Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Extended DOF-Handler for BETL2

Bachelor Thesis

F. Hillebrand

June 19, 2017

Advisor: Prof. Dr. R. Hiptmair
Seminar for Applied Mathematics, ETH Zürich

Abstract

This thesis provides implementations for BETL2 that allows for varying sets of degrees of freedom for entities with the same reference element type. This functionality is among others used in hp-finite element methods.

The new implementations provide documentation using Doxygen and are outlined again in this thesis. Furthermore, their correctness has been tested using an example of linear and quadratic Lagrangian finite elements as well as hp-Lagrangian based on the results of the Bachelor's Thesis 'Dirichlet Boundary Value Problem on Deformed Domains' by Christian Baumann [10].

A small performance assessment in comparison to the previous implementation is also given.

Acknowledgements

I'd like to thank Christian Baumann for giving me access to his source code, answering questions and allowing me to use his mesher as well as compare his results to mine.

Contents

Contents v

1 Introduction 1
1.1 Motivation 1
1.2 Remarks 1

2 Implementation 3
2.1 Finite Element Basis 4
2.1.1 `FEBasis` 6
2.1.2 `FEBasisWrapper` 6
2.2 Degree-of-Freedom-Handler 7
2.2.1 Containers for Degrees of Freedom 7
2.3 Finite Element Space 10
2.3.1 Filtering Degrees of Freedom 12
2.3.2 Linear Combination 12

3 Examples 15
3.1 Linear Lagrangian Finite Elements 15
3.1.1 Problem Description 15
3.1.2 Validation 16
3.1.3 Performance 16
3.2 Quadratic Lagrangian Finite Elements 16
3.2.1 Problem Description and Correctness 16
3.2.2 Validation 17
3.2.3 Performance 18
3.3 hp-Lagrangian Finite Elements 19
3.3.1 Problem Description 19
3.3.2 Meshes 20
3.3.3 Solving the BVP 20
3.3.4 Validation 22

4 List of Classes and Files **25**

5 Conclusion **27**

Bibliography **29**

Chapter 1

Introduction

1.1 Motivation

Different variants of the finite element method, such as hp-FEM, necessitate the functionality that cells of the same type may differ in their local set of degrees of freedom. In BETL2 [1], the task of handling the degrees of freedom is taken care of by the classes `DofHandler`, `FESpace` and `FEBasis`. None of which support this functionality.

The task of this Bachelor's thesis is to extend BETL2 such that cells of the same type may also differ in their local set of degrees of freedom. However, old classes are not replaced since they provide a more efficient data layout. Furthermore, the new classes are tested using different examples (based on Lagrangian finite elements) and a in-line documentation for them is provided using Doxygen [4].

1.2 Remarks

BETL2 provides finite element spaces for $H(\text{curl}, \Omega)$ and $H(\text{div}, \Omega)$ besides Lagrangian finite elements for $H^1(\Omega)$. In this thesis only functionalities for Lagrangian finite elements have been considered and tested.

Plots appearing in this thesis were created using Python [9] and Matplotlib[8]. Most of the code was written in C++11 [2] and as a reference [3] was used. BETL2 heavily relies on the linear algebra library Eigen [5] (version 3.2.7). CMake is used for easy compilation.

The code for this thesis can be found on GitLab at https://gitlab.ethz.ch/hifabian/Code_DofHandler. It includes an instruction on how to compile the programs. However, this code does not include BETL2 itself.

Chapter 2

Implementation

As mentioned in the introduction, the management of the degrees of freedom, both locally and globally, is primarily handled by the classes `DofHandler`, `FESpace` and `FEBasis`. The classes `DofHandler` and `FEBasis` are directly created respectively defined by the user while `FESpace` is created internally by the `DofHandler`.

In order to understand the working principle of an implementation of a finite element method using BETL2, the three classes are briefly explained and an example to retrieve an instance of the class `FESpace` is shown in Listing 2.1 for the previous implementation and in Listing 2.2 for the implementation provided by this thesis.

Listing 2.1: How to access a `FESpace` object in BETL2 in the old implementation

```
[...]

// Using Lagrangian finite element space with piecewise linear basis
    functions
using febasis_t = betl2::fe::FEBasis< betl2::fe::Linear,
    betl2::fe::FEBasisType::Lagrange >;

// Creating dofHandler object
using dofHandler_t = betl2::fe::DofHandler< febasis_t,
    betl2::fe::FESContinuity::Continuous, gridViewFactory_t >;
dofHandler_t dh;

dh.distributeDofs(gridVewFactory);

// Retrieving fespace
const auto& fespace = dh.fespace();
```

Listing 2.2: How to access a `FESpace` object in BETL2 in the new implementation

```
[...]

// Creating febasis
using febasis_t = betl2::fe::ex::FEBasis< gridViewFactory_t >;
febasis_t febasis(gridViewFactory);

// Setting febasis as in the linear Lagrangian case
// Default value for a multiplicity is 0
for( auto& e : gridView.template entities<codimVertex>( ) )
  febasis.setMult(e, 1);

// Alternatively in this case use:
// using febasis_t = betl2::fe::ex::FEBasisWrapper<
    betl2::fe::Linear, betl2::fe::FEBasisType::Lagrange,
    gridViewFactory_t >;
// febasis_t febasis;


// Creating dofHandler object
using dofHandler_t = betl2::fe::ex::DofHandler< febasis_t,
    gridViewFactory_t >;
dofHandler_t dh;

dh.distributeDofs(gridViewFactory, febasis);

// Retrieving fespace
const auto& fespace = dh.fespace();
```

The purpose of the class `DofHandler` is to distribute the degrees of freedom on a grid and give access to an instance of the class `FESpace`. The class `FEBasis` determines the number of degrees of freedom associated with a given reference element type of an entity for the old implementation and for the new implementation associated with an entity. The `FESpace`, after being created by the `DofHandler`, is used in the code to traverse the grid and to access degrees of freedom.

