All examples can be found in this repository

https://bitbucket.org/rhaas80/advcarpet

and can be downloaded using GetComponents

./GetComponents https://bitbucket.org/rhaas80/advcarpet/raw/master/thornlists/advcarpet.th
What is Carpet?

- the de-facto standard *driver* for Cactus
- allocates and maintains memory for all grid functions, grid arrays, grid scalars
- responsible to call scheduled routines in the proper order
- handles all parallelism between nodes
- block structure true adaptive mesh refinement with subcyling in time
- multi-patch support for multiple coordinate patches
- provides IO routines for grid functions, grid arrays, grid scalars
- handles checkpointing and recovery
- provides interpolators to interpolate data to any coordinate
- provides reduction operations similar to MPI_Reduce
- provides methods to set up a grid structure
What will be covered in this lecture?

- structure of a schedule.ccl file
- how Carpet walks the schedule
  - how Cactus' timebins are handled
  - what GLOBAL, LEVEL, SINGLEMAP modes do
  - what LOOP-LOCAL does
  - how to use the BEGIN_*_LOOP and END_*_LOOP macros
  - when OutputGH happens
- Carpet’s Reduction interface

Most of this can be found in the Carpet documentation on https://carpetcode.org/.
The Cactus schedule

```cactus
schedule WaveToyC_Evolution as WaveToy_Evolution at EVOL {
    SYNC: scalarevolve
} "Evolution of 3D wave equation"
```

```cactus
schedule WaveToyC_Boundaries as WaveToy_Boundaries at EVOL AFTER WaveToy_Evolution {
    OPTIONS: level
} "Boundaries of 3D wave equation"
```

```cactus
schedule GROUP ApplyBCs as WaveToyC_ApplyBCs at EVOL after WaveToy_Boundaries {
} "Apply boundary conditions"
```

- `schedule.ccl` tells Cactus which routines to schedule and what storage to allocate
- the union of all `schedule.ccl` files prescribe a partial ordering among all the scheduled routines → Cactus constructs an explicit ordering from this: the “schedule”
- `SYNC` statements instruct the driver to fill in points that could not be computed due to the stencil width
- `modes` tell the driver what data a scheduled function will access
Scheduling in Cactus

- Cactus itself defines INITIAL, EVOL, and ANALYSIS bins
- Bins allow independent thorns to cooperate with the Cactus flesh by defining slots for computations
- GROUPs extend this idea for thorns to define slots, e.g., MoL_CalcRHS and form the major way for thorns to interact
- Initially, there were no SYNC statements; instead, codes would call CCTK_SynGroups when needed. SYNC statements mimic this: schedule is a set of instructions, not tasks!

```
SCHEDULE Func BEFORE Func1 AFTER Func2
```
schedule.ccl syntax

A basic schedule block (see the appendices of the UserGuide for full details) looks like this:

```
schedule [GROUP] <function name|group name> IN <group>
    [AS <alias>] \ 
    [BEFORE|AFTER <function name>] \ 
{
    [LANG: <language>]
    [OPTIONS: <option>,<option>...] 
    [SYNCHRONISE: <group>,<group>...]
} "Description of function"
```

for example

```
schedule WaveToyC_Evolution as WaveToy_Evolution at EVOL
{
    LANG: C
    SYNC: scalarevolve
} "Evolution of 3D wave equation"
```
Cactus’ scheduler knows of two types of scheduled items

**GROUPS** which let you re-use a series of functions, and also define new scheduling *slots* for other thorns to schedule their function in. This is *primary* means by which Cactus thorns interact.

**FUNCTIONS** are actual C or Fortran functions that operate on the data in your simulation. Cactus needs to know which language they are written in to correctly pass information about the defined grid variables to them.

Groups that you may encounter include `MoL_CalcRHS` and `MoL_PostStep` which are used in conjunction with the `MethodOfLines` thorn to compute the right-hand-side of an evolution equation. Other groups of interest might be `ADMBase_SetADMVars` which is used as a marker after which you may assume that the metric, lapse and shift are valid. Finally some groups, most prominently `ApplyBCs` which is defined by the `Boundary` are used by client thorns to call routines in another thorn.
For historical reason, `schedule.ccl` is also the location where one activates (schedules...) storage for grid variables defined in `interface.ccl`.

