Carpet under the hood

Erik Schnetter <schnetter@uni-tuebingen.de>

May 3, 2003

Abstract

This document describes the internal workings of the Carpet arrangement. Its intended readership are people who extend Carpet, or who use Carpet more thant the average user. This document is supposed to be read in conjuction with and guiding through the source code.

Contents

1 Overview

The Carpet driver, which lives in the Carpet arrangement, is divided into several parts. The thorn `Carpet` is the main driver piece; it provides all the routines and structures that Cactus expects from it. The thorn `CarpetLib` is the workhorse that does all the bookkeeping and data shuffling. Those two alone form a valid Cactus driver; the other thorns provide additional functionality. The thorns `CarpetInterp`, `CarpetReduce`, and `CarpetSlab` provide the corresponding interpolation, reduction, and slabbing interfaces. The thorns `CarpetIOASCII` and `CarpetIOFlexIO` provide I/O methods. Finally, thorn `CarpetRegrid` provides a user interface to select where and what to refine. (The actual refinement is handled in `CarpetLib`.)

2 Terminology

Carpet is called "Carpet" because a carpet consists of many individual patches.

Carpet is a mesh refinement driver. It knows about a hierarchy of *refinement levels*, where each level is decomposed into a set of cuboid *grid patches*. For historic reasons it also has a notion of *multigrid levels*, but those are currently unused. They might conceivably be reactivated to form multigrid stacks to solve elliptic equations. The grid patch is the smallest unit of grid points that Carpet deals with. Carpet parallelises by assigning sets of grid patches to processors.

A multi-patch run is a run where more than one grid patch (of the same refinement level) is assigned to a single processor. This is a situation that can occur even without refinement. This is also a situation that cannot occur with PUGH, so that most thorns cannot handle this situation. In multi-patch runs one has to distinguish between *local mode*, where one has access to a single grid patch, and *global mode*, where one cannot access individual grid patches, but can instead perfom global operations such as synchronisation, interpolation, or reduction. This part of Cactus is currently (2003-04-30) undergoing changes.

Carpet uses vertex-centered refinement. That is, each coarse grid point coincides with a fine grid point. To *regrid* means to select a new set of grid patches for each refinement level. To *recompose* the grid hierarchy means to move data around. Regridding is only about bookkeeping, while recomposing is about data munging.

Each grid patch can be divided in up to four zones: the interior, the outer boundary, and the ghost zone, and the refinement boundary. The interior is where the actual compuations go on. The outer boundary is where the users' outer boundary condition is applied; from Carpet's point of view, these two are the same. (The only difference is that Carpet sets `cctk_bbox` correspondingly.) The ghost zones are boundaries to other grid patches on the same refinement level (that might live on a different processor). The refinement boundary is the boundary of the refined region in a level, and it is filled by prolongation

(interpolation) from the next coarser level. Both the ghost zones and the prolongation boundary are filled by *synchronising*.

Grid patches that are on the same refinement level never overlap except with their ghost zones. Conversly, all ghost zones must overlap with a non-ghost zone of another grid patch of the same level. All refinement boundaries must overlap with a grid patch on the next coarser level. (This is also called *proper nesting*.)

Except for exceptions, Carpet numbers grid point indices and time levels with integers. It counts always in terms of the finest grid, so that coarser grids have *strides* that are powers of the refinement factor. This has the advantage that different refinement levels can use the same global numbering scheme.

The grid patches are described by a *bounding box* (abbreviated bbox, see `CarpetLib/src/bbox.*`). This is a triplet of *vectors* (see `CarpetLib/src/vect.*`), where each triplet specifies *lower bound*, *upper bound*, and *stride*, much as is conventional in Fortran. Triplets are enclosed in round parentheses $(\cdot : \cdot : \cdot)$, and vectors are enclosed in square brackets $[\cdot, \cdot, \cdots]$. A typical grid patch might have a bounding box which is denoted by $([0,0,0] : [20,20,20] : [2,2,2])$. This is to be read as $(\text{lower} : \text{upper} : \text{stride})$, meaning that the grid patch has one corner grid point at $[0,0,0]$, the diagonally opposite corner grid point at $[20,20,20]$, and the grid points are spaced two "fine grid spacings" apart. This grid patch contains $11 \times 11 \times 11$ grid points. Empty bboxes have an upper bound that is strictly lower than the lower bound. The files `CarpetLib/src/vect.*` contains many useful routines to deal with short vectors, and the files `CarpetLib/src/bbox.*` contain routines deal with an algebra of bboxes. The files `CarpetLib/src/bboxset.*` contain routines that handle sets of bboxes.

