Implementation of Discontinuous Galerkin Finite Element Method on Polygonal Meshes

Bachelor Project Report supervised by Prof. Dr. Ralf Hiptmair

Tarzis Maurer, ETH Zürich, 17-934-274

July, 2023

Abstract

Discontinuous Galerkin Finite Element Methods are universal, do not lack in stability which is featured in classical FEMs and can be implemented on general polytopic meshes. An infrastructure for the problem solving of a linear degenerate second-order boundary value problem on such general poygonal meshes in a discontinuous finite element setting is added to the FEM library *LehrFEM++*.

1. Introduction

It is known that Classical Finite Element Methods (FEMs) lack sufficient stability when applied to hyperbolic or ”nearly” hyperbolic problems. Oscillations in the approximated solution can show up, mostly around regions where the gradient of the analytical solution is large. Discontinuous Galerkin Finite Element Methods (DGFEMs) allow for general non-self-adjoint PDEs to be solved without those stabilization issues. Additionally, polytopic meshes can be used (opposed to strictly triangular, rectangular or hybrid meshes) because there are no continuity constraints between neighbouring cells. This report describes a Bachelor Project which extends the C++ open-source Finite Element Method library *LehrFEM++* [1], used for teaching of the course *Numerical Methods for Partial Differential Equations* [2] at ETH Zürich. The two main additions are made up of an environment for the handling of general polygonal meshes within the program and functionalities to solve the general linear degenerate second order convection-diffusion-reaction boundary value problem

$$-\operatorname{div}(\mathbf{A}(\mathbf{x})\cdot\operatorname{grad}u)+\operatorname{div}(\mathbf{b}(\mathbf{x})u)+c(\mathbf{x})u=f\ \ \textit{in}\ \ \Omega\subset\mathbb{R}^2, \tag{1}$$

$$u=g\ \textit{on}\ \Gamma_D\cup\Gamma_-\quad \textit{and}\quad \mathbf{A}(\mathbf{x})\cdot\operatorname{grad}u\cdot\mathbf{n}(\mathbf{x})\ \ \textit{on}\ \ \Gamma_N. \tag{2}$$

Where $\mathbf{A}:\Omega\to\mathbb{R}^{2,2}$ is a positive semi-definite bounded matrix field, $\mathbf{b}$ a continuous vector field on Ω and c : $\Omega\to\mathbb{R}$ a bounded function. $\Omega\in\mathbb{R}^2$ is an open domain in two dimensions. Γ_D, Γ_- and Γ_N are disjoint sections of the boundary $\partial\Omega$ of the domain.

All theoretical aspects of this project closely follow [3]. The error analysis etc. is presented there and will not be repeated here. The book lies the mathematical foundation for the weak formulation of the problem 1 & 2 and especially the symmetric interior penalty (SIP)(also see [7]). The correctness of the implementation can be proven by the method of manufactured solutions.

A number of symbols need to be introduced for their further use. A subdivision of the domain Ω into disjoint open (and polygonal) elements κ is denoted by $\mathscr{T}_h$. $V^{\mathbf{p}}(\mathscr{T}_h)$ is the discrete finite element space defined over the domain. $\mathscr{F}_h$ denotes the set of faces of codimension 1 associated with $\mathscr{T}_h$. This set is further subdivided into interior faces $\mathscr{F}_h^{\mathscr{I}}$ and faces on the boundary $\mathscr{F}_h^{\mathscr{B}}$.

As stated above, this project extends *LehrFEM++*, which already incorporated many functionalities. Meshes with triangular, quadrilateral or hybrid partitions were already incorporated in the library and classical FEM with nodal basis functions in the discrete space had been implemented.

Contents

2. Overview of changes in LehrFEM++

Numerous additions in *LehrFEM++* emerge from this project. They are not merged into the main branch of the library at this point. To have a visual overview of what is newly implemented, see figure 1. Most additions are in the newly created namespace *lf::dgfe* as one goal of the project is to minimize interfering with existing infrastructure. The additions in the other namespaces *lf::mesh*, *lf::assemble*, *lf::base* and *lf::io* feature tests with Google's C++ test framework and are working correctly. Functionalities regarding the solving of 1 & 2 in a DGFEM setting are collected in the namespace *lf::dgfe*. Smaller parts of this module, e.g. the class `lf::dgfe::BoundingBox` are also tested. More complex utilities are shown to be correct by way of numerical experiments, whose results can be found in section 6. The current state of the project can be found in the github repository `https://github.com/tarzm/lehrfempp/tree/dgfe`.

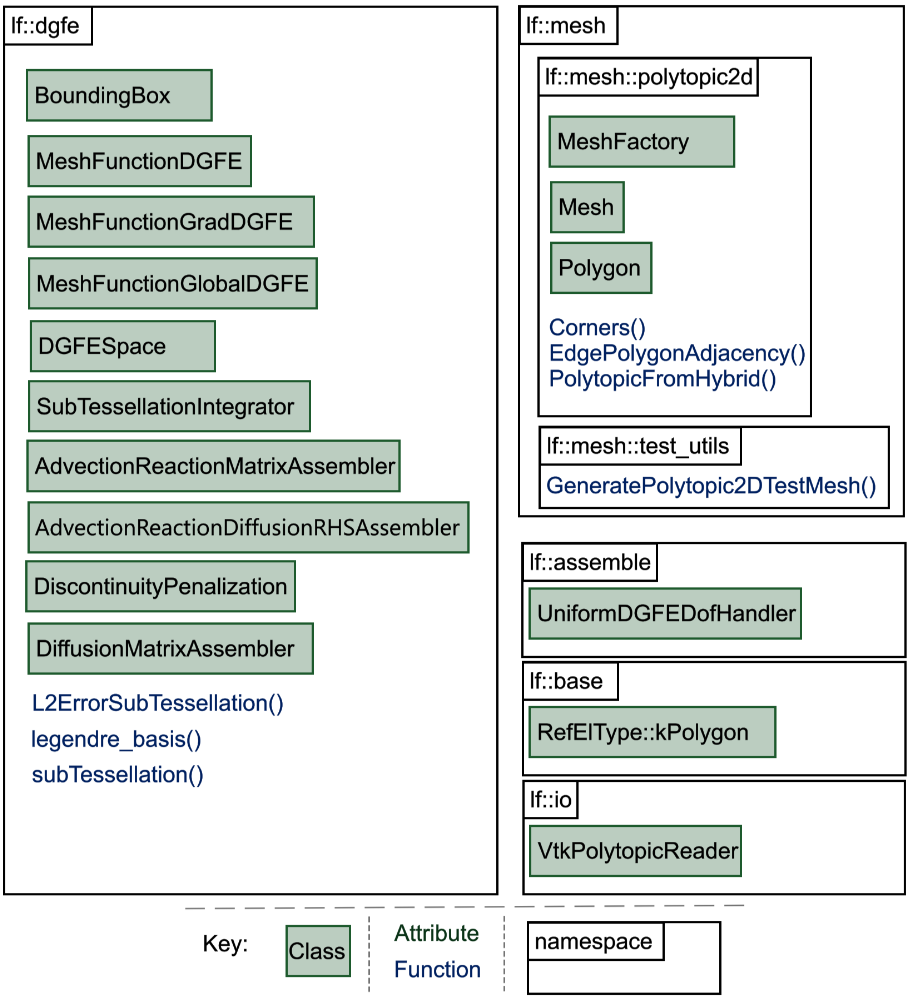


Figure 1: Overview of LEHRFEM++ changes from this project. The visualization is not complete but a base to discuss upon.

3. Mesh

3.1. Theory

The application of discontinuous Galerkin finite element methods generally allows for choosing general piecewise polynomial trial and test spaces and has - more important for this section - no restrictions on the choice of subdivision of the computational domain. That is the reason why polygonal or polytopic meshes have been introduced and added to the *LehrFEM++* library to allow for more general meshes. As the error analysis in [3, sec. 5.2] requires shape regularity (defined and discussed also in [3]) of the polytopic cells in the mesh, this property is implied for all meshes used in the polytopic setting. It can be assumed that the mesh generator used for the project provides shape-regular meshes. More on the generator in the next section.

The variational formulation of the boundary value problem 1 & 2 includes the integration of jump terms over all edges in the mesh (see equation 24, further information in section 5). To evaluate jump terms over internal edges, a reference from each edge to the two adjacent polygonal cells is needed, otherwise an iteration over possibly all polygons would be required for each evaluation. This functionality is provided and discussed in section 3.3.

3.2. Mesh Generation

Polytopic meshes used in the project have been either made with *PolyMesher* [6], were programmed by hand inside the library or were originally an existing hybrid mesh from *LehrFEM++* and then interpreted as a polytopic mesh.

PolyMesher creates meshes with linear convex polygons. They are Voronoi Tessellations of the desired domain. All functionalities to generate meshes with *PolyMesher* can be found in the *LehrFEM++* library in `lf/mesh/polytopic2d/polymesher`. The matlab file `lf/mesh/polytopic2d/polymesher/mesh.m` provides the code used to generate meshes with 2^2 up to 2^{12} cells (a graphical representation of 4 of these are depicted in figure 2). It writes `.txt`-files with information about the nodes and the elements of the mesh. Another file `lf/mesh/polytopic2d/polymesher/vtk_writer.py` `lf/io/plot_mesh.py`, written in Python then reads the information and packs it into a `.vtk`-file. Finally this file can be read by the new *LehrFEM++* class `lf::io::VtkPolytopicReader` which then provides the mesh in the internal *LehrFEM++* representation.

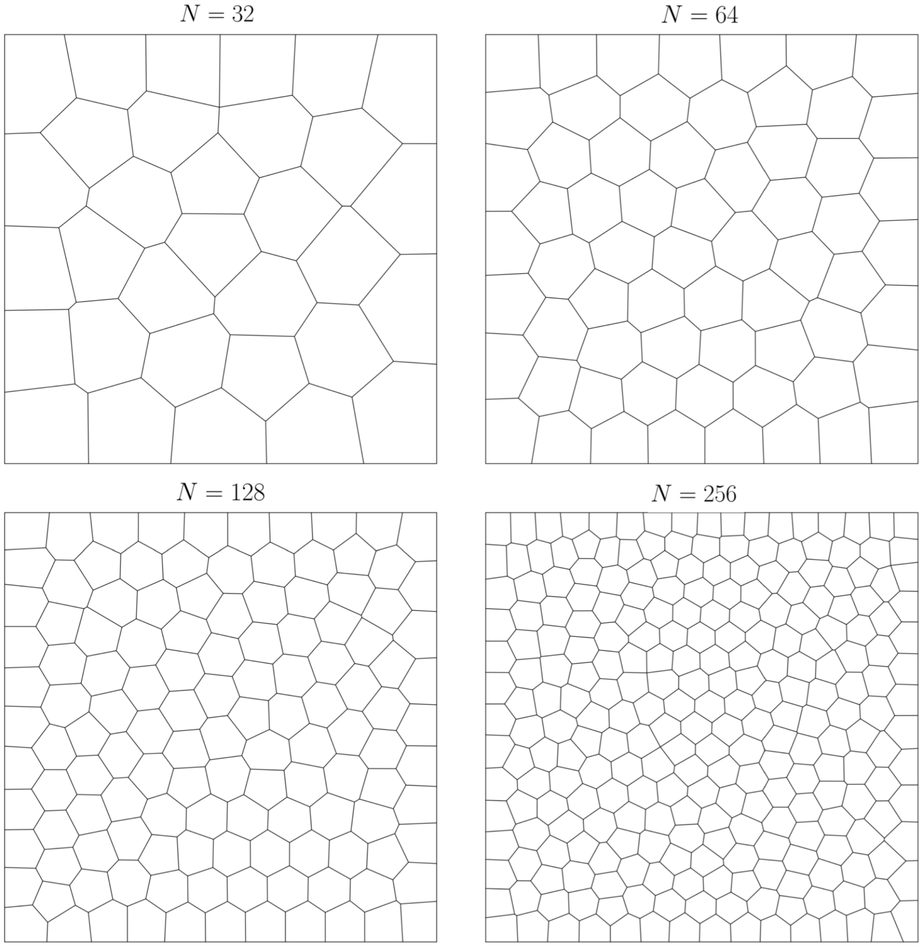


Figure 2: Visualizations of Voronoi Tessellations generated by *PolyMesher*. N describes the number of polygonal cells present in the mesh. All meshes are subdivisions of the unit square domain $(0, 1)^2$. They are created via *lf::io::writeMatplotlib()* and `lf/io/plot_mesh.py`.

3.3. Implementation

An overview of the additions to the namespace *lf::mesh* is depicted in figure 3. A new namespace *lf::mesh::polytopic2d* holds the majority of new functionalities regarding polygonal meshes.

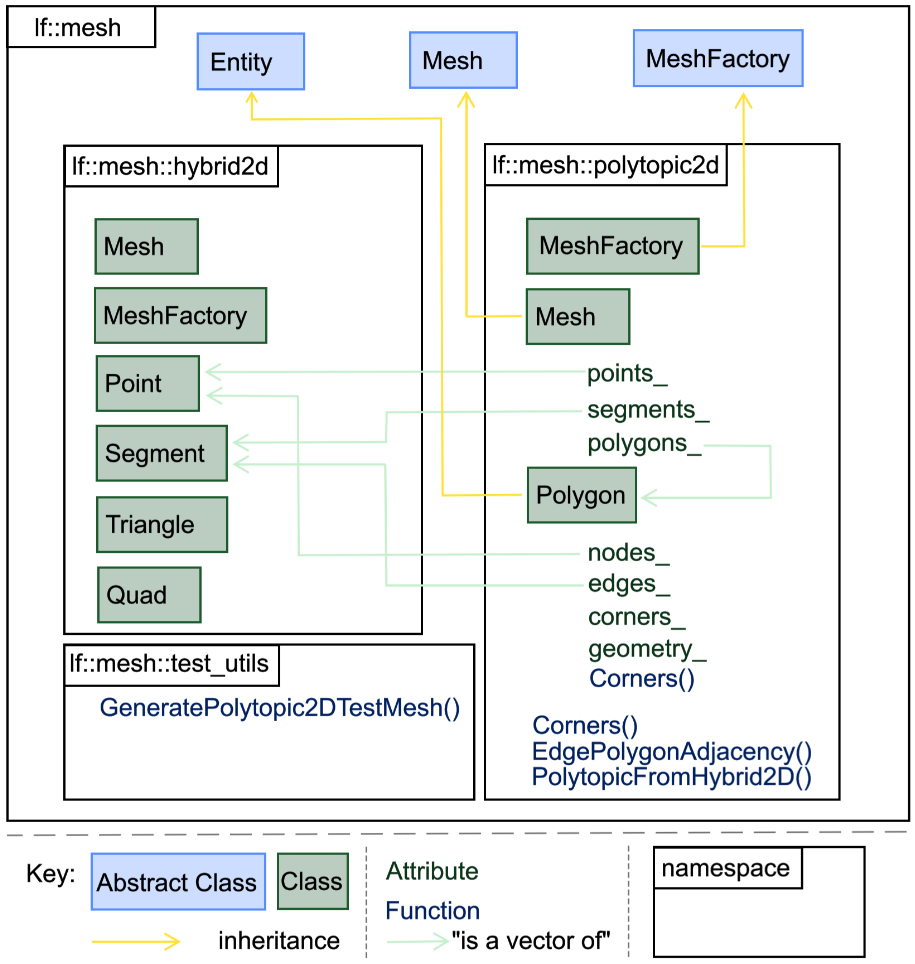


Figure 3: Visualization of the namespace *lf::mesh*. It is not complete but rather an overview of what is discussed in this section. A key to the symbols can be found at the bottom of the figure.

