Gabriel Lefloch

19 April – 20 August, 2021

Visualization and analysis of tree-based Adaptive Mesh Refinement simulations

Internship report

Master's degree in scientific computing and modelling
at the University of Rennes 1

Receiving institution: CEA Paris-Saclay
Internship advisor: Damien Chapon
Scientific advisor: Loïc Strafella

Faculty advisor: Mariko Dunseath-Terao

Contents

1 *Acknowledgements*

I would like to greatly thank Mr. Damien Chapon for his advice and guidance during this internship, which was essential to my global understanding of the concepts at hand. I would also like to deeply thank Mr. Loïc Strafella for his help and time, which was vital to the progress of my work. I am also grateful for the help and support of Prof. Mariko Dunseath-Terao. I'd like to acknowledge the work and assistance of Sylvain Kern, creator of this Tufte-styled LaTeX class.

2 *The CEA*

The French Alternative Energies and Atomic Energy Commission (CEA) is a research and development establishment playing a role in four main areas: defence and security; low carbon energies (nuclear and renewable energies); technological research for the industry and fundamental research in the physical sciences and life sciences.[2] This internship took place in the Institute of Research into the Fundamental Laws of the Universe (IRFU), which is located in the Paris-Saclay center – one of the nine in France. To be more specific, it was held in the Laboratory of Software Engineering for Scientific Applications (LILAS). This laboratory is part of one of the seven departments of the aforementioned Institute, and is known as the DEDIP: Electronics, Detectors and Computing department. As per their website's description:[4] *"The main vocation of the Department of Sensor Electronics and Computer Science for Physics is to invent and build the ambitious and innovative detection instruments of the future, essential to the progress of physics studied at IRFU"*. The LILAS laboratory focuses on developing specific software tools and modules needed for the various physics experiments held in its department.

3 *Goals*

For over 10 years, the DEDIP has been developing efficient and practical tools for astrophysicists of the IRFU. These tools, such as PyMSES, are used to explore, analyse and visualize simulation outputs with complex data structures, namely tree-based AMR grids. In order to process these massive data outputs, the purpose-specific scientific tools need to be remarkably optimized. A recently developed data structure called *vtkHyperTreeGrid* specifically made for AMR trees made its way in the C++ library VTK. Although this data format thrives on the plethora of generic VTK tools, it lacks specific data processing functionalities. That is why the internship will focus on the development of new efficient data processing algorithms for the vtkHyperTreeGrid format, in collaboration with the teams of the Mili-

tary Applications Division (DAM) and Fundamental Research Division (DRF).

Namely, we aim at developing a particle mapping tool, 1D profiling tool and a sub-region extraction tool. Seeing as all three have been successfully implemented, we also take on the making of a volumetric data splatting tool.

In the long term, these algorithms are to be integrated into a future Python package called PyMSES 5.0. The novel package will provide useful tools for not only the astrophysicists of the IRFU, but more broadly to the entire scientific community using point-wise AMR grids.

4 *Introduction*

Preamble. With a growing demand to visualize and analyse large simulation data outputs, *Adapative Mesh refinement* offers a trade-off between numerical accuracy, memory footprint and computational cost. This structured meshing method is efficient at tracking fine details in large regions, whilst taking up to 80% less space in memory than other non-structured meshing strategies.[5] This report will focus on the presentation and analysis of several utilities that help exploit and process tree-based AMR structures.

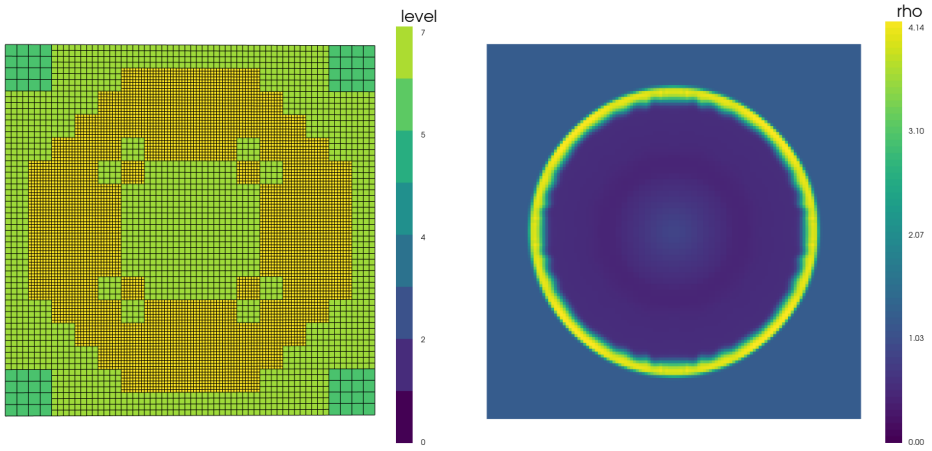
Context. The term *Adaptive Mesh Refinement* – originally coined in 1984 by J. Berger and J. Oliger[1] – is a method used to dynamically adapt the precision of a structured mesh in a numerical simulation whilst its solution is being calculated. AMR comes in response to a need for lighter meshing structures. Indeed, with the arrival of pre-exascale computing machines,[3] data outputs of large-scale simulations using unstructured meshes with fully described connectivity are too cumbersome to store and visualize. In contrast, AMR is a trade-off between numerical accuracy, memory footprint, computational cost and can therefore improve visualization capabilities as well as resource usage.

While there are several approaches to AMR – such as block-based or tree-based – the following report will solely focus on the latter flavor. This meshing strategy, also known as point-wise AMR, can be found in RAMSES, a massively parallel hydrodynamical code for self-gravitating magnetized flows. It can be used in a broad set of domains, like stellar formation; proto-planetary systems; supernova explosions; interstellar medium; galactic dynamics and formation of large-scale cosmological structures to name a few.[11] Its meshing method allows for deep spatial resolution without excessive memory overhead. However, its I/O performances were highly bottle-necked when scaled to many-core architectures: format specific binary files were used for restarts/checkpoints as well as post-processing, which led to unwieldy

and heavy outputs. What's more, the data format used to describe it's meshes was proprietary and demanded purpose-specific tools.[8;9;13] L. Strafella and D. Chapon's recent work[10] circumvents both dilemmas by integrating in RAMSES an efficient parallel I/O library called *Hercule*, and by splitting the main data outputs into two formats: one used for checkpoints (and therefore only read by the code itself), and another for post-processing called *LightAMR*. The latter format aims at giving a standard and self-descriptive way to characterize AMR trees, with memory efficiency in mind.

5 *The LightAMR standard*

AMR meshes are refined in areas of simulations that are turbulent or sensitive, so as to capture detail only where it is needed. Contrary to fixed resolution grids or meshes, this method allows for deep resolution on large-scale simulations. Tree-based AMR meshes are grids that are refined by subdividing their cells into sub-cells. As an example, let's consider a 2D "blast" simulation in which we follow the propagation of a mechanical wave through a bounded square space. The blast is most turbulent near the propagated wave departing the center. We should expect a higher number of cells in that region. As we can see in the figure 1, the level depth of the AMR grid in 2a matches the corresponding high density zones in 2b.



(a) Representation of the depth or level of the grid (i.e. the subdivision level).

(b) Representation of the density of the matter.

Figure 1. VTK 2D rendering of the tree-based mesh of a blast simulation. The image is a snapshot at $t \approx 0.06s$, with a box length of 1.0 (unitless).

The name *tree-based* AMR comes from the fact that each cell can be represented by a vertex of a tree. There is therefore a duality in the representation of this meshing strategy. LightAMR is a standard used to describe AMR trees. In fact, it is a first step towards unifying the different types of formats used to outline this meshing strategy. It is self-descriptive, in that it uses key-value pairs to give a name to each data entry. It is also memory efficient, because it fully exploits

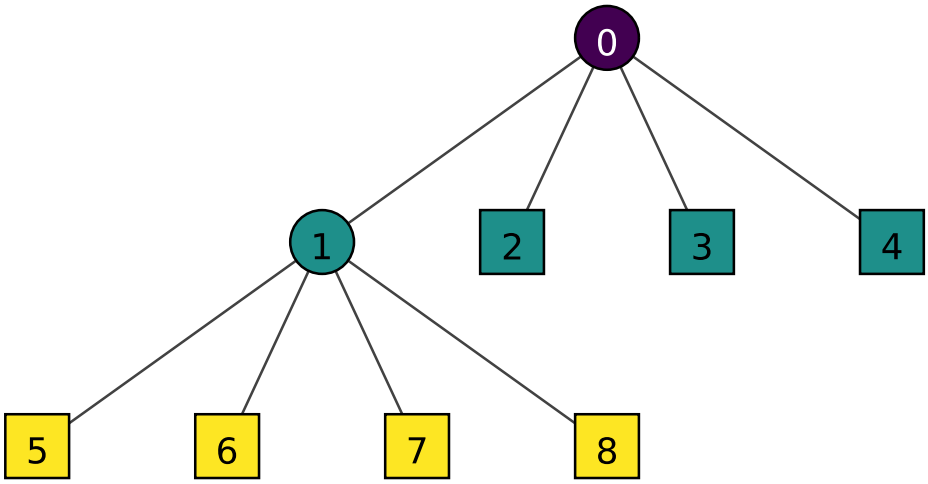


Figure 2. Depiction of a tree according to the definition 5.1. Each color of a vertex matches a tree depth level. Squares delineate leaf nodes, and circles strict nodes. The upmost vertex (in purple) is called the root. Finally, the labels are what describe the tree: 1 for parent and 0 for leaf.

the implicit structure of tree-based grids and can be compressed for storage with an algorithm tailored to the format.[7]

Before going any further in the description of the standard, let us define a *tree*:

Definition 5.1 (Tree)**.** Given a set of vertices V, we define an undirected edge with the following requirements:

(1) any two vertices are connected by a path,
(2) no set of edges can form a closed polygon.

We define the *root* of our tree by one and only one element of V. We call *leaf* a node that has no edge leaving it, and *strict node* a vertex that is not a leaf. Henceforth, a tree is defined by the couple (V, E), where V is the set of all vertices of the tree and E all its directed edges. The figure 2 gives us an example of a tree.

Remark. Let us note that directed edges are unambiguously deduced from the undirected ones, as any vertex of V has a unique path to the root.[1] From this, we define the *depth* – also known as *level*, see figure 5 – of a given vertex by the number of edges in its path from the root.

Remark.

- The root will always be represented at the top of the tree in this report. Thus, the tree is always a descending tree.
- The duality between the grid representation and the tree representation is unique, so long as we define an isomorphic mapping function between the set of vertices and the set of cells. As we'll see later on, a space filling function such as the Morton space filling curve can be used to get a unique vertex-cell pair.

Description of a tree in LightAMR. As we can see in the figure 2, each node label describes whether it is a leaf of a strict node. (0 being a leaf of the tree, 1 being a strict node). This is the basis behind the description of trees in the LightAMR standard. Indeed, AMR trees are detailed by a list or array called `isParentInt` comprised only of zeros

1 Indeed, according to (1) any vertex can be connected to the root by a given path, and the axiom (2) ensures the unicity of that path (multiple paths would lead to a closed polygon)

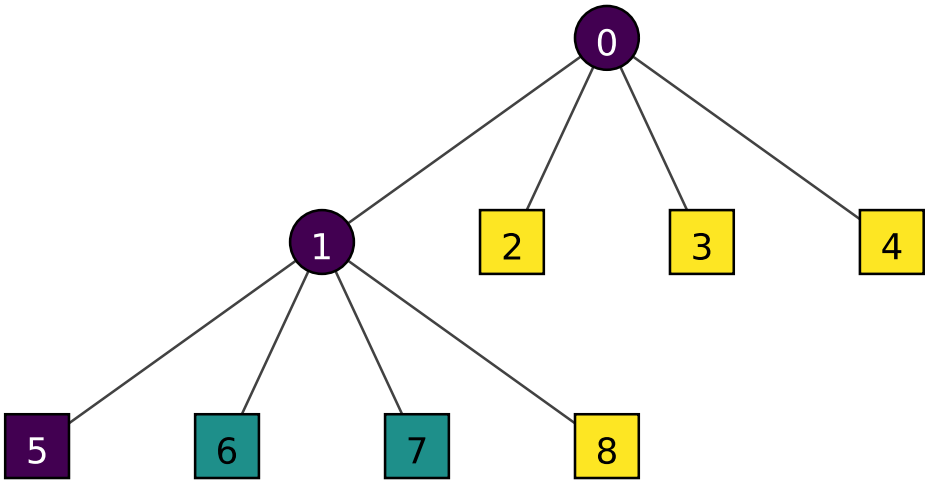


Figure 3. Domain decomposition of the previously considered tree. In this example, there are three domains indicated by the respective colors.

and ones. As an example, the `isParentInt` description of the figure 2 would be:

```
isParentInt:  1 | 1 0 0 0 | 0 0 0 0
```

More generally, if we define a 1 as being a strict node (that will necessarily have child nodes) and a 0 as a leaf node (indicating the end of a branch), from any `isParentInt` array we can deduce a corresponding tree representation, given two crucial parameters that are the dimension $d \in \mathbb{N}^*$ and the branching factor $f \in \mathbb{N}^*$. The number of children of each strict node is given by f^d.[2]

2 For the figure 2, the branching factor f is 2 and the dimension is 2. There are therefore $2^2 = 4$ children at each strict node, as is depicted.

