A Hierarchical Solver for Time-Harmonic Maxwell's Equations

Zur Erlangung des akademischen Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

von der KIT-Fakultät für Mathematik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

Pascal Kraft
geb. in Karlsruhe

Tag der mündlichen Prüfung: 13.04.2022

Hauptreferent: Prof. Dr. Willy Dörfler
Korreferent: Prof. Dr. Christian Wieners

Foreword

While I have loved the work on the project and the results make me proud, the enormous difficulties of the implementation and the sometimes extremely tedious nature of debugging a large code base has been difficult and I want to thank my partner, Hanna Becker, for supporting me when I needed it. I could not have done it without you!

I want to thank Prof. Dr. Willy Dörfler for his guidance and help as well as his trust and honesty. Working with you has been a pleasure and the working group has felt like home – at least before the Corona pandemic.

I want to thank my family for setting me on this path and supporting me on it and for always being there for me.

In no particular order I want to thank Julian Ott (for countless debates and lots of coffee), Stefan Findeisen and Mariia Sukova (for being great office partners), Fabian Castelli, Zoltan Veszelka and Niklas Baumgarten for the perfect measure of fun and competitiveness on the foosball table, Lars Machinek and Markus Meier for countless lunch breaks and coffee at the AKK combined with fruitful debate and fun.

Beyond these people, I have gotten to know a lot of people from many countries working on a wide range of fascinating topics – it has been a pleasure working with all of you.

Lastly, I want to thank the CRC 1173 for funding my research and providing me with the opportunity to fully focus on this work. The work put in by the administrative staff and the members creates the ideal conditions for ideas to flourish and for work to get done.

Contents

1 Summary and Conclusion

This work is the result of the author's time as a PhD student at the Karlsruhe Institute of Technology (KIT) and, in part, an extension of their Master's thesis. We present a hierarchical sweeping preconditioner, a modern and scalable method of solving large systems of linear equations derived by the discretization of the time-harmonic formulation of Maxwell's equations by means of the finite element method.

While there are some methods of solving such systems (direct solvers, classical sweeping, simplified models) it has been the author's observation that none of these scale well to system sizes as they may be required in fields like chip-to-chip interconnect optimization or other industrial fields of interest. As a consequence, this work is focused on the topic of providing a numerical scheme to solve the aforementioned partial differential equation on a larger scale. We will include an introduction to the required theoretical basis for this work and provide some numerical experiments. Additionally, we will be providing the full source code of the implemented method to make it easier for the reader to incorporate advancements from this work into their own.

In chapter 2 we introduce some basic nomenclature and in chapter 3 we provide a short motivation for this work and real-world applications. This is followed by an introduction to some basic concepts in chapter 4. This includes Maxwell's equations, modal theory, boundary conditions and transformation optics. Although many of these topics are likely known to the reader – we reference introductory materials to these topics and provide an overview if that is not the case.

Chapter 5 gives an introduction to the method of finite elements and sweeping preconditioners before introducing the hierarchical sweeping preconditioner – the central innovation presented in this work. We discuss several details about this concept, such as the choice of absorbing boundary condition and both direct and iterative solvers involved in the scheme. We also provide numerical results to show the properties of this method.

In chapter 6 we first provide an introduction to shape optimization which is tailored towards the application of classic optimization theory to wave propagation in waveguides. In this chapter, we discuss a hybrid method of domain and material optimization based on transformation optics that blends the advantages of both techniques and integrates well with the finite element method and thus sweeping preconditioners.

We end this work with an overview of the provided code base in chapter 7 and the complete documentation of the code in chapter 8. Since this documentation is extensive and should be used with internal links, it will only be included in electronic versions of this document.

2 Nomenclature

Nomenclature

2.1 Basic Terminology

In this work, we will represent

- scalar-valued quantities by normal, lowercase letters,
- vector-valued quantities by bold, lowercase letters,
- the components of a vector by a subscript index and
- matrices and operators by normal capital letters.

In systems that contain matrices in block notation, we will be using normal font for these blocks and block vectors.

The identity operator on a given space (i.e. $\mathbb{R}$) will be written as $\boldsymbol{Id}_{\mathbb{R}}$.

We will be using transformation optics for transformations dependent on a vector of parameters $p \in \mathbb{R}^N$. Whenever any object is marked by either $\cdot_p$ or $\cdot^p$, this states that the value relates to values in a coordinate system, transformed with the space-transformation for the parameters p. If multiple individual sets of parameters are relevant, p can also have indices. In that case p^i refers to the i-th set of parameters and p_j to the j-th parameter in the vector, so p^i_j references the j-th component of the i-th set of parameters. In this work, we will never apply an exponent to a parameter vector.

2.2 Fields

The electric field as known from physical observations:

$$\mathcal{E}(t) : \mathbb{R}^3 \to \mathbb{C}^3 \quad \text{for every} \quad t \geq 0.$$

The electric field in a transformed coordinate system where the transformation is described by a vector of parameters $p \in \mathbb{R}^{N_p}$:

$$\mathcal{E}_p(t) : \mathbb{R}^3 \to \mathbb{C}^3 \quad \text{for every} \quad t \geq 0.$$

Once a time-harmonic ansatz has been employed to resolve the time-dependence of the electric field we will write

$$\boldsymbol{E} : \mathbb{R}^3 \to \mathbb{C}^3 \quad \text{and} \quad \boldsymbol{E}_h : \mathbb{R}^3 \to \mathbb{C}^3$$

for the continuous and the discretized E-field. We will use $\boldsymbol{E}^*$ to denote the adjoint state and use the equivalent notation as introduced for electric fields for magnetic fields replacing $\boldsymbol{E}$ with $\boldsymbol{H}$ and $\mathcal{E}$ with $\mathcal{H}$.

The fundamental mode of a waveguide on a 2D-plane is written as

$$\boldsymbol{F}_0 : \mathbb{R}^2 \to \mathbb{C}^3.$$

There are two material properties, permittivity and permeability, which will be used in the following notations: In physical settings as ϵ (μ) and in the transformed setting ϵ_p (μ_p). They are defined as

$$\epsilon, \mu : \mathbb{R}^3 \to \mathbb{R} \quad \text{and}$$
$$\epsilon_p, \mu_p : \mathbb{R}^3 \to \mathbb{C}^{3\times 3} \quad \text{with}$$

$$\epsilon = \epsilon_0\, \epsilon_r\, \boldsymbol{\sigma} \qquad \text{and} \qquad \mu = \mu_0\, \mu_r\, \boldsymbol{\sigma} \tag{2.1}$$

for a transformation tensor $\boldsymbol{\sigma} : \mathbb{R}^3 \to \mathbb{R}^{3\times 3}$. For the materials under consideration in this work it holds that $\mu_r = 1$.

2.3 Domains

Without restriction, we regard problems in three spatial dimensions. We mainly distinguish between two terms: *computational domain* Ω_C and *domain of interest* Ω_I.

The domain of interest is the domain on which we intend to solve the partial differential equation in question. This solution, however, can require spatial truncation, which, if PML (see section 4.4.3) is used, introduces artificial domains which are not included in the domain of interest. Hardy-Space infinite elements also introduce artificial domains. In general the following holds

$$\Omega_I \subseteq \Omega_C \subset \mathbb{R}^3. \tag{2.2}$$

The waveguide will typically be considered in a transformed coordinate system, in which it is a axis parallel, rectangular cuboid. We choose z to be the propagation direction of the transmitted signal and will therefore be interested in how a signal propagates between some z_{in} and z_{out} with $z_{\text{in}} < z_{\text{out}}$ and the input and output interfaces:

$$\Gamma_{\text{in}} := \Omega_I|_{z=z_{\text{in}}} \qquad \Gamma_{\text{out}} := \Omega_I|_{z=z_{\text{out}}}. \tag{2.3}$$

Optimization problems specifically and waveguide computation in general will additionally split the domain of interest into one part considered the interior of the waveguide, the *core* Ω_I^{co} and the exterior of the waveguide, the *cladding* Ω_I^{cl}. For these domains, ϵ_r will take the values

$$\epsilon_{\text{r}}|_{\Omega_I^{\text{co}}} = 2.3409 \quad \text{and} \tag{2.4}$$
$$\epsilon_{\text{r}}|_{\Omega_I^{\text{cl}}} = 1.8496 \tag{2.5}$$

leading to the refractive indices of core and cladding

$$n_{\text{co}} = \sqrt{\epsilon_{\text{r}}|_{\Omega_I^{\text{co}}}} = 1.53 \quad \text{and} \quad n_{\text{cl}} = \sqrt{\epsilon_{\text{r}}|_{\Omega_I^{\text{cl}}}} = 1.36. \tag{2.6}$$

Additionally, we define the output interface of the waveguide core

$$\Gamma_{\text{out}} := \Omega_I^{\text{co}}\big|_{z=z_{\text{out}}}. \tag{2.7}$$

Any domain $\Omega_\#$ will be open and we will typically introduce them via their closure $\overline{\Omega_\#}$ with $\Omega_\# = \operatorname{int}(\overline{\Omega_\#})$ with the usual notation $\partial\Omega_\# = \overline{\Omega_\#} \setminus \Omega_\#$.

2.4 Constants

The complex unit will be written $\sqrt{-1} = \mathrm{i}$. We will also use the physical constants

$$\epsilon_0 \approx 8.8541878128 \cdot 10^{-12} \frac{As}{Vm},$$
$$\mu_0 \approx 1.25663706212 \cdot 10^{-6} \frac{N}{A^2},$$

for whom it holds

$$c = \frac{1}{\sqrt{\epsilon_0 \mu_0}},$$

in which $c = 2.99792458 \cdot 10^8 \frac{m}{s}$ is the speed of light.

For waveguides, we will be using $\lambda = 1550nm$, which is a standard wavelength for many applications in photonics.

2.5 Special functions and function spaces

We will use the symbol $\mathcal{L}$ to express the Laplace transform of a suitable function $f : [0, \infty) \to \mathbb{C}$

$$(\mathcal{L}f)(s) := \int_0^\infty f(t)e^{-st}\,\mathrm{d}t \tag{2.8}$$

as well as the parameter-dependent *Möbius* transformation

$$m_{\kappa_0} : \mathbb{C} \setminus \{1\} \to \mathbb{C}, \quad z \mapsto \mathrm{i}\kappa_0 \frac{z+1}{z-1} \tag{2.9}$$

for some $\kappa_0 \in \mathbb{C}$ with $\mathrm{Re}(\kappa_0) > 0$.

We will use the standard (see for example [Mon92]) definition of square integrable functions $L^2(\Omega)$, which is a Hilbert Space, on a domain Ω and the associated inner product

$$\langle u, v\rangle = \int_\Omega u \cdot \overline{v}\,\mathrm{d}x. \tag{2.10}$$

Additionally, we will be using the space of curl-conforming functions $H(\mathrm{curl}, \Omega)$ defined as

$$H(\mathrm{curl}, \Omega) = \{\boldsymbol{v} \in L^2(\Omega)^3 : \mathrm{curl}\,\boldsymbol{v} \in L^2(\Omega)\}. \tag{2.11}$$

3 Introduction

3.1 Motivation

Optoelectronic components consist of two major building-blocks: a substrate and the elements placed on them. Some functional groups work better or exclusively on a certain substrate material and this fact motivates multi-chip setups. In these setups, functional groups are placed on multiple wavers or different substrate wavers and have to be connected after the placing of the groups is completed. An example discussed in [Bil+18] consists of four such parts:

- distributed feedback lasers on an indium phosphate substrate,
- modulators on a silicon-on-insulator chip,
- arrayed-waveguide grating on a TriPlex chip and
- a single-moded fibre (SMF) to carry the signal these components produce.

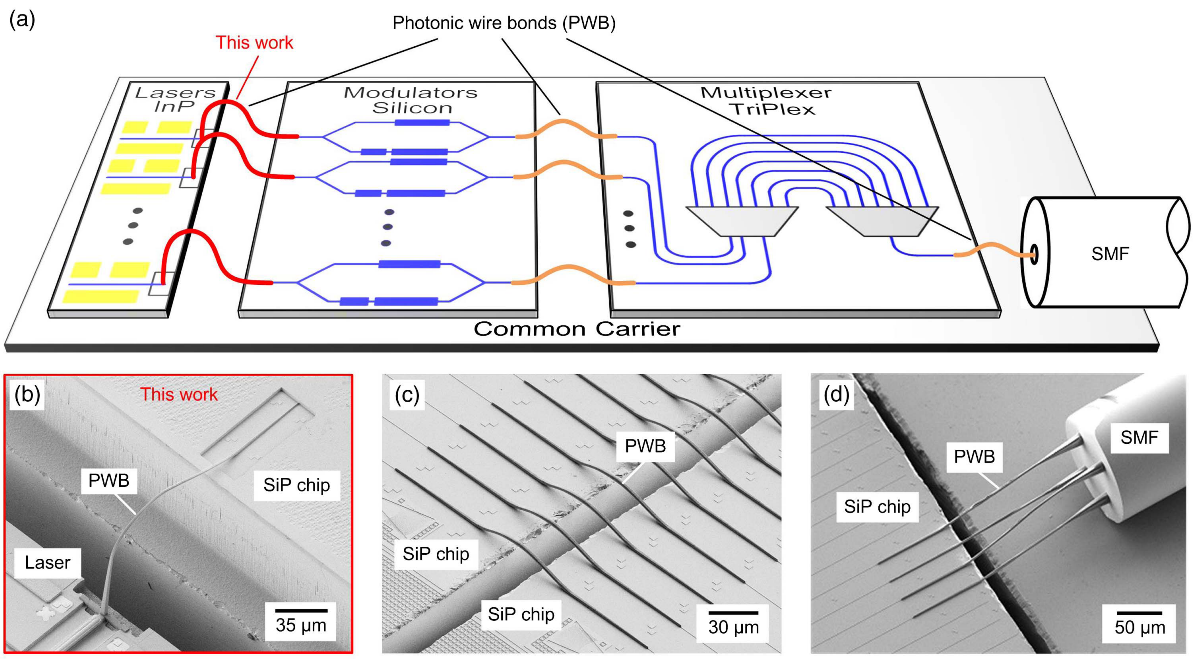


Figure 3.1: For details on this setup see [Bil+18].

These chips would be placed on a common carrier and connected by photonic wire bonds (PWB). Because placing the functional groups on their substrates and the substrates on the common carrier cannot be performed perfectly, the PWB-connections should take the actual position of the connectors after the groups have settled into account and should not be positioned ahead of time. The exact position of the connectors will vary with every manufactured chip and PWBs do not offer lossless signal transmission. As a consequence, a numerical scheme is required, that computes an ideal shape for the PWB connections given the actual position of the connectors. Such a scheme should be as fast as possible, because the manufacturing process would be waiting for the result. PWBs for chip-to-chip interconnects are a field of active research and industry scale manufacturing of high quality products is not yet possible.

Project C4 of the CRC 1173 "Wavephenomena" at KIT is aimed at providing algorithms for this problem. The numerical computation of a solution of the complete Maxwell-system that describes this setting in adequate spatial resolution as presented in this work takes minutes at best or days at worst. It is not likely that these run times can be reduced drastically enough to make them viable for online computation during the manufacturing process. Therefore, simplified models have been proposed in [Neg+18] (FMA) and in [Ott17] (Helmholtz and Half-Space-Matching). These methods, however, make assumptions on the solution, which may not always hold. To evaluate in which settings which fast method is most applicable, a complete solver was proposed to serve as a benchmarking tool to measure the fast methods against. This work describes the development and implementation of such a complete model.

The resulting method, however, extends beyond the scope of PWBs and is applicable to more general scattering problems arising from Maxwell's equations such as meta-materials with chiral properties [Gan+10] or cavity resonances as in [Kar+10]. The PWB example is merely a set of example applications.

3.2 Scope of this Work

In this document I present the work I have done on two related topics over the course of my PhD studies. In my master's thesis, I worked on a scheme to apply adjoint-based optimization to waveguides where finite elements are used to discretize Maxwell's equations. The work was centered around leveraging optimal numerical schemes to solve an otherwise difficult problem: time-harmonic Maxwell's equations. This set of equations can be rewritten as a second order partial differential equation (PDE). Numerical experiments show that the classic finite element method (FEM) runs into severe difficulties when applied to large 3D geometries, such as waveguides. The standard process here goes as follows:

1. **Discretize the PDE using finite elements.** This will generate a system matrix (A) and a right-hand side vector (b).
2. **Precondition the system.** We construct a matrix M that approximates the inverse of A.
3. **Solve the system.** We apply a solver for linear sets of equations to our system $AMy = b$ and $y = M^{-1}x$.

This game-plan is applicable in many settings and its simplicity is the reason for the great success of the finite element method. On paper, this remains true for time-harmonic Maxwell's equations. There are, however, substantial roadblocks once we go further into detail, which highlight that specific characteristics of the solutions of Maxwell's equations make this process impossible or at least more difficult. Without diving deep into the mathematical details in this introductory chapter, two problems can be identified on a qualitative level:

- Oscillation: Because the solutions of Maxwell's equations are highly oscillatory, the resolution of our finite elements, i.e. the number of unknowns per volume unit, has to be high to be able to accurately describe the solution. Large numbers of degrees of freedom in turn require lots of memory and make the application of solvers and preconditioners more numerically expensive.
- Coupling: Electromagnetic waves in a waveguide ideally do not dissipate – this is why we use them for telecommunication. This property has a negative effect on the linear system we are trying to solve. If a wave inserted at some point into the computational domain can travel over large distances without losing most of its amplitude, it is implied that the inverse of the system-matrix is not sparse, i.e., most entries of the inverse of the system matrix are not zero. Imagine standing on a large, open field and someone smoking a kilometer away. You would not smell the smoke produced a kilometer away, because it dissipates and the amount of smoke reaching your nose would be so small that you would not be able to perceive of it. On the other hand: If that person was using a laser-pointer and pointing it directly at you, you would notice it nearly independently of the distance. If we consider the propagation of smoke, the solution at your own location and the solution at the other

person's location do not couple (at least not very much). For the light propagation, however, they do. This implies that the inverse operator is dense and therefore expensive to compute and store on a computer.

There are multiple ways of dealing with these issues, some of which we will touch on in this work. We focused on ways, where we maintain the most physically accurate way of solving the system rather than switching to a somewhat similar system that does not have that problem. As a consequence, the preconditioner and solver we propose in this work are capable of solving time-harmonic Maxwell's equations at large scale, albeit being difficult to implement and run. The scheme is built for large-scale applications. To comply with modern expectations of Good Scientific Practice (GSP) this document in its electronic form also includes the complete, documented code. It will be referenced throughout this work to give the reader the possibility to extract building-blocks of the code rather than using the complete application. We also provide installation instructions for the code so it can be used as a tool.

This dissertation was part of the C4 project of the collaborative research center 1173 Wave Phenomena at KIT. The title of this project is *Modeling, design and optimization of 3D waveguides* and all these topics are discussed in this work. The project partners provided a set of waveguide shapes that serve as a benchmark set for different schemes to model losses of waveguides. Because the scheme we propose is highly efficient, we are also able to add another layer on top of merely computing solutions of Maxwell's equations: We also provide an adjoint-based optimization method, that allows for efficient shape optimization. The field of electromagnetic design does not only relate to loss-minimization of waveguides – also the topic of electromagnetic cloaking has been discussed extensively in the literature in recent years [HZH09], [LZ16], [LGG15] or [Rah+08]. The ability to perform optimization for the full time-harmonic Maxwell setting is remarkable since the base-problem is considered to be computationally expensive and thus optimization as an additional layer seems unrealistic. The method we propose requires a minimal amount of solutions and is capable of optimizing large geometries for reduced signal losses. It can also be extended for more generic problems like optical signal splitters. The main effort in this work went into the development of the hierarchical sweeping preconditioner as a way to extend the applicability of sweeping preconditioners – the chapter on optimization merely serves as an example of a possible application while hierarchical sweeping preconditioners are not restricted to the topic of optimization.

At the end of this work, we present several ways to develop this method further. For example, the hierarchical sweeping preconditioner can utilize GPUs for improved performance. We can also construct more advanced optimization schemes based on the proposed adjoint based computation of shape gradients to tackle specific problems. Additionally, we will provide details on the libraries used to build this implementation and some guidance and experiences with the development of a code base of this size.

4 Electromagnetics and Waveguide Theory

We want to compute how light propagates through waveguides. Therefore, our first step is an introduction of what light is. We will introduce the translation of the physical systems into mathematical ones and we will introduce **Maxwell's equations** and derive from them the **time-harmonic second order pde**. Once this foundation is laid, we will move on to give an introduction to the **modal theory of waveguides**. Formulating the kind of scattering problems we will be working on in this work, will require domain truncation techniques such as **Hardy-Space Infinite Elements (HSIE)** and **Perfectly Matched Layers (PML)**. While there are multiple ways of introducing PML, we will be choosing a path via **Transformation Optics** since it is a technique we will also rely on for the representation of geometry in forward problems as well as the formulation of a parameter space for an optimization problem in chapter 6.

4.1 Maxwell's Equations

While we will give some details about time-harmonic Maxwell's equations, we will refer to more complete introductions on this topic, such as [Mon92] and [KH14].

In the basic differential form there are 4 equations given for the vector fields $\mathcal{E}$, $\mathcal{D}$, $\mathcal{H}$, $\mathcal{B}$ and $\mathcal{J}$ as well as the scalar field ρ:

Faraday's Law of Induction states

$$\frac{\partial \mathcal{B}}{\partial t} + \nabla \times \mathcal{E} = \mathbf{0} \tag{4.1}$$

and states that a changing magnetic field induces an electric field.

Ampere's Law connects the remaining vector fields as

$$\frac{\partial \mathcal{D}}{\partial t} - \nabla \times \mathcal{H} = -\mathcal{J} \tag{4.2}$$

which we can read as *"A changing electric field induces a magnetic field or a transport of electrical charges"*.

Gauss' Electric and Magnetic Laws state that

$$\nabla \cdot \mathcal{D} = \rho \qquad \text{and} \qquad \nabla \cdot \mathcal{B} = 0 \tag{4.3}$$

expressing that electrically charged particles generate electric fields and magnetically charged particles do not exist.

In the next step, we will make the assumption of **linear materials**, stating that

$$\mathcal{B} = \mu \mathcal{H} \qquad \text{and} \qquad \mathcal{D} = \epsilon \mathcal{E} \tag{4.4}$$

where $\epsilon, \mu : \mathbb{R}^3 \to \mathbb{R}^{3\times 3}$. ϵ is called the **dielectric** or **permittivity tensor** and μ the **permeability tensor**.

An important distinction at this point are **isotropic** and **anisotropic** media. In the former case, both ϵ and μ can be written as

$$\mu = \mu_0 \, \mu_r \, I \qquad \text{and} \qquad \epsilon = \epsilon_0 \, \epsilon_r \, I \tag{4.5}$$

with $\mu_r, \epsilon_r : \mathbb{R}^3 \to \mathbb{R}$, $I = \boldsymbol{Id}_{\mathbb{R}^{3\times 3}}$ and $\mu_0, \epsilon_0 \in \mathbb{R}$ are positive constants known from observation. While the materials in the physical applications of this work are generally isotropic, we will nonetheless focus on the more general, anisotropic case. This has multiple reasons:

- When we perform domain truncation, we will be using PML, which introduces anisotropic material tensors as described in [Ged96] and theorem 4.3.1.
- Transformation optics introduces anisotropic materials as well and offers substantial simplifications for the implementation of efficient shape-optimization schemes as discussed in chapter 6.

Following **Ohm's Law** we define

$$\mathcal{J} = \sigma \, \mathcal{E} + \mathcal{J}_{\text{ext}} \tag{4.6}$$

where $\mathcal{J}_{\text{ext}}$ is the external current density and $\sigma : \mathbb{R}^3 \to \mathbb{R}$ is the **conductivity** of the medium. For our purposes, we will be assuming $\sigma = 0$ (a **dielectric** medium) and $\mathcal{J}_{\text{ext}} = 0$.

We insert eq. (4.6) with the properties of an anisotropic, linear medium eq. (4.4) into eq. (4.2) and eq. (4.1) and find the first order system

$$\begin{aligned} \nabla \times \mathcal{E} + \mu_0 \mu \frac{\partial \mathcal{H}}{\partial t} &= \mathbf{0} \\ \nabla \times \mathcal{H} - \epsilon_0 \epsilon \frac{\partial \mathcal{E}}{\partial t} &= \mathbf{0}. \end{aligned} \tag{4.7}$$

4.2 Time-harmonic Ansatz and Second Order PDE

As a next step, we will assume that the solution is **time-harmonic** which means it can be written as

$$\mathcal{E}(x,t) = e^{-\mathrm{i}\omega t} \boldsymbol{E}(x) \qquad \text{and} \qquad \mathcal{H}(x,t) = e^{-\mathrm{i}\omega t} \boldsymbol{H}(x) \tag{4.8}$$

for $\boldsymbol{E}, \boldsymbol{H} : \mathbb{R}^3 \to \mathbb{C}^3$ and a frequency $\omega \in \mathbb{R}$.

Inserting the time-harmonic fields into the first order eq. (4.7) yields

$$\begin{aligned} \nabla \times \boldsymbol{E} &= \mathrm{i}\omega\mu_0\mu \approx \quad \text{and} \\ \nabla \times \approx &= -\mathrm{i}\omega\epsilon_0\epsilon\boldsymbol{E}. \end{aligned} \tag{4.9}$$

In our setting, the right-hand side of eq. (4.6) is $\mathbf{0}$ and we can insert that fact into eq. (4.2). Additionally we combine eq. (4.8) with eq. (4.2) and eq. (4.1). By these steps we derive

$$\nabla \times \left(\mu^{-1} \nabla \times \boldsymbol{E}\right) - \epsilon\omega^2 \boldsymbol{E} = \mathbf{0} \qquad \text{in } \Omega, \tag{4.10}$$

$$\nabla \cdot (\epsilon \boldsymbol{E}) = \mathbf{0} \qquad \text{in } \Omega. \tag{4.11}$$

For more details see [Mon92].

4.3 Transformation Optics

We investigate how a change in the coordinate system changes the first order eq. (4.9). A first formulation of this method can be found in [WP96] and is formulated on the whole of $\mathbb{R}^3$.

Theorem 4.3.1 (Transformation Optics)**.** *For a given one-to-one and onto coordinate transformation $\boldsymbol{q} \in C^2(\mathbb{R}^3, \mathbb{R}^3)$ with the unit vectors $\boldsymbol{u}_i(\boldsymbol{x}) = \boldsymbol{q}^{-1}(\boldsymbol{q}(\boldsymbol{x}) + \mathbf{e}_i) - \boldsymbol{x}$ for $i \in \{1,2,3\}$ and the euclidean unit vectors $\mathbf{e}_i$ we can write eq. (4.7) in the transformed coordinate system as*

$$\begin{aligned} \nabla_{\boldsymbol{q}} \times \widehat{\boldsymbol{E}} \; - \; \mathrm{i}\mu_0 \widehat{\boldsymbol{\mu}}\, \widehat{\boldsymbol{H}} \; &= \; \mathbf{0}, \\ \nabla_{\boldsymbol{q}} \times \widehat{\boldsymbol{H}} \; + \; \mathrm{i}\epsilon_0 \widehat{\boldsymbol{\epsilon}}\, \widehat{\boldsymbol{E}} \; &= \; \mathbf{0} \end{aligned} \tag{4.12}$$

for the transformed material tensors

$$\widehat{\epsilon}_{ij} = \epsilon g_{ij} \left| \boldsymbol{u}_1 \cdot (\boldsymbol{u}_2 \times \boldsymbol{u}_3) \right| Q_1 Q_2 Q_3 (Q_i Q_j)^{-1} \quad \textit{and} \tag{4.13}$$

$$\widehat{\mu}_{ij} = \mu g_{ij} \left| \boldsymbol{u}_1 \cdot (\boldsymbol{u}_2 \times \boldsymbol{u}_3) \right| Q_1 Q_2 Q_3 (Q_i Q_j)^{-1}. \tag{4.14}$$

We have used the definitions

$$q_i(\boldsymbol{x}) = (\boldsymbol{q}(\boldsymbol{x}))_i\,, \tag{4.15}$$

$$Q_{ij} = \frac{\partial x_1}{\partial q_i}\frac{\partial x_1}{\partial q_j} + \frac{\partial x_2}{\partial q_i}\frac{\partial x_2}{\partial q_j} + \frac{\partial x_3}{\partial q_i}\frac{\partial x_3}{\partial q_j}, \tag{4.16}$$

$$Q_i = \sqrt{Q_{ii}} \quad \textit{and} \tag{4.17}$$

$$\boldsymbol{g} = \begin{pmatrix} \boldsymbol{u}_1 \cdot \boldsymbol{u}_1 & \boldsymbol{u}_1 \cdot \boldsymbol{u}_2 & \boldsymbol{u}_1 \cdot \boldsymbol{u}_3 \\ \boldsymbol{u}_2 \cdot \boldsymbol{u}_1 & \boldsymbol{u}_2 \cdot \boldsymbol{u}_2 & \boldsymbol{u}_2 \cdot \boldsymbol{u}_3 \\ \boldsymbol{u}_3 \cdot \boldsymbol{u}_1 & \boldsymbol{u}_3 \cdot \boldsymbol{u}_2 & \boldsymbol{u}_3 \cdot \boldsymbol{u}_3 \end{pmatrix}^{-1}. \tag{4.18}$$

Proof. See [WP96]. □

Remark (Transformation Optics)**.** *The concept of transformation optics can be visualized very effectively on the example of a linear coordinate transformation such as $\boldsymbol{q} : \mathbb{R}^3 \to \mathbb{R}^3 : \boldsymbol{x} \mapsto 3\boldsymbol{x}$. In the new coordinates, the wavelength of the solution would be reduced by a factor of 3. From physics we know, however, that a reduction in wavelength is associated with a higher optical density of the material, so the expected outcome would be that the optical density of the material would be improved in this system, which is exactly what happens if we insert our coordinate transformation into the equations listed above. The refractive index $n = \sqrt{\epsilon_r}$ of a material has the effect of reducing the wavelength by a factor of $\frac{1}{n}$ compared to the vacuum wavelength so we would expect to see $\epsilon_r = 9$ in the new coordinate system, which is exactly what we get when we insert our transformation.*

The transformations we will be using in this work will typically have a shape like

$$\boldsymbol{T}_p(x_1, x_2, x_3) = (x_1, x_2 - m_p(x_3), x_3)^T. \tag{4.19}$$

In this formulation, $m_p(x_3) : \mathbb{R} \to \mathbb{R}$ is the vertical displacement of the central axis of the waveguide given by a set of N parameters $p \in \mathbb{R}^N$. This transformation has the disadvantage of varying the height of the waveguide in the direction of light propagation, meaning that the waveguide is slimmer in the propagation direction if $\frac{\partial m_p}{\partial x_3}(\cdot) \neq 0$. The example waveguides provided in our project, however, have this property, too, meaning that the relatively simple formulation eq. (4.19) is reasonable. For the rectangular waveguide geometries described in section 5.4.5, this transformation for an appropriate set of parameters P will map the bent waveguide shape in the physical coordinate system onto an axis-parallel rectangular cuboid.

For a cylindrical and bent waveguide of radius $r_p(x_3)$ and vertical displacement $m_p(x_3)$ we can use the transformation

$$\boldsymbol{q}_p(x_1, x_2, x_3) = \left(\frac{x_1}{r_p(x_3)}, \frac{x_2 - m_p(x_3)}{r_p(x_3)}, x_3 \right)^T. \tag{4.20}$$

Theorem 4.3.2 (Jacobian Formulation)**.** *For a one-to-one and onto coordinate transformation* $\boldsymbol{q}_p(x_1, x_2, x_3) \in C^1(\mathbb{R}^3)$ *with the jacobian*

$$\mathcal{J} = \begin{pmatrix} \frac{\partial q_1}{\partial x_1} & \frac{\partial q_1}{\partial x_2} & \frac{\partial q_1}{\partial x_3} \\ \frac{\partial q_2}{\partial x_1} & \frac{\partial q_2}{\partial x_2} & \frac{\partial q_2}{\partial x_3} \\ \frac{\partial q_3}{\partial x_1} & \frac{\partial q_3}{\partial x_2} & \frac{\partial q_3}{\partial x_3} \end{pmatrix} \tag{4.21}$$

we can rewrite the formulation of the material tensors above to

$$\widehat{\boldsymbol{\epsilon}_p} = \frac{\mathcal{J}\,\boldsymbol{\epsilon}\,\mathcal{J}^T}{\det(\mathcal{J})} \tag{4.22}$$

and

$$\widehat{\boldsymbol{\mu}_p} = \frac{\mathcal{J}\,\boldsymbol{\mu}\,\mathcal{J}^T}{\det(\mathcal{J})}. \tag{4.23}$$

Proof. A complete proof of this can be found in [NZG10]. □

This formulation lends itself well to be implemented in numerical code because the jacobian can be computed either explicitly or by symbolic differentiation. Once $\mathcal{J}$ is implemented, $\mathcal{J}^T$ and $\det(\mathcal{J})$ can be computed cheaply. The original formulation (see theorem 4.3.1) requires the computation of the unit-vectors in the transformed coordinate system as well as all the terms in the jacobian, which are part of the terms Q_i. The formulation based on the jacobian, however, can be computed very efficiently and the determinant of $\mathcal{J}$ can also be implemented efficiently since it is a constant expression for 3×3 matrices.

4.4 Boundary Conditions

So far, we have discussed Maxwell's equations on a possibly unbounded, generic domain $\Omega \subset R^3$. We plan to solve this set of equations by means of numeric discretization, which will introduce a number of degrees of freedom that scales linearly with the volume of the computational domain – as a consequence, we cannot solve the problem on an infinite domain. We will therefore need to truncate Ω to a finite subset of $\mathbb{R}^3$ and impose appropriate boundary conditions on the surface.

In the following subsections, we will discuss some basic types of boundary conditions we will employ in this work. Section 4.4.1 introduces the most basic boundary condition – setting the tangential part of the solution on the boundary to zero. Next, Dirichlet boundaries will be introduced, which set the values on the surface to a fixed value that is not necessarily zero and finally, we will introduce the absorbing boundary conditions **Perfectly Matched Layers** in section 4.4.3 and **Hardy Space Infinite Elements** in section 4.4.4.

4.4.1 PEC

Perfect Electrical Conductor (or **PEC**) boundary conditions are derived from the physical model of metals. In materials with free charges, no electrical field can build up because the free charges move to negate the effect. As a consequence, the electrical field is zero in an electrical conductor. If the domain is truncated by a PEC layer, it means that the surface tangential components of the E-field are zero. Nédélec-elements (see section 5.1.2) are perfectly suited for the application of this boundary condition, since their surface degrees of freedom represent the tangential components of the solution. Application of this boundary condition on a given surface therefore only requires the identification of all cell faces on the surface, enumerating their degrees of freedom and forcing their value to be zero.

Regarding wave propagation, such a boundary condition in a physical setting creates reflections (the field generated by the movement of electrons reacting to the incoming field). This method should therefore only be applied to a metal object in the domain of interest or in areas, where the field can be assumed to be (tangentially) zero. Otherwise, incoming signals will be handled incorrectly. While we will not discuss any metal objects as part of the computational domain in this work, we will be using PEC boundary conditions as part of our PML implementation (see section 4.4.3).

Mathematically, this kind of boundary condition is described by the expression

$$\boldsymbol{E}(\boldsymbol{x}) \times \boldsymbol{n}_{\Gamma_0}(\boldsymbol{x}) = \boldsymbol{0} \quad \forall \boldsymbol{x} \in \Gamma_0, \tag{4.24}$$

where $\boldsymbol{n}(\boldsymbol{x})$ is the outer orthogonal vector to some surface Γ_0 at $\boldsymbol{x}$. We will be using $\Gamma_0 = \partial\Omega_C \setminus \Gamma_{\text{in}}$.

4.4.2 Dirichlet

If the tangential components of the solution are known to be equal to the tangential components of $\boldsymbol{E}_I$ on a surface Γ_D, we can set

$$\boldsymbol{E}(\boldsymbol{x}) \times \boldsymbol{n}_{\Gamma_D}(\boldsymbol{x}) = \boldsymbol{E}_I(\boldsymbol{x}) \times \boldsymbol{n}_{\Gamma_D}(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Gamma_D. \tag{4.25}$$

In this case, we call $\boldsymbol{E}_I$ an incident field. These boundary conditions are commonly referred to as Dirichlet boundary conditions and we can see that PEC boundary conditions are a special case of these conditions.

This formulation becomes much more straight forward in the case of an axis-parallel surface Γ_D. We assume w.l.o.g. that $\Gamma_D = \mathbb{R}\times\mathbb{R}\times\{0\}$ and the waveguide is parallel to the z-axis with the signal travelling from the surface Γ_D along the z-direction towards $+\infty$. We find the outward normal vector

$$\boldsymbol{n}(\boldsymbol{x}) = (0,0,-1)^T \quad \forall \boldsymbol{x} \in \Gamma_D. \tag{4.26}$$

Inserting eq. (4.26) in eq. (4.25) we find

$$\begin{pmatrix} -E_2 \\ E_1 \\ 0 \end{pmatrix}(\boldsymbol{x}) = \boldsymbol{E}(\boldsymbol{x}) \times \boldsymbol{n}_{\Gamma_D}(\boldsymbol{x}) = \boldsymbol{E}_I(\boldsymbol{x}) \times \boldsymbol{n}_{\Gamma_D}(\boldsymbol{x}) = \begin{pmatrix} -(E_I)_2 \\ (E_I)_1 \\ 0 \end{pmatrix}(\boldsymbol{x}). \tag{4.27}$$

We observe that this boundary condition is equivalent to setting the tangential components of the solution to the values of the tangential components of the incident field. How to derive a signal $\boldsymbol{E}_I$ will be discussed in section 4.5.

4.4.3 PML

Next, we will regard the coordinate transformation

$$q_1 = x_1, \tag{4.28}$$
$$q_2 = x_2 \quad \text{and} \tag{4.29}$$
$$q_3 = \sigma\, x_3 \tag{4.30}$$

for $\sigma \in \mathbb{C}$ and $\sigma \neq 0$.

We evaluate the definitions in eq. (4.15) for eq. (4.28) and find

$$\boldsymbol{Q} = (Q_{ij}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \sigma^{-2} \end{pmatrix} \tag{4.31}$$

$$Q_1 = 1 \qquad Q_2 = 1 \qquad Q_3 = \sigma^{-1}. \tag{4.32}$$

Inserting these results in eq. (4.13) we find

$$\widehat{\epsilon}_{11} = \widehat{\epsilon}_{22} = \epsilon_{11} \underbrace{1}_{g_{33}} \quad \underbrace{\sigma^{-1}}_{|\boldsymbol{u}_1 \cdot (\boldsymbol{u}_2 \times \boldsymbol{u}_3)|} \quad \underbrace{\sigma^{-1}}_{Q_1 Q_2 Q_3 (Q_1 Q_1)^{-1}} = \epsilon_{11}\, \sigma^{-2} \tag{4.33}$$

$$\widehat{\epsilon}_{33} = \epsilon_{33} \underbrace{\sigma^{2}}_{g_{33}} \quad \underbrace{\sigma^{-1}}_{|\boldsymbol{u}_1 \cdot (\boldsymbol{u}_2 \times \boldsymbol{u}_3)|} \quad \underbrace{\sigma}_{Q_1 Q_2 Q_3 (Q_3 Q_3)^{-1}} = \epsilon_{33}\, \sigma^{2}. \tag{4.34}$$

For a given direction (here we use x_3 as our example), we find tensors of the shape

$$\epsilon = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a^{-1} \end{pmatrix} \quad \text{for some} \tag{4.35}$$

$$a \in \{x \in \mathbb{C} \,:\, \mathrm{Im}(x) > 0\}. \tag{4.36}$$

These are the material tensors attained for the so-called **uniaxial PML** or **UPML**. This transformation has the effect of damping the field exponentially and can therefore be used for domain truncation. We will now regard the implications of this technique in more detail.

Our goal is to build a system that we can solve numerically. Our current system eq. (4.9) is, however, not restricted to a finite domain, implying an infinite number of degrees of freedom upon discretization. Truncation of this real domain to a domain of interest introduces a need for a truncation method. The most used technique for the truncation of Maxwell's equations from a global formulation to a restriction to a domain of interest is the method of **Perfectly Matched Layers (PML)**. A basic introduction to PML can be found in [Ber94] – the more specialized version of this method UPML was formulated in [Ged96].

