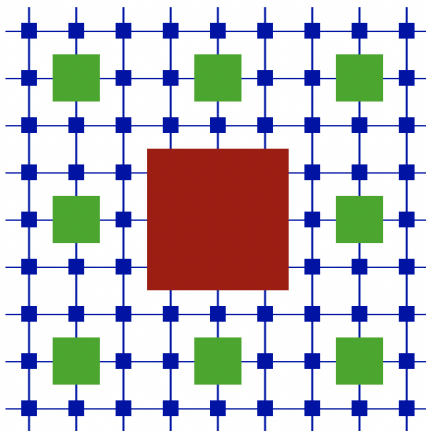


# CarpetX

Erik Schnetter <eriks@email.com>  
Lucas Timotheo Sanches <lucas.t.s.carneiro@gmail.com>  
Steve Brandt <steves@email.com>

October 12, 2023



## Abstract

**TODO: Maybe change this description**

CarpetX is a **Cactus** driver based on **AMReX**, a software framework for block-structured AMR (adaptive mesh refinement). CarpetX is intended for the **Einstein Toolkit**.

# 1 Introduction

TODO: This document should cover the topics listed here.

TODO: We should have some words explaining what **CarpetX** is.

1. `loop.hxx/where_t/ghosts_inclusive`
2. What loop functions should be used? `loop_device` or the other ones?
3. Picture of grid regions

## 2 Building and using standard images

## 3 Loops over grid elements

Contrary to **Carpet**, in **CarpetX**, loops over grid elements are not written explicitly. Operations that are to be executed for every grid element (cells, edges or points) are specified via C++ *lambda functions*, also known as closures or anonymous functions.

These objects behave like regular C++ functions, but can be defined *inline*, that is, on the body of a function or as an argument to another function.

An important concept to grasp with lambda function is *captures*. If a lambda (let us call this the child function) is defined in the body of an outer function (let us call this the parent function), the child can access variables define in the parent function, provided that these variables are *captured*. The two most relevant modes of capture while using **CarpetX** are *capture by reference* (denoted with the `&` sign in the square brackets denoting the start of the lambda) and *capture by value* (denoted by an `=` sign inside the square brackets of the lambda declaration).

When running on GPUs, capture by values are *required*. This is because data must be copied from host (CPU side) memory to device (GPU side) memory in order to be executed.

The API for writting loops in **CarpetX** is provided by the **Loop** thorn. To use it, one must add

```
1  REQUIRES Loop
```

to the thorn's `configuration.ccl` file and

```
1  INHERITS: CarpetX
2  USES INCLUDE HEADER: loop.hxx           # For using CarpetX on CPUs
3  USES INCLUDE HEADER: loop_device.hxx    # For using CarpetX on GPUs
```

to the thorn's `interface.ccl` file. Furthermore, one must include the **Loop** API header file in all source files where the API is needed by adding

```
1  #include <loop_device.hxx>
```

to the beginning of the source file.

To understand how to utilize the Loop API within Cactus scheduled functions, let us consider the following excerpt from the `schedule.cc1` file of the WaveToyX thorn, included in the CarpetX repository:

```

1  SCHEDULE WaveToyX_RHS IN ODESolvers_RHS
2  {
3      LANG: C
4      READS: state(everywhere)
5      WRITES: rhs(interior)
6      SYNC: rhs
7  } "Calculate scalar wave RHS"

```

This schedule block declares to Cactus that a C++ function (with C linkage) called WaveToyX\_RHS should be executed on the ODESolvers\_RHS schedule bin (for further information on ODESolvers, see Sec. 6).

The first few lines of C++ source for WaveToyX\_RHS read

```

1  extern "C" void WaveToyX_RHS(CCTK_ARGUMENTS) {
2      DECLARE_CCTK_ARGUMENTSX_WaveToyX_RHS;
3      DECLARE_CCTK_PARAMETERS;
4      .
5      .
6      .
7  }

```

The macros `DECLARE_CCTK_ARGUMENTSX_WaveToyX_RHS`, `CCTK_ARGUMENTS` and `DECLARE_CCTK_PARAMETERS` allow the thorn writer to access parameters and grid functions declared in the thorn's `.cc1` files. Note that Cactus now supports the `DECLARE_CCTK_ARGUMENTSX_FUNC_NAME` macro, where `FUNC_NAME` is the name of a grid function declared in the `schedule.cc1` file. These macros restrict the access of a function to its schedule-declared grid functions. More importantly, it provides a variable called `grid` which can be used to access the functionalities of the Loop API.