In addition to the `IsParentInt` array, lightAMR also comes with an array called `IsMaskInt`, which describes which vertices are to be displayed/taken into account or not. `IsMaskInt` is equally composed only of 0's and 1's – a 1 signifies that the point is hidden or *masked* while a 0 signifies the opposite. This array description will come in handy for both visualization and domain decomposition, as we'll see in forthcoming sections.

Domain decomposition. In massively parallel simulation codes like RAMSES, many calculations or processes are carried out simultaneously over many physical processors so as to gain in available computation memory and lower the wall-clock time of the large simulations. In the case of tree-based AMR grids, carrying multiple calculation processes on a same tree can be done by separating the tree in multiple domains and then concurrently do the calculations on each domain. That can be done by partitioning the grid into multiple sub-blocks. There are multiple methods for domain decomposition of the grid among the processors, such as the famous Peano-Hilbert space-filling curve. The figure 3, based of the figure 5 illustrates an example of a domain decomposition. In this case, there are three distinct domains.

Checking the coherence of LightAMR. When a tree-based AMR structure is saved to a file using the LightAMR format, we may add a number of additional arrays and variables so as not to have to re-compute properties of the tree every time we need them. These arrays can be for example: the number of vertices per level, the number of domains, the refinement factor f, the dimension d, the total number of cells, the total number of levels per domain, etc. The HDF5 file format can be used to store these details, as it employs a data format that stores key-value pairs of information in a hierarchical fashion.

However, as some errors can slip into the tree descriptions per domain, the coherence of the `isParentInt` and `isMaskInt` arrays has to be verified according to the other pre-computed or predefined variables. e.g: does the number of vertices per level array match the actual number of vertices in `isParentInt`? Is the `isMaskInt` coherent?[3] etc. These verifications are listed in the appendix as Python implementations and use HDF5 file structures.

3 If a vertex is marked as masked, but one of its child vertices is marked as not masked, there is a coherence problem. Indeed, if the parent vertex is masked, all the children must be aswell.

6 *A need for a specialized tree-based AMR object: The Hyper Tree Grid*

Point-wise AMR grids lack widespread support in large visualization libraries, and generally[4] have to be converted into well known unstructured meshes in order to be visualized, which defeats the purpose of our tree-based grids. Moreover, their peculiar data-structure demands specialized tools in order to traverse and manipulate them. The DAM's recent work circumvents this dilemma by integrating tree-based data structures and tools[5] into the well established visualisation library VTK allowing for a scalable and feature rich tool-set to visualize AMR based simulation outputs. The object is called the *HyperTreeGrid*, and was envisioned with 2 goals in mind: integrate the object into the VTK ecosystem in order to gain compatibility with the wealth of VTK tools and design high performance algorithms specific[6] to tree-based AMR grids. Let us define a Hyper Tree Grid:

4 There are visualization and analysis tools that have tree-based AMR grid implementations, such as the yt project. [13]

5 Namely special constructs to traverse the trees and iso-contour filters

6 e.g : for the traversal of a tree, specialized constructs called cursors and supercursors were developed.

Definition 6.1 (Hyper Tree Grid)**.** In dimension $d \in \mathbb{N}^*$ with a branching factor $f \in \mathbb{N}^*$,[7] a HTG object is a type of data structure that can be represented as a tree where each strict node has exactly f^d children. Data can be attached to any one of its vertices.

7 In our case, d and f are restrained to: $d \in \{1, 2, 3\}$, $f \in \{2, 3\}$

Visual representations. The HTG object intrinsically has two visual facets: it may be represented as a tree graph or as a set of geometric shapes, as seen in the figure 5. Let us note that there is a trivial bijection between each node of (a) shape of (b), so long as we define the order in which the node indexes appear in (b). This order is given by the Morton space filling curve, which works in both 2D and 3D as seen in the figure 4.

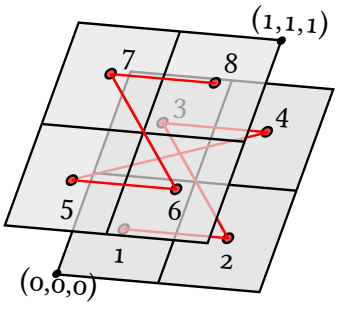
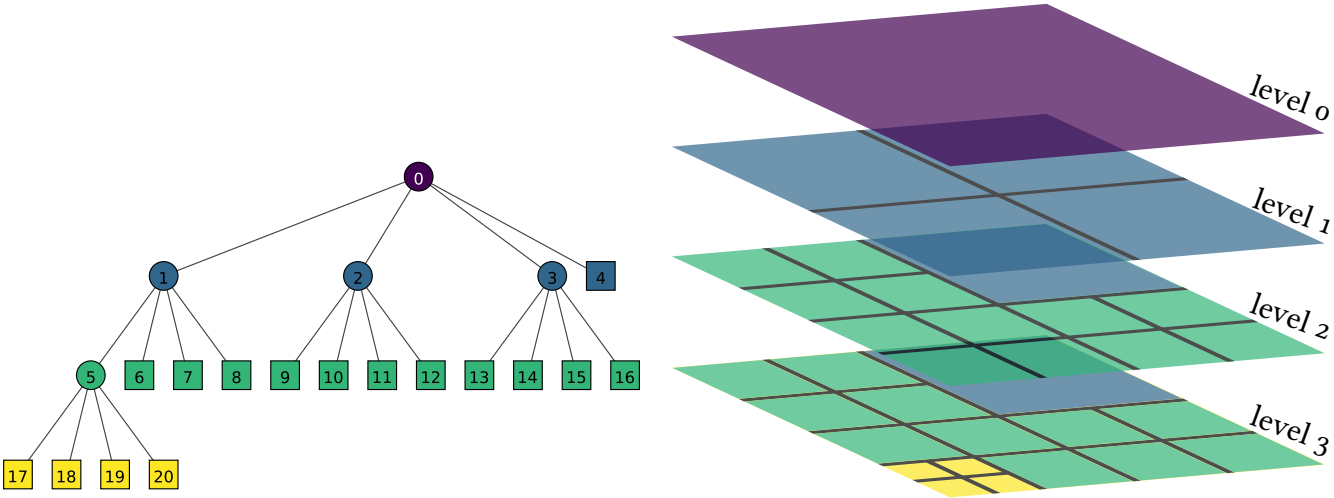


Figure 4. This schematic reflects how the morton space filling curve is used to define the order in which nodes of a tree are represented in a 3D HTG.



(a) Tree representation.

(b) Stacked levels of geometric representation.

Figure 5. Two representations of a HTG object with $d = 2$ and $f = 2$. On the left (a) is a tree graph of the object, while on the right (b) is a layered decomposition of a VTK-like rendering. Each color represents a level from 0 to 3. In (a), the numbers on each vertex express the global node indexes.

Remark. In fine, only the level 3 of the geometric representation (b) in the figure 5 is used to represent a HTG in VTK.

Implementation. As mentioned earlier, the heart of the object is baked into a C++ scientific visualization library known as VTK. Any action on a HTG object is therefore done through the library's already existing paradigms.[8] Creating a HTG is done by instancing it's VTK object, then by using a special construct called a *cursor* to define and navigate the branches of our tree. In fact, *cursors* (see definition 6.2) are a core element of the object as they are designed to fully take advantage of the implicit geometry of trees: *Depth-First search* is generally used to descend the tree and, depending on the used cursor, certain pre-computations are done to have access to neighboring cell information.

8 For instance, adding data the cells of a tree demands the usage of special VTK data structures such as C++vtkPointData.

Definition 6.2 (Cursor)**.** For a given Hyper Tree Grid H, a *cursor* is a structure pointing to one of the vertices of H used to traverse and access the nodes of its structure.

While there are several cursor implementations that vary in complexity, all are equipped with two fundamental functions:

```
ToChild(i) : move to the child of index i,
             relative to the current vertex.
ToParent() : go up one vertex (except if at root).
```

The other implementations – known as *supercursors* – build on top of these primitives, adding functionalities that give access to neighboring cells; the coordinates of the cell centers; bounds of the current cell; etc.

Memory storage. Finding an efficient way to store a HTG object is critical, as certain simulation AMR outputs can be quite sizeable. Let's consider a 2D "blast" simulation in which we follow the propagation of a mechanical wave through a bounded space, depicted in the figure 1. The resulting AMR tree has 12629 vertices and 8 levels of depth. The memory size of the *Hyper Tree Grid* VTK object is 47456 Bytes, while the equivalent representation of the tree with an unstructured grid object takes up 735232 Bytes: the unstructured grid object is over 15 times bigger than it's HTG counterpart. This difference in memory size is achieved by only storing all strict nodes and the index of their first children nodes, also called *eldest nodes*.

However, the VTK Hyper Tree Grid Object isn't particularly well adapted to storing AMR meshes on the long term. Large RAMSES outputs can have tree depth of up to 16, amounting to over $\sum_{k=0}^{15} 8^k = \frac{1-8^{16}}{1-8} \approx 4 \times 10^{13}$ vertices (assuming that every branch is fully subdivided). That yields 160 Tera Bytes, if one cell is stored in 4 Bytes. That is a job much more well-suited for the lightAMR standard.

7 *Particle reading and mapping*

The Hyper Tree Grid object is capable of attaching VTK Data structures to its leafs, such as scalar or vector fields for subsequent visualization or data processing. However, the data must first be mapped to each cell of the HTG: for point particles, the spatial geometry of the HTG and the particle coordinates have to be taken into account so as to correctly map the position of the particles to the HTG.

The particles are stored according to the LightAMR format in HDF5 files.[6] They are accessed through the HDF5 Python library, which gives access to the groups and attributes of a kin file with dictionary-like data structures. In our case, the particles are stored as sub-groups of domain groups, as laid out in the figure . To be more precise, they are stored as Python dictionaries, where the keys are domains and the values are dictionaries of the particle data arrays. The following code snippet outlines the data-structure of the resulting read particles.

```
particle_dict = {
  # Particles of domain 0
  "0": {"ids": np.array([...]), "position_x": np.array([...])
    , ...}
  # Particles of domain 1
  "1": ...,
  ...
}
```

As we can see, each domain key has a dictionary of arrays. The full list of keys is: `"ids"`; `"mass"`; `"position_x"`; `"position_y"`; `"position_z"`. Most are self-explanatory, except the "ids" key that is a unique identifier of each particle generated by RAMSES during the simulation.

Mapping the particles requires time efficient algorithms, as the AMR trees can be quite sizeable. Indeed, for an HTG with a depth of 15 levels (typical in large simulations), there can be in a worst case scenario up to $\sum_{k=0}^{15} 8^k = \frac{1-8^{16}}{1-8} \approx 4 \times 10^{13}$ cells.

