ETH zürich D MATH

Implementation of Efficient Finite Element Solvers in Dynamic Programming Languages

A case study in the Julia programming language

Till Ehrengruber

June 2017

Bachelor Thesis

Advisors: Prof. Dr. Ralf Hiptmair
Department of Mathematics, ETH Zürich

Contents

Acknowledgements

First of all I would like to thank my advisor Prof. Dr. Ralf Hiptmair for giving me the opportunity to develop this library as part of my bachelor thesis and for sharing some of his knowledge of the subject with me that was especially useful in the design decisions that go beyond the problem examined in this thesis. Furthermore, I want to thank my family for their support and guidance. I want to especially thank my brother Max for his patience with me during the hard times that are probably part of the formation process of every thesis. I also want to thank all of the Julia developers for their work and discussions on some of the issues that occurred during development and last but not least the users on the Julia IRC channel for the many fruitful discussions.

Abstract

In this thesis I present the implementation of the TipiFEM library, a framework for the implementation of finite element solvers in Julia. At the time of writing the library supports linear and quadratic Lagrangian finite elements on two dimensional domains discretized using triangles and quadrilaterals. Its design however allows extension to three dimensional domains and different finite elements. The different building blocks of the finite element solver are explained. This includes a simple yet general and efficient representation of computational meshes. A topology construction algorithm for meshes with topological dimension two is developed that can easily be extended to meshes with topological dimension three. Furthermore, datastructures for numerical integration and finite element spaces as well as computer representations of finite element bases are developed. The assembly procedure of the Galerkin matrix and right hand side vector is explained briefly. Then a second order elliptic boundary value problem with inhomogeneous Dirichlet and Neumann boundary conditions is solved on a hybrid mesh and its runtime is discussed. In the end the Julia programming language is evaluated by pointing out its strengths and weaknesses.

1

Introduction

Finite element methods are a numerical technique for finding approximate solutions to weak formulations of boundary value problems arising in various fields of engineering. Typically the sought solution is the distribution of some field variable like electrical charge, temperature or heat flux. Nowadays it has become a key technology used for computational modeling and simulation in engineering. A typical finite element software package consists of three modules

- **Mesh generator** (*Input*: CAD data, *Output*: Mesh file)
  Discretizes the domain on which the problem is formulated into small pieces.
- **Finite element solver** (*Input*: Parameters, Mesh file, *Output*: Degrees of freedom)
  Numerically solves the boundary value problem on the mesh and stores the solution.
- **Post-Processor / Visualizer** (*Input*: Galerkin Solution)

In this thesis we will concentrate on the finite element solver. We will develop the datastructures and algorithms required to solve a model problem general enough to build the foundation of an efficient framework applicable to a wider range of problems that can be solved using finite element methods. The model problem will be a second order elliptic boundary value problem with inhomogenous Dirichlet and Neumann boundary conditions posed on a two dimensional domain. Furthermore, we will discuss the implementation of such a framework in the Julia programming language and some aspects of Julia that played a role during the design process. Lastly a brief performance evaluation is carried out.

1.1. TipiFEM

The framework developed as part of this thesis is TipiFEM. It is freely available at `https://github.com/tehrengruber/TipiFEM` and distributed under the GNU Affero General Public License. TipiFEM is compatible with Julia v0.6.

Design principles & goals At various time points in the development of a software library design choices are being made that influence what a library is good at and what not. To guide the developer in such choices a set of principles and their relevance is very handy. In TipiFEM the following design principles, in order of descending relevance, have been choosen:

1. Extendability
2. Performance
3. Explicitness
4. Readability

Considering the title of this thesis the ordering of the first two principles might be surprising. However, it was decided that solving harder problems in a reasonable amount of time is more important to the library than solving easy problems in the shortest amount of time. A design goal of TipiFEM is to give mathematicans and programmers a common ground to rapidly develop new methods for the solution of boundary value problems[1] and while attributing performance higher relevance then extendibility the archieved performance is pointless if one can not implement complex problems for which extendibility is crucial because there already exists a dozen of performant libraries for simple problems.

1.2. The Choice of a Progamming Language

The idea of generating efficient machine code from high-level, generic code is probably as old as the first computers. It was then realized that writing machine code by hand is time comsuming and error prone. A major step towards realization of this idea had been made with the development of FORTRAN, C and C++, which remain the default choice for computationally intensive tasks. Although these three languages have been extended steadily since their first appereances in the mid 50s for FORTRAN, the early 70s for C and the early 80s for C++, it has to be recognized that today dynamic languages like MATLAB or Python enjoy high popularity in the scientific community, used for both research and teaching, due to their increased productivity, ability to closely resemble mathematical formulas and low language complexity. But since many of these languages were not designed with the goal of high performance in mind [2], computationally intensive tasks are often carried out by libraries written in a static language, accessed by means of a foreign function interface[3]. Due to this discrepancy in performance it is then often beneficial to write code in a form that uses operations implemented in a static language. For arithmetic operations this typically means rewriting for loops in terms of linear algebra

[1]Note that TipiFEM is not there yet.

[2]Matlab for example was developed initially to give students an easy way to interact with LINPACK and EISPACK

[3]A prominent example for such a library is NumPy http://www.numpy.org/

operations commonly referred to as *vectorization*[4]. Finite element matrix assembly however is naturally written in terms of a loop over mesh elements with rather short loop bodies (in terms of runtime). Dynamic languages tend to be slow in this regard, giving rise to the development of vectorized algorithms to save the property of efficienty [1]. Julia is different in this regard as its for loops can easily archieve the speed of equivalent loops in static languages like the ones given before. While this is nothing particulary new in computer science, Julia comes with a remarkable set of features that make it very attractive for the efficient implementation of numerical algorithms: Its syntax can easily resemble mathematical expressions with only minor differences in notation. Furthermore, the basic syntax is easy to read especially for programmers that are familar with Matlab. Most notably Julia also has features for metaprogramming, allowing sophisticated code generation by means of macros operating at the level of abstract syntax trees. Last but not least Julia has bindings to LAPACK and SuiteSparse for linear algebra operations.

1.3. Motivation

During a university course about numerical methods for partial differential equations I worked with DUNE[5], a modular toolbox for solving partial differential equations, and BETL[6], a framework with DUNE style interface, mainly concerned with providing a toolbox for Galerkin boundary element methods. Both libraries include tools for the implementation of finite element methods. This was my first contact with finite element methods and therefore implementation of first codes were done in a try and error fashion. During implementation I observed that I spent only a fraction of time dealing with the implementation of algorithms and was mostly waiting for compilation to finish and understanding how the library works internally. This distribution of time was not because the two frameworks were designed badly, but because of the programming workflow in C++ intrinsic to most static languages. Every fix in my code, no matter how small, required an additional step of compilation, significantly slowing down productivity. Furthermore, derivations on paper differed significantly from the implementation in the code, requiring frequent checks to verify that the implementation matched the derivation.

[4]The Matlab manual explicitly names performance as reason for vectorization see https://mathworks.com/help/matlab/matlab_prog/vectorization.html

[5]`http://dune-project.org/`

[6]`http://www.sam.math.ethz.ch/betl/`

2

Mathematical concepts

A rigourous discussion of finite element methods is far beyond the scope of this thesis. In this section only a set of definitions is given to introduce the reader to the notation, conventions, and theory used later on.

2.1. Cell

Definition 2.1 (Polytope) *A (convex) d-polytope is the convex hull of a finite set of points in* $\mathbb{R}^d$.

Definition 2.2 (Facet) *The facets of a d-polytope* P *are the polytope elements of* P *with dimension* $d-1$.

Definition 2.3 (Face) *The faces of a d-polytope* P *are the polytope elements of* P *with dimension less than d.*

Definition 2.4 (Skeleton) *The skeleton of a d-polytope* P *is the set of all polytope elements of* P *with dimension less or equal to d. Put differently the skeleton of* P *is* P *itself and its faces.*

Definition 2.5 (Cell) *A closed (resp. open) cell* $K \subset \mathbb{R}^n$ *is a bounded closed (resp. open) set with nonempty interior and piecewise smooth boundary. Therefore a polytope is a cell and its interior is an open cell.*

Definition 2.6 (Reference triangle) *The reference triangle is the triangle with vertices in* $\mathbb{R}^2$

$$\widehat{\boldsymbol{p}}_1 = (0,0), \quad \widehat{\boldsymbol{p}}_2 = (1,0), \quad \widehat{\boldsymbol{p}}_3 = (0,1)$$

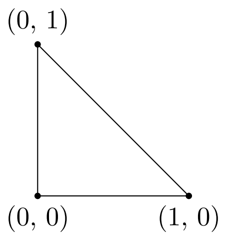
Definition 2.7 (Reference square) *The reference square is the square with vertices in* $\mathbb{R}^2$

$$\widehat{\boldsymbol{p}}_1 = (0,0), \quad \widehat{\boldsymbol{p}}_2 = (1,0), \quad \widehat{\boldsymbol{p}}_3 = (1,1), \quad \widehat{\boldsymbol{p}}_4 = (0,1)$$

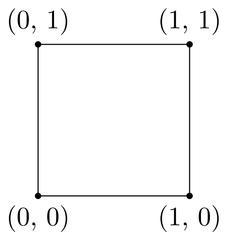
Definition 2.8 (Reference edge) *The reference edge is the edge with end points/vertices in* $\mathbb{R}$

$$\widehat{\boldsymbol{p}}_1 = 0, \quad \widehat{\boldsymbol{p}}_2 = 1$$

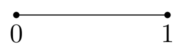
Put differently the reference edge is the interval $[0,1]$.



(a) The reference triangle



(b) The reference quadrilateral



(c) The reference edge

2.2. Mesh

In general the term mesh describes a subdivision of a bounded set $\Omega \subset \mathbb{R}^D$ into open cells of dimension less than or equal D. Here we stick to a more specific definition of a mesh with topological dimension two.

Definition 2.9 (Mesh) *We say that $\mathcal{M} = \mathcal{V} \cup \mathcal{E} \cup \mathcal{T}$ is a mesh if the following properties hold*

1. *$\mathcal{T}$ is a finite set of flat, bounded, open and non-degenerate triangles and quadrangles, the mesh elements.*
2. *$\mathcal{E}$, respectively $\mathcal{V}$, is the set of all facets, respectively vertices, of elements in $\mathcal{T}$.*
3. *$\overline{\Omega}$ is covered by $\mathcal{M}$*
4. *For $T_i, T_j \in \mathcal{T}$ with $i \neq j$ the intersection $\overline{T_i} \cap \overline{T_j}$ is either empty, a node, or an edge of both T_i and T_j.*

Definition 2.10 (Incidency) *We say that a cell is incident to another cell if one cell is either a face to the other cell or if the intersection of the two cells is a facet.*

Definition 2.11 (Adjacency / Neighbourhood) *We say that a cell is adjacent to another cell or is a neighbour to another cell if both cells share a common facet.*

2.3. Polynomials

Definition 2.12 (Multivariate polynomials) *The space of univariate polynomials of degree $p \in \mathbb{Z}^{\geq 0}$*

$$\mathcal{P}_p(\mathbb{R}) = \{x \mapsto \sum_{0 \leq \alpha \leq p} c_\alpha x^\alpha, c_\alpha \in \mathbb{R}\}$$

The space of 2-variate polynomials of degree $p \in \mathbb{Z}^{\geq 0}$

$$\mathcal{P}_p(\mathbb{R}^2) := \left\{ (x_1, x_2) \in \mathbb{R}^2 \mapsto \sum_{\substack{\alpha_1,\alpha_2 \geq 0, \\ \alpha_1+\alpha_2 \leq p}} c_{\alpha_1,\alpha_2} x_1^{\alpha_1} x_2^{\alpha_2},\ c_{\alpha_1,\alpha_2} \in \mathbb{R} \right\}$$

Definition 2.13 (Tensor product polynomials) *The space of tensor product polynomials of degree $p \in \mathbb{Z}^{\geq 1}$*

$$\mathcal{Q}_p(\mathbb{R}^2) := \{(x_1, x_2) \mapsto p_1(x_1)p_2(x_2),\ p_1 \in \mathcal{P}_p(\mathbb{R}),\ p_2 \in \mathcal{P}_p(\mathbb{R})\}$$

2.4. Finite Element

Definition 2.14 (Finite Element) *A finite element is a triplet (K, P, Σ) such that*

- *K is a cell*
- *P is a s-dimensional vector space of functions $p : K \to \mathbb{R}$ where $s \in \mathbb{Z}^{\geq 1}$*
- *$\Sigma = (\sigma_1, \sigma_2, \ldots, \sigma_s)$ is an ordered basis of the dual space to P. The elements $\sigma_1, \sigma_2, \ldots, \sigma_s$ are called local degrees of freedom and their indices are called local degree of freedom indices.*

The elements $b_1, \ldots, b_s \in P$ with $\sigma_i(b_j) = \delta_{ij}$ are called local shape functions or nodal basis.

