



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

PAUL SCHERRER INSTITUT



---

# BUILDING BLOCKS FOR FINITE ELEMENT COMPUTATIONS IN IPPL

---

## BACHELOR THESIS

in Computational Science and Engineering

Department of Mathematics

ETH Zurich

written by

LUKAS BÜHLER

supervised by

Dr. A. Adelmann (ETH & PSI)

scientific advisers

Sonali Mayani (PSI)

Dr. M. Frey (University of St Andrews)

Dr. S. Muralikrishnan (Jülich Supercomputing Center)

February 3, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Methodology</b>	<b>2</b>
2.1	Finite Element Method (FEM)	2
2.1.1	Variational formulation	3
2.1.2	Discretization (Meshing)	4
2.2	Matrix-free Method for FEM computations	5
2.3	Independent Parallel Particle Layer (IPPL)	6
2.4	Arguments against using an external FEM library	6
2.5	Focus and Scope of this Thesis	7
<b>3</b>	<b>Implementation</b>	<b>8</b>
3.1	Software Architecture	8
3.2	Finite Element Spaces, Meshes and Indexing	10
3.2.1	Mesh vertex and element index mapping	10
3.2.2	The FiniteElementSpace Class	10
3.3	Elements and Transformations	12
3.3.1	Defining the Reference Elements	12
3.3.2	Transformations	13
3.3.3	Element classes	15
3.4	Lagrangian Finite Element Methods	16
3.4.1	Lagrangian shape functions	16
3.4.2	LagrangeSpace class	17
3.5	Quadrature	20
3.5.1	Gaussian-Quadrature	20
3.5.2	Gauss-Jacobi Quadrature	23
3.5.3	Tensor-product Quadrature	23
3.6	Matrix-free Cell-oriented Assembly Algorithm with Local Computations	26
3.6.1	Matrix-free Assembly Algorithm for the Stiffness Matrix	26
3.6.2	Assembly Algorithm for the Load Vector	28
3.6.3	Essential Boundary conditions	29
3.7	Solver	31
3.8	Results	32
<b>4</b>	<b>Conclusion</b>	<b>36</b>
4.1	Future Work	37

---

<b>Appendix A Resources and Details</b>	<b>41</b>
A.1 Resources . . . . .	41
A.2 IPPL Build Instructions . . . . .	41
A.3 Documentation Build Instructions . . . . .	43
A.4 Plot Generation . . . . .	44
A.4.1 Lagrangian finite element space plots . . . . .	44
A.4.2 Convergence plots . . . . .	44
<b>Appendix B Implementation Helpers</b>	<b>45</b>
B.1 Higher-order Lagrangian finite elements . . . . .	45
B.2 Non-homogeneous Dirichlet boundary conditions . . . . .	45
B.3 Other Finite Elements and their spaces . . . . .	46
B.4 Bilinear Transformations . . . . .	47
B.5 Periodic boundary conditions . . . . .	47

# List of Figures

3.1	Software architecture of the FEM framework, showing the classes with their template arguments. . . . .	9
3.2	IPPL FEM framework source file structure. . . . .	9
3.3	Illustration of mesh vertex indexing and element indexing. . . . .	10
3.4	Illustration of the local vertex numbering on the reference edge element (1D). . .	12
3.5	Illustration of the local vertex numbering on a quadrilateral element (2D). . . .	12
3.6	Illustration of the local vertex numbering on a hexahedral element (3D). . . . .	13
3.7	Illustration of a 2D transformation (scaling and translation only). . . . .	14
3.8	First-order Lagrange shape functions. . . . .	17
3.9	First-order Lagrangian shape functions in 2D. . . . .	18
3.10	Second-order Lagrange shape functions. . . . .	19
3.11	Third-order Lagrange shape functions. . . . .	19
3.12	2D Tensor-product of 7-point Gauss-Legendre quadrature nodes on the unit-square.	24
3.13	1D Sinusoidal convergence . . . . .	33
3.14	2D Sinusoidal convergence . . . . .	34
3.15	3D Sinusoidal convergence . . . . .	35

# Chapter 1

## Introduction

Many fundamental physics equations governing natural phenomena cannot be solved analytically. However, using computer simulations we can solve complex equations numerically. This process can be accelerated using supercomputers. Supercomputers, or computer clusters, underwent many evolutionary changes in pursuit of more performance, measured by the number of computations per second, more precisely, the number of floating point operations per second (FLOP/s). In the past few years, there was a race to the first exascale supercomputer, a supercomputer that is capable of calculating at least  $10^{18}$  IEEE 754 double precision (64 bit) floating point operations per second (Exaflop/s). The term ‘exascale computing’ refers to computing systems being capable of that. In June of 2022, the Frontier system, located at Oak Ridge National Laboratory (ORNL), achieved a High-Performance Linpack (HPL) benchmark score of 1.102 Exaflop/s and was crowned the first true exascale machine [1].

Moving to the era of exascale, heterogeneous computing architectures, with different kinds of processors and/or possibly different kinds of GPUs in the same system, are unavoidable. Because of this, we seek to provide tools to computational physicists to carry out extreme-scale simulations on the next generation of computing architectures. The IPPL (Independent Parallel Particle Layer) library [2] is a performance portable framework for Particle-In-Cell (PIC) methods in C++. It is being developed by collaborators from Paul Scherrer Institute (PSI), Jülich Supercomputing Center (JSC) and the University of St Andrews. Currently, IPPL supports electrostatic PIC simulations. Development of a full electromagnetic (EM) solver is ongoing.

EM solvers are useful for simulating Free Electron Lasers, where the radiation interacts with free electrons, whose motions are governed by the electromagnetic Maxwell equations.

The goal of this thesis is to explore one of the numerical methods used for EM solvers, the Finite Element Method (FEM) and implement the building blocks for it. The thesis is structured as follows: First, we present the methodology and the software framework, then we cover the implementation of the building blocks and finally we discuss the result with convergence plots, followed by our conclusion and a few words on future work.

# Chapter 2

## Methodology

For electromagnetic (EM) simulations, it is common to use the Finite Difference Time Domain (FDTD) scheme [3]. The FDTD scheme discretizes the time and space derivatives with central finite differences, hence, it only has 2nd-order accuracy.

Another method that can be used to solve the Maxwell equations is the Finite Element Method (FEM). When solving EM problems with FEM, the space discretization is done using FEM and other schemes such as Runge-Kutta methods are used to solve in the time domain. This is known as Finite Element Time Domain (FETD).

FEM has multiple advantages compared to finite differences. It allows for more complex geometries and higher-order elements (i.e. higher-order basis functions for the finite element spaces) to improve accuracy. Using higher-order elements to improve the accuracy is known as  $p$ -refinement. The higher-order elements allow FEM to achieve better accuracy without having to refine the mesh and therefore improve the accuracy without really affecting the runtime, scalability and memory footprint. It is also possible to refine the mesh, just as in the finite difference scheme, which is known as  $h$ -refinement. It is even possible to do both with  $hp$ -refinement.

Furthermore, a matrix-free assembly algorithm can be used, which gives the method an even smaller memory footprint and thus further improves the performance on GPUs.

In this work, we show an implementation of a Finite Element Method framework in IPPL [2] with a matrix-free assembly algorithm and the possibility for  $p$ -refinement.

### 2.1 Finite Element Method (FEM)

The Finite Element Method (FEM) is a method to solve partial differential equations (PDEs). To use it, boundary value problems need to be formulated into a variational boundary problem (weak form). After that, the FEM consists of three main steps: Firstly, discretizing or ‘meshing’ the domain with a finite number of ‘elements’. Secondly, approximating the solution of the PDE on each of the elements and finally assembling the solutions from the elements in a linear system of equations (LSE). This LSE can then be solved with any LSE solver. FEM produces sparse problems, which is why sparse LSE solvers are often used. In this work, however, we use the conjugate gradient algorithm to solve the LSE instead, since this allows us to avoid having to explicitly build the left-hand-side matrix (matrix-free). For more details on the matrix-free method, see section 2.2.

### 2.1.1 Variational formulation

**Definition 1** (Linear form). *Given a vector space  $V$  over  $\mathbb{R}$ , a linear form  $\ell$  is a mapping  $\ell : V \mapsto \mathbb{R}$  that satisfies [4, Def. 0.3.1.4]*

$$\ell(\alpha u + \beta v) = \alpha \ell(u) + \beta \ell(v), \quad \forall u, v \in V, \quad \forall \alpha, \beta \in \mathbb{R}. \quad (2.1)$$

**Definition 2** (Bilinear form). *Given a vector space  $V$  over  $\mathbb{R}$ , a bilinear form  $a$  on  $V$  is a mapping  $a : V \times V \mapsto \mathbb{R}$ , for which it holds [4, Def. 0.3.1.4]*

$$\begin{aligned} a(\alpha_1 v_1 + \beta_1 u_1, \alpha_2 v_2 + \beta_2 u_2) = \\ \alpha_1 \alpha_2 a(v_1, v_2) + \alpha_1 \beta_2 a(v_1, u_2) + \beta_1 \alpha_2 a(u_1, v_2) + \beta_1 \beta_2 a(u_1, u_2), \end{aligned} \quad (2.2)$$

$$\forall u_i, v_i \in V, \alpha_i, \beta_i \in \mathbb{R}, i = 1, 2.$$

**Definition 3** (Continuous Linear Form). *Given a normed vector space  $V$  with norm  $\|\cdot\|$ , a linear form  $\ell : V \rightarrow \mathbb{R}$  is continuous on  $V$  if [4, Def 1.2.3.42]*

$$\exists C > 0 : \quad |\ell(v)| \leq C \|v\| \quad \forall v \in V, \quad (2.3)$$

holds for  $C \in \mathbb{R}$ .

**Definition 4** (Continuous Bilinear Form). *Given a normed vector space  $V$  with norm  $\|\cdot\|$ , a bilinear form  $a : V \times V \rightarrow \mathbb{R}$  on  $V$  is continuous, if [4, Def 1.2.3.42]*

$$\exists K > 0 : \quad |a(u, v)| \leq K \|u\| \|v\| \quad \forall u, v \in V_0, \quad (2.4)$$

holds for  $K \in \mathbb{R}$ .

**Definition 5** ((Galerkin) Linear Variational Problem (LVP)). *We define a linear variational problem (LVP) as [4, Def. 1.4.1.7]*

$$u \in V : \quad a(u, v) = \ell(v) \quad \forall v \in V, \quad (2.5)$$

where  $V$  is a vector space, with norm  $\|\cdot\|_V$ ,  $a : V \times V \mapsto \mathbb{R}$  is a continuous bilinear form and  $\ell : V \mapsto \mathbb{R}$  is a continuous linear form. The function  $u \in V$  is called the trial function and the functions  $v \in V$  are called the test functions.

In a generalized linear variational problem, the test and trial functions can be from different vector spaces, defined on the same subspace. Then these two vector spaces are called the test space and the trial space. However, in this work, we are only considering Galerkin Finite Element Methods where the test and the trial space are the same vector space, by definition.

#### Example: Poisson Equation

The strong form of the Poisson equation with homogeneous Dirichlet boundary conditions is

$$\begin{aligned} -\Delta u &= f & u &\in \Omega, \\ u &= 0 & u &\in \partial\Omega, \end{aligned} \quad (2.6)$$

where  $\Omega$  is the domain,  $\partial\Omega$  is the boundary of the domain  $\Omega$  and  $f : \Omega \mapsto \mathbb{R}$  is the source function.

The weak-form or variational equation of the Poisson equation with homogeneous Dirichlet boundary conditions is

$$\int_{\Omega} \nabla u \cdot \nabla v \, d\mathbf{x} = \int_{\Omega} f v \, d\mathbf{x}, \quad (2.7)$$

where  $v \in V$  is the test function and  $u \in V$  is the trial function, and  $f : \Omega \rightarrow \mathbb{R}$  is the source function. The derivation can be found in [4].

The bilinear form and the linear of this problem are therefore

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, d\mathbf{x}, \quad (2.8)$$

$$\ell(v) = \int_{\Omega} f v \, d\mathbf{x}. \quad (2.9)$$

### 2.1.2 Discretization (Meshing)

Recall the definition of the linear variational problem (Def. 5):

$$u \in V : \quad a(u, v) = \ell(v) \quad \forall v \in V, \quad (2.5)$$

where  $a : V \times V \mapsto \mathbb{R}$  is the bilinear form and  $\ell : V \mapsto \mathbb{R}$  is the linear form,  $u$  is the trial function and  $v$  are the test functions.

The first part of the discretization (meshing) step is the replacement of the infinite-dimensional vector space  $V$  in the linear variational problem with a finite-dimensional subspace  $V_h \subset V$  [4, Chapter 2.2.1].

**Definition 6** (Discrete (linear) variational problem (DVP)). *The discrete variational problem (DVP) is defined as [4, Def. 2.2.1.1]*

$$u_h \in V_h : \quad a(u_h, v_h) = \ell(v_h), \quad \forall v_h \in V_h, \quad (2.10)$$

where  $u_h$  is the discretized trial function or Galerkin solution,  $v_h$  are the discretized test functions,  $a : V_h \times V_h \rightarrow \mathbb{R}$  is a continuous bilinear form and  $\ell : V_h \rightarrow \mathbb{R}$  is a continuous linear form.

The next step in the Galerkin Discretization is the definition of the basis functions for the discrete variational problem.

We choose an ordered basis  $\{b_h^1, \dots, b_h^N\}$  of  $V_h$  with  $N := \dim V_h$  where for every  $v_h \in V_h$  there are unique coefficients  $\nu_i \in \mathbb{R}, i \in \{1, \dots, N\}$ , such that  $v_h = \sum_{i=1}^N \nu_i b_h^i$  [4, Def. 0.3.1.2].