That is why we'll compare and benchmark four candidate algorithms that map particles to a tree-based AMR grid. The first traverses the HTG once only and directly finds all the particles that fit in each cell. The second uses a different strategy: for each particle, we descend the tree and find the corresponding cell, skipping the parent cells that do not contain the particle. For the last two, instead of mindlessly descending the tree testing which cell contains the particle, we directly calculate in which quadrant or octant the particle is located and descend into it. The difference between the last two is that the third is implemented with a recursive function while the fourth is iterative.

Let us note that a recursive algorithm is used in this tool to descend the tree, and is structured as laid out in the code snippet 1. The recursive strategy is used throughout the report in light of the way cursors are implemented; contrary to the iterative analogue, it is quite straightforward to implement a recursive descent algorithm. The iterative version would demand checking and using the `isParentInt` and `isMaskInt` lists. Furthermore, as we will see in the sections below, there are some relatively easy-to-implement optimizations to this descent method. In this recursion, the cursor descends the tree level-by-level first, and then explores the breadth of the tree, as seen in the figure 6.

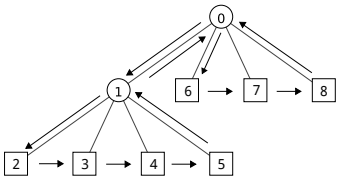


Figure 6. Order in which the nodes are met by the cursor.

Listing 1. General algorithm used to fully descend and explore an AMR tree. The cursor descends the tree down to the level defined by `maxLOD`, which stands for maximum level of detail.

```
def descendTreeRec(cursor, maxLOD):
    if( cursor.GetLevel() >= maxLOD ):
        return

    if( cursor.IsLeaf() ):
        return
    else:
        for ison in range(cursor.GetNumberOfChildren()):
            cursor.ToChild(ison)
            descendTreeRec(cursor, maxLOD)
            cursor.ToParent()
```

Two main benchmarking tests will be led. For the first, we fix the dimension d, raffinement factor f and maximal depth of the tree `maxLevel`. Then we evaluate the mapping time of each algorithm in $n = 10$ points, where each point is a set of randomly generated particles. The sets range in cardinality from `minN` to `maxN`. We are therefore evaluating the mapping time as a function of the number of particles to map. For the second, we fix the dimension d, raffinement factor f and the particle dataset that will be mapped with a cardinality of $p \in \mathbb{N}^*$. Then we retain the mapping time of the particle dataset over 10 different random AMR trees of increasing depths (ranging from `minDepth` to `maxDepth`. Thus, we evaluate the mapping time as a function of the depth of the tree.

Both benchmark results can be found in the figures 7 and 8. For the first test, all the curves seem to scale linearly. The third and fourth algorithms outmatch the two first in the three dimensional case. However, in 2D the first algorithm bests all the others.

For the second test, the curves are clearly no longer linear. The first algorithm is most notable, as it seems to follow an exponential growth in both dimensions. This can be an obvious pre-indicator that it does not scale well over large AMR tree structures. While the second algorithm fares the worst in 2D, is still leads ahead of the first in 3D. In both dimensions, the algorithms three and four lay unsurpassed.

Results analysis. Both benchmark reveal that the first algorithm can be used as a mapping strategy in 2D, so long as the tree is not exceedingly deep. Indeed, the figure Figure 8 proves that the algorithm scales terribly in deep (depth=13+) trees, no matter the dimension. This is to be expected, as the first algorithm uses a binary search algorithm to filter out all the particles that fit in the x bounds of a cell. From the found x-matching particles, we select the ones that match the y bounds and then z bounds if in 3D. The extra evaluation done in 3D dramatically lowers the overall performance.

The better performance scaling of the algorithms 2,3,4 observed in the figures 9a,9b is directly linked to the *cursors* used to navigate the tree. These constructs are extremely lightweight and fully exploit the tree nature of point-wise AMR grids for moving from vertex to vertex. In all the other cases, the algorithms 3 and 4 are quite comparable and seem to always outmatch the others. This can be explained by the fact that the descent to the correct cell is done automatically and optimally. There is no search, contrary to the algorithms 1 and 2. The figure 9 will help us illustrate the algorithm: to find in which quadrant the particle lies, we simply have to evaluate the following conditions:

$$x \in \left[x_{min}; \frac{x_{min} + x_{max}}{2}\right], y \in \left[y_{min}; \frac{y_{min} + y_{max}}{2}\right] \implies \text{P in quadrant 1} \tag{1}$$

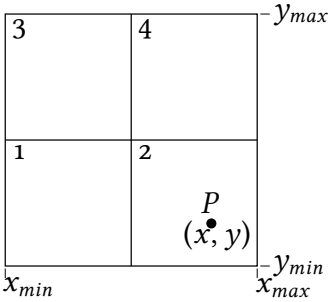
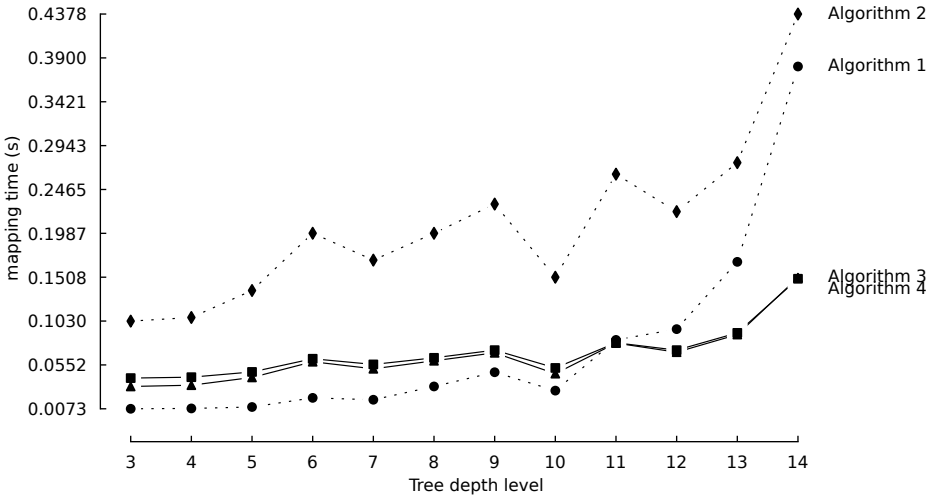
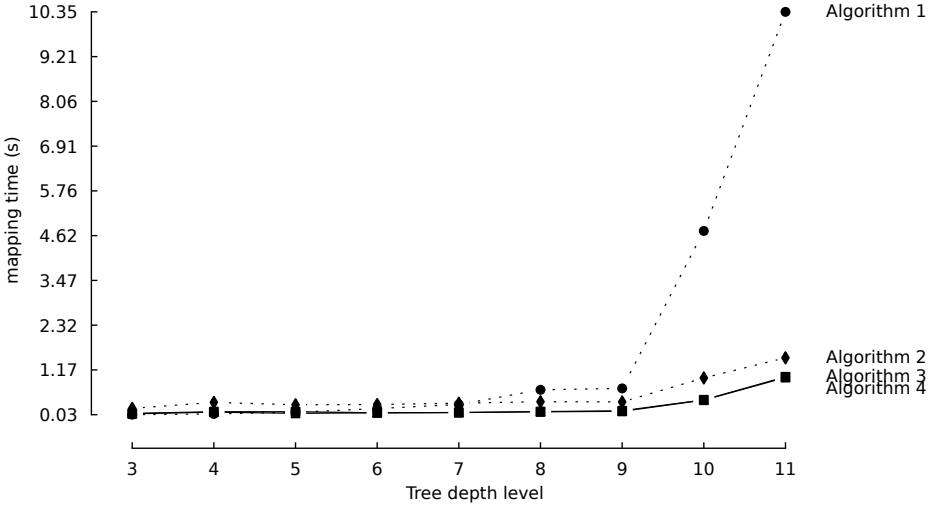


Figure 9. Schematic of a 2D cell subdivided into 4 subcells, ordered by the z-curve illustrated in figure 4. The black dot represents a particle P, with (x, y) as it's coordinates.



(a) 2D tree, $d = 2$.



(b) 3D tree, $d = 3$.

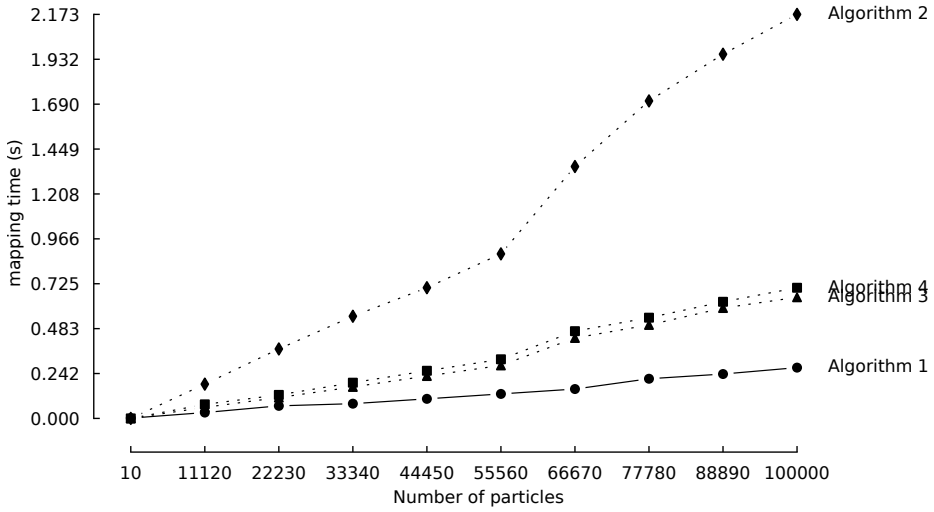
Figure 7. Mapping time of randomly generated particles ranging in numbers from 10 to 100000 using a random AMR tree with a maximal depth of 10. (`maxLevel` = 10, `minN`=100000, `maxN`=1000000, $f = 2, n = 10,$).

$$x \in \left[\frac{x_{min} + x_{max}}{2}; x_{max}\right], y \in \left[y_{min}; \frac{y_{min} + y_{max}}{2}\right] \implies \text{P in quadrant 2} \tag{2}$$

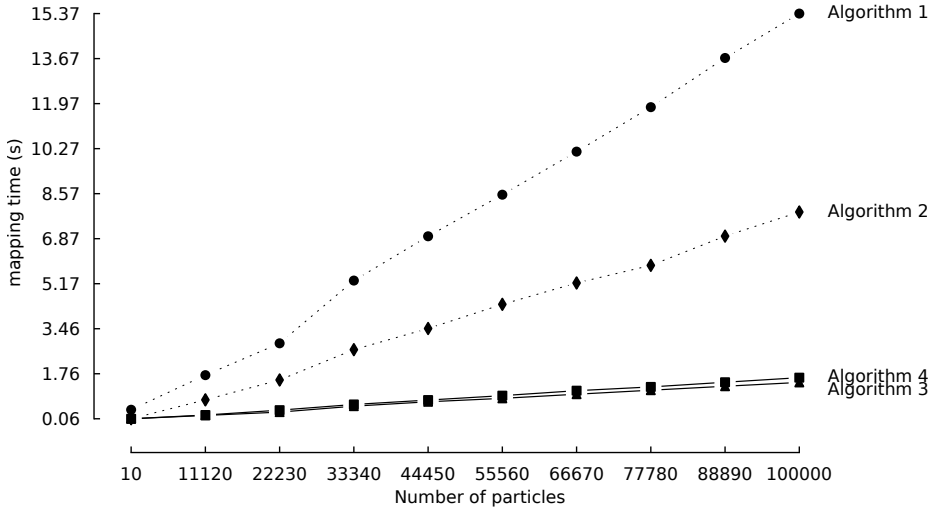
$$x \in \left[x_{min}; \frac{x_{min} + x_{max}}{2}\right], y \in \left[\frac{y_{min} + y_{max}}{2}; y_{max}\right] \implies \text{P in quadrant 3} \tag{3}$$

$$x \in \left[\frac{x_{min} + x_{max}}{2}; x_{max}\right], y \in \left[\frac{y_{min} + y_{max}}{2}; y_{max}\right] \implies \text{P in quadrant 4} \tag{4}$$

In the example given in the margin figure 9, it is the equation 2 that holds true. By iteratively (or recursively) applying this strategy, we directly find the leaf of the tree containing P.