Definition 2.15 (Internal degree of freedom) *An internal degree of freedom is a degree of freedom that is associated with the interior of its finite element.*

Definition 2.16 (Boundary degree of freedom) *A boundary degree of freedom is a degree of freedom that is not an internal degree of freedom.*

Definition 2.17 (Lagrange element) *A Langrange element is a finite element (K, P, Σ) such that, given a set of points $a_1, \ldots, a_s \in K$, the i-th degree of freedom is defined as $\sigma_i : P \to \mathbb{R}, p \mapsto p(a_i)$. Then by definition $\sigma_i(b_j) = b_j(a_i) = \delta_{ij}$. The points a_i are called local interpolation points and i is the local interpolation node index.*

Note that for the specification of the nodal basis of a Lagrange element only the interpolation points need to be specified. Therefore only the interpolation nodes for different Lagrange elements are given later on.

Definition 2.18 (Associated cell) *Given an interpolation node $a \in K$ the cell associated with a is the open cell in the skeleton of K that contains a. Since every degree of freedom of a Lagrange element is associated naturally to an interpolation node, every interpolation node and then also every local shape function is associated with a cell.*

Definition 2.19 (Cell type constrained interpolation node index) *Let $A = \{a_1, \dots, a_s\}$ be an ordered set of interpolation nodes. If one removes all interpolation nodes that are not associated to a cell of type $[C]$*[7]*, one obtains a new ordered set A_C. The cell type constrained interpolation node index of an interpolation node $\boldsymbol{a}_i$ is then the index of $\boldsymbol{a}_i$ in A_C.*

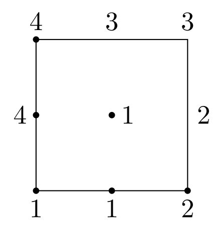


Figure 2.2: Cell type constrained interpolation node indices of the $\mathbb{Q}_2(K)$ element

Definition 2.20 ($\mathbb{P}_p$ element) *A $\mathbb{P}_p(K)$ element is a Lagrange element (K, P, Σ) such that*

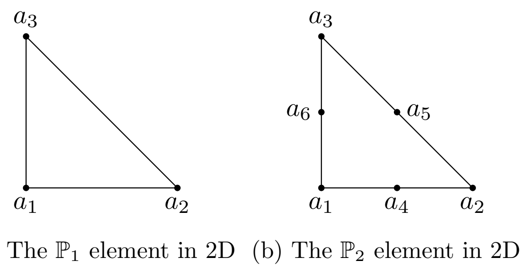
- *K is a simplex (e.g. a line segment or a triangle)*
- *$P = \mathcal{P}_p(K)$*

We omit the general specification of the interpolation nodes of the $\mathbb{P}_p(K)$ element here and specify them only for K being either a triangle T or an edge E.

Definition 2.21 (Triangle element) *The triangle element is a $\mathbb{P}_p(K)$ element where K is a triangle with vertices $v_1, v_2, v_3 \in \mathbb{R}^2$.*

Its interpolation nodes are

- *$p = 1$:*
$$a_1 = v_1, \quad a_2 = v_2, \quad a_3 = v_3$$
- *$p = 2$:*
$$\begin{aligned} &a_1 = v_1, \quad a_2 = v_2, \quad a_3 = v_3, \\ &a_4 = \frac{v_1 + v_2}{2}, \quad a_5 = \frac{v_2 + v_3}{2}, \quad a_6 = \frac{v_3 + v_1}{2} \end{aligned}$$



(a) The $\mathbb{P}_1$ element in 2D (b) The $\mathbb{P}_2$ element in 2D

[7]This notation is inspired by the standard notation of equivalence classes. Later on we will just specify the cell type by saying that C is for example a triangle.

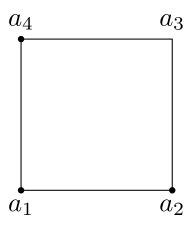
Definition 2.22 (Edge element) *The edge element is a $\mathbb{P}_p(K)$ element where K is an edge with endpoints $v_1, v_2 \in \mathbb{R}$.*

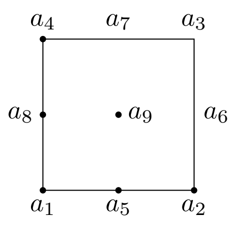
Its interpolation nodes are

- *$p = 1$:*
$$a_1 = v_1, \quad a_2 = v_2$$
- *$p = 2$:*
$$a_1 = v_1, \quad a_2 = v_2, \quad a_3 = \frac{v_1 + v_2}{2}$$

Definition 2.23 ($\mathbb{Q}_p$ element) *The $\mathbb{Q}_p(K)$ element is a Lagrange element such that*

- *K is parallelotop (e.g. square)*
- *$P = \mathcal{Q}_p(K)$*





(a) The $\mathbb{P}_1$ element in 2D (b) The $\mathbb{P}_2$ element in 2D

2.5. Function Spaces

Definition 2.24 (Sobolev space $H^1(\Omega)$) *The space of integrable functions on Ω with square integrable gradient*

$$H^1(\Omega) = \{v : \Omega \to \mathbb{R} \text{ integrable } : \int_\Omega |\, \mathit{grad}\, v(\boldsymbol{x})|^2 \mathrm{d}\boldsymbol{x} < \infty\}$$

Definition 2.25 (Lagrangian finite element space) *The space of p-th degree Lagrangian finite element functions on $\mathcal{M}$*

$$\mathcal{S}_p^0(\mathcal{M}) := \left\{ v \in C^0(\overline{\Omega}) : v_{|K} \in P(K) \quad \forall K \in \mathcal{T} \right\}$$

where P is the space of local shape functions of the finite element of K.

2.6. Variational Problem

Definition 2.26 (Linear variational problem) *A variational problem posed on an affine space V and a vector space V_0 of the form*

$$u \in V : a(u, v) = l(v) \qquad \forall v \in V_0$$

is called a linear variational problem if $a : V \times V_o \to \mathbb{R}$ is a bilinear form and $l : V_0 \to \mathbb{R}$ is a linear form.

Definition 2.27 (Discrete variational problem) *Given a continuous variational problem the corresponding discrete variational problem is obtained by choosing a finite dimensional trial space V_N and test space $V_{N,0}$, where N is the dimension of the two spaces. The discrete variational problem then reads:*

Find $u_N \in V_N$ such that:

$$a(u_N, v_N) = l(v_N) \qquad \forall v_N \in V_{0,N}$$

2.6.1. Solution of a Linear Discrete Variational Problem

The Galerkin solution u_N of a linear discrete variational problem is obtained by

1. Introduction of an (ordered) basis $\mathfrak{B}_N = \{b_N^1, \dots, b_N^N\} \subset V_{N,0}$ for the trial and test space.
2. Solution of the linear system of equations $A\mu = \varphi$ where

$$\boldsymbol{A} = \left(a\left(b_N^k, b_N^j\right)\right)_{j,k=1}^N \in \mathbb{R}^{N,N} \qquad \text{(Galerkin matrix)}$$

$$\boldsymbol{\varphi} = \left(l\left(b_N^j\right)\right)_{j=1}^N \in \mathbb{R}^N \qquad \text{(Right hand side vector)}$$

$$\boldsymbol{\mu} = (\mu_1, \dots, \mu_N)^T \in \mathbb{R}^N \qquad \text{(Coefficient vector)}$$

   The functions b_N^i are called the global basis functions or global shape functions, the matrix A Galerkin matrix, φ right hand side vector and μ coefficient vector. The elements μ_i of $\boldsymbol{\mu}$ are the global degrees of freedom and the index i is the global degree of freedom index.

3. Recovery of the solution by $u_N = \sum_{k=1}^N \mu_k b_N^k$

To keep the discussion focused, we use the same basis for both spaces V_N and $V_{N,0}$.

3

Model problem

Before we start with description of the finite element solver we define a simple boundary value problem which we use to develop all building blocks of a finite element method. The problem is a second order elliptic boundary value problem with Dirichlet and Neumann boundary conditions posed on a bounded and connected domain $\Omega \subset \mathbb{R}^2$. Let $u : \Omega \to \mathbb{R}$, $g : \Gamma_D \to \mathbb{R}$, $h : \Gamma_N \to \mathbb{R}$ be scalar valued functions, where Γ_D and Γ_N are disjoint subsets of Ω such that $\partial\Omega = \Gamma_D \cup \Gamma_N$. The function g specifies Dirichlet data and h specifies Neumann data.

Strong formulation
Find u such that:

$$\begin{aligned} -\operatorname{div}(\sigma(x)\operatorname{grad} u) &= f && \text{in } \Omega \\ u &= g && \text{on } \Gamma_D \\ (\sigma(x)\operatorname{grad} u)\cdot n &= h && \text{on } \Gamma_N \end{aligned}$$

Weak formulation
Find $u \in V = \{v \in H^1(\Omega) \mid u_{|\Gamma_D} \equiv g\}$ with $V_0 = \{v \in H^1(\Omega) \bigm| v_{|\Gamma_D} \equiv 0\}$ such that

$$a(u,v) = l(v) \qquad \forall v \in V_0$$

where

$$\begin{aligned} a(u,v) &= \int_\Omega \sigma(x)\operatorname{grad} u \cdot \operatorname{grad} v \,\mathrm{d}x \\ l(v) &= \int_\Omega f v \,\mathrm{d}x + \int_{\partial\Omega} ((\sigma(x)\operatorname{grad} u)\cdot n)\; v \,\mathrm{d}S(x) \\ &= \int_\Omega f v \,\mathrm{d}x + \int_{\Gamma_N} h v \,\mathrm{d}S(x) \end{aligned}$$

4

Mesh datastructure

In this section we will develop a datastructure to represent a mesh in computer memory. A suitable datastructure must be both space efficient and time efficient in the sense that a low amount of memory is required to store the datastructure and that the operations on the mesh are fast. The following operations are of particular importance:

1. Cell traversal
2. Retrieval of topological information (incidence relationships between elements, faces and facets)
3. Retrieval of geometrical information (location, shape)

4.1. Concepts

In TipiFEM a representation inspired by the one given in [2] has been chosen and is based on the basic concepts of a cell identifier, cell geometry, cell connectivity, mesh function, mesh connectivity and mesh topology. The concepts themselves are generic in the sense that they can be applied to very different types of meshes.

