# AMRD V2 Reference Manual

Frans Pretorius
*Department of Physics*
*Princeton University*
*Princeton, NJ, 08544*

Branson Stephens
*Department of Physics*
*Princeton University*
*Princeton, NJ, 08544*

**Contents**



## I. INTRODUCTION

AMRD (adaptive mesh refinement driver) is a library consisting of a main function, **amrd()**, which implements a Berger and Oliger (B&O) style adaptive mesh refinement (AMR) driver, with modifications to allow one to solve elliptic equations within the B&O time stepping framework, as outlined in [1]. A standard, adaptive, full approximation storage (FAS) multigrid (MG) algorithm is thus also provided. The user specifies the particular partial differential equations (PDEs) to solve via a set of "hook" functions that are called by AMRD at appropriate times. AMRD is built on top of PAMR [3], and so is parallel ready. AMRD is still very much under development, as is this document. Comments, questions, bug reports (no guarantees that they will be fixed!), etc. should be sent to

`fransp@phys.ualberta.ca`

The description of AMRD is sub-divided into the following sections. Section II describes how to invoke the driver, Section III lists all the hook functions, Section IV lists all the run-time parameters, and Section V describes a couple of simple example programs included in the distribution.

At this stage only a C-language interface exits, though a fortran interface may be forthcoming. The type **real** used throughout the interface is inherited from PAMR, and is currently defined as (see /include/internal_opts.h)

```
#define real double
```

**true** and **false** refer to C-style booleans, namely zero for **false** and non-zero for **true** (though these are *not* types defined in the PAMR headers).

Arrays are required to be indexed fortran style, i.e. first index varies most rapidly. Also, all array indices passed to and returned from various PAMR functions use fortran style numbering (element 1 is the first element in the array); however example code statements below are in C and adjust these numbers to C style indexing.

## II. INITIALIZATION

The driver is invoked via a single call:

```
void amrd(int argc, char **argv,
          int (*app_id)(void),
          void (*app_var_pre_init)(char *pfile),
          void (*app_var_post_init)(char *pfile),
          void (*app_AMRH_var_clear)(void),
          void (*app_free_data)(void),
          void (*app_t0_cnst_data)(void),
          real (*app_evo_residual)(void),
          real (*app_MG_residual)(void),
          void (*app_evolve)(int iter),
          real (*app_MG_relax)(void),
          void (*app_L_op)(void),
          void (*app_pre_io_calc)(void),
          void (*app_scale_tre)(void),
          void (*app_post_regrid)(void),
          void (*app_post_tstep)(int L),
          void (*app_fill_ex_mask)(real *mask, int dim, int *shape, real *bbox, real excised),
          void (*app_fill_bh_bboxes)(real *bbox, int *num, int max_num));
```

The **argc** and **argv** arguments are the standard C command line information passed via **main()**. At this stage the only command line argument interpreted by **amrd()** is the name of a parameter file. The call to **amrd()** starts the driver, which then reads all the options from the parameter file, initializes the PAMR context, and then starts the evolution. At various moments during this process, user *hook* functions, as specified by the **app_...**, arguments are called, and are expected to perform the actual PDE specific numerics. Hook functions passed in the **amrd()** parameter list that are not needed must still be defined, however they can immediately **return**. Since the first version of AMRD several new hook functions have been added; they are described in Sec.III H, and must be defined prior to the call to **amrd()** if used. All hook functions are described in the following section.

## III. USER HOOK FUNCTIONS

AMRD uses PAMR to manage the grid hierarchy. When calling a hook function that is expected to perform numerical operations on grid functions, AMRD initializes a sequential iterator in PAMR to loop through all the local grids (see the section *Grid-function access* in the PAMR reference manual [3]), and for each grid calls the desired hook function. Therefore, such hook functions can access all relevant grid information via the corresponding PAMR functions, but should *not* use any iterator functions; i.e., the numerical hooks only operate on single grids at a time, and do not need to know anything about grid distribution issues. In the listings below, hook functions that operate on individual grids have a */* grid function hook */* comment.

### A. Custom generation of the initial hierarchy

```
int app_id(void);
```

This function is called after all parameters have been read and the base level(s) [1] have been allocated. If the return is **true**, then the user has created all subsequent levels in the hierarchy and appropriately initialized them, and evolution starts immediately. Otherwise, one of AMRD's default hierarchy initialization mechanisms is invoked.

### B. Reading custom parameters

```
void app_var_pre_init(char *pfile);
void app_var_post_init(char *pfile);
```

These two hook functions are called before (**app_var_pre_init**) and after (**app_var_post_init**) AMRD parameters have been read from the parameter file **pfile** and the base hierarchy has been initialized. Several utility functions, listed below, are provided to read simple data structures in RNPL format [2] from the parameter file.

```
void AMRD_int_param(char *pfile, char *name, int *var, int size);
void AMRD_real_param(char *pfile, char *name, real *var, int size);
void AMRD_str_param(char *pfile, char *name, char **var, int size);
void AMRD_ivec_param(char *pfile, char *name, int *var, int size);
```

In all cases above, **pfile** is the parameter file name, **name** is the name of the parameter, **var** is a pointer to an array where the result should be stored, and **size** refers to the expected size of the array (note that currently for integer, real and string arrays, the size must *exactly* match the size of the object in the parameter file, if it is present). If the desired parameter is not in the parameter file, nothing is written to the memory locations (so pre-initialize with defaults).

The following 3 routines are version of the above that read in variable sized arrays, and are analogues of the corresponding "*_v*" functions in RNPL:

```
void AMRD_int_param_v(char *pfile, char *name, int **var, int *size);
void AMRD_real_param_v(char *pfile, char *name, real **var, int *size);
void AMRD_str_param_v(char *pfile, char *name, char ***var, int *size);
```

Prior to calling, *var* should be set to null, and afterward will point to the corresponding array of size *size*, or still null if the parameter was not found in the input file. *var* is allocated with *malloc* (as are the individual character arrays in *var* for the string case), and should be *free*'d by the caller.

