Using Richardson Extrapolation in SAMRAI

Introduction

Support for Richardson extrapolation error estimation has been enhanced in SAMRAI version 1.2. Richardson extrapolation may be combined with other cell refinement methods, such as gradient detection or refining static regions. This document is divided into three sections. The first section describes class organization for refinement options, the second describes the implementation of the Richardson extrapolation algorithm, and the last section gives examples of how to use the various tagging options. Send questions and requests for additional information via email to `samrai@llnl.gov`.

Class organization

The class `GriddingAlgorithm` drives the overall process of creating levels in an AMR hierarchy as well as regridding individual levels. This class requires operations, supplied through a Strategy pattern interface called `TagAndInitializeStrategy`, that tag cells on a level for refinement. The class `StandardTagAndInitialize` is a specific instantiation of that interface that provides common cell tagging operations for structured AMR. As the name implies, this class performs two main functions, tagging cells for refinement and initializing data on a new patch hierarchy level. The class supports three methods for selecting cells for refinement: gradient detection, Richardson extrapolation, and refining static box regions specified in an input file. To supply alternative refinement and initialization routines, one can implement a new subclass of the `TagAndInitializeStrategy` interface.

Problem-specific operations needed to select cells for refinement are supplied to the `StandardTagAndInitialize` class via the Strategy pattern interface called `StandardTagAndInitStrategy`. This class declares methods used to initialize a level: `itializeLevelData()`and `resetHierarchyConfiguration()` and to tag cells for refinement `applyRichardsonExtrapolation()` and `applyGradientDetector()`. Several other virtual functions needed for the Richardson extrapolation algorithm also appear in the interface since Richardson extrapolation needs to interact with integration routines. The methods are:

- `advanceLevel()`
- `resetTimeDependentData()`
- `resetDataToPreadvanceState()`
- `coarsenDataForRichardsonExtrapolation()`

Note that several of these methods are shared by the `TimeRefinementLevelStrategy` interface which declares problem-specific time integration operations needed by the class `TimeRefinementLevelIntegrator`. For example, concrete implementations of these abstract methods are provided in `HyperbolicLevelIntegrator`, which is derived from `StandardTagAndInitStrategy`. Figure 2 shows the class layout for the case of the Euler example application.

