Cactus Documentation

Release 0.0.1

Yu Liu

Jun 12, 2020

Contents:

Einstein Toolkit is composed of several different modules and is developed by researchers from different institutions throughout the world. Documentation for several modules is very brief.

I decided to write this documentation. One goal is to make Cactus easy for beginners to understand and modify, with minimal effort.

This documentation provided a useful guide for the parameter file and thorn writing.

Note: Documentation is unofficial and not yet finished.

CHAPTER 1

Get Start

1.1 Introduction

The largest user base for Cactus is in the field of numerical relativity where, for example, over 100 components are now shared among over fifteen different groups through the Einstein Toolkit.

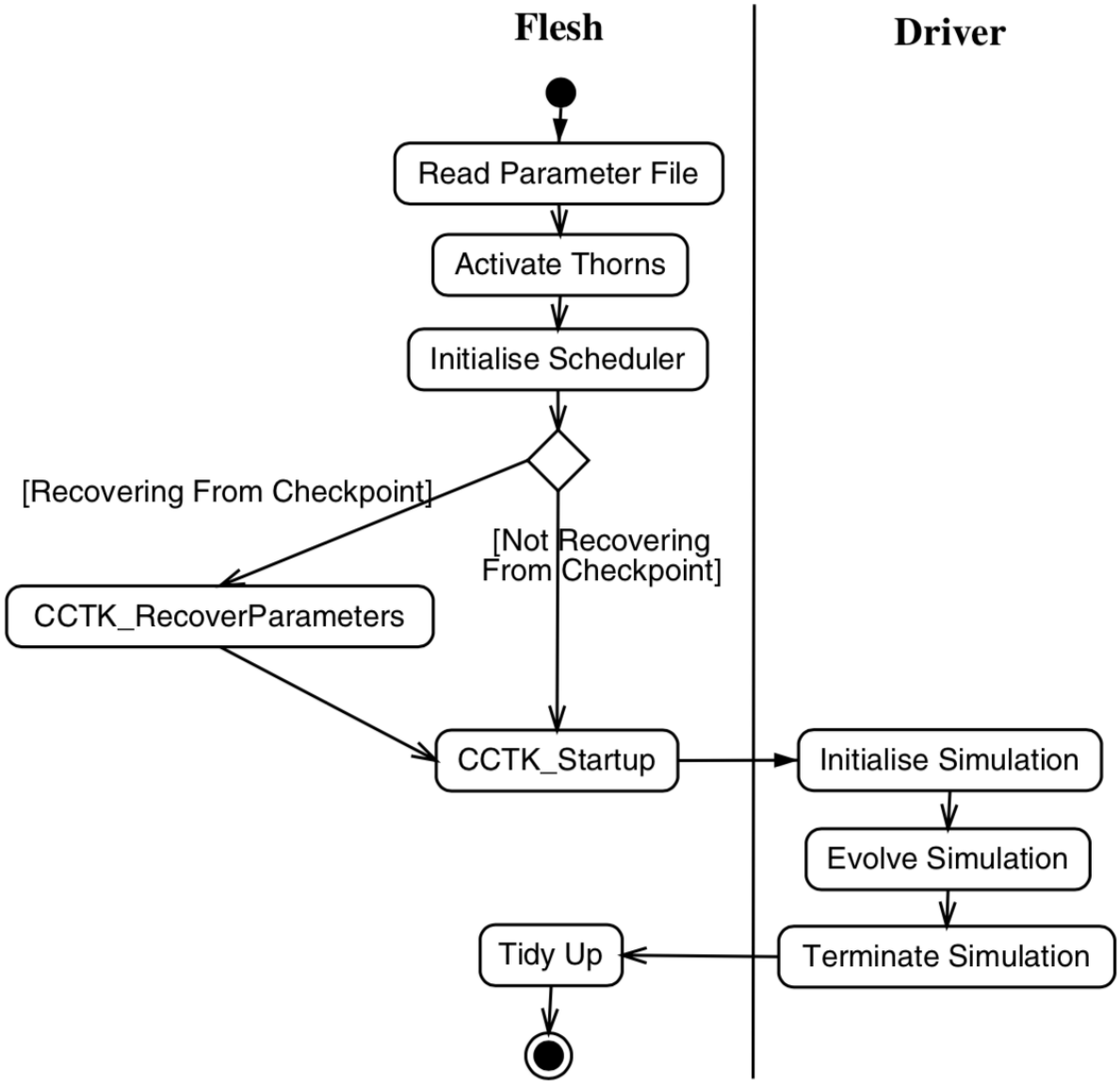
Cactus is a component framework. Its components are called thorns whereas the framework itself is called the flesh. The flesh is the core of Cactus, it provides the APIs for thorns to communicate with each other, and performs a number of administrative tasks at build–time and run–time.

Note: The Cactus API supports C/C++ and F77/F90 programming languages for the thorns. This makes it easier for scientists to turn existing codes into thorns which can then make use of the complete Cactus infrastructure, and in turn be used by other thorns within Cactus.

1.1.1 Flesh

At run-time the flesh parses a user provided parameter file which defines which thorns are required and provides key-value pairs of parameter assignments. The flesh then activates only the required thorns, sets the given parameters, using default values for parameters which are not specified in the parameter file, and creates the schedule of which functions provided by the activated thorns to run at which time.

Note: In order to ensure code portability, the flesh is written in ANSI C. Thorns, however, may be written in C, C++, FORTRAN 77 or Fortran 90. In order to ensure that thorns are platform independent, the configuration script determines various machine-specific details, such as the presence or absence of certain non-ANSI C functions or the sizes of variables, which vary from platform to platform – in order to avoid problems with this, e.g. in cross-language calls or parallel jobs distributed across heterogeneous platforms, Cactus defines a set of types, such as *CCTK_REAL*, *CCTK_INT*, which are guaranteed to have the same size on all platforms and in all languages. This also means that runs can produce exactly the same results on different machines, and checkpoint files created on one machine can be restarted on any other.


Flesh
Driver
Read Parameter File
Activate Thorns
Initialise Scheduler
[Recovering From Checkpoint]
[Not Recovering From Checkpoint]
CCTK_RecoverParameters
CCTK_Startup
Initialise Simulation
Evolve Simulation
Terminate Simulation
Tidy Up

1.1.2 Driver

Drivers are responsible for memory management for grid variables, all parallel operations, Input/Output(IO) and mesh refinement.

Note: basic parallelisation operations:

- ghost-zone synchronisation between sub-domains;
- generalised reduction operators;
- generalised interpolation operators.

There are two driver thorns: the unigrid *PUGH* driver and the adaptive mesh refinement (AMR) *Carpet* driver. Which driver is used is determined by which is activated at run-time.

1.1.3 Modularity

A thorn is the basic working module within Cactus. All user supplied code goes into thorns, which are, by and large, independent of each other. The connection from a thorn to the flesh or to other thorns is specified in configuration files that are parsed at compile time and used to generate glue code that encapsulates the external appearance of a thorn.

A thorn consists of a subdirectory of an arrangement containing four administrative files written in the Cactus Configuration Language (CCL):

- *interface.ccl*: Defines the thorn interface and inheritance along with variables and aliased functions.
- *param.ccl*: This defines the parameters that are used to control the thorn.
- *schedule.ccl*: This defines which functions are called from the thorn and when they are called. It also handles memory and communication assignment for grid variables.

Thorns can also contain

- *configuration.ccl*: This file is optional for a thorn. If it exists, it contains extra configuration options of this thorn.
- a subdirectory called *src*, which should hold source files and compilation instructions for the thorn.
- a subdirectory *src/include* for include files.
- a *README* containing a brief description of the thorn.
- a *doc* directory for documentation.
- a *par* directory for example parameter files.
- a *test* subdirectory may also be added, to hold the thorn's test suite.

Thorns are grouped into arrangements. This is a logical grouping of thorns which is purely for organisational purposes. The arrangements live in the arrangements directory of the main Cactus directory.

There exists a special include source mechanism to generate include files from sections of user code, allowing inlining of code between different thorns. This mechanism arguably breaks the separation between thorns, but can lead to large performance gains if automatic, compiler-dependent cross-file inlining is not available or not reliable. The include source mechanism supports arbitrary languages, including Fortran, and the inlined code is surrounded by guards making it possible to enable or disable the inlined code at run time. One particular disadvantage of this mechanism is namespace pollution, similar to manually inlined code.

1.1.4 Accuracy

we made the following choices:

- 4th order accurate finite differences,
- 4th order accurate Runge-Kutta time integrator,
- 3 timelevels for evolved grid functions,
- 3 ghostzones for interprocess synchronization,
- 5th order accurate spatial and 2nd order accurate temporal interpolation at mesh refinement boundaries,
- 5th order Kreiss-Oliger dissipation terms added to the right hand side of the evolution equations,

1.1.5 Visualization

For visualizing 1-dimensional ASCII output, standard tools like matplotlib are often used; for 2- and 3-dimensional HDF5 output, VisIt is popular (freely available) options.

I written a python library named CactusTool The source code for CactusTool is hosted on GitHub at https://github.com/YuLiumt/CactusTool

You can clone it with

```
$ git clone https://github.com/YuLiumt/CactusTool.git
```

and run

```
$ pip install -e .
```

1.2 Install

Users of the Einstein Toolkit are encouraged to register to become one of its.

1.2.1 Required Software

The main requirements are:

- Client tools for Source Code Repositories: CVS, SVN and git
- Compilers: C, C++ and Fortran 90
- MPI implementation: This is needed for the Carpet driver
- Standard development tools: Perl, etc.

Linux

```
# On Debian/Ubuntu/Mint use this command:
$ sudo apt-get install -y subversion gcc git numactl libgsl-dev libpapi-dev python␣
↪libhwloc-dev make libopenmpi-dev libhdf5-openmpi-dev libfftw3-dev libssl-dev␣
↪liblapack-dev g++ curl gfortran patch pkg-config libhdf5-dev libjpeg-turbo?-dev
# On Fedora use this command:
```

(continues on next page)

(continued from previous page)

```
$ sudo dnf install -y libjpeg-turbo-devel gcc git lapack-devel make subversion gcc-
↪c++ which papi-devel python hwloc-devel openmpi-devel hdf5-openmpi-devel openssl-
↪devel libtool-ltdl-devel numactl-devel gcc-gfortran findutils hdf5-devel fftw-devel␣
↪patch gsl-devel pkgconfig
$ module load mpi/openmpi-x86_64 # You will have to repeat the module load command␣
↪each time you would like to compile or run the code.
# On Centos use this command:
$ sudo yum install -y epel-release
$ sudo yum install -y libjpeg-turbo-devel gcc git lapack-devel make subversion gcc-
↪c++ which papi-devel hwloc-devel openmpi-devel hdf5-openmpi-devel openssl-devel␣
↪libtool-ltdl-devel numactl-devel gcc-gfortran hdf5-devel fftw-devel patch gsl-devel
```

Mac

MacPorts

1. Install Xcode from the Apple App Store. In addition agree to Xcode license in Terminal

   ```
   $ sudo xcodebuild -license
   ```

2. install MacPorts for your version of the Mac operating system, if you did not already install it

3. Next, please install the following packages, using the commands:

   ```
   $ sudo port -N install pkgconfig gcc9 openmpi-gcc9 fftw-3 gsl jpeg zlib␣
   ↪hdf5 +fortran +gfortran openssl ld64 +ld64_xcode
   $ sudo port select mpi openmpi-gcc9-fortran
   ```

Homebrew

1. Install Homebrew

   ```
   $ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
   ↪Homebrew/install/master/install)"
   ```

2. Next, please install the following packages, using the commands:

   ```
   $ brew install gnuplot pkg-config gcc fftw gsl hdf5 hwloc jpeg openssl␣
   ↪pkg-config szip open-mpi
   ```

1.2.2 GetComponents Tools

A script called GetComponents is used to fetch the components of the Einstein Toolkit. You may download and make it executable it as follows:

```
$ cd ~/
$ curl -kLO https://raw.githubusercontent.com/gridaphobe/CRL/ET_2019_03/GetComponents
$ chmod u+x GetComponents
```

Below checks out Cactus, the Einstein Toolkit thorns, the Simulation Factory and example parameter files into a directory named Cactus.

```
$ ./GetComponents --parallel https://bitbucket.org/einsteintoolkit/manifest/raw/ET_
↪2019_03/einsteintoolkit.th
```

Error:

- Some versions of svn might show problems with the parallel checkout. If you see errors like (svn: E155037: Previous operation has not finished), try without the –parallel option.
- svn: E155004: Could not update module <Thorn>

```
$ cd <Thorn_Path>
$ svn cleanup
```

1.2.3 The traditional way of compiling Cactus

The traditional way of compiling Cactus without SimFactory.

```
$ make ET-config options=<*.cfg> THORNLIST=<thornlist>
```

This creates a configuration called "ET", but any other name could be chosen here.

Once the configuration is done, the compilation process is simply

```
$ make -j <number of processes> ET
```

If everything is compiled correctly, the executable *cactus_ET* will be created under *./exe/*.

The typical procedure for running is

```
$ mpirun -np <num procs> ./exe/cactus_ET <parameter file>
```

1.2.4 SimFactory Tools

SimFactory needs to be configured before it can be used. The recommended way to do this is to use the setup command. SimFactory contains general support for specific operating systems commonly used on workstations or laptops, including Mac OS, Ubuntu, Cent OS and Scientific Linux. To configure SimFactory for one of these, you need to find the suitable files in *simfactory/mdb/optionlists* and *simfactory/mdb/runscripts* and specify their names on the sim setup command line.

Note: Newer versions do not have ubuntu.cfg. All flavors of linux should compile with generic.cfg.

```
# for Debian
$ ./simfactory/bin/sim setup-silent --optionlist=debian.cfg --runscript debian.sh
# for Ubuntu, Mint
$ ./simfactory/bin/sim setup-silent --optionlist=ubuntu.cfg --runscript debian.sh
# for Fedora (you may have to log out and back in if you have just intalled mpich to␣
↪make the module command work)
$ module load mpi
$ ./simfactory/bin/sim setup-silent --optionlist=fedora.cfg --runscript debian.sh
# OSX+MacPorts
$ ./simfactory/bin/sim setup-silent --optionlist=osx-macports.cfg --runscript osx-
↪macports.run
```

(continues on next page)

(continued from previous page)

```
# OSX+Homebrew
$ ./simfactory/bin/sim setup-silent --optionlist=osx-homebrew.cfg --runscript generic-
↪mpi.run
```

Note: Generally, configuring SimFactory means providing an optionlist, for specifying library locations and build options, a submit script for using the batch queueing system, and a runscript, for specifying how Cactus should be run, e.g. which mpirun command to use.

Follow the on-screen prompts. This will output your choices in the configuration file *simfactory/etc/defs.local.ini*.

Note: It is likely that you will have to further customise this file. You can see some possible option settings in *simfactory/etc/defs.local.ini.example*.

SimFactory needs to have a machine definition for every machine that it is run on. If you are using a machine that SimFactory already has a definition for, such as a well-known supercomputer used by others in the Cactus community, then no additional setup is required. If, however, you are running SimFactory on an individual laptop or on an unsupported supercomputer, the setup command will also create a new machine definition for the local machine in *./repos/simfactory2/mdb/machines/<hostname>.ini*. You may also have to add extra information to this file.

Note: A machine consists of a certain number of nodes, each of which consists of a certain number of cores.

The user chooses the total number of threads (–procs). The user can also choose the number of threads per process (–num-threads) and the number of threads per core (–num-smt). Additionally, the user can also specify the number of cores per node (–ppn) and the number of threads per node (–ppn-used). The number of nodes is always chosen automatically.

Note that nodes and cores are requested from the queuing system, while processes and threads are started by SimFactory.

For more details you can see https://simfactory.bitbucket.io/simfactory2/userguide/processterminology.html

Note: The main SimFactory binary is called “sim” and is located in simfactory/bin. You can execute SimFactory explicitly as ./simfactory/bin/sim, but we recommend that you set up a shell alias in your shell startup file so that you can just use the command “sim”. For bash users this file is .bashrc on Linux. Add the following to the shell startup file:

```
$ alias sim=./simfactory/bin/sim
```

Error:

- checking whether the C compiler (gcc-8 -g -std=c11 -lgfortran) works... no

Building the Einstein Toolkit

Assuming that SimFactory has been successfully set up on your machine, you should be able to build the Einstein Toolkit with

```
$ ./simfactory/bin/sim build --mdbkey make 'make -j2' --thornlist ../einsteintoolkit.
↪th | cat
```

The most important argument to this command is the *–thornlist* option, as this tells Cactus which thorns from your source tree you want to include in the configuration.

Note: Note that, typically, one will not be compiling all the thorns provided with the ET. Compilation is time-consuming, and different configurations also take a significant amount of disk space. One therefore typically builds a thornlist that is as small as possible, including only the required thorns. Care should be taken, though, as there are often non-trivial dependencies between thorns. If one thorn which is required by another thorn is not mentioned in the thornlist, compilation will abort (with the corresponding error message).

Running the Einstein Toolkit

Simulations must always be created before they can be submitted or run. Since it is very common to want to create a simulation and immediately submit or run it, SimFactory provides the *create-run* and *create-submit* commands. These commands create the simulation and then either run or submit it immediately.

Note: If you are working on a laptop or workstation, you can run SimFactory simulations directly in your terminal without going via a queuing system. However, if you are running SimFactory on a supercomputer with a queuing system, you cannot run simulations directly using the run command - they must instead be submitted to the queuing system, such as the PBS queuing system.

SimFactory needs to know a name for the simulation as well as what parameter file to use. You can either specify the name on the command line and give the parameter file with the *–parfile* option.

example

```
$ ./simfactory/bin/sim create-run static_tov --parfile=par/static_tov_small_short.par␣
↪--procs=2 --num-threads=1 --ppn-used=2  --walltime=8:0:0 | cat
```

It is often useful to use a parameter file script, rather than a parameter file, as a basic description of a simulation. For example, when performing a convergence test, many parameters might change between simulations, and changing them all manually is tedious and error-prone.

A parameter file script is a file with a ".rpar" extension which, when executed, generates a file in the same place but with a ".par" extension. You can write a parameter file script in any language.

We provide examples in python.

```
#!/usr/bin/env python

import sys
import re
from string import Template

dtfac = 0.5

lines = """
Time::dtfac                     = $dtfac
```

(continues on next page)

(continued from previous page)

```
"""

data = open(re.sub(r'(.*)\.rpar$', r'\1.par', sys.argv[0]), 'w')
data.write(Template(lines).substitute(locals()))
```

These parameter file scripts look like standard Cactus parameter files but with $var variable replacements (in this case for the dtfac variable). You can define new variables and do calculations in the header and use the variables in the main body.

If you want to use the Cactus $parfile syntax, you need to escape the $

```
IO::out_dir = $$parfile
```

CHAPTER 2

Thorn Guide

Typographical Conventions:

- $< \cdots >$ Indicates a compulsory argument.
- $[\ldots]$ Indicates an optional argument.
- $|$ Indicates an exclusive or.

2.1 Parameter File Syntax

A parameter file is used to control the behaviour of a Cactus executable. It specifies initial values for parameters as defined in the various thorns' param.ccl files. The name of a parameter file is often given the suffix .par, but this is not mandatory.

A parameter file is a text file whose lines are either comments or parameter statements. Comments are begin with either '#' or '!'. A parameter statement consists of one or more parameter names, followed by an '=', followed by the value for this parameter.

The first parameter statement in any parameter file should set ActiveThorns, which is a special parameter that tells the program which thorns are to be activated. Only parameters from active thorns can be set (and only those routines scheduled by active thorns are run). By default, all thorns are inactive.

2.2 Flesh

2.2.1 parameter

- Description of this simulation

```
>>> Cactus::cctk_run_title = "Description of this simulation"
```

- Give detailed information for each warning statement

```
>>> Cactus::cctk_full_warnings = yes
WARNING level 3 from host ubuntu process 0
    while executing schedule bin CCTK_BASEGRID, routine IOASCII::IOASCII_Choose1D
    in thorn IOUtil, file /home4/yuliu/Cactus/arrangements/CactusBase/IOUtil/src/
↪Utils.c:361:
    -> IOUtil_1DLines: Found no default Cartesian coordinate system associated␣
↪with grid variables of dimension 2, and no slice center index coordinates were␣
↪given either - slice center will not be set up for output of 1D lines from 2D␣
↪variables
>>> Cactus::cctk_full_warnings = no
WARNING[L3,P0] (IOUtil): IOUtil_1DLines: Found no default Cartesian coordinate␣
↪system associated with grid variables of dimension 2, and no slice center index␣
↪coordinates were given either - slice center will not be set up for output of␣
↪1D lines from 2D variables
```

Level	Description
0	abort the Cactus run
1	the results of this run will be wrong,
2	the user should know about this, but the problem is not terribly surprising
3	this is for small problems that can probably be ignored, but that careful people may want to know about
4	these messages are probably useful only for debugging purposes

- Print the scheduling tree to standard output

```
>>> Cactus::cctk_show_schedule = yes
if (recover initial data)
    Recover parameters
endif
Startup routines
    [CCTK_STARTUP]
Startup routines which need an existing grid hierarchy
    [CCTK_WRAGH]
Parameter checking routines
    [CCTK_PARAMCHECK]
Initialisation
    if (NOT (recover initial data AND recovery_mode is 'strict'))
        [CCTK_PREREGRIDINITIAL]
        Set up grid hierarchy
        [CCTK_POSTREGRIDINITIAL]
        [CCTK_BASEGRID]
        [CCTK_INITIAL]
        [CCTK_POSTINITIAL]
        Initialise finer grids recursively
        Restrict from finer grids
        [CCTK_POSTRESTRICTINITIAL]
        [CCTK_POSTPOSTINITIAL]
        [CCTK_POSTSTEP]
    endif
    if (recover initial data)
        [CCTK_BASEGRID]
        [CCTK_RECOVER_VARIABLES]
        [CCTK_POST_RECOVER_VARIABLES]
    endif
    if (checkpoint initial data)
        [CCTK_CPINITIAL]
    endif
```

(continues on next page)

(continued from previous page)

```
    if (analysis)
        [CCTK_ANALYSIS]
    endif
Output grid variables
do loop over timesteps
    [CCTK_PREREGRID]
    Change grid hierarchy
    [CCTK_POSTREGRID]
    Rotate timelevels
    iteration = iteration+1
    t = t+dt
    [CCTK_PRESTEP]
    [CCTK_EVOL]
    Evolve finer grids recursively
    Restrict from finer grids
    [CCTK_POSTRESTRICT]
    [CCTK_POSTSTEP]
    if (checkpoint)
        [CCTK_CHECKPOINT]
    endif
    if (analysis)
        [CCTK_ANALYSIS]
    endif
    Output grid variables
enddo
Termination routines
    [CCTK_TERMINATE]
Shutdown routines
    [CCTK_SHUTDOWN]
Routines run after changing the grid hierarchy:
    [CCTK_POSTREGRID]
>>> Cactus::cctk_show_schedule = no
None
```

- Provide runtime of each thorn

```
>>> Cactus::cctk_timer_output = "full"
================================================================================================
Thorn           | Scheduled routine in time bin           | gettimeofday [secs] |␣
↪getrusage [secs]
================================================================================================
CoordBase       | Register a GH extension to store the coo|          0.00000400 |␣
↪       0.00000000
------------------------------------------------------------------------------
↪-----------------
                | Total time for CCTK_STARTUP             |          0.00000400 |␣
↪       0.00000000
================================================================================================
------------------------------------------------------------------------------
↪-----------------
                | Total time for simulation               |          0.00004400 |␣
↪       0.00000000
================================================================================================
>>> Cactus::cctk_timer_output = "off"
None
```

- Condition on which to terminate evolution loop

```
>>> Cactus::terminate = "iteration"
>>> Cactus::cctk_itlast = 0
----------------
it  |          |
    |    t     |
----------------
  0 |    0.000 |
>>> Cactus::terminate = "iteration"
>>> Cactus::cctk_itlast = 5
----------------
it  |          |
    |    t     |
----------------
  0 |    0.000 |
  1 |    1.000 |
  2 |    2.000 |
  3 |    3.000 |
  4 |    4.000 |
  5 |    5.000 |
>>> Cactus::terminate = "time"
>>> Cactus::cctk_initial_time = 10
>>> Cactus::cctk_final_time = 15
----------------
it  |          |
    |    t     |
----------------
  0 |   10.000 |
  1 |   11.000 |
  2 |   12.000 |
  3 |   13.000 |
  4 |   14.000 |
  5 |   15.000 |
```

2.3 Grid

2.3.1 CoordBase

The CoordBase thorn provides a method of registering coordinate systems and their properties.

CoordBase provides a way for specifying the extent of the simulation domain that is independent of the actual coordinate and symmetry thorns. This is necessary because the size of the physical domain is not necessarily the same as the size of the computational grid, which is usually enlarged by symmetry zones and/or boundary zones.

Note: The black hole "source" region has a length scale of GM/c^2, where G is Newton's constant, M is the total mass of the two black holes, and c is the speed of light. The gravitational waves produced by the source have a length scale up to $\sim 100GM/c^2$. The source region requires grid zones of size $\lesssim 0.01GM/c^2$ to accurately capture the details of the black holes' interaction, while the extent of the grid needs to be several hundred GM/c^2 to accurately capture the details of the gravitational wave signal.