The concept of these media is that the imaginary part of the material tensors has the effect of exponentially dampening the incident wave as it propagates outward. After traversing the entire PML domain, the wave amplitude is assumed to be damped far enough to fulfill PEC boundary conditions (see section 4.4.1), meaning that the tangential components of the field are zero on the boundary. To motivate the assumption that this effect holds, [Zsc+05] introduces the following example:

We assume that $\epsilon_r, \mu_r \in \mathbb{R}$ are constant on $\Omega_r := \mathbb{R} \times \mathbb{R} \times (0, +\infty)$, the right half space. It is known that in this setting, a TE mode solution of the Maxwell system satisfies the Helmholtz equation

$$\frac{\partial^2 E_y}{\partial x^2}(x) + \underbrace{\left(\omega^2 \mu \epsilon - k_z^2\right)}_{\phi^2} E_y(x) = 0 \quad \text{for} \quad x \in (0, +\infty). \tag{4.37}$$

The general solution to this system can be written as

$$E_y(x) = C_1 e^{\mathrm{i}\phi x} + C_2 e^{-\mathrm{i}\phi x}. \tag{4.38}$$

With the definition that the real part of the square root is positive we call the first term an outgoing wave and the second part an incoming wave. If a wave therefore propagates into Ω_r and no backward reflections occur in the physical setting, we have $C_1 \neq 0$ and $C_2 = 0$. Using the Laplace transformation eq. (2.8) we find

$$\mathcal{L}E_y(s) = \frac{C_1}{s - \mathrm{i}\phi} + \frac{C_2}{s + \mathrm{i}\phi}. \tag{4.39}$$

The two terms on the right hand side introduce one pole each, at $s_\pm = \pm i\phi$. $\mathcal{L}E_y$ is holomorphic everywhere else and therefore, the following statements are equivalent:

1. $C_2 = 0$.
2. The Laplace transform of the solution of eq. (4.37) $\mathcal{L}E_y$ is holomorphic on the lower complex half plane.
3. The solution E_y of eq. (4.37) is called *outgoing*.

This is referred to as the **Pole condition** which we will get back to later on in this work.

We now observe the general solution along the line $(1 + \mathrm{i}\sigma)x$ for $\sigma \in \mathbb{R}$ with $\sigma > 1$ and find

$$E_y(x) = C_1 \underbrace{e^{\mathrm{i}\phi\,(1+\mathrm{i}\sigma)\,x}}_{\to 0,\quad x\to\infty} + C_2 \underbrace{e^{-\mathrm{i}\phi\,(1+\mathrm{i}\sigma)\,x}}_{\to\infty,\quad x\to\infty}. \tag{4.40}$$

If we set $E_y(d) = 0$ for the PML domain thickness d we find

$$0 = C_1 e^{\mathrm{i}\phi\,(1+\mathrm{i}\sigma)\,d} + C_2 e^{-\mathrm{i}\phi\,(1+\mathrm{i}\sigma)\,d}$$

and thus

$$\frac{C_2}{C_1} = -e^{2\mathrm{i}\phi\,(1+\mathrm{i}\sigma)\,d} = -e^{2\mathrm{i}\phi d} e^{-2\phi\sigma d} \tag{4.41}$$

leading to

$$\frac{|C_2|}{|C_1|} = \left|e^{2\mathrm{i}\phi d}\right| \cdot \left|e^{-2\phi\sigma d}\right| = \underbrace{\left|e^{-2\phi\sigma d}\right|}_{\to 0,\quad d\to+\infty}. \tag{4.42}$$

So as we increase d or σ and assume $C_1 = 1$ we find that C_2 decays exponentially. We therefore use $\sigma = 1 + \mathrm{i}\sigma_d$ for $\sigma_d \in \mathbb{R}$.

The second property of the PML we need to investigate is the question of reflections on the surface of the PML domain. For the numerical solution to be representative of the physical wave propagation, the artificial PML medium may not introduce artificial reflections as the wave passes from the domain of interest into the PML medium. This topic is discussed in [Ged96]. In the continuous case for

$$\boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_x^z & 0 & 0 \\ 0 & \epsilon_y^z & 0 \\ 0 & 0 & \epsilon_z^z \end{pmatrix} \quad \text{and} \quad \boldsymbol{\mu} = \begin{pmatrix} \mu_x^z & 0 & 0 \\ 0 & \mu_y^z & 0 \\ 0 & 0 & \mu_z^z \end{pmatrix}, \tag{4.43}$$

where $\cdot^z$ denotes the fact that this material is associated with a spatial truncation in z-direction, it is shown that for

$$\epsilon_x^z = \epsilon_y^z = (\epsilon_z^z)^{-1} = \mu_x^z = \mu_y^z = (\mu_z^z)^{-1} \tag{4.44}$$

the PML is reflection-less. This, however, only holds in the continuous case – for the discretized setting the authors in [TH05] show numerical reflections on the surface of the medium, which can be reduced in several ways:

- By increasing the spatial resolution and reducing the value σ.
- By increasing the degree of the method that is being used for the discretization, i.e. the order of the finite element (see chapter 5).

- By applying the PML via a smooth and increasing profile

$$\epsilon_z^z = 1 + \mathrm{i}\sigma(z) \quad \text{for} \quad \sigma(z) = \left(\frac{z}{d}\right)^k \sigma_{\max} \tag{4.45}$$

  for a PML medium beginning at $z = 0$ and typical values $k = 3$ or $k = 4$.

While the first two options increase the number of degrees of freedom directly, the third options has a similar, indirect effect. The dampening effect of the PML near the interface is very low since z^k suppresses the dampening effect. Merely increasing the value of $\sigma_{\max}$, however, does not solve the problem. If the reflections depend on the difference between σ on neighboring quadrature points, increasing $\sigma_{\max}$ increases the amount of reflections as shown in fig. 4.1.

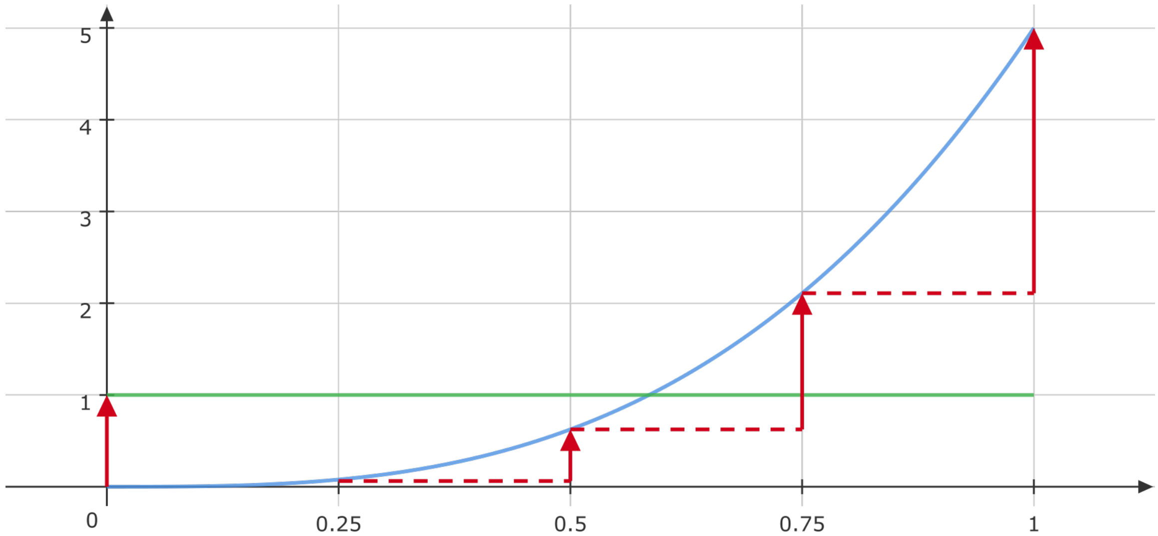


Figure 4.1: Comparison of constant σ (green) and $\sigma(x) = 5x^3$ (blue) for $h = 0.25$ in the PML domain. Red arrows indicate the difference of $\sigma(x)$ at the evenly spaced quadrature points. For constant σ, we only observe a jump at $x = 0$. For the increasing profile $\sigma(x)$ (the blue curve) we observe no jump at $x = 0$ and smaller increases for neighboring quadrature points 0, 0.25 and 0.5. Beyond that, however, the increase between neighboring quadrature points is larger than that at 0 for constant σ.

The dampening performance of the PML depends on

$$\int_0^d \sigma(x)\, \mathrm{d}x = \frac{1}{k+1}\sigma_{\max}$$

so for a similar dampening performance of PML domains for $k = 0$, $k = 3$ and $k = 4$ we would find

$$\sigma_{\max}^{k=0} = \frac{1}{4}\sigma_{\max}^{k=3} = \frac{1}{5}\sigma_{\max}^{k=4}. \tag{4.46}$$

The increased value of $\sigma_{\max}$, however, means that the steps of $\sigma(x)$ on neighboring quadrature points increase – once again introducing reflections. Ultimately, this can only be resolved by increasing the number of degrees of freedom via either h or p refinement.

We can employ the ansatz presented above for truncation in every spatial direction and thus envelop the computational domain in layers of PML to limit the computational domain to a finite subset of $\mathbb{R}^3$. A 2D schematic overview of a PML setup with an input interface and PML for domain truncation in the 3 other directions ($-y$, $+y$ and $+z$) is shown in fig. 4.2. In [Sch15] the author discusses additional details about applications of PML media for domain truncation such as different boundary conditions on the back side of the PML medium.

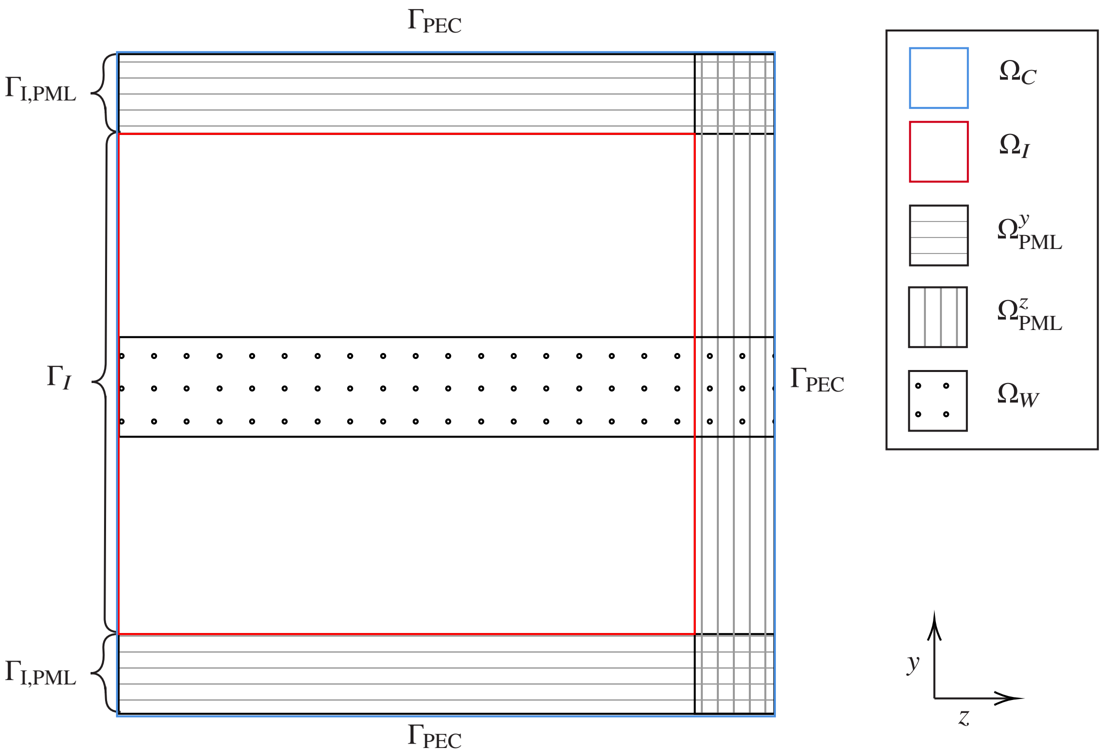


Figure 4.2: Schematic of the computational domain Ω_C, the domain of interest Ω_I, PML domains for truncation in $\pm y$ direction Ω^y_{PML} and in z direction Ω^z_{PML}, the waveguide domain Ω_W as well as the PEC boundary Γ_{PEC} and the input interface Γ_I with its overlap on PML domains $\Gamma_{\text{I,PML}}$

4.4.4 HSIE

Remark. *We will not use this method for domain truncation in this work. These elements according to the literature and our own initial testing have better properties initially when comparing them with PML. They approximate the solution by oscillating functions in the exterior and even for low polynomial degree in the Hardy space, they consistently outperform PML methods. However, these elements can not be used in sweeping preconditioners. To explain that problem we introduce them briefly and then discuss the problem as soon as it arises in section 5.6.*

For both Helmholtz- and Maxwell-type scattering problems, **Hardy-Space Infinite Elements** have been introduced as an alternative method of truncating the computational domain. They use a definition of outgoing solutions to derive a closed formulation of the integrals arising from a variational approach with Nédélec-elements (see section 5.1.2) on infinite cells. We will introduce the building blocks of this method here to elaborate on details that complicate the application of this method to a sweeping preconditioner and refer the reader to the relevant literature (see [NS11] and [Nan+11]) for more detailed introductions of specific topics as well as numerical results.

Definition 4.4.1 (Hardy Spaces). *For*

$$\mathbb{C}^{\pm} = \{s \in \mathbb{C} : \operatorname{Im}(\pm s) > 0\} \tag{4.47}$$

and

$$g^{\pm}(u, y) = \int_{\mathbb{R}} |u(x \pm \mathrm{i}y)|^2 \,\mathrm{d}x \tag{4.48}$$

we define the Hardy-Space $H^{\pm}(\mathbb{R})$ as the set of functions u that are L^2 boundary values of some function v, which is holomorphic on $\mathbb{C}^{\pm}$ (respectively) with uniformly bounded $g^{\pm}(v, y)$ for $y > 0$.

Lemma 4.4.1 (Hilbert Spaces)**.** *The spaces $H^{\pm}(\mathbb{R})$ equipped with*

$$\langle u, v\rangle = \int_{\mathbb{R}} u \cdot v \,\mathrm{d}x \tag{4.49}$$

are Hilbert spaces.

Proof. See [Dur70]. □

Lemma 4.4.2 (Paley–Wiener)**.** *The Hardy spaces $H^{+}(\mathbb{R})$ and $H^{-}(\mathbb{R})$ have the representation*

$$H^{\pm}(\mathbb{R}) = \left\{ u \in L^2(\mathbb{R}) \ : \ \int_{\mathbb{R}} e^{\pm \mathrm{i}st} u(s) \,\mathrm{d}s = 0 \quad \textit{for almost all } t > 0 \right\} \tag{4.50}$$

and they provide an orthogonal decomposition of $L^2(\mathbb{R})$.

Proof. See [Hof14]. □

Definition 4.4.2 (Möbius transformation and $H^{+}(S^1)$)**.** *For $S^1 = \{z \in \mathbb{C} : |z| = 1\}$, the complex unit disc $D = \{z \in \mathbb{C} : |z| < 1\}$ and $\kappa_0 > 0$, we define the* ***Möbius transformation***

$$\phi_{\kappa_0} : D \longrightarrow \mathbb{C}^{-} \ x \mapsto \mathrm{i}\kappa_0 \frac{z+1}{z-1} \tag{4.51}$$

and the Hardy space of the unit disc

$$H^{+}(S^1) := \left\{ \frac{(u \circ \phi_{\kappa_0})(z)}{z-1} \,|\, u \in H^{-}(\mathbb{R}) \right\}. \tag{4.52}$$

This notation has been introduced to facilitate the definition of an outgoing solution of the problem

$$u'' + \kappa^2 u(x) = 0 \tag{4.53}$$

for $x \geq 0$. This problem has the general solution

$$u(x) = C_1 e^{i\kappa x} + C_2 e^{-i\kappa x}. \tag{4.54}$$

We call such a solution u **incoming** iff $C_1 = 0$ and **outgoing** iff $C_2 = 0$.

Lemma 4.4.3 (Pole Condition)**.** *A solution u of eq. (4.54) is outgoing if it holds that*

- *$\mathcal{L}u$ (see eq. (2.8)) has no poles with negative imaginary part,*
- *we can express $\mathcal{L}u$ as*

$$\mathcal{L}u(s) = \frac{C_1}{s - \mathrm{i}\kappa} + \frac{C_2}{s + \mathrm{i}\kappa}, \tag{4.55}$$

- *and $\mathcal{L}u$ belongs to $H^{-}(\mathbb{R})$.*

Proof. See [Sch02]. □

Combining the parameter dependent Möbius transformation (see eq. (2.9)) and the Laplace transformation (see eq. (2.8)), we define the core operator used in Hardy space infinite elements:

$$\mathcal{H}_{\kappa_0} u(z) := \mathcal{L}u\left(m_{\kappa_0}(z)\right)\,(z-1)^{-1}. \tag{4.56}$$

Lemma 4.4.4. *For $u, v \in H^-(\mathbb{R})$,*

$$U := \mathcal{H}_{\kappa_0} u \quad \text{and} \quad V := \mathcal{H}_{\kappa_0} v \tag{4.57}$$

the following equation holds:

$$a(U,V) := \int_0^\infty u(\xi)v(\xi)\,\mathrm{d}\xi = -\frac{\mathrm{i}\kappa_0}{\pi}\int_{S^1} U(z)V(\overline{z})|\,\mathrm{d}z|. \tag{4.58}$$

Proof. See [HN09]. □

Using this identity, we can now express the external integrals and their integrals to infinity as elements of the Hardy space $H^+(S^1)$ and to transform their integrals to the bounded domain S^1, where we can evaluate them explicitly. We know that monomials form a basis of $H^+(S^1)$ and will therefore express elements in $H^+(S^1)$ by this basis.

Next, we need a way to express derivatives ∂_ξ. First we define

$$\mathcal{T}_\pm(u_0, U)(z) = \frac{u_0 + (z \pm 1)U(z)}{2}. \tag{4.59}$$

Lemma 4.4.5. *For u continuously differentiable with*

$$\lim_{z\to\infty} e^{-zt}u(z) = 0 \tag{4.60}$$

it holds that

$$\mathcal{L}(u')(z) = z\mathcal{L}(u)(z) - u_0, \tag{4.61}$$

where $u_0 = u(0)$.

Proof. We apply integration by parts and find

$$\mathcal{L}(u')(z) = \int_{t=0}^\infty e^{-zt}u'(t)\,\mathrm{d}t \tag{4.62}$$

$$= \int_{t=0}^\infty ze^{-zt}u(t)\,\mathrm{d}t + \left[u(z)e^{-zt}\right]_{t=0}^\infty \tag{4.63}$$

$$= z\mathcal{L}(u)(z) - u_0. \tag{4.64}$$

□

We define $\mathcal{H}_{\kappa_0}(u) = \frac{1}{\mathrm{i}\kappa_0}\mathcal{T}_-(u_0, U)$ and find

$$\mathcal{H}_{\kappa_0}(\partial_\xi u)(z) = m_{\kappa_0}(z\mathcal{L}(u)(z) - u_0) \tag{4.65}$$

$$= \mathrm{i}\kappa_0 \frac{z+1}{i-1}\frac{u_0 + (z-1)\widehat{U}(z)}{2\mathrm{i}\kappa_0} - \frac{u_0}{z-1} \tag{4.66}$$

$$= \frac{u_0 + (z+1)\widehat{U}(z)}{2} \tag{4.67}$$

$$= \mathcal{T}_+(u_0, \widehat{U})(z) \tag{4.68}$$

$$= \mathrm{i}\kappa_0 \mathcal{T}_+\mathcal{T}_-^{-1}\mathcal{H}_{\kappa_0}(u)(z). \tag{4.69}$$

where $\widehat{U}$ is defined by

$$\mathcal{H}_{\kappa_0}(u) = \frac{1}{\mathrm{i}\kappa_0}\mathcal{T}_-(u(0), \widehat{U}). \tag{4.70}$$

With this observation, we define

$$\widehat{\partial_\xi} := \mathrm{i}\kappa_0 \mathcal{T}_+ \mathcal{T}_-^{-1}. \tag{4.71}$$

We can now express functions in the exterior domain by their basis vectors in $H^+(S^1)$ and evaluate both integrals to infinity as well as derivatives in the infinite direction by basic operations on polynomials.

Based on these definitions, [Nan+11] introduces the following formulations:

$$\int_K \boldsymbol{u} \cdot \boldsymbol{v} \,\mathrm{d}x = A_{G,T}\left(((\mathcal{D} \otimes \boldsymbol{Id})U_1, U_2, U_3)^T, ((\mathcal{D} \otimes \boldsymbol{Id})V_1, V_2, V_3)^T\right) \quad \text{and} \tag{4.72}$$

$$\int_K \nabla \times \boldsymbol{u} \cdot \nabla \times \boldsymbol{v} \,\mathrm{d}x = A_{C,T}\left(\begin{pmatrix} (\mathcal{I} \otimes \nabla_{\widehat{x}}^{(1)})U_3 - (\mathcal{I} \otimes \nabla_{\widehat{x}}^{(2)})U_2 \\ (\boldsymbol{Id} \otimes \nabla_{\widehat{x}}^{(2)})U_1 - (\partial_\xi \otimes \boldsymbol{Id})U_3 \\ (\partial_\xi \otimes \boldsymbol{Id})U_2 - (\boldsymbol{Id} \otimes \nabla_{\widehat{x}}^{(1)})U_1 \end{pmatrix}, \begin{pmatrix} (\mathcal{I} \otimes \nabla_{\widehat{x}}^{(1)})V_3 - (\mathcal{I} \otimes \nabla_{\widehat{x}}^{(2)})V_2 \\ (\boldsymbol{Id} \otimes \nabla_{\widehat{x}}^{(2)})V_1 - (\partial_\xi \otimes \boldsymbol{Id})V_3 \\ (\partial_\xi \otimes \boldsymbol{Id})V_2 - (\boldsymbol{Id} \otimes \nabla_{\widehat{x}}^{(1)})V_1 \end{pmatrix}\right) \tag{4.73}$$

where K denotes the reference cell and one uses

$$\mathcal{I}^{-1} = \mathcal{D} = \boldsymbol{Id} + \frac{1}{2\mathrm{i}\kappa_0}\begin{pmatrix} -1 & 1 & & \\ 1 & -3 & 2 & \\ & 2 & -5 & \ddots \\ & & \ddots & \ddots \end{pmatrix}, \tag{4.74}$$

$$G^{-1} = C = \frac{\widehat{\boldsymbol{J}}\widehat{\boldsymbol{J}}^T}{|\widehat{\boldsymbol{J}}|} \quad \text{and} \tag{4.75}$$

$$\begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix}(\xi, \widehat{x}) = \widehat{\boldsymbol{J}}^T(\widehat{x})\begin{pmatrix} \mathcal{H}_{\kappa_0} u_1 \circ F(\xi, \widehat{x}) \\ \mathcal{D}\mathcal{H}_{\kappa_0} u_2 \circ F(\xi, \widehat{x}) \\ \mathcal{D}\mathcal{H}_{\kappa_0} u_3 \circ F(\xi, \widehat{x}) \end{pmatrix}, \tag{4.76}$$

$$A_{X,T}\left((\phi_1, \phi_2, \phi_3)^T, (\psi_1, \psi_2, \psi_3)^T\right) = \int_T \sum_{i,j=1}^{3} X_{ij}(\widehat{x})\, a\left(\phi_i(\cdot, \widehat{x}), \psi_j(\cdot, \widehat{x})\right) \mathrm{d}\widehat{x} \quad \text{and} \tag{4.77}$$

$$a(U, V) = -\frac{\mathrm{i}\kappa_0}{\pi}\int_{S^1} U(z)V(\overline{z})|\,\mathrm{d}z|. \tag{4.78}$$

In this construction, F is a parametrization of the external cell and $\widehat{x}$ is a surface coordinate and the operators $\mathcal{I}$ and $\mathcal{D}$ are expressed in the monomial basis of $H^+(S^1)$. For HSIE, ϵ and μ have to be constant on every cell, meaning that for a cell c of the surface triangulation it holds

$$\epsilon(F(\xi, \widehat{x})) = \epsilon_c \quad \forall \widehat{x} \in c, \xi \in [0, \infty). \tag{4.79}$$

There are two main ways to do this: star-shaped with a base-point, or axis-parallel. For the star-shaped concept, one chooses a point V_0 in the interior domain and chooses the infinite direction as the connecting vectors of V_0 with the vertices of the surface triangulation. For this choice, we set

$$F(\xi, \widehat{x}) = \widehat{x} + \xi(\widehat{x} - V_0) \tag{4.80}$$

and additionally factorize

$$\boldsymbol{J}(\xi, \widehat{x}) = \widehat{\boldsymbol{J}}(\widehat{x})\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1+\xi & 0 \\ 0 & 0 & 1+\xi \end{pmatrix} \tag{4.81}$$

If one chooses the infinite direction to be axis-parallel (in our notation the x coordinate), we define

$$F(\xi, \widehat{x}) = (\xi, \widehat{x})^T \tag{4.82}$$

and

$$\boldsymbol{J} = \widehat{\boldsymbol{J}} = \boldsymbol{Id}\,. \tag{4.83}$$

This concludes the required definitions for the introduction of Hardy space infinite elements. For the definition of the infinite element to discretize problems based on this transformation, see section 5.1.3. These elements are part of the code base, but there is a crucial problem that prevents them from being applied in the sweeping preconditioner, which is the core part of this work. For more details on the problems arising from their use in a sweeping preconditioner see section 5.3.3.

4.5 Modal Theory

Having introduced domain truncation methods and derived the partial differential equation most suited to modelling by finite elements, we turn to the derivation of input signals we wish to observe propagating through the geometries we model. For periodic structures with certain properties there are field profiles that propagate lossless under ideal conditions – these combinations of a geometry, a losslessly propagating field and a wavenumber κ are called modal configurations. The value of κ is usually derived from the physical application, in which a source of light dictates the wavelength of the signal propagating in the structure.

As the initial step for the computation of modes, we go back to eq. (4.9) and assume that the solution has the shape

$$\mathcal{E}(x,y,z) = \boldsymbol{E}(x,y)\, e^{\mathrm{i}\beta z} \quad \text{and} \tag{4.84}$$

$$\mathcal{H}(x,y,z) = \boldsymbol{H}(x,y)\, e^{\mathrm{i}\beta z} \tag{4.85}$$

for some $\beta > 0$. For the computation of modes, we demand the system to be periodic in the z direction. These are not the settings we will be attempting to solve later on in this dissertation, where we will be looking at freeform waveguides that are decidedly non-periodic in the z direction and where we use transformation optics to map the geometry onto a z-periodic geometry. At this point, however, we do not require transformation optics and the modes of a waveguide also impose no requirement that the materials in the system be anisotropic or anisomorphic. As a consequence, we will be assuming linear materials and simplifying the given equations to the Helmholtz system by means of the so-called Helmholtz decomposition. As a consequence, the property of guiding the mode perfectly only holds for an infinite waveguide with periodicity in the z direction and will no longer hold as soon as we change the geometry.

Using the common simplification $\kappa^2 := \omega^2 \mu_0 \epsilon_0 \epsilon_r$ in the second order system eq. (4.10) we find

$$\nabla \times \nabla \times \left(\boldsymbol{E}(x,y) e^{\mathrm{i}\beta z} \right) + \kappa^2 \boldsymbol{E}(x,y) e^{\mathrm{i}\beta z} = \boldsymbol{0} \quad \text{on } \mathbb{R}^3 \tag{4.86}$$

4.5.1 The Rectangular Waveguide

While there are analytical ways of computing modes for special geometries (such as cylindrical waveguides), we will focus on the more general techniques that also work for rectangular waveguide geometries. One important difference between some specialized approaches and the more general approach is the fact that for rectangular waveguide geometries, it cannot be assumed that $E_z = 0$.

Discretized Mixed Eigenvalue Problem

A way to compute the modes based on the finite element method is described in [Pom+07]. This method uses a mixed finite element method, in which the components orthogonal to the propagation direction and the component in propagation direction are dealt with separately. We first introduce the following notation:

$$\boldsymbol{\epsilon} = \begin{pmatrix} \boldsymbol{\epsilon}_\perp & \mathbf{0} \\ \mathbf{0}^T & \epsilon_z \end{pmatrix} \quad \text{and} \quad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_\perp & \mathbf{0} \\ \mathbf{0}^T & \mu_z \end{pmatrix}, \tag{4.87}$$

$$\boldsymbol{E}(x,y) = (\boldsymbol{E}_\perp(x,y), E_z(x,y))^T, \tag{4.88}$$

$$S = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \tag{4.89}$$

$$\nabla_\perp = (\partial_x, \partial_y)^T \tag{4.90}$$

and rewrite eq. (4.86) as

$$\begin{pmatrix} (S\nabla_\perp \mu_z^{-1} \nabla_\perp \cdot S) - \beta^2 S\boldsymbol{\mu}_\perp^{-1} S & -\mathrm{i}\beta S\boldsymbol{\mu}_\perp^{-1} S\nabla_\perp \\ -\mathrm{i}\beta\nabla_\perp \cdot S\boldsymbol{\mu}_\perp^{-1} S & \nabla_\perp \cdot S\boldsymbol{\mu}_\perp^{-1} S\nabla_\perp \end{pmatrix} \begin{pmatrix} \boldsymbol{E}_\perp \\ E_z \end{pmatrix} = \begin{pmatrix} \omega^2\boldsymbol{\epsilon}_\perp & \mathbf{0} \\ \mathbf{0}^T & \omega^2\epsilon_z \end{pmatrix} \begin{pmatrix} \boldsymbol{E}_\perp \\ E_z \end{pmatrix}. \tag{4.91}$$

Finally, one defines $\widetilde{E}_z = \beta E_z$ and

$$A = \begin{pmatrix} S\nabla_\perp \mu_z^{-1} \nabla_\perp \cdot S - \omega^2 \boldsymbol{\epsilon}_\perp & -\mathrm{i}S\boldsymbol{\mu}_\perp^{-1} S\nabla_\perp \\ \mathbf{0}^T & \nabla_\perp \cdot S\boldsymbol{\mu}_\perp^{-1} S\nabla_\perp - \omega^2\epsilon_z \end{pmatrix} \tag{4.92}$$

and

$$B = \begin{pmatrix} S\boldsymbol{\mu}^{-1} S & 0 \\ \mathrm{i}\nabla_\perp \cdot S\boldsymbol{\mu}_\perp^{-1} S & 0 \end{pmatrix} \tag{4.93}$$

and finds

$$A \begin{pmatrix} \boldsymbol{E}_\perp \\ \widetilde{E}_z \end{pmatrix} = \beta^2 B \begin{pmatrix} \boldsymbol{E}_\perp \\ \widetilde{E}_z \end{pmatrix} \tag{4.94}$$

for some $\beta > 0$. This eigenvalue problem for $(\boldsymbol{E}, \beta)$ is formulated on $\mathbb{R}^2$ but PML can be used to truncate the problem to a finite computational domain.

The truncated problem can then be discretized using 2D Nédélec elements for $\boldsymbol{E}_\perp$ and standard nodal elements for $\widetilde{E}_z$. This process is described in [Zsc+05], [Pom+07] and [Agh+08]. [Pom+07] discusses the convergence of this method for various element degrees and refinement strategies. The authors find that for elements of order 1 or 2 and depending on the mesh refinement strategy, there are situations in which the convergence of the imaginary part of the eigenvalues could not be guaranteed. For element of higher order, however, stable convergence was observed for all refinement strategies.

Since these computations are based on a 2D triangulation, the computational cost does not exceed the limitations of standard direct solvers.

Variational Effective Index Approximation

An alternative to the finite element method for mode computation is the **Variational Effective Index Approximation** described in [Iva+09]. In a first step, this method regards a slab waveguide with $\epsilon_s(x)$ and a polarized field with $E_x = 0$. Computing modes is generally introduced as a problem on $\mathbb{R}^3$ (or $\mathbb{R}^2$ and $\mathbb{R}$). Once numerical computation is required, these problems can be truncated by means of PML.

For the 2D setup, one computes the slab waveguide solution

$$\begin{pmatrix} E_x^s & E_y^s & E_z^s \\ H_x^s & H_y^s & H_z^s \end{pmatrix}(x,z) = \begin{pmatrix} 0 & \widetilde{E}_y(x) & 0 \\ \widetilde{H}_x(x) & 0 & \widetilde{H}_x(z) \end{pmatrix} e^{\mathrm{i}\beta z}. \tag{4.95}$$

With this assumption on the form of the slab waveguide solution, eq. (4.86) simplifies to

$$\frac{\partial^2 \widetilde{E}_y}{\partial x^2}(x) + \kappa^2 \epsilon_s(x)\widetilde{E}_y(x) = \beta^2 \widetilde{E}_y(x). \tag{4.96}$$

Next, we assume that the solution of the slab problem can be used to factorize the solution of the actual waveguide geometry as follows:

$$\begin{pmatrix} E_x & E_y & E_z \\ H_x & H_y & H_z \end{pmatrix}(x,y,z) = \begin{pmatrix} 0 & E_y^s(x)\widetilde{E}_y(y,z) & E_y^s(x)\widetilde{E}_z(y,z) \\ H_x^s(x)\widetilde{H}_x(y,z) & H_z^s(x)\widetilde{H}_y(y,z) & H_z^s(x)\widetilde{H}_z(y,z) \end{pmatrix} \tag{4.97}$$

Using the variational argument that solutions of eq. (4.9) are stationary points of the functional

$$f(\boldsymbol{E},\approx) = \int_{\mathbb{R}^3} \left(\boldsymbol{E}\cdot(\nabla\times\approx) + \approx\cdot(\nabla\times\boldsymbol{E}) - \mathrm{i}\omega\epsilon_0\epsilon\boldsymbol{E}^2 + \mathrm{i}\omega\mu_0\mu\approx^2 \right)\,\mathrm{d}V, \tag{4.98}$$

the authors in [Iva+09] derive the factors in eq. (4.97) to be

$$\begin{pmatrix} \widetilde{E}_x & \widetilde{E}_y & \widetilde{E}_z \\ \widetilde{H}_x & \widetilde{H}_y & \widetilde{H}_z \end{pmatrix}(y,z) = \frac{\mathrm{i}\beta}{\kappa^2\epsilon_{\mathrm{eff}}} \begin{pmatrix} 0 & \partial_z\widetilde{H}_x & -\partial_y\widetilde{H}_x \\ (-\mathrm{i}\kappa^2\epsilon_{\mathrm{eff}}\beta^{-1})\widetilde{H}_x & \partial_y\widetilde{H}_x & \partial_z\widetilde{H}_x \end{pmatrix}, \tag{4.99}$$

with

$$\epsilon_{\mathrm{eff}}(y,z) := \frac{\beta^2}{\kappa^2} + \frac{\int_{\mathbb{R}}(\epsilon(x,y,z)-\epsilon_s(x))(\widetilde{E}_y(x))^2\,\mathrm{d}x}{\int_{\mathbb{R}}(\widetilde{E}_y(x))^2\,\mathrm{d}x}. \tag{4.100}$$

Next, the domain is split into homogeneous, axis-parallel cuboids Ω^i, such that $\epsilon(x,y,z)|_{\Omega^i}$ is constant and finds the solution

$$\widetilde{H}_x(y,z) = c e^{-\mathrm{i}(k_y y + k_z z)} \tag{4.101}$$

for

$$k_y^2 + k_z^2 = \kappa^2\epsilon_{\mathrm{eff}} \tag{4.102}$$

and $k_y, k_z \geq 0$. For a waveguide that is parallel to the z axis we set $k_y = 0$.

In the special case of rectangular waveguides, $\epsilon(x,y,z)$ is constant in the z direction. We define $\rho^2 = \kappa^2\epsilon_{\mathrm{eff}}$ and divide eq. (4.102) by ρ^2. We find

$$\sqrt{\frac{k_y^2}{\rho^2} + \frac{k_z^2}{\rho^2}} = 1. \tag{4.103}$$

With the definition $\theta = \cos^{-1}\left(\frac{k_z}{\rho}\right)$ eq. (4.97) can be written as

$$\begin{pmatrix} \widetilde{E}_x & \widetilde{E}_y & \widetilde{E}_z \\ \widetilde{H}_x & \widetilde{H}_y & \widetilde{H}_z \end{pmatrix}(y,z) = \frac{c\beta}{\rho} e^{-\mathrm{i}\rho(-\sin\theta y + \cos\theta z)} \begin{pmatrix} 0 & \cos\theta & \sin\theta \\ \frac{\rho}{\beta} & -\sin\theta & \cos\theta \end{pmatrix}. \tag{4.104}$$

The variational effective index method thus first solves a 1D problem for the material distribution $\epsilon_s(x)$ to compute $\widetilde{E}_y(x)$ and β by solving eq. (4.96). This is done using the finite element method in 1D with linear basis functions as described in [ISH13, section 6.1]. The result is used to compute the effective material property $\epsilon_{\text{eff}}(y,z)$ via eq. (4.100), in which $\widetilde{E_y}(x)$ is decaying exponentially outside of the slab waveguide. and then uses it to construct a polarized solution of the 2D problem by eq. (4.104). While there are some strong assumptions that go into this method and [Iva+09] shows in their numerical results some situations in which the approximation is far from the precision of a 3D finite difference time domain (FDTD) computation, it does provide a very fast way to compute a mode profile. In our computations we have found that the computed mode profiles propagate nearly lossless with a signal retention of $> 95\%$ after several dozen wavelengths.

Remark. *This method is so fast, in fact, that a web browser can compute it in high resolution without noticeable run time. In our computations we generally use the fundamental mode of the waveguide, which is the eigenfunction for the largest eigenvalue, but this is not necessary.*

Experiment 1. ***Setup:*** *We use the EIMS solver [Ham22] based on the variational effective index approximation to compute the modes of a rectangular waveguide. The computation is performed on a 2D mesh of $[-3.6, 5.4] \times [-5, 5]\mu m$ with mesh size $h = 0.02\mu m$ and the waveguide at $[0, 1.8] \times [-1, 1]$ with $\epsilon_r = 1.36$ outside of the waveguide and $\epsilon_r = 1.53$ inside. The dimensions of the waveguide are $2\mu m$ width and $1.8\mu m$ height. We compute the first TE mode $TE_{0,0}$ as an input signal for our computations.*

Results: *The solver returns the propagation constant $\beta = 5.9560\mu m^{-1}$. Additionally, the retrieved mode profile is displayed in fig. 4.3 and fig. 4.4.*

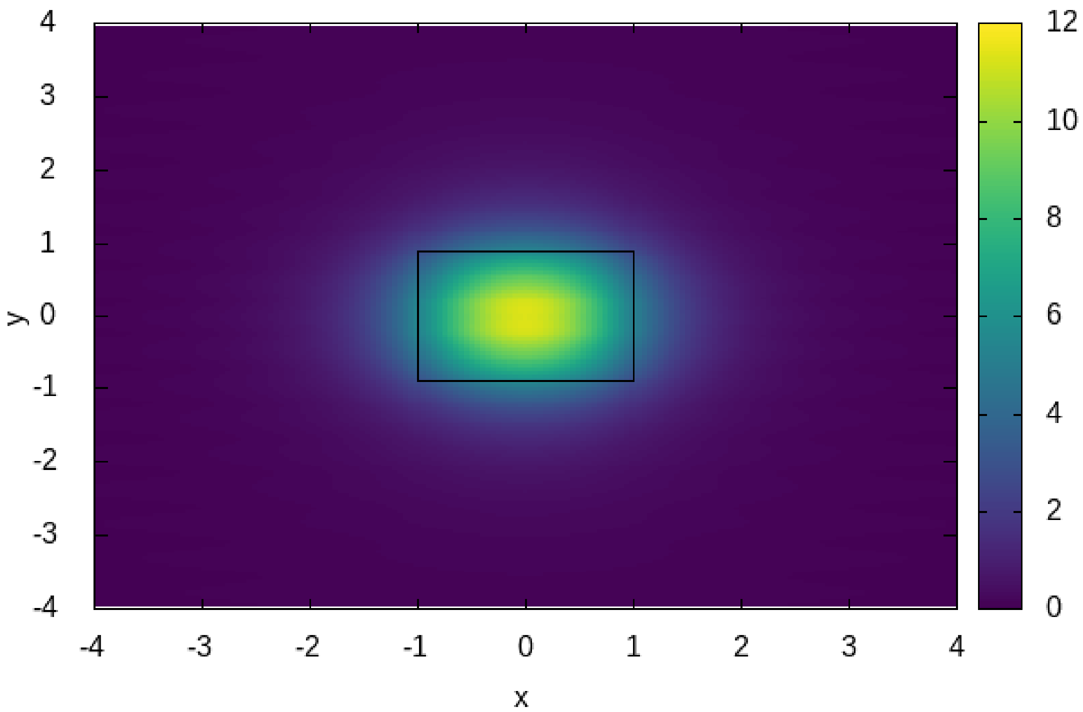


Figure 4.3: Real part of E_x as computed by the variational effective index approximation experiment 1.