For the sake of concreteness we give an example of a simple hybrid mesh.

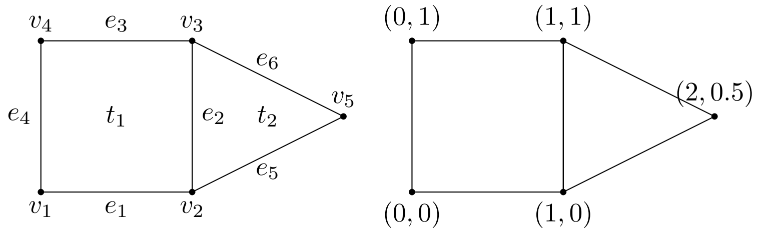


Figure 4.1: Simple hybrid mesh (Topological information on the left, geometrical information on the right)

Cell type id A *cell type id* or *cell type identifier* is an integer specifying the type of a cell. Which cell type id belongs to which cell type may be chosen arbitrarily. For the sake of simplicity however the choice made in the Gmsh[8] file format was taken[9].

ctid	Cell type
15	Vertex
1	Edge
2	Triangle
3	Quadrilateral

Figure 4.2: Selected cell types and their cell type identifier

Cell index A *cell index* is an integer unique within the set of all cells of equal cell type.

Cell id A *cell id* or *cell identifier* is a pair $(ctid, i)$ where $ctid$ is a cell type id (i.e. triangle, edge, vertex) and i is the cell index. Since a cell id uniquely identifies a cell, the two terms are often used synonymously later on even though the notion of cell id is just about identification while the notion of cell is also about geometry and connectivity. We denote the cell id of a cell $K \subset \mathcal{M}$ by $id(K)$.

The mesh given in Figure 4.1 contains 13 cells. Considering only the edges with $ctid$ equal to one the cell indices are: $\{(1,1),(1,2),(1,3),(1,4)\}$

Cell Geometry A *cell geometry* is a matrix of dimension $n_\mathcal{V} \times d$ associated to a single cell, where $n_\mathcal{V}$ is the number of vertices of the cell and d the dimension of the ambient space. The rows of the matrix contain the coordinates of the vertices of the cell in counter clock wise ordering.

The cell geometry of the quadrangle t_1 from Figure 4.1 is:

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Cell Connectivity A *cell connectivity* is a tuple $(ctid_1 \to ctid_2, (i_1, \dots, i_n))$ containing the cell indices $i_1, \dots, i_n$ of a cell with cell type id $ctid_2$ incident to a single cell with cell type id $ctid_1$. The cell to which a cell connectivity object belongs to is called the incidenter and the incident cells are called the incidentees. Here $ctid_1$ is the cell type id of the incidenter, or the incidenter type and $ctid_2$ is the cell type id of the incidentees, or the incidentee type. The indices $i_1, \dots, i_n$ in combination with the incidentee type give us the cell identifiers of all incident cells. Note that since the incident cells are specified by a tuple, we can talk about the orientation of a cell. Therefore for example an edge has a source and a sink.

Let $ctid_1$ be the incidenter type and $ctid_2$ the incidentee type of a cell connectivity then we call this a connectivity from $ctid_1$ to $ctid_2$. If the incidente type is omitted we implicitly mean that $ctid_2$ is a vertex.

[8]See `http://gmsh.info/doc/texinfo/gmsh.html#MSH-ASCII-file-format`
[9]Note that in Gmsh a cell is called a geometrical entity

The cell connectivity of the quadrangle t_1 and its edges from Figure 4.1 is $(3 \to 1, (1, 2, 3, 4))$.

Definition 4.1 *A canonical form of a cell connectivity with its vertices is another cell connectivity such that, given two cell connectivities with different orientation, the canonical form of the two is equal.*

The choice of a canonical form is arbitrary. For an edge its canonical form is the connectivity itself if the vertex index of the first vertex is larger than the second and otherwise the connectivity of the edge with opposite orientation.

Mesh function The basic building block to store cell geometry and connectivity objects is a mesh function that maps cells, respectively their cell identifiers to arbitrary objects. In mathematical notation a mesh function may be written as some function $f_{\mathcal{M}} : X \to Y$ where $X \subseteq \{id(K) \mid K \in \mathcal{M}\}$.

We define a set of operations on mesh functions that may be written in mathematical notation:

- domain($f_{\mathcal{M}}$) $= X$
- image($f_{\mathcal{M}}$) $= Y$
- graph($f_{\mathcal{M}}$) $= \{(x, f_{\mathcal{M}}(x)) \mid x \in X\}$
- push($f_{\mathcal{M}}, i, v$) $= g_{\mathcal{M}}$ with graph($g_{\mathcal{M}}$) $=$ graph($f_{\mathcal{M}}$) $\cup \{(i, v)\}$
- length($f_{\mathcal{M}}$) $= |X|$

A real valued mesh function may assign material parameters to mesh elements. Another boolean valued mesh function may describe which cells of a mesh are part of the mesh boundary.

Mesh connectivity A *mesh connectivity* $c_{\mathcal{M},i \to j}$ is a mesh function mapping i-dimensional cells to their cell connectivity which contains j-dimensional cells. This is just another form of writing down the incidence relation between two sets of cells, but in a way that given a cell id allows efficient retrieval of its incident cells if implemented correctly. Since the incidence relation is symmetric for every mesh connectivity $c_{\mathcal{M},i \to j}$, there is an equivalent mesh connectivity $c_{\mathcal{M},j \to i}$ differing only in representation.

Considering a mesh connectivity mapping i-dimensional cells to j-dimensional cells we can distinguish the following three cases:

- $i > j$: A map from i-dimensional cells to their j-dimensional faces.
- $i = j$: A map from i-dimensional cells to their neighboring cells (i.e. cells with a common facet).
- $i < j$: A map from i-dimensional cells to j-dimensional cells, which contain the i-dimensional cell.

It is worth noting that the mesh connectivity mapping codimension zero cells to dimension zero cells, i.e. vertices, already provides complete topological information from which all other mesh connectivities may be reconstructed. Furthermore, from the definition of incidency it follows that there is no mesh connectivity $c_{\mathcal{M},0\to 0}$ because vertices have no facets.

Mesh geometry A *mesh geometry* is a mesh function $g_{\mathcal{M},i}$ mapping i-dimensional cells to their cell geometry.

The mesh geometry of the mesh given in Figure 4.1 is:

cell id	cell geometry	commentary
$(3,1)$	$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$	Cell geometry of the quadrangle t_1
$(2,1)$	$\begin{pmatrix} 1 & 0 \\ 2 & 0.5 \\ 1 & 1 \end{pmatrix}$	Cell geometry of the triangle t_2

Mesh topology The *mesh topology* $\mathcal{I}$ contains the mesh connectivities of a mesh. It is convenient to write the mesh topology in matrix form $(\mathcal{I})_{ij} = c_{\mathcal{M},i\to j}$. The matrix contains a maximum of $(D+1)^2 - 1$ connectivities where D is the topological dimension of the mesh. For a fully bidirectional topology representation of a mesh with topological dimension two the matrix is as follows:

$$\mathcal{I} = \begin{bmatrix} - & c_{\mathcal{M},0\to 1} & c_{\mathcal{M};0\to 2} \\ c_{\mathcal{M},1\to 0} & c_{\mathcal{M},1\to 1} & c_{\mathcal{M};1\to 2} \\ c_{\mathcal{M},2\to 0} & c_{\mathcal{M},2\to 1} & c_{\mathcal{M};2\to 2} \end{bmatrix}$$

More restricted representations can be noted simple by dropping some connectivities. In a fully unidirectional representation the upper triangular part of the matrix is empty, while all other entries being set. In a restricted bidirectional representation only the lower triangular part and the entry (D,D) are set.

4.2. (Data)types and Datastructures

Except for the cell type id, which is just an integer, each of the concepts given in the previous section is represented by a type and potentially subtypes of that type equipped with a set of methods in the `TipiFEM.Meshes` module. This module however does not specify the properties of the mesh cells. From now on we will call this the abstract mesh implementation, while a concrete mesh implementation also specifies the properties of the mesh cells. The `TipiFEM.PolytopalMesh` module contains such a concrete mesh implementation for a mesh with polytopal cells, leveraging the types and methods of the `TipiFEM.Meshes` module. We begin with the discussion of how each concept corresponds to a datatype in the abstract mesh implementation.

4.2.1. Abstract Mesh Implementation

`Cell` The cell type is represented by a singleton type[10] that subtypes the abstract type `Cell`. The `Cell` type is part of the abstract mesh implementation while the singleton type is declared in the concrete mesh implementation. The cell type identifier may be accessed by calling `cell_type_id(T)` with `T` a subtype of `Cell`.

`Id` The primitive type[10] `Id{T}` represents a cell type identifier. It stores only the index of the cell, while the cell type `T` is encoded as a type parameter. In order to avoid writing the type `T` and potentially the cell type id parameter of `T` explicitly, the concrete mesh implementation may contain a string macro[11] such that, in the case of a polytopal mesh, the cell id `Id{Polytope{15}}(1)`, in mathematical notation $(15, 1)$, may be written as `Id"1-node point"(1)`.

`Geometry` The immutable composite type[10] `Geometry{T, world_dim, T}`, where `T` is a subtype of `Cell`, `world_dim` is the dimension of the ambient space as an integer, and `T` is a subtype of `Real` (i.e. `Float64` if the coordinates are stored in double precision), represents the cell geometry. It is a subtype of `StaticMatrix` from the `StaticArrays` package and therefore behaves like a statically sized matrix. An SIMD[12] friendly storage layout was choosen where the coordinates of one dimension are contiguous in memory or, put differently, the coordinates of one vertex are stored in a row of the matrix.

```
Geometry{Triangle, 2, Float64}((0, 0), (1, 0), (0, 1))
```

Figure 4.3: Construction of the `Geometry` of the reference triangle

`Connectivity` The immutable composite type[10] `Connecitvity{T1, T2}`, where `T1` and `T2` are subtypes of `Cell` represent a cell connectivity. It is a subtype of `StaticVector`, therefore behaving like a statically sized vector.

```
Connectivity"4-node quadrangle → 1-node point"(1, 2, 3, 4)
```

Figure 4.4: Construction of the `Connectivity` of the quadrilateral t_2 from Figure 4.1

`MeshFunction` Conceptionally all mesh functions can be written using the same mathematical notation. A representation in memory suitable for the design of efficient algorithms however depends on the properties of the input values,

[10]See `https://docs.julialang.org/en/latest/manual/types.html`

[11]See `https://docs.julialang.org/en/latest/manual/metaprogramming/#non-standard-string-literals`

[12]Single instruction, multiple data

i.e. the cell identifiers element of the domain. Therefore the following three datatypes, each of which has its own datastructure, were designed:

- `GenericMeshFunction`: The cell type of all input values is not known at the time of construction. Input values of different cell type can be stored in arbitrary order.
- `HomogeneousMeshFunction`: All input values are of equal cell type known at the time of construction.
- `HeterogeneousMeshFunction`: The cell types of all input values are known at the time of construction and multiple cell types are allowed. Input values of the same cell type are packed together in memory.