Parameter

- Specifying the extent of the physical domain.

```
>>> CoordBase::domainsize = "minmax" # lower and upper boundary locations
>>> CoordBase::xmin = -540.00
>>> CoordBase::ymin = -540.00
>>> CoordBase::zmin = -540.00
>>> CoordBase::xmax =  540.00
>>> CoordBase::ymax =  540.00
>>> CoordBase::zmax =  540.00
>>> CoordBase::spacing = "gridspacing" # grid spacing
>>> CoordBase::dx    = 1
>>> CoordBase::dx    = 18.0
>>> CoordBase::dy    = 18.0
>>> CoordBase::dz    = 18.0
INFO (Carpet): CoordBase domain specification for map 0:
    physical extent: [-540,-540,-540] : [540,540,540]   ([1080,1080,1080])
    base_spacing   : [18,18,18]
```

- Boundary description.

```
>>> CoordBase::boundary_size_x_lower = 3
>>> CoordBase::boundary_size_y_lower = 3
>>> CoordBase::boundary_size_z_lower = 3
>>> CoordBase::boundary_size_x_upper = 3
>>> CoordBase::boundary_size_y_upper = 3
>>> CoordBase::boundary_size_z_upper = 3
>>> CoordBase::boundary_internal_x_lower = "no"
>>> CoordBase::boundary_internal_y_lower = "no"
>>> CoordBase::boundary_internal_z_lower = "no"
>>> CoordBase::boundary_internal_x_upper = "no"
>>> CoordBase::boundary_internal_y_upper = "no"
>>> CoordBase::boundary_internal_z_upper = "no"
>>> CoordBase::boundary_staggered_x_lower = "no"
>>> CoordBase::boundary_staggered_y_lower = "no"
>>> CoordBase::boundary_staggered_z_lower = "no"
>>> CoordBase::boundary_staggered_x_upper = "no"
>>> CoordBase::boundary_staggered_y_upper = "no"
>>> CoordBase::boundary_staggered_z_upper = "no"
>>> CoordBase::boundary_shiftout_x_lower = 0
>>> CoordBase::boundary_shiftout_y_lower = 0
>>> CoordBase::boundary_shiftout_z_lower = 0
>>> CoordBase::boundary_shiftout_x_upper = 0
>>> CoordBase::boundary_shiftout_y_upper = 0
>>> CoordBase::boundary_shiftout_z_upper = 0
INFO (Carpet): Boundary specification for map 0:
    nboundaryzones: [[3,3,3],[3,3,3]]
    is_internal   : [[0,0,0],[0,0,0]]
    is_staggered  : [[0,0,0],[0,0,0]]
    shiftout      : [[0,0,0],[0,0,0]]
INFO (Carpet): CoordBase domain specification for map 0:
    interior extent: [-522,-522,-522] : [522,522,522]   ([1044,1044,1044])
    exterior extent: [-576,-576,-576] : [576,576,576]   ([1152,1152,1152])
```

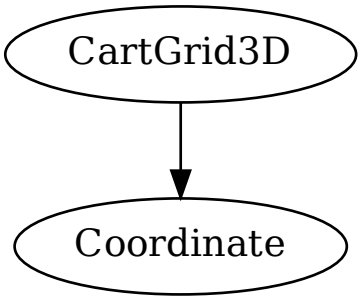
Warning

- The boundary must be contained in the active part of the domain boundaries <= domain_active

```
>>> CoordBase::xmin = -200.0
>>> CoordBase::xmax = +200.0
```

2.3.2 CartGrid3D

CartGrid3D allows you to set up coordinates on a 3D Cartesian grid in a flexible manner.



Parameter

- Get specification from CoordBase

```
>>> CartGrid3D::type = "coordbase"
```

- Get specification from MultiPatch

```
>>> CartGrid3D::type = "multipatch"
>>> CartGrid3D::set_coordinate_ranges_on = "all maps"
```

Output

- 3D Cartesian grid coordinates

```
>>> CarpetIOHDF5::out2D_vars = "grid::coordinates"
```

2.3.3 SymBase

Thorn SymBase provides a mechanism by which symmetry conditions can register routines that handle this mapping when a global interpolator is called.

2.4 Adaptive Mesh Refinement

It is often the case in simulations of physical systems that the most interesting phenomena may occur in only a subset of the computational domain. In the other regions of the domain it may be possible to use a less accurate approximation, thereby reducing the computational resources required, and still obtain results which are essentially similar to those obtained if no such reduction is made.

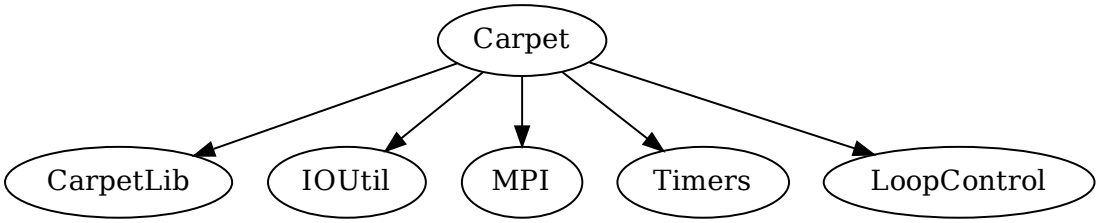
In particular, we may consider using a computational mesh which is non-uniform in space and time, using a finer mesh resolution in the "interesting" regions where we expect it to be necessary, and using a coarser resolution in other areas. This is what we mean by mesh refinement (MR).

2.4.1 Carpet

Carpet is a mesh refinement driver for Cactus. It knows about a hierarchy of refinement levels, where each level is decomposed into a set of cuboid grid patches. The grid patch is the smallest unit of grid points that Carpet deals with. Carpet parallelises by assigning sets of grid patches to processors.

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 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.

Note: Carpet does not handle staggered grids. Carpet does not provide cell-centered refinement. Carpet always enables all storage.



Parameter

- Use the domain description from CoordBase to specify the global extent of the coarsest grid.

```
>>> Carpet::domain_from_coordbase = yes
```

- Use the domain description from MultiPatch

```
>>> Carpet::domain_from_multipatch = yes
```

- Maximum number of refinement levels, including the base level

```
>>> Carpet::max_refinement_levels = 1
```

- Ghost zones in each spatial direction

> **Note:** 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.

```
>>> Carpet::ghost_size = 3
INFO (Carpet): Base grid specification for map 0:
    number of coarse grid ghost points: [[3,3,3],[3,3,3]]
```

- Fill past time levels from current time level after calling initial data routines.

> **Note:** The boundary values of the finer grids have to be calculated from the coarser grids through interpolation. An because the time steps on the finer grids are smaller, there is not always a corresponding value on the coarser grids available. This makes it necessary to interpolate in time between time steps on the coarser grids. three time levels allow for a second order interpolation in time.

```
>>> Carpet::init_fill_timelevels = "yes"
>>> Carpet::init_3_timelevels = "no" # Set up 3 timelevels of initial data
```

- Carpet currently supports polynomial interpolation, up to quadratic interpolation in time, which requires keeping at least two previous time levels of data. It also supports up to quintic interpolation in space, which requires using at least three ghost zones.

```
>>> Carpet::prolongation_order_space = 5
>>> Carpet::prolongation_order_time  = 2
```

- Refinement factor

```
>>> Carpet::refinement_factor = 2
```

- Temporal refinement factors over the coarsest level

```
>>> grid::dxyz = 18
>>> time::dtfac = 0.25
>>> Carpet::max_refinement_levels    = 7
>>> Carpet::time_refinement_factors  = "[1,1,2,4,8,16,32]"
INFO (Time): Timestep set to 4.5 (courant_static)
INFO (Time): Timestep set to 4.5 (courant_static)
INFO (Time): Timestep set to 2.25 (courant_static)
INFO (Time): Timestep set to 1.125 (courant_static)
INFO (Time): Timestep set to 0.5625 (courant_static)
INFO (Time): Timestep set to 0.28125 (courant_static)
INFO (Time): Timestep set to 0.140625 (courant_static)
```

- Use buffer zones

```
>>> Carpet::use_buffer_zones = "yes"
INFO (Carpet): Buffer zone counts (excluding ghosts):
    [0]: [[0,0,0],[0,0,0]]
    [1]: [[9,9,9],[9,9,9]]
    [2]: [[9,9,9],[9,9,9]]
    [3]: [[9,9,9],[9,9,9]]
```

(continues on next page)

(continued from previous page)

```
    [4]: [[9,9,9],[9,9,9]]
    [5]: [[9,9,9],[9,9,9]]
    [6]: [[9,9,9],[9,9,9]]
```

- Carpet uses vertex-centered refinement. That is, each coarse grid point coincides with a fine grid point.

```
>>> Carpet::refinement_centering = "vertex"
```

- Print detailed statistics periodically

```
>>> Carpet::output_timers_every = 512
>>> Carpet::schedule_barriers = "yes" # Insert barriers between scheduled items,␣
↪so that timer statistics become more reliable (slows down execution)
>>> Carpet::sync_barriers = "yes" # Insert barriers before and after syncs, so␣
↪that the sync timer is more reliable (slows down execution)
```

- Try to catch uninitialised grid elements or changed timelevels.

> **Note:** 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.

```
>>> Carpet::check_for_poison = "no"
>>> Carpet::poison_new_timelevels = "yes"
>>> CarpetLib::poison_new_memory = "yes"
>>> CarpetLib::poison_value      = 114
```

- Base Multigrid level and factor

```
>>> Carpet::convergence_level = 0
>>> Carpet::convergence_factor = 2
INFO (Carpet): Adapted domain specification for map 0:
    convergence factor: 2
    convergence level : 0
```

Output

- File name to output grid coordinates.

```
>>> Carpet::grid_coordinates_filename = "carpet-grid.asc"
```

Warning

- INFO (Carpet): There are not enough time levels for the desired temporal prolongation order in the grid function group "ADMBASE::METRIC". With Carpet::prolongation_order_time=2, you need at least 3 time levels.

2.4.2 CarpetLib

This thorn contains the backend library that provides mesh refinement. CarpetLib contains of three major parts:

- a set of generic useful helpers

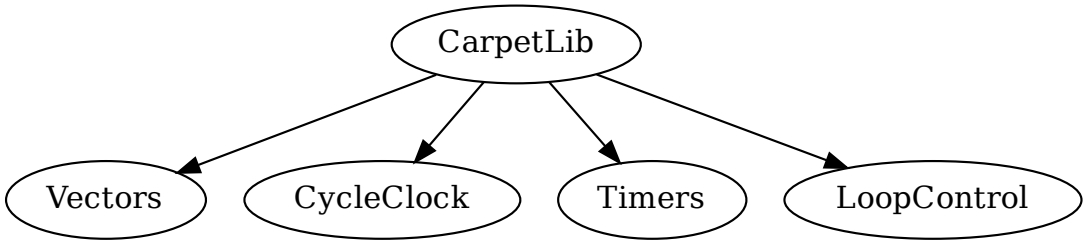
A class `vect<T,D>` provides small D-dimensional vectors of the type T. A vect corresponds to a grid point location. The class `bbox<T,D>` provides D-dimensional bounding boxes using type T as indices. A bbox defines the location and shape of a grid patch. Finally, `bboxset<T,D>` is a collection of bboxes.

- the grid hierarchy and data handling

  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.

- interpolation operators.

  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.



Parameter

- Provide one extra ghost point during restriction for staggered operators.

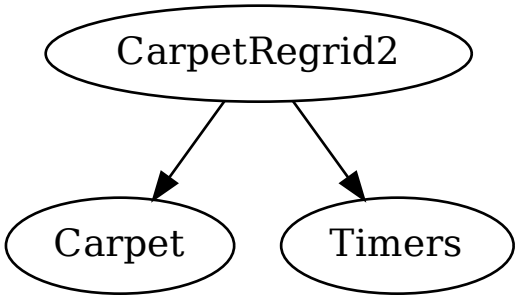
```
>>> CarpetLib::support_staggered_operators = "yes"
```

2.4.3 CarpetRegrid2

Set up refined regions by specifying a set of centres and radii about them. All it does is take a regridding specification from the user and translate that into a `gh`.

- `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. As all grid functions have the same number of ghost zones, there exists also only one `dh` for all grid functions.

Note: 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.



Parameter

- Set up refined regions by specifying a set of centres and radii about them.

```
>>> Carpet::max_refinement_levels = 7
>>> CarpetRegrid2::num_centres    = 2
>>> CarpetRegrid2::num_levels_1   = 7
>>> CarpetRegrid2::num_levels_2   = 7
>>> CarpetRegrid2::position_x_1   = -15.2
>>> CarpetRegrid2::position_x_2   =  15.2
>>>
>>> CarpetRegrid2::radius_1[1]    =  270.0
>>> CarpetRegrid2::radius_1[2]    =  162.0
>>> CarpetRegrid2::radius_1[3]    =   94.5
>>> CarpetRegrid2::radius_1[4]    =   40.5
>>> CarpetRegrid2::radius_1[5]    =   27.0
>>> CarpetRegrid2::radius_1[6]    =   13.5
>>>
>>> CarpetRegrid2::radius_2[1]    =  270.0
>>> CarpetRegrid2::radius_2[2]    =  162.0
>>> CarpetRegrid2::radius_2[3]    =   94.5
>>> CarpetRegrid2::radius_2[4]    =   40.5
>>> CarpetRegrid2::radius_2[5]    =   27.0
>>> CarpetRegrid2::radius_2[6]    =   13.5
>>>
>>> Carpet::refinement_factor = 2
INFO (CarpetRegrid2): Centre 1 is at position [-15.2,0,0] with 7 levels
INFO (CarpetRegrid2): Centre 2 is at position [15.2,0,0] with 7 levels
INFO (Carpet): Grid structure (superregions, coordinates):
    [0][0][0]   exterior: [-576.000000000000000,-576.000000000000000,-576.
↪000000000000000] : [576.000000000000000,576.000000000000000,576.
↪000000000000000] : [18.000000000000000,18.000000000000000,18.000000000000000]
    [1][0][0]   exterior: [-369.000000000000000,-351.000000000000000,-351.
↪000000000000000] : [369.000000000000000,351.000000000000000,351.
↪000000000000000] : [9.000000000000000,9.000000000000000,9.000000000000000]
    [2][0][0]   exterior: [-216.000000000000000,-202.500000000000000,-202.
↪500000000000000] : [216.000000000000000,202.500000000000000,202.
↪500000000000000] : [4.500000000000000,4.500000000000000,4.500000000000000]
    [3][0][0]   exterior: [-130.500000000000000,-114.750000000000000,-114.
↪750000000000000] : [130.500000000000000,114.750000000000000,114.
↪750000000000000] : [2.250000000000000,2.250000000000000,2.250000000000000]
```

(continues on next page)

(continued from previous page)

```
    [4][0][0]   exterior: [-66.375000000000000,-50.625000000000000,-50.
↪625000000000000] : [66.375000000000000,50.625000000000000,50.625000000000000] :␣
↪[1.125000000000000,1.125000000000000,1.125000000000000]
    [5][0][0]   exterior: [-47.250000000000000,-32.062500000000000,-32.
↪062500000000000] : [47.250000000000000,32.062500000000000,32.062500000000000] :␣
↪[0.562500000000000,0.562500000000000,0.562500000000000]
    [6][0][0]   exterior: [-31.218750000000000,-16.031250000000000,-16.
↪031250000000000] : [31.218750000000000,16.031250000000000,16.031250000000000] :␣
↪[0.281250000000000,0.281250000000000,0.281250000000000]
```

- Regrid every n time steps

```
>>> CarpetRegrid2::regrid_every = 32 # 2**(max_refinement_levels - 2)
```

- Minimum movement to trigger a regridding

```
>>> CarpetRegrid2::movement_threshold_1 = 0.10 # Regrid only if the regions have␣
↪changed sufficiently
INFO (CarpetRegrid2): Refined regions have not changed sufficiently; skipping␣
↪regridding
```

Warning

- PARAMETER ERROR (CarpetRegrid2): The number of requested refinement levels is larger than the maximum number of levels specified by Carpet::max_refinement_levels

```
>>> Carpet::max_refinement_levels = <number>
```

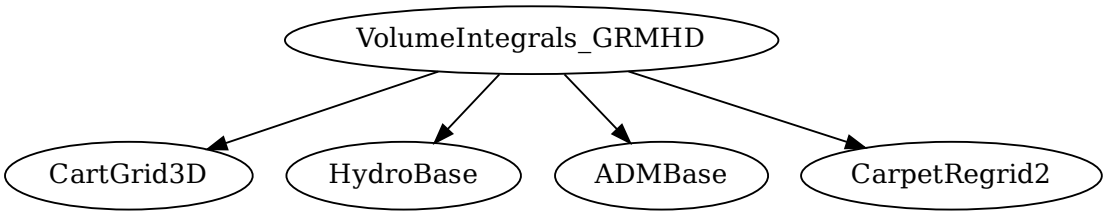
2.4.4 VolumeIntegrals_GRMHD

Thorn for integration of spacetime quantities. The center of mass (CoM) can be used to set the AMR center.

$$\rho_* = \alpha u^0 \sqrt{det(g)} \rho_0 = \Gamma \sqrt{det(g)} \rho_0$$

$$CoM^i = \frac{\int \rho_* x^i dV}{\int \rho_* dV}$$

Note: In a binary neutron star simulation, the maximum density is ~ 1, with atmosphere values surrounding the stars of at most $\sim$ 1e-6. The concern is that a bad conservatives-to-primitives inversion called before this diagnostic might cause Gamma to be extremely large. Capping Gamma at 1e4 should prevent the CoM integral from being thrown off significantly, which would throw off the NS tracking.



Parameter

- Set verbosity level: 1=useful info; 2=moderately annoying (though useful for debugging)

```
>>> VolumeIntegrals_GRMHD::verbose = 1
```

- How often to compute volume integrals?

```
>>> VolumeIntegrals_GRMHD::VolIntegral_out_every = 32 # 2**(max_refinement_levels␣
↪- 2)
```

- Number of volume integrals to perform

```
>>> VolumeIntegrals_GRMHD::NumIntegrals = 6
>>>
>>> # Integrate the entire volume with an integrand of 1. (used for testing/
↪validation purposes only).
>>> VolumeIntegrals_GRMHD::Integration_quantity_keyword[1] = "one"
>>>
>>> # To compute the center of mass in an integration volume originally centered␣
↪at (x,y,z) = (-15.2,0,0) with a coordinate radius of 13.5. Also use the center␣
↪of mass integral result to set the ZEROTH AMR center.
>>> VolumeIntegrals_GRMHD::Integration_quantity_keyword[2] = "centerofmass"
>>> VolumeIntegrals_GRMHD::volintegral_sphere__center_x_initial         [2] = -15.
↪2
>>> VolumeIntegrals_GRMHD::volintegral_inside_sphere__radius            [2] =  13.
↪5
>>> VolumeIntegrals_GRMHD::amr_centre__tracks__volintegral_inside_sphere[2] =  0
>>>
>>> # Same as above, except use the integrand=1 (for validation purposes, to␣
↪ensure the integration volume is approximately 4/3*pi*13.5^3).
>>> VolumeIntegrals_GRMHD::Integration_quantity_keyword[3] = "one"
>>> VolumeIntegrals_GRMHD::volintegral_sphere__center_x_initial         [3] = -15.
↪2
>>> VolumeIntegrals_GRMHD::volintegral_inside_sphere__radius            [3] =  13.
↪5
>>>
>>> # The neutron star originally centered at (x,y,z) = (+15.2,0,0). Also use the␣
↪center of mass integral result to set the ONETH AMR center.
>>> VolumeIntegrals_GRMHD::Integration_quantity_keyword[4] = "centerofmass"
>>> VolumeIntegrals_GRMHD::volintegral_sphere__center_x_initial         [4] =  15.
↪2
>>> VolumeIntegrals_GRMHD::volintegral_inside_sphere__radius            [4] =  13.
↪5
```

(continues on next page)

(continued from previous page)

```
>>> VolumeIntegrals_GRMHD::amr_centre__tracks__volintegral_inside_sphere[4] =  1
>>>
>>> # Same as above, except use the integrand=1 (for validation purposes, to␣
↪ensure the integration volume is approximately 4/3*pi*13.5^3).
>>> VolumeIntegrals_GRMHD::Integration_quantity_keyword[5] = "one"
>>> VolumeIntegrals_GRMHD::volintegral_sphere__center_x_initial         [5] =  15.
↪2
>>> VolumeIntegrals_GRMHD::volintegral_inside_sphere__radius            [5] =  13.
↪5
>>>
>>> # Perform rest-mass integrals over entire volume.
>>> VolumeIntegrals_GRMHD::Integration_quantity_keyword[6] = "restmass"
```

- Enable output file

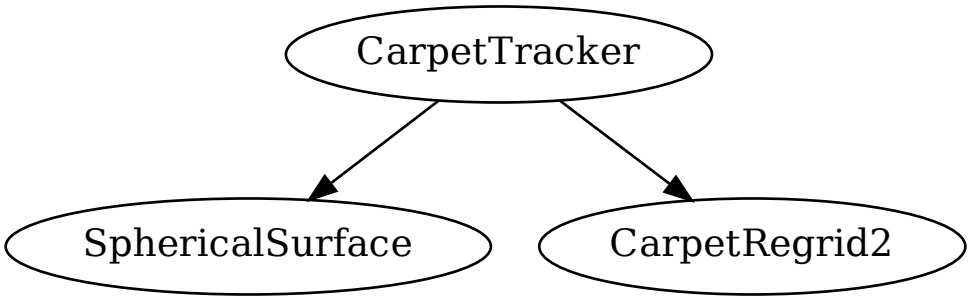
```
>>> VolumeIntegrals_GRMHD::enable_file_output = 1
>>> VolumeIntegrals_GRMHD::outVolIntegral_dir = "volume_integration"
```

- Gamma speed limit

```
>>> VolumeIntegrals_GRMHD::CoM_integrand_GAMMA_SPEED_LIMIT = 1e4
```

2.4.5 CarpetTracker

Object coordinates are updated by CarpetTracker, which provides a simple interface to the object trackers PunctureTracker and NSTracker in order to have the refined region follow the moving objects.



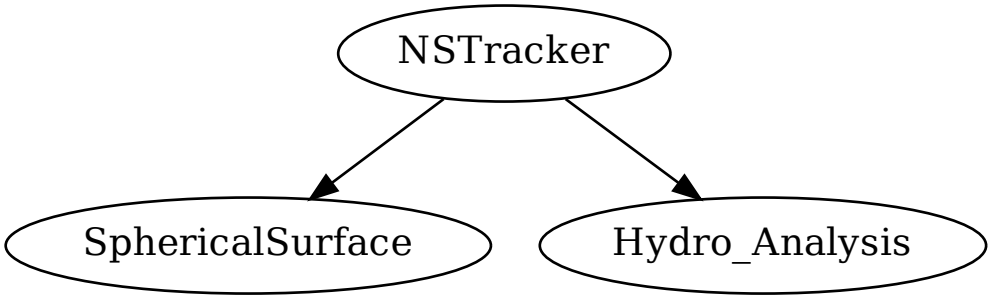
Parameter

- Spherical surface index or name which is the source for the location of the refine regions. (Maximum number of tracked surfaces less than 10)

```
>>> CarpetTracker::surface[0] = 0
<surface index>
>>> CarpetTracker::surface_name[0] = "Righthand NS"
<surface name>
```

2.4.6 NSTracker

This thorn can track the location of a neutron star, e.g. to guide mesh refinement.



Parameter

- Index or Name of the sperical surface which should be moved around

```
>>> NSTracker::NSTracker_SF_Name          = "Righthand NS"
>>> NSTracker::NSTracker_SF_Name_Opposite = "Lefthand NS"
<surface name>
>>> NSTracker::NSTracker_SF_Index          = 0
>>> NSTracker::NSTracker_SF_Index_Opposite = 1
<surface index>
```

- Allowed to move if new location is not too far from old.

```
>>> NSTracker::NSTracker_max_distance = 3
```

- grid scalar group containing coordinates of center of star

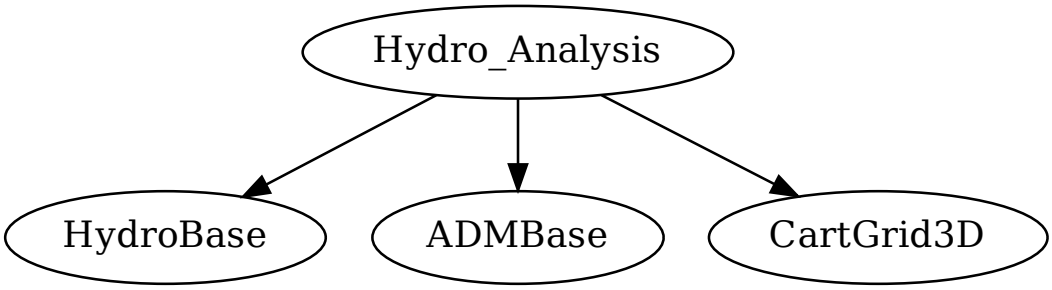
```
>>> NSTracker::NSTracker_tracked_location = "Hydro_Analysis::Hydro_Analysis_rho_
↪max_loc" # location of maximum
position_x[NSTracker_SF_Index] = Hydro_Analysis_rho_max_loc
position_x[NSTracker_SF_Index_Opposite] = - Hydro_Analysis_rho_max_loc
>>> NSTracker::NSTracker_tracked_location = "Hydro_Analysis::Hydro_Analysis_rho_
↪core_center_volume_weighted" # center of mass
```

- Time after which to stop updating the SF

```
>>> NSTracker::NSTracker_stop_time = -1
```

2.4.7 Hydro_Analysis

This thorn provides basic hydro analysis routines.



