The CactusDoc documentation pseudo arrangement

Erik Schnetter <schnetter@aei.mpg.de>

Date: 2006/06/06 14:02:34

Abstract

This arrangement CactusDoc contains documentation of concepts than span multiple thorn, yet are not part of the flesh.

1 Tags Tables: Optional Attributes for Cactus Grid Variables

[This section last updated on Date: 2006/06/06 14:23:28 .]

Cactus defines a "tags table" mechanism where a Cactus key-value table is associated with each Cactus group of grid variables. By default the flesh sets up an empty table for each group. If a group is declared with a `tags="..."` clause in its `interface.ccl`, the flesh uses this optional string to initialize the key-value tags table for this group. Keys in a tags table are conventionally taken to have case-insensitive semantics. The value string (which must be deliminated by a single or double quotes in `interface.ccl`)) is interpreted by the function `Util_TableSetFromString()` (see the Cactus Reference Manual for a description of this function).

Currently the contents of a tags table are not evaluated by the CST parser and not used by the flesh; it's entirely up to thorns to agree among themselves as to what should be in the tags table and how it should be interpreted.

You can get the tags table's table handle for a specified group by calling the flesh function `CCTK_GroupTagsTable()` or `CCTK_GroupTagsTableI()`. Once you have this table handle, you can access the tags table itself using the standard key-value table API as documented in the Cactus Reference Manual.

Most thorns which use tags-table information only look at the tags table once (when they start up), and won't notice if the information is changed later on. Therefore, modifying the tags table is probably not a good idea (it's likely to cause confusion if some thorns change their behavior based on the new tags-table entries, while other thorns don't change).

The following sections document some common conventions for tags-table entries.

1.1 Tensor Types

The following tags-table entries describe the tensor types of grid functions/arrays. These are widely used, both by **CactusEinstein** and by other arrangements/thorns:

`tensortypealias`
: This value associated with this tags-table key gives the primary tensor type (behavior under coordinate transformation), and may be one of the following strings:

  `"scalar"` for a 3-scalar or group of 3-scalars; this is generally assumed if no `tensortypealias` tag is present.

  `"u"` for a group of 3 grid functions/arrays storing a contravariant 3-vector T^i.

  `"d"` for a group of 3 grid functions/arrays storing a covariant 3-vector T_i.

  `"uu_sym"` for a group of 6 grid functions/arrays storing a contravariant rank 2 symmetric 3-tensor S^{ij}.

  `"dd_sym"` for a group of 6 grid functions/arrays storing a covariant rank 2 symmetric 3-tensor S_{ij}.

  `"4scalar"` for a 4-scalar or group of 4-scalars.

  `"4u"` for a group of 4 grid functions/arrays storing a contravariant 4-vector T^a.

  `"4d"` for a group of 4 grid functions/arrays storing a covariant 4-vector T_a.

  `"4uu_sym"` for a group of 10 grid functions/arrays storing a contravariant rank 2 symmetric 4-tensor S^{ab}.

  `"4dd_sym"` for a group of 10 grid functions/arrays storing a covariant rank 2 symmetric 4-tensor S_{ab}.

  Note that for multi-index tensors, different thorns have different conventions about the order of the grid functions. Row-major order by indices is most common (for example, indices 11, 12, 13, 22, 23, 33 for a rank 2 symmetric 3-tensor), but some thorns may use column-major order instead (in this example, indices 11, 21, 31, 22, 23, 33). This also illustrates that for multi-index symmetric tensors, there are multiple conventions about which subset of algebraically-independent components is actually stored.

`tensormetric`
: The string value associated with this tags-table key gives the full name (Thorn::Gridfn) of the grid variable holding the tensor 3-metric with respect to which the other tensor-transformation properties are defined.

`tensorweight`
: For a tensor density, this gives the weight. If omitted, most code will use a default weight of 0.