The abstract type[10] `MeshFunction{K, V}` is the supertype of all these datatypes, where `I` is the cell type of the cell ids in the domain of the mesh function and `V` the type of the values in the domain of the mesh function. Most functionality is implemented in a generic fashion and inherited by subtyping from `MeshFunction` and implementing the `domain`, `image`, `getindex`, `setindex` operations. Which of the three datatypes `GenericMeshFunction`, `HomogeneousMeshFunction` and `HeterogeneousMeshFunction` should be used can be decided if the type parameters `K` and `V` are known. Therefore the type `MeshFunction` has a constructor that, given `K` and `V`, returns an empty mesh function of appropiate type.

```
# construct an empty mesh function mapping polytopes to integers
generic_mf = MeshFunction(Polytope, Int)
# construct an empty mesh function mapping triangles and
#  quadrangles to integers
heterogeneous_mf = MeshFunction(Union{Polytope"3-node triangle",
  Polytope"4-node quadrangle"}, Int)
# construct an empty mesh function mapping triangles to integers
homogenous_mf = MeshFunction(Polytope"3-node triangle", Int)
```

Figure 4.5: Construction of empty generic-, heterogenous-, homogenous-mesh functions.

`GenericMeshFunction` This type is the most flexible one because no information on the cell type of the input values is required at the time of construction. Initially this was used for loading a mesh from file. Now a `HeterogeneousMeshFunction` is used for this purpose and the allowed cell types are obtained by querying the concrete mesh implementation for all admissible cells. The type however is still present because there exist some narrow cases where it might be useful.

`HomogeneousMeshFunction{K, V, II, VI}` This type can only represent mesh functions mapping cell ids of a single cell type `K`. The type parameter `V` is the element type of the image, `II` the type of the domain iterator and `VI` the type of the image iterator. The type has two fields: `ids`, the domain iterator and `values`, the image iterator. Let `id` be a cell id in the domain of the mesh function stored in the `i`-th element of the domain iterator then its value is

the `i`-th element of the value iterator. From now on we assume that `II` and `VI` are random access iterators because it greatly simplifies runtime complexity considerations.
The `HomogeneousMeshFunction` supports four types of domain iterators:

- `OneTo`: The domain is a range of ids from `Id{K}(1)` to `Id{K}(stop)`, where `stop` is an integer equal to the number of elements in the domain.
- `UnitRange`: The domain is a range of ids from `Id{K}(start)` to `Id{K}(stop)`, where `start` and `stop` are integers.
- `StepRange`: The domain is a range of ids from `Id{K}(start)` to `Id{K}(stop)` in steps of length `step`, where `start`, `stop` and `step` are integers.
- `AbstractArray`: The domain is an array of cell ids.

Depending on the type of the domain iterator the memory requirements of the domain and runtime of the `getindex` operation differ. In case the type of the domain iterator is either `OneTo`, `UnitRange`, or `StepRange` only a constant amount of memory is required. Furthermore, the runtime of the `getindex` operation is constant because the index in the image iterator of the value that belongs to an element in the domain can be calculated. For example in case of `OneTo` the value belonging to the cell id `Id{K}(i)` is the `i`-th value of the image iterator. In case the type of the domain is a `Vector`, which is a subtype of `AbstractArray`, each cell id is stored explicitly therefore requiring memory space proportional to the length of the mesh function.

```
# construct a homogeneous mesh function
#  an empty domain iterator of type OneTo and image iterator
#  of type Vector are used by default
mf = MeshFunction(Polytope"1-node point", Int)
# add some values. cell ids are allocated automatically
push!(mf, 2)
push!(mf, 3)
push!(mf, 4)
# print the mesh function
# output:
# 3 element HomogenousMeshFunction Polytope"1-node point" → Int64
#  (1, 2)
#  (2, 3)
#  (3, 4)
display(mf)
# access the value belonging to the Id"1-node point"(1)
#  returns 2
mf[Id"1-node point"(1)]
# iterate over all values and print their values
for v in mf
  println(v)
end
# iterate over all indices and values and print their values
for (cid, v) in graph(mf)
  println(cid, " ", v)
end
```

Figure 4.6: Operations on a homogeneous mesh function with domain iterator of type `OneTo` and image iterator of type `Vector` mapping vertices to integers.

`HeterogeneousMeshFunction{K, V, II, VI}` This type can represent mesh functions mapping cell ids of multiple cell types. The type is essentially just a wrapper around multiple homogeneous mesh functions. Additionally to the methods defined on all mesh functions, methods are provided to return the wrapped mesh functions for a particular cell type.

```
# construct a hybrid mesh function
heterogeneous_mf = MeshFunction(Union{Polytope"3-node triangle",
  Polytope"4-node quadrangle"}, Int)
# add some values
# ...
# extract the homogeneous mesh function mapping from triangles
#  to integers
homogeneous_mf[Polytope"3-node triangle"]
```

Figure 4.7: Extraction of a homogeneous mesh function from a hybrid mesh function

Note that one should not use `for` loops over mesh functions because they are not type stable[13]. Instead one should use for example the `foreach` function.

`MeshTopology` The `MeshTopology` type represents the mesh topology. The type has two fields. One is a tuple storing the mesh connectivities and the other one is a tuple of booleans storing which mesh connectivities have been constructed.

`Mesh` Given the datastructures above the datastructure representing a mesh is quite simple. The `Mesh` type represents a mesh. The type has the following three type parameters:

- `K`: A subtype of `Cell` representing the cell types of codimension zero cells. For a hybrid mesh `K` is a union type.
- `world_dim`: The dimension of the ambient space.
- `REAL_`: A subtype of `Real` representing real numbers.

Furthermore, the `Mesh` type has the following three fields:

- `vertices`: A mesh function mapping vertex ids to their coordinates.
- `topology`: The mesh topology of the mesh.
- `cell_groups`: An associative array that maps cell tags to the cell ids with that tag.

4.2.2. Concrete Mesh Implementation

TipiFEM currently only contains a single concrete mesh implementation for polytopal cells. The implementation is contained in the `TipiFEM.PolytopalMesh` module. It contains functions specifying the dimension of all supported polytopes, their number of vertices and rules to construct the facet connectivity and geometry. Furthermore functionality to access reference cells and evaluate coordinate transformations mapping coordinates on the reference cells to cells, as well as their Jacobians, is provided.

Code examples

Below are some code examples to demonstrate mesh construction and retrieval of topological and geometrical information.

[13]see `https://docs.julialang.org/en/stable/manual/performance-tips/#write-type-stable-functions`

```
# construct mesh instance
mesh = Mesh(Union{Polytope"3-node triangle",
                  Polytope"4-node quadrangle"})
# add some vertices
add_vertex!(mesh, 0, 0)
add_vertex!(mesh, 0, 1)
add_vertex!(mesh, 1, 1)
add_vertex!(mesh, 1, 0)
add_vertex!(mesh, 3, 0.5)

# query the number of cells
number_of_cells(mesh, Polytope"1-node point") == 5

# connect vertices
add_cell!(mesh, Polytope"4-node quadrangle", 1, 2, 3, 4)
add_cell!(mesh, Polytope"3-node triangle", 2, 5, 3)

# populate topology (restricted bidirectional by default)
populate_connectivity!(mesh)

# extract mesh that contains only the boundary
boundary(mesh)
```

Figure 4.8: Construction of the hybrid mesh in 4.1

```
using Gmsh
mesh = Gmsh.load(filename, mesh_dim=2, world_dim=2)
```

Figure 4.9: Loading a Gmsh file from disk

```
# print Geometry instances of all line segments
for geo in geometry(mesh, Polytope"2-node line")
  display(geo)
end
# print the Geometry of all facets of codimension zero cells
for geo in geometry(mesh)
  display(facets(geo))
end
# calculate the volume / area of a mesh
mapreduce(volume, +, geometry(mesh))
```

Figure 4.10: Access to geometrical information

```
# print the cell connectivity of all codimension zero cells
#  with their facets
for conn in connectivity(mesh, Codim{0}(), Codim{1}())
  display(conn)
end
```

Figure 4.11: Access to topological information

```
# retrieve the geometry of the triangle reference element
ref_tria = reference_element(Polytope"3-node triangle"())
# construct the geometry of a triangle
tria=Geometry{Polytope"3-node triangle", 2, Float64}(
  (0., 0.), (1., 0.), (2., 0.))
# obtain global coordinates from local coordinates
#  return (2, 0)
local_to_global(quad, SVector{2, Float64}(0., 1.))
```

Figure 4.12: Reference elements and coordinate transformation

4.3. Mesh Topology Construction

After a mesh has been constructed, e.g. by adding cells manually or by loading the data from a Gmsh file, only the mesh connectivity from codimension zero cells to dimension zero cells, i.e. $c_{\mathcal{M},2\to 0}$ since we are in two dimensions, is constructed. For the assembly procedure discussed later on we additionally need the connectivity $c_{\mathcal{M},1\to 0}$ to obtain the cell identifiers of edges associated to two dimensional cells. We begin with the description of the construction of $c_{\mathcal{M},1\to 0}$ from $c_{\mathcal{M},2\to 0}$. This construction procedure can easily be extended to construct also $c_{\mathcal{M},2\to 1}$ and $c_{\mathcal{M},2\to 2}$, resulting in a restricted bidirectional topology representation. In order to construct $c_{\mathcal{M},2\to 1}$ one only needs to determine which edges are located on the boundary and which are located in the interior of the mesh. Then, given a two dimensional cell, one can add all of its edges onto $c_{\mathcal{M},2\to 1}$ if they are either on the boundary or if the first vertex id of the edge is larger than the second. The latter condition ensures that an edge is only added only once. Note that this only works if the source of the edge of one triangle is equal the corresponding edge of the neighbouring triangle. To determine whether an edge is on the boundary one just needs to find another triangle containing this edge. In the beginning of the development of TipiFEM two algorithms were tested to construct this $c_{\mathcal{M},1\to 0}$. Both of these algorithms were rather slow such that a new third algorithm was developed outperforming the first two ones by far. We begin with the sketch of the first two algorithms and discuss the reasons for their bad performance.

Hash table based algorithm The first algorithm was based on a hash table datastructure. While a hash table has constant average complexity for the lookup operation thus rendering an algorithm with complexity linear to the number of codimension zero cells the memory accesses suffers from poor locality of reference[14] and computational cost for the computation of the hash values such that the constants hidden in the asymptotic runtime made this algorithm rather slow.

Binary search based algorithm An improvement to this was an algorithm that is based on binary search where first, one iterates over all codimension zero cells, then adds all edges and the cell id of its codimension zero cell to an array, thus obtaining an array containing all edges on the boundary once and all edges in the interior of the domain twice. Then one copies this array, obtaining a second array and sorts it by the vertex connectivity of the contained edges. Equipped with these two arrays one iterates over the unsorted one and runs for every contained edge binary search on the sorted array to find an edge with opposite direction. If such an edge was not found the current edge is an edge on the boundary and we add it to a third array. Otherwise we only add it if the cell index of the source is larger than the cell index of its sink. Again the latter condition ensures that an edge is only added once. Afterwards the array contains every edge only once and we can assign cell identifiers to the edges using its index in the third array. The asymptotic runtime of this algorithm is $\mathcal{O}(n \cdot \log(n))$ where n is the number of codimension zero cells. Even though the asymptotic runtime of this algorithm is higher than for the hash table based algorithm, it was faster even for meshes with millions of elements because it has better locality of reference [14].

