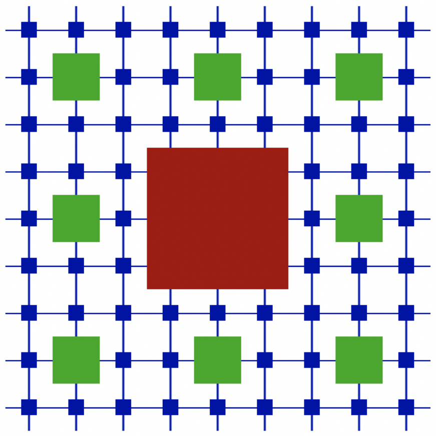
CarpetX

Erik Schnetter <eriks@email.com>
Lucas Timotheo Sanches <lucas.t.s.carneiro@gmail.com>
Steven R. Brandt <sbrandt@cct.lsu.edu>

October 17, 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 Writing CCL files

3.1 `configuration.ccl`

TODO: talk about requirements for bringing carpetx things in for accessing the APIs

3.2 `interface.ccl`

TODO: talk about new tags

3.3 `schedule.ccl`

TODO: Talk about important new schedule bins and the read/write safeguards on schedules.

3.4 Preprocessor macros

TODO: Talk about new macros and new macro features. The test below is mostly ok, butneeds to be adapted

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 `.ccl` 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.ccl` file. These macros restrict the access of a function to it's schedule-declared grid functions. More importantly, it provides a variable called `grid` which can be used to access the functionalities of the `Loop` API.

4 Loops over grid elements

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 defined 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, captures by value are *required* and captures by reference are *forbidden*. 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 writing loops in `CarpetX` is provided by the `Loop` thorn. To use it, one must add

```
REQUIRES Loop
```

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

```
INHERITS: CarpetX
USES INCLUDE HEADER: loop.hxx         # For using CarpetX on CPUs
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

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

to the beginning of the source file.

TODO: Figure out whether we need both of these header files

4.1 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 that 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

4.2 Loop methods

Loop API functions are methods of the `GridDescBase` or `GridDescBaseDevice` classes which contain functions for looping over grid elements on the CPU or GPU, respectively.TODO: Is this correct? Do we need these two classes? Can we make it with on. The macro `DECLARE_CCTK_ARGUMENTSX_FUNCTION_NAME` provides a variable called `grid`, which is an instance of either of these classes. The name of each looping method is formed according to

`loop_` + <loop region>+ [`_device`]

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

```
  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

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

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

```
template <int CI, int CJ, int CK, ..., typename F>
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 and be either 0 or 1. 0 means that this direction will be looped over grid vertices, while 1 means that it will be looped over cell centers.

2. `CJ`: Centering index for the second grid direction. Must be set explicitly and be either 0 or 1. 0 means that this direction will be looped over grid vertices, while 1 means that it will be looped over cell centers.

3. `CK`: Centering index for the third grid direction. Must be set explicitly and be either 0 or 1. 0 means that this direction will be looped over grid vertices, while 1 means that it will be looped over cell centers.

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.

Note that centering indexes can be mixed: setting the indices to $(1, 0, 0)$, for instance, creates loops over faces on the `x` direction. 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.

4.3 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 4.2.

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

```
// append ; if assigning to a variable
[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. 4.4) 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

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

4.4 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 grind point.
3. `X`: A 3-element array containing the point's coordinates.
4. DX: A 3-element array containing the point's grid spacing.
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.

4.5 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` where 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

```
extern "C" void LoopExample_RHS(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTS_LoopExample_RHS;
  DECLARE_CCTK_PARAMETERS;

  // The grid variable is implicitly defined via the CCTK macros
  // A 0/1 in template parameters indicate that a grid is vertex/cell
      centered
  grid.loop_int<0, 0, 0>(
    grid.nghostzones,

    // The loop lambda
    [=] (const Loop::PointDesc &p) {
      using std::pow;

      const CCTK_REAL hx = p.DX[0] * p.dX[0];
      const CCTK_REAL hy = p.DX[1] * p.dX[1];
      const CCTK_REAL hz = p.DX[2] * p.dX[2];

      const CCTK_REAL dudx = u(p.I - p.DI[0]) - 2 * u(p.I)
        + u(p.I + p.DI[0])/hx;

      const CCTK_REAL dudy = u(p.I - p.DI[1]) - 2 * u(p.I)
        + u(p.I + p.DI[1])/hy;

      const CCTK_REAL dudz = u(p.I - p.DI[2]) - 2 * u(p.I)
        + u(p.I + p.DI[2])/hz;

      u_rhs(p.I) = rho(p.I);
      rho_rhs(p.I) = ddu;

    } // Ending of the loop lambda
  ); // Ending of the loop_int call
}
```

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`

4.6 SIMD Vectorization of loops

TODO: See SIMDWaveToyX/src/simdwavetoyx.cxx TODO: Do the fused SIMD operations declared in Arith can be used?