3 The driver

The driver consists of the two thorns `Carpet` and `CarpetLib`. `Carpet` is the front end to Cactus, while `CarpetLib` is the back end to the machine. `Carpet` specifies the grid shape, decides when to allocate and deallocate storage, cycles through thes schedule bins, and passes all internal information in the `cGH` structure to the thorns.

3.1 Specifying the grid extent

`Carpet` defines the usual parameters necessary to specify the extent of the grid. Everything that has to do with coordinates and symmetries is handled elsewhere, and the driver does not know about that.

The `global_*` parameters specify the global extent of the coarsest grid. Not all of this grid needs to be covered by grid patches. It is conceivable to have an L-shaped simulation domain without any refinement. This situation can be described to Carpet by specifying a global shape that is the convex hull of the domain, and then using two cuboid grid patchs to fill in the shape of the L.

The ghost_* parameters specify the number of ghost zones. The periodic* parameters are unused; they are only there because some thorns look at these parameters. Carpet itself does not supply periodic boundary conditions; they have to be handled by another thorn. The size of the prolongation boundary is the same as the number of ghost zones.

The parameter max_refinement_levels specifies the maximum number of levels that can be present in a run, including the base level. This parameter, together with refinement_factor, define the grid point numbering scheme, which (see above) depends on the finest possible grid. However, none of the finer levels will be activated automatically. The multigrid_* parameters are unused.

The parameter base_extents specifies the shapes of the grid patches that are present on the coarsest grid. This can be used to set up e.g. an L-shaped domain. The parameter base_outerbounds specifies which of the grid patches' boundaries are to be treated as outer boundaries, i.e. for which of those cctk_bbox should be set to 1.

Carpet currently ignores enable_all_storage and always enables all storage. This is because it is not yet clear how individual grid function can be allocated and deallocated while still keeping enough data for the boundary prolongation.

Checksumming and poisoning are means to find thorns that alter grid variables that should not be altered, or that fail to fill in grid variables that they should fill in.

None of the above specifies anything about refined grids. Refined grid are created and destroyed at run time, possibly guided by the thorn CarpetRegrid which provides a nice user interface.

3.2 The timeline

It is Carpet's task to walk through the schedule bins and call all user routines. Only some fairly fundamental initialisation happens in the flesh before Carpet takes control. The overall picture of what happens when is:

1. Startup (see file Carpet/src/CarpetStartup.cc). This is the only scheduled routine; everything else happens by overloading and registering. This routine does nothing but registering and overloading.

2. SetupGH (see file Carpet/src/SetupGH.cc). This routine does the bulk of initialising Carpet. It sets up the internal structures for all grid variables.

3. Initialise (see file Carpet/src/Initialise.cc). This routine walks the initialisation part of the scheduling bins.

4. Evolve (see file Carpet/src/Evolve.cc). This routine walks the evolution part of the scheduling bins. It also contains the main evolution loop.

5. Shutdown (see file `Carpet/src/Shutdown.cc`). This routine walks the shutdown part of the scheduling bins. Normally, nothing interesting happens here.

These stages are explained in the following sections.

3.2.1 Initialisation

(See file `Carpet/src/Initialise.cc`.) In this stage Carpet initialises the simulation. This includes setting up the grids, calling routines to register symmetries and boundary conditions, as well as calculating the actual initial data on several refinement levels.

There are three parameters influencing initial data generation, and it does not make sense to set more than one to "yes":

```
MoL::initial_data_is_crap
Carpet::init_each_timelevel
Carpet::init_3_timelevels
```

That is, you have four methods, and the default (all no) gives you wrong data on the past timelevels and hence wrong data on the interpolated refinement boundaries when you use second order time interpolation. For first order time interpolation, all four methods are identical.

With all three parameters set to "no" Carpet traverses the scheduling bins in the following order:

1. Set `cctk_iteration` to zero
2. Set `cctk_time` to the initial time
3. PARAMCHECK
4. Loop over refinement levels, starting from coarsest:
5. BASEGRID
6. INITIAL
7. POSTINITIAL
8. POSTSTEP
9. Regrid (possibly creating new levels)
10. End loop over refinement levels
11. Restrict from finer to coarser grids
12. Loop over refinement levels, starting from coarsest:
13. RECOVER_VARIABLES
14. CPINITIAL
15. ANALYSIS
16. OutputGH

17. End loop over refinement levels

In the beginning, only the coarsest level exists. The first loop starts by initialising this level. At the end of this loop, more levels are created if desired. This makes it possible to make this decision depend on an automatic refinement criterion.