### C. Specifying initial data

```
void app_AMRH_var_clear(void); /* grid function hook */
void app_free_data(void); /* grid function hook */
void app_t0_cnst_data(void); /* grid function hook */
```

[1] if the maximum possible depth of the hierarchy specified via **AMRD_max_lev** is greater than 1, then 'base' level consists of two fully refined levels due to the self-shadow hierarchy method used to compute truncation errors

These routines allow one to initialized grid functions in the hierarchy. **app_AMRH_var_clear()** is expected to set all variables in the AMR hierarchy to their "zero" values. **app_free_data()** is called during the generation of the initial hierarchy, and is expected to set the initial conditions for all freely specifiable fields at the initial time level (specified via **AMRD_ic_n**). **app_t0_cnst_data** is called after every multigrid V-cycle when calculating the initial data, and can be used to enforce algebraic (or other) constraints amongst initial data fields.

### D. Multigrid routines

```
real app_MG_residual(void); /* grid function hook */
real app_MG_relax(void); /* grid function hook */
void app_L_op(void); /* grid function hook */
```

These routines are required by the built-in FAS multigrid algorithm (with V-cycling), to solve elliptic equations of the form

$$\mathcal{L}[f] = 0, \tag{1}$$

where $\mathcal{L}$ is a differential operator acting on a variable $f$. The FAS algorithm introduces a new right hand side $R_f$ on coarser levels of the hierarchy, modifying the above equation to

$$\mathcal{L}[f] = R_f. \tag{2}$$

**app_MG_residual** returns some norm of the residual of the MG variables, *and* expects the point-wise residual $\mathcal{L}[f] - R_f$ to be stored in the grid function **f_res** for each MG variable **f**. $R_f$ is stored in the variable **f_rhs**. The function **app_MG_relax** is expected to perform one smoothing (relaxation) sweep of (2) over the grid, and return an estimate of the norm of the residual on that grid. **app_L_op** computes the result of the differential operator acting on the each MG variable **f** (i.e. $\mathcal{L}[f]$), in a region of the grid specified by the grid function **cmask**—where **cmask[i,j,...]==AMRD_CMASK_ON**—and stores the result in **f_lop**. Parts of the grid where **cmask[i,j,...]==AMRD_CMASK_OFF** should *not* be modified by the user routine.

### E. Evolution routines

```
void app_evolve(int iter, int *ifc_mask); /* grid function hook */
real app_evo_residual(void); /* grid function hook */
```

**app_evolve** is expected to perform 1 iteration of some evolution scheme, updating unknowns at the most advanced time level. The argument **iter** tells what the iteration (starting at 1) within the current time step is. The second argument was added for evolutions with hydro. It is an an integer array containing the flux correction mask which is needed for the Berger-Colella algorithm [4]. **app_evo_residual** returns some norm of the residual of the evolution equations.

### F. Excision routines

```
void app_fill_ex_mask(real *mask, int dim, int *shape, real *bbox, real excised);
void app_fill_bh_bboxes(real *bbox, int *num, int max_num);
```

**app_fill_ex_mask** is expected to set all points within excised regions in the grid function **mask** to **excised**, and some other value outside of excised regions. **NOTE:** this function is *not* called within a PAMR sequential iterator environment, and could be called at times when the hierarchy is undefined (in between a regrid). Thus no PAMR functions should be called during the execution of this hook. To this end, the mask grid functions' dimension (**dim**), shape array (**shape**), and coordinate bounding box (**bbox**) are passed as arguments.

**app_fill_bh_bboxes** is expected to fill an array of bounding box structures ($[x\_1, x\_2, y\_1, y\_2, ...]$), each describing the smallest rectangular region containing a given excised region in the domain. This is used with the **AMRD_TRE_sgpbh** option (see section IV G below). **bbox** is a preallocated array of memory of size **2*dim*max_num**. **max_num** is the maximum number of distinct excised regions that are supported, and upon return **num** should be set to the actual number of excised regions.

```
void AMRD_repopulate(int n, int def_order);
```

**AMRD_repopulate** is *not* a hook function; rather it is a utility routine that can be used to "repopulate" grid points if the excision mask moves during evolution. When this function is called, *all* hyperbolic (at *all* times) and elliptic variables, over *all* levels, are extrapolated **n** grid-points in from the existing excision boundary (using **def_order**-order extrapolation by default, but this can be changed on a variable-by-variable basis via the **ex_repop1_vars,...** excision options—see Sec.IV H). After the extrapolation, all **rg_diss_vars** are smoothed if **rg_diss_eps** $> 0$ (see Sec.IV G).

### G. Miscellaneous routines

```
void app_pre_io_calc(void); /* grid function hook */
void app_scale_tre(void); /* grid function hook */
void app_post_regrid(void); /* grid function hook */
void app_post_tstep(int L);
```

**app_pre_io_calc** is called prior to any output of grid functions to disk, to allow one to calculate any diagnostic grid functions outside of the evolution loop. **app_scale_tre** gives the user the option to manipulate the standard truncation error (TE) estimates for the set of variables for which such estimates are computed. Two global variables can be of help in this regard: **AMRD_num_f_tre_vars** is an integer describing the number of TE variables, and **AMRD_f_tre** is an array of **AMRD_num_f_tre_vars** pointers to the corresponding grid function data. After **app_scale_tre** is called, the actual TE estimate used by AMRD is calculated as the point-wise $\ell_2$ norm of this set of TE estimate variables. **app_post_regrid** is called after regridding. **app_post_tstep** is called after each evolution step has been completed. The level number of the most recent step is passed in **L**.