Inserting this basis representation into the variational equation 2.10 yields:

$$v_h \in V_h \Rightarrow v_h = \nu_1 b_h^1 + \dots + \nu_N b_h^N, \quad \nu_i \in \mathbb{R}, \quad (2.11)$$

$$u_h \in V_h \Rightarrow u_h = \mu_1 b_h^1 + \dots + \mu_N b_h^N, \quad \mu_i \in \mathbb{R}, \quad (2.12)$$

where the number  $N$  is the dimension of the discrete vector space  $V_h$  and  $\nu_i, \mu_i \in \mathbb{R}, i \in \{1, \dots, N\}$  are unique coefficients.

The basis functions also have to satisfy the cardinal basis property, as given by definition 7.

**Definition 7** (Cardinal basis property).

$$b_h^j(\mathbf{x}_i) = \begin{cases} 1 & i = j, \\ 0 & \text{else} \end{cases}, \quad i, j \in \{1, \dots, N\}. \quad (2.13)$$

Note: In this work, we refer to the basis functions on the global vector space as the basis functions and to the basis functions restricted to an element  $K$  as the *shape functions*.



**Definition 8** (Shape function). *For an element  $K$ , we define the shape functions as the basis functions, restricted to the element  $K$ :*

$$b_K^i := b_{h|K}^i. \quad (2.14)$$

Note: The letter  $K$  for the shape function  $b_K^i$  may be omitted where the context is clear. Inserting the definitions of the basis functions into the variational equation

$$a(u_h, v_h) = \ell(v_h) \quad \forall u_h, v_h \in V_{0,h}, \quad (2.15)$$

$$\sum_{k=1}^N \sum_{j=1}^N \mu_k \nu_j a(b_h^k, b_h^j) = \sum_{j=1}^N \nu_j \ell(b_h^j) \quad \forall \nu_1, \dots, \nu_N, \mu_1, \dots, \mu_N \in \mathbb{R}, \quad (2.16)$$

$$\sum_{j=1}^N \nu_j \left( \sum_{k=1}^N \mu_k a(b_h^k, b_h^j) - \ell(b_h^j) \right) = 0 \quad \forall \nu_1, \dots, \nu_N, \mu_1, \dots, \mu_N \in \mathbb{R}, \quad (2.17)$$

$$\sum_{k=1}^N \mu_k a(b_h^k, b_h^j) = \ell(b_h^j) \quad \text{for } j = 1, \dots, N. \quad (2.18)$$

**Definition 9** (Stiffness matrix). *The stiffness matrix (or Galerkin matrix) is the matrix of the bilinear form evaluations defined as:*

$$\mathbf{A} = \left[ a(b_h^j, b_h^i) \right]_{i,j=1}^N, \in \mathbb{R}^{N,N}. \quad (2.19)$$

**Definition 10** (Load vector). *The load vector (or right-hand-side vector) is the vector of linear form evaluations defined as:*

$$\varphi = [\ell(b_h^i)]_{i=1}^N \in \mathbb{R}^N. \quad (2.20)$$

To compute each entry of the stiffness matrix and load vector, the integrals from the bilinear form and linear form are approximated. This is done with numerical quadrature (numerical integration).

We arrive at the linear system of equations (LSE):

$$\mathbf{A}\boldsymbol{\mu} = \varphi. \quad (2.21)$$

## 2.2 Matrix-free Method for FEM computations

In 2017, Ljungkvist showed that matrix-free finite element algorithms have many benefits on modern manycore processors and graphics cards (GPUs) compared to alternative sparse matrix-vector products [5]. Additionally, in 2023, Settgaest et. al showed that in the context of the conjugate gradient (CG) method, the matrix-free approach compares favorably even for low-order FEM [6].

Motivated by those findings, we chose to implement a matrix-free assembly algorithm that integrates into the conjugate gradient algorithm, which is already implemented in IPPL.

The CG method is an iterative method and terminates once the norm of the residual is smaller than the specified tolerance  $\epsilon \in \mathbb{R}_{>0}$ . The pseudocode for the CG algorithm in IPPL is given in Algorithm 1.

```

 $\mathbf{x} \leftarrow$  initial guess, (usually  $\mathbf{0}$ )
 $\mathbf{b} \leftarrow \varphi$ 
 $\mathbf{p} \leftarrow \mathbf{A}\mathbf{x}$ 
 $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{p}$ 
while  $\|\mathbf{r}\|_2 < \epsilon$  do
     $\mathbf{z} \leftarrow \mathbf{A}\mathbf{p}$ 
     $\alpha \leftarrow \frac{\mathbf{r}^\top \mathbf{r}}{\mathbf{p}^\top \mathbf{z}}$ 
     $\mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{p}$ 
     $\mathbf{r}_{\text{old}} \leftarrow \mathbf{r}$ 
     $\mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{z}$ 
     $\beta \leftarrow \frac{\mathbf{r}^\top \mathbf{r}}{\mathbf{r}_{\text{old}}^\top \mathbf{r}_{\text{old}}}$ 
     $\mathbf{p} \leftarrow \mathbf{r} + \beta \mathbf{p}$ 
end

```

**Algorithm 1:** Pseudocode of the CG algorithm implemented in IPPL.

In the CG method, the stiffness matrix  $\mathbf{A}$  is used in the initialization and in the while loop itself, and in both cases, it is multiplied with a vector. The implemented FEM framework provides an assembly function to replace this matrix-vector product  $\mathbf{A}\mathbf{x}$ . The assembly function will return the same resulting vector, but without building the full stiffness matrix  $\mathbf{A}$ , thus being “matrix-free”.

More detailed information regarding the implementation of this assembly algorithm is given in section 3.6.

## 2.3 Independent Parallel Particle Layer (IPPL)

The Independent Parallel Particle Layer (IPPL) [2] [7] is a performance portable C++ library for Particle-Mesh methods. It is a portable, massively parallel toolkit using the Message Passing Interface (MPI) for inter-processor communication, HeFFTe [8] as a Fast Fourier Transform (FFT) library and Kokkos [9] for hardware portability.

## 2.4 Arguments against using an external FEM library

The Finite Element Method has been studied in great detail already and has been implemented for many languages, environments and use cases. There are also many C++ FEM libraries, for example, MFEM [10] and deal.II [11].

The possibility of interfacing IPPL with an external library to introduce the Finite Element Method was considered, but it was decided against for several reasons. Firstly, new dependencies can bring problems in the future and they further complicate the compilation and installation of IPPL. Secondly, almost all external libraries use non-standard or custom datatypes. Since the performance of IPPL on supercomputers is very important, it is crucial to avoid data copies, which would occur when using a different data type to support an external library, as they incur high data movement costs, especially on GPUs.

## 2.5 Focus and Scope of this Thesis

The focus of this Bachelor thesis is the software design and implementation of the building blocks for the Finite Element Method in the IPPL [2] library. The focus was on structured, rectilinear grids with rectangular hexahedral (“brick”) elements and the possibility for  $p$ -refinement.

The term ‘building blocks’ refers to the different parts that are necessary for a functioning FEM implementation. For example, a class for numerical integration (quadrature), classes for mesh and degree of freedom (DOF) index mapping and assembly functions for building the resulting linear system of equations.

The framework implemented in this thesis supports first-order Lagrangian finite elements in one to three dimensions with a matrix-free assembly function that interfaces directly into the CG algorithm. It supports the basic midpoint quadrature rule and the polynomial Gauss-Jacobi quadrature rule. For essential boundary conditions, only homogeneous Dirichlet boundary conditions are supported at the moment.

Additionally, a solver to solve the Poisson equation was implemented as a proof-of-concept.

## Chapter 3

# Implementation

In this chapter, the first section (3.1) will go into the general organization and software architecture of the FEM framework in IPPL. The following sections will go into the specifics of the FEM building blocks and their implementation.

### 3.1 Software Architecture

From a software architecture perspective, the FEM framework was designed to have high modularity and is easily extensible as a result. A quick overview is given in the Unified Modeling Language (UML) diagram in figure 3.1.

The main interface for all the building blocks is the abstract `FiniteElementSpace` class. It has a reference to the mesh, the reference element and the quadrature rule that is used in the assembly.

The IPPL `Mesh` class is abstract as well and other mesh classes like the IPPL `UniformMesh` class inherit from it. At the moment, only rectilinear meshes are supported in IPPL.

The reference element class is abstract and it defines the transformation functions. It has pure virtual functions to force its child classes to implement the functions to support these transformations. Implemented classes that inherit from it are the `EdgeElement`, `QuadrilateralElement` and `HexahedralElement` classes.

The `Quadrature` class is also abstract and implements the functions for transforming 1D quadrature weights and nodes to tensor product quadrature nodes and weights on the reference elements. At the moment there are two different quadrature rules implemented that inherit from the `Quadrature` base class. The `MidpointQuadrature` rule that was implemented for testing and the `GaussJacobiQuadrature` class which implements the Gauss-Jacobi quadrature rule.

The `LagrangeSpace` class inherits from the abstract `FiniteElementSpace` base class and defines the degree of freedom (DOF) operations and assembly functions for the Lagrangian finite elements.

The `FEMSolver` class is used to showcase the FEM framework. It solves the Poisson equation using first-order Lagrangian finite element methods with the existing conjugate gradient algorithm from IPPL and the Gauss-Jacobi quadrature rule.

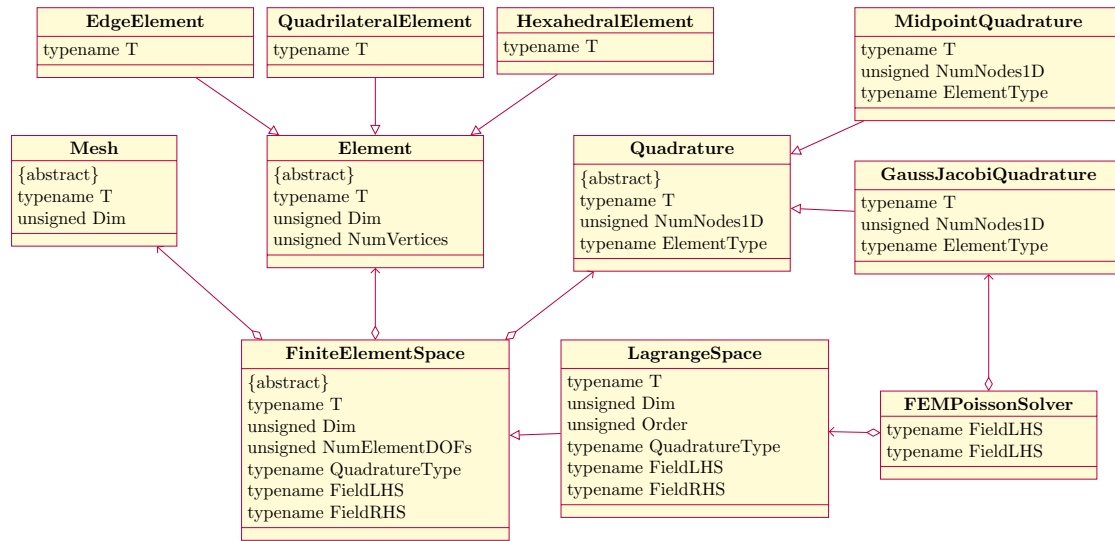


Figure 3.1: Software architecture of the FEM framework, showing the classes with their template arguments.

```

ippl/
├── src/
│   └── FEM/
│       ├── Elements/
│       │   ├── EdgeElement.h
│       │   ├── EdgeElement.hpp
│       │   ├── Element.h
│       │   ├── Element.hpp
│       │   ├── HexahedralElement.h
│       │   ├── HexahedralElement.hpp
│       │   ├── QuadrilateralElement.h
│       │   └── QuadrilateralElement.hpp
│       ├── Quadrature/
│       │   ├── GaussJacobiQuadrature.h
│       │   ├── GaussJacobiQuadrature.hpp
│       │   ├── MidpointQuadrature.h
│       │   ├── MidpointQuadrature.hpp
│       │   ├── Quadrature.h
│       │   └── Quadrature.hpp
│       ├── CMakeLists.txt
│       ├── FiniteElementSpace.h
│       ├── FiniteElementSpace.hpp
│       ├── LagrangeSpace.h
│       └── LagrangeSpace.hpp
  
```

Figure 3.2: IPPL FEM framework source file structure.

## 3.2 Finite Element Spaces, Meshes and Indexing

To implement the Finite Element Method, functions are needed to map mesh vertices to elements and vice-versa. In our implementation, this is implemented in the `FiniteElementSpace` class. It is also the base class for the `LagrangeSpace` class that implements Lagrangian finite elements. The `FiniteElementSpace` class is an abstract class that implements the mesh helper functions and also declares pure virtual member functions for degree-of-freedom, reference element shape function evaluations and assembly functions.

### 3.2.1 Mesh vertex and element index mapping

At this time, IPPL implements only structured, rectilinear grid meshes. In FEM, each mesh cell is assigned an element. The elements are indexed starting from zero (C++ indexing) in this implementation.

To define the position of a vertex in the structured rectilinear grid mesh, we define a vector that has  $N$  entries for an  $N$  dimensional mesh with each entry representing the position of a mesh cell in the  $n$ -th dimension. In the implementation, the type alias for this vector is called `NDIndex`, following the naming convention from IPPL.

For example, in figure 3.3 the `NDIndex` of vertex 13 would be  $(3, 2)$ . The element `NDIndex` of  $(3, 1)$  corresponds to the element with index 7.