`MoL::initial_data_is_crap` performs all steps as indicated. After 17, when the evolution starts, `MoL` copies the current timelevels to the past timelevels.

`Carpet::init_each_timelevel` loops the steps 5 to 8 over all timelevels, setting `cctk_time` differently each time.

Finally, `Carpet::init_3_timelevels` performs all steps in order, but evolves each level forward one and backwards two steps, creating two past time levels.

The parameter that specifies the number of refinement levels is not a Carpet parameter, but a CarpetRegrid parameter. CarpetRegrid determines item 9, i.e., whether to create a new, finer level when the coarser levels have been initialised. CarpetRegrid has a host of other parameters, and it can decide item 9 also by a different means, e.g. —in principle— through the local truncation error.

3.2.2 Evolution

(See file `Carpet/src/Evolve.cc`.) In this stage Carpet performs the main time evolution loop. This is further complicated by the fact that finer grids need to take more and smaller time steps than coarser grids. In Carpet's time step counting scheme, which is based on the finest grid time steps, this means that the coarser grids are skipped in the remaining time steps. Thus the elegant recursive scheme is flattened out. The scheduling bins in the main time evolution loop are traversed in the following order:

1. Advance `cctk_iteration`
2. Loop over refinement levels, starting from coarsest:
3. If the current level needs to be treated at this iteration:
4. Calculate current `cctk_time`
5. Cycle time levels
6. PRESTEP
7. EVOL
8. POSTSTEP
9. Regrid
10. End loop over refinement levels
11. Restrict from finer to coarser grids
12. Loop over refinement levels, starting from coarsest:

13. If the current level needs to be treated at this iteration:
14. CHECKPOINT
15. ANALYSIS
16. OutputGH
17. End loop over refinement levels

The condition whether a refinement level needs to be treated at the current iteration is different for the two loops. In the first loop, the coarse grids need to be advanced before the finer grids, so the condition is $\mathit{iter} \bmod \mathit{stride} = 1$. Here *iter* is the current iteration, and *stride* the stride of the current refinement level, i.e. the factor by which the finest grid is finer than the current grid. In the second loop above, the coarse grids need to be treated after the finer grids, so that the condition reads $\mathit{iter} \bmod \mathit{stride} = \mathit{stride}$.

3.3 Calling scheduled routines

(See file `Carpet/src/CallFunction.cc`.) The process by which the scheduling bins are traversed is different from the process which actually calls the routines within the scheduling bins. The former has to do with mesh refinement, making sure that the coarse and fine grids are evolved in the right order. The latter has to do with treating multiple patches, i.e. with local mode and global mode operations, as mentioned above.

In the function `CallFunction`, all the arguments that are passed to the scheduled routines have to be set up. Additionally, the `cGH` structure has to be filled in. Some fields in the `cGH` structure are always kept up-to-date during the refinement level loops, such as the time step size and the grid spacing. The file `Carpet/src/helper.cc` contains helper routines that allow easy looping over refinement levels and over grid patches. (Grid patches are also called *compoments* in Carpet. The expression component seems to be confusing, so that I switched to using *patch* instead. Some source code still reflects the old conventsion.)

The function `CallFunction` first distinguishes between global mode functions and local mode functions.

Global mode functions are called once (on each processor). They are passed all the global data, such as `cctk_gsh` and `cctk_delta_space`, but none of the local data, such as `cctk_lsh` or `cctk_bbox`. Grid functions are not accessible, and they are passed as null pointers. However, grid scalars and grid arrays are accessible. There is an untested gateway to directly call local mode functions from global mode functions.

Local mode functions might be called several times (on each processor), once for each grid patch that is assigned to this processor. They receive the global data as well as data for a single grid patch. It is illegal to perform global operations, such as synchronisation, interpolation, or reduction, in local mode.

The distinction between global and local mode is only important for multi-patch runs. For single-patch runs, the distinction does not exist.

Multi-patch runs are only necessary when there are more grid patches on a refinement level than there are processors. This is normally not the case for fixed mesh refinement. Things are different for adaptive mesh refinement, which can create many refined regions.