4.3.1. Sorting Based Algorithm

The thrid algorithm, which is the one currently used in TipiFEM, is based again on sorting, but this time without the need for binary search. Furthermore, this algorithm constructs also $c_{\mathcal{M},2\to 1}$ and $c_{\mathcal{M},2\to 2}$ For its implementation we need the additional concept of a facet tuple.

Definition 4.2 (Facet tuple) *A facet tuple is a quadruple (i, j, b, c) associated with a facet of a cell. The elements of the tuple are:*

> *i: A cell id of the codimension zero cell that contains the facet.*
>
> *j: An integer storing the facet index of the facet.*
>
> *b: A boolean storing whether the facets vertex connectivity is in canonical form.*
>
> *c: The canonical form of the facets vertex connectivity.*

The facet tuples of all facets of a cell can be constructed given only its cell id and vertex connectivity.

[14] See `https://en.wikipedia.org/wiki/Locality_of_reference`

Restricted bidirectional version Now we will sketch a version of the topology construction algorithm that constructs a restricted bidirectional topological representation.

We first define a set of variables that store the data used in the algorithm:

`mesh_conn`	Mesh connectivity from codimension zero cells to their vertices. Obtained from the mesh instance that is an input argument to the algorithm.
`facets_mesh_conn`	Mesh connectivity of codimension one cells (edges) with their vertices. Empty at the beginning of the algorithm.
`neighbour_mesh_conn`	Mesh connectivity of codimension zero cells (triangles, quadrilaters) with codimension zero cells.
`el_facet_mesh_conn`	Mesh connectivity of codimension zero cells (triangles, quadrilaters) with their facets (edges). Empty at the beginning of the algorithm.
`facet_tuples`	Temporary array of facet tuples.
`boundary_facet_tuples`	Array of facet tuples whose facets are on the boundary.
`boundary_facet_ids`	Cell ids of facets that are part of the boundary of the mesh.
`boundary_facet_tuples`	Facet tuples of facets that are part of the boundary of the mesh.

Now we come to the algorithm itself:

1. Iterate over all codimension zero cells of the mesh stored in `mesh_conn`, construct their facet tuples and add them to the `facet_tuples` array.

2. Sort the facet tuples in `facet_tuples` by the facets vertex connectivity. Since the vertex connectivity is given in canonical form, the facet tuples associated with the same facet are next to each other in the sorted array.

3. This step constructs the topological information of interior facets. First iterate over the sorted facet tuples and do a case distinctions:

    - If the vertex connectivity in the facet tuple in the last iteration is the same as in the facet tuple in the current iteration the facet is an interior facet and we do the following:
        1. Remove the last facet tuple from `boundary_facet_tuples` since it is associated with an interior facet.
        2. Assign a new cell id to the facet (both facet tuples are associated to the same facet).
        3. Store its vertex connectivity in `facets_mesh_conn`.
        4. Store in `neighbour_mesh_conn` that the two codimension zero cells of the current and last facet tuple are neighbours.
        5. For the current and the last facet tuple: Fetch the cell connectivity of the codimension zero cell in the facet tuple with its

facets from `el_facet_mesh_conn`. Using the face index given in the facet tuple store in the obtained connectivity the previously assigned cell id.

- Otherwise add the current facet tuple to `boundary_facet_tuples`. Note that it is only determined in the next iteration whether the current facet is on the boundary. Therefore its facet tuple needs to be removed again in the next iteration if the facet is an interior facet.

4. For each of the facet tuples in `boundary_facet_tuples` do the following:
    1. Assign a new cell id to the facet.
    2. Add the newly assigned id to `boundary_facet_ids`.
    3. Store its vertex connectivity in `facets_mesh_conn`. If the canonical form differs from the original vertex connectivity, flip the orientation. This flipping ensures that normals of facets on the boundary can be calculated given only its geometry.
    4. Store that the codimension zero cell has no neighbour.
    5. Fetch the cell connectivity of the codimension zero cell in the facet tuple with its facets from `el_facet_mesh_conn`. Using the face index given in the facet tuple store, in the obtained connectivity, the previously assigned cell id.
5. Tag all facets in `boundary_facet_ids` with the tag `:boundary`.

Staged restricted bidirectional version The previously described restricted bidirectional version suffers from the problem that the `facet_tuples` array contains cell ids to codimension zero cells of different cell type. This leads to a situation where access to `el_facets_mesh_conn` and `neighbour_mesh_conn` involves a substantial overhead because of type instability[13]. The idea of the staged restricted bidirectional version is then:

1. Run step 1 - 4 of the naive restricted bidirectional version separately on all codimension zero cells of a single cell type. In 2D this means that one runs the naive version first for all triangles and then for all quadrilaterals. Now the obtained `boundary_facet_tuples` may contain facet tuples associated with facets that are the intersection of two cells with different type.
2. Run step 2 - 5 on `boundary_facet_tuples` instead of `facet_tuples`.

The last step is still type unstable but usually the number of facets that are the intersection of two cells is low. Therefore the last step takes only a fraction of time and does not need to be optimized any further. Overall this algorithm performs significantly better than the two previous ones because only a few sort operations and for loops over elements consecutive in memory are required. For a mesh consisting of 2097152 cells the algorithm still requires about 13 seconds to finish on a 4th generation Intel i7 processor with 2.1GHz. The reason for the rather bad performance is a type inference9 issue currently present in Julia and a speedup of $\approx$ 6x was measured when the algorithm was restricted to cells of a single cell type. This algorithm is right now the one used in TipiFEM. Later on it might be extended such that a fully bidirectional topology representation is constructed.

5

Numerical Integration / Quadrature

In case the coefficient σ or source term f of our model problem[3] 3 is given only in procedural form numerical quadrature is mandatory to evaluate the integrals in the linear and bilinear form. Composite quadrature is then used to split the integral on a mesh $\mathcal{M}$ into cell contributions, required for a cell oriented assembly procedure. The approximation of the integral $\int_C f(x)\mathrm{d}x$ can be written in terms of a P-point local quadrature rule

$$\int_C f(x)\mathrm{d}x \approx s \sum_{l=1}^{P_k} \omega_k^C f(\zeta_l^C) \qquad C \in \mathcal{M},\, P_K \in \mathbb{N},\, \omega_l^C \in \mathbb{R},\, \zeta_l^C \in C$$

with s a scaling factor, ω_l^C the weights and ζ_l^C the quadrature nodes. The weights and quadrature nodes however are specific the cell C. To avoid recomputation of the quadrature nodes and weights, their values are only stored for a quadrature rule defined on the reference cell $\hat{C}$. We will call this a local quadrature rule. The quadrature rule on the cell C is then obtained using a coordinate transformation $\Phi_C : \hat{C} \to C, \hat{x} \mapsto x$.

$$\int_C f(x)\mathrm{d}x = \int_{\hat{C}} f(\Phi_C(\hat{x}))\, |\det(D\Phi_C)|\, \mathrm{d}\hat{x} \approx \hat{s} \sum_{l=1}^{P_k} \hat{\omega}_k^C f(\hat{\zeta}_l^C)$$
$$C \in \mathcal{M},\, P_K \in \mathbb{N},\, \omega_l^C \in \mathbb{R},\, \zeta_l^C \in C$$

5.1. Datastructures

All quadrature related datastructures are implemented in the `TipiFEM.Quadrature` module. The actual quadrature rules itself are implemented in the concrete mesh implementation. We will now discuss the datastructures and their usage in TipiFEM.

`Quadrule` An instance of the immutable composite type[10] `Quadrule{C, num_points, REAL_T}`, where `C` is a cell type, `num_points` is the number of quadrature points and `REAL_T` is a type representing real numbers, stores the weights, quadrature points and scaling factor of a quadrature rule.

```
@computed type Quadrule{C <: Cell, num_points, REAL_T}
  w::SVector{num_points, REAL_T}
  x::SVector{num_points,
             dim(C) == 1 ? REAL_T : SVector{dim(C), REAL_T}}
  scale::REAL_T
end
```

Figure 5.1: Definition of the `Quadrule` datastructure

Local quadrature rules are accessible by means of the `quadrule` function that takes the type of a quadrature rule and returns the weights, quadrature points and scaling factor of such a rule.

```
ŵs,x̂s,ŝ = quadrule(Quadrule{Polytope"3-node triangle",6,Float64})
```

Figure 5.2: Access to a six-point quadrature rule on the reference triangle in double precision.

`QuadruleList` Some functions that rely on quadrature, e.g. the element matrix assembler, can be written in a generic fashion such that they may be evaluated on different types of cells. In that case the `Quadrule` datastructure is insufficient to access local quadrature rule data because it can only store the data for a single reference cell. For this purpose the `QuadruleList` datastructure exists that stores multiple local quadrature rules in a single datastructure. Access to a quadrature rule is then by evaluation of the `[]` operator on the quadrature rule list with the cell type for which one wants to obtain the data.

```
# construct quadrature rule list
quadrule_list = QuadruleList(
  Quadrule{Polytope"2-node line", 3, Float64},
  Quadrule{Polytope"3-node triangle", 12, Float64},
  Quadrule{Polytope"4-node quadrangle", 12, Float64})
# access quadrature rule data for the reference triangle
ŵs, x̂s, ŝ = quadrule_list[Polytope"3-node triangle"]
```

Figure 5.3: Constrution of a `QuadruleList` and access to the quadrature rule data for the reference triangle

6

Finite Element Space & Finite Element Basis

6.1. Finite Element Basis

All information about a finite element can be accessed via methods that take an argument of type `FEBasis` and a cell type. The `FEBasis` type itself is a singleton type[10] and thus does not store any data except for its type parameters. The type has two type parameters. The first parameter `basis_type` is a symbol specifying the type of the element (i.e. `:Lagrangian`) and the second parameter `approx_order` is an integer from which the number of local shape functions can be deduced. In case of the $\mathbb{P}_p$ and $\mathbb{Q}_p$ element the `approx_order` is simply p. At the time of writing only the $\mathbb{P}_p$ and $\mathbb{Q}_p$ element for $p = 1, 2$ are implemented. Therefore we will only discuss these elements here.

```
# Linear Lagrangian finite element basis
basis = FEBasis{:Lagrangian, 1}()
# Second order Lagrangian finite elements
basis = FEBasis{:Lagrangian, 2}()
```

Drawbacks While it is convenient to use the polynomial degree as a type parameter for the Lagrangian element this prevents the usage of Lagrange elements which are not the $\mathbb{P}_p$ or $\mathbb{Q}_p$ element but are still defined on simplices or parallelotops. Therefore the interface of `FEBasis` might change in the feature.

Local degrees of freedom Information about a local degrees of freedom can be accessed by methods that take an argument of type `LocalDOF`. The `LocalDOF` type is a singleton type[10] with three type parameters:

- `basis`: A subtype of `FEBasis`.
- `K`: A subtype of `Cell` specifying the cell type.
- `idx`: An index specifying which degree of freedom is represented.

Note that the type does not store the value of the degree of freedom but is only used for accessing information.

In case `basis` is a subtype of `FEBasis{:Lagrangian}` the following methods can be called on an instance of `LocalDOF`:

- `associated_cell`: Return the associated cell of the interpolation node which belongs to the local degree of freedom.
- `local_face_indices`: Return the index of the face to which the interpolation node of the local degree of freedom is associated to.
- `interpolation_node`: Return the interpolation node which is associated to the degree of freedom.
- `local_shape_function`: Returns the shape function that belongs to the dof.
- `grad_local_shape_function`: Returns the gradient of the shape function that belongs to the dof.