Polygon

A new class `Polygon` describes all cells (codimension 0) in the 2D mesh. As depicted in figure 3, it inherits from the abstract base class `lf::mesh::Entity`. Its functionalities are therefore very similar to the ones of a `lf::mesh::hybrid2d::Triangle` and `lf::mesh::hybrid2d::Quad`. To keep things simple and use existing components of *LehrFEM++*, the edges and nodes of `Polygon` are objects of types from the namespace *lf::mesh::hybrid2d*. There are a number of differences to the entities of codimension 0 in the hybrid case:

- All polygons have the newly created dummy reference element `lf::base::RefElType::kPolygon`.

- Polygons have no fixed number of nodes and edges. The attributes *nodes_* and

```cpp
Eigen::MatrixXd Corners(const lf::mesh::Entity* ent){
    LF_VERIFY_MSG(ent->RefEl() == lf::base::RefEl::kPolygon(),
      "This method is only implemented for Polygons");
    return dynamic_cast<const lf::mesh::polytopic2d::Polygon*>(ent)->Corners();
}
```

Listing 1: Definition of the function *lf::mesh::polytopic2d::Corners()*.

*edges_* are of type vector with variable length instead of an array with fixed length.

- The attribute *geometry_* is a nullpointer. Due to polygons not having a common reference element, there is also no affine mapping between a general polygon and all instances in the mesh. Therefore the `lf::geometry::Geometry` object (More on this in [2, sec. 2.7.2.3]) has no relevance for polygons. How basis functions are created on polygons is discussed in section 4.
- There is a new attribute *corners_* which stores the coordinates of the nodes of the polygon in a 2xn matrix, where n is the number of nodes in the polygon. As discussed above, polygons have no `lf::geometry::Geometry` object which stores its topological information in one place. Rather than having to iterate over its nodes every time coordinates are needed, *Corners()* provides that functionality. In most parts of the program, a polygon is interpreted as the base class `lf::mesh::Entity`, so directly calling *Corners()* from it will cause an error. That is the reason for the existence of the function *Corners()* which takes an `lf::mesh::Entity` as an argument. It will then perform a dynamic cast to make the *Corner()* member function work. The definition of this function is showed in listing 1.

Apart from those differences, the `Polygon` class works like its hybrid equivalents.

Mesh

The class `Mesh` in the polytopic setting provides the same functionalities as in the hybrid setting. All entities of codimension 0 are of type `lf:.mesh::polytopic2d::Polygon`. As has been mentioned above, a functionality which provides a reference from all `lf::mesh::Segments` to their adjacent polygons in the mesh is needed. This is implemented in *EdgePolygonAdjacency()*. It takes a pointer to a mesh as an argument and returns a data set containing pointers to both polygons for each segment and the local indices of the segment in the polygons. If the segment is on the boundary, i.e. has only one adjacent polygon, the pointer to the second polygon is a null pointer. The declaration of the function can be seen in listing 2.
A function which is very important, but nowhere depicted here is *lf::mesh::utils::flagEntitiesOnBoundary()*. Polytopic meshes and polygons are implemented such that the function works exactly the same as in the hybrid setting.

```
using PolygonPair = std::pair<std::pair<const lf::mesh::Entity*, size_type>,
            std::pair<const lf::mesh::Entity*, size_type>>;
/**
 * @brief Constructs A CodimMeshDataSet that contains the adjacencies
 *        of the Segements. Each Segment is adjacent to either two Polygons
 *        (inner Segment) or one Polygon (boundary Segment). Returns a pair
 *        of pairs. First object of the inner pair is
 *        a pointer to the polygon, the second is the local idx of
 *        the edge in that polygon.
 *
 * @param mesh_ptr The mesh used.
 * @return lf::mesh::utils::CodimMeshDataSet<PolygonPair> The constructed
 * CodimMeshDataSet
 */
lf::mesh::utils::CodimMeshDataSet<PolygonPair> EdgePolygonAdjacency(
            std::shared_ptr<const lf::mesh::Mesh> mesh_ptr);
```

Listing 2: Declaration of the function *lf::mesh::polytopic2d::EdgePolygonAdjacency()*.

```
explicit MeshFactory(dim_t dim_world, bool check_completeness = true,
        bool unit_square = true): dim_world_(dim_world),
        check_completeness_(check_completeness), unit_square_(unit_square) {}
```

Listing 3: Declaration of the constructor of `lf::mesh::polytopic2d::MeshFactory()`.

MeshFactory

Like in the hybrid setting, `lf::mesh::polytopic2d::MeshFactory` inherits from the abstract base class `lf::mesh::MeshFactory` and thus incorporates almost identical components. Two functionalities are added.

As described above, most meshes used in the project are generated by *PolyMesher* and are subdivisions of the unit square. *PolyMesher* collapses small edges into one node (in the middle of the collapsed edge) in its iterative mesh generation process. This results in nodes in the mesh that should be exactly on the boundary of the unit square, but due to the edge collapsing they are not. To counter this problem, a new member *unit_square* was added to the `MeshFactory`. It can be passed as an optional construction argument, as shown in listing 5. If set to true, all added coordinates closer to the boundary than the macro *COORD_TOLERANCE* (currently set to 10^{-7}) are ”cleaned” and set to exactly the boundary coordinate.

In order to be able to compare the hybrid and polytopic meshes numerically, the function *polytopicFromHybrid()* can be used. Every hybrid mesh can be interpreted as a general polytopic one and this is what the function does. Its declaration is shown in listing 4.

Additionally, a polytopic equivalent to *lf::mesh::test_utils::GenerateHybrid2DTestMesh()* is implemented as *lf::mesh::test_utils::GeneratePolytopic2DTestMesh()*. Two poly-

```
/**
 * @brief returns a polytopic 2D mesh from a hybrid 2D mesh
 */
std::shared_ptr<lf::mesh::Mesh> PolytopicFromHybrid2D(
        std::shared_ptr<const lf::mesh::Mesh> mesh_ptr);
```

Listing 4: Declaration of the function `PolytopicFromHybrid2D()`.

topic test meshes are available. One is depicted in figure 4. The second one is a subdivision of the unit square by four equal-sized squares.

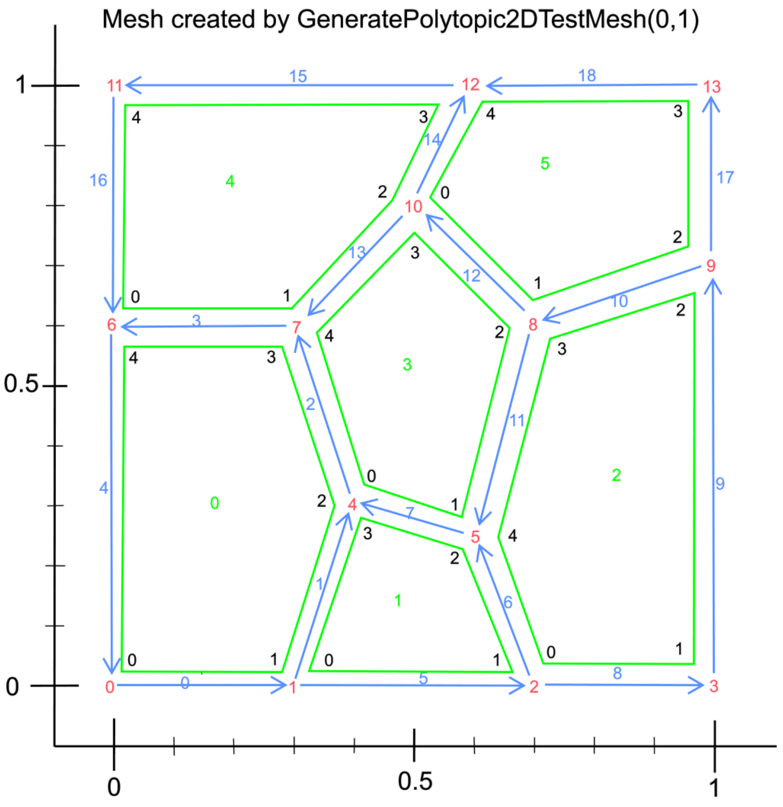


Figure 4: Visualization of the mesh retrieved with *lf::mesh::test_utils::GeneratePolytopic2DTestMesh(0,1)*.

IO Functionalities

The io-part of the polytopic mesh module is not visualized in figure 3. Mainly a new class `lf::io::VtkPolytopicReader` is implemented to read polytopic mesh data from VTK-files and provide the polytopic mesh. A sample usage is shown in listing 5.

In order to be able to plot polytopic meshes like their hybrid counterparts, *lf::io::writeMatplotlib()* is slightly adjusted to accept entities of codimension 0 with the

```
std::filesystem::path here = __FILE__;
auto mesh_file_name = "msh_files/unit_square_voronoi_64_cells.vtk";
auto mesh_file = here.parent_path().string() + mesh_file_name;
lf::io::VtkPolytopicReader reader
        (std::make_unique<lf::mesh::polytopic2d::MeshFactory>(2), mesh_file);
auto mesh = reader.mesh();
//example loop over polygonal cells
for (auto polygon : mesh->Entities(0)) { ... }
```

Listing 5: Example usage of `lf::io::VtkPolytopicReader`.

reference element `lf::base::RefElType::kPolygon`. The output of this function (which is a .csv file) can then be plotted via `lf/io/plot_mesh.py`.

4. Discrete Finite Element Space

4.1. Theory

As stated above, the general theoretical decisions of the project follow the book [3]. Basis functions of the discrete finite element $V^{\mathbf{p}}(\mathscr{T}_h)$ space in a discontinuous finite element setting are not subject to any continuity constraints between cells (thus the term ”discontinuous”). A rather simple construction of basis functions is achieved by mapping a polynomial space defined on a reference bounding box κ_R to the axis-aligned bounding box B_κ of each specific polygonal cell κ in the mesh. This space is then restricted to κ. The axis-aligned bounding box is the minimal cartesian axis-aligned rectangle such that all vertices of the cell either are inside the rectangle, or on its boundary, i.e. $\overline{\kappa} \subseteq \overline{B}_\kappa$. The reference bounding box is defined as $\kappa_R \coloneqq (-1, 1)^2$. Via an affine mapping $\mathbf{F}_\kappa$, the bounding box κ_R is mapped to a polygon:

$$\mathbf{F}_\kappa(\hat{\mathbf{x}}) = \mathbf{x} = J_\kappa \hat{\mathbf{x}} + \mathbf{c}. \tag{3}$$

With $J_\kappa \coloneqq diag(h_1, h_2)$, $\mathbf{c} \coloneqq (m_1, m_2)^\intercal$, $\hat{\mathbf{x}}$ a general point in κ_R and $\mathbf{x}$ its image in B_κ. Additionally $h_i, i = 1, 2$ is half the length of the i-th side of B_κ and m_i is the midpoint of the i-th side of B_κ. See figure 5 for a sketch of this mapping.

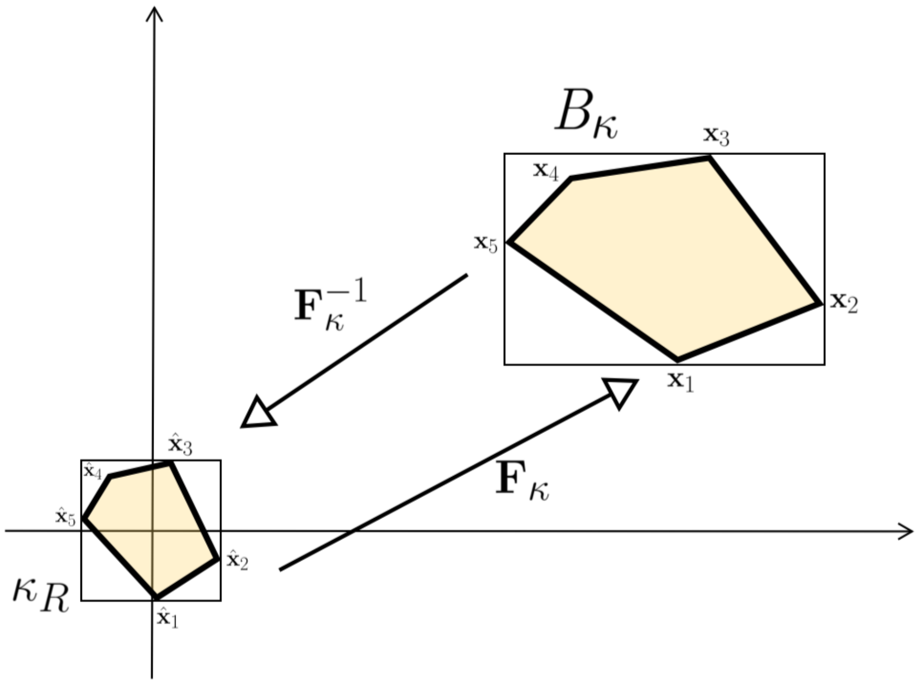


Figure 5: Mapping between the reference bounding box κ_R and the bounding box B_κ of a cell κ in the mesh.

For polynomial basis functions, tensor product Legendre Polynomials in two dimensions are used. $\left\{\hat{L}_i(\hat{x})\right\}_{i=0}^{2}$ describes the family of $L^2(-1,1)$-orthonormal one-dimensional Legendre polynomials (their respective degree being i), in this project namely the ones in table 1. There are two options in the program: 1D-legendre polynomials of maximum degree 1 or maximum degree 2.

i	$\hat{L}_i(\hat{x})$
0	1
1	$\hat{x}$
2	$\frac{1}{2}(3\hat{x}^2 - 1)$

Table 1: One-dimensional legendre polynomials with polynomial degree i

The basis functions $\hat{\Phi}_i(\hat{\mathbf{x}})$ on the reference bounding box are then defined as

$$\hat{\Phi}_i(\hat{\mathbf{x}}) = \hat{L}_{i_x}(\hat{x})\hat{L}_{i_y}(\hat{y}) \tag{4}$$

With i_x and i_y being the polynomial degrees of the 1D-legendre polynomials in the direction of the first and second axis of the cartesian coordinate system. The plots and explicit numbering of $\hat{\Phi}_i(\hat{\mathbf{x}})$ is depicted in figure 6 in case of one-dimensional legendre polynomials of maximum degree two. There is also an option for the usage of one-dimensional legendre polynomials of maximum degree one. More of this choice in section 4.2.