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

where `timelevels` is the number of timelvels for which storage is allocated. If multiple `schedule.ccl` files request storage for the same group of grid variables, then Cactus uses the maximum of all requested timelevels. The `timelevels` value can be an actual number or the name of a parameter of the thorn.

Typically these statements are found outside of schedule blocks, but if inside then storage persists only during the execution of the schedule block. This can be used to allocate memory for temporary grid functions, but is tricky to get right.
schedule.ccl files can contain (C) if statements, that can be used to change which functions or group are included in Cactus’ schedule. Since the schedule is constructed at the very beginning of a simulation, you cannot use this to change which functions are scheduled at runtime. It also means that all information used in the if statements must be available at the beginning of the simulation, which typically means that only parameters can be used.

```c
if (<condition>) {
    [<schedule statements>]
} else if (<condition>) {
    [<schedule statements>]
} else {
    [<schedule statements>]
}
```

The condition used can be any valid C expression.
At the beginning of the run Cactus outputs the linear ordering of the all scheduled groups and routines,

\[\text{[CCTK\_EVOL]}\]

WaveToyC::WaveToy\_Evolution: Evolution of 3D wave equation
WaveToyC::WaveToy\_Boundaries: [level] Boundaries of 3D wave equation
GROUP WaveToyC\_ApplyBCs: Apply boundary conditions
  GROUP BoundaryConditions: Execute all boundary conditions
    Boundary::Boundary\_ApplyPhysicalBCs: Apply all requested local physical \ boundary conditions
    Boundary::Boundary\_ClearSelection: [level] Unselect all grid variables for \ boundary conditions
Evolve finer grids recursively
Restrict from finer grids

which is an essential help when adding new functionality to Cactus since it lets you quickly check whether your newly introduced scheduled function is called at the correct point during the evolution.
Cactus itself defines INITIAL, EVOL, and ANALYSIS bins

- bins allow independent thorns to cooperate by defining \textit{slots} for computations
- GROUPs extend this idea for thorns do define slots, e.g. MoL\_CalcRHS and form the major way for thorns to interact
- initially there were no SYNC statements, instead codes would call CCTK\_SynGroups when needed. SYNC statements mimic this: schedule is a set of instructions, not tasks!

- PUGH's domain decomposition creates one component per process
- schedule is processed exactly once per bin
- SYNC statements execute after functions, “fixes” points not computed by the function
- ApplyBCs group contain functions to apply physical (or symmetry) boundaries, need ghost zones to be valid
- SYNC and ApplyBCs form a unit
Block structured AMR works on refined regions, each of which has uniform resolution and the same memory layout as a uniform grid. This makes it very easy to adapt an existing unigrid code to block structured AMR.

A parallel AMR code involves 3 data movement operations:

- **prolongation**, which is the interpolation of data from a coarse grid into a fine grid
- **restriction**, which is the interpolation of data from a fine grid into a coarse grid
- **synchronization**, which is exchange of data within a refinement level
Recursive Procedure Integrate (level)
Repeat \((h_f/h_e)_{level}\) times
  Regridding time?—error estimation for grids at level \(level\) and finer
  Step \(\Delta t_{level}\) on all grids at level \(level\)
If \((level + 1\ exists)\nThen Begin
  Integrate \((level + 1)\)
  Update \((level, level + 1)\)
End
level = 0 (*coarsest grid level*)
Integrate \((level)\)
Mesh refinement with Carpet

Carpet provides two types of functionality that go beyond what PUGH provides

- support for multiple grid patches (maps)
- adaptive mesh refinement

To make this work with Cactus’ schedule and the established use of it by PUGH, Carpet introduces *modes*. They tell Carpet in which context to call a scheduled routine, or alternatively what a scheduled routine operates on.
Example: 2 level WaveToy (I/III)

schedule WaveToyC_Evolution as WaveToy_Evolution at EVOL {
    SYNC: scalarevolve
} "Evolution of 3D wave equation"

schedule WaveToyC_Boundaries as WaveToy_Boundaries at EVOL AFTER WaveToy_Evolution {
    OPTIONS: level
} "Boundaries of 3D wave equation"

schedule GROUP ApplyBCs as WaveToyC_ApplyBCs at EVOL after WaveToy_Boundaries {
} "Apply boundary conditions"

using the parfile gaussian-simple.par

- 1 MPI rank
- 2 levels of mesh refinement
- no multipatch
- leapfrog timestepping scheme (1 step)
- cactus_sim gaussian-simple.par | \
  gawk "/CCTK_EVOL/,/CCTK_POSTSTEP/{if(/call at/ && !/CartGrid3D/) print}"
Example: 2 level WaveToy (II/III)

