This code is the result of a project supervised by Dennis Ogiermann and Maximilian Köhler and carried out by Esther Koch and Leonid Ryvkin.

1 Goals of the project

The goal of the project was to implement a subtype of the abstract type `JuAFEM.AbstractGrid` in a way enabling certain grid modifications (refinements and derefinements of cells). We worked against the following specification.

Design and constraints:

- A grid is a collection of cells (Elements) of a certain dimension d in an n-dimensional euclidean space.
- The cells can be convex quadrangles ($d = 2$, $n = 2$ or $n = 3$) or convex hexahedra (cube-like elements, $d = n = 3$).
- Cells are attached to each other along their faces.

Basic functionality:

- In addition to all functionality of the JuAFEM grid (recovering vertices, edges, faces, cells, giving certain collections of these entities labels, etc.) it should be possible to refine/derefine cells.
- Refining a cell means subdividing it into several subcells (dividing a cell into 4 subcells for $d = 2$, or 8 subcells for $d = 3$). The refinements always happen along the centers of opposing sides, cf. Figure 1.

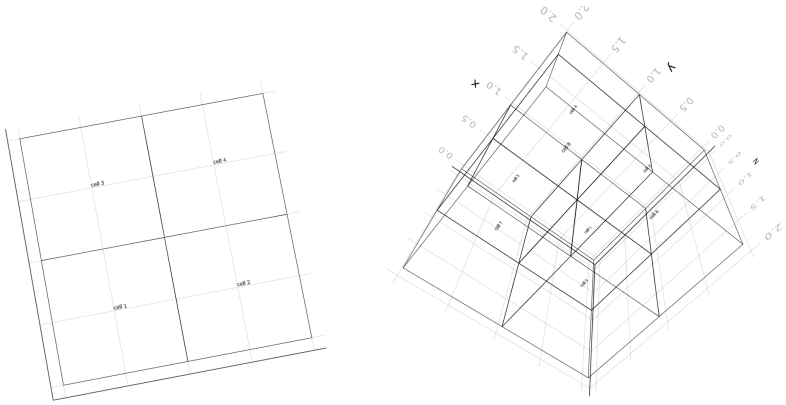

Figure 1: isotropic split of 2D resp. 3D cell

- Derefining simply reverts the refining operation.

- The biggest difficulty, if only one of two neighbouring cells is refined, then suddenly a big face is attached to several small ones. Some nodes turn up at the interior of cells and faces (i.e. are not on the corners). This information has to be recoverable from the grid in a way that JuAFEM can work with.

Stretch goals:

- Anisotropic splits: Splitting a cell into subcells might also be anisotropic, i.e. the split is carried out only along some axes. In $d = 2$ an anisotropic split yields 2 subcells of a cell and for $d = 3$ we get 2 or 4 subcells), cf. Figure 2.

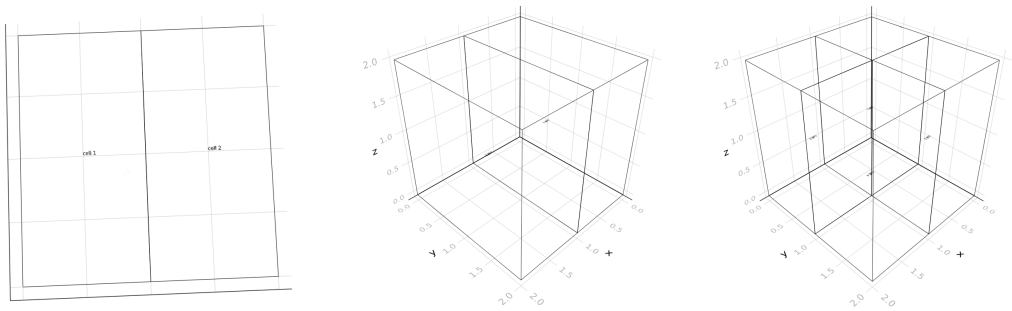

Figure 2: The ways to anisotropically split 2D resp. 3D cell

- Non-manifold grids: In basic grids only two cells can be joined by a face. However, sometimes it is useful (especially for $d = 2$) to allow for multi-cell joints, see e.g. Figure 3