3.4 Grid arrays and grid scalars

Grid scalars are implemented as zero-dimensional grid arrays with `DISTRIB=CONSTANT`.

Grid arrays are implemented as grid functions, where each grid array group has their own refinement hierarchy that consists of a single level only and is never changed at run time. Grid arrays with less than 3 dimension are extended to have an extent of 1 (and no ghost zones) in the remaining dimensions, so that all quantities in Carpet have 3 dimensions[1]. `DISTRIB=CONSTANT` grid arrays are implemented by internally enlarging the grid array in the z direction, and then distributing this array onto the processors.

3.5 Flesh interfaces

The flesh has many interfaces that need to be filled in by a driver. These are in particular all the routines that are overloaded in the SetupGH stage. Those overloaded routines as well as other helper function are implemented in the following files:

`Carpet/src/Checksum.cc` catching illegal changes to grid variables

`Carpet/src/Comm.cc` synchronisation, prolongation

`Carpet/src/Cycle.cc` time level handling

`Carpet/src/Poison.cc` catching uninitialised grid variables

`Carpet/src/Restrict.cc` restriction from finer to coarser grids

`Carpet/src/Storage.cc` enabling and disabling storage

`Carpet/src/helpers.cc` small low-level helper routines

`Carpet/src/variables.cc` the global variables that keep Carpet's current state (this is used instead of a GH extension — should probably be changed some time)

Most of these files are fairly self-contained, and they mostly marshal the actual work to `CarpetLib`.

[1] This is set by a compile-time constant and could be changed to allow for grid functions and arrays with more than 3 dimensions.

3.6 Interfaces to other thorns

Some other thorns, mostly from the Carpet arrangement, do need to access internal data of Carpet. Carpet keeps its internal state in global variables which are declared in `Carpet/src/carpet_public.hh` and defined in `Carpet/src/variables.cc`. Entities that can be accessed from C are declared in `Carpet/src/carpet_public.h`; some of these would be quite useful if they were provided by the flesh.

3.7 Missing parts

`Carpet` does not handle staggered grids. `Carpet` does not provide cell-centered refinement. `Carpet` always enables all storage. `Carpet` does not run efficiently in parallel.

4 The workhorse

While `Carpet` provides the necessary interfaces to the flesh, the grunt work is actually done by `CarpetLib`. This thorn grew from an earlier mesh refinement of mine (Erik Schnetter) library that was independent of Cactus. It has in the mean time been thoroughly changed, and it does not make sense any more to use it independent of Cactus. `CarpetLib` contains of three major parts: a set of generic useful helpers, the grid hierarchy and data handling, and interpolation operators. Especially the latter could probably be separated out. While `CarpetLib` is written in C++, the interpolators are written in FORTRAN77.

4.1 The helpers

The helpers correspond closely to Carpet's terminology. A class `vect<T,D>` provides small `D`-dimensional vectors of the type `T`, with all the operators that one has learned to enjoy from Haskell and Fortran 90. A `vect` corresponds to a grid point location. The class `bbox<T,D>` provides `D`-dimensional bounding boxes using type `T` as indices. A `bbox` defines the location and shape of a grid patch. Finally, `bboxset<T,D>` is a collection of `bboxes`. `bboxsets` are a useful extension of the algebra of bboxes, as it closes the `bbox` algebra under the union operation.

The files `CarpetLib/src/defs.*` defines useful small helpers and instantiates the STL templates. `CarpetLib/src/dist.*` provides some routines around MPI. Carpet is closely coupled to MPI and does not work without it.

(Instead of inserting switches into Carpet to make it work without MPI, it would make more sense to use a dummy version of MPI. PETSc does contain such a dummy version. It is also easily possible to use a free MPI version such as MPICH and use that to run on a single processor. However, I cannot see any real need for making Carpet work without MPI.)

4.2 The grid hierarchy

The grid hierarchy is described by a set of classes. Except for the actual data, all structures and all information is replicated on all processors.

`gh` is a grid hierarchy. It describes, for each refinement level, the location of the grid patches. This `gh` does not contain ghost zones or prolongation boundaries. There exists only one common `gh` for all grid functions.