[CCTK_EVOL]
WaveToyC::WaveToy_Evolution: Evolution of 3D wave equation
WaveToyC::WaveToy_Boundaries: [level] Boundaries of 3D wave equation
GROUP WaveToyC.ApplyBCs: Apply boundary conditions
    GROUP BoundaryConditions: Execute all boundary conditions
        Boundary::Boundary_ApplyPhysicalBCs: Apply all requested local physical boundary conditions
        Boundary::Boundary_ClearSelection: [level] Unselect all grid variables for boundary conditions
Evolve finer grids recursively
Restrict from finer grids
Example: 2 level WaveToy (III/III)

[r1=0] Evolution I at iteration 1 time 0.01 (global) (meta)
[r1=0] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[r1=0] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[r1=0] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[r1=0] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[r1=1] Evolution I at iteration 1 time 0.005 (global) (meta)
[r1=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[r1=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[r1=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[r1=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[r1=1] Evolution I at iteration 2 time 0.01 (global) (meta)
[r1=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[r1=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[r1=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[r1=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection
Example: 2 level WaveToy (III/III)

[rl=0] Evolution I at iteration 1 time 0.01 (global) (meta)
[rl=0] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[rl=0] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[rl=0] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[rl=0] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[rl=1] Evolution I at iteration 1 time 0.005 (global) (meta)
[rl=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[rl=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[rl=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[rl=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[rl=1] Evolution I at iteration 2 time 0.01 (global) (meta)
[rl=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[rl=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[rl=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[rl=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection
Example: 2 level WaveToy (III/III)

[r=0] Evolution I at iteration 1 time 0.01 (global) (meta)
[r=0] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[r=0] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[r=0] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[r=0] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[r=1] Evolution I at iteration 1 time 0.005 (global) (meta)
[r=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[r=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[r=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[r=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[r=1] Evolution I at iteration 2 time 0.01 (global) (meta)
[r=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[r=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[r=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[r=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection
Example: 2 level WaveToy (III/III)

[rl=0] Evolution I at iteration 1 time 0.01 (global) (meta)
[rl=0] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[rl=0] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[rl=0] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[rl=0] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[rl=1] Evolution I at iteration 1 time 0.005 (global) (meta)
[rl=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[rl=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[rl=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[rl=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection

[rl=1] Evolution I at iteration 2 time 0.01 (global) (meta)
[rl=1] Local mode call at CCTK_EVOL to WaveToyC::WaveToy_Evolution
[rl=1] Level mode call at CCTK_EVOL to WaveToyC::WaveToy_Boundaries
[rl=1] Local mode call at BoundaryConditions to Boundary::Boundary_ApplyPhysicalBCs
[rl=1] Level mode call at ApplyBCs to Boundary::Boundary_ClearSelection
Modes once more

Modes let you do things that do not fit into the Berger-Oliger notion of an operation on a grid, e.g. telling Cactus which boundary condition to apply or computing a reduction over the whole grid.
LOCAL operates on a single grid component, e.g. compute the RHS
SINGLEMAP operate son a single map patch (no real good example) zones
LEVEL operates on all grid components in a level, e.g. ghost zone exchange
GLOBAL operates on the whole grid, e.g. interpolation or reductions
LOOP-XXX call me once one each XXX in available in mode
What is available in each mode?

LOCAL  grid functions, cctk_lsh, cctk_bbox
This is the mode that is used by the majority of science codes, e.g. ML_BSSN, GRHydro, IllinoisGRMHD, WaveToy

SINGLEMAP  cctk_nghostzones, cctk_origin_space, cctk_delta_space
rarely used, could be used for the SelectBCs call

LEVEL  SYNC and the schedule happen here
rarely used, could be used to set / change grid scalars used in IF and WHILE schedule statements. CCTK_SyncGroups must be called in this mode. However Carpet will execute a SYNC in this mode, even if the SYNC statement is attached to a function of LOCAL or SINGLEMAP mode.

