Curved Domain Adaptive Mesh Refinement with Hexahedra

Sandro Elsweijer

July 19, 2021

Abstract: In tree-based adaptive mesh refinement (AMR) we store refinement trees in the cells of an unstructured coarse mesh. This lets us combine the speed and simpler management of structured refinement trees with the more flexible mesh generation of the unstructured coarse mesh.

But this creates a conflict between performance and geometrical accuracy. If we favor speed we reduce the cells in our coarse mesh and hence reduce the accuracy of our geometrical representation. If we want more accurate results we generate a finer coarse mesh and lose performance by managing more cells in our unstructured coarse mesh.

To mitigate this conflict we present the prototype of an geometry description which we implement in an already existing library. With this description we build geometry adapted hexahedral refinement trees, which also support high-order curved boundary cells. We also present examples on how to use this description.

Moreover, we test the speedup of this new algorithm compared with coarse meshes with different geometrical errors.

1 Introduction

The geometry of a product is often the key to its function. One of the best examples are airfoils, where the shape influences their characteristics and therefore their use cases [10].

Moreover, the geometry influences more than that like its rigidity, mass, or cost. But for simulating different properties of our product we have to solve partial differential equations (PDEs), which contain spatial derivatives. These can be resolved with the help of spatial discretization of our domain. But by discretizing with polyhedra we lose geometrical accuracy.

The smaller our polyhedra, the smaller our geometrical errors get. However, too small polyhedra result in increased computing times and therefore greater cost. To mitigate this conflict we use adaptive mesh refinement (AMR), which refines or coarsens cells concerning an a priori defined criterion. These refinements happen locally and are therefore able to reduce our overall mesh size. [11, 14]

In this case we build on the previous works of Burstedde and Holke [7, 3, 8], who developed the tree-based AMR library `t8code`. The library provides the user with parallel management of adaptive meshes with different polygons and polyhedra (triangles, quadrilaterals, tetrahedrons, hexahedrons, prisms, and pyramids) and high-level algorithms like adapt, partition, balance, and ghost. A more detailed description of the library and its algorithms can be found in the aforementioned publications.

Tree-based AMR uses already existing meshes and refines their cells, but keeps track of the refinement with so-called refinement trees. These refinement trees are a solution for storing information in a tree-like shape. This can help with the data management which arises in computational simulations. Furthermore, it has the benefit of cutting computing times by significant amounts and still have a good discretization around important regions. [2, 5]

Thus, by using AMR we improve our spatial discretization, but in our case, we do not improve the discretization of our geometry. That can result in computational errors in especially important regions like the boundary layer in a computational fluid dynamics (CFD) simulation.

This problem is inherent to the way most meshes are saved. After the mesh generation, the vertices and their relations are stored, but there is no way of getting to the data of the meshed geometry because the geometry does not get saved along with the mesh.

Consequently, we want to introduce and implement a new geometry description, which is built on an already existing geometry representation capability from `t8code` and tackle the aforementioned problem. Until now this geometry base class has a limited significance for practical use because the geometries have to be programmed manually.

This new geometry implementation can then process geometry-infused mesh files, which we have to develop as well. In these geometry-infused mesh files, the geometry data is linked with the mesh data in such a way, that we can extract the information about which feature of which cell lies on which geometric feature. But this has to be done after the feasibility of the new geometry implementation is verified.

Nonetheless, we want to use this base class to build our own automatic geometry description, which can be used without the need of programming a geometry by hand. Instead, we can then use the geometry-infused mesh files.

We do this with the help of the `Open CASCADE Technology (OCCT)` CAD kernel [15]. We use the CAD Kernel to handle the geometries, which we link to our cells. In Fig. 1 we can see such a mesh consisting of six cells which are positioned to represent a hollow cylinder. Each cell has a linked geometry on the in- and outside.

Take note, that the geometrical description influences each refined cell and therefore improves the accuracy of our geometry with each refinement step. This is the main contribution of this paper. We map the one- or two-dimensional geometries from the cell faces or edges into the three-dimensional space.

This is needed because geometries are often meshed with the help of boundary representations. These boundary representations outline a volume with the help of its boundary (points, curves, and surfaces); therefore, the mesh is created and mapped to the boundary components, not its volume.

In addition, the geometry influences more than just the refinement of the AMR cells. The geometry description can also be accessed by the solver just like in high-order meshes. This way we can for example position our integration points with greater accuracy. These high-order

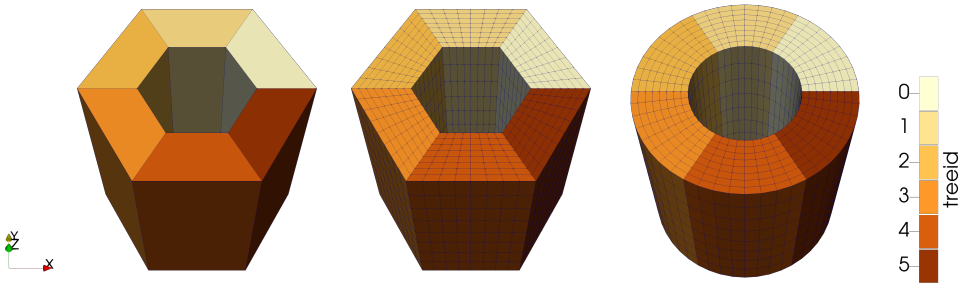


Figure 1: The cmesh of a hollow cylinder (left) consists of 6 refinement trees. When refined three times the resulting mesh has the same shape (middle). But with curved domain refinement the shape approaches the shape of the cylinder (right).

curved domain meshes are addressed in [17, 12, 16, 13, 18, 22, 6]. These publications address this only in the context of static high-order meshes, but not in combination with AMR.

The geometry base class uses a reference volume, which gets mapped to the real cell volume with the help of a geometric description. The reference and real cell volumes can be hexahedrons, tetrahedrons, prisms, and pyramids. In this paper, we will settle for hexahedrons because of their simpler tensor product structure.