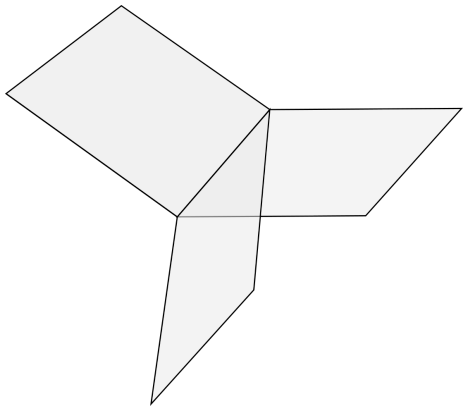

Figure 3: A non-manifold joint of three 2D-cells at one boundary in three dimensions.

- Grids with varying cell dimension.[1]

[1] Our realization does not exclude building such grids. However, the current refinement and derefinement functionality might not work properly close to the places where different dimensions meet.

2 Technical documentation

2.1 Introduction and Code structure

The module FEMRefinements.jl provides a realization of the `JuAFEM.AbstractGrid` type. The grid we construct is build from linear quadrilateral/ hexahedral cells. Adjacent cells are connected through their boundaries. Cells, their boundaries, their boundaries boundaries etc. are all realized as instances of an `Element` struct, relizing the `JuAFEM.AbstractCell` type. In addition to basic `JuAFEM.grid` functionalities, our grid class `AdaptiveGrid` admits the refinement of cells (substituting a cell with a collection of smaller cells in a particular way). For the refinement procedure to be reversible, the old cells are not erased, but kept in the data structure. Hence, `Elements` store a lot of information: Relations between cells and subcells (boundaries and sub-boundaries), attachements to neighbouring cells, node locations. The core of our module are the structs `Element` and `AdaptiveGrid` together with the functions `refine!` and `derefine!`. The code is structured as follows:

- **src/FEMRefinements.jl:** This file mainly contains the imports, exports and and includes the actual code files.
- **src/elements.jl:** Contains the `Element` struct and its functions. In addition to reimplementing JuAFEM `Cell` methods (`Element` implements the abstract type `JuAFEM.AbstractCell`), also some methods for tree traversal are provided. Furthermore, some methods are provided to find out which nodes are on/in elements (mostly only working for boundary elements).
- **src/refinements.jl:** This file contains the methods for refining and derefining cells. As these operations are local, no access to the grid is needed. The main subtleties are managing the masters of the faces and avoiding to create the same node twice (when a cell close to a subdivided cell is split, we want to reuse the same nodes for all new cells where possible).[2]
- **src/grid.jl:** Here the core struct `AdaptiveGrid` subtyping the `JuAFEM.AbstractGrid` type is implemented, along with its methods. Due to refinements and derefinements, nodes, faces and cells have no persistent integer number, however JuAFEM needs numbered lists as outputs, so most of the functions internally translate between the external linear numbering and the internal tree structure. The only parameter really necessary for the interior processing of the grid is `rootcells`, the other ones `cells, faces, nodes` are only stored, because they are accessed to often to be generated anew every time. The generation is carried out using the `update_leafmesh!` function.
- **src/utils.jl:** This file, written by M.K. and D.O. contains some functions necessary to use the grid with JuAFEM.
- **src/grid_generators.jl:** Create a standard quadrilateral resp. hexahedral grid from two resp. 3 vectors.
- **src/visualization.jl:** A few grid and solution plotting methods, mostly for debugging. Written by M.K.

[2] For instance in the first step of the splitting procedure in Figure 4 five new nodes appear, and in the last step only 4, because the node on the center of the left hand side of the right cell already exists (it is the center of the righthand side of the left cell).