Parameter

- Look for the value and location of the maximum of rho

```
>>> Hydro_Analysis::Hydro_Analysis_comp_rho_max = "true"
>>> Hydro_Analysis::Hydro_Analysis_average_multiple_maxima_locations = "yes" #␣
↪when finding more than one global maximum location, average position
>>> Hydro_Analysis::Hydro_Analysis_comp_rho_max_every = 32 # 2**(max_refinement_
↪levels - 2)
```

- Look for the location of the volume-weighted center of mass

```
>>> Hydro_Analysis::Hydro_Analysis_comp_vol_weighted_center_of_mass = "yes"
```

- Look for the proper distance between the maximum of the density and the origin (along a straight coordinate line, not a geodesic)

```
>>> Hydro_Analysis::Hydro_Analysis_comp_rho_max_origin_distance = "yes"
>>> ActiveThorns = "AEILocalInterp"
>>> Hydro_Analysis::Hydro_Analysis_interpolator_name = "Lagrange polynomial␣
↪interpolation (tensor product)"
```

Output

- Coordinate location of the maximum of rho

```
>>> IOScalar::outScalar_vars = "Hydro_Analysis_rho_max"
>>> IOScalar::outScalar_vars = "Hydro_Analysis::Hydro_Analysis_rho_max_loc"
```

- coordinate location of the volume weightes center of mass

```
>>> IOScalar::outScalar_vars = "Hydro_Analysis_rho_center_volume_weighted"
```

- proper distance between the maximum of the density and the origin (along a straight coordinate line)

```
>>> IOScalar::outScalar_vars = "Hydro_Analysis::Hydro_Analysis_rho_max_origin_
↪distance"
```

Warning

- Cannot get handle for interpolation ! Forgot to activate an implementation providing interpolation operators (e.g. LocalInterp)?

```
>>> ActiveThorns = "LocalInterp"
```

- No driver thorn activated to provide an interpolation routine for grid arrays

```
>>> ActiveThorns = "CarpetInterp"
```

- No handle found for interpolation operator 'Lagrange polynomial interpolation (tensor product)'

```
>>> ActiveThorns = "AEILocalInterp"
```

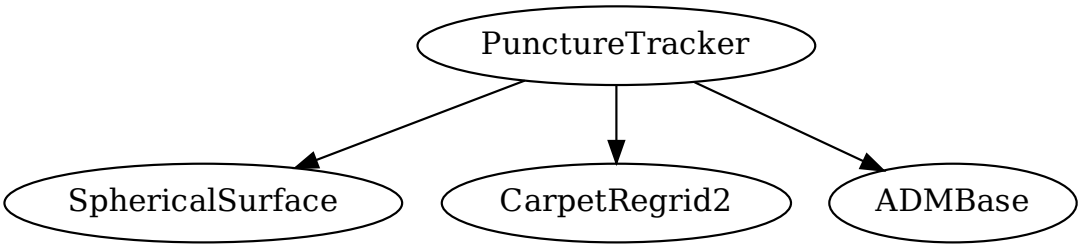
- No handle: '-2' found for reduction operator 'sum'

```
>>> ActiveThorns = "LocalReduce"
```

2.4.8 PunctureTracker

PunctureTracker track BH positions evolved with moving puncture techniques. The BH position is stored as the centroid of a spherical surface (even though there is no surface) provided by SphericalSurface.

$$pt\_loc = pt\_loc\_p - dt \times pt\_beta$$



Parameter

- A spherical surface index where we can store the puncture location

```
>>> PunctureTracker::which_surface_to_store_info[0] = 0
>>> PunctureTracker::track                       [0] = yes
>>> PunctureTracker::initial_x                   [0] =
>>> PunctureTracker::which_surface_to_store_info[1] = 1
>>> PunctureTracker::track                       [1] = yes
>>> PunctureTracker::initial_x                   [1] =
```

Warning

- No handle found for interpolation operator 'Lagrange polynomial interpolation'

```
>>> ActiveThorns = "AEILocalInterp"
```

- Error

```
>>> ActiveThorns = "SphericalSurface"
>>> SphericalSurface::nsurfaces = 2
>>> SphericalSurface::maxntheta = 66
>>> SphericalSurface::maxnphi   = 124
>>> SphericalSurface::verbose   = yes
```

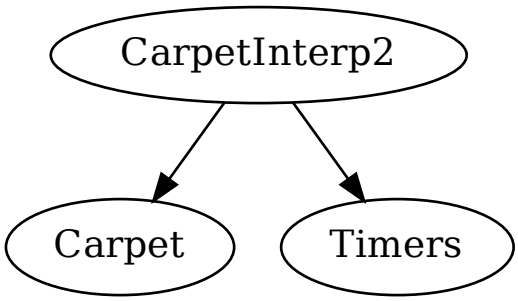
Output

- Location of punctures

```
>>> IOASCII::out0D_vars = "PunctureTracker::pt_loc"
```

2.4.9 CarpetInterp/CarpetInterp2

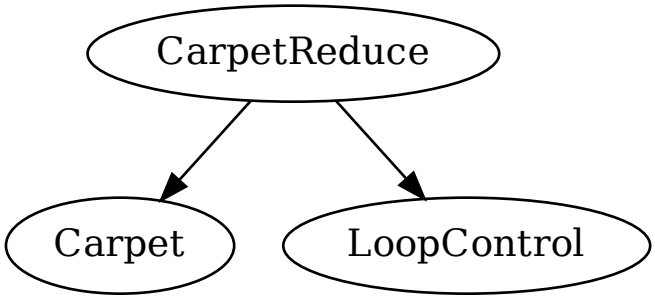
This thorn provides a parallel interpolator for Carpet.



2.4.10 CarpetReduce

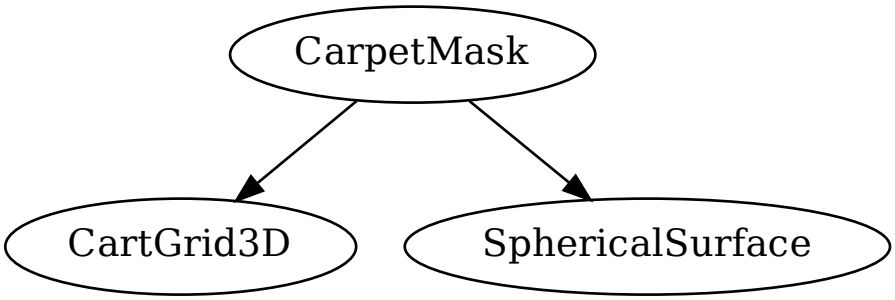
This thorn provides parallel reduction operators for Carpet. This thorn now uses a weight function. The weight is 1 for all "normal" grid points.

Note: The weight is set to 0 on symmetry and possible the outer boundary, and it might be set to $1/2$ on the edge of the boundary. The weight is also reduced or set to 0 on coarser grids that are overlaid by finer grid.



2.4.11 CarpetMask

Remove unwanted regions from Carpet's reduction operations. This can be used to excise the interior of horizons.



Parameter

- Exclude spherical surfaces with shrink factor

> **Note:** iweight = 0 if dr <= radius_{sf} * excluded_surface_factor

```
>>> CarpetMask::excluded_surface[0] = 0 # index of excluded surface
>>> CarpetMask::excluded_surface_factor[0] = 1
```

2.5 Initial Data

2.5.1 InitBase

Thorn InitBase specifies how initial data are to be set up.

Parameter

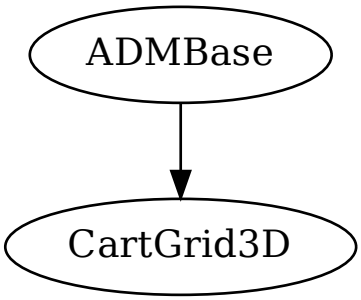
- Procedure for setting up initial data ["init_some_levels", "init_single_level", "init_two_levels", "init_all_levels"]:

```
>>> InitBase::initial_data_setup_method = "init_all_levels"
```

2.5.2 ADMBase

This thorn provides the basic variables used to communicate between thorns doing General Relativity in the 3+1 formalism.

- alp
- betax betay betaz
- dtalp
- dtbetax dtbetay dtbetaz
- gxx gyy gzz gxy gxz gyz
- kxx kyy kzz kxy kxz kyz



Parameter

- Initial data value

```
>>> ADMBase::initial_data    = "Meudon_Bin_NS"
>>> ADMBase::initial_lapse   = "Meudon_Bin_NS"
>>> ADMBase::initial_shift   = "zero"
>>> ADMBase::initial_dtlapse = "Meudon_Bin_NS"
>>> ADMBase::initial_dtshift = "zero"
```

- evolution method

```
>>> ADMBase::evolution_method         = "ML_BSSN"
>>> ADMBase::lapse_evolution_method   = "ML_BSSN"
>>> ADMBase::shift_evolution_method   = "ML_BSSN"
>>> ADMBase::dtlapse_evolution_method = "ML_BSSN"
>>> ADMBase::dtshift_evolution_method = "ML_BSSN"
```

- Number of time levels

```
>>> ADMBase::lapse_timelevels  = 3
>>> ADMBase::shift_timelevels  = 3
>>> ADMBase::metric_timelevels = 3
```

Output

- ADM

```
>>> CarpetIOHDF5::out2D_vars = "ADMBase::curv
>>>                             ADMBase::lapse
>>>                             ADMBase::metric
>>>                             ADMBase::shift"
```

Warning

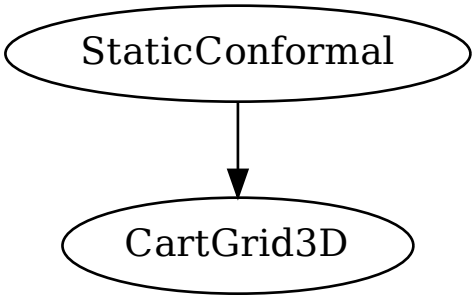
- The variable "ADMBASE::alp" has only 1 active time levels, which is not enough for boundary prolongation of order 1

```
>>> ADMBase::lapse_timelevels = 3
>>> ADMBase::shift_timelevels = 3
>>> ADMBase::metric_timelevels = 3
```

2.5.3 StaticConformal

StaticConformal provides aliased functions to convert between physical and conformal 3-metric values.

- psi: Conformal factor



Parameter

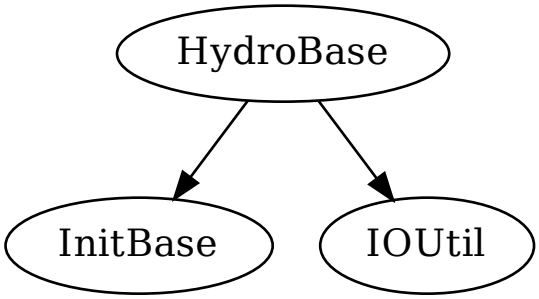
- Metric is conformal with static conformal factor, extrinsic curvature is physical

```
>>> ADMBase::metric_type = "static conformal"
```

2.5.4 HydroBase

HydroBase defines the primitive variables

- rho: rest mass density ρ
- press: pressure P
- eps: internal energy density ϵ
- vel[3]: contravariant fluid three velocity v^i
- w_lorentz: Lorentz Factor W
- Y_e: electron fraction Y_e
- Abar: Average atomic mass
- temperature: temperature T
- entropy: specific entropy per particle s
- Bvec[3]: contravariant magnetic field vector defined as
- Avec[3]: Vector potential
- Aphi: Electric potential for Lorentz Gauge



Parameter

- Number of time levels in evolution scheme

```
>>> InitBase::initial_data_setup_method = "init_all_levels"
>>> HydroBase::timelevels = 3
rho[i] = 0.0;
rho_p[i] = 0.0;
rho_p_p[i] = 0.0;
```

- The hydro initial value and evolution method (rho, vel, w_lorentz, eps)

```
>>> HydroBase::initial_hydro = "zero"
>>> HydroBase::evolution_method = "none"
```

- Other initial value and Evolution method

```
>>> HydroBase::initial_Avec = "none"
>>> HydroBase::initial_Aphi = "none"
```

```
>>> HydroBase::initial_Bvec = "none"
>>> HydroBase::Bvec_evolution_method = "none"
```

```
>>> HydroBase::initial_temperature = "none"
>>> HydroBase::temperature_evolution_method = "none"
```

```
>>> HydroBase::initial_entropy = "none"
>>> HydroBase::entropy_evolution_method = "none"
```

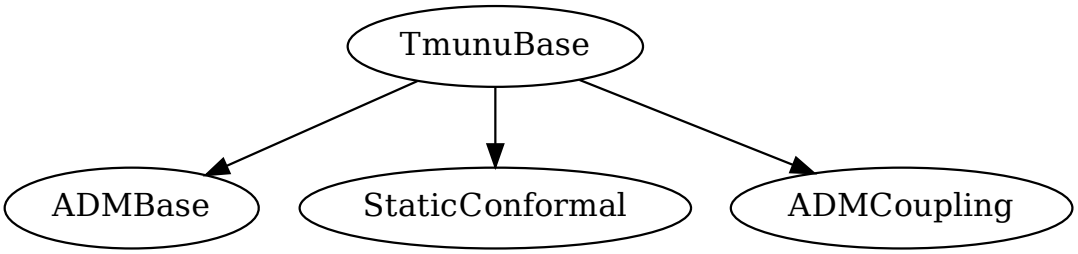
```
>>> HydroBase::initial_Abar = "none"
>>> HydroBase::Abar_evolution_method = "none"
```

```
>>> HydroBase::initial_Y_e = "none"
>>> HydroBase::Y_e_evolution_method = "none"
```

2.5.5 TmunuBase

Provide grid functions for the stress-energy tensor

$$T_{ab} = \begin{pmatrix} eTtt & eTtx & eTty & eTtz \\ & eTxx & eTxy & eTxz \\ & & eTyy & eTyz \\ & & & eTzz \end{pmatrix}$$



Parameter

- Should the stress-energy tensor have storage?

```
>>> TmunuBase::stress_energy_storage = yes
```

- Should the stress-energy tensor be calculated for the RHS evaluation?

```
>>> TmunuBase::stress_energy_at_RHS = yes
```

- Number of time levels

```
>>> TmunuBase::timelevels = 3
```

2.5.6 Meudon_Bin_NS

Import LORENE Bin_NS binary neutron star initial data.

Parameter

- Input file name containing LORENE data

```
>>> Meudon_Bin_NS::filename = "resu.d"
```

- Initial data EOS identifyer

```
>>> Meudon_Bin_NS::filename =
>>> Meudon_Bin_NS::eos_table =
```

2.5.7 Seed_Magnetic_Fields

Since the LORENE code cannot yet compute magnetized BNS models.

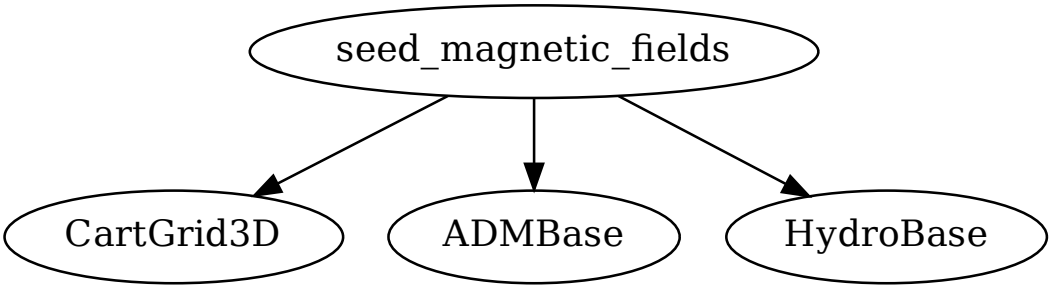
The following sets up a vector potential of the form

$$A_\phi = \varpi^2 A_b max[(X - X_{cut}), 0],$$

where ϖ is the cylindrical radius: $\sqrt{x^2 + y^2}$, and $X \in \{\rho, P\}$ is the variable P or ρ specifying whether the vector potential is proportional to P or ρ in the region greater than the cutoff.

This formulation assumes that A_r and $A_\theta = 0$; only A_ϕ can be nonzero. Thus the coordinate transformations are as follows:

$$A_x = -\frac{y}{\varpi^2} A_\phi$$
$$A_y = \frac{x}{\varpi^2} A_\phi$$



Parameter

- A-field prescription ["Pressure_prescription", "Density_prescription"]:

```
>>> Seed_Magnetic_Fields::Afield_type = "Pressure_prescription"
```

- Multiply A_ϕ by ϖ^2?

```
>>> Seed_Magnetic_Fields::enable_varpi_squared_multiplication = "yes"
```

- Magnetic field strength parameter.

```
>>> Seed_Magnetic_Fields::A_b = 1e-3
```

- Cutoff pressure, below which vector potential is set to zero. Typically set to 4% of the maximum initial pressure.

```
>>> Seed_Magnetic_Fields::P_cut = 1e-5
```

- Magnetic field strength pressure exponent $A_\phi = \varpi^2 A_b max[(P - P_{cut})^{n_s}, 0]$.

```
>>> Seed_Magnetic_Fields::n_s = 1
```

- Cutoff density, below which vector potential is set to zero. Typically set to 20% of the maximum initial density.

```
>>> Seed_Magnetic_Fields::rho_cut = 0.2 # If max density = 1.0
```

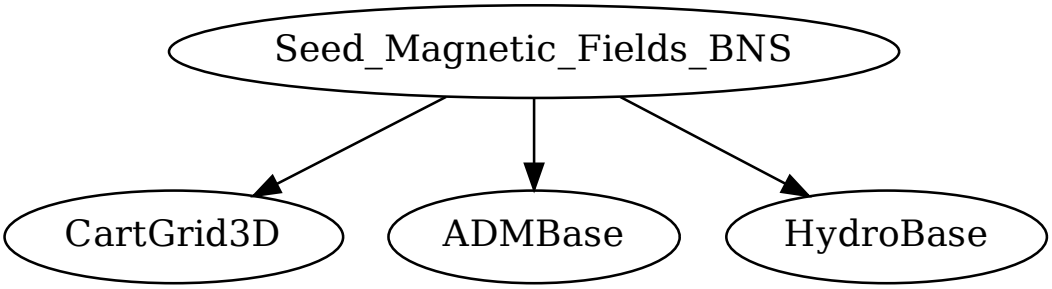
- Define A fields on an IllinoisGRMHD staggered grid?

```
>>> Seed_Magnetic_Fields::enable_IllinoisGRMHD_staggered_A_fields = "yes"
```

2.5.8 Seed_Magnetic_Fields_BNS

Thorn *Seed_Magnetic_Fields* set seeds magnetic fields within a single star. This thorn simply extends the capability to two stars, centered at $(x_1, 0, 0)$ and $(x_2, 0, 0)$ (LORENE sets up the neutron stars along the x-axis by default).

$$A_\phi = \varpi^2 A_b max[(P - P_{cut})^{n_s}, 0],$$



Parameter

- Magnetic field strength parameter.

```
>>> Seed_Magnetic_Fields_BNS::A_b = 1e-3
```

- Cutoff pressure, below which vector potential is set to zero. Typically set to 4% of the maximum initial pressure.

```
>>> Seed_Magnetic_Fields_BNS::P_cut = 1e-5
```

- Magnetic field strength pressure exponent.

```
>>> Seed_Magnetic_Fields_BNS::n_s = 1
```

- Define A fields on an IllinoisGRMHD staggered grid?

```
>>> Seed_Magnetic_Fields_BNS::enable_IllinoisGRMHD_staggered_A_fields = "no"
```

- Which field structure to use [“poloidal_A_interior”, “dipolar_A_everywhere”]:

```
>>> Seed_Magnetic_Fields_BNS::enable_IllinoisGRMHD_staggered_A_fields =
↪"yes" # This requires a staggered grid
>>> Seed_Magnetic_Fields_BNS::A_field_type = "poloidal_A_interior" #␣
↪interior to the star
>>> Seed_Magnetic_Fields_BNS::x_c1 = -15.2 # x coordinate of NS1 center
>>> Seed_Magnetic_Fields_BNS::x_c2 = 15.2 # x coordinate of NS2 center
>>> Seed_Magnetic_Fields_BNS::r_NS1 = 13.5 # Radius of NS1. Does not have␣
↪to be perfect, but must not overlap other star.
>>> Seed_Magnetic_Fields_BNS::r_NS2 = 13.5 # Radius of NS2
```

$$A_\phi = \varpi^2 A_b max[(P - P_{cut})^{n_s}, 0]$$

```
>>> Seed_Magnetic_Fields_BNS::enable_IllinoisGRMHD_staggered_A_fields =
↪"yes" # This requires a staggered grid
>>> Seed_Magnetic_Fields_BNS::A_field_type = "dipolar_A_everywhere"
>>> Seed_Magnetic_Fields_BNS::x_c1 = -15.2 # x coordinate of NS1 center
>>> Seed_Magnetic_Fields_BNS::x_c2 = 15.2 # x coordinate of NS2 center
>>> Seed_Magnetic_Fields_BNS::r_NS1 = 13.5 # Radius of NS1. Does not have␣
↪to be perfect, but must not overlap other star.
>>> Seed_Magnetic_Fields_BNS::r_NS2 = 13.5 # Radius of NS2
>>> Seed_Magnetic_Fields_BNS::r_zero_NS1 = 1.0 # Current loop radius
>>> Seed_Magnetic_Fields_BNS::r_zero_NS2 = 1.0
>>> Seed_Magnetic_Fields_BNS::I_zero_NS1 = 0.0 # Magnetic field loop␣
↪current of NS1
>>> Seed_Magnetic_Fields_BNS::I_zero_NS2 = 0.0
```

$$A_\phi = \frac{\pi r_0^2 I_0}{\left(r_0^2 + r^2\right)^{3/2}} \left(1 + \frac{15 r_0^2 \left(r_0^2 + \varpi^2\right)}{8 \left(r_0^2 + r^2\right)^2}\right)$$

2.5.9 TwoPunctures

Create initial for two puncture black holes using a single domain spectral method.

Following York's conformal-transverse-traceless decomposition method, we make the following assumptions for the metric and the extrinsic curvature

$$\gamma_{ij} = \psi^4 \delta_{ij}$$
$$K_{ij} = \psi^{-2} \left(V_{j,i} + V_{i,j} - \frac{2}{3}\delta_{ij} \operatorname{div} \boldsymbol{V}\right)$$

The initial data described by this method are conformally flat and maximally sliced, $K = 0$. With this ansatz the Hamiltonian constraint yields an equation for the conformal factor ψ

$$\triangle\psi + \frac{1}{8}\psi^5 K_{ij} K^{ij} = 0$$

while the momentum constraint yields an equation for the vector potential $\boldsymbol{V}$

$$\triangle \boldsymbol{V} + \frac{1}{3} \operatorname{grad}(\operatorname{div} \boldsymbol{V}) = 0$$

Note: TwoPunctures Thorn is restricted to problems involving two punctures.

One can proceed by choosing a non-trivial analytic solution of the Bowen-York type for the momentum constraint,

$$\boldsymbol{V} = \sum_{n=1}^{2} \left(-\frac{7}{4\left|\boldsymbol{x}_n\right|}\boldsymbol{P}_n - \frac{\boldsymbol{x}_n \cdot \boldsymbol{P}_n}{4\left|\boldsymbol{x}_n\right|^3}\boldsymbol{x}_n + \frac{1}{\left|\boldsymbol{x}_n\right|^3}\boldsymbol{x}_n \times \boldsymbol{S}_n \right)$$

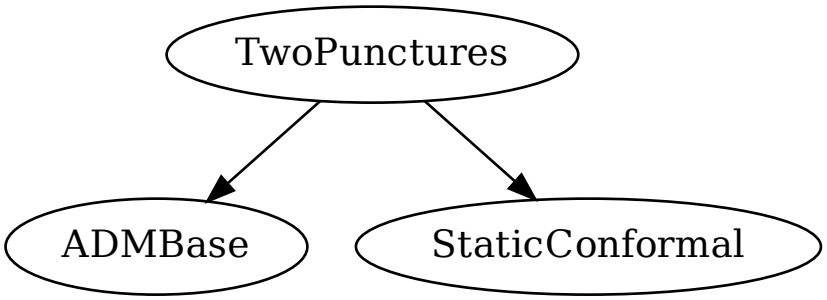
Hamiltonian constraint is obtained by writing the conformal factor ψ as a sum of a singular term and a finite correction u

$$\psi = 1 + \sum_{n=1}^{2} \frac{m_n}{2\left|\boldsymbol{x}_n\right|} + u$$

It is impossible to unambiguously define local black hole masses in general. In the following we choose the ADM mass

$$M_{\pm}^{ADM} = \left(1 + u_{\pm}\right) m_{\pm} + \frac{m_+ m_-}{2D}$$

Here m_+ and m_- are the values of u at each puncture.



Parameter

- initial_data

```
>>> ADMBase::initial_data  = "twopunctures"
>>> ADMBase::initial_lapse = "twopunctures-averaged"
>>> ADMBase::initial_shift   = "zero"
>>> ADMBase::initial_dtlapse = "zero"
>>> ADMBase::initial_dtshift = "zero"
```

- Coordinate of the puncture

```
>>> TwoPunctures::par_b            = 5.0
>>> TwoPunctures::center_offset[0] = -0.538461538462
```

- ADM mass of Black holes

```
>>> TwoPunctures::target_M_plus  = 0.553846153846
>>> TwoPunctures::target_M_minus = 0.446153846154
>>> TwoPunctures::adm_tol = 1.0e-10
```

(continues on next page)

(continued from previous page)

```
INFO (TwoPunctures): Attempting to find bare masses.
INFO (TwoPunctures): Target ADM masses: M_p=0.553846 and M_m=0.446154
INFO (TwoPunctures): ADM mass tolerance: 1e-10
INFO (TwoPunctures): Bare masses: mp=1, mm=1
INFO (TwoPunctures): ADM mass error: M_p_err=0.500426421474965, M_m_err=0.
↪607857653302066
. . .
INFO (TwoPunctures): Bare masses: mp=0.518419372531011, mm=0.391923877275946
INFO (TwoPunctures): ADM mass error: M_p_err=2.35933494963092e-12, M_m_err=8.
↪5276896655273e-11
INFO (TwoPunctures): Found bare masses.
>>> TwoPunctures::target_M_plus  = 0.553846153846
>>> TwoPunctures::target_M_minus = 0.446153846154
>>> TwoPunctures::par_m_plus  = 0.553846153846
>>> TwoPunctures::par_m_minus = 0.446153846154
>>> TwoPunctures::adm_tol = 1.0e-10
INFO (TwoPunctures): Attempting to find bare masses.
INFO (TwoPunctures): Target ADM masses: M_p=0.553846 and M_m=0.446154
INFO (TwoPunctures): ADM mass tolerance: 1e-10
INFO (TwoPunctures): Bare masses: mp=0.553846153846, mm=0.446153846154
INFO (TwoPunctures): ADM mass error: M_p_err=0.0334459078036595, M_m_err=0.
↪0445419016377125
. . .
INFO (TwoPunctures): Bare masses: mp=0.518419372531011, mm=0.391923877275946
INFO (TwoPunctures): ADM mass error: M_p_err=2.35933494963092e-12, M_m_err=8.
↪5276896655273e-11
INFO (TwoPunctures): Found bare masses.
```

- momentum of the puncture

```
>>> TwoPunctures::par_P_plus [1] = +0.3331917498
>>> TwoPunctures::par_P_minus[1] = -0.3331917498
```

- spin of the puncture

```
>>> TwoPunctures::par_S_plus [1] = 0.0
>>> TwoPunctures::par_S_minus[1] = 0.0
```

- A small number to smooth out singularities at the puncture locations

```
>>> TwoPunctures::TP_epsilon = 1e-6
```

- Tiny number to avoid nans near or at the pucture locations

```
>>> TwoPunctures::TP_Tiny    = 1.0e-2
```

- Print screen output while solving

```
>>> TwoPunctures::verbose = yes
INFO (TwoPunctures): Bare masses: mp=0.553846153846, mm=0.446153846154
Newton: it=0         |F|=7.738745e-02
bare mass: mp=0.553846       mm=0.446154
bicgstab:  itmax 100, tol 7.738745e-05
bicgstab:     0   6.428e-01
bicgstab:     1   1.010e+00   1.021e+00   0.000e+00   6.116e-01
bicgstab:     2   7.551e-02   1.622e+00   1.531e-02   4.085e-01
```

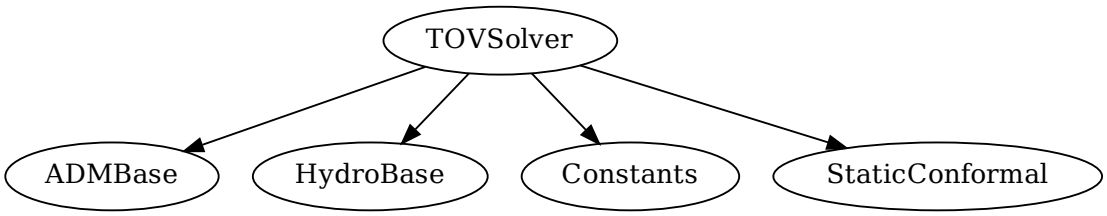
(continues on next page)

(continued from previous page)

```
bicgstab:     3   1.561e-02   2.836e-01   2.396e-02   8.846e-01
bicgstab:     4   7.358e-03   2.473e-01  -1.079e-01   9.778e-01
bicgstab:     5   3.429e-04   9.104e+00  -7.954e-01   4.003e-01
bicgstab:     6   6.564e-05   3.724e-01  -4.164e-01   1.293e+00
Newton: it=1          |F|=1.149396e-03
INFO (TwoPunctures): ADM mass error: M_p_err=0.0334459078036595, M_m_err=0.
↪0445419016377125
>>> TwoPunctures::verbose = no
INFO (TwoPunctures): Bare masses: mp=0.553846153846, mm=0.446153846154
INFO (TwoPunctures): ADM mass error: M_p_err=0.0334459078036595, M_m_err=0.
↪0445419016377125
```

2.5.10 TOVSolver

This thorn provides initial data for TOV star(s) in isotropic coordinates. The Tolman-Oppenheimer-Volkoff solution is a static perfect fluid "star".



