CartGrid3D

Gabrielle Allen
Gerd Lanfermann
Joan Masso
Jonathan Thornburg

Date: 2003/08/22 20:56:08

Abstract

`CartGrid3D` allows you to set up coordinates on a 3D Cartesian grid in a flexible manner. You can choose different grid domains (*eg* octant) to allow you to exploit any symmetry in your problem. `CartGrid3D` also provides routines for registering symmetries of grid functions and applying symmetry conditions across the coordinate axes.

1 Specifying the Grid Symmetry

You specify the grid symmetry (or lack thereof) with the `grid::domain` parameter:

`grid::domain = "full"`
: There are no symmetries.

`grid::domain = "bitant"`
: The grid includes only the $z \geq 0$ half-space (plus symmetry zones); there is a reflection symmetry across the $z = 0$ plane.

`grid::domain = "quadrant"`
: The grid includes only the $\{x \geq 0, y \geq 0\}$ quadrant (plus symmetry zones); there is a reflection symmetry across both the $x = 0$ plane and the $y = 0$ plane.

`grid::domain = "octant"`
: The grid includes only the $\{x \geq 0, y \geq 0, z \geq 0\}$ octant (plus symmetry zones); there is a reflection symmetry across each of the $x = 0$ plane, the $y = 0$ plane, and the $z = 0$ plane.

In each case except `grid::domain = "full"`, symmetry zones are introduced just on the "other side" of each symmetry grid boundary. Each symmetry zone has a width (perpendicular to the boundary) of `driver::ghost_size` extra grid points. For centered 2nd order finite differencing, a width of `driver::ghost_size = 1` should be sufficient, but for (centered) 4th order finite differencing, or for upwinded 2nd order, a width of `driver::ghost_size = 2` is needed. Making `driver::ghost_size` too large is fairly harmless (it just slightly reduces performance), but making it too small will almost certainly result in horribly wrong finite differencing near the symmetry boundaries, and may also result in core dumps from out-of-range array accessing.

Note that the symmetry zones must be explicitly included in `driver::global_nx`, `driver::global_ny`, and `driver::global_nz`, but should *not* be included in any of the `grid::type = "byrange"` parameters `grid::xmin`, `grid::xmax`, `grid::ymin`, `grid::ymax`, `grid::zmin`, `grid::zmax`, `grid::xyzmin`, and/or `grid::xyzmax` described in the next section.

Note also that `driver::global_nx`, `driver::global_ny`, and `driver::global_nz` do *not* include any ghost zones introduced for multiprocessor synchronization. (For more information on ghost zones, see the section "Ghost Size" in the "Cactus Variables" chapter of the Cactus Users' Guide.)

2 Specifying the Grid Size, Range, and Spacing

`CartGrid3D` provides several different methods for setting up the integer *grid size* (*eg* 128), floating-point *grid spacing* (*eg* 0.1), and floating-point *grid range* (*eg* 12.8).[1] You specify which method to use, with the `grid::type` parameter:

`grid::type = "byrange"`
: You specify the x, y, and z grid ranges, either with separate `grid::xmin`, `grid::xmax`, `grid::ymin`, `grid::ymax`, `grid::zmin`, and `grid::zmax` parameters, or with the `grid::xyzmin` and `grid::xyzmax` parameters. The grid spacings are then determined automagically from this information and the `driver::global_nx`, `driver::global_ny`, and `driver::global_nz` grid-size parameters. You should also choose the `grid::domain` parameter consistent with all these other parameters. (It's not clear whether or not the code ever explicitly checks this.)

`grid::type = "box"`
: This is a special case of `grid::type = "byrange"` with the grid ranges hard-wired to `grid::xyzmin = -0.5` and `grid::xyzmax = +0.5`.

`grid::type = "byspacing"`
: You specify the x, y, and z grid spacings, either with separate `grid::dx`, `grid::dy`, and `grid::dz` parameters, or with the `grid::dxyz` parameter. You also specify the grid symmetry with the `grid::domain` parameter. The x, y, and z grid ranges are then determined automagically from this information and the `driver::global_nx`, `driver::global_ny`, and `driver::global_nz` grid-size parameters: Each coordinate's range is chosen to be either symmetric about zero, or to extend from 0 up to a maximum value.

There are also a number of optional parameters which can be used to specify whether or not it's ok to have a grid point with an x, y, and/or z coordinate exactly equal to 0:

`grid::no_originx`, `grid::no_originy`, `grid::no_originz`, `grid::no_origin`
: These parameters are all deprecated — don't use them!

`grid::avoid_originx`
: This is a Boolean parameter; if set to true (`grid::avoid_originx = "true"` or `grid::avoid_originx = "yes"` or `grid::avoid_originx = 1`) then the grid will be "half-centered" across $x = 0$, *ie* there will be grid points at $\ldots$, $x = -\frac{3}{2}\Delta x$, $x = -\frac{1}{2}\Delta x$, $x = +\frac{1}{2}\Delta x$, $x = +\frac{3}{2}\Delta x$, $\ldots$, but not at $x = 0$.

`grid::avoid_originy`
: Same thing for y.

`grid::avoid_originz`
: Same thing for z.

