if(myid==1)write(*,*)'Error in the namelist:
   tuto_efficiency must be larger than 0'
         nml_ok=.false.
      endif

      if(tuto_timescale<=0)then
         if(myid==1)write(*,*)'Error in the namelist:
   tuto_timescale must be larger than 0'
         nml_ok=.false.
      endif

      if(myid==1)write(*,*)'TUTO:
   tuto_efficiency=',tuto_efficiency,',
   tuto_timescale=',tuto_timescale

   end subroutine read_tuto_params
```

In the main routine of *amr/read_params.f90*, we add the call:

```
call read_poisson_params(1,nml_ok)
call read_tuto_params(1,nml_ok)
```

Remark that because we defined the variables directly in the subroutine, we avoided having to update the Makefile to add new module files.

6. Bonus: namelist and Python

If you want to read/write namelist with python, check out the `f90nml` package.

Initial conditions

In this chapter, we cover how initial conditions are prepared and read. There are several ways to provide the starting state of a RAMSES simulation. For basic geometries, the user can use the parameters in the `init_params` namelist block. For more advanced setups that can be described

analytically, there are the `condinit` subroutines. Finally, RAMSES also supports input from files with specified formats.

Contents

1. Analytical initial conditions on the grid with `condinit`

RAMSES users may already have experience with implementing initial conditions using the `condinit` routine. This routine is called by `init_flow_fine` during the setup phase of the simulation. At the time of writing, there are two versions available in the public version: one for hydro and one for mhd. Remark that recently, the system has been reworked to support various default setups (rather than replying on the patch system).

As input, the `condinit` routine receives the cell center positions (in code units) of the `nn` cells in the vector sweep, as well as the cell size of the current level. From this information, the primitive variables are then calculated. Several prescriptions are available by default and can be selected by setting `condinit_kind` in the namelist. One can easily add their own analytical initial conditions as a new subroutine following the existing examples. Finally, the primitive variables are converted to the conservative ones. The conservative variables are then returned to `init_flow_fine` through the array `u`, where they are written to `uold`.

Exercise

Implement a new `condinit_type` that adds a sinusoidal perturbation on a uniform density background in 1D: \(\rho(x) = \rho_0 [1 + A \cos(\frac{2\pi x}{\lambda})]\) The pressure is set to the same value as the density.

Solution

```
...
  case('jeans_instability_cos')
    call jeans_instability_cos_condinit(x, q, dx, nn)
...
!
================================================================
subroutine jeans_instability_cos_condinit(x,q,dx,nn)
```

```
  use amr_parameters
  use hydro_parameters
  use constants, only:pi

  implicit none
  integer ::nn                                ! Number of cells
  real(dp)::dx                                ! Cell size
  real(dp),dimension(1:nvector,1:nvar)::q ! Primitive variables
  real(dp),dimension(1:nvector,1:ndim)::x ! Cell center
position.
  !
================================================================
  ! sinusoidal perturbation of amplitude A
  !
================================================================
  integer::i
  real(dp),parameter::A=1d-4, lambda=0.5

  ! Call built-in initial condition generator to init the fields
  call region_condinit(x,q,dx,nn)

  do i=1,nn
    ! density
    q(i,1) = 1+A*cos(2*pi*x(i,1)/lambda)
    ! pressure
    q(i,3)=q(i,1)
  end do

end subroutine jeans_instability_cos_condinit
```

2. Input file formats

Another way to provide the initial conditions is through files, by setting the namelist parameters `initfile` and `filetype`. For the variables on the grid, supported formats are ascii and grafic (see *init_flow_fine.f90*), while for particles ascii, grafic, and gadget are available (see *init_part.f90*, and the chapter on particles).

If you want to add your own input file, good luck.

Outputting

In this chapter we cover

- How outputs are written, structured, and extended.
- The link between I/O routines and the AMR (adaptive mesh refinement) hierarchy and MPI domain decomposition.

Contents

1. The different output files in a RAMSES snapshot

Exercise

Look in a ramses output. What files are there? (If you have no outputs laying around, you can run one of the test suite cases with `-d`, for example `./run_test_suite.sh -t 1 -p 2 -d` and look in the individual test folders, e.g. *tests/hydro/advect*)

Solution

Several types of files can be found in a RAMSES snapshot *output_xxxxx/* (non-exhaustive):

- General simulation info:
    - info_xxxxxx.txt: units, boxlen, output time, cosmological constants,...
    - info_rt_xxxxxx.txt: info on photon groups and chemistry
    - header_xxxxxx.txt: number of particles per family
- Data outputted by each MPI process (yyyyy):
    - amr_xxxxx.outyyyyyy: the grid linked list variables, grid center position, octree variables, loadbalancing and refinement map
    - hydro_xxxxx.outyyyyyy: all hydro variables
    - rt_xxxxx.outyyyyyy:
    - grav_xxxxx.outyyyyyy: gravitational potential and acceleration * part_xxxxx.outyyyyyy: particle fields
- File descriptors: these files list which variables are outputted in the corresponding data files and in which order they are:
    - hydro_file_descriptor.txt
    - rt_file_descriptor.txt
    - part_file_descriptor.txt
- Data files outputted by cpu 0 only:
    - sink_xxxxxx.csv
    - stellar_xxxxxx.csv
- Information about the execution:
    - namelist.txt: copy of the input namelist file
    - timer_xxxxxx.txt: execution time per module
- Information about the compilation:
    - compilation.txt: date, commit hash, etc
    - makefile.txt: copy of the Makefile used

- patches.txt: copy of the content of any patch files

2. The main output routine: `dump_all`

RAMSES outputs simulation data through a centralized routine, `dump_all` in *amr/output_amr.f90*, which orchestrates the writing of all output files. It is executed by each MPI process, and so each MPI process produces its own set of files in the snapshot.

Each module typically has its own `output_xxxx.f90` file / subroutine to deal with outputting, which is then called by `dump_all`.

3. Outputting the AMR structure

The outputting of the AMR structure is handled by the routine `backup_amr` which can be found in `amr/output_amr.f90`. It writes:

- the grid linked list variables,
- the grid center position, and the octree variables
- the loadbalancing map
- refinement map.

Normally, there is no need to alter something here.

4. Outputting the variables on the grid (hydro, rt, gravity)

Outputting the hydro variables, which are stored in `uold` (see the chapter on Hydrodynamics Section 2.2), is done by the routine `backup_hydro` found in the file *hydro/output_hydro.f90*. The structure of the routine is as follows:

Click to show the code