`tensorparity`
: For a (pseudo)tensor, this gives the parity. The parity can be either $+1$ or -1. The parity defines whether the quantity has a different sign change behaviour under odd parity transformations, such as e.g. reflections. E.g., Pseudoscalars and axial vectors have negative parity. If omitted, most code will use a default parity of $+1$.

`tensorspecial`
: This is an "escape-hatch" tag used for things which are "sort of tensors, but not really". If present, it may have one of the following string values describing variables in the AEI BSSN formalism (gr-qc/0003071):

  `"phi"` for the BSSN logarithmic-conformal-factor $\phi \equiv \frac{1}{12} \log \det[g_{ij}]$.

  `"gamma"` for the BSSN contracted-conformal-Christoffel-symbols $\tilde{\Gamma}^i \equiv \tilde{g}^{mn} \tilde{\Gamma}^i{}_{mn} \equiv -\partial_j \tilde{g}^{ij}$.

Some thorns, particularly those for multi-patch computations, may also use the following tags-table entry:

`tensorbasis`
: The string value associated with this tags-table key specifies the type of tensor basis being used:

  `"global xyz"` means that the tensor basis is a global Cartesian one, the same for all patches (and hence that this group should not be tensor-transformed when it's interpolated from one patch to another)

  `"local"` means that the tensor basis is a local per-patch one (which varies from one patch to another); usually this will be the obvious local-Cactus-coordinates coordinate basis

  Other values may also be allowed for this key; see the documentation for individual multi-patch infrastructures for further details (including a discussion of what defaults are assumed for this).

1.1.1 Example

For example, the BSSN 3-metric might be declared with an `interface.ccl` entry like this:

```
real conformal_metric                              type=GF timelevels=3    \
     tags='tensortypealias="dd_sym"                                        \
           tensorweight=-0.66666666666666667                               \
           tensormetric="BSSNBase::conformal_metric"'
{
g_tilde_dd_11
g_tilde_dd_12
g_tilde_dd_13
g_tilde_dd_22
g_tilde_dd_23
g_tilde_dd_33
} "conformal 3-metric $\tilde{g}_{ij}$"
```

Notice the syntax here: the `tags=` string is enclosed in single quotes (`tags='...'`), and may contain double-quoted strings.

1.2 Checkpointing of Cactus Grid Variables

During checkpointing all variables which have global storage assigned are saved in a checkpoint file for later recovery. For some variables this isn't really necessary because they are set up at `BASEGRID` and remain constant thereafter, or they are used only as temporaries which don't need to be initialized from a checkpoint during recovery.

`checkpoint`
: This boolean key-value tag is meant as a hint to checkpoint methods as to whether a Cactus grid variable group needs to be checkpointed.

  `"yes"`
  : The group needs to be saved in a checkpoint. This is the default if no such tag is specified in a group's tags table.

  `"no"`
  : The group may be omitted from a checkpoint. For a recovery Cactus run, user code in application thorns makes sure that variables in this group are properly initialized.

1.3 Carpet-specific Tags

This section lists tags table entries which are specific to the mesh-refinement driver thorn `Carpet`. For a detailed description of these tags please refer to the Carpet documentation.

`Prolongation`
: This string key-value tag speficifies which method Carpet should use to prolongate variables in this group. Possible values are:

  `"none"`
  : do not prolongate this group

  `"Copy"`
  : use simple copying for prolongation (needs only one time level)

  `"Lagrange"`
  : use Lagrange interpolation (this is the default if no prolongation operator has been specified for this group)

  `"TVD"`
  : use TVD stencils (for hydro)

  `"ENO"`
  : use ENO stencils (for hydro)

  `"WENO"`
  : use WENO stencils (for hydro)

`ProlongationParameter = <parameter_name>`
: Rather than naming a fixed prolongation operator at compile time via the `Prolongation` tag in an interface.ccl file, it is also possible to specify it for each group only at runtime: the `ProlongationParameter` tag key expects a string value denoting the full name of a Cactus **STRING** or **KEYWORD** parameter which specifies the actual prolongation operator (from the same set listed above).
  The `ProlongationParameter` and `Prolongation` tags are mutually exclusive, one can either use one or the other in an inteface.ccl file.
  Currently Carpet evaluates the prolongation parameter information only once at startup, it is assumed that prolongation operators for individual groups do not change afterwards.