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

```
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

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

By default, this option is set to `"no"`. Finally, to control the step size of the time integrator, it is possible to set the configuration parameter `CarpetX::dtfac`, which defaults to 0.5, is defined as

$$\texttt{dtfac} = \texttt{dt}/\min(\texttt{delta\_space}) \tag{3}$$

where min(`delta_space`) refers to the smallest step size defined in the `CarpetX` grid and `dt` is the time integrator step.

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, ρ) 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 an excerpt from 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  ...
```

In lines1-5, 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. 3). The `TAGS` entry is particularly important in this instance. It consists of a single quote string (marked by `'`) with space separated key-value pars 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. In lines 7-11, the `rhs` group is declared with two real grid functions, `u_rhs` and `rho_rhs`. These variables will be responsible for holding the RHS data of the PDE, which will in turn be used by `ODESolvers`.

The next step is to schedule the execution of functions into their correct schedule groups. The most relevant schedule groups provided by `ODESolvers` are `ODESolvers_RHS` and `ODESolvers_PostStep`. The former is the group where one evaluates the RHS of the state vector everywhere on the grid and the latter is where boundary conditions are applied to the state vector, and projections are applied if necessary. For example, looking at `WaveToyX`'s `schedule.ccl` file, one sees

```
SCHEDULE WaveToyX_RHS IN ODESolvers_RHS
{
  LANG: C
  READS: state(everywhere)
  WRITES: rhs(interior)
  SYNC: rhs
} "Calculate scalar wave RHS"

```

Name	Type	Possible Values	Default Value	Description
`shape_n`	String	`"sphere"` or `"cube"`	`"sphere"`	Shape of refined region
`active_n`	Boolean	`"yes"` or `"no"`	`"yes"`	Is this box active?
`num_levels_n`	Single integer	$[1, 30]$	1	Number of refinement levels for this box
`position_x_n`	Single real number	Any real	0.0	x-position of this box
`position_y_n`	Single real number	Any real	0.0	y-position of this box
`position_z_n`	Single real number	Any real	0.0	z-position of this box
`radius_n`	30 element array of reals	-1.0 or positive real	-1.0 (radius ignored)	Radius of refined region for this level
`radius_x_n`	30 element array of reals	-1.0 or positive real	-1.0 (radius ignored)	x-radius of refined region for this level
`radius_y_n`	30 element array of reals	-1.0 or positive real	-1.0 (radius ignored)	y-radius of refined region for this level
`radius_z_n`	30 element array of reals	-1.0 or positive real	-1.0 (radius ignored)	z-radius of refined region for this level

Table 2: Configuration parameters for a single (1 out of 3) box that can be defined in parameter files using the BoxInBox thorn.

```
 9 SCHEDULE WaveToyX_Energy IN ODESolvers_PostStep
10   {
11     LANG: C
12     READS: state(everywhere)
13     WRITES: energy(interior)
14     SYNC: energy
15   } "Calculate scalar wave energy density"
```

The schedule statement from lines 1-7 schedules the function that computes the RHS of the wave equation. Note that the function reads the state on the whole grid and writes to the RHS grid variables in the interior. With `CarpetX`, grid functions read and write statements are enforced: You cannot write to a variable which was declared as read only in the `schedule.ccl` file. Lines 9-15 exemplify the scheduling of a function in the `ODESolvers_PostStep` group, which is executed after `ODESolvers_RHS` during the time stepping loop. In this particular example, the scheduled function is computing the energy associated with the scalar wave equation system. These are all the required steps for using `ODESolvers` to solve a PDE system via the method of lines.

7 Implementing boundary conditions

8 Adding and controlling AMR

8.1 Box-in-box AMR

`CarpetX` supports fixed mesh refinement via the so called box-in-box paradigm. This capability is provided by the `BoxInBox` thorn. Using it is very simple and similar to `Carpet`'s `CarpetRegrid2` usage.

All configuration of boxes and levels are performed within configuration files. `BoxInBox` supports adding 3 “boxes” or “centers”. Each box can be configured as summarized in Tab. 2. The `n` suffix should be replaced by `1`, `2` or `3` for configuring the corresponding boxes. Each box can be shaped differently as either Cartesian-like cubes or spheres and support configuring up to 30 levels. Level's positions and radii can be set independently for each dimension. Note that for each box the `active`, `num_levels` and `position_(xyz)` field are stored as grid scalars. Each of the 30 refinement level radii and `x, y, z` individual radii for each box are also stored as grid arrays. This allows these parameters to be changed during a simulation run, allowing for moving boxes. This is useful, for example, when implementing a puncture tracker.

