Afivo: a framework for finite volume simulations on adaptively refined quadtree and octree grids

Jannis Teunissen, Ute Ebert

February 17, 2016

Abstract

Afivo is a small framework for doing numerical simulations on adaptively refined quadtrees (2D) and octrees (3D). A geometric multigrid solver suitable for these grids is included. Compared to other simulation frameworks, a 'feature' of Afivo is that it provides less functionality, which can make it easier to adapt. For example, only shared-memory parallelization (OpenMP) is included, so that no code for parallel load balancing or communication is required. The framework is available as free/open source software.

1 Introduction

Numerical simulations can often be sped up by having a different mesh in different parts of the domain. There is usually a trade-off: with a more flexible mesh, the cost per computational cell increases. For example, computing a second order approximation of the Laplacian is straightforward on a uniform Cartesian grid. But on an unstructured triangle mesh, such an operation is more complicated: first the neighbors of a cell need to be determined, and then some form of interpolation is required.

Here, we present a framework for simulations on adaptively refined quadtree (2D) and octree (3D) meshes. The main advantage of such meshes is that they provide adaptivity while keeping the grid structure simple. In D dimensions, a quadtree or octree grid consist of boxes that each contain N^D cells. A box with a grid spacing h can be subdivided into 2^D smaller boxes with grid spacing $h/2$. By refining boxes up to different levels, local refinements can be created. In Afivo '2:1 balance' is ensured, which means that between neighboring cells the refinement level differs by at most one. The framework is implemented in Fortran 2003, and is available under the `GNU GPLv3` license.

Below, we first discuss the motivation for developing Afivo, by comparing it to some alternatives. Then the basic data structures and methods are described in sections 3 and 4. In section 5, design choices for parallelization and ghost cells are discussed, after which the multigrid implementation is described in section 6.

2 Motivation and alternatives

There exist numerous frameworks for doing (parallel) numerical computations. Table 1 lists frameworks that operate on structured grids. There are perhaps even more unstructured grid and/or finite element frameworks, but we do not discuss these here.

Name	Application	Language	Parallel	Mesh
Boxlib [2]	General	C/F90	MPI/OpenMP	Block-str.
Chombo [3]	General	C++/Fortran	MPI	Block-str.
AMRClaw	Flow	F90/Python	MPI/OpenMP	Block-str.
SAMRAI [4]	General	C++	MPI	Block-str.
AMROC	Flow	C++	MPI	Block-str.
Paramesh [5]	General	F90	MPI	Orthtree
Dendro [6]	General	C++	MPI	Orthtree
Peano [7]	General	C++	MPI/OpenMP	Orthtree
Gerris [8]	Flow	C	MPI	Orthtree
Ramses [9]	Self gravitation	F90	MPI	Orthtree

Table 1: A list of frameworks for parallel numerical computations on adaptively refined but structured numerical grids. For each framework, the typical application area, programming language, parallelization method and mesh type is listed. This list is largely taken from Donna Calhoun's homepage [10].

Two types of structured meshes are commonly used: *block-structured* meshes and *orthtree*[1] meshes. Examples of these meshes are shown in figure 1. All the listed frameworks use MPI (which stands for Message Passing Interface) for parallelization. This is a distributed memory technique, so that multiple processors connected by a network can be used in parallel. Some codes also support OpenMP, which is a shared-memory parallelization technique that requires processor cores to have access to the same memory.

Block-structured meshes are more general than orthtree meshes: any orthtree mesh is also a block-structured mesh, whereas the opposite is not true. Some of the advantages and disadvantages of these approaches are:

- In a block-structured mesh, blocks can have a flexible size. Computations on larger blocks are typically more efficient. When ghost cells are required (virtual cells on the boundary of a block), the overhead is smaller when blocks are larger.
- For an orthtree grid, there is a trade-off: larger block sizes allow for more efficient computations, but reduce the adaptivity of the mesh. For a block structured grid, there is a similar trade-off: in principle it can be refined in a more flexible way, but adding many refined blocks increases the overhead.
- The connectivity of the mesh is simpler for an orthtree mesh, because each block has the same number of cells, and blocks are refined in the same way. This also ensures a simple relation between fine and coarse meshes. These properties make operations such as prolongation and restriction easier to implement, especially in parallel.

2.1 Motivation: a brief history

Now given the fact that there are already several frameworks available, what was the motivation for developing another one? The main reason was that a *simple* or *basic* framework seemed to be missing – at least to our knowledge. Our motivation came from work on time-dependent simulations of streamer discharges. These discharge have a multiscale nature, and require a fine mesh in the region where they grow. Furthermore, at every time step Poisson's equation has to

[1]Note that we here refer to quadtrees and octrees as *orthtrees*, because the general name for quadrants and octants is orthants [1].

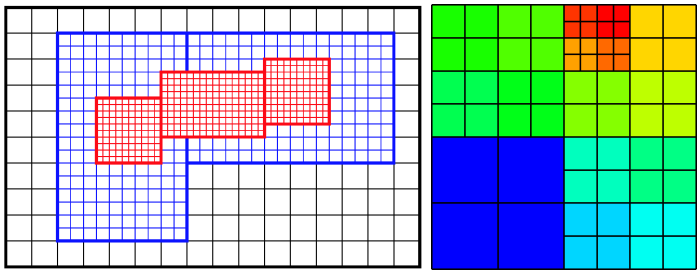

Figure 1: Left: example of a block-structured grid, taken from [11]. Right: an quadtree grid consisting of boxes of 2×2 cells.

be solved. A streamer model that uses a uniform Cartesian grid is therefore computationally expensive, especially for 3D simulations.

In [12], Paramesh was used for streamer simulations. The main bottleneck in this implementation was however the Poisson solver. Other streamer models (see e.g., [13, 14, 15]) had the same problem, because the non-local nature of Poisson's equation makes an efficient parallel solution difficult, especially on an adaptively refined grid. An attractive solution method to get around this is geometric multigrid, discussed in section 6.

We first considered implementing multigrid in Paramesh [5], which already includes an *alpha* version of a multigrid solver with the following comment [16]:

> This is an ALPHA version of this feature. You should be aware that it may be 'buggy'. Also, construction of multigrid algorithms and AMR is much less straightforward than incorporating AMR into finite-volume hydro codes.

Because Paramesh does not seem to be actively maintained, we decided not to move forward with it after experiencing several problems (creating and visualizing output, performance with a large number of blocks, code organization).

Next, we considered Boxlib [2], an actively maintained framework which is also used in Chombo [3]. Boxlib contains a significant amount of multigrid code, including several examples that demonstrate how a solver can be set up and used. After spending some time getting familiar with the framework, we tried to modify the multigrid solver to our needs. This involves operations like: get the coarse grid values next to refinement boundaries to perform a special type of ghost cell filling (see section 6.4). Although such tasks are definitely possible in Boxlib, they are not trivial to implement. The reason is that the framework is quite large and supports many features, uses MPI parallelization and block-structured grids.