The next sections will discuss the purposes and functionalities of these classes and their replacements in more detail. One main reason why varying sets of local degrees of freedom are not supported lies with the `FEBasis`. It will be explained first followed by the `DofHandler` then `FESpace`.

2.1 Finite Element Basis

As mentioned, in the previous implementation the class `FEBasis` provides access to the degrees of freedom for any given reference element type. Its specific implementation in BETL2 provides in addition access to basis functions and the corresponding differential operator applied to them. List-

ing 2.3 shows a possible implementation of a class for a finite element basis in BETL2 based on the previous implementations.

Listing 2.3: Generic form of a valid FEBasis

```
class GenericFEBasis {
public:
  typedef eth::base::unsigned_t size_type;

  // Type of finite element space
  static const betl2::fe::FEBasisType feBasisType( );

  // Number of degrees of freedom associated
  // with a reference type
  template< eth::base::RefElType RET >
  static constexpr size_type multiplicity( );

  // Number of degrees of freedom associated
  // with a reference type including its subentities
  template< eth::base::RefElType RET >
  static constexpr size_type numDofs( );

  // Runtime support for multplicity
  static size_type multiplicity( const eth::base::RefElType );
  // Runtime support for numDofs
  static size_type numDofs( const eth::base::RefElType );
};
```

The fact that it is the reference type of an entity that decides the number of degrees of freedom and not the entity itself, is one of the reason BETL2 did not support a varying set of degrees of freedom for entities with the same reference element type. A generic replacement for the class FEBasis is given by 2.4.

Listing 2.4: Generic form of a valid new FEBasis

```
class GenericFEBasis {
public:
  typedef eth::base::unsigned_t size_type;

  // A way to set the number of degrees of freedom
  template< int CODIM >
  void setMult( const entity_t<CODIM>&, size_type );

  // Type of finite element space
  static const betl2::fe::FEBasisType feBasisType( );

  // Number of degrees of freedom associated with an entity
  template< int CODIM >
  size_type multiplicity( const entity_t<CODIM>& ) const;
```

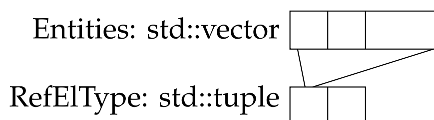

Figure 2.1: Illustration of the storage of the number degrees of freedom. The `std::tuple` stores all reference element types while the `std::vector` stores the number of degrees of freedom for each entity of that reference type.

```
15
16    // Number of degrees of freedom associated with an element
          including its subentities
17    size_type numDofs( const element_t& ) const;
18  };
```

This thesis provides two default implementations: `FEBasis` in the namespace `ex` and `FEBasisWrapper` in the same namespace.

2.1.1 `FEBasis`

The class `FEBasis` in the namespace `ex` provides a function `setMult` for setting the degrees of freedom associated with an entity. It stores the number of degrees of freedom using a class `NumDofDataSet` whose structure is similar to `DataSet` discussed in 2.2.1. Figure 2.1 provides an illustration.

An entity is identified using the index set provided by the grid view. Determining the index can, however, be quite slow due to the function `lexical_cast` called from asserts. This can be remedied by setting the macro `NDEBUG`. This effect is again discussed in chapter 3.

It should be noted that the function `numDofs` is only supported for elements and not all entities. This is done because it saves some memory and because the function is only used for elements internally. The computation for the total number of degrees of freedom of elements can be done on the fly.

2.1.2 `FEBasisWrapper`

The class `FEBasisWrapper` provides a wrapper for the original class `FEBasis`. It provides both member functions listed in 2.3 and in 2.4 and provides an interface for `FEBasis` in the new implementation while also supporting old functionalities.

There is one key difference in the implementation of `FEBasisWrapper` and the old `FEBasis` connected with how BETL2 handles a piece-wise constant basis. In the previous implementation in the case of constant Lagrangian elements the degree of freedom belonging to an element is associated with every entity within that element. As later discussed in section 2.2, BETL2

handles this case differently while the new implementation does not. Because of this, `FEBasisWrapper` uses a specialization for the constant case where it associates the degree of freedom solely with the element (entity with co-dimension 0) but not its sub-entities.

2.2 Degree-of-Freedom-Handler

In BETL2, the class `DofHandler` is responsible for distributing the degrees of freedom, represented by a class `Dof`, on the elements of a grid and giving access to a finite element space object. It differentiates between a continuous and discontinuous space. In most cases only the continuous case is used since it allows for elements to share degrees of freedoms. In the discontinuous case all degrees of freedom are associated with only one element. This is de facto a special case of the continuous space where degrees of freedom solely lie on entities with co-dimension 0. Because of this, in the new implementation only the continuous space is implemented. This, however, leads to some complications already discussed in 2.1.2.

The class `DofHandler` inherits from a class called `DofDistributionPolicy` which differentiates between the continuous and discontinuous space. The continuous case first distributes degrees of freedom on vertices, followed by all other entities. After having distributed the degrees of freedom on all entities, they are associated with elements via pointers.

The classes `DofHandler` and `DofDistributionPolicy` in the namespace `ex` replace their counterparts. The distribution of the degrees of freedom is handled as in the continuous case of the old implementation. The only difference is that the storage is no longer fixed size and has to be adjusted for the number of degrees of freedom on each entity as well as for each element.