Figure 3.3 illustrates the mesh vertex indexing (black) and element indexing (red). Each vertex index is associated with the vertex `NDIndex`, as well as each element associated with the element `NDIndex`.

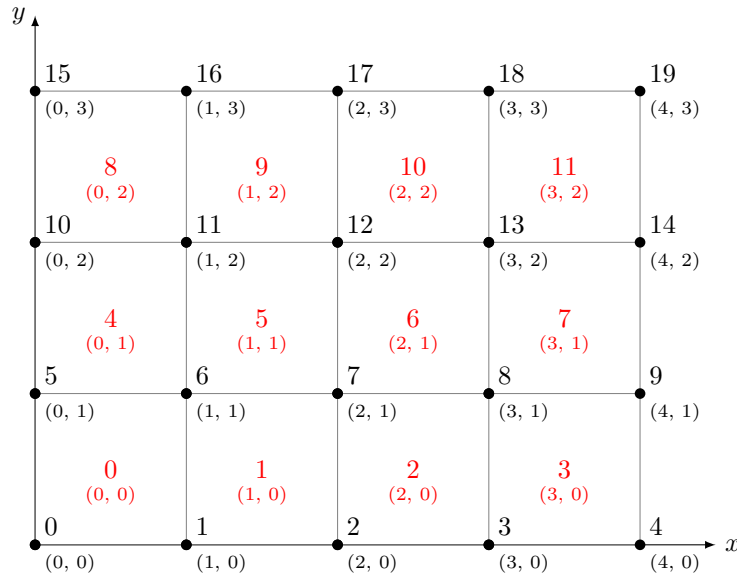


Figure 3.3: Illustration of mesh vertex indexing and element indexing.

### 3.2.2 The FiniteElementSpace Class

The `FiniteElementSpace` class takes in template arguments for the floating point type `T`, the dimension `Dim` and the number of degrees of freedom on a reference element `NumElementDOFs`

the quadrature rule type `QuadratureType` and the two types for the left-hand-side IPPL `Field` and the right-hand-side IPPL `Field`.

This class declares additional pure virtual member functions in the header file, that child classes need to define. These functions are for degrees of freedom, reference element shape functions evaluations and assembly operations and they will be introduced in section 3.4.2.

### 3.3 Elements and Transformations

#### 3.3.1 Defining the Reference Elements

In this section the three basic reference elements, for one, two and three dimensions, are defined. They could be described as n-cuboid shapes because they are shaped like the cells of rectilinear structured grids. For consistency with the implementation, the indices start at zero (C++ indexing).

##### Edge Element (1D)

In a one-dimensional mesh, an element is a line segment or an edge. The reference edge element is one-dimensional and starts at 0 and ends at 1. The first vertex with index 0 is at 0, and the second vertex with index 1 is at 1. the indexing is visualized in figure 3.4.

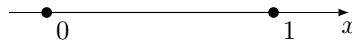


Figure 3.4: Illustration of the local vertex numbering on the reference edge element (1D).

##### Quadrilateral Element (2D)

The 2D structured grid cells can be described by a quadrilateral. For rectilinear grids those cells are rectangular. However, the quadrilateral reference element in this framework aims to describe quadrilaterals, which include parallelograms and by definition every closed, non-degenerate shape with four vertices.

The local vertices are ordered according to the right-hand rule. The first vertex of the reference element is at  $(0,0)$  and has index 0. The second vertex of the reference element is located at  $(1,0)$  and has index 1. The vertex with index 2 is at  $(1,1)$  and the last vertex with index 3 is at  $(0,1)$ . See figure 3.5 for an illustration of the vertex indexing.

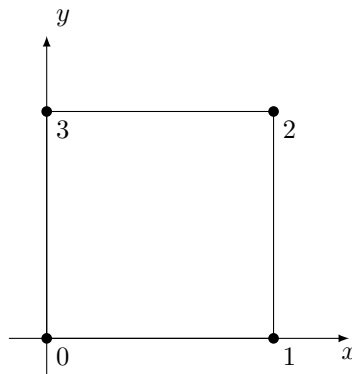


Figure 3.5: Illustration of the local vertex numbering on a quadrilateral element (2D).



### Hexahedral Element (3D)

Three-dimensional structured grids have hexahedral grid cells. In rectilinear structured grids from IPPL, the cells can be described as bricks or rectangular cuboids. The indexing follows the indexing from FEMSTER [12]. See figure 3.6 for an illustration.

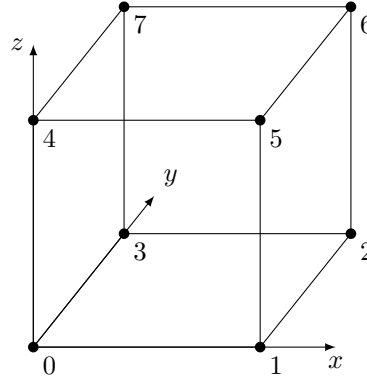


Figure 3.6: Illustration of the local vertex numbering on a hexahedral element (3D).

#### 3.3.2 Transformations

In order to compute the values of the shape functions on the reference element  $\hat{K}$  and transform them back to the global element  $K$ , we need a function that maps the global element to the reference element. For this, we define the transformation function  $\Phi_K$  below.

**Definition 11** (Transformation Function). *The transformation function  $\Phi_K$  maps points from the local coordinate system of the reference element  $\hat{K}$  to the global coordinate system of the global element  $K$ :*

$$\Phi_K : \hat{K} \mapsto K. \quad (3.1)$$

To transform a point  $\hat{\mathbf{x}} \in \hat{K}$  to its point in the global coordinate system  $\mathbf{x} \in K$  we use:

$$\mathbf{x} = \Phi_K(\hat{\mathbf{x}}). \quad (3.2)$$

This is called a local-to-global transformation.

The opposite from  $\mathbf{x} \in K$  to  $\hat{\mathbf{x}} \in \hat{K}$  is the global-to-local transformation using the inverse of the transformation function  $\Phi_K^{-1}$ :

$$\hat{\mathbf{x}} = \Phi_K^{-1}(\mathbf{x}). \quad (3.3)$$

At the time of writing, IPPL only supports rectilinear structured grid meshes for which affine transformations are sufficient. Thus, only affine transformations were implemented, instead of more general bilinear transformations. For more information on how to update this implementation to use bilinear transformations instead, see section B.4 in the Appendix.

The difference between the bilinear and affine transformations is that the affine transformations can only handle transformations to parallelograms whereas bilinear transformations cover all possible transformations for quadrilaterals.

**Definition 12** (Affine Transformation). *An affine transformation combines a linear transformation and a translation and is defined as*

$$\Phi_K(\hat{\mathbf{x}}) = \mathbf{F}_K \hat{\mathbf{x}} + \boldsymbol{\tau}_K, \quad (3.4)$$

where  $\hat{\mathbf{x}}$  is the point to transform in the reference coordinate system,  $\mathbf{F}_K$  is the transformation matrix and  $\boldsymbol{\tau}_K$  is the translation vector.

Also, on a side note, not implementing bilinear transformations means that technically the `QuadrilateralElement` class currently only supports parallelograms instead of all quadrilaterals. Similarly, the `HexahedralElement` does not support transformations to all hexahedrons.

Also, since we assume rectilinear grids, we can go one step further because the linear transformation in the affine transformation can be simplified even more to just a scaling transformation. Thus we can use a transformation matrix as follows:

$$\mathbf{F}_K := \begin{bmatrix} \mathbf{v}_x^1 - \mathbf{v}_x^0 & 0 & 0 \\ 0 & \mathbf{v}_y^2 - \mathbf{v}_y^0 & 0 \\ 0 & 0 & \mathbf{v}_z^4 - \mathbf{v}_z^0 \end{bmatrix}. \quad (3.5)$$

Equation 3.5 is the scaling transformation matrix for three dimensions, where  $\mathbf{v}^i \in \mathbb{R}^d$  is the  $i$ -th local vertex of the element with  $d$  entries for each dimension.

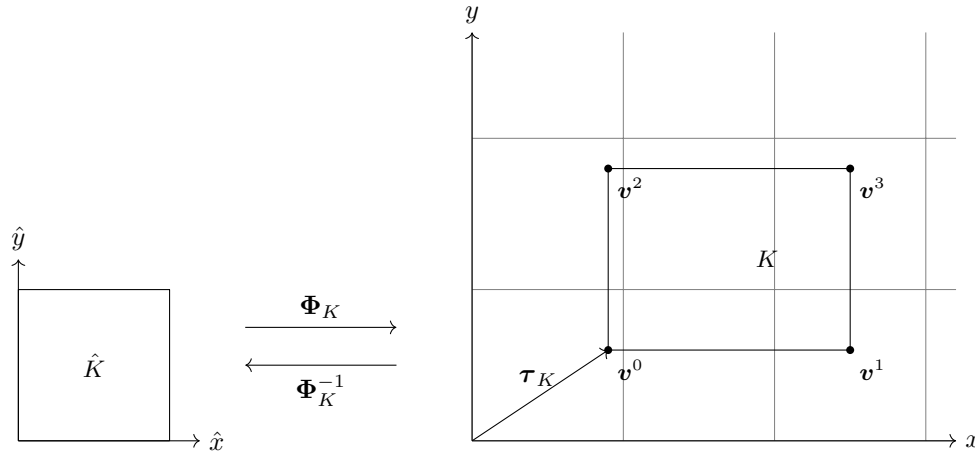


Figure 3.7: Illustration of a 2D transformation (scaling and translation only).

### Applying Affine Transformations

In this subsection, the transformation will be applied to integrals and gradients.

First, we define how to apply the transformation to functions using the *pullback* operation.

**Definition 13** (Pullback). *The pullback  $\Phi_K^* f$  of a function  $f$  is defined as [4]:*

$$(\Phi_K^* f)(\hat{\mathbf{x}}) := f(\Phi_K(\hat{\mathbf{x}})), \quad \hat{\mathbf{x}} \in \hat{K}. \quad (3.6)$$

*It applies the transformation function  $\Phi_K$  to the input of the function  $f$ .*

**Definition 14** (Transformation Jacobian). *The Transformation Jacobian is defined as [4]:*

$$\mathbf{D}\Phi_K(\hat{\mathbf{x}}) = \left[ \frac{\partial(\Phi_K(\hat{\mathbf{x}}))_i}{\partial x_j} \right]_{i,j=1}^d. \quad (3.7)$$

In this work, with the simplifications applied from before, the Jacobian  $\mathbf{D}\Phi_K$  is equal to the transformation matrix:

$$\mathbf{D}\Phi_K(\hat{\mathbf{x}}) = \mathbf{F}_K. \quad (3.8)$$

The pullback can be applied to the gradient of a function as follows:

$$\Phi_K^*(\nabla_{\mathbf{x}} u)(\hat{\mathbf{x}}) = (\mathbf{D}\Phi_K(\hat{\mathbf{x}}))^{-\top} (\nabla_{\hat{\mathbf{x}}}(\Phi_K^* u))(\hat{\mathbf{x}}) \quad (3.9)$$

$$= (\mathbf{D}\Phi_K(\hat{\mathbf{x}}))^{-\top} (\nabla_{\hat{\mathbf{x}}} u)(\Phi_K(\hat{\mathbf{x}})), \quad (3.10)$$

with the notation:  $\mathbf{S}^{-\top} := (\mathbf{S}^{-1})^\top = (\mathbf{S}^\top)^{-1}$ .

Also, the inverse of the Transformation Jacobian is equal to the inverse of the transformation matrix:

$$\mathbf{D}\Phi_K^{-1}(\hat{\mathbf{x}}) = \mathbf{F}_K^{-1}. \quad (3.11)$$

The determinant of the Transformation Jacobian is therefore also equal to the determinant of the transformation matrix:

$$|\det \mathbf{D}\Phi_K(\hat{\mathbf{x}})| = |\det \mathbf{F}_K|. \quad (3.12)$$

### 3.3.3 Element classes

The **Element** class is the base class for all the reference elements. The reference element classes that inherit from it, define the indexing and position of the local vertices. They also define the transformation functions for the local-to-global transformations.

The base **Element** class is abstract and dimension-independent and it takes in arguments for the dimension **Dim**, as well as the datatype **T** and the number of vertices **NumVertices**. The number of vertices is required at compile time to define the size of the IPPL Vector, which also takes the size as a template argument at compile time.

For now, transformations to lower dimensions are not supported. Therefore the dimension template argument has to have the same dimension as the mesh.

### 3.4 Lagrangian Finite Element Methods

In this work, we are looking at Lagrangian FEM on structured rectilinear grids, also known as Tensor-Product Lagrangian FEM [4].

The Lagrangian finite element space defines a set of piecewise continuous polynomials of degree  $p$  over the mesh. They are only piecewise over the element boundaries.

**Definition 15** ((Tensor-product) Lagrangian finite element spaces). *We define the space of  $p$ -th degree Lagrangian finite element functions on a tensor product mesh  $\mathcal{M}$  as [4]:*

$$\mathcal{S}_p^0(\mathcal{M}) := \{v \in C^0(\bar{\Omega}) : v|_K \in \mathcal{Q}_p(K) \ \forall K \in \mathcal{M}\}, \quad (3.13)$$

where  $\bar{\Omega}$  is the domain of the mesh, including the boundary,  $\mathcal{Q}_p(K)$  is the set of basis functions that are tensor products of polynomials of degree  $p$  spanning the element  $K$  and  $v|_K$  are the functions  $v$  restricted to element  $K$ .

The space  $\mathcal{S}_p^0$  contains continuous, piecewise polynomial functions that are made up of “stitched together” basis functions from the elements in mesh  $\mathcal{M}$ .