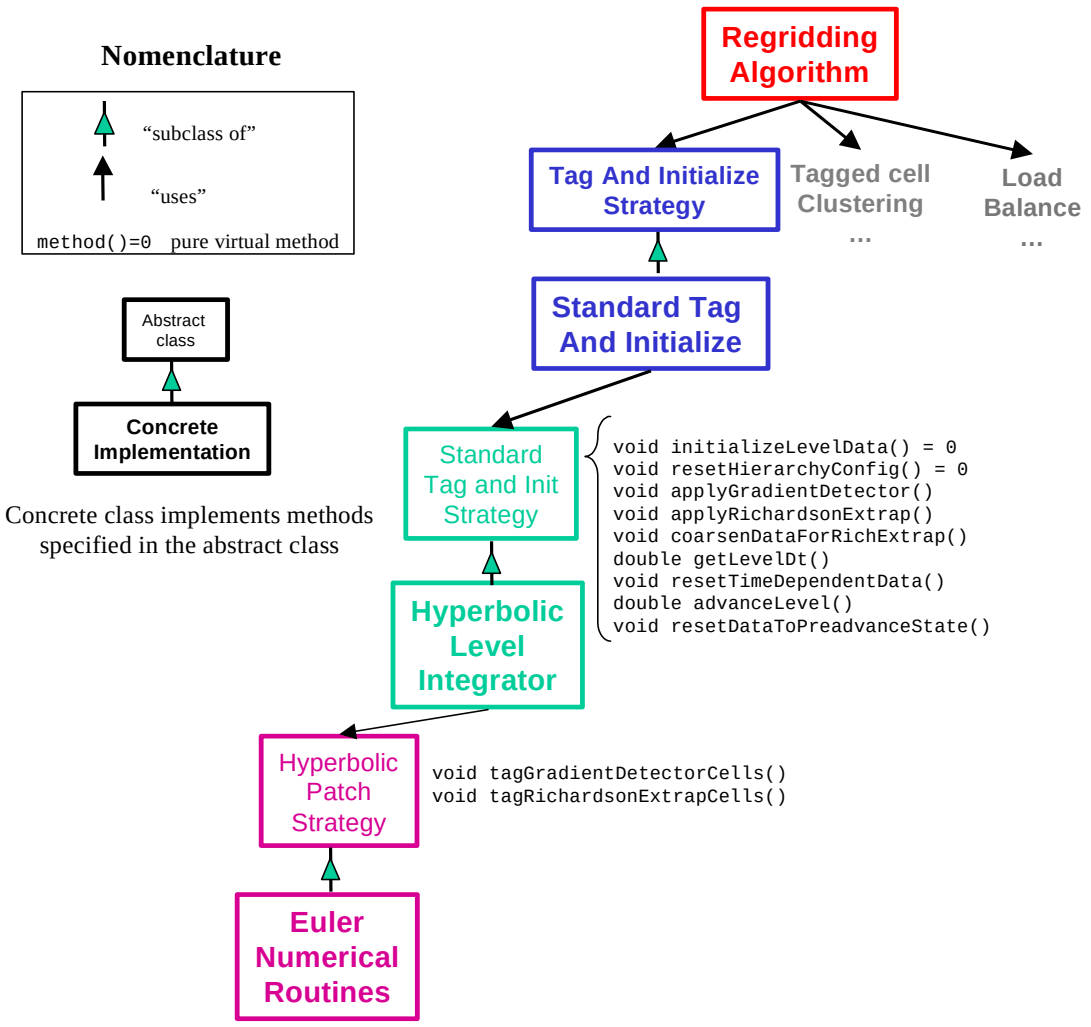


Figure 1. Class structure for Euler application

Note the two methods to initialize a new level are declared *pure virtual* since they must be provided in all cases. The rest of the methods are NOT *pure virtual* and are supplied with default "empty" implementations that will simply "drop through" when called if no implementation is provided in a subclass. As a result, users only need to supply operations for the refinement strategy they want. However, care must be exercised by users to supply all methods required by the desired refinement strategy since, for example, the compiler will not complain if one does not supply a needed method. We have tried to add sufficient error checking and warning messages to help users catch these problems at run-time.

Richardson Extrapolation

This section discusses the use of Richardson extrapolation as it is implemented in the classes described earlier. We note that there are other ways to implement the Richardson extrapolation procedure. For concreteness, we discuss the interaction of error estimation and time integration operations using the `HyperbolicLevelIntegrator class`. As it is implemented in

SAMRAI, Richardson extrapolation can be used easily with any level integration scheme that couples to the `TimeRefinementIntegrator` class.

During Richardson extrapolation error estimation, the solution is advanced in time in two different ways for the hierarchy level under consideration for refinement. The first advance occurs on the level itself. The second occurs on a coarsened version of that level. These two solutions are then compared to determine which cells to refine. In contrast, gradient detection performs no integration of the data. Although Richardson extrapolation is more expensive computationally than gradient detection, Richardson extrapolation can be made generic with respect to the magnitude of the quantities involved and if less heuristically-based than methods that attempt to refine around large gradients.

The bulk of the Richardson extrapolation algorithm itself is implemented in two methods in the `StandardTagAndInitialize` class: `preprocessRichardsonExtrapolation()` and `tagCellsUsingRichardsonExtrapolation()`. The first of these methods is used to integrate the data on a coarsened version of the level. The second method integrates the data on the level itself and calls a user routine to compare the two solutions. The description below summarizes the major steps performed in the algorithm. Steps 1 – 2 are performed in `preprocessRichardsonExtrapolation()` and steps 3 – 7 are performed in `tagCellsUsingRichardsonExtrapolation()`.

1. **Create a coarser version of the level on which cells are being selected for refinement.** The ratio used to generate the coarser level is called the `error_coarsen_ratio`. This integer is computed as the greatest common divisor of the refinement ratio relating the current hierarchy level to its next coarser (for level zero, we use the ratio between it and level one). The current algorithm only allows coarsen ratios of 2 or 3.