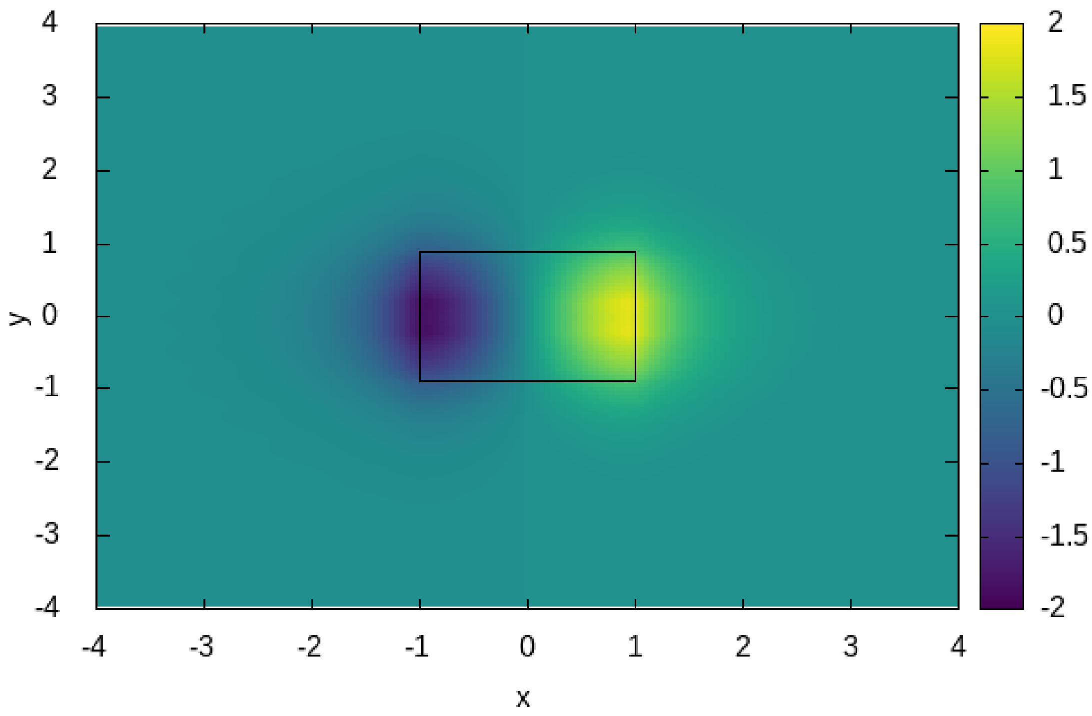


Figure 4.4: Imaginary part of E_z as computed by the variational effective index approximation experiment 1.

4.6 Signal Input

We have now provided an overview of what modes of a waveguide are. The next question we will be discussing are methods of introducing signals into a computational domain. The two methods we consider are Dirichlet boundary values on the tangential component on the surface and a method of computing a forcing term $\boldsymbol{F}$ that introduces the incident field in the interior of a computational domain instead of the surface.

4.6.1 Dirichlet Data

The common way of coupling a signal into the computational domain is to use similar steps as described above (see section 4.4.2) and impose Dirichlet boundary conditions on an input interface. This method, however, has one flaw: Let us consider the setup in fig. 4.5. In this setup, one imposes Dirichlet data on Γ_I and the domain of interest contains an object Ω_S, a scatterer, with PEC boundary conditions on the surface. This object will reflect and we can decompose the solution of this problem into two parts: If the incoming field is a propagating wave which is reasonably lossless, then a part of it will be reflected by the scatterer and reflected back onto the input interface. The field at the input interface is therefore the sum of the incident field $\boldsymbol{E}_I$ and the scattered field $\boldsymbol{E}_S$. The Dirichlet boundary condition, however, is applied to $\boldsymbol{E} = \boldsymbol{E}_I + \boldsymbol{E}_S$ and we find that $\boldsymbol{E}_I = \boldsymbol{E} - \boldsymbol{E}_S$. Therefore, the incident field does not fulfill the boundary condition if the scattered field is not equal to zero somewhere on the input interface.

As a consequence, this boundary condition should only be applied if backward reflections from the interior are negligible. It is preferable to derive a system, in which the scattered field is not subject to the input conditions. A way of achieving this will be proposed in the next section.

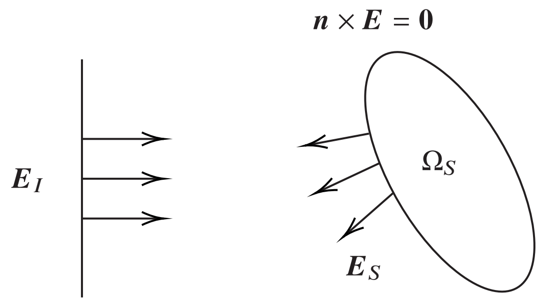


Figure 4.5: Schematic of a setup in which a scatterer Ω_S reflects an incoming wave $\boldsymbol{E}_I$ partially back onto the input interface.

4.6.2 Tapered Coupling

Ideally, we want backward reflections to be subject to an absorbing boundary condition. We therefore want a PML surface on the side of the input interface. This leaves us with the question of how to couple the input signal into the computational domain. Essentially, there are two paths to achieve this:

1. To generate a version of a PML that absorbs the scattered field well enough while also introducing the signal. This could be achieved by computing the damped signal on the backside of the PML and to set this signal instead of a PEC boundary condition. The field is damped exponentially, however, so this value would be multiple orders of magnitude smaller than the amplitude of the signal we are trying to generate.
2. To use a right-hand-side term to introduce the signal inside of Ω_C instead of a boundary condition.

Initially, we split the domain into three parts:

$$\Omega_1 = \mathbb{R} \times \mathbb{R} \times (-\infty, z_{\min}), \tag{4.105}$$
$$\Omega_T = \mathbb{R} \times \mathbb{R} \times z \in [z_{\min}, z_{\max}] \quad \text{and} \tag{4.106}$$
$$\Omega_0 = \mathbb{R} \times \mathbb{R} \times (z_{\min}, +\infty). \tag{4.107}$$

On these three domains, we investigate the equation

$$\nabla \times \mu^{-1} \nabla \times \boldsymbol{A} + \omega^2 \epsilon \boldsymbol{A} = \mathbf{0} \tag{4.108}$$
$$\nabla \cdot \epsilon \boldsymbol{A} = \mathbf{0} \tag{4.109}$$

and assume linear materials. We additionally introduce a field split by a function $\phi \in C^2(\mathbb{R}) : \mathbb{R} \to [0, 1]$ with the properties

$$\phi(z) = 1 \quad \text{for } z \leq z_{\min} \quad \text{and} \tag{4.110}$$
$$\phi(z) = 0 \quad \text{for } z \geq z_{\min}. \tag{4.111}$$

Next, we split the field $\boldsymbol{A}$ as

$$\boldsymbol{A} = \underbrace{\phi \boldsymbol{A}}_{-\boldsymbol{F}} + (1 - \phi)\boldsymbol{A}. \tag{4.112}$$

Our goal is to replace the part $\phi \boldsymbol{A}$ by a forcing term $\boldsymbol{F}$ and subtract it from eq. (4.108). On Ω_1 we will be using the PML to truncate the computational domain, on Ω_T we taper in the signal and Ω_0 will be our domain of interest.

On Ω_0 and Ω_1, both $e^{i\beta z}\boldsymbol{E}_0(x,y)$ – the signal we want to couple in – as well as $\mathbf{0}$ solve Maxwell's equations. The individual parts of the right side of eq. (4.112), however, do not. We compute

$$\boldsymbol{F}(x,y,z) = -\nabla\times\mu^{-1}\nabla\times\left(\phi(z)e^{i\beta z}\boldsymbol{E}_0(x,y)\right) + \omega^2\epsilon\phi(z)e^{i\beta z}\boldsymbol{E}_0(x,y). \tag{4.113}$$

Next, we define

$$\phi(z) = 2\left(\frac{z-z_{\text{min}}}{z_{\text{max}}-z_{\text{min}}}\right)^3 - 3\left(\frac{z-z_{\text{min}}}{z_{\text{max}}-z_{\text{min}}}\right)^2 + 1, \tag{4.114}$$

which fulfills eq. (4.110).

With this definition of $\phi(z)$ and the fact, that $\boldsymbol{E}_0$ does not depend on z, we define

$$\boldsymbol{E}_I(x,y,z) = \phi(z)e^{i\beta z}\begin{pmatrix} f(x,y) \\ g(x,y) \\ h(x,y)\end{pmatrix}. \tag{4.115}$$

Next we can evaluate the curl-terms:

$$\begin{aligned}
\frac{(\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_x}{e^{\mathrm{i}\beta z}} &= \left(\phi(z)\left(\frac{\partial^2 g}{\partial x\partial y}(x,y) - \frac{\partial^2 f}{\partial y^2}(x,y) + \beta^2 f(x,y) + \mathrm{i}\beta\frac{\partial h}{\partial x}(x,y)\right)\right) \\
&\quad + f(x,y)\left(-\frac{\partial^2\phi}{\partial z^2}(z) - 2\mathrm{i}\beta\frac{\partial\phi}{\partial z}(z)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial h}{\partial x}(x,y),
\end{aligned} \tag{4.116}$$

$$\begin{aligned}
\frac{(\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_y}{e^{\mathrm{i}\beta z}} &= \phi(z)\left(\frac{\partial^2 f}{\partial x\partial y}(x,y) - \frac{\partial^2 g}{\partial x^2}(x,y) + \beta^2 g(x,y) + \mathrm{i}\beta\frac{\partial h}{\partial y}(x,y)\right) \\
&\quad + g(x,y)\left(-\frac{\partial^2\phi}{\partial z^2}(z) - 2\mathrm{i}\beta\frac{\partial\phi}{\partial z}(z)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial h}{\partial y}(x,y),
\end{aligned} \tag{4.117}$$

$$\begin{aligned}
\frac{(\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_z}{e^{\mathrm{i}\beta z}} &= \phi(z)\left(\mathrm{i}\beta\frac{\partial f}{\partial x}(x,y) + \mathrm{i}\beta\frac{\partial g}{\partial y}(x,y) - \frac{\partial^2 h}{\partial x^2}(x,y) - \frac{\partial^2 h}{\partial y^2}(x,y)\right) \\
&\quad + \frac{\partial\phi}{\partial z}(z)\left(\frac{\partial f}{\partial x}(x,y) + \frac{\partial g}{\partial y}(x,y)\right).
\end{aligned} \tag{4.118}$$

We can now use the property, that $g(x,y) = 0$ for modes computed by the variational effective index approximation (see section 4.5.1) and thus

$$\begin{aligned}
\frac{(\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_x}{e^{\mathrm{i}\beta z}} &= \phi(z)\left(-\frac{\partial^2 f}{\partial y^2}(x,y) + \beta^2 f(x,y) + \mathrm{i}\beta\frac{\partial h}{\partial x}(x,y)\right) \\
&\quad - f(x,y)\left(\frac{\partial^2\phi}{\partial z^2}(z) + 2\mathrm{i}\beta\frac{\partial\phi}{\partial z}(z)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial h}{\partial x}(x,y), \\
\frac{(\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_y}{e^{\mathrm{i}\beta z}} &= \phi(z)\left(\frac{\partial^2 f}{\partial x\partial y}(x,y) + \mathrm{i}\beta\frac{\partial h}{\partial y}(x,y)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial h}{\partial y}(x,y), \\
\frac{(\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_z}{e^{\mathrm{i}\beta z}} &= \phi(z)\left(\mathrm{i}\beta\frac{\partial f}{\partial x}(x,y) - \frac{\partial^2 h}{\partial x^2}(x,y) - \frac{\partial^2 h}{\partial y^2}(x,y)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial f}{\partial x}(x,y).
\end{aligned} \tag{4.119}$$

We can now compute the derivatives of h and f numerically. Additionally we can use

$$\phi'(z) = \frac{6(z-z_{\text{min}})(z-z_{\text{max}})}{(z_{\text{max}}-z_{\text{min}})^3} \quad\text{and}\quad \phi''(z) = \frac{12(z-z_{\text{min}})}{(z_{\text{max}}-z_{\text{min}})^3} - \frac{6}{(z_{\text{max}}-z_{\text{min}})^2} \tag{4.120}$$

to compute $\boldsymbol{F}$. We find

$$\boldsymbol{F}(x,y,z) = \begin{pmatrix} F_x(x,y,z) \\ F_y(x,y,z) \\ F_z(x,y,z) \end{pmatrix} = - \begin{pmatrix} (\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_x - \omega^2\epsilon\phi(z)\mathrm{e}^{\mathrm{i}\beta z} f(x,y) \\ (\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_y \\ (\nabla\times\nabla\times\boldsymbol{E}_I(x,y,z))_z - \omega^2\epsilon\phi(z)\mathrm{e}^{\mathrm{i}\beta z} h(x,y) \end{pmatrix}, \tag{4.121}$$

$$\begin{aligned} F_x(x,y,z) = -\mu^{-1} e^{\mathrm{i}\beta z}\Bigg[\phi(z)\left(-\frac{\partial^2 f}{\partial y^2}(x,y) + \beta^2 f(x,y) + \mathrm{i}\beta\frac{\partial h}{\partial x}(x,y)\right) \\ -f(x,y)\left(\frac{\partial^2\phi}{\partial z^2}(z) + 2\mathrm{i}\beta\frac{\partial\phi}{\partial z}(z)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial h}{\partial x}(x,y)\Bigg] + \omega^2\epsilon e^{\mathrm{i}\beta z} f(x,y), \end{aligned} \tag{4.122}$$

$$F_y(x,y,z) = -\mu^{-1} e^{\mathrm{i}\beta z}\left[\phi(z)\left(\frac{\partial^2 f}{\partial x\partial y}(x,y) + \mathrm{i}\beta\frac{\partial h}{\partial y}(x,y)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial h}{\partial y}(x,y)\right], \tag{4.123}$$

$$F_z(x,y,z) = -\mu^{-1} e^{\mathrm{i}\beta z}\left[\phi(z)\left(\mathrm{i}\beta\frac{\partial f}{\partial x}(x,y) - \frac{\partial^2 h}{\partial x^2}(x,y) - \frac{\partial^2 h}{\partial y^2}(x,y)\right) + \frac{\partial\phi}{\partial z}(z)\frac{\partial f}{\partial x}(x,y) - \omega^2\epsilon\mu h(x,y)\right]. \tag{4.124}$$

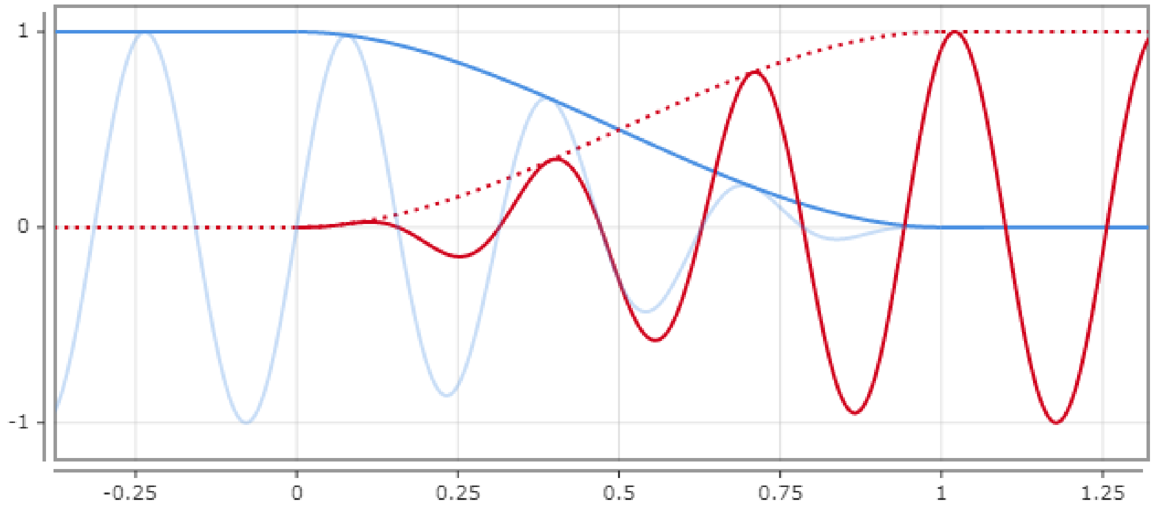


Figure 4.6: $\phi(z)$ (blue) and $(1-\phi(z))\,\mathrm{Re}(E_{I,x})$ (red) for $z_{\min} = 0$ and $z_{\max} = 1$. The red dotted line shows $(1-\phi(z))$ and the light blue line represents $\phi(z)\mathrm{Re}(E_{I,x})$.

4.7 The Weak Formulation

At this point, we have introduced the second order PDE, the derivation of an input signal by various means and boundary conditions. The problems we will be considering in this work in general are of the shape

$$\begin{aligned} \nabla\times\left(\boldsymbol{\mu}^{-1}\nabla\times\boldsymbol{E}\right) - \boldsymbol{\epsilon}\omega^2\boldsymbol{E} = \boldsymbol{F} \qquad & \text{in } \Omega_C, \\ \nabla\cdot(\boldsymbol{\epsilon}\boldsymbol{E}) = 0 \qquad & \text{in } \Omega_C, \\ \boldsymbol{E}\times\boldsymbol{n}_C = \boldsymbol{0} \qquad & \text{on } \partial\Omega_C, \end{aligned} \tag{4.125}$$

if we use tapered coupling, or

$$\begin{aligned} \nabla \times \left(\boldsymbol{\mu}^{-1}\nabla \times \boldsymbol{E}\right) - \boldsymbol{\epsilon}\omega^2\boldsymbol{E} &= \boldsymbol{0} && \text{in } \Omega_C, \\ \nabla \cdot (\boldsymbol{\epsilon}\boldsymbol{E}) &= 0 && \text{in } \Omega_C, \\ \boldsymbol{E} \times \boldsymbol{n}_I - \boldsymbol{E}_I \times \boldsymbol{n}_I &= \boldsymbol{0} && \text{on } \Gamma_I, \\ \boldsymbol{E} \times \boldsymbol{n}_I &= \boldsymbol{0} && \text{on } \partial\Omega_C \setminus \Gamma_I, \end{aligned} \tag{4.126}$$

if we use Dirichlet data on the input interface Γ_I, where $\boldsymbol{n}_I$ is the normal vector of the surface Γ_I and $\boldsymbol{E}_I$ is the input field. In both cases, the domains Ω_C, i.e. the computational domains, include a PML medium on all sides for eq. (4.125) and all directions except the input interface for eq. (4.126).

As remarked in section 4.6.1, using Dirichlet data to couple the signal into the system has the downside of interfering with backward reflections, since the condition on the boundary effectively prescribes

$$(\widetilde{\boldsymbol{E}_I} + \boldsymbol{E}_S) \times \boldsymbol{n}_I = \boldsymbol{E}_I \times \boldsymbol{n}_I. \tag{4.127}$$

where we have split $\boldsymbol{E}$ into the effective input signal $\widetilde{\boldsymbol{E}_I}$ and a scattered field $\boldsymbol{E}_S$. The effective input signal is only similar to the boundary constraints if $\boldsymbol{E}_S \ll \widetilde{\boldsymbol{E}_I}$. If we compute a straight waveguide or a Hertzian dipole in free space, where we can reasonably assume that no backward reflections exist, then this boundary condition can be used. It has the advantage of requiring no PML on the input interface and is therefore numerically cheaper. Once backward reflections become considerable, however, we should switch to tapered signal coupling.

For these two problems we can derive weak formulations by the Galerkin method. We multiply the first equations in eq. (4.125) and eq. (4.126) by the complex conjugate of a smooth test function $\overline{\boldsymbol{\phi}}$ and find for eq. (4.125)

$$\int_{\Omega_C} (\boldsymbol{\mu}^{-1}\nabla\times\boldsymbol{E})\cdot\nabla\times\overline{\boldsymbol{\phi}} - \boldsymbol{\epsilon}\omega^2\boldsymbol{E}\cdot\overline{\boldsymbol{\phi}}\,\mathrm{d}V + \int_{\partial\Omega}\left[\boldsymbol{n}_C\times\boldsymbol{\mu}^{-1}(\nabla\times\boldsymbol{E})\right]\cdot\left[(\boldsymbol{n}_C\times\overline{\boldsymbol{\phi}})\times\boldsymbol{n}_C\right]\,\mathrm{d}A = \int_{\Omega_C}\boldsymbol{F}\cdot\overline{\boldsymbol{\phi}}\,\mathrm{d}V. \tag{4.128}$$

If we choose our test functions $\boldsymbol{\phi}$ from the space

$$H^0(\mathrm{curl}, \Omega_C) = \{\boldsymbol{u} \in H(\mathrm{curl}, \Omega_C) : \boldsymbol{n}_C \times \boldsymbol{u} = \boldsymbol{0} \quad \text{on} \quad \partial\Omega_C\}, \tag{4.129}$$

then we can use the additional identity

$$\int_{\partial\Omega}\left[\boldsymbol{n}_C\times\boldsymbol{\mu}^{-1}(\nabla\times\boldsymbol{E})\right]\cdot\underbrace{\left[(\boldsymbol{n}_C\times\overline{\boldsymbol{\phi}})\times\boldsymbol{n}_C\right]}_{=\boldsymbol{0}}\,\mathrm{d}A = 0. \tag{4.130}$$

With eq. (2.10) and eq. (4.130) we write eq. (4.128) as

$$\langle\boldsymbol{\mu}^{-1}\nabla\times\boldsymbol{E}, \nabla\times\boldsymbol{\phi}\rangle - \langle\boldsymbol{\epsilon}\omega^2\boldsymbol{E}, \boldsymbol{\phi}\rangle = \langle\boldsymbol{F}, \boldsymbol{\phi}\rangle. \tag{4.131}$$

We define the sesquilinear form $a_T : H^0(\mathrm{curl}, \Omega_C) \times H^0(\mathrm{curl}, \Omega_C) \to \mathbb{C}$ by

$$a_T(\boldsymbol{u}, \boldsymbol{v}) = \langle\mu^{-1}\nabla\times\boldsymbol{u}, \nabla\times\boldsymbol{v}\rangle - \langle\boldsymbol{\epsilon}\omega^2\boldsymbol{u}, \boldsymbol{v}\rangle \tag{4.132}$$

and for the right hand side of the equation we define $r : H^0(\mathrm{curl}, \Omega_C) \to \mathbb{C}$ by

$$r_T(\boldsymbol{u}) = \int_{\Omega_C} \boldsymbol{F}\cdot\overline{\boldsymbol{u}}\,\mathrm{d}V. \tag{4.133}$$

For the case of Dirichlet data, we use the space

$$H_D^0(\mathrm{curl},\Omega_C) = \left\{\boldsymbol{u} \in H(\mathrm{curl},\Omega_C) : \boldsymbol{n}_C \times \boldsymbol{u} = \boldsymbol{0} \text{ on } \Omega_C \setminus \Gamma_I \text{ and } \boldsymbol{u}_\perp \in L^2(\Gamma_I)^3\right\}, \tag{4.134}$$

find the equation

$$\int_{\Omega_C} (\boldsymbol{\mu}^{-1}\nabla \times \boldsymbol{E}) \cdot \nabla \times \overline{\boldsymbol{\phi}} - \boldsymbol{\epsilon}\omega^2 \boldsymbol{E} \cdot \overline{\boldsymbol{\phi}} \,\mathrm{d}V = \boldsymbol{0} \tag{4.135}$$

and define the sesquilinear form $a_D : H_D^0(\mathrm{curl},\Omega_C) \times H_D^0(\mathrm{curl},\Omega_C) \to \mathbb{C}$ defined by

$$a_D(\boldsymbol{u},\boldsymbol{v}) = \langle \mu^{-1}\nabla \times \boldsymbol{u}, \nabla \times \boldsymbol{v}\rangle - \langle \boldsymbol{\epsilon}\omega^2 \boldsymbol{u}, \boldsymbol{v}\rangle. \tag{4.136}$$

In the following sections, we will only focus on the first equations in eqs. (4.125) and (4.126). To illustrate the reasoning behind this, consider $\phi \in H^1(\Omega_C)$. It is clear that $\nabla\phi \in H(\mathrm{curl},\Omega_C)$, since

$$\nabla \times \nabla\phi = \boldsymbol{0} \in (L^2(\Omega_C))^3. \tag{4.137}$$

We then choose $\boldsymbol{\psi} = \nabla\phi$, and insert this into eq. (4.135) and find

$$\int_{\Omega_C} (\boldsymbol{\mu}^{-1}\nabla \times \boldsymbol{E}) \cdot \underbrace{\nabla \times \overline{\boldsymbol{\psi}}}_{=\boldsymbol{0}} = \int_{\Omega_C} \boldsymbol{\epsilon}\omega^2 \boldsymbol{E} \cdot \overline{\boldsymbol{\psi}}. \tag{4.138}$$

Therefore, if $\omega \neq 0$ we find

$$\boldsymbol{0} = \int_{\Omega_C} \boldsymbol{\epsilon}\boldsymbol{E} \cdot \overline{\boldsymbol{\psi}} = \int_{\Omega_C} \boldsymbol{\epsilon}\boldsymbol{E} \cdot \nabla\overline{\phi} = -\int_{\Omega_C} \nabla \cdot (\boldsymbol{\epsilon}\boldsymbol{E})\phi \,\mathrm{d}V + \int_{\partial\Omega_C} \boldsymbol{n}_C \cdot \boldsymbol{\epsilon}\boldsymbol{E}\phi \,\mathrm{d}A. \tag{4.139}$$

Due to the Dirichlet constraints on the boundary of the computational domain, we know that the variation should be zero on the boundary and choose $\phi|_{\partial\Omega_C} = 0$. Therefore, the weak formulation of the divergence condition holds.

Problem 1 can now be written as

$$\text{Find } \boldsymbol{u} \in H^0(\mathrm{curl},\Omega_C) : a(\boldsymbol{u},\boldsymbol{v}) = r(\boldsymbol{v}) \quad \text{for all } \boldsymbol{v} \in H^0(\mathrm{curl},\Omega_C) \tag{4.140}$$

and problem 2 as

$$\begin{aligned} \text{Find } \boldsymbol{u} \in H_D^0(\mathrm{curl},\Omega_C) : a(\boldsymbol{u},\boldsymbol{v}) = \boldsymbol{0} \quad &\text{for all } \boldsymbol{v} \in H_D^0(\mathrm{curl},\Omega_C) \\ &\text{such that } \boldsymbol{u} \times \boldsymbol{n}_{\Gamma_{\mathrm{in}}} = \boldsymbol{E}_0 \times \boldsymbol{n}_{\Gamma_{\mathrm{in}}} \text{ on } \Gamma_{\mathrm{in}}. \end{aligned} \tag{4.141}$$

Remark. *In the numerical computations, the additional condition in problem 2 will be enforced by projecting the solution onto Γ_I and setting the values of the degrees of freedom of edges and faces on Γ_I to the resulting values. It is a property of Nédélec-elements that these degrees of freedom form the tangential space. We can then condense the resulting constraints into the set of linear equations. This way, we will not compute the values of these boundary degrees of freedom but instead set their value and move the terms to the right hand side.*

4.7.1 Existence and Uniqueness of Solutions

First, let us note that the sesquilinear forms a_D and a_T are not coercive due to the term $-\langle \boldsymbol{\epsilon}\omega^2 \boldsymbol{u}, \boldsymbol{v}\rangle$. Therefore, the conditions of the Cea-Lemma and Lax-Milgram theorem are not fulfilled. Since we will be using PML for the truncation of the computational domain, we know that there exists a ball B of non-zero radius such that $\mathrm{Im}(\boldsymbol{\epsilon})$ is strictly positive on B. By theorem 4.17 in [Mon92] we find existence and uniqueness of the solution in the respective spaces $H^0(\mathrm{curl},\Omega_C)$ and $H_D^0(\mathrm{curl},\Omega_C)$. The proof is based on first establishing uniqueness on the ball B and then using a unique continuation theorem to extend the solution to Ω_C.

4.8 Implementation

The boundary conditions mentioned in this chapter are all implemented in the code base in classes derived from the class BoundaryCondition. The PEC boundary condition is named EmptySurface, Dirichlet surfaces are implemented in DirichletSurface objects, PML surfaces as PMLSurface objects and Hardy space infinite elements are provided in the HSIESurface class. All these implementations can be found in the folder /Code/BoundaryCondition.

The code currently holds no implementation of the computation of modes. Originally, there was also code to model cylindrical waveguides for which the modes can be computed analytically. This part of the code was removed, however, since the specific meshes required to model cylindrical waveguides had disadvantages in the implementation of the hierarchical sweeping preconditioner. For the rectangular waveguide case, the modes must be computed numerically as discussed above. Tools to compute and export these modes exist and are even available online, such as `https://www.computational-photonics.eu/eims.html`. The code, therefore, has the functionality of importing mode profiles from input files. One such mode is present for the default configuration of a $2\mu m$ by $1.8\mu m$ rectangular waveguide with the material properties listed in chapter 2 and the standard operating wavelength of $1550nm$ that is frequently used in photonics.

As an alternative to waveguide computations, there is also an implementation of a Hertzian dipole in PointSourceField that can be used to simulate a Hertzian dipole in free space.

To prescribe these boundary values, there are two options: Dirichlet data (using DirichletSurface see section 4.6.1) or tapered coupling (see section 4.6.2), which can be activated using a parameter in the case file, and which introduces an additional PML domain for the backward reflected field.

5 Numerical Methods

In the previous chapter we presented how time-harmonic Maxwell's equations arise from physics and have dealt with basic questions like spatial truncation to enable us to restrict the domain on which we seek a solution to said equations to a smaller subdomain. We also introduced basic modal theory for waveguides, which will provide the boundary values we use to complete the setup of a scattering problem. In this chapter, we will focus on the numerical aspects of solving the equations we formulated using finite elements and introduce the core concepts. We will begin by introducing finite elements and move on to Nédélec-elements, which are the type of finite element we will be using. These elements enable us to transform our partial differential equation into a set of linear equations we can express as a system $Ax = b$. We will see that this system is numerically difficult to solve and introduce the concept of preconditioning and more specifically sweeping preconditioners, which will serve as the basis for the hierarchical sweeping preconditioner. The hierarchical sweeping preconditioner is a method of extending the range of applicability of sweeping preconditioners and the central innovation presented in this work.

5.1 Finite Elements

5.1.1 General Introduction

Finite element methods require a triangulation of the computational domain. In this work we will generally be using hexahedral, axis-parallel meshes as described in [Mon92, chapter 6.1] and [Zag06].

While this is a special case and more elaborate meshes can be interesting in some applications, these restrictions have massive advantages concerning implementation and the use of transformation optics to map a physical geometry onto a more suitable computational domain will enable us to still solve the partial differential equation on a wide variety of geometries. As we will see in chapter 6, this will not be the only advantage of using transformation optics instead of mesh adaptation.

A triangulation consists of two sets: a set of vertices and a set of edges. The set of edges lists pairs of vertices that are connected by an edge. In this work, we focus on axis parallel, hexahedral meshes, which are meshes in 3D space in which each vertex is connected to its neighbors via axis-parallel edges.

A finite element consists of three parts:

1. A reference cell. Since we work on 3D meshes, our reference cell is $K = [0, 1]^3$.
2. A function space P_K of finite dimension on the reference cell with a set of basis elements.
3. A set of degrees of freedom M. These are linear functionals on P_K that are unisolvent, meaning that a set of values of the degrees of freedom uniquely identifies an element in P_K.

The process of solving a partial differential equation using that finite element then consists of the steps of first discretizing the computational domain of the problem by images of the reference cell, evaluating the weak formulation for the basis function on a quadrature of the reference cell to assemble a linear set of equations and finally, solving that system to compute the values of the degrees of freedom, which, in turn, uniquely identify the approximate solution in P_K.

For an introduction to the general theory of the method of finite elements, we refer to [Mon92] or [BW76].

5.1.2 Nédélec's curl-conforming elements

All our computations are performed on hexahedra and we therfore introduce the Nédélec element on hexahedra. We define

Definition 5.1.1 (Tensor Product Polynomial Space)**.** *Let*

$$Q_{i,j,k} := \{\textit{polynomials of maximum degree } i \textit{ in } x_1, j \textit{ in } x_2, \textit{ and } k \textit{ in } x_3\}$$

from [Mon92, p. 109].

In the following definition, $\widetilde{\cdot}$ denotes objects on the original triangulation while objects without tilde are defined on the reference element.

Definition 5.1.2 (Nédélec's curl-conforming element)**.** *For an integer $k \geq 1$, Nédélec's curl-conforming element of order k is defined as the tuple $(K, P_K, M(u))$ with the reference element*

$$K := [0,1]^3,$$

and the polynomial space

$$P_K := Q_{k-1,k,k} \times Q_{k,k-1,k} \times Q_{k,k,k-1}.$$

The set of degrees of freedom consists of three types. For $\boldsymbol{u} \in (H^{1/2+\delta}(\widetilde{K}))^3$ with $\delta > 0$ such that $\nabla \times \boldsymbol{u} \in (L^p(\widetilde{K}))^3$ where $\widetilde{K}$ is a cell in the triangulation and $p > 2$ these are defined as follows:

1. *The edge dofs for the 12 edges e with tangential unit vector $\boldsymbol{t}$ are defined as*

$$M_e(\boldsymbol{u}) = \left\{ \int_e \boldsymbol{u} \cdot \boldsymbol{t} q \,\mathrm{d}s \quad \textit{for every } q \in P_{k-1}(e) \right\} \tag{5.1}$$

2. *The face dofs for the 6 faces f with normal vector $\boldsymbol{n}$ are defined as*

$$M_f(\boldsymbol{u}) = \left\{ \int_f \boldsymbol{u} \times \boldsymbol{n} \cdot \boldsymbol{q} \,\mathrm{d}A \quad \textit{for each } \boldsymbol{q} \in Q_{k-2,k-1}(f) \times Q_{k-1,k-2}(f) \right\}. \tag{5.2}$$

3. *The cell dofs are defined as*

$$M_K(u) = \left\{ \int_K \boldsymbol{u} \cdot \boldsymbol{q} \,\mathrm{d}V \quad \textit{for all } \boldsymbol{q} \in Q_{k-1,k-2,k-2} \times Q_{k-2,k-1,k-2} \times Q_{k-2,k-2,k-1} \right\}. \tag{5.3}$$

Theorem 5.1.1. *The element described in definition 5.1.2 is* ***curl conforming*** *and* ***unisolvent*** *and it holds*

$$V_h = \{\widetilde{\boldsymbol{u}}_h \in H(\mathrm{curl}, \Omega) : \widetilde{\boldsymbol{u}}_h|_{\widetilde{K}} \in Q_{k-1,k,k} \times Q_{k,k-1,k} \times Q_{k,k,k-1} \textit{ for all cells } \widetilde{K} \textit{ in the triangulation}\}.$$

Proof. (See [Mon92, Theorem 6.5]). □

The term curl-conforming states that $V_h \subset H(\mathrm{curl}, \Omega)$ and unisolvency is the linear independence of the individual degrees of freedom. In the element of lowest order, there are no face or cell dofs, only edge dofs. These are most commonly used since they generate the lowest bandwidth for the system matrix.

5.1.3 Hardy Space Infinite Elements

For a more detailed introduction we refer to [Nan+11]. In general, all the degrees of freedom are either a combination of the surface Nédélec element continued by the operator $\mathcal{T}_+$ from eq. (4.59) or the surface nodal element continued by $\mathcal{T}_{-}$. For Hardy-space infinite elements of a given degree $N > 0$ this yields the following types of degrees of freedom:

1. **Edge functions**: for a every edge e and every basis function on that edge $\boldsymbol{v}_i$ we define the function
$$\boldsymbol{V_i^e} = \begin{pmatrix} 0 \\ \Psi_{-1} \otimes \boldsymbol{v}_i^e \end{pmatrix} \tag{5.4}$$
2. **Surface functions**: similarly for every surface s and every basis function of that surface v_i^s we define the basis function
$$\boldsymbol{V_i^s} = \begin{pmatrix} 0 \\ \Psi_{-1} \otimes \boldsymbol{v}_i^s \end{pmatrix} \tag{5.5}$$
3. **Ray functions**: For every edge e which originates in the node n the ray functions
$$\boldsymbol{V_k^r} = \begin{pmatrix} \phi_k \otimes w_n^i \\ \mathbf{0} \end{pmatrix} \quad \text{for } k = -1, \ldots, N. \tag{5.6}$$
4. **Tangential infinite face functions**: for every edge e and every basis function on that edge of the surface Nédélec element $\boldsymbol{v}_i$
$$\boldsymbol{V_k^{fa}} = \begin{pmatrix} 0 \\ \Psi_k \otimes \boldsymbol{v}_i^e \end{pmatrix} \quad \text{for } k = 0, \ldots, N. \tag{5.7}$$
5. **Orthogonal infinite face functions**: for every edge e and every basis function of the H^1 hexahedral volume element on that edge $\boldsymbol{h}_i$
$$\boldsymbol{V_k^{fb}} = \begin{pmatrix} \psi_k \otimes \boldsymbol{h}_i \\ \mathbf{0} \end{pmatrix} \quad \text{for } k = -1, \ldots, N. \tag{5.8}$$
6. **Tangential infinite cell functions**: for every surface s and every basis function v_i^s of that surface we define the basis function $\boldsymbol{v}_i$
$$\boldsymbol{V_k^{ca}} = \begin{pmatrix} 0 \\ \Psi_k \otimes \boldsymbol{v}_i^s \end{pmatrix} \quad \text{for } k = 0, \ldots, N. \tag{5.9}$$
7. **Orthogonal infinite cell functions**: for every basis function of the f_i of the H^1 hexahedral volume element on the surface s the basis functions:
$$\boldsymbol{V_k^{cb}} = \begin{pmatrix} \psi_k \otimes \boldsymbol{f}_i \\ \mathbf{0} \end{pmatrix} \quad \text{for } k = -1, \ldots, N. \tag{5.10}$$

In this notation, the first components of the basis function denote the outward or infinite direction and the second component is the two-dimensional surface coordinate. We use the definitions

$$\phi_{-1} = \mathcal{T}_{+}(1,0) \tag{5.11}$$

$$\phi_j = \mathcal{T}_{+}(0,(\cdot)^j) \quad \text{for } j = 0,\dots,N \tag{5.12}$$

$$\boldsymbol{\Psi}_{-1} = \frac{1}{\mathrm{i}\kappa_0}\mathcal{T}_{-}(1,0) \quad \text{and} \tag{5.13}$$

$$\Psi_j = \frac{1}{i\kappa_0}\mathcal{T}_{-}(0,(\cdot)^j) \quad \text{for } j = 0,\dots,N. \tag{5.14}$$

The operators $\mathcal{T}_{\pm}$ have been introduced in eq. (4.59) and κ_0 is the parameter used to construct the Möbius transformation at the core of the HSIE method. The space $\boldsymbol{V}_h$ can then be defined as the set of all functions of these types for all edges, faces, cells and surface degrees of freedom of the surface triangulation.

Remark. *The HSIE dofs can also be understood as a transformed Nédélec element because every type of dof corresponds to a dof of the Nédélec element. The only difference is, that we differentiate between those dofs that point into one of the two surface-tangential directions and the dofs that point in the outward direction. By this representation, edge functions are the edge functions on the surface, the surface functions are the the surface functions, the ray functions are the edge functions pointing outward. The two types of infinite face functions are the face functions either pointing outward or surface parallel and the two types of infinite cell functions are the surface parallel and orthogonal cell functions.*

In [Nan+11], the authors show that this infinite element is a curl-conforming element. We introduce these elements because of two properties:

- They have the advantage of modeling the outward propagating solution by an oscillating function. This is more appropriate than the PML since the solution is oscillating.
- HSIE polynomials of degree 5 and higher are a common default value. While this seems much, it does not impact the size of the system very much since there is only one layer of infinite elements. There are, however, substantially more degrees of freedom per cell which leads to a higher bandwidth of the associated blocks in the stiffness matrix.