Parameter

- TOV star initial data

```
>>> ADMBase::initial_data            = "tov"
>>> ADMBase::initial_lapse           = "tov"
>>> ADMBase::initial_shift           = "tov"
>>> ADMBase::initial_dtlapse         = "zero"
>>> ADMBase::initial_dtshift         = "zero"
```

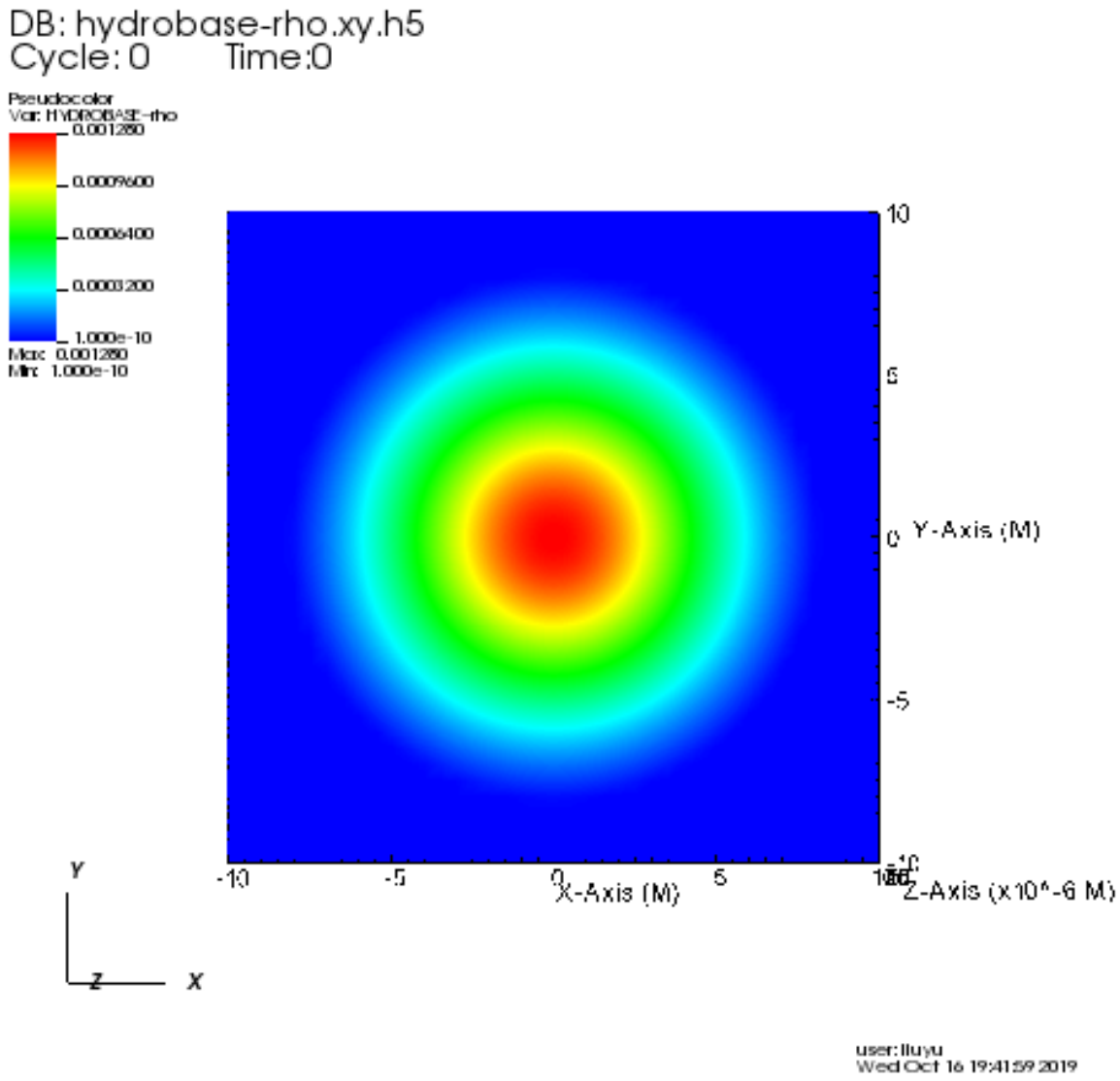
- Set up a TOV star described by a polytropic equation of state $p = K\rho^{\Gamma}$

```
>>> TOVSolver::TOV_Rho_Central[0] = 1.28e-3
>>> TOVSolver::TOV_Gamma           = 2.0
>>> TOVSolver::TOV_K               = 100.0
```

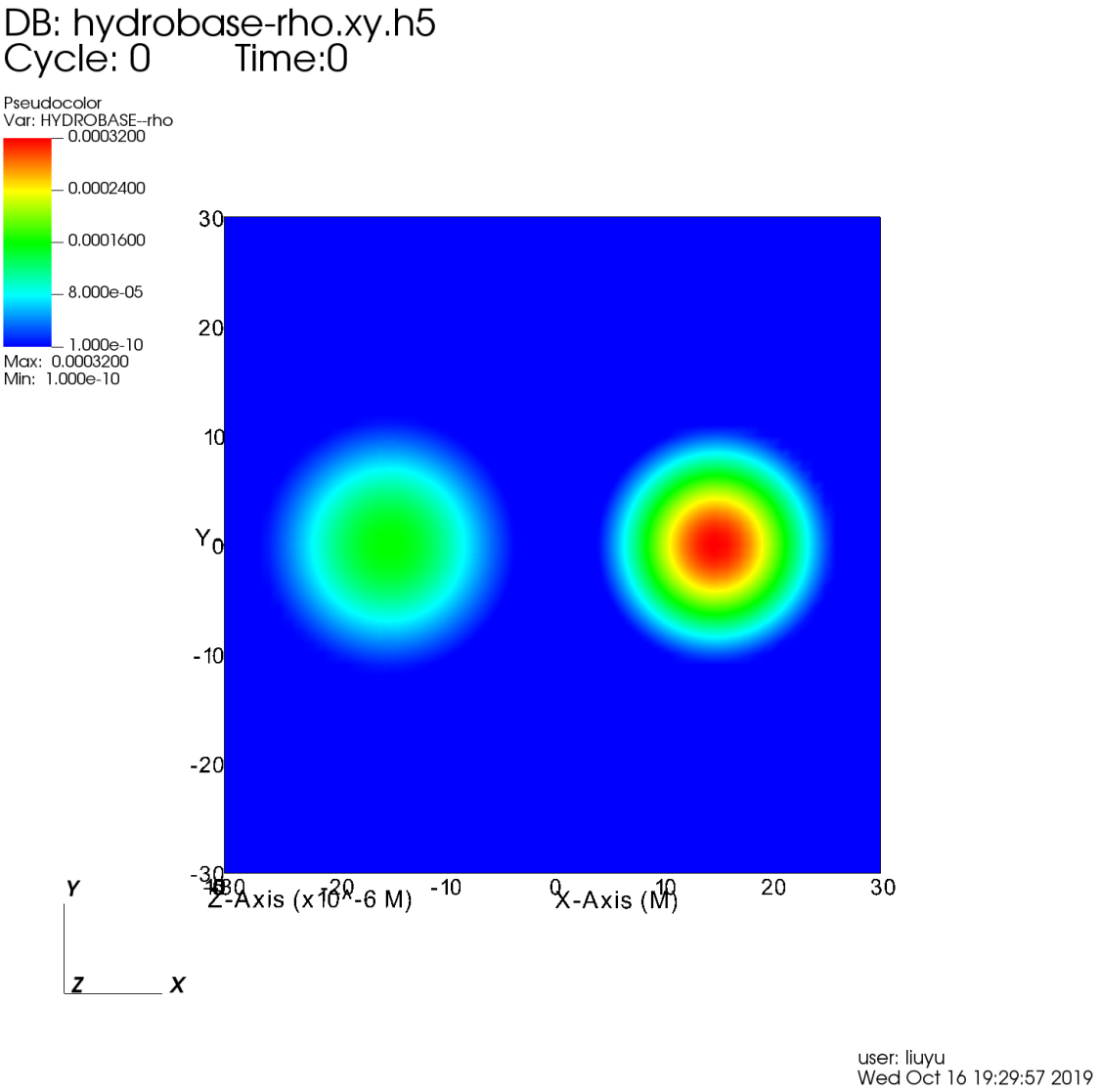
- Velocity of neutron star

```
>>> TOVSolver::TOV_Velocity_x[0]  = 0.1
>>> TOVSolver::TOV_Velocity_y[0]  = 0.2
>>> TOVSolver::TOV_Velocity_z[0]  = 0.3
```

- Two or more of TOVs


DB: hydrobase-rho.xy.h5
Cycle: 0 Time:0
Pseudocolor
Var: HYDROBASE-rho
0.001280
0.0009600
0.0006400
0.0003200
1.000e-10
Max: 0.001280
Min: 1.000e-10
10
5
0
-5
-10
Y-Axis (M)
-10
-5
0
5
X-Axis (M)
Z-Axis (x10^-6 M)
Y
Z
X
user: lluyu
Wed Oct 16 19:41:59 2019

```
>>> Tovsolver::TOV_Num_TOVs        = 2
>>> Tovsolver::TOV_Num_Radial      = 200000
>>> Tovsolver::TOV_Combine_Method = "average"
>>> Tovsolver::TOV_Rho_Central[0] = 0.16e-3
>>> Tovsolver::TOV_Position_x[0]  = -15.0
>>> Tovsolver::TOV_Rho_Central[1] = 0.32e-3
>>> Tovsolver::TOV_Position_x[1]  = 15.0
```



2.5.11 Exact

All of these exact spacetimes have been found useful for testing different aspect of the code.

This thorn sets up the 3+1 ADM variables for any of a number of exact spacetimes/coordinates. Optionally, any 4-metric can be Lorentz-boosted in any direction. As another option, the ADM variables can be calculated on an arbitrary slice through the spacetime, using arbitrary coordinates on the slice.

Parameter

- The exact solution/coordinates
- By default, this thorn sets up the ADM variables on an initial slice only. However, setting ADMBase::evolution_method so you get an exact spacetime, not just a single slice.

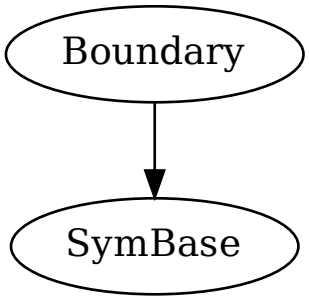
```
>>> ADMBase::evolution_method = "exact"
```

2.6 Boundary

2.6.1 Boundary

Provides a generic interface to boundary conditions, and provides a set of standard boundary conditions for one, two, and three dimensional grid variables.

- scalar boundary conditions
- flat boundary conditions
- radiation boundary conditions
- copy boundary conditions
- Robin boundary conditions
- static boundary conditions



Parameter

- Register routine

```
>>> Boundary::register_scalar = "yes"
>>> Boundary::register_flat = "yes"
>>> Boundary::register_radiation = "yes"
>>> Boundary::register_copy = "yes"
>>> Boundary::register_robin = "yes"
```

(continues on next page)

(continued from previous page)

```
>>> Boundary::register_static = "yes"
>>> Boundary::register_none = "yes"
```

Warning

- The aliased function 'SymmetryTableHandleForGrid' (required by thorn 'Boundary') has not been provided by any active thorn !

```
>>> ActiveThorns = "SymBase"
```

2.7 Evolution

2.7.1 Time

Calculates the timestep used for an evolution by either

- setting the timestep directly from a parameter value
- using a Courant-type condition to set the timestep based on the grid-spacing used.

Parameter

- The standard timestep condition dt = dtfac*max(delta_space)

```
>>> grid::dxyz = 0.3
>>> time::dtfac = 0.1
---------------------------------
   it   |          | WAVETOY::phi |
        |    t     | norm2        |
---------------------------------
     0  |    0.000 |   0.10894195 |
     1  |    0.030 |   0.10892065 |
     2  |    0.060 |   0.10885663 |
     3  |    0.090 |   0.10874996 |
```

- Absolute value for timestep

```
>>> time::timestep_method = "given"
>>> time::timestep = 0.1
---------------------------------
   it   |          | WAVETOY::phi |
        |    t     | norm2        |
---------------------------------
     0  |    0.000 |   0.10894195 |
     1  |    0.100 |   0.10870525 |
     2  |    0.200 |   0.10799700 |
     3  |    0.300 |   0.10682694 |
>>> time::timestep_method = "given"
>>> time::timestep = 0.2
---------------------------------
   it   |          | WAVETOY::phi |
```

(continues on next page)

(continued from previous page)

```
       |    t      | norm2        |
-----------------------------------
     0 |    0.000 |   0.10894195 |
     1 |    0.200 |   0.10799478 |
     2 |    0.400 |   0.10520355 |
     3 |    0.600 |   0.10072358 |
```

2.7.2 MoL

The Method of Lines (MoL) converts a (system of) partial differential equation(s) into an ordinary differential equation containing some spatial differential operator. As an example, consider writing the hyperbolic system of PDE's

$$\partial_t q + A^i(q)\partial_i B(q) = s(q)$$

in the alternative form

$$\partial_t q = L(q)$$

Given this separation of the time and space discretizations, well known stable ODE integrators such as Runge-Kutta can be used to do the time integration.

Parameter

- chooses between the dierent methods.

```
>>> MoL::ODE_Method = "RK4"
INFO (MoL): Using Runge-Kutta 4 as the time integrator.
```

- controls the number of intermediate steps for the ODE solver. For the generic Runge-Kutta solvers it controls the order of accuracy of the method.

```
>>> MoL::MoL_Intermediate_Steps = 4
```

- controls the amount of scratch space used.

```
>>> MoL::MoL_Num_Scratch_Levels = 1
```

Warning

- When using the efficient RK4 evolver the number of intermediate steps must be 4, and the number of scratch levels at least 1.

```
>>> MoL::MoL_Intermediate_Steps = 4
>>> MoL::MoL_Num_Scratch_Levels = 1
```

2.7.3 Dissipation

Add fourth order Kreiss-Oliger dissipation to the right hand side of evolution equations.

The additional dissipation terms appear as follows

$$\partial_t \boldsymbol{U} = \partial_t \boldsymbol{U} + (-1)^{(p+3)/2} \epsilon \frac{1}{2^{p+1}} \left(h_x^p \frac{\partial^{(p+1)}}{\partial x^{(p+1)}} + h_y^p \frac{\partial^{(p+1)}}{\partial y^{(p+1)}} + h_z^p \frac{\partial^{(p+1)}}{\partial z^{(p+1)}} \right) \boldsymbol{U}$$

where h_x, h_y, and h_z are the local grid spacings in each Cartesian direction. ϵ is a positive, adjustable parameter controlling the amount of dissipation added, and must be less that 1 for stability.

Parameter

- Dissipation order and strength

```
>>> Dissipation::order = 5
>>> Dissipation::epsdis = 0.1
```

Note: Currently available values of order are $p \in \{1, 3, 5, 7, 9\}$. To apply dissipation at order p requires that we have at least $(p + 1)/2$ ghostzones respectively.

- List of evolved grid functions that should have dissipation added

```
>>> Dissipation::vars = "ML_BSSN::ML_log_confac
                         ML_BSSN::ML_metric
                         ML_BSSN::ML_trace_curv
                         ML_BSSN::ML_curv
                         ML_BSSN::ML_Gamma
                         ML_BSSN::ML_lapse
                         ML_BSSN::ML_shift
                         ML_BSSN::ML_dtlapse
                         ML_BSSN::ML_dtshift"
```

2.7.4 ML_BSSN

The code is designed to handle arbitrary shift and lapse conditions. Gauges are the commonly used $1 + log$ and $\tilde{\Gamma}$-driver conditions with advection terms.

The hyperbolic K-driver slicing conditions have the form

$$\left(\partial_t - \beta^i \partial_i\right) \alpha = -f(\alpha) \alpha^2 \left(K - K_0\right)$$

The hyperbolic Gamma-driver condition have the form

$$\partial_t^2 \beta^i = F \partial_t \tilde{\Gamma}^i - \eta \partial_t \beta^i.$$

where F and η are, in general, positive functions of space and time. We typically choose $F = 3/4$. For some reason, a simple space-varying prescription for η is implemented

$$\eta(r) := \frac{2}{M_{TOT}} \begin{cases} 1 & \text{for } r \leq R \text{ (near the origin)} \\ \frac{R}{r} & \text{for } r \geq R \text{ (far away)} \end{cases}$$

This is a generalization of many well known slicing and shift conditions.

Parameter

- Evolution method

```
>>> ADMBase::evolution_method         = "ML_BSSN"
>>> ADMBase::lapse_evolution_method   = "ML_BSSN"
>>> ADMBase::shift_evolution_method   = "ML_BSSN"
>>> ADMBase::dtlapse_evolution_method = "ML_BSSN"
>>> ADMBase::dtshift_evolution_method = "ML_BSSN"
```

- K-driver slicing conditions: $\frac{d\alpha}{dt} = -f\alpha^n K$

```
>>> ML_BSSN::harmonicN = 1
>>> ML_BSSN::harmonicF = 2.0
[1+log slicing condition]
```

- Gamma-driver condition: F

```
>>> ML_BSSN::useSpatialShiftGammaCoeff = 0
>>> ML_BSSN::ShiftGammaCoeff = <F>
```

$$F(r) = F$$

```
>>> ML_BSSN::useSpatialShiftGammaCoeff = 1
>>> ML_BSSN::ShiftGammaCoeff = <F>
>>> ML_BSSN::spatialShiftGammaCoeffRadius = 50
```

$$F(r) = Min[1, e^{1-\frac{r}{R}}] \times F$$

- Gamma-driver condition: η

```
>>> ML_BSSN::useSpatialBetaDriver = 0
>>> ML_BSSN::BetaDriver = <eta>
```

$$\eta(r) = \eta$$

```
>>> ML_BSSN::useSpatialBetaDriver = 1
>>> ML_BSSN::BetaDriver = <eta>
>>> ML_BSSN::spatialBetaDriverRadius = <R>
```

$$\eta(r) = \frac{R}{Max[r, R]} \times \eta$$

- Enable spatially varying betaDriver

```
>>> ML_BSSN::useSpatialBetaDriver = 1
```

- Advect Lapse and shift?

```
>>> ML_BSSN::advectLapse = 1
>>> ML_BSSN::advectShift = 1
```

- Boundary condition for BSSN RHS and some of the ADMBase variables.

```
>>> ML_BSSN::rhs_boundary_condition = "scalar"
```

- Whether to apply dissipation to the RHSs

```
>>> ML_BSSN::epsDiss = 0.0
>>>
>>> Dissipation::epsdis = 0.1
>>> Dissipation::order = 5
>>> Dissipation::vars = "ML_BSSN::ML_log_confac
>>>                      ML_BSSN::ML_metric
>>>                      ML_BSSN::ML_trace_curv
>>>                      ML_BSSN::ML_curv
>>>                      ML_BSSN::ML_Gamma
>>>                      ML_BSSN::ML_lapse
>>>                      ML_BSSN::ML_shift
>>>                      ML_BSSN::ML_dtlapse
>>>                      ML_BSSN::ML_dtshift"
```

- Enforced minimum of the lapse function

```
>>> ML_BSSN::MinimumLapse = 1.0e-8
```

- Finite differencing order

```
>>> ML_BSSN::fdOrder = 4
```

Warning

- Insufficient ghost or boundary points for ML_BSSN_InitialADMBase2Interior

```
>>> ML_BSSN::fdOrder = 8
>>> driver::ghost_size = 5
>>> CoordBase::boundary_size_x_lower = 5
>>> CoordBase::boundary_size_y_lower = 5
>>> CoordBase::boundary_size_z_lower = 5
>>> CoordBase::boundary_size_x_upper = 5
>>> CoordBase::boundary_size_y_upper = 5
>>> CoordBase::boundary_size_z_upper = 5
```

- Range error setting parameter ‘ML_BSSN::initial_boundary_condition’ to ‘extrapolate-gammas’

```
>>> ActiveThorns = "ML_BSSN_Helper CoordGauge"
```

2.7.5 ML_BSSN_Helper

Warning

- The function ExtrapolateGammas has not been provided by any active thorn.

```
>>> ActiveThorns = "NewRad"
```

2.7.6 illinoisGRMHD

IllinoisGRMHD solves the equations of General Relativistic MagnetoHydroDynamics (GRMHD) using a high-resolution shock capturing scheme and the Piecewise Parabolic Method (PPM) for reconstruction.

IllinoisGRMHD evolves the vector potential A_μ (on staggered grids) instead of the magnetic fields (B^i) directly, to guarantee that the magnetic fields will remain divergenceless even at AMR boundaries.

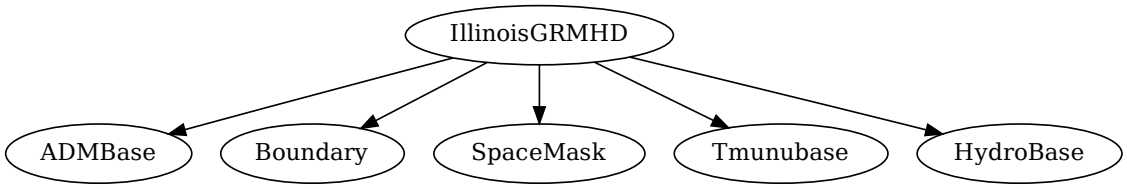
IllinoisGRMHD currently implements a hybrid EOS of the form

$$P\left(\rho_0, \epsilon\right) = P_{\text{cold}}\left(\rho_0\right) + \left(\Gamma_{\text{th}} - 1\right)\rho_0\left[\epsilon - \epsilon_{\text{cold}}\left(\rho_0\right)\right]$$

where P_{cold} and ϵ_{cold} denote the cold component of P amd ϵ respectively, and Γ_{th} is a constant parameter which determines the conversion efficiency of kinetic to thermal energy at shocks. The function $\epsilon_{\text{cold}}\left(\rho_0\right)$ is related to $P_{\text{cold}}\left(\rho_0\right)$ by the first law of thermodynamics,

$$\epsilon_{\text{cold}}\left(\rho_0\right) = \int \frac{P_{\text{cold}}\left(\rho_0\right)}{\rho_0^2} d\rho_0$$

The Γ-law EOS $P = (\Gamma - 1)\rho_0\epsilon$ is adopted. This corresponds to setting $P_{\text{cold}} = (\Gamma - 1)\rho_0\epsilon_{\text{cold}}$, which is equivalent to $P_{\text{cold}} = \kappa\rho_0^\Gamma$ (with constant κ), and $\Gamma_{\text{th}} = \Gamma$. In the absence of shocks and in the initial data $\epsilon = \epsilon_{\text{cold}}$ and $P = P_{\text{cold}}$



Parameter

- Determines how much evolution information is output

```
>>> IllinoisGRMHD::verbose = "no"
```

```
>>> IllinoisGRMHD::verbose = "essential"
```

```
>>> IllinoisGRMHD::verbose = "essential+iteration output"
```

- Floor value on the energy variable tau and the baryonic rest mass density

```
>>> IllinoisGRMHD::tau_atm
>>> IllinoisGRMHD::rho_b_atm
```

- Hybrid EOS

```
>>> IllinoisGRMHD::gamma_th = 2.0
>>> IllinoisGRMHD::K_poly =
```

- Chosen Matter and EM field boundary condition

```
>>> IllinoisGRMHD::EM_BC = "outflow"
>>> IllinoisGRMHD::Matter_BC = "copy"
```

```
>>> IllinoisGRMHD::EM_BC = "frozen"
>>> IllinoisGRMHD::Matter_BC = "frozen" #If Matter_BC or EM_BC is set to FROZEN,␣
↪BOTH must be set to frozen!
```

- Debug. If the primitives solver fails, this will output all data needed to debug where and why the solver failed.

```
>>> IllinoisGRMHD::conserv_to_prims_debug = 1
```

2.7.7 ID_conerter_ILGRMHD

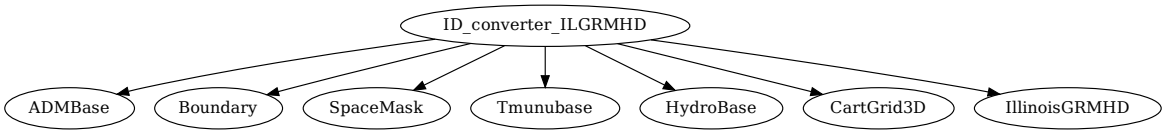
IllinoisGRMHD and HydroBase variables are incompatible. The former uses 3-velocity defined as $v^i = u^i/u^0$, and the latter uses the Valencia formalism definition of v^i as measured by normal observers, defined as:

$$v^i_{(n)} = \frac{u^i}{\alpha u^0} + \frac{\beta^i}{\alpha}$$

In addition, IllinoisGRMHD needs the A-fields to be defined on *staggered* grids, and HydroBase does not yet support this option.