### H. Optional hook functions

Some hook functions are not specified in the **amrd()** argument list, but can be declared prior to the call to **amrd()** (this is the mechanism that should have been used to declare all hook functions, though for "historic" reasons only newer hooks are specified as such). The function **amrd_set_xxx_hook** is used to define the **xxx** hook function.

```
void amrd_set_app_pre_tstep_hook(void (*app_pre_tstep_f)(int L));
void app_pre_tstep(int L);
```

**app_pre_tstep** is called prior to taking a time step on level $L$.

```
void amrd_set_app_user_cp_hook(void (*app_pre_tstep_f)(int save_restore, char *data), int cp_data_size)
void app_user_cp(int save_restore, char *data);
```

**app_user_cp** is used for check-pointing (see Sec.IV J). Prior to saving to a check-point file, **app_user_cp** is called with **save_restore** equal to **AMRD_CP_SAVE**, and the user program can fill in a block of memory pointed to by **data**, of size **cp_data_size**, with any local state information that cannot be read from the parameter file. After restarting from a previously saved check-point file, **app_user_cp** is called with **save_restore** equal to **AMRD_CP_RESTORE**, and **data** will point to a block of memory containing the data previously saved.

Additional hook functions have been added for hydrodynamical evolutions. The following hooks were added for the flux correction step of the Berger-Colella algorithm [4]:

```
void app_fcs_var_clear(int, int *);
void amrd_set_app_fcs_var_clear_hook(void (*app_fcs_var_clear)(int, int *));
void app_flux_correct();
void amrd_set_app_flux_correct_hook(void (*app_flux_correct)());
void app_post_flux_correct();
void amrd_set_app_post_flux_correct_hook(void (*app_post_flux_correct)());
```

The following two hooks were added because of special treatment required by conserved hydrodynamic variables before they are injected or interpolated. Namely, "extensivization" refers to multiplying a grid function by the local volume element at each cell center. This is done before communication, and then undone ("intensivization") after the communication step.

```
extern void (*app_extensivize_intensivize)(int);
void amrd_set_app_extensivize_intensivize_hook(void (*app_extensivize_intensivize)(int));

extern void (*app_extensivize_intensivize_mg)(int);
void amrd_set_app_extensivize_intensivize_mg_hook(void (*app_extensivize_intensivize_mg)(int));
```

These functions became necessary because of the coordinate compactification, which causes the ratio of cell volumes to differ slightly from the exact ratio expected from the refinement ratio. This breaks mass conservation unless you first multiply, for example, the density by the volume of each cell. The grid function now contains the mass inside each cell. These masses are then injected to the underlying coarse grid by simply adding them together. After the injection, the density grid function is divided by the volume elements to restore it to its original state. There is a separate version of the hook for multigrid because it must call the appropriate pointer initialization subroutine.

## IV. OPTIONS

The following describe all of AMRD's options, specified in the input parameter file. These options include specifying the number and type of variables, and controlling various aspects of the AMR/MG algorithms. Most of the options are declared as global variables (with an AMRD_ prefix that does *not* appear in the parameter file names), and so can be accessed by the user programs during execution (see the amrd.h header file).

**NOTE:** RNPL is used to read parameters from the file, and from AMRD V2 onwards a later version of RNPL that can handle arbitrary length array parameters is required. A consequence of this is that the "**num_p**" integers for **p** array variables required in the earlier version of AMRD are no longer used.

The format of the parameter listings below are as follows:

**name** (*type*, default=...):

Current scalar types include *real, integer, string* and *ivec*. Array types are shown as *real[size], integer[size]* and *string[size]*. The size is either a constant, or as discussed above, a integer size variable that must be declared elsewhere in the parameter file. Some parameters do not have defaults, and are required to be present in the input file.

### A. Variable definition

The following parameters tell AMRD what variables to define over the grid hierarchy. Variable options (such as the kind of interpolation, restriction, etc.) are discussed in later sections.

**hyperbolic_vars** (*string[...]*, default=[]): The list of "hyperbolic" variables. They are defined in both the AMR (for all **num_evo_tl** time levels) and MG hierarchies, but are assumed to play the role of passive "source" functions during MG.

**elliptic_vars** (*string[...]*, default=[]): The list of "elliptic" variables. They are defined in both the AMR (for all **num_evo_tl** time levels) and MG hierarchies, but are assumed to play the role of passive "source" functions during evolution time steps. For each elliptic variable **f** that is defined here, AMRD will create the following list of additional variables (see Sec. III D for a description of the MG variables; the extra AMR variables are used to implement the "extrapolation and delayed solution" technique to deal with elliptics in Berger and Oliger AMR, as discussed in [1]): **f_rhs** (in MG hierarchy), **f_res** (MG), **f_lop** (MG), **f_rhs** (MG), **f_brs** (AMR, one time level only), **f_extrap_tm1** (AMR, one time level only), **f_extrap_tm2** (AMR, one time level only).

**elliptic_vars_t0** (*string[...]*, default=[]): The list of "elliptic" variables that *only* exist at $t = t_0$ for the purposes of initial data calculation. They are only defined in the MG hierarchy.

**AMRH_work_vars** (*string[...]*, default=[]): The list of AMR "work" variables; i.e. variables that are only defined in the AMR hierarchy at one time level.

**MGH_work_vars** (*string[...]*, default=[]): The list of MG "work" variables; i.e. variables that are only defined in the MG hierarchy.

**TRE_vars** (*string[...]*, default=[]): A subset of the list of **hyperbolic_vars** that will be used for truncation error estimated. For each variable **f** listed here, a one time level AMR variable **f_tre** will be created to hold the corresponding variable's truncation error estimate (only calculated prior to regridding).

**fc_vars** (*string[...]*, default=[]): A subset of the list of **hyperbolic_vars** that will require flux corrections according to the Berger-Colella algorithm [4]. This should be the list of “conserved” evolution variables in a set of hyperbolic conservation laws (as in hydrodynamics, MHD, etc.). For each variable **f** listed here, a one time level AMR variable **f_fcs** will be created to store the corresponding variable's accumulated flux correction.