2.2.1 Containers for Degrees of Freedom

Figure 2.2 shows the storage layout. On the left side is the container for the degrees of freedom. It is first divided manually by co-dimension, then in the `DataSet` further by reference element type, followed by a list of entities and lastly an array for the degrees of freedom on an entity represented by the object `Dof`. On the right side we have the pointers for the elements. This time we only have entities of co-dimension 0 (triangles and quadrilaterals for 2 dimensions) which are stored in `std::vectors`. For each entity, or element, we then have again an array but this time with pointers to the container of the degrees of freedom. It is done this way to account for shared degrees of freedom. A concrete example is given in Figure 2.3.

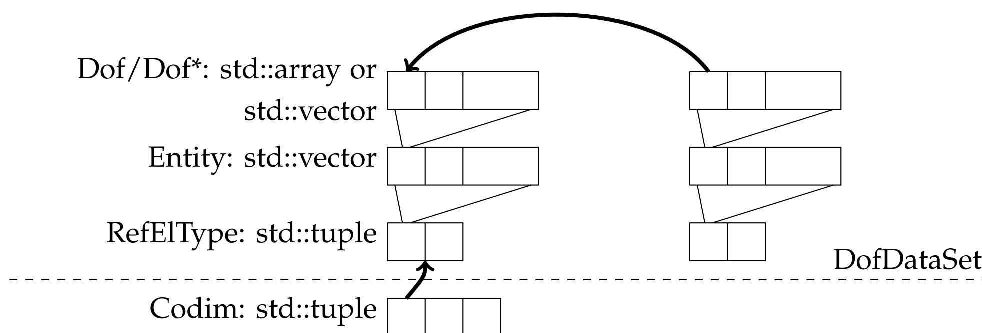

Figure 2.2: Illustration of the storage for the new and the old continuous case

In the case of a varying set of local degrees of freedom one could remove the lowest layer of `DataSet` but this leads to various problems connected to the index set of the grid view. This index set is used to identify an entity. However, it is based on reference element types and not co-dimension, i.e. only entities that share the same reference element type belong to one index set.

The storage for the degrees of freedom was originally handled using the class `GridDataSet` provided by ethGenericGrid. This class uses something it calls `MultiplicityPair` to determine the number of degrees of freedom for a given reference type. Due to the homogeneous number of degrees of freedom, it can use fixed-size array for storage of specific entities.

These fixed-size array do not work in the extended case and are replaced with dynamically allocated arrays (using `std::vector`) in the class `DataSet`. No special function is provided to resize the arrays but they can be resized by retrieving the array itself and then manually employing a resize. This works well since they are only resized while distributing degrees of freedom. 2.5 outlines the functionalities supported by `DataSet`. The implementation is mostly identical to ethGenericGrid's `GridDataSet`.

Listing 2.5: An outline of DataSet

```
1  template< class DATA_TYPE, class GRID_VIEW_FACTORY_T,
       eth::base::RefElType... REFS >
2  class DataSet {
3  public:
4    // Some typedefs
5    [...]
6
7    // Constructor
8    DataSet(gridViewFactory_t gridViewFactory);
9
10   // Data at index i of entity e
11   template< int CODIM >
```