As we will see later on, sweeping preconditioners depend on an internal application of an absorbing boundary condition to split the computational domain into parts. Since we consider sweeping preconditioners at their numerical limits in this work, a cheap but effective boundary condition could drastically improve the performance of a sweeping technique. We will discuss where this leads to in section 5.6.

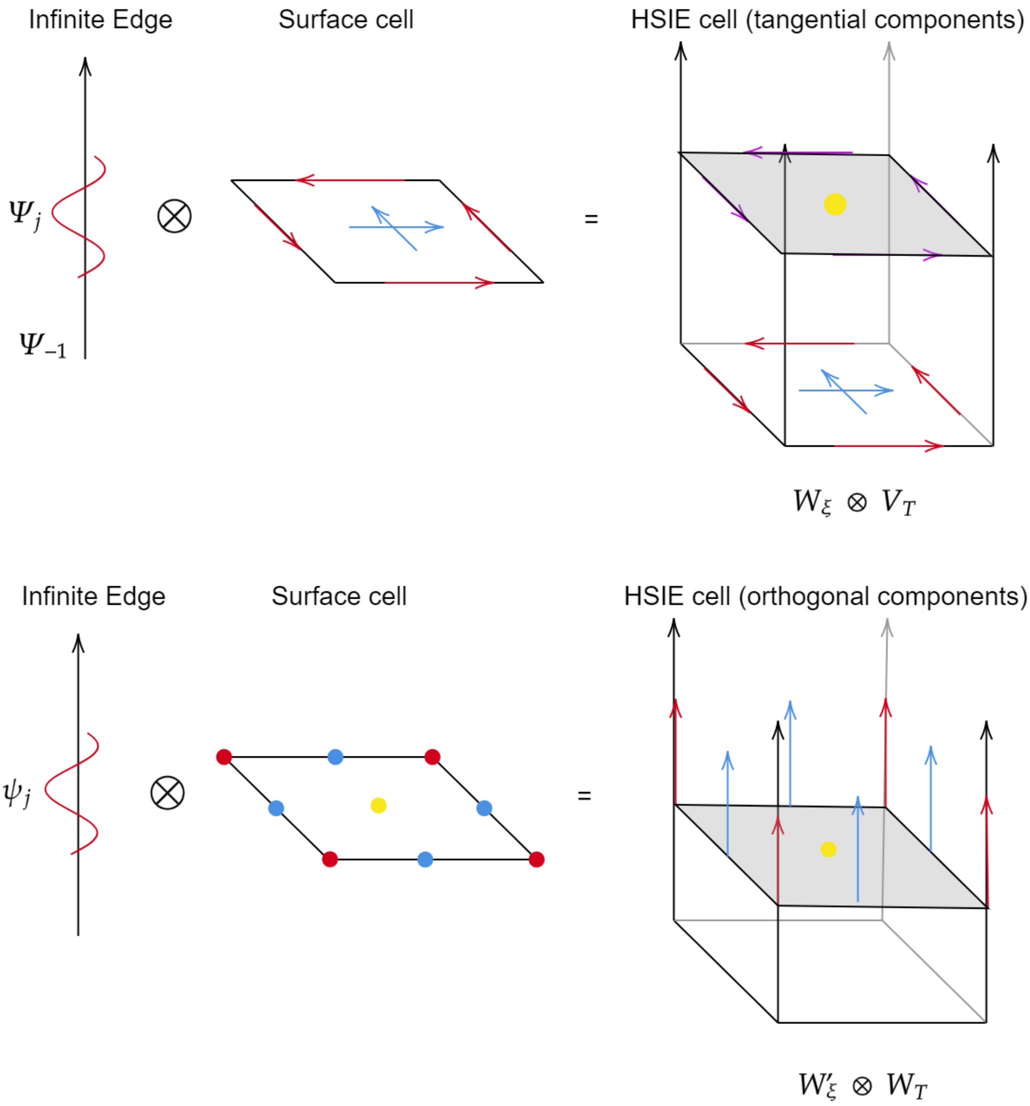


Figure 5.1: Sketch of the combination of surface dofs with Hardy space polynomials to create the basis functions. In the top row on the right: red represents the edge functions, blue the surface functions, pink the tangential infinite face functions and yellow the tangential infinite cell functions. In the bottom row: red represents the ray functions, blue the orthogonal infinite face functions and yellow the orthogonal infinite cell functions.

5.2 Problem Definitions

Now that we have introduced the finite element method we define two versions of the problems we will consider in our numerical experiments: The time-harmonic Maxwell's equations on a bounded domain truncated by PML where we use either Dirichlet data or tapered coupling to couple a signal into the system.

Definition 5.2.1 (Problem 1: Dirichlet Data)**.** *Provided a computational domain*

$$\overline{\Omega}_C = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}] \subset \mathbb{R}^3 \tag{5.15}$$

with

$$x_{min} < x_{max} \quad and \quad y_{min} < y_{max} \quad and \quad z_{min} < z_{max}, \tag{5.16}$$

numbers n_x, n_y and n_z of cells in the x, y and z, an order $o \in \mathbb{N}$ with $n \geq 0$ and an input signal $\boldsymbol{E}_I(x, y)$, we call solving eq. (4.126) using Nédélec elements of order k ***problem 1****.*

Definition 5.2.2 (Problem 2: Tapered Coupling)**.** *Provided a computational domain*

$$\overline{\Omega}_C = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \times [z_{min}, z_{max}] \subset \mathbb{R}^3 \tag{5.17}$$

with

$$x_{min} < x_{max} \quad \text{and} \quad y_{min} < y_{max} \quad \text{and} \quad z_{min} < z_{max} \tag{5.18}$$

numbers n_x, n_y and n_z of cells in the x, y and z, an order $k \in \mathbb{N}$ with $n \geq 0$ and an input signal $\boldsymbol{E}_I$ represented by $\boldsymbol{J}$ (see section 4.6.2), we call solving eq. (4.125) using Nédélec elements of order k ***problem 2****.*

5.3 Sweeping Preconditioners

5.3.1 Motivation

Employing the techniques introduced before, we could come to the conclusion that the problem has been solved. The steps are as follows:

- Choose a domain of interest.
- Enclose it in PML to truncate the domain.
- Compute an input signal for the specified waveguide.
- Discretize the domain using Nédélec elements.
- Assemble the system and right-hand-side.
- Solve the system.
- Evaluate any functional of interest on the solution (i.e. signal loss etc).

This is, however, where problems begin to arise: As mentioned in [TEY12], the oscillatory nature of the Green's function of Maxwell's equations causes the system matrix A to be highly indefinite and ill-conditioned. The high, oscillatory nature of the solution and non-existent dispersion of the solution cause the inverse of A to be a dense matrix. We observe the following two issues:

1. Direct solvers, such as **UMFPACK** (see [Dav04]) or **MUMPS** (see [Ame+01]), have a very high memory consumption that scales with the square of the number of degrees of freedom in the system. With current PC-architectures, algorithms and time-constraints, this leads to a very low limit on the number of degrees of freedom and thus the volume of the computational domain. This is in part the reason why in publications the number of degrees of freedom ends at around 10^6 (such as [Bur+06]) because the limits of direct solvers are reached at this point.
   Remark. *There is a fundamental issue here, that extends to any direct solver: A direct solver necessarily has the memory consumption of the storage of the inverse of the system matrix. As a consequence, while there are some modern distributed direct solvers like PARDISO (see [SG04]), their memory consumption will still be in excess of what can be afforded. For example: Suppose we have a system of 10^8 degrees of freedom (as we will solve in this work). A full matrix of this size would hold 10^{16} complex valued entries. At double precision, storage of this object would require $1.6 \cdot 10^{17}$ Bytes of memory or 160 petabytes, which is 32 times the memory capacity of the currently largest HPC system in the world (see Top500 List, November 2021).*
2. The common alternative, iterative solvers, runs into a similar issue: The high condition number of the matrix A has the effect of halting the convergence of such algorithms as **GMRES** (see [SS86])

and **CG** (see [Kel]) converge at such a low rate that the number of stored iteration results would reach the same level as that of direct solvers. This effect is well-known and documented in works like [TEY12], [BH96] or [Mon92, chapter 13].

To construct a solution in spite of this, we will begin by attempting to alleviate the problems arising from the application of a direct solver such as UMFPACK or MUMPS to a system with many degrees of freedom. In chapter 4, we saw that a scattering problem can be truncated to reduce its size by means of an absorbing boundary condition such as PML. We will employ this technique to split our problem into smaller chunks and construct solutions on these local parts. We will then use block-structure arguments to construct solutions for the original problem.

Remark. *We will be using the terms* ***global*** *and* ***local*** *problem in this work. Global will always refer to the complete problem we are trying to solve. We will be splitting the global problem into smaller parts and applying methods to these parts. In the context of such a method applied to a part of the larger global problem, we will refer to the smaller partial problems as local problems. This terminology is derived from the parallelization techniques that will be used on HPC systems to numerically solve these problems. The global problem will be distributed across many processors and they will solve it in tandem – the local problems, however, are either only stored and solved on one computer or on a subgroup of the full compute-system.*

Initial Approach

We will begin by recounting the work done in [TEY12] to introduce **sweeping preconditioners** with a **moving PML** method for spatial truncation adapting the work to our notation. Further details on the principles of these methods can be found in [GZ16] and [GZ18].

Let

$$\overline{\Omega}_C = [-1,1] \times [-1,1] \times [-1,1] \quad \text{and} \tag{5.19}$$
$$\Gamma_I = [-1,1] \times [-1,1] \times \{-1\}. \tag{5.20}$$

The numerical examples will use a different computational domain but it will still be an axis parallel cuboid. As per the nomenclature in this document, this domain already includes the PML domains required to truncate the global problem. We split this domain into subdomains N in the z direction, resulting in

$$\Omega_i = (-1,1) \times (-1,1) \times \left(-1 + \frac{(i-1)}{2N}, -1 + \frac{i}{2N}\right) \quad \text{for } i \in \{1,\dots,N\}. \tag{5.21}$$

We choose our mesh size such that it aligns with this splitting, meaning that the number of cells in the sweeping direction z is a multiple of N. Additionally, we sort the degrees of freedom with an ascending numbering in z direction based on the central point of their associated base structure (cell, face or edge). Cell degrees of freedom will only have support on one subdomain, since they only have support on one cell and each cell is part of one subdomain. Face and edge degrees of freedom, however, can either have support on one or two subdomains (two if they are associated with a face or edge on the interface between two neighboring subdomains). This allows us to define the index sets for the degrees of freedom associated with the subdomains for both the discretization of the domain of interest as well as the boundary methods applied on the surface:

$$\mathcal{I}_1 := \{j : \text{degree of freedom } j \text{ has support on } \Omega_1\} \quad \text{and} \tag{5.22}$$
$$\mathcal{I}_i := \{j : \text{degree of freedom } j \text{ has support on } \Omega_i \text{ but not on } \Omega_{i-1}\} \quad \text{for } i \in \{2,\dots,N\}. \tag{5.23}$$

This means that every degree of freedom is only included in one set and if it is active on two subdomains, it is counted in the lower one of the two. The splitting of a subdomain into subdomains for an axis-parallel, hexahedral mesh is depicted in fig. 5.2.

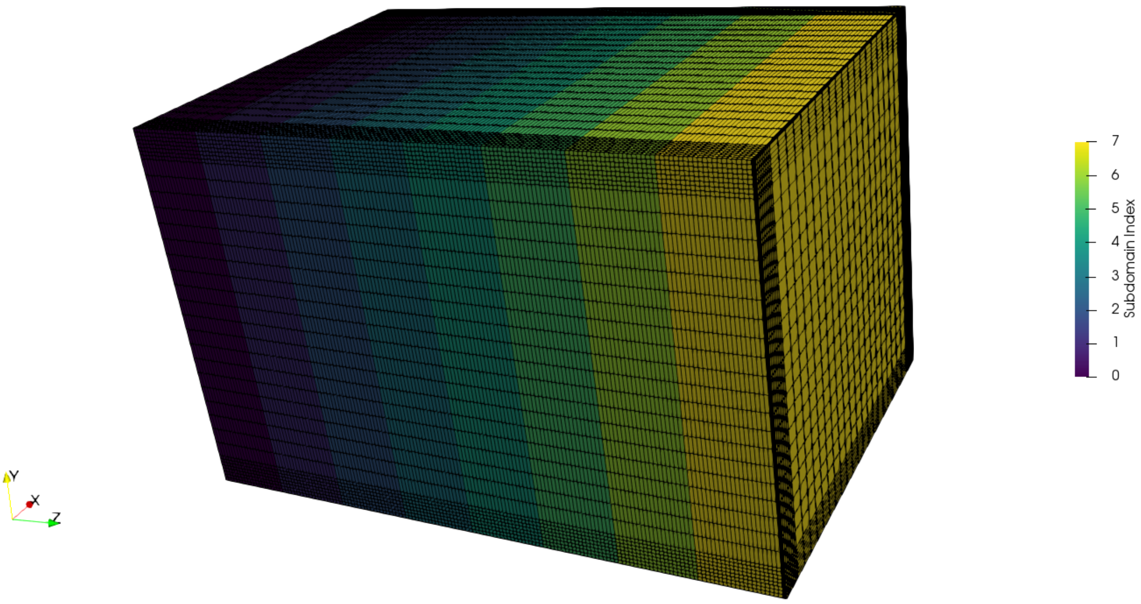


Figure 5.2: Subdomain split for a hexahedral, axis-parallel mesh.

Discretizing the system

$$\nabla \times \boldsymbol{\mu} \nabla \times \boldsymbol{E} - \boldsymbol{\epsilon}\omega^2 \boldsymbol{E} = \boldsymbol{F} \qquad \text{in } \Omega_C, \tag{5.24a}$$

$$\boldsymbol{E} \times \boldsymbol{n} = \boldsymbol{0} \qquad \text{on } \partial\Omega_C \setminus \Gamma_I \tag{5.24b}$$

yields a linear system of equations of the form

$$A\boldsymbol{u} = \boldsymbol{b}, \tag{5.25}$$

where we call $A \in \mathbb{C}^{D \times D}$ the stiffness or system matrix, $\boldsymbol{b}$ the right hand side and $D \in \mathbb{N}$ the number of degrees of freedom. The computed solution $\boldsymbol{E}_h$ is defined as

$$\boldsymbol{E}_h = \sum_{i=1}^{D} u_i \boldsymbol{\phi}_i, \tag{5.26}$$

where $\boldsymbol{\phi}_i$ are the shape functions of our finite element. Using the numbering of degrees of freedom we introduced earlier and a block notation style, where $A(\mathcal{I}_i, \mathcal{I}_j)$ is the part of A consisting of only the rows whose index is included in $\mathcal{I}_i$ and whose columns are included in $\mathcal{I}_j$. We can write eq. (5.25) as

$$\begin{pmatrix} A(\mathcal{I}_1,\mathcal{I}_1) & A(\mathcal{I}_1,\mathcal{I}_2) & & \\ A(\mathcal{I}_2,\mathcal{I}_1) & A(\mathcal{I}_2,\mathcal{I}_2) & \ddots & \\ & \ddots & \ddots & A(\mathcal{I}_{N-1},\mathcal{I}_N) \\ & & A(\mathcal{I}_N,\mathcal{I}_{N-1}) & A(\mathcal{I}_N,\mathcal{I}_N) \end{pmatrix} \begin{pmatrix} u(\mathcal{I}_1) \\ u(\mathcal{I}_2) \\ \vdots \\ u(\mathcal{I}_N) \end{pmatrix} = \begin{pmatrix} b(\mathcal{I}_1) \\ b(\mathcal{I}_2) \\ \vdots \\ b(\mathcal{I}_N) \end{pmatrix}. \tag{5.27}$$

This system is indefinite and therefore the computation of an LDL^T factorization is inadvisable (see [NW06, Chapter 3.4], [Fan10]). Successively applying a Schur-complement process can be used, however, to construct a blockwise LDL^T factorization of the system, yielding

$$A = L_1 \dots L_{N-1} \begin{pmatrix} S_1 & & & \\ & S_2 & & \\ & & \ddots & \\ & & & S_N \end{pmatrix} L_{N-1}^T \dots L_1^T, \tag{5.28}$$

with the Schur-complement matrices S_i defined by

$$A_{i,j} = A(\mathcal{I}_i, \mathcal{I}_j), \tag{5.29}$$

$$S_1 = A_{1,1}, \tag{5.30}$$

$$S_i = A_{i,j} - A_{i,i-1} S_{i-1}^{-1} A_{i-1,i} \quad i \in \{2, \dots, N\} \text{ and} \tag{5.31}$$

$$L_i(\mathcal{I}_j, \mathcal{I}_k) = \begin{cases} \boldsymbol{Id} & j = k \\ A_{j,k} S_i^{-1} & i = j - 1 = k \\ 0 & \text{else.} \end{cases} \tag{5.32}$$

To show these identities, we start with $N = 2$, which yields the system

$$\underbrace{\begin{pmatrix} \boldsymbol{Id} & 0 \\ A_{2,1} A_{1,1}^{-1} & \boldsymbol{Id} \end{pmatrix}}_{L_1} \begin{pmatrix} A_{1,1} & 0 \\ 0 & A_{2,2} - A_{2,1} A_{1,1}^{-1} A_{1,2} \end{pmatrix} \underbrace{\begin{pmatrix} \boldsymbol{Id} & (A_{2,1} A_{1,1}^{-1})^T \\ 0 & \boldsymbol{Id} \end{pmatrix}}_{L_1^T} \tag{5.33}$$

$$= \begin{pmatrix} A_{1,1} & 0 \\ A_{2,1} & A_{2,2} - A_{2,1} A_{1,1}^{-1} A_{1,2} \end{pmatrix} \begin{pmatrix} \boldsymbol{Id} & A_{1,1}^{-T} A_{1,2} \\ 0 & \boldsymbol{Id} \end{pmatrix} \tag{5.34}$$

$$= \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = A. \tag{5.35}$$

This can be repeated to verify the formulation.

Inserting eq. (5.28) in eq. (5.25) and applying the inverses of the individual matrices yields the equation

$$\begin{pmatrix} u(\mathcal{I}_1) \\ u(\mathcal{I}_2) \\ \vdots \\ u(\mathcal{I}_N) \end{pmatrix} = L_1^{-T} \dots L_{N-1}^{-T} \begin{pmatrix} S_1^{-1} & & & \\ & S_2^{-1} & & \\ & & \ddots & \\ & & & S_N^{-1} \end{pmatrix} L_{N-1}^{-1} \dots L_1^{-1} \begin{pmatrix} b(\mathcal{I}_1) \\ b(\mathcal{I}_2) \\ \vdots \\ b(\mathcal{I}_N) \end{pmatrix}. \tag{5.36}$$

To build the terms S_i^{-1} we can start by inverting S_1 and then sequentially constructing the following matrices and inverting them by evaluating eq. (5.31). Once the matrices S_i^{-1} have been computed, the exact application of the inverse written as a pseudo-code algorithm takes the form presented in algorithm 1.

Remark. *In this notation, the for-loops get evaluated from left to right. That means that the last for-loop in the algorithm above starts at $i = N - 1$ and iterates down to $i = 1$.*

In this algorithm, the first loop initializes the vector to contain the solution with the value of b. The second loop applies the blocks $L_{N-1}^{-1} \dots L_1^{-1}$ from eq. (5.36), where the block $u(\mathcal{I}_{i+1})$ is caused by the identity term in eq. (5.32) and the other term is the evaluation of the middle term in eq. (5.32). The third loop evaluates the application of the block-diagonal matrix and the third loop applies $L_1^{-T} \dots L_{N-1}^{-T}$ consecutively.

Algorithm 1 Algorithm for the application of the full inverse of the system matrix.

for $1 \le i \le N$ **do**
  $u(\mathcal{I}_i) \leftarrow b(\mathcal{I}_i)$
end for

for $1 \le i \le N-1$ **do**
  $u(\mathcal{I}_{i+1}) \leftarrow u(\mathcal{I}_{i+1}) - A_{i+1,i} S_i^{-1} u(\mathcal{I}_i)$
end for

for $1 \le i \le N$ **do**
  $u(\mathcal{I}_i) \leftarrow S_i^{-1} u(\mathcal{I}_i)$
end for

for $N-1 \ge i \ge 1$ **do**
  $u(\mathcal{I}_i) \leftarrow u(\mathcal{I}_i) - S_i^{-1} A_{i,i+1} u(\mathcal{I}_{i+1})$
end for

There is one main problem with the computation of the matrices S_i^{-1}: The system matrix is sparse and therefore $A_{i,i}$ is sparse, too. Thus, computing S_1^{-1} only requires the inversion of a sparse matrix where we can make use of a sparsity pattern and expect a direct solver to perform well. The matrices S_i for $i \neq 1$, however, depend on the inverse of S_1, which is a dense matrix. As a consequence, the computational cost of inverting these matrices is high. Additionally, these matrices have to be computed sequentially. The effort of the preparation of all matrices S_i^{-1} is in $O(N_{\text{dofs}}^{\frac{7}{3}})$ (see [EY11]). This implies that this method cannot be applied to large systems. We therefore move away from the ansatz of constructing the inverse of A precisely and instead assemble an approximation.

As described in [TEY12], we now regard the first m block rows from eq. (5.27) and set $A(\mathcal{I}_{m,m+1}) = \mathbf{0}$ which leads to the system

$$\begin{pmatrix} A(\mathcal{I}_1,\mathcal{I}_1) & A(\mathcal{I}_1,\mathcal{I}_2) & & \\ A(\mathcal{I}_2,\mathcal{I}_1) & A(\mathcal{I}_2,\mathcal{I}_2) & \ddots & \\ & \ddots & \ddots & A(\mathcal{I}_{m-1},\mathcal{I}_m) \\ & & A(\mathcal{I}_m,\mathcal{I}_{m-1}) & A(\mathcal{I}_m,\mathcal{I}_m) \end{pmatrix} \begin{pmatrix} u(\mathcal{I}_1) \\ u(\mathcal{I}_2) \\ \vdots \\ u(\mathcal{I}_m) \end{pmatrix} = \begin{pmatrix} b(\mathcal{I}_1) \\ b(\mathcal{I}_2) \\ \vdots \\ b(\mathcal{I}_m) \end{pmatrix}. \tag{5.37}$$

The condition $A(\mathcal{I}_{m,m+1}) = \mathbf{0}$ is equivalent to PEC boundary condition (see section 4.4.1) on $\overline{\Omega_m} \cap \overline{\Omega_{m+1}}$. The system eq. (5.37) is therefore the discretization of the problem

$$\nabla \times \boldsymbol{\mu} \nabla \times \boldsymbol{E} - \boldsymbol{\epsilon}\omega^2 \boldsymbol{E} = \boldsymbol{F} \qquad \text{in } \bigcup_{i=1}^{m} \Omega_i, \tag{5.38a}$$

$$\boldsymbol{E} \times \boldsymbol{n} = \mathbf{0} \qquad \text{on } \partial \bigcup_{i=1}^{m} \Omega_i \setminus \Gamma_I. \tag{5.38b}$$

In eq. (5.36) we observe that there is no block L_N^{-1} or L_N^{-T}, meaning that the right and left side of the diagonal matrix have no effect on S_N^{-1}. Therefore, evaluating all the matrix products in eq. (5.36) will result in the block at position (N,N) being S_N^{-1}. The same holds for the inverse of the system of reduced size m. For a solution of eq. (5.41) it therefore holds that

$$u(\mathcal{I}_m) = S_m^{-1} b(\mathcal{I}_m). \tag{5.39}$$

At this point, we still have no better way of computing S_m^{-1} so we will not be able to evaluate this equation efficiently. If we consider this system embedded in a larger one, however, we observe that to fulfill the role of S_m^{-1} in eq. (5.36) it only has to be a good approximation on the index set $\mathcal{I}_m$. If we truncate the computational system by filling the subdomain $m-1$ with an absorbing medium, assemble the system of equations resulting from that setup and invert the resulting system matrix, that matrix, by definition fulfills this requirement.

We define the auxiliary domains

$$\Omega_i^{\mathrm{pml}} = (-1,1) \times (-1,1) \times \left(-1 + \frac{i}{2N} - d, -1 + \frac{i}{2N}\right] \quad \text{for } i \in \{2, \ldots, N\} \tag{5.40}$$

where d is the thickness of the PML layer and introduce the auxiliary problems

$$\nabla \times \boldsymbol{\mu} \nabla \times \boldsymbol{E} - \boldsymbol{\epsilon}\omega^2 \boldsymbol{E} = \boldsymbol{F} \qquad \text{in } \Omega_i \cup \Omega_i^{\mathrm{pml}}, \tag{5.41a}$$

$$\boldsymbol{E} \times \boldsymbol{n} = \boldsymbol{0} \qquad \text{on } \partial\left(\Omega_i \cup \Omega_i^{\mathrm{pml}}\right). \tag{5.41b}$$

In eq. (5.41b) $\boldsymbol{n}$ is the outward normal vector for the domain $\left(\Omega_i \cup \Omega_i^{\mathrm{pml}}\right)$. For each subdomain except the first, we construct an extended local problem, which consists of the original pde on the i-th subdomain, truncated by a PML layer in the $(i-1)$st subdomain. The PML introduces additional degrees of freedom that are not part of the original system and which we will sort after the degrees of freedom present in the i-th block. Discretizing these auxiliary problems yields the matrices $\widetilde{S}_i$, which are sparse finite element matrices that we can factorize cheaply and define the operators

$$H_i^{-1}\boldsymbol{v} = \left(\widetilde{S}_i^{-1}\begin{pmatrix}\boldsymbol{v}\\ \boldsymbol{0}\end{pmatrix}\right)^T \begin{pmatrix}\boldsymbol{Id}\\ \boldsymbol{0}\end{pmatrix} \quad \text{and} \tag{5.42}$$

$$H_1^{-1}\boldsymbol{v} = A_{1,1}^{-1}\boldsymbol{v}. \tag{5.43}$$

Remark. *The auxiliary problem uses an additional PML domain and therefore has some additional degrees of freedom. All the operator H_i^{-1} does, is to extend the vector $\boldsymbol{v}$ with zeros for these additional dofs. It then applies the inverse of the system matrix of the local problem and extracts the first* $\dim(\boldsymbol{v})$ *components from the solution, which simply abandons the solution computed in the added PML domain.*

Since the matrices $\widetilde{S}_i$ are stiffness matrices of a local problem and the problem on the subdomain only has $\frac{1}{m}$ times the size of the original system, they are much cheaper to factorize than A^{-1}. This is the crucial point for sweeping preconditioners: When A^{-1} can no longer be applied due to cost, the operators $\widetilde{S}_i^{-1}$ may still be applicable.

As a next step, we replace the application of the precise Schur-blocks S_i^{-1} in the first version of the algorithm with the approximated local operators H_i^{-1} and find the basic 1D sweeping algorithm from [TEY12] as listed in algorithm 2.

As a first observation, we note that in both the first and second for-loop (specifically lines 6 and 10 of algorithm 2), we compute the costly term $H_i^{-1}u(\mathcal{I}_i)$. We can therefore reduce the cost of the algorithm by rephrasing it to a version that only contains three loops instead of four (see algorithm 3).

This reformulation saves a third of the applications of the operators H_i^{-1} compared to [TEY12].

Algorithm 2 Algorithm for the application of the approximate inverse to a vector $\boldsymbol{b}$.

for $1 \leq i \leq N$ **do**
  $u(\mathcal{I}_i) \leftarrow b(\mathcal{I}_i)$
end for

for $1 \leq i \leq N-1$ **do**
  $u(\mathcal{I}_{i+1}) \leftarrow u(\mathcal{I}_{i+1}) - A_{i+1,i}H_i^{-1}u(\mathcal{I}_i)$
end for

for $1 \leq i \leq N$ **do**
  $u(\mathcal{I}_i) \leftarrow H_i^{-1}u(\mathcal{I}_i)$
end for

for $N-1 \geq i \geq 1$ **do**
  $u(\mathcal{I}_i) \leftarrow u(\mathcal{I}_i) - H_i^{-1}A_{i,i+1}u(\mathcal{I}_{i+1})$
end for

Algorithm 3 Updated version of algorithm 2.

for $1 \leq i \leq N$ **do**
  $u(\mathcal{I}_i) \leftarrow b(\mathcal{I}_i)$
end for

for $1 \leq i \leq N-1$ **do**
  $u(\mathcal{I}_i) \leftarrow H_i^{-1}u(\mathcal{I}_i)$
  $u(\mathcal{I}_{i+1}) \leftarrow u(\mathcal{I}_{i+1}) - A_{i+1,i}u(\mathcal{I}_i)$
end for

$u(\mathcal{I}_N) \leftarrow H_N^{-1}u(\mathcal{I}_N)$

for $N-1 \geq i \geq 1$ **do**
  $u(\mathcal{I}_i) \leftarrow u(\mathcal{I}_i) - H_i^{-1}A_{i,i+1}u(\mathcal{I}_{i+1})$
end for

5.3.2 PML Tuning

One of the downsides of using a PML medium to truncate the computational domain is the seemingly arbitrary choice of the parameters. If we choose to go the route of an increasing value of sigma towards the outside of the PML-medium as presented in eq. (4.45), we have to choose

- the value σ_{max}, which is the value of σ on the outer surface,
- the scaling exponent, which is typically either $k = 0$ for constant σ within the PML region or $2 \leq k \leq 4$ for increasing σ towards the outer boundary,
- the number of cell-layers the PML-medium consists of and
- the thickness of the domain.

The PML should not influence the wavelength very much, so we choose the number of cell layers and the thickness of the domain in a similar way we choose them for the interior domain. We use ≈ 10 cell-layers per wavelength and set the thickness of the PML domain equal to the wavelength. These values, however, are educated guesses and should be evaluated further in numerical experiments.

For the remaining properties, we have less of an indication which values to use. The fundamental problem with these parameters is the existence of sweet-spots for their values. If σ_{max} is chosen too small, the PML-medium will not dampen the incoming wave quickly enough and it will be reflected on the metallic boundary condition we place on the outside surface. If, on the other hand, we choose the value too large, there will be reflections inside the PML medium. In theory, the norm of the PML-tensor scales continuously with the outward coordinate. Numerical discretization, however, causes an evaluation of the material tensor at discrete locations. If σ_{max} is chosen very large, the difference of the norm of the material tensors between quadrature points will also increase and this will cause reflections.

The same problem can be observed for the number of cell layers: For very few cell layers, on the one hand, the absorption of the PML will not be efficient, introducing numerical errors. Increasing the number of cell layers on the other hand, drastically increases the size of the system matrix and thus the numerical cost of solving the system.

For illustration we consider this setup: Let the domain of interest be a cube meshed by 10 cells in each coordinate direction. Using Nédélec elements of lowest order we find the number of inner dofs

$$N_i = 3 \cdot 11 \cdot 11 \cdot 10 = 3630. \tag{5.44}$$

In the next step, we wrap this domain in a PML, which is 5 cells thick in each direction. We find the total number of dofs

$$N_t = 3 \cdot 21 \cdot 21 \cdot 20 = 26460 \approx 7.3 \cdot N_i. \tag{5.45}$$

In this system, less then 14% of the dofs would be used to solve the partial differential equation in the domain of interest and 86% are PML dofs.

5.3.3 Problems with Sweeping Preconditioners

We have established that sweeping preconditioners can be used to split a large domain of interest into smaller parts, impose either PEC or absorbing boundary conditions on the interfaces and then construct a preconditioner for the original system from direct solvers applied to the problems on the subdomains. We now regard the question of scaling this method to very large domains of interest. Two main problems arise:

Scaling of the sweeping method for increasing numbers of subdomains. As a toy problem, we will use the example of a straight waveguide with Dirichlet data on the input interface at $z = 0$. We set a fixed width in the x and y direction and discretize in those directions by a fixed amount of cells. We want to run a series of computations for an increasing number of subdomains n in the z direction. We choose a subdomain-length in the z direction z_s and set the total length of the domain of interest to $n\, z_s$. In essence, we compose the computational domain of building blocks of the same size for an increasing number of processes. As the input signal we choose the fundamental mode of the input waveguide. In this setup, we expect the signal from the input interface to propagate without backward reflection because the fundamental mode propagates without losses and therefore the conservation of energy requires the signal to travel along the waveguide without loss.

These kinds of setups are ideal for sweeping preconditioners because each reflection would require another cycle of forward and backward sweeping to incorporate in the solution. In our setup, the only need for multiple sweeping cycles arises from discretization errors, such as numerical dispersion and reflections from PML domains.

In experiment 2 we investigate the runtime and number of GMRES steps required to achieve convergence. We observe that, as the number of subdomains increases, the number of GMRES steps increases linearly. This is due to the linear increase of errors introduced by PML-layers (because the number of PML-layers in the system increases linearly, since there are 5 PML layers per subdomain for the subdomain problems) and of errors introduced by numerical dispersion (because these errors scale with the volume of the computational domain, which increases linearly with the number of subdomains).

As we increase the number of subdomains, the number of sequential steps in the sweeping algorithm (and thus the runtime of a single application of the preconditioner) scales linearly. Additionally, we require more steps and thus we see the total runtime of the algorithm scale quadratically. We see this behaviour in fig. 5.3.

Experiment 2. ***Setup:*** *For* $i \in \{1, \dots, 16, 20, 24, 28, 32, 64, 128, 256\}$ *we split the domain* $\Omega_I = [-2, 2] \times [-1.8, 1.8] \times [0, 2.0 \cdot i]$ *into* i *equal subdomains in the* z *direction. Each subdomain consists of* 20 *cells in each direction. We set the mode computed in experiment 1 as a Dirichlet value on the input side* $(-z)$ *and wrap the domain in* 10 *cells of PML with* σ *from eq.* (4.45) *for* $k = 3$*,* $\sigma_{max} = 30$ *and* $d = 0.5$*. We then solve eq.* (4.126) *discretized by Nédélec elements of order* 1 *using algorithm 3 and GMRES solver with absolute convergence criterion* 10^{-6}*. Including the PML domains we find* $\overline{\Omega_C} = [-2-d, 2+d] \times [-1.8-d, 1.8+d] \times [0, 2.0 \cdot i + d]$
Results: *The run times of the solver and the number of GMRES steps are plotted in fig. 5.3 where we see that the number of steps scales linearly (approximated by* $g(n) = 0.09n + 5.34$*) and that the time to solve scales quadratically (approximated by* $f(n) = 0.25n^2 + 9n + 800$*).*

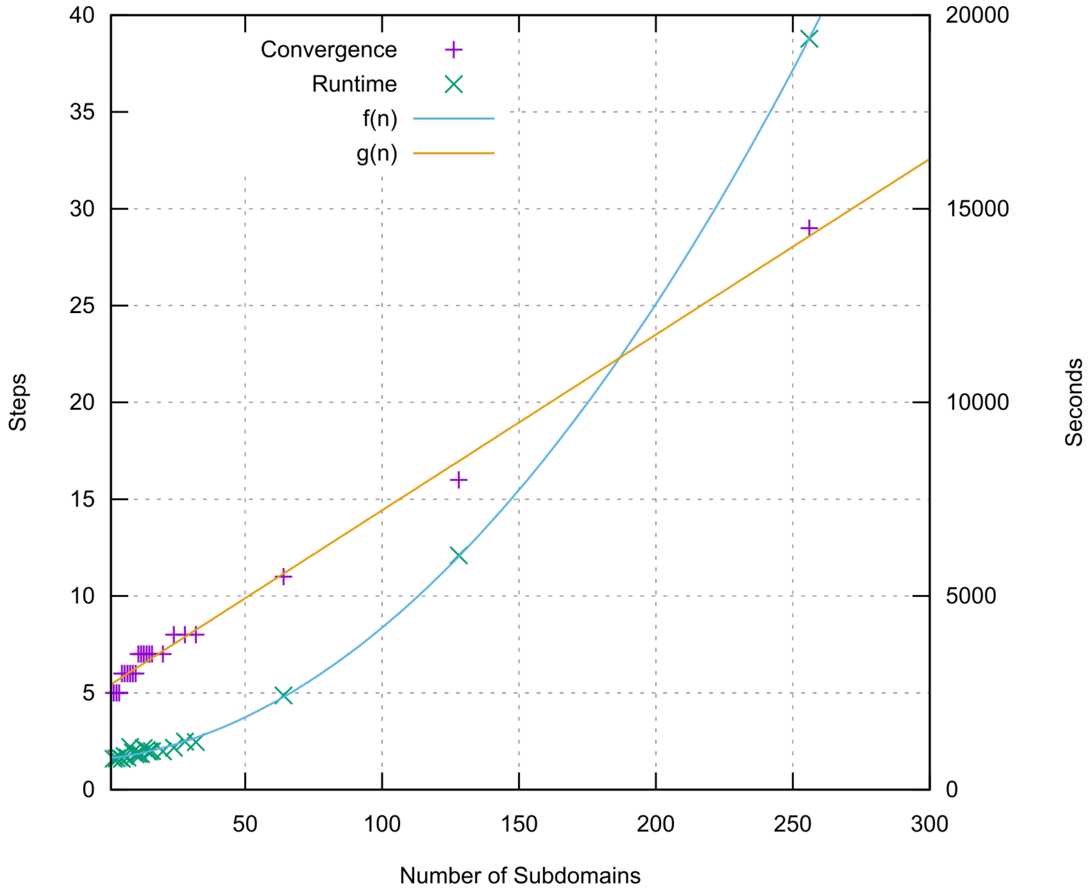


Figure 5.3: Results of experiment 2

Construction of solvers for the sweeping subdomains. Let us assume that the domain of interest requires a minimum of $100 \times 100 \times 100$ cells for the solution to be resolved appropriately and that we need

10 cell layers of PML to truncate a partial problem. Additionally, we will assume that on the computers at our disposal, we will only be able to solve a system with 500.000 dofs with a direct solver. A subdomain composed of s cell layers would then have

$$N_d = \underbrace{(s+20)\cdot 121^2}_{z \text{ dofs}} + \underbrace{2\cdot(s+21)\cdot 121\cdot 120}_{x \text{ and } y \text{ dofs}} \tag{5.46}$$

$$= 292820 + 14641s + 609840 + 29040s \tag{5.47}$$

$$= 902660 + 43681s \tag{5.48}$$

degrees of freedom. As a consequence, we would not be able to solve the subdomain problems even if we chose $s = 1$. Not only are we not able to solve the subdomain problems, but even if we were able to solve for $s = 1$, our individual problems would be one layer of cells each, wrapped in 10 cells of PML in each direction. If the PML introduces even a small numerical error in that case, it will influence every degree of freedom in the system directly because every degree of freedom in the domain of interest would also couple to at least one PML surface. In this case, we would additionally have 100 subdomains in the sweeping direction, which, as we saw above, has a negative impact on the convergence rate.

5.4 A Hierarchical Sweeping Preconditioner

5.4.1 Concept

As discussed above, we note the fact that in very large 3D simulations using PML to truncate the computational domain can lead to situations in which even the subdomain problems are too large to be solved directly. We observe, however, that the problems we solve on the subdomains are of the same nature as the global problem: a 3D fem simulation of the same partial differential equation with similar boundary conditions. We therefore choose to solve this system using an iterative solver with a sweeping preconditioner. We can repeat this procedure until the subdomain problems are small enough to be solved with a direct solver, which is typically at around 200.000 degrees of freedom. Choosing the new sweeping direction orthogonally to the original sweeping direction has the advantage of allowing us to split the computational domain into subdomains with more control over the shape of the subdomains, eventually arriving at cube-shaped domains on the lowest level, in which the influence of errors introduced by boundary conditions are not as dominant as they would be in a 1-cell-layer in the xy-plane.

5.4.2 Description of the Method

To evaluate algorithm 3 the main numerical cost is to compute the application of the local solvers H_i^{-1} since all other steps are only vector operations and block-matrix-vector products, which are cheap in comparison to solving a linear set of equations. For a basic 1D sweeping scheme, these sets are solved directly, since the indefinite nature and ill conditioning of these matrices still prevents the application of an iterative solver.

We, however, propose the application of a sweeping preconditioner to these systems. This allows for these subdomain problems to consist of more cells and therefore degrees of freedom. For the given problem of applying a sweeping preconditioner to a vector, two properties have to hold:

1. The system has to be discretized fine enough for the local solvers to be a good approximation of the local inverse and

2. we must be able to evaluate the local solver (in view of memory and run time).

One-dimensional sweeping preconditioners enforce these two conditions in one step. For the hierarchical sweeping preconditioner, we weaken this restriction: Let us first assume that the global problem we are attempting to solve is well discretized by, $h \approx \frac{\lambda}{20}$. Since the subdomains are subsets of the triangulation of the global problem, they maintain that same grid constant h and thus automatically fulfill condition 1.