In our experience, a large number of scientific simulations fit into the memory of a desktop machine or cluster node, which nowadays typically have 16 or 32 gigabytes of RAM. A practical reason for this is that for larger problems, the visualization of the results becomes quite challenging. For the application we had in mind, efficient large scale parallelism would anyway be hard, due to the non-locality of the Poisson equation. Furthermore, writing parallel code with good scaling takes considerable effort, for which the manpower and resources are often lacking. This inspired us to develop a framework that uses shared-memory parallelism, which makes many operations much simpler, because all data can directly be accessed. The goal was to create a relatively simple framework that could easily be modified, to provide an option in between the 'advanced' distributed-memory codes of table 1 and simple uniform grid computations.

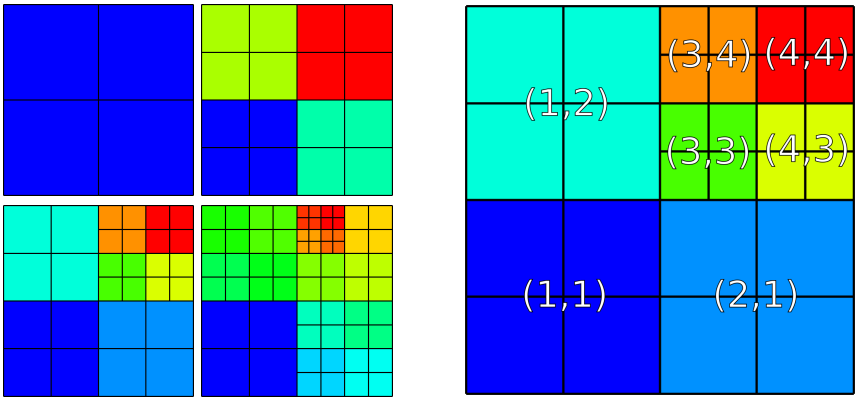

Figure 2: Left: Example of a quadtree mesh that gets refined. Here boxes contain 2×2 cells, and different boxes have different colors. Right: the spatial indices of the boxes. When a box with indices (i, j) is refined, its children have indices $(2i - 1, 2j - 1)$ up to $(2i, 2j)$.

3 Overview of data structures

We now start with the description of Afivo's implementation. First, the properties of *orthtree* meshes are discussed. Three data types are used to store these meshes: boxes, levels and trees. These data types are described below.

3.1 Orthtree meshes

In Afivo, quadtree (2D) and octree (3D) meshes are used, which can be described by the following rules:

- The mesh is constructed from boxes that each contain N^D cells, where D is the number of coordinates and N is an even number.
- When a box with a grid spacing h is refined, it is subdivided in 2^D 'children' with grid spacing $h/2$.
- The difference in refinement level for adjacent boxes is at most one. This is called '2:1 balance'.

Figure 2 shows an example of a quadtree that gets refined. All the boxes are stored in a single one-dimensional array, so that an integer index can be used to point to a box, see section 3.4 below.

3.2 Box data type

A *box* is the basic mesh unit in Afivo. Each box consists of N^D grid cells, where N has to be an even number and D is the spatial dimension. In figure 2, there is for example a box with 2×2 cells at coordinate $(1, 1)$. Each box stores its parent, an array of 2^D children and an array of $2D$ neighbors. In figure 3a, the indices of the children and the neighbors are shown. A special value `a5_no_box` (which is zero) is used to indicate that a parent, child or neighbor does not exist. In the case of neighbors, boundary conditions are specified by negative numbers.

Two types of cell data are supported by default: cell-centered data and face-centered data, see figure 3b. These are stored in $D + 1$-dimensional arrays, so that multiple variables can be

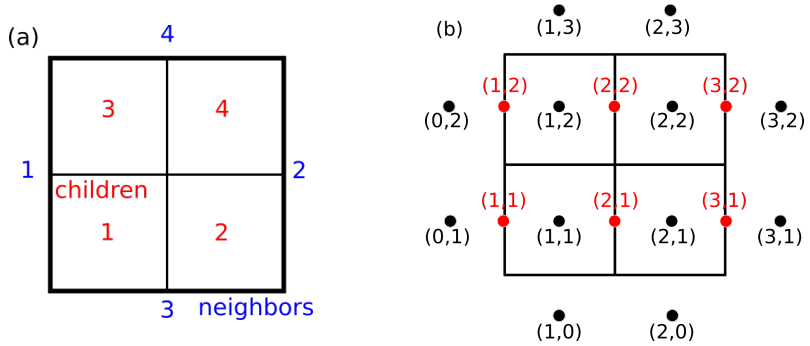

Figure 3: a) Each box contains an array of children and neighbors. The ordering of these arrays is shown here for a 2D box. Red: children, blue: neighbors. b) Location and indices of the cell-centered variables (black dots) and the face-centered variables in the x-direction (red dots) for a box of 2×2 cells.

stored per location. Furthermore, boxes contain some 'convenience' information, such at their refinement level, minimum coordinate and spatial index.

3.3 Level data type

The *level* data type contains three lists:

- A list with all the boxes at refinement level l
- A list with the *parents* (boxes that are refined) at level l
- A list with the *leaves* (boxes that are not refined) at level l

This separation is often convenient, because some algorithms operate only on leaves while others operate on parents or on all boxes. These lists contain the integer indices of the boxes in the tree data structure described below.

3.4 Tree data type

The tree data type contains all the data of the mesh. Most importantly, it stores two arrays: one that contains all the boxes and one that contains all the levels[2]. Some other information is also stored: the current maximum refinement level, the number of cells per box-dimension N, the number of face and cell-centered variables and the grid spacing Δx at the coarsest level.

4 Methods

In this section we give a brief overview of the most important methods in Afivo. The names of the methods for two-dimensional meshes are used, which have the prefix `a2`. The three-dimensional analogs have, not surprisingly, a prefix `a3`.

[2]Since Afivo is implemented in Fortran, these arrays start at index one.

```
! Initialize tree
call a2_init(tree, & ! Tree to initialize
     box_size, &     ! Number of cells per coordinate in a box
     n_var_cell, &   ! Number of face-centered variables
     n_var_face, &   ! Number of cell-centered variables
     dr)             ! Distance between cells on base level

! Set the spatial index and neighbors
ix_list(1:2, 1) = 1  ! One box at (1,1)
nb_list(1:4, 1) = 1  ! Periodic box is its own neighbor

! Create the base mesh
call a2_set_base(tree, ix_list, nb_list)
```

Figure 4: Fortran code fragment that shows how a base mesh can be constructed. In this case, there is one box at $(1, 1)$, with periodic boundary conditions.