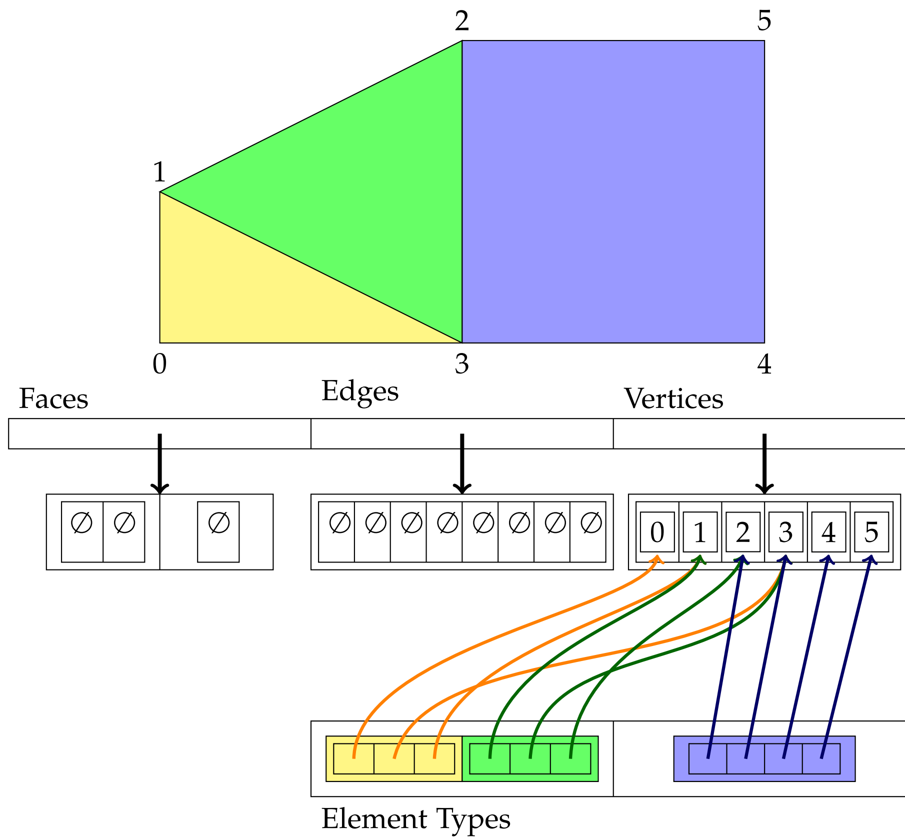

Figure 2.3: Concrete example of the storage for the degrees of freedom. Only vertices have degrees of freedom and each vertex has exactly one thereof. ∅ indicates an empty container.

```
12    dataType& data( const entity_t< CODIM >& e, size_type i );
13    template< int CODIM >
14    const dataType& data( const entity_t< CODIM >& e, size_type i )
          const;
15
16    // Data at index i of a subentity with index subIndex within entity
          e
17    template< int CODIM >
18    dataType& data( const entity_t< CODIM >& e, size_type subIndex,
          size_type i );
19    template< int CODIM >
20    const dataType& data( const entity_t< CODIM >& e, size_type
          subIndex, size_type i ) const;
21
22    // Data of entity e (can employ resize() here)
23    template< eth::base::RefElType RET >
24    std::vector<dataType_t>& data( const entity_t< dimMesh -
          eth::base::ReferenceElement<RET>::dimension >& e );
```

```
25 };
```

The classes GridDataSet and DataSet just provide general containers suitable for a grid. The concrete definitions of the data sets used for the degrees of freedoms are determined via classes DofDataSetFactory. Their explanations are omitted because they do not add much.

2.3 Finite Element Space

The class FESpace is the workhorse of solving a finite element problem using BETL2. Besides the afore-mentioned use to traverse the mesh and to access the degrees of freedom, it also stores the degrees of freedom and provides filter functions to access indices for degrees of freedom lying on intersections. An outline of the functions provided by the class FESpace is given by 2.6.

Listing 2.6: Functions provided by FESpace

```
1  template< class FE_BASIS_T, class GRID_VIEW_FACTORY_T, class
       LINEAR_COMBINATION >
2  class FESpace {
3  [...]
4  public:
5
6    // Returns container of stored dofs
7    const dofDataSet_t& dofsOnElements( ) const;
8    // Returns the grid factory
9    const GRID_VIEW_FACTORY_T& gridFactory( ) const
10
11   // Returns the space's linear combination
12   const linearCombination_t& linearCombination( ) const;
13   linearCombination_t& linearCombination( );
14
15   // Returns true if the space is continuous
16   constexpr static bool isContinuous( );
17
18   // Iterator to beginning of element collection
19   inline const_element_iterator begin( ) const;
20   // Iterator to end of element collection
21   inline const_element_iterator end( ) const;
22
23   // Iterator to beginning of degrees of freedom on an element
24   inline const_dof_iterator begin( const element_t& ) const;
25   inline dof_iterator begin( const element_t& );
26   // Iterator to end of degrees of freedom on an element
27   inline const_dof_iterator end( const element_t& ) const;
28   inline dof_iterator end( const element_t& );
29
```

```
30   // Global index of a degree of freedom
31   inline size_type globalIndex( const const_dof_iterator& ) const;
32   // Local index of a degree of freedom
33   inline size_type localIndex( const const_dof_iterator&, const
         element_t& ) const;
34
35   // Renumber a degree of freedom
36   void renumberDof( const element_t&, const IndexPair<size_type>&,
         size_type );
37
38   // Filter local indices of degrees of freedom given an intersection
         index
39   // Only entities with codimension CODIM are filtered
40   template< int CODIM >
41   inline std::vector< int > filter( const element_t&, int ) const;
42   // Filter all entities
43   inline std::vector< int > filterIndices( const element_t&, int )
         const;
44
45   // Filter degrees of freedom of all entities given an intersection
         index
46   std::vector< Dof* > filterAll( const element_t& e, int );
47   std::vector< const Dof* > filterAll ( const element_t&, int ) const;
48
49   // Get pairs of local-global indices for an element
50   std::vector< IndexPair<size_type> > indices( const element_t& e )
         const;
51   // Get pairs of local-global indices for an element given an
         intersection index
52   std::vector< IndexPair<size_type> > indices( const element_t& e,
         const int ) const;
53
54   // Number of degrees of freedom
55   size_type numDofs( ) const;
56   // Number of elements
57   size_type numElements( ) const;
58 };
```

The class itself did not change much. The new implementation FESpace in the namespace ex simply provides an extra function getFEBasis that gives access to the FEBasis object. However, classes used by some of its member functions needed to be adjusted: FilterDofs and DefaultLinearCombination. Both will be discussed in the following subsections.

BETL2 provides ways to only access constrained finite element spaces with the class ConstrainedFESpace. This class should be able to be used in place of FESpace but due to this new function will not be. Because of this, dummy replacements have been provided for it and for its dependent classes (which

do not change) in the namespace `ex`.

2.3.1 Filtering Degrees of Freedom

The class `FilterDofs` provides an `operator()` that takes as an input an element, an intersection index and a constant reference to a finite element space and returns an array of local indices of degrees of freedom belonging to the intersection. In order to find the local indices of the degrees of freedom, it needs to count up to the entities associated with the intersection. In the old implementation the reference type of an entity determined the number of degrees of freedom and therefore they could easily be summed up for an offset. In the new implementation every entity needs to be inspected and added separately to an offset. The change is quite simple but still necessitates a new class in the namespace `ex`.

2.3.2 Linear Combination

The term linear combination here refers to the enforcement of the linearity of basis functions across edges or faces and is here only discussed for Lagrangian finite elements.

For Lagrangian finite elements, this is a problem that only arises for cubic or higher order polynomials. Figure 2.4 illustrated what is happening: The edge between the two triangles is orientated differently within each triangle causing that the ordering of local indices does not match up, i.e. the global indices are swapped with respect to both triangles. However, note that only the ordering of local indices determines which basis function are associated with which degrees of freedom (the finite element basis class solely stores the number of degrees of freedom). Due to this, the local element matrix and the local element vector need to be permuted to match.

