Creating a Multiblock Geometry in *SAMRAI*

Noah S. Elliott

1 Introduction

The multiblock functionality in *SAMRAI* is intended to allow the use of *SAMRAI*'s structured AMR infrastructure on problem domains that are not logically rectangular on the whole, but can be decomposed into logically rectangular subdomains, or blocks. This document describes the steps needed to set up a multiblock domain and provides some information on some multiblock-specific usage needed in the filling of data.

2 Creating a `GridGeometry` for a mutliblock mesh

This section describes how to use input set up an object of class `GridGeometry` that is able to manage a domain for a multiblock problem. Figure 1 shows an example domain, consisting of five blocks, that will be used for reference in this section.

2.1 Defining the Index Spaces for the Blocks

`GridGeometry` is a concrete class inheriting from `BaseGridGeometry` that is capable of managing a domain for a multiblock problem. The input database used to construct a `GridGeometry` provides a set of boxes that describe the index spaces of each of the blocks in the multiblock domain. Each block has an index space which is defined independently from all other blocks and is identified by an integer block number, which is usually accessed via the class `BlockId`

The following example shows how the index spaces for each block are specified for the domin shown in Figure 1:

```
GridGeometry {
   num_blocks = 5

   domain_boxes_0 = [ (0,0) , (7,8) ]
   domain_boxes_1 = [ (0,0) , (6,8) ]
   domain_boxes_2 = [ (0,0) , (6,4) ]
   domain_boxes_3 = [ (0,0) , (4,6) ]
   domain_boxes_4 = [ (0,0) , (7,6) ]

   ...
}
```

The integer suffixes for each entry are used to assign the block number for each block. The values must be from 0 to n-1, n being the number of blocks.

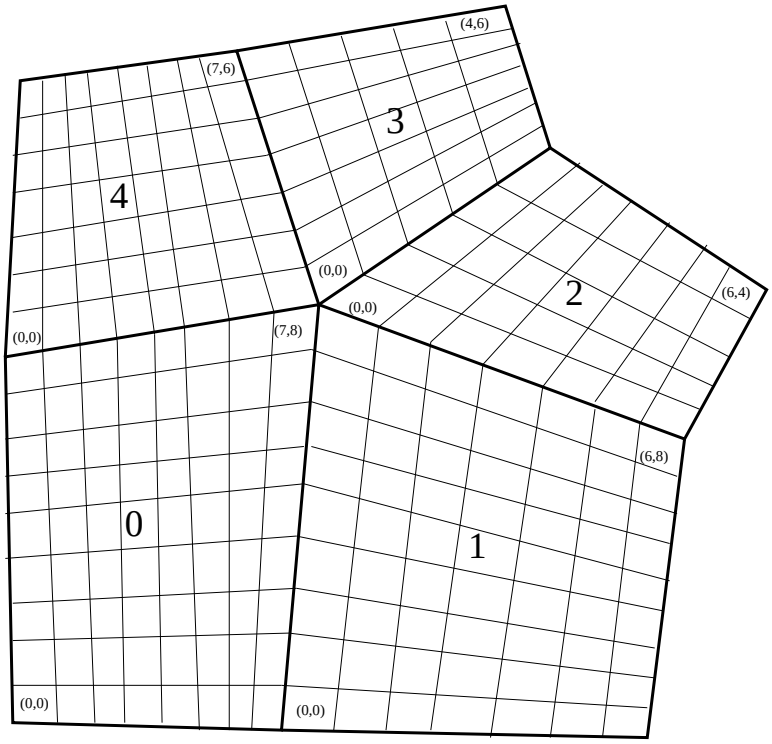

Figure 1: Example 5 block domain

2.2 Input to describe singularities

In addition to index space input, a multiblock `GridGeometry` also requires input that is used to define the relationships and relative locations of the different blocks within the multiblock domain. First, a block of input is required to describe the location of each singularity point.

```
GridGeometry {
   ...

   Singularity0 {
      blocks = 0,1,2,3,4

      sing_box_0 = [(8,9),(8,9)]
      sing_box_1 = [(-1,9),(-1,9)]
      sing_box_2 = [(-1,-1),(-1,-1)]
      sing_box_3 = [(-1,-1),(-1,-1)]
      sing_box_4 = [(8,-1),(8,-1)]
   }

   ...
}
```

For each singularity point, the `blocks` input specifies which blocks touch the singularity, and the `sing_box_*` inputs tell where each block abuts the singularity. In two dimensions, the required input is

the single-cell box that lies immediately outside the block and touches the block only at the singularity point. In three dimensions, a singularity can occur at either a corner or an edge. If the singularity is a corner, then the input is the same as in two dimensions. If it is on an edge, the the input box must be a box that runs along the singularity edge and is width one in the other two directions. This box also must be located immediately outside the block and touch the block only along the singularity edge.