The polynomial basis functions of a general polygon κ in the mesh are then given by mapping $\hat{\mathit{\Phi}}_i(\hat{\mathbf{x}})$ via $\mathbf{F}_\kappa$ to B_κ and restricting its support to κ.

$$\mathit{\Phi}_{i,\kappa}(\mathbf{x}) = \hat{\mathit{\Phi}}_i(\mathbf{F}_\kappa^{-1}(\mathbf{x})) \quad \forall \mathbf{x} \in \kappa \subset B_\kappa \quad \forall \kappa \in \mathscr{T}_h \tag{5}$$

Finally, the discrete space $V^{\mathbf{p}}(\mathscr{T}_h)$ is spanned by all basis functions $\mathit{\Phi}_{i,\kappa}$ on all polygonal cells.

$$V^{\mathbf{p}}(\mathscr{T}_h) = \operatorname{span}\{\mathit{\Phi}_{i,\kappa}\} \tag{6}$$

4.2. Implementation

As mentioned before, most of the new components in *LehrFEM++* are integrated in the new namespace *lf::dgfe*. It contains both basic functionalities of a discrete Galerkin finite element space as well as the algorithms used to assemble the Galerkin matrix and right-hand side vector of equation 24. A visual overview of *lf::dgfe* is depicted in figure 7.

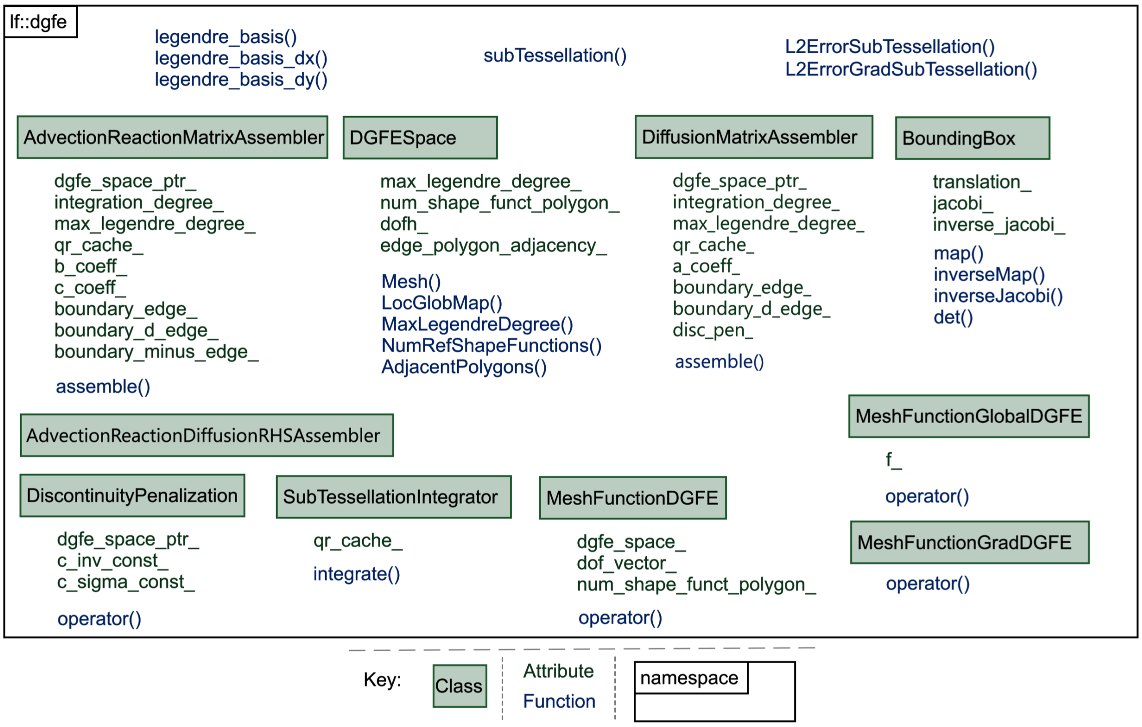
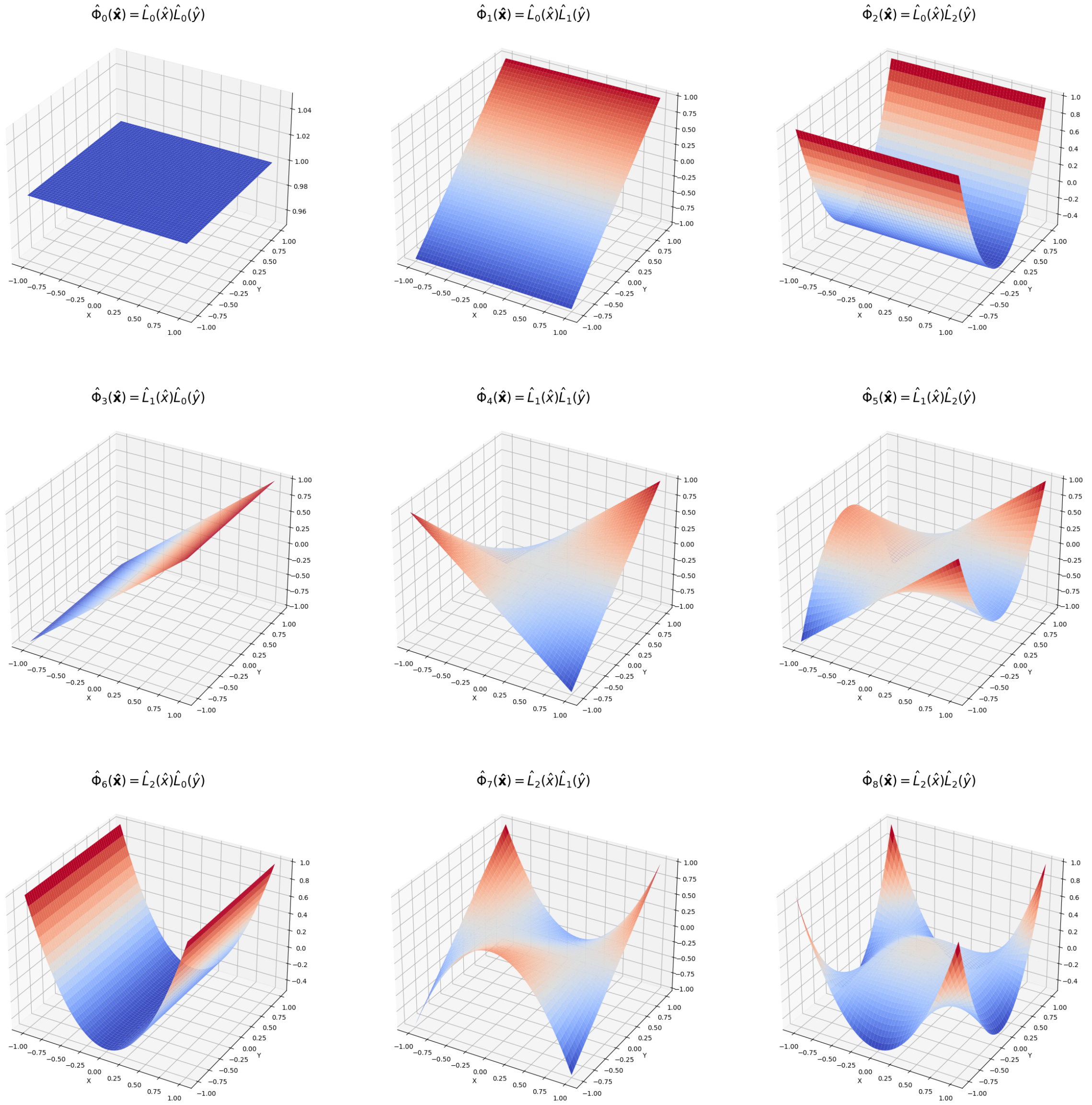


Figure 7: Visualization of the namespace *lf::dgfe*. It is not complete but rather an overview of what is discussed in the report. A key to the symbols can be found at the bottom of the figure.



$\hat{\Phi}_0(\hat{\mathbf{x}}) = \hat{L}_0(\hat{x})\hat{L}_0(\hat{y})$

$\hat{\Phi}_1(\hat{\mathbf{x}}) = \hat{L}_0(\hat{x})\hat{L}_1(\hat{y})$

$\hat{\Phi}_2(\hat{\mathbf{x}}) = \hat{L}_0(\hat{x})\hat{L}_2(\hat{y})$

$\hat{\Phi}_3(\hat{\mathbf{x}}) = \hat{L}_1(\hat{x})\hat{L}_0(\hat{y})$

$\hat{\Phi}_4(\hat{\mathbf{x}}) = \hat{L}_1(\hat{x})\hat{L}_1(\hat{y})$

$\hat{\Phi}_5(\hat{\mathbf{x}}) = \hat{L}_1(\hat{x})\hat{L}_2(\hat{y})$

$\hat{\Phi}_6(\hat{\mathbf{x}}) = \hat{L}_2(\hat{x})\hat{L}_0(\hat{y})$

$\hat{\Phi}_7(\hat{\mathbf{x}}) = \hat{L}_2(\hat{x})\hat{L}_1(\hat{y})$

$\hat{\Phi}_8(\hat{\mathbf{x}}) = \hat{L}_2(\hat{x})\hat{L}_2(\hat{y})$

Figure 6: Plots of tensor product Legendre Polynomials on the unit square in two dimensions used as basis functions $\hat{\Phi}_i(\hat{\mathbf{x}})$. This is the specific case when the maximal degree of one-dimensional Legendre polynomials used is 2.

BoundingBox

The in section 4.1 discussed axis-aligned bounding box is implemented as the class `lf::dgfe::BoundingBox`. It is a fairly lightweight class that is constructed and deleted ”on the fly” within Galerkin assembly algorithms. Once initialized for a specific polygon κ (constructor in listing 6), it implements $\mathbf{F}_\kappa$ as the member function *map()* (listing 7) and the inverse $\mathbf{F}_\kappa^{-1}$ as *inverseMap()* (listing 8). For the integration of functions and gradients of basis functions, the determinant of J_κ and single entries of the inverse of J_κ in equation 3 are needed. The two member functions *det()* and *inverseJacobi()* provide this information.

```
BoundingBox(const lf::mesh::Entity &entity);
```

Listing 6: Declaration of the constructor of `lf::dgfe::BoundingBox`.

```
/**
 * @brief maps from reference bounding box to the cell's bounding box
 *
 * @param corners local points to be mapped into global coordinates
 * @return Eigen::Matrix Global points
 */
Eigen::MatrixXd map(const Eigen::MatrixXd corners);
```

Listing 7: Declaration of `lf::dgfe::BoundingBox::map()`.

```
/**
 * @brief Maps global coordinates into reference bounding box
 *
 * @param corners global points
 * @return Eigen::MatrixXd local points
 */
Eigen::MatrixXd inverseMap(const Eigen::MatrixXd corners);
```

Listing 8: Declaration of `lf::dgfe::BoundingBox::inverseMap()`.

Basis Functions

The tensor product Legendre Polynomials $\hat{\Phi}_i(\hat{\mathbf{x}})$ from section 4.1 are implemented in the function *legendre_basis()*. Its declaration is shown in listing 9.

```
/**
 * @brief returns 2D basis function at coordinate defined on
 * reference bounding box
 *
 * @param n index of basis function on reference bounding box
 * @param max_degree maximum polynomial degree of 1D legendre polnynomials
 * present in basis
 * @param coord point for which the polynomial is evaluated
 */
scalar_t legendre_basis(size_type n, size_type max_degree,
        const Eigen::Vector2d &coord);
```

Listing 9: Declaration of *lf::dgfe::legendre_basis()*.

Furthermore, the derivatives of $\hat{\Phi}_i(\hat{\mathbf{x}})$ in x- and y-dimension are implemented. The declaration in x-direction is in listing 10.

```
/**
 * @brief returns partial derivative in x of 2D reference basis function at coord
 * defined on reference bounding box
 *
 * @note !! DO NOT FORGET TO MULTIPLY WITH ENTRY (0, 0) OF
 * THE INVERSE JACOBI OF THE REFERENCE BOX MAPPING !!
 *
 * @param n nth basis function of
 * @param max_degree maximum degree of 1D legendre polnynomials present in basis
 * @param coord point for which the polynomial is evaluated
 */
scalar_t legendre_basis_dx(size_type n, size_type max_degree,
                            const Eigen::Vector2d &coord);
```

Listing 10: Declaration of *lf::dgfe::legendre_basis_dx()*.

As can be read in listing 10, it is necessary to multiply the derivatives of $\hat{\Phi}_i(\hat{\mathbf{x}})$ with the corresponding entries of J_κ^{-1} from equation 3. This is due to the chain rule of derivatives.

$$\frac{\partial}{\partial x}\Phi_{i,\kappa}(\mathbf{x}) = \frac{\partial}{\partial x}\left[\hat{\Phi}_i(\mathbf{F}_\kappa^{-1}(\mathbf{x}))\right] = \frac{\partial}{\partial x}\left(\hat{\Phi}_i\right)\left(\mathbf{F}_\kappa^{-1}(\mathbf{x})\right)\cdot\frac{\partial}{\partial x}\left(\mathbf{F}_\kappa^{-1}(\mathbf{x})\right) \tag{7}$$

More functions regarding Legendre Polynomials are implemented, but their discussion is omitted here for clarity and priority reasons.

Mesh Functions

A MeshFunction is one of *LehrFEM++*'s concepts. A MeshFunction must particularily overload the bracket operator as shown in listing 11. The concept can be seen in `https://craffael.github.io/lehrfempp/group__mesh__function.html`.

```
std::vector<R> operator()(const lf::mesh::Entity& e,
            const Eigen::MatrixXd& local) const
```

Listing 11: Overloading of the bracket operator to satisfy *LehrFEM++*'s concept of a MeshFunction.

It has been discussed above that there exists no general parametric mapping from a reference polygon to all polygons in a mesh. Therefore the "local" coordinates used in listing 11 have a different meaning in the polytopic setting. Here, the local coordinates are coordinates in the reference bounding box κ_R. The entity e must be a polygon such that the local coordinates can be mapped to global coordinates in the mesh via the mapping $\mathbf{F}_\kappa$. Three different mesh function classes exist. `MeshFunctionGlobalDGFE` is initialized with a lambda function $f(\mathbf{x})$ like in listing 13. `MeshFunctionDGFE` is initialized with a vector holding the coefficients of a basis expansion of a function $f(\mathbf{x})$. `MeshFunctionGradDGFE` works exactly like `MeshFunctionDGFE` but returns the gradient of the function for which the basis expansion is given.

Numerical Integration

As [3, sec. 6.3.1] indicates , developing numerical quadrature rules on polygons is not trivial. To keep it rather simple and universal, triangular sub-tessellations of the polygons are used to then apply standard quadrature rules on each triangle. This approach is computationally inefficient compared to other methods [4, sec. 2.1]. But its implementation is straight-forward and it can be used to integrate any function given in procedural form. The sub-tessellation of polygons is realized in *lf::dgfe::subTessellation()* (listing 12). Given a polygon with n nodes it returns n objects of type `lf::geometry::Tria01` in a vector. Each of them is created from the coordinates of the barycenter of the cell and those of two adjacent nodes. Note that this only works with convex polygons (which *PolyMesher* produces exclusively).

```
/**
 * @brief returns a vector of triangle geometry objects resulting
 * from subdividing a polyon from its barycenter into triangles.
 */
std::vector<std::unique_ptr<lf::geometry::Tria01>> subTessellation(
                        const lf::mesh::Entity *polygon);
```

Listing 12: Declaration of *lf::dgfe::subTessellation()*.