   Solution data is set on the coarser level by the time integration class in the `coarsenDataForRichardsonExtrapolation()` method. It is important to be aware of the error estimation time and the time associated with the solution data in order to apply the algorithm correctly. For example, the state of the data during this coarsening phase in `HyperbolicLevelIntegrator` is represented in Figure 3. At initialization time (i.e., when the AMR hierarchy is constructed initially) all data is at the simulation start time (e.g., t = 0). At other regridding times, solution data will exist at two different times indicated with *variable context*s "CURRENT" and "NEW" in the hyperbolic integrator. During the data coarsening phase in either case, we coarsen the "CURRENT" data on the hierarchy patch level to the "CURRENT" data on the coarsened version of this level. Thus, the data on the coarsened level is in a proper state for time integration. We note that the data coarsening function `coarsenDataForRichardsonExtrapolation()` is called twice during the Richardson extrapolation process. A boolean flag called "before_advance" in the argument list distinguishes the two cases. The flag is true for the current case in which we initialize the data for the coarsened hierarchy level.

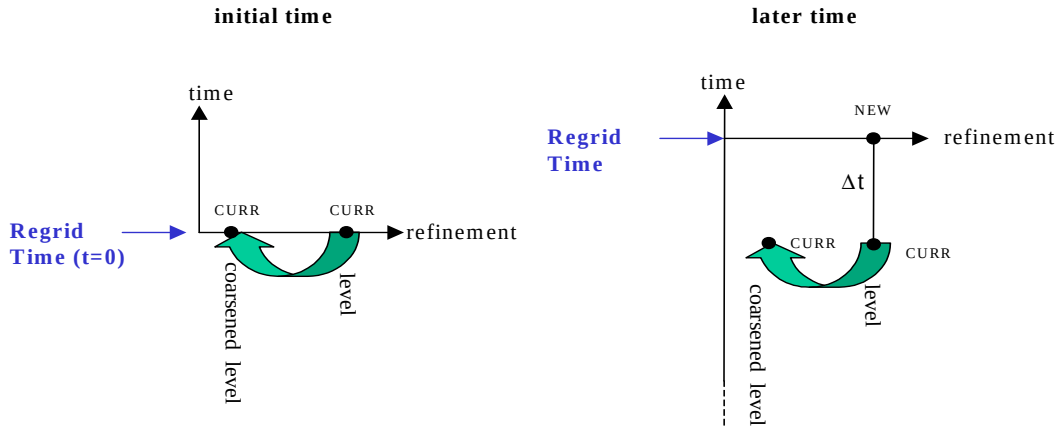


Figure 3. Coarsen "current" data from level being regridded to a coarser version of the level.

2. **Advance data on the coarser level.** After initializing the data on the coarser level, we integrate the coarser level over a single time increment defined by Δt_{coarse} = (`error_coarsen_ratio` * Δt). Here Δt is the most recent time increment used to advance the solution on the hierarchy level under consideration. This operation also occurs in the method `preprocessRichardsonExtrapolation()`. At initialization time, the solution will be advanced on the coarser level to time t_{regrid} + error_coarsen_ratio * Δt. At later times, we advance to time t_{regrid} + (error_coarsen_ratio-1) * Δt. See Figure 4.