4.1 Creating the initial mesh

In Afivo the coarsest mesh, which covers the full computational domain, is not supposed to change. To create this mesh there is a routine `a2_set_base`, which takes as input the spatial indices of the coarse boxes and their neighbors. In figure 4, a 2D example is shown for creating a single coarse box at index $(1, 1)$. This box is its own neighbor in all four directions, or in other words, there are periodic boundary conditions. Physical (non-periodic) boundaries can be indicated by a negative index for the neighbor. By adjusting the neighbors one can specify different geometries, the possibilities include meshes that contain a hole, or meshes that consist of two isolated parts. The treatment of boundary conditions is discussed in section 4.3.

4.2 The refinement procedure

Mesh refinement can be performed by calling the `a2_adjust_refinement` routine, which should be passed a user-defined refinement function. This function is then called for each box, and should set the refinement flag of the box to one of three values: refine (add children), derefine (remove this box) or keep refinement. It is also possible to set the refinement flags for other boxes than the current one, which can for example be useful to extend refinements to a neighbor.

In Afivo, a box is either fully refined (with 2^D children) or not refined. Furthermore, 2:1 balance is ensured, so that there is never a jump of more than one refinement level between neighboring boxes. These constraints are automatically handled, so that the user-defined refinement function does not need to impose them.

Each call to `a2_adjust_refinement` changes the mesh by at most one level. To introduce larger changes one should call the routine multiple times. A number of rules is used to make the user-supplied refinement consistent:

- Only leaves can be removed (because the grid changes by at most one level at a time)
- A box flagged for refinement will always be refined, including neighbors that are required for 2:1 balance
- Boxes cannot be removed if that would violate 2:1 balance
- If all the 2^D children of a box are flagged for removal, and the box itself not for refinement, then the children are removed

- Boxes at level one cannot be removed
- Boxes cannot be refined above the maximum allowed refinement level

The `a2_adjust_refinement` routine returns information on the added and removed boxes per level, so that a user can set values on the new boxes or clean up data on the removed ones.

When boxes are added or removed in the refinement procedure, their connectivity is automatically updated. References to a removed box are removed from its parent and neighbors. When a new box is added, its neighbors are found through its parent. Three scenarios can occur: the neighbor can be one of the other children of the parent, the neighbor can be a child from the neighbor of the parent, or the neighbor does not exist. In the latter case, there is a refinement boundary, which is indicated by the special value `a5_no_box`.

4.3 Filling of ghost cells

When working with numerical grids that are divided in multiple parts, it is often convenient to use *ghost cells*. The usage of ghost cells has two main advantages: algorithms can operate on the different parts without special care for the boundaries, and algorithms can straightforwardly operate in parallel.

In Afivo each box has one layer of ghost cells for its cell-centered variables, as illustrated in figure 3b. The built-in routines only fill the ghost cells on the sides of boxes, not those on the corners. The reasons for implementing ghost cells in this way are discussed in sections 5.1 and 5.2.

For each side of a box, ghost cells can be filled in three ways

- If there is a neighboring box, then the ghost cells are simply copied from this box
- If there is a physical boundary, then the box is passed to a user-defined routine for boundary conditions
- If there is a refinement boundary, then the box is passed to another user-defined routine

Physical boundaries are indicated by negative values for the neighbor index, and these values are passed on to the user-defined routine. In this way, one can set up different types of boundary conditions.

4.4 Interpolation and restriction

Because corner ghost cells are not filled by Afivo, interpolation schemes cannot use diagonal elements. Instead of the standard bilinear and trilinear interpolation schemes, a $2-1-1$ and $1-1-1-1$ scheme is used in 2D and 3D, respectively. These interpolation schemes use information from the closest and second-closest neighbors; the 2D case is illustrated in figure 5. Zeroth-order interpolation is also included, in which the coarse values are simply copied without any interpolation. As a restriction method (going from fine to coarse) Afivo just includes averaging, in which the parent gets the average value of its children.

A user can of course implement higher order interpolation and restriction methods, by using information from additional grid locations. It is generally quite complicated to do this consistently near refinement boundaries.

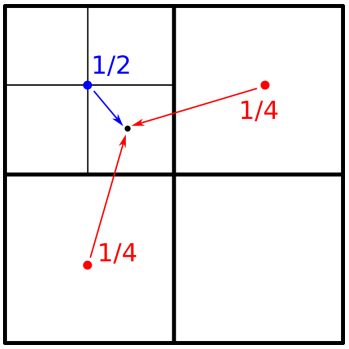

Figure 5: Schematic drawing of $2-1-1$ interpolation. The three nearest coarse grid values are used to interpolate to the center of a fine grid cell. Note that the same interpolation scheme can be used for all fine grid cells, because of the symmetry in a cell-centered discretization.

4.5 The list of boxes

In Afivo, all the boxes are stored in a single array. New boxes are always added to the end of the array, which means that removed boxes leave a 'hole'. There is a route `a2_tidy_up` to tidy up the array: all the unused boxes are moved to the end of the array, and the boxes that are still in use are sorted by their refinement level. Furthermore, for each level, the boxes are ranked according to their Morton index [17].

Sometimes, extra storage is required when `a2_adjust_refinement` has to add new boxes to the mesh. In such a case, the array of boxes is simply resized so that there is enough space.

4.6 Producing output

It is important that one is able to quickly and conveniently visualize the results of a simulation. Afivo supports two output formats: VTK unstructured files and Silo files.

For VTK files, Afivo relies on the unstructured format, which support much more general grids than quadtree and octree meshes. This format should probably only be used for grids of moderate sizes (e.g., 10^5 or 10^6 cells), because visualizing larger grids can be computationally expensive. Although there is some support for octrees in VTK, this support does not yet extend to data visualization programs such as Paraview [18] and Visit [19].

Afivo also supports writing Silo files. These files contain a number of Cartesian blocks ('quadmeshes' in Silo's terminology) that can each contain multiple boxes. This is done by starting with a region R that contains a single box. If all the neighbors to the left of R exist, have no children and are not yet included in the output, then these neighbors are added to R. The procedure is repeated in all directions, until R can no longer grow. Then R represents a rectangular collection of boxes which can be added to the output, and the procedure start again from a new box that is not yet included. This merging of boxes is done because writing and reading a large number separate meshes can be quite costly with the Silo library.

5 Design discussion

5.1 One ghost cell

There are essentially two ways to implement ghost cells in a framework such as Afivo.

1. Ghost cells are not stored for boxes. When a computation has to be performed on a box, there are typically two options: algorithms can be made aware of the mesh structure, or a box can be temporarily copied to an enlarged box on which ghost cells are filled.