Table 1. Storage location on grid of the magnetic field B^i and vector potential $\mathcal{A}_\mu$. Note that $\boldsymbol{P}$ is the vector of primitive variables $\{\rho_0, P, v^i\}$

Variable(s)	storage location
Metric terms, $\boldsymbol{P}$, ρ_*, $\tilde{S}_i$, $\tilde{\tau}$	(i,j,k)
B^x, $\tilde{B}^x$	$(i+\frac{1}{2},j,k)$
B^y, $\tilde{B}^y$	$(i,j+\frac{1}{2},k)$
B^z, $\tilde{B}^z$	$(i,j,k+\frac{1}{2})$
A_x	$(i,j+\frac{1}{2},k+\frac{1}{2})$
A_y	$(i+\frac{1}{2},j,k+\frac{1}{2})$
A_z	$(i+\frac{1}{2},j+\frac{1}{2},k)$
$\sqrt{\gamma}\Phi$	$(i+\frac{1}{2},j+\frac{1}{2},k+\frac{1}{2})$



Parameter

- Single Gamma-law EOS:

```
>>> ID_converter_ILGRMHD::Gamma_Initial = 2.0
>>> ID_converter_ILGRMHD::K_Initial     = 123.64110496340211
```

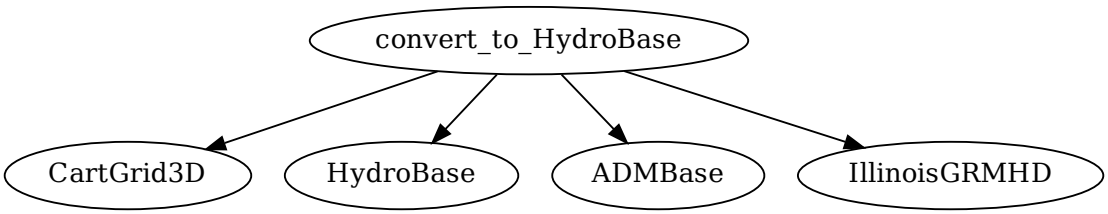
2.7.8 convert_to_HydroBase

Convert IllinoisGRMHD-compatible variables to HydroBase-compatible variables. Used for compatibility with HydroBase/ADMBase analysis thorns in the Einstein Toolkit.

Parameter

- How often to convert IllinoisGRMHD primitive variables to HydroBase primitive variables? This needed for particle tracer.

```
>>> convert_to_HydroBase::convert_to_HydroBase_every = 0
```



2.8 Analysis

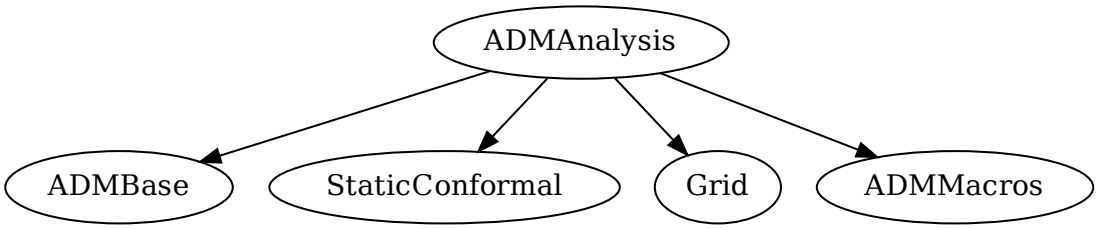
2.8.1 ADMAnalysis

This thorn does basic analysis of the metric and extrinsic curvature tensors.

It calculates

- the trace of the extrinsic curvature (trK)
- the determinant of the metric (detg)
- the components of the metric in spherical coordinates (grr,grq,grp,gqq,gqp,gpp)
- the components of the extrinsic curvature in spherical coordinates (Krr,Krq,Krp,Kqq,Kqp,Kpp)
- components of the Ricci tensor
- the Ricci scalar

(q refers to the theta coordinate and p to the phi coordinate.)



Parameter

- Project angular components onto $r * dtheta$ and $r * sin(theta) * dphi$

```
>>> ADMAnalysis::normalize_dtheta_dphi = "yes"
```

Warning

- Cannot output variable because it has no storage

Output

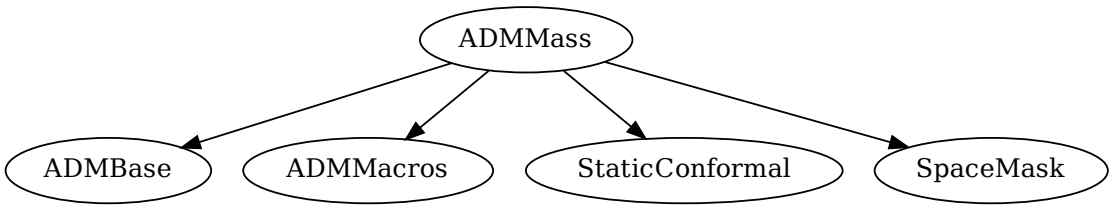
- The Ricci scalar

```
>>> IOHDF5::out2D_vars = "ADMAnalysis::ricci_scalar"
```

2.8.2 ADMMass

Thorn ADMMass can compute the ADM mass from quantities in ADMBase.

Note: ADMMass uses the ADMMacros for derivatives. Thus, converegence of results is limited to the order of these derivatives (ADMMacros::spatial_order).



Parameter

- Number of measurements

```
>>> ADMMass::ADMMass_number = 1
```

Note: Multiple measurements can be done for both volume and surface integral, but the limit for both is 100 (change param.ccl if you need more).

- ADMMass provides several possibilities to specify the (nite) integration domain for surface integral

```
>>> ADMMass::ADMMass_surface_distance[0] = 12
```

- ADMMass provides several possibilities to specify the (nite) integration domain for volume integral

```
>>> ADMMass::ADMMass_use_all_volume_as_volume_radius = "yes"
Use the whole grid for volume integration
>>> ADMMass::ADMMass_use_surface_distance_as_volume_radius = "yes"
ADMMass_surface_distance is used to specify the integration radius.
>>> ADMMass::ADMMass_volume_radius[0] =12
ADMMass_volume_radius species this radius.
```

- Should we exclude the region inside the AH to the volume integral?

```
>>> ADMMass::ADMMass_Excise_Horizons = "yes"
```

Warning

- WARNING[L2,P0] (ADMMass): radius < 0 / not set, not calculating the volume integral to get the ADM mass.

```
>>> ADMMass::ADMMass_surface_distance[0] = 12
```

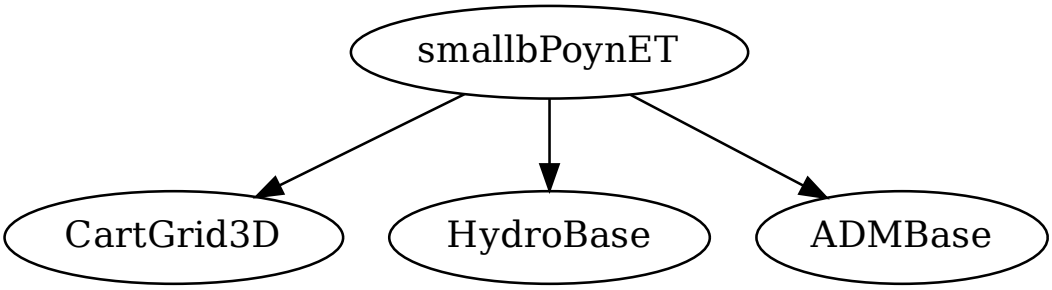
Output

- The results for the volume integral, the usual surface integral and the sorface integral including the lapse

```
>>> IOASCII::out0D_vars  = "ADMMass::ADMMass_Masses"
```

2.8.3 smallbPoynET

Using HydroBase velocity and magnetic field variables, as well as ADMBase spacetime metric variables, smallbPoynET computes b^i, and three spatial components of Poynting flux. It also computes $-1 - u_0$, which is useful for tracking unbound matter.



Parameter

- How often to compute smallbPoyn?

```
>>> smallbPoynET::smallbPoynET_compute_every = 0
```

Output

- Poynting flux

```
>>>
```

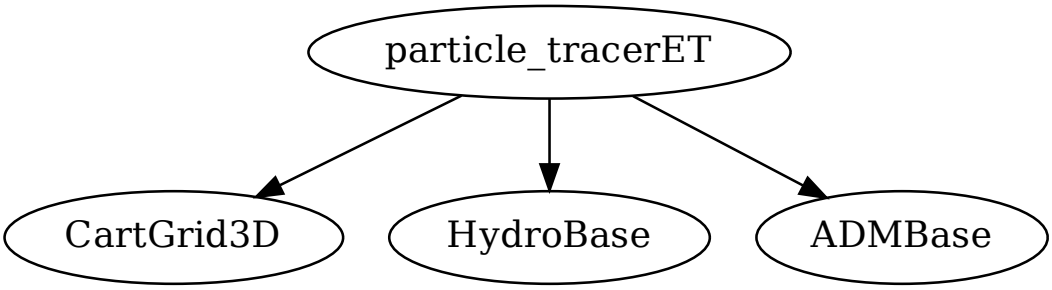
- b^i

```
>>> CarpetIOHDF5::out2D_vars = "smallbPoynET::smallb2
                                smallbPoynET::smallbx
                                smallbPoynET::smallby
                                smallbPoynET::smallbz"
```

- ```
  >>> CarpetIOHDF5::out2D_vars = "smallbPoynET::minus_one_minus_u_0"
  ```

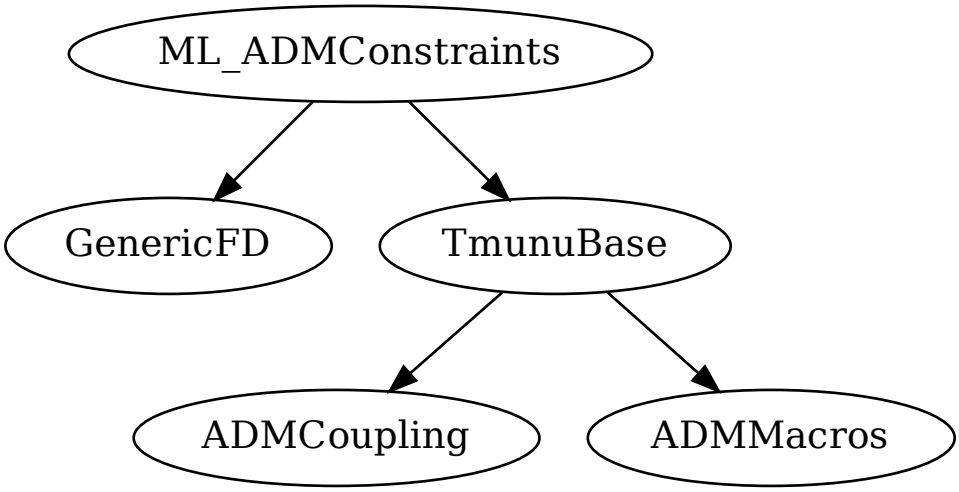
2.8.4 particle_tracerET

This thorn provides essential functionality for self-consistent visualizations of the dynamics of magnetic field lines over time in a GRMHD simulation.
  ```



2.8.5 ML_ADMConstraints

ML_ADMConstraints calculates the ADM constraints violation, but directly using potentially higher-order derivatives, and is, in general, preferred over ADMConstraints.



Output

- Hamiltonian constraint

```
>>> IOHDF5::out2D_vars = "ML_ADMConstraints::ML_Ham"
```

- Momentum constraints

```
>>> IOHDF5::out2D_vars = "ML_ADMConstraints::ML_mom"
```

2.8.6 QuasiLocalMeasures

Calculate quasi-local measures such as masses, momenta, or angular momenta and related quantities on closed two-dimentional surfaces, including on horizons.

Parameter

- Input a surface that the user specifies and can calculate useful quantities

```
>>> QuasiLocalMeasures::num_surfaces   = 1
>>> QuasiLocalMeasures::spatial_order  = 4
>>> QuasiLocalMeasures::interpolator = "Lagrange polynomial interpolation"
>>> QuasiLocalMeasures::interpolator_options = "order=4"
>>> QuasiLocalMeasures::surface_name[0] = "waveextract surface at 100"
```

Output

- Scalar quantities on the surface

```
>>> IOASCII::out0D_vars  = "QuasiLocalMeasures::qlm_scalars"
```

2.8.7 AHFinderDirect

In 3+1 numerical relativity, it's often useful to know the positions and shapes of any black holes in each slice.

Finding Apparent Horizons in a numerical spacetime. It calulates various quantities like horizon area and its corresponding mass.

Note: The main complication here is that AHFinderDirect needs an initial guess for an AH shape, and if this initial guess is inaccurate AHFinderDirect may fail to find the AH.

Parameter

- How often should we try to find apparent horizons?

```
>>> AHFinderDirect::find_every = 128 # every course
```

- Number of apparent horizons to search for

```
>>> AHFinderDirect::N_horizons = 2
```

- Move the origins with the horizons

```
>>> AHFinderDirect::move_origins = yes
```

- Which surface should we store the info?

```
>>> AHFinderDirect::origin_x [1] =
>>> AHFinderDirect::initial_guess__coord_sphere__x_center[1] =
>>> AHFinderDirect::initial_guess__coord_sphere__radius [1] =
>>> AHFinderDirect::which_surface_to_store_info [1] = 2
>>> AHFinderDirect::track_origin_source_x        [1] = "PunctureTracker::pt_loc_
↪x[0]"
>>> AHFinderDirect::track_origin_source_y        [1] = "PunctureTracker::pt_loc_
↪y[0]"
>>> AHFinderDirect::track_origin_source_z        [1] = "PunctureTracker::pt_loc_
↪z[0]"
>>> AHFinderDirect::max_allowable_horizon_radius [1] = 3
```

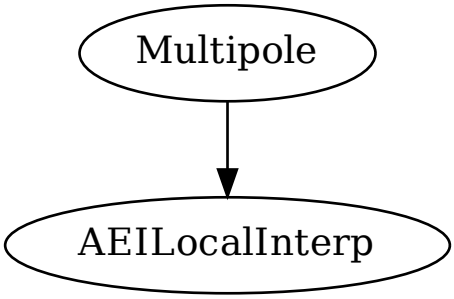
2.8.8 Multipole

Multipole thorn can decompose multiple grid functions with any spin-weight on multiple spheres. A set of radii for these spheres, as well as the number of angular points to use, can be specied.

The angular dependence of a field $u(t, r, \theta, \varphi)$ can be expanded in spin-weight s spherical harmonics

$$u(t, r, \theta, \varphi) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} C^{lm}(t, r)_s Y_{lm}(\theta, \varphi)$$

where the coefficients $C^{lm}(t, r)$ are given by



Parameter

- Decide the number and radii of the coordinate spheres on which you want to decompose.

```
>>> Multipole::nradii    = 3
>>> Multipole::radius[0] = 10
>>> Multipole::radius[1] = 20
>>> Multipole::radius[2] = 30
>>> Multipole::variables = "MyThorn::u"
```

- How many points in the theta and phi direction?

```
>>> Multipole::ntheta = 120
>>> Multipole::nphi   = 240
```

- The maximum l mode to extract

```
>>> Multipole::l_max = 8
```

- Output an HDF5 file for each variable containing one dataset per mode at each radius

```
>>> Multipole::output_hdf5  = yes
```

2.8.9 WeylScal4

Calculate the Weyl Scalars for a given metric given the fiducial tetrad.

Parameter

- Finite differencing order

```
>>> WeylScal4::fdOrder = 8
```

- Which scalars to calculate

```
>>> WeylScal4::calc_scalars = "psis"
```

- Compute invariants

```
>>> WeylScal4::calc_invariants = "always"
```

2.9 Numerical

Provides numerical infrastructure thorns.

2.9.1 SpaceMask

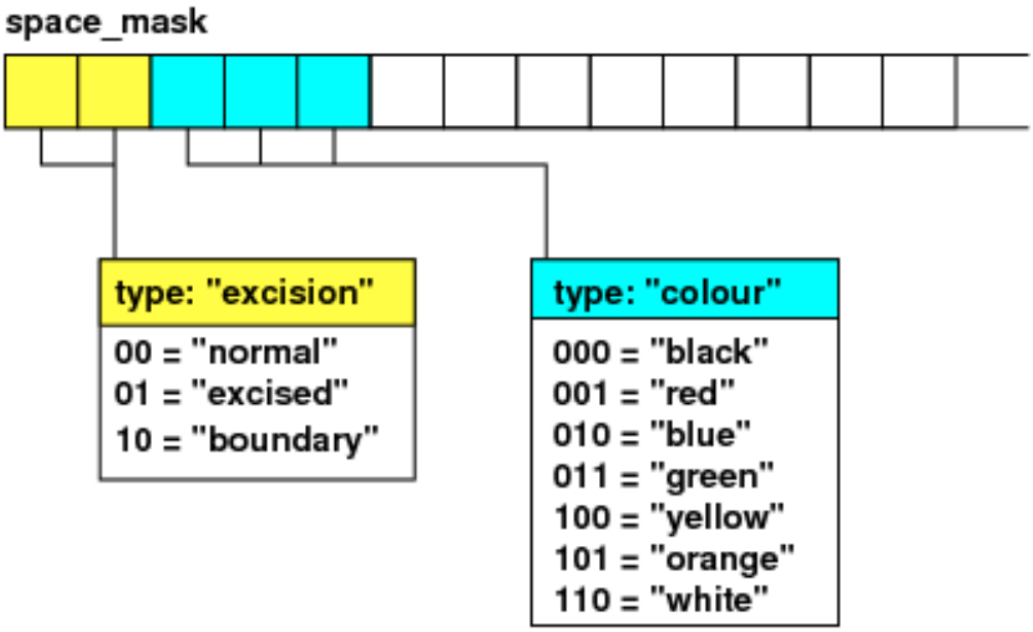
The mask is a grid function which can be used to assign a state to each point on the grid. It is used as a bit eld, with different bits, or combinations of bits, assigned to represent various states.

For instance, a programmer wants to record whether each point of the grid has state “interior”, “excised” or “boundary”. 2-bits of the mask (enough to represent the three possible states of the type) are allocated to hold this information. If a new type is required, bits are allocated to it from the remaining free bits in the mask.

Parameter

- Turn on storage for mask

```
>>> SpaceMask::use_mask = "yes"
```



2.9.2 SphericalSurface

SphericalSurface defines two-dimensional surfaces with spherical topology. The thorn itself only acts as a repository for other thorns to set and retrieve such surfaces, making it a pure infrastructure thorn. One thorn can update a given spherical surface with information, and another thorn can read that information without having to know about the first thorn.

Parameter

- Number of surfaces

```
>>> SphericalSurface::nsurfaces = 2
```

- Surface Definition: Maximum number of grid points in the theta amd phi direction

```
>>> SphericalSurface::maxntheta = 39
>>> SphericalSurface::maxnphi   = 76
```

- Surface Definition and set resolution according to given parameters. Some of spherical surface index may be used by PunctureTracker.

```
>>> SphericalSurface::name         [0] = "Righthand NS"
>>> SphericalSurface::ntheta       [0] = 39 # must be at least 3*nghoststheta
>>> SphericalSurface::nphi         [0] = 76 # must be at least 3*nghostsphi
>>> SphericalSurface::nghoststheta[0] = 2
>>> SphericalSurface::nghostsphi   [0] = 2
```

- Place surface at a certain radius

```
>>> SphericalSurface::set_spherical[0] = yes
>>> SphericalSurface::radius       [0] = 250
```

2.10 Utility

2.10.1 NaNChecker

The NaNChecker thorn can be used to analyze Cactus grid variables (that is grid functions, arrays or scalars) for NaN (Not-a-Number) and innite values.

Parameter

- How often to check for NaNs

```
>>> NaNChecker::check_every = 128
```

- Groups and/or variables to check for NaNs

```
>>> NaNChecker::check_vars = "all" # List of full group and/or variable names, or
↪'all' for everything
[1mWARNING level 1 from host ubuntu process 0
while executing schedule bin NaNChecker_NaNCheck, routine NaNChecker::NaNChecker_
↪NaNCheck_Check
in thorn NaNChecker, file /home4/yuliu/Cactus/arrangements/CactusUtils/NaNChecker/
↪src/NaNCheck.cc:875:
->[0m There were 142 NaN/Inf value(s) found in variable 'HYDROBASE::rho'
```

- What to do if a NaN was found

```
>>> NaNChecker::action_if_found = "terminate"
[1mWARNING level 1 from host ubuntu process 0
while executing schedule bin CCTK_POSTSTEP, routine NaNChecker::NaNChecker_
↪TakeAction
in thorn NaNChecker, file /home4/yuliu/Cactus/arrangements/CactusUtils/NaNChecker/
↪src/NaNCheck.cc:251:
->[0m 'action_if_found' parameter is set to 'terminate' - scheduling graceful␣
↪termination of Cactus
>>> NaNChecker::action_if_found = "just warn"
```

- Tracking and Visualizing NaNs Positions

```
>>> NaNChecker::out_NaNmask = "yes"
>>> NaNChecker::out_NaNmask = "no"
```

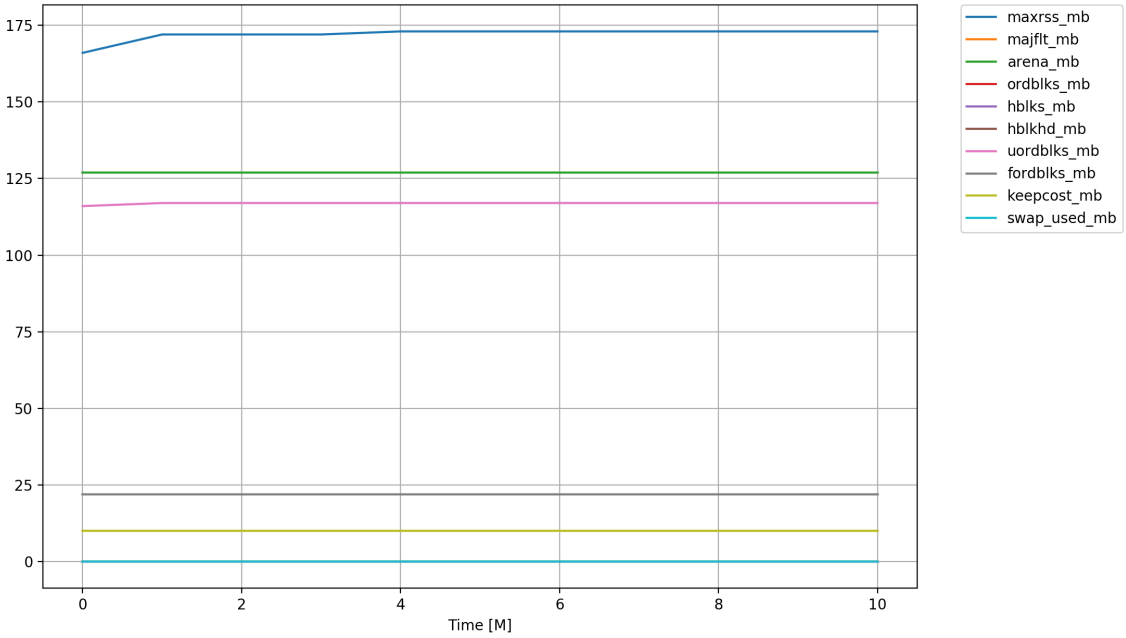
2.10.2 SystemStatistics

Thorn SystemStatistics collects information about the system on which a Cactus process is running and stores it in Cactus variables. These variables can then be output and reduced using the standard Cactus methods such as IOBasic and IOScalar.

Output

- Run memory statistics in MB

```
>>> IOBasic::outInfo_vars  = "SystemStatistics::maxrss_mb{reductions =
↪'maximum'}"
>>> IOScalar::outScalar_vars = "SystemStatistics::process_memory_mb"
--------------------------------
Iteration      Time | *axrss_mb
                    |   maximum
--------------------------------
        0     0.000 |       166
       32     1.000 |       172
       64     2.000 |       172
       96     3.000 |       172
[systemstatistics-process_memory_mb.maximum.asc]
```



2.10.3 Trigger

Trigger can be used to change parameters depending on data from the simulation.

2.10.4 TerminationTrigger

This thorn watches the elapsed walltime. If only n minutes are left before the some limit set by the user, it triggers termination of the simulation.

Parameter

- Walltime in HOURS allocated for this job

```
>>> TerminationTrigger::max_walltime = 1
```

- When to trigger termination in MINUTES

```
>>> TerminationTrigger::on_remaining_walltime = 1
```

- Output remaining wall time every n minutes

```
>>> TerminationTrigger::output_remtime_every_minutes = 5
```

- Create an empty termination file at startup

```
>>> TerminationTrigger::create_termination_file = yes
```

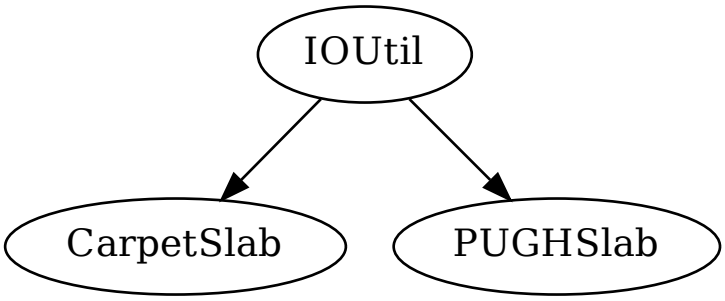
2.11 I/O

In Carpet, a local grid (a "cuboid" that has a uniform spacing in each axis, and lives on a single processor) has a number of attributes:

- **reflevel** - This is an integer specifing the grid's "refinement level" in the Berger-Oliger algorithm.
- **map** - This is an integer specifying the "map" (grid patch) at this refinement level.
- **component** - This is an integer specifying one of the local grids in this map/patch.

2.11.1 IOUtil

Thorns providing IO methods typically have string parameters which list the variables which should be output, how frequently (i.e. how many iterations between output), and where the output should go.



Parameter

- The name of the directory to be used for output.

```
>>> IO::out_dir = $parfile
```

- How often, in terms of iterations, each of the Cactus I/O methods will write output.

```
>>> IO::out_every = 2
------------------------------
it |          | *::coarse_dx |
   |    t     | scalar value |
------------------------------
 0 |    0.000 |   0.25000000 |
 2 |    2.000 |   0.25000000 |
 4 |    4.000 |   0.25000000 |
 6 |    6.000 |   0.25000000 |
 8 |    8.000 |   0.25000000 |
```

- writing to file is performed only by processor zero. This processor gathers all the output data from the other processors and then writes to a single le.

```
>>> IO::out_mode = "onefile"
```

- Every processor writes its own chunk of data into a separate output le.