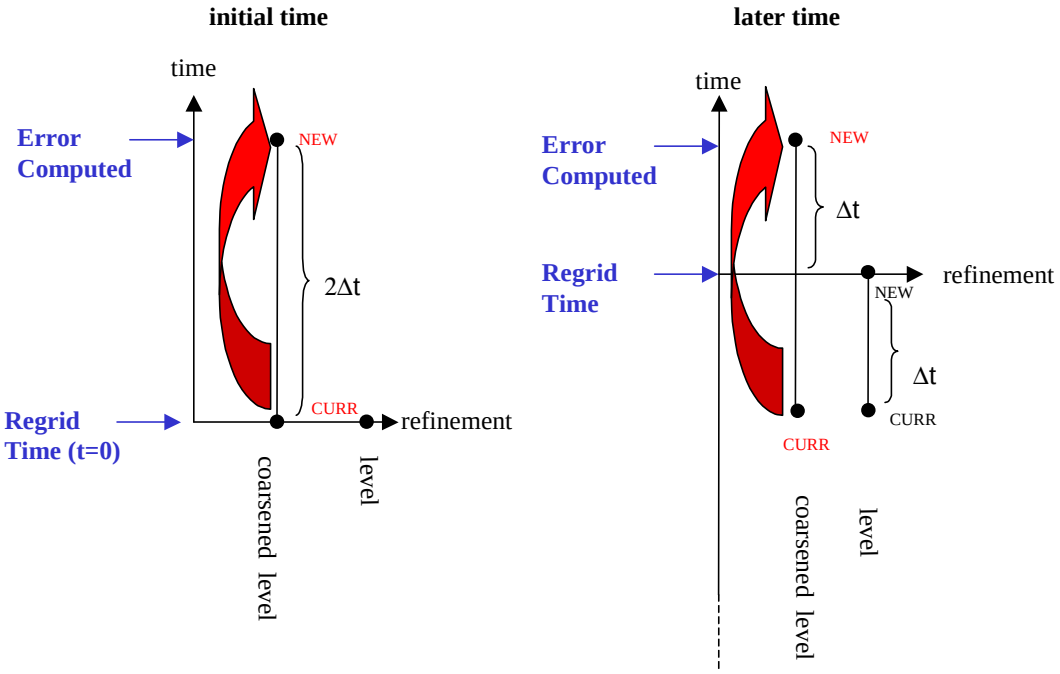


Figure 4. Advance data in time on coarser level

3. **Advance data on the hierarchy level on which error estimation is performed.** So that we can compare two different solutions at the same integration time, the solution on

hierarchy level is integrated to the same time as that on the coarsened version of the level. At initialization time, we integrate over `error_coarsen_ratio` steps of size Δt. At later times, we integrate over `error_coarsen_ratio-1` steps of size Δt. Note that at initialization time, no integration steps have been performed on the level before error estimation is called, so we have only "CURRENT" data to advance. At later times, we have both "CURRENT" and "NEW" data. "CURRENT" data corresponds to the initial integration time for the coarser level. "NEW" data corresponds to the regrid time. Before we can advance the data on the hierarchy level, the solution data must be reset. This is done by the integrator in the function `resetTimeDependentData()`.

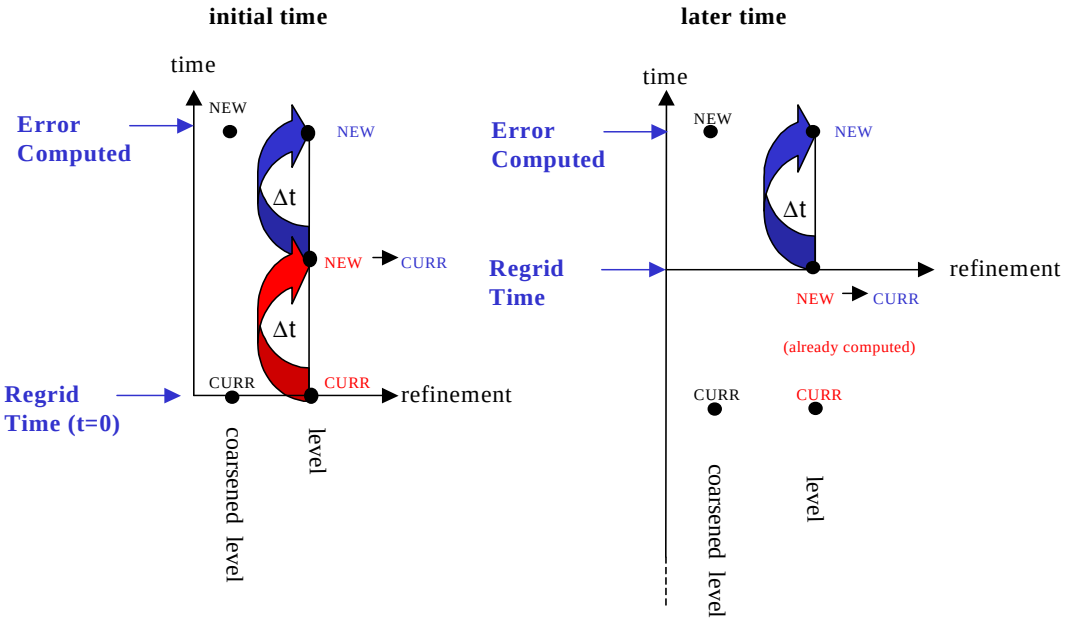


Figure 5. Advance data on level.