#### 3.4.1 Lagrangian shape functions

Together with the Lagrangian finite element space, the basis functions have to be defined as well. The basis functions are defined according to the policy of interpolation nodes through the cardinal basis property (see def. 7).

**Definition 16** (Lagrange Polynomials). *Given a set  $\{x_0, \dots, x_n\} \subset \mathbb{R}$  of nodes, the Lagrange polynomials are [13]*

$$L_i(x) := \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}, \quad i = 0, \dots, n. \quad (3.14)$$

The Lagrange interpolating polynomial is the unique polynomial of the lowest degree that interpolates a set of points [13]. They satisfy the cardinal basis property on the interpolation nodes.

To define polynomial functions of degree  $p$  on the elements, Lagrangian FEM utilizes Lagrangian polynomials, hence the name. We have a fixed number of degrees of freedom (DOFs) per element that correspond to the local shape functions of the element. The shape functions satisfy the cardinal basis property that they evaluate to one at the location of the corresponding DOF and to zero at the locations of all the other DOFs.

#### First-order shape functions

In one dimension, first-order Lagrangian shape functions are polynomials of degree 1, or linear polynomials, and interpolate two points. The basis functions for first-order Lagrangian FEM are the well-known tent functions [4]. The shape functions are:

$$\hat{b}^i(x) = \begin{cases} 1 - x & i = 0, \\ x & i = 1. \end{cases} \quad (3.15)$$

See figure 3.8 for a plot of 1D first-order Lagrangian shape functions.

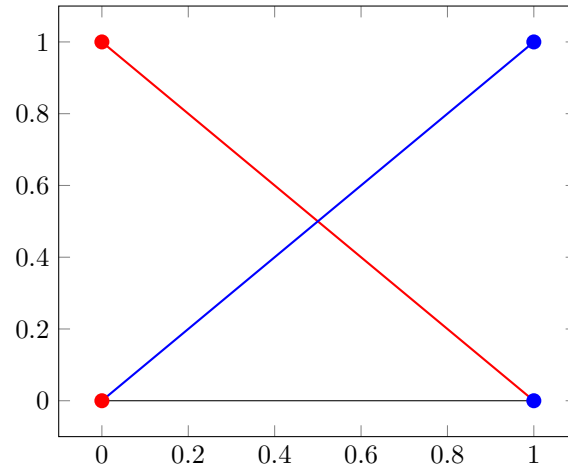


Figure 3.8: First-order Lagrange shape functions.

In two dimensions, the first-order Lagrangian shape functions are the tensor-product of 1D 1st-order shape functions of the y- and the x-dimension:

$$\hat{b}^i(x, y) = \begin{cases} (1-x)(1-y) & i = 0, \\ x(1-y) & i = 1, \\ xy & i = 2, \\ (x-1)y & i = 3. \end{cases} \quad (3.16)$$

See figure 3.9 for a plot of 2D first-order Lagrangian shape functions.

### Higher-order shape functions

Higher-order Lagrangian methods have more degrees of freedom per element. Second-order Lagrangian finite element methods have an additional degree of freedom in the middle of each edge, in two dimensions also a degree of freedom in the middle of each face and in three dimensions also an additional degree of freedom in the middle of each cell.

In 1D the shape functions are polynomials of degree two, specifically, the Lagrangian interpolation polynomials interpolating the points 0,  $\frac{1}{2}$  and 1:

$$\hat{b}^i(x) = \begin{cases} (1-x)(\frac{1}{2}-x) & i = 0, \\ x(1-x) & i = 1, \\ x(\frac{1}{2}-x) & i = 2. \end{cases} \quad (3.17)$$

The definition for even higher-order Lagrangian finite element spaces and shape functions follows from the definition of the second-order Lagrangian shape functions (Eq. 3.17) and the interpolating polynomials for more degrees of freedom.

#### 3.4.2 LagrangeSpace class

The `LagrangeSpace` class inherits from the `FiniteElementSpace` class and overrides and defines the virtual degree of freedom and assembly member functions.

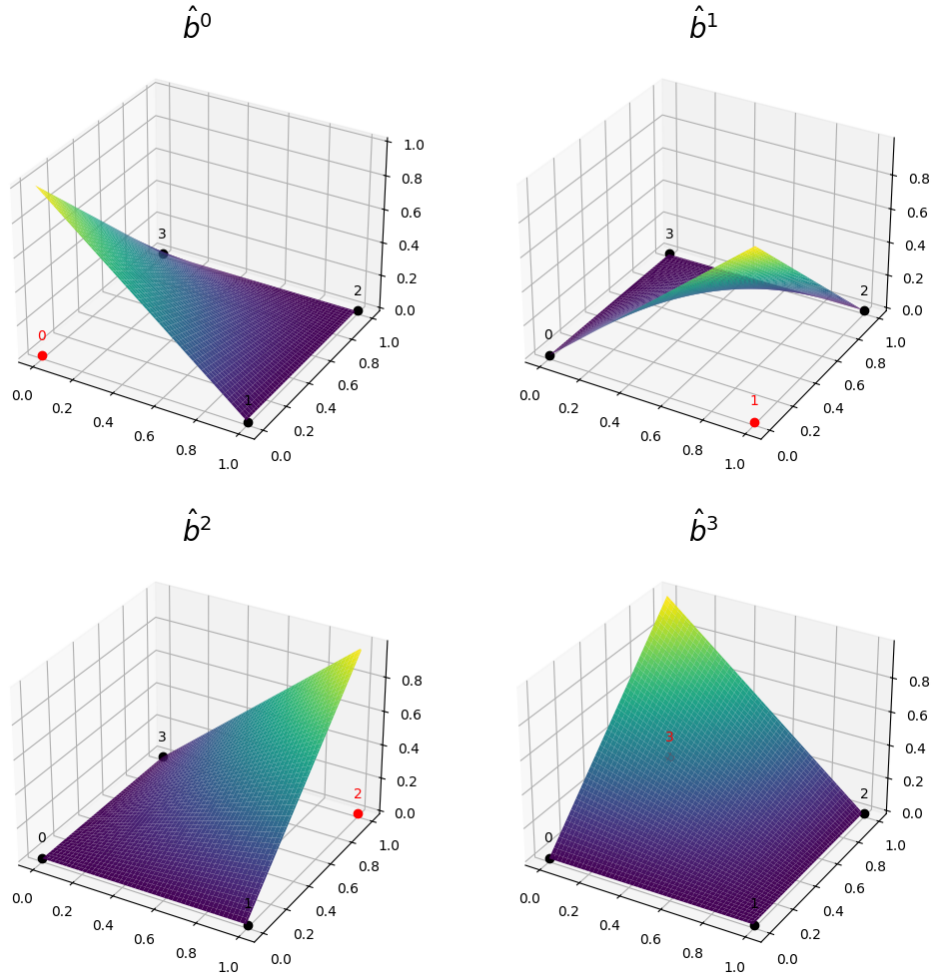


Figure 3.9: First-order Lagrangian shape functions in 2D.

The `LagrangeSpace` class does not represent the Lagrangian finite element space in a mathematical sense. It instead serves as a container for all the DOF and assembly operations corresponding to the rules implied by the Lagrangian finite element space for a specified order.

It takes template arguments for the floating point type `T`, the dimension `Dim`, the order `Order` as well as an argument for the type of the reference element, the type of the quadrature rule and left-hand-side and right-hand-side field types.

To inherit from the `FiniteElementSpace` class, it also has to pass a template argument for the number of element degrees of freedom `NumElementDOFs`, which is computed at compile time from the mesh dimension and the order of the `LagrangeSpace`.

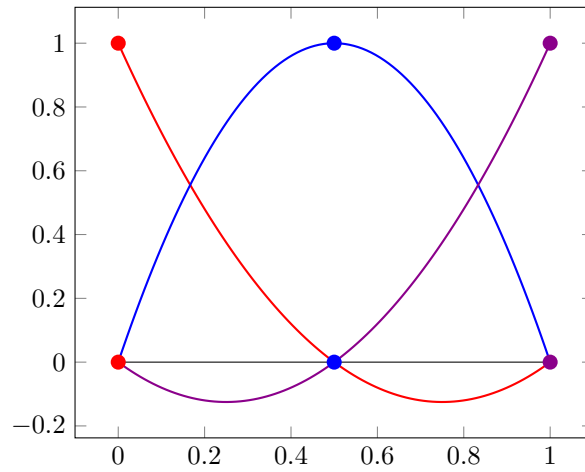


Figure 3.10: Second-order Lagrange shape functions.

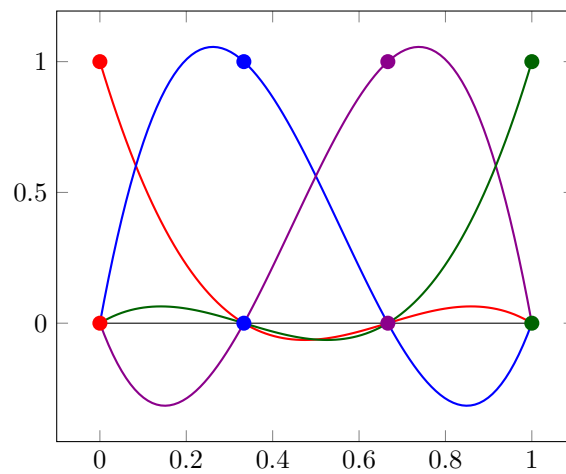


Figure 3.11: Third-order Lagrange shape functions.

## 3.5 Quadrature

Numerical quadrature refers to numerical integration in one dimension.

**Definition 17** (*n*-point Quadrature rule). *An  $m$ -point quadrature rule on  $[a, b]$ ,  $n \in \mathbb{N}$  is defined as [4]*

$$\int_a^b f(t) \, dt \approx \sum_{j=1}^n w_j f(x_j), \quad (3.18)$$

where  $w_j \in \mathbb{R}$  are the quadrature weights and  $x_j \in [a, b]$  are the quadrature nodes.

A quadrature rule defines a way to approximate a definite integral of a function using a weighted sum of the values of the function at specific points, also defined by the quadrature rule, called the quadrature nodes or points. Quadrature rules have a fixed, total number of points  $n$  on which they evaluate the function. For each point  $j$  of the  $n$  points, the rule also defines a weight for the weighted sum.

In this work, numerical quadrature is required to approximate the integral in the computation of the element matrix of the cell-oriented assembly algorithm (see section 3.6).

There are many different quadrature rules, with different properties. To talk about the accuracy of quadrature rules, one can use the *degree of exactness* of a quadrature rule, often abbreviated as just the *degree* of a quadrature rule.

**Definition 18** (Degree of a quadrature rule). *A quadrature rule has degree (of exactness)  $m$  if  $m$  is the maximal degree of the polynomials for which the quadrature rule is guaranteed to be exact [13, Def. 7.4.1.1].*

This means the quadrature rule of degree  $m$  provides exact solutions (instead of just approximations) for all polynomial functions  $f$  with degree lower or equal to  $m$ . For non-polynomial functions or polynomials with a degree greater than  $m$ , it only provides an approximation.

In literature, it is often the order that is used to compare different quadrature rules.

**Definition 19** (Order of a quadrature rule). *The order  $p$  of a quadrature rule of degree  $m$  is: [13, Def. 7.4.1.1]*

$$p = m + 1. \quad (3.19)$$

Some examples of the degrees, and orders, of quadrature rules are:

- Midpoint (Rectangle) rule: Degree 1 (Order 2).
- Trapezoidal rule: Degree 1 (Order 2).
- Simpson rule (3-point): Degree 3 (Order 4).
- $n$ -point Gaussian quadrature rule: Degree  $2n - 1$  (Order  $2n$ ).

### 3.5.1 Gaussian-Quadrature

**Definition 20** (*n*-point Gaussian quadrature rule). *An  $n$ -point Gaussian quadrature rule is a quadrature rule that yields exact results for polynomials of degree  $2n - 1$  or less. It is defined on  $[-1, 1]$  as:*

$$\int_{-1}^1 f(x) \, dx \approx \sum_{i=1}^n w_i f(x_i). \quad (3.20)$$

Therefore, by definition, a Gaussian quadrature rule has degree  $2n - 1$  and order  $2n$ .



The weights  $w_j \in \mathbb{R}$ ,  $j = 1, \dots, n$  of an  $n$ -point quadrature rule on  $[-1, 1]$  are given by [13, Thm. 7.4.1.6]:

$$w_j = \int_{-1}^1 L_{j-1}(t) dt, \quad j = 1, \dots, n, \quad (3.21)$$

where  $L_k$ ,  $k = 0, \dots, n-1$  is the  $k$ -th Lagrange polynomial associated with the ordered set of quadrature nodes  $\{x_1, \dots, x_n\}$ .

### Gauss-Legendre Quadrature

The simplest example of a Gaussian quadrature rule is the Gauss-Legendre quadrature rule, which uses the Legendre polynomials

**Definition 21** (Legendre Polynomials). *The  $n$ -th Legendre polynomial  $P_n$  is defined by [13, Def. 7.4.2.16]:*

$$\int_{-1}^1 P_n(t)q(t) dt = 0 \quad \forall q \in \mathcal{P}_{n-1}, \quad (3.22)$$

where  $\mathcal{P}_{n-1}$  is the set of polynomials of degree smaller or equal to  $n-1$ .