**past_bdy_interp_vars** (*string[...]*, default=[]): For each substep in a hydrodynamics evolution, AMR boundary conditions on conserved hydro variables must be supplied by interpolation on *both* the origin and destination time levels. This requires modification of the “past” time level, hence the name of the list. This is in contrast to the usual AMR boundary condition procedure, which applies only to the *destination* time level.

### B. Hierarchy definition

The following parameters define basic properties of the grid hierarchy.

**dim** (*integer*): The spatial dimension, which currently can be 1,2 or 3.

**num_evo_tl** (*integer*): The number of time levels in the AMR hierarchy.

**ic_n** (*integer*, default=2): Which time level is considered $t = t_0$ for initial data calculation.

**base_shape (*integer[dim]*, default=[3,3,...])**: The base grid shape.

**base_bbox (*real[2*dim]*, default=[-1,1,-1,1,...])**: The base grid coordinate bounding box.

**max_lev** (*integer*, default=1): The maximum depth of the AMR hierarchy. **max_lev=1** is therefore unigrid, and note that because of the “self-shadow hierarchy” technique used to compute truncation error estimates, if **max_lev>1** level 2 is always fully refined, hence the first two levels effectively become the base level in an AMR evolution.

**t0** (*real*, default=0.0): The initial time.

**rho_sp** (*integer*, default=2): This variable sets the spatial refinement ratio for all levels in the hierarchy to the *same* value **rho_sp**.

**rho_tm** (*integer*, default=2): This variable sets the temporal refinement ratio for all levels in the hierarchy to the *same* value **rho_tm**.

**rho_sp_all** (*integer[max_lev]*, default=[2,...]): **rho_sp_all[i-1]** defines the spatial refinement ratio of level **i** (taking precedence over any **rho_sp** declaration).

**rho_tm_all** (*integer[max_lev]*, default=[2,...]): **rho_tm_all[i-1]** defines the spatial refinement ratio of level **i** (taking precedence over any **rho_tm** declaration).

**periodic** (*integer[dim]*, default=[0,...]): If set to a number other than 0, the corresponding spatial boundary is considered periodic, and PAMR will take care of properly enforcing periodic boundary conditions.

**lambda** (*real*, default=1.0): The local CFL factor; i.e. **dt=lambda*min(dx,dy,...)**.

### C. Parallel options

**ghost_width** (*integer[dim]*, default=[2,...]): The size (in units of the grid spacing, *i.e.* cell width) of the ghost region to add along interior grid boundaries that are created by PAMR when splitting a grid across several nodes.

**gdm_grid_by_grid** (*integer*, default=[0,...]): The grid distribution method. Currently only two options are supported, level-by-level (default) and grid-by-grid (**gdm_grid_by_grid!=0**). With level-by-level, PAMR will try to split the volume taken up by all the grids at a given level into $N$ pieces, where $N$ are the number of nodes partaking in the run. With grid-by-grid, PAMR will try to split the volume taken up by *each* grid into $N$ pieces.

**gdm_align** (*integer*, default=[0,...]): If non-zero then ghost regions are padded so that all local child grid boundaries lie on parent grid-lines.

**gdm_no_overlap** (*integer*, default=[0,...]): If non-zero, then all overlapping grids in the hierarchy, prior to distribution in a parallel environment, are clipped to remove the overlap (option not yet implemented in PAMR).

**min_width** (*integer[dim]*, default=[3,...]): The minimum size that a grid could get split into.

### D. AMR/evolution options

**steps** (*integer*, default=1): The number of coarse level (1) time steps to perform.

**evo_max_iter** (*integer*, default=50): The maximum number of iterations per time step.

**evo_min_iter** (*integer*, default=1): The minimum number of iterations per time step.

**evo_tol** (*real*, default=0.0): The tolerance for the hyperbolic equations during the evolution step; i.e. if the residual returned by the hook function **app_evolve** is less than or equal to **evo_tol** the equations are considered solved.

**evo_ssc** (*integer*, default=1): This flag, when non-zero, specifies separate stopping criteria for hyperbolic and elliptic equations on the finest level on an evolution step. This means that if the hyperbolic residual drops below **evo_tol** while the elliptic residual is still above **MG_tol**, then subsequent iterations only call the MG routines, and vice-versa. If **evo_ssc**=0, then the MG and hyperbolic evolution routines are called until both residuals drop below their respective tolerances.

**np1_initial_guess** (*real*, default=0): How the advanced time level $t^{n+1}$ is initialized prior to the evolution step. Current options are 0 and 1. **np1_initial_guess**=0 means "do nothing", and since time-levels are cyclically switched from one time step to the next, the advanced time level will contain data from the most retarded time level of the previous time step. **np1_initial_guess**=1 copies the data from $t^n$ to $t^{n+1}$.

**MG_extrap_method** (*integer*, default=0): (experimental ... leave at the default for now)

**eps_diss** (*real*, default=0): Kreiss-Oliger style dissipation parameter for both **tn_diss_vars** and **tnp1_diss_vars**.

**tn_eps_diss** (*real*, default=0): Kreiss-Oliger style dissipation parameter for **tn_diss_vars**; overrides the **eps_diss** if present.

**tnp1_eps_diss** (*real*, default=0): Kreiss-Oliger style dissipation parameter for **tnp1_diss_vars**; overrides the **eps_diss** if present.

**diss_bdy** (*integer*, default=0): Set to 1 to extend the dissipation stencil all the way to grid boundaries (following [5] otherwise only interior points are smoothed.

**tnp1_diss_vars** (*string[...]*, default=[]): The list of most-advanced time level ($t^{n+1}$) grid functions to smooth *after* an evolution step has completed.

**tn_diss_vars** (*string[...]*, default=[]): The list of next-to-most-advanced time level ($t^n$) grid functions to smooth *before* an evolution step has begun.