2. Each box includes ghost cells, either a fixed number for all variables or a variable-dependent number.

Storing ghost cells can be quite costly. For example, adding two layers of ghost cells to a box of 8^3 cells requires $(12/8)^3 = 3.375$ times as much storage. With one layer, about two times as much storage is required. Not storing ghost cells prevents this extra memory consumption. However, some operations can become more complicated to program, for example when some type of interpolation depends on coarse-grid ghost cells. Furthermore, one has to take care not to unnecessarily recompute ghost values, and parallelization becomes slightly harder.

If ghost cells are stored for each box, then there are still two options: store a fixed number of them for each variable, or let the number of ghost cells vary per variable. In Afivo, we have opted for the simplest approach: there is always one layer of ghost cells for cell-centered variables. For numerical operations that depend on the nearest neighbors, such as computing a second order Laplacian, one ghost cell is enough. When additional ghost cells are required, these can of course still be computed, there is just no default storage for them.

5.2 No corner ghost cells

In Afivo, corner ghost cells are not used. The reason for this is that in three dimensions, the situation is quite complicated: there are eight corners, twelve edges and six sides. It is hard to write an elegant routine to fill all these ghost cells, especially because the corners and edges have multiple neighbors. Therefore, only the sides are considered in Afivo. This means that Afivo is not suitable for stencils with diagonal terms.

5.3 OpenMP for parallelism

The two conventional methods for parallel computing are OpenMP (shared memory) and MPI (communicating tasks). Afivo was designed for small scale parallelism, for example using at most 16 cores, and therefore only supports OpenMP. Compared to an MPI implementation, the main advantage of OpenMP is simplicity: data can always be accessed, sequential (user) code can easily be included, there is no need for load balancing and no communication between processes needs to be set up. For problems that require large scale parallelism, there are already a number of frameworks available, as discussed in section 2.

Most operations in Afivo loop over a number of boxes, for example the leaves at a certain refinement level. All such loops have been parallelized by adding OpenMP statements around them, for example as in figure 6.

The parallel speedup that one can get depends on the cost of the algorithm that one is using. The communication cost (updating ghost cells) is always about the same, so that an expensive algorithm will show a better speedup. Furthermore, on a shared memory system, it is not unlikely for an algorithm to be memory-bound instead of CPU-bound.

6 Multigrid

Multigrid can be seen as a technique to improve the convergence of a relaxation method, by using a hierarchy of grids. Afivo comes with a built-in geometric multigrid solver, to solve problems of

```
do lvl = 1, tree%max_lvl
   !$omp parallel do private(id)
   do i = 1, size(tree%lvls(lvl)%ids)
      id = tree%lvls(lvl)%ids(i)
      call my_method(tree%boxes(id))
   end do
   !$omp end parallel do
end do
```

Figure 6: Fortran code fragment that shows how to call `my_method` for all the boxes in a tree, from level 1 to the maximum level. Within each level, the routine is called in parallel using OpenMP.

the form

$$A(u) = \rho, \tag{1}$$

where A is a (nearly) elliptic operator, ρ the right-hand side and u the solution to be computed. In discretized form, we write equation (1) as

$$A_h(u_h) = \rho_h, \tag{2}$$

where h denotes the mesh spacing at which the equation is discretized.

There already exists numerous sources on the foundations of multigrid, the different cycles and relaxation methods, convergence behaviour and other aspects, see for example [20, 21, 22, 23]. Here we will not provide a general introduction to multigrid. Instead we briefly summarize the main ingredients, and focus on one particular topic: how to implement multigrid on an adaptively refined quadtree or octree mesh. On such a mesh, the solution has to be specified on all levels. Therefore we use FAS multigrid, which stands for Full Approximation Scheme. Below, the implementation of the various multigrid components in Afivo are described.

6.1 The V-cycle

Suppose there are levels $l = l_{\text{min}}, l_{\text{min}} + 1, \ldots, l_{\text{max}}$, then the FAS V-cycle can be described as

1. For l from l_{max} to $l_{\text{min}} + 1$, perform N_{down} relaxation steps on level l, then update level $l - 1$ (see below)

2. Perform N_{base} relaxation steps on level l_{min}, or apply a direct solver

3. For l from $l_{\text{min}} + 1$ to l_{max}, perform a correction using the data from level $l - 1$

$$u_h \leftarrow u_h + I_H^h(v_H - v'_H), \tag{3}$$

   then perform N_{up} relaxation steps on level l. (See below for the notation)

The first two steps require some extra explanation. Let us denote the level $l - 1$ grid by H and the level l grid by h, and let v denote the current approximation to the solution u. Furthermore, let I_H^h be an interpolation operator to go from coarse to fine and I_h^H a restriction operator to go from fine to coarse. For these operators, the schemes described in section 4.4 are used.
In the first step, the coarse grid is updated in the following way

1. Set $v_H \leftarrow I_h^H v_h$, and store a copy v'_H of v_H

2. Compute the the fine grid residual $r_h = \rho_h - A_h(v_h)$

3. Update the coarse grid right-hand side

$$\rho_H \leftarrow I_h^H r_h + A_H(v_H) \tag{4}$$

This last equation can also be written as

$$\rho_H \leftarrow I_h^H \rho_h + \tau_h^H,$$

where τ_h^H is given by [20, 21, 22, 24, 23]

$$\tau_h^H = A_H(I_h^H v_h) - I_h^H A_h(v_h). \tag{5}$$

This term can be seen as a correction to ρ on the coarse grid. When a solution u_h is found such that $A_h(u_h) = \rho_h$, then $u_H = I_h^H u^h$ will be a solution to $A_H(u_H) = \rho_H$.

In the second step, relaxation takes place on the coarsest grid. In order to quickly converge to the solution with a relaxation method, this grid should contain very few points (e.g., 2×2 or 4×4 in 2D). Alternatively, a direct solver can be used on the coarsest grid, in which case it can be larger. Such a direct method has not yet been built into Afivo, although this is planned for the future. As a temporary solution, additional coarse grids can be constructed below the coarsest quadtree/octree level. For example, if a quadtree has boxes of 16×16 cells, then three levels can be added below it (8×8, 4×4 and 2×2), which can then be used in the multigrid routines.

6.2 The FMG-cycle

The full multigrid (FMG) cycle that is implemented in Afivo works in the following way.

1. If there is no approximation of the solution yet, then set the initial guess to zero on all levels, and restrict ρ down to the coarsest grid using I_h^H. If there is already an approximation v to the solution, then restrict v down to the coarsest level. Use equation (4) to set ρ on coarse grids.