```
! Open the file and the file descriptor
...
! write some information to the header
write(unit_out) ncpu
...
! Loop over refinement levels
do ilevel = 1, nlevelmax
   ! Loop over physical boundaries and MPI domains
   do ibound = 1, nboundary+ncpu
```

```
      ! Get the head of grid linked list and number of elements in
it
      if (ibound <= ncpu) then
         ncache = numbl(ibound, ilevel)
         istart = headl(ibound, ilevel)
      else
         ncache = numbb(ibound-ncpu, ilevel)
         istart = headb(ibound-ncpu, ilevel)
      end if
      write(unit_out) ilevel
      write(unit_out) ncache
      if (ncache > 0) then
         ! allocate work arrays to gather grid indices and variable
values
         allocate(ind_grid(1:ncache), xdp(1:ncache))
         ! Gather the grid index ind_grid by following the linked
list
         igrid = istart
         do i = 1, ncache
            ind_grid(i) = igrid
            igrid = next(igrid)
         end do
         ! Loop over cells in the grids
         do ind = 1, twotondim
            iskip = ncoarse+(ind-1)*ngridmax
            ! Write the data of all fields in order
            ! starting with the Euler variables
            do ivar = 1, neul-1
               if (ivar == 1) then
                  ! Write density
                  do i = 1, ncache
                     xdp(i) = uold(ind_grid(i)+iskip, 1)
                  end do
                  field_name = 'density'
               else
                  ! Write velocity field
                  do i = 1, ncache
                     xdp(i) = uold(ind_grid(i)+iskip, ivar)/
max(uold(ind_grid(i)+iskip, 1), smallr)
                  end do
                  field_name = 'velocity_' // dim_keys(ivar - 1)
               end if
               call generic_dump(field_name, info_var_count, xdp,
unit_out, dump_info_flag, unit_info)
            end do
            ...
      end do
      deallocate(ind_grid, xdp)
   end if
end do
end do
close(unit_out)
```

An equivalent exists for outputting the content of `rt_old`, done by the subroutine `rt_backup_hydro` in *rt/rt_output_hydro.f90*. Similarly, the outputting of the gravity variables `phi`, `f` and optionally `rho` is handled by the routine `backup_poisson` in *poisson/output_poisson.f90*.

Generalized, the part of the inner loop that does the outputting of a variable looks like this:

```
! Gather the values of the variable ivar into the work array xdp
do i = 1, ncache
   xdp(i) = uold(ind_grid(i)+iskip, ivar)
end do
! Specify the name of the variable
field_name = 'my_variable_name'
! Pass data to dump utils for writing
call generic_dump(field_name, info_var_count, xdp, unit_out,&
                & dump_info_flag, unit_info)
```

The routine `generic_dump` (defined in the file *io/dump_utils.f90*) is performing the `write` instruction for any type. When adding new variables on the grid, the corresponding backup routine needs to be updated to include a new block following this structure. The name you choose for your new variable will appear in the file descriptor, so make sure to choose something informative.

5. Outputting particle fields

The routine for outputting particles, `backup_part`, can be found in *pm/output_part.f90*. The code is simpler than for grid variables, since we do not follow any linked list, but simply loop over all allocated particle memory and check whether a particle exists at each memory location. The structure of the code is as follows:

Click to show the code