**Definition 22** ( $n$ -point Gauss-Legendre Quadrature). *A  $n$ -point Gauss-Legendre quadrature rule has the nodes at the zeros (roots) of the  $n$ -th Legendre polynomial. The weights are chosen according to equation 3.21.*

$n$	Points $x_i$	Weights $w_i$
1	$x_1 = 0$	$w_1 = 2$
2	$x_1 = -0.57735026918962...$ $x_2 = 0.57735026918962...$	$w_1 = 1$ $w_2 = 1$
3	$x_1 = -0.77459666924148...$ $x_2 = 0$ $x_3 = 0.77459666924148...$	$w_1 = 0.88888888888888...$ $w_2 = 0.55555555555555...$ $w_3 = 0.88888888888888...$
4	$x_1 = -0.86113631159405...$ $x_2 = -0.33998104358485...$ $x_3 = 0.33998104358485...$ $x_4 = 0.86113631159405...$	$w_1 = 0.34785484513745...$ $w_2 = 0.65214515486254...$ $w_3 = 0.65214515486254...$ $w_4 = 0.34785484513745...$
5	$x_1 = -0.90617984593866...$ $x_2 = -0.53846931010568...$ $x_3 = 0$ $x_4 = 0.53846931010568...$ $x_5 = 0.90617984593866...$	$w_1 = 0.23692688505618...$ $w_2 = 0.47862867049936...$ $w_3 = 0.56888888888888...$ $w_4 = 0.47862867049936...$ $w_5 = 0.23692688505618...$
6	$x_1 = -0.93246951420315...$ $x_2 = -0.66120938646626...$ $x_3 = -0.23861918608319...$ $x_4 = 0.23861918608319...$ $x_5 = 0.66120938646626...$ $x_6 = 0.93246951420315...$	$w_1 = 0.17132449237917...$ $w_2 = 0.36076157304813...$ $w_3 = 0.46791393457269...$ $w_4 = 0.46791393457269...$ $w_5 = 0.36076157304813...$ $w_6 = 0.17132449237917...$
7	$x_1 = -0.94910791234275...$ $x_2 = -0.74153118559939...$ $x_3 = -0.40584515137739...$ $x_4 = 0$ $x_5 = 0.40584515137739...$ $x_6 = 0.74153118559939...$ $x_7 = 0.94910791234275...$	$w_1 = 0.12948496616887...$ $w_2 = 0.27970539148927...$ $w_3 = 0.38183005050511...$ $w_4 = 0.41795918367346$ $w_5 = 0.38183005050511...$ $w_6 = 0.27970539148927...$ $w_7 = 0.12948496616887...$

Table 3.1: Table with Gauss-Legendre quadrature nodes and weights for one to seven points [14].

### 3.5.2 Gauss-Jacobi Quadrature

If the function to integrate  $f(x)$ ,  $f : \mathbb{R} \mapsto [-1, 1]$  has endpoint singularities, the integrand can be rewritten as  $f(x) = (1-x)^\alpha(1+x)^\beta g(x)$ , where  $g$  is just the integrable function without the endpoint singularities at  $\{-1, 1\}$ . This is known as the Gauss-Jacobi quadrature rule.

**Definition 23** ( $n$ -point Gauss-Jacobi quadrature rule).

$$\int_{-1}^1 f(x) dx = \int_{-1}^1 (1-x)^\alpha(1+x)^\beta g(x) dx \approx \sum_{i=1}^n w_i g(x_i), \quad (3.23)$$

with  $\alpha, \beta \geq -1$ .

It is important to note that the Gauss-Legendre quadrature is a special case of the Gauss-Jacobi quadrature with  $\alpha = \beta = 0$ . Some other special cases of the Gauss-Jacobi quadrature are the Chebyshev-Gauss quadrature of the first kind ( $\alpha = \beta = -\frac{1}{2}$ ), Chebyshev-Gauss quadrature of the second kind ( $\alpha = \beta = \frac{1}{2}$ ) and Gauss-Gegenbauer quadrature ( $\alpha = \beta$ ).

The Jacobi polynomials, like the Legendre polynomials, are orthogonal polynomials and are defined through a recurrence relation.

#### Algorithm

To compute the Gauss-Jacobi quadrature nodes, the implementations of deal.II [11], GSL [15], and LehrFEM++ [16] were analyzed. All of the mentioned libraries use an iterative algorithm to compute the zeros of the polynomials with the recurrence relation using Newton's method [13, Chapter 8.5].

The implemented algorithm is mostly based on the implementation from LehrFEM++ [16], since it is best suited our requirements. For Newton's method, the implementation remained the same as in LehrFEM++.

The steps in the algorithm are:

1. Iterate over the number of quadrature points to compute (roots of Gauss-Jacobi polynomial).
2. For each point, make an initial (educated) guess.
3. Apply Newton's method on the initial guess to find the root.
4. Compute the weight corresponding to the point using the root.

The implemented algorithm (see Code 1) also provides the possibility to use different initial guesses. LehrFEM++ [16] uses some custom initial guesses compared to deal.II [11], which uses the Chebychev nodes for initial guesses.

### 3.5.3 Tensor-product Quadrature

In this chapter, up to now, only one-dimensional quadrature was considered. However, in this framework, the assembly functions use the quadrature nodes on the reference elements, which have the same dimension as the mesh that is used.

The reference elements are  $n$ -cuboids and the quadrature nodes follow the vertices of a tensor-product mesh.

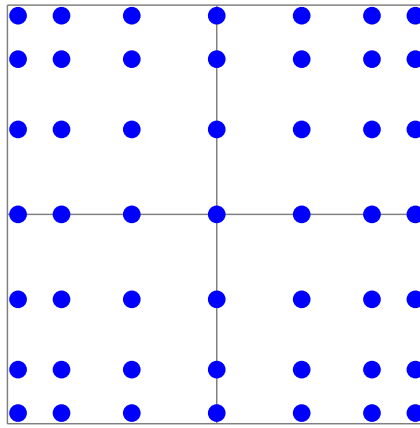


Figure 3.12: 2D Tensor-product of 7-point Gauss-Legendre quadrature nodes on the unit-square.

```

// Compute the root of the Jacobi polynomial
for (std::size_t i = 0; i < NumNodes1D; ++i) {
    // initial guess depending on which root we are computing
    if (initial_guess_type == InitialGuessType::LehrFEM) {
        z = this->getLehrFEMInitialGuess(i, integration_nodes);
    } else if (initial_guess_type == InitialGuessType::Chebyshev) {
        z = -this->getChebyshevNodes(i);
    } else {
        throw std::runtime_error("Unknown initial guess type");
    }

    std::size_t its = 1;
    do {
        // refinement by Newton's method (from LehrFEM++)
        temp = 2.0 + alfbet;

        p1 = (alpha - beta + temp * z) / 2.0;
        p2 = 1.0;
        for (std::size_t j = 2; j <= NumNodes1D; ++j) {
            p3 = p2;
            p2 = p1;
            temp = 2 * j + alfbet;
            a = 2 * j * (j + alfbet) * (temp - 2.0);
            b = (temp - 1.0) * (alpha * alpha - beta * beta + temp * (temp - 2.0) * z);
            c = 2.0 * (j - 1 + alpha) * (j - 1 + beta) * temp;
            p1 = (b * p2 - c * p3) / a;
        }
        pp = (NumNodes1D * (alpha - beta - temp * z) * p1
              + 2.0 * (NumNodes1D + alpha) * (NumNodes1D + beta) * p2)
              / (temp * (1.0 - z * z));

        z1 = z;
        z = z1 - p1 / pp; // Newtons Formula

        if (its > this->min_newton_iterations_m && Kokkos::abs(z - z1) <= tolerance) {
            break;
        }
        ++its;
    } while (its <= this->max_newton_iterations_m);

    integration_nodes[i] = z;

    // Compute the weight of the Gauss-Jacobi quadrature
    weights[i] =
        Kokkos::exp(Kokkos::lgamma(alpha + NumNodes1D) + Kokkos::lgamma(beta + NumNodes1D)
                    - Kokkos::lgamma(NumNodes1D + 1.)
                    - Kokkos::lgamma(static_cast<double>(NumNodes1D) + alfbet + 1.0))
        * temp * Kokkos::pow(2.0, alfbet) / (pp * p2);
}

```

Code 1: Abbreviated Algorithm for the computation of the Gauss-Jacobi quadrature nodes and weights.

### 3.6 Matrix-free Cell-oriented Assembly Algorithm with Local Computations

Assembly in FEM refers to the computation of the entries of the stiffness matrix  $\mathbf{A}$  and the entries of the right-hand side vector, or load vector,  $\boldsymbol{\varphi}$  of the linear system of equations:  $\mathbf{A}\boldsymbol{\mu} = \boldsymbol{\varphi}$ .

Recall the definition from section 2.1 of the stiffness matrix (Def. 9):

$$\mathbf{A} = \left[ a(b_h^j, b_h^i) \right]_{i,j=1}^N, \in \mathbb{R}^{N,N}, \quad (2.19)$$

and the load vector (Def. 10):

$$\boldsymbol{\varphi} = [\ell(b_h^i)]_{i=1}^N \in \mathbb{R}^N. \quad (2.20)$$

For each element, we define the *element stiffness matrix* and *element load vector*.

**Definition 24** (Element stiffness matrix and load vector). *Given a mesh element  $K$  and the local shape functions  $\{\hat{b}^1, \dots, \hat{b}^M\}$ , where  $M$  is the number of local shape functions, we define the element stiffness matrix:*

$$\mathbf{A}_K := \left[ a(b_K^j, b_K^i) \right]_{i,j=1}^M \in \mathbb{R}^{M,M}, \quad (3.24)$$

and the element load vector:

$$\boldsymbol{\varphi}_K := [\ell(b_K^i)]_{i=1}^M \in \mathbb{R}^M. \quad (3.25)$$

The element stiffness matrix  $\mathbf{A}_K$  has as many rows and columns as there are local degrees of freedom. For example, for a one-dimensional edge element with the degrees of freedom on the vertices,  $\mathbf{A}_K$  is a two-by-two matrix. For a 2D mesh and a quadrilateral element with the degrees of freedom also on the vertices, it is a four-by-four matrix.

The stiffness matrix  $\mathbf{A}$  can be ‘assembled’ by iterating over all the elements, computing the local element stiffness matrices for every element and then using them to accumulate the contribution to the global stiffness matrix. However, in this work, we are using a ‘matrix-free’ assembly algorithm and are not storing the full global stiffness matrix  $\mathbf{A}$  at any point in time. The implementation of the matrix-free assembly algorithm is covered in the next section.

#### 3.6.1 Matrix-free Assembly Algorithm for the Stiffness Matrix

As briefly explained in section 2.2, we implemented a matrix-free assembly algorithm that interfaces with the Conjugate Gradient (CG) method and returns the product of the stiffness matrix  $\mathbf{A}$  and an arbitrary vector  $\mathbf{x}$  that is passed to the function:  $\mathbf{A}\mathbf{x}$ . We call this assembly function `evaluateAx` and in addition to the vector  $\mathbf{x}$ , it also takes in a C++ Lambda function, named `eval`, as an input argument.

The `eval` function evaluates the approximation of the bilinear form for a given problem. It is defined outside of the assembly function and passed as an argument to make the assembly function independent of the PDE that is to be solved. Later in this section, we derive the assembly function for the Poisson equation and in the end, we arrive at the assembly function that was implemented together with the corresponding `eval` function for the Poisson equation.

The assembly algorithm is cell-oriented, meaning we iterate over all the elements of the mesh instead of the degrees of freedom. For each element, we perform two steps. First, we compute the element stiffness matrix  $\mathbf{A}_K$  and second, we compute the contribution of this element with its corresponding element stiffness matrix, and the vector  $\mathbf{x}$  to the matrix-vector product  $\mathbf{A}\mathbf{x}$  that is returned.

Let's start at the first step: Computing the element stiffness matrix  $\mathbf{A}_K$  for a given element  $K$ . The entries of the element stiffness matrix are defined using the element bilinear form  $a(\cdot, \cdot)$ , which stems from the weak form of the problem. The bilinear form of the Poisson equation with homogeneous Dirichlet boundary conditions is (see section 2.1.1):

$$(\mathbf{A}_K)_{i,j} = a(b_K^i, b_K^j) = \int_K \nabla b_K^j \cdot \nabla b_K^i \, d\mathbf{x}. \quad (3.26)$$

However, instead of integrating over a different element  $K$  each iteration, we can transform this integral onto a reference element  $\hat{K}$  with unit dimensions and compute it there ahead of time and then simply do the transformation for every element.

We will briefly explain the variables used in this derivation.  $I$  and  $J$  are the global degrees of freedom corresponding to the local degrees of freedom  $i$  and  $j$ . Where a degree of freedom  $I$  corresponds to one or more local degrees of freedom  $i$  for different elements if the elements are adjacent and it lies on the boundaries of those elements. Given a local degree of freedom  $i$ , and an element  $K$ , the choice of the global degree of freedom  $I$  is unique. In the other direction, if a global degree of freedom  $I$  is on an element  $K$ , the choice of the local degree of freedom for the element is unique as well. We defined  $b_K^I$  and  $b_K^J$  to be the shape functions corresponding to the local degrees of freedom  $i$  and  $j$  of element  $K$ , defined on the global coordinate system. The shape functions of the reference element  $\hat{K}$  are  $\hat{b}_i$  with  $i \in 1, \dots, M$ , where  $M$  is the number of degrees of freedom on an element.