GLOBAL  grid scalars, grid arrays, cctk_time, cctk_delta_time
This is where CCTK_Reduce and CCTK_InterpGridArrays should be called
How Carpet traverses the schedule

FOR each iteration
  FOR each refinement level (coarse to fine)
    FOR each scheduled function
      FOR each map
        FOR each component
          call function
        SYNC ghosts
        PROLONGATE refinement boundaries
      RESTRICT to coarser level

SINGLEMAP and LEVEL modes do the “obvious” thing and replace the component and map loops by a call to the function. GLOBAL and its variants GLOBAL-EARLY and GLOBAL-LATE on the other hand are more complicated . . .
How Carpet traverses the schedule

FOR each iteration
  FOR each refinement level (coarse to fine)
    FOR each scheduled function
      FOR each map
        FOR each component
          call function
          SYNC ghosts
          PROLONGATE refinement boundaries
        RESTRICT to coarser level

SINGLEMAP and LEVEL modes do the “obvious” thing and replace the component and map loops by a call to the function. GLOBAL and its variants GLOBAL-EARLY and GLOBAL-LATE on the other hand are more complicated...
GLOBAL mode

A GLOBAL, GLOBAL-EARLY or GLOBAL-LATE function is executed along with either the functions on the coarsest (early) or finest (late) refinement level executing at this iteration. GLOBAL is *early* in EVOL and *late* in the other bins.

FOR each iteration
  FOR each refinement level (coarse to fine)
    FOR each scheduled function
      IF GLOBAL-EARLY AND coarse
        call function
      IF GLOBAL-LATE AND fine
        call function
      IF GLOBAL AND ((coarse AND EVOL) OR (fine AND NOT EVOL))
        call function

Ordering wrt LOCAL functions executing at the same iteration is given by AFTER / BEFORE conditions!
Example: modes (I/II)

```c
schedule ModesExample::Function AS LocalFunc IN CCTK_EVOL {
    "a local function"
}

schedule ModesExample::Function AS SingleMapFunc IN CCTK_EVOL AFTER LocalFunc{
    OPTIONS: singlemap
    "a singlemap function"
}

schedule ModesExample::Function AS LevelFunc IN CCTK_EVOL AFTER SingleMapFunc{
    OPTIONS: level
    "a level function"
}

schedule ModesExample::Function AS GlobalFunc IN CCTK_EVOL AFTER LevelFunc {
    OPTIONS: global
    "a global function"
}

schedule ModesExample::Function AS GlobalEarlyFunc IN CCTK_EVOL AFTER LocalFunc {
    OPTIONS: global-early
    "a global-early function"
}

schedule ModesExample::Function AS GlobalLateFunc IN CCTK_EVOL BEFORE LocalFunc {
    OPTIONS: global-late
    "a global-late function"
}
```
Example: modes (II/II)

cactus_sim modesexample.par | grep -E "Evolution I at iter|to ModesExample"

[rl=0] Evolution I at iteration 1 time 0.01 (global) (meta)
Global mode call to ModesExample::GlobalEarlyFunc
[rl=0] Local mode call to ModesExample::LocalFunc
[rl=0] Singlemap mode call to ModesExample::SingleMapFunc
[rl=0] Level mode call to ModesExample::LevelFunc
Global mode call to ModesExample::GlobalFunc

[rl=1] Evolution I at iteration 1 time 0.005
[rl=1] Local mode call to ModesExample::LocalFunc
[rl=1] Singlemap mode call to ModesExample::SingleMapFunc
[rl=1] Level mode call to ModesExample::LevelFunc
Global mode call to ModesExample::GlobalLateFunc

[rl=1] Evolution I at iteration 2 time 0.01 (global) (meta)
Global mode call to ModesExample::GlobalEarlyFunc
[rl=1] Local mode call to ModesExample::LocalFunc
[rl=1] Singlemap mode call to ModesExample::SingleMapFunc
[rl=1] Level mode call to ModesExample::LevelFunc
Global mode call to ModesExample::GlobalFunc