2. For $l = l_{\min}, l_{\min} + 1, \ldots, l_{\max}$

   - Store the current approximation v_h as v_h'
   - If $l > l_{\min}$, perform a coarse grid correction using equation (3)
   - Perform a V-cycle starting at level l, as described in the previous section

6.3 Gauss Seidel red-black

In Afivo, we have implemented Gauss Seidel red-black or GS-RB as a relaxation method. This method is probably described in almost all textbooks on multigrid, such as [20, 21, 22, 23], so we just give a very brief description.

The *red-black* refers to the fact that points are relaxed in an alternating manner, using a checkerboard-like pattern. For example, in two dimensions with indices (i, j) points can be labeled *red* when $i + j$ is even and *black* when $i + j$ is odd. Now consider equation (2), which typically relates a value $u_h^{(i,j)}$ to neighboring values and the source term ρ. If we keep the values of the neighbors fixed, then we can determine the value $u_h^{(i,j)}$ that locally solves the linear equation. This is precisely what is done in GS-RB: the linear equations are solved for all the red points while keeping the old black values, and then vice-versa.

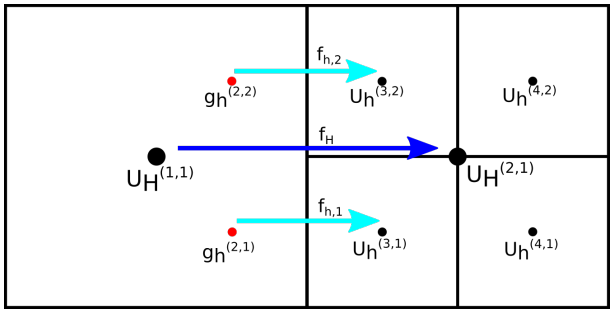

Figure 7: Two coarse cells, of which the right one is refined. The cell centers are indicated by dots. There are two ghost values (red dots) on the left of the refinement boundary. Fluxes across the refinement boundary are indicated by arrows.

6.4 Conservative filling of ghost cells

The finer levels will typically not cover the complete grid in Afivo, so that ghost cells have to be used near refinement boundaries. These ghost cells can be filled in multiple ways, which will affect the multigrid solution and convergence behavior. Here we consider *conservative* schemes for filling ghost cells [24, 21]. A conservative scheme ensures that the coarse flux across a refinement boundary equals the average of the fine fluxes, see figure 7.

Ensuring consistent fluxes near refinement boundaries helps in obtaining a *consistent* solution. For example, if we consider a general equation of the form $\nabla \cdot \vec{F} = \rho$, then the divergence theorem gives

$$\int_V \rho \, dV = \int_V \nabla \cdot \vec{F} \, dV = \int \vec{F} \cdot \vec{n} \, dS, \tag{6}$$

where the last integral runs over the surface of the volume V, and $\vec{n}$ is the normal vector to this surface. This means that when fine and coarse fluxes are consistent, the integral over ρ will be same on the fine and the coarse grid.

The construction of a conservative scheme for filling ghost cells is perhaps best explained with an example. Consider a 2D Poisson problem

$$\nabla^2 u = \nabla \cdot (\nabla u) = \rho, \tag{7}$$

with a standard 5-point stencil for the Laplace operator

$$L_h = h^{-2} \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}. \tag{8}$$

With this stencil, the coarse flux f_H across the refinement boundary in figure 7 is given by

$$f_H = [u_H^{(2,1)} - u_H^{(1,1)}]/H, \tag{9}$$

and on the fine grid, the two fluxes are given by

$$f_{h,1} = [u_h^{(3,1)} - g_h^{(2,1)}]/h, \tag{10}$$

$$f_{h,2} = [u_h^{(3,2)} - g_h^{(2,2)}]/h. \tag{11}$$

The task is now to fill the ghost cells $g_h^{(2,1)}$ and $g_h^{(2,2)}$ in such a way that the coarse flux equals the average of the fine fluxes, i.e., such that

$$f_H = (f_{h,1} + f_{h,2})/2 \tag{12}$$

To relate $u_H^{(2,1)}$ to the refined values u_h, the restriction operator I_h^H needs to be specified. In our implementation, this operator does averaging over the children, which can be represented as

$$I_h^H = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}. \tag{13}$$

The constraint from equation (12) can then be written as

$$g_h^{(2,1)} + g_h^{(2,2)} = u_H^{(1,1)} + \frac{3}{4}\left(u_h^{(3,1)} + u_h^{(3,2)}\right) - \frac{1}{4}\left(u_h^{(4,1)} + u_h^{(4,2)}\right). \tag{14}$$

Any scheme for the ghost cells that satisfies this constraint will be a conservative discretization.

Bilinear *extrapolation* (similar to standard bilinear interpolation) gives the following scheme for $g_h^{(2,1)}$

$$g_h^{(2,1)} = \frac{1}{2}u_H^{(1,1)} + \frac{9}{8}u_h^{(3,1)} - \frac{3}{8}\left(u_h^{(3,2)} + u_h^{(4,1)}\right) + \frac{1}{8}u_h^{(4,2)}. \tag{15}$$

(The scheme for $g_h^{(2,2)}$ should then be obvious.) Another option is to use only the closest two neighbors for the extrapolation, which gives the following expression for $g_h^{(2,1)}$

$$g_h^{(2,1)} = \frac{1}{2}u_H^{(1,1)} + u_h^{(3,1)} - \frac{1}{4}\left(u_h^{(3,2)} + u_h^{(4,1)}\right). \tag{16}$$

This last scheme is how refinement-boundary ghost cells are filled by default in Afivo.

6.4.1 Three-dimensional case

In three spatial dimensions, the 5-point stencil of equation (8) becomes a 7-point stencil with -6 at the center, and the restriction operator has eight entries of 1/8. The analog of equation (16) then becomes

$$g_h^{(2,1,1)} = \frac{1}{2}u_H^{(1,1,1)} + \frac{5}{4}u_h^{(3,1,1)} - \frac{1}{4}\left(u_h^{(4,1,1)} + u_h^{(3,2,1)} + u_h^{(3,1,2)}\right). \tag{17}$$

6.4.2 Change in ε at cell face

For the more general equation with a coefficient ε

$$\nabla \cdot (\varepsilon \nabla u) = \rho, \tag{18}$$

we consider a special case: ε jumps from ε_1 to ε_2 at a cell face. Local reconstruction of the solution shows that a gradient $(\phi_{i+1} - \phi_i)/h$ has to be replaced by

$$\frac{2\,\varepsilon_1\varepsilon_2}{\varepsilon_1\varepsilon_2}\,\frac{\phi_{i+1} - \phi_i}{h}, \tag{19}$$

or in other words, the gradient is multiplied by the harmonic mean of the ε's (see for example chapter 7.7 of [21]). The 5-point stencil for the Laplacian can be modified accordingly.