The class `SubTessellationIntegrator` uses *lf::dgfe::subTessellation()* to integrate over polygonal cells. A demonstration for the integration of a function over a polygonal is shown in listing 13. It makes use of the class `lf::dgfe::MeshFunctionGlobalDGFE` which is discussed above. For the integration on the triangle geometries resulting from the sub-tessellation, *LehrFEM++*'s internal quadrature infrastructure is used. SubTessellationIntegrator has a lf::quad::QuadruleCache as a private attribute for efficiency. For a general function $f(x)$, its integration over a polygon is implemented in the following way:

$$\int_{\kappa} f(x)dx = \sum_{T\in\mathscr{S}}\int_{T} f(x)dx \approx \sum_{T\in S}\sum_{l=1}^{p}\hat{w}_l f\left(\Psi_T\left(\hat{\xi}_l\right)\right)\left|\det D\Psi_T\left(\hat{\xi}_l\right)\right|. \tag{8}$$

With Ψ_T being *LehrFEM++*'s mapping from the reference triangle $\hat{T}$ to a general triangle T (Note: in [2, sec. 2.8.1], the local-global mapping is denoted by Φ_K. In this report, Φ already stands for basis functions). T denotes a triangle of the sub-tessellation $\mathscr{S}$ of κ. Additionally, $\hat{w}_l$ are the weights of the quadrature rule employed on the reference triangle, $\hat{\xi}_l$ are the quadrature points on the reference triangle and $\left|\det D\Psi_T\left(\hat{\xi}_l\right)\right|$ are the gramian determinants of the mapping Ψ_T. For MeshFunctions $\mathscr{M}$, the following relation needs to be taken into account and is implemented in SubTessellationIntegrator:

$$f\left(\Psi_T\left(\hat{\xi}_l\right)\right) = \mathscr{M}\left(\kappa\,,\mathbf{F}_{\kappa}^{-1}\left(\Psi_T\left(\hat{\xi}_l\right)\right)\right). \tag{9}$$

```cpp
//get mesh
auto mesh_ptr = lf::mesh::test_utils::GeneratePolytopic2DTestMesh(0,1);
//lambda x^2 + e^(x*y) for mesh function
auto exponential_lambda = [](Eigen::Vector2d x) -> double {
    return x[0]*x[0] + exp(x[0] * x[1]);
};
lf::dgfe::MeshFunctionGlobalDGFE<decltype(exponential_lambda)>
                    exp_msh_funct(exponential_lambda);
lf::dgfe::SubTessellationIntegrator<double, decltype(exp_msh_funct)>
        exp_integrator;
int integration_degree = 10;
double sum = 0.0;
//loop over cells and integrate
for (auto cell : mesh_ptr->Entities(0)){
    sum += exp_integrator.integrate(*cell, exp_msh_funct, integration_degree);
}
```

Listing 13: Demonstration of the integration of $x^2 + e^{x*y}$ over a polytopic mesh of the unit square (test mesh shown in figure 4).

The function *integrate()* takes the degree of exactness of the quadrature rule used for an argument as shown in listing 13.

The infrastructure for numerical integration in the discontinuous setting is completed by two functions to calculate the error over a mesh. More precisely, they calculate the L^2-norm of the difference of two functions. Both of them are implemented as templated functions. For two sclar-valued functions f and g defined on a domain Ω, *L2ErrorSubTessellation()* calculates the following expression:

$$\|f - g\|_{L^2(\Omega)} = \left(\int_\Omega \|f(\mathbf{x}) - g(\mathbf{x})\|^2 \mathrm{d}\mathbf{x}\right)^{\frac{1}{2}}. \tag{10}$$

L2ErrorGradSubTessellation() does the same routine, but here f and g have to be provided as vector-valued functions. The functions feature the word "error" because if one passes the known true solution of a PDE and the calculated approximation in the discrete space, the L^2-norm of the error of the approximation is received.

DGFE Space

A new class `lf::assemble::UniformDGFEDofHandler` does the handling of degrees of freedom in a discontinuous setting and is situated where its classical equivalents are: In `lf/assemble/dofhandler.h`. It is an implementation of the abstract base class `lf::dgfe::DofHandler`.
Most functionalities to solve equation 24 numerically are collected in the class `lf::dgfe::DGFESPace`. It contains a pointer to a `lf::mesh::polytopic2d::Mesh` , the `lf::assemble::UniformDGFEDofHandler` which is used as well as a `lf::mesh::utils::CodimMeshDataSet` with the information about adjacent polygons of each segment (discussed in section 3.1). The `lf::dgfe::DGFESpace` constructor takes a polytopic mesh as a first argument and the maximum polynomial degree of one-dimensional Legendre Polynomials used in the basis functions in table 1. There are only two options available at the moment. Either it is set to 2 which results in the basis functions on the reference bounding box κ_R being exactly as depicted in figure 6. Or it is set to 1 which leads to a basis of 4 functions per polygon, namely those in the top left corner in 6. The DofHandler of the discrete space is initialized in the space's constructor.

5. Boundary Value Problem

5.1. Theory

As has been mentioned multiple times before, the notation used here is almost identical to [3] to make direct links to the book possible. Deriving and explaining all parts of the variational formulation of 1 & 2 would go far beyond the scope of this project and would be a copy of the work done in [3]. What is discussed here are all parts from the book which are necessary to implement and solve the variational formulation of 1. First off, there is a need to introduce a series of operators and symbols used. Let κ_i and κ_j be two adjacent polygons of $\mathscr{T}_h$. F describes the interior face they have in common $F = \partial\kappa_i \cap \partial\kappa_j$. The outward unit normal vectors with respect to κ_i and κ_j on F are indentified with $\mathbf{n}_{\kappa_i}$ and $\mathbf{n}_{\kappa_j}$. Then, v is a general scalar-valued function and $\mathbf{q}$ is a general vector-valued function. In the discontinuous setting it is important to describe precisely to which cell, κ_i or κ_j, a function trace on a common face belongs to. For this reason $\left(v_{\kappa_i}^+, \mathbf{q}_{\kappa_i}^+\right)$ and $\left(v_{\kappa_j}^+, \mathbf{q}_{k_j}^+\right)$ are used to distinguish traces of the functions v and $\mathbf{q}$ taken from the interior of the two cells. Now the average operator can be introduced. For $\mathbf{x} \in F \in \mathscr{F}_h^{\mathscr{I}}$ the averages of v and $\mathbf{q}$ are given by

$$\{\!\{v\}\!\} := \frac{1}{2}\left(v_{\kappa_i}^+ + v_{\kappa_j}^+\right), \quad \{\!\{\mathbf{q}\}\!\} := \frac{1}{2}\left(\mathbf{q}_{\kappa_i}^+ + \mathbf{q}_{\kappa_j}^+\right) \quad . \tag{11}$$

The complementary jump operator is defined by:

$$[\![v]\!] := v_{\kappa_i}^+\mathbf{n}_{\kappa_i} + v_{\kappa_j}^+\mathbf{n}_{\kappa_j}, \quad [\![\mathbf{q}]\!] := \mathbf{q}_{\kappa_i}^+ \cdot \mathbf{n}_{\kappa_i} + \mathbf{q}_{\kappa_j}^+ \cdot \mathbf{n}_{\kappa_j} \quad . \tag{12}$$

If the face $F \in \mathscr{F}_h^{\mathscr{B}}$ is on the boundary of the mesh such that there is only one adjacent cell κ_i, the operators become:

$$\{\!\{v\}\!\} := v_{\kappa_i}^+, \quad \{\!\{\mathbf{q}\}\!\} := \mathbf{q}_{\kappa_i}^+, \quad [\![v]\!] := v_{\kappa_i}^+\mathbf{n}_{\kappa_i} \quad [\![\mathbf{q}]\!] := \mathbf{q}_{\kappa_i}^+ \cdot \mathbf{n}_{\kappa_i} \quad . \tag{13}$$

And the upwind jump operator defined on interior faces $F \in \mathscr{F}_h^{\mathscr{I}}$ is denoted by:

$$\lfloor v \rfloor := v_\kappa^+ - v_\kappa^- \quad . \tag{14}$$

The discontinuity penalization function $\sigma : \mathscr{F}_h^{\mathscr{I}} \cup \mathscr{F}_h^D \to \mathbb{R}$ is defined as:

$$\sigma(\mathbf{x}) := \begin{cases} C_\sigma \max_{\kappa\in\{\kappa^+,\kappa^-\}} \left\{ C_{\mathrm{INV}}\left(p, \kappa, F\right) \frac{A_F|_\kappa \; p_\kappa^2 \; |F|}{|\kappa|} \right\}, & \mathbf{x} \in F \in \mathscr{F}_h, F \subset \partial\kappa^+ \cap \partial\kappa^-, \\ C_\sigma A_F C_{\mathrm{INV}}\left(p_\kappa, \kappa, F\right) \frac{p_\kappa^2 \; |F|}{|\kappa|}, & \mathbf{x} \in F \in \mathscr{F}_h^D, F \subset \partial\kappa. \end{cases} \tag{15}$$

with $A_F := \|\sqrt{\mathbf{A}}\mathbf{n}\|_{L^\infty(F)}^2$, for every face $F \subset \partial\kappa, F \in \mathscr{F}_h^{\mathscr{I}} \cup \mathscr{F}_h^D$, and C_σ a sufficiently large positive constant. The maximal total polynomial degree of the tensor-product Legendre Polynomial basis function is denoted as p_κ. It is the result of adding the maximum polynomial degree of the used one-dimensional Legendre Polynomials. The term p is the polynomial degree appearing in the definition of shape-regularity of

meshes in [3, sec. 3.1, Def. 10] and in this project is defined as the maximum polynomial degree of the used one-dimensional Legendre Polynomials.
The function C_{INV} appearing in 15 is defined as:

$$C_{\text{INV}}(p, \kappa, F) := C_{\text{inv}} \min \left\{ \frac{|\kappa|}{\sup_{\kappa_{\text{b}}^F \subset \kappa} |\kappa_{\text{b}}^F|}, p^{2(d-1)} \right\}. \tag{16}$$

With C_{inv} being a positive constant and $\kappa_{\text{b}}^F \in \mathscr{F}_{\text{b}}^\kappa$. This $\mathscr{F}_{\text{b}}^\kappa$ is the family of all triangles (simplices) contained in κ having at least one common face with κ. Then, $\kappa_{\text{b}}^F \in \mathscr{F}_{\text{b}}^\kappa$ is a simplex which shares the specific face $F \subset \partial\kappa$. A visualization of the implementation of this is depicted in figure 8.

For expressions $v_\kappa^\pm$ it will always be clear to which element $\kappa, \kappa \in \mathscr{T}_h$ the functions correspond to. Therefore the subscript κ is suppressed from now on. A further subdivision of the boundary of the domain $\partial\Omega$ is needed to comply with the notation of [3]:

$$\partial_0\Omega := \left\{ \mathbf{x} \in \partial\Omega : \sum_{i,j=1}^{2} a_{ij}(\mathbf{x})\mathbf{n_i}\mathbf{n_j} > 0 \right\}. \tag{17}$$

With a_{ij} being entries of the diffusion tensor and $\mathbf{n} = (n_1, n_2)^\intercal$ the outward unit normal vector to $\partial\Omega$. The part set of the boundary which is not $\partial_0\Omega$ is further divided:

$$\begin{aligned} \partial_-\Omega &:= \{\mathbf{x} \in \partial\Omega \backslash \partial_0\Omega : \mathbf{b}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) < 0\} \\ \partial_+\Omega &:= \{\mathbf{x} \in \partial\Omega \backslash \partial_0\Omega : \mathbf{b}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) \geq 0\}. \end{aligned} \tag{18}$$

Also $\partial_0\Omega$ is further divided into two sets, namely the set where Dirichlet Boundary Conditions are employed, denoted by $\partial\Omega_D$, and the part where Neumann Boundary Conditions are enforced, which is $\partial\Omega_N$. Therefore $\partial\Omega = \partial_-\Omega \cup \partial_+\Omega \cup \partial\Omega_N \cup \partial\Omega_D$.
Analogously to $\partial_-\Omega$, there exists a subset of the boundary of cells which appears in the DGFEM vraiational formulation of equations 1 and 2:

$$\partial_-\kappa := \{\mathbf{x} \in \partial\kappa : \mathbf{b}(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) < 0\}. \tag{19}$$

The full variational formulation of equation 1 derived in [3, sec. 5.1] is now presented. As stated above, no reasoning or derivation will be given, find all of this in [3, sec. 5.1]. First off, the reaction-advection part of the equation results in:

$$\begin{aligned} B_{\text{ar}}(w, v) := &\sum_{\kappa \in \mathscr{T}_h} \int_\kappa (\nabla \cdot (\mathbf{b}w) + cw)v \; \mathrm{d}\mathbf{x} - \sum_{\kappa \in \mathscr{T}_h} \int_{\partial_-\kappa \backslash \partial\Omega} (\mathbf{b} \cdot \mathbf{n}) \lfloor w \rfloor v^+ \; dS \\ &- \sum_{\kappa \in \mathscr{T}_h} \int_{\partial_-\kappa \cap (\partial\Omega_{\text{D}} \cup \partial_-\Omega)} (\mathbf{b} \cdot \mathbf{n}) w^+ v^+ \mathrm{d}S \quad . \end{aligned} \tag{20}$$

Then, the diffusion part is

$$
\begin{aligned}
\hat{B}_{\mathrm{d}}(w,v) := \sum_{\kappa \in \mathscr{T}_h} \int_{\kappa} a \nabla w \cdot \nabla v \ \mathrm{d}\mathbf{x} + \int_{\mathscr{F}_h^{\mathscr{I}} \cup \mathscr{F}_h^{D}} \sigma [\![w]\!] \cdot [\![v]\!] \mathrm{d}S \\
- \int_{\mathscr{F}_h^{\mathscr{I}} \cup \mathscr{F}_h^{D}} \left(\{\!\{ \sqrt{a} \boldsymbol{\mathit{\Pi}}_{L^2} (\sqrt{a} \nabla w) \}\!\} \cdot [\![v]\!] + \{\!\{ \sqrt{a} \boldsymbol{\mathit{\Pi}}_{L^2} (\sqrt{a} \nabla v) \}\!\} \cdot [\![w]\!] \right) \mathrm{d}S
\end{aligned}
\tag{21}
$$

Where $\boldsymbol{\mathit{\Pi}}_{L^2} : \left[L^2(\Omega)\right]^2 \to \left[V^{\mathbf{p}}\left(\mathscr{T}_h\right)\right]^2$ denotes the orthogonal L^2-projection onto the finite element space $\left[V^{\mathbf{p}}\left(\mathscr{T}_h\right)\right]^2$.