In section 2 we outline the fundamental concepts needed to understand this work. Then we will introduce our new geometry description in section 3. After that in section 4 we will take a look at the implementation and some examples. Lastly, we will look at the performance of this description in section 5 and discuss the results in section 6.

2 Fundamental concepts

In this section, we explain the fundamental concepts which lead to our new proposed algorithm.

For the generation of an adaptive mesh, we need an already existing unstructured mesh, whose elements can then be recursively refined. In 3D, `t8code` can convert conforming meshes consisting of tetrahedra, hexahedra, prisms, and pyramids. The input mesh can also contain multiple of these shapes, as long as the connecting faces of two neighbors have the same shape and no hanging nodes occur.

The following concepts are particularly important and inherent to the AMR library `t8code` in which our description will be implemented. These are the concepts of forest-based adaptive meshes with space-filling curves. Forest-based adaptive meshes are constructed from the aforementioned conforming meshes. We will reference these meshes as coarse mesh or short cmesh. To refine this cmesh every cmesh cell gets a recursive refinement rule, which states how the cell should be refined. The refinement levels start at level 0, which is the cmesh cell itself.

Furthermore, we can connect the descendant cells to the ancestor cell to get a tree-like structure with the cmesh cell at its root and the most refined cells as its leaves. We call the tree, its refinement function, and the cmesh cell a refinement tree. The collection of all cmesh trees is known as a forest.

In `t8code` only the leaves of the trees are stored and we can put them into a one-dimensional array, which is described as linear tree storage. This is possible because only the leaves are needed in a numerical simulation. To store the leaves linearly a space filling curve (SFC) index is needed. This index assigns a natural number to each cell so that the combination of a refinement level and an index describes only one cell in the tree. [19, 20, 21]

It is possible to hand in additional information to a refinement tree. This additional information is then stored as an attribute of this tree. We can use this for example for storing the geometrical information about our cmesh cell.

We generate and store the geometrical information itself with the help of the `OCCT` CAD kernel. This way we do not have to worry about all the different possible geometry descriptions and we get a unified way to evaluate the geometries. We will refer to these `OCCT` geometries as curves and surfaces. They should not be confused with the geometries of a hexahedral cell, which we name edges and faces.

The evaluation of a `OCCT` geometry is similar to the geometry description we have to implement. To get a point along a one-dimensional geometry or curve we need one parameter u. u has to be in the range of valid parameters $[u_{\min}, u_{\max}]$ and returns a set of three coordinates along the curve c:

$$\begin{aligned} c : [u_{\min}, u_{\max}] &\to \mathbb{R}^3, \\ u &\mapsto c(u). \end{aligned}$$

In addition we need two parameters u, v to describe every point on a surface s:

$$\begin{aligned} s : [u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}] &\to \mathbb{R}^3, \\ \begin{pmatrix} u \\ v \end{pmatrix} &\mapsto s \begin{pmatrix} u \\ v \end{pmatrix}. \end{aligned}$$

3 Defining a geometrical description

The geometrical evaluation of our hexahedral cells relies on the aforementioned description. This description has to be built on the already existing geometry evaluation base class. This class provides us with all information we need. Among other things we get a set of coordinates from inside a unit cube, which we have to map into our hexahedron $f : [0,1]^3 \to \mathbb{R}^3$.

The currently used description, which maps the provided coordinates into the hexahedron, is a trilinear interpolation between the eight vertices. This method suffices for hexahedra with plane faces and straight edges. But to include hexahedra with non-linear geometries linked to them we need to modify or even replace it.

This also means, that our new description will be more complex and therefore slow down the computation. Nevertheless, we are willing to make this trade-off between performance and accuracy.

Therefore, we show a modified version of the trilinear interpolation, which includes nonlinear geometries. For illustrative purposes, we start with an explanation of a 2D quadrilateral and a bilinear interpolation (Fig. 2).

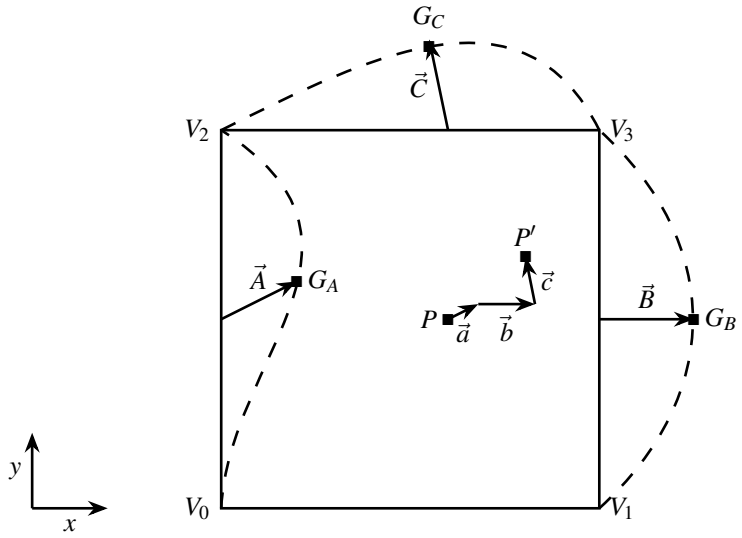


Figure 2: Linear interpolation based geometry description. The deviation from the linear interpolation gets calculated for each edge. After that the vectors get multiplied with their reference coordinate and added to the midpoint calculated by a bilinear interpolation.

Let $V = (X,Y) \in [0,1]^2$ be a point in the reference square. Then we can compute its picture in a quadrilateral with geometries linked to it.

Furthermore, let our quadrilateral have three geometries linked to it, we denote them with G_A, G_B and G_C. The four vertices of the quadrilateral are marked with V_0 to V_3. These vertices also store their parameters on the geometries; we denote them with $U_{A0}, U_{A2}, U_{B1}, U_{B3}, U_{C2}$ and U_{C3}. U_{A0} stands for the parameter of vertex V_0 on geometry G_A.