When a jump in ε occurs on a coarse cell face, it will also be located on a fine cell face, see figure 7. In this case, the ghost cell schemes described above for constant ε still ensure flux conservation. The reason is that the coarse and fine flux are both weighted by a factor $2\,\varepsilon_1\varepsilon_2/(\varepsilon_1\varepsilon_2)$.

6.4.3 Cylindrical case

In cylindrical coordinates, the Laplace operator can be written as

$$\nabla^2 u = \frac{1}{r}\partial_r(r\partial_r u) + \partial_z^2 u = \partial_r^2 u + \frac{1}{r}\partial_r u + \partial_z^2 u, \tag{20}$$

where we have assumed cylindrical symmetry (no ϕ dependence). At a radius $r \neq 0$, the 5-point stencil is

$$L_h = h^{-2}\begin{bmatrix} & 1 & \\ 1-\frac{h}{2r} & -4 & 1+\frac{h}{2r} \\ & 1 & \end{bmatrix}. \tag{21}$$

With the cell-centered grids in Afivo, radial grid points are located at $(i-\frac{1}{2})h$ for $i = 1, 2, 3, \ldots$, which means we do not have to consider the special case $r = 0$. For this type of grid indexing, the 5-point stencil can also be written as

$$L_h = h^{-2}\begin{bmatrix} & 1 & \\ \frac{2i-2}{2i-1} & -4 & \frac{2i}{2i-1} \\ & 1 & \end{bmatrix}. \tag{22}$$

If we do not modify the restriction operator, then the ghost cells can still be filled with the schemes from equations (16) and (17). One way to interpret this is that fluxes are computed in the same way in cylindrical coordinates, although their divergence is weighted by the radius:

$$\nabla \cdot \vec{F} = \frac{1}{r}\partial_r(rF_r) + \ldots \tag{23}$$

From figure 7, we can see that for refinement in the r-direction, the coarse and fine flux are 'weighted' by the same radius. For the fluxes in the z-direction, the computations are the same as for the Cartesian case. Note that when the restriction operator is changed to include radial weighting, these arguments are no longer valid.

6.5 Multigrid test problems

In this section we present two test problems to demonstrate the multigrid behavior on a partially refined mesh. We use the 'method of manufactured solutions': from an analytic solution the right-hand side and boundary conditions are computed. Two test problems are considered, a constant-coefficient Poisson equation

$$\nabla^2 u = \nabla \cdot (\nabla u) = \rho \tag{24}$$

and a cylindrical version with a coefficient ε

$$\frac{1}{r}\partial_r(r\varepsilon\partial_r u) + \partial_z(\varepsilon\partial_z u) = \rho, \tag{25}$$

both on a two-dimensional rectangular domain $[0,1] \times [0,1]$. For the second case, ε has a value of 100 in the lower left quadrant $[0, 0.25] \times [0, 0.25]$, and a value 1 in the rest of the domain. In both cases, we pick the following solution for u

$$u(r) = \exp(|\vec{r} - \vec{r}_1|/\sigma) + \exp(|\vec{r} - \vec{r}_2|/\sigma), \tag{26}$$

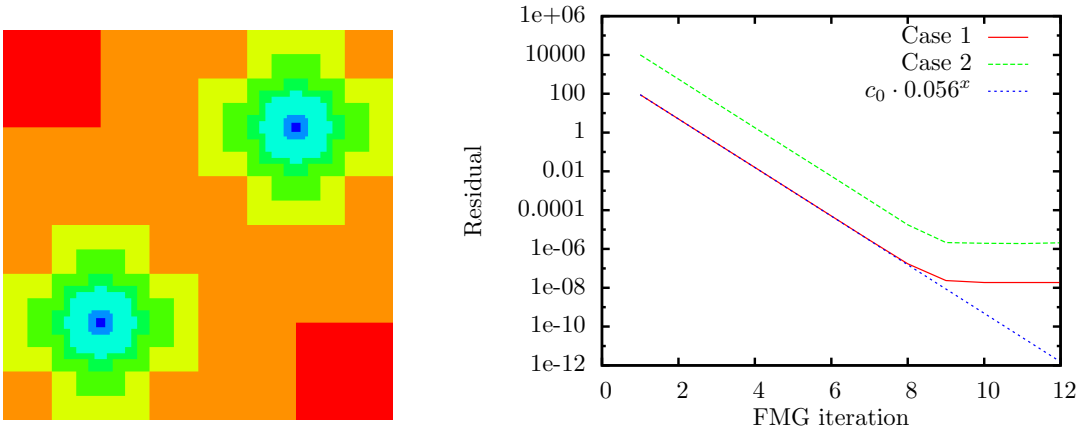

Figure 8: Left: mesh spacing used for the multigrid examples, in a $[0,1] \times [0,1]$ domain. Each step in color is a factor two in refinement, with red indicating $\Delta x = 2^{-5}$ and the darkest blue indicating $\Delta x = 2^{-12}$. Right: the maximum residual versus FMG iteration, case 1 corresponds to equation (24) and case 2 to equation (25).

where $\vec{r_1} = (0.25, 0.25)$, $\vec{r_2} = (0.75, 0.75)$ and $\sigma = 0.04$. An analytic expression for the right-hand side ρ is obtained by plugging the solution in equations (24) and (25)[3], and the solution itself is used to set boundary conditions.

The two different problems can now be solved numerically. For the cylindrical case with the varying ε, a modified Laplacian operator is used, as described in section 6.4. The Gauss Seidel red-black relaxation methods are also modified, because they depend on the applied operator, see section 6.3. For these examples, we have used $N_{\text{down}} = N_{\text{up}} = N_{\text{base}} = 2$ (number of down/up/base smoothing steps), and a coarsest grid of 2×2 cells.

It is possible to do adaptive mesh refinement in multigrid, for example by using an estimate of the local truncation error based on equation (5) (see also chapter 9 of [21]). Such a technique is not used here, instead the refinement criterion is based on the right-hand side: refine if $\Delta x^2 |\rho| > 5.0 \times 10^{-4}$. The resulting mesh spacing is shown in figure 8a.

In both cases, one FMG (full multigrid) cycle is enough to achieve convergence up to the discretization error, which was approximately 10^{-4} for the mesh of figure 8a. Consecutive FMG cycles have a negligible effect on the absolute error, although they do reduce the residual $r = \rho - \nabla^2 u$. The maximum value of $|r|$ is shown versus iteration number in figure 8b. The convergence behaviour is similar for both cases, with each iteration reducing the residual by a factor of about 0.056. The offset between the lines is caused by the $\varepsilon = 100$ region, which locally amplifies the source term by a factor of 100.

7 Implementing a plasma fluid model

To illustrate how Afivo can be used, we describe the implementation of a simple 2D/3D *plasma fluid model* for streamer discharges below. For simplicity, photoionization is not included in this example. A review of fluid models for streamer discharges can be found in [15].