(a) 2D tree, $d = 2$.



(b) 3D tree, $d = 3$.

Figure 8. Mapping time of a generated particle dataset onto 10 random AMR trees ranging in depth from 3 to 14. (`minDepth`=3, `maxDepth`=14, $f = 2, n = 10, p = 10000$).

Finally, we may ask ourselves whether it is best to use an iterative or recursive algorithm. By singling out both the algorithm 3 and 4 (figure 10) of figure 9b, we notice that the recursive flavor of the two is marginally faster. However, choosing between the two is of little importance: both perform virtually the same, and recursion is very language specific. What's more, recursion may be easier to read in well suited cases but may also use more memory than iteration.

To recap, we can look at the average times for both benchmark tests per dimension (figure 11). It is clear that either the methods 3 or 4 should be used. The first can be used in the specific case of 2D trees.

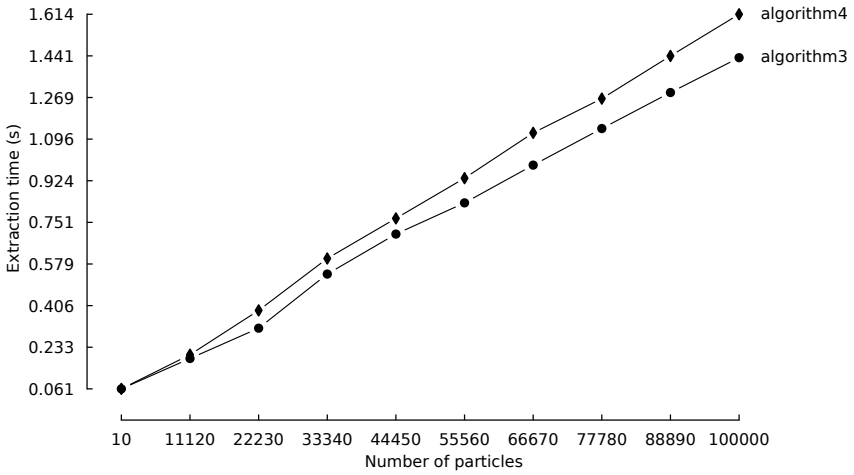


Figure 10. A plot of 9b, but only with the algorithms 3 and 4.

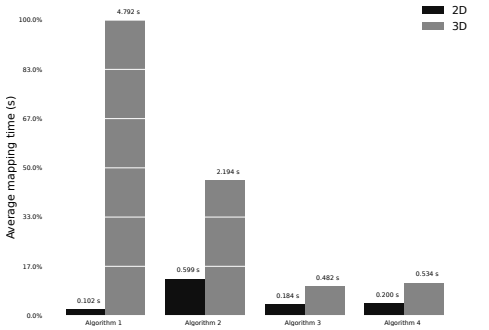


Figure 11. Average mapping times of both benchmarks per dimension.

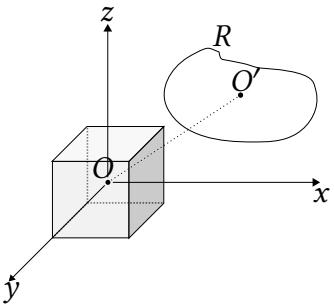


Figure 12. Illustration of the two coordinate systems of origins O and O', with $\mathcal{H}$ represented in O and R in O'.

Remark. The trees used for the benchmarks are randomly generated. The code used to generate them can be found in the appendix **??**.

8 *Extraction of a HTG sub-region*

Certain post-processing tools do not always require the entire HTG but only a sub-region of it. This is the case when we want to map particles to a smaller given region so as to lower calculation times, or analyse fields of a small part of the HTG and not the entire domain, etc. That is why we may try to implement a function that extracts and returns a sub-domain of a given Hyper Tree Grid object with a given Level-Of-Detail (LOD).

Method construction. Let us consider two Cartesian coordinate systems of origins O and O'. Both are are defined by the basis vectors $(\mathbf{u_x}, \mathbf{u_y}, \mathbf{u_z})$. Let $\mathcal{H}$ be a unitary cubic HTG object, centered to the origin O. We then define a region R as the set of all triplets $(x, y, z) \in \mathbb{R}^3$ meeting the condition $C(x, y, z)$ in the second coordinate system of origin O' . i.e:

$$R = \{\forall (x, y, z) \in \mathbb{R}^3 \mid C(x, y, z) \text{ holds true}\}$$

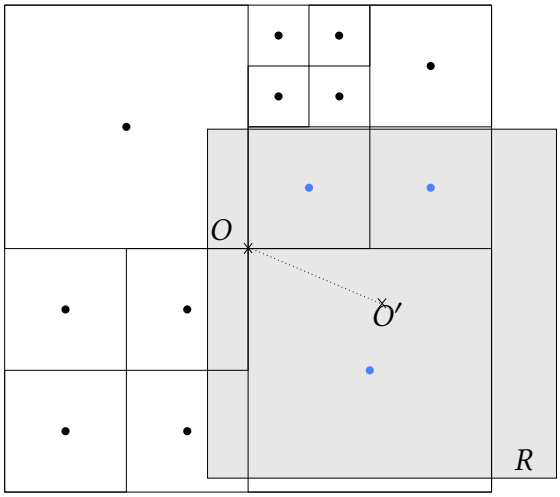


Figure 13. 2D Illustration of the conditions used to determine if a cell in a given region R. Each cell center is depicted by a dot. The blue dots are the cell centers inside the region R. This concept is trivially extended to 3D space.

The figure 12 gives an example representation of the two coordinate systems in O and O'. A cell of $\mathcal{H}$ can be considered in R if it's center point coordinates are in R (see illustration 13). As a initial solution, we could descend in every cell of the tree down to a certain LOD and simply check if the cell center point is contained in R. However, the center point $(P_{\text{center}})_O = (x_P, y_P, z_P)$ of a cell of $\mathcal{H}$ is expressed in the $(O, \mathbf{u_x}, \mathbf{u_y}, \mathbf{u_z})$ coordinate system and needs to first be expressed in $(O', \mathbf{u_x}, \mathbf{u_y}, \mathbf{u_z})$, where $O' = (x_{O'}, y_{O'}, z_{O'})$:

$$(P_{\text{center}})_{O'} = (x_{P'}, y_{P'}, z_{P'}) = (x_P - x_{O'}, y_P - y_{O'}, z_P - z_{O'})$$

In this report, only cuboidal and spherical regions are implemented. However, other types of regions can easily be implemented into the `Region` class. For a spherical region R_{sphere}, C_r may be defined as follows:

$$C_r(x, y, z) = \begin{cases} 1 & \text{if } x^2 + y^2 + z^2 \leq r \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where r is the radius of R_{sphere}. For a cuboidal region R_{cube}, the condition function $C_{w,h,l}$ takes three parameters $(w, h, l) \in \mathbb{R}_+$ which are respectively the lengths along the x-axis, y-axis and z-axis of the cuboid. $C_{w,h,l}$ is defined as such:

$$C_{w,h,l}(x, y, z) = \begin{cases} 1 & \text{if } \begin{array}{l} x_{O'} - -\frac{1}{2}w \leq x \leq x_{O'} + \frac{1}{2}w, \\ y_{O'} - -\frac{1}{2}h \leq y \leq y_{O'} + \frac{1}{2}h, \\ z_{O'} - -\frac{1}{2}l \leq z \leq l_{O'} + \frac{1}{2}l \end{array} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

Implementation. The extraction tool is to be implemented in python. It can be envisioned as a class `HyperTreeGridExtractor` that inherits from a parent class `Region` which defines regions and a method for finding if a point is in the region, as depicted in the figure 14. Such a class structure allows for a modular approach to how

the regions are defined and checked, because there is no modification needed in the `HyperTreeGridExtractor` class. Regions can be added and checked in the `Region` class, according to the users' needs. The code extract 2 is a minimal structure of the fully implemented `Region` class. At its instantiation, it initialises the center and bounds of the domain R, as well as a class method name `regionCheckFunction` used to check the HTG. A generic method `self.isInRegion()` is assigned to `regionCheckFunction` and can be used without further specification of the region in the subclass `HyperTreeGridExtractor`. The code extract 3 is also a barebone structure of the full implementation. On instantiation, its parent class is initialized and the `self.extract()` method can be called. The `extract` method invokes a recursive function `self._recursive_extraction()` that navigates the flagging[9] which cells are in the aforementioned region and which aren't.

Once we have updated the `isMaskInt` list by flagging the cells that aren't in R, we use a method designed to prune the branches that are masked. This reduces the original HTG $\mathcal{H}$ to a new smaller one $\mathcal{H}'$ that is encapsulated by the region R.

9 Cells are flagged by specifying if they are masked or not. A masked cell is designated by a 0 in `isMaskInt`

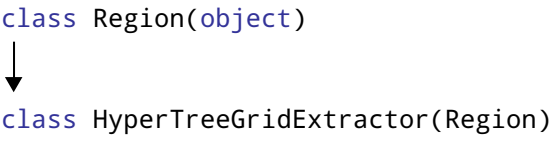


Figure 14. Class inheritance and structure of the extractor tool.

```python
class Region(object):
  _cellCenter = [0]*3

  def __init__(self, center, regionBounds,
    regionCheckFunction):
    # Center of region
    self._center = np.array(center)
    # Array like defining the bounds
    self._regionBounds = regionBounds
    # Define the generic method isInRegion()
    # used to check if cell is in region
    try:
      self.isInRegion = eval("self."+regionCheckFunction)
    except:
      print("regionCheckFunction method is not defined in
    Region Class")

  # The following methods are the available names for
    regionCheckFunction
  def isInSphericalRegion(self):
    # Code to check if _cellCenter is in region
    pass
    return True
```

Listing 2. Main structure of `Region` class. Region parameters are passed in the `__init__` function, so as to define it's center, bounds and check function used to validate if a cell is in the region in a generic fashion.

```
21
22    def isInCuboidalRegion(self):
23      # Code to check if _cellCenter is in region
24      pass
25      return True
26
27    # Other region functions
```

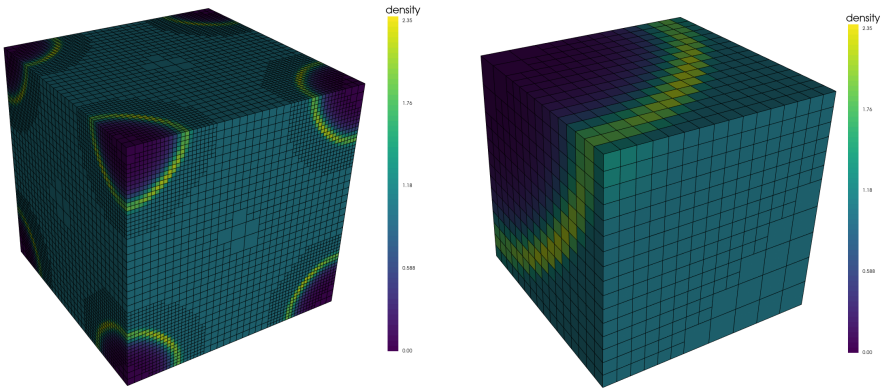
```
1  class HyperTreeGridExtractor(Region):
2
3    def __init__(self, center, regionBounds,
       regionCheckFunction):
4      # Initilaize the region parent class
5      super().__init__(center, regionBounds,
       regionCheckFunction)
6
7    def extract(self, HTG):
8      # Define a cursor
9      cursor = HTG.NewNonOrientedGeometryCursor(0, True)
10     # Recursively go through the tree
11     self._recursive_extraction(cursor)
12     # Remove the cells that are flagged as not in the
       region
13     clean_HTG = self.__magic_squeeze2(HTG)
14
15     return clean_HTG
16
17   def _recursive_extraction(self, cursor):
18     cursor.GetPoint(self._cellCenter)
19
20     # Use generic method defined in parent class
21     if( (not self.isInRegion()) and cursor.IsLeaf() ):
22       # Flag leafs to be removed
23       cursor.SetMask(True)
24       return
25
26     if( not cursor.IsLeaf() ):
27       for ison in range(0, cursor.GetNumberOfChildren()):
28         cursor.ToChild(ison)
29         self._recursive_extraction(cursor)
30         cursor.ToParent()
```

Listing 3. Main structure of `HyperTreeGridExtractor` class. As we can see, checking if a cell is in the region remains completely generic.