**diss_all_past** (*integer*, default=0): If non-zero, specifies that all past time levels of variables in **tn_diss_vars** should be smoothed (using **tn_eps_diss**) prior to the evolution step, not only level $t^n$.

**tnp1_liipb_vars** (*string[...]*, default=[]):

**tnp1_liiab_vars** (*string[...]*, default=[]):

**tnp1_liibb_vars** (*string[...]*, default=[]):

Experimental ... reinitialize these variables via linear interpolation next to (1 grid point in from) AMR (**tnp1_liiab_vars**), physical (**tnp1_liipb_vars**) or both (**tnp1_liibb_vars**) boundaries after each iteration of the hyperbolic equations.

**interp_AMR_bdy_vars** (*string[...]*, default=[]): Experimental (intended for MG variables) ... initialize these variables at their AMR boundaries, at points that have no parent point, via interpolation from points that do, before the evolution step.

**interp_AMR_bdy_order** (*integer*, default=4): The order of interpolation to use for **interp_AMR_bdy_vars**.

**max_t_interp_order** (*integer*, default=2): The (maximum) order of temporal interpolation to use to set coarse level, AMR boundary conditions. At this stage only second and third order interpolation is supported, and third order interpolation requires at least 3 time levels.

**t_interp_substeps** (*integer[...]*, default=[]): For time-stepping schemes that have substeps integrating to fractional times between $t_n$ and $t_{n+1}$, this array tells AMRD what fractional time to associate with level $n+1$ at each substep. Then, at iteration $i$ (beginning at $i = 1$), boundaries will be set via interpolation in time from the parent level to time $t_n + t\_interp\_substeps[i-1] * dt$. If no values are supplied for this variable, or more iterations than elements are taken, then a value of 1 is used. For example, a typical 4th order Runge-Kutta integration will have $t\_interp\_substeps = [0, 0.5, 0.5, 1]$.

**re_interp_width** (*integer*, default=0): If $> 0$, then after fine level evolution, but just before the fine-to-coarse level injection step, a region of size **re_interp_width** points on the fine level adjacent to AMR boundaries are reinterpolated from the parent (coarse) to child (fine) grids.

**re_interp_width_c** (*integer*, default=0): Same as above, except for cell centered data. This is crucial for hydro evolutions, since the AMR boundaries must be re-interpolated *before* injection to preserve conservation. Moreover, the interpolation itself must be done conservatively (**PAMR_FIRST_ORDER_CONS**).

**amr_inject** (*string[...]*, default=[]): The list of variables to be injected from a fine level to the next coarser levels, as per the B&O scheme. For cell centered data, injection is done with nearest-neighbor averaging (**PAMR_NN_AVERAGE**).

**amr_inject_extensive** (*string[...]*, default=[]): The list of variables which need to be multiplied by the local volume element before injection, and then divided by it afterward (see Section III H). It is envisioned that this will only be applied to cell-centered variables and that **PAMR_NN_ADD** will be used as the extensive injection type.

**amr_interp2** (*string[...]*, default=[]): The list of variables that will have their AMR boundaries set via $2^{nd}$ order interpolation in space and time from the next coarser level, as per the B&O scheme. For cell centered data, this interpolation is done with the MC limiter (**PAMR_MC**).

**amr_interp4** (*string[...]*, default=[]): The list of variables that will have their AMR boundaries set via $4^{th}$ order interpolation in space and $2^{nd}$ order interpolation in time from the next coarser level, as per the B&O scheme.

**amr_interp_extensive** (*string[...]*, default=[]): The list of variables which need to be multiplied by the local volume element before interpolation, and then divided by it afterward (see Section III H). It is envisioned that this will only be applied to cell-centered variables and that **PAMR_FIRST_ORDER_EXTENSIVE** will be used as the extensive injection type.

**amr_sync** (*string[...]*, default=[]): The list of variables that need to have their ghost regions synchronized after each iteration of the evolution equations.

**amr_transfer2** (*string[...]*, default=[]): The list of variables that will be initialized on a new patch of fine level after a regrid, either by copying data from an existing overlapping patch on the same level, or by $2^{nd}$ order spatial and temporal interpolation from the next coarser level.

**amr_transfer4** (*string[...]*, default=[]): The list of variables that will be initialized on a new patch of fine level after a regrid, either by copying data from an existing overlapping patch on the same level, or by $4^{th}$ order spatial and $2^{nd}$ order temporal interpolation from the next coarser level.

**amr_transfer_extensive** (*string[...]*, default=[]): The list of variables that will be initialized on a new patch of fine level after a regrid, either by copying data from an existing overlapping patch on the same level, or by $1^{st}$ order extensive interpolation from the next coarser level. Note that this type of transfer is only envisioned for hydro variables, which do not need temporal interpolation.

**amr_bdy_interp1** (*string[...]*, default=[]): This list allows the user to specify different interpolation types in the AMR boundary and the bulk. Currently, this list applies only to cell-centered variables (i.e., vertex-centered variables always use the same interpolation type in the AMR boundaries as in the bulk). Variables in this list will be interpolated using a first order conservative scheme in the AMR boundaries. In practice, this is necessary when one wants to do global interpolations on the extensive version of the grid function. However, interpolating the extensive quantity is not necessary in AMR boundaries, since strict conservation is not needed there.

**amr_bdy_interp2** (*string[...]*, default=[]): Same as previous, except that variables in this list will be interpolated using a second order scheme (currently the MC limiter) in the AMR boundaries.

The following options define lists of variables that have even or odd character across a given physical boundary. At this stage these flags are only utilized when applying dissipation.