- **test/runtests.jl** Tests for the `AdaptiveGrid` are provided. Contains an adaptation of most of the gridtests from JuAFEM and many additional tests for the refinement and derefinement interface.
- **Examples/HeatEquation.jl** Adaptation of the HeatEquation example of JuAFEM with our AdaptiveGrid class, written by M.K. and D.O.
- **Examples/HeatEquationMixed.jl** Further adaptation of the HeatEquation example of JuAFEM using a MixedDoFHandler, written by M.K. and D.O.
- **Examples/helmholtz.jl** Adaptation of the Helmholtz example of JuAFEM to our grid class, written by M.K. and D.O.

2.2 Design choices

- **Cells:** A cell, realized by the struct `Element` is given by a list of its `nodes` (corners, points in n-dimensional space)[3]. The order of the corners plays an important role and is fixed by the JuAFEM convention.
- **Boundaries:** The cells are not connected directly, but through their boundaries. The boundaries are also instances of `Element`. Hence, any `Element` has parameters `boundaryof` and `boundaries`. For cells, the `boundaryof` parameter is set to `nothing`, for a boundary it points to the cell it is boundary of. For an element of dimension d, `boundaries` is an array of elements of dimension $d-1$, sorted by the JuAFEM convention. For 0-dimensional elements (given by one node) `boundaries` is left empty.
- **Tree structure:** For the bookkeeping of the refinements, a tree structure is chosen. When a cell is subdivided it stays in place and just gets the subcells as children. The tree-like structure relating the big cell and its subcells is saved in the `parent`, `children` parameters of `Element`. When a cell c is split, the boundaries of the (appropriate) subcells s_i are also linked to the boundaries of c through their parent and children parameters. However, boundaries of boundaries are not connected via parent-children anymore for simplicity reasons.
- **Attachments of boundaries:** The following approach is often utilized for adaptive grids[4]: The boundaries of cells are called *halffaces*. A face is made up of 1 halfface on one side (the master) and a set of halffaces spanning the same region on the other side.[5] To register, whether a halfface is the master `Element` has a `ismaster` boolean parameter, which is superfluous for non-cell-boundary elements. In order to allow for non-manifold structures, i.e. faces joining more than two sides, our faces contain one master and possibly several sets of slaves. Hence we will avoid the misleading notion of halffaces in the sequel. For an example of how master and slave halffaces behave, see Figure 4 below. As in our implementation of the grid, cells are not erased when they are split (cf. Tree structure), instead of pointing at a set of "small boundaries", we can point at their common ancestor. This is realized by the `attachedto`

[3] In general there might be more nodes than corners (e.g. for quadratic instead of linear interpolation), however currently the package only supports linear elements

[4] Zhao, Xinglin, et al. "Conformal and non-conformal adaptive mesh refinement with hierarchical array-based half-facet data structures." Procedia Engineering 124 (2015): 304-316.

[5] This method does not allow two 2-dimensional boundaries in 3-dimensional space, of which one is split horizontally and the other vertically to be attached to each other.

parameter of `Element`. It is a list of elements, which we only use for boundaries of cells (i.e. not for cells themselves and not for boundaries of boundaries of cells etc.). When a cell boundary is also a boundary of the grid, its `attachedto` is empty, when it is a slave, `attachedto` points only to its master and when it is a master `attachedto` points to all of its slaves. The attachedto interface is only kept up to date for leaves of the tree, i.e. the boundaries that are not further subdivided, for a simpler code and better performance.[6]

- **The actual grid** The core of the `AdaptiveGrid` struct is given by an array of its `rootcells`. All other data can be recovered through them by vertical traversal (parent to child) and horizontal traversal (cell to boundary to attachedto to boundaryof etc).

- **Refinement algorithm** The core of the package are the `refine!` (and derefine) functions. These functions call `split!` on a series of cells, and then execute `update_leafmesh`. The `split!` method first creates the necessary new nodes, for splitting a given cell c. Then it checks the neighbourhood of the cell for whether some of these nodes already exist and removes the unnecessary new ones. Then the (up to 8) subcells s_i of c are created and attached to one another. The last and most subtle step is gluing the s_i into the grid, along the boundaries of c. This is delegated to `attach_children!`. `attach_children!` has three options: either the boundary of c is a slave (very simple case), or it can be made a slave (using `make_master!` on another boundary it is attached to, simple case) or the master has to be split, in which case the slaves have to be distributed correctly among the subboundaries of the master.