Putting these two parts together results in the bilinear form of the variational formulation:

$$
B(w,v) := B_{\mathrm{ar}}(w,v) + \hat{B}_{\mathrm{d}}(w,v) \tag{22}
$$

The linear term of the variational formulation is:

$$
\begin{aligned}
\hat{\ell}(v) := \sum_{\kappa \in \mathscr{T}_h} \int_{\kappa} f v \ \mathrm{d}\mathbf{x} - \sum_{\kappa \in \mathscr{T}_h} \int_{\partial - \kappa \cap (\partial \Omega_{\mathrm{D}} \cup \partial - \Omega)} (\mathbf{b} \cdot \mathbf{n}) g_{\mathrm{D}} v^{+} \mathrm{d}s \\
- \int_{\partial \Omega_{\mathrm{D}}} g_{\mathrm{D}} \left(\sqrt{a} \mathbf{\Pi}_{L^2} (\sqrt{a} \nabla v) \cdot \mathbf{n} - \sigma v \right) \mathrm{d}s + \int_{\partial \Omega_{\mathrm{N}}} g_{\mathrm{N}} v \ \mathrm{d}s.
\end{aligned}
\tag{23}
$$

Finally, the full DGFEM approximation of the original problem 1: Find $u_h \in V^{\mathbf{p}}(\mathscr{T}_h)$ such that:

$$
B\left(u_h, v_h\right) = \hat{\ell}\left(v_h\right) \quad \forall v_h \in V^{\mathbf{p}}\left(\mathscr{T}_h\right). \tag{24}
$$

5.2. Implementation

Discontinuity Penalization

The SIP is implemented as a class as seen in figure 7. Listing 14 shows its constructor. The class owns a pointer to the discrete space used so it can access the data set which contains information about the adjacent polygons of each face in the mesh. How to set the constants C_{inv} and C_{σ} is not defined clearly in [3]. They are rather heuristic values whose correctness can be shown by the method of manufactured solutions. And they are used for the theoretical error analysis in [3]. The variable A_F needs to be calculated in the program before calling *operator()* of the SIP on a face.

The implementation of the SIP is rather straight-forward. A visualization of the calculation of $\kappa_{\mathrm{b}}^{F} \in \mathscr{F}_{\mathrm{b}}^{\kappa}$ is depicted in figure 8. This is implemented in a function *simplexAreas()* which returns a vector of the areas of $\kappa_{\mathrm{b}}^{F} \in \mathscr{F}_{\mathrm{b}}^{\kappa}$ as shown in figure 8. The declaration of the constructor and the *operator()* of the SIP-class is shown in listing 14.

```
DiscontinuityPenalization(std::shared_ptr<const lf::dgfe::DGFESpace>
                          dgfe_space_ptr, scalar_t c_inv_constant, scalar_t
                          c_sigma_constant) :
                          dgfe_space_ptr_(std::move(dgfe_space_ptr)),
                          c_inv_const_(c_inv_constant),
                          c_sigma_const_(c_sigma_constant) {}

scalar_t operator()(const lf::mesh::Entity &edge, scalar_t A_f) const ;
```

Listing 14: Declaration of the constructor of `lf::dgfe::DiscontinuityPenalization` and the declaration of its operator().

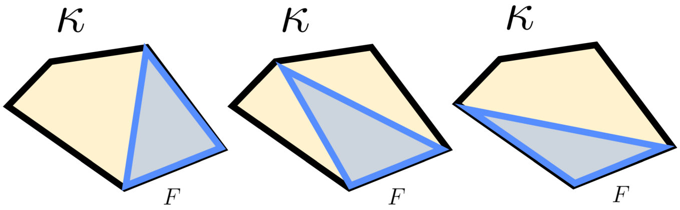


Figure 8: Visualization of the explicit calculation of $\kappa_\mathrm{b}^F \in \mathscr{F}_\mathrm{b}^\kappa$. For a cell κ and a face $F \in \partial\kappa$ the three blue triangles are the κ_b^F which are taken into account.

Assembly of Galerkin Matrix and RHS Vector

Assembling the Galerkin Matrix (eq.21) correctly was the hardest step of the project. This process, including the simplification of the more complex terms and their transformation into pseudocode is presented here. All terms which cannot be implemented straight-forward and include jump(12), average(11) or upwind jump(14) operators are discussed.

Diffusion The second and third addend of equation 21 both have different forms for interior and boundary segments. Equations 12, 11 and 13 are used to arrive at the following results.
Firstly the second addend of the diffusion term is presented. For interior edges:

$$
\begin{aligned}
&\int_{\mathscr{F}_h^{\mathscr{I}}} \sigma [\![w]\!] \cdot [\![v]\!] \; dS \\
&= \sigma \int_{\mathscr{F}_h^{\mathscr{I}}} (w_i \cdot \mathbf{n}_i + w_j \cdot \mathbf{n}_j)\,(v_i \cdot \mathbf{n}_i + v_j \cdot \mathbf{n}_j) \;\; dS \\
&= \sigma \int_{\mathscr{F}_h^{\mathscr{I}}} (w_i \cdot \mathbf{n}_i - w_j \cdot \mathbf{n}_i)\,(v_i \cdot \mathbf{n}_i - v_j \cdot \mathbf{n}_i) \;\; dS \\
&= \sigma \int_{\mathscr{F}_h^{\mathscr{I}}} w_i \cdot v_i \cdot \mathbf{n}_i^2 - w_i \cdot v_j \cdot \mathbf{n}_i^2 - w_j \cdot v_i \cdot \mathbf{n}_i^2 + w_j \cdot v_j \cdot \mathbf{n}_i^2 \; dS \\
&= \sigma \int_{\mathscr{F}_h^{\mathscr{I}}} w_i \cdot v_i - w_i \cdot v_j - w_j \cdot v_i + w_j \cdot v_j \; dS
\end{aligned} \tag{25}
$$

And for boundary edges:

$$
\begin{aligned}
&\int_{\mathscr{F}_h^{D}} \sigma [\![w]\!] \cdot [\![v]\!] \; dS \\
&= \sigma \int_{\mathscr{F}_h^{D}} w \cdot \mathbf{n} \cdot v \cdot \mathbf{n} \; dS \\
&= \sigma \int_{\mathscr{F}_h^{D}} w \cdot v \; dS
\end{aligned} \tag{26}
$$

This second addend of equation 21 is assembled in the Galerkin Matrix with algorithm 1.

Algorithm 1: Assembly of the Galerkin Matrix regarding the second addend of equation 21.

```
for each $e \in \mathscr{F}_h^{\mathscr{I}}$ do
    for $v$ = 1 to # basis functions per cell do
        for $w$ = 1 to # basis functions per cell do
            Add $\sigma \int_e w_i \cdot v_i \; dS$ to Galerkin Matrix at (DOF(v$_i$), $DOF(w_i)$)
            Add $-\sigma \int_e w_i \cdot v_j \; dS$ to Galerkin Matrix at (DOF(v$_j$), $DOF(w_i)$)
            Add $-\sigma \int_e w_j \cdot v_i \; dS$ to Galerkin Matrix at (DOF(v$_i$), $DOF(w_j)$)
            Add $\sigma \int_e w_j \cdot v_j \; dS$ to Galerkin Matrix at (DOF(v$_j$), $DOF(w_j)$)
        end
    end
end
for each $e \in \mathscr{F}_h^{D}$ do
    for $v$ = 1 to # basis functions per cell do
        for $w$ = 1 to # basis functions per cell do
            Add $\sigma \int_e w \cdot v \; dS$ to Galerkin Matrix at (DOF(v), DOF(w))
        end
    end
end
```

Note that w_i and w_j refer to traces of the basis functions on two different cells κ_i and κ_j adjacent to edge e. When the edge is on the boundary, there is only one adjacent cell with the basis functions w and v.

The third addend of equation 21 features an orthogonal L^2-projection $\boldsymbol{\mathit{\Pi}}_{L^2}$ onto the finite element space $V^{\mathbf{p}}(\mathscr{T}_h)$. Although the projection plays a factor in the methods' ability to solve problems with strong discontinuities, its discussion and also its implementation is omitted in this project for simplicity and clarity reasons. They remain a task for the future.

In the following the expansion of one part of the third addend of 21 is presented. The whole addend features two terms which are exactly the same except that trial (w) and test (v) functions are swapped. Shown is only one of those two parts. For interior edges:

$$\begin{aligned}
&\int_{\mathscr{F}_h^{\mathscr{I}}} \{\!\!\{a\nabla w\}\!\!\} \cdot [\![v]\!]\; dS \\
&= \int_{\mathscr{F}_h^{\mathscr{I}}} \frac{1}{2}\left(a\nabla w_i + a\nabla w_j\right)\cdot\left(v_i\cdot\mathbf{n_i} + v_j\cdot\mathbf{n_j}\right) dS \\
&= \frac{1}{2}\int_{\mathscr{F}_h^{\mathscr{I}}} \left(a\nabla w_i + a\nabla w_j\right)\cdot\left(v_i\cdot\mathbf{n_i} - v_j\cdot\mathbf{n_i}\right) dS \\
&= \frac{1}{2}\int_{\mathscr{F}_h^{\mathscr{I}}} a\nabla w_i\cdot v_i\cdot\mathbf{n_i} - a\nabla w_i\cdot v_j\cdot\mathbf{n_i} + a\nabla w_j\cdot v_i\cdot\mathbf{n_i} - a\nabla w_j\cdot v_j\cdot\mathbf{n_i}\; dS
\end{aligned} \tag{27}$$

And for boundary edges:

$$\begin{aligned}
&\int_{\mathscr{F}_h^{D}} \{\!\!\{a\nabla w\}\!\!\} \cdot [\![v]\!]\; dS \\
&= \int_{\mathscr{F}_h^{D}} a\nabla w\cdot v\cdot\mathbf{n}\; dS
\end{aligned} \tag{28}$$

Which results in algorithm 2 to assemble its entries in the Galerkin Matrix.

Algorithm 2: Assembly of the Galerkin Matrix regarding one part of the third addend of equation 21.

```
for each $e \in \mathscr{F}_h^{\mathscr{I}}$ do
    for $v$ = 1 to # basis functions per cell do
        for $w$ = 1 to # basis functions per cell do
            Add $-\frac{1}{2}\int_e a\nabla w_i \cdot v_i \cdot \mathbf{n_i}\ dS$ to Galerkin at $(\text{DOF}(\text{v}_i), DOF(w_i))$
            Add $\frac{1}{2}\int_e a\nabla w_i \cdot v_j \cdot \mathbf{n_i}\ dS$ to Galerkin at $(\text{DOF}(\text{v}_j), DOF(w_i))$
            Add $-\frac{1}{2}\int_e a\nabla w_j \cdot v_i \cdot \mathbf{n_i}\ dS$ to Galerkin at $(\text{DOF}(\text{v}_i), DOF(w_j))$
            Add $\frac{1}{2}\int_e a\nabla w_j \cdot v_j \cdot \mathbf{n_i}\ dS$ to Galerkin at $(\text{DOF}(\text{v}_j), DOF(w_j))$
        end
    end
end
for each $e \in \mathscr{F}_h^{D}$ do
    for $v$ = 1 to # basis functions per cell do
        for $w$ = 1 to # basis functions per cell do
            Add $\int_e a\nabla w \cdot v \cdot \mathbf{n}\ dS$ to Galerkin at (DOF(v), DOF(w))
        end
    end
end
```

Advection-Reaction And the last term which is extensively discussed appears in equation 20 and features the upwind jump operator (14).

$$\begin{aligned} &\int_{\partial_-\kappa\setminus\partial\Omega} (\mathbf{b}\cdot\mathbf{n})\lfloor w \rfloor v^+\ dS \\ =&\int_{\partial_-\kappa\setminus\partial\Omega} (\mathbf{b}\cdot\mathbf{n})\cdot(w^+ - w^-)\cdot v^+\ dS \\ =&\int_{\partial_-\kappa\setminus\partial\Omega} \mathbf{b}\cdot\mathbf{n}\cdot w^+\cdot v^+ - \mathbf{b}\cdot\mathbf{n}\cdot w^-\cdot v^+\ dS \end{aligned} \tag{29}$$

Here, w^+ and v^+ refer to basis functions defined on the current cell in the sum while w^- refers to a basis function defined on another cell adjacent to $\partial_-\kappa\setminus\partial\Omega$ (see equation 19 for further explanation on this set). The corresponding algorithm is displayed in algorithm 3.

Algorithm 3: Assembly of the Galerkin Matrix regarding the second addend of equation 20.

```
for each $\kappa \in \mathscr{T}_h$ do
    for each $e \in \partial\_\kappa \backslash \partial\Omega$ do
        for $v$ = 1 to # basis functions per cell do
            for $w$ = 1 to # basis functions per cell do
                Add $-\int_e \mathbf{b} \cdot \mathbf{n} \cdot w^+ \cdot v^+ \; dS$ to Galerkin at
                 $(\mathrm{DOF(v^+)}, DOF(w^+))$
                Add $\int_e \mathbf{b} \cdot \mathbf{n} \cdot w^+ \cdot v^+ \; dS$ to Galerkin at $(\mathrm{DOF(v^+)}, DOF(w^-))$
            end
        end
    end
end
```

In contrast to the assembly of the LSE components of the variational formulation in the classical setting, the DGFEM version routines cannot be trivially passed to a general assembly algorithm that iterates over cells of codimension 0. The equations 20, 21 and 23 all feature parts of integration over sets of faces in the mesh rather than cells only. Additionally, also the cell-oriented assembly depends on information from the edges of the mesh.

Extensive routines for the DGFEM LSE assembly are implemented in classes representing the equations 20, 21 and 23. The classes `AdvectionReactionMatrixAssembler` and `DiffusionMatrixAssembler` assemble entries in the Galerkin matrix themselves and are not passed to *LehrFEM++*'s assembling algorithms *lf::assemble::AssembleMatrixLocally()*. The same goes for the class which assembles the linear term (right hand side vector of 24 `AdvectionReactionDiffusionRHSAssembler`.

For a full example of the usage of all essential parts of the implementation refer to the complete listing 15 in the appendix.

6. Numerical Experiments

Finite Element Methods are proven to be correct by the method of manifactured solutions. A known solution is inserted into the PDE, the problem is solved with according boundary conditions and the convergence of the approximation error is studied.

In this case 1 and 2 are solved on a series of Voronoi Tessellations (of which some are displayed in figure 2) of the unit square $(0,1)^2$ generated by *PolyMesher*. Neumann boundary conditions are employed on Γ_N which is made up of all edges of which both nodes are on the side $x = 1$. The rest of the boundary belongs to Γ_D and is employed with dirichlet boundary conditions. The diffusion coefficient is defined by $a(x,y) = \delta I_2$ with $\delta = \sin(4 \cdot \Pi \cdot (x+y))^2 + 1$. The advection coefficient is $[2-xy, 2-x^2]^T$

and the reaction coefficient $c = (1 + x) \cdot (1 + y)^2$.
This problem has the analytical solution $u_{true} = 1 + \sin(\Pi \cdot (1 + x) \cdot (1 + y)^2 \cdot \frac{1}{8})$. The parameters of the discontinuity penalization in equation 15 and 16 are set to $C_{\text{inv}} = 0.5$ and $C_\sigma = 20$. Figure 9 shows that the L_2 error of the approximated solution converges as is expected and shown in [3, sec. 6.4.1].