We start at the definition of the entry of the element stiffness matrix at row  $i$  and column  $j$ , corresponding to the local degrees of freedom  $i$  and  $j$  of the element  $K$  (Equation 3.27). To compute the integral on the reference element  $\hat{K}$  instead of the global element  $K$  we use the transformation  $\Phi_K$  and its pullback  $\Phi_K^*$  (Equation 3.28). The transformations are defined in section 3.3.2. Next, the pullback is applied to the gradients. This is done to evaluate the gradients of the local shape function on the reference element instead (Equation 3.29). This way, all the computations are done on the reference element and can be precomputed. After that, we can use numerical quadrature (see section 3.5) to approximate the integral (Equation 3.30).

$$(\mathbf{A}_K)_{i,j} = \int_K \nabla b_K^J(\mathbf{x}) \cdot \nabla b_K^I(\mathbf{x}) \, d\mathbf{x}, \quad (3.27)$$

$$= \int_{\hat{K}} \Phi_K^* \nabla b_K^J(\hat{\mathbf{x}}) \cdot \Phi_K^* \nabla b_K^I(\hat{\mathbf{x}}) |\det \mathbf{D}\Phi_K(\hat{\mathbf{x}})| \, d\hat{\mathbf{x}}, \quad (3.28)$$

$$= \int_{\hat{K}} (\mathbf{D}\Phi_K)^{-\top} \nabla \hat{b}^j(\hat{\mathbf{x}}) \cdot (\mathbf{D}\Phi_K)^{-\top} \nabla \hat{b}^i(\hat{\mathbf{x}}) |\det \mathbf{D}\Phi_K(\hat{\mathbf{x}})| \, d\hat{\mathbf{x}}, \quad (3.29)$$

$$\approx \sum_k^{N_{\text{Int}}} \hat{\omega}_k (\mathbf{D}\Phi_K)^{-\top} \nabla \hat{b}^j(\hat{\mathbf{q}}_k) \cdot (\mathbf{D}\Phi_K)^{-\top} \nabla \hat{b}^i(\hat{\mathbf{q}}_k) |\det \mathbf{D}\Phi_K(\hat{\mathbf{q}}_k)|, \quad (3.30)$$

where  $\hat{\omega}_k$  is the weight of the quadrature rule associated with  $\hat{\mathbf{q}}_k$ , which is the  $k$ -th quadrature node on the reference element  $\hat{K}$ .

The eval function is then defined as:

$$\text{eval}(i, j, k) := (\mathbf{D}\Phi_K(\hat{\mathbf{q}}_k))^{-\top} \nabla \hat{b}^j(\hat{\mathbf{q}}_k) \cdot (\mathbf{D}\Phi_K(\hat{\mathbf{q}}_k))^{-\top} \nabla \hat{b}^i(\hat{\mathbf{q}}_k) |\det \mathbf{D}\Phi_K(\hat{\mathbf{q}}_k)| \quad (3.31)$$

The second step in each iteration of the assembly function is the computation of the contribution of the element to the global stiffness matrix  $\mathbf{A}$ . The assembly function returns a vector  $\mathbf{z}$ , which is the product of the global stiffness matrix  $\mathbf{A}$  and the vector  $\mathbf{x}$ :  $\mathbf{z} = \mathbf{A}\mathbf{x}$ . Each entry  $z_I$  at the global degree of freedom  $I$  is the dot-product of the  $I$ -th row of  $\mathbf{A}$  and vector  $\mathbf{x}$ :  $z_I = (\mathbf{A})_{I,:} \cdot \mathbf{x}$ .

Recall that the local degree of freedom  $i$  in element  $K$  corresponds to a global degree of freedom  $I$ . This mapping is stored in the `FiniteElementSpace` class and is computed once for each element and then stored at the beginning of each iteration of the assembly algorithm.

The  $i$ -th entry in the list of global degrees of freedom `globalDOFs` for element  $K$ , is the index  $I$  of the global degree of freedom. Therefore the contribution to the global stiffness matrix  $\mathbf{A}$  of an element stiffness matrix  $\mathbf{A}_K$  is computed with:

```

for  $i \in \text{localDOFs}$  do
   $I = \text{globalDOFs}[i]$ 
  for  $j \in \text{localDOFs}$  do
     $J = \text{globalDOFs}[j]$ 
     $(\mathbf{A})_{I,J} \leftarrow (\mathbf{A})_{I,J} + (\mathbf{A}_K)_{i,j}$ 
  end
end

```

To compute the contribution of an element and its corresponding element matrix to the resulting vector  $\mathbf{z}$  we therefore have to add to the  $z_I$  entry:

$$z_I = z_I + (\mathbf{A}_K)_{i,j} \cdot x_J \quad (3.32)$$

The pseudocode for the full algorithm for the assembly of the left-hand side (`evaluateAx`) is given in Algorithm 2.

### 3.6.2 Assembly Algorithm for the Load Vector

The assembly algorithm for the right-hand-side is very similar to the left-hand side:

1. Assembly of the element load vector.
2. Accumulation of the contribution to  $\boldsymbol{\varphi}$ .

The linear form of the load vector is (from section 2.1):

$$\ell(b_K^I) = \int_K f(\mathbf{x}) b_K^I(\mathbf{x}) \, d\mathbf{x}, \quad (3.33)$$

where  $f$  is the source function and  $b_K^I$  is the shape function of the  $I$ -th global degree of freedom of element  $K$ .

Using the method outlined in the previous section, we arrive at the algorithm for the computation of the load vector shown in Algorithm 3. For this assembly function of the right-hand side, we don't pass an `eval` function but the right-hand side source function  $f$ .



```

Input:  $\mathbf{x}$ ,  $\text{eval}(i, j, k)$ 
 $\mathbf{z} \leftarrow \mathbf{0}$  // Resulting vector to return
for Element  $K$  in Mesh do
    localDOFs  $\leftarrow$  getLocalDOFsForElement( $K$ )
    globalDOFs  $\leftarrow$  getGlobalDOFsForElement( $K$ )
    // 1. Compute the Element matrix  $\mathbf{A}_K$ 
    for  $i \in \text{localDOFs}$  do
        for  $j \in \text{localDOFs}$  do
             $(\mathbf{A}_K)_{i,j} \approx \sum_k^{N_{\text{Int}}} \hat{\omega}_k \underbrace{(\mathbf{D}\Phi_K(\hat{\mathbf{q}}_k))^{-\top} \nabla \hat{b}^j(\hat{\mathbf{q}}_k) \cdot (\mathbf{D}\Phi_K(\hat{\mathbf{q}}_k))^{-\top} \nabla \hat{b}^i(\hat{\mathbf{q}}_k) | \det \mathbf{D}\Phi_K(\hat{\mathbf{q}}_k)|}_{=:\text{eval}(i,j,k)}$ 
        end
    end
    // 2. Compute  $\mathbf{z} = \mathbf{A}\mathbf{x}$  contribution with  $\mathbf{A}_K$ 
    for  $i \in \text{localDOFs}$  do
         $I = \text{globalDOFs}[i]$ 
        for  $j \in \text{localDOFs}$  do
             $J = \text{globalDOFs}[j]$ 
             $\mathbf{z}_I \leftarrow \mathbf{z}_I + (\mathbf{A}_K)_{i,j} \cdot \mathbf{x}_J$ 
        end
    end
end
return  $\mathbf{z}$ 

```

**Algorithm 2:** evaluateAx (for the specific case of the Poisson equation).

### 3.6.3 Essential Boundary conditions

In FEM, Dirichlet boundary conditions are synonymous with essential boundary conditions.

Essential boundary conditions are enforced on the functions in the trial space. Natural boundary conditions or Neumann boundary conditions, are enforced through the variational equation. They therefore appear in the bilinear form and are implemented in the `eval` function of the problem.

#### Homogeneous Dirichlet Boundary Conditions

To implement homogeneous Dirichlet boundary conditions, or zero Dirichlet boundary conditions, one can set the entries of the stiffness matrix corresponding to the degrees of freedom on the boundary to zero [4, Chapter 2.7.6].

To support them, the left-hand-side assembly algorithm was modified to skip all DOFs on the boundary when accumulating the contributions from the element stiffness matrices to force them to be zero.

Due to time constraints, we must leave heterogeneous Dirichlet boundary conditions, and other types of boundary conditions, to be implemented in future work.

```

Input:  $x, f$ 
 $\varphi \leftarrow \mathbf{0}$  // Resulting load vector to return
for Element  $K$  in Mesh do
    localDOFs  $\leftarrow$  getLocalDOFsForElement( $K$ )
    globalDOFs  $\leftarrow$  getGlobalDOFsForElement( $K$ )
    // 1. Compute the Element matrix  $\mathbf{A}_K$ 
    for  $i \in \text{localDOFs}$  do
         $(\varphi_K)_i \approx \sum_k^{N_{\text{Int}}} \hat{\omega}_k f(\Phi_K(\hat{x})) \hat{b}^i(\hat{q}_k) |\det \mathbf{D}\Phi_K(\hat{q}_k)|$ 
    end
    // 2. Compute load vector  $\varphi$  contribution of  $\varphi_K$ 
    for  $i \in \text{localDOFs}$  do
         $I = \text{globalDOFs}[i]$ 
         $\varphi_I \leftarrow \varphi_I + (\varphi_K)_i$ 
    end
end
return  $\varphi$ 

```

**Algorithm 3:** evaluateLoadVector (for the specific case of the Poisson equation).

### 3.7 Solver

The `FEMPoissonSolver` class inherits from the IPPL `Poisson` class and follows in functionality the existing IPPL `PoissonCG` class. This new solver class provides a constructor, a solve function and getters for the total number of iterations and the residue after solving. The constructor takes in a left-hand-side IPPL `Field`, a right-hand-side IPPL `Field` and the source function  $f$ . In the constructor, the right-hand-side field values are then set according to the `evaluateLoadVector` assembly function with the given source function  $f$ .

Inside the `solve` function, the `eval` lambda function for the Poisson equation is defined. The `eval` function is a function that depends on the PDE weak form and is used to decouple the assembly function from the problem itself. It is passed to the assembly function which provides the arguments needed to solve the problem.

For the Poisson equation, the `eval` function is:

$$\text{eval}(i, j, k) = (\mathbf{D}\Phi_K^{-\top}) \nabla \hat{b}^j(\mathbf{q}_k) \cdot (\mathbf{D}\Phi_K^{-\top}) \nabla \hat{b}^i(\mathbf{q}_k) |\det \mathbf{D}\Phi_K|, \quad (3.34)$$

where  $i$  and  $j$  are the indices for the local degrees of freedom used to select the corresponding reference element shape function and  $\mathbf{q}_k$  is the  $k$ -th integration point.

It is important to note, that everything in this `eval` function can be pre-computed because everything is evaluated on the reference element (see section 3.3).

Inside the `FEMPoissonSolver`, the transformation variables  $\mathbf{D}\Phi^{-\top}$  and  $|\det \mathbf{D}\Phi|$  are pre-computed but the values of the basis functions at the quadrature nodes  $k$  are precomputed inside the assembly function instead and passed by reference to the `eval` function. In the future, the signature of the `eval` function might need tweaking to support problems that have more than the gradient of the basis in their weak form.

```
const auto poissonEquationEval =
    [DPhiInvT, absDetDPhi](
        const std::size_t& i, const std::size_t& j,
        const Vector<Vector<Tlhs, Dim>, NumElementDOFs>& grad_b_q_k) {
        return dot((DPhiInvT * grad_b_q_k[j]), (DPhiInvT * grad_b_q_k[i])).apply()
            * absDetDPhi;
    };

```

Code 2: `eval` function implementation from `FEMPoissonSolver`.

In the `FEMPoissonSolver`, after pre-computation of the transformation variables, the IPPL `PCG` (Preconditioned Conjugate Gradient) algorithm class is used to solve the linear system of equations.

Currently, the `FEMPoissonSolver` is using Gauss-Legendre quadrature with five integration points in one dimension. It is used by initializing the `GaussJacobiQuadrature` class with the parameters  $\alpha = \beta = 0$ . points and first-order Lagrangian Finite Elements using the `LagrangeSpace` class.

The dimension of the mesh is the same as the dimension of the input fields that are passed to the constructor.

### 3.8 Results

Using the `FEMPoissonSolver`, outlined in the previous section (see section 3.7), we computed the solutions for a given problem and studied the convergence to the exact solution for decreasing mesh spacings (mesh refinement). The accuracy was tested for one, two and three dimensions and the convergence was good for each of them.

The problem solved was the Poisson equation with a sinusoidal source function and homogeneous Dirichlet boundary conditions. In one dimension, the problem was:

$$-\Delta u = \pi^2 \sin(\pi x), \quad x \in [-1, 1], \quad (3.35)$$

with  $u(-1) = 0$  and  $u(1) = 0$ . The exact solution to this problem is:

$$u(x) = \sin(\pi x). \quad (3.36)$$

To solve this problem in two and three dimensions, the following  $N$ -dimensional variant of the same problem was used:

$$-\Delta u = N\pi^2 \prod_{i=1}^N \sin(\pi x_i), \quad \mathbf{x} \in [-1, 1]^N, \quad (3.37)$$

where  $N \in \mathbb{N}$  is the number of dimensions, the solution function  $u$  is defined as:  $u : \mathbb{R}^N \rightarrow \mathbb{R}$ , with the  $N$ -dimensional point  $\mathbf{x} \in \mathbb{R}^N$  with its elements  $\mathbf{x}_i \in \mathbb{R} : \mathbf{x}_i \in \mathbf{x}$  with  $i \in \{1, \dots, N\}$ . Also, the function  $u$  is zero on the boundaries:  $u(\mathbf{x}) = 0$  if for any entry  $\mathbf{x}_i \in \mathbf{x}$ , it holds that  $\mathbf{x}_i \in \{-1, 1\}$  for  $i \in \{1, \dots, N\}$ .