### 3.1 Loop regions

Before actually writing any code that iterates over grid elements, one must choose *which* elements are to be iterated over. We shall refer to the set of points in the grid hierarchy will be iterated over when a loop is executed as a *Loop region*. The following regions are defined in the Loop API:

1. All: This region refers to all points contained in the grid. Denoted in code by the `all` suffix.
2. Interior: This region refers to the interior of the grid. Denoted in code by the `int` prefix.
3. Outermost interior: This region refers to the outermost "boundary" points in the interior. They correspond to points that are shifted inwards by `= cctk_nghostzones[3]` from those that CarpetX identifies as boundary points. From the perspective of CarpetX (or AMReX), these do not belong in the outer boundary, but rather the interior. This excludes ghost faces, but includes ghost edges/corners on non-ghost faces. Loop over faces first, then edges, then corners. Denoted in code by the `outermost_int` suffix.

TODO: Picture of grid regions

## 3.2 Loop methods

Remember that the macro `DECLARE_CCTK_ARGUMENTSX_FUNC_NAME` provides a variable called `grid`, which is an instance of either `GridDescBase` or `GridDescBaseDevice` classes which contain functions for looping over grid elements on the CPU or GPU, respectively. The name of each looping method is formed according to

$$\text{loop\_} + \langle \text{loop region} \rangle + [\text{\_device}]$$

For example, to loop over boundaries using the CPU one would call

```
1  grid.loop_bnd<...>(...);
```

To obtain a GPU equivalent version, one would simply append `_device` to the function name. Thus, for example, to loop over the interior using a GPU, one would call

```
1  grid.loop_int_device<...>(...);
```

Let us now look at the required parameter of loop methods. The typical signature is as follows

```
1  template <int CI, int CJ, int CK, ..., typename F>
2  void loop_REG_PU(const vect<int, dim> &group_nghostzones, const F &f);
```

The template parameters meanings are as follows:

1. **CI**: Centering index for the first grid direction. Must be set explicitly.
2. **CJ**: Centering index for the second grid direction. Must be set explicitly.
3. **CK**: Centering index for the third grid direction. Must be set explicitly.
4. **F**: The type signature of the lambda function passed to the loop. It is not required to be set explicitly and is automatically deduced by the compiler.

Function parameter meanings are as follows:

1. **group\_nghostzones**: The number of ghost zones in each direction of the grid. This can be obtained by calling `grid.nghostzones`.
2. **f**: The C++ lambda to be executed on each step of the loop.

## 3.3 Loop Lambdas

We shall now discuss the syntax and the available elements of the lambda functions that are to be fed to the Loop methods described in Section 3.2.

To start, let us be reminded of the general syntax of a lambda function in C++:

```

1 // append ; if assigning to a variable
2 [capture_parameter] (argument_list) -> return_type { function_body }

```

When running on GPUs, the `capture_parameter` field used should always be `=`, indicating pass by value (copy) rather than `&`, indicating pass by reference. The `argument_list` of the lambda should receive only one element of type `PointDesc` (which will be described on Sec. 3.4) and the lambda must return no value, which means that `return_type` can be omitted altogether.

This means that a typical lambda passed to a loop method will have the form

```

1 [=] (const Loop::PointDesc &p) {
2     // loop body
3 }

```

### 3.4 The PointDesc type and loop lambda body

The `PointDesc` type provides a complete description of the current grid element in the loop. The following members are the ones that are expected to be used more often:

1. `I`: A 3-element array containing the grid point indices.
2. `DI`: A 3-element array containing the direction unit vectors from the current grid point.
3. `X`: A 3-element array containing the point's coordinates.
4. `DX`: A 3-element array containing the point's grid spacings.
5. `iter`: The current loop iteration.

In the body of a loop lambda, grid functions declared in the thorn's `interface.ccl` file are available as `GF3D2` objects, which are C++ wrappers around native `Cactus` grid functions. These objects are accessible by directly calling them as functions taking arrays of grid indices as input. Such indices, in turn can be obtained by directly accessing `PointDesc` members.

### 3.5 Example: Computing a RHS with finite differences

Let us now combine the elements describe thus far into a single example. Let us suppose that the following system of PDEs is implemented in `Cactus`:

$$\partial_t u = \rho \tag{1}$$

$$\partial_t \rho = \partial_x^2 u + \partial_y^2 u + \partial_z^2 u \tag{2}$$

Let us suppose that the grid functions `u` and `rho` were made available, while grid functions `u_rhs` and `rho_rhs` are their corresponding RHS storage variables. The function that computes the RHS of Eqs. (1)-(2) can be written as

```

1 extern "C" void LoopExample_RHS(CCTK_ARGUMENTS) {
2     DECLARE_CCTK_ARGUMENTS_LoopExample_RHS;
3     DECLARE_CCTK_PARAMETERS;
4
5     // The grid variable is implicitly defined via the CCTK macros
6     // A 0/1 in template parameters indicate that a grid is vertex/cell
       centered
7     grid.loop_int<0, 0, 0>(
8         grid.nghostzones,
9
10        // The loop lambda
11        [=] (const Loop::PointDesc &p) {
12            using std::pow;
13
14            const CCTK_REAL hx = p.DX[0] * p.dX[0];
15            const CCTK_REAL hy = p.DX[1] * p.dX[1];
16            const CCTK_REAL hz = p.DX[2] * p.dX[2];
17
18            const CCTK_REAL dudx = u(p.I - p.DI[0]) - 2 * u(p.I)
19                + u(p.I + p.DI[0])/hx;
20
21            const CCTK_REAL dudy = u(p.I - p.DI[1]) - 2 * u(p.I)
22                + u(p.I + p.DI[1])/hy;
23
24            const CCTK_REAL dudz = u(p.I - p.DI[2]) - 2 * u(p.I)
25                + u(p.I + p.DI[2])/hz;
26
27            u_rhs(p.I) = rho(p.I);
28            rho_rhs(p.I) = ddu;
29
30        } // Ending of the loop lambda
31    ); // Ending of the loop_int call
32 }