To picture the surface of the geometrically deformed quadrilateral we firstly have to do a bilinear interpolation between the vertex coordinates V_0 to V_3. We do this based on the reference coordinates X and Y from our point V. This gives us the point P:

$$\begin{aligned} P = &(V_0 \times [1-X] + V_1 \times X) \times (1-Y) \\ &+ (V_2 \times [1-X] + V_3 \times X) \times Y. \end{aligned}$$

In addition, we want to include the displacement generated by the geometries G_A, G_B, G_C. For this, we iterate over each not linear edge and calculate the displacement between the geometries and the linear edges. To get the corresponding coordinates of the geometries (G_A, G_B, G_C) we need the stored parameters of the vertices.

The parameters of the vertices represent their position on the geometry. Thus we can insert them into our geometry and the geometry returns the same coordinates as our vertex has. We can also interpolate between them to get a point between the vertices, but still on the geometry. We do these interpolations again based on the reference coordinates X and Y. In this case we use X to interpolate between U_{C2} and U_{C3} and Y to interpolate between U_{A0} and U_{A2}, as well as between U_{B1} and U_{B3}.

After we insert the interpolated parameters into the geometries we can calculate the displacement vector between the geometry and a linear interpolation between the vertices connected to the geometry. Note, that we have to invert the X and Y reference coordinates according to the vertex positions.

$$\begin{aligned}
\vec{A} &= G_A\left(U_{A0} \times [1-Y] + U_{A2} \times Y\right) - V_0 \times [1-Y] + V_2 \times Y, \\
\vec{B} &= G_B\left(U_{B1} \times [1-Y] + U_{B3} \times Y\right) - V_1 \times [1-Y] + V_3 \times Y, \\
\vec{C} &= G_C\left(U_{C2} \times [1-X] + U_{C3} \times X\right) - V_2 \times [1-X] + V_3 \times X.
\end{aligned} \tag{1}$$

With these vectors and the point P calculated by a bilinear interpolation we can calculate the new point P' with respect to the geometry. This is done by scaling each vector $\vec{A}, \vec{B}, \vec{C}$ from eq. (1) by the orthogonal reference coordinates and adding the scaled vectors $\vec{a}, \vec{b}, \vec{c}$ to our point P:

$$\begin{aligned}
\vec{a} &= \vec{A} \times (1-X), \\
\vec{b} &= \vec{B} \times X, \\
\vec{c} &= \vec{C} \times Y, \\
P' &= P + \vec{a} + \vec{b} + \vec{c}.
\end{aligned}$$

When we extend this description into the three-dimensional hexahedron we have to distinguish between surfaces and curves. Surfaces can be linked either to an edge or a face, curves can only be linked to edges. But for us, it is irrelevant if an edge is linked to a curve or a surface, only the amount of parameters changes.

Nevertheless, it is important if we have a linked edge or a linked face. A face has only one orthogonal reference coordinate, an edge has two. Therefore we have to scale the displacement vectors of a face only with one reference coordinate and a vector of an edge has to be scaled with two coordinates.

An example of such a hexahedron with the curve G_A linked to it is shown in Fig. 3. To convert the displacement vector $\vec{A}$ from eq. (1) to the scaled vector $\vec{a}$ we have to multiply it with both of its orthogonal reference coordinates Y and Z:

$$\vec{a} = \vec{A} \times (1-Y) \times Z.$$

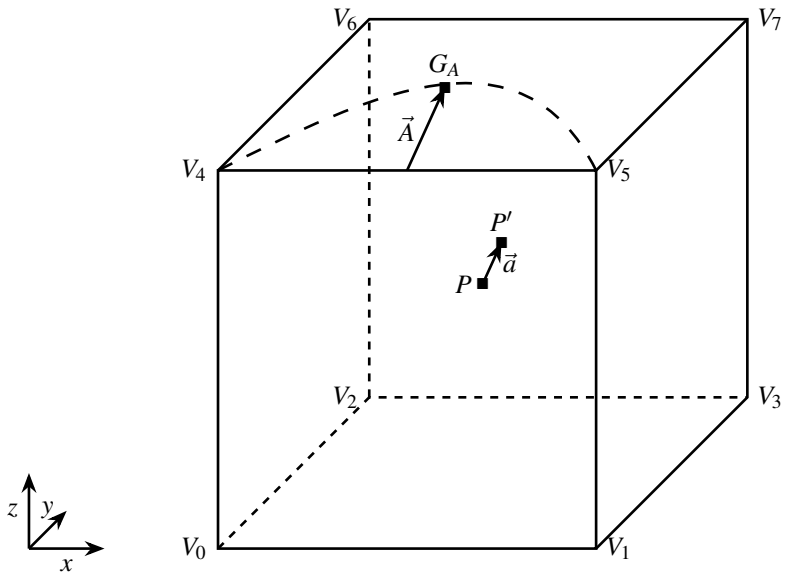


Figure 3: Linear interpolation based geometry description. The deviation from the linear interpolation gets calculated for each edge. After that the vectors get multiplied by their reference coordinate and added to the midpoint calculated by a bilinear interpolation.

4 Implementation

In this section, we look at the already given infrastructure on which the algorithm is built. After that, we take a look at the implementation itself and discuss a few examples of how to use it.

4.1 Implementation of the description

To implement the description we have to take a look at the given infrastructure. We already mentioned that `t8code` needs a so-called cmesh to build on. This cmesh consists of cmesh cells, which are converted to refinement trees. Every refinement tree stores its shape, the coordinates of its vertices, a tree of its refined cells, a SFC and a refinement function.

Moreover, a tree can store additional information in form of attributes. These attributes can be accessed at every time and we will use these attributes to store data like our curve or surface geometries.

A pseudocode of the implemented algorithm is shown in Fig. 5. Here we can see, how the algorithm converts the reference coordinates and the coordinates of the vertices into the output coordinates.