```
>>> IO::out_mode = "proc"
```

Note: For a run on multiple processors, scalar, 1D, and 2D output will always be written from only processor zero (that is, required data from all other processors will be sent to processor zero, which then outputs all the gathered data). For full-dimensional output of grid arrays this may become a quite expensive operation since output by only a single processor will probably result in an I/O bottleneck and delay further computation. For this reason Cactus offers different I/O modes for such output which can be controlled by the *IO::out_mode* parameter, in combination with *IO::out_unchunked* and *IO::out_proc_every*.

- Checkpointing

```
>>> IO::checkpoint_ID = "yes"              # Checkpoint initial data
INFO (CarpetIOHDF5): Dumping initial checkpoint at iteration 0, simulation time 0
>>> IO::checkpoint_every = 1               # How often to checkpoint
>>> IO::checkpoint_on_terminate = "yes"    # Checkpoint after last iteration
INFO (CarpetIOHDF5): Dumping termination checkpoint at iteration 2432, simulation␣
↪time 47.5
>>> IO::checkpoint_dir = "../checkpoints" # Output directory for checkpoint files
[checkpoint.chkpt.it_0.file_0.h5]
[checkpoint.chkpt.it_0.file_1.h5]
. . .
[checkpoint.chkpt.it_128.file_0.h5]
. . .
```

- Recover

```
>>> IO::recover_dir = "../checkpoints" # Directory to look for recovery files
>>> IO::recover = "autoprobe"
```

Warning

- No driver thorn activated to provide storage for variables

```
>>> ActiveThorns = "CarpetSlab"
AMR driver provided by Carpet
>>> ActiveThorns = "PUGHSlab"
Driver provided by PUGH
```

2.11.2 IOBasic

Thorn IOBasic provides I/O methods for outputting scalar values in ASCII format into files and for printing them as runtime information to screen.

- This method outputs the information into ASCII files named "<scalar_name>.{asc|xg}" (for CCTK_SCALAR variables) and "<var_name>_<reduction>.{asc|xg}" (for CCTK_GF and CCTK_ARRAY variables where reduction would stand for the type of reduction operations (eg. minimum, maximum, L1, and L2 norm)
- This method prints the data as runtime information to stdout. The output occurs as a table with columns containing the current iteration number, the physical time at this iteration, and more columns for scalar/reduction values of each variable to be output.

Reduction Operations

- The minimum of the values

$$\min := \min_i a_i$$

- The maximum of the values

$$\max := \max_i a_i$$

- The norm1 of the values

$$\frac{\Sigma\,|a_i|}{count}$$

- The norm2 of the values

$$\sqrt{\frac{\sum_i |a_i|^2}{count}}$$

Parameter

- Print the information of CCTK_SCALAR variables

```
>>> IOBasic::outInfo_vars = "grid::coarse_dx"
-------------------------------
it  |          | *::coarse_dx |
    |    t     | scalar value |
-------------------------------
  0 |    0.000 |   0.25000000 |
```

- Print the information of CCTK_GF and CCTK_ARRAY variables with the type of reduction

```
>>> IOBasic::outInfo_vars = "wavetoy::phi"
>>> IOBasic::outInfo_reductions = "minimum maximum"
-----------------------------------------------
it  |          | WAVETOY::phi                 |
    |    t     | minimum      | maximum       |
-----------------------------------------------
  0 |    0.000 | 7.104375e-13 |   0.99142726 |
>>> IOBasic::outInfo_vars = "wavetoy::phi{reductions = 'norm2'}"
-------------------------------
it  |          | WAVETOY::phi |
    |    t     | norm2        |
-------------------------------
  0 |    0.000 |   0.10894195 |
```

- Outputs CCTK_SCALAR variabless into ASCII files

```
>>> IOBasic::outScalar_vars = "grid::coarse_dx"
[~/simulations/example/output-0000/example/coarse_dx.xg]
"Parameter file /home4/yuliu/simulations/example/output-0000/example.par
"Created Sep 05 2019 05:05:37-0400
"x-label time
"y-label GRID::coarse_dx
"coarse_dx v time
0.0000000000000     0.2500000000000
```

Warning

- WARNING[L1,P0] (IOBasic): Unknown reduction operator ‘minimum’. Maybe you forgot to activate thorn LocalReduce? (Driver provided by Carpet)

```
>>> ActiveThorns = "CarpetIOBasic CarpetReduce"
```

2.11.3 IOASCII

Thorn IOASCII provides I/O methods for 1D, 2D, and 3D output of grid arrays and grid functions into files in ASCII format.

Parameter

- Outputs CCTK_GF and CCTK_ARRAY variables into ASCII files

```
>>> IOASCII::out1D_every = 1
>>> IOASCII::out1D_style = "gnuplot f(x)"
>>> IOASCII::out1D_vars = "wavetoy::phi"
[~/simulations/example1/output-0000/example1/phi_x_[1][1].asc]
#Parameter file /home4/yuliu/simulations/example/output-0000/example.par
#Created Sep 07 2019 03:55:52-0400
#x-label x
#y-label WAVETOY::phi (y = 0.1500000000000, z = 0.1500000000000), (yi = 1, zi = 1)
#Time = 0.0000000000000
-0.1500000000000            0.9914272633971
```

(continues on next page)

(continued from previous page)

```
0.1500000000000              0.9914272633971
0.4500000000000              0.9689242170281
0.7500000000000              0.9254388283880
. . .
```

Warning

- The aliased function 'Hyperslab_GetList' (required by thorn 'IOASCII') has not been provided by any active thorn ! (Driver provided by Carpet)

```
>>> ActiveThorns = "CarpetIOASCII"
```

2.11.4 CarpetIOBasic

This thorn provides info output for Carpet.

Parameter

- Variables to output in scalar form

```
>>> IOBasic::outInfo_vars = "ADMBase::gxx"
-----------------------------------------------
Iteration      Time |              ADMBASE::gxx
                    |      minimum      maximum
-----------------------------------------------
        0     0.000 |    1.0000000    1.0000000
```

Warning

- Reduction operator "maximum" does not exist (maybe there is no reduction thorn active?)

```
>>> ActiveThorns = "CarpetReduce"
```

2.11.5 CarpetIOScalar

This thorn provides scalar output for Carpet.

Parameter

- Variables to output in scalar form

```
>>> IOScalar::outScalar_vars = ""
```

- Write one file per group instead of per variable

```
>>> IOScalar::one_file_per_group = yes
```

2.11.6 CarpetIOASCII

This thorn provides ASCII output for Carpet. The CarpetIOASCII I/O methods can output any type of CCTK grid variables (grid scalars, grid functions, and grid arrays of arbitrary dimension); data is written into separate les named "<varname>.asc".

It reproduces most of the functionality of thorn IOASCII from the standard CactusBase arrangement. Where possible the names of parameters and their use is identical. However, this thorn outputs considerably more information than the standard IOASCII thorn. Information about, e.g., the renement level and the index position of the output are also given. All the output can be visualized using gnuplot.

Parameter

- Variables to output in 1D ASCII file format

```
>>> IOASCII::out1D_vars = "ADMBase::gxx"
[~/simulations/example/output-0000/example/gxx.x.asc]
# 1D ASCII output created by CarpetIOASCII
# created on ubuntu by yuliu on Sep 10 2019 at 03:33:33-0400
# parameter filename: "/home4/yuliu/simulations/example/output-0000/example.par"
#
# gxx x (gxx)
#
# iteration 0   time 0
# time level 0
# refinement level 0   multigrid level 0   map 0   component 0
# column format: 1:it       2:tl    3:rl 4:c 5:ml   6:ix 7:iy 8:iz  9:time  10:x␣
↪11:y 12:z  13:data
. . .
>>> IOASCII::out2D_vars = "ADMBase::gxx"
[~/simulations/example/output-0000/example/gxx.xy.asc]
# 2D ASCII output created by CarpetIOASCII
# created on ubuntu by yuliu on Sep 10 2019 at 04:14:22-0400
# parameter filename: "/home4/yuliu/simulations/example/output-0000/example.par"
#
# gxx x y (gxx)
#
# iteration 0   time 0
# time level 0
# refinement level 0   multigrid level 0   map 0   component 0
# column format: 1:it       2:tl    3:rl 4:c 5:ml   6:ix 7:iy 8:iz  9:time  10:x␣
↪11:y 12:z  13:data
0   0       0 0 0   0 0 12  0       -12 -12 0       1
0   0       0 0 0   1 0 12  0       -11 -12 0       1
0   0       0 0 0   2 0 12  0       -10 -12 0       1
. . .
0   0       0 0 0   0 1 12  0       -12 -11 0       1
0   0       0 0 0   1 1 12  0       -11 -11 0       1
0   0       0 0 0   2 0 12  0       -10 -11 0       1
. . .
0   0       0 0 0   0 2 12  0       -12 -10 0       1
0   0       0 0 0   1 2 12  0       -11 -10 0       1
0   0       0 0 0   2 2 12  0       -10 -10 0       1
>>> IOASCII::out3D_vars = "ADMBase::gxx"
[~/simulations/example/output-0000/example.par]
# 3D ASCII output created by CarpetIOASCII
# created on ubuntu by yuliu on Sep 10 2019 at 04:19:51-0400
```

(continues on next page)

(continued from previous page)

```
# parameter filename: "/home4/yuliu/simulations/example/output-0000/example.par"
#
# gxx x y z (gxx)
#
# iteration 0   time 0
# time level 0
# refinement level 0   multigrid level 0   map 0   component 0
# column format: 1:it   2:tl    3:rl 4:c 5:ml   6:ix 7:iy 8:iz  9:time  10:x 11:y␣
↪12:z  13:data
0       0       0 0 0   0 0 0   0       -12 -12 -12     1
0       0       0 0 0   1 0 0   0       -11 -12 -12     1
0       0       0 0 0   2 0 0   0       -10 -12 -12     1
. . .
0       0       0 0 0   0 1 0   0       -12 -11 -12     1
0       0       0 0 0   1 1 0   0       -11 -11 -12     1
0       0       0 0 0   2 1 0   0       -10 -11 -12     1
. . .
0       0       0 0 0   0 2 0   0       -12 -10 -12     1
0       0       0 0 0   1 2 0   0       -11 -10 -12     1
0       0       0 0 0   2 2 0   0       -10 -10 -12     1
. . .
0       0       0 0 0   0 0 1   0       -12 -12 -11     1
0       0       0 0 0   1 0 1   0       -11 -12 -11     1
0       0       0 0 0   2 0 1   0       -10 -12 -11     1
. . .
0       0       0 0 0   0 1 0   0       -12 -11 -11     1
0       0       0 0 0   1 1 0   0       -11 -11 -11     1
0       0       0 0 0   2 1 0   0       -10 -11 -11     1
. . .
0       0       0 0 0   0 2 0   0       -12 -10 -11     1
0       0       0 0 0   1 2 0   0       -11 -10 -11     1
0       0       0 0 0   2 2 0   0       -10 -10 -11     1
. . .
0       0       0 0 0   0 0 1   0       -12 -12 -10     1
0       0       0 0 0   1 0 1   0       -11 -12 -10     1
0       0       0 0 0   2 0 1   0       -10 -12 -10     1
. . .
0       0       0 0 0   0 1 0   0       -12 -11 -10     1
0       0       0 0 0   1 1 0   0       -11 -11 -10     1
0       0       0 0 0   2 1 0   0       -10 -11 -10     1
. . .
0       0       0 0 0   0 2 0   0       -12 -10 -10     1
0       0       0 0 0   1 2 0   0       -11 -10 -10     1
0       0       0 0 0   2 2 0   0       -10 -10 -10     1
```

- Write one file per group instead of per variable

```
>>> IOASCII::out3D_vars = "ADMBase::gxx"
>>> IOASCII::one_file_per_group = yes
[~/simulations/example/output-0000/example/admbase-metric.xyz.asc]
# 3D ASCII output created by CarpetIOASCII
# created on ubuntu by yuliu on Sep 10 2019 at 04:28:57-0400
# parameter filename: "/home4/yuliu/simulations/example/output-0000/example.par"
#
# ADMBASE::METRIC x y z (admbase-metric)
#
# iteration 0   time 0
```

(continues on next page)

(continued from previous page)

```
# time level 0
# refinement level 0   multigrid level 0   map 0   component 0
# column format: 1:it   2:tl    3:rl 4:c 5:ml   6:ix 7:iy 8:iz  9:time  10:x 11:y␣
↪12:z  13:data
# data columns: 13:gxx 14:gxy 15:gxz 16:gyy 17:gyz 18:gzz
>>> IOASCII::out3D_vars = "ADMBase::gxx"
>>> IOASCII::one_file_per_group = no
[~/simulations/example/output-0000/example/gxx.xyz.asc]
```

2.11.7 CarpetIOHDF5

Thorn CarpetIOHDF5 provides HDF5-based output to the Carpet mesh refinement driver in Cactus. The CarpetIOHDF5 I/O method can output any type of CCTK grid variables (grid scalars, grid functions, and grid arrays of arbitrary dimension); data is written into separate les named "<varname>.h5". **HDF5 is highly recommended over ASCII for performance and storage-size reasons.**

Note: The default is to output distributed grid variables in parallel, each processor writing a file <varname>.file_<processor ID>.h5. Unchunked means that an entire Cactus grid array (gathered across all processors) is stored in a single HDF5 dataset whereas chunked means that all the processor-local patches of this array are stored as separate HDF5 datasets (called chunks). Consequently, for unchunked data all interprocessor ghostzones are excluded from the output. In contrast, for chunked data the interprocessor ghostzones are included in the output. When visualising chunked datasets, they probably need to be recombined for a global view on the data. This needs to be done within the visualisation tool.

Parameter

- Variables to output in CarpetIOHDF5 file format. The variables must be given by their fully qualied variable or group name.

```
>>> IOHDF5::out_vars = "ADMBase::gxx"
```

- Parallel (chunked) Output of Grid Variables or unchunked of Grid Variables.

```
>>> IO::out_mode = "onefile"
>>> IO::out_unchunked = 1
[gxx.h5]
>>> IO::out_mode = "proc"
[gxx.file_0.h5]
[gxx.file_1.h5]
[gxx.file_2.h5]
. . .
```

- Do checkpointing with CarpetIOHDF5

```
>>> IOHDF5::checkpoint = "yes"
```

include:: CactusBase.rst

include:: Llama.rst

include:: CactusNumerical.rst

include:: EinsteinBase.rst

include:: EinsteinEOS.rst

include:: EinsteinEvolve.rst

include:: CactusConnect.rst

CHAPTER 3

Thorn Write

Naming Conventions:

- Thorn names must not start with the word "Cactus" (in any case).
- Routine names have to be unique among all thorns.
- Names of global variables have to be unique among all thorns.

The following naming conventions are followed by the flesh and the supported Cactus arrangements. They are not compulsory.

- Parameters: lower case with words separated by an underscore. Examples: my_first_parameter.
- Routine names and names of global variables: Prefixed by thorn name with an underscore, then capitalised words, with no spaces. Examples: MyThorn_StartUpRoutine.

3.1 A brief introduction to C

C is a programming language originally developed for developing the Unix operating system. It is a low-level and powerful language, but it lacks many modern and useful constructs. C++ is a newer language, based on C, that adds many more modern programming language features that make it easier to program than C.

A C program should be written into one or more text files with extension ".c".

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

3.1.1 Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

Each variable in C has a specific type, which determines how much space it occupies in storage and how the bit pattern stored is interpreted.

Type	Description
char	A single character.
int	An integer.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

To declare a variable you use

```
<variable type> <name of variable>;
```

Variables can be initialized (assigned an initial value) in their declaration.

```
<variable type> <name of variable> = <value of variable>;
```

Constants

Constants refer to fixed values that the program may not alter during its execution. There are two simple ways in C to define constants

- Using `#define` preprocessor.
- Using `const` keyword.

The #define Preprocessor

Given below is the form to use `#define` preprocessor to define a constant

```
#define <identifier> <value>
```

The const Keyword

The word "const" in front of a type name means that the variable is constant and thus its value cannot be modified by the program. Constant variables are initialised when they are declared:

```
const <variable type> <name of variable> = <value of variable>;
```

Arrays

Arrays are special variables which can hold more than one value under the same variable name, organised with an index.

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows

```
<type> <arrayName> [ <arraySize> ];
```

The arraySize must be an integer constant greater than zero and type can be any valid C data type.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. If you omit the size of the array, an array just big enough to hold the initialization is created.

Accessing Array Elements

An element is accessed by indexing the array name. For example

```
double salary = balance[9];
```

The above statement will take the 10^{th} element from the array and assign the value to `salary` variable.

Strings

Strings are actually one-dimensional array of characters terminated by a null character '0'.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows

```
char greeting[] = "Hello";
```

Structures

When programming, it is often convenient to have a single name with which to refer to a group of a related values. Structures provide a way of storing many different values in variables of potentially different types under the same name.

Defining a Structure

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows

```
struct <tag> {
    <members>;
};
```

Where Tag is the name of the entire type of structure and Members are the variables within the struct.

Accessing Structure Members

To access any member of a structure, we use the member access operator '.'.

```
struct <tag> <structure_variables>; // Declare <structure_variables> of type <tag>
<structure_variables>.<members>
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable

```
struct <tag> *<struct_pointer>;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows

```
<struct_pointer> = &<structure_variables>;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows

```
<struct_pointer> -> <members>
```

typedef

The C programming language provides a keyword called `typedef`, which you can use to give a type a new name.

```
typedef <existing_name> <alias_name>
```

Local Variable

Variables can be declared at any point in the code, provided that, of course, they are declared before they are used. The declaration is valid only within the local block, i.e. within the region limited by braces "{ }".

Size

To get the exact size of a type or a variable on a particular platform, you can use the `sizeof` operator.

```
int a
sizeof(a)
```

Type Casting

Converting one datatype into another is known as type casting or, type-conversion. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable.

```
(type_name) expression
```

3.1.2 Functions

A function is simply a collection of commands that do "something". You can either use the built-in library functions or you can create your own functions.

They must all be declared before there is a call to them. A function declaration tells the compiler about a function's name, return type, and a list of arguments.

```
<return_type> <function_name>(<arguments>)
```

A function definition provides the actual body of the function.

```
<return_type> <function_name>(<arguments>) {
    /* Here goes your code */
}
```

Here are all the parts of a function:

- **Return Type** - A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword `void`.
- **Function Name** - This is the actual name of the function.
- **Arguments** - When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function.
- **Function Body** - The function body contains a collection of statements that define what the function does.

Note: In C, arguments are copied by value to functions, which means that we cannot change the arguments to affect their value outside of the function. To do that, we must use pointers.

3.1.3 Pointers

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

```
<type> *<varname>;
```

Here, type is the pointer's base type; it must be a valid C data type and varname is the name of the pointer variable.

The cool thing is that once you can talk about the address of a variable, you'll then be able to go to that address and retrieve the data stored in it, use the `*`. The technical name for this doing this is dereferencing the pointer.

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration.

Memory allocation

The function `malloc`, residing in the *stdlib.h* header file, is used to initialize pointers with memory from free store. The argument to `malloc` is the amount of memory requested (in bytes), and `malloc` gets a block of memory of that size and then returns a pointer to the block of memory allocated.

Since different variable types have different memory requirements, we need to get a size for the amount of memory `malloc` should return. So we need to know how to get the size of different variable types. This can be done using the keyword `sizeof`, which takes an expression and returns its size. For example,

```
#include <stdlib.h>

int *ptr = malloc(sizeof(int));
```

This code set `ptr` to point to a memory address of size int. The memory that is pointed to becomes unavailable to other programs. This means that the careful coder should free this memory at the end of its usage lest the memory be lost to the operating system for the duration of the program (this is often called a memory leak because the program is not keeping track of all of its memory). The free function returns memory to the operating system.

```
free(ptr);
```

Passing pointers to functions in C

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Return pointer from functions in C

3.1.4 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions.

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language.

Operator	Description
+	Adds two operands.
-	Subtracts second operand from the first.
*	Multiplies both operands.
/	Divides numerator by de-numerator.
%	Modulus Operator and remainder of after an integer division.
++	Increment operator increases the integer value by one.
- -	Decrement operator decreases the integer value by one.

Relational Operators

The following table shows all the relational operators supported by C.

Operator	Description
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.

Logical Operators

Following table shows all the logical operators supported by C language.

Operator	Description
&&	Called Logical AND operator.
||	Called Logical OR Operator.
!	Called Logical NOT Operator.

3.1.5 Decision

The `if` statement allows us to check if an expression is true or false, and execute different code according to the result.

```
if ( expression ) {
    /* Here goes your code */
}
else {
    /* Here goes your code */
}
```

Note: C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Conditional Operator `? :` can be used to replace `if...else` statements. It has the following general form

```
Exp1 ? Exp2 : Exp3;
```

3.1.6 File I/O

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access to handle file on your storage devices.

Opening Files

You can use the `fopen()` function to create a new file or to open an existing file. This call will initialize an object of the type `FILE`, which contains all the information necessary to control the stream. The prototype of this function call is as follows

```
FILE *fp;
fp = fopen(filename, mode);
```

Here, filename is a string literal, which you will use to name your file, and access mode can have one of the following values

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Closing a File

To close a file, use the `fclose()` function. The prototype of this function is

```
fclose(fp);
```

The `fclose()` function returns zero on success, or EOF if there is an error in closing the file.

Writing a File

Following is the simplest function to write argument to a stream

```
fprintf(fp, comments);
```

Reading a File

Use `fscanf()` function to read strings from a file, but it stops reading after encountering the first space character.

```
fscanf(fp, "%s", buff);
```

3.1.7 Library

The C standard library provides numerous built-in functions that your program can call. To access the standard functions that comes with your compiler, you need to include a header with the `#include` directive.

assert.h

TODO

math.h

TODO

stdio.h

TODO

stdlib.h

TODO

string.h

string.h is a header file that contains many functions for manipulating strings.

int strcmp(const char * str1, const char * str2);
Compare two strings.

Parameters

- **str1** (*char*) – C string to be compared.
- **str2** (*char*) – C string to be compared.

Result

- **0** (*int*) - the contents of both strings are equal
- **<0** (*int*) - the first character that does not match has a lower value in ptr1 than in ptr2
- **>0** (*int*) - the first character that does not match has a greater value in ptr1 than in ptr2

3.2 Cactus Configuration Language

Cactus Configuration Language (CCL) files are text files which tells the flesh all it needs to know about the thorns.

CCL files may contain comments introduced by the hash # character, which indicates that the rest of the line is a comment. If the last non-blank character of a line in a CCL file is a backslash \, the following line is treated as a continuation of the current line.

3.2.1 The interface.ccl File

The interface configuration file consists of:

- A header block giving details of the thorn's relationship with other thorns.
- A block detailing which include files are used from other thorns, and which include files are provided by this thorn.
- Blocks detailing aliased functions provided or used by this thorn.

- A series of blocks listing the thorn's global variables.

implementation name

The implementation name is declared by a single line at the top of the file

```
implements: <name>
```

The implementation name must be unique among all thorns.

Inheritance relationships between thorns

There are two relationship statements that can be used to get variables (actually groups of variables) from other implementations.

- Gets all **Public** variables from implementation <name>, and all variables that <name> has in turn inherited

  ```
  Inherits: <name>
  ```

  A thorn cannot inherit from itself. Inheritance is transitive (if A inherits from B, and B inherits from C, then A also implicitly inherits from C), but not commutative.

- Gets all **Protected** variables from implementation <name>, but, unlike inherits, it defines a transitive relation by pushing its own implementation's **Protected** variables onto implementation <name>.

  ```
  Friend: <name>
  ```

  A thorn cannot be its own friend. Friendship is associative, commutative and transitive (if A is a friend of B, and B is a friend of C, then A is implicitly a friend of C).

Include Files

Cactus provides a mechanism for thorns to add code to include files which can be used by any other thorn. Such include files can contain executable source code, or header/declaration information.

```
INCLUDE[S] [SOURCE|HEADER]: <file_to_include> in <file_name>
```

This indicates that this thorn adds the code in <file to include> to the include file <file name>.

Any thorn which uses the include file must declare this in its *interface.ccl* with the line

```
USES INCLUDE [SOURCE|HEADER]: <file_name>
```

If the optional `[SOURCE|HEADER]` is omitted, `HEADER` is assumed. Note that this can be dangerous, as included source code, which is incorrectly assumed to be header code, will be executed in another thorn even if the providing thorn is inactive. Thus, it is recommended to always include the optional `[SOURCE|HEADER]` specification.

Thorn variables

Cactus variables are used instead of local variables for a number of reasons:

- Cactus variables can be made visible to other thorns, allowing thorns to communicate and share data.
- Cactus variables can be distributed and communicated among processors, allowing parallel computation.

- A database of Cactus variables, and their attributes, is held by the flesh, and this information is used by thorns, for example, for obtaining a list of variables for checkpointing.

Cactus variables are placed in groups with homogeneous attributes, where the attributes describe properties such as the data type, group type, dimension, ghostsize, number of timelevels, and distribution.

```
[<access>:]

<data_type> <group_name>[[<number>]] [TYPE=<group_type>] [DIM=<dim>] [TIMELEVELS=<num>
↪] [SIZE=<size in each direction>] [DISTRIB=<distribution_type>] [GHOSTSIZE=
↪<ghostsize>] [TAGS=<string>]
{
    <variable_name>,  <variable_name>,  <variable_name>
} ["<group_description>"]
```

Currently, the names of groups and variables must be distinct. The options TYPE, DIM, etc., following <group name> must all appear on one line.

Access

There are three different access levels available for variables

- **Public**: Can be 'inherited' by other implementations.
- **Protected**: Can be shared with other implementations which declare themselves to be friends of this one.
- **Private**: Can only be seen by this thorn.

By default, all groups are **private**, to change this, an access specification of the form `public:` or `protected:`

Data Type

Cactus supports integer, real, complex and character variable types, in various different sizes. Normally a thorn should use the default types (**CCTK_INT**, **CCTK_REAL**, **CCTK_COMPLEX**) rather than explicitly setting the size, as this gives maximum portability.

Vector Group

If [number] present, indicates that this is a vector group.

Group Types

Groups can be either scalars, grid functions (GFs), or grid arrays.

- **SCALAR**: This is just a single number.
- **GF**: This is the most common group type. A GF is an array with a specific size, set at run time in the parameter file, which is distributed across processors. All GFs have the same size, and the same number of ghostzones. Groups of GFs can also specify a dimension, and number of timelevels.
- **ARRAY**: This is a more general form of the GF. Each group of arrays can have a distinct size and number of ghostzones, in addition to dimension and number of timelevels. The drawback of using an array over a GF is that a lot of data about the array can only be determined by function calls, rather than the quicker methods available for GFs.

DIM

DIM defines the spatial dimension of the ARRAY or GF. The default value is `DIM=3`.

Timelevels

TIMELEVELS defines the number of timelevels a group has if the group is of type ARRAY or GF, and can take any positive value. The default is one timelevel.

Size and Distrib

A Cactus grid function or array has a size set at runtime by parameters. This size can either be the global size of the array across all processors (`DISTRIB=DEFAULT`), or, if `DISTRIB=CONSTANT`, the specified size on each processor. If the size is split across processors, the driver thorn is responsible for assigning the size on each processor.