Let's examine an example in which we define a cuboidal region R with side lengths equal to 1,[10] centered in $O' = (1, 1, 1)$. We take a cubic Hyper Tree Grid $\mathcal{H}$ made from a blast simulation with side lengths of 2 and center $O = (0, 0, 0)$. This region therefore extracts the $(1, 1, 1)$ corner octant of $\mathcal{H}$. The figure 15 is the output result of the

10 $(w, h, l) = (1, 1, 1)$

code given in 3. As we can see, the mapped scalar density field in 16a has been retained 16b.



(a) Original HTG $\mathcal{H}$. **(b)** Extracted HTG $\mathcal{H}'$.

Figure 15. Example of an extracted region of a given Hyper Tree Grid Object $\mathcal{H}$, made from a blast simulation. On the left (a) is the original, with a scalar density field mapped to its cells. On the right (b) is an extracted region R of $\mathcal{H}$, where $R = \{(x, y, z) \in \mathbb{R}^3 \ | 0.5 < x, y, z < 1\}$.

Performance. It may be interesting to see how well the tool scales when extracting regions with increasing levels of depths, for a given HTG object. That is why we shall perform 12 extractions of two regions R_{sphere} and R_{cuboid} on the aforementioned blast simulation found in 13. Each extraction will be done with an increasing LOD ranging from 4 to 15, the maximal level of the simulation output. The resulting benchmarks are plotted in the figure 16. Unsurprisingly, the extraction times of both regions seem to have the same global variation trends, with the cuboidal region fairing overall less well than the spherical region. There are however more operations done for the cuboidal region than its spherical equivalent, as seen in the equations 5 and 6. To be more specific, $C_r(x, y, z)$ in 5 accounts for ≈ 7 operations while $C_{w,h,l}$ accounts for $6 * 3 = 18$ operations.

As for the scaling over tree depth levels of the region extraction tool, the results in the figure 16 are not very indicative of how well the tool performs. However, by plotting the region extraction times as a function of the descent times of a cursor descending the tree to each LOD (see 17, we notice a roughly linear relationship between the two. This proves that the tool will scale approximately as well as the highly optimized Hyper Tree Grid cursor object, or at least as well as the function `isInRegion()` which is what degrades the linear relationship found in 17. A poorly optimized `isInRegion()` function may therefore greatly hinder the extraction performances.

9 *Extraction of a 1D profile of a scalar field*

When displaying a 3D hyper Tree Grid object through VTK, only the outermost surface and its attached data fields can be represented. Yet, visualizing the internals of HTGs is a crucial element in the scientific

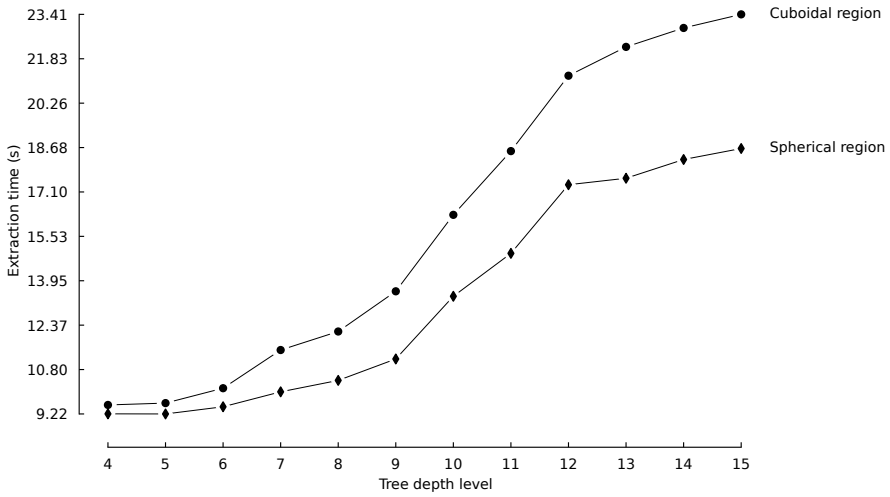


Figure 16. Extraction times of both regions as a function of AMR tree depth levels ranging from 4 to 15.

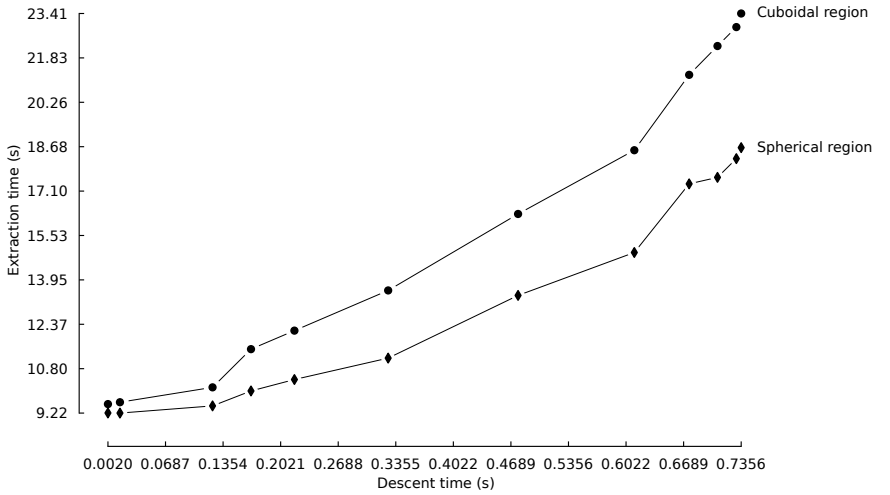


Figure 17. Extraction times of both regions as a function of the descent times of a cursor in the same tree. The descent and extraction times are done over levels ranging from 4 to 15.

analysis of simulation outputs. That is why we will examine two main approaches to the problem: both involve representing the intersection of a HTG object with either a 1D curve or a 2D surface. Seeing as the latter approach has already been tackled and integrated into VTK,[5] we will focus on the 1D intersection tool. The main goal of this tool is to extract the intersection between the cells of a HTG $\mathcal{H}$ and a line segment S in space. Once the cells are extracted, one can plot the values attached to them as a function of the length of the segment. This section will be based of an efficient Ray-box intersection algorithm.[12] We start by working in a 2D space for visualization purposes, then generalize to three dimensions.

Method construction. Let S be a segment in a Cartesian coordinate system of origins O. S is defined by two points $P_a = (x_a, y_a)$ and $P_b = (x_b, y_b)$. Let's now consider the parametric equations that defines S:

$$\begin{cases} x(t) &= x_a + (x_b - -x_a)t \\ y(t) &= y_a + (y_b - -y_a)t \end{cases} \quad t \in [0, 1] \tag{7}$$

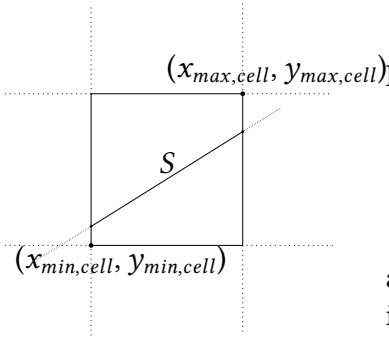


Figure 18. Illustration of 2D cell geometry.

From the equation 7, we solve for t and deduce the following:

$$\frac{x - x_a}{x_b - -x_a} = \frac{y - y_a}{y_b - -y_a} \tag{8}$$

Let C be a square cell defined by two points $C_{min} = (x_{min,cell}, y_{min,cell})$ and $C_{max} = (x_{max,cell}, y_{max,cell})$, as is shown in the illustration 18. S intersects the sides of C if the following equations hold true:

$$\begin{cases} x_{min} &\leq x \leq x_{max} \\ y_{min} &\leq y \leq y_{max} \end{cases} \tag{9}$$

$$\implies \begin{cases} \frac{x_{min,cell} - -x_a}{x_b - -x_a} &\leq \frac{x - -x_a}{x_b - -x_a} \leq \frac{x_{max,cell} - -x_a}{x_b - -x_a} \\ \frac{y_{min,cell} - -y_a}{y_b - -y_a} &\leq \frac{y - -y_a}{y_b - -y_a} \leq \frac{y_{max,cell} - -y_a}{y_b - -y_a} \end{cases} \tag{10}$$

When combining the equations 10 and 8, new minimal boundary intersection requirements can be derived. Consider $x = x_{min,cell}$, which gives

$$\frac{x_{min,cell} - -x_a}{x_b - -x_a} \leq \frac{x_{min,cell} - -x_a}{x_b - -x_a} \leq \frac{x_{max,cell} - -x_a}{x_b - -x_a} \tag{11}$$

$$10 \text{ and } 8 \implies \frac{y_{min,cell} - -y_a}{y_b - -y_a} \leq \frac{x_{min,cell} - -x_a}{x_b - -x_a} \leq \frac{y_{max,cell} - -y_a}{y_b - -y_a} \tag{12}$$

Therefore, S is out of the x-axis range $[x_{min}, x_{max}]$ of C if

$$\frac{y_{max,cell} - -y_a}{y_b - -y_a} < \frac{x_{min,cell} - -x_a}{x_b - -x_a}$$

If we note

$$t_{x,min} = \frac{x_{min,cell} - -x_a}{x_b - -x_a} \tag{13}$$

$$t_{y,min} = \frac{y_{min,cell} - -y_a}{y_b - -y_a} \tag{14}$$

$$t_{x,max} = \frac{x_{max,cell} - -x_a}{x_b - -x_a} \tag{15}$$

$$t_{y,max} = \frac{y_{max,cell} - -y_a}{y_b - -y_a} \tag{16}$$

then we directly deduce the two following boundary conditions that reveal if S intersects C or not:

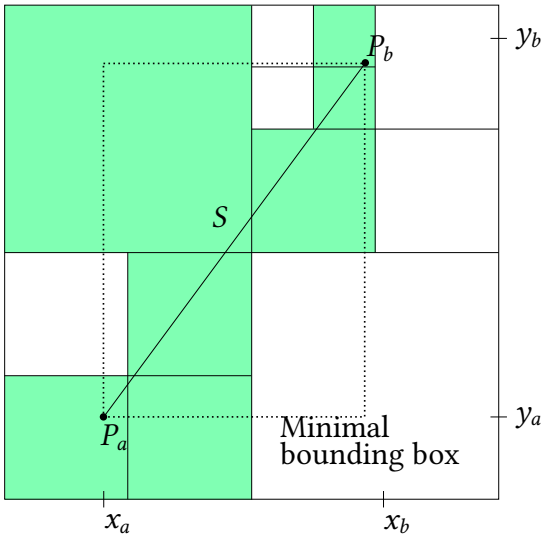


Figure 19. Illustration of the minimal bounding box of a segment S in 2D. The colored cells are the ones intersected by the segment.

1. $t_{y,max} < t_{x,min}$
2. $t_{x,max} < t_{y,min}$

The conditions are trivially extended to three dimensions by defining two extra variables t_{min} and t_{max} such that

$$t_{min} = \max(t_{x,min}, t_{y,min}) \tag{17}$$
$$t_{max} = \min(t_{x,max}, t_{y,max}) \tag{18}$$

These two variables inform us on the minimal bounding box of S. In 3D, we can now add the final boundary condition:

3. $t_{z,max} < t_{min}$
4. $t_{max} < t_{z,min}$

By once again calculating $t_{min} = \max(t_{z,min}, t_{min})$ and $t_{max} = \min(t_{z,max}, t_{max})$, we can at last verify if the limits of our segment S have not been exceeded. i.e:

$$0 \leq t_{min} \leq t_{max} \leq 1 \tag{19}$$

If 19 is not true, the cell is not in the minimal bounding box of S. The figure 19 gives an illustration of the minimal bounding of a segment S.

