# Lateral Boundary Conditions for the MPAS system: Requirements and Design

MPAS Development Team

March 18, 2010

# Contents

# Chapter 1

# Summary of Problem

Within the MPAS framework we need to support a variety of model configurations. Notably, we need to support model configurations that span the entire surface of the sphere (termed "global"), configurations that span the infinite plane via double periodic boundary conditions ("doubly-periodic"), configurations that produce a channel system on the plane or on the surface of the sphere ("channel") and configurations that support an arbitrary number of islands in either a planar or spheric setting ("basin").

At present, MPAS is configured in $(x, y, z)$. As a result, supporting spherical or planar geometries is obtained by construction. Also, topologically the global and doubly-periodic domains are identical; both are singly-connected domains meaning that any closed loop can be continuously deformed to match any other closed loop. Requirements for supporting global domains will also cover doubly-periodic domains. Furthermore, the current implementation of MPAS supports both global and doubly-periodic geometries, so no additional requirements are needed for those systems. We must only insure not to break the "global" ability when adding support for the channel and basin configurations.

Both the channel and basin system bring with them the notion of a boundary, i.e. a place where the system requires user-specified values in order to solve the problem. Within the discrete system, there are various ways to implement a boundary, ranging from methods that try to match discrete domain to a user-specified boundary location (i.e. immersed boundary, shaved-cells, etc.) to methods that make the boundary match the discrete mesh. Supporting different "types" of boundary specification is a huge effort, not one to be undertaken at this early stage in development. Rather, our goal here is to get some notion of a boundary into MPAS in as simple a

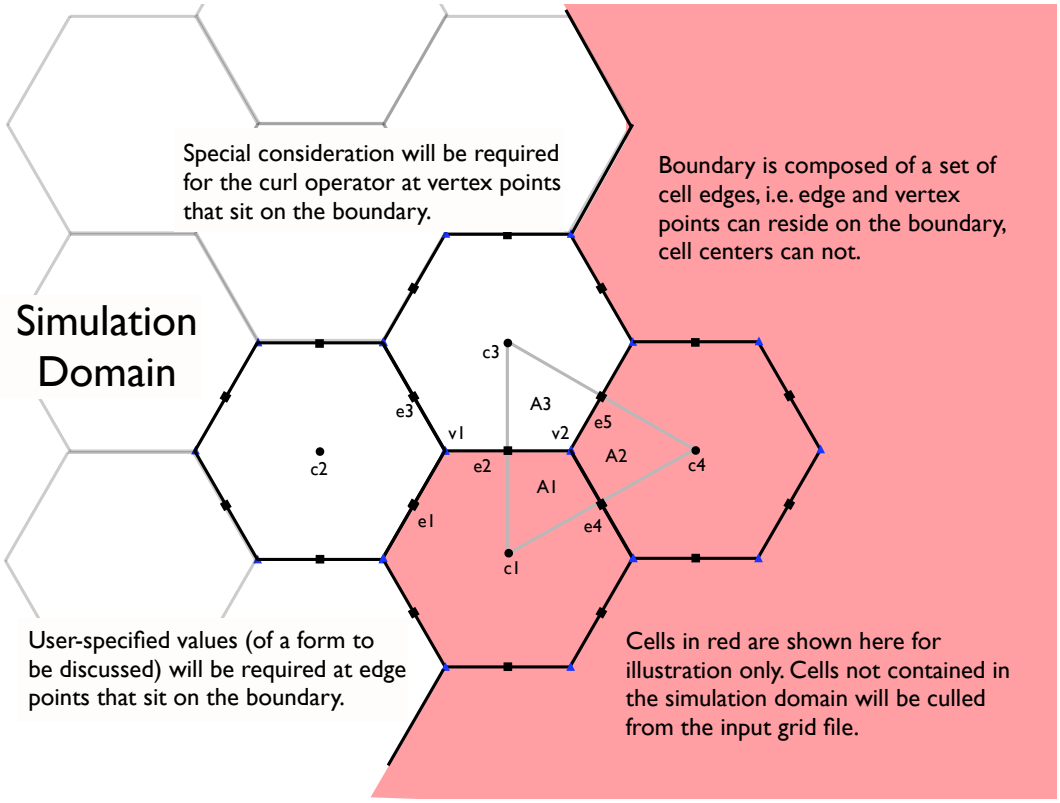


Figure 1.1: A graphical representation of the discrete boundary.

way as possible.