4. **Coarsen integrated solution data from hierarchy level to the coarsened level.** Now that we two solutions at the same time, one on the hierarchy level and one on a coarsened version of the hierarchy level, we need to move one of these to the level holding the other so that we can compare the two on a grid with the same spatial resolution. This is done using the `coarsenDataForRichardsonExtrapolation()` routine. Recall that this routine contains a boolean value "before_advance" in its argument list to indicate where in the algorithm it is called. For the case we are now describing, the boolean is false. In `HyperbolicLevelIntegrator`, the data is coarsened from the "NEW" context to the "NEW" context.

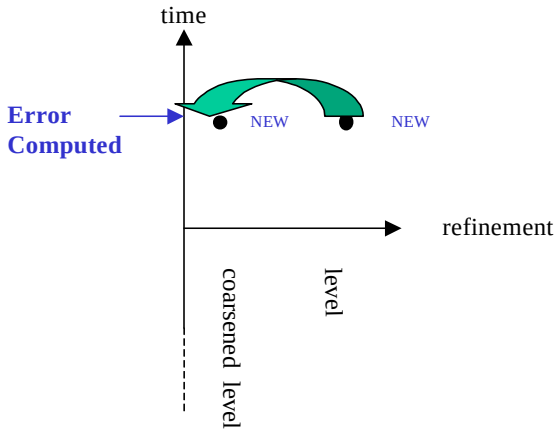


Figure 6. Coarsen data advanced on level to the coarsened level.

5. **Tag cells on the coarser level.** Comparing the two solutions, we identify cells to tag for refinement (or de-refinement). For example, the `HyperbolicLevelIntegrator` calls the `tagRichardsonExtrapolationCells()` method in hyperbolic patch strategy interface for this.

6. **Refine tag data from the coarsened level to the hierarchy level subject to error estimation.** We now have tagged cells for refinement on the coarser level, we need to refine the tags to the hierarchy level so that a refinement of that level can be generated. We simply copy the tagged cells on the coarsened level to cells covering the same region on the hierarchy level.

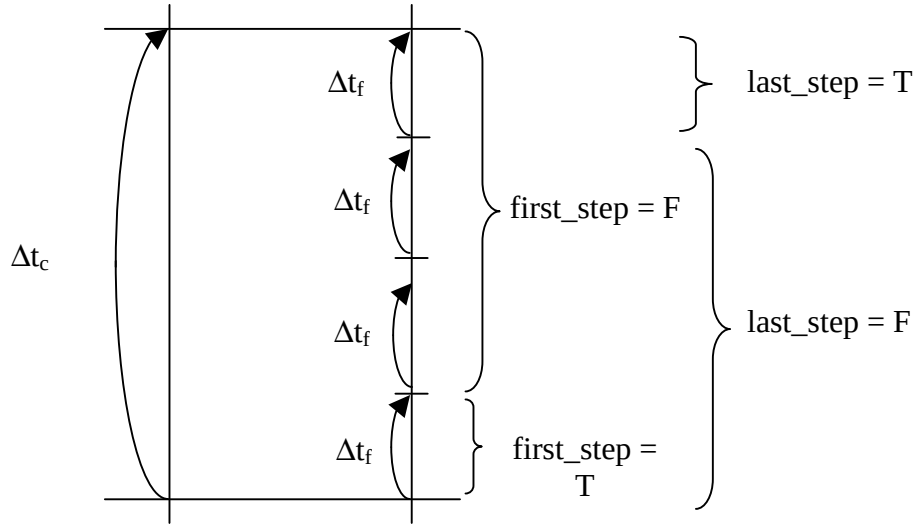
7. **Reset data on the hierarchy level to a state suitable for the next time integration step.** The algorithm calls the method `resetDataToPreadvanceState()` for this.

In the case of a hyperbolic problem like the Euler example, the Richardson extrapolation algorithm calls methods shared by two interfaces `TimeRefinementLevelStrategy` and `StandardTagAndInitStrategy`. These are described in the following table:

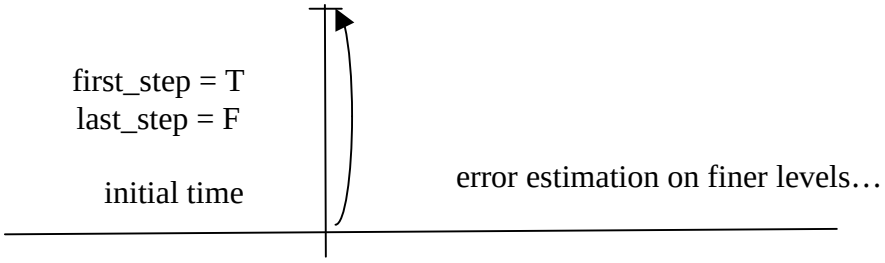
method name	What it does…
`advanceLevel()`	Advance solution one timestep
`resetTimeDependentData()`	Reset data pointers after advance