Implementation. The tool is implemented in Python as a class that takes 2 parameters to be initialized: `HTG`, `field`.[11] `HTG` is a Hyper Tree Grid object, and `field` is a string designating the name of the scalar field that will be intersected. This scalar field must already be mapped to `HTG`. The class has three public methods that a user may call – `findIntersection()`, `displaySegments()`, `plotData()` – and 5 private methods as outlined in the code snippet 4. `findIntersection()` can be called successively so as to extract multiple different segments with the same `extractLineData(object)` class.

When `findIntersection()` is called, we use `__initializeSegment()` to store the segment's intrinsic properties and elements necessary

11 Two others are optional `maxLevel=99`, `intersectionField=None`. `maxLevel` specifies the maximum tree depth level used when traversing the tree. pyintersectionField is the name of a scalar field that is attached to the `HTG` if it is specified. The field gives information on which cells have been intersected.

for testing intersections, such as the direction vector, the magnitude, etc. Additionally, we pre-compute some values used in the algorithm laid out in,[12] such as $\frac{1}{direction}$ for performance gains. After those initializations, we calculate the minimal tree depth level `minLevel` used when descending the tree with the usual recursive algorithm (see code block 1). This value is critical to finding data in the HTG when the segment is entirely enclosed by cells near the root and does not intersect them. Let d be the length of a given segment, L the length of the sides of a given cubic Hyper Tree Grid and l_{min} the minimal tree depth level. Then l is found by solving:

$$\begin{aligned} \frac{L}{2^{l}_{min}} &\leq d \\ \implies \frac{1}{\log(2)} \log \frac{L}{d} &\leq l_{min} \end{aligned}$$

Now that all the pre-computations have been done, we may descend the tree recursively while improving on the usual algorithm in listing 1. Instead of descending in every cell, we exclusively descend into the ones intersected by the segment S. This allows for a more direct descent to the desired cells, with considerable performance gains. The code block 5 is a Python implementation of the method.

```
class extractLineData(object):
  """
  Methods
  -------
  findIntersection()
  displaySegments()
  plotData()
  """
  def __init__(self, \Gls{HTG}, field, maxLevel=99,
    intersectionField=None):
    self.HTG = HTG
    self.dimension = HTG.GetDimension()
    self.maxLevel = maxLevel
    self.field = field
    ...
  def __initializeSegment(self):
    ...
  def __validatePoints(self):
    ...
  def __initMask(self):
    ...
  def __intersects(self, cellBounds):
    ...
  def __findIntersectionRec(self):
    ...
  def findIntersection(self, startPoint=None, endPoint=None
    ):
    ...
  def displaySegments(self):
    ...
  def plotData(self, logY = False, select = None):
    ...
```

Listing 4. Barebone structure of the `extractLineData` class, accompanied by its docstrings and comments. For the full code snippet, please refer to the appendix **??**

```
def __findIntersectionRec(self):
    self.cursor.GetBounds(self.bounds)
    IsLeaf = self.cursor.IsLeaf()
    intersects = self.__intersects(self.bounds)

    # Check if current cell intersects (unless we're before
     minLevel)
    return if( self.cursor.GetLevel() > self.minLevel )
    return if( not intersects )
    if( IsLeaf or self.cursor.GetLevel() == self.maxLevel )
    :
        # Add data
```

Listing 5. Recursive algorithm used to descend an AMR tree, while avoiding cells that do not intersect the segment S.

```
11          return
12      for ison in range( self.cursor.GetNumberOfChildren() ):
13          self.cursor.ToChild(ison)
14          self.__findIntersectionRec()
15          self.cursor.ToParent()
```

Extraction benchmarks. Following the previous benchmark efforts, we may evaluate how well the tool fares when extracting segments over increasing tree depth levels of a given tree, as well as over increasing lengths of segments. This will give us an idea of how the extraction tool performs depending on the used segment (see figure 20). Conjointly, the scalability of the tool can be tested by assessing the calculation times of a segment over varying tree depths (see 21). For these tests, we use a RAMSES output tree of a blast simulation in 3D space, the same as in 15. The tree has 16 levels of depth and is decomposed in 7 domains.

Results and analysis. The figure 20 unfolds an unexpected result: the shorter the segment, the longer the extraction time. Indeed, the highest extraction time of the subfigure 21a is 0.45s, is also smallest segment length (0.025), while the rest of the curve stagnates at 0.12ms. What's more, the subfigure 21b confirms the previous results by delineating in which groups the extraction time of 100 random segments lie. The general trend prevails: the shorter the segment, the longer the extraction time – or to be more precise, the longer the segment the shorter the extraction time. I believe that the main reason behind this behaviour is directly linked to the `minLevel` variable, which defines up to which minimal tree depth level the cursor has to descend to find cells that intersect the segment. The descent to `minLevel` is not optimized, as we cannot always compute its intersections with other cells; it may be entirely encompassed by a cell without touching the edges. For example, we consider a small segment of normalized length $d = 0.025$ in a unitary HTG. The minimal tree depth level would evaluate to

$$\frac{1}{\log 2} \log\left(\frac{1}{0.025}\right) \approx 5 \text{ Levels}$$

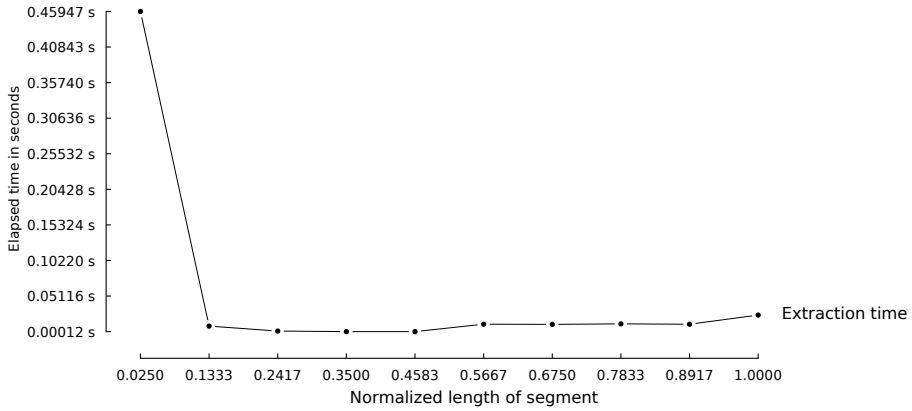
This example would amount to the descent of all the branches of the HTG up to level 5. When comparing the extraction time of such an example to the extraction time of a longer segment that would have a much more optimized descent in addition to a smaller `minLevel`,[12] the results portrayed in 20 may be deemed quite reasonable.

As for the figure 21, it is clear that extraction times increase linearly with the tree depth level. However, there seems to be an unexplained plateau for levels of depth 12 and up.[13] Further testing is required to validate whether the plateau is a recurrence or simply an artifact of the chosen segment. Even so, the results are encouraging as the tool

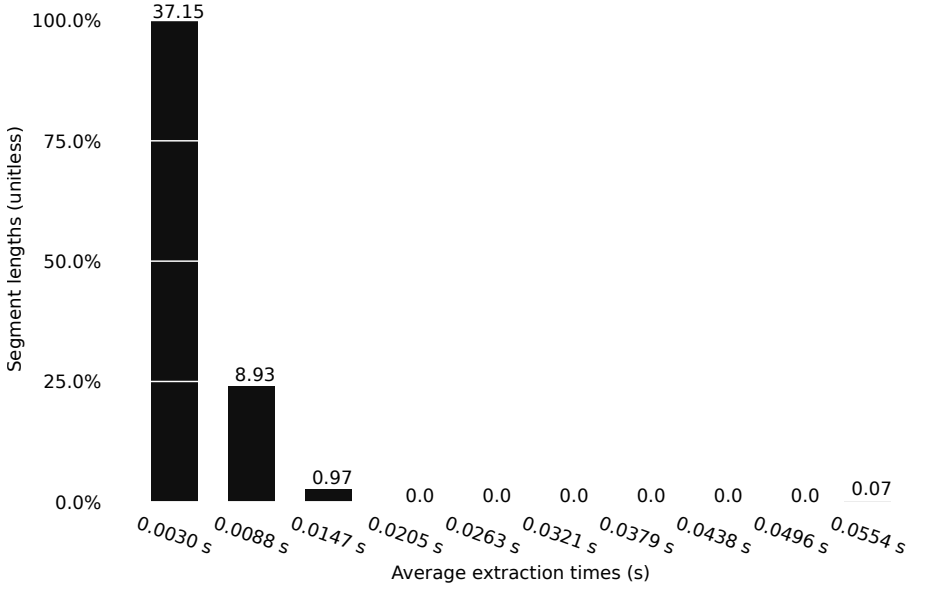
12 For example, if $d = 0.6$, `minLevel` ≈ 0

13 The AMR tree used has 16 levels of depth. The curve should keep increasing linearly up to 16 levels of depth

scales linearly in time over trees that grow exponentially in number of vertices. This is once again most probably linked to the optimization listed in the code snippet 5.



(a) Extraction time of 10 segments of same direction but increasing lengths.



(b) Histogram of extraction time of 100 random segments. The values atop the bars denote the exact segment lengths. (unitless)

Figure 20. Both figures exhibit how the tool behaves with different sizes of segments in a given AMR tree. On the left in (a), we visualize the extraction time of a segment increasing in length, but always with the same direction in space. On the right (b), is a histogram of 100 segments in varying length and direction, binned by their extraction times.
Note: each bin is represented by the average value of the time interval bin, not the time interval itself.

10 *Volumetric data Splatting applied to Hyper Tree Grid Objects*

Splatting is a method used to visualize volumetric datasets. In our case, it is used to project 3D AMR grid scalar fields onto a 2D surface, so as to have a global overview of a simulation output. Splatting has already been implemented for tree-based grids by the team behind the RAMSES post-processing tool *PyMSES*.[8] However, the method does

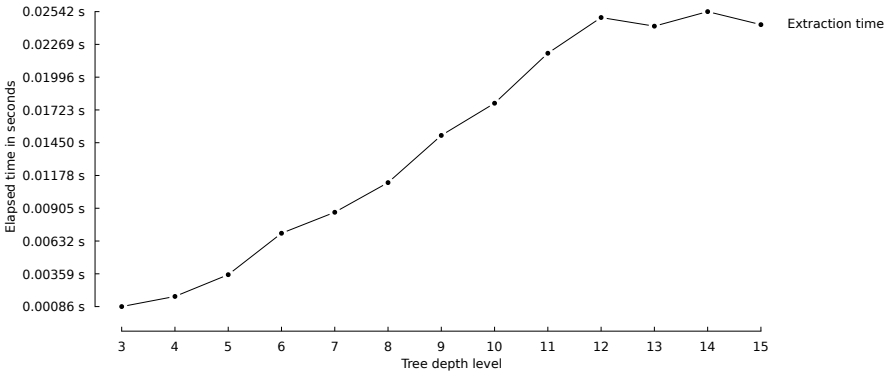


Figure 21. Extraction times of a segment for 13 different maximal levels of depth. The levels range from 9 to 15. The segment is defined by the points P_a = [-boxlen, -boxlen, -boxlen] and P_b = [boxlen, boxlen, boxlen].

not yet exist for VTK Hyper Tree Grid data structures. That is why we will focus on adapting a splatting method tailored to Hyper Tree Grid Objects.

Method overview. This splatting technique can be described over a few main steps, giving us an overview of what is done to retrieve a splatted Hyper Tree Grid image.