To visualize the impact of the first condition we ran a series of computations, in which we left all parameters equal and only increased the length of the computational domain. This has the effect of continuously decreasing the quality of the discretization of the domain and as a consequence, condition one deteriorates. The experiment is described in experiment 3 and the resulting increase in required iterations for GMRES to converge as well as the increase in run time are shown in fig. 5.4.

Experiment 3 (Sweeping on a stretched domain)**.** ***Setup:*** *For this experiment we use the same setup as in experiment 2 – a straight waveguide, 1D sweeping, PML for truncation, Dirichlet data for the signal input. The only difference being that we vary the system length without increasing the cell count or the subdomain count. As a consequence, the cells increase in length in the z direction. We use* $20 \times 20 \times 20$ *cells per process and* 8 *processes leading to a total of* $20 \times 20 \times 160$ *cells.*
Result: *As the length increases, so does the required number of steps until the convergence criterion* (10^{-6}) *is reached. The time scales equally. The results are shown in fig. 5.4. A typical criterion for the approximation of waves is to use* 10 *nodes per wavelength to approximate the wave reasonably well. In this experiment, this criterion holds for system lengths of up to* $32\mu m$ *(because of* $\lambda_0 = 1.55\mu m$ *and* $n = \sqrt{2.3409} = 1.53$ *and therefore* $\lambda \approx 1\mu m$*). We see in fig. 5.4 that the preconditioner deteriorates beyond this point.*

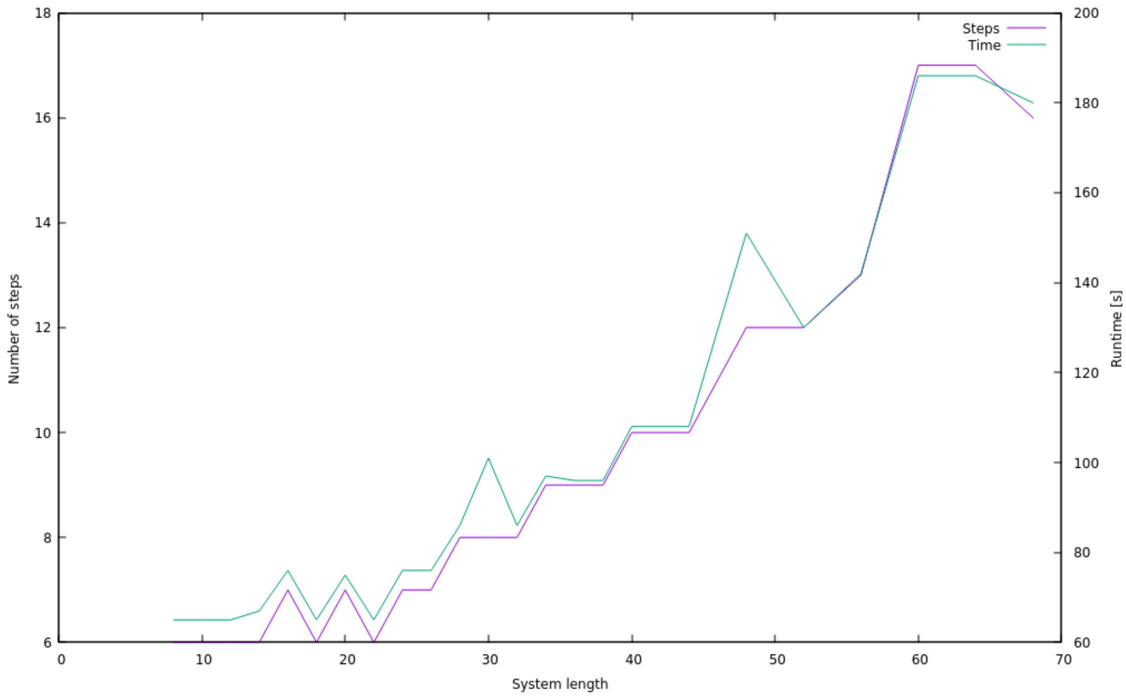


Figure 5.4: Required steps and runtime of a GMRES solver with a 1D sweeping preconditioner for an increasing length of the computational domain.

We saw in section 5.3.3 that sweeping preconditioners do not scale optimally as the number of subdomains increases. It is therefore preferable to set a maximum number of subdomains to split the global problem into. These subdomains may be too large to be solved with a direct solver and in this situation, we propose the application of an iterative solver with a sweeping preconditioner, whose sweeping direction is orthogonal to that of the original sweeping preconditioner. The stability of this scheme relies on the same properties as the global sweeping preconditioner – the only remaining question is the compatibility of the lower level sweeping preconditioner with boundary conditions on this level.

We will introduce a formulation of this scheme that generalizes the concept of the moving PML method used in 1D sweeping preconditioners in the following section and provide numerical results for the convergence of this scheme at scales for which 1D sweeping preconditioners are no longer applicable.

5.4.3 The Hierarchical Sweeping Algorithm

Remark. *We will be introducing the higher level sweeping preconditioners for problems of type* 1 *(i.e. Dirichlet data to couple the signal into the computational domain) as described in definition 5.2.1. The concept is not dependent on this, however, and can easily be transferred for problem* 2*.*

For a problem of type 1 with

$$x_{\min} = y_{\min} = z_{\min} = 0 \quad \text{and} \quad x_{\max} = y_{\max} = z_{\max} = 1, \tag{5.49}$$

three integers $2 \leq n_x, n_y, n_z$ and the PML layer thickness d we define

$$L_x = \frac{x_{\max} - x_{\min}}{n_x} \tag{5.50}$$

$$L_y = \frac{y_{\max} - y_{\min}}{n_y} \tag{5.51}$$

$$L_z = \frac{z_{\max} - z_{\min}}{n_z} \tag{5.52}$$

and the intervals

$$I_{\#}^{(i)} = ((i-1)L_{\#}, iL_{\#} + d) \quad \text{for } 1 \leq i < n_{\#} \quad \text{and} \tag{5.53}$$

$$I_{\#}^{(n_{\#})} = ((n_{\#} - 1)L_{\#}, 1)\,. \tag{5.54}$$

with $\# \in \{x, y, z\}$. Next we introduce the sweeping domains as follows:

Definition 5.4.1 (Sweeping domains)**.** *For the sweep in the z direction we define the subdomains*

$$\Omega_C^{(i)} = (0,1) \times (0,1) \times I_z^{(i)} \quad \textit{for } 1 \leq i \leq n_z. \tag{5.55}$$

For the sweep in the y direction we define the subdomains

$$\Omega_C^{(i,j)} = (0,1) \times I_y^{(j)} \times I_z^{(i)} \quad \textit{for } 1 \leq i \leq n_z \textit{ and } 1 \leq j \leq n_y. \tag{5.56}$$

For the sweep in the x direction we define the subdomains

$$\Omega_C^{(i,j,k)} = I_x^{(k)} \times I_y^{(j)} \times I_z^{(i)} \quad \textit{for } 1 \leq i \leq n_z \textit{ and } 1 \leq j \leq n_y \textit{ and } 1 \leq k \leq n_x. \tag{5.57}$$

Remark. *The PML thickness of the auxiliary layers can be chosen lower than the subdomain lengths L_x, L_y and L_z.*

Definition 5.4.2 (Problem 3: Sweeping Operators)**.** *For level* 1 *sweeping, we define the operator S^i as the system matrices for problem* 1 *constructed on $\Omega_C^{(i)}$. For the x and y directions, the boundary conditions remain unchanged. For the upper boundary in z direction we apply an auxiliary PML (as depicted in fig. 5.5) if $i < n_z$. For $i = n_z$, the boundary condition of Ω_C is inherited (PML). On the lower boundary in z direction for $i > 0$ we apply PEC boundary conditions (see the remarks below and structure of eq. (5.37) and fig. 5.5) – for $i = 0$ this is the input interface and those boundary conditions are used.*

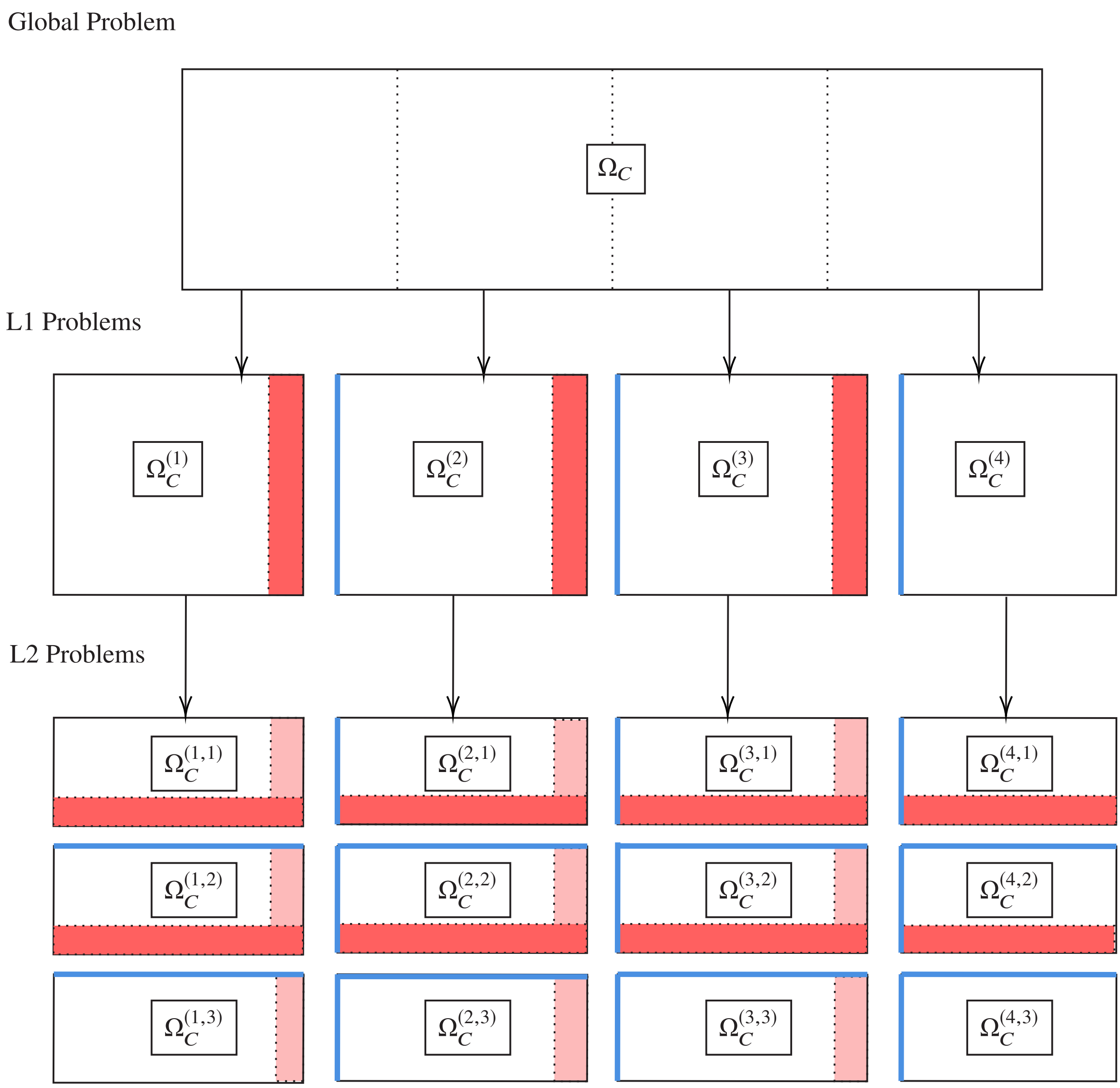


Figure 5.5: Schematic of the computational domain Ω_C split for a 2D sweeping preconditioner. Darker red areas are PML domains introduced on the current sweeping level, brighter red for the previous level. PML domains in the global problem are not highlighted. Blue boundaries depict PEC boundary conditions introduced by the sweeping preconditioner.

For level 2 sweeping, we define the operator S^{ij} as the system matrices for problem 1 constructed on $\Omega_C^{(ij)}$. For the x directions, the boundary conditions remain unchanged. For the upper boundary in y direction we apply an auxiliary PML (as depicted in fig. 5.5) and on the lower boundary in y direction we apply PEC boundary conditions (see the remarks below and structure of eq. (5.37) and fig. 5.5). For the z directions, the boundary conditions are the same as for S^i.

For level 3 sweeping, we define the operator S^{ijk} as the system matrices for problem 1 constructed on $\Omega_C^{(ijk)}$. For the upper boundary in x direction we apply an auxiliary PML and on the lower boundary in x direction we apply PEC boundary conditions (see the remarks below and structure of eq. (5.37) and fig. 5.5). For the y and z directions, the boundary conditions are the same as for S^{ij}.

Degrees of freedom introduced by auxiliary PML layers are always sorted to the end of the system and for the number of auxiliary PML degrees of freedom in the respective system n_a with total size N and the number of global degrees of freedom $n_g = N - n_a$, we define the operators

$$H^i : \mathbb{R}^{n_g} \to \mathbb{R}^{n_g}, \boldsymbol{v} \mapsto (S^i)^{-1} \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{0} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{Id} \\ \boldsymbol{0} \end{pmatrix}, \tag{5.58}$$

$$H^{ij} : \mathbb{R}^{n_g} \to \mathbb{R}^{n_g}, \boldsymbol{v} \mapsto (S^{ij})^{-1} \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{0} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{Id} \\ \boldsymbol{0} \end{pmatrix} \quad \textit{and} \tag{5.59}$$

$$H^{ijk} : \mathbb{R}^{n_g} \to \mathbb{R}^{n_g}, \boldsymbol{v} \mapsto (S^{ijk})^{-1} \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{0} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{Id} \\ \boldsymbol{0} \end{pmatrix} \tag{5.60}$$

with $\boldsymbol{Id} \in \mathbb{R}^{n_g \times n_g}$ and $\boldsymbol{v} \in \mathbb{R}^{n_g}$.

These operators perform the following task: They take a vector of size n_g, extend it by zeroes to the size N (since the auxiliary problem has the additional PML domain), solve problem 1 represented by the application of the inverse of the system matrix $S^{\cdots}$ and extract the first n_g components from the result by multiplying with a rectangular matrix, which is the identity in the first n_g rows and zero in the rest.

Remark. *The operators H^i with $d = L_z$ are the standard operators for the moving PML method. Our formulation, however, is more general, which has already been alluded to in [TEY12], where the choice $d = L_z$ was made for a simplification of the formulation since the PML materials were explicitly listed (see section 2.1 in that publication).*

Algorithm 4 Algorithm for the hierarchical sweeping preconditioner to a vector $\boldsymbol{b}$: Level 1

1: **for** $1 \le i \le n_z$ **do**
2: $\quad u(\mathcal{I}_i) \leftarrow b(\mathcal{I}_i)$
3: **end for**
4:
5: **for** $1 \le i \le n_z - 1$ **do**
6: $\quad u(\mathcal{I}_i) \leftarrow H^i u(\mathcal{I}_i)$
7: $\quad u(\mathcal{I}_{i+1}) \leftarrow u(\mathcal{I}_{i+1}) - A_{i+1,i} u(\mathcal{I}_i)$
8: **end for**
9:
10: $u(\mathcal{I}_{n_z}) \leftarrow H^{n_z} u(\mathcal{I}_{n_z})$
11:
12: **for** $n_z - 1 \ge i \ge 1$ **do**
13: $\quad u(\mathcal{I}_i) \leftarrow u(\mathcal{I}_i) - H^i A_{i,i+1} u(\mathcal{I}_{i+1})$
14: **end for**

For solving the linear systems arising from the application of H^i in lines 6, 10 and 13, we use GMRES with the level 2 preconditioner defined by algorithm 5.

Algorithm 5 Algorithm for the hierarchical sweeping preconditioner to a vector $\boldsymbol{b}$: Level 2 on a given subdomain i

for $1 \le j \le n_y$ **do**
  $u(\mathcal{I}_j) \leftarrow b(\mathcal{I}_j)$
end for

for $1 \le j \le n_y - 1$ **do**
  $u(\mathcal{I}_j) \leftarrow H^{ij} u(\mathcal{I}_j)$
  $u(\mathcal{I}_{j+1}) \leftarrow u(\mathcal{I}_{j+1}) - S^i_{j+1,j} u(\mathcal{I}_j)$
end for

$u(\mathcal{I}_{n_y}) \leftarrow H^{in_y} u(\mathcal{I}_{n_y})$

for $n_y - 1 \ge j \ge 1$ **do**
  $u(\mathcal{I}_j) \leftarrow u(\mathcal{I}_j) - H^{ij} S^i_{j,j+1} u(\mathcal{I}_{j+1})$
end for

If the problems on this level are still too large to be solved directly, we can once again solve them iteratively using GMRES with the level 3 sweeping preconditioner defined by algorithm 6.

Algorithm 6 Algorithm for the hierarchical sweeping preconditioner to a vector $\boldsymbol{b}$: Level 3 on a given subdomain ij

for $1 \le k \le n_x$ **do**
  $u(\mathcal{I}_k) \leftarrow b(\mathcal{I}_k)$
end for

for $1 \le k \le n_x - 1$ **do**
  $u(\mathcal{I}_k) \leftarrow H^{ijk} u(\mathcal{I}_k)$
  $u(\mathcal{I}_{k+1}) \leftarrow u(\mathcal{I}_{k+1}) - S^{ij}_{k+1,k} u(\mathcal{I}_k)$
end for

$u(\mathcal{I}_{n_x}) \leftarrow H^{ijn_x} u(\mathcal{I}_{n_x})$

for $n_x - 1 \ge k \ge 1$ **do**
  $u(\mathcal{I}_k) \leftarrow u(\mathcal{I}_k) - H^{ijk} S^{ij}_{k,k+1} u(\mathcal{I}_{k+1})$
end for

Remark. *It is important to note that the index sets $\mathcal{I}_\xi$ are different on the levels since they only include the degrees of freedom on the respective level plus the auxiliary PML dofs.*

In one of the numeric examples we will show later, we chose $n_x = 3$, $n_y = 3$ and $n_z = 39$ and used a level 3 sweeping preconditioner. To evaluate this scheme efficiently on an HPC system we perform the following steps:

1. For each highest level subdomain we start one process, $3 \cdot 3 \cdot 39 = 351$ in total.
2. Each process assembles its local problem S^{ijk} and computes the LDL^T factorization of $(S^{ijk})^{-1}$.
3. We form groups for all processes that share a sweep in the x direction. Each one of these groups constructs its system S^{ij}. In the example, these groups always have 3 members.
4. We form groups for all processes that share a sweep in the y direction. Each one of these groups constructs its system S^i. In the example, these groups always have $3 \cdot 3 = 9$ members.

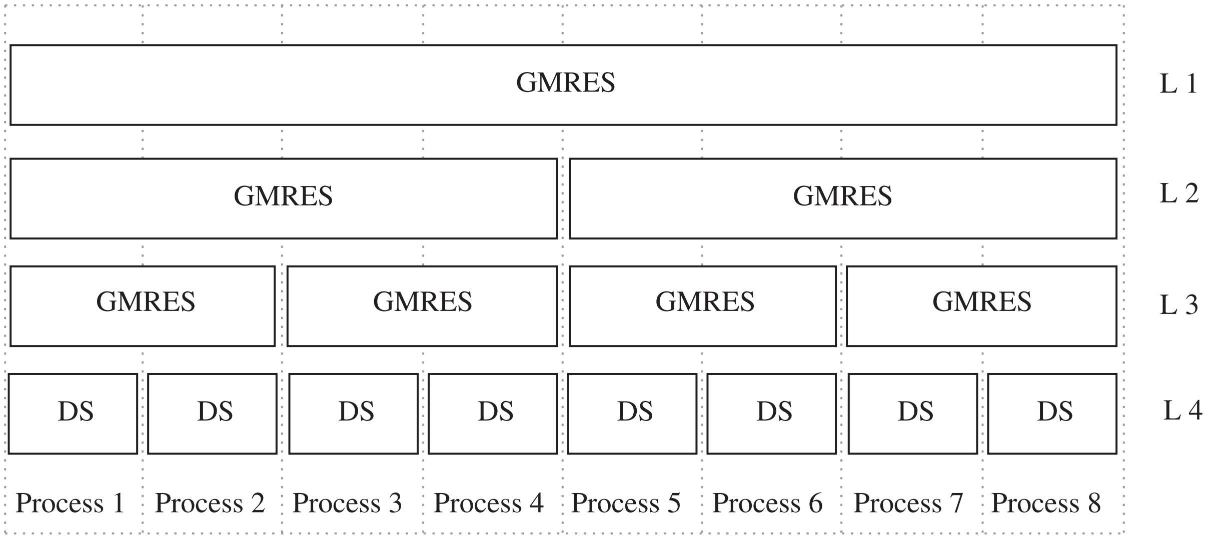


Figure 5.6: Distribution of solvers across processes in a hierarchical sweeping setup.

5. All processes assemble the global system A together.

The smallest possible case for a level 3 sweeping preconditioner is for a total of $8 = 2 \cdot 2 \cdot 2$ subdomains. For this example, fig. 5.6 shows how the solvers are parallelized. From left to right we see a column for each process. On the highest level 4, i.e. locally, each process uses a direct solver. On level 3, we have sweeps in x-direction with 2 processes participating in each. These are used as preconditioners for level 2 sweeps, in which 4 processes contribute to precondition the system. On the lowest level, we have one GMRES solver that is parallelized across all participating processes, solving the global system.

5.4.4 Choice of the Iterative Solver

So far, we have not discussed the choice of an iterative solver in detail. The matrices A, S^i, S^{ij} and S^{ijk} are indefinite so we require a generic solver, i.e. a solver that only assumes the existence of the inverse.

As a study of the options at our disposal, we will be using the generic solvers *GMRES*, *BiCGStab* and *TFQMR*. In addition to the solvers mentioned above, we will also be using a Richardson iteration since it will turn out to be a useful comparison.

Remark. *The cost of adding vectors to the Arnoldi-Basis in GMRES increases for every additional step. It is therefore common to restart GMRES after certain numbers of steps, resetting the Arnoldi-Basis. A typical notation is GMRES*(n) *for GMRES that restarts every* n *steps. We will ignore this detail in this work since the cost of the preconditioner is typically so high in these cases that we would not compute this many steps. The default for* n *in many implementations is* 30*, but sweeping preconditioners should typically converge in less then* 10 *steps.*

Solver	Subdomains	Steps	Avg. Res. Reduction	Time [s]	Number of dofs
GMRES	8	11	0.1717	779	427 520
GMRES	4	11	0.1997	83	226 816
TFQMR	8	14	0.3359	1050	427 520
TFQMR	4	14	0.3012	127	226 816
BICGS	8	7	0.0543	889	427 520
BICGS	4	7	0.0592	121	226 816
RICHARDSON	8	22	0.4488	1438	427 520
RICHARDSON	4	24	0.5333	200	226 816

Table 5.1: Listed results of experiment 4.

Experiment 4. ***Setup:*** *For $i \in \{4, 8\}$ we split the domain $\overline{\Omega_I} = [-2, 2] \times [-1.8, 1.8] \times [0, 2.0 \cdot i]$ into i equal subdomains in the z direction. Each subdomain consists of 16 cells in each direction. We set the mode computed in experiment 1 as a Dirichlet value on the input side ($-z$) and wrap the domain in 8 cells of PML with σ from eq. (4.45) for $k = 1$, $\sigma_{max} = 10$ and $d = 0.5$. We then solve eq. (4.126) discretized by Nédélec elements of order 1 using the algorithm 3 and a variety of iterative solvers: GMRES, TFQMR, BICGS and the Richardson iteration with absolute convergence criterion 10^{-6}. Including the PML domains, we use the computational domains $\overline{\Omega_C} = [-2-d, 2+d] \times [-1.8-d, 1.8+d] \times [0, 2.0 \cdot i + d]$. Each of the subdomains is stored on an individual process and the computation is therefore done in parallel using 4 or 8 parallel processes.*
Results: *In table 5.1 we list the results. For each solver, we have two runs. One for 4 subdomains / processes and one for 8 to show that the results are similar within the parameter region where sweeping is assumed to work well. We list the number of steps until the iterative solver converges or aborts (for all except Richardson we have set a step count limit of 20 performed steps). The next column lists the average of $\frac{r_{i+1}}{r_i}$ for the i-th residual norm r_i. Next, we list the run time of the iterative solver and the number of degrees of freedom involved used in the global problem.*

Our first remark is that for CG the method does not converge. This is expected because the system is indefinite. It is an important remark, however, because while A is not positive definite, this might still be the case for the preconditioned system. Consider the case in which the preconditioner P is exact, meaning $P = A^{-1}$. In that case, $P\,A = \boldsymbol{Id}$ is positive definite and symmetric. We do observe, however, that the preconditioned system still has these properties.

The remaining solver options yield low numbers of applications and compared to the application of the preconditioner, they have low numerical cost. As a consequence, since the number of steps until convergence are lower for TFQMR and BiCGStab than for GMRES, we would assume that those two solvers are better suited to our application. This, however is incorrect. Both TFQMR and BiCGStab require two matrix-vector multiplications per step, and, as a consequence, two applications of the preconditioner. We see therefore, that all the schemes require very similar numbers of applications of the preconditioner. This is why we added the Richardson iteration to this list.

	Level 4	Level 3	Level 2	Level 1
Number of Dofs	155 760	308 368	610 480	19 678 880
Number of PML Dofs	121 896	209 448	321 567	8 590 376
Ratio non-PML/PML	21%	32%	47%	56%
Number of solver calls	216 960	14 695	770	1
Solver	direct	GMRES_x	GMRES_y	GMRES_z

Table 5.2: Details about the hierarchy levels. $\text{GMRES}_\#$ refers to a GMRES solver using a sweeping preconditioner in # direction.

Definition 5.4.3 (Richardson Iteration)**.** *For the equation* $A\boldsymbol{x} = \boldsymbol{b}$ *with* $A, P^{-1} \in \mathbb{GL}_n$ *and* $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$*, we call the iteration*

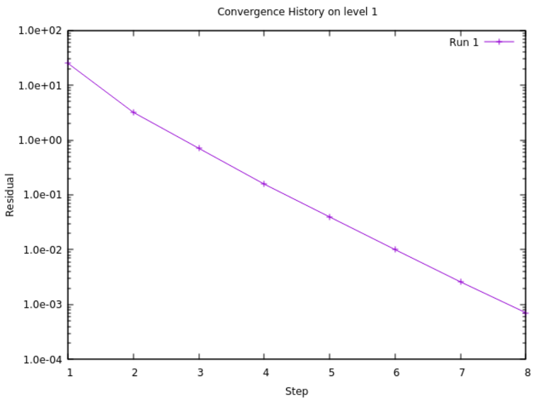
$$\boldsymbol{x}^{(i+1)} = P^{-1}\boldsymbol{b} - P^{-1}A\boldsymbol{x}^{(i)} + \boldsymbol{x}^{(i)} \tag{5.61}$$

for $\boldsymbol{x}^{(0)} = \boldsymbol{0}$ *the preconditioned Richardson Iteration and* P^{-1} *the preconditioner.*

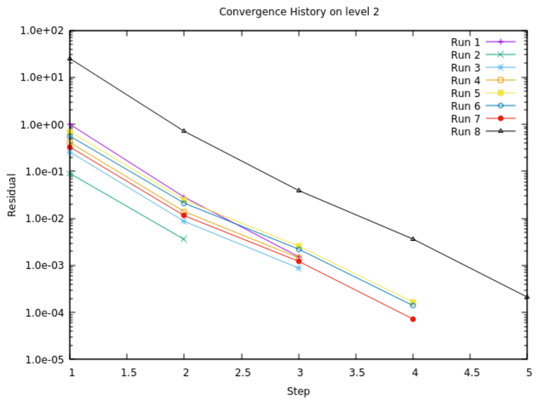
In this iteration, we perform one matrix-vector product per iteration, that we can precondition by applying the approximate inverse. Compared to the Krylov-methods, this method does not create a vector space in which the solution has minimal residual, so the convergence is solely dependent on the repeated application of the preconditioner. We can therefore assume that the Richardson iteration should serve as a lower limit for other solvers since it is cheap to apply. If a solver requires more applications of the preconditioner than the Richardson iteration, the actual solver does not contribute positively to the convergence of the method.

5.4.5 Numeric Results

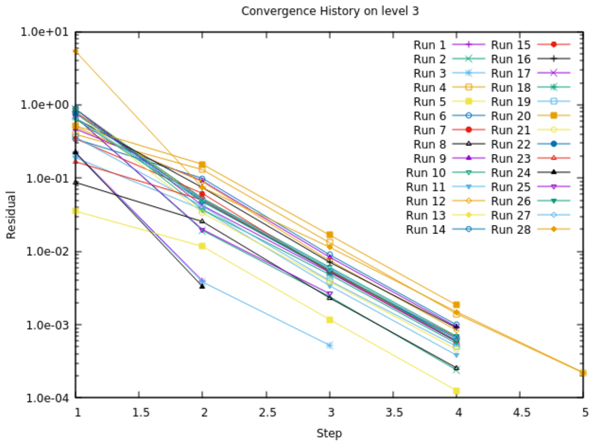
Experiment 5 (Waveguide with vertical displacement)**.** ***Setup:*** *We use a hierarchical sweeping preconditioner with all three levels to compute a mode (computed by section 4.5.1) propagating in a waveguide of width* $2\mu m$ *and height* 1.8μ *of a predefined shape. We use Dirichlet input data and Nédélec elements of lowest order. The global problem consists of* $48 \times 48 \times 1560$ *cells and we set* $n_x = n_y = 3$ *and* $n_z = 39$ *leading to* $3 \times 3 \times 39 = 351$ *parallel processes. The entire domain is wrapped in* 10 *cell layers of PML medium of thickness* $0.5\mu m$ *with the scaling exponent* $k = 3$ *and* $\sigma_{max} = 20$*. As a solver we use GMRES and the algorithms 4 to 6 as preconditioners. We use the computational domain* $\overline{\Omega_C} = [-3, 3] \times [-2.7, 2.7] \times [0, 100.5]$.
Results: *The system has a total of* 19 678 880 *degrees of freedom in the global problem. Solving the problem took* 39.5 *hours and required* 933.72 *GB of Memory on the BWUniCluster 2.0. (JobId* 20506636*). The numbers of dofs by level and type are listed in table 5.2. For convergence rate of the solvers see fig. 5.7 and for plots of solutions and relevant data see figs. 5.9 to 5.11 and 6.2.*



(a) Convergence of the GMRES run on level 1.



(b) Convergence of the GMRES runs on level 2.



(c) Convergence of the GMRES runs on level 3.

Figure 5.7: Comparison of the convergence rates on levels 1, 2 and 3

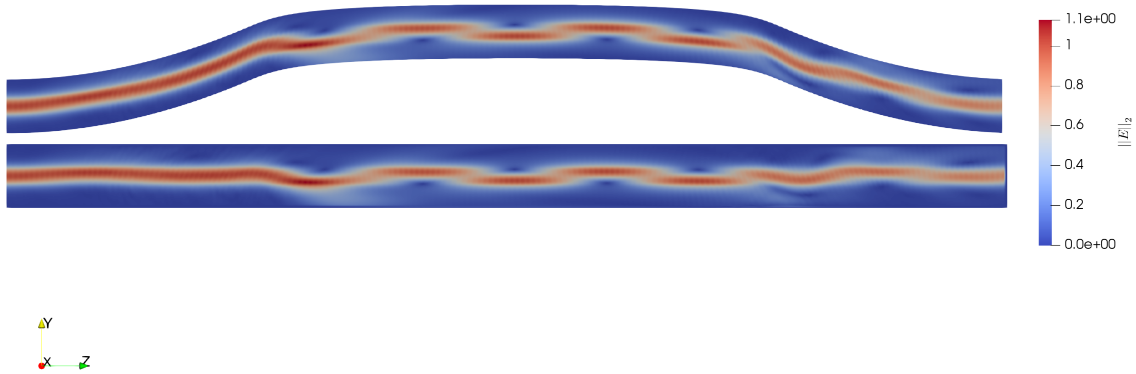


Figure 5.8: Euclidian norm of the E-field in experiment 5 along the central waveguide axis in the physical (top) and mathematical coordinate system (bottom).

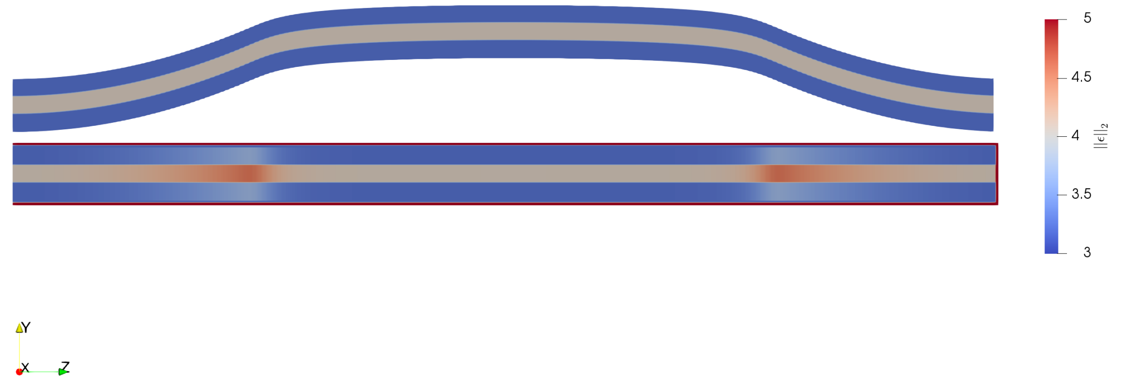


Figure 5.9: Norm of ϵ for experiment 5 of the transformed (top) and non-transformed system (bottom). Important: The transformed system also includes the PML which causes the red line around the computational domain.

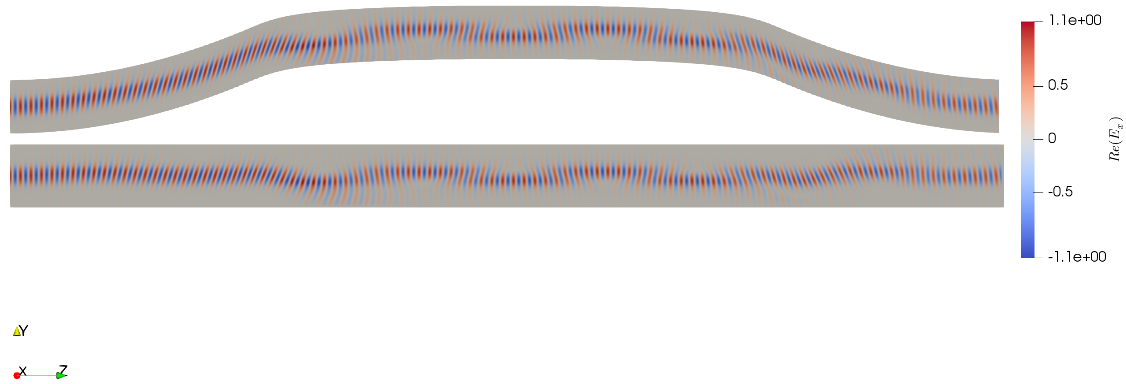


Figure 5.10: Real part of the solution of experiment 5 for both the transformed (top) and non-transformed coordinate system (bottom).

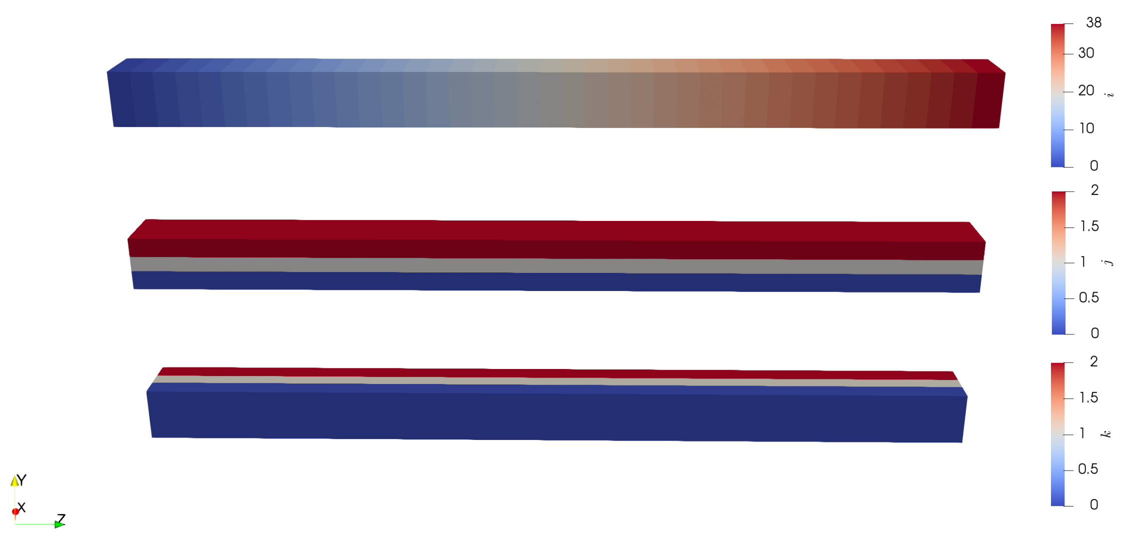


Figure 5.11: Subdomain association of the computational domain listing visualizing the indices i (top), j (middle) and k (bottom).

5.4.6 Convergence of the Method

In essence, this method only relies on the properties of a 1D sweeping preconditioner and convergence criteria are therefore similar. Sweeping preconditioners rely on 2 premises for convergence:

1. The individual problems on the subdomains are well-posed meaning a solution exists in the function space we are searching in.
2. We can solve the problems on the subdomain.

For the first point, it is important to point out a conceptional difference between multigrid-type methods and sweeping preconditioners: sweeping preconditioners perform no coarsening or refinement between the global problem and the problems on the subdomain. The subdomain problem triangulation is a subset of the global triangulation with possibly some additional PML domain. Therefore, if a condition on the mesh constant h holds on any one of the levels, it holds on all of them.

For the question of whether the second condition can be fulfilled, we rely on the fact that 1D sweeping preconditioners converge, since on every successive level we perform a 1D sweep. We continue doing this until the individual problems are small enough to be solved directly. It is important to note, however, that this does not scale infinitely since the number of solver applications scales exponentially with the sweeping level.

We have found that full convergence to the target absolute convergence criterion is not required for the y and x sweeps. In our numerical experiments a relative convergence criterion of 10^{-2} against the residual of the higher level solvers current residual yielded similar results. It is not required to compute an extremely accurate solution on these levels since the vector that is passed in from the method on the level above still has a larger error and it is more important to compute the sweeps quickly then to evaluate them at maximum precision.

The relative convergence criterion 10^{-2} delivered basically unaffected convergence rates in test runs and is therefore set as default in our computations. This can also be observed in the convergence plots in fig. 5.7 where on levels 1 and 2 the computations are aborted for larger residuals for the earlier runs.

5.4.7 Runtime Prediction

We make the following assumptions:

- The time to apply the direct solvers t_s is the same on every process because the system size is the same and geometries are similar.
- Communication is cheap.
- Storing a solver context and loading it takes t_l seconds.
- The time required to perform a GMRES step based on the number of degrees of freedom is known as a function of degrees of freedom per process and a number of processes.
- An estimate of the number of GMRES steps required to reach the convergence cutoff is available.

We can then estimate both CPU-time and wall time based on different load distributions. The space for optimization arises from the fact that only during applications of GMRES for the global system all processes solve at the same time. While a load-distribution will always increase the wall time (i.e. the time it takes for the program to complete) one expects that the increase in wall time can be negligible in some settings while on the other hand saving large parts of the resources. Using fewer processes of a cluster reduces the cost. These costs can be assumed to be linear in the number of nodes and the wall time of the application. Therefore, an increase of the wall time by 10% with a reduction of resources to half would lead to a 45% decrease in real-world cost of the simulation.

To perform this load balancing we would have to predict the number of steps GMRES for a given sweep will likely require to converge and an estimate on how long a single application of a local operator as well as the computation of a GMRES step takes. All these parameters can be predicted based on the number of subdomains, the direct solver in use and experience from previous runs. This, however, would primarily make sense in a large scale industrial application to minimize cost or to maximize throughput on a given HPC system.

As mentioned in section 5.4.2, there are two kinds of operations that take most of the time during the application of the method:

- Application of GMRES to a system on the sweeping levels. These are performed in parallel on all processes that are part of the current sweep.
- Application of a direct solver on the lowest level. These are performed on only one process.

Considering the fact that applying a direct solver is time consuming, it makes sense to find room for performance improvements in these blocking segments of the algorithm.