This is done by the class `DefaultLinearCombination` which creates a permutation matrix and can be applied to local element matrices and element vectors using the in BETL2 provided classes `ImposeLinearCombinations` and `ImposeLinearCombination` respectively.

BETL2 provides this in the Lagrangian case only for cubic polynomials in 2 dimensions. To generalize this (but still only for 2 dimensions), a new class `DefaultLinearCombination` was created in the namespace `ex` that permutes any edge that has the wrong orientation regardless of the number of degrees of freedom on it. Of course, in the case of none or one degree of freedom nothing changes.

Currently BETL2 does not allow a user to set the linear combination class for a constrained finite element space and is instead forced to use the default implementation. Since the constrained finite element already has a dummy

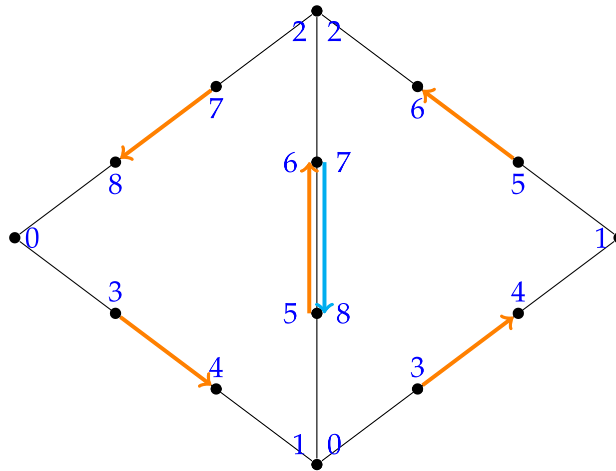

Figure 2.4: The edge between the triangles is orientated differently for the two triangles causing the local indices to be increasing in one triangle and decreasing in the other

replacement due to the missing function, this bug does not show. Nevertheless, the dummy replacement implements a proposed, simple fix. The fix necessitates however that any passed finite element space object provides a type definition `linearCombination_t`.

Chapter 3

Examples

Three examples were looked at to test the correctness of the implementation: Linear Lagrangian finite elements, quadratic Lagrangian finite elements and hp-Lagrangian finite elements. The former two are compared to the previous implementation in BETL2 using the specialised versions. The latter is compared to the results of [10].

In addition to correctness, performance of the new implementations are assessed for the former two cases. To that end, the sparse matrix solver was omitted as it adds a constant time to both implementation and can heavily depend on the algorithm used. In addition, the creation of the `gridFactory` object is also ignored. The only parts considered are the creation of the the classes `FEBasis`, `DofHandler`, `FESpace` and the assembly of both the Galerkin matrix and the right-hand-side vector. This is done to emulate a realistic implementation. However, it should be noted that the assembly of local element matrices and vectors also add a constant time which can dominate the performance assessment.

Everything has been run on a Lenovo laptop 64 bits with Intel®Core™i5-4210U CPU @ 1.70GHz.

All meshes used here where either created using GMSH[6] version 2.10.1 or converted to a GMSH-like file.

3.1 Linear Lagrangian Finite Elements

3.1.1 Problem Description

For linear Lagrangian finite elements the boundary value problem (BVP) given by

$$\begin{aligned} u(\mathbf{x}) - \Delta u(\mathbf{x}) &= 1, \quad \mathbf{x} \in \Omega \\ \nabla u(\mathbf{x}) \cdot \mathbf{n} + u(\mathbf{x}) &= 0, \quad \mathbf{x} \in \partial\Omega \end{aligned}$$

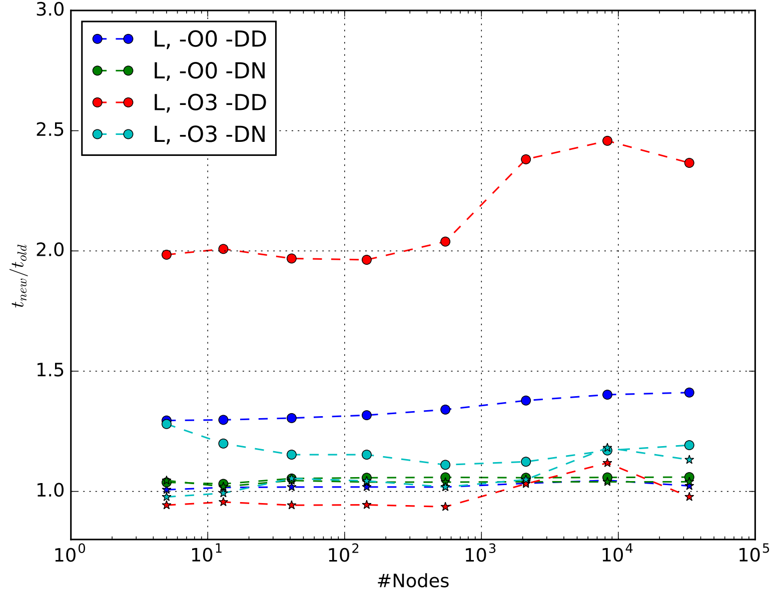

Figure 3.1: Relative performance using different compiler flags averaged over 6 runs (Compiler: `g++ (5.4.0) -std=c++11`)

where $\Omega \subset \mathbb{R}^2$ is solved. This equivalent to the weak form

$$u \in H^1(\Omega) : \int_\Omega uv + \nabla u \cdot \nabla v \mathrm{d}\mathbf{x} + \int_{\partial\Omega} uv \mathrm{d}S = \int_\Omega v \mathrm{d}\mathbf{x}, \quad \forall v \in H^1(\Omega)$$

which is taken from exercise 7 problem 2 and 4 from the lecture [11].