1. Retrieving scalar data of cells per level: for each level, we collect the scalar data attached to a cell, as well as the center point coordinates of the cell.
2. Projecting the retrieved data onto a given surface, per level: we project each cell center point that we have collected onto a given surface.
3. Data binning: once we have obtained projected points per level, we compute their 2D histograms. Each cell center point contributes its associated scalar field value to the histogram.
4. Convolving the histograms: we now have l histograms, which are basically (n, n) matrices. We convolve each one of them by a given convolution kernel – say a 2D Gaussian function. This kernel will depend on the size of the HTG cells. However, because of the implicit geometry of tree-based grids, we only need to know the tree depth level of the cell to find its size.[14] For a 2D Gaussian function, it's σ parameter will therefore be a function of the tree depth level.
5. Summing the obtained images: the final step consists in simply summing on a cell-by-cell basis all the convolved histograms. The resulting matrix can be interpreted as an image, where each one if its values is mapped to a color scale.

14 Let $\mathcal{H}$ be a square Hyper Tree grid with sides of length d. A cell of $\mathcal{H}$ at the level l will have sides of length $\frac{d}{2^l}$.

Projection. The first step is straightforward as it simply demands traversing the tree and retrieving cell information. The second less so, as there are multiple ways of projecting a point in space onto a 2D surface – namely perspective projections and parallel projections.

However, by lack of time, we will simply consider orthogonal projections. This method is fairly elementary, as it only involves the dot product of the point to be projected with an axis considered as the "depth" axis. The depth axis is the axis closest to the normal vector of the projection surface, and can be determined by calculating the angle separating the two.

Let $(\mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_z)$ be the basis vectors of a Cartesian coordinate system of origins $\mathcal{O}$. $\mathcal{P}$ is a plane defined by a point in space p and a unitary normal vector $\mathbf{n} \in \mathbb{R}$. The depth axis is found by calculating along which of the three basis vectors $(\mathbf{u}_x, \mathbf{u}_y, \mathbf{u}_z)$ the normal vector $\mathbf{n}$ has an angle α in $I = [-\frac{\pi}{4}; \frac{\pi}{4}]$, as illustrated in the margin figure 22. Once the depth axis is found, the projection p' of $p = (p_x, p_y, p_z)$ onto $\mathcal{P}$ is given by

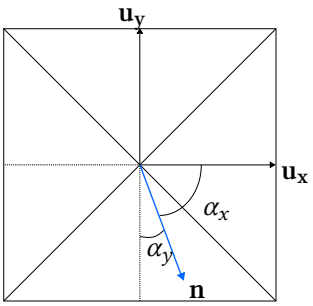


Figure 22. Illustrative schematic of how the depth axis is found, in two dimensions. In this figure, the α_y angle is the only one lying in I. Therefore $\mathbf{u}_y$ is the depth axis.

$$p' = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \tag{20}$$

where $(a, b, c) \in \{0; 1\}$. Only one of the three reals is null and designates the depth axis. If we take the figure 22 as an example, we would have:

$$p' = \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} p'_x \\ 0 \\ p'_z \end{bmatrix} \tag{21}$$

the following code block is an example Python implementation of how to determine the depth axis.

```
1  def findDepthAxis(self):
2      for index in range(3):
3        alpha = np.pi/4.0
4        n = self.normal[index]
5        if( -alpha < np.arccos(n) < alpha ):
6          return index
```

Listing 6. Python implementation of a method used to find the depth axis of a surface in a HTG object. `self.normal` is the normal vector $\mathbf{n}$ of the surface $\mathcal{S}$.

Binning. Now that we have a method to project our data points onto our surface $\mathcal{S}$, we may bin the projected points onto a grid, which is essentially the same as making a 2D histogram. The grid has the same length and height as the Hyper Tree Grid used, so that all the projected points fit in the dimensions of the grid. However, the granularity of the grid – or the number of cells – is given by a parameter that we may call `outputSize`, as it is this parameter what will dictate the final image resolution. Indeed, the produced grid is a matrix $\mathbb{M}$ where each value $m_{(i,j)}$ is the value of histogram grid cell (i, j). (see margin figure 23).

The figure 24 depicts all 16 levels of projected points of a RAMSES blast simulation output. The projected field is that of density.

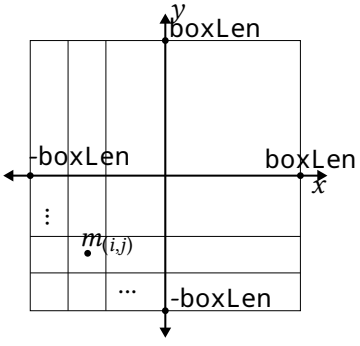


Figure 23. Depiction of the histogram grid used to bin the projected field points. `outputSize`^{2} is the number of cells and `boxLen` is the half of the total side lengths of a cubic Hyper Tree Grid.

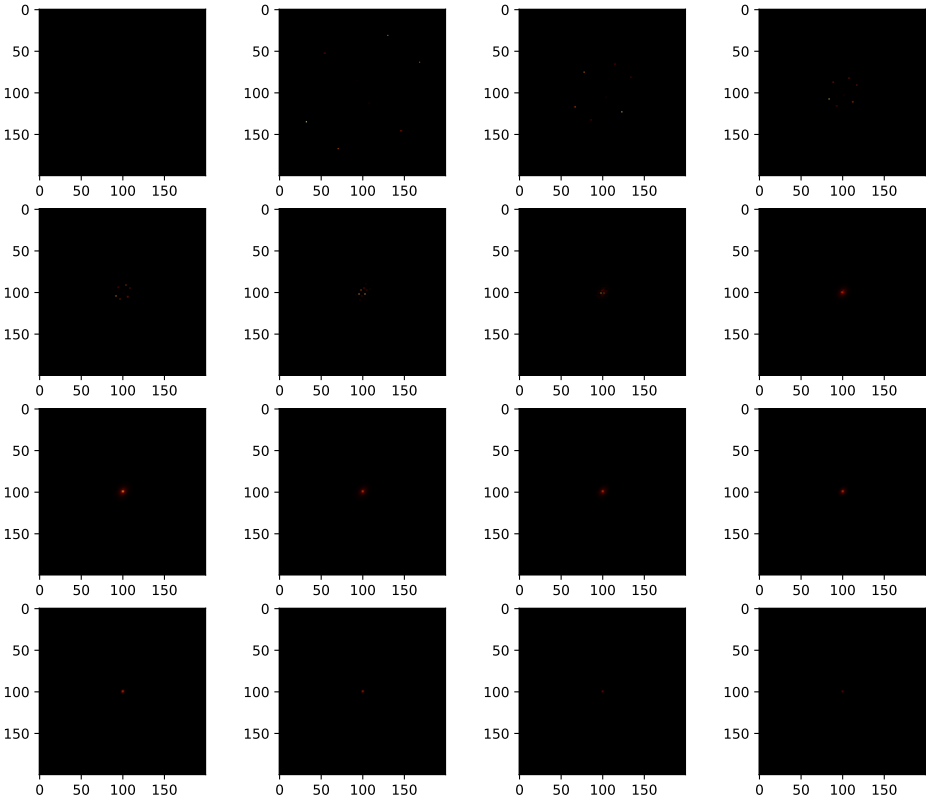


Figure 24. All projection images produced by the binning of the projected points of a HTG object of 16 levels. The HTG object is built with the output of a RAMSES blast simulation. The splatted field is a scalar density field.

Convolution. The matrices obtained from the binning process are therefore of size (`outputSize`, `outputSize`), and are convolved with a given convolution kernel. We choose the two dimensional Gaussian function g as a kernel:

$$g(x, y, l) = A_l e^{\frac{-x^2-y^2}{2\sigma_l^2}}$$

where $A_l = \frac{1}{2\sigma_l^2}$ and $\sigma_l = \frac{\texttt{outputsize}}{2^l}$.[15] A and σ are functions of the level l used to project the points. So the deeper the level l, the smaller the kernel g. That is also why convolving the full kernel images with the bin images would be inefficient, as we could retain the exclusively the positive parts of the kernels for convolutions. To do so, we need to define a minimal value ε that gives a threshold on what is considered null in the kernel images. From there, we can evaluate what dimensions of the kernel images we have to retain, in order to keep only the parts greater than ε.

15 σ is evaluated as the actual length of a cell in the level l.

$$\begin{aligned} & g(x, y) > \varepsilon \\ \implies & Ae^{\frac{-x^2-y^2}{2\sigma^2}} > \varepsilon \\ \implies & \frac{-x^2 - y^2}{2\sigma^2} > \ln\left(\frac{\varepsilon}{A}\right) \\ \implies & x^2 + y^2 < 2\sigma^2 \ln\left(\frac{\varepsilon}{A}\right) = r^2 \end{aligned}$$

The final equation is in fact the Cartesian equation of a circle of radius r. All we need to calculate to extract the positive values of the Gaussian kernels is:

$$x = y = \sqrt{2\sigma^2 \ln\left(\frac{\varepsilon}{A}\right)}$$

The figure 26 portrays Gaussian kernels images that have an original size of `outputSize` = (200, 200). The image sizes decrease as we descend in the levels. At the twelfth level, the kernel image is discarded as its values are too small. In order to convolve the projected points images that are of size `outputSize` (as seen in the figure 24) with the smaller kernel images, we simply pad the kernel images with zeros. This enables us to maintain the same sizes between the images. By reducing the size of the kernels used for convolution, we gain in calculation times, as we can see in the graph 25. The curve labeled "optimized" uses the technique we just laid out, while the other simply convolves the full kernel images of size `outputSize` with the projection images in 24. As we can see, the non optimized convolutions have longer calculation times and increase with a greater slope than the optimized counterparts.

The resulting convolutions are given in the figure 27.

Remark. The convolutions are computed with the convolution theorem 22, which uses Fourier transforms.

$$g(x_n, y_n) * i(x_n, y_n) = \mathcal{F}^{-1}\left\{G(x_n, y_n) \times I(x_n, y_n)\right\} \tag{22}$$

where $*$ denotes convolution, $\times$ point-wise multiplication, $\mathcal{F}$ the Fourier transform operator and G, H the Fourier transformed functions of g, h.

This convolution method is privileged over classic convolutions, as it employs highly optimized algorithms such as fast Fourier transforms.

Final result. Now that all the convolutions have been computed, we may simply add all the images together. The resulting image is a single matrix whose values are mapped to a color scale, as seen in 28. This figure gives us an insight on the global distribution in space of the density scalar field of the simulation output.

11 *Conclusion*

Summary. This report details the AMR-related topics I have investigated over the course of the internship. Before tackling the main problems of interest, I spent a preliminary time getting accustomed to the already existing code base as well as the general concepts relative to point-wise Adaptive Mesh Refinement. This groundwork allowed

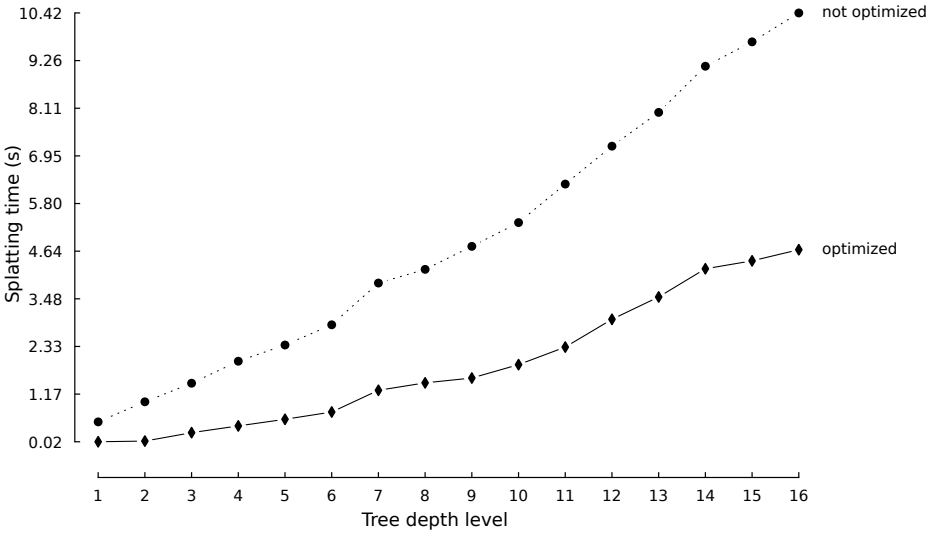


Figure 25. Comparison of the splatting times of the optimized kernel convolutions with the non optimized kernel convolutions over 16 levels of depth.