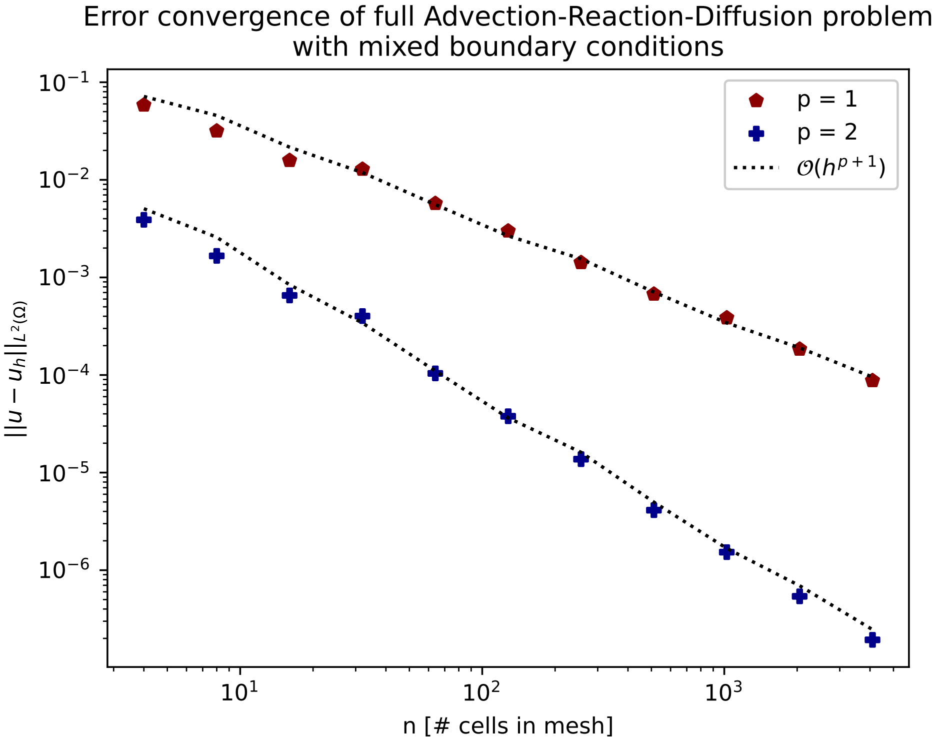


Figure 9: Convergence of the L_2-error of solutions to 1 & 2 using discontinuous Galerkin FEM. Constants set for the SIP are $C_{\text{inv}} = 0.5$ and $C_\sigma = 20$. P describes to polynomial degree of one-dimensional Legendre polynomials used for basis functions. The maxmimum mesh width of the Voronoi tessellations is denoted by h.

7. Learnings and Outlook

As is usual in programming projects, many problems that have not been not on the radar before were encountered. Already the incorporation of polytopic meshes into *LehrFEM++* and especially their generation and data import into the program was full of problems. The code provided for *PolyMesher* did not work out of the box. One actual bug had to be found and multiple adaptions were necessary for the process to work as planned. The bug will be reported to the authors of *PolyMesher*. A big learning is that one should first go for the straight-forward and universal implementations before trying to go for an elegant and efficient type. This was particularly the case for the numerical integration used on the polytopic mesh. Displayed in the appendix B that the original idea was to use an algorithm used for the very fast integration of homogeneous functions, presented in [4]. After some troubles of actually implementing it (it is still present on the current repository branch but only works for polynomial functions), it was noticed that is not universal enough for the use in this project. In general, many parts were implemented without being absolutely necessary. But that is okay for a Bachelor project. Self-organization and pragmatic problem solving were needed. The challenge was interesting and an instructive experience.

As mentioned before, the implementation of the orthogonal L^2-projection $\boldsymbol{\Pi}_{L^2}$ onto the finite element space $V^{\mathbf{p}}(\mathscr{T}_h)$ remains a task.

Most algorithms implemented in the project can surely be implemented a lot more efficiently. The focus is set strictly to correct results while performance optimizations are kept to a minimum.k

References

[1] Hiptmair, R., Casagrande, R. et al, *LEHRFEM++*, Simplistic Finite Element Framework for research and education, `https://github.com/craffael/LehrFEMpp`

[2] Hiptmair, Ralf. "Numerical Methods for Partial Differential Equations". Spring 2023, ETH Zurich. `https://people.math.ethz.ch/~grsam/NUMPDEFL/NUMPDE.pdf`

[3] Cangiani, A., Dong, Z., Geourgoulis, E. H., Houston, P. (2017). *hp-Version Discontinuous Galerkin Methods on Polygonal and Polyhedral Meshes*. SpringerBriefs in Mathematics. `https://doi.org/10.1007/978-3-319-67673-9`

[4] Antonetti, P. F., Houston, P., Pennesi, G. (2018). *Fast Numerical Integration on Polytopic Meshes with Applications to Discontinuous Galerkin Finite Element Methods*. Journal of Scientific Computing (2018) 77:1339–1370. `https://doi.org/10.1007/s10915-018-0802-y`

[5] A. Cangiani, E.H. Georgoulis, P. Houston, hp-Version discontinuous Galerkin methods on polygonal and polyhedral meshes. Math. Models Methods Appl. Sci. **24**(10),(2014)

[6] Talischi, C., Paulino, G. H., Pereira, A., Menezes, I. F. M. (2012). *PolyMesher: a general-purpose mesh generator for polygonal elements written in Matlab*. `https://link.springer.com/article/10.1007/s00158-011-0706-z`

[7] Georgoulis, E. H., Lasis A., *A note on the design of hp-version interior penalty discontinuous Galerkin finite element methods for degenerate problems* . IMA Journal of Numerical Analysis **26**(2) (2006) `https://doi.org/10.1093/imanum/dri038`

A. Script Example

```cpp
/** @file
 *  @brief Bachelor Thesis DGFEM
 *  @author Tarzis Maurer
 *  @date July 23
 *  @copyright ETH Zurich
 */

#include <cstdlib>
#include <filesystem>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>
#include <iomanip>

#include <lf/mesh/polytopic2d/polytopic2d.h>
#include <lf/mesh/hybrid2d/hybrid2d.h>
#include <lf/mesh/mesh.h>
#include <lf/io/io.h>
#include <lf/mesh/utils/utils.h>
#include <lf/base/base.h>
#include <lf/dgfe/dgfe.h>
#include <lf/fe/fe.h>
#include <lf/uscalfe/uscalfe.h>

#include "lf/mesh/test_utils/test_meshes.h"

//function to write error to a file
void write_error_file(std::string run_name, double c_inv, int c_sigma, int num_cells, std::string error_type, double error){
    //error file
    std::setprecision(17);
    auto c_inv_str = std::to_string(c_inv);
    c_inv_str.resize(4);
    auto c_sigma_str = std::to_string(c_sigma);
    std::string out_file_name = "measurements/" + run_name + "/" + std::to_string(num_cells) + "_" + c_inv_str
                                 + "_" + c_sigma_str + "_" + error_type + ".txt";
    std::ofstream out_file(out_file_name);
    out_file << error;
    out_file.close();
}

int main(int argc, char *argv[]){

```

```
//############ USAGE of this script in bash ########################
/*

./projects.dgfem.full_bvp <RUN_NAME> <C_INV> <C_SIGMA> <LIST OF # cells in mesh separated by space>

EXAMPLE:

`\newpage`

./projects.dgfem.full_bvp my_run_name 0.5 20 4 8 16 32 64 128 256 512 1024 2048 4096

*/

//get run arguments: name and constants for discontinuity penalty
std::string run_name = argv[1];
double c_inv = std::stod(argv[2]);
double c_sigma = std::stod(argv[3]);
std::cout << "C_inv: " << c_inv << " and C_sigma: " << c_sigma << "\n";

//set the integration degree used for assembling Galerkin Matrix and RHS
int integration_degree = 15;

//----------------------PREPARE COEFFICIENTS------------------------
// Scalar valued reaction coefficient c
auto c_coeff_lambda = [](Eigen::Vector2d x) -> double {
    return (1 + x[0]) * (1 + x[1]) * (1 + x[1]);
};
lf::dgfe::MeshFunctionGlobalDGFE m_c_coeff{c_coeff_lambda};
//Vector valued advection coefficient b
auto b_coeff_lambda = [](Eigen::Vector2d x) -> Eigen::Vector2d {
    return (Eigen::Vector2d{2 - x[0]* x[1] , 2 - x[0]*x[0]});
};
lf::dgfe::MeshFunctionGlobalDGFE m_b_coeff{b_coeff_lambda};
// Scalar valued div of advection coeff
auto div_b_coeff_lambda = [](Eigen::Vector2d x) -> double {
    return -x[1];
};
lf::dgfe::MeshFunctionGlobalDGFE m_div_b_coeff{div_b_coeff_lambda};
// 2x2 diffusion tensor A(x)
auto a_coeff_lambda = [](Eigen::Vector2d x) -> Eigen::Matrix<double, 2, 2> {
    double entry = 1.0 + (std::sin(4.0 * M_PI *(x[0] + x[1]))) * (std::sin(4.0 * M_PI *(x[0] + x[1])));
    return (Eigen::Matrix<double, 2, 2>() << entry, 0.0, 0.0, entry).finished();
};
lf::dgfe::MeshFunctionGlobalDGFE m_a_coeff{a_coeff_lambda};
```

```
//----------------------END PREPARE COEFFICIENTS------------------------


//----------------------PREPARE PRESCRIBED FUNCTIONS------------------------
//define eulers number
const double E =
↪  2.7182818284590452353602874713526624977572470936999595749669676277;
// Scalar valued prescribed function gD
auto gD_lambda = [](Eigen::Vector2d x) -> double {
    return 1.0 + std::sin(M_PI * (1.0 + x[0]) * (1.0 + x[1]) * (1.0 + x[1]) *
↪  0.125);
};
lf::dgfe::MeshFunctionGlobalDGFE m_gD{gD_lambda};

auto gN_lambda = [E](Eigen::Vector2d x) -> double {
    return 0.39269908169872414*std::pow(1 + x[1],2)*
        std::cos(0.39269908169872414*(1.0 + x[0])*std::pow(1 + x[1],2))*
        (1.0 + std::pow(std::sin(4*M_PI*(x[0] + x[1])),2));
};
lf::dgfe::MeshFunctionGlobalDGFE m_gN{gN_lambda};

// Scalar valued prescribed function f
auto f_lambda = [](Eigen::Vector2d x) -> double {
    return 0.7853981633974483*(1 + x[0])*(2 - std::pow(x[0],2))*(1 + x[1])*
    std::cos(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2)) +
   0.39269908169872414*std::pow(1 + x[1],2)*(2 - x[0]*x[1])*
    std::cos(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2)) -
   x[1]*(1 + std::sin(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))) +
   (1 + x[0])*std::pow(1 + x[1],2)*
    (1 + std::sin(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))) -
   19.739208802178716*(1 + x[0])*(1 + x[1])*
    std::cos(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))*
    std::cos(4*M_PI*(x[0] + x[1]))*std::sin(4*M_PI*(x[0] + x[1])) -
   9.869604401089358*std::pow(1 + x[1],2)*
    std::cos(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))*
    std::cos(4*M_PI*(x[0] + x[1]))*std::sin(4*M_PI*(x[0] + x[1])) -
   0.7853981633974483*(1 + x[0])*
    std::cos(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))*
    (1 + std::pow(std::sin(4*M_PI*(x[0] + x[1])),2)) +
   0.6168502750680849*std::pow(1 + x[0],2)*std::pow(1 + x[1],2)*
    std::sin(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))*
    (1 + std::pow(std::sin(4*M_PI*(x[0] + x[1])),2)) +
   0.15421256876702122*std::pow(1 + x[1],4)*
    std::sin(0.39269908169872414*(1 + x[0])*std::pow(1 + x[1],2))*
    (1 + std::pow(std::sin(4*M_PI*(x[0] + x[1])),2));
};
lf::dgfe::MeshFunctionGlobalDGFE m_f{f_lambda};
```

```cpp
//---------------------END PREPARE PRESCRIBED FUNCTIONS-----------------------

//------------------------LOOP over Meshes--------------------------
//loop over meshes
for (int i = 4; i < argc; i++){

    //read number of cells
    std::string num_cells = argv[i];

    //get mesh
    std::filesystem::path here = __FILE__;
    auto mesh_file = here.parent_path().string() + "/msh_files/unit_square_voronoi_" + num_cells + "_cells.vtk";
    lf::io::VtkPolytopicReader reader(std::make_unique<lf::mesh::polytopic2d::MeshFactory>(2), mesh_file);
    auto mesh_ptr = reader.mesh();

    //write mesh for python drawing
    //lf::io::writeMatplotlib(*mesh_ptr, "./csvs/" + std::to_string(mesh_ptr->NumEntities(0)) + ".csv");

    //dgfe space p = 1
    lf::dgfe::DGFESpace dgfe_space(mesh_ptr, 1);
    auto dgfe_space_ptr = std::make_shared<lf::dgfe::DGFESpace>(dgfe_space);

    //---------------------PREPARE BOUNDARY EDGE SETS-----------------
    auto boundary_edge = lf::mesh::utils::flagEntitiesOnBoundary(mesh_ptr, 1);
    //boundary_N_edge
    lf::mesh::utils::CodimMeshDataSet<bool> boundary_n_edge(mesh_ptr, 1, false);
    //boundary_0_edge
    lf::mesh::utils::CodimMeshDataSet<bool> boundary_0_edge(mesh_ptr, 1, false);
    //boundary_D_edge
    lf::mesh::utils::CodimMeshDataSet<bool> boundary_d_edge(mesh_ptr, 1, false);
    //boundary_minus_edge
    lf::mesh::utils::CodimMeshDataSet<bool> boundary_minus_edge(mesh_ptr, 1, false);
    //boundary_plus_edge
    lf::mesh::utils::CodimMeshDataSet<bool> boundary_plus_edge(mesh_ptr, 1, false);

    //setup qr rule for segments
    const lf::quad::QuadRule qr_s = lf::quad::make_QuadRule(lf::base::RefEl::kSegment(), integration_degree);
    // qr points
    const Eigen::MatrixXd zeta_ref_s{qr_s.Points()};
    //weights
    Eigen::VectorXd w_ref_s{qr_s.Weights()};

    //BOUNDARY SETS ASSEMBLY
    for (auto cell : mesh_ptr->Entities(0)){
        for (auto edge : cell->SubEntities(1)){
            if (boundary_edge(*edge)){
                //normal n
                auto polygon_pair = dgfe_space_ptr->AdjacentPolygons(edge);
                auto normal = lf::dgfe::outwardNormal(lf::geometry::Corners(*(edge->Geometry())));
                //if orientation of edge in polygon is negative, normal has to be multiplied by -1;
                normal *= (int) (cell->RelativeOrientations()[polygon_pair.first.second]);

                lf::dgfe::BoundingBox box(*cell);
                // qr points mapped to segment
                Eigen::MatrixXd zeta_global_s{edge->Geometry()->Global(zeta_ref_s)};
                // qr points mapped back into reference bounding box to retrieve values
                Eigen::MatrixXd zeta_box_s{box.inverseMap(zeta_global_s)};
                //gramian determinants
                Eigen::VectorXd gram_dets_s{edge->Geometry()->IntegrationElement(zeta_ref_s)};
                auto a_evaluated = m_a_coeff(*cell, zeta_box_s);
                double boundary_0_sum = 0.0;

                for (int i = 0; i < gram_dets_s.size(); i++){
                    boundary_0_sum += normal.dot(a_evaluated[i] * normal) * gram_dets_s[i] * w_ref_s[i];
                }
                //BOUNDARY 0 #############
                if (boundary_0_sum > 0){
                    boundary_0_edge(*edge) = true;
                    //HERE DIRICHLET AND NEUMANN ################
                    auto corners = lf::geometry::Corners(*(edge->Geometry()));
                    if(corners(0,0) == 1.0 && corners(0,1) == 1.0){ //whole edge on side x = 1
                        boundary_n_edge(*edge) = true;
                    } else {
                        boundary_d_edge(*edge) = true;
                    }
                } else { //BOUNDARY_plus and BOUNDARY_minus ###############
                    auto b_evaluated = m_b_coeff(*cell, zeta_box_s);
                    double boundary_plus_sum = 0.0;
                    for (int i = 0; i < gram_dets_s.size(); i++){
```

```
              boundary_0_sum += b_evaluated[i].dot(normal) * gram_dets_s[i] * w_ref_s[i];
            }
            if (boundary_plus_sum < 0){
              boundary_minus_edge(*edge) = true;
            } else {
              boundary_plus_edge(*edge) = true;
            }
          }
        }
      }
    }
    //----------------------END PREPARE BOUNDARY EDGE SETS------------------------

    //----------------------ASSEMBLE GALERKIN MATRIX & RHS------------------------
    //set up discontinuity penalization
    lf::dgfe::DiscontinuityPenalization disc_pen(dgfe_space_ptr, c_inv, c_sigma);
    unsigned n_dofs = dgfe_space_ptr->LocGlobMap().NumDofs();

    //galerkin matrix initialization
    lf::assemble::COOMatrix<double> A(n_dofs, n_dofs);
    A.setZero();
    //diffusion assembler
    lf::dgfe::DiffusionMatrixAssembler<decltype(A), double, decltype(m_a_coeff), decltype(boundary_edge)>
            diffusionAssembler(dgfe_space_ptr, m_a_coeff, boundary_edge, boundary_d_edge, integration_degree, disc_pen);
    //advection reaction matrix assembler
    lf::dgfe::AdvectionReactionMatrixAssembler<decltype(A), double, decltype(m_b_coeff), decltype(m_c_coeff), decltype(boundary_edge), decltype(m_div_b_coeff)>
            advectionReactionAssembler(dgfe_space_ptr, m_b_coeff, m_div_b_coeff, m_c_coeff, boundary_edge, boundary_d_edge, boundary_minus_edge, integration_degree);
    //assemble matrix
    diffusionAssembler.assemble(A);
    advectionReactionAssembler.assemble(A);

    //rhs initialization
    Eigen::VectorXd rhs(n_dofs);
    rhs.setZero();
    //RHS Assembler
    lf::dgfe::AdvectionReactionDiffusionRHSAssembler<double, decltype(m_a_coeff), decltype(m_b_coeff), decltype(boundary_edge), decltype(m_f), decltype(m_gD),
                          decltype(m_gN),
↪ decltype(rhs)>
              rhsAssembler(dgfe_space_ptr, m_f, m_gD, m_gN,
↪ m_a_coeff, m_b_coeff, boundary_minus_edge,
              boundary_d_edge, boundary_n_edge, integration_degree,
↪ disc_pen);
  //assemble rhs vector
  rhsAssembler.assemble(rhs);
  //----------------------END ASSEMBLE GALERKIN MATRIX &
↪ RHS-----------------------

  //----------------------SOLVE LSE-----------------------
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  Eigen::VectorXd sol_vec = solver.solve(rhs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
  //----------------------END SOLVE LSE-----------------------

  //----------------------MESH FUNCTION AND ERROR
↪ CALCULATION-----------------------
  lf::dgfe::MeshFunctionDGFE<double> dgfe_mesh_function(dgfe_space_ptr,
↪ sol_vec);

  //calculate L2 error of solution with "overkill" QR
  double mesh_func_l2_error = lf::dgfe::L2ErrorSubTessellation<double,
↪ decltype(dgfe_mesh_function), decltype(m_gD)>(dgfe_mesh_function, m_gD,
↪ mesh_ptr, 30);
  //----------------------END MESH FUNCTION AND ERROR
↪ CALCULATION-----------------------

  //----------------------WRITE ERROR TO FILE-----------------------
  write_error_file(run_name, c_inv, c_sigma, std::stoi(num_cells), "L2",
↪ mesh_func_l2_error);

  std::cout << "L2 Error for " << num_cells << " cells: \t" <<
↪ mesh_func_l2_error << "\n";
  //---------------------END WRITE ERROR TO FILE-----------------------

}
//-----------------------END LOOP over Meshes----------------------------------

return 0;
} //end main
```