**even_vars_x0min** (*string[...]*, default=[]):

**even_vars_x0max** (*string[...]*, default=[]):

**even_vars_x1min** (*string[...]*, default=[]):

**even_vars_x1max** (*string[...]*, default=[]):

**even_vars_x2min** (*string[...]*, default=[]):

**even_vars_x2max** (*string[...]*, default=[]):

**odd_vars_x0min** (*string[...]*, default=[]):

**odd_vars_x0max** (*string[...]*, default=[]):

**odd_vars_x1min** (*string[...]*, default=[]):

**odd_vars_x1max** (*string[...]*, default=[]):

**odd_vars_x2min** (*string[...]*, default=[]):

**odd_vars_x2max** (*string[...]*, default=[]):

### E. MG options

**min_mg_cwidth** (*integer[dim]*, default=[3,...]): The minimum grid size for the coarsest level in the MG hierarchy (independent of parallelization issues).

**MG_max_iter** (*integer*, default=50): The maximum number of vcycles to perform.

**MG_min_iter** (*integer*, default=1): The minimum number of vcycles to perform.

**MG_pre_swp** (*integer*, default=3): The number of pre CGC (coarse-grid-correction) smoothing sweeps to perform.

**MG_pst_swp** (*integer*, default=3): The number of post CGC smoothing sweeps to perform.

**MG_tol** (*real*, default=0.0): The tolerance for the MG equations i.e. if the residual returned by the hook function **app_MG_residual** is less than or equal to **evo_tol** the equations are considered solved.

**MG_crtol** (*real*, default=1.0e-3): Currently AMRD "solves" the coarsest grid equations using relaxation. The parameter **MG_crtol** specifies how much the residual on the coarsest level should be reduced by *relative* to the current next-to-coarsest residual, for the coarse grid problem to be considered solved.

**MG_w0** (*real*, default=1.0): An "under-relaxation" parameter. The residuals driving the RHS's of the coarser grid differential operators are multiplied by **MG_w0**, and correspondingly the coarse grid corrections are multiplied by **1/MG_w0** before being applied to the finer level. A value of **MG_w0** on the order of 0.9 to 0.95 is useful for certain PDEs.

**MG_eps_c** (*real*, default=1.0): A "correction" to apply to the extrapolation of MG variables on the coarser levels during a constrained evolution—see Section 2.70 of [1].

**mg_interp2** (*string[...]*, default=[]): The list of variables that will have linearly interpolated coarse grid corrections applied during the vcycle.

**mg_hw_restr** (*string[...]*, default=[]): The list of variables that will be restricted to coarser levels in the MG hierarchy via half-weight restriction.

**mg_fw_restr** (*string[...]*, default=[]): The list of variables that will be restricted to coarser levels in the MG hierarchy via full-weight restriction.

**mg_sync** (*string[...]*, default=[]): The list of variables that need to have their ghost regions synchronized after each smoothing sweep.

### F. Initial data/hierarchy options

**id_method** (*integer*, default=0): The method used to generate the initial hierarchy. Current options are

**0**: use truncation error estimates of **TRE_vars** variables, calculated by taking a set of single coarse-step evolutions;

**1**: use a truncation error estimate of the elliptic equations from the initial MG solve.

**id_pl_method** (*integer*, default=0): The method used to initialize past time levels. Current options are

**0**: first order extrapolation (i.e. straight copy);

**1**: first order extrapolation, then evolve the entire hierarchy backwards, then forwards one coarse step via the B&O time-stepping scheme;

**2**: evolve all levels of the hierarchy backwards in time by **id_pl_steps** "small" steps (as specified by a time step **id_pl_lambda*dx_f**, where **dx_f** is the smallest mesh spacing in the hierarchy), and then extrapolate/interpolate the solution to the initial retarded time levels.

**3**: no nothing (or user will do it somewhere else, such as via **app_t0_cnst_data**). Do *not* use this option with constrained evolution, as the past data needed to extrapolate the MG variables will not be initialized.

**id_pl_steps** (*integer*, default=1): see **id_pl_method**

**id_pl_lambda** (*real*, default=0.1): see **id_pl_method**

**MG_cnst_data_vars** (*string[...]*, default=[]): The list of variables that will be set via **app_t0_cnst_data** following each vcycle.

**MG_cnst_data_vars** (*string[...]*, default=[]):

**init_depth** (*integer*, default=2, maximum=10):

**init_bbox_#** (*real[2*dim]*):

**init_bbox_b_#** (*real[2*dim]*):

# is a number between 3 and **init_depth**. If **init_depth** is greater than 2, then an initial guess to the grid structure of levels 3..**init_depth** can be specified via bounding boxes **init_bbox_#** (required) and **init_bbox_b_#** (optional). Thus, at present two initial grids per level can be specified. Note that it is the users responsibility to make sure that the bounding boxes "fit" into the hierarchy; i.e., no bounds checking, nor grid alignment is performed. Also, depending on the initial hierarchy construction options, the specified structure may not be the one that is actually used for the initial hierarchy (you can freeze this initial hierarchy in by using the **regrid_min_lev** option, setting it to **init_depth**).

### G. Regridding/clustering options

**TRE_max** (*real*, default=0.0): The maximum estimated truncation error. The truncation error at each point is calculated as the $\ell_2$ norm of the estimated truncation errors of all the variables specified in **TRE_vars**. This is then used, modulo buffers, to determine the set of grids in the hierarchy. The estimated truncation error of a given variable is calculated using the self-shadow hierarchy technique, as described in [1].

**TRE_norm** (*integer*, default=0): If non-zero, then the TE estimate of each variable is divided by an approximate $\ell_2$ norm of the magnitude of the corresponding variable.

**TRE_buffer** (*integer*, default=0): The number of buffer cells (relative to the parent level) to add to the region of high TE.

**TRE_ibc_buffer** (*integer*, default=0): The number of buffer cells adjacent to AMR boundaries in which to clear (zero) the TE estimate.

**TRE_ibcp_buffer** (*integer*, default=0): Experimental ... the same as **TRE_ibc_buffer**, except the calculation is performed *locally*; i.e. **TRE_ibcp_buffer** clears the TRE adjacent to parallel interior boundaries in addition to AMR boundaries, and hence the result could depend rather strongly on the number of nodes and grid distribution.