Ghost Zones

Cactus is based upon a distributed computing paradigm. That is, the problem domain is split into blocks, each of which is assigned to a processor. For hyperbolic and parabolic problems the blocks only need to communicate at the edges. It defaults to zero.

TAGS

TAGS defines an optional string which is used to create a set of key-value pairs associated with the group. The keys are case independent. The string (which must be deliminated by single or double quotes) is interpreted by the function `Util_TableSetFromString()`.

Function

Cactus offers a mechanism for calling a function in a different thorn where you don't need to know which thorn is actually providing the function, nor what language the function is provided in. Function aliasing is also comparatively inefficient, and should not be used in a part of your code where efficiency is important.

If any aliased function is to be used or provided by the thorn, then the prototype must be declared with the form:

```
<return_type> FUNCTION <alias>(<arg1_type> <intent1>, ...)
```

- The `<return_type>` must be either void, CCTK_INT, CCTK_REAL, CCTK_COMPLEX, CCTK_POINTER, or CCTK_POINTER_TO_CONST. Standard types such as int are not allowed. The keyword `SUBROUTINE` is equivalent to `void FUNCTION`.
- The name of the aliased function `<alias>` must contain at least one uppercase and one lowercase letter and follow the C standard for function names.
- The type of an argument, `<arg*_type>`, must be one of scalar types CCTK_INT, CCTK_REAL, CCTK_COMPLEX, CCTK_POINTER, CCTK_POINTER_TO_CONST, or an array or pointer type CCTK_INT ARRAY, CCTK_REAL ARRAY, CCTK_COMPLEX ARRAY, CCTK_POINTER ARRAY. The scalar types are assumed to be not modifiable. If you wish to modify an argument, then it must have intent OUT or INOUT (and hence must be either a CCTK_INT, a CCTK_REAL, or a CCTK_COMPLEX, or an array of one of these types).

- The intent of each argument, <intent*>, must be either IN, OUT, or INOUT. The C prototype will expect an argument with intent IN to be a value and one with intent OUT or INOUT to be a pointer.
- CCTK_STRING must appear at the end of the argument list.

Using an Aliased Function

To use an aliased function you must first declare it in your interface.ccl file. Declare the prototype as, for example,

```
/* this function will be either required in your thorn by */
REQUIRES FUNCTION <alias>
/* or optionally used in your thorn by */
USES FUNCTION <alias>
```

A prototype of this function will be available to any C routine that includes the cctk.h header file.

Providing a Function

To provide an aliased function you must again add the prototype to your interface.ccl file. A statement containing the name of the providing function and the language it is provided in, must also be given. For example,

```
PROVIDES FUNCTION <alias> WITH <provider> LANGUAGE <providing_language>
```

As with the alias name, <provider> must contain at least one uppercase and one lowercase letter, and follow the C standard for function names. It is necessary to specify the language of the providing function; no default will be assumed. Currently, the only supported values of <providing_language> are C and Fortran.

Testing Aliased Functions

The calling thorn does not know if an aliased function is even provided by another thorn. Calling an aliased function that has not been provided, will lead to a level 0 warning message, stopping the code. In order to check if a function has been provided by some thorn, use the CCTK_IsFunctionAliased function described in the function reference section.

3.2.2 The param.ccl File

Parameters are the means by which the user specifies the runtime behaviour of the code. Each parameter has a data type and a name, as well as a range of allowed values and a default value. These are declared in the thorn's *param.ccl* file.

The full specification for a parameter declaration is

```
[<access>:]

[EXTENDS|USES] <parameter_type> <parameter name>[[<len>]] "<parameter_description>"␣
↪[AS <alias>] [STEERABLE=<NEVER|ALWAYS|RECOVER>]
{
    <PARAMETER_RANGES>
} <default_value>
```

The options AS, STEERABLE, etc., following <parameter description>, must all appear in one line.

Access

There are three access levels available for parameters:

- **Global**: These parameters are seen by all thorns.
- **Restricted**: These parameters may be used by other implementations if they so desire.
- **Private**: These are only seen by this thorn.

Inheritance relationships between thorns

To access **restricted** parameters from another implementation, a line containing

```
shares: <name>
```

Each of these parameters must be qualified by the initial token **USES** or **EXTENDS**, where

- **USES**: indicates that the parameters range remains unchanged.
- **EXTENDS**: indicates that the parameters range is going to be extended.

For example, the following block adds possible values to the keyword <par> originally defined in the implementation <name>, and uses the REAL parameter <par>.

```
shares: <name>

EXTENDS KEYWORD <par>
{
    "KEYWORD"   :: "A description of the parameter"
}

USES CCTK_REAL <par>
```

Note that you must compile at least one thorn which implements <name>.

parameter type

Parameters can be of these types:

- **CCTK_INT**: Can take any integral value

  The range specification is of the form

  ```
  INT <par> "A description of the parameter"
  {
      \\ Each range may have a description associated with it by placing a␣
  ↪``::`` on the line, and putting the description afterwards.
      lower:upper:stride :: "Describing the allowed values of the parameter.
  ↪ lower and upper specify the lower and upper allowed range, and stride␣
  ↪allows numbers to be missed out. A missing end of range (or a `*`)␣
  ↪indicates negative or positive infinity."
  } <default value>
  ```

- **CCTK_REAL**: Can take any floating point value

```
REAL <par> "A description of the parameter"
{
    \\ Each range may have a description associated with it by placing a␣
↪``::`` on the line, and putting the description afterwards.
    lower:upper :: "Describing the allowed values of the parameter. lower␣
↪and upper specify the lower and upper allowed range. A missing end of␣
↪range (or a `*`) implies negative or positive infinity. "
} <default value>
```

- CCTK_KEYWORD: A distinct string with only a few known allowed values.

```
KEYWORD <par> "A description of the parameter"
{
    "KEYWORD_1"   :: "A description of the parameter"
    "KEYWORD_2"   :: "A description of the parameter"
    "KEYWORD_3"   :: "A description of the parameter"
} <default value>
```

- CCTK_STRING: An arbitrary string, which must conform to a given regular expression. To allow any string, the regular expression "" should be used. Regular expressions and string values should be enclosed in double quotes.
- CCTK_BOOLEAN: A boolean type which can take values 1, t, true, yes or 0, f, false, no.

```
BOOLEAN <par> "A description of the parameter"
{
} <default value>
```

Name

The <parameter name> must be unique within the scope of the thorn. <len> indicates that this is an array parameter of len values of the specified type. <alias> allows a parameter to appear under a different name in this thorn, other than its original name in another thorn.

Steerable

By default, parameters may not be changed after the parameter file has been read, or on restarting from checkpoint. A parameter can be changed dynamically if it is specified to be steerable. It can then be changed by a call to the flesh function `CCTK_ParameterSet`.

The value RECOVERY is used in checkpoint/recovery situations, and indicates that the parameter may be altered until the value is read in from a recovery par file, but not after.

3.2.3 The schedule.ccl File

A schedule configuration file consists of:

- Assignment statements to switch on storage for grid variables for the entire duration of program execution.
- Schedule blocks to schedule a subroutine from a thorn to be called at specific times during program execution in a given manner.
- Conditional statements for both assignment statements and schedule blocks to allow them to be processed depending on parameter values.

Assignment Statements

Assignment statements, currently only assign storage.

These lines have the form:

```
STORAGE: <group>[timelevels]
```

If the thorn is active, storage will be allocated, for the given groups, for the duration of program execution (unless storage is explicitly switched off by some call to `CCTK_DisableGroupStorage` within a thorn).

The storage line includes the number of timelevels to activate storage for, this number can be from 1 up to the maximum number or timelevels for the group, as specified in the defining *interface.ccl* file. Alternatively timelevels can be the name of a parameter accessible to the thorn. The parameter name is the same as used in C routines of the thorn, fully qualified parameter names of the form `thorn::parameter` are not allowed.

The behaviour of an assignment statement is independent of its position in the schedule file (so long as it is outside a schedule block).

Schedule Blocks

The flesh knows about everything in *schedule.ccl* files, and handles sorting scheduled routines into an order. Each schedule block in the file schedule.ccl must have the syntax

```
schedule [GROUP] <function|schedule group name> AT|IN <schedule bin|group name> [AS
↪<alias>] [WHILE <variable>] [IF <variable>] [BEFORE|AFTER <item>|(<item> <item> ...
↪)]
{
    [LANG: <FORTRAN|C>]
    [STORAGE: <group>[timelevels]]
    [TRIGGERS: <group>]
    [SYNC: <group>]
    [OPTIONS: <option>]
    [TAGS: [list of keyword=value definitions]]
    [READS: <group>]
    [WRITES: <group>]
} "A description"
```

Schedule Bins

Each schedule item is scheduled either AT a particular scheduling bin, or IN a schedule group. A list of the most useful schedule bins for application thorns is given here.

In the absence of any ordering, functions in a particular schedule bin will be called in an undetermined order.

Group

If the optional GROUP specifier is used, the item is a schedule group rather than a normal function. Schedule groups are effectively new, user-defined, schedule bins. Functions or groups may be scheduled IN these, in the same way as they are scheduled AT the main schedule bins.

Schedule Bin	Description
CCTK_STARTUP	For routines which need to be run before the grid hierarchy is set up, for example, for function registration.
CCTK_PARAMCHECK	For routines that check parameter combinations for potential errors. Routines registered here only have access to the grid size and the parameters.
CCTK_INITIAL	For routines which generate initial data.
CCTK_PRESTEP	Tasks performed before the main evolution step.
CCTK_EVOL	The evolution step.
CCTK_POSTSTEP	Tasks performed after the evolution step.
CCTK_ANALYSIS	Routines which can analyze data at each iteration. This time bin is special in that ANALYSIS routines are only called if output from the routine is requested, e.g. in the parameter file

Schedule Options

The options define various characteristics of the schedule item.

- **AS**: This assigns a new name to a function for scheduling purposes.
- **WHILE**: This specifies a CCTK_INT grid scalar which is used to control the execution of this item. As long as the grid scalar has a nonzero value, the schedule item will be executed repeatedly.
- **IF**: This specifies a CCTK_INT grid scalar which is used to control the execution of this item. If the grid scalar has a nonzero value, the schedule item will be executed, otherwise the item will be ignored. If both an IF and a WHILE clause are present, then the schedule is executed according to the following pseudocode:

```
IF condition
    WHILE condition
        SCHEDULE item
    END WHILE
END IF
```

- **BEFORE** or **AFTER**: These specify either a function or group before or after which this item will be scheduled.

Further details

The schedule block specifies further details of the scheduled function or group.

- **LANG**: This specifies the language of the routine. Currently this is either C or Fortran.
- **STORAGE**: The STORAGE keyword specifies groups for which memory should be allocated for the duration of the routine or schedule group. The storage status reverts to its previous status after completion of the routine or schedule group.
- **TRIGGER**: List of grid variables or groups to be used as triggers for causing an ANALYSIS function or group to be executed.
- **SYNC**: The keyword SYNC specifies groups of variables which should be synchronised (that is, their ghost-zones should be exchanged between processors) on exit from the routine.
- **OPTIONS**: Often used schedule options are local (also the default), level, or global. These options are interpreted by the driver, not by Cactus.
- **TAGS**: Schedule tags. These tags must have the form keyword=value, and must be in a syntax accepted by `Util_TableCreateFromString`.
- **READS**: READS is used to declare which grid variables are read by the routine.
- **WRITES**: WRITES is used to declare which grid variables are written by the routine.

Modes

When looping over grids, Carpet has a standard order to loop over grid attributes.

```
# meta mode
begin loop over mglevel (convergence level)

    # global mode
    begin loop over reflevel (refinement level)

        # level mode
```

(continues on next page)

(continued from previous page)

```
        begin loop over map

            # singlemap mode
            begin loop over component
                # local mode
            end loop over component

        end loop over map

    end loop over reflevel

end loop over mglevel
```

When you schedule a routine (by writing a schedule block in a schedule.ccl file), you specify what mode it should run in. If you don't specify a mode, the default is local. Since convergence levels aren't used at present, for most practical purposes meta mode is identical to global mode. Similarly, unless you're doing multipatch simulations, singlemap mode is identical to level mode.

Variables	meta	global	level	singlemap	local
mglevel	No	Yes	Yes	Yes	Yes
reflevel	No	No	Yes	Yes	Yes
map	No	No	No	Yes	Yes
component	No	No	No	No	Yes
CCTK_ORIGIN_SPACE	No	No	Yes	Yes	Yes
CCTK_DELTA_SPACE	No	No	Yes	Yes	Yes
CCTK_DELTA_TIME	No	No	Yes	Yes	Yes
cctk_origin_space	No	Yes	Yes	Yes	Yes
cctk_delta_space	No	Yes	Yes	Yes	Yes
cctk_delta_time	No	Yes	Yes	Yes	Yes
grid scalars	No	Yes	Yes	Yes	Yes
grid arrays	No	Yes	Yes	Yes	Yes
grid functions	No	No	No	No	Yes

Local

Grid functions are defined only in local mode. Since most physics code needs to manipulate grid functions, it therefore must run in local mode. However, in general code scheduled in local mode will run multiple times (because it's nested inside loops over mglevel, reflevel, map, and component). Sometimes you don't want this. For example, you may want to open or close an output file, or initialize some global property of your simulation. Global modes are good for this kind of thing.

Level

Synchronization and turning storage on/off happen in level mode. Boundary conditions must be selected in level mode. Cactus output must be done in level mode.

Reduction/interpolation of grid arrays and/or grid functions may be done in either level mode (applying only to that refinement level), or in global mode (applying to all refinement levels).

Singelmap

Singelmap mode is mostly useful only if you're doing multipatch simulations.

Querying and Changing Modes

Carpet has various functions to query what mode you're in, and functions and macros to change modes. These are all defined in *Carpet/Carpet/src/modes.hh*, and are only usable from C++ code.

To use any of these facilities, put the line `uses include: carpet.hh` in your *interface.ccl*, then `include "carpet.hh"` in your C++ source code (this must come after `include "cctk.h"`).

To query the current mode, just use any of the Boolean predicates `is_meta_mode()`, `is_global_mode()`, ..., `is_local_mode()`.

```
#include <cassert>
#include "cctk.h"
#include "carpet.hh"

void my_function(...)
{
    // make sure we're in level mode
    assert(Carpet::is_level_mode());
}
```

Conditional Statements

Besides schedule blocks, it's possible to embed C style if/else statements in the schedule.ccl file. These can be used to schedule things based upon the value of a parameter.

```
if (<conditional-expression>)
{
    [<assignments>]
    [<schedule blocks>]
}
```

<conditional-expression> can be any valid C construct evaluating to a truth value. Such conditionals are evaluated only at program startup, and are used to pick between different static schedule options.

3.2.4 The Source File

Compile

By default, the CCTK looks in the `src` directory of the thorn for source files.

The Cactus make system looks for a file called `make.code.defn` in that directory (if there is no file called `Makefile` in the `src` directory). At its simplest, this file contains two lines

```
SRCS = <list of all source files in this directory>
SUBDIRS = <list of all subdirectories, including subdirectories of subdirectories>
```

Each subdirectory listed should then have a `make.code.defn` file containing just a `SRCS =` line, a `SUBDIRS =` line will be ignored.

Then you need to build the code. The command you need to run is the following:

```
make <name>
```

Each configuration has a *ThornList* which lists the thorns to be compiled in. When this list changes, only those thorns directly affected by the change are recompiled.

Routines

Any source file using Cactus infrastructure should include the header file `cctk.h` using the line

```
#include "cctk.h"
```

Any routine using Cactus argument lists (for example, all routines called from the scheduler at time bins between CCTK_STARTUP and CCTK_SHUTDOWN) should include at the top of the file the header

```
#include "cctk_Arguments.h"
```

A Cactus macro CCTK_ARGUMENTS is defined for each thorn to contain:

- General information about the grid hierarchy.
- All the grid variables defined in the thorn's *interface.ccl*.
- All the grid variables required from other thorns as requested by the `inherits` and `friend` lines in the *interface.ccl*.

These variables must be declared at the start of the routine using the macro DECLARE_CCTK_ARGUMENTS.

Any routine using Cactus parameters should include at the top of the file the header

```
#include "cctk_Parameters.h"
```

All parameters defined in a thorn's *param.ccl*. Booleans and Integers appear as CCTK_INT types (with nonzero/zero values for boolean yes/no), Reals as CCTK_REAL, and Keywords and String parameters as CCTK_STRING. These variables are read only, and changes should not be made to them. The effect of changing a parameter is undefined (at best). To compare a string valued parameter use the function `CCTK_Equals()`.

The parameters should be declared at the start of the routine using them with the macro DECLARE_CCTK_PARAMETERS.

Example

The C routine MyCRoutine is scheduled in the schedule.ccl file, and uses Cactus parameters. The source file should look like

```
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

void MyFunction(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS
    DECLARE_CCTK_PARAMETERS

    /* Here goes your code */
};
```

The C++ routine MyCRoutine is scheduled in the schedule.ccl file, and uses Cactus parameters. The source file should look like

```
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

extern "C" void MyFunction(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS
    DECLARE_CCTK_PARAMETERS

    /* Here goes your code */
};
```

The Fortran routine MyCRoutine is scheduled in the schedule.ccl file, and uses Cactus parameters. The source file should look like

```
#include "cctk.h"
#include "cctk_Arguments.h"
#include "cctk_Parameters.h"
#include "cctk_Functions.h"

subroutine MyFunction(CCTK_ARGUMENTS)

    implicit none

    DECLARE_CCTK_ARGUMENTS
    DECLARE_CCTK_PARAMETERS
    DECLARE_CCTK_FUNCTIONS

    /* Here goes your code */
end
```

Specifically for C Programmers

Grid functions are held in memory as 1-dimensional C arrays. Cactus provides macros to find the 1-dimensional index which is needed from the multidimensional indices which are usually used. There is a macro for each dimension of grid function. Below is an artificial example to demonstrate this using the 3D macro `CCTK_GFINDEX3D`:

```
for (k=0; k<cctk_lsh[2]; ++k) {
    for (j=0; j<cctk_lsh[1]; ++j) {
        for (i=0; i<cctk_lsh[0]; ++i) {
            int const ind3d = CCTK_GFINDEX3D(cctkGH,i,j,k);
            rho[ind3d] = exp(-pow(r[ind3d],2));
        }
    }
}
```

Here, `CCTK_GFINDEX3D(cctkGH,i,j,k)` expands to `((i) + cctkGH->cctk_lsh[0]*((j)+cctkGH->cctk_lsh[1]`
In Fortran, grid functions are accessed as Fortran arrays, i.e. simply as `rho(i,j,k)`.

To access vector grid functions, one also needs to specify the vector index. This is best done via the 3D macro `CCTK_VECTGFINDEX3D`:

```
for (k=0; k<cctk_lsh[2]; ++k) {
    for (j=0; j<cctk_lsh[1]; ++j) {
        for (i=0; i<cctk_lsh[0]; ++i) {
            /* vector indices are 0, 1, 2 */
            vel[CCTK_VECTGFINDEX3D(cctkGH,i,j,k,0)] = 1.0;
            vel[CCTK_VECTGFINDEX3D(cctkGH,i,j,k,1)] = 0.0;
            vel[CCTK_VECTGFINDEX3D(cctkGH,i,j,k,2)] = 0.0;
        }
    }
}
```

Cactus Variables

The Cactus variables which are passed through the macro `CCTK_ARGUMENTS` are

Variables	Description
cctkGH	A C pointer identifying the grid hierarchy.
cctk_dim	An integer with the number of dimensions used for this grid hierarchy.
cctk_lsh	An array of cctk_dim integers with the local grid size on this processor.
cctk_ash	An array of cctk_dim integers with the allocated size of the array.
cctk_gsh	An array of cctk_dim integers with the global grid size.
cctk_iteration	The current iteration number.
cctk_delta_time	A CCTK_REAL with the timestep.
cctk_time	A CCTK_REAL with the current time.
cctk_delta_space	An array of cctk_dim CCTK_REALs with the grid spacing in each direction.
cctk_nghostzones	An array of cctk_dim integers with the number of ghostzones used in each direction.
cctk_origin_space	An array of cctk_dim CCTK_REALs with the spatial coordinates of the global origin of the grid.
cctk_lbnd	An array of cctk_dim integers containing the lowest index (in each direction) of the local grid, as seen on the global grid.
cctk_ubnd	An array of cctk_dim integers containing the largest index (in each direction) of the local grid, as seen on the global grid.
cctk_bbox	An array of 2*cctk_dim integers, which indicate whether the boundaries are internal boundaries (e.g. between processors), or physical boundaries. A value of 1 indicates a physical (outer) boundary at the edge of the computational grid, and 0 indicates an internal boundary.
cctk_levfac	An array of cctk_dim integer factors by which the local grid is refined in the corresponding direction with respect to the base grid.
cctk_levoff	Two arrays of cctk_dim integers describing the distance by which the local grid is offset with respect to the base grid, measured in local grid spacings.
cctk_timefac	The integer factor by which the time step size is reduced with respect to the base grid.
cctk_convlevel	The convergence level of this grid hierarchy. The base level is 0, and every level above that is coarsened by a factor of cctk_convfac.
cctk_convfac	The factor between convergence levels. The relation between the resolutions of different convergence levels is $\Delta x_L = \Delta x_0 \cdot F^L$, where L is the convergence level and F is the convergence factor. The convergence factor defaults to 2.

CCTK Routines

Providing Runtime Information

To write from thorns to standard output (i.e. the screen) at runtime, use the macro CCTK_INFO. For example,

```
CCTK_INFO("Hellow World")
```

will write the line:

```
INFO (MyThorn): Hellow World
```

Including variables in the info message use CCTK_VINFO. For example,

```
CCTK_VINFO("The integer is %d", myint);
```

Here are some commonly used conversion specifiers:

specifiers	type
%d	int
%g	real (the shortest representation)
%s	string

For a multiprocessor run, only runtime information from processor zero will be printed to screen by default.

Error Handling, Warnings and Code Termination

The Cactus macros CCTK_ERROR and CCTK_VERROR should be used to output error messages and abort the code. The Cactus macros CCTK_WARN and CCTK_VWARN should be used to issue warning messages during code execution.

Along with the warning message, an integer is given to indicate the severity of the warning.

Macros	Level	Description
CCTK_WARN_ABORT	0	abort the Cactus run
CCTK_WARN_ALERT	1	the results of this run will be wrong,
CCTK_WARN_COMPLAIN	2	the user should know about this, but the problem is not terribly surprising
CCTK_WARN_PICKY	3	this is for small problems that can probably be ignored, but that careful people may want to know about
CCTK_WARN_DEBUG	4	these messages are probably useful only for debugging purposes

A level 0 warning indicates the highest severity (and is guaranteed to abort the Cactus run), while larger numbers indicate less severe warnings. For example,

```
CCTK_WARN(CCTK_WARN_ALERT, "Your warning message");
CCTK_ERROR("Your error message");
```

Note that if the flesh parameter cctk_full_warnings is set to true, then CCTK_ERROR and CCTK_WARN automatically include the thorn name, the source code file name and line number in the message. The default is to omit the source file name and line number.

Including variables in the warning message use CCTK_VERROR and CCTK_VWARN. For example,

```
CCTK_VWARN(CCTK_WARN_ALERT, "Your warning message, including %f and %d", myreal,␣
↪myint);
CCTK_ERROR("Your warning message, including %f and %d", myreal, myint);
```

Iterating Over Grid Points

A grid function consists of a multi-dimensional array of grid points. These grid points fall into several types:

- **interior**: regular grid point, presumably evolved in time
- **ghost**: inter-process boundary, containing copies of values owned by another process
- **physical boundary**: outer boundary, presumably defined via a boundary condition
- **symmetry boundary**: defined via a symmetry, e.g. a reflection symmetry or periodicity

---

Note: Grid points in the edges and corners may combine several types. For example, a point in a corner may be a ghost point in the x direction, a physical boundary point in the y direction, and a symmetry point in the z direction.

The size of the physical boundary depends on the application. The number of ghost points is defined by the driver; the number of symmetry points is in principle defined by the thorn implementing the respective symmetry condition, but will in general be the same as the number of ghost points to avoid inconsistencies.

The flesh provides a set of macros to iterate over particular types of grid points.

- Loop over all grid points

```
CCTK_LOOP3_ALL(name, cctkGH, i,j,k)
{
    /* body of the loop */
} CCTK_ENDLOOP3_ALL(name);
```

- Loop over all interior grid points

```
CCTK_LOOP3_INT(name, cctkGH, i,j,k)
{
    /* body of the loop */
} CCTK_ENDLOOP3_INT(name);
```

- Loop over all physical boundary points

  `LOOP_BND` loops over all points that are physical boundaries (independent of whether they also are symmetry or ghost boundaries).

```
CCTK_LOOP3_BND(name, cctkGH, i,j,k, ni,nj,nk)
{
    /* body of the loop */
} CCTK_ENDLOOP3_BND(name);
```

- Loop over all "interior" physical boundary point

  `LOOP_INTBND` loops over those points that are only physical boundaries (and excludes any points that belongs to a symmetry or ghost boundary).

```
CCTK_LOOP3_INTBND(name, cctkGH, i,j,k, ni,nj,nk)
{
    /* body of the loop */
} CCTK_ENDLOOP3_INTBND(name);
```

In all cases, name should be replaced by a unique name for the loop. i, j, and k are names of variables that will be declared and defined by these macros, containing the index of the current grid point. Similarly ni, nj, and nk are names of variables describing the (outwards pointing) normal direction to the boundary as well as the distance to the boundary.

Interpolation Operators

There are two different flesh APIs for interpolation, depending on whether the data arrays are Cactus grid arrays or processor-local, programming language built-in arrays.