`grid::avoid_origin`
: Same thing for all 3 axes x and y and z, *ie* no grid point will have $x = 0$ or $y = 0$ or $z = 0$.

3 An Example

Here is an example of setting up a grid using the `PUGH` unigrid driver. The relevant parts of the parameter file are as follows:

```
# PUGH
driver::ghost_size = 2
driver::global_nx = 61
driver::global_ny = 61
driver::global_nz = 33
```

[1] If you're AMR-ing, this all refers to the coarsest or base grid.

```
# CartGrid3D
grid::avoid_origin = "no"
grid::domain = "bitant"
grid::type = "byrange"
grid::xmin = -3.0
grid::xmax = +3.0
grid::ymin = -3.0
grid::ymax = +3.0
grid::zmin =  0.0
grid::zmax = +3.0
```

The resulting Cactus output (describing the grid) is as follows:

```
INFO (CartGrid3D): Grid Spacings:
INFO (CartGrid3D):  dx=>1.0000000e-01  dy=>1.0000000e-01  dz=>1.0000000e-01
INFO (CartGrid3D): Computational Coordinates:
INFO (CartGrid3D):  x=>[-3.000, 3.000]  y=>[-3.000, 3.000]  z=>[-0.200, 3.000]
INFO (CartGrid3D): Indices of Physical Coordinates:
INFO (CartGrid3D):  x=>[0,60]  y=>[0,60]  z=>[2,32]
INFO (PUGH): Single processor evolution
INFO (PUGH): 3-dimensional grid functions
INFO (PUGH):   Size: 61 61 33
```

Since there's no symmetry in the x and y directions, the grid is set up just as specified, with floating-point coordinates running from -3.0 to 3.0 inclusive, and 61 grid points with integer grid indices $[0, 60]$ (C) or $[1, 61]$ (Fortran).

However, in the z direction there's a reflection symmetry across the $z = 0$ plane, so the specified range of the grid, $z \in [0.0, 3.0]$, is automagically widened to include the symmetry zone of `driver::ghost_size = 2` grid points. The grid thus actually includes the range of floating-point coordinates $z \in [-0.2, 3.0]$. The original specification of 33 grid points is left alone, however, so the grid points have integer array indices $[0, 32]$ (C) or $[1, 33]$ (Fortran). The "physical" (*ie* non-symmetry-zone) part of the grid is precisely the originally-specified range, $z \in [0.0, 3.0]$, and has the integer array indices $[2, 32]$ (C) or $[3, 33]$ (Fortran).

4 Coordinates

`CartGrid3D` defines (registers) four coordinate systems: `cart3d`, `cart2d`, `cart1d`, and `spher3d`.

The Cartesian coordinates supplied by this thorn are grid functions with the standard names `x`, `y`, and `z`. To use these coordinates you need to inherit from `grid`, *ie* you need to have an

```
inherits: grid
```

line in your `interface.ccl` file. In addition a grid function `r` is provided, containing the radial coordinate from the origin where

$$r = \sqrt{x^2 + y^2 + z^2}$$

`CartGrid3D` registers the lower and upper range of each coordinate with the flesh.

5 Symmetries for Grid Functions

If your problem and initial data allow it, `CartGrid3D` allows you to enforce even or odd parity for any grid function at (across) each coordinate axis. For a function $\phi(x, y, z)$, even parity symmetry on the x-axis means

$$\phi(-x, y, z) = \phi(x, y, z)$$

while odd parity symmetry means

$$\phi(-x, y, z) = -\phi(x, y, z)$$

Note that the symmetries will only be enforced if a symmetry domain is chosen (that is, if `grid::domain` is set to something other than `grid::domain = "full"`.

5.1 Registering Symmetry Behaviour

Each grid function can register how it behaves under a coordinate change using function calls in `CartGrid3D`. These symmetry properties can then be used by other thorns, for example `CactusBase/Boundary` uses them to enforce symmetry boundary conditions across coordinate axes. Symmetries should obviously be registered before they are used, but since they can be different for different grids, they must be registered *after* the `CCTK_STARTUP` timebin. The usual place to register symmetries is in the `CCTK_BASEGRID` timebin.

For example, to register the symmetries of the *xy* component of the metric tensor from C, you first need to get access to the include file by putting the line

```
uses include: Symmetry.h
```

in your `interface.ccl` file. Then in your thorn you can write (C)

```
#include "Symmetry.h"
static int one=1;
int sym[3];
sym[0] = -one;
sym[1] = -one;
sym[2] =  one;
SetCartSymVN(cctkGH, sym,"ADMBase::gxy");
```

5.2 Calling Symmetry Boundary Conditions

`CartGrid3D` provides the following two routines to apply symmetry boundary conditions to a variable group:

```
CartSymGI(cGH *GH, int *gi)
CartSymGN(cGH *GH, const char *gn)
```

and for a specific variable it provides:

```
CartSymVI(cGH *GH, int *vi)
CartSymVN(cGH *GH, const char *gn)
```

A group or variable can be specified by its index value or name (use the 'I' or 'N' version respectively). The Fortran versions of these functions take an additional first argument, which is an integer which will hold the return value.