```
! write some information to the header
write(unit_out) ncpu
write(unit_out) ndim
write(unit_out) npart
...
! allocate float work array
allocate(xdp(1:npart))
! Write float properties in order, starting with the position
do idim = 1, ndim
   ! gather data in work array
   ipart = 0
   do i = 1, npartmax
      if (levelp(i) > 0) then ! check if particle exists
         ipart = ipart+1 ! count the number of particles
         xdp(ipart) = xp(i, idim)
```

```
      end if
    end do
    ! write data
    call generic_dump("position_"//dim_keys(idim), ivar, xdp,
 unit_out, dump_info, unit_info)
 end do
 ...
 ! Write integer properties
 ...
```

Remark that the counter `ivar` is updated automatically in `generic_dump`.

Generalized, new particle fields can be added to the output by adding a new block of the form:

```
! allocate work array of the correcto type if needed
allocate(work_array(1:npart))

! Gather data into work array by looping over all allocated space
ipart = 0
do i = 1, npartmax
   ! check if particle exists
   if (levelp(i) > 0) then
      ipart = ipart+1
      ! retrieve value
      work_array(ipart) = my_particle_array(i)
   end if
end do
! send the data to be outputted
call generic_dump("my_field_name", ivar, work_array, unit_out,
dump_info, unit_info)