`CCTK_InterpGridArrays()` function interpolates a list of CCTK grid variables (in a multiprocessor run these are generally distributed over processors) on a list of interpolation points. The grid topology and coordinates are implicitly specified via a Cactus coordinate system. The interpolation points may be anywhere in the global Cactus grid. Additional parameters for the interpolation operation can be passed in via a handle to a key-value options table.

```
#include "cctk.h"
#include "util_Table.h"

/* Pointer to a valid Cactus grid hierarchy. */
const cGH *GH;

/* Number of dimensions in which to interpolate. */
#define N_DIMS 3

/* Handle to the local interpolation operator as returned by CCTK_InterpHandle. */
const int operator_handle = CCTK_InterpHandle(interpolator_name);
if (operator_handle < 0) {
    CCTK_VWARN(CCTK_WARN_ABORT, "Couldn't find interpolator \"%s\"!", interpolator_
↪name);
}

/* Handle to a key-value table containing zero or more additional parameters for the␣
↪interpolation operation. */
const int param_table_handle = Util_TableCreateFromString(interpolator_pars);
if (param_table_handle < 0) {
    CCTK_VWARN(CCTK_WARN_ABORT, "Bad interpolator parameter(s) \"%s\"!", interpolator_
↪pars);
}

/* Handle to Cactus coordinate system as returned by CCTK_CoordSystemHandle. */
const int coord_system_handle = CCTK_CoordSystemHandle(coord_name);
if (coord_system_handle < 0) {
    CCTK_VWARN(CCTK_WARN_ABORT, "Couldn't get coordinate-system \"%s\"!", coord_name);
}

/* The number of interpolation points requested by this processor. */
#define N_INTERP_POINTS 1000

/* (Pointer to) an array of N dims pointers to 1-D arrays giving the coordinates of␣
↪the interpolation points requested by this processor. These coordinates are with␣
↪respect to the coordinate system defined by coord system handle. */
CCTK_REAL interp_x[N_INTERP_POINTS], interp_y[N_INTERP_POINTS], interp_z[N_INTERP_
↪POINTS];
const void *interp_coords[N_DIMS];

interp_coords[0] = (const void *) interp_x;
interp_coords[1] = (const void *) interp_y;
interp_coords[2] = (const void *) interp_z;

/* The number of input variables to be interpolated. */
#define N_INPUT_ARRAYS 2
```

(continues on next page)

(continued from previous page)

```
/* (Pointer to) an array of N_input_arrays CCTK grid variable indices (as returned by␣
↪CCTK_VarIndex) specifying the input grid variables for the interpolation. */
CCTK_INT input_array_variable_indices[N_INPUT_ARRAYS];
input_array_variable_indices[0] = CCTK_VarIndex("my_thorn::var1");
input_array_variable_indices[1] = CCTK_VarIndex("my_thorn::var2");

/* The number of output arrays to be returned from the interpolation. Note that N_
↪output_arrays may differ from N_input_arrays. */
#define N_OUTPUT_ARRAYS 2

/* Giving the data types of the 1-D output arrays pointed to by output_arrays[]. */
CTK_INT output_array_type_codes[N_OUTPUT_ARRAYS]
output_array_type_codes[0] = CCTK_VARIABLE_REAL
output_array_type_codes[0] = CCTK_VARIABLE_COMPLEX

/* (Pointer to) an array of N_output_arrays pointers to the (user-supplied) 1-D␣
↪output arrays for the interpolation. */
void *output_arrays[N_OUTPUT_ARRAYS];
CCTK_REAL output_for_real_array [N_INTERP_POINTS];
CCTK_COMPLEX output_for_complex_array[N_INTERP_POINTS];
output_arrays[0] = (void *) output_for_real_array;
output_arrays[1] = (void *) output_for_complex_array;

int status = CCTK_InterpGridArrays(
    GH,
    N_DIMS,
    operator_handle,
    param_table_handle,
    coord_system_handle,
    N_INTERP_POINTS,
    CCTK_VARIABLE_REAL, // Giving the data type of the interpolation-point coordinate␣
↪arrays pointed to by interp_coords[].
    interp_coords,
    N_INPUT_ARRAYS,
    input_array_variable_indices,
    N_OUTPUT_ARRAYS,
    output_array_type_codes,
    output_arrays
)

if (status < 0) {
    CCTK_WARN(CCTK_WARN_ABORT, "error return from interpolator!");
}
```

`CCTK_InterpLocalUniform()` interpolate a list of processor-local arrays which define a uniformly-spaced data grid.

Reduction Operators

A reduction operation can be defined as an operation on variables distributed across multiple processor resulting in a single number. Typical reduction operations are: sum, minimum/maximum value, and boolean operations. The different operators are identified by their name and/or a unique number, called a handle.

There are two different flesh APIs for reduction, depending on whether the data arrays are Cactus grid arrays or processor-local, programming language built-in arrays.

`CCTK_ReduceGridArrays()` reduces a list of CCTK grid arrays (in a multiprocessor run these are generally distributed over processors).

```
#include "cctk.h"
#include "util_Table.h"

/* Pointer to a valid Cactus grid hierarchy. */
const cGH *GH;

/* Handle to the local reduction operator as returned by CCTK_
↪LocalArrayReductionHandle(). */


const int status = CCTK_ReduceGridArrays(
    GH,
    0, // The destination processor
    param_table_handle,
    N_INPUT_ARRAYS,
    input_array_variable_indices,
    M_OUTPUT_VALUES,
    output_value_type_codes,
    output_values
);
```

`CCTK_ReduceLocalArrays()` performs reduction on a list of local grid arrays.

Utility Routines

As well as the high-level `CCTK_*` routines, Cactus also provides a set of lower-level `Util_*` utility routines which thorns developers may use. Cactus functions may need to pass information through a generic interface. In the past, we often had trouble passing `extra` information that wasn't anticipated in the original design. Key-value tables provide a clean solution to these problems. They're implemented by the `Util_Table*` functions.

3.3 ParamCheck

In your *schedule.ccl* file.

```
SCHEDULE MyCRoutine_ParamCheck AT CCTK_PARAMCHECK
{
    LANG: C
} "ParamCheck"
```

In your code.

```
#include "cctk.h"

#include "cctk_Arguments.h"
#include "cctk_Parameters.h"

void MyCRoutine_ParamCheck(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS;
    DECLARE_CCTK_PARAMETERS;
```

(continues on next page)

(continued from previous page)

```
    if(! CCTK_EQUALS(metric_type, "physical") &&
       ! CCTK_EQUALS(metric_type, "static conformal"))
    {
        CCTK_PARAMWARN("Unknown ADMBase::metric_type - known types are \"physical\"␣
↪and \"static conformal\"");
    }
}
```

3.3.1 API

int **CCTK_IsThornActive**(const char* *thorn*)

Reports whether a thorn was activated in a parameter file.

Parameters

- **thorn** (*char*) – The character-string name of the thorn

Result **status** (*int*) This function returns a non-zero value if thorn was activated in a parameter file, and zero otherwise.

```
>>> if (CCTK_IsThornActive ("MoL")) {
>>>     /* Here goes your code */
>>> }
```

char **CCTK_ParameterValString**(const char **name*, const char **thorn*)

Get the string representation of a parameter's value.

Discussion The memory must be released with a call to `free()` after it has been used.

Parameters

- **name** (*char*) – Parameter name
- **thorn** (*char*) – Thorn name (for private parameters) or implementation name (for restricted parameters)

Result **valstring** (*char*) - Pointer to parameter value as string

Error **NULL** - No parameter with that name was found.

```
>>> char *valstring = CCTK_ParameterValString("cctk_run_title", "Cactus")
>>> assert( valstring != NULL );
>>> free(valstring);
```

CCTK_ParameterGet(const char **name*, const char **thorn*, int **type*)

Get the data pointer to and type of a parameter's value.

Parameters

- **name** (*char*) – Parameter name
- **thorn** (*char*) – Thorn name (for private parameters) or implementation name (for restricted parameters)
- **type** (*int*) – If not NULL, a pointer to an integer which will hold the type of the parameter

Result **valstring** (*char*) - Pointer to the parameter value

Error **NULL** - No parameter with that name was found.

```
>>> const void *ghost_ptr = CCTK_ParameterGet("ghost_size", "Carpet", NULL);
>>> assert( ghost_ptr != NULL );
>>> int ghost = *(const int *)ghost_ptr;
```

3.4 Coordinate

3.4.1 Global coordinates

Compute the global Cactus xyz coordinates of the current grid point.

```
#include "cctk.h"

void MyThorn_MyFunction(CCTK_ARGUMENTS)
{
    int i, j, k;
    for (k = 0; k < cctk_lsh[2]; ++k) {
        for (j = 0; j < cctk_lsh[1]; ++j) {
            for (i = 0; i < cctk_lsh[0]; ++i) {
                // variables declared with 'const' added become constants and cannot␣
↪be altered by the program.
                const int ind3d = CCTK_GFINDEX3D(cctkGH,i,j,k);

                const CCTK_REAL xL = x[ind3d];
                const CCTK_REAL yL = y[ind3d];
                const CCTK_REAL zL = z[ind3d];
                const CCTK_REAL rL = r[ind3d];
            }
        }
    }
}
```

```
#include "cctk.h"

void MyThorn_MyFunction(CCTK_ARGUMENTS)
{
    int i, j, k;
    /* Do not compute in ghost zones */
    for(i=ghost; i<cctk_lsh[0]-ghost; i++) {
        for(j=ghost; j<cctk_lsh[1]-ghost; j++) {
            for(k=ghost; k<cctk_lsh[2]-ghost; k++) {
                const int ind3d = CCTK_GFINDEX3D(cctkGH,i,j,k);

                const CCTK_REAL xL = x[ind3d];
                const CCTK_REAL yL = y[ind3d];
                const CCTK_REAL zL = z[ind3d];
                const CCTK_REAL rL = r[ind3d];
            }
        }
    }
}
```

3.4.2 CartGrid3D

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

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

You first need to get access to the include le by putting the line in your *interface.ccl* file.

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

Symmetries should obviously be registered before they are used, but since they can be dierent for dierent grids, they must be registered after the `CCTK_STARTUP` timebin. The usual place to register symmetries is in the `CCTK_BASEGRID` timebin.

```
SCHEDULE MyCRoutine_RegisterSymmetry AT CCTK_BASEGRID
{
    LANG: C
    OPTIONS: global
} "Register symmetry"
```

```
#include "cctk.h"
#include "cctk_Arguments.h"
#include "Symmetry.h"

void MyCRoutine_RegisterSymmetry(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS;

    int sym[3];
    int ierr;

    sym[0] = 1;
    sym[1] = 1;
    sym[2] = 1;

    /* Applies symmetry boundary conditions from variable name */
    ierr = SetCartSymVN(cctkGH, sym, "ADMAnalysis::Ricci23");
    /* error information */
    if(ierr) {
        CCTK_VWarn(0, __LINE__, __FILE__, "Thorn_Name", "Error returned from function␣
↪SetCartSymVN");
    }
}
```

3.4.3 Boundary

The implementation *Boundary* provides a number of aliased functions, which allow application thorns to register routines which provide a particular physical boundary condition, and also to select variables or groups of variables to have boundary conditions applied to whenever the *ApplyBCs* schedule group is scheduled.

```
SCHEDULE MyCRoutine_Boundaries
{
    LANG: C
}  "Select boundary conditions"

SCHEDULE GROUP ApplyBCs as MyCRoutine_Boundaries
{
} "Apply boundary conditions"
```

To select a grid variable to have a boundary condition applied to it, use one of the following aliased functions:

```
#include "cctk.h"
#include "cctk_Arguments.h"

void MyCRoutine_Boundaries(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS;

    int ierr;

    /* Select an entire variable group, using its name. */
    err = Boundary_SelectGroupForBC(
        cctkGH, // Grid hierarchy pointer
        CCTK_ALL_FACES, // CCTK_ALL_FACES corresponds to the set of all faces of the␣
↪domain.
        1,
        -1, // use default values
        "ADMAnalysis::ricci_scalar", // group name: (<implementation>::<group_name>)
        "Flat" // The boundary conditions available are Scalar, Flat, Radiation, Copy,
↪ Robin, Static, and None.
    );
    if (err < 0) {
        CCTK_WARN(2, "Error in applying flat boundary condition");
    }
}
```

Each of these functions takes a faces specication, a boundary width, and a table handle as additional arguments. The faces specification is a single integer which identifies a set of faces to which to apply the boundary condition. The boundary width is the thickness, in grid points, of the boundaries. The table handle identifies a table which holds extra arguments for the particular boundary condition that is requested.

3.5 Initial Data

3.5.1 Timelevel

These are best introduced by an example using finite differencing. Consider the 1-D wave equation

$$\frac{\partial^2 \phi}{\partial t^2} = \frac{\partial^2 \phi}{\partial x^2}$$

To solve this by partial differences, one discretises the derivatives to get an equation relating the solution at different times. There are many ways to do this, one of which produces the following difference equation

$$\phi(t+\Delta t, x) - 2\phi(t,x) + \phi(t-\Delta t, x) = \frac{\Delta t^2}{\Delta x^2}\{\phi(t, x+\Delta x) - 2\phi(t,x) + \phi(t, x-\Delta x)\}$$

which relates the three timelevels $t + \Delta t$, t, $t - \Delta t$.

All timelevels, except the current level, should be considered read-only during evolution, that is, their values should not be changed by thorns. The exception to this rule is for function initialisation, when the values at the previous timelevels do need to be explicitly filled out.

```
if (timelevels == 1) {
    STORAGE: rho[1]
}
else if (timelevels == 2) {
    STORAGE: rho[2]
}
else if (timelevels == 3) {
    STORAGE: rho[3]
}
```

```
#include <cctk.h>
#include <cctk_Arguments.h>
#include <cctk_Parameters.h>

void MyCRoutine_Zero(CCTK_ARGUMENTS)
{
    const int np = cctk_ash[0] * cctk_ash[1] * cctk_ash[2];

    if (CCTK_ActiveTimeLevels(cctkGH, "HydroBase::rho") >= 1) {
#pragma omp parallel for
        for (int i = 0; i < np; ++i) {
            rho[i] = 0.0; \* Set rho to 0 *\
        }
    }

    if (CCTK_ActiveTimeLevels(cctkGH, "HydroBase::rho") >= 2) {
#pragma omp parallel for
        for (int i = 0; i < np; ++i) {
            rho_p[i] = 0.0;
        }
    }

    if (CCTK_ActiveTimeLevels(cctkGH, "HydroBase::rho") >= 3) {
#pragma omp parallel for
        for (int i = 0; i < np; ++i) {
            rho_p_p[i] = 0.0;
        }
    }
}
```

3.6 Numerical

3.7 Analysis

3.7.1 Calculation

- determinant

```
#include <stddef.h>
#include "cctk.h"

void MyThorn_MyFunction(CCTK_ARGUMENTS)
{
    int i, j, k;
    for (k = 0; k < cctk_lsh[2]; ++k) {
        for (j = 0; j < cctk_lsh[1]; ++j) {
            for (i = 0; i < cctk_lsh[0]; ++i) {
                int index = CCTK_GFINDEX3D(cctkGH,i,j,k);

                double gxxL = gxx[index];
                double gxyL = gxy[index];
                double gxzL = gxz[index];
                double gyyL = gyy[index];
                double gyzL = gyz[index];
                double gzzL = gzz[index];

                double det = -gxzL*gxzL*gyyL + 2*gxyL*gxzL*gyzL -␣
↪gxxL*gyzL*gyzL - gxyL*gxyL*gzzL + gxxL*gyyL*gzzL;
                double invdet = 1.0 / det;
                double gupxxL=(-gyzL*gyzL + gyyL*gzzL)*invdet;
                double gupxyL=( gxzL*gyzL - gxyL*gzzL)*invdet;
                double gupyyL=(-gxzL*gxzL + gxxL*gzzL)*invdet;
                double gupxzL=(-gxzL*gyyL + gxyL*gyzL)*invdet;
                double gupyzL=( gxyL*gxzL - gxxL*gyzL)*invdet;
                double gupzzL=(-gxyL*gxyL + gxxL*gyyL)*invdet;
            }
        }
    }
}
```

3.7.2 Volume integral

```
#include "cctk.h"
#include "cctk_Arguments.h"

void MyCRoutine(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS;

    CCTK_INT reduction_handle;

    reduction_handle = CCTK_ReductionHandle("sum");
    if (reduction_handle < 0) {
        CCTK_WARN(0, "Unable to get reduction handle.");
    }
    CCTK_Reduce(
        cctkGH,
        -1,
        reduction_handle,
        1,
        CCTK_VARIABLE_REAL,
        &ADMMass_VolumeMass[*ADMMass_LoopCounter],
        1,
```

(continues on next page)

(continued from previous page)

```
        CCTK_VarIndex("ADMMass::ADMMass_VolumeMass_GF")
    )
    // TODO ADMMass
}
```

3.7.3 Surface integral

```
#include "cctk.h"
#include "cctk_Arguments.h"

void MyCRoutine(CCTK_ARGUMENTS)
{
    DECLARE_CCTK_ARGUMENTS;

    // TODO ADMMass
}
```

3.8 I/O

3.8.1 API

int **CCTK_VarIndex**(const char **varname*)

Get the index for a variable.

Discussion The variable name should be the given in its fully qualified form, that is `<implementation>::<variable>` for PUBLIC or PROTECTED variables and `<thorn>::<variable>` for PRIVATE variables.

Parameters

- **varname** (*char*) – The name of the variable.

Error

- **-1** - no variable of this name exists
- **-2** - no failed to catch error code from `Util_SplitString`
- **-3** - given full name is in wrong format
- **-4** - memory allocation failed

```
>>> index = CCTK_VarIndex("evolve::phi");
>>> index = CCTK_VarIndex("evolve::vect[0]");
```

int **CCTK_GroupIndexFromVarI**(int *varindex*)

Given a variable index, returns the index of the associated group

Parameters

- **varindex** (*int*) – The index of the variable

Result **groupindex** (*int*) - The index of the group

```
>>> index = CCTK_VarIndex("evolve::phi");
>>> groupindex = CCTK_GroupIndexFromVarI(index);
```

char **CCTK_FullName** (int *index*)
Given a variable index, returns the full name of the variable

Discussion The full variable name is in the form `<implementation>::<variable>` for PUBLIC or PROTECTED variables and `<thorn>::<variable>` for PRIVATE variables.

Parameters

- **index** (*int*) – The variable index

Result implementation (*char*) - The full variable name

```
>>> name = CCTK_FullName(index);
```

int **CCTK_VarTypeI** (int *index*)
Provides variable type index from the variable index

Discussion The variable type index indicates the type of the variable. Either character, int, complex or real. The group type can be checked with the Cactus provided macros for CCTK_VARIABLE_INT, CCTK_VARIABLE_REAL, CCTK_VARIABLE_COMPLEX or CCTK_VARIABLE_CHAR.

Parameters

- **index** (*int*) – The variable index

Result type (*int*) - The variable type index

```
>>> vtype = CCTK_VarTypeI(index);
>>> if (vtype == CCTK_VARIABLE_REAL) {
>>>     /* Here goes your code */
>>> }
```

int **CCTK_GroupTypeI** (int *group*)
Provides a group type index given a group index

Discussion A group type index indicates the type of variables in the group. The group type can be checked with the Cactus provided macros for CCTK_SCALAR, CCTK_GF, CCTK_ARRAY.

Parameters

- **group** (*int*) – Group index.

Error -1 - the given group index is invalid.

```
>>> gtype = CCTK_GroupTypeI(gindex);
>>> if (gtype == CCTK_GF) {
>>>     /* Here goes your code */
>>> }
```

void **CCTK_VarDataPtrI** (const cGH * *cctkGH*, int *timelevel*, int *index*)
Returns the data pointer for a grid variable from the variable index.

Parameters

- **cctkGH** – pointer to CCTK grid hierarchy
- **timelevel** (*int*) – The timelevel of the grid variable
- **index** (*int*) – The index of the variable

```
>>> CCTK_REAL *data = NULL;
>>> vindex = CCTK_VarIndex("evolve::phi");
>>> data = (CCTK_REAL*) CCTK_VarDataPtrI(cctkGH, 0, vindex);
```

CCTK_GroupData (int *group_index*, cGroup* *group_data_buffer*)
Given a group index, returns information about the group and its variables.

Discussion The cGroup structure contains (at least) the following members:

- **grouptype** (*int*) - group type
- **vartype** (*int*) - variable type
- **disttype** (*int*) - distribution type
- **dim** (*int*) - dimension (rank) of the group
- **numvars** (*int*) - number of variables in the group
- **numtimelevels** (*int*) - declared number of time levels for this group's variables
- **vectorgroup** (*int*) - 1 if this is a vector group,0 if it's not
- **vectorlength** (*int*) - vector length of group (i.e. number of vector elements)
- **tagstable** (*int*) - handle to the group's tags table

Parameters

- **group_index** (*int*) – The group index for which the information is desired
- **group_data_buffer** (*int*) – Pointer to a cGroup structure in which the information should be stored.

Error **-1** - group index is invalid. **-2** - group_data_buffer is NULL.

```
>>> cGroup group_info;
>>> int group_index = CCTK_GroupIndex("BSSN_MoL::ADM_BSSN_metric");
>>> CCTK_GroupData(group_index, &group_info);
>>> CCTK_VINFO("Dim: %d, numvars: %d", group_info.dim, group_info.numvars);
```

int **CCTK_OutputVarAs** (const cGH **cctkGH*, const char **variable*, const char **alias*)
Output a single variable as an alias by all I/O methods.

Discussion If the appropriate file exists the data is appended, otherwise a new file is created. Uses `alias` as the name of the variable for the purpose of constructing a filename.

Parameters

- **cctkGH** – pointer to CCTK grid hierarchy
- **variable** (*char*) – full name of variable to output
- **alias** (*char*) – alias name to base the output filename on

Result **istat** (*int*) - the number of IO methods which did output of variable

Error **negative** - if no IO methods were registered

```
>>> CCTK_OutputVarAs(cctkGH, "HydroBase::rho", "rho");
```

int **CCTK_OutputVarAsByMethod** (const cGH **cctkGH*, const char **variable*, const char **method*, const char **alias*)
Output a variable variable using the method method if it is registered. Uses alias as the name of the variable for

the purpose of constructing a filename. If the appropriate file exists the data is appended, otherwise a new file is created.

Parameters

- **cctkGH** – pointer to CCTK grid hierarchy
- **variable** (*char*) – full name of variable to output
- **method** (*char*) – method to use for output
- **alias** (*char*) – alias name to base the output filename on

Result istat (*int*) - zero for success

Error negative - indicating some error

int **CCTK_OutputVarByMethod** (const cGH **cctkGH*, const char **variable*, const char **method*)

Output a variable variable using the IO method method if it is registered. If the appropriate file exists the data is appended, otherwise a new file is created.

Parameters

- **cctkGH** – pointer to CCTK grid hierarchy
- **variable** (*char*) – full name of variable to output
- **method** (*char*) – method to use for output

Result istat (*int*) - zero for success

Error negative - indicating some error

3.9 Utility

3.9.1 API

int **CCTK_isinf** (double *x*)

int **CCTK_isnan** (double *x*)

3.10 Functions

3.10.1 find_closest

```
#include <cctk.h>
#include <math.h>

int find_closest(const cGH *cctkGH, const int *cctk_lsh,
                 const CCTK_REAL *cctk_delta_space, int ghost,
                 CCTK_REAL *coord, CCTK_REAL coord_min, int dir)
{
    int i, ijk, min_i = -1;
    CCTK_REAL min = 1.e100;

    for(i=ghost; i<cctk_lsh[dir]-ghost; i++) {
        ijk = CCTK_GFINDEX3D(cctkGH, (dir==0)?i:0, (dir==1)?i:0, (dir==2)?i:0);
```

(continues on next page)

(continued from previous page)

```
        if (fabs(coord[ijk] - coord_min) < min) {
            min = fabs(coord[ijk] - coord_min);
            min_i = i;
        }
    }
    return min_i;
}
```

CHAPTER 4

Lorene

LORENE is a set of C++ classes. It provides tools to solve partial differential equations by means of multi-domain spectral methods.

4.1 Multi-domain grid

Setup of a multi-domain grid

```
>>> int nz = 3 ;     // Number of domains
>>> int nr = 7 ;     // Number of collocation points in r in each domain
>>> int nt = 5 ;     // Number of collocation points in theta in each domain
>>> int np = 8 ;     // Number of collocation points in phi in each domain
>>> int symmetry_theta = SYM ; // symmetry with respect to the equatorial plane
>>> int symmetry_phi = NONSYM ; // no symmetry in phi
>>> bool compact = true ; // external domain is compactified
>>> // Multi-domain grid construction:
>>> Mg3d mgrid(nz, nr, nt, np, symmetry_theta, symmetry_phi, compact) ;
>>> cout << mgrid << endl ;
```

CHAPTER 5

Kranc

5.1 Introduction

Kranc is a suite of Mathematica-based computer-algebra packages, which comprise a toolbox to convert certain (tensorial) systems of partial differential evolution equations to parallelized C code for solving initial boundary value problems. **It does this by generating Cactus thorns, allowing use of all the infrastructure provided by Cactus.**

Kranc generates code and Cactus CCL files for:

- Declaring the grid functions which the simulation will use;
- Registering the grid functions for the evolved variables with the MoL thorn;
- Computing the right hand sides of evolution equations so that the time integrator can compute the evolved variables at the next time step;
- Computing finite differences, both using built-in definitions of standard centred finite differencing operators, as well as allowing the user to create customised operators;
- Performing a user-specified calculation at each point of the grid.

5.1.1 Get start

Run Kranc to build the <thornname> thorn:

```
Kranc/Bin/kranc <thornname>
```

There should be a new directory, <thornname>, which is the Cactus thorn which has been newly generated from the <thornname>.m Kranc script.

Note: The scripts look for a Kranc installation in the "m" directory, in the Cactus root directory and in $HOME as well as in the location where they find the kranc shell-script if it is in the $PATH. If your Kranc (or kranc) directory

is in neither of these places, then you can set the environment variable KRANCPATH to the location of your Kranc installation.

5.2 Data structures

Mathematica does not have the concept of a C++ class or a C structure, in which collections of named objects are grouped together for ease of manipulation. Instead, we have defined a Kranc structure as a list of rules of the form `key -> value`.

5.2.1 Name

The name of a calculation is a string which will be used as the function name in Cactus.

```
>>> Name -> "Example"
```

Note: Only the Name key is required; all the others take default values if omitted.

5.3 Creating a Kranc thorn

```
CreateThorn[groups, directory, thornName, namedArgs]
```

Note: If you want to use TensorTools tensors in calculations, you must call the CreateThornTT function instead of this one.

CHAPTER 6

Reference

- Löffler, F. et al. The Einstein Toolkit: a community computational infrastructure for relativistic astrophysics. Class. Quantum Grav. 29, 115001 (2012).
- Pollney, D., Reisswig, C., Schnetter, E., Dorband, N. & Diener, P. High accuracy binary black hole simulations with an extended wave zone. arXiv:0910.3803 [gr-qc] (2009) doi:10.1103/PhysRevD.83.044045.
- Zilhão, M. & Löffler, F. An Introduction to the Einstein Toolkit. Int. J. Mod. Phys. A 28, 1340014 (2013).
- J. Thornburg, "A fast apparent horizon finder for three-dimensional Cartesian grids in numerical relativity," Classical and Quantum Gravity, vol. 21, no. 2, pp. 743–766, Jan. 2004.

Index

C