This problem has the exact solution:

$$u(\mathbf{x}) = \prod_{i=1}^N \sin(\pi \mathbf{x}_i). \quad (3.38)$$

This test along with these sinusoidal functions are implemented in `TestFEMPoissonSolver`.

Different numbers of mesh vertices of powers of two were used in each dimension. For a given number of mesh vertices  $n$ , the mesh spacing  $h$  is given by:

$$h = \frac{2}{n-1}. \quad (3.39)$$

The tolerance used in the CG algorithm was chosen to be  $10^{-13}$ .

In table 3.2 the number of mesh vertices  $n$  is given in the ‘Size column’, the ‘Spacing’ is computed via equation 3.39, the ‘Relative Error’ is the relative error of the computed solution compared to the analytical solution of the problem, the ‘Residue’ is the residue from the CG algorithm (see Algorithm 1) and the number of iterations of the CG algorithm is given in the ‘Iterations’ column.

Size	Spacing	Relative Error	Residue	Iterations
4	0.6666666666666666	7.080989919320194e-09	4.440892098500626e-16	1
8	0.2857142857142857	1.249845927172312e-12	2.094764613337708e-15	1
16	0.1333333333333333	6.407420160623263e-16	5.325889242270741e-15	1
32	0.06451612903225806	1.779759178867111e-15	1.742186992884441e-14	1
64	0.03174603174603174	1.789223255831461e-16	6.654398136858673e-14	1
128	0.01574803149606299	3.503924460135148e-15	5.157582646417248e-14	2
256	0.007843137254901961	7.471370361275646e-15	8.253791443434346e-14	3
512	0.003913894324853229	2.094557289636767e-15	4.997105441592253e-14	5
1024	0.001955034213098729	2.178897100917102e-14	4.308783938700041e-14	177

Table 3.2: TestFEMPoissonSolver Output for the 1D sinusoidal problem.

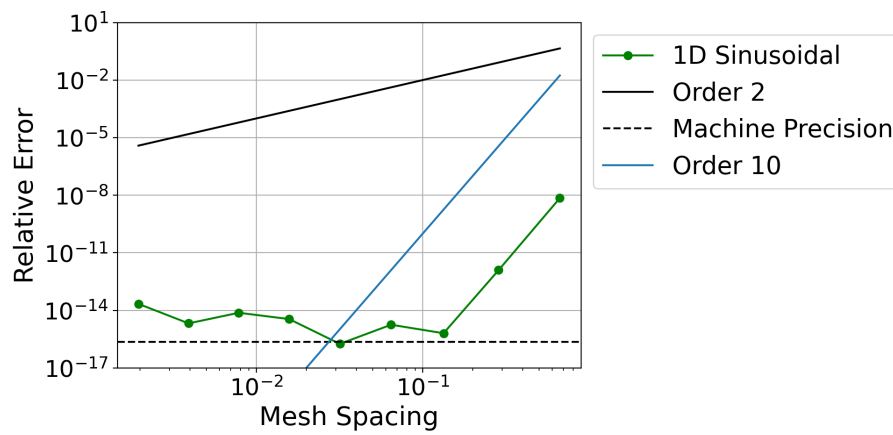


Figure 3.13: 1D Sinusoidal convergence

For one dimension, the solution converges with order 10 until it gets close to machine precision (figure 3.13). This is much faster than anticipated and we assume it is due to the smooth, periodic nature of the right-hand side chosen. At the moment the solver only uses first-order Lagrangian finite element methods, which are linear functions (see figure 3.8). There is a phenomenon for the trapezoidal quadrature rule, where it converges rapidly when applied to analytic functions on periodic intervals [17].

We think the convergence we are seeing for one dimension might be related to this phenomenon, however, this was not investigated further but it might be interesting for future work.

Size	Spacing	Relative Error	Residue	Iterations
4	0.6666666666666666	0.367835959800294	8.881784197001252e-16	1
8	0.2857142857142857	0.06877172889870629	2.054662609310059e-15	1
16	0.1333333333333333	0.01470558224055019	6.673718396468935e-15	1
32	0.06451612903225806	0.003428048120959612	1.226637297968919e-14	1
64	0.03174603174603174	0.0008291654851627174	2.735745689195284e-14	1
128	0.01574803149606299	0.0002039888701591884	1.157673913473065e-14	2
256	0.007843137254901961	5.059492109621404e-05	1.26186832683033e-14	3
512	0.003913894324853229	1.259908312652751e-05	7.423183529194849e-15	119
1024	0.001955034213098729	3.143610123038628e-06	3.751402450611288e-15	265

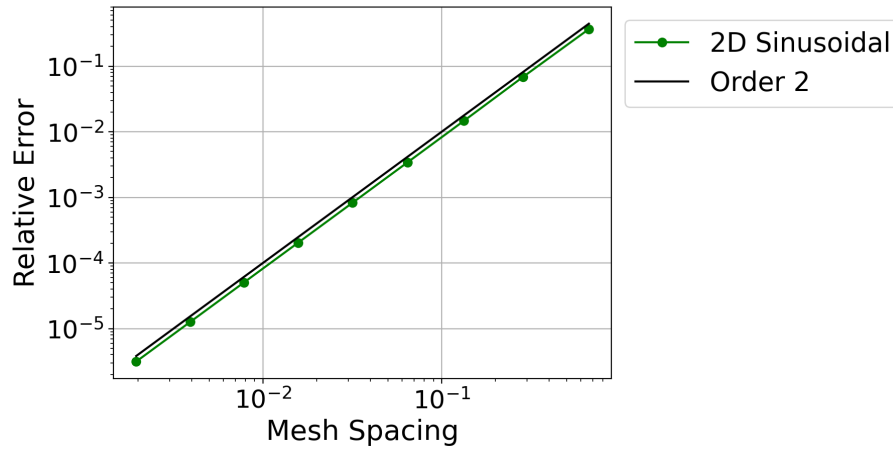
Table 3.3: `TestFEMPoissonSolver` Output for the 2D sinusoidal problem.

Figure 3.14: 2D Sinusoidal convergence

In two dimensions, the fast convergence from the one-dimensional study disappears and it follows second-order convergence. This is the minimal rate of convergence expected for FEM but it is to be expected for first-order Lagrangian finite elements.

Size	Spacing	Relative Error	Residue	Iterations
4	0.6666666666666666	0.8709752261711512	1.368774871883577e-15	1
8	0.2857142857142857	0.1422730084945544	8.763930132546234e-15	1
16	0.1333333333333333	0.02962741863012481	1.814959731383155e-14	1
32	0.06451612903225806	0.006867847755836543	1.019648580795529e-14	1
64	0.03174603174603174	0.00165901848558236	1.623920791356438e-14	1
128	0.01574803149606299	0.0004080193515153442	3.485107117806992e-15	10

Table 3.4: `TestFEMPoissonSolver` Output for the 3D sinusoidal problem.

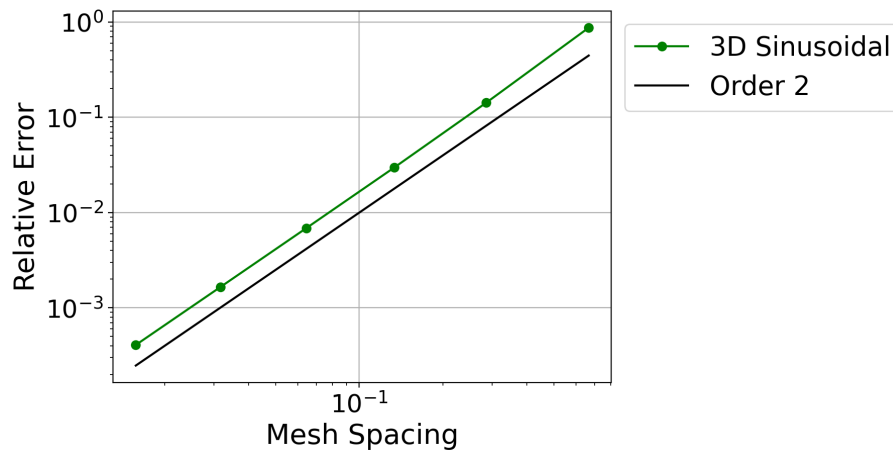


Figure 3.15: 3D Sinusoidal convergence

In three dimensions, we get a very similar picture as in two dimensions. We also observe second-order convergence.

Note: For three dimensions, the relative error was not computed for more than 128 mesh vertices (in each dimension) due to the high computational load and the serial nature of this implementation.

## Chapter 4

# Conclusion

This thesis and the implemented building blocks for it are the first big step into FEM for IPPL. It not only contains the building blocks for FEM, it can already be considered a FEM framework by itself, with the inclusion of the solver. We have shown the implementation from the ground up and have been able to achieve good convergence using the solver.

We started by designing the general software architecture of the `C++` FEM framework in IPPL, with extensibility in mind. Then, element classes along with dimension-independent affine transformations were implemented, followed by the `FiniteElementSpace` base class handling the mapping of the mesh to the elements. Then the `LagrangeSpace` class was implemented, which supports first-order Lagrangian finite elements, for one- to three-dimensional meshes. It was also designed to be easily extensible to higher-order basis functions. Next, a base class for numerical quadrature was added which has methods for creating tensor-product quadrature points and weights for all the reference elements. It serves as the basis of the Midpoint quadrature rule and the Gauss-Jacobi quadrature rule which were also implemented in this work. The latter uses an iterative algorithm to compute the roots of the Jacobi-Polynomial. With the main building blocks taken care of, a special matrix-free assembly algorithm, interfacing with the CG method, was implemented for ideal performance on GPUs, in the future. And finally, the `FEMPoissonSolver` was added as a proof-of-concept and as an example for future FEM solvers.



## 4.1 Future Work

This thesis is still only the beginning for FEM in IPPL. In future work, one could continue with higher-order Lagrangian finite elements, adding support for more boundary conditions (i.e. non-homogeneous Dirichlet boundary conditions or periodic boundary conditions) and process- and thread-level parallelization. Then it would be interesting to do some timing studies and GPU benchmarks. After that, more `FinitElementSpace` classes could be implemented (i.e. Nédélec or Raviart-Thomas finite elements), on the road to solving the electromagnetic Maxwell equations.

We provide some guidance and our thoughts on the implementation of some of these topics in Appendix B.

## Acknowledgements

I would like to thank the AMAS group from PSI and Dr. Andreas Adelman for their support and their trust to let me begin the work of implementing FEM in the IPPL framework. I especially want to thank my scientific advisor Ms. Sonali Mayani who supported me greatly throughout this thesis, it was great working with her.

# Bibliography

- [1] “June 2022 | TOP500,” Dec. 2023. [Online]. Available: <https://www.top500.org/lists/top500/2022/06/>
- [2] M. Frey, A. Vinciguerra, S. Muralikrishnan, S. Mayani, V. Montanaro, and A. Adelmann, “IPPL-framework/ipp: IPPL-3.1.0,” Sep. 2023. [Online]. Available: <https://zenodo.org/doi/10.5281/zenodo.8389192>
- [3] K. Yee and J. Chen, “The finite-difference time-domain (FDTD) and the finite-volume time-domain (FVTD) methods in solving Maxwell’s equations,” *IEEE Transactions on Antennas and Propagation*, vol. 45, no. 3, pp. 354–363, Mar. 1997, conference Name: IEEE Transactions on Antennas and Propagation. [Online]. Available: <https://ieeexplore.ieee.org/document/558651>
- [4] R. Hiptmair, “Numerical Methods for (Partial) Differential Equations,” 2023. [Online]. Available: <https://people.math.ethz.ch/~grsam/NUMPDEFL/NUMPDE.pdf>
- [5] K. Ljungkvist, “Matrix-Free Finite-Element Computations On Graphics Processors with Adaptively Refined Unstructured Meshes,” in *25th High Performance Computing Symposium (HPC 2017)*. Virginia Beach, VA, USA: Society for Modeling and Simulation International (SCS), 2017. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3108097>
- [6] R. R. Settghost, Y. Dudouit, N. Castelletto, W. R. Tobin, B. C. Corbett, and S. Klevtsov, “Performant low-order matrix-free finite element kernels on GPU architectures,” Aug. 2023, arXiv:2308.09839 [cs, math]. [Online]. Available: <http://arxiv.org/abs/2308.09839>
- [7] S. Muralikrishnan, M. Frey, A. Vinciguerra, M. Ligotino, A. J. Cerfon, M. Stoyanov, R. Gayatri, and A. Adelmann, “Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures,” Nov. 2022, arXiv:2205.11052 [physics]. [Online]. Available: <http://arxiv.org/abs/2205.11052>
- [8] A. Ayala, S. Tomov, A. Haidar, and J. Dongarra, “heFFTe: Highly Efficient FFT for Exascale,” in *Computational Science – ICCS 2020*, ser. Lecture Notes in Computer Science, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020, pp. 262–275.
- [9] H. Carter Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, Jul. 2014, institution: Sandia National Lab. (SNL-NM), Albuquerque, NM (United States) Number: SAND-2013-5603J Publisher: Elsevier. [Online]. Available: <https://www.osti.gov/biblio/1106586>