In the case of a piecewise linear approximations for v and u on a triangular mesh, we can solve these integrals analytical as shown in [11, 3.6.5.1].

3.1.2 Validation

The new implementation does not change the order of the degrees of freedom compared to the old implementation thanks to their similarity in distributing the degrees of freedom. Because of this, the resulting vectors can directly be subtracted. This shows that results of both implementation match up to machine precision validating that the implementation is correct for the case of one degree of freedom per vertex.

3.1.3 Performance

Figure 3.1 shows the ratio of averaged (using the arithmetic mean over 6 runs) runtime of the different implementations. The lines using the dots are for the class `FEBasis` while the stars are for the class `FEBasisWrapper`.

As hinted at in an earlier section, there is a significant difference in the ratio of runtime when the macro `NDEBUG` is set (-N) compared to when not (-D). An in-depth analysis using GPROF [7] revealed that this is due to calls to the boost function `lexical_cast` that is performed by asserts in the new implementation of `FEBasis` among others.

Without debugging and no optimization both implementation perform equally well. However, setting optimization on improves the old implementation better than the new albeit not much ending with a ratio of around 1.25. This is not significantly slower than the previous implementation.

The difference disappears almost completely when using the wrapper finite element basis indicating that it is mainly the implementation of the new class `FEBasis` that slows down the code and not the class `DataSet`.

3.2 Quadratic Lagrangian Finite Elements

3.2.1 Problem Description and Correctness

For quadratic Lagrangian finite elements the same BVP

$$\begin{aligned} u(\mathbf{x}) - \Delta u(\mathbf{x}) &= 1, \quad \mathbf{x} \in \Omega \\ \nabla u(\mathbf{x}) \cdot \mathbf{n} + u(\mathbf{x}) &= 0, \quad \mathbf{x} \in \partial\Omega \end{aligned}$$

where $\Omega \subset \mathbb{R}^2$ and with the weak form

$$u \in H^1(\Omega) : \int_\Omega uv + \nabla u \cdot \nabla v \mathrm{d}\mathbf{x} + \int_{\partial\Omega} uv \mathrm{d}S = \int_\Omega v \mathrm{d}\mathbf{x}, \quad \forall v \in H^1(\Omega)$$

is solved.

In the case of piecewise quadratic approximations of u and v on a triangular mesh, the integrals are solved using a quadrature rule provided by BETL2. This is more compute intensive than the previous example where the integrals are solved analytically over the triangles. In addition, more basis-functions and thus more entries have to be computed.

3.2.2 Validation

Again the difference between the two resulting vectors can directly be computed and the error is around machine precision. Thus, we conclude that the implementation is correct.

3.2.3 Performance

Again we look at the performance. It still hold that the dots represent the finite element basis using the class `FEBasis` in the namespace `ex` while the

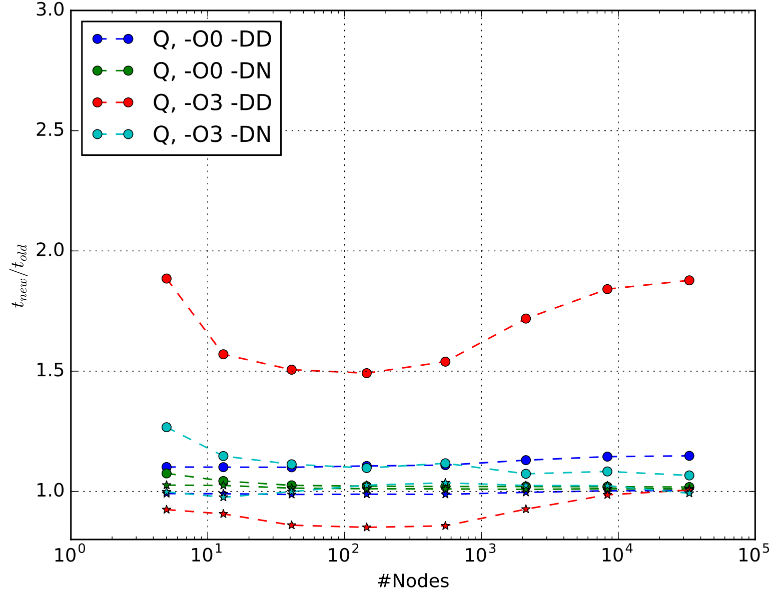

Figure 3.2: Relative performance using different compiler flags averaged over 6 runs (Compiler: `g++ (5.4.0) -std=c++11`)

stars are for `FEBasisWrapper`. -N stands for the macro `NDEBUG` while -D stands for `DEBUG`. Two different optimization settings are compare: None and O3.

Similar behaviour as to 3.2 is observed but the difference is slightly lower. This can be attributed to the already mentioned higher computational intensive local element matrix and vector which add a larger constant time factor.

It also appears that in the case of -D with O3 the new implementation outperforms the previous one. This is most likely due to some skips of asserts in `DataSet` and thus calling `lexical_cast` less.

3.3 hp-Lagrangian Finite Elements

3.3.1 Problem Description

In this example the Dirichlet boundary value problem as described by [10] is solved. For convenience it is restated here:

$$
\begin{aligned}
-\Delta u(\mathbf{x}) &= 0, \quad \mathbf{x} \in \Omega \\
u(\mathbf{x}) &= g(\mathbf{x}), \quad \mathbf{x} \in \partial\Omega
\end{aligned}
$$

where $g \in H^1(\Omega)$ is a given function.