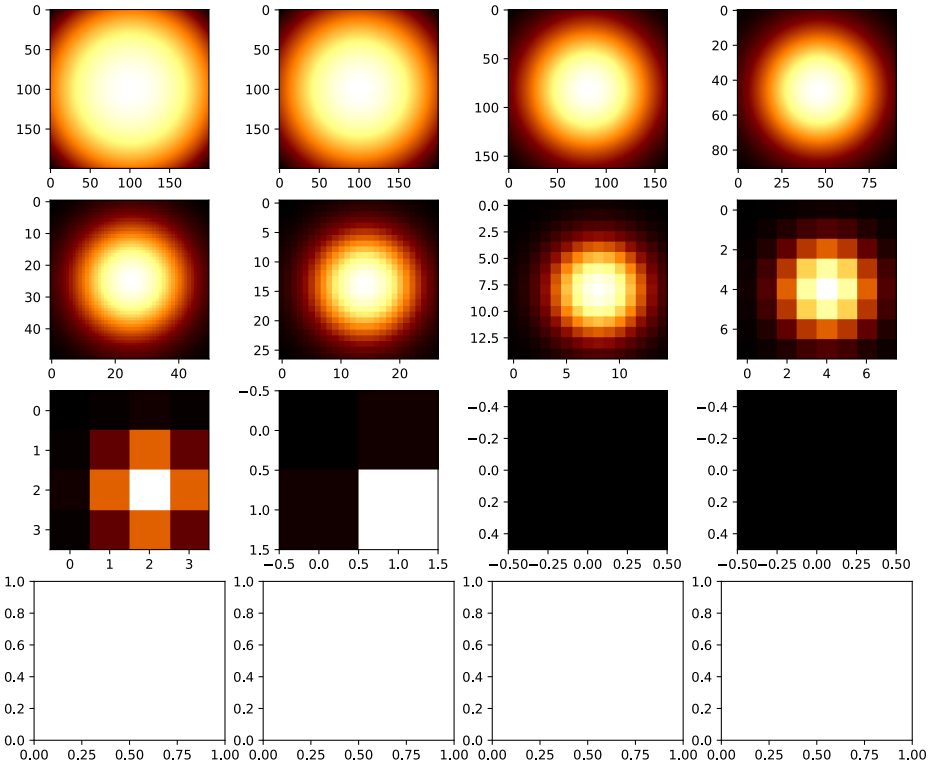


Figure 26. All kernel images produced by a HTG object of 16 levels (from left to right). The four last are not displayed as they are too small to be used for convolutions.

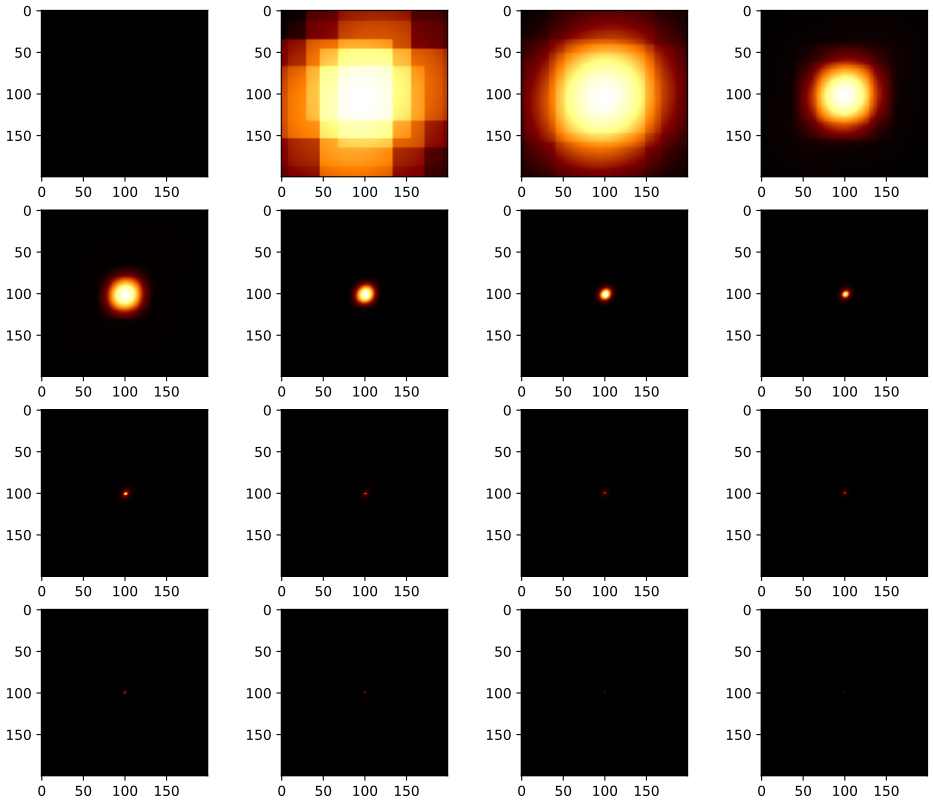


Figure 27. Resulting images of the convolution of the images 26 by 24. The HTG object is built with the output of a RAMSES blast simulation. The splatted field is a scalar density field.

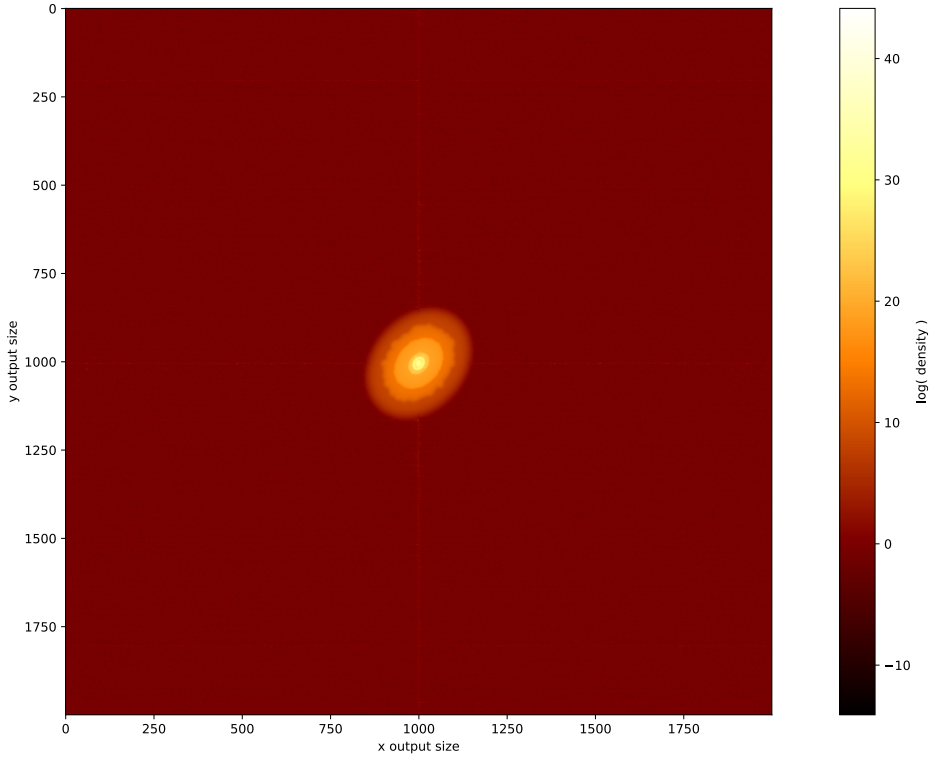


Figure 28. Final splat image of a 3D blast simulation. The logarithm of the density field of the simulation is represented.

for a fluid transition onto the bigger projects, the first of which being a particle mapping utility for Hyper Tree Grids. This tool enables end-users to associate sizeable arrays of particles to cells of a HTG object with reasonable waiting times. In fact, performance is a key word for all the covered topics, in light of the considerably large data-sets produced by modern simulation software; fortunately all the developed tools exhibit promising performance results. Indeed, the 1D profile extraction tool manages to find intersections of a line segment with the cells of a HTG with more than acceptable efficacy, thanks to an efficient ray-box tracing algorithm adapted to our data-structures. The volumetric data splatting tools achieves encouraging performance results as well, and, like the other tools, scales well over increasing tree depth levels, mainly as a result of the optimized FFT algorithm used. Finally, The Hyper Tree Grid sub-region extractor is an essential tool that may be used in conjunction with all the aforementioned post-processing tools, as it enables the analysis of targeted zones of a simulation output, lowering computation and memory costs.

On a more personal note, this internship has been a great opportunity for me to get a glimpse into what it means to work in a research facility. I have enjoyed every moment of the work and have realized that I greatly appreciate data visualization as a whole. During my time on campus, I discovered the Institute of Research of the Fundamental laws of the Universe and its divisions; this gave me a greater understanding of what is currently done in astrophysical simulations. All in all, I believe that this work experience has sharpened my goals for what I wish to do in a time soon to come.

12 *Perspectives*

The tools I have developed are far from finished, as there is only so much that can be done in four months. In addition to the general performance improvements that could be found for all of the utilities, the are a few main problems yet to be handled: the splatting class includes a real-time VTK visualization tool that has yet to be worked out. Indeed, the splatting times are too slow for the moment, and make for a sloppy/choppy real time visualization. In the long term, the tools will have to be ported to C++ for actual performance gains. They may hopefully be integrated into the VTK ecosystem, allowing for a mainstream and widespread adoption of standardized tools to post-process and visualize tree-based AMR data-structures.

References

[1] Marsha J. Berger and Joseph Oliger. *Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations*, volume 53. 1984.

[2] CEA. The cea's homepage. URL https://www.cea.fr/.

[3] CEA DAM. The cea's tera computing machine. URL http://www-hpc.cea.fr/en/complexe/tera.htm.

[4] CEA DEDIP. The edeip's homepage. URL http://irfu.cea.fr/en/Phocea/Vie_des_labos/Ast/ast_service.php?id_unit=521.

[5] Philippe P. Pébaÿ Guénolé Harel, Jacques-Bernard Lekien. Visualization and analysis of large-scale, tree-based, adaptive mesh refinement simulations with arbitrary rectilinear geometry. Technical report, 2 2017.

[6] hdf5. Official hdf5 documentation. URL https://docs.h5py.org/en/stable/.

[7] D.Chapon L.Strafella. Amr grid data structures standardization and compression for post-processing:the ramses use case.

[8] Thomas GUILLET Marc LABADENS, Damien CHAPON. Official pymses website. URL http://irfu.cea.fr/Projets/PYMSES/intro.html.

[9] osiris. Osyris: python visualization utility for ramses. URL https://osyris.readthedocs.io/en/latest/index.html.

[10] Loic Strafella and Damien Chapon. Boosting i/o and visualization for exascale era using hercule: test case on RAMSES. *Journal of Physics: Conference Series*, 1623:012019, sep 2020. URL https://doi.org/10.1088/1742-6596/1623/1/012019.

[11] R. Teyssier. *Cosmological hydrodynamics with adaptive mesh refinement*, volume 385, page 337–364. EDP Sciences, 4 2002. doi: 10.1051/0004-6361:20011817. URL http://dx.doi.org/10.1051/0004-6361:20011817.

[12] Amy Williams, Steve Barrus, R. Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. *J. Graphics Tools*, 10:49–54, 01 2005. doi: 10.1145/1198555.1198748.

[13] Yt. Yt an analysis and visualization toolkit for volumetric data. URL https://yt-project.org/doc/index.html.

Glossary

AMR Adaptive Mesh Refinement. 4–9, 11–15, 21, 25–27, 31, 34

FFT Fast Fourier Transform: Algorithm that computes the discrete Fourier transform.. 34

HTG Hyper Tree Grid. 3, 9–12, 16, 18, 20, 21, 24, 26, 28–30, 32–34

LOD Level Of Detail: The maximal depth of a given AMR tree.. 16, 17, 20

VTK Visualization Tool Kit. 4, 6, 9–11, 20, 21, 28, 34

13 *Appendices*

[CONFIDENTIAL]