- [10] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cervený, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, “MFEM: A modular finite element methods library,” *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, Jan. 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0898122120302583>
- [11] W. Bangerth, R. Hartmann, and G. Kanschat, “deal.II—A general-purpose object-oriented finite element library,” *ACM Transactions on Mathematical Software*, vol. 33, no. 4, pp. 24–es, Aug. 2007. [Online]. Available: <https://dl.acm.org/doi/10.1145/1268776.1268779>
- [12] P. Castillo, R. Rieben, and D. White, “FEMSTER: An object-oriented class library of high-order discrete differential forms,” *ACM Transactions on Mathematical Software*, vol. 31, no. 4, pp. 425–457, Dec. 2005. [Online]. Available: <https://dl.acm.org/doi/10.1145/1114268.1114269>
- [13] R. Hiptmair, “Numerical Methods for Computational Science and Engineering,” 2020. [Online]. Available: [https://people.math.ethz.ch/~grsam/NCSE20/NumCSE\\_Lecture\\_Document.pdf](https://people.math.ethz.ch/~grsam/NCSE20/NumCSE_Lecture_Document.pdf)
- [14] A. N. Lowan, N. Davids, and A. Levenson, “Table of the zeros of the Legendre polynomials of order 1-16 and the weight coefficients for Gauss’ mechanical quadrature formula,” *Bulletin of the American Mathematical Society*, vol. 48, no. 10, pp. 739–743, 1942. [Online]. Available: <https://www.ams.org/bull/1942-48-10/S0002-9904-1942-07771-8/>
- [15] B. Gough, *GNU Scientific Library Reference Manual - Third Edition*, 3rd ed. Network Theory Ltd., 2009.
- [16] craffael, “LehrFEM++,” Aug. 2023, original-date: 2018-04-06T14:21:27Z. [Online]. Available: <https://github.com/craffael/lehrfempp>
- [17] L. N. Trefethen and J. A. C. Weideman, “The Exponentially Convergent Trapezoidal Rule,” *SIAM Review*, vol. 56, no. 3, pp. 385–458, Jan. 2014. [Online]. Available: <http://epubs.siam.org/doi/10.1137/130932132>

# Appendix A

## Resources and Details

### A.1 Resources

- IPPL repository (GitHub): <https://github.com/IPPL-framework/ippl>
- IPPL FEM branch (GitHub): <https://github.com/s-mayani/ippl/tree/fem-framework>
- IPPL build scripts (GitHub): <https://github.com/IPPL-framework/ippl-build-scripts>
- Student repository (PSI GitLab): [https://gitlab.psi.ch/AMAS-students/buehler\\_bsc](https://gitlab.psi.ch/AMAS-students/buehler_bsc)

### A.2 IPPL Build Instructions

1. Make sure the following prerequisites are installed:

- curl (<https://curl.se>)
- git (<https://git-scm.com>)
- Make (<https://www.gnu.org/software/make>)
- CMake (<https://cmake.org>)
- A compatible C++ compiler:
  - GCC (<https://gcc.gnu.org>)
  - Clang (<https://clang.llvm.org>)
- MPI, for example OpenMPI (<https://www.open-mpi.org>)
- GoogleTest (<https://github.com/google/googletest>)

2. Clone the build script repository.

```
git clone https://github.com/IPPL-framework/ippl-build-scripts.git
```

This will clone four Bash scripts into a new `ippl-build-scripts` directory:

- 100-build-kokkos
- 200-build-heffte
- 300-build-ippl

- 999-build-everything

The scripts are used to set up all the dependencies, as well as for building IPPL itself.

The commit used at the time of writing of the `ippl-build-scripts` repository was commit `e9a3e1f`, in later commits, some steps might be different.

To switch to that commit, run: `git checkout e9a3e1f`

3. Set up the build scripts to use the FEM branch.

The IPPL FEM framework was developed in a separate fork of the IPPL repository (listed as “IPPL FEM branch” in the ‘Resources’ section). At the time of writing, the FEM framework introduced in this thesis was not yet merged into any branch of the official IPPL GitHub repository, but instead lives in a branch in the forked repository. If the FEM framework was merged into the `main` branch in the meantime, this step can be safely ignored. If it was not yet merged, the URL of the IPPL repository needs to be altered in the `300-build-ippl` Bash script and the branch needs to be switched.

The unaltered script (`300-build-ippl`) contains the following snippet given in Code 3.

```
if [ -d "ippl" ]
then
    echo "Found existing IPPL source directory"
else
    echo "Clone ippl repo ... "
    if [ ! -z "$USE_SSH" ]; then
        git clone git@github.com:IPPL-framework/ippl.git
    else
        git clone https://github.com/IPPL-framework/ippl.git
    fi
fi
```

Code 3: Unaltered script snippet from `300-build-ippl`

In this snippet, we need to alter the repository URL to the URL of the repository that contains the FEM framework. Additionally, we add `git` commands to switch the branch after cloning.

The script should be altered to make the above snippet look like the following snippet given in Code 4.

After altering the script, it will now clone the desired repository and switch to the FEM branch.

4. Run the build scripts to install the dependencies and build ippl.

All that is left to do is run the build scripts with the desired options to build the IPPL repository. The available options can be found in the `README` of the build scripts repository under “Flags”.

To build everything, run the following command:

```
./999-build-everything -t serial -kfi --export
```

The command will also set the required environment variables. The `-t` argument sets the build target, `-k` installs Kokkos, `-f` installs heFFTe and `-i` installs IPPL.

```
if [ -d "ippl" ]
then
    echo "Found existing IPPL source directory"
else
    echo "Clone ippl repo ... "
    if [ ! -z "$USE_SSH" ]; then
        git clone git@github.com:s-mayani/ippl.git ippl
    else
        git clone https://github.com/s-mayani/ippl.git ippl
    fi
    # Navigate into the cloned folder to switch the branch
    cd ippl
    git fetch
    git switch fem-framework
    cd ..
fi
```

Code 4: Altered script snippet from 300-build-ippl

Running the command from above, will create a directory for ippl and inside it a build directory `ippl-build-scripts/ippl/build_serial/`. This build directory will contain all the executables. To rebuild in the future, navigate inside this directory and use `make`.

### A.3 Documentation Build Instructions

The IPPL framework has a Doxygen documentation that can be built locally, and viewed in the browser. To build it, make sure you have Doxygen and its prerequisites installed. Then navigate into the `doc/` directory in the IPPL repository and run the following command:

```
doxygen Doxyfile
```

Then simply open the file `doc/html/index.html` in the browser of your choice to read the documentation.

## A.4 Plot Generation

Each of the plots used in this work is produced in two steps. In the first step, the data is generated by running a test executable, which is compiled when building IPPL, and the generated data is stored in a file. This will be either a Comma-Separated Values (CSV) file or a text file that stores the raw output. In the second step, the data is plotted using Python and a Jupyter Notebook.

The Jupyter Notebooks can be found in the student repository (PSI GitLab) inside the `plots` directory. Inside the `plots` directory, there are two directories, `convergence` containing the convergence plots and the corresponding Jupyter notebook and `lagrange_space` containing two directories for 1D and 2D.

### A.4.1 Lagrangian finite element space plots

For the `LagrangeSpace` plots, the executables are inside the build directory (`build_serial`, `build_openmp`, etc.), under `<build_dir>/test/FEM/`. Where an executable for 1D and one for 2D can be found: `TestLagrangeSpace1D` and `TestLagrangeSpace2D`. Their source files can be found under `src/test/FEM/`.

After running the executables and before running the notebooks, copy the generated CSV files `1D_lagrange_local.basis.csv` and `2D_lagrange_local.basis.csv` to the same directory as the notebooks named

To create the plot for 1D, run the Jupyter notebook named `plot_lagrange_1d.ipynb` at `plots/lagrange_space/1d` in the student GitLab repository and for 2D run the Jupyter Notebook `plot_lagrange_2d.ipynb` at `plots/lagrange_space/2d`.

This will generate the plot from figure 3.9.

### A.4.2 Convergence plots

For the FEM convergence plot, the test is located in the `solver` directory in the build folder. The test is called `TestFEMPoissonSolver`. (`<build_dir>/solver/TestFEMPoissonSolver`). The source file for this test is located at `/src/test/solver/TestFEMPoissonSolver.cpp`. Run the executables and store the output in a text file. I chose the naming scheme of `sinus1d.dat` to `sinus3d.dat` depending on the number of dimensions.

Run the executable from the build directory with the command below, by replacing `<Dim>` with the integer 1, 2, or 3 for 1D, 2D and 3D:

```
./test/solver/TestFEMPoissonSolver <Dim> --info 5 > sinus<Dim>d.dat
```

The output will be piped into a file called `sinus1d.dat`, `sinus2d.dat` or `sinus3d.dat` in the build directory. After running the solver test for all three dimensions, copy all three files for the three dimensions to `plots/convergence` in the student GitLab repository. Then simply run the Jupyter Notebook in the same directory called `convergence.ipynb`.

This will generate the plots from figures 3.13, 3.14 and 3.15.



## Appendix B

# Implementation Helpers

The purpose of this section is to give some guidance and thoughts on the implementation of some of the topics mentioned in the Future Work section (section 4.1). Since we have not implemented the problems described in this section, we are not able to provide all the details for the respective implementations at this time, but we can provide some pointers as to where the current implementation needs to change and provide some additional thoughts where applicable.

### B.1 Higher-order Lagrangian finite elements

Something we planned on implementing, but ran out of time for, were higher-order Lagrangian finite elements. The implementation in the `LagrangeSpace` class will need to be updated as well as the corresponding unit tests. The `LagrangeSpace` class already has a template argument for the order that is used in some functions and unit tests. In the `LagrangeSpace` class both the DOF functions need to be updated, as well as the functions that evaluate the shape function and the gradient of the shape function on the reference element at a given point on the reference element.

### B.2 Non-homogeneous Dirichlet boundary conditions

For non-homogeneous Dirichlet boundary conditions, the assembly function `evaluateAx` in the `LagrangeSpace` needs to be updated, as well as the corresponding unit tests. The first change should be making the assembly function know which boundary condition it is dealing with. The type of boundary condition is stored in the field and should be retrieved from the LHS field object.

Currently, in the assembly function, it assumes homogeneous Dirichlet boundary conditions. Under this assumption, the current implementation skips all the DOFs on the boundary when accumulating the contribution to the resulting vector  $\mathbf{z}$ . This is done to set the assembly matrix to zero for degrees of freedom on the boundary. This behavior will need to be modified to implement the non-homogeneous Dirichlet boundary conditions. Please refer to [4, Chapter 2.7.6] for more details on how to implement them.

### B.3 Other Finite Elements and their spaces

An essential step in implementing the electromagnetic solver with the Finite Element Time Domain (FETD) scheme is the implementation of the Nédélec and Raviart-Thomas finite element methods.

To add a new finite element type, one needs to implement a new class that inherits from the `FiniteElementSpace` class and implements all the pure virtual functions from it. Meaning, that all the functions needed for the mapping from the local DOFs to the global DOFs and vice-versa need to be implemented, as well as the functions that evaluate the shape function and its gradient on the reference element. These functions should be implemented to be dimension-independent if possible, or at least work for 1D, 2D and 3D. As the next step, these implementations should also work for higher-order elements of the new type.

The functions that need to be implemented are:

<code>numGlobalDOFs</code>	Returns the number of global degrees of freedom (DOFs) in the space for a given mesh.
<code>getLocalDOFIndex</code>	Returns the local DOF for a given element and a given global DOF.
<code>getGlobalDOFIndex</code>	Returns the global DOF given an element and a local DOF.
<code>getLocalDOFIndices</code>	Returns the vector of local DOFs for the reference element for this space.
<code>getGlobalDOFIndices</code>	Returns the global DOFs for a given element.
<code>getGlobalDOFNDIndices</code>	Returns a vector of <code>NDIndex</code> for the global DOFs of a given element.
<code>evaluateRefElementShapeFunction</code>	Returns the scalar value of the reference element shape function for the given local DOF and given point on the reference element.
<code>evaluateRefElementShapeFunctionGradient</code>	Returns the scalar value of the gradient of the reference element shape function for a given local DOF and local point on the reference element.
<code>evaluateAx</code>	This is the assembly function for the left-hand side. For reference refer to the implementation in <code>LagrangeSpace</code> .
<code>evaluateLoadVector</code>	This is the assembly function for the right-hand side. For reference refer to the implementation in <code>LagrangeSpace</code> .

Also, a very helpful resource for getting around the framework is the Doxygen documentation. To build the documentation, refer to Appendix A.3.

After implementing this class a new solver should be implemented, to use this new type of finite elements. For that refer to the implementation of the `FEMPoissonSolver` which uses Lagrangian finite elements.

## B.4 Bilinear Transformations

We have implemented affine transformations in this work only, since IPPL only supports rectilinear meshes at this time and thus affine transformations suffice. Implementing bilinear transformations only really makes sense when IPPL supports unstructured meshes.

To implement bilinear transformations, the implementations of the element classes need to be updated. The difficulty with bilinear transformations compared to affine transformations is that the transformation cannot be simply described by multiplication with a transformation matrix and a translation. The element classes handle all the transformations, namely `Element`, `EdgeElement`, `QuadrilateralElement` and `HexahedralElement`. The functions `globalToLocal` and `localToGlobal` are the main interfaces for the transformations and should stay, but their implementation will need to change to support bilinear transformations.

For the theory behind bilinear transformations, refer to [4, Chapter 2.8.2].

## B.5 Periodic boundary conditions

During the implementation of the solver, we shortly thought about implementing periodic boundary conditions, as we were very limited in the number of problems we were able to solve using only homogeneous Dirichlet boundary conditions.

My idea on that was that one could manipulate only the stiffness matrix to implement periodic boundary conditions. On a cuboid mesh, one could remove the degrees of freedom of one of the two opposing boundaries and manipulate the stiffness matrix to be as if the degrees of freedom from one boundary are on both opposing boundaries.



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):

Building blocks for Finite Element computations in IPPL

**Authored by** (in block letters):

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

**Name(s):**

Bühler

**First name(s):**

Lukas

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**

Pfäffikon SZ, 1.2.2024

**Signature(s)**

L. Bühler

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