**TRE_exc_buffer** (*integer*, default=0): The number of buffer cells around excised regions in which to clear (zero) the TE estimate, on levels greater than or equal to **TRE_exc_buffer_lmin**

**TRE_exc_buffer_lmin** (*integer*, default=0): see above

**regrid_interval** (*integer*, default=4): How often, in steps, to regrid on each level.

**regrid_min_lev** (*integer*, default=2): The minimum level beyond which regridding can occur (in other words, levels **1..regrid_min_lev** are fixed in time).

**cls_method** (*integer*, default=0): Clustering method. Currently the only method offered is to cover regions of high TE with minimal rectangles.

**cls_merge_dist** (*integer*, default=0): If clusters are within **cls_merge_dist** grid points of one another, then merge them into a single encompassing cluster.

**cls_align_mode** (*integer*, default=0): MG imposes restrictions on child grid alignment; if **cls_align_mode=0(1)** clusters are shrunk (expanded), if necessary, to conform to MG's requirements.

**TRE_sgpbh** (*integer*, default=0): If non-zero specifies a single grid (before parallel distribution) per black hole when excising.

**skip_frg** (*integer*, default=1): If non-zero then skips the first regrid.

**rg_diss_vars** (*string[...]*, default=[]): A list of variables to apply standard Kreiss-Oliger dissipation to after regridding.

**rg_eps_diss** (*real*, default=0.0): The amount of dissipation to apply to **rg_diss_vars** variables.

**regrid_script** (*integer*, default=0): The grid hierarchy can be recorded (**regrid_script=1**) to an ASCII file during evolution, or read (**regrid_script=2**) from such a file to by-pass the TE-based hierarchy mechanism. This allows one to (for instance) do a "standard" convergence test. **NOTE:** when reading the hierarchy structure from a script, *no* checking is done to ensure that the grids are consistent with MG, are properly nested, etc. So modify such scripts at your own risk!

**regrid_script_name** (*string*, default=[]): The file name to read or write a regrid script, if **regrid_script!=0**.

### H. Excision options

**do_ex** (*integer*, default=0): If $> 0$, then turn on PAMR's excision support functions. Furthermore, a variable called "chr" will be defined to store the excision characteristic mask, where a point is excised if the corresponding value in chr is set to **ex** below. *special value: if* **do_ex**$< 0$*, then excision is turned on, and "chr" will be used as the "eps" array with dissipation*

**ex** (*real*, default=1.0):

**ex_repop#_vars** (*string[...]*, default=[]): During a call to **AMRD_repopulate**, the list of variables that will be extrapolated using #-order extrapolation (currently # can be between 1 and 4), over-riding the **def_order** parameter of **AMRD_repopulate** (see Sec.III F).

## I. Output options

**save_tag** (*string*, default=[]): A character string to attach to front all output variable file names.

**save_ivec** (*ivec*, default=[]): An index vector specifying the set of coarse grid times to save the desired grid functions on *all* levels of the hierarchy to disk (in the current directory).

**save_mg_vars** (*string[...]*, default=[]): The list of MG variables to save.

**save_#_vars** (*string[...]*, default=[]): The list of AMR variables to save, where the # refers to the time level number. Note that for a 2 or more time level scheme, grid functions are saved to disk when time level 2 is the most advanced time (specifically ,the time sequence is 2,3,4,...,bf num_evo_tl,1).

**save_ivec_#** (*ivec*, default=[]): Index vectors specifying the set of times to save level # grid functions to disk.

## J. Check-pointing

The following flags control PAMR/ARMD's checkpointing mechanism:

**cp_save_fname** (*string*, default=""): A file name tag for saving new check-point files.

**cp_restore_fname** (*string*, default=""): A file name tag for restoring the state from an existing set of check-point files.

**cp_restart** (*boolean*, default=**false**): If **true**, then evolution is continued from a prior set of check-point files, with tag **cp_restore_fname**. The number of nodes in the new run does not need to be the same as the run that created the check-point data.

**cp_first_delta_t_hrs**, (*integer*, default=0): If greater than zero, the approximate amount of run-time to elapse before the first check-point.

**cp_delta_t_hrs**, (*integer*, default=0): If greater than zero, the approximate amount of run-time between check-points (except possibly the first one, with can be set with **cp_first_delta_t_hrs** above).

When check-pointing during an **n** node parallel run, the internal state of the entire hierarchy is saved to a set of files *cp_save_fname_X_#.sdf, cp_save_fname_AMRD_X_#.sdf*, where $X \in A..Z, a..z$ is a letter labeling the particular check-point, and $\# \in 0..(n-1)$ is the rank of the node. $X$ starts at $A$, and after each checkpoint is "incremented". After $Z$, the label changes to $a$, and after $z$ the label resets to $A$. **Thus, at present, a maximum of 52 unique check-point files can be written per run, and after the label wraps around old check-point files will be over-written.**

When restarting a run, many of the parameters are retrieved from the parameter file. Care should be taken when changing parameters upon a restart. For example, it is "safe" to change dissipation parameters, truncation error estimate thresholds, the maximum depth of the hierarchy (the change will only take effect after the next regrid), the list of output variables, and similar parameters. Others, such as the grid resolution, the CFL factor (if during an AMR evolution with more than one time level and the temporal refinement ratio is greater than 1), etc., should not be altered (the changes will either be ignored, or could have unpredictable effects).

Internal user parameters can be saved/restored using the **app_user_cp** hook; see Sec.III H.

## K. Miscellaneous options

The **..._DV_trace** options below will eventually allow one to save variables at intermediate points in an iteration for debugging purposes, though at this stage these options are not fully supported yet.

**echo_params** (*integer*, default=1): If non-zero, then all parameters are echoed to the screen at run time, in addition to the list of PAMR variables that have been defined.