`InterpNumTimelevels = <levels>`
: This tag specifies how many timelevels should be used for time interpolation.

2 Post-processing data from the CactusEinstein infrastructure

I (Erik Schnetter <schnetter@cct.lsu.edu>) was asked how to run thorn IsolatedHorizon on data that are stored in a file. Here is what I answered. I thought I should conserve it for posteriority.

2.1 Step 1

You do the following. I assume that you have the 3-metric and the extrinsic curvature in HDF5 files. You set up a parameter file for a grid structure that contains the region around the horizon. The refinement level structure and grid spacing etc. needs to be the same as in the HDF5 files, but the grids can be

much smaller. You can also leave out some finer grids, i.e., reduce the number of levels. However, the coarse grid spacing must remain the same. The symmetries must also be the same.

You then use the file reader and thorn AEIThorns/IDFileADM to read in the ADM variables from the files. The parameter file does not need to activate BSSN_MoL or any time evolution mechanism. IDFileADM acts as provider for initial data, so you don't need any other initial data either.

You set up your parameter file so that the AH finder runs, stores the horizon shape in SphericalSurface, and IsolatedHorizon accesses these data.

This gives you the time-independent variables on the horizon, i.e., mostly the spin. It also allows you to look for apparent horizons if you don't know where they are.

2.2 Step 2

If you also want time-dependent data on the horizon, e.g. its 3-determinant, then you also have to perform some time steps. You can either read in lapse and shift from files, or you can set them arbitrarily (e.g. lapse one, shift zero). You also need to activate a time evolution thorn, i.e., BSSN_MoL, MoL, Time, etc. In order to fill the past time levels, just choose MoL::initial_data_is_crap. If you have hydrodynamics, you will also need to read in the hydro variables.

You only need to perform two time steps. Remember that the output of IsolatedHorizon for iteration 0 and 1 are incorrect or very inaccurate, since the past time levels are not correct, and hence the time derivatives that IsolatedHorizon calculates are wrong. However, iteration 2 should be good. (You can also perform 5 iterations and cross-check.)

2.3 Step 3

If you do not have the extrinsic curvature, but have the 3-metric, lapse, and shift for consecutive time steps (that is, if you have data suitable for finding event horizons), then you need to reconstruct the extrinsic curvature first. There is a thorn AEIThorns/CalcK that helps you. It reads the data for the 4-metric timestep after timestep, calculates the time derivative of the 3-metric through finite differencing in time, and then determines the extrinsic curvature from that, and writes it to a file. Once you have it, you can start as in step 1. CalcK has a small shell script that tells you what to do.

2.4 Additional remarks

In general, things become more interesting if a static conformal factor is involved (since you now have more variables), especially if you output the static conformal factor only once (since it is static), which means that you have to mix variables from different time steps.

The thorns that I mentioned above have some examples. In general, this is NOT a "just do it" action; you have to know what you are doing, since you have to put the pieces together in your parameter file and make sure that everything is consistent. We may have a vision that you just call a script in a directory that contains output files and the script figures out everything else, but we're not there yet. All the ingredients are there, but you'll have to put them together in the right way. Think Lego.

You could start by reading the documentation of the file reader in CactusBase/IOUtil (to find out how it

locates and reads files) and the CactusEinstein initial data mechanism (since IDFileADM provides initial data). After that, step 1 is straightforward.

Step 2 is also easy if you know how to evolve data in time.

Step 3 is conceptually simple, but technically complicated because it involves moving data around in HDF5 files. If you need to do that, I can explain it in more detail.