First, we calculate the output coordinates with a trilinear interpolation. This gives us the coordinates as if no geometry is present. After that, we have to calculate the deviation vector for each geometry on our hexahedron. This step is split into two for-loops to distinguish between edges and faces. But they function similarly.

To do this we iterate over each edge and check if it is linked to a geometry via a refinement tree attribute. If an edge is linked to a geometry we save the geometry and the parameter of each vertex on this geometry. This information is also stored as an attribute of the refinement tree.

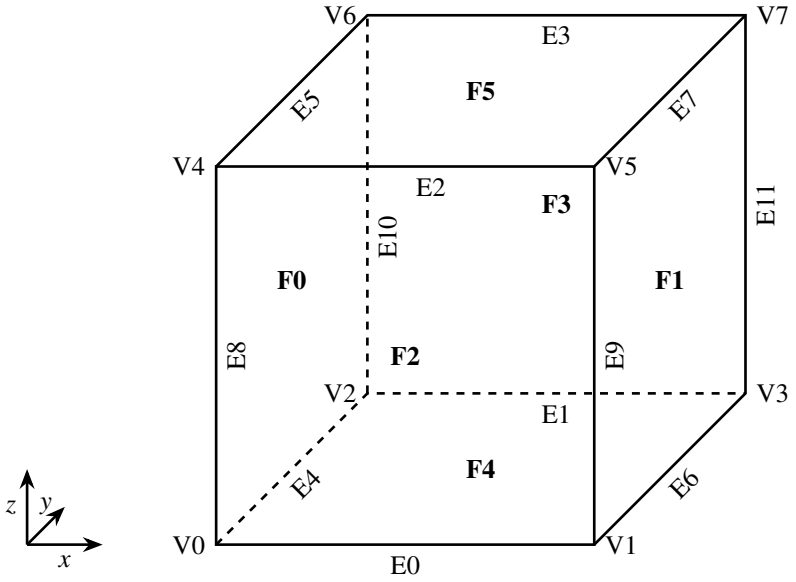


Figure 4: Shown are the indices of a hexahedron. Each set of components (vertices, edges and faces) has its own set of indices.

After that, we can do a linear interpolation between the vertex parameters with the help of the reference coordinates. But for this, we need to know which coordinate (x, y, or z) of the reference coordinates we have to use for this interpolation.

This is why we give each edge of a hexahedron an index i, which represents in which direction this edge points. The indices are shown in Fig. 4. The edges with $i < 4$ point in the x-direction, edges with $4 \leq i < 8$ point in the y-direction and edges with $i \geq 8$ point in the z-direction. If we take a look at edge 9 we know, that we have to use the z-coordinate for the linear interpolation.

With the calculated parameters, we can evaluate our geometry and thus get the first point to generate our displacement vector.

However, we also need the second point, which we get by a linear interpolation between the vertices of our edge. But for this, we need to know, which vertices belong to our edge. This is solved via data sets. These data sets store which vertices are connected to which edges and which faces.

We now have both points and subtract them to get the displacement vector. According to our description, we now have to scale this vector with both of the orthogonal reference coordinates. Because of the indices of our edge we already know which of these are orthogonal. In the case of edge 9 this would be the x- and y-coordinate.

But we also need to know if we have to invert these orthogonal coordinates (e.g. $1-x$). This information is also encoded in our indices from Fig. 4.

We can convert the directions of our coordinate system into directional indices, so let $x \equiv 0$, $y \equiv 1$ and $z \equiv 2$. We can also cycle trough these indices, so that $2+1 \equiv 0$ and $0-1 \equiv 2$.

Then we can distinguish four different scenarios s for an edge, which points in direction d:

0. Both orthogonal coordinates have to be inverted
1. The coordinate with index $d-1$ has to be inverted
2. The coordinate with index $d+1$ has to be inverted
3. No coordinate has to be inverted

These four scenarios are encoded into the index i of the edge. To decode them we just have to calculate

$$s = i \bmod 4 \tag{2}$$

and the result gives us which scenario we have.

Edge 9 for example satisfies $i \geq 8$ and therefore points in the z-direction ($d = 2$). Furthermore we need to multiply the displacement vector of edge 9 with x and $1-y$. This is represented by scenario 1. If we input the edge index into equation (2) we also get 1.

With these rules, we can determine the scaling factors of an edge just by its index. After scaling the vector we only have to add the vector to our total displacement and move on with the next geometry.

After cycling through each edge we have to do the same thing with each face. But here we have to do bilinear interpolations between the parameters and vertex coordinates. However, the scaling of the vector only needs one orthogonal coordinate.

Finally, we just have to add all our aggregated vectors to the outgoing coordinates.

```
input  : Reference coordinates (ref_coords); vertex coordinates (vertices)
output : Output coordinates (out_coords)
out_coords = trilin_inter (ref_coords, vertices);
/* calculate and scale edge displacements                                    */
foreach edge do
    if is_linked (edge) then
        geom = get_geom_attribute (edge);
        parameters = get_parameter_attribute (edge);
        displacement = evaluate (geom, lin_inter (ref_coords, parameters));
        displacement −= lin_inter (ref_coords, get_vertices_of_edge (edge, vertices));
        displacement *= get_orthogonal_coords (edge, ref_coords);
        tot_displacement += displacement
    end
end
/* calculate and scale face displacements                                    */
foreach face do
    if is_linked (face) then
        geom = get_geom_attribute (face);
        parameters = get_parameter_attribute (face);
        displacement = evaluate (geom, bilin_inter (ref_coords, parameters));
        displacement −= bilin_inter (ref_coords, get_vertices_of_face (face, vertices));
        displacement *= get_orthogonal_coords (face, ref_coords);
        tot_displacement += displacement
    end
end
out_coords += tot_displacement;
```

Figure 5: Pseudocode of the modified trilinear interpolation algorithm. The displacements generated by the linked geometries are added to the result of a trilinear interpolation.