Figure 1 summarizes the type of boundary that this document intends to discuss. The discrete boundary will be composed of a set of edges. In effect, the boundary is made to match the discrete mesh; a cell is either fully in the simulation domain or is removed.

Velocity is solved at the center of cell edges. As a result, when a boundary is present, velocity points will exist on that boundary. In addition, vorticity is computed at cell vertices based on the velocity field. Special consideration will have be given to the computation of vorticity at vertex points along the boundary.

## 1.1 Requirements

A list of requirements related to supporting lateral boundaries of the type discussed above is as follows:

1. Users can specify an arbitrary number of closed, non-overlapping loops based on edge number. Users will specify these loops as simply an arbitrarily-ordered list of boundary edges.

3. Users can specify a time-indepent value of the normal component of velocity at each boundary edge. (Note: supporting time-varying velocities makes sense, but it seems appropriate to wait on this until we get a solid time management system in place). This boundary velocity will be read during the initialization of the simulation.

4. A mechanism will be provided to insure that every boundary edge is connected to exactly two other boundary edges. This testing will likely occur at during the creation of the initial condition data (i.e. before simulation runtime).

5. A mechanism will be provided to insure that when starting at any boundary edge, say, edge $e$ and moving counter-clockwise that the loop returns to edge $e$, i.e. all loops are closed.

6. A check will be performed (probably during the loop test that every boundary edge has one (and exactly one) cellsOnEdge that is set to zero.

## 1.2 Design

The design philosophy is to implement support for boundaries in MPAS in a sensible, but minimalist manner. We must keep in mind that we are still building prototype code, so we should implement support for boundaries with this in mind. The conceptual design is as follows:

1. Add the "boundaryEdge" attribute to the grid derived data type. boundaryEdge will have a default of 0 (integer). boundaryEdge will be of size (nVertLevels, nEdges). Values of 1 will denote that the edge resides on the boundary. boundaryEdge will be contained in the grid.nc file, read in at the initialization of the simulation and made available anytime the grid

derived data type is available.

2. Add the "boundaryVelocity" attribute to the state derived data type. boundaryVelocity will have a default value of 0 (double precision) and will be of size (nVertLevels, nEdges). Where boundaryEdge(iLevel, iEdge) == 1, the computed velocity(iLevel, iEdge) will be replaced by boundaryVelocity(iLevel, iEdge).

3. The model is presently constructed with the assumptions that every edge is associated with exactly two cell centers. At every boundary edge one (and exactly one) of these cells will be removed because it is not a part of the simulation domain (e.g. see Figure 1). Changing the assumption that $nCellsOnEdge \equiv 2$ would result in a great deal of work due to "touching" a significant amount of code. Our design will not change this assumption, but rather referenced cells that do not exist will be given a value of zero when grid.nc is generated. For example, Figure 1 shows that edge $e1$ is associated with cells $c1$ and $c2$ (assume $c4 > c3 > c2 > c1$ and $e5 > e4 > e3 > e2 > e1$ in Figure 1. In a global simulation, $c1$ and $c2$ are a part of the simulation domain and, thus,

```
cellsOnEdge(e1,1) = c1
cellsOnEdge(e1,2) = c2
```

but when $c1$ does not exist because $e1$ is boundary edge, users will provide

```
cellsOnEdge(e1,1) = c2
cellsOnEdge(e1,2) = 0
```

in the grid.nc file. By construction, only one of these cells can be "outside" the simulation domain so only one entry in cellsOnEdge(e1,:) can be non-zero. Furthermore, we will assume that only the "second" cell, i.e. cellsOnEdge(e1,2), can be zero. The model assumes throughout that positive normal vectors point **from** cellsOnEdge(iEdge,1) **toward** cellsOnEdge(iEdge,2) for all iEdge. We will remain consistent with that definition, thus *a positive velocity will always be directed away from the simulation domain at boundary edge points.*

4. It is also assumed in MPAS that every vertex is associated with exactly nVertexDegree edges and nVertexDegree cell centers. nVertexDegree can have a value of 3 (Voronoi diagrams) or a value of 4 (quadrilateral meshes). When boundary edges are present, some of the edges and cells in the arrays

edgesOnVertex and cellsOnVertex will not exist. As above, we will place zeroes in these arrays when the referenced entity does not exist.