- **Derefinement algorithm** works analogously calls `derefine!` on an individual cell c, where the only subtlety is gluing the boundary of c back into the grid, which is realized by the `unattach_children` function.

[6] Like in a tree, where only the few outer layers are actually alive, and the inside can has only supporting functions.

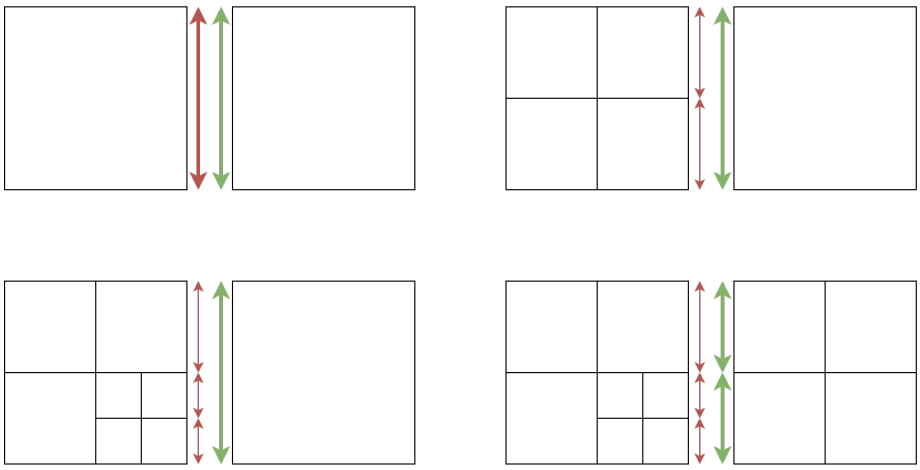

Figure 4: The development of master-slave relationships for two neighbouring cells, masters are green and slaves are red. In the first picture there is one face (one master attached to one slave). Then the left cell is split, and we have a master attached to two slaves (still one face). After a further split, the master has three slaves (still one face). Finally the right cell, and hence the master splits, now we have two masters, one with one slave and one with two (Now we have 2 faces). Note that in the last image the top master and slave could be interchanged.

2.3 Costs of the functions

In the runtime expectations described below, we assume n to be the number of original cells and d be the maximal depth or refinement. We further assume that the number of non-manifold-joints (i.e. the number of places where more than two boundaries are joined) is not significant. Especially, the space required by a grid can go up to $\mathcal{O}(8^d \cdot n)$. We omit methods with constant runtime. Most methods in the element.jl and refinements.jl files operate locally and don't depend on n directly:[7]

- **Methods with depth-linear runtime** $\mathcal{O}(d)$ Mostly due to (simple) tree traversal
  - `ultimate_ancestor`
  - `findlowestancestorwithnodes`
  - `descent_until_split`
  - `getdepth`
- **Methods with depth-exponential runtime** $\mathcal{O}(8^d)$ (3D case), mostly due to tree traversal with branching (i.e. doing something for all children)
  - `collect_children`
  - `getcontainednodes`, `getinteriornodes`, `getneighbourhood`, `getnodeneighborhood`
  - `have_identical_leaves`

[7]The only exception is `eltocell`, which needs to go through the whole $\mathcal{O}(8^d \cdot n)$ long nodelist, to return the indices of the corners of the element

- `make_master!` (as a big boundary can be connected to many small ones)
- `attach_children!`, `unattach_children!` and hence `derefine!`.
- `split!` and `isotropic_split!`, mostly due to checking, whether the necessary new nodes are already in the grid. More specifically getting the `nodeneighbourhood` and applying `substitute_nodes!` are the costly part.

On the other hand, methods in grid.jl, typically have to find indices of elements in a list (node, cell, face etc.) and thus have runtimes of at least $\mathcal{O}(n \cdot 8^d)$ i.e. linear in the total number of nodes (or cells). Especially, the method `update_leafmesh!` and hence `refine!` and `derefine!` have this runtime.