The domain Ω is described by its boundary as

$$\partial\Omega := \Big\{ r(\varphi)\big(\sin\varphi, \cos\varphi\big)^\mathsf{T} \subset \mathbb{R}^2, \quad 0 \leq \varphi \leq 2\pi \Big\}$$

where $r : [0, 2\pi] \to \mathbb{R}$ is a continuous functions given by the real Fourier series

$$r(\varphi) := 1 + \sum_{j=1}^{N} \Big(c_j y_j \cos(j\varphi) + s_j z_j \sin(j\varphi) \Big)$$

with parameters $-1 \leq y_j, z_j \leq 1$ and

$$\sum_{j=1}^{\infty} \Big(|c_j| + |s_j| \Big) \leq \frac{1}{2}$$

Using a mapping approach this problem can be transformed to the unit disc $B_1 \subset \mathbb{R}^2$ using the diffeomorphism $\Phi : B_1 \to \Omega$ defined as

$$\Phi(\hat{\mathbf{x}}) := \hat{\mathbf{x}} + \mathbf{w}(\hat{\mathbf{x}})$$

and by the auxiliary problem

$$\begin{aligned} -\Delta \mathbf{w}(\hat{\mathbf{x}}) &= 0, \quad \hat{\mathbf{x}} \in B_1 \\ \mathbf{w}(\hat{\mathbf{x}}) &= \begin{pmatrix} \big(r(\varphi) - 1\big)\cos\varphi \\ \big(r(\varphi) - 1\big)\sin\varphi \end{pmatrix}, \quad \hat{\mathbf{x}} \in \partial B_1 \end{aligned}$$

with φ being the polar coordinate on ∂B_1.

The weak forms of these two BVPs are given by

$$\begin{aligned} u \in H^1(\Omega) : \int_\Omega \nabla u \cdot \nabla v \mathrm{d}\mathbf{x} = 0, \quad \forall v \in H^1_0(\Omega) \\ u = g \quad \text{on} \quad \partial\Omega \end{aligned}$$

and

$$\begin{aligned} w_1 \in H^1(B_1) : \int_{B_1} \nabla w_1 \cdot \nabla v \mathrm{d}\mathbf{x} = 0, \quad \forall v \in H^1_0(B_1) \\ w_2 \in H^1(B_1) : \int_{B_1} \nabla w_2 \cdot \nabla v \mathrm{d}\mathbf{x} = 0, \quad \forall v \in H^1_0(B_1) \\ \mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} \big(r(\varphi) - 1\big)\cos\varphi \\ \big(r(\varphi) - 1\big)\sin\varphi \end{pmatrix} \quad \text{on} \quad \partial B_1 \end{aligned}$$

A derivation can be found in [10].

3.3.2 Meshes

[10] also provides meshes. These meshes are defined using the classes `mesh`, `element` and a matrix `nodes`. In order to use these meshes in BETL2, a converter file `meshConverter.cpp` was created that takes a mesh object and a node matrix and writes as a Gmsh-like file.

The original meshes store the order p of an element as well as if a degree of freedom is missing or not ($m \in \{0,1\}$). This is replaced using a physical tag given by $p^2 - m$. Furthermore, the meshes are structured such that a missing degree of freedom is located on the first edge. This is also true in the converted mesh. This last property is especially useful for determining the basis-functions and building the local element matrices.

3.3.3 Solving the BVP

The first problem concerning the displacement **w** is solved in a similar fashion as the previous two examples. The final problem however uses isoparametric elements which are given by the displacement **w** and the basis-functions.

BETL2 does not support isoparametric elements but one can easily pass the displacements as arguments to the local assemblers and then explicitly incorporate it into the computations. The implementation of the local stiffness matrix is shown in Listing 3.1.

Listing 3.1: Local stiffness matrix

```
1  // RET          Reference element type of the element
2  // POINTS       Number of interpolation points on the element (dofs)
3  // BUILDER_DATA Class that provides extra information (for the
       displacements)
4  // ELEMENT_T    Class of the element
5  template< eth::base::RefElType RET, int POINTS, typename
       BUILDER_DATA, class ELEMENT_T >
6  struct LocStiffMatIso
7  {
8    static const int size = POINTS; // Size of matrix
9    static const int quad = 16;  // Quadrature rule
10   static const int dim = 2;    // Dimension of mesh
11
12   // Returns the local stiffness matrix
13   inline static matrix_t<size, size> eval(const BUILDER_DATA& data,
         const ELEMENT_T& el)
14   {
15       // Get geometry object and element area
16       const auto& geom = el.geometry();
17
18       // Get displacements of dofs of this element
```

```
19      const auto& w = data.w_;
20
21      // Result via integration
22      matrix_t<size,size> result;
23      result.setZero();
24
25      // Get quadrature rule
26      typedef betl2::quad::Quadrature< RET, quad > quadRule_t;
27      // Get quadrature points and weights (scaled)
28      const auto& xi = quadRule_t::getPoints();
29      const auto& wi = quadRule_t::getWeights() *
            quadRule_t::getScale();
30
31      // Get graident basis function evaluation at quadrature points
32      using gradFuncts = GradFun< RET, size >;
33      const auto gradEval = gradFuncts::Eval( xi );
34
35      // Transformed interpolation points (dofs)
36      Eigen::MatrixXd nodes = geom.global(detail::ReferencePoints<
            RET, POINTS >::get());
37      nodes += w.transpose(); // Shift nodes
38
39      // Determinants of Jacobians at quadrature points: Dphi' = sum(
            (dof)*gradBasis.transpose() )
40      matrix_t<1,quad> detJi;
41      // Jacobians transposed
42      matrix_t<dim,dim*quad> JT; JT.setZero();
43      // Compute Jacobians and their determinants:
44      // Loop over all xi
45      for( int i = 0; i < quad; ++i ) {
46        // Loop over dofs
47        for( int d = 0; d < size; ++d) {
48            JT.template block<dim,dim>(0, i*dim) +=
49              (nodes.col(d) * gradEval.template
                  block<1,dim>(d,i*dim)).transpose();
50        }
51        detJi(i) = (JT.template block<dim,dim>(0,
              i*dim)).determinant();
52      }
53
54      // Compute coefficients
55      const auto coeff = detJi.cwiseProduct( wi );
56
57      for( int i = 0; i < xi.cols(); ++i ) {
58        // Get gradient of basis function for current xi_i
59        const matrix_t<size,dim> g_i = gradEval.template block<
              size,dim >(0, i*dim)
60                         * (JT.template block< dim,dim >(0,
                             dim*i)).transpose().inverse();
```