The first step we can take is to observe that the system matrices do not change during the application of the algorithm. Direct solvers typically compute an LU factorization of the system matrix and as long as that system matrix does not change, the factorization remains valid. We can make use of this property and only compute the factorization once. Additionally, because the system matrices on the local problems do not depend on each other, we can also compute the factorizations independently. We can therefore preface the hierarchical sweeping preconditioner with a factorization step for the systems on the local level that is perfectly parallelizable. In numerical experiments we found the runtime of the factorization compared with the application of the pre-computed factorization to be roughly 100 times longer for the type of system matrices that occur in this setting.

Solver implementations often check if the system matrix has been changed from the last time the solver was called and only perform a factorization if such a change has occurred. This is often referred to as caching. As a consequence, only the first solver applications in the scheme would be 100 times slower than any successive ones. The algorithm calls the local level solvers sequentially, however, so all these factorizations would be computed sequentially, too. Such an implementation would be highly inefficient on a HPC system.

If the direct solver used in the scheme does not provide factorization as a separate function call, the easy way to circumvent sequential processing would be to apply all direct solvers to a random right-hand side in parallel ($\mathbf{0}$ for example) before the first application of the preconditioner. Since this effort is perfectly parallelizable, the computation of all the factorizations only takes as long as computing one factorization (in walltime). If this step is skipped, the computation will occur the first time the factorization is required in the preconditioner, which occurs sequentially for all processes and thus the computation of the factorizations would occur sequentially.

Once this issue has been addressed, using the local solvers comes down to the application of the factorized version of the inverse of the system matrix to a vector. It makes sense to make use of all numerical libraries on any available HPC system to run this preconditioner due to the number of basic matrix-vector and vector-vector operations. For example, using the Intel Math Kernel Library for our implementation on the HPC system that was used for the numerical results, yielded a performance improvement of $\approx$ 20%. This is not surprising, because most of the runtime is spent on either the direct solvers or GMRES, which are both library implementations, that make use of the most efficient Basic Linear Algebra Subprograms (BLAS see [Bla+02]) at their disposal (like ATLAS [WD99] or Intel MKL [09]).

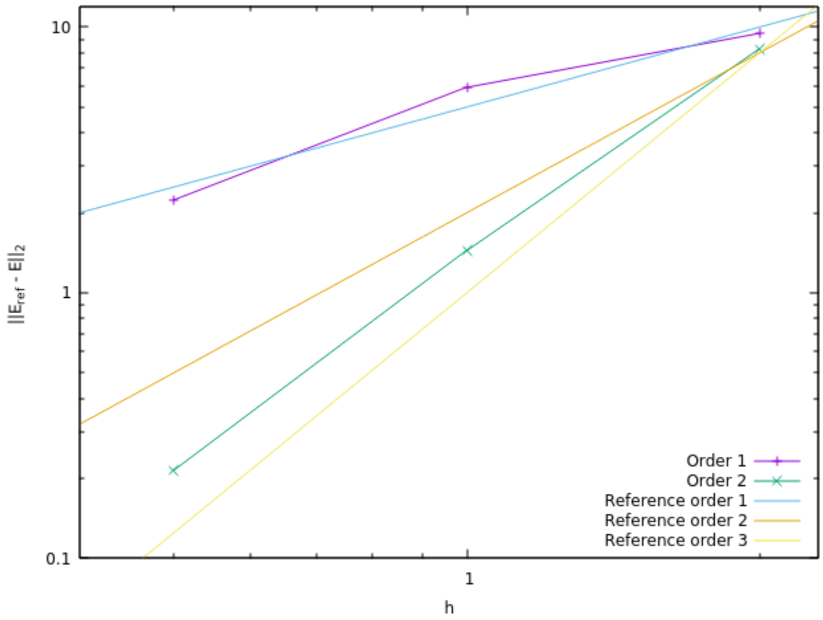


Figure 5.12: Convergence plot of the numerical error in experiment 6

However, the fact that we apply direct solvers on the lowest level offers room for performance improvement. The application of the direct solver consists of matrix-vector products, which are a perfectly parallelizable action in themselves. At the time of writing this document, the GPU HPC library CUDA by Nvidia does not provide a direct solver for indefinite, complex-valued matrices. However, if such a solver becomes available in the future, running the direct solvers on a GPU could yield considerable performance improvements. In absence of such a solver, it could also be considered, if available, to extract the factorization matrices from the direct solver, copy them to a GPU and apply them there. The CPUs computing the GMRES steps and the GPUs applying the lowest level solvers would only have to exchange the factorization before the method starts and the right-hand side vectors during the lowest-level sweep. Depending on the memory available on a GPU and the sizes of systems involved, the GPUs could also handle the local solver for multiple CPU processes that are neighbors on the lowest sweeping level, thus removing the requirement for communication via the CPU (GPU1 -> CPU1 -> CPU2 -> GPU2).

5.5 Validation of the Method

Experiment 6 (Convergence Analysis)**.** ***Setup:*** *We ran a series of convergence studies for a straight waveguide of length* $2.5\mu m$ *with an input signal computed by experiment 1 in the formulation of problem 1, i.e. as Dirichlet values. We truncated the domain by 6 cell layers of a PML medium with* $\sigma_{max} = 10$ *and* $k = 3$*. We used* 2×2 *processes for a 2 level sweeping preconditioner. The mesh consisted of* $n \times n \times n$ *cells for* $n \in \{2, 4, 8, 16\}$ *and we used the solution for* $n = 16$ *(*$h = 0.25\mu m$*) as a reference for a convergence study. We used Nédélec elements of order* $o = 1$ *and* $o = 2$ *to check if the expected improved convergence rate for higher element orders can be validated. According to [Mon92, Theorem 5.41] we expect to see the* L^2 *errors develop equal or better than* Ch^o*.*
Results: *The improved convergence rate of higher order elements is clear (see fig. 5.12) and details about the runs are listed in table 5.3.*

-	Order 1				Order 2			
h	$\|\|E_{\text{ref}} - E\|\|_2$	dim V_h	Steps	Time[s]	$\|\|E_{\text{ref}} - E\|\|_2$	dim V_h	Steps	Time
2	9.477486	7808	16	76	8.296882	58032	15	304
1	5.925429	15138	6	26	1.440584	114212	5	94
0.5	2.230410	40262	5	29	0.213845	308748	4	162
0.25	-	148494	4	101	-	1155548	4	896

Table 5.3: Results of experiment 6. E_{ref} is the solution for the finest mesh $h = 0.25$

5.6 Sweeping Preconditioners with HSIE

As stated in [TEY12], using PML for the truncation of the computational domain is not required – it can be replaced by other truncation techniques. We described in section 5.3.2 that there are several issues with PML, such as the large number of parameters that require tuning as well as the high number of degrees of freedom they introduce into the system. Since the scheme of hierarchical sweeping does not require the use of PML specifically as an absorbing boundary condition, we are free to explore alternative options.

The next steps are as follows: We build a mesh of the exterior domain by defining a set of semi-infinite cells, that extend from the surface of the domain of interest to infinity. We will then transform the cell to a cartesian frustrum as a reference cell for a finite element and adapt the definition of a Nédélec-element, where we express the vector-components associated with the external coordinate by polynomials in $H^+(S^1)$. Derivatives and integrals of the external direction can be reformulated as operations on monomials in $H^+(S^1)$ and thus represented as matrix operations for vector representations of polynomials.

Remark. *At this point, we can already see the core advantage of HSIE over PML. We have derived the basic principles, but the only decisions we need to make in the application of the method are the*

- *choice of* κ_0*,*
- *construction or choice of an external direction*
- *choice of a maximum polynomial degree of the Hardy space polynomials.*

An appropriate value of κ_0 *can be determined by experiments but some hints are given in the relevant literature. The choice of the external direction is influenced by the important factors, that, on the one hand, the cells must be non-degenerate, i.e. the rays from the surface of the domain of interest in the infinite direction may not intersect. Furthermore, the material property* ϵ *has to be constant on each external cell. These two properties heavily restrict the possible choices for exterior triangulations.*

Lastly, we need to choose a maximum polynomial degree that has an effect similar to an increase of the number of cell-layers for a PML surface, since it also increases the number of degrees of freedom while increasing the precision of the approximation of the solution in the exterior domain.

Next we will assume a simple setup to visualize the issues arising from the application of HSIE in a sweeping preconditioner: Let the domain of interest Ω_I be the unit cube $[0,1]^3 \subset \mathbb{R}^3$ and $\epsilon(x) = 1$ on all of $\mathbb{R}^3 \setminus \Omega_I$. In this setup, we can employ HSIE to truncate the domain of interest as stated above. First, we need to choose a way to pick the infinite direction for each vertex on the surface so we can construct the semi-infinite rays, which will be the edges of the Hardy space infinite elements.

Two methods come to mind:

- to pick a point p_0 inside Ω_I and to use the ray $(\alpha + 1)\boldsymbol{x} - p_0$ for $\alpha \geq 0$ or

- to use axis-parallel rays. Since we have chosen a cube to be the domain of interest, we can always pick an orthogonal direction on the surfaces.

Both methods, however, run into a similar issue. The choice of the external direction has to be the same across sweeping hierarchy levels. This is due to the fact that we copy the solution components from lower to higher and from higher to lower sweeping level. If the infinite directions are not consistent across sweeping levels, this will introduce error terms and thus reduce the quality of the preconditioner.

Figure 5.13 visualizes the issues arising across sweeping levels that prevent us from using these elements in this setup. In the first row, we see a setup in which the infinite direction is chosen to be axis parallel. This leads to a straight forward implementation of the HSIE since the derivative of the mapping from the infinite cell to the reference element is zero. However, for neighboring sides of the inner domain that both are treated using HSIE, the outermost degrees of freedom need to be coupled since the system would be underdetermined otherwise. This case is not currently treated in publications on the topic since the default formulation of the method breaks down in this case: The method decomposes the integral over the entire cell into a surface integral and an integral over $(0, \infty)$ for the infinite direction. On the corner domain (highlighted in red in fig. 5.13), however, the surface triangulation is the intersection of the two surfaces of the inner domain – an edge of the domain and therefore 1D. Considering this same problem in 3D for three neighboring surfaces that share a corner, the situation is even more extreme, in that the shared surface triangulation is only the corner point of the inner domain. The basic mechanism behind HSIE therefore breaks down in these cases.

In the rows two and three of fig. 5.13 two other choices of infinite direction are visualized, which also run into a problem: While these choices work for one single domain, as soon as a neighbor is added, the infinite cells would overlap. For such meshes, the derivation of the method degenerates.

Remark. *While these details have not been further examined by the author, there appear to be ways around these restrictions. Firstly, it may be possible to construct an edge and corner term for the case in which cartesian infinite directions are used. This, however, extends beyond the scope of this work. Secondly, there are alternative formulations of infinite elements, such as complex scaled elements (see [NW22] and [Wes20]) which are still under development and have not yet been extended to Maxwell's equations, which might prove to make the handling of such situations easier in the future.*

5.7 Central Innovation of this Work

Let us again consider experiment 5 of a waveguide of $\approx 100\mu m$ length. When we solved this problem using a hierarchical sweeping preconditioner, we split the domain in $3 \times 3 \times 39$ subdomains, each containing $16 \times 16 \times 60$ cells. If we wanted to solve the same system using a 1D sweeping preconditioner, what would the partitioning look like?

Including the PML layers, the global system consists of $68 \times 68 \times 1580$ cells. For $i \times j \times k$ cells, we get a system of

$$N = (i+1)jk \; + \; i(j+1)k \; + \; ij(k+1) \tag{5.62}$$

degrees of freedom. If we further assume, that we can only solve a system for 200 000 degrees of freedom per process, we find the restriction on k

$$200\,000 \geq 4692k + 4692k + 4624k = 14008k. \tag{5.63}$$

And therefore

$$k \leq 14. \tag{5.64}$$

Therefore our sweeping problems would be $68 \times 68 \times 14$ cells large. Consider additionally, that the subproblems have a PML in the sweeping direction and that for the PML layers we use 10 cell layers, then

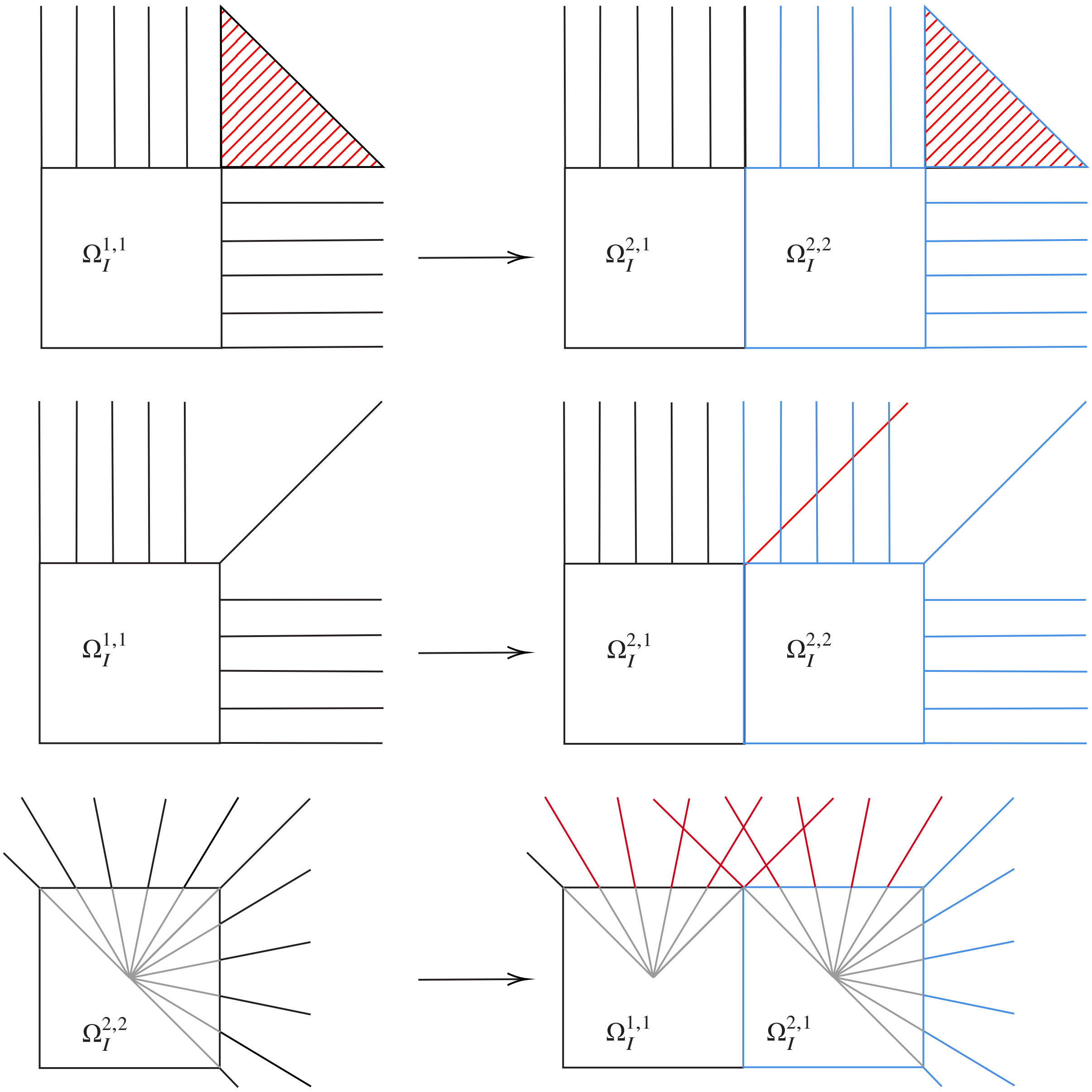


Figure 5.13: Different choices of infinite directions and their respective problems in a sweeping setting.

the actual inner problem we are solving on the subdomain can only be 4 cells wide. We would therefore require $\frac{1560}{4} = 390$ processes. We have seen in experiment 2 that the sweeping preconditioner deteriorates as the number of subdomains increases. Not only does this scale badly, but there is also another problem: The waveguide examples lend themselves naturally to sweeping because they can usually be decomposed in the propagation direction and the orthogonal directions where the propagation direction is the canonical sweeping direction. What happens if we regard a cube-shaped computational domain? Let us assume we need $100 \times 100 \times 100$ cells and once again we use 10 for PML. The constraint introduced above then evaluates to

$$200\,000 \geq 10100k + 10100k + 10000k = 30200k \tag{5.65}$$

For this problem we could only solve six cell layers as a subdomain problem which is less then even the PML would require. We cannot construct a 1D sweeping preconditioner for this problem.

With the hierarchical sweeping preconditioner, on the other hand, we can split this problem into $5 \times 5 \times 5$ subdomains consisting of $20 \times 20 \times 20$ cells each (plus PML). This partitioning leads to $\approx 196\,000$ degrees of freedom. For 5 processes per sweeping direction we can also expect to see fast convergence on every level so we still have a lot of room to increase the number of subdomains if required.

5.8 Implementation

The interior domains (domain of interest) are implemented in the class InnerDomain, the PML used for domain truncation in PMLSurface. Every process receives a part of the global geometry and constructs all level problems for that domain. To this end, there are two main classes: The LocalProblem class uses an InnerDomain and PMLSurfaces to build the lowest level problem. It then applies a direct solver to it. On higher levels, the NonLocalProblem class is used, which can also have NeighborSurface boundaries to construct fem systems that are not completely locally owned. If it is called upon to solve the system, GMRES is used. In these GMRES solvers, either the LocalProblem or a lower-level NonLocalProblem is used as a preconditioner.

6 Electromagnetic Design

In this chapter we will focus on the topic of numerical optimization. We will begin by introducing basic theory of numeric optimization in general and shape optimization specifically in section 6.1. This part will introduce the concept of a shape gradient, which we will discuss in detail in section 6.1.2. We will discuss the two main conceptual perspectives on shape optimization (**Material Optimization** and **Shape Optimization**) and introduce a hybrid method, that utilizes transformation optics and adjoint-based shape-gradient computation.

Once a method of computing a shape-gradient is available, we have to choose an optimization algorithm, which, in this work, is mainly a matter of picking the algorithm, that requires the least evaluation of a shape gradient for our application.

Finally, we will present numerical examples in section 6.3. It is important to note, however, that the method described in this work can be used for a much wider variety of applications than the examples we present here. In section 6.4 we will discuss further possible applications.

Initially, we will introduce two common ways of performing shape optimization: Material derivative based methods and domain derivative based methods.

6.0.1 Material Derivatives

The ansatz of using material derivatives is based on computing the derivative of the loss functional in the direction of a finite non-infinitesimal material change. In a 2D-problem formulation we use a grid to split the computational domain cells. Each cell can either be part of the waveguide core or cladding. If the cell is part of the waveguide, $\epsilon_r = \epsilon_{in}$. Else $\epsilon_r = \epsilon_{out}$.

The computation of $\frac{\partial \mathcal{L}}{\partial x_i}$ – the derivative of the loss-functional in direction of a material change – can then be computed as $(E, \delta\epsilon_d E_0)$ where $\epsilon_d = \epsilon_{out} - \epsilon_{in}$ is a constant material tensor difference across the entire domain. In an update step we include all cells, for which the derivative is negative and exclude all those for which it is positive. We can repeat these steps and change the subdomain grid.

Advantages

This scheme is easy to implement. For a given shape, we compute E and E_0. Then we generate a set of cells to consider for inclusion or exclusion from the waveguide interior. For each cell we compute the derivative of the loss-functional and then select the cells that reduce losses. These steps are repeated until a condition for termination is reached.

Disadvantages

This scheme includes no method for regularization in the sense that arbitrarily complex solutions can be generated because the optimization is ultimately performed in the set of all possible splits of the domain into two subsets. Most of these possible states, however, could not be manufactured (see for example [Ott17]). It is a common result for these kinds of optimization results to include islands of material which are not physically stable in 3D.

To restrict the optimization to geometries that can be manufactured a regularization has to be performed, i.e. a projection to a space of reduced dimension that only contains solutions that can be manufactured. It can prove difficult to show that the quality of the regularized shape is close to the quality of the shape before regularization. While the ability to find non-connected optimal solutions can be advantageous in some situations, it is not required for 3D free-form waveguides and the necessary restrictions could be complex.

6.0.2 Domain Derivative

When optimizing the shape of an object that has an interior and exterior, we can go another way. We regard the surface as the relevant domain and try to determine if the surface should be bent outward or inward to encompass more or less material. This leads to a functional defined on the surface of the shape, which maps the E-field and the adjoint state onto a value indicating to either extend or retract the surface at any given surface location.

This framework, by operating only on the surface, can make regularization easier and provide a formulation that builds on infinitesimal change rather than finite change as in the previous case of material optimization.

Advantages

As stated above, the advantages are fairly obvious. The functional evaluation on the surface makes it possible to only compute the gradient for directions that are relevant (on the surface we can make this judgment) and then perform optimization. Another advantage of this scheme is the restriction of the relevant functional to the surface of the domain rather than the volume, which reduces the dimension and hence the computational cost.

Disadvantages

Some problems remain in this methodology. Formulating the functionals that have to be evaluated on the surface to compute the shape gradients is specific to the shape we regard and therefore it cannot be easily done as an automatic method. Furthermore, we have to cover edge-cases in which the topology of the optimized shape changes. If, for example, the geometry consists of two parallel waveguides and we change their geometry, we need to consider the case that their surfaces come into contact, changing the surface of the waveguide.

6.0.3 Transformation-based Optimization

The solution proposed in this work combines the ease of implementation of material derivatives with the implied regularization of shape optimization based approaches. For the domain of interest, we begin by defining a basic concept.

Electromagnetic design usually deals with the optimization of the shape of a discontinuous material distribution, such as the core of a waveguide or the position of a reflective surface. For this method, we will introduce a so-called **fundamental geometry**, that contains all the material discontinuities, such that every variant of the shape can be achieved by transforming the fundamental geometry with a coordinate transformation, yielding an additional material tensor, which is continuous with respect to the value of the shape parameters.

Definition 6.0.1 (Parameterizations)**.** *For a space of viable geometries*

$$G = \{\Omega_W \subseteq \mathbb{R}^3 : \Omega_W \textit{ is a valid shape}\}, \tag{6.1}$$

*and the parameter space $P = \mathbb{R}^N$ we call $\mathcal{P} \in C(G, P)$ a **parameterization** (of G) if $\mathcal{P}^{-1}$ exists. For a given shape $g \in G$ we call*

$$(p)_i := (\mathcal{P}(g))_i, \quad 1 \leq i \leq N \tag{6.2}$$

*its **parameter values** or p the parameter vector.*

We will be using the shorthand $\Omega_p := \mathcal{P}^{-1}(p)$ for $p \in P$. The parameterization maps a shape of a waveguide onto its parameter vector and the inverse does the opposite.

Definition 6.0.2 (Fundamental geometry)**.** *We call Ω_F a **fundamental geometry** if there exists a mapping $\Phi : G \to C_2(\mathbb{R}^3, \mathbb{R}^3)$ such that*

$$\Phi(\Omega_p)(\Omega_p) = \Omega_F \tag{6.3}$$

for every $p \in P$ and Φ is continuous w.r.t

$$\langle g_1, g_2 \rangle_G = \int_{\Omega_I} |\mathbb{1}_{g_1} - \mathbb{1}_{g_2}| \,\mathrm{d}V \quad \forall g_1, g_2 \in G. \tag{6.4}$$

In simple terms: For every parameter vector $p \in P$, $\Phi(\Omega_p)$ is a coordinate transformation that maps Ω_p onto Ω_F. Additionally, small changes of parameter values cause small changes of the geometry.

Remark. *We introduce fundamental geometries because we will only implement our scheme on the fundamental geometry. It is similar to the reference cell in finite element methods in that we perform our computations on this domain rather than the actual geometry of the waveguide. We employ transformation optics to transform the real waveguide geometry onto the fundamental geometry, compute the solutions of forward problems and the adjoint state on the transformed domain and once we have found an optimal geometry, we can transform it back into the physical coordinate system.*

6.1 Shape Optimization

The reason to investigate the possibility to efficiently compute solutions of Maxwell's equations in this work is to enable shape optimization. Optimization schemes in numerical applications are built around equations like

$$\widetilde{p} = \arg\min_{p \in P} L(\boldsymbol{u}) \tag{6.5}$$

where $\boldsymbol{u}$ is the solution of a pde involving p. There are many introductions to this topic such as [Wal14].

In our case, we have the following specific details: The vector p describes a shape of a waveguide connecting two ports. The goal of the method is to find a waveguide shape such that the signal transmission from the input to the output is in some sense maximized (we will define this in more detail later). Since we formulated the optimization problem as a minimization, we will define $L(\boldsymbol{u})$ as a *loss* functional, since maximization of signal transmission is equivalent to minimization of signal losses.

As a first step, we need to specify the terms in eq. (6.5). The manufacturing process of a waveguide places constraints on the shapes we consider to be viable. The waveguide needs to be a connected shape to be physically stable and we do not allow shapes that extend out of the domain of interest. The waveguide shape is therefore a closed subset of $\mathbb{R} \times \mathbb{R} \times [z_{in}, z_{out}]$. For a given material configuration p we can compute $E_p(\boldsymbol{x})$ by numerical means as has been discussed in chapter 5 (see experiment 5). To state the process in detail, we pick our computational domain and fundamental geometry. For rectangular waveguides, this fundamental geometry is an axis-parallel cuboid and the coordinate transformation $\Phi(\Omega_W)$ maps the waveguide geometry in physical coordinates Ω_W onto the fundamental geometry. We then use this transformation to compute the material tensors ϵ_p and μ_p. We can then solve problem 1 (see definition 5.2.1 using Nédélec elements on the computational domain and PML for spatial truncation.

Additionally, we can use experiment 1 to compute the incident field E_I and compute the right-hand term $\boldsymbol{F}$ via eq. (4.121).

The question we will be discussing in the following is how a given figure of merit of the signal changes if we change the parameterization, i.e. the shape.

We begin by defining our figure of merit – the loss functional

$$L : H(\mathrm{curl}, \Omega_C) \to \mathbb{R}, \ \boldsymbol{u} \mapsto -\left| \int_{\Gamma_{\mathrm{out}}} \boldsymbol{F}_0 \cdot \overline{\boldsymbol{u}} \, \mathrm{d}A \right|^2 . \tag{6.6}$$

Other choices of the loss functional are possible to tailor the optimization towards certain applications but this version is reasonably generic and sensible. This loss functional measures the excitation of the mode $\boldsymbol{F}_0$ on the output interface. Some alternative choices of loss functionals are introduced in [Sem+15], which this introduction is in part based on.

Backward reflections are expected when performing shape optimization so we should not use the system definition 5.2.2 but instead opt for definition 5.2.1 (tapered signal coupling) to allow for the backward propagation of reflections. For a waveguide shape parametrized by p we recall a_T and r_T and recall that for a solution $\boldsymbol{E}_p$ with the materials $\boldsymbol{\epsilon} = \epsilon_p$ and $\boldsymbol{\mu} = \mu_p$ of eq. (4.125) it holds

$$\underbrace{\langle \mu_p^{-1} \nabla \times \boldsymbol{E}_p, \nabla \times \boldsymbol{v} \rangle - \langle \epsilon_p \omega^2 \boldsymbol{E}_p, \boldsymbol{v} \rangle}_{= a_T(\boldsymbol{E}_p, \boldsymbol{v})} - \underbrace{\langle \boldsymbol{F}, \boldsymbol{v} \rangle}_{= r_T(\boldsymbol{v})} = 0 \quad \forall \boldsymbol{v} \in H(\mathrm{curl}, \Omega_C). \tag{6.7}$$

6.1.1 Finite Differences

Since we are able to solve the forward problem, i.e. to compute the E-field for a given geometry, we can also compute finite differences for these parameters. This would involve solving the forward problem eq. (6.7) for a parameter vector p, to then evaluate the loss functional L for the solution and to repeat these two steps for the parameter vector $p + h\boldsymbol{e}_i$ where $\boldsymbol{e}_i$ is the i-th unit vector and $h > 0$ is a reasonably small step width. We would then subtract the values of the loss functional for these two solutions and divide by h. The problem with this method arises from the pure numerical cost. Each component of the shape gradient requires the solution of a forward problem. For shape optimization applications it can be important to use a large set of shape parameters and in that situation, it is not feasible to compute the shape gradient via finite differences. Instead, we will focus on a method to compute the shape gradient at the cost of one additional solution of a forward problem that is identical to the original problem in cost.

6.1.2 The Adjoint Method

We define $\boldsymbol{\mu}_0 = \boldsymbol{\mu}^{-1}$ for convenience and the Lagrange function $\mathcal{L}$ as follows

$$\mathcal{L}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{u}, \boldsymbol{v}) = L(\boldsymbol{u}) - \langle \boldsymbol{\mu}_0 \nabla \times \boldsymbol{u}, \nabla \times \boldsymbol{v} \rangle + \langle \boldsymbol{\epsilon} \omega^2 \boldsymbol{u}, \boldsymbol{v} \rangle - \langle \boldsymbol{F}, v \rangle. \tag{6.8}$$

For an optimal shape parametrization P all partial derivatives of this function have to be zero. We compute

$$0 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{u}}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{u}, \boldsymbol{v}; \boldsymbol{\phi}) = \frac{\partial L}{\partial \boldsymbol{u}}(\boldsymbol{u}; \boldsymbol{\phi}) - \langle \boldsymbol{\mu}_0 \nabla \times \boldsymbol{\phi}, \nabla \times \boldsymbol{v} \rangle + \langle \boldsymbol{\epsilon} \omega^2 \boldsymbol{\phi}, \boldsymbol{v} \rangle \tag{6.9}$$

and

$$0 = \frac{\partial \mathcal{L}}{\partial \boldsymbol{v}}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{u}, \boldsymbol{v}; \boldsymbol{\psi}) = -\langle \boldsymbol{\mu}_0 \nabla \times \boldsymbol{u}, \nabla \times \boldsymbol{\psi} \rangle + \langle \boldsymbol{\epsilon} \omega^2 \boldsymbol{u}, \boldsymbol{\psi} \rangle - \langle \boldsymbol{F}, \boldsymbol{\psi} \rangle. \tag{6.10}$$

With eq. (6.7) and $\boldsymbol{u} = \boldsymbol{E}_p$, $\boldsymbol{\epsilon} = \boldsymbol{\epsilon}_p$ and $\boldsymbol{\mu}_0 = \boldsymbol{\mu}_p^{-1}$ the second of these conditions is fulfilled. For the first condition we define the adjoint state $\boldsymbol{E}_p^*$ as the solution of the problem

$$\langle \boldsymbol{\mu}_p^{-1} \nabla \times \boldsymbol{E}_p^*, \nabla \times \boldsymbol{\phi} \rangle - \langle \epsilon_p \omega^2 \boldsymbol{E}_p^*, \boldsymbol{\phi} \rangle = \frac{\partial L}{\partial \boldsymbol{u}}(\boldsymbol{E}_p; \boldsymbol{\phi}) \quad \forall \boldsymbol{\phi} \in H^0(\mathrm{curl}, \Omega_C). \tag{6.11}$$

With this definition, we find that

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{v}}(\boldsymbol{\mu}_p^{-1}, \boldsymbol{\epsilon}_p, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \cdot) = 0 \tag{6.12}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{u}}(\boldsymbol{\mu}_0^{-1}, \boldsymbol{\epsilon}_p, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \cdot) = 0. \tag{6.13}$$

Next, we note that L is not complex differentiable. Therefore we reinterpret the derivative as a real derivative by identifying $\mathbb{C}$ with $\mathbb{R}^2$ and find the derivative in the sense of Fréchet:

$$\frac{\partial L}{\partial \boldsymbol{u}}(\boldsymbol{E}_p; \boldsymbol{\phi}) = -2\Re\left(\int_{\Gamma_{\mathrm{out}}} \boldsymbol{F}_0 \cdot \overline{\boldsymbol{E}_p}\, \mathrm{d}A \int_{\Gamma_{\mathrm{out}}} \boldsymbol{F}_0 \cdot \overline{\boldsymbol{\phi}}\, \mathrm{d}A\right). \tag{6.14}$$

Inserting this equality into eq. (6.11) and with the simplification $\kappa = \int_{\Gamma_{\mathrm{out}}} \boldsymbol{F}_0 \cdot \overline{\boldsymbol{E}_p}\, \mathrm{d}A$ we can write the adjoint problem as

$$\text{Find } \boldsymbol{E}_p^* \text{ s.t.} \quad a_T(\boldsymbol{E}_p^*, \boldsymbol{\phi}) = -2\Re\left(\kappa \int_{\Gamma_{\mathrm{out}}} \boldsymbol{F}_0 \cdot \overline{\boldsymbol{\phi}}\, \mathrm{d}A\right) \quad \forall \boldsymbol{\phi} \in H^0(\mathrm{curl}, \Omega_C). \tag{6.15}$$

This problem is equivalent to solving the forward problem for a different right-hand side. Next, we turn to variations w.r.t. $\boldsymbol{\mu}_0$ and $\boldsymbol{\epsilon}$. We find

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu_0}}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \delta \boldsymbol{\mu}_0) = \int_{\Omega_C} \delta \boldsymbol{\mu}_0 \nabla \times \boldsymbol{E}_p \cdot \nabla \times \overline{\boldsymbol{E}_p^*}\, \mathrm{d}V \tag{6.16}$$

and

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\epsilon}}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \delta \boldsymbol{\epsilon}) = -\omega^2 \int_{\Omega_C} \delta \boldsymbol{\epsilon} \boldsymbol{E}_p \cdot \overline{\boldsymbol{E}_p^*}\, \mathrm{d}A. \tag{6.17}$$

Since both $\boldsymbol{\mu}$ and $\boldsymbol{\epsilon}$ depend on the shape parameter vector p, we combine these two terms and find

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial p_i}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \delta p_i) &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{\mu}_0}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \frac{\partial \boldsymbol{\mu}_0}{\partial p_i} \delta p_i) + \frac{\partial \mathcal{L}}{\partial \boldsymbol{\epsilon}}(\boldsymbol{\mu}_0, \boldsymbol{\epsilon}, \boldsymbol{E}_p, \boldsymbol{E}_p^*; \frac{\partial \boldsymbol{\epsilon}}{\partial p_i} \delta p_i) && (6.18)\\
&= \int_{\Omega_C} \frac{\partial \boldsymbol{\mu}_0}{\partial p_i} \delta p_i \nabla \times \boldsymbol{E}_p \cdot \nabla \times \overline{\boldsymbol{E}_p^*}\, \mathrm{d}V - \omega^2 \int_{\Omega_C} \frac{\partial \boldsymbol{\epsilon}}{\partial p_i} \delta p_i \boldsymbol{E}_p \cdot \overline{\boldsymbol{E}_p^*}\, \mathrm{d}V && (6.19)\\
&= \delta p_i \int_{\Omega_C} \frac{\partial \boldsymbol{\mu}_0}{\partial p_i} \nabla \times \boldsymbol{E}_p \cdot \nabla \times \overline{\boldsymbol{E}_p^*} - \omega^2 \frac{\partial \boldsymbol{\epsilon}}{\partial p_i} \boldsymbol{E}_p \cdot \overline{\boldsymbol{E}_p^*}\, \mathrm{d}V && (6.20)
\end{aligned}$$

For further details on the underlying variational techniques, see [Ott17; Lal+13, Chapter 6].

To compute these gradient components at a given material configuration p we perform the following steps:

1. Solve the forward problem for the material configuration p by solving definition 5.2.1 where the material properties $\boldsymbol{\epsilon}_p$ and $\boldsymbol{\mu}_p$ are given via transformation optics with the mapping $\Phi(\Omega_p)$. We use Nédélec elements to discretize the problem on the computational domain and compute the electric field in the geometry described by p as $\boldsymbol{E}_p$.

2. We evaluate the loss functional for $\boldsymbol{E}_p$.

3. We solve the adjoint problem as stated in eq. (6.15). The right-hand-side term $\boldsymbol{F}$ from problem one is replaced with the forcing term of the adjoint state.

4. We compute the components of the shape gradient by evaluating eq. (6.20) where we can use the same quadrature we have used for our finite element assembly loop to evaluate the integrals. We define $p^{i+} = p + h_{\text{opt}} e_i$ where e_i is the i-th unit vector and the partial derivatives of $\boldsymbol{\epsilon}$ by p_i can be evaluated at a given position $\boldsymbol{x}$ with the optimization step width $h_{\text{opt}} > 0$ small and the approximation
$$\frac{\partial \boldsymbol{\epsilon}_p}{\partial p_i}(\boldsymbol{x}) \approx \frac{\boldsymbol{\epsilon}_{p^{i+}}(\boldsymbol{x}) - \boldsymbol{\epsilon}_p(\boldsymbol{x})}{h_{\text{opt}}}. \tag{6.21}$$
In simple terms: For a given parametrization p we can compute the material tensors and we can therefore also evaluate finite differences. Alternatively, the analytic form of the material tensors can be derived by the parameter values symbolically. The same argument holds for $\boldsymbol{\mu}_0$

Once these steps are performed we know the value of the loss functional as well as its derivatives in the space of shape parameters. We can therefore use a generic optimization scheme to iteratively optimize the choice of p from P.

6.2 Optimization Algorithms

For optimization algorithms we have two principal options: We can either ignore the history of the iteration or use it to approximate the second derivative at our current state.

6.2.1 Gradient Methods

The simplest version to perform an iteration based on shape gradients is the method of steepest descent. For this method, we pick the component of the shape gradient that has the largest absolute value and update the corresponding component of the parameter vector by a certain step size. There is a large family of methods that compute the descent direction by
$$d^k = -D^k \nabla f(x^k) \tag{6.22}$$
for a positive definite matrix D^k, the step index k, the figure of merit evaluated for the current state $f(x^k)$. For $D^k = I$ this method is the method of steepest descent. Since the matrix D^k is positive definite, the property
$$(\nabla f(x^k))^T(-D^k)\nabla f(x^k) < 0 \tag{6.23}$$
always holds and therefore the partial derivative of f in the direction d^k is negative.

The method stated above for the cheap computation of the shape gradient does not extend to the second derivative in an equally applicable way and we have no way to efficiently compute the Hessian of f. There are however so-called **Quasi-Newton Methods** that approximate the Hessian by using the previous steps and evaluations of the figure of merit. One such scheme is the BFGS algorithm that we will discuss next.

6.2.2 Quasi-Newton Methods

In the class of Quasi-Newton methods, two well-known algorithms are **Broyden-Fletcher-Goldfarb-Shanno** (or **BFGS**, see [Fle87]) and the **Davidon-Fletcher-Powell**-method (or **DFP**, see [Dav91]).

The BFGS method starts by guessing an initial value $\boldsymbol{x}^{\mathbf{0}}$ and Hessian matrix B^0 and setting $k = 0$. The steps are to

1. Compute the descent direction by solving $B^k d^k = -\nabla f(x_k)$.
2. A line search is performed in this direction to determine an appropriate step width a^k, which can either be done directly by computing values of f along the line, or, much more commonly, indirectly via inequalities such as the Wolfe conditions.
3. Update $x^{k+1} = x^k + a^k d^k$ i.e. perform the step.
4. Compute the gradient difference $ddf^k = \nabla f(x^{k+1}) - \nabla f(x^k)$ which will be used to update the approximation of the Hessian B.
5. Update the Hessian to $B^{k+1} = B^k + \frac{ddf^k\,(ddf^k)^T}{(a^k\,ddf^k)^T\,d^k} - \frac{B^K\,d^k\,(d^k)^T\,(B^k)^T}{(d^k)^T\,B^k\,d^k}$.
6. Increment k.

As an abort condition either the value of f, the convergence rate or the step count can be used.

In our numerical experiments, we use the implementation of BFGS that is provided by the deal.II library and we use smooth waveguide shapes of low polynomial order to construct initial geometries connecting a given input and output connector.

6.3 Numerical Results

We will be discussing a specific shape of a waveguide in this section: Waveguides with a vertical displacement between the input and output connectors. For these waveguides, the input and output connectors are located are the same rectangle in the xy plane, only the z coordinate is different. This problem without further constraints would simply yield the straight waveguide connection from the input to the output. To make things more interesting, however, we introduce a vertical displacement in the center. This request states that the waveguide has to be shifted by the vector $(0, 1.5)^T\,\mu m$ between the two connectors. The system length is set to $6\mu m$. Additionally, we require the waveguide to be symmetric with respect to the plane $z = 3$.

This is a very small geometry which has two consequences: Signal transmission is basically always good since the structure we are optimizing is small compared to the wavelength (vacuum wavelength of $1.55\mu m$) and we can compute these shape quickly.