Local interpolation nodes A local interpolation node is represented by the `LocalInterpolationNode` singleton type[10]. The following methods can be called on an instance of `LocalInterpolationNode`:

- `coordinates`: Return the coordinates of the interpolation node.
- `associated_cell`: The cell associated to the interpolation node.

The local interpolation nodes of a Lagrange element are accessible via the `interpolation_nodes(::FEBasis{:Lagrangian}, ::Cell)` method.

```
# Prints:
# [0.0, 0.0]
# [1.0, 0.0]
# [1.0, 1.0]
# [0.0, 1.0]
for node in interpolation_nodes(FEBasis{:Lagrangian, 1}(),
    Polytope"4-node quadrangle"())
  println(coordinates(node))
end
```

Figure 6.1: Print interpolation nodes of the $\mathbb{Q}_1$ element.

Local shape functions Additionally to accessing the local shape functions and their gradients by means of the `LocalDOF` type, all of them can be accessed at once by means of calling `local_shape_functions`, respectively `grad_local_shape_functions`, on a `FEBasis` instance and a cell type.

```
# construct a FEBasis
basis = FEBasis{:Lagrangian, 2}()
# select a cell type
K = Polytope"3-node triangle"()
# specify some points at which we want to evaluate
#  the shape functions and their gradients
x̂=SVector{2, Float64}(0.5, 0)
x̂s=SMatrix{2, 2, Float64}(1, 0.5, 0, 0)
#
# SISD version
#
# evaluate local shape functions
local_shape_functions(basis, K, x̂s)
# evaluate gradients of local shape functions
grad_local_shape_functions(basis, K, x̂s)
#
# SIMD/vectorized version
#
# evaluate local shape functions
local_shape_functions(basis, K, x̂s)
# evaluate gradients of local shape functions
grad_local_shape_functions(basis, K, x̂s)
```

Figure 6.2: Evaluation of local shape functions and their gradients

Boundary (local) degrees of freedom The boundary degrees of freedom of a finite element can be accessed by calling `boundary_dofs` with a `FEBasis` instance and a cell type.

Internal (local) degrees of freedom The internal degrees of freedom of a finite element can be accessed by calling `internal_dofs` with a `FEBasis` instance and a cell type.

6.2. Finite Element Space

A Langrangian finite element space is represented by the `FESpace` datatype. The finite element basis is stored as a type parameter. The datatype has the following fields:

- `mesh`: The `Mesh` instance on which the finite element space is defined.
- `dofh`: The `DofHandler` that maps a cell id and a local degree of freedom index to a global degree of freedom index.
- `constraints`: An array of `IndexMappings` that maps indices of degrees of freedoms to degrees of freedom. The field is used to store Dirichlet degrees of freedom. In the future it might also be used to store `IndexMappings` from indices of degrees of freedom to a indices of other degrees of freedom

with an associated weight. This could then be used to do block elimination of internal degrees of freedom.

- `active_cells`: A cell id iterator. No cells that are faces of cells that are already contained in `active_cells` need to be stored. Only contributions of local shape functions on these cells are distributed by the Galerkin matrix / right hand side assembler. For a Neumann term the active cells are codimension 1 cells.

- `active_cells_mask` A boolean valued mesh function on all cells (i.e. quadrilaterals, triangles, edges, vertices) in the mesh. All contributions of local shape functions associated to cells for which this function evaluates to false are dropped.

```
# setup trial space
trial_space = FESpace(basis, mesh)

# evaluate analytic solution at all Dirichlet nodes
dirichlet_term = map(u, interpolation_nodes(trial_space,
  tagged_cells(mesh, :boundary)))

# impose dirichlet boundary conditions
add_constraints!(trial_space, dirichlet_term)

# retrieve degrees of freedom on the triangle with
#  id 2 that are active
active_dofs(trial_space, Id"3-node triangle"())
```

Figure 6.3: Construction of a trail space with (inhomogeneous) Dirichlet boundary conditions and retrieval of active degrees of freedom.

7

Assembly

In this section the assembly procedure of the Galerkin matrix and right hand side vector is described.

7.1. `DofHandler` - Local to Global Mapping

The distribution of local contributions relies on an function `locglobmap`: $\mathcal{M} \times \mathbb{N} \to \mathbb{N}, (K, i) \mapsto j$ that maps a cell K and the index of the local degree of freedom i to the index of a global degree of freedom j. We will now construct such a mapping that ensures that the finite element space is H_1 conforming. The constructed mapping can only be used if a fixed finite element basis is used, which is currently a constraint in TipiFEM anyway.

- **Initialization procedure** First write down the cell types in some order, e.g. ordered by ascending dimension. Afterwards assign consecutive indices starting at one to the cell types. Then assign to each cell type an integer called offset by recursion

$$\begin{aligned} \text{offset}(1) &= 0 \\ \text{offset}(n) &= \text{offset}(n-1) + \#(i) \cdot \text{multiplicty}(i) \end{aligned}$$

  where $\#i$ is the number of cells and multiplicty(i) is the number of internal degrees of freedom of the cells with type i in $\mathcal{M}$. Then by definition the offsets are separated such that one can assign to each degree a unique integer in the range of its associated cell type.

  For a hybrid mesh the offsets read:

i	cell type	offset(i)
1	vertices	0
2	edges	offset(1) + #vertices · multiplicity(vertices)
3	triangles	offset(2) + #edges · multiplicity(edges)
4	quadrilaterals	offset(3) + #triangles · multiplicity(triangles)

  This initialization procedure needs to be done only once.

- **Evaluation procedure**

1. Fetch the cell index `k` of the cell to which the local degree of freedom with index i is associated to.
2. Fetch the cell constrained interpolation node index `l` of the local degree of freedom with index i.
3. Now the index of the global degree of freedom is:
   $i =$ `offset(cell_type(`K`)) + k + l`.

This approach takes only memory space proportional to the number of cell types instead of proportional to the number of degrees of freedom and is very fast to construct and evaluate.

The index mapping discussed previously is represented by the `DofHandler` type in TipiFEM. The type can be constructed by passing a `Mesh` instance and a `FEBasis` to its constructor. The constructor then does the initialization procedure. Given a cell id the global indices of the degrees of freedom can be obtained by means of the `[]` operator.

```
dofh = DofHandler(mesh, basis)
dofs = dofh[Id"3-node triangle"(1)]
```

Figure 7.1: Construction of a `DofHandler` object and access to the global indices of the degrees of freedom on the triangle with cell index 1

7.2. Element Matrix Assembly

In TipiFEM the element matrix assembler is a function that takes the finite element basis of the trial space and the finite element basis of the test space and returns a function. The returned function is a function that takes a cell identifier and its geometry and returns the element matrix belonging to the cell with the given cell identifier. This two step evaluation procedure is done because it avoids capturing the finite element basis from the enclosing scope in which the assembler has been defined and instead hands over the obligation to pass the basis to the Galerkin matrix assembler. The element matrix assembly itself has to be implemented by the user of the library.

The element matrix assembler for the model problem [3] can be found below:

```
function element_matrix_assembler(basis::B,_::B) where B<:FEBasis
  function (cid::Id{K}, geo::Geometry{K,world_dim,REAL_T}) where
      {K <: Cell, world_dim, REAL_T <: Real}
    # the number of local shape functions on K
    n = number_of_local_shape_functions(basis, K())
    # get quadrature rule data
    ŵs, x̂s, ŝ = quadrule_list[K]
    # initialize empty element matrix
    el_mat = zeros(MMatrix{n, n, Float64})
    # assemble element matrix
    for (ŵ, x̂) in zip(ŵs, x̂s)
      let DΦ⁻¹=DΦ⁻¹(geo, x̂),
        grads = map(grad -> DΦ⁻¹*grad,
          grad_local_shape_functions(basis, K(), x̂)),
        det_DΦ = integration_element(geo, x̂),
        x = local_to_global(geo, x̂)
        for i in 1:n
          for j in 1:n
            grad_bi = grads[i]
            grad_bj = grads[j]
            el_mat[i, j] += det_DΦ * ŵ * grad_bi ⋅ grad_bj * σ(x)
          end
        end
      end
    end
    # apply scaling factor
    el_mat*ŝ
  end
end
```

Figure 7.2: Element matrix assembler for the bilinear form $a(u,v) = \int_K \sigma(x)\,\mathrm{grad}\,u \cdot \mathrm{grad}\,v\,\mathrm{d}x$

7.3. Element Vector Assembly

The implementation of the element vector is analogous to the one of the element matrix. The element vector assembler is a function that takes the finite element basis of the test space and returns a function that takes the cell id and its geometry and returns the corresponding element vector. Again the element vector assembler has to be implemented by the user.

```
function element_vector_assembler(basis::FEBasis)
  function (cid::Id{K}, geo::Geometry{K,world_dim,REAL_T}) where
        {K <: Cell, world_dim, REAL_T <: Real}
      # the number of local shape functions on K
      n = number_of_local_shape_functions(basis, K())
      # get quadrature rule data
      ŵs, x̂s, ŝ = quadrule_list[K]
      # initialize empty element matrix
      el_vec = zeros(MVector{n, Float64})
      # assemble element matrix
      for (ŵ, x̂) in zip(ŵs, x̂s)
          let lsfs = local_shape_functions(basis, K(), x̂),
              det_DΦ = integration_element(geo, x̂),
              x = local_to_global(geo, x̂)
              for i in 1:n
                  el_vec[i] += det_DΦ * ŵ * lsfs[i] * f(x)
              end
          end
      end
      # apply scaling factor
      el_vec*=ŝ
      # return element vector
      el_vec
  end
end
```

Figure 7.3: Element vector assembler for the linear form $l(v) = \int_K f(x)v \, \mathrm{d}x$

7.4. Galerkin Matrix Assembly

The Galerkin matrix assembler in TipiFEM is implemented in the `matrix_assembler` function that takes an element matrix assembler defined by the user and two instances of `FESpace` one of which is the trial space and the other on is the test space. The actual assembly is then the usual one and works as follows:

1. The Galerkin matrix assembler evaluates the element matrix assembler with the basis of the trial and test space and obtains the actual element matrix assembler taking a cell id and its geometry.

2. The Galerkin matrix assembler iterates over all active cells in the finite element space with the lowest amount of active cells.

   (a) Evaluates the actual element matrix assembler.

   (b) Distributes the elements of the element matrix assembler that belong to cells that are not marked inactive in the `active_cell_mask` field of the trial and test space.