For any iVertex the entries in edgesOnVertex(iVertex,:) and cellsOnVertex(iVertex,:) are always listed in a counter clockwise direction (CCW). We will require that this convention hold even when edges and cells in that list do not exist and are zeroes. For example, in a global simulation we would have

```
cellsOnVertex(v2,1) = c1
cellsOnVertex(v2,2) = c4
cellsOnVertex(v2,3) = c3

edgesOnVertex(v2,1) = e2
edgesOnVertex(v2,1) = e4
edgesOnVertex(v2,1) = e5
```

but when $c1$, $c4$ and $e4$ do not exist, these arrays will become

```
cellsOnVertex(v2,1) = 0
cellsOnVertex(v2,2) = 0
cellsOnVertex(v2,3) = c3

edgesOnVertex(v2,1) = e2
edgesOnVertex(v2,1) = 0
edgesOnVertex(v2,1) = e5
```

5. In order to avoid referencing cells and edges that have a value of 0 during the simulation, at runtime all entries in cellsOnEdge, edgesOnVertex and cellsOnVertex with a value of zero will be given a value of nCells+1 or nEdges+1, as appropriate.[1] In order for this to be valid, allocation of derived data types will have be of size nCells+1 and nEdges+1 for cell and edge quantities, respectively. Note that nCells and nEdges varies across processors in distributed memory simulations, so this replacement has to be done after decomposition. This also means that during the construction of the halo regions, tests for cells and edges with a value of zero will have to be done.

[1] In the grid.nc file, nCells, nVertices and nEdges refer to the total number of cells, vertices and edges, respectively, on the entire mesh. In the simulation code, these same variables refer to the number of cells, vertices and edges on a specific processor.

6. In order to avoid the reconstruction of the tangent velocity at edge boundary points, the value of nEdgesOnEdge(iEdge) will be set to zero if boundaryEdge(iEdge) = 1. Note that the array edgesOnEdge(iEdge) will be full (and unchanged from the global simulation domain) when boundaryEdge(iEdge)=0 and will never be reference when boundaryEdge(iEdge)=1.

7. We compute the area associated with each vertex by summing up the "kite" areas (see A1, A2 and A3 in Figure 1). In a global simulation, A1, A2 and A3 would all be non-zero. In this simulation with boundaries we will have A1=0 and A2=0. In its full glory, the expression is

```
kiteAreaOnVertex(v2, 1) = 0
kiteAreaOnVertex(v2, 2) = 0
kiteAreaOnVertex(v2, 3) = A3
```

By zeroing those kite areas that lie outside the simulation domain, we can retain the definition that cell areas (areaCell) are the sum the associated kite areas and that vertex areas (areaTriangle) are also the sum of associated kite areas.

8. The curl operator at boundary vertex points requires special attention since it will involve boundary edges and edges that have been removed because they reside outside the simulation domain. In a global simulation, the curl operator at vertex v2 would be

```
vorticity(v2)=(u(e4)*dc(e4)-u(e5)*dc(e5)-u(e2)*dc(e2))/areaTriangle(v2)
```

The issue here is that e4 will not exist when the boundary shown in Figure 1 is present. Assuming that the areaTriangle is computed as specified above, the curl operator could remained unchanged so long the term u(e4)*dc(e4) returns a value of zero.[2] When boundaries are present, e4 will not exist and, thus, this memory location will point to the entry at nEdges+1. A very easy way to force u(e4)*dc(e4) is to always specify u(nEdges+1)=0 and set dc(nEdges+1)=1. Setting dc(nEdges+1)=1 insures that we do not get NaNs when computing gradients.

[2]When going from the global simulation to one having boundaries, the normal vector at e2 changes sign because c1 is culled and we assume that the normal is always directed outwards at boundary edges. As a result the direction of a positive u(e2) also changes sign. This should be accommodated without modification in the present curl operator because the vertices in verticesOnEdge will also flip. We will have to check to make sure this happens correctly.

The design for the curl operator will be to leave it unchanged, but to insure that u(nEdges+1) always returns a zero.

## 1.3 Modified Routines and interfaces

To be completed.

**Initialization**

**Operators**