2.4 Basic use

```
using FEMRefinements #our module
using JuAFEM #appropriate version necessary, enabling use of external grids
import Makie # just for plotting

#define 3 vectors
v1 = Vec((2.0, 0.0, 0.0))
v2 = Vec((0.0, 2.0, 0.0))
v3 = Vec((0.0, 0.0, 2.0))

#generate a grid
grid = generate_grid3D(2, 2, 2, v1, v2, v3);

#refine all cells isotropically
FEMRefinements.refine!(grid, collect(1:getncells(grid)))

#refine one cell anisotropically along the first axis
FEMRefinements.refine!(grid, [1], [[1]])

#get nonconformal information (i.e. information about hanging nodes)
ncinfo = getncinfo(grid)

# plot the grid
Makie.mesh(grid)
```

The output of the above plot can be seen in Figure 5.

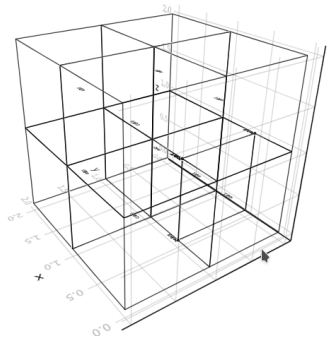

Figure 5: Plot of the 3D-grid generated and refined above

2.5 State of the code, future work

- Tested to some extend:
  - quadrilateral 2-dimensional grids in 2 and 3 dimensions.
  - hexahedral 3-dimensional grids.
  - refinement (isotropically or anisotropically) and derefinement.
  - output in a JuAFEM compatible way.
  - information on hanging nodes, i.e. nodes that are not on the corners of master-boundaries (= faces).
- Implemented, untested:
  - The possibility to have grids with more complex topology (tori, rings, spheres etc).
  - Non-manifold grids (with more than two sides touching).
- partially implemented/unimplemented:
  - 1-dimensional grids (easy to add).
  - Grids of mixed dimension (Here some change in JuAFEM would be necessary, splitting near the dimension jumps could be difficult).
  - Higher dimensions (Currently not possible due to limitations of Tensors.jl. Implementation changes could be necessary for finding node-neighbourhoods of elements and checking whether a node is in an element, where the linear algebra is currently not dimension-agnostic).
- Possible problems:
  - Currently the way to check whether a certain node is already in the grid is by checking coordinates, this is not precise and might lead to problems at very refined (small) cells. Similarly, the checks of whether a node is in a cell are numerical calculations and not algebraic and thus could also break due to machine precision problems.

  - To improve performance, currently when an new node candidate is added to the grid, not all existing nodes are checked, when trying to find whether it already exists, but only a node-neighborhood of the cell. The generation of this neighbourhood could be unreliable. [8]
  - The management of which boundaries are masters and how they are attached to each other is quite fiddly.Problems in reassigning the master can easily occur, especially when having non-manifold meshes in 3 dimensions.
  - We do not follow the JuAFEM convention for ordering edges of a 3-dimensional cell, because for us they are just boundaries of boundaries. [9]

- Potential improvements:
  - More dimension-agnostic approach, especially for the refinement parts.
  - Algebraic instead of numeric way of finding nodes nearby.
  - performance improvements e.g. by not having edges (and 0-dimensional elements!) as separate entities. Use of static data structures instead of Arrays. Different data structures for different types of element to avoid unnecessary field.
  - A grid construction toolkit, (where one can, for example, glue grids to one another along elements) could be quite useful.
  - Instead of throwing an error when `refine!`is called on a set of cells where some cells can not be refined, the refinement procedure should just throw a warning and refine wherever possible.
  - Most methods in grid.jl have a very bad runtime, as the indices of elements and nodes have to be retrieved from very long lists. This could be improved by appropriately saving the indices directly in the cells or buffering them.

[8] This is rather a 3D than a 2D problem and could be mitigated by also attaching edges etc. to each other and not only faces. However that could harm performance and needs some coding work.

[9] If needed, this can be easily changed.