! deallocate work array if needed
deallocate(work_array)
```

Be careful to choose a work array of the type corresponding to the type of your new particle array. Give the variable an informative name.

6. Other files

Some other output files exist that do not follow the traditional rules. For example:

- clumpfinder: `write_clump_field` in *pm/output_clump.f90* (writes clump field)
- sinks: `output_sink_csv` in *pm/output_sink.f90*
- stellars: `output_stellar_csv` in *pm/output_stellar.f90*

7. Restarts

When restarting a simulation, ramses will read the information that was previously outputted. It is thus important that the parts of the code that read the input for restarting, match those of the outputting.

The restart parts are not encapsulated in their own routines, but can be found in the various init routines when searching for `if(nrestart>0)`.

Exercises

Exercise

Add a new variable to uold that will keep track of the temperature.

Exercise

Add a file descriptor for the gravity output.

Godunov solver

Contents

1. Solving hydro with the Finite Volume Method (FVM)

Several numerical methods exist to solve the hydrodynamic equations on a grid. RAMSES uses a finite volume method, more specifically an explicit second-order predictor-corrector Godunov scheme (see further), to integrate this conservative system of equations. This subset of methods is particularly well-suited for solving hyperbolic systems of conservation laws like the Euler equations. The method ensures conservation of mass, momentum, and energy across cell boundaries by construction.

In finite volume methods, the computational domain is discretized into control volumes V_i called cells. Remark that when using AMR, these volumes can have different sizes. Writing the system equation in integral form results in

$$\frac{d}{dt} \int_{V_i} \mathbb{U} \, dV + \int_{\partial V_i} \mathbb{F} \cdot \hat{n} \, dS, \, dA = \int_{V_i} \mathbb{S} \, dV$$

When defining the cell-averaged conserved quantities as

$$\mathbb{U}_i(t) = \frac{1}{|V_i|} \int_{V_i} \mathbb{U}(\mathbf{x}, t) \, dV ,$$

and the numerical flux through face f as $\mathbb{F}_f$, the semi-discrete update of the conservative variables becomes:

$$\frac{d \mathbb{U}_i}{dt} = -\frac{1}{|V_i|} \sum_{f \in \text{faces}} \mathbb{F}_f A_f + \mathbb{S}_i$$

with A_f the area of face f and $\mathbb{S}_i$ the cell-averaged source term.

Numerical fluxes $\mathbb{F}_f$ are obtained by solving Riemann problems at each interface between cells (see further), using the states on the left $\mathbb{U}_L$ and right $\mathbb{U}_R$ of the interface. For this, the interface states themselves first have to be interpolated from the cell-centered values. To compute the fluxes RAMSES used the MUSCL-Hancock scheme, which is a second-order Godunov method (more on this further).

Discretizing further, the conservative variable update becomes (in 1D):

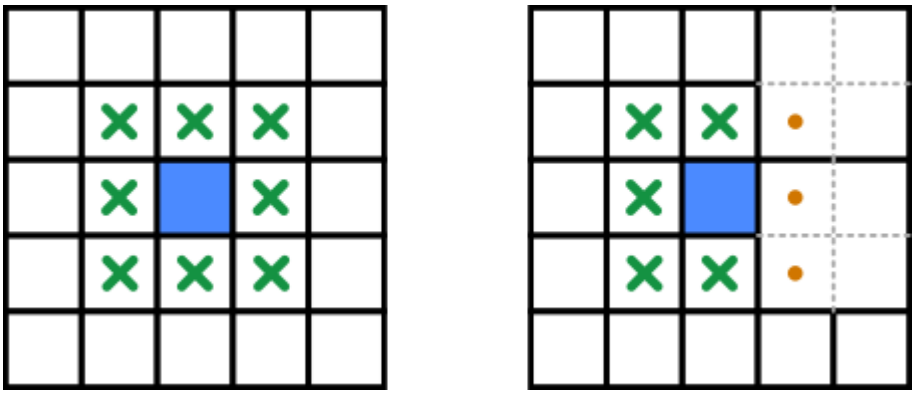
$$\mathbb{U}_i^{n+1} = \mathbb{U}_i^{n} - \frac{\Delta t}{\Delta x} (\mathbb{F}_{i+1/2} - \mathbb{F}_{i-1/2}) + \Delta t \, \mathbb{S}_i$$

where $\mathbb{F}_{i\pm1/2}$ are the fluxes going through opposing faces of the cell.

In the code, the entry point for hydro solver is `godunov_fine`. This routine uses the standard RAMSES structure to gather `nvector` cells to be send of to the core calculation routine `godfine1`.

Inside `godfine1`, the first step is to gather a stencil of 6 x 6 x 6 cells (in 3D) neighboring cells (this is needed to have second order accurary in the slope calculation, see later). If there is AMR, the neighboring cells may not be on the same refinement level and an interpolation is performed. In

that case, the coarse level is virtually refined at the fine level, and the hydro variables are interpolated (done by the routine `interpol_fine`).



Then, the subroutine `unsplit` is called, which calculates the fluxes. The name comes from the implementation being 'unsplit', meaning that the fluxes are computed simultaniously for all spatial directions (see further). The fluxes are always computed at the left side of the cell.

Finally, at the end of `godfine1`, `unew` is updated using the fluxes from all directions (unsplit method).

How source terms are treated will be discussed further.

2. Unsplit MUSCL-Hancock scheme for computing the fluxes

By default, RAMSES uses the MUSCL-Hancock scheme for computing the numerical fluxes across cell faces. This is a predictor-corrector extension of Godunov's method that allows for second-order accuracy by using

- piecewise linear reconstruction of the cell states, in contrast to Godunov's original piecewise constant
- a half step prediction for time evolution.

The scheme is extended to multiple dimension in an unsplit fashion, which is why the corresponding entry subroutine is named `unsplit`. This means that the fluxes in all directions are computed at the same time, that is in one timestep. Transverse corrections are included. This approach avoids splitting errors, better preserves symmetry and is overall more accurate for multi-dimensional simulations.

The steps for obtaining the flux are as follows.