[3]Note that jumps in ε also contribute to the source term ρ.

7.1 Model formulation

We use the so-called drift-diffusion-reaction approximation:

$$\partial_t n_e = -\nabla \cdot \vec{j}_e + \bar{\alpha} \left| \vec{j}_e \right|, \tag{27}$$

$$\partial_t n_i = \bar{\alpha} \left| \vec{j}_e \right|, \tag{28}$$

$$\vec{j}_e = -\mu_e n_e \vec{E} - D_e \nabla n_e, \tag{29}$$

where n_e is the electron density, n_i the positive ion density, $\vec{j}_e$ the electron flux, $\bar{\alpha}$ the effective ionization coefficient, μ_e the electron mobility, D_e the electron diffusion coefficient and $\vec{E}$ the electric field. The above equations are coupled to the electric field, which we compute in the electrostatic approximation:

$$\vec{E} = -\nabla \phi, \tag{30}$$

$$\nabla^2 \phi = -\rho/\varepsilon_0 \tag{31}$$

$$\rho = e(n_i - n_e), \tag{32}$$

where ϕ is the electric potential, ε_0 the permittivity of vacuum and e the elementary charge. The electric potential is computed with the multigrid routines described in section 6.

We make use of the *local field approximation* [25], so that μ_e, D_e and $\bar{\alpha}$ are all functions of the local electric field strength $E = \left| \vec{E} \right|$. These coefficients can be obtained experimentally, or they can be computed with a Boltzmann solver [26, 27] or particle swarms [28].

7.2 Flux calculation and time stepping

The electron flux is computed as in [13]. For the diffusive part, we use central differences. The advective part is computed using the Koren limiter [29]. The Koren limiter was not designed to include refinement boundaries, and we use linear interpolation to obtain fine-grid ghost values. These ghost cells lie inside a coarse-grid neighbor cell, and we limit them to twice the coarse values to preserve positivity. (We would like to improve this in the future.)

Time stepping is also performed as in [13], using the explicit trapezoidal rule, also known as the modified Euler's method. The time step is determined by a CFL condition for the electron flux and the dielectric relaxation time, as in [13].

7.3 Refinement criterion

Our refinement criterion contains two components: a *curvature monitor* c_ϕ for the electric potential and a monitor $\bar{\alpha}\Delta x$ which gives information on how well the ionization length $(1/\bar{\alpha})$ is resolved. For both, we use the maximum value found in a box in order to decide whether to (de)refine it.

Since $\nabla^2 \phi = -\rho/\varepsilon_0$, the curvature monitor can be computed as $c_\phi = \Delta x^2 |\rho|/\varepsilon_0$. The quantity $\bar{\alpha}\Delta x$ is computed by locating the highest electric field in the box, and looking up the corresponding value of $\bar{\alpha}$. The combined refinement criterion is then as follows, where later rules can override earlier ones:

- If $\bar{\alpha}\Delta x < 0.1$ and $\Delta x < 25\,\mu$m, derefine.
- If $t < 2.5\,$ns, ensure that there is enough refinement around the initial seed to resolve it.
- If $\bar{\alpha}\Delta x > 1.0$ and $c_\phi > 0.1\,$Volt, refine.

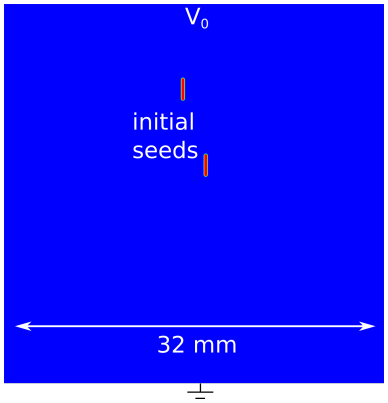

Figure 9: Cross section through the center of the three-dimensional simulation domain. The ionized seeds with a density of $10^{20}\,\mathrm{m}^{-3}$ electrons and ions are indicated in red. There is a background density of $5 \times 10^{15}\,\mathrm{m}^{-3}$ electrons and ions, and the background electric field points down with a magnitude $E_0 = 2.5\,\mathrm{MV/m}$.

7.4 Simulation conditions and results

A cross section through the computational domain of $(32\,\mathrm{mm})^3$ is shown in figure 9. The background field points down, with a magnitude $E_0 = 2.5\,\mathrm{MV/m}$, which is about $5/6^{\mathrm{th}}$ of the critical field. The background field is applied by grounding the bottom boundary of the domain, and applying a voltage at the top. At the other sides of the domain we use Neumann boundary conditions for the potential. We use transport coefficients (e.g., $\bar{\alpha}$, μ_e) for atmospheric air, but for simplicity photoionization has not been included. Instead a background density of $5 \times 10^{15}\,\mathrm{m}^{-3}$ electrons and positive ions is present.

Two seeds of electrons and ions locally enhance the background electric field, see figure 9. These seeds have a density of $10^{20}\,\mathrm{m}^{-3}$, a width of about 0.3 mm and a length of 1.6 mm. The electrons from these seeds will drift upwards, enhancing the field at the bottom of the seed where a positive streamer can form.

In figure 10, the time evolution of the electron density is shown, and in figure 11 the electric field is shown. Two positive streamers grow downwards from the ionized seeds. The upper one is attracted to the negatively charged end of the lower one, and connects to it at around 9.5 ns. The three-dimensional simulation took about 3.5 hours on a 16-core machine, and eventually used about 1.3×10^7 grid cells.

References

References

[1] Erick Wong. Name of the generalization of quadtree and octree? [Online; accessed 17-July-2015].

[2] Ann S. Almgren et al. Boxlib. [Online; accessed 22-July-2015].

[3] P. Colella, D. T. Graves, J. N. Johnson, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo software package for AMR applications - design document, 2011.

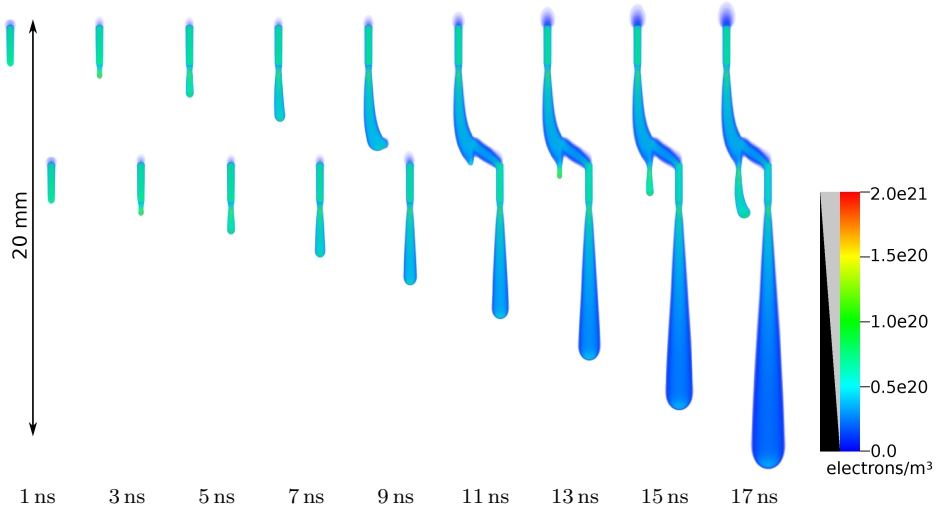

Figure 10: A three-dimensional simulation showing two positive streamers propagating downwards. The upper one connects to the back of the lower one. The electron density is shown using volume rendering, for which the opacity is indicated in the legend; low densities are transparent.