We first need to find a space of parameters for which the shapes are valid. To achieve this we use a piece-wise polynomial function $f(z)$. Our goal is to use the coordinate transformation $\boldsymbol{\phi}(x, y, z) = (x, y - f(z), z)$ to transform the real waveguide shape onto the fundamental waveguide shape, which is a straight waveguide connecting the input and output connectors.

Definition 6.3.1 (Shape Functions)**.** *For a $N > 2 \in \mathbb{N}$ and the interval length $L \in \mathbb{R}$ we define the nodes*

$$z_i = ih = i\frac{L}{N} \quad \textit{for } i \in \{0, \dots, N\} \tag{6.24}$$

and the function $f : [0, L] \to \mathbb{R}$ by

$$f(0) = f_0 \tag{6.25}$$

$$f(L) = f_L \tag{6.26}$$

$$f'(z_i) = f_i' \quad \textit{for } i \in \{0, \dots, N\} \tag{6.27}$$

$$f'(z_i + d) = f_i' + \frac{d}{h}(f_{i+1}' - f_i') \quad \textit{for } d \in [0, h], i \in \{0, \dots, N-1\} \quad \textit{and} \tag{6.28}$$

$$f(z) = f_0 + \int_0^L f'(z)dz \quad \textit{for } z \in [0, L]. \tag{6.29}$$

For given values $f_0, f_L, f'_0, \dots, f'_{N-2}$ *and* f'_N *the value of* f'_{N-1} *is uniquely defined by*

$$f'_{N-1} = \frac{f_N - f(x_{N-2})}{L} - \frac{f'_{N-2} + f'_N}{2} \tag{6.30}$$

To fulfill the symmetry requirement of the shape, we will only model the function on half the space and evaluate the rest via the symmetry condition. We set $L = 3$, $N = 5$ and

$$f_0 = 0 \tag{6.31}$$
$$f_L = 1.5 \tag{6.32}$$
$$f'_0 = 0 \quad \text{and} \tag{6.33}$$
$$f'_N = 0. \tag{6.34}$$

For our choice of $N = 5$ the values f'_1, f'_2 and f'_3 are our shape parameters.

We define $P = \mathbb{R}^3$ and for $p \in P$ we identify $p_i = f'_{i+1}$. ϕ is differentiable and invertible on $\mathbb{R} \times \mathbb{R} \times [0, 3]$ and, by symmetric extension, on $\mathbb{R} \times \mathbb{R} \times [0, 6]$.

Experiment 7. ***Setup:*** *We use the domain of interest* $\overline{\Omega_I} = [-2, 2] \times [-1.8, 1.8] \times [0, 6]$ *truncated by a PML domain of width* 1.5 *in all directions leading to* $\overline{\Omega_C} = [-3.5, 3.5] \times [-3.3, 3.3] \times [-1.5, 7.5]$*. The PML is 10 layers thick and we discretize the interior by* $12 \times 12 \times 48$ *cells. We use 4 processes for a sweep in the z direction and Nédélec elements of lowest order. As our initial guess for the parameter values* p^0 *we choose* $(0, 0, 0)^T$ *and we use the input mode profile* $\boldsymbol{F}_0$ *computed in experiment 1. The signal loss in the waveguide is computed as*

$$L(\boldsymbol{E}) = 1 - \left| \int_{\Gamma_{out}} \boldsymbol{F}_0 \cdot \overline{\boldsymbol{E}} \, \mathrm{d}A \right|^2 . \tag{6.35}$$

The global system has $\dim V_h = 214\,272$ *degrees of freedom.*
Results: *In the first steps, the stepwidth increases since the loss functional gets lower in every step. This eventually leads to a step (step 6) that massively reduces the signal transmission instead of improving it because the step width is too large. As the optimization continues, the loss functional recovers and the method converges to a final state as the step width goes to zero. The loss functional starts at* 0.73% *and is reduced to* 0.48%*, a reduction by* 65%*. The exact values during the optimization are shown in table 6.1. We compute these shapes on a relatively coarse grid, and therefore we see in fig. 6.3 that the convergence rate of the solver decreases considerably when the waveguide is bent. A sharp turn in the waveguide shape implies a large derivative of f and therefore a larger norm of the Jacobian of the transformation. This directly impacts the material tensors and thus the wavelength of the solution. To keep the convergence rate of the sweeping preconditioner constant we would need higher spatial resolution.*

The decaying convergence of GMRES in the shape optimization process motivates once again that schemes that can handle higher spatial resolution are required. 1D sweeping preconditioners reach a breaking point when the cross section of the computational domain becomes large and or a generally high spatial resolution or polynomial degree is required.

6.4 Further Possible Applications

So far, we have focused on examples in which the shape of a waveguide was subject to optimization. These setups consist of two connectors: one input and one output. In the method itself, there is, however, no requirement for such a setup.

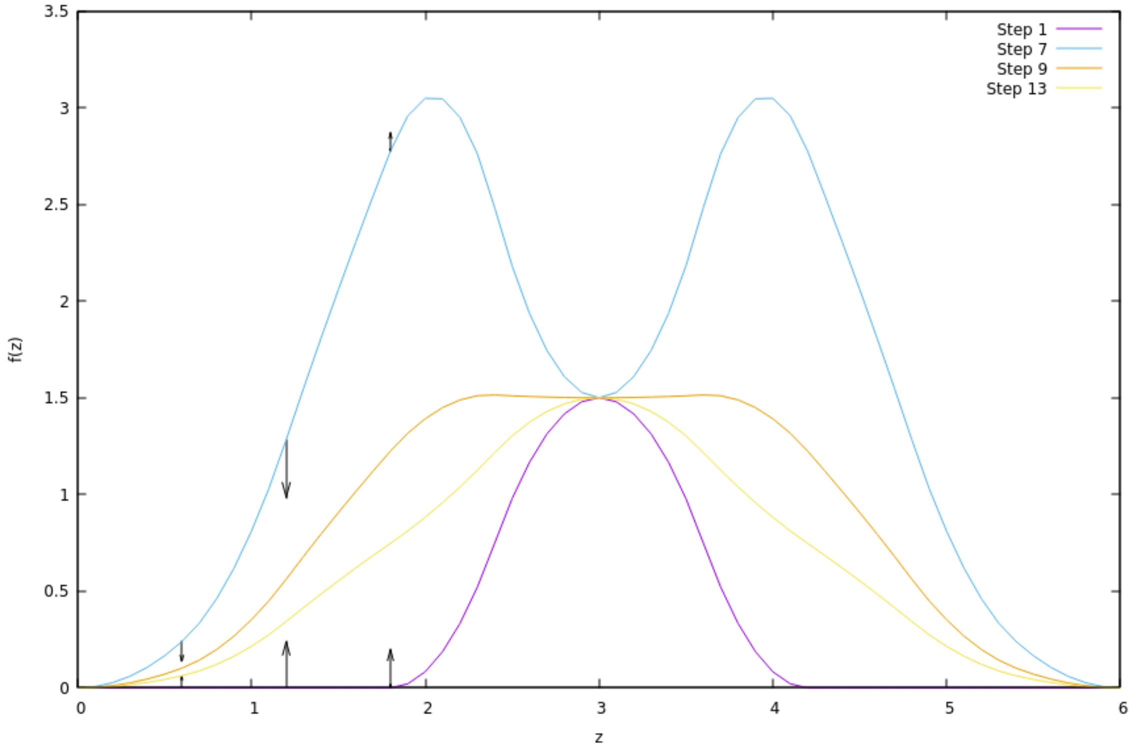


Figure 6.1: Results from experiment 7. The degrees of freedom of the shape functions are located at $z = 0.6$, $z = 1.2$ and $z = 1.8$. The arrows are the shape gradient in the of the individual state for each of these nodes of the two most extreme states, step 1 and 7. Their length is scaled by -100. The sign minus is included because we perform loss minimization so the shape gradient components are negative for directions of signal quality *improvement* directions.

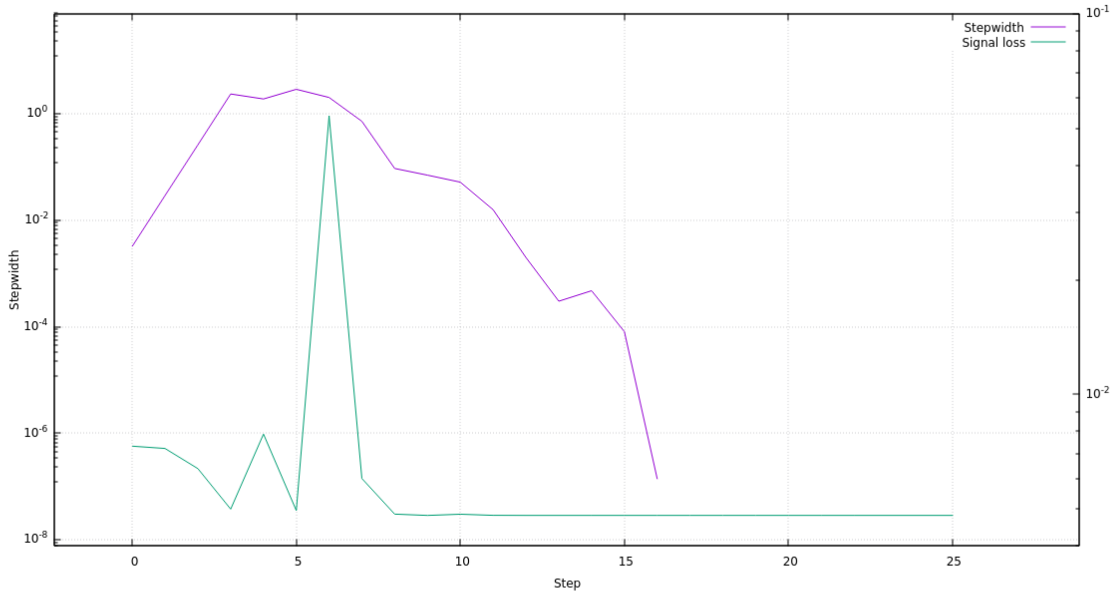


Figure 6.2: Results from experiment 7. The green line shows the error going from 0.73% to 0.48% and the stepwidth converges to zero. See the results section of experiment 7 for more details.

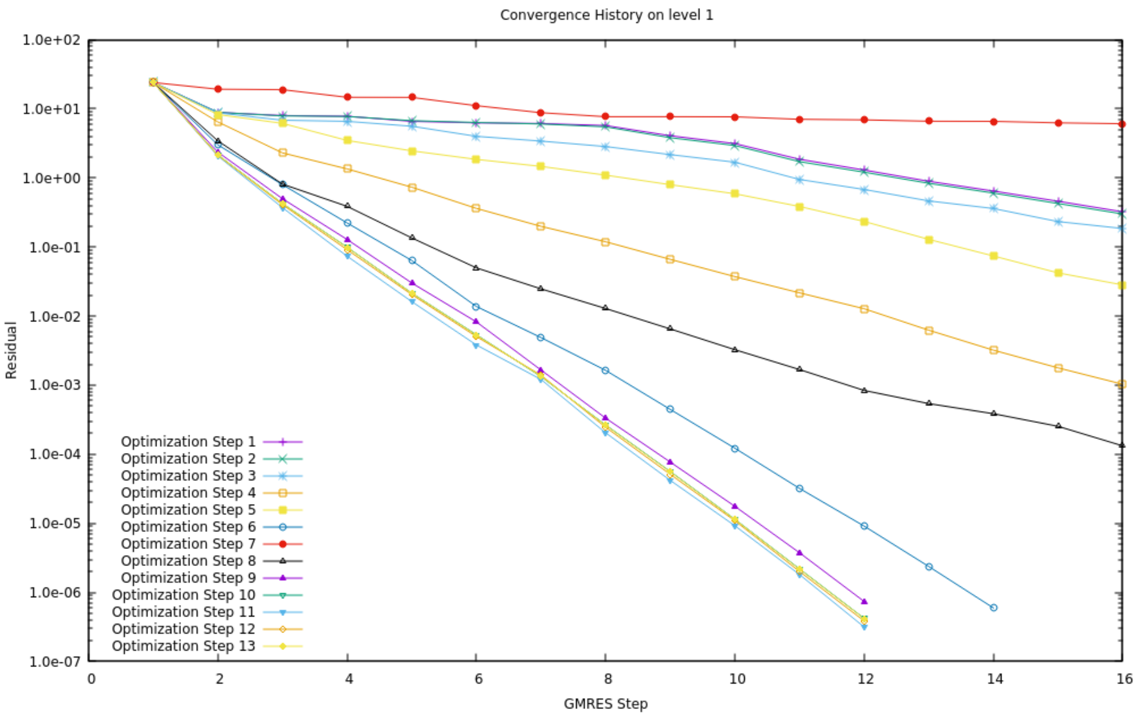


Figure 6.3

Step	Step width	Signal loss
1	-	0.73%
2	1.46e-4	0,72%
3	1.16e-3	0,63%
4	1.96e-3	0,50%
5	4.05e-3	0,78%
6	4.11e-3	0,49%
7	6.92e-2	5.39%
8	6.77e-2	0,60%
9	1.64e-3	0,48%
10	5.36e-5	0,48%
11	5.09e-5	0,48%
12	4.41e-5	0,48%
13	6.11e-6	0,48%
14	6.82e-7	0,48%

Table 6.1: Development of the step width and loss functional during the optimization procedure.

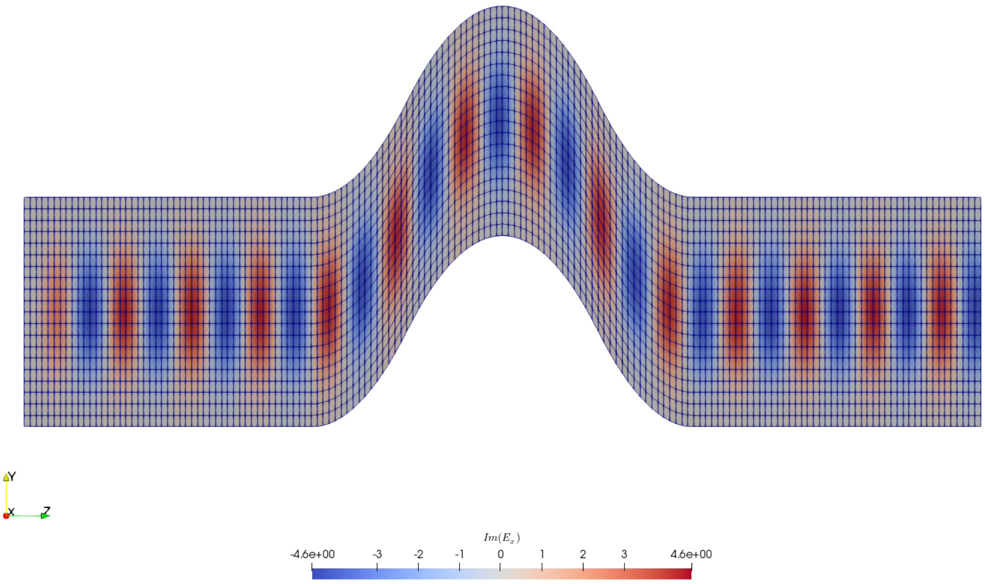


Figure 6.4: Exemplary visualization of the field on a cross-section along the waveguide core.

Some loss of generality occurred when we decided in section 6.3 to focus on shapes, which are symmetric in the propagation direction of the signal. The idea behind this assumption was, that in that case, the adjoint state, which is required for the computation of the shape gradient, can be computed by performing a coordinate transformation of the primal solution.

This was a simplification of the problem that drastically reduces the duration of our numerical experiments (cutting it in half). We made this assumption because it fits the settings we were dealing with but it is in no way required. If we drop this assumption, we can compute the adjoint state with the same solver we use for the primal state. As a consequence, no such symmetry is required for this method.

Furthermore, we have only focused on examples containing one signal input and one signal output. One frequently used application of shape optimization algorithms in electromagnetic design is the case of so-called *Beam Splitters*. Let's assume we have an input connector that is multi-moded with m modes used for telecommunication. A beam splitter is a device, that extracts the m individual signals to separate waveguides. Our geometry would consist of one input and m outputs. The input signal would be the sum of all m modes on the input waveguide and we define a separate loss-functional for each output and the corresponding signal we wish to extract into that waveguide. For each of the outputs we find an individual adjoint state and we can compute a shape-gradient for the individual loss-functionals. We define our main loss functional as the sum of the individual loss-functionals and the main shape-gradient as the sum of the individual shape-gradients. Performing an optimization algorithm as described above yields a series of shapes of improving signal qualities in the individual waveguides. We could improve this method further by including terms in the loss functional, that penalize cross-talk amongst the output waveguides, but such considerations are beyond the scope of this work.

In chapter 4 we decided to assume anisotropic, inhomogeneous media for this method. While this would not be necessary for handling waveguide examples (unless transformation optics or PML are applied) this also enables us to consider more complex materials, such as photonic crystals.

7 Implementation

This chapter is intended to enable the reader to understand the challenges we faced during the implementation of the numerical scheme described in chapter 5 and chapter 6. While the commented source code is included in chapter 8, this chapter serves to guide you towards the answer for individual questions by introducing the structure of the implementation and the reasons behind it.

The main steps of the algorithm and their co-dependence is shown in fig. 7.1.

7.1 HPC Setting and Requirements

The high memory requirements as well as the difficulty of applying standard algorithms in an optimal manner to large amounts of data, make it necessary to view the methods described in this document in a **High Performance Computing** (HPC) setting. The hierarchical sweeping preconditioner is designed to be easily parallelizable and the shape optimization method is performed on E-field evaluations of

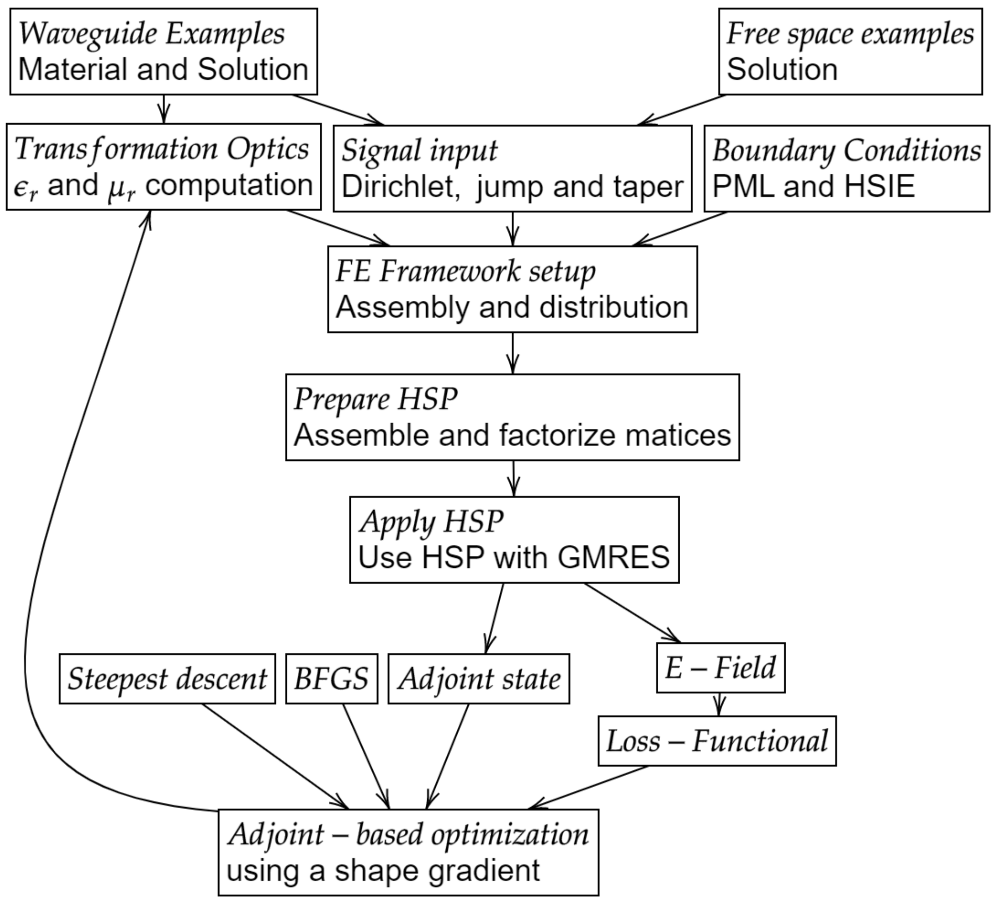


Figure 7.1: Structural Components of the provided Waveguide Solver Code

very high resolution, thus implying that parallelization also makes sense for this part. As a consequence, scaleability and performance as well as access to efficient implementations of basic operations are metrics for the appropriate choice of a setting to implement this method in.

We implemented these parts (except the computation of the input signals for the free space and waveguide examples) in C++. The language is available on every HPC system. As a finite element framework, deal.II (see [Arn+21]) was used. The code has been written to be compatible with the version 9.3 of deal.II and since the framework is under constant further development, it cannot be guaranteed that it will be compatible with future versions. Deal.II provides wrappers around basic data types like matrices and vectors and their distributed versions, in which parts of the same object are stored on different processes. The deal.II framework also provides two implementations of Nédélec-elements (see [KL17]) and integrates well with platform-dependent implementations of the MPI standard, which provides tools for distributed memory parallelization on HPC clusters.

Over the last decades, desktop and laptop computers have changed dramatically, however, to a point, where parallelized workloads with a low number of processes can also be handled on consumer electronics. At the time of writing this document, consumer PCs can have up to 64 CPU cores and their memory can be up to 128GB (RAM, not to be confused with storage). As a consequence, the code base provided with this dissertation, can be used on desktop PCs and Laptops to simulate geometries of low to medium volume.

The code is based around PETSc for data types and deal.II for everything else from fem to output generation. As a consequence, a build of those libraries in required for this code to compile. Since we work with complex numbers, both PETSc and deal.II have to be configured to support this. Additionally, either UMFPACK or MUMPS should be used as a direct solver and need to be provided or built within the dependencies. It is not possible to provide installation instructions for all possible target systems, especially because HPC systems usually require a lot of individual configuration. The code is written in such a way, however, that once the dependencies work, this code should work, too. It supports both Intel and Gnu compilers and an installation script is available for Ubuntu Linux systems based on the 20.04 distribution, which can be considered a very mainline operating system. Since the installation is mainly a bash-script this part can be executed on almost every Linux type system (or Windows with WSL). It begins with an installation command for base-packages, however, which has to be adapted for other package managers.

Beyond the point of installation, there is currently a bug in deal.II that makes an adaptation in deal.II source code required to compile the waveguide solver. A bug report has been issued but not resolved at the time of writing this chapter. Instructions on how to perform this adaptation can be found in the Readme.md file in the code repository.

If you want to read the code and understand how theoretical aspects from this work are implemented in the code, we recommend reading the code documentation, which is part of this document for your convenience. You can extract any code, that is of interest for you from there and navigate between building-blocks easily. Reading this code, we recommend keeping one detail in mind: The code is optimized by runtime, not by local performance. This means that we have only invested effort in optimization where it actually improves the runtime of the code. Some of the examples run for up to three days for the evaluation of a single E-field without shape optimization. While the setup might be inefficient in these cases, it does not really matter because the runtime of the setup is only some minutes. We have therefore invested most of our time into making the time-consuming parts perform well rather than optimizing other parts.

If your goal is to run the code, we would redirect you to the code repository. The best starting-point to get the code running is the file Readme.md, which provides instructions about the installation and the structure of the repository.

In line with GSP recommendations, the folder RUN contains scripts that perform the numerical experiments discussed in this work. Some of them can be performed locally, some require an HPC system. The ones that require an HPC system are built to work with the BWUniCluster 2.0 at KIT and thus use the Slurm batch system. You will not be able to run these scripts without that batch system, but it should be simple to translate the parameter values to any other HPC batch system since there are no specific dependencies on Slurm.

7.2 The Hierarchical Structure

To make sweeping possible in multiple ways, we included a hierarchical implementation. This is based around the abstract base class HierarchicalProblem from which two classes are derived:

- LocalProblem: This class builds a fem system, assembles the system matrix and vectors and applies a direct solver to it.
- NonLocalProblem: This class also builds a fem system but the objects are distributed, because this object describes a sweeping setting. The system matrix and vectors are shared among all processes involved in the sweep. Additionally, it uses a GMRES solver to solve the system. As a preconditioner for this GMRES solver it uses a pointer to it's member named child, which either points to another non-local problem (for hierarchical sweeping) or to a LocalProblem, which then solves directly.

In theory, the stack of NonLocalProblems could be as high as one wants, but, since every level assembles a large matrix and thus requires a lot of memory, this was only built to work with level 1, 2 or 3 sweeping. A cube shaped computational domain with 20 subdomains in each spatial direction would require a total of 8000 processes and represent something like $420 \times 420 \times 420$ cells. This would build a system matrix of size $\approx 0.25 \cdot 10^9$ degrees of freedom and could still be extended without the preconditioner running into the convergence issues discussed in section 5.3.3.

7.3 Parameters

There are three ways to provide parameter values to this scheme:

- The case file: This file contains all the parameters to describes **what** you want to solve. This includes what the domain of interest is shaped like, parameters of the waveguide and materials and the type of input signal.
- The run file: This file describes **how** you want to solve it. This includes solver parameters, process distribution and PML settings.
- Overrides: Since the first two sets of parameters are provided via input files, it can be interesting to pass some parameter overrides from the terminal when executing the program. This has the advantage of not requiring an adapted parameter file. Say you want to run the same simulation and only vary the number of PML cell layers used for domain truncation, then you reuse the same case- and run-file but adapt the cell layer count with an override parameter.

To see which parameters can be provided, please see the classes Parameter, ParameterReader and ParameterOverride.

7.4 Current Features

The code provides the following features:

- Waveguide simulations for rectangular waveguides. These can either be straight, of predefined shape, or only provide a vertical shift across the length of the computational domain. The waveguide consists of an interior and exterior specified by constant scalar values of ϵ_r.
- Waveguide shape optimization. In this case, the geometric restrictions are the same as above but the geometry is described by shape functions, which have degrees of freedom. After each step, the loss functional is evaluated and a shape update is computed via an adjoint based method to compute the shape gradient. BFGS is used to compute an update.
- Open space simulations. It is also possible to run the code without a waveguide. As an example for this, the solution of the Hertzian diapole [KH14] has been implemented and made available as an input signal.
- Level 1, 2 and 3 sweeping. The solver can be configured to sweep in either only the z direction, the z and y direction or in z, y and x direction. On the lowest level it uses a direct solver (MUMPS).
- PML boundary method. The code uses an adapted version of the moving PML method for sweeping and also employs PML for domain truncation in the assembled problems. There is also an implementation of Hardy space infinite elements in the code base that is based on the first row in fig. 5.13. As long as the corner and edge cell assembly problem is not addressed, however, it does not work for sweeping.
- FEM of "any" order. The code uses either Nédélec-elements of order 1 or 2. Higher orders have not been tested since it is not required. The code is capable of handling all three types of inner degrees of freedom (edge, face and cell) and therefore all building blocks should work for Nédélec elements of any order. Nédélec elements of high order, however, create system matrices of large bandwidth and are therefore numerically costly. As a consequence, we have only experimented with elements of lowest order and order 1. The same order is used for PML boundary conditions and HSIE.
- Output: The code creates a sub–folder in the folder Solutions for every run. After the run has completed successfully, this folder will include the output log, convergence graphs of GMRES solvers, a run description with some core properties of the setup and the field output in .vtk and .vtu file format. The .vtu files, whose name begins with "_" are union files that incorporate all the output generated on level. As a consequence these are the best ones to open in software like Paraview. If there is an error while opening these files, individual files are also available, that contain all the data from the PML regions and the inner domain from every process separately.

7.5 Possible Improvements

There are two main features that can be considered as next extensions of this code:

- The local problem uses a direct solver to solve the system. This is the dominant amount of numerical cost involved in the solver and building this direct solver makes up most of the memory consumption. Direct solvers, like the ones employed in this code, are based on matrix factorizations, so their application to an input vector consists of matrix-vector products. GPUs are much faster at performing such tasks at near ideal speedup (if the communication time is taken into account) and would thus be ideally suited to perform this task. At the time of writing this document, there is no solver for GPUs that uses complex arithmetic and is capable of solving indefinite systems. Additionally, this would have to be accessible from C++ – ideally through deal.II interfaces.

- If the problems expressed in fig. 5.13 are addressed by providing either an appropriate interpolation of dof values for rows 2 or 3 or a way to compute the coupling term in row 1, HSIE could be used in the sweeping preconditioner. Matrix blocks generated by HSIE are less expensive to solve than the ones originating from PML truncation. HSIE need less degrees of freedom and their matrix blocks have better properties, therefore this would provide the opportunity to have more degrees of freedom in the domain of interest. In the current version of the code, up to 80% of the dofs can be PML dofs. This would drastically reduce runtime and numerical cost.

7.6 Progress.md

The entire development of this code took over 5 years. Towards the end of this time-frame, we used a work diary to keep track of the progress we made and listed steps we took to alleviate problems. This document is part of the repository. If you wish to adapt the code and run into problems, searching this document for related code words can be helpful since it includes most of the latest changes we made and their motivation. As another remark: The public version of the git repository is built around the master-branch. Many attempts to add specific features, however, were performed on branches that were sometimes abandoned, if the feature became irrelevant or if a different implementation was attempted. For example, I tried migrating from DOUBLE to COMPLEX<DOUBLE> several times before deal.II implemented all the features required for me to do so successfully. I also tried GPU solvers, other direct and iterative solvers etc.

7.7 Participation

The code for this project in its current state will be made publicly available. It will remain under development in which you can partake if you are interested. Comments and feature requests are welcome. The development of this code was only possible through the funding of my dissertation through the DFG and we therefore consider it important to make this work available to anyone.

8 Appendix 1: Code

On the following pages you will find the documentation of the codebase used for this project.

The code is available at `https://git.scc.kit.edu/mx3529/waveguide-problem`.

Introduction

1 Topics of this project

This project began as the implementation used in the thesis for the title of Master of Science by Pascal Kraft at the KIT. It is continued for his PHD studies and possibly as an introduction to deal.II for other students in the same research group. This project, apart from mathematical goals, aims at creating a clear and reusable implementation of the the finite element method for Maxwell's equations in a range of performance values, that enable the inclusion of an optimization-scheme without crippling time- or CPU-time consumption. Therefore the code should fulfill the following criteria:

1. The code should be readable to starters (educational purpose),
2. The code should be maintainable (re-usability),
3. The code should be paralellizable via MPI or CUDA (both will be tested as a part of the PhD-proceedings),
4. The code should perform well under the given circumstances,
5. The code should give scientific results and not only operate on marginal domains of parameter-values,
6. The code should be portable to other hardware-specifications then those on the given computer at the workspace (i.e. the performance should be usable in large-scale computations for example in Super Computers of the KIT's SCC.

These demands led to the introduction of a software development scheme for the work on the code based on agile-development and git.

2 Prerequisites of this project

In order to be able to work with this code it is important to first achieve a fundamental understanding of the following topics: First and foremost, an understanding of the finite element method is required and completely irreplaceable. There exists extensive documentation on this topic and the reader should be aware of the fact, that the mathematical background cannot be understood without this knowledge. However, there are further demands. The programming-language of both this project and deal.II itself is C++. This language also forms the backbone of CUDA and many other relevant libraries. It is to be considered inevitable in this field. The choice of this language reduces the importance of a high performance implementation on the code level. Also it should be noted that there exists a very large documentation about deal.II which might help the reader understand this code. Lastly deal.II is basically only available on Linux since it nearly always requires a build-process which would not be possible without enormous problems on a different OS. As far as mathematical knowledge is concerned, a basic education in linear algebra, Krylov subspace methods, transformation-optics, functional analysis, optics and optimization theory will further the understanding of both the code and this documentation of it.

Hierarchical Index

1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Class Index

1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

File Index

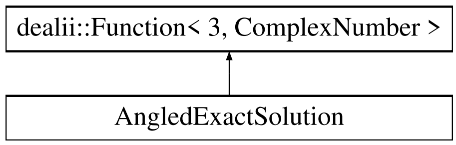
1 File List

Here is a list of all documented files with brief descriptions:

Class Documentation

1 AngledExactSolution Class Reference

Inheritance diagram for AngledExactSolution:



Public Member Functions

- std::vector< std::string > **split** (std::string) const
- ComplexNumber **value** (const Position &p, const unsigned int component) const
- void **vector_value** (const Position &p, dealii::Vector< ComplexNumber > &value) const
- dealii::Tensor< 1, 3, ComplexNumber > **curl** (const Position &in_p) const
- dealii::Tensor< 1, 3, ComplexNumber > **val** (const Position &in_p) const
- Position **transform_position** (const Position &in_p) const

1.1 Detailed Description

Definition at line 12 of file AngledExactSolution.h.

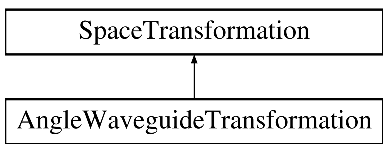
The documentation for this class was generated from the following files:

- Code/Solutions/AngledExactSolution.h
- Code/Solutions/AngledExactSolution.cpp

2 AngleWaveguideTransformation Class Reference

```
#include <AngleWaveguideTransformation.h>
```

Inheritance diagram for AngleWaveguideTransformation:



Public Member Functions

- Position math_to_phys (Position coord) const

  Transforms a coordinate in the mathematical coord system to physical ones.
- Position phys_to_math (Position coord) const

  Transforms a coordinate in the physical coord system to mathematical ones.
- dealii::Tensor< 2, 3, double > get_J (Position &coordinate) override

  Compute the Jacobian of the current transformation at a given location.
- dealii::Tensor< 2, 3, double > get_J_inverse (Position &coordinate) override

  Compute the Jacobian of the current transformation at a given location and invert it.
- double get_det (Position coord) override

  Get the determinant of the transformation matrix at a provided location.
- dealii::Tensor< 2, 3, ComplexNumber > get_Tensor (Position &coordinate) override

  Get the transformation tensor at a given location.
- dealii::Tensor< 2, 3, double > get_Space_Transformation_Tensor (Position &coordinate) override

  Get the real part of the transformation tensor at a given location.
- void estimate_and_initialize ()

  At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.
- Vector< double > get_dof_values () const

  Other objects can use this function to retrieve an array of the current values of the degrees of freedom of the functional we are optimizing.
- unsigned int n_free_dofs () const

  This function returns the number of unrestrained degrees of freedom of the current optimization run.
- unsigned int n_dofs () const

  This function returns the total number of DOFs including restrained ones.
- void Print () const

  Console output of the current Waveguide Structure.

Additional Inherited Members

2.1 Detailed Description

Definition at line 20 of file AngleWaveguideTransformation.h.

2.2 Member Function Documentation

estimate_and_initialize()

`void AngleWaveguideTransformation::estimate_and_initialize ( ) [virtual]`

At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.

Therefore the values for the degrees of freedom need to be estimated. This function sets all variables to appropiate values and estimates an appropriate shape based on averages and a polynomial interpolation of the boundary conditions on the shape.

Implements SpaceTransformation.

Definition at line 70 of file AngleWaveguideTransformation.cpp.

```
70                                                            {
71
72 }
```

get_det()

```
double AngleWaveguideTransformation::get_det (
             Position   )  [override], [virtual]
```

Get the determinant of the transformation matrix at a provided location.

Returns

double determinant of J.

Reimplemented from SpaceTransformation.

Definition at line 39 of file AngleWaveguideTransformation.cpp.

```
39                                                        {
40   if(!is_constant || !is_det_prepared) {
41     det = determinant(get_J(c));
42     is_det_prepared = true;
43   }
44   return det;
45 }
```

References get_J().

Referenced by get_Space_Transformation_Tensor().

get_dof_values()

`Vector< double > AngleWaveguideTransformation::get_dof_values ( ) const [virtual]`

Other objects can use this function to retrieve an array of the current values of the degrees of freedom of the functional we are optimizing.

This also includes restrained degrees of freedom and other functions can be used to determine this property. This has to be done because in different cases the number of restrained degrees of freedom can vary and we want no logic about this in other functions.

Reimplemented from SpaceTransformation.

Definition at line 74 of file AngleWaveguideTransformation.cpp.

```
74                                                             {
75   Vector<double> ret;
76   return ret;
77 }
```

get_J()

```
dealii::Tensor< 2, 3, double > AngleWaveguideTransformation::get_J (
              Position & ) [override], [virtual]
```

Compute the Jacobian of the current transformation at a given location.

Returns

Tensor<2,3,double> Jacobian matrix at the given location.

Reimplemented from SpaceTransformation.

Definition at line 16 of file AngleWaveguideTransformation.cpp.

```
16                                                                          {
17   if(!is_constant || !is_J_prepared) {
18     dealii::Tensor<2, 3, double> ret;
19     ret[0][0] = 1;
20     ret[1][1] = 1;
21     ret[2][2] = 1;
22     ret[2][1] = -0.2;
23     J_perm = ret;
24     is_J_prepared = true;
25   }
26   return J_perm;
27 }
```

Referenced by get_det(), get_J_inverse(), and get_Space_Transformation_Tensor().

get_J_inverse()

```
dealii::Tensor< 2, 3, double > AngleWaveguideTransformation::get_J_inverse (
              Position & ) [override], [virtual]
```

Compute the Jacobian of the current transformation at a given location and invert it.

Returns

Tensor<2,3,double> Inverse of the jacobian matrix at the given location.

Reimplemented from SpaceTransformation.

Definition at line 29 of file AngleWaveguideTransformation.cpp.

```
29                                                                                   {
30   if(!is_constant || !is_J_inv_prepared) {
31     dealii::Tensor<2, 3, double> ret = get_J(c);
32     ret = invert(ret);
33     J_inv_perm = ret;
34     is_J_inv_prepared = true;
35   }
36   return J_inv_perm;
37 }
```

References get_J().

get_Space_Transformation_Tensor()

```
Tensor< 2, 3, double > AngleWaveguideTransformation::get_Space_Transformation_Tensor (
            Position & ) [override], [virtual]
```

Get the real part of the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 92 of file AngleWaveguideTransformation.cpp.

```
92                                                                {
93   Tensor<2, 3, double> ret;
94   ret[0][0] = 1;
95   ret[1][1] = 1;
96   ret[2][2] = 1;
97   return (get_J(p) * ret * transpose(get_J(p))) / get_det(p);
98 }
```

References get_det(), and get_J().

Referenced by get_Tensor().

get_Tensor()

```
Tensor< 2, 3, ComplexNumber > AngleWaveguideTransformation::get_Tensor (
            Position & ) [override], [virtual]
```

Get the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 66 of file AngleWaveguideTransformation.cpp.

```
66                                                    {
67   return get_Space_Transformation_Tensor(position);
68 }
```

References get_Space_Transformation_Tensor().

math_to_phys()

```
Position AngleWaveguideTransformation::math_to_phys (
            Position coord ) const [virtual]
```

Transforms a coordinate in the mathematical coord system to physical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the mathematical system

Returns

Position Coordinate in the physical system

Implements SpaceTransformation.

Definition at line 49 of file AngleWaveguideTransformation.cpp.

```
49                                                                    {
50   Position ret;
51   ret[0] = coord[0];
52   ret[1] = coord[1];
53   ret[2] = coord[2] + GlobalParams.PML_Angle_Test*coord[1];
54   return ret;
55 }
```

n_dofs()

`unsigned int AngleWaveguideTransformation::n_dofs ( ) const [virtual]`

This function returns the total number of DOFs including restrained ones.

This is the lenght of the array returned by Dofs().

Reimplemented from SpaceTransformation.

Definition at line 87 of file AngleWaveguideTransformation.cpp.

```
87                                                     {
88   return 0;
89 }
```

phys_to_math()

```
Position AngleWaveguideTransformation::phys_to_math (
              Position coord ) const  [virtual]
```

Transforms a coordinate in the physical coord system to mathematical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the physical system

Returns

Position Coordinate in the mathematical system

Implements SpaceTransformation.

Definition at line 57 of file AngleWaveguideTransformation.cpp.

```
57                                                                    {
```

```
    Position ret;
    ret[0] = coord[0];
    ret[1] = coord[1];
    ret[2] = coord[2] - GlobalParams.PML_Angle_Test*coord[1];
    return ret;
}
```

The documentation for this class was generated from the following files:

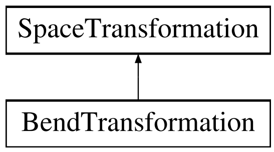
- Code/SpaceTransformations/AngleWaveguideTransformation.h
- Code/SpaceTransformations/AngleWaveguideTransformation.cpp

3 BendTransformation Class Reference

This transformation maps a 90-degree bend of a waveguide to a straight waveguide.