The `advanceLevel()` method is used in a number of situations that require different behavior. Four booleans in the argument list, `first_step`, `last_step`, `regrid_advance`, `level_in_hierarchy`, help to distinguish the cases describe next:

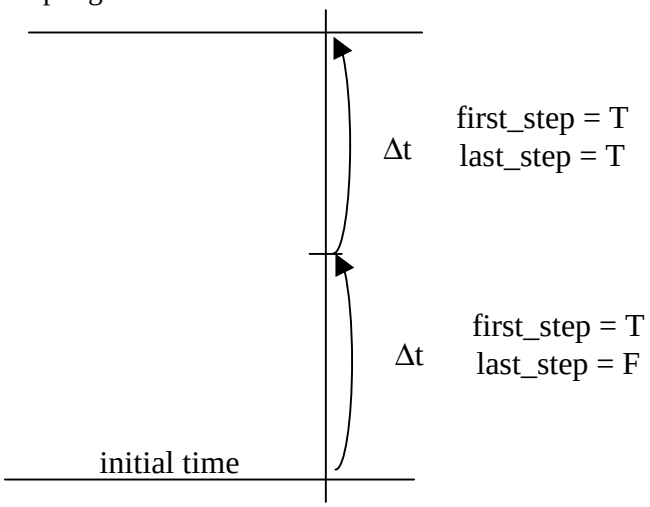
1. **Regular time integration advance.** During the standard integration process (i.e., not during regridding), the first and last step flags identify where we are in the timestep sequence, where the sequence is defined by the steps taken between consecutive advance steps on the next coarser level. In this case, `regrid_advance` is false and `level_in_hierarchy` is true.



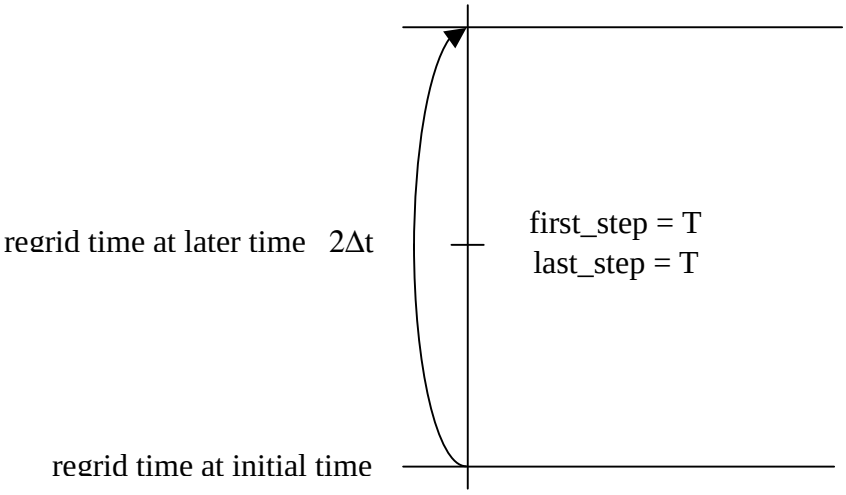
2. **Advance a level at *initial time* to supply boundary conditions before Richardson extrapolation error estimation on finer levels.** Since Richardson extrapolation advances the data, it may require time-dependent boundary values from coarser levels in the hierarchy. Thus, coarser levels must be integrated in time during the initial construction of the hierarchy when Richardson extrapolation is performed. Since this step is performed at initialization for regridding on a level in the hierarchy, `first_step` is true, `last_step` is false, `regrid_advance` is true, and `level_in_hierarchy` is true.