`dh` is a data hierarchy. It extends the notion of a `gh` by ghost zones and prolongation boundaries. The `dh` does most of the bookkeeping work, deciding which grid patches interact with what other grid patches through synchronisation, prolongation, restriction, and boundary prolongation. Unexpected situations are often caught in one of `dh`'s many self checks. As all grid functions have the same number of ghost zones, there exists also only one `dh` for all grid functions.

`th` is a time hierarchy. It extends the notion of a `gh` by multiple time levels. There exists one `th` per grid function group. This is a small class that keeps track of the current time on the different refinement levels. (Note that different refinement levels usually live at different times.)

`gf` is a grid function of a certain variable type. There is one instance of `gf` for every grid function, whether it has storage or not. Each `gf` is associated with a `dh` and a `th` and holds the storage for all levels and all patches. It provides interfaces to access and modify these data, either directly or through interpolation operators. `gf` also handles the data movement during a regridding operation.

`ggf` is an abstract superclass of `gf` which is independent of the variable type. This is necessary in C++ to prevent egregious code duplication due to class templates. Most of the routines in `gf` are actually declared in `ggf`, and they either are virtual functions themselves, or they call virtual functions that are declared in `gf`.

`data` is a container for a grid patch of a certain variable type. This is a glorified multi-dimensional array that knows how to move between processors. `data` is not only used to store the grid patches that make up a `gf`, it is also used to move parts of patches around, e.g. for synchronisation or prolongation.

`gdata` is an abstract superclass of `data` for much the same reasons as for `ggf`. All information that is independent of the variable type is kept in `gdata`.

4.3 The interpolators

There are three kinds of "interpolators": for prolongation, for restricting, and for copying. The latter is only a glorified hyperslabber that moves parts of grid patches between grid patches.

The interpolators used for restriction and prolongation are different from those used for the generic interpolation interface in Cactus. The reason is that interpolation is expensive, and hence the interpolation operators used for restriction and prolongation have to be streamlined and optimised. As one knows the location of the sampling points for the interpolation, one can calculate the coefficients in advance, saving much time compared to calling a generic interpolation interface.

4.3.1 Restriction

Restriction operators move data from finer to coarser grids. They are typically called after both the coarse and the fine grid have been advanced to the same time, and they overwrite parts of the coarse grid with information from the fine grid, coupling the coarse grid evolution to the fine grid evolution. In principle, there could be restriction operators with different orders of accuracy. Currently only a single restriction operator is implemented that uses sampling.

The interface of the restriction operator (see file `CarpetLib/src/restrict_3d_real8.F77`) is

```
subroutine restrict_3d_real8
      (src, srciext, srcjext, srckext,
       dst, dstiext, dstjext, dstkext,
       srcbbox, dstbbox, regbbox)

   integer    srciext, srcjext, srckext
   CCTK_REAL8 src(srciext,srcjext,srckext)
   integer    dstiext, dstjext, dstkext
   CCTK_REAL8 dst(dstiext,dstjext,dstkext)
   integer    srcbbox(3,3), dstbbox(3,3), regbbox(3,3)
```

This interpolator assumes that space has three dimensions. The arrays `src` and `dst` contain the source (fine) and destination (coarse) grid patches, stored in Fortran order, as is customary in Cactus. The arrays `src` and `dst` have the shapes `(srciext,srcjext,srckext)` and `(dstiext,dstjext,dstkext)`, respectively — this corresponds to the `cctk_lsh` field in the `cGH` structure.

The three bboxes describe the location and shape of the two arrays and of the region that should be prolongated in the global grid point index system. That is, while the two arrays `src` and `dst` are stored as dense arrays, they correspond to grid patches which in general have non-unit strides in the global index system. As prolongation is an operation that is performed between overlapping grids, the prolongation region is the same for both the coarse and the fine grids.

A few constraints must hold for these data. For example, the shapes of the arrays must be the same as the shapes defined by the bounding boxes; the strides in the bounding boxes must differ by the refinement factor; the bounding boxes must overlap, and the region's bounding box must be contained in

the arrays bounding boxes, etc. Checking these constraints makes up about three quarters of the restriction routine.

The bboxes themselves are here represented as Fortran arrays. Their meaning is

`bbox(:,1)` lower boundary (inclusive)

`bbox(:,2)` upper boundary (inclusive)

`bbox(:,3)` stride

4.3.2 Prolongation

There are many prolongation operators implemented. They differ in the order of their interpolation in space (first and third, or linear and cubic interpolation) and in time (first and second, or linear and quadratic). The higher the order of interpolation, the larger is the stencil, i.e. the more ghost zones and time levels are necessary, and the more expensive the operation becomes.