`#include <BendTransformation.h>`

Inheritance diagram for BendTransformation:



Public Member Functions

- Position math_to_phys (Position coord) const override

  Transforms a coordinate in the mathematical coord system to physical ones.
- Position phys_to_math (Position coord) const override

  Transforms a coordinate in the physical coord system to mathematical ones.
- dealii::Tensor< 2, 3, ComplexNumber > get_Tensor (Position &coordinate) override

  Get the transformation tensor at a given location.
- dealii::Tensor< 2, 3, double > get_Space_Transformation_Tensor (Position &coordinate) override

  Get the real part of the transformation tensor at a given location.
- void estimate_and_initialize () override

  At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.
- void Print () const override

  Console output of the current Waveguide Structure.

Additional Inherited Members

3.1 Detailed Description

This transformation maps a 90-degree bend of a waveguide to a straight waveguide.

This transformation determines the full arch-length of the 90-degree bend as the length given as the global-z-length of the system. It can then determine all properties of the transformation. The computation of the material tensors is performed via symbolic differentiation instead of the version chosen in other transformations. This ansatz is therefore the one most easy to use for a new transformation.

The bend transformation also has internal sectors for the option of shape transformation. The y-shifts represent an inward or outward shift in radial direction, the width remains the same.

Definition at line 27 of file BendTransformation.h.

3.2 Member Function Documentation

estimate_and_initialize()

`void BendTransformation::estimate_and_initialize ( ) [override], [virtual]`

At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.

Therefore the values for the degrees of freedom need to be estimated. This function sets all variables to appropiate values and estimates an appropriate shape based on averages and a polynomial interpolation of the boundary conditions on the shape.

Implements SpaceTransformation.

Definition at line 41 of file BendTransformation.cpp.

```
41                                                  {
42   return;
43 }
```

get_Space_Transformation_Tensor()

```
Tensor< 2, 3, double > BendTransformation::get_Space_Transformation_Tensor (
            Position & ) [override], [virtual]
```

Get the real part of the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 36 of file BendTransformation.cpp.

```
36                                                                {
37   Tensor<2, 3, double> transformation;
38   return transformation;
39 }
```

Referenced by get_Tensor().

get_Tensor()

```
Tensor< 2, 3, ComplexNumber > BendTransformation::get_Tensor (
            Position & )  [override], [virtual]
```

Get the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 31 of file BendTransformation.cpp.

```
31                                              {
32   return get_Space_Transformation_Tensor(position);
33 }
```

References get_Space_Transformation_Tensor().

math_to_phys()

```
Position BendTransformation::math_to_phys (
            Position coord ) const  [override], [virtual]
```

Transforms a coordinate in the mathematical coord system to physical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the mathematical system

Returns

Position Coordinate in the physical system

Implements SpaceTransformation.

Definition at line 18 of file BendTransformation.cpp.

```
18                                                      {
19   Position ret;
20
21   return ret;
22 }
```

phys_to_math()

```
Position BendTransformation::phys_to_math (
            Position coord ) const  [override], [virtual]
```

Transforms a coordinate in the physical coord system to mathematical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the physical system

Returns

Position Coordinate in the mathematical system

Implements SpaceTransformation.

Definition at line 24 of file BendTransformation.cpp.

```
24                                                          {
25   Position ret;
26
27   return ret;
28 }
```

The documentation for this class was generated from the following files:

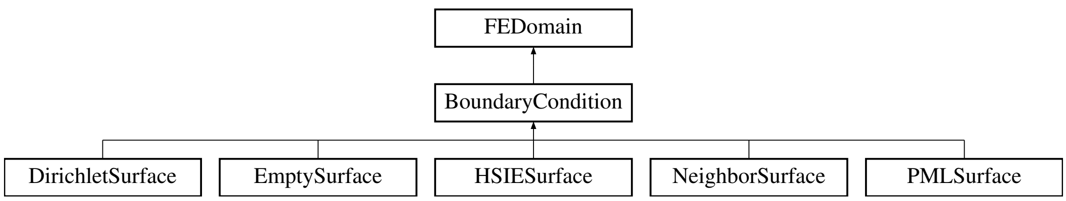
- Code/SpaceTransformations/BendTransformation.h
- Code/SpaceTransformations/BendTransformation.cpp

4 BoundaryCondition Class Reference

This is the base type for boundary coniditions. Some implementations are done on this level, some in the derived types.

`#include <BoundaryCondition.h>`

Inheritance diagram for BoundaryCondition:



Public Member Functions

- **BoundaryCondition** (unsigned int in_bid, unsigned int in_level, double in_additional_coordinate)
- virtual void initialize ()=0

  Not all data for objects of this type will be available at time of construction.
- virtual std::string output_results (const dealii::Vector< ComplexNumber > &in_solution, std::string filename)=0

  Writes output for a provided solution to a file with the provided name.
- virtual bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid)=0

  Checks if a 2D coordinate is on the a surface of the boundary methods domain.
- void set_mesh_boundary_ids ()

If the boundary condition has its own mesh, this function iterates over the mesh and sets boundary ids on the mesh.

- auto get_boundary_ids () -> std::vector< BoundaryId >

  Returns a vector of all boundary ids associated with dofs in this domain.

- virtual auto get_dof_association () -> std::vector< InterfaceDofData >=0

  Returns a vector of all degrees of freedom shared with the inner domain.

- virtual auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData >=0

  More general version of the function above that can also handle interfaces with other boundary ids.

- virtual auto get_global_dof_indices_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< DofNumber >

  Specific version of the function above that provides the indices in the returned vector by their globally unique id instead of local numbering.

- virtual void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗constraints)=0

  If this object owns degrees of freedom, this function fills a sparsity pattern for their global indices.

- virtual void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix ∗matrix, NumericVectorDistributed ∗rhs, Constraints ∗constraints)=0

  Fills a provided matrix and right-hand side vector with the data related to the current fem system under consideration and related to this boundary condition.

- virtual void finish_dof_index_initialization ()

  Handles the communication of non-locally owned dofs and thus finishes the setup of the object.

- virtual auto make_constraints () -> Constraints

  Builds a constraint object that represents fixed values of degrees of freedom associated with this object.

- double boundary_norm (NumericVectorDistributed ∗solution)

  Computes the L2-norm of the solution passed in on the shared interface with the interior domain.

- double boundary_surface_norm (NumericVectorDistributed ∗solution, BoundaryId b_id)

  Computes the L2-norm of the solution passed in as an argument on the solution passed in as the second argument.

- virtual unsigned int cells_for_boundary_id (unsigned int boundary_id)

  Counts the number of cells associated with the boundary passed in as an argument.

- void print_dof_validation ()

  In some cases we have more then one option to validate how many dofs a domain should have.

- void force_validation ()

  Triggers the internal validation routine.

- virtual unsigned int n_cells ()

  Counts the number of cells used in the object.

Public Attributes

- const BoundaryId **b_id**
- const unsigned int **level**

- const double **additional_coordinate**
- std::vector< InterfaceDofData > **surface_dofs**
- bool **surface_dof_sorting_done**
- bool **boundary_coordinates_computed** = false
- std::array< double, 6 > **boundary_vertex_coordinates**
- DofCount **dof_counter**
- int **global_partner_mpi_rank**
- int **local_partner_mpi_rank**
- const std::vector< BoundaryId > **adjacent_boundaries**
- std::array< bool, 6 > **are_edge_dofs_owned**
- DofHandler3D **dof_handler**

4.1 Detailed Description

This is the base type for boundary coniditions. Some implementations are done on this level, some in the derived types.

There are several deriveed classes for this type: Dirichlet, Empty, Hardy, PML and Neighbor. Details about them can be found in the derived classes. To the rest of the code, the most relevant functions are:

- Handling the dofs (number of dofs and association to boundaries)
- Assembly (of sparsity pattern and matrices)
- Building constraints

For the boundary numbering, I always use the scheme 0 = -x, 1 = +x, 2 = -y, 3 = +y, 4 = -z and 5 = +z for all domain types. All domains are cuboid, so there are always 6 surfaces in the coordinate orthogonal directions, so the code always considers one interior domain and 6 surfaces, which each need a boundary condition associated with them.

Boundary conditions in this code have three types of surfaces (best visualized with a pml domain, i.e. a FE-domain):

- The surface shared with the inner domain, This is always one.
- The sufaces shared with other boundary conditions, There are always four neighbors since there are always six boundary methods for a domain and the boundary conditions handle the outer sides of this domain like the sides of a cube.
- An outward surface, where dofs only couple with the interior of this boundary condition domain (if that exists).

Similar to all objects in this code, these objects have an initialize function that is implemented in the derived classes. It is important to note, that boundary conditions can introduce their own degrees of freedom to the system assemble and are therefore derived from the abstract base class FEDomain, which basically means they have owned and locally active dofs and these may need to be added to sets of degrees of freedom or handled otherwise.

Definition at line 41 of file BoundaryCondition.h.

4.2 Member Function Documentation

boundary_norm()

```
double BoundaryCondition::boundary_norm (
            NumericVectorDistributed * solution )
```

Computes the L2-norm of the solution passed in on the shared interface with the interior domain.

This function evaluates the provided dof values as a solution on the surface connected to the interior domain. That function is then integrated across the surface as an L2 integral.

Parameters

solution	The provided values of the degrees of freedom related to this boundary condition.

Returns

The function returns the L2 norm of the function computed along the surface connecting the boundary condition with the interior domain.

Definition at line 96 of file BoundaryCondition.cpp.

```
96                                                               {
97    double ret = 0;
98    for(unsigned int i = 0; i < global_index_mapping.size(); i++) {
99      ret += norm_squared(in_v->operator()(global_index_mapping[i]));
100    }
101    return std::sqrt(ret);
102 }
```

boundary_surface_norm()

```
double BoundaryCondition::boundary_surface_norm (
            NumericVectorDistributed * solution,
            BoundaryId b_id )
```

Computes the L2-norm of the solution passed in as an argument on the solution passed in as the second argument.

Thisi function performs the same action as the previous function but does so an an arbitrary surface of the boundary condition instead of only working for the surface facing the interior domain.

Parameters

solution	The values of the degrees of freedom to be used for this computation. These dof values represent an electircal field that can be integrated over the somain surface.
b_id	The boundary id of the surface the function is supposed to integrate across.

Returns

The function returns the L2 norm of the field provided in the solution argument across the surface b_id.

Definition at line 104 of file BoundaryCondition.cpp.

```
104                                                                                                    {
105   double ret = 0;
106   auto dofs = get_dof_association_by_boundary_id(in_bid);
107   for(auto it : dofs) {
108     ret += norm_squared(in_v->operator()(it.index));
109   }
110   return std::sqrt(ret);
111 }
```

References get_dof_association_by_boundary_id().

cells_for_boundary_id()

```
unsigned int BoundaryCondition::cells_for_boundary_id (
             unsigned int boundary_id )  [virtual]
```

Counts the number of cells associated with the boundary passed in as an argument.

It can be useful for testing purposes to count the number of cells forming a certain surface. Imagine if you will a domain discretized by 3 cells in x-direction, 4 in y and 5 in z-direction. The suraces for any combination of 2 directions then have a known number of cells. We can use this knowledge to test if our mesh-coloring algoithms work or not.

Parameters

boundary_id	The boundary we are counting the cells for.

Returns

The number of cells the method found that connect directly with the boundary boundary_id

Reimplemented in PMLSurface.

Definition at line 113 of file BoundaryCondition.cpp.

```
113                                                    {
114   return 0;
115 }
```

fill_matrix()

```
virtual void BoundaryCondition::fill_matrix (
             dealii::PETScWrappers::MPI::SparseMatrix * matrix,
             NumericVectorDistributed * rhs,
             Constraints * constraints )  [pure virtual]
```

Fills a provided matrix and right-hand side vector with the data related to the current fem system under consideration and related to this boundary condition.

Most of a fem code is preparation to assemble a matrix. This function is the last step in that process. Once dofs have been enumerated and materials and geometries setup, this function performs the task of filling a system matrix with the contributions to the set of linear equations. Called after the previous function, this function writes the actual values into the system matrix that were marked as non-zero in the previous function. The same function exists on the InnerDomain object and these objects together build the entire system matrix.

See also

InnerDomain::fill_matrix()

Parameters

matrix	The matrix to fill with the entries related to this object.
rhs	If dofs in this system are inhomogenously constraint (as in the case of Dirichlet data or jump coupling) the system has a non-zero right hand side (in the sense of a linear system A∗x = b). It makes sense to assemble the matrix and the right-hand side together. This is the vector that will store the vector b.
constraints	The constraint object is used to determine values that have a fixed value and to use that information to reduce the memory consumption of the matrix as well as assembling the right-hand side vector.

Implemented in HSIESurface, PMLSurface, NeighborSurface, EmptySurface, and DirichletSurface.

fill_sparsity_pattern()

```
virtual void BoundaryCondition::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * constraints )  [pure virtual]
```

If this object owns degrees of freedom, this function fills a sparsity pattern for their global indices.

The classes local and non-local problem manage matrices to solve either directly or iteratively. Matrices in a HPC setting that are generated from a fem system are usually sparse. A sparsity pattern is an object, that describes in which positions of a matrix there are non-zero entries that require storing. This function updates a given sparsity pattern with the entries related to this object. An important sidemark: In deal.II there are constraint object which store hanging node constraints as well as inhomogenous constraints like Dirichlet data. When filling a matrix, there can sometimes be ways of making use of such constraints and reducing the required memory this way.

See also

deal.II description of sparsity patterns and constraints

Parameters

in_dsp	The sparsity pattern to be updated
constraints	The constraint object that is used to perform this action effectively

Implemented in HSIESurface, PMLSurface, NeighborSurface, EmptySurface, and DirichletSurface.

finish_dof_index_initialization()

`void BoundaryCondition::finish_dof_index_initialization ( ) [virtual]`

Handles the communication of non-locally owned dofs and thus finishes the setup of the object.

In cases where not all locally active dofs are locally owned (for example for two pml domains, the dofs on the shared surface are only owned by one of two processes) this function handles the numbering of the dofs once the non-owned dofs have been communicated.

Reimplemented in HSIESurface, PMLSurface, and NeighborSurface.

Definition at line 87 of file BoundaryCondition.cpp.

```
87                                                          {
88
89 }
```

force_validation()

`void BoundaryCondition::force_validation ( )`

Triggers the internal validation routine.

Prints an error message if invalid.

This is for internal use. It validates if all dofs have a value that is valid in the current scope. Since this is mainly a core implementation concern there is only an error message printed to the console - errors in this code should no longer be occuring.

Definition at line 147 of file BoundaryCondition.cpp.

```
147                                             {
148   if(Geometry.levels[level].surface_type[b_id] != SurfaceType::NEIGHBOR_SURFACE) {
149
150
151     for(unsigned int surf = 0; surf < 6; surf++) {
152         if(surf != b_id && !are_opposing_sites(b_id, surf)) {
153           std::vector<InterfaceDofData> d = get_dof_association_by_boundary_id(surf);
154           bool one_is_invalid = false;
155           unsigned int count_before = 0;
156           unsigned int count_after = 0;
157           for(unsigned int index = 0; index < d.size(); index++) {
158             if(!is_dof_owned[d[index].index]) {
159               if(global_index_mapping[d[index].index] >= Geometry.levels[level].n_total_level_dofs) {
160                 one_is_invalid = true;
161                 count_before ++;
162               }
163             }
164           }
165           if(one_is_invalid) {
166             std::vector<unsigned int> local_indices(d.size());
167             for(unsigned int i = 0; i < d.size(); i++) {
168               local_indices[i] = d[i].index;
169             }
170             set_non_local_dof_indices(local_indices,
       Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id));
171             for(unsigned int index = 0; index < d.size(); index++) {
172               if(!is_dof_owned[d[index].index]) {
173                 if(global_index_mapping[d[index].index] >= Geometry.levels[level].n_total_level_dofs) {
174                   count_after ++;
175                 }
176               }
```

```
            }
          }
        }
    }
  }
}
```

get_boundary_ids()

`std::vector< unsigned int > BoundaryCondition::get_boundary_ids ( ) -> std::vector<BoundaryId>`

Returns a vector of all boundary ids associated with dofs in this domain.

Returns

> The returned vector contains all boundary IDs that are relevant on this domain.

Definition at line 72 of file BoundaryCondition.cpp.

```
72                                                            {
73   return (Geometry.surface_meshes[b_id].get_boundary_ids());
74 }
```

get_dof_association()

`virtual auto BoundaryCondition::get_dof_association ( ) -> std::vector< InterfaceDofData > [pure virtual]`

Returns a vector of all degrees of freedom shared with the inner domain.

For those boundary conditions that generate their own dofs (HSIE, PML and Neighbor) we need to figure out dpf sets that need to be coupled. For example: The PML domain has dofs on the surface shared with the interior domain. These should have the same index as their counterpart in the interior domain. To this goal, we exchange a vector of all dofs on the surface we have previously sorted. That way, we only need to call this function on the interior domain and the boundary method and identify the dofs in the two returned vectors that have the same index.

See also

> InnerDomain::get_surface_dof_vector_for_boundary_id()

Returns

> InterfaceDofData always contains a reference points and index for every index found on the surface. The reference points are used for sorting, the index is the actual data used by the caller.

Implemented in HSIESurface, PMLSurface, EmptySurface, NeighborSurface, and DirichletSurface.

get_dof_association_by_boundary_id()

`virtual auto BoundaryCondition::get_dof_association_by_boundary_id ( BoundaryId in_boundary_id ) -> std::vector< InterfaceDofData > [pure virtual]`

More general version of the function above that can also handle interfaces with other boundary ids.

This function typically holds the actual implementation of the function above as well as implementations for the boundaries shared with other boudnary conditions. It differs in all the derived types.

See also

PMLSurface::get_dof_association_by_boundary_id()

Parameters

boundary_id	This is the boundary id as seen from this domain.

Returns

InterfaceDofData always contains a reference points and index for every index found on the surface. The reference points are used for sorting, the index is the actual data used by the caller.

Implemented in HSIESurface, PMLSurface, EmptySurface, DirichletSurface, and NeighborSurface.

Referenced by boundary_surface_norm(), and get_global_dof_indices_by_boundary_id().

get_global_dof_indices_by_boundary_id()

```
std::vector< DofNumber > BoundaryCondition::get_global_dof_indices_by_boundary_id (
            BoundaryId in_boundary_id ) -> std::vector<DofNumber>  [virtual]
```

Specific version of the function above that provides the indices in the returned vector by their globally unique id instead of local numbering.

Lets say a Boundary Condition has 1000 own degrees of freedom then the method above will return dof ids in the range [0,1000] whereas this function will return the index ids in the numbering relevant to the current sweep of local problem which is globally unique to that problem.

This function performs the same task as the one above but returns the global indices of the dofs instead of the local ones.

See also

get_dof_association()

Parameters

boundary_id	This is the boundary id as seen from this domain.

Returns

At this point, the base_points are no longer required since this function gets called later in the preparation stage. For that reason, this function does not return the base points of the dofs anymore and instead only returns the dof indices. The indices, however, are still in the same order.

Definition at line 76 of file BoundaryCondition.cpp.

```
76                                                                                                      {
77   std::vector<InterfaceDofData> dof_data = get_dof_association_by_boundary_id(in_boundary_id);
78   std::vector<DofNumber> ret;
79   for(unsigned int i = 0; i < dof_data.size(); i++) {
80     ret.push_back(dof_data[i].index);
```

```
  }

  ret = transform_local_to_global_dofs(ret);
  return ret;
}
```

References get_dof_association_by_boundary_id(), and FEDomain::transform_local_to_global_dofs().

initialize()

```
virtual void BoundaryCondition::initialize ( )  [pure virtual]
```

Not all data for objects of this type will be available at time of construction.

This function exists on many objects in this code and handles initialization once all data is configured.

Typically, this function will perform actions like initializing matrices and vectors and enumerating dofs. It is part of the typical pattern Construct -> Initialize -> Run -> Output -> Delete. However, since this is an abstract base class, this function cannot be implemented on this level. No data needs to be passed as an argument and no value is returned. Make sure you understand this function before calling or adapting it on a derived class.

See also

> This function is also often implemented in deal.II examples and derives its name from there.

Implemented in HSIESurface, PMLSurface, EmptySurface, DirichletSurface, and NeighborSurface.

is_point_at_boundary()

```
virtual bool BoundaryCondition::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid )  [pure virtual]
```

Checks if a 2D coordinate is on the a surface of the boundary methods domain.

This function is currently only being used for HSIE. It checks if a point on the interface shared between the inner domain and the boundary method is also at a surface of that boundary, i.e. if this point is also relevant for another boundary method.

See also

> HSIESurface::HSIESurface::get_vertices_for_boundary_id()

Parameters

in_p	The point in the 2D parametrization of the surface.
in_bid	The boundary id of the other boundary condition, for which it should be checked if this point is on it.

Returns

Returns true if this is on such an edge and false if it isn't.

Implemented in HSIESurface, PMLSurface, EmptySurface, DirichletSurface, and NeighborSurface.

make_constraints()

```
Constraints BoundaryCondition::make_constraints ( ) -> Constraints [virtual]
```

Builds a constraint object that represents fixed values of degrees of freedom associated with this object.

For a Dirichlet-data surface, this writes the dirichlet data into the AffineConstraints object. In a PML Surface this writes the zero constraints of the outward surface to the constraint object. Constraint objects can be merged. Therefore this object builds a new one, containing only the constraints related to this boundary contidion. It can then be merged into another one.

Returns

Returns a new constraint object relating only to the current boundary condition to be merged into one for the entire local computation-

Reimplemented in EmptySurface, DirichletSurface, and PMLSurface.

Definition at line 91 of file BoundaryCondition.cpp.

```
91                                                  {
92   Constraints ret(global_dof_indices);
93   return ret;
94 }
```

n_cells()

```
unsigned int BoundaryCondition::n_cells ( ) [virtual]
```

Counts the number of cells used in the object.

For msot derived types, this is the number of 2D surface cells of the inner domain. For PML, however the value is the number of 3D cellx. It is always the number of steps a dof_handler iterates to handle the matrix filling operation.

Returns

The number of cells.

Reimplemented in PMLSurface.

Definition at line 184 of file BoundaryCondition.cpp.

```
184                                         {
185   return 0;
186 }
```

output_results()

```
virtual std::string BoundaryCondition::output_results (
            const dealii::Vector< ComplexNumber > & in_solution,
            std::string filename )  [pure virtual]
```

Writes output for a provided solution to a file with the provided name.

In some cases (currently only the PMLSurface) the boundary condition can have its own mesh and can thus also have data to visualize. As an example of the distinction: For a surface of Dirichlet data (DirichletSurface) all the boundary does is set the degrees of freedom on the surface of the inner domain to the values they should have. As a consequence, the object has no interior mesh and the it can be checked in the output of the inner domain if the boundary method has done its job correctly so no output is required. For a PML domain, however, there is an interior mesh in which the solution is damped. Visual output of the solution in the PML domain can be helpful to understand problems with reflections etc. As a consequence, this function will usually be called on all boundary conditions but most won't perform any tasks.

See also

PMLSurface::output_results()

Parameters

in_solution	This parameter provides the values of the local dofs. In the case of the PMLSurface, these values are the computed E-field on the degrees of freedom that are active in the PMLDomain, i.e. have support in the PML domain.
filename	The output will typically be written to a paraview-compatible format like .vtk and .vtu. This string does not contain the file endings. So if you want to write to a file solution.vtk you would only provide "solution".

Returns

> This function returns the complete filename to which it has written the data. This can be used by the caller to generate meta-files for paraview which load for example the solution on the interior and all adjacent pml domains together.

Implemented in PMLSurface, EmptySurface, DirichletSurface, NeighborSurface, and HSIESurface.

print_dof_validation()

`void BoundaryCondition::print_dof_validation ( )`

In some cases we have more then one option to validate how many dofs a domain should have.

This is one way of computng that value for comparison with numbers that arise from the compuataion directly.

This is an internal function and should be used with caution. The function only warns the user. It does not abort the execution.

Definition at line 117 of file BoundaryCondition.cpp.

```
117                                              {
118   unsigned int n_invalid_dofs = 0;
119   for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
120     if(global_index_mapping[i] >= Geometry.levels[level].n_total_level_dofs) {
121       n_invalid_dofs++;
122     }
123   }
124   if(n_invalid_dofs > 0) {
125     std::cout « "On process " « GlobalParams.MPI_Rank « " surface " « b_id « " has " « n_invalid_dofs «
      " invalid dofs." « std::endl;
126     for(unsigned int surf = 0; surf < 6; surf++) {
127       if(surf != b_id && !are_opposing_sites(b_id, surf)) {
128         unsigned int invalid_dof_count = 0;
129         unsigned int owned_invalid = 0;
130         auto dofs = get_dof_association_by_boundary_id(surf);
131         for(auto dof:dofs) {
132           if(global_index_mapping[dof.index] >= Geometry.levels[level].n_total_level_dofs) {
133             invalid_dof_count++;
134             if(is_dof_owned[dof.index]) {
135               owned_invalid++;
136             }
137           }
138         }
139         if(invalid_dof_count > 0) {
140           std::cout « "On process " « GlobalParams.MPI_Rank « " surface " « b_id « " there were "«
      invalid_dof_count « "(" « owned_invalid « ") invalid dofs towards "« surf « std::endl;
141         }
142       }
143     }
144   }
145 }
```

set_mesh_boundary_ids()

`void BoundaryCondition::set_mesh_boundary_ids ( )`

If the boundary condition has its own mesh, this function iterates over the mesh and sets boundary ids on the mesh.

Consider, as an example, a PML domain. For such a domain we have one surface facing the inner domain, 4 surfaces facing other boundary conditions and the remainder of the boundary condition faces outward. All of these surfaces have to be dealt with individually. On the boundary facing the interior we need to identify the dofs with their equivalent dofs on the interior domain. On durfaces shared with other boundary conditions we have to decide on ownership and set them properly (if the other boundary condition is a Dirichlet Boundary, for example, we need to enforce a PML-damped dirichlet data. If it is a neighbor surface, we need to perform communication with the neighbor. etc.) For the outward surface on the other hand we need to set metallic boundary conditions. To make these actions more efficient, we set boundary ids on the cells, so after that we can simply derive the operation required on a cell by asking for its boundary id and we can also simply get all dofs that require a certain action simply by their boundary id.

See also

PMLSurface::set_mesh_boundary_ids()

Definition at line 22 of file BoundaryCondition.cpp.

```
22                                                 {
23      auto it = Geometry.surface_meshes[b_id].begin_active();
24      std::vector<double> x;
25      std::vector<double> y;
26      while(it != Geometry.surface_meshes[b_id].end()){
27        if(it->at_boundary()) {
28          for (unsigned int face = 0; face < GeometryInfo<2>::faces_per_cell; ++face) {
29            if (it->face(face)->at_boundary()) {
30              dealii::Point<2, double> c;
31              c = it->face(face)->center();
32              x.push_back(c[0]);
33              y.push_back(c[1]);
34            }
35          }
36        }
37        ++it;
38      }
39      double x_max = *max_element(x.begin(), x.end());
40      double y_max = *max_element(y.begin(), y.end());
41      double x_min = *min_element(x.begin(), x.end());
42      double y_min = *min_element(y.begin(), y.end());
43      it = Geometry.surface_meshes[b_id].begin_active();
44      while(it != Geometry.surface_meshes[b_id].end()){
45      if (it->at_boundary()) {
46        for (unsigned int face = 0; face < dealii::GeometryInfo<2>::faces_per_cell;
47            ++face) {
48          Point<2, double> center;
49          center = it->face(face)->center();
50          if (std::abs(center[0] - x_min) < 0.0001) {
51            it->face(face)->set_all_boundary_ids(
52                edge_to_boundary_id[this->b_id][0]);
53          }
54          if (std::abs(center[0] - x_max) < 0.0001) {
55            it->face(face)->set_all_boundary_ids(
56                edge_to_boundary_id[this->b_id][1]);
57          }
58          if (std::abs(center[1] - y_min) < 0.0001) {
59            it->face(face)->set_all_boundary_ids(
60                edge_to_boundary_id[this->b_id][2]);
61          }
62          if (std::abs(center[1] - y_max) < 0.0001) {
63            it->face(face)->set_all_boundary_ids(
64                edge_to_boundary_id[this->b_id][3]);
65          }
66          }
67      }
68      ++it;
69    }
70 }
```

The documentation for this class was generated from the following files:

- Code/BoundaryCondition/BoundaryCondition.h
- Code/BoundaryCondition/BoundaryCondition.cpp

5 BoundaryInformation Struct Reference

Public Member Functions

- **BoundaryInformation** (unsigned int in_coord, bool neg)

Public Attributes

- unsigned int **inner_coordinate**
- bool **negate_value**

5.1 Detailed Description

Definition at line 127 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

6 CellAngelingData Struct Reference

Public Attributes

- EdgeAngelingData **edge_data**
- VertexAngelingData **vertex_data**

6.1 Detailed Description

Definition at line 86 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

7 CellwiseAssemblyData Struct Reference

Public Member Functions

- **CellwiseAssemblyData** (dealii::FE_NedelecSZ< 3 > ∗fe, DofHandler3D ∗dof_handler)
- void **prepare_for_current_q_index** (unsigned int q_index)
- Tensor< 1, 3, ComplexNumber > **Conjugate_Vector** (Tensor< 1, 3, ComplexNumber > input)

Public Attributes

- QGauss< 3 > **quadrature_formula**
- FEValues< 3 > **fe_values**
- std::vector< Position > **quadrature_points**
- const unsigned int **dofs_per_cell**
- const unsigned int **n_q_points**
- FullMatrix< ComplexNumber > **cell_mass_matrix**
- FullMatrix< ComplexNumber > **cell_stiffness_matrix**
- dealii::Vector< ComplexNumber > **cell_rhs**
- const double **eps_in**
- const double **eps_out**
- const double **mu_zero**
- MaterialTensor **transformation**
- MaterialTensor **epsilon**
- MaterialTensor **mu**
- std::vector< DofNumber > **local_dof_indices**
- DofHandler3D::active_cell_iterator **cell**
- DofHandler3D::active_cell_iterator **end_cell**
- const FEValuesExtractors::Vector **fe_field**

7.1 Detailed Description

Definition at line 166 of file RectangularMode.cpp.

The documentation for this struct was generated from the following file:

- Code/ModalComputations/RectangularMode.cpp

8 CellwiseAssemblyDataNP Struct Reference

Public Member Functions

- **CellwiseAssemblyDataNP** (dealii::FE_NedelecSZ< 3 > ∗fe, DofHandler3D ∗dof_handler)
- void **set_es_pointer** (ExactSolution ∗in_es)
- void **prepare_for_current_q_index** (unsigned int q_index)
- Tensor< 1, 3, ComplexNumber > **Conjugate_Vector** (Tensor< 1, 3, ComplexNumber > input)
- Tensor< 1, 3, ComplexNumber > **evaluate_J_at** (Position p)

Public Attributes

- QGauss< 3 > **quadrature_formula**
- FEValues< 3 > **fe_values**

- std::vector< Position > **quadrature_points**
- const unsigned int **dofs_per_cell**
- const unsigned int **n_q_points**
- FullMatrix< ComplexNumber > **cell_matrix**
- const double **eps_in**
- const double **eps_out**
- const double **mu_zero**
- Vector< ComplexNumber > **cell_rhs**
- MaterialTensor **transformation**
- MaterialTensor **epsilon**
- MaterialTensor **mu**
- std::vector< DofNumber > **local_dof_indices**
- DofHandler3D::active_cell_iterator **cell**
- DofHandler3D::active_cell_iterator **end_cell**
- bool **has_input_interface** = false
- const FEValuesExtractors::Vector **fe_field**
- Vector< ComplexNumber > **incoming_wave_field**
- IndexSet **constrained_dofs**
- Tensor< 1, 3, ComplexNumber > **J**
- ExactSolution ∗ **es_for_j**

8.1 Detailed Description

Definition at line 160 of file InnerDomain.cpp.

The documentation for this struct was generated from the following file:

- Code/Core/InnerDomain.cpp

9 CellwiseAssemblyDataPML Struct Reference

Public Member Functions

- **CellwiseAssemblyDataPML** (dealii::FE_NedelecSZ< 3 > ∗fe, DofHandler3D ∗dof_handler)
- Position **get_position_for_q_index** (unsigned int q_index)
- void **prepare_for_current_q_index** (unsigned int q_index, dealii::Tensor< 2, 3, ComplexNumber > epsilon, dealii::Tensor< 2, 3, ComplexNumber > mu_inverse)
- Tensor< 1, 3, ComplexNumber > **Conjugate_Vector** (Tensor< 1, 3, ComplexNumber > input)

Public Attributes

- QGauss< 3 > **quadrature_formula**
- FEValues< 3 > **fe_values**

- std::vector< Position > **quadrature_points**
- const unsigned int **dofs_per_cell**
- const unsigned int **n_q_points**
- FullMatrix< ComplexNumber > **cell_matrix**
- Vector< ComplexNumber > **cell_rhs**
- std::vector< DofNumber > **local_dof_indices**
- DofHandler3D::active_cell_iterator **cell**
- DofHandler3D::active_cell_iterator **end_cell**
- const FEValuesExtractors::Vector **fe_field**

9.1 Detailed Description

Definition at line 385 of file PMLSurface.cpp.

The documentation for this struct was generated from the following file:

- Code/BoundaryCondition/PMLSurface.cpp

10 ConstraintPair Struct Reference

Public Attributes

- unsigned int **left**
- unsigned int **right**
- bool **sign**

10.1 Detailed Description

Definition at line 211 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

11 ConvergenceOutputGenerator Class Reference

Public Member Functions

- void **set_title** (std::string in_title)
- void **set_labels** (std::string x_label, std::string y_label)
- void **push_values** (double x, double y_num, double y_theo)
- void **write_gnuplot_file** ()
- void **run_gnuplot** ()

11.1 Detailed Description

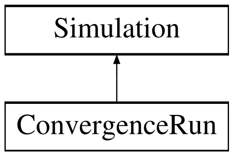
Definition at line 5 of file ConvergenceOutputGenerator.h.

The documentation for this class was generated from the following files:

- Code/OutputGenerators/Images/ConvergenceOutputGenerator.h
- Code/OutputGenerators/Images/ConvergenceOutputGenerator.cpp

12 ConvergenceRun Class Reference

Inheritance diagram for ConvergenceRun:



Public Member Functions

- ConvergenceRun ()

  Construct a new Convergence Run object The constructor does nothing.
- void prepare () override

  Solve the reference problem and setup the others.
- void run () override

  Solves the coarser problems and computes their theoretical and numerical error.
- void write_outputs ()

  Writes the results of the convergence study to the command line.
- void prepare_transformed_geometry () override

  Not implemented / not required here.
- void set_norming_factor ()

  Computes and stores the max vector component of the reference solutions norm.
- double compute_error_for_two_eval_vectors (std::vector< std::vector< ComplexNumber >> a, std::vector< std::vector< ComplexNumber >> b)

  Computes the L2 difference of two solutions, i.e.

12.1 Detailed Description

Definition at line 9 of file ConvergenceRun.h.

12.2 Member Function Documentation

compute_error_for_two_eval_vectors()

```
double ConvergenceRun::compute_error_for_two_eval_vectors (
            std::vector< std::vector< ComplexNumber >> a,
            std::vector< std::vector< ComplexNumber >> b )
```

Computes the L2 difference of two solutions, i.e.

the reference solution and another one. As a consequence the order of the provided vectors does not matter.

Parameters

a	first solution vector
b	other solution vector

Returns

double L2 norm of the difference.

Definition at line 141 of file ConvergenceRun.cpp.

```
141                               {
142   double local = 0.0;
143   for(unsigned int i = 0; i < a.size(); i++) {
144     double x = std::abs(a[i][0] - b[i][0]);
145     double y = std::abs(a[i][1] - b[i][1]);
146     double z = std::abs(a[i][2] - b[i][2]);
147     local += std::sqrt(x*x + y*y + z*z);
148   }
149   local /= evaluation_positions.size();
150   local *= (Geometry.local_x_range.second - Geometry.local_x_range.first) *
       (Geometry.local_y_range.second - Geometry.local_y_range.first) * (Geometry.local_z_range.second -
       Geometry.local_z_range.first);
151   double ret = dealii::Utilities::MPI::sum(local, MPI_COMM_WORLD);
152   ret /= norming_factor;
153   return ret;
154 }
```

prepare()

```
void ConvergenceRun::prepare ( )  [override], [virtual]
```

Solve the reference problem and setup the others.

In a convergence run we have the reference solution on the finest grid and then a set of other sizes as the actual data. This function solves the reference problem and prepares the others.

Implements Simulation.

Definition at line 39 of file ConvergenceRun.cpp.

```
39                              {
40   print_info("ConvergenceRun::prepare", "Start", LoggingLevel::DEBUG_ONE);
41   GlobalParams.Cells_in_x = GlobalParams.convergence_max_cells;
42   GlobalParams.Cells_in_y = GlobalParams.convergence_max_cells;
43   GlobalParams.Cells_in_z = GlobalParams.convergence_max_cells;
44   Geometry.initialize();
45   mainProblem = new NonLocalProblem(GlobalParams.Sweeping_Level);
46   mainProblem->initialize();
47   for(auto it = Geometry.levels[0].inner_domain->triangulation.begin_active(); it !=
       Geometry.levels[0].inner_domain->triangulation.end(); it++) {
```

```
    evaluation_positions.push_back(it->center());
  }
  for(unsigned int i = 0; i < evaluation_positions.size(); i++) {
    NumericVectorLocal local_solution(3);
    GlobalParams.source_field->vector_value(evaluation_positions[i], local_solution);
    std::vector<ComplexNumber> local_solution_vector;
    for(unsigned int j = 0; j < 3; j++) {
      local_solution_vector.push_back(local_solution[j]);
    }
    evaluation_exact_solution.push_back(local_solution_vector);
  }
  mainProblem->assemble();
  mainProblem->compute_solver_factorization();
  mainProblem->solve_with_timers_and_count();
  mainProblem->output_results();
  mainProblem->empty_memory();
  base_problem_n_dofs = mainProblem->compute_total_number_of_dofs();
  base_problem_n_cells = mainProblem->n_total_cells();
  base_problem_h = mainProblem->compute_h();
  evaluation_base_problem = mainProblem->evaluate_solution_at(evaluation_positions);
  base_problem_theoretical_error = compute_error_for_two_eval_vectors(evaluation_base_problem,
    evaluation_exact_solution);
  delete mainProblem;
  print_info("ConvergenceRun::prepare", "End", LoggingLevel::DEBUG_ONE);
}
```

run()

`void ConvergenceRun::run ( )  [override], [virtual]`

Solves the coarser problems and computes their theoretical and numerical error.

Then calls write_outputs().

Implements Simulation.

Definition at line 73 of file ConvergenceRun.cpp.

```
73                          {
74     print_info("ConvergenceRun::run", "Start", LoggingLevel::PRODUCTION_ONE);
75     for(unsigned int run_index = 0; run_index < GlobalParams.convergence_cell_counts.size()-1;
       run_index++) {
76       GlobalParams.Cells_in_x = GlobalParams.convergence_cell_counts[run_index];
77       GlobalParams.Cells_in_y = GlobalParams.convergence_cell_counts[run_index];
78       GlobalParams.Cells_in_z = GlobalParams.convergence_cell_counts[run_index];
79       Geometry.initialize();
80       otherProblem = new NonLocalProblem(GlobalParams.Sweeping_Level);
81       otherProblem->initialize();
82       otherProblem->assemble();
83       otherProblem->compute_solver_factorization();
84       otherProblem->solve_with_timers_and_count();
85       std::vector<std::vector<ComplexNumber» other_evaluations =
       otherProblem->evaluate_solution_at(evaluation_positions);
86       double numerical_error = compute_error_for_two_eval_vectors(evaluation_base_problem,
       other_evaluations);
87       double theoretical_error = compute_error_for_two_eval_vectors(evaluation_exact_solution,
       other_evaluations);
88       numerical_errors.push_back(numerical_error);
89       theoretical_errors.push_back(theoretical_error);
90       std::string msg = "Result: " + std::to_string(GlobalParams.convergence_cell_counts[run_index]) + "
       found numerical error " + std::to_string(numerical_error) + "and theoretical error " +
       std::to_string(theoretical_error);
91       print_info("ConvergenceRun::run", msg , LoggingLevel::PRODUCTION_ONE);
92       unsigned int temp_ndofs = otherProblem->compute_total_number_of_dofs