**pamr_trace_lev** (*integer*, default=0): see the PAMR reference manual [3]

**MG_trace** (*integer*, default=0):

**MG_DV_trace** (*integer*, default=-1):

**MG_DV_trace_t_on** *real*, default=1.0e-15):

**MG_DV_trace_t_off** *real*, default=1.0e-15):

**ID_DV_trace** (*integer*, default=0):

**evo_DV_trace** (*integer*, default=0):

**evo_DV_trace_t_on** *real*, default=1.0e-15):

**evo_DV_trace_t_off** *real*, default=1.0e-15):

**evo_trace** (*integer*, default=0): Set from 0...3 to have the program output less...more information on the iteration during each evolution step.

**calc_global_var_norms** (*integer*, default=0): *descriptions to follow*

**calc_global_var_norm_type** (*integer*, default=1):

**calc_global_var_norm_floor** (*real*, default=1.0):

## V. EXAMPLES

This section describes the two example programs **wave** and **nbs** included in the distribution.

### A. wave

**wave** is located in the directory

```
.../pamr/examples/wave/
```

This program solves a 1,2 or 3D flat space wave equation $\Box\Phi = 0$ in a rectangular domain, with either Dirichlet or period boundary conditions. A 3 time-level, second order accurate leap frog style discretization scheme is used. The following subsections describe the parameter and code files in a bit more detail, though I think the best way to understand how things work is to peruse the source files (which are relatively simple in this case).

#### 1. *Parameter Files*

For conceptual clarity, the input parameters are split into two files, the fixed parameters in

```
.../pamr/examples/wave/wave.fparam
```

and the run-time dependent parameters (the ellipsis ... in the above path name denote the directory where pamr was installed). Three run-time parameter files, for a 1, 2 and 3d sample, are included in

```
.../pamr/examples/wave/run_1d/wave.rtparam
.../pamr/examples/wave/run_2d/wave.rtparam
.../pamr/examples/wave/run_3d/wave.rtparam
```

The fixed parameter file defines parameters that should be the same for all evolutions. This includes the number of time levels, the variable definition (which in this case is the single scalar field variable **phi**), interpolation, injection and synchronization options. The run-time parameter file defines parameters that could change from run to run, such as the number of time steps, scalar field initial data, regridding intervals, etc. AMRD expects a single parameter file, thus to run a simulation a sequence of commands such as the following could be used:

```
> cd .../pamr/examples/wave/run_1d/
> cat ../wave.fparam wave.rtparam > wave.param
> mpirun -np 2 ../wave wave.param
```

#### 2. Source Files

The source is split into two main files plus a header file:

```
.../pamr/examples/wave/wave.c
.../pamr/examples/wave/wave.h
.../pamr/examples/wave/sf_evo.f
```

**wave.c** is the main program, which calls **amrd()** and contains all the required hook functions. Note the usage of the PAMR library functions, in particular in the function **ldptr()**, which is called for all grid based hook functions and queries the library about information pertaining to the current grid. The actual numerics are performed in the fortran file **sf_evo.f**—see the comments in the file for more details.

### B. nbs

**nbs** is located in the directory

```
.../pamr/examples/nbs/
```

This program solves the Schroedinger equation with Newtonian potential in 3D flat space (and can in principle be used to solve for Newtonian boson stars, hence the name; however, neither the initial nor boundary conditions are setup for doing so at this stage).

$$i\frac{\partial\Phi}{\partial t} = -\frac{1}{2}\nabla^2\Phi + V\Phi \tag{3}$$

$$\nabla^2 V = \Phi\bar{\Phi}, \tag{4}$$

where $\Phi$ is a complex field (with $\bar{\Phi}$ its complex conjugate) and $V$ is a real potential. Dirichlet boundary conditions are used for both $\Phi$ and $V$. In the code $\Phi$ is represented by its real and imaginary components $\Phi \equiv \Phi_r + i\Phi_i$.

The above is a coupled elliptic-parabolic system of equations, however in the code the parabolic Schroedinger equation is treated as a hyperbolic equation (solved using a two step Crank-Nicholson like scheme). This imposes a rather restrictive CFL condition on the time step, and requires that the temporal refinement ratio be greater than the spatial one. Thus the code is not very useful for doing physics, however the purpose of this example is to demonstrate how to solve coupled elliptic/hyperbolic systems with AMRD.

#### 1. Parameter Files

As with **wave** (see Sec.V A 1), the input parameters are split into two files, the fixed parameters in

```
.../pamr/examples/nbs/nbs.fparam
```

and the run-time dependent parameters

```
.../pamr/examples/nbs/run/nbs.rtparam
```

**nbs** is run in the same way as **wave**:

```
> cd .../pamr/examples/nbs/run/
> cat ../nbs.fparam nbs.rtparam > nbs.param
> mpirun -np 2 ../nbs nbs.param
```

#### 2. Source Files

The source is split into two main files plus a couple of include files:

```
.../pamr/examples/nbs/nbs.c
.../pamr/examples/nbs/nbs.h
.../pamr/examples/nbs/num.f
.../pamr/examples/nbs/cmask.inc
```

The structure of the code is very similar to that of **wave** (see Sec.V A 2); **nbs.c** is the main program interfacing to the libraries, and **num.f** contains the numerical routines.

---

[1] F. Pretorius, "Numerical Simulations of Gravitational Collapse", *Univ. of British Columbia Ph.D. Thesis* (2002)
[2] R.L. Marsa and M.W. Choptuik, "The RNPL User's Guide", http://laplace.physics.ubc.ca/Members/marsa/rnpl/users_guide/users_guide.html (1995)
[3] F. Pretorius, "PAMR Reference Manual", (2004).
[4] M. J. Berger and P. Colella, "Local Adaptive Mesh Refinement for Shock Hydrodynamics", J. Comp. Phys. **82**, 64 (1989).
[5] G. Calabrese, L. Lehner, O. Reula, O. Sarbach, M. Tiglio "Summation by parts and dissipation for domains with excised regions", gr-qc/0308007