Schedule used:

LocalFunc
SingleMapFunc AFTER LocalFunc
LevelFunc AFTER SingleMapFunc
GlobalFunc AFTER LevelFunc
GlobalEarlyFunc BEFORE LocalFunc
GlobalLateFunc AFTER LocalFunc
Example: modes (II/II)

cactus_sim modesexample.par | grep -E "Evolution I at iter|to ModesExample"

[rl=0] Evolution I at iteration 1 time 0.01 (global) (meta)
Global mode call to ModesExample::GlobalEarlyFunc
[rl=0] Local mode call to ModesExample::LocalFunc
[rl=0] Singlemap mode call to ModesExample::SingleMapFunc
[rl=0] Level mode call to ModesExample::LevelFunc
Global mode call to ModesExample::GlobalFunc

[rl=1] Evolution I at iteration 1 time 0.005
[rl=1] Local mode call to ModesExample::LocalFunc
[rl=1] Singlemap mode call to ModesExample::SingleMapFunc
[rl=1] Level mode call to ModesExample::LevelFunc
Global mode call to ModesExample::GlobalLateFunc

[rl=1] Evolution I at iteration 2 time 0.01 (global) (meta)
Global mode call to ModesExample::GlobalEarlyFunc
[rl=1] Local mode call to ModesExample::LocalFunc
[rl=1] Singlemap mode call to ModesExample::SingleMapFunc
[rl=1] Level mode call to ModesExample::LevelFunc
Global mode call to ModesExample::GlobalFunc

Schedule used:
LocalFunc
SingleMapFunc AFER LocalFunc
LevelFunc AFTER SingleMapFunc
GlobalFunc AFTER LevelFunc
GlobalEarlyFunc BEFORE LocalFunc
GlobalLateFunc AFTER LocalFunc
Example: modes (II/II)

cactus_sim modesexample.par | grep -E "Evolution I at iter|to ModesExample"

Schedule used:

LocalFunc
SingleMapFunc AFER LocalFunc
LevelFunc AFTER SingleMapFunc
GlobalFunc AFTER LevelFunc
GlobalEarlyFunc BEFORE LocalFunc
GlobalLateFunc AFTER LocalFunc
Cactus’ Reduction interface is modelled after MPI. There are two general types of reductions, \textit{point-like}

\begin{itemize}
  \item maximum
  \item minimum
\end{itemize}

and \textit{integral like}

\begin{itemize}
  \item sum
  \item count
  \item norm2
  \item ...
\end{itemize}

Carpet provides “i” flavors of the reductions

\begin{itemize}
  \item imaximum
  \item isum
  \item icount
\end{itemize}

with the exact set being defined in CarpetReduce/src/reduce.cc.

Since these are collective MPI operations they \textit{must not} be called in LOCAL mode and \textit{should} be called in a GLOBAL mode.
Integral reductions in Carpet

The sum reduction is almost the Riemann sum definition of an integral

\[
\text{Reduce(”sum”, } \phi \text{)} \sim \sum_i \phi_i \Delta V_i
\]

with the “almost” being that one has to multiple the reduction result by `cctk_delta_space` to incorporate the physical volume of the cells.

At faces of refined regions where the centers of a fine and a coarse cell coincide and each must contribute only 1/2.

Thus the reduction is computed as:

\[
\text{Reduce(”sum”, } \phi \text{)} = \phi \times \text{weight} \times \Delta V_c + \phi \times \text{weight} \times \Delta V_f
\]

In general the value that each point contributes is weighted by its weight when accumulating the reduction result.
Since extrema are defined as values at the point, instead of weighting by the weight they simply use the weight to indicate whether to ignore a point (weight = 0) or include it (weight > 0).