Listing 15: Script example to produce the measurements with $p = 1$ in figure 9.

B. Proposed Changes

Here the proposed changes to the *LehrFEM++* library which have been formulated in the beginning of the project.

Changes LehrFem++ version 2 (polytopic meshes & DGFE)

Before starting with the changes to the different modules I present the method I will apply to integrate the necessary terms over polytopes and segments. I will closely follow the paper of Antonetti P., Houston P. and Pennesi G.: "Fast Numerical Integration on polytopic Meshes with Applications to Discontinuous Galerkin Finite Element Methods"; https://link.springer.com/article/10.1007/s10915-018-0802-y. It provides information on how to assemble the Galerkin Matrix. The polygons' axis-aligned bounding boxes are mapped to the reference bounding box $(-1,1)^2$.

The computations for the element matrices is then done on the mappings of the polygons within $(-1,1)^2$. The reason for this is the choice of Legendre Polynomials defined on $(-1,1)^2$ as basis functions of the discrete space V^h. More of this in a later section.

The transformation $F_\mathcal{K}: \hat{B} \rightarrow B_\mathcal{K}$ maps the reference bounding box $\hat{B} := (-1,1)^2$ to the axis-aligned bounding box $B_\mathcal{K}$ of one specific polygon $\mathcal{K}$ in the mesh.

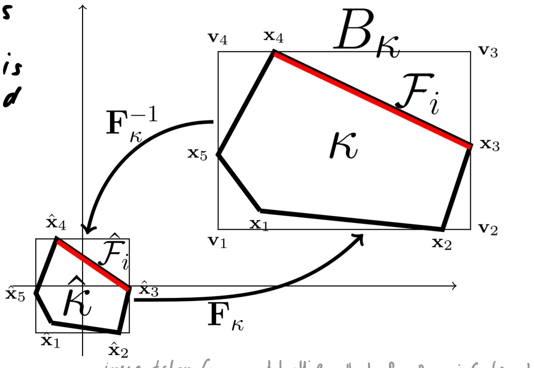
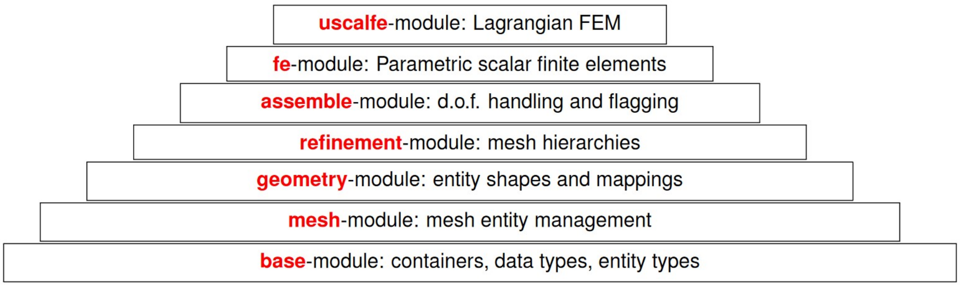


image taken from: Antonetti P., Houston P., Pennesi G. (2018) Fast Numerical Integration on Polytopic Meshes with Applications to Discontinuous Galerkin Finite Elemen Methods

$F_\mathcal{K}(\hat{x}) = J_\mathcal{K}\hat{x} + t_\mathcal{K}$ with $J_\mathcal{K} \in \mathbb{R}^{2\times 2}$ being the Jacobian of the transformation, $t \in \mathbb{R}^2$ the translation between $[0,0]^T$ and the baricenter of $B_\mathcal{K}$. $F_\mathcal{K}$ is affine, $J_\mathcal{K}$ therefore diagonal. $J_\mathcal{K} := \operatorname{diag}(h_1, h_2)$ with h_1, h_2 being half the lengths of $B_\mathcal{K}$'s sides.

In the following sections I will go over some of LehrFEM's modules which are depicted below. The order in which I go through them is in a bottom-up manner and the changes proposed such that no dependencies from higher to lower module are created. Existing functionalities are preserved, LehrFEM merely is extended.


uscalfe-module: Lagrangian FEM
fe-module: Parametric scalar finite elements
assemble-module: d.o.f. handling and flagging
refinement-module: mesh hierarchies
geometry-module: entity shapes and mappings
mesh-module: mesh entity management
base-module: containers, data types, entity types

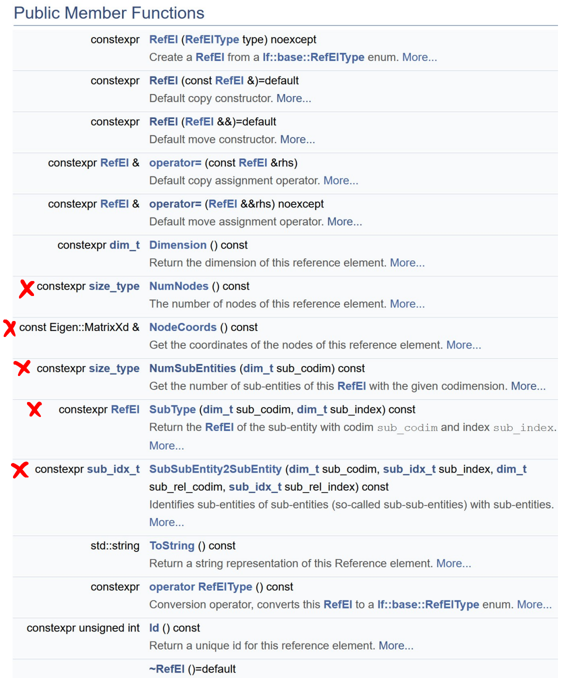
11 Base-RefEl

As can be seen above, there is no reference element for polygons. To comply with higher modules, which require an entity in the mesh to have a RefEl, a "disabled RefEl" for polygons is introduced. Most functionalities of the other reference elements are not provided, asserts will fail if the corresponding member function of the class is called.

First off, there is a new type of RefEl:

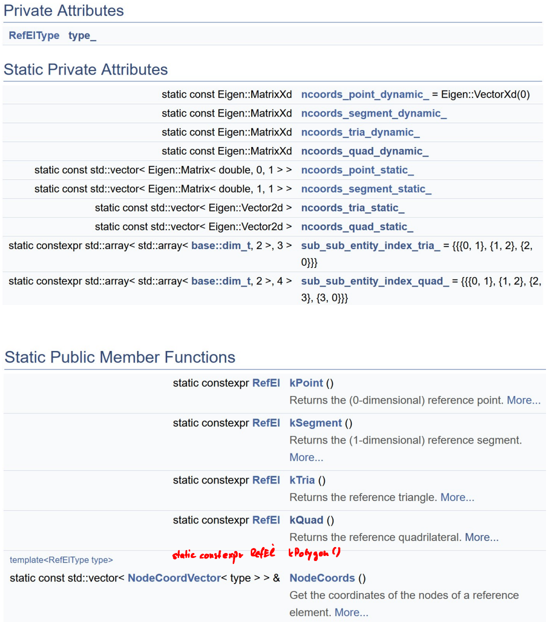
```
enum class RefElType : unsigned char {
        kPoint   = 1,
        kSegment  = 2,
        kTria   = 3,
        kQuad   = 4,
        kPolygon  = 3,
};
```

Here are screenshots of the functions and attributes of lf::base::RefEl:



Public Member Functions

constexpr	RefEl (RefElType type) noexcept Create a RefEl from a lf::base::RefElType enum. More...
constexpr	RefEl (const RefEl &)=default Default copy constructor. More...
constexpr	RefEl (RefEl &&)=default Default move constructor. More...
constexpr RefEl &	operator= (const RefEl &rhs) Default copy assignment operator. More...
constexpr RefEl &	operator= (RefEl &&rhs) noexcept Default move assignment operator. More...
constexpr dim_t	Dimension () const Return the dimension of this reference element. More...
✗ constexpr size_type	NumNodes () const The number of nodes of this reference element. More...
✗ const Eigen::MatrixXd &	NodeCoords () const Get the coordinates of the nodes of this reference element. More...
✗ constexpr size_type	NumSubEntities (dim_t sub_codim) const Get the number of sub-entities of this RefEl with the given codimension. More...
✗ constexpr RefEl	SubType (dim_t sub_codim, dim_t sub_index) const Return the RefEl of the sub-entity with codim `sub_codim` and index `sub_index`. More...
✗ constexpr sub_idx_t	SubSubEntity2SubEntity (dim_t sub_codim, sub_idx_t sub_index, dim_t sub_rel_codim, sub_idx_t sub_rel_index) const Identifies sub-entities of sub-entities (so-called sub-sub-entities) with sub-entities. More...
std::string	ToString () const Return a string representation of this Reference element. More...
constexpr	operator RefElType () const Conversion operator, converts this RefEl to a lf::base::RefElType enum. More...
constexpr unsigned int	Id () const Return a unique id for this reference element. More...
	~RefEl ()=default



Private Attributes

RefElType	type_

Static Private Attributes

static const Eigen::MatrixXd	ncoords_point_dynamic_ = Eigen::VectorXd(0)
static const Eigen::MatrixXd	ncoords_segment_dynamic_
static const Eigen::MatrixXd	ncoords_tria_dynamic_
static const Eigen::MatrixXd	ncoords_quad_dynamic_
static const std::vector< Eigen::Matrix< double, 0, 1 > >	ncoords_point_static_
static const std::vector< Eigen::Matrix< double, 1, 1 > >	ncoords_segment_static_
static const std::vector< Eigen::Vector2d >	ncoords_tria_static_
static const std::vector< Eigen::Vector2d >	ncoords_quad_static_
static constexpr std::array< std::array< base::dim_t, 2 >, 3 >	sub_sub_entity_index_tria_ = {{{0, 1}, {1, 2}, {2, 0}}}
static constexpr std::array< std::array< base::dim_t, 2 >, 4 >	sub_sub_entity_index_quad_ = {{{0, 1}, {1, 2}, {2, 3}, {3, 0}}}

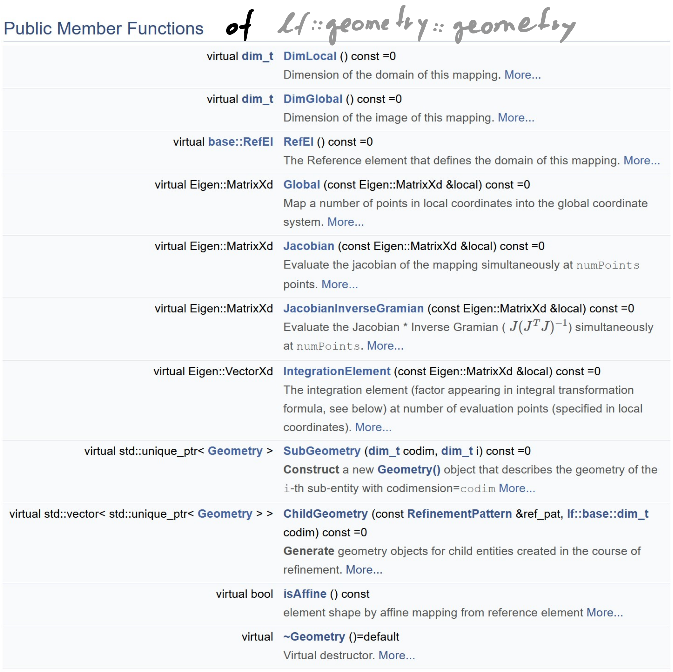
Static Public Member Functions

static constexpr RefEl	kPoint () Returns the (0-dimensional) reference point. More...
static constexpr RefEl	kSegment () Returns the (1-dimensional) reference segment. More...
static constexpr RefEl	kTria () Returns the reference triangle. More...
static constexpr RefEl	kQuad () Returns the reference quadrilateral. More...
static constexpr RefEl	kPolygon ()
template<RefElType type> static const std::vector< NodeCoordVector< type > > &	NodeCoords () Get the coordinates of the nodes of a reference element. More...