3. Return the Galerkin matrix (in triplet form)

```
A = matrix_assembler(element_matrix_assembler, trial_space,
  test_space)
```

Figure 7.4: Galerkin matrix assembly

7.5. Right Hand Side Vector Assembly

The right hand side vector assembler in TipiFEM is implemented in the `vector_assembler` function which takes an element vector assembler and the test space. There is nothing exceptional about the assembler. It just iterates over all active cells and distributes the entries of the element matrix to the coefficient vector.

```
b = vector_assembler(element_vector_assembler, test_space)
```

Figure 7.5: Right hand side vector assembly

7.6. Incorporation of Neumann Boundary Conditions

For the incorporation of Neumann boundary condition one just needs to construct a `FESpace` instance where the active cells are the cells of the boundary on which Neumann boundary conditions are present and then assemble the right hand side vector with that finite element space and an element vector assembler that evaluates the Neumann term of the variational equation.

```
# get the neumann cells
neumann_cells = tagged_cells(mesh, :neumann_cells)

# construct a finite element space for the neumann cells
neumann_fespace = FESpace(basis, mesh, neumann_cells)

# assemble right hand side b
# ...

# incorporate Neumann boundary conditions
b += vector_assembler(element_vector_neumann, neumann_fespace)
```

Figure 7.6: Incorporation of Neumann boundary conditions

7.7. Incorporation of Dirichlet Boundary Conditions

Currently the elimination technique is used to incorporate Dirichlet boundary conditions. Implementation wise this means that the matrix assembler skips all rows of the Galerkin matrix that correspond to degrees of freedom where Dirichlet boundary conditions are imposed. Then after the assembly of the Galerkin matrix the `incorporate_constraints` method must be evaluated by the user. This method sets the diagonal entries of Dirichlet degrees of freedom to one and modifies the right hand side vector by setting the degrees of freedom on which Dirichlet conditions have been imposed to the fixed values that are stored in the `constraints` field of the trial space.

```
# get the dirichlet cells
dirichlet_cells = tagged_cells(mesh, :dirichlet_cells)

# evaluate analytic solution at all dirichlet nodes
dirichlet_term = map(u, interpolation_nodes(trial_space,
  dirichlet_cells))

# impose dirichlet boundary conditions
add_constraints!(trial_space, dirichlet_term)

# assemble Galerkin matrix and rhs vector
# ...

# incorporate Dirichlet boundary conditions into A and b
incorporate_constraints(trial_space, A, b)
```

Figure 7.7: Incorporation of Dirichlet boundary conditions in TipiFEM

This technique has the disadvantage that the resulting Galerkin matrix is not symmetric anymore and contains a higher amount of nonzero elements. Therefore it is planed to use the augmentation technique in later versions of TipiFEM.

8 Case Study

Now that we have developed all building blocks to solve the model problem [3] we proceed by solving the problem on a hybrid mesh with inhomogeneous Dirichlet and inhomogeneous Neumann boundary data. We use the following manufactured solution u and coefficient σ:

$$\begin{aligned} u(x,y) &= \sin(x)\sin(y) \\ \sigma(x,y) &= xy \end{aligned}$$

The domain Ω is the union of the two sets Ω_Q and Ω_T.

$$\begin{aligned} \Omega_Q &= \left\{ (x,y) \in \mathbb{R}^2 \;\middle|\; 0 \le x \le 1, \quad 0 \le y \le 1 \right\} \\ \Omega_T &= \left\{ (x,y) \in \mathbb{R}^2 \;\middle|\; 1 \le x \le 2, \quad \frac{x-1}{2} \le y \le 1 - \frac{x-1}{2} \right\} \\ \Omega &= \Omega_Q \cup \Omega_T \end{aligned}$$

Using the strong formulation of the model problem [3] we can derive f.

$$\begin{aligned} f(x,y) &= -\operatorname{div}\left(\sigma(x)\operatorname{grad} u\right) \\ &= -x\cos(x)\sin(y) - x\sin(x)\cos(y) + 2xy\sin(x)\sin(y) \end{aligned}$$

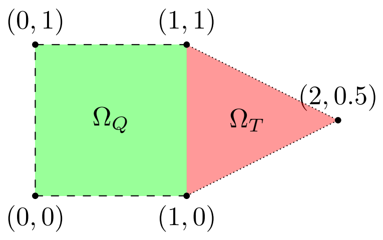


Figure 8.1: The domain Ω with inhomogeneous Dirichlet boundary conditions on the dashed line and inhomogeneous Neumann boundary conditions on the densely dotted line.

The set Ω_Q was subdivided using quadrilaterals and Ω_T using triangles. Afterwards the timings were collected for both linear and quadratic Lagrangian

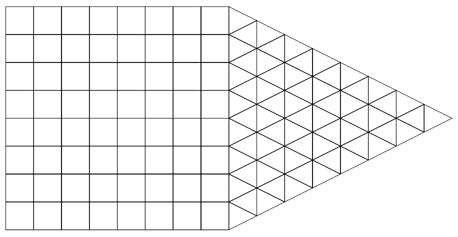


Figure 8.2: Hybrid mesh 4.1 after three regular refinement steps.

N	Setup time	Assembly time	Sparse solver time	$\|\|u - u_N\|\|_0$
5	0.213573	0.356063	0.000425389	3.37509e-3
12	0.00358171	0.0010706	0.000506341	3.03735e-4
35	0.00352524	0.00125792	0.00184714	2.38240e-5
117	0.00363856	0.00215343	0.00110623	1.71460e-6
425	0.00402946	0.00557717	0.0149168	1.17279e-7
1617	0.00453351	0.022046	0.0240581	7.77739e-9
6305	0.00515485	0.0841215	0.0569778	5.06050e-10
24897	0.00404427	0.193442	0.353058	3.25407e-11
98945	0.0143459	0.893896	1.15355	2.07687e-12
394497	0.00789053	3.35427	5.09676	1.31907e-13

Table 8.1: Timings in seconds for linear Lagrangian finite elements on a sequence of meshes created by regular refinement. N is the number of degrees of freedom.

finite elements on a sequence of meshes that were created by regular refinement of the mesh in Figure 4.1 using the Gmsh[3] mesh generator. The Pardiso solver[15] was used to solve the sparse linear system of equations. On both triangles and quadrilaterals a 12-point quadrature rule was used to calculate to element matrices and vectors. On edges a three point quadrature rule was used. The timings were collected on a computer with an *Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz* processor and 8GB of DDR3 memory. The timings can be found in Table 8.1 and 8.2. The setup time is just the time needed to construct the `FESpace` instances. The assembly time is the time needed to compute the Galerkin matrix and right hand side vector, the incorporation of Dirichlet and Neumann boundary conditions, and the conversion from triplets to CSC format.

Scaling behavior Figure 8.3 shows a log-log plot of the number of degrees of freedom compared to the assembly time. The plot suggests that the assembly procedure scales like $\mathcal{O}(N)$, where N is the number of degrees of freedom. The nonlinear increase in runtime before the fourth refinement step can be explained by the memory hierarchy of the computer. No explanation on the bump in the sixth refinement step was found. The bump persisted across multiple runs and a detailed memory traffic analysis that might explain the behavior was not feasible.

[15]The `Pardiso.jl` package was used to call the Pardiso library. See: https://github.com/JuliaSparse/Pardiso.jl

N	Setup time	Assembly time	Sparse solver time	$\|\|u - u_N\|\|_0$
12	0.00243636	0.421545	0.000470896	6.43083e-5
35	0.00377534	0.0011894	0.000772178	7.67773e-7
117	0.00360992	0.00198156	0.00792169	1.08240e-8
425	0.00420641	0.00379326	0.00797907	1.62498e-10
1617	0.00455445	0.0120698	0.02377	2.46452e-12
6305	0.00584542	0.0546963	0.130999	3.74386e-14
24897	0.00359999	0.113003	0.480396	5.69799e-16
98945	0.00480784	0.479446	1.7126	8.69804e-18
394497	0.00698595	2.28516	7.38854	1.33855e-19
1575425	0.0121419	8.18097	36.5552	2.75770e-21

Table 8.2: Timings in seconds for quadratic Lagrangian finite elements on a sequence of meshes created by regular refinement. N is the number of degrees of freedom.

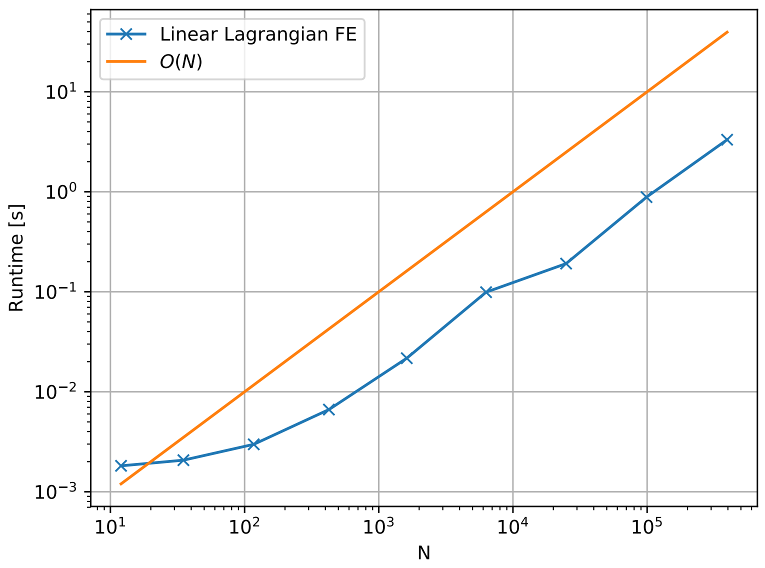


Figure 8.3: Number of degrees of freedom plotted against the runtime of the assembly of the Galerkin matrix and right hand side vector. The runtime on the unrefined mesh is omitted..

Evaluation First of all it should be noted that the relativly long runtimes for the lowest value of N are an artefact, stemming from the fact that in the first run Julia is still compiling code. The evaluation of the timings without a theoretical performance analysis or direct comparisons to other libraries is quite difficult. Looking at the timings given in [1] the timings seem rather competitive. A detailed comparison however is not meaningful since the timings given there are for the Galerkin matrix assembly only, a constant σ, and a different quadrature, and are therefore omitted. Considering only single core performance I conclude from the timings that TipiFEM's design goal of being a performant library has been met. However, considering that nearly 50% of the assembly time is spend in the garbage collector and no caching of the evaluation of shape functions and their gradients has been done shows that there is still optimization potential.

9

Discussion of the Julia Programming Language

The development of TipiFEM was my first development of a larger library in Julia. In this section I want to discuss the my experiences with the language. Most topics covered in this section are not tied to the implementation presented in this paper and might be interesting to everyone that develops numerical libraries in Julia or is assessing Julia's current state to see whether it is suitable for a given use case.

9.1. Type Inference

Type inference algorithm Dynamically typed programming languages require by definition no type declaration on variables. The process of automatically deducing the type of expressions is termed type inference and the algorithm that executes this deduction is a type inference algorithm.

Abstract types An abstract type is a type that cannot be instantiated directly[10]. A type that is not abstract, thus can be instantiated directly, is a concrete type or leaf type.

Problem description In Julia abstract types have no structure, i.e. no fields, and therefore their memory layout is not known to the compiler. Therefore accessing fields of abstract types induces a significant overhead. This induces significant overhead in case abstract types are accessed frequently and this should therefore be avoided [16]. However, even though all types of expressions that return concrete types may be deducible in theory, in practice this process sometimes fails. I observed two major causes where the process failed:

- Instantiation of types whose type parameters were specified by constant expressions[17]. Since currently type inference is run after constant folding/propagation[18], the type cannot be deduced because the type parameters are only known after type inference. This problems are difficult to find

[16]This is also one of the reasons for the various mesh function datatypes.
[17]For example `2*2` is a constant expression.
[18]See `https://en.wikipedia.org/wiki/Constant_folding`

because the type annotated code, that can be displayed in julia, contains the result of the constant expression only, thus hiding its origin.

- Access to fields of types that are so complex that the type inference gives up on them. This problems are nearly impossible to find without knowledge of the type inference algorithm. This is because every reduction of the code may just render it simple enough such that the types are infered well enough.