- Convert conservative cell-centered variables to primitive variables (`ctoprim`)
- Calculate the limited slopes (TVD) for the primitive variables that will be used to reconstruct the state at the cell edges and evaluate space derivative for the time evolution below (MUSCL part, `uslope`)

- Evolve (or `trace`) the cell centered states forward in time for half a time step and then project on cell faces (Hancock part).

- Solve the Riemann problem at each interface using predicted left/right states to obtain the fluxes \(F_{i+1/2}\) `cmpflxm` ("compute flux minus", because only the left flux is calculated)

These fluxes are then used to update the conserved variables. Below, we go into each step in more detail. You can visit the following page for a more complete description of the MUSCL-Hancock scheme: https://ammar-hakim.org/sj/hancock-muscl.html

Switch to primitive variables

First, the conservative variables are converted into their primitive form. This is done by the subroutine `ctoprim`, which has as input the cell-centered conservative variables \(\mathbb{U}\) and as output the corresponding primitive variables \(\mathbb{Q}\).

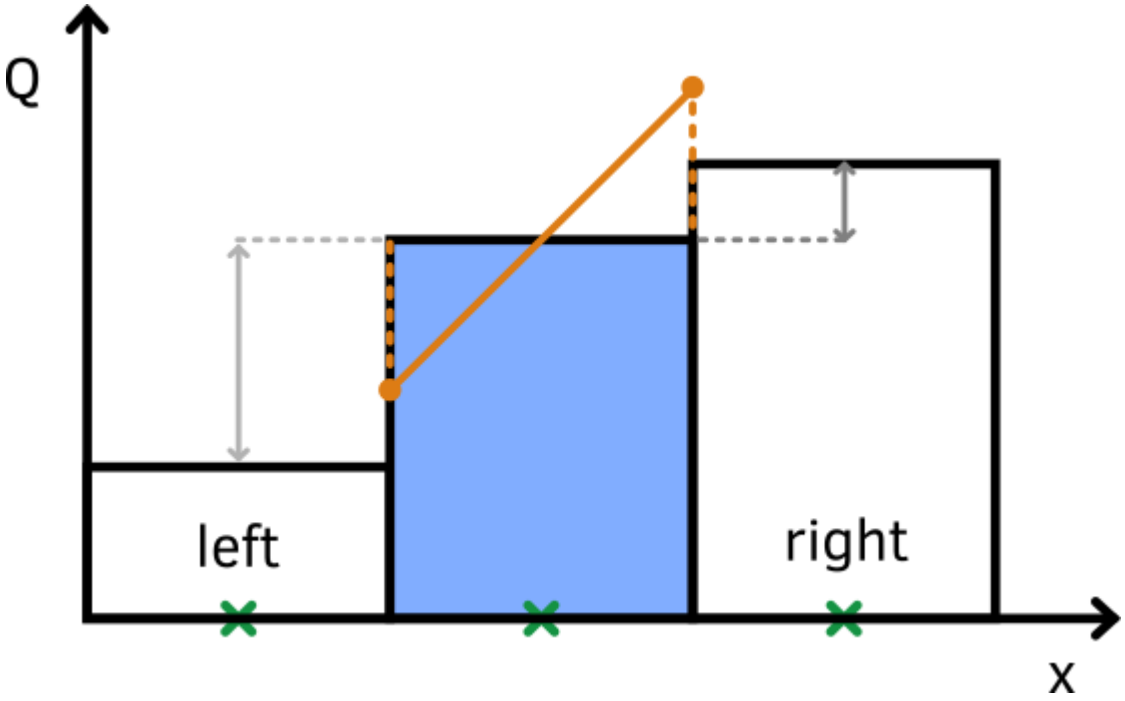
While not strictly required, it has become standard practice to use primitive variables for higher-order Godunov schemes like the MUSCL-Hancock scheme. This choice improves both accuracy and stability of the method. Primitive variables tend to vary more smoothly, especially across contact discontinuities and shocks, making them better suited for linear reconstruction and slope limiting. Reconstructing in conserved variables can introduce spurious oscillations or unphysical states such as negative densities or pressures. By switching to primitive variables during reconstruction and time prediction, the scheme maintains physical realism while achieving second-order accuracy.

A subtlety arises when gravity is included in the simulation. In that case, half a gravity predictor step is applied to the velocity in this routine. See more further.

Compute the slopes

In the MUSCL-Hancock scheme, the goal of the reconstruction step is to approximate the solution inside each cell using a linear profile, instead of a constant value. Given the cell-averaged primitive variables \(Q_i\), the slope \(\Delta Q_i\) is computed using the cell-centered values of the neighboring cells. A slope limiter is applied to control oscillations near discontinuities:

\(\Delta Q_i = \text{Limiter}(Q_{i+1}-Q_i, Q_i - Q_{i-1})\)



For example, the average slope without limiting would be simply

$\Delta Q_i = \frac{(Q_{i+1} - Q_i) + (Q_i - Q_{i-1})}{2}$,

which can introduce new extrema in the reconstructed field. This configuration is prone to oscillations. Total variation diminishing (TVD) limiters prevent the apparition of these spurious oscillations.

In RAMSES, calculating the limited slope is done by the subroutine `uslope`. Several slope limiters are implemented (minmod, monotonized centered, etc...).

Predictor step: evolution of cell boundary values for half a time step

The predictor step advances these states forward by half a timestep to account for their local evolution (Source term part) and reconstruct the state at the left and right interface of the cell. This increases temporal accuracy to second order.

First, the cell centered primitive variables are advanced for hal a timestep. The evolution is governed by the Euler equations, linearized around the reconstructed primitive states. In practice, this is done by applying the method of lines to estimate the time derivative and then updating the interface states. In 1D

$\rho^{n+1/2}_i=\rho^n_i - (u^n_i\Delta_x \rho^n_i+\rho^n_i\Delta_x u^n_i)\Delta t/\Delta x$

$u^{n+1/2}_i=u^n_i - (u^n_i\Delta_x u^n_i+\Delta_x P^n_i/\rho^n_i)\Delta t/\Delta x$

In the case of multiple dimensions, transverse corrections are applied, accounting for the coupling between spatial directions. These are cross-derivative terms that arise when wave propagation in

one direction is affected by gradients in perpendicular directions, which is crucial for preserving accuracy and symmetry in 2D and 3D flows.

In 2D

$$\rho^{n+1/2}_i=\rho^n_i - (u^n_i\Delta_x \rho^n_i+\rho^n_i\Delta_x u^n_i)\Delta t/\Delta x - (u^n_i\Delta_y \rho^n_i+\rho^n_i\Delta_y u^n_i)\Delta t/\Delta y$$

$$u^{n+1/2}_i=u^n_i - (u^n_i\Delta_x u^n_i+\Delta_x P^n_i/\rho^n_i)\Delta t/\Delta x - (v^n_i\Delta_y u^n_i)\Delta t/\Delta y$$

$$v^{n+1/2}_i=v^n_i - (u^n_i\Delta_x v^n_i)\Delta t/\Delta x - (v^n_i\Delta_y v^n_i +\Delta_y P^n_i/\rho^n_i)\Delta t/\Delta y$$

Then, we define left and right interface states at the boundaries of each cell:

$$Q_i^m = Q_i^{n+1/2} - \frac{1}{2} \Delta Q_i\\ Q_i^p = Q_i^{n+1/2} + \frac{1}{2} \Delta Q_i$$

These reconstructed values represent the linearly extrapolated states at each interface, from within cell ii. This reconstruction is applied in each spatial direction independently.

The predictor step is perfomed in the subroutine `trace`

Solving the Riemann problem to obtain the fluxes through cell interfaces