The prolongation operators live in the files `CarpetLib/src/prolongate_3d_real8*.F77`, and the file names indicate their orders: *n*`tl` stands for n time levels, and `o`*n* stands for an order n interpolation in space (which uses a stencil that is $n+1$ grid points wide).

Apart from taking more than one `src` array argument when using more than one time level, the interface to the prolongation operator is equivalent to that of the restriction operator described above.

5 Regridding, how and where and when

The thorn `Carpet` provides a routine `RegisterRegridRoutine` where one can register a regridding routine. Such a regridding routine does not have to actually regrid anything, it only has to return the new desired grid hierarchy, i.e. basically a description of a `gh`.

Thorn `CarpetRegrid` provides a user interface to the regridding routines in Carpet. All it does is take a regridding specification from the user and translate that into a `gh`. As usual, the parts where the computer has to listen to what a human being intends are the most complicated.

As humans are usually more adept at getting used to computers than the other way around, it is useful and probably necessary to get acquainted with how Carpet thinks in order to make it do what is intended.

Carpet does not deal with real-valued coordinates. Carpet deals with integer grid point locations only, and it counts grid points in terms of the finest possible grid (not the finest currently existing grid). The finest possible grid is defined by the maximum number of refinement levels set in `Carpet`. Changing this parameter will change the meaning of many other values in parameter files, such as e.g. iteration numbers (termination, output). The only parameter that is specified in terms of the coarsest grid is the shape of the coarsest grid in the `global_*` parameters of `Carpet`. I therefore suggest to set

max_refinement_levels to some large number (e.g. 10), and then not changing it while experimenting with other parameter settings.

Carpet also does not know about symmetries. When specifying the location of a fine grid in terms of grid points, it is the responsibility of the user to place the fine grid correctly. For that one has to take ghost zones and symmetry zones into account.

It is also possible to specify the fine grid locations in terms of real-valued coordinates. In this case, CarpetRegrid translates these into integer grid points. A good translation is quite complicated, because it has to take many user expectations into account, such as the location of the origin, staggering with respect to the origin, symmetry boundary conditions, the number of ghost zones etc. The current translation is naive and leads to unexpected results in many cases. A routine that does most of the time what the user expects while being easy to understand is probably important for the ease of use of Carpet, but it might be some time until it is written.

CarpetRegrid contains also a routine that measures the error, as provided in a grid function, and the automatically decides where to refine. This is called AMR (adaptive mesh refinement) if it works efficiently.

Much of CarpetRegrid is just slabbed together in an attempt to find out what people need and expect. The thorn is a mess, and a complete rewrite might be a good idea, once one knows what exactly the rewritten thorn should do.

6 Random ramblings

Carpet uses the STL, because the STL provides very useful container classes such as vectors, sets, and lists. Writing these abstract datatypes oneself does not make sense in these times. It makes much more sense to politick computer administrators to upgrade their software.

The STL and CarpetLib's classes need to be instantiated explicitly. Several compilers have several "automatic" schemes that handle all template issues "just fine". Except they don't. One wants to select the following: No automatic inclusion of .cc files, no automatic template instantiation at link time. Instead, most templates are instantiated explicitly by CarpetLib. It is also necessary to specify to instantiate used templates automatically. The explicit instantiations of CarpetLib's classes live in the .cc files corresponding to the .hh file that define the templates. The STL templates are instantiated in the file CarpetLib/src/defs.cc.

Carpet makes extensive use of the assert() macro in C. This is a quick and easy way to ensure that a certain condition holds. Assert statements abort the code if the condition does not hold. Although I try to provide useful error messages to the user, many unexpected cases are only caught deep inside Carpet and manifest themselves as assertion failures. If you report an assertion failure, it is vitally important to remember theaccompanying file name and line

number. It would also be useful to extract from the core file a stack backtrace and the values of the local variables of the current stack frame.

Using symmetry boundary conditions such as octant mode is currently still awkward in Carpet. There are several reasons for this: `CarpetRegrid` does not know about symmetries, and hence doesn't take them into account when choosing refinement regions. The symmetry conditions on the finer grid might be different from the conditions on the coarser grids, and the symmetry thorns cannot cope with this, so this situation must be avoided: one cannot use `avoid_origin=yes`, because the finer grids all have `avoid_origin=no` due to the vertex-centred refinement.