The two causes stated were a major cause of performance degradation in TipiFEM and lead to a situation in which most of my time assigned to optimization was spend on finding occurrences in which types were not infered well enough, even though the actual fix was often easy. This time however could have been better spend on other optimizations. Especially because I only optimized performance critical parts where I used concrete datatypes anyway.

Possible solutions & workarounds

- Wrap constant expressions into a pure functions. Sometimes this works, sometimes it doesn't. This problem will probably disappear after Julia issue #5560[19] is closed.
- Increase the `tupletype_len`, `tuple_depth`, `tuple_splat`, `apply_union_enum` of Julia's inference algorithm. Unfortunately this requires a recompilation of Julia.

Furthermore, I think that Julia would profit significantly from a mechanism to annotate expressions in which all variables and accessed fields are required to be concrete types[20]. This could then be verified by the compiler and hints on the location should be printed. I assume this would eliminate most of the time spend on fixing problems with inference.

9.2. Metaprogramming / Code generation

The two most important features for metaprogramming in Julia are Lisp-style macros and `@generated` functions. Both of this features are heavily used in TipiFEM and I want to give an example demonstrating their superiority compared to macros in C/C++ and template metaprogramming in C++.

`@dim_dispatch` macro The first example is the `@dim_dispatch` macro that, given a function definition with type parameter `K`, with `K` a codimension zero cell that takes arguments of type `Dim` or `Codim`, generates a set of wrapper functions by altering the function signature such that each `Dim` argument can be specified as a `Codim` argument and vice versa. One of the functions that makes use of this macro is `connectivity(msh::Mesh{K}, i::Dim, j::Dim) where K`

[19]See `https://github.com/JuliaLang/julia/issues/5560`

[20]A discussion about this can be found at `https://github.com/JuliaLang/julia/issues/10980`.

`<: Cell` that, given a mesh and two dimensions, returns the mesh connectivity $c_{\mathcal{M},i\to j}$. Given the definition of this function the macro automatically detects that the `i` and `j` argument are of type `Dim` and generates three additional functions with the following signatures:

- `connectivity(msh::Mesh{K}, i::Codim, j::Dim) where K <: Cell`
- `connectivity(msh::Mesh{K}, i::Dim, j::Codim) where K <: Cell`
- `connectivity(msh::Mesh{K}, i::Codim, j::Codim) where K <: Cell`

Macros in C/C++ are not capable of such transformations because they do not allow for examination of the abstract syntax tree, which makes them useful only in a much smaller subset of cases. It should be noted that there are ways of doing something similar in C++11, but certainly not with the same productivity and elegance that Julia's Lisp-style macros offer.

9.3. Memory Allocation

Except for immutable composite objects, objects in Julia are usually allocated on the heap[21]. While this is very convenient and in certain cases also more efficient than reference counting often allocation and garbage collection are a bottleneck. This is especially true if an object allocated inside of a subroutine does not escape to the calling subroutine. In that case allocation on the stack[22] is often significantly faster. There exist mainly two workarrounds for this: Either one allocates the object in global scope once or if the caller calls the callee multiple times, allocates the object in the caller and passes it as an additional argument to the callee. The first approach however is problematic because it is not thread-safe and the second approach is too intrusive and harnesses flexibility. The only clean solution is to do escape analysis to avoid the allocation on the heap in such cases and since this is currently under active development[23] it was decided that the performance degradation that I observed in TipiFEM can be tolerated until development of this feature has completed. The element matrix assembler currently suffers from this issue. There 50% of the time is spent on garbage collection only. I think that this is currently a huge disadvantage over C/C++.

9.4. Volatility of the Language and Package Ecosystem

At the time I started developing TipiFEM in the end of 2016 the most recent stable version of Julia was 0.5. I quickly switched to the nightly version 0.6 mainly because of the new features introduced to the type system in that version.

[21]See `https://en.wikipedia.org/wiki/Dynamic_memory_allocation`
[22]See `https://en.wikipedia.org/wiki/Stack-based_memory_allocation`
[23]See `https://github.com/JuliaLang/julia/pull/8134`

Now in June 2017 Julia 0.6 is mostly finished. During that time I have observed a lot of changes that were sometimes quite significant and broke a lot of my code. I want to give two examples because they allow to assess whether Julia is already mature enough for production use.

Example 1 - `@generated` function A lot of type instability issues that I encountered occurred because of `if` statements that branch depending on the dimension of a cell, given as an argument. Since the cell type was often accessible at compile time, i.e. because it was a type parameter, those branches were easily eliminated by the use of generated functions that just evaluated the branch condition at compile time and only returned the code of the taken branch. Such functions occurred often in the abstract mesh implementation. The definition of the dimension of a cell is located in the concrete mesh implementation and therefore compiled at a later time point. At some point in the development cycle of 0.6, generated functions became restricted to use only functions that were defined before the generated function was defined. Thus the example given above broke. While there are reasons for this change it has still cost me days to rewrite my code such that it worked again. Such breaking changes are quite rare, but they occur and I do not expect them to disappear until at least version 1.0.

Example 2 - Switch from `FixedSizeArrays.jl` to `StaticArrays.jl` At first I used the `FixedSizeArrays.jl` package to represent vectors and matrices of fixed size because they are much more efficient for small vector/matrix sizes than the built in vector and matrix types where the size is not stored in a type parameter. This package however was deprecated at some point because new features in Julia version 0.6 allowed for a more powerful implementation. I then switched to the new `StaticArrays.jl`. This however required rewriting all code that used fixed size vectors and matrices and additionally all of my datastructures that inherited functionality from types in the `FixedSizeArrays.jl` package. Therefore one should be aware that even packages that are used by quite a few other Julia packages because they provide basic functionality may just be abandoned and will stop working in more recent versions of Julia.

The lesson I learned from these two examples is that the Julia language and also its package ecosystem are currently very volatile, which increases development time and makes its prediction difficult.

9.5. Introspects

Since Julia aims to be a high performance language, it comes with a set of tools to analyse the different stages of its compilation process. These tools were heavily used during the development of TipiFEM to spot potential performance bottlenecks like dynamic memory allocation, unnecessary bound checks, incomplete constant propagation or missing function inlining. Below you find a concrete example of how easy it is in Julia to gather information about the generated machine code:

```
julia> @code_native connectivity(mesh, Dim{2}(), Dim{0}())
        .text
Filename: mesh.jl
        pushq        %rbp
        movq        %rsp, %rbp
Source line: 133
        movq        8(%rdi), %rax
        movq        (%rax), %rax
        movq        48(%rax), %rax
        popq        %rbp
        retq
        nopw        %cs:(%rax,%rax)
```

Figure 9.1: x86 assembly code of the `connectivity function`.

9.6. Contributions to the Package Ecosystem

9.6.1. `ExtendedParametricTypes` & `ComputedFieldTypes`

Julia version 0.6 does not support specifying field types of a parametric type that depend on one of the type parameters. During early development this has shown to be an unfeasible constraint in writing an efficient and maintainable mesh implementation. Most of the datatypes in the mesh implementation depend in some way on the dimension or the number of vertices of a cell. Specifying these values as an additional type parameter, which is how the `StaticArrays` package deals with this issue, is not only redundant, but it also requires writing special constructors and time consuming type stability optimizations for every datatype. I have therefore developed a package that uses metaprogramming to add this functionality in a type stable fashion thus eliminating the need to optimize for type stability for every new datatype. While the type stability has proven to be very difficult it was essential for performance reasons and could be achieved by using what Julia calls a pure function. ”Inspired” by this package Jameson Nash, a Julia maintainer, has developed an improved package `ComputedFieldTypes` providing similar functionality. This package was a major improvement in terms of code quality and ease of use but suffered from some type instability issues in complicated cases required in TipiFEM. I can report to have successfully resolved these issues and a pull request asking to merge my improvements back is currently pending.

9.6.2. `SimpleRepeatIterator`

During the population of the topology datastructure an iterable type was required to efficiently represent the local indices of all edges. Since the local indices are the same for every cell of one cell type, storing them explicitly is unnecessarily wasteful in terms of memory space and bandwidth usage. Thus Julia's built in function `repeat` function cannot be used because it stores every element explicitly. I have therefore developed the `SimpleRepeatIterator.jl` package providing the same functionality as Julia's built in `repeat` function, but without the wasteful allocation. The package is freely available under the MIT Expat license at `https://github.com/tehrengruber/SimpleRepeatIterator.jl`.

9.6.3. `EvalInModuleREPLMode`

While Julia is a Just-In-Time compiled language with a REPL[24], which allows writing code line by line, the development of code that is located inside a package/module, in this case TipiFEM, was tideous at the beginning. One had to either modify the source files and recompile the complete package or wrap the new code inside an `eval` call. This lead to a significant decrease in productivity that was regained by the `EvalInModuleREPLMode` package. The package introduces a new REPL mode such that when one presses `:` one can type a module in which one wants to evaluate code in and afterwards paste in the code to be evaluated. If no module is specified the previously used one is chosen. Therefore with this new package in most cases pressing `:` followed by `<Enter>` and pasting in the code is enough to evaluate code inside a module, significantly increasing productivity of package development.

[24]Read–Eval–Print Loop

10
Conclusions

In the previous chapters the TipiFEM library has been introduced. Its mesh datastructure has been described and example codes have been given of how it can be used. Additionally the mesh topology construction algorithm for hybrid meshes with topological dimension two has been given. Afterwards the usage of quadrature rules and quadrature rule lists in TipiFEM has been explained. Later on the assembly procedure of the Galerkin matrix and right hand side vector has been described. Then a second order elliptic boundary value problem has been solved using TipiFEM. Its timings showed that the implementation is indeed efficient. Therefore the main goal of this thesis to develop an efficient finite element solver in a dynamic programming language has been achieved. My experiences and the timings of the solver showed that Julia is a very promosing language, but also that there are some problems with type inference and memory allocation that need to be solved before it will be ready for production use.

Bibliography

[1] François Cuvelier, Caroline Japhet, and Gilles Scarella, “An efficient way to assemble finite element matrices in vector languages,” *BIT Numerical Mathematics*, vol. 56, no. 3, pp. 833–864, dec 2015.

[2] Anders Logg, “Efficient representation of computational meshes,” 2012.

[3] Christophe Geuzaine and Jean-François Remacle, “Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities,” *International Journal for Numerical Methods in Engineering*, 2008.

[4] Prof. R. Hiptmair, Prof. Ch. Schwab, Prof. H. Harbrecht, Dr. V. Gradinaru, Dr. A. Chernov, and Prof. P. Grohs, “Lecture slides numerical methods for partial differential equation,” `http://www.sam.math.ethz.ch/~hiptmair/tmp/NPDE/NPDE16.pdf`, Accessed: 2017-05-10.

[5] Susanne Brenner and Ridgway Scott, *The Mathematical Theory of Finite Element Methods (Texts in Applied Mathematics)*, Springer, 2007.

[6] Ewgenij Gawrilow and Michael Joswig, *polymake: a Framework for Analyzing Convex Polytopes*, pp. 43–73, Birkhäuser Basel, Basel, 2000.

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, jan 2017.

[8] Cosmin G. Petra, Olaf Schenk, Miles Lubin, and Klaus Gärtner, “An augmented incomplete factorization approach for computing the schur complement in stochastic optimization,” *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C139–C162, 2014.

[9] Cosmin G. Petra, Olaf Schenk, and Mihai Anitescu, “Real-time stochastic optimization of complex energy systems on high-performance computers,” *IEEE Computing in Science & Engineering*, vol. 16, no. 5, pp. 32–42, 2014.