The last step is to compute the flux through the cell faces given the states to the left and right of the interface. This is a Riemann problem, i.e. a discontinuity with a value on the left and another value on the right, which can be solved numerically using Riemann solvers. RAMSES users have the choice between several different Riemann solvers. For more info on Riemann solvers, see elsewhere.

In the code, the subroutine `cmpflxm` is called for each spatial direction. Inside `cmpflxm`, the requested Riemann solver is called. The output array `flux` is then updated with the 1D flux for the requested spatial direction.

If AMR is active, the coarse level is updated during the fine level update at fine-to-coarse boundary.

Exercise

1. Find in the code where the coarse level is virtually refined. Does it imply some interpolation?
2. Find in the code where the coarse level update is done? What does it imply for conservative variable evolution? What about coarse-to-fine boundary?

Solution

1. in ‘umuscl.f90’. Quantities are conserved. The coarse-to-fine boundary is never considered, the flux being set to zero.

Credits

The general organization and conception of the first RAMSES Developer School was done by: J. Blaizot (Coord.), N. Brucy, C. Cadiou, T. Colman, B. Commerçon, M. Farcy, M. Gonzalez Rey, J. Rosdahl, J. Sorce, M. Trebitsch. The lecture notes were written by **J. Blaizot** (How to use Git, Radiative cooling and heating, subgrid modelling utilities), **N. Brucy** (Refinement schemes and implementation), **C. Cadiou** (MPI communications), **T. Colman** (overview of the code, hydrodynamics, mesh data structures, MPI communications, namelist parameters, initial conditions, outputting, Godunov solver, gravity), **B. Commercon** (Godunov solver, gravity), **J. Rosdahl** (Radiative cooling and heating), **M. Trebitsch** (Particles, subgrid modelling utilities).

Warning

NB: You will need a **GitHub account** to follow this tutorial or contribute to RAMSES. If you don’t already have one, please sign up at https://github.com/. We also strongly recommend that you **setup ssh keys** following the GitHub documentation (go to your account `settings`, into the `SSH and GPG keys` section, and follow instructions from there).

How to use Git

RAMSES uses Git to manage code development and version control. Git is a distributed version control system designed to track changes in source code during software development. It allows multiple developers to collaborate efficiently, manage different versions of a project, and revert to previous states if needed. With Git, one can work offline, create new branches for new developments, and merge them back to the main code seamlessly. While Git is a very powerful tool for community development, using it in practice does require a well-defined community strategy, and some basic understanding of Git’s inner workings. We have summarized the RAMSES Git strategy in a reference document, that we expand here with practical exercises.

There are many Git documentations online. We recommend this simple introduction or that one, as well as our recorded RAMSES SNO webinar on Git Fundamentals by Noé Brucy.

RAMSES Git model summary

The public version of RAMSES is maintained by the `ramses-organisation` account on GitHub, and it is accessible here : https://github.com/ramses-organisation/ramses.

The Git model of RAMSES relies on **one single branch**, called **dev**, which contains the latest public version of the code. Everyone can submit code updates to this branch through pull requests. These will then go through a validation process before they are merged into the main code or rejected. For obvious reasons, only a few selected persons can edit directly the code on the ramses-organisation repository, create branches there, or validate pull requests onto the public version. We thus encourage the following model:

1. Create your own fork of the ramses-organisation/ramses repository.
2. In your fork, create a branch out of the `dev` branch to start a new development. Make changes and commit regularly.
3. While developing, make sure to stay in sync it with the upstream version.
4. Once ready, submit a pull request to the upstream repository.

All these steps are explained in detail below.

1. Creating a new fork

The first thing to do is to **create your own fork of the ramses repository**. A fork is basically a copy of a repository. The original repository is called the **upstream** repository. In your fork, you can experiment and make changes without affecting the upstream version. The fork system allows you to synchronise your copy with the upstream repo, and to push your developments upstream through pull requests once they are done. An insightful documentation on forks is accessible at this link.

One can fork any repository, and **in this tutorial we will fork the repository ``Mentuhotep-II/ramses``**, which is itself a fork of the repository `ramses-organisation/ramses` that can be viewed as *the source*.

Exercise: Create a fork of the RAMSES repository. Follow the GitHub documentation but navigate to the forked ramses repository (Mentuhotep-II/ramses) instead of their “octocat/Spoon-Knife” example.

Once you have created your fork `YOUR-USERNAME/ramses`, you can get the code on your computer the usual way:

```
# using SSH :
git clone git@github.com:YOUR-USERNAME/ramses.git
```

2. Making changes in a new branch

Now that you have your own copy of the ramses repository, you can start making changes. In order to do this, a good practice is to create a new branch from the `dev` branch, in your fork. This new branch should have a name that refers to the development you are undertaking. In the terminal, this is done as follows:

```
cd ramses                          # go into your fork's directory
git checkout dev                   # go into the branch dev (of your
fork)
git checkout -b my-new-feature # create a new branch called my-new-
feature.
```

At this point, the new branch `my-new-feature` exists only locally on your computer. In order to make it visible/accessible online (in your fork), you need to *push* with the following instruction:

```
git push --set-upstream origin my-new-feature
```

Once this is done, you can check on GitHub page of your ramses fork (https://github.com/YOUR-USERNAME/ramses) that the branch `my-new-feature` appears in the branches.

As you develop, you should regularly *commit* your changes. This helps keeping track of your developments. With Git, a *commit* affects the history on your local computer only, and you also need to *push* developments to the original repository (i.e. your fork) online. We provide a list of basic Git instructions below that you will need.

⚠ Warning

Basic Git instructions

``git branch`` tells you on which branch you are and lists the local branches (use `git branch -r` to see the list of remote branches).

``git switch branch-name`` goes to branch `branch-name` (you can also use **``git checkout branch-name``** for that).

``git checkout -b new-branch`` creates a new branch (from the one you were in).

``git status`` lists the files that have been modified in the repository. This list also tells you which files are *staged for commit* (i.e. will be included in the next commit), and which are not.

``git add path/filename`` includes the file `path/filename` in the next commit.

``git commit -m "message"`` commits changes (staged for commit with `git add`) to your local history. The message passed in quotes should provide a quick description of the code modifications.

``git push`` pushes your local history to the original online repository.

Exercise: Create a new branch. On that branch, add a comment in the file `amr/constants.f90`, commit and push your changes. Use the GitHub web interface to check that your changes have been saved. :::spoiler Solution After editing the file and saving your changes use the following commands.

```
git status    # check which files have been modified
git add amr/constants.f90    # add amr/constants.f90 to the next
commit.
git commit -m "Added a test comment to amr/constants.f90."  # make
the commit.
git push      # push the commit online.
```

3. Staying in sync with the public version

The public version of the code is updated regularly, and it may happen that it changes while you are making changes on your fork. It is important to sync the `dev` branch of your fork regularly with the original (*upstream*) version. This will make things much easier later if you want to share your developments with the community, and it is also necessary to benefit from bug fixes.

3.1. Synching your fork

The simplest way to sync your fork with the upstream version is to click on the “Sync fork” button on the GitHub page of your fork.

You can also do this in the terminal as follows. First, setup the upstream version as the public version of RAMSES.

```
cd ramses       # go to your local repository
git remote add upstream git@github.com:ramses-organisation/ramses.git
git remote -v
```

This should print the following lines on the terminal:

```
origin git@github.com:YOUR-USERNAME/ramses.git (fetch)
origin git@github.com:YOUR-USERNAME/ramses.git (push)
upstream git@github.com:ramses-organisation/ramses.git (fetch)
upstream git@github.com:ramses-organisation/ramses.git (push)
```

Once this is done, you can sync with the following command.

```
git fetch upstream    # fetch the latest changes
git checkout dev      # go to the dev branch
git pull --rebase --autostash upstream dev   # do the magic ...
```

This last line will replay your local commits on top of the fetched changes and will automatically stash your local changes before pulling and reapply them after completion to avoid conflicts with uncommitted changes.

3.2. Merging the updated code into your branch

Once you have synched your fork, the `dev` branch on your fork matches the upstream version. You should now get the code on your computer with the following commands:

```
cd ramses           # go to your local ramses directory
git switch dev      # switch to the dev branch
git pull            # download the updated code
```

Once you have the updated code, you need to incoroporate it into your `my-new-feature` branch. This operation is called a **merge** in Git, and is done as follows.

```
git switch my-new-feature
git merge dev
```

This will likely generate **conflicts**, which are occurences where Git cannot decide automatically how combine code from two versions. In such cases, Git will return `CONFLICT` warnings with the list of files that are concerned. You need to edit the files to resolve the conflicts. Conflicts are highlighted with the following format:

```
<<<<<<< HEAD
  ! Here some code in my-new-feature.
  real(dp) :: my_new_feature_secret_variable
=======
  ! Here is some different code from dev, at the same location...
```

```
  real(dp),paramter :: thirteen = 13.0d0
>>>>>>> dev
```

In this simple example, you want to keep the two codes and simply have to remove the lines introduced by Git to produce:

```
! Here some code in my-new-feature.
real(dp) :: my_new_feature_secret_variable

! Here is some different code from dev, at the same location...
real(dp),paramter :: thirteen = 13.0d0
```

Once you have solved all conflicts, simply commit with