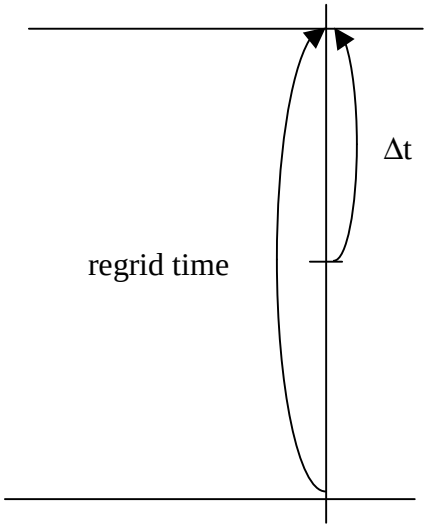
3. **Advance a level at initial time during Richardson extrapolation.** This corresponds to step 3 in the sequence of Richardson extrapolation steps above. Here, `regrid_advance` is true, and `level_in_hierarchy` is true. The values of the first and last step arguments are illustrated below:



4. **Advance on coarsened version of the level during regridding.** This corresponds to step 2 in the Richardson extrapolation algorithm described above. Only a single advance step is taken on the coarse level so `first_step` and `last_step` are both true in this case. The `regrid_advance` argument is also true, since this advance is applied during regridding. Since this advance is applied to the coarsened version of the level being regridded, which is NOT in the hierarchy, `level_in_hierarchy` is false. The only difference in this case at the initial time and at a later time is the actual regrid time with respect to the time of the data during the time integration. This is illustrated below:



5. **Advance on level being regridded at *later time*.** This corresponds to step 3 (at later time only) in the Richardson extrapolation algorithm above. Here, `first_step` is false when the level is the coarsest level in the hierarchy to synchronize with other levels at the regrid time. Otherwise, it is true. The `last_step` argument is true. The `regrid_advance` argument is true. The `level_in_hierarchy` argument is true.



Richardson extrapolation error criterion

If your application code employs the `GriddingAlgorithm` and `HyperbolicLevelIntegrator` classes in SAMRAI and you wish to use Richardson extrapolation, the only method you need to add to your code is `tagCellsForRichardsonExtrapolation`() method in you concrete hyperbolic patch strategy subclass. In this method, you will compare the solution advanced on the coarser level and the solution advanced on the finer level and then coarsened to the coarser level and tag cells where this comparison indicates a sufficiently high truncation error.

Note that standard usage of Richardson extrapolation for error estimation in SAMR for time dependent problems assumes that the spatial and temporal truncation error of the numerical methods is the same order. When this is the case, we can assume that the local truncation error of the time integration scheme is

$$\varepsilon = C\Delta t^{n+1} \tag{1}$$

where n is the global order in time of the scheme and C is some (generally unknown) constant. If we take r steps of size Δt on a fine level, and one step of size $r\Delta t$ on a coarser level, the truncation errors on the coarse and fine levels will be:

$$\varepsilon_f = C \cdot r\Delta t^{n+1} \qquad\qquad \varepsilon_c = C(r\Delta t)^{n+1} = Cr^{n+1}\Delta t^{n+1}$$

The local truncation error may therefore be estimated as

$$\varepsilon = C\Delta t^{n+1} = \frac{\varepsilon_c - \varepsilon_f}{r^{n+1} - r} \tag{2}$$

Usually, we wish to tag where the global error – the error accumulated over the course of the simulation – is greater than some specified tolerance. To estimate the global error we multiply

the local truncation error by an estimate of the total number of timesteps we will take in the simulation. We can estimate the number of advance steps using a simple formula

$$\#steps = \frac{L}{s\Delta t}$$

where L is a characteristic length scale associated with the problem domain, and s is some characteristic propagation speed (i.e., wave speed), and Δt is the current timestep. This implies that we should tag cells for which:

$$\frac{\varepsilon_c - \varepsilon_f}{r^{n+1} - r} \approx \frac{\left|\varpi_c - \varpi_f\right|}{r^{n+1} - r} \frac{L}{s\Delta t} > tol \tag{3}$$

where ω is the quantity being monitored for errors.

See the source code for either the Euler or linear advection sample problem for a specific example that uses Richardson extrapolation.

Specifying tagging options and other users issues

The `StandardTagAndInitialize` class allows one to use a combination of static refine regions, gradient detection, and Richardson extrapolation. This section summarizes the different ways in which these options may be invoked.