4.2 Usage of the implementation

The process of creating a OCCT linked curved domain cmesh manually has some more steps than the process of creating a regular cmesh. These additional steps are *highlighted* in the following list:

1. Setting the vertex coordinates of the cmesh hexahedra
2. Connecting the faces of the hexahedra
3. *Generating the geometries*
4. *Generating the vertex parameters*
5. *Generating a list about which faces are linked to a surface*
6. *Generating a list about which edges are linked to a curve*

7. *Handing the generated information as attributes to the refinement trees*

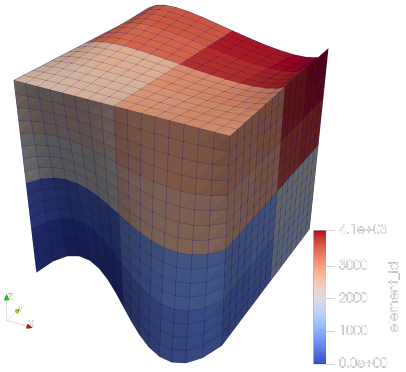
8. Committing the cmesh

Just like every other manually created cmesh, the coordinates of the vertices of the hexahedra have to be specified. After that, we can sew the faces together, so that we have a numerical connection between them. When creating a regular cmesh we would be ready to commit and use the mesh. But when creating a curved domain cmesh we have to do some extra steps.

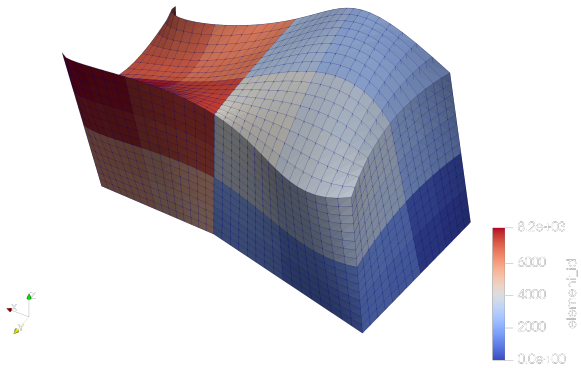
First, we need to generate the geometries which we want to link to our cmesh. All vertices of an edge or a face have to lie on this geometry. We also need to know the parameters these vertices have on the geometry. After this, we can handle the geometries and the parameters as attributes to our cmesh cells.

Additionally, we need lists about which edge or face has a geometry linked to it. These lists can also be saved as an attribute. The last step is to commit the cmesh, which means all the previous steps get executed.

In the following, we will clarify the use of the algorithm with a few examples. These examples should show the general use of one- and two-dimensional geometries to deform one or multiple hexahedrons.



(a) A level 4 refinement tree with splines linked to edge 0 and 3. It is also possible to link surfaces to edges, but these cells do not look different.



(b) Two level 4 refinement trees share the same surface on face 5. One tree is colored bluish, while the other is reddish.

Figure 6: These examples show the usage of the geometry description. It is possible to link multiple refinement trees to one geometry or even link multiple geometries to one refinement tree. The cells are color-coded by their id.

Example 1: B-spline The first example shows how we can link B-splines to the edges of a hexahedron. For this, we generate a hexahedron and give it some arbitrary coordinates. In this case, we use the coordinates of a unit cube.

Then we can generate two B-splines, which link to the edges E0 and E9. In this case, the parameters are 0 for vertex V0 and V6 and 1 for vertex V1 and V7.

The last thing we have to generate is a list that states which edge has which curve and that no face carries any surface.

After handing this information as attributes to our refinement trees we can commit the mesh. The result of refining this cmesh four times can be seen in Fig. 6a.

Example 2: B-spline surface This example shows how to link two hexahedra to the same surface. Just like in the aforementioned example we start by generating the vertex coordinates of the hexahedrons.

In this case, we reuse four vertices, so that our hexahedrons touch each other. After that, we can sew the faces of the hexahedra together.

The geometry used in this example is a surface, which gets linked to face F5 of both hexahedra. This is done by giving one hexahedron parameters between $[0,0]$ and $[0.5,1]$ and the other hexahedron gets parameters between $[0.5,0]$ and $[1,1]$.

Furthermore, we create a list that defines which face gets which geometry. Here both faces get the same geometry.

After giving this information to the refinement trees we commit and refine the cmesh. The result of refining the cmesh four times is visible in Fig. 6b.

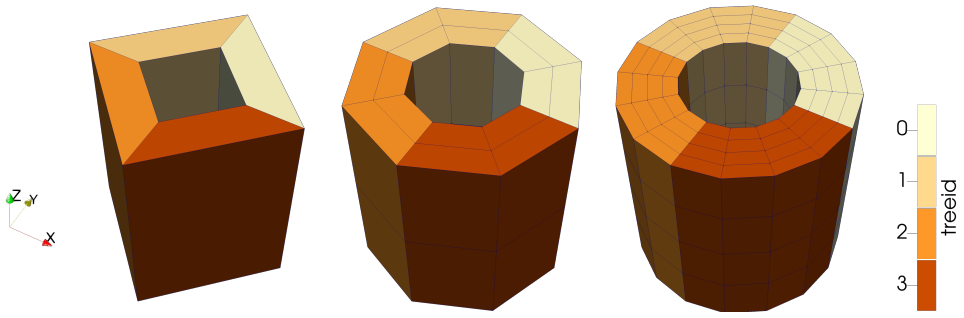


Figure 7: The cylinder example shows how multiple geometries can be linked to multiple refinement trees and therefore improve the geometric accuracy with reach refinement. Shown are the level 0-2 refinements of four refinement trees. Note that the cells of the curved domain geometry description are shown as polygons because ParaView [1] only evaluates the corners of the cells. In reality, the cells are curved to the geometry.

Example 3: Cylinder The last example shows how to link multiple refinement trees to the same two geometries. In addition, we use this example to benchmark the new algorithm in section 5.