2.3 Defining Neighbor relationships

The remaining input for `GridGeometry` describes the relationships between neighboring blocks. Blocks are said to be neighbors if they abut each other at any point, edge or face. Every pair of neighbors must be specified in the `GridGeometry` input. Shown here are some of the `BlockNeighbors` entries needed for the Figure 1 example.

```
GridGeometry {

   ...

   BlockNeighbors0 {
      block_a = 0
      block_b = 1

      rotation_b_to_a = "I_UP", "J_UP"
      point_in_a_space = 8,0
      point_in_b_space = 0,0
   }

   BlockNeighbors1 {
      block_a = 0
      block_b = 2

      rotation_b_to_a = "I_UP", "J_UP"
      point_in_a_space = 8,9
      point_in_b_space = 0,0
      is_singularity = TRUE
   }

   BlockNeighbors2 {
      block_a = 1
      block_b = 2

      rotation_b_to_a = "I_UP", "J_UP"
      point_in_a_space = 0,9
      point_in_b_space = 0,0
   }

   BlockNeighbors3 {
      block_a = 2
      block_b = 3

      rotation_b_to_a = "J_DOWN", "I_UP"
```

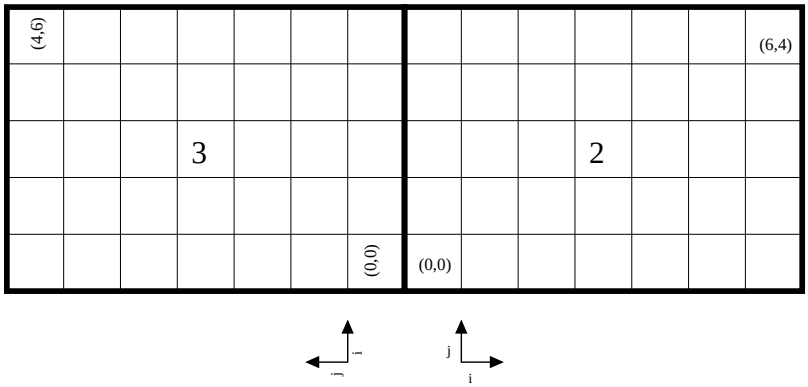

Figure 2: Index spaces of blocks 2 and 3

```
      point_in_a_space = -1,0
      point_in_b_space = 0,0
   }

   ...

}
```

In each `BlockNeighbors` entry, one block is arbitrarily chosen to be `block_a` and the other `block_b`. `rotation_b_to_a` is used to specify how block b's index space is aligned in comparison to block a's. As an example, for the `BlockNeighbors3` entry above, we consider only the two blocks in question, 2 and 3, as if the other blocks did not exist. Figure 2 shows the index spaces of the blocks 2 and 3 and the respective alignments of their i-j axes.

The positive i direction on block 2 is equivalent to the negative j direction on block 3. Thus the first entry for `rotation_b_to_a` is `"J_DOWN"`. Likewise the positive j direction on block 2 is equivalent to the positive i direction on block 3, so the second entry is `"I_UP"`. In general, the entries for `rotation_b_to_a` are determined by travelling in the positive direction for each dimension in block a, and identifying the equivalent axis and direction on block b.

For the entries `point_in_a_space` and `point_in_b_space`, we choose any cell-centered point in block a's index space and assign it to `point_in_a_space`. It does not matter whether or not the point is in the interior of block a's domain. Then we identify the index location of that same point in block b's index space, and assign it to `point_in_b_space`.

The three entries `rotation_b_to_a`, `point_in_a_space`, and `point_in_b_space` are sufficient to describe a unique neighbor relationship between two adjacent blocks that do not touch only at a singularity. When two blocks touch only at a singularity, as in the `BlockNeighbors1` example, another input parameter, `is_singularity` is needed and must be set to `TRUE`.