To specify tagging options through input, pass an input database to the constructor to `StandardTagAndInitialize`. The input key `tagging_method` allows one to indicate the tagging criteria using an array of strings. Valid string choices are "GRADIENT_DETECTOR", "RICHARDSON_EXTRAPOLATION", and "REFINE_BOXES". You may use combinations of the gradient detector, Richardson extrapolation, and static refine boxes. If refine boxes are chosen, you must supply a `RefineBoxes{ }`database entry which specifies the prescribed regions where refinement is to occur. Note that the entries specify regions to refine on a given level; they do not specify the level itself. The following example shows how to use an input file to combine a gradient detector with static refine boxes defined on the first two levels in a hierarchy.

foo.C

```
Pointer<StandardTagAndInitialize> tagging_alg =
     new StandardTagAndInitialize(
          "StandardTagAndInitialize",
               hyp_level_integrator,
          input_db->
               getDatabase("StandardTagAndInitialize"));
```

input.file

```
StandardTagAndInitialize{
     tagging_method = "GRADIENT_DETECTOR", "REFINE_BOXES"
     RefineBoxes{
```

```
        level_0 = [(15,0),(29,14)], [(18,15),(29,20)]
        level_1 = [(65,10),(114,40)]
}
```

The user-specified refine boxes can also be modified to change over course of the the simulation. This functionality requires the user to supply a specified time interval along with the set of refine boxes for each time interval. For example, to change refine boxes from a first to second set at time 1.0, supply the following for the "RefineBoxes" input entry

```
RefineBoxes{
    Level0{
        times = 0.0, 1.0
        boxes_0=[(15,0),(29,14)], [(18,15),(29,20)]
        boxes_1=[(18,3),(32,17)], [(21,18),(32,23)]
    }
    Level1{
        times = 0.0, 1.0
        boxes_0 = [(65,10),(114,40)]
        boxes_1 = [(68,13),(117,43)]
    }

}
```

See comments in the header file for the `TagAndInitializeStrategy` class for further information on ways to specify adaptive refinement through input over designated time or cycle intervals in the simulation.

Input files in the Euler and linear advection sample problems provide additional examples of cell tagging options specified via input. Also, see the header file comments in the `StandardTagAndInitialize` class for more details about input file options for regridding.

It is important to note that the `StandardTagAndInitialize` class manages only the algorithms used for selecting cells for refinement. The actual tagging of cells for a gradient detector or Richardson extrapolation must be implemented by the user. These operations are supplied through the interface methods in the `StandardTagAndInitStrategy` class. The Euler and linear advection sample problems illustrate how this is done when using the `HyperbolicLevelIntegrator` class. No user routines are needed when static refine boxes are used.

Some users problems may wish to change the tagging criteria at different points in the simulation (e.g. start the simulation with static refine boxes but turn them off at some point). For these instances, the `StandardTagAndInitialize` class provides methods to turn refinement options on and off. For example, the default state of a `StandardTagAndInitialize` object (no input database provided), will set the tagging type to be gradient detector. We can add to the gradient detector a set of static refine boxes using the calls:

```
tagging_alg->turnOnRefineBoxes();
BoxArray l0_rboxes = …                  // refine boxes for level 0
tagging_alg->resetRefineBoxes(l0_rboxes, 0);
```

At a later time, the boxes can be reset to something else, or the refine boxes may be turned off:

```
tagging_alg->turnOffRefineBoxes();
```

Similar methods are available to turn on/off gradient detectors and Richardson extrapolation. See the `StandardTagAndInitialize` class header documentation for more details.

Regridding Boxes I/O

The `GriddingAlgorithm` class allows one to read and write regrid boxes to/from a data file. For example, the following input entry:

```
GriddingAlgorithm {
      …
      write_regrid_boxes = TRUE
      regrid_boxes_filename = "regrid_boxes"
}
```

will output the level boxes constructed during the regridding process each time a hierarchy level is constructed to the file names "regrid_boxes". If we switch "write_regrid_boxes=TRUE" to "read_regrid_boxes = TRUE", the gridding algorithm will read the set of level boxes from the file. Then, rather than performing the usual tag operations to generate levels in the hierarchy, the boxes read from the file will be used. We find this useful for testing the performance of the library in certain circumstances. However, this capability is not designed for general consumption and requires some care to use. For more information about this feature, please consult the SAMRAI team (samrai@llnl.gov).

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.
Document UCRL-TM-202188.