This example consists of multiple refinement trees, which resemble a hollow cylinder with an outer diameter of 1, an inner diameter of 0.5, and a height of 1. The cells on the inner and outer cylinder mantles are linked to the same cylinder geometries.

We generate the coordinates of our vertices with a coordinate transformation from cylindrical to Cartesian coordinates. After that, we generate our two cylinder geometries and use the attributes to link them to the cells. The result of refining this cmesh is shown in Fig. 7.

5 Performance

In this section, we analyze the performance of the implemented algorithm. To compare it to the linear algorithm we use the example from section 4.2.

In this example, we have two cylinder geometries, which are linked to an amount of cmesh cells. Moreover, we integrated a feature to specify the level of the cmesh. The level 0 cmesh consists of $4 \times 8^0 = 4$ cells, while the level 1 cmesh consists of $4 \times 8^1 = 32$ cells, the level 2 cmesh of $4 \times 8^2 = 256$ cells and so on.

However, it is important to notice, that only the outward-facing cells have a geometry linked to them. So in practice, the 4 cells of the level 0 cmesh have two geometries linked to them, the 32 cells of level 1 have only one geometry linked to them and at level 2 we have 128 cells with a geometry linked to them and 128 more without any geometry. But these cells get processed by the same algorithm.

In comparison to that, we can create the same cylinder with the same amount of cmesh cells, but with the linear geometry description. With these two coarse meshes, we can compare the geometry algorithms.

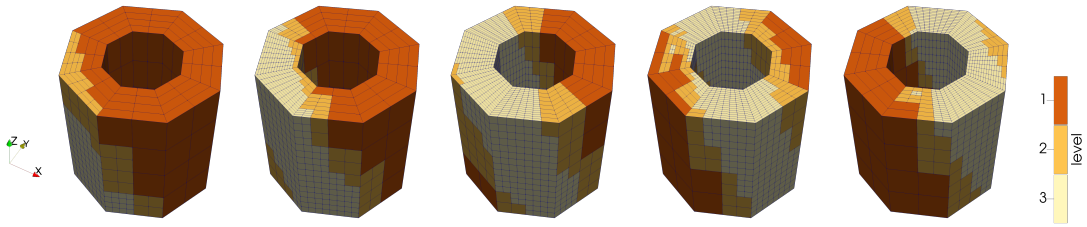
For this, we are going to advance a plane through the cylinders and all cells in certain proximity get refined by 3 levels. The distance gets measured from the center of the cell. The cmeshes have different levels c; to get a consistent amount of cells we will also refine the cmesh by l levels so that $c + l = 6$. The plane advances in 5 time steps. An example of this is shown in Fig. 8

These coarse mesh levels will lead to deviation from the original geometry when using the linear geometry description. In the case of the level 0 coarse mesh, the maximum distance between a vertex and the geometry will be 1.46×10^{-1}. The maximum distance for level 3 is 2.41×10^{-3} and for level 6 is 3.76×10^{-5}. However, these values are dependent on the cmesh level as well as the geometry, so these values can not be transferred to other geometries.

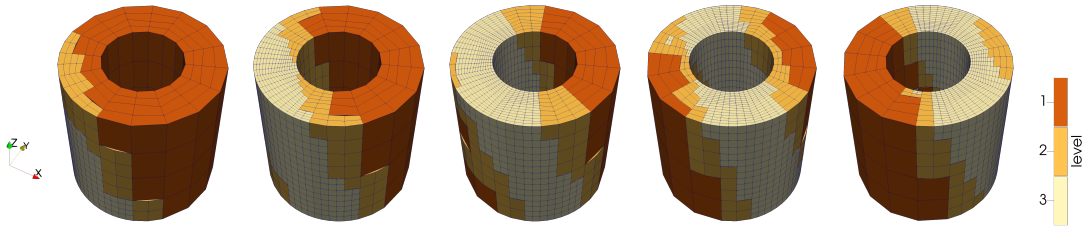
After that, we will add the time spent on the algorithms partition, adapt, balance and ghost. The partition algorithm uses the SFC to partition the cells for each process. Adapt refines the cells using the refinement function of the tree. The balance function refines cells in multiple iterations so that the level difference between neighboring cells is not larger than 1. The ghost function searches for the neighboring cells across refinement trees. These functions are covered in more detail in [7].

We will also measure how much time the geometry calculations need. This way we can show the influence of the geometry calculations on the overall runtime.

Note that we test the AMR algorithms excusively. A real-world analysis with a PDE solver is important and relevant, but beyond of the scope of this paper. However, it will be part of future works.



(a) Linear geometry description



(b) New curved domain geometry description

Figure 8: A plane is moved through the cylinder cmeshes from section 4.2. One cmesh uses the linear geometry description, while the other uses the new curved domain geometry description. The cmeshes are of level $c = 1$. Each refinement tree is refined once ($l = 1$). Cells near the plane get refined twice and balanced afterwards. Note that the cells of the curved domain geometry description are shown as polygons because ParaView only evaluates the corners of the cells. In reality, the cells are curved to the geometry.

The tests were conducted on a Intel Xeon W-2175 Processor with 14 cores at 2.5 GHz and with 128 GB of RAM. We used commit `180c678499e951432f7835594fc581ebbc9dc077` [9] on a `t8code` fork. The command line inputs for the $c = 3$ runs are:

```
mpirun -np 14 ./examples/timings/t8_time_forest_partition \
-O -l3 -r3 -C3 --xmin -0.7 --xmax -0.3 -b -T 1 -D 0.2 -o -g

mpirun -np 14 ./examples/timings/t8_time_forest_partition \
-L -l3 -r3 -C3 --xmin -0.7 --xmax -0.3 -b -T 1 -D 0.2 -o -g
```

The results are shown in Fig. 9.

As expected the runtime of the geometry evaluation increases with the more complex description. In the case of cmesh level 0, it increases by a factor of nearly 6.6. But when the level of the cmesh is increased the mean runtime of an evaluation of the curved domain description decreases, because there are more cells with no geometry linked to them. In the case of cmesh level 3, only

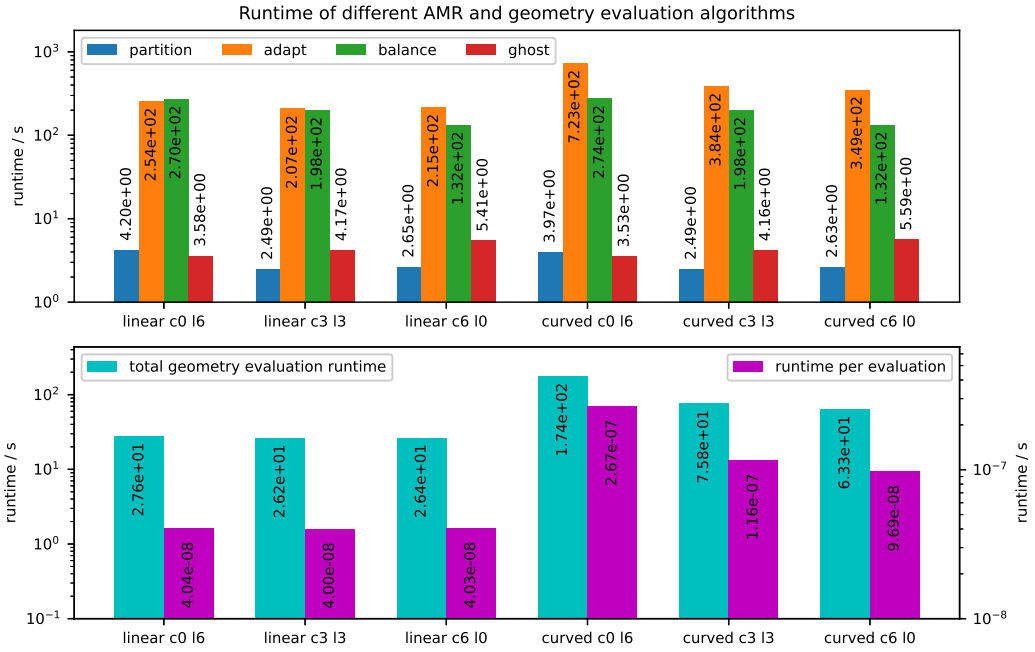


Figure 9: Runtime of different AMR algorithms. Shown are the mean runtime per process and timestep accumulated over each timestep (top). The runtime was measured with variations in the cmesh level c and geometry description. We also measured the total mean time spent on evaluating the geometry descriptions (bottom) and the mean runtime per evaluation.

one fourth of the cmesh cells have a geometry linked to it, with level 6 this is further reduced to $1/32$. This results in an increase in runtime in comparison to the linear interpolation by a factor of 2.9 for cmesh level 3 and a factor of 2.4.

However, the curved domain geometry description time still does not approach the linear geometry description time. This is because we have to check for each cell if a geometry is present. But this can be mitigated with further code optimization.

Besides that, the adapt algorithm is the only one, which uses the geometry description. Consequently, only the adapt algorithms runtime is affected by the new geometry description. In the case of the level 0 cmesh, the runtime nearly triples, but this approaches an increase of only 50% with higher cmesh levels. Note that this is only the case because our adaptation is based on the location of the cell. If the refinement function would use another, not geometry-linked refinement criterion, the geometry would not be evaluated. But we chose this criterion to get and observe the influence.

Furthermore, the remaining algorithms only in- or decrease their runtime with the amount of cmesh cells. the balance algorithm speeds up with a greater amount of cmesh cells, while the ghost algorithm slows down. But the overall runtime is dependent on much more factors, like the level the cells are adaptively refined to or the geometry of the mesh itself. More on the runtime of these algorithms can be read in [7, 3, 4].

6 Conclusion

In the previous sections, we looked at how we can describe a geometry-deformed hexahedron and implemented this into the AMR library `t8code`. We also looked at some examples of how to use this implemented description. Lastly, we looked at the performance of the newly implemented algorithm and compared it to the previously used linear geometry description.

In this section, we are going to evaluate the previous sections and take a look at the following research.

As already mentioned there are multiple publications about curved domain, high order meshes [17, 12, 16, 13, 18, 22, 6], but none of them combine this approach with tree-based AMR.

Therefore, this paper moves one step ahead and applies it to a new region. But we have to evaluate if this suffices for calling this method useful. To do that we have to look foremost at the performance.

As we have seen in section 5, the runtime of the curved geometry description can peak as high as 6.6 times the runtime of the linear geometry description. But this is only the case if every cell has two geometries (the inner and outer cylinder) linked to it. In typical use cases, we use meshes with way fewer geometry-linked cells, because only the boundary cells are geometry-linked.

Besides that, the runtime of the curved geometry evaluation can be improved with code optimization. Then the runtime of a cell with no geometry evaluated with the curved description is nearly the runtime of the same cell evaluated with the linear description. The only difference is, that the curved description has to check if a geometry is present.

Furthermore, the percental increase in the runtime of the curved geometry description does not translate exactly in the runtime increase of our high-level algorithms. It only affects the adapt runtime, while the algorithms partition, balance, and ghost are not affected.

On the other hand, we do not only get a geometry-based refinement of the boundary cells, but we also get a more accurate high-order mesh. We improve from a linear geometry approximation to a geometry approximation with the same polynomial degree as the high-order mesh. Yet we can not say, what this means for example for the boundary layer in a CFD simulation, but theoretically, this results in a gain in accuracy.

Therefore, we still got some subsequent work to do. Firstly we want to test this algorithm with a numerical application and look for accuracy improvements. Moreover, we need to adapt this geometry description for the other, by `t8code` supported, element types (triangles, tetrahedrons, prisms, and pyramids) and implement the coupling to a mesh generator, to obtain the relevant additional geometry information.

In addition, the results of this paper could also be the first step towards the implementation of improved inflation or prism layer control in `t8code`. With the information about which cells carry a geometry and where the geometry lies, we get more possibilities in controlling the refinement of inflation layers themselves.

In summary, we want to advance this approach to ease its use and to give it a broad spectrum of applicability. We strive to deepen the interaction between geometries and their numerical simulation, always in pursuit of greater accuracy.

References

[1] Lisa Avila and Utkarsh Ayachit, eds. *The ParaView guide: Updated for ParaView version 4.3*. Full color version. Los Alamos: Kitware, 2015. ISBN: 978-1-930934-30-6.

[2] Marsha J. Berger and Joseph Oliger. "Adaptive mesh refinement for hyperbolic partial differential equations". In: *Journal of Computational Physics* 53.3 (1984), pp. 484–512. ISSN: 0021-9991. DOI: 10.1016/0021-9991(84)90073-1.

[3] Carsten Burstedde and Johannes Holke. "A tetrahedral space-filling curve for nonconforming adaptive meshes". In: *SIAM Journal on Scientific Computing* 38 (2016), pp. C471–C503. DOI: 10.1137/15M1040049.

[4] Carsten Burstedde and Johannes Holke. "Coarse Mesh Partitioning for Tree-Based AMR". In: *SIAM Journal on Scientific Computing* Vol. 39 (2017), pp. C364–C392. DOI: 10.1137/16M1103518.

[5] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. "p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees". In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: 10.1137/100791634.

[6] S. Dey, R. M. O'Bara, and M. S. Shephard. "Towards curvilinear meshing in 3D: the case of quadratic simplices". In: *Computer-Aided Design* 33.3 (2001), pp. 199–209. ISSN: 0010-4485. DOI: 10.1016/S0010-4485(00)00120-2.

[7] Johannes Holke. "Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types". PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2018. URL: https://d-nb.info/1173789790/34.

[8] Johannes Holke and Carsten Burstedde. *t8code: Parallel AMR on hybrid non-conforming meshes*. 2015. URL: https://github.com/holke/t8code (visited on 07/19/2021).

[9] Johannes Holke, Carsten Burstedde, and Sandro Elsweijer. *t8code: Parallel AMR on hybrid non-conforming meshes*. 2021. URL: https://github.com/sandro-elsweijer/t8code/commit/180c678499e951432f7835594fc581ebbc9dc077 (visited on 07/19/2021).

[10] Eastman N Jacobs, Kenneth E Ward, and Robert M Pinkerton. *The characteristics of 78 related airfoil sections from tests in the variable-density wind tunnel*. NACA Technical Report No. 460. 1933. URL: https://ntrs.nasa.gov/citations/19930091108 (visited on 07/19/2021).

[11] Richard I. Klein. "Star formation with 3-D adaptive mesh refinement: the collapse and fragmentation of molecular clouds". In: *Journal of Computational and Applied Mathematics* 109.1 (1999), pp. 123–152. ISSN: 0377-0427. DOI: 10.1016/S0377-0427(99)00156-9.

[12] Xiao-Juan Luo et al. "Automatic p-version mesh generation for curved domains". In: *Engineering with Computers* 20.3 (Sept. 1, 2004), pp. 273–285. ISSN: 1435-5663. DOI: 10.1007/s00366-004-0295-1.

[13] Gianmarco Mengaldo et al. "Industry-Relevant Implicit Large-Eddy Simulation of a High-Performance Road Car via Spectral/hp Element Methods". In: *CoRR* abs/2009.10178 (2020). URL: https://arxiv.org/abs/2009.10178.

[14] Andreas Müller et al. “Comparison between adaptive and uniform discontinuous Galerkin simulations in dry 2D bubble experiments”. In: *Journal of Computational Physics* 235 (2013), pp. 371–393. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2012.10.038.

[15] *Open CASCADE Technology*. URL: https://www.opencascade.com/open-cascade-technology/ (visited on 09/07/2021).

[16] O. Sahni et al. “Curved boundary layer meshing for adaptive viscous flow simulations”. In: *Finite Elements in Analysis and Design* 46.1 (2010), pp. 132–139. ISSN: 0168-874X. DOI: 10.1016/j.finel.2009.06.016.

[17] Mark S. Shephard et al. “Adaptive mesh generation for curved domains”. In: *Applied Numerical Mathematics* 52.2 (2005), pp. 251–271. ISSN: 0168-9274. DOI: 10.1016/j.apnum.2004.08.040.

[18] S. J. Sherwin and J. Peiró. “Mesh generation in curvilinear domains using high-order elements”. In: *International Journal for Numerical Methods in Engineering* 53.1 (2002), pp. 207–223. DOI: 10.1002/nme.397.

[19] Hari Sundar, Rahul S. Sampath, and George Biros. “Bottom-Up Construction and 2:1 Balance Refinement of Linear Octrees in Parallel”. In: *SIAM Journal on Scientific Computing* 30.5 (2008), pp. 2675–2708. DOI: 10.1137/070681727.

[20] Hari Sundar et al. “Low-constant parallel algorithms for finite element simulations using linear octrees”. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC ’07*. Ed. by Becky Verastegui. event-place: New York, New York, USA. ACM Press, 2007, p. 1. ISBN: 978-1-59593-764-3. DOI: 10.1145/1362622.1362656.

[21] Tobias Weinzierl. “The Peano Software-Parallel, Automaton-based, Dynamically Adaptive Grid Traversals”. In: *ACM Transactions on Mathematical Software* 45.2 (2019), pp. 1–41. ISSN: 0098-3500. DOI: 10.1145/3319797.

[22] Zhong Q. Xie et al. “The generation of arbitrary order curved meshes for 3D finite element analysis”. In: *Computational Mechanics* 51.3 (Mar. 1, 2013), pp. 361–374. ISSN: 1432-0924. DOI: 10.1007/s00466-012-0736-4.