These configurations are subjected to (and restricted by) two additional `CarpetX` configurations, namely `CarpetX::regrid_every`, which controls how many iterations should pass before checking if the box grid variables have changed and `CarpetX::max_num_levels` which controls the maximum number of allowed refinement levels.

As an example, we present a configuration file excerpt for creating two refinement boxes with the `BoxInBox` thorn

```
BoxInBox::num_regions = 2

BoxInBox::num_levels_1 = 2
BoxInBox::position_x_1 = -0.5
BoxInBox::radius_x_1[1] = 0.25
BoxInBox::radius_y_1[1] = 0.5
BoxInBox::radius_z_1[1] = 0.5

BoxInBox::num_levels_2 = 2
BoxInBox::position_x_2 = +0.5
BoxInBox::radius_x_2[1] = 0.25
BoxInBox::radius_y_2[1] = 0.5
BoxInBox::radius_z_2[1] = 0.5
```

TODO: This example is not tested. Test it, maybe add a MovingBoxToy thorn to carpetx. Is this correct? Where to schedule the function that changes the boxes?

```
extern "C" void MoveBoxes(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTSX_MoveBoxes;
  DECLARE_CCTK_PARAMETERS;

  using std::sin;
  using std::cos;

  const CCTK_REAL omega = M_PI / 4;

  const auto r0 = position_x_1;
  const auto r1 = position_x_2;

  // Settings for box 1
  position_x[0] = r0 * cos(omega * cctk_time);
  position_y[0] = r0 * sin(omega * cctk_time);

  // Settings for box 2
  position_x[1] = r1 * cos(omega * cctk_time);
  position_y[1] = r1 * sin(omega * cctk_time);
}
```

The `MoveBoxes` function should then be scheduled in the `postinitial` and `poststep` bins before the `EstimateError` group defined in `BoxInBox` as

```
SCHEDULE MoveBoxes AT postinitial BEFORE EstimateError
{
  LANG: C
  WRITES: CarpetX::position_x CarpetX::position_y
} "Update refinement boxes positions"

SCHEDULE MoveBoxes AT poststep BEFORE EstimateError
{
  LANG: C
  WRITES: CarpetX::position_x CarpetX::position_y
} "Update refinement boxes positions"
```

8.2 Advanced AMR

CarpetX supports non-fixed (adaptive) mesh refinement. For cell level control of AMR, CarpetX provides user with a cell centered and non-checkpointed grid function called regrid_error. Users are responsible for filling this grid function with real value however they see fit. Once it is filled, the configuration parameter CarpetX::regrid_error_threshold controls regridding: If the values stored in regrid_error are larger than what is set in regrid_error_threshold, the region gets refined. Additionally, the configuration parameter CarpetX::regrid_every controls how many iterations should pass before checking if the error threshold has been exceeded. The parameter CarpetX::max_num_levels controls the maximum number of allowed refinement levels.

Note that CarpetX **does not** provide a "standardized" regrid error routine. This is because refinement criteria are highly specific to the problem being solved via AMR, and thus there is no one size fits all error criteria. This might seem inconvenient, but ultimately it allows for users to have higher degrees of customization in their AMR codes. For demonstration purposes, we shall now provide a routine that estimates the regrinding error as TODO: what? Provide a good starter example. This implementation could be used as a starting point for codes that wish to use different error criteria in their AMR grids.

```
  extern "C" void EstimateError(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTSX_EstimateError;
  DECLARE_CCTK_PARAMETERS;

  // The template indices indicate this a loop over cell centers
  // Remember that regrid_error is a cell centered grid function
  grid.loop_int_device<1, 1, 1>(
      grid.nghostzones,
      [=] (const Loop::PointDesc &p) {
        // TODO: Give a simple example
        regrid_error(p.I) = 0.0;
      });
}
```

Once defined, EstimateError should be scheduled in both the postinitial and poststep bins. The poststep bin gets called right after a new state vector has been calculated, and is thus the proper place to analyze it. The postinitial scheduling is also necessary for computing the initial refinement after initial conditions have been set up. A thorn making use of the regrid_error AMR mechanism should then add the following to its schedule.ccl file:

```
SCHEDULE EstimateError AT postinitial
{
  LANG: C
  READS: state(everywhere)
  WRITES: CarpetX::regrid_error(interior)
} "Estimate error for regridding"

SCHEDULE EstimateError AT poststep
{
  LANG: C
  READS: state(everywhere)
  WRITES: CarpetX::regrid_error(interior)
} "Estimate error for regridding"
```

9 Analyzing data

9.1 OpenPMD

9.2 SILO

10 Interpolation

11 Acknowledgements

References