```
git commit  # Git will suggest an adequate commit message in this
case
git push    # broadcast your commit
```

Exercise: Set your upstream to Mentuhotep-II/ramses. Make some minor change in the file amr/constants.f90. Wait for a greenlight from the lecturer (who will introduce some change in the upstream version) and sync your fork with the upstream version.

4. Running the test suite

All changes should be tested to make sure they didn't break anything. This can be done by running the test suite as explained in the documentation. Note that they are also run on the github repository before any merge.

Exercise: Run the all the hydro tests. Make a change and see if the test suite still pass. Then explore the `tests` folder to understand how the tests are done.

5. Creating a pull request

Creating a pull request is the way to incorporate your developement in the upstream version of the code. The simplest way to do this, after you committed and pushed your work, is from the GitHub web interface. Simply click on the "Compare and Create a Pull Request" button and follow the instructions.

GitHub Management

All developments related to the RAMSES community are hosted under the Ramses organisation. The owner of the organisation is Romain Teyssier, but the management of the organisation and the repositories is done by the RAMSES community.

In order to encourage contributions and to promote a more inclusive and community-driven development, contributors to the code will be listed online (to reward their effort) and be granted permission based on their level of engagement. We distinguish three user roles:

- **Administrators**: Have full access to all repositories and can manage the organisation (`admin` role on GitHub). Their number should be kept to a minimum and they should be trusted members of the community. They are in charge of administrating the GitHub organisation. This includes notably creating/deleting repositories, managing user permissions, configuring default branches and protections on those.
- **Maintainers**: Have write access to the repositories and can merge pull requests (`maintain` role on GitHub). They are responsible for the day-to-day management of the repositories. This includes notably merging pull requests and closing solved issues. They should be active members of the community with a good understanding of the codebase.
- **Contributors**: Can manage issues and pull requests (`triage` role on GitHub). They are in charge of reviewing pull requests, suggest changes and approve pull requests, as well as answering issues. Anyone who has made a significant contribution to RAMSES, including code contribution or community management (e.g. creating GitHub issues or responding to other users). This role rewards active members of the community, and should be used to encourage new contributors, especially early-career researchers.

Contributors and user roles

Current members

Administrators

- Romain Teyssier
- Corentin Cadiou

Maintainers

- Tine Colman

- Noé Brucy

Contributors

- Robel Geda

Building the documentation

Some notes about the Sphinx and Read-The-Docs (RTD) documentation builders:

- The documentation has two sections, the user documentation section with files located in the 'wiki' directory and developer documentation in the `dev_docs` directory. When a PDF is generated, only the wiki version is included.
- Requirements for the docs can be installed via `pip install -r doc/rtd_requirements.txt`.
  - For MacOS pdf builds you may need to install `pango` as follows `brew install pango`. See `sphinx_simplepdf` docs at the link below.
- The build can be configured in `doc/conf.py`.
  - To build pdf locally run `sphinx-build -M html doc` or in the `doc` folder `make html`. The html will be built in `_build/html/index.html`
- To use markdown files, i.e `.md` files, the `myst_parser` extension is used. Documentation for `myst_parser`.
- To convert the docs to PDF, `sphinx_simplepdf` extension is used.
  - The RTD build is configured in `.readthedocs.yaml` under the `commands` list.
  - If `sphinx_simplepdf` ever breaks, you can remove the custom build instructions there.
  - To build pdf locally run `sphinx-build -M simplepdf . _build`.
  - Documentation for `sphinx_simplepdf`.
- As of this version, the color scheme for the docs are the following (can be changed in `doc/conf.py`):
  - `primary_color = '#000000'`
  - `secondary_color = '#FFD587'`
  - `text_color = '#000000'`

- If creating a page without listing it in the toctree, add the page to `doc/wiki/orphan_pages.rst`.
  - If you do not do this, you will get the following error: `WARNING: document isn't included in any toctree`.

Acknowledgements

The development of the RAMSES code has been initiated and coordinated by the main author, Romain Teyssier. The main author would like to thank all co-authors who took an active role in the development of this version. They are cited in alphabetical order.

- Dominique Aubert (radiative transfer, initial conditions)
- Edouard Audit (radiative transfer)
- Andreas Bleuler (sink particle, clump finder)
- Stephane Colombi (cooling and atomic physics)
- Benoit Commercon (radiative transfer, MHD)
- Stephanie Courty (cooling and atomic physics)
- Julien Devriendt (Hilbert curve)
- Emmanuel Dormy (MHD)
- Yohan Dubois (supernovae feedback, AGN feedback, MHD)
- Sebastien Fromang (MHD, relativistic HD)
- Claudio Gheller (GPU optimisation)
- Matthias Gonzalez (radiative transfer, initial conditions)
- Thomas Guillet (Multigrid Poisson solver)
- Patrick Hennebelle (MHD)
- Troels Haugboelle (MHD, gravity solver)
- Astrid Lamberts (relativistic HD)
- Davide Martizzi (AGN feedback, clump finder)
- Aake Nordlund (MHD, gravity solver)
- Joakim Rosdahl (radiative transfer)

- Yann Rasera (star formation)
- Philippe Series (code optimization)
- Neil Vaytet (automatic testing)
- Philippe Wautelet (code optimization)