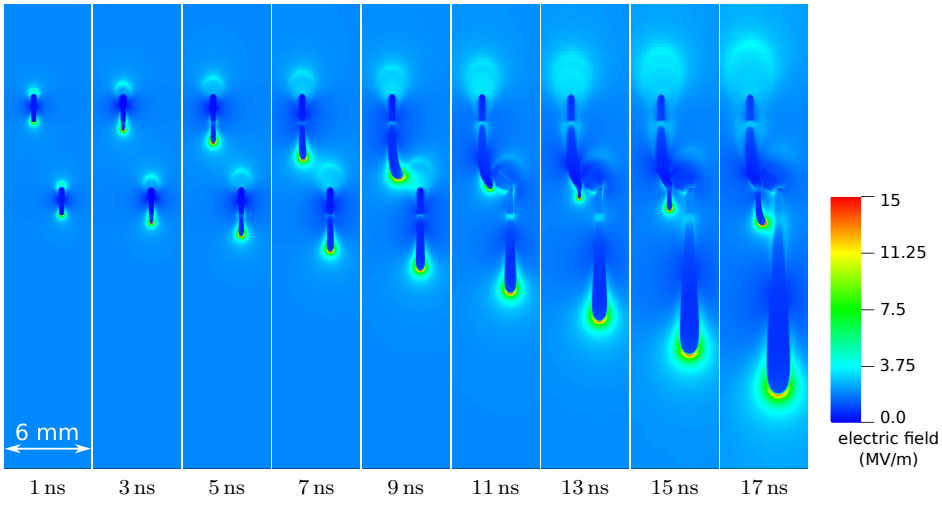

Figure 11: Cross section through the three-dimensional domain showing the time evolution of the electric field. The full height of the domain is shown (32 mm), but only 6 mm of the width.

[4] Richard D. Hornung, Andrew M. Wissink, and Scott R. Kohn. Managing complex data and geometry in parallel structured amr applications. *Engineering with Computers*, 22(3-4):181–195, Aug 2006.

[5] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, Apr 2000.

[6] Rahul S. Sampath, Santi S. Adavani, Hari Sundar, Ilya Lashuk, and George Biros. Dendro: Parallel algorithms for multigrid and amr methods on 2:1 balanced octrees. *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008.

[7] Tobias Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Technische Universität München, 2009.

[8] Stéphane Popinet. Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries. *Journal of Computational Physics*, 190(2):572–600, Sep 2003.

[9] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement. *A&A*, 385(1):337–364, Apr 2002.

[10] Donna Calhoun. Adaptive mesh refinement resources. [Online; accessed 17-July-2015].

[11] John Bell. Block Structured AMR Short Course: Lecture 1. [Online; accessed 17-July-2015].

[12] S. Pancheshnyi, P. Ségur, J. Capeillère, and A. Bourdon. Numerical simulation of filamentary discharges with parallel adaptive mesh refinement. *Journal of Computational Physics*, 227(13):6574–6590, Jun 2008.

[13] C. Montijn, W. Hundsdorfer, and U. Ebert. An adaptive grid refinement strategy for the simulation of negative streamers. *Journal of Computational Physics*, 219(2):801–835, Dec 2006.

[14] Chao Li, Ute Ebert, and Willem Hundsdorfer. Spatially hybrid computations for streamer discharges: II. fully 3D simulations. *Journal of Computational Physics*, 231(3):1020–1050, Feb 2012.

[15] A. Luque and U. Ebert. Density models for streamer discharges: Beyond cylindrical symmetry and homogeneous media. *Journal of Computational Physics*, 231(3):904–918, Feb 2012.

[16] Kevin Olson. Alpha version of paramesh multigrid support. [Online; accessed 17-July-2015].

[17] G.M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. *IBM Research Report*, 1966.

[18] Kitware. Paraview. [Online; accessed 22-July-2015].

[19] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. Chapman and Hall/CRC, Oct 2012.

[20] Achi Brandt and Oren E. Livne. *Multigrid Techniques*. Society for Industrial & Applied Mathematics (SIAM), Jan 2011.

[21] U. Trottenberg, C.W. Oosterlee, and A. Schuller. *Multigrid*. Elsevier Science, 2000.

[22] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial (2^{nd} Ed.)*. Society for Industrial & Applied Mathematics, Philadelphia, PA, USA, 2000.

[23] Wolfgang Hackbusch. Multi-grid methods and applications. *Springer Series in Computational Mathematics*, 1985.

[24] D. Bai and A. Brandt. Local mesh refinement multilevel techniques. *SIAM Journal on Scientific and Statistical Computing*, 8(2):109–134, Mar 1987.

[25] Chao Li, W. J. M. Brok, Ute Ebert, and J. J. A. M. van der Mullen. Deviations from the local field approximation in negative streamer heads. *Journal of Applied Physics*, 101(12):123305, 2007.

[26] G J M Hagelaar and L C Pitchford. Solving the boltzmann equation to obtain electron transport coefficients and rate coefficients for fluid models. *Plasma Sources Science and Technology*, 14(4):722–733, Oct 2005.

[27] Saša Dujko, Ute Ebert, Ronald D. White, and Zoran Lj. Petrović. Boltzmann equation analysis of electron transport in a N_2-O_2 streamer discharge. *Japanese Journal of Applied Physics*, 50(8):08JC01, Aug 2011.

[28] Chao Li, Ute Ebert, and Willem Hundsdorfer. Spatially hybrid computations for streamer discharges with generic features of pulled fronts: I. planar fronts. *Journal of Computational Physics*, 229(1):200–220, Jan 2010.

[29] B. Koren. A robust upwind discretization method for advection, diffusion and source terms. In C.B. Vreugdenhil and B. Koren, editors, *Numerical Methods for Advection-Diffusion Problems*, pages 117–138. Braunschweig/Wiesbaden: Vieweg, 1993.