```

## 4 SIMD Vectorization of loops

## 5 Using flux

## 6 Time integration using ODESolvers

In `CarpetX`, time integration of PDEs via the Method of Lines is provided by the `ODESolvers` thorn. This makes time integration tightly coupled with the grid driver, allowing for more optimization opportunities and better integration.

From the user's perspective, `ODESolvers` is very similar (and sometimes even more straightforward) the MoL thorn, but a few key differences need to be observed. Firstly, not all integrators available to MoL

Name	Description
constant	The state vector is kept constant in time
Euler	Forward Euler method
RK2	Explicit midpoint rule
RK3	Runge-Kutta's third-order method
RK4	Classic RK4 method
SSPRK3	Third-order Strong Stability Preserving Runge-Kutta
RKF78	Runge-Kutta-Fehlberg 7(8)
DP87	Dormand & Prince 8(7)
Implicit Euler	Implicit Euler method

Table 1: Available methods in `ODESolvers`

are available to `ODESolvers`. The list of all supported methods is displayed in Tab. 1. Method selection occurs via configuration file, by setting

```
1 ODESolvers::method = "Method name"
```

and the default method used if none other is set is "RK2".

Additionally, users can set verbose output from the time integrator by setting

```
1 ODESolvers::verbose = "yes"
```

By default, this option is set to "no". Note that, in contrast to `MoL`, the number of parameters that are required to be set via configuration file for a successful time integration has been significantly reduced.

To actually perform time evolution, the PDE system of interest needs to be declared to `Cactus` as a set of Left-Hand Side (or LHS, or more commonly *state vector*) grid functions plus a set of Right-Hand Side (RHS) grid functions. The RHS grid functions correspond exactly to the right-hand side of the evolution equations while the state vectors stores the variables being derived in time in the current time step. More time steps can be stored internally, depending on the time integrator of choice, but this is an implementation detail that is supervised automatically by `ODESolvers`. To make this clear, consider the PDE system comprised of Eqs. (1)-(2). In this example, the state vector would be the set  $(u, \rho)$  while the right-hand side would be all elements to the right of the equal signs. Note that derivative appearing on the RHS are only derivatives in space. By discretizing space with a grid and replacing continuous derivatives with finite approximations (by using finite differences, for instance) the time-space dependent PDE system now becomes a ODE system in time, with the state vector being the sought variables. By providing the RHS of the PDE system, `ODESolvers` can apply the configured time stepping method and compute the next time steps of the state vector.

To see how `ODESolvers` is used in practice, let us turn once again to the `WaveToyX` example, bundled with `CarpetX`. To begin, let us look at this example's `interface.ccl` file

```
1 CCTK_REAL state TYPE=gf TAGS='rhs="rhs" dependents="energy error"'
2 {
3   u
4   rho
5 } "Scalar wave state vector"
6
7 CCTK_REAL rhs TYPE=gf TAGS='checkpoint="no"'
8 {
```

```

9      u_rhs
10     rho_rhs
11 } "RHS of scalar wave state vector"
12
13 CTK_REAL energy TYPE=gf TAGS='checkpoint="no"'
14 {
15     eps
16 } "Energy density"
17
18 CTK_REAL error TYPE=gf TAGS='checkpoint="no"'
19 {
20     u_err
21     rho_err
22 } "Error in scalar wave state vector"

```

In lines 1-5, the group of real grid functions called **state**, consisting of grid function **u** and **rho**, is declared. The **TYPE=gf** entry indicates that the variables in this group are grid functions (more details on Sec. 8). The **TAGS** entry is particularly important in this instance. It consists of a single quote string (marked by **'**) with space separated key-value pairs of the form **key="value"**. The **rhs="rhs"** pair indicates that these grid functions have an associated RHS group, that is, a group of variables with grid functions responsible for storing the PDE system's RHS and this group is called **"rhs"** which is defined later in lines 7-11. This information is used by **ODESolvers** while taking a time step and is tightly coupled to Cactus file parsers. **TODO: Explain the dependents key.**

## 7 Implementing boundary conditions

## 8 Writing CCL files

## 9 Adding and controlling AMR

## 10 Analyzing data

### 10.1 OpenPMD

### 10.2 SILO

## 11 Acknowledgements

## References