Similarly the “i” reductions are identical to their non-“i” counterparts but do not take the volume of the cells into account, ie:

$$\text{Reduce}("i\text{sum}", \phi) = \phi \times \text{weight} + \phi \times \text{weight}$$
Calling the interpolator

Carpet uses the “old” interpolator interface, which is described in the UserGuide (only):

```c
int index = CCTK_VarIndex(const char *varname);
int CCTK_ReductionHandle(const char *reduction_name);
int CCTK_Reduce(const cGH *GH, /* cctkGH */
    int proc, /* who receives result */
    int operation_handle, /* operation to perform */
    int num_out_vals, /* normally 1 */
    int type_out_vals, /* CCTK_VARIABLE_REAL */
    void *out_vals, /* array for outputs */
    int num_in_fields, /* how many variables to process */
    ...); /* the varindexes of the inputs */
```

where the "..." are the results of CCTK_VarIndex calls, e.g. for WaveMoL::phi.
Example: computing the amplitude of a wave

The thorn WaveMaximum demonstrates how to use the reduction calls. To test it run `cactus_sim gaussian-maxium.par` which outputs

```
INFO (WaveMaximum): Found maxval of 1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Time</th>
<th>*MUM::maxval</th>
<th>WAVETOY::phi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>minimum</td>
<td>maximum</td>
</tr>
<tr>
<td>0</td>
<td>0.000</td>
<td>1.000000000</td>
<td>2.678637e-33</td>
</tr>
<tr>
<td>1</td>
<td>0.005</td>
<td>0.9850748</td>
<td>6.618259e-33</td>
</tr>
<tr>
<td>2</td>
<td>0.010</td>
<td>0.9555940</td>
<td>1.318430e-32</td>
</tr>
</tbody>
</table>
```

INFO (WaveMaximum): Found maxval of 0.955594
schedule WaveMaximum_Maximum IN CCTK_ANALYSIS
{
  LANG: C
  OPTIONS: GLOBAL
} "Compute maximum of wave"

void WaveMaximum_Maximum(CCTK_ARGUMENTS) {
  DECLARE_CCTK_PARAMETERS;
  int max_handle = CCTK_ReductionHandle("maximum");
  int varidx = CCTK_VarIndex("WaveToy::phi");
  int ierr = CCTK_Reduce(cctkGH,
    -1,   /* make result available to all procs */
    max_handle, /* reduction op to perform */
    1,    /* number of outputs (usually same as inputs) */
    CCTK_VARIABLE_REAL, /* output type for all outputs */
    maxval, /* pointer to output */
    1,    /* 1 input grid variables (functions, scalars or or arrays) */
    varidx, /* the variable indices of the grid variables, as with printf */
  );
  CCTK_VINFO("Found maxval of %g", (double)*maxval);
}
Carpet offers macros to control macros that let C code explicitly control the modes and loops. The first set is `BEGIN XXX LOOP...END XXX LOOP` where XXX is one of `REFLEVEL` to loop over refinement levels, `coarse to fine`, `LOCAL_MAP` to loop over all maps, *which exists on this MPI rank*, `LOCAL_COMPONENT` to loop over all components, *which exist on this MPI rank*. All `BEGIN XXX LOOP` macros take `cctkGH` as their sole argument, `END XXX LOOP` takes no argument.

Inside the loop use `DECLARE_CCTK_ARGUMENTS` to set up `cctkGH` and grid function variables:

```c
BEGIN_LOCAL_COMPONENT_LOOP(cctkGH) {
    DECLARE_CCTK_ARGUMENTS;
    phi[CCTK_GFINDEX3D(cctkGH, i,j,k)] = 42.;
} END_LOCAL_COMPONENT_LOOP;
```
A set of macros `ENTER XXX_MODE...LEAVE XXX_MODE` exist that escape from whichever mode Carpet is in and enter the desired mode. This is useful to for example perform a global `CCTK_Reduce` call from within a `LOCAL` routine. All `BGEIN XXX_MODE` macros take `cctkGH` as an argument.

These macros are:

- `BEGIN_GLOBAL_MODE...END_GLOBAL_MODE`
- `BEGIN_LEVEL_MODE...END_LEVEL_MODE`
- `BEGIN_SINGELEMAP_MODE...END_SINGELEMAP_MODE`
- `BEGIN_LOCAL_MODE...END_LOCAL_MODE`

and after the `END XXX_MODE` macros, Carpet is back to the original mode (and level, map and component).
All macros (and some more) are defined in Carpet/src/modes.hh and accessible by requesting the carpet.hh header in interface.ccl:

USES INCLUDE: carpet.hh

and adding a

#include "carpet.hh"

to the C++ code.
There's no example, instead his is part of of the hands-on activity.
Questions?