I decided to not even include the information on the number of nodes of the polytope in the reference element. The changes to the functions are therefore all very small. All functions that are constexpr will remain so. The ones marked with a ✗ will fail an assert if they are called upon a RefEl of type_ kPolygon. All other functions will be implemented for polygons accordingly.

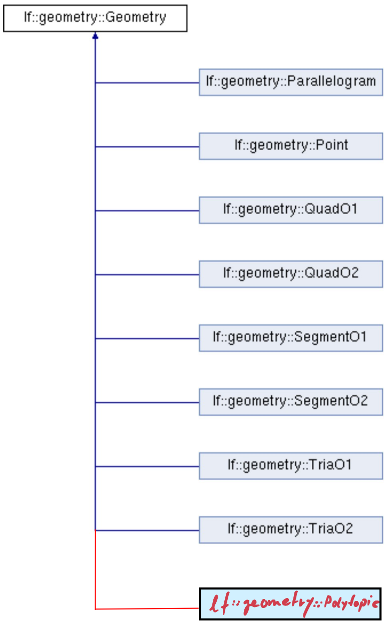
2 Geometry

The computations for the element matrices are done directly on the polygons. The mappings between $[-1,1]^2$ and the axis-aligned bounding boxes $\mathcal{B}_x$ of each element is trivial and will be computed on-the-fly when required. Therefore, the functionalities of the geometry module are not used in the polytopic mesh setting.
To comply with the requirements of lf::mesh::Entity , a polygon still has a pointer to a lf::geometry::Geometry object. This pointer could just be set to NULL. But I decided to create dummy geometry classes that just have asserts in every public function. This makes debugging easier, in case one nevertheless tries to call member functions of polytopic geometry objects.
The screenshot below shows all public member functions lf::geometry::Geometry interface. The added class (annotated screenshot on the right) will have them filled with asserts. All polytopic mesh entities will have a geometry pointer to a lf::geometry::Polytopic object.



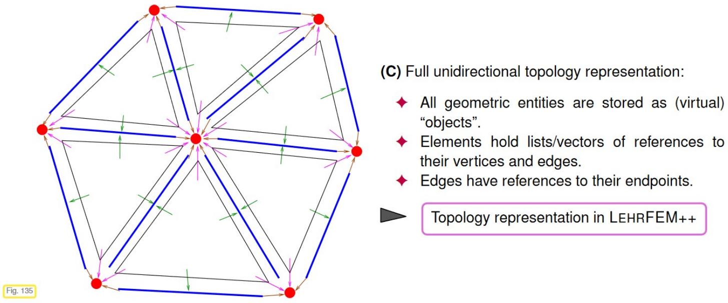
Public Member Functions of lf::geometry::geometry

virtual dim_t	DimLocal () const =0 Dimension of the domain of this mapping. More...
virtual dim_t	DimGlobal () const =0 Dimension of the image of this mapping. More...
virtual base::RefEl	RefEl () const =0 The Reference element that defines the domain of this mapping. More...
virtual Eigen::MatrixXd	Global (const Eigen::MatrixXd &local) const =0 Map a number of points in local coordinates into the global coordinate system. More...
virtual Eigen::MatrixXd	Jacobian (const Eigen::MatrixXd &local) const =0 Evaluate the jacobian of the mapping simultaneously at numPoints points. More...
virtual Eigen::MatrixXd	JacobianInverseGramian (const Eigen::MatrixXd &local) const =0 Evaluate the Jacobian * Inverse Gramian ($J(J^TJ)^{-1}$) simultaneously at numPoints. More...
virtual Eigen::VectorXd	IntegrationElement (const Eigen::MatrixXd &local) const =0 The integration element (factor appearing in integral transformation formula, see below) at number of evaluation points (specified in local coordinates). More...
virtual std::unique_ptr< Geometry >	SubGeometry (dim_t codim, dim_t i) const =0 Construct a new Geometry() object that describes the geometry of the i-th sub-entity with codimension=codim More...
virtual std::vector< std::unique_ptr< Geometry > >	ChildGeometry (const RefinementPattern &ref_pat, lf::base::dim_t codim) const =0 Generate geometry objects for child entities created in the course of refinement. More...
virtual bool	isAffine () const element shape by affine mapping from reference element More...
virtual	~Geometry ()=default Virtual destructor. More...



3 Mesh

LehrFEM++ comes with a full unidirectional topology representation as can be seen in this screenshot taken from the NumPDE lecture document:



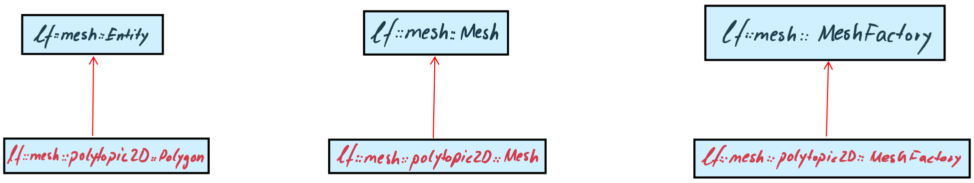
(C) Full unidirectional topology representation:

- All geometric entities are stored as (virtual) "objects".
- Elements hold lists/vectors of references to their vertices and edges.
- Edges have references to their endpoints.

▶ Topology representation in LEHRFEM++

The bilinear form corresponding to the symmetric interior penalty also features terms that are to be computed on segments and require knowledge of the adjacent polygons.
If the segments do not hold any data on the 2 polygons they belong to, finding the two would have a complexity of $\mathcal{O}(N)$, N being the number of polygons in the mesh.
To avoid this, during the initialization of the mesh, a lf::mesh::utils::CodimMeshDataSet is created that stores pointers to the two adjacent polygons (or only one in case of a boundary edge).

The new namespace lf::mesh::polytopic2D is introduced. It will feature 3 classes who are depicted below, together with the abstract base class they inherit from.


lf::mesh::Entity
lf::mesh::Mesh
lf::mesh::MeshFactory
lf::mesh::polytopic2D::Polygon
lf::mesh::polytopic2D::Mesh
lf::mesh::polytopic2D::MeshFactory

The lf::mesh::polytopic2D::Polygon will have the same functionalities and members as lf::mesh::hybrid2D::Triangle except that it stores its nodes and edges in vectors instead of arrays.
The Points and Segments of the mesh will be objects of the classes of the hybrid setting. They will have a "disabled RefEl" and a "dummy Geometry" as members though.
The new lf::mesh::polytopic2D::MeshFactory will initialize the polytopic mesh as well as the lf::mesh::utils::CodimMeshDataSet from above from a .gmsh file.

$\boxed{3}$ Shape Functions & LSE assembly

As proposed in the paper on politopic meshes integration on politopic meshes in the introduction, the family of one-dimensional and L^2-orthonormal Legendre polynomials defined over $L^2(-1,1)$ will be used as basis functions to span a standard polynomial space.

$$\mathcal{L}_n(x) = \frac{L_n(x)}{\|L_n\|_{L^2(-1,1)}} \quad , \quad \text{with} \quad L_n = \frac{1}{2^n n!}\frac{d}{dx}\left[(x^2-1)^n\right]$$

Then, basis functions for $\mathcal{P}_p(\hat{\mathcal{B}})$ are defined. I is a multi-index: $I = (i_1, i_2)$. The basis functions on $\hat{\mathcal{B}}$ are denoted by $\{\hat{\phi}_I\}_{0 \leq |I| \leq p}$ $(I = i_1 + i_2)$

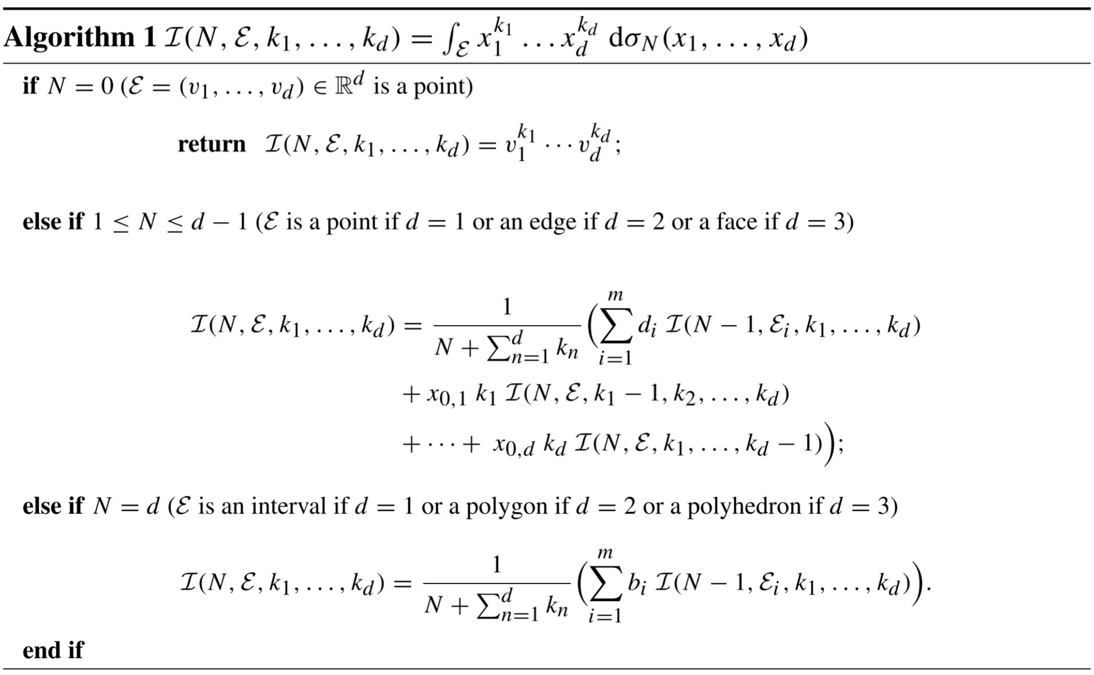
$$\hat{\phi}_I(\hat{x}) - \hat{\phi}_I(\hat{x}_1, \hat{x}_2) = \mathcal{L}_{i_1}(\hat{x}_1)\mathcal{L}_{i_2}(\hat{x}_2)$$

Then, the polynomial spaces on the bounding boxes $\mathcal{B}_\kappa$ is defined by

$$\phi_{I,\kappa}(x) = \hat{\phi}_I(F_\kappa^{-1}(x)) \quad \forall\, x \in \kappa \in \mathcal{B}_\kappa \qquad \forall I: 0 \leq |I| \leq p_\kappa$$

Finally $\{\phi_{I,\kappa} : 0 \leq |I \leq p_\kappa ,\ \kappa \in \overset{\text{set of polygons in the mesh}}{\mathcal{T}_h}\}$ is the basis of the discrete space V_h. On each element, there is a bijective relation between $\{I = (i_1, i_2)\}$ and $\{1, 2, \dots, \dim(\mathcal{P}_p(\kappa))\}$

In LehrFEM, functionalities regarding this basis will be collected in a new module lf::dgfe. It will also feature the integration of those basis functions over polytopes and segments in the mesh. The method used is the A1 from the paper of Antonetti P. Houston P. and Pennesi G. Here a screenshot of the pseudo code:



Algorithm 1 $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = \int_{\mathcal{E}} x_1^{k_1} \dots x_d^{k_d}\, \mathrm{d}\sigma_N(x_1, \dots, x_d)$

```
if $N = 0$ ($\mathcal{E} = (v_1, \dots, v_d) \in \mathbb{R}^d$ is a point)
        return $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = v_1^{k_1} \cdots v_d^{k_d}$;
else if $1 \leq N \leq d-1$ ($\mathcal{E}$ is a point if $d = 1$ or an edge if $d = 2$ or a face if $d = 3$)
        $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = \frac{1}{N + \sum_{n=1}^{d} k_n}\Big(\sum_{i=1}^{m} d_i\, \mathcal{I}(N-1, \mathcal{E}_i, k_1, \dots, k_d)$
                $+ x_{0,1}\, k_1\, \mathcal{I}(N, \mathcal{E}, k_1 - 1, k_2, \dots, k_d)$
                $+ \cdots + x_{0,d}\, k_d\, \mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d - 1)\Big)$;
else if $N = d$ ($\mathcal{E}$ is an interval if $d = 1$ or a polygon if $d = 2$ or a polyhedron if $d = 3$)
        $\mathcal{I}(N, \mathcal{E}, k_1, \dots, k_d) = \frac{1}{N + \sum_{n=1}^{d} k_n}\Big(\sum_{i=1}^{m} b_i\, \mathcal{I}(N-1, \mathcal{E}_i, k_1, \dots, k_d)\Big)$.
end if
```

image taken from: Antonetti P., Houston P., Pennesi G. (2018): Fast Numerical Integration on Polytopic Meshes with Applications to Discontinuous Galerkin Finite Element Methods

The lf::dgfe module will feature entity matrix/vector providers which can be used with lf::assemble routines to assemble the rhs & lhs of the Discontinuous Galerkin LSE of the boundary value problem

$$\boxed{\begin{array}{l} -\mathrm{div}(A(x)\,\mathrm{grad}(u)) + \mathrm{div}(b(x)\,u) + c(x)\,u = f \quad \text{in } \Omega \subset \mathbb{R}^2 \\ \qquad\qquad u = g \quad \text{on } \Gamma_D \cup \Gamma_- \\ \qquad\qquad A(x)\,\mathrm{grad}\,u \cdot n(x) = h \quad \text{on } \Gamma_N \end{array}}$$

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Implementation of Discontinuous Galerkin Finite Element Method on Polygonal Meshes

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Maurer	Tarzis

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 31.07.23	T.

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.