Table 3.1: Results of $u^\intercal Au$

Results from [10]	Results from this thesis	Error
4.11729656262623	4.11729656326183	$6.356 \cdot 10^{-10}$
4.81167689406564	4.81294668956745	$1.270 \cdot 10^{-3}$

```
61          // Add to result
62          result += coeff(i) * g_i * g_i.transpose();
63        }
64        // Return result
65        return std::move(result);
66    }
67  }; // end struct LocStiffMatIso
```

Because of the varying degrees of freedom and varying order of polynomials, basis-functions have to be determined and implemented. Unlike [10], this implementation only supports up to cubic polynomials. The basis-functions were determined by solving a linear system of equations if BETL2 did not provide them already. [10] lists analytical formulas for both the basis-functions and their gradients. However, one should keep in mind that BETL2 uses a different convention for both the reference elements and the numbering the degrees of freedom.

3.3.4 Validation

In order to validate the results, only small deformation ([10, 5.2.1]) were looked at and only the first two meshes were used. Unlike in the previous two examples, the vectors cannot directly be subtracted. Instead, the energy norms (computed as $u^\intercal Au$ where A is the Galerkin matrix and u is the solution vector) are compared. These are not direct results presented in [10] but can easily be computed by the provided code.

The obtained norms are compared against each other in Table 3.1. The differences arise from the different quadrature rules that were used. This thesis uses a very high but constant quadrature rule compared to the ones used in [10] where an adaptive choice was made. Higher or lowering either quadrature rules will yield the same results.

Chapter 4

List of Classes and Files

This chapter provides a list of files and classes that have been implemented in this thesis.

- dof_distribution_policy_extended.hpp
  - DofDistributionPolicy
- ex_data_set.hpp
  - DataSet
  - DataSetTraits
- ex_dof_data_set_factories.hpp
  - DofDataSetFactory
- ex_dof_handler.hpp
  - DofHandler
- ex_febasis.hpp
  - FEBasis
  - FEBasisTraits
- ex_fespace.hpp
  - FESpace
- ex_filter_dofs.hpp
  - FilterDofs
- ex_linear_combination.hpp
  - DefaultLinearCombination
- febasis_wrapper.hpp
  - FEBasisWrapper
  - FEBasisWrapperTraits
- num_dof_data_set.hpp
  - NumDofDataSet
- num_dof_data_set_factories.hpp
  - NumDofDataSetFactory

Chapter 5

Conclusion

This chapter will briefly repeat the most important results of this thesis and provide an outlook for further work.

Implementations to extend BETL2 to allow for varying sets of local degrees of freedom have been successfully provided for the Lagrangian case. They have been tested using 2 dimensional linear, cubic and hp-Lagrangian finite element spaces. The new implementations while still being slower than the optimised versions for fixed numbers of degrees of freedom can still keep up with them.

A documentation for these new implementations has been provided in-line using Doxygen. However, the rest of BETL2 still lacks most of its documentation.

In addition, various functionalities provided by BETL2 are not supported for these new implementations. To name a few important ones:

- `betl2::DofInterpolator`
- `betl2::InterpolationGridFunction`
- Support for `Div`, `Curl`, `LagrangeHierachical`
- Access to basis functions and reference interpolation points

Bibliography

[1] BETL2 - Boundary Element Template Library 2. http://www.sam.math.ethz.ch/betl/. accessed June 2017.

[2] C++. https://isocpp.org. accessed June 2017.

[3] C++ reference. http://en.cppreference.com/w/. accessed June 2017.

[4] Doxygen. http://www.stack.nl/~dimitri/doxygen/. accessed June 2017.

[5] Eigen. http://eigen.tuxfamily.org/. accessed June 2017.

[6] Gmsh. http://gmsh.info. accessed June 2017.

[7] Gprof. https://www.gnu.org/software/binutils/. accessed June 2017.

[8] Matplotlib. http://matplotlib.org. accessed June 2017.

[9] Python. https://www.python.org. accessed June 2017.

[10] Christian Baumann. Dirichlet Boundary Value Problems on Deformed Domains. Bachelor's thesis, ETH Zürich, 2017. https://gitlab.com/chbauman/Bachelor_Thesis_Christian_Baumann.

[11] Ralf Hiptmair. Numerical Methods for Partial Differential Equations. http://www.sam.math.ethz.ch/~hiptmair/tmp/NPDE/NPDE16.pdf, 2016. accessed June 2017.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

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

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

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

Title of work (in block letters):

Extended DOF-Handler for BETL2

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Hillebrand	Fabian

With my signature I confirm that

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

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

Place, date

Oberarth, 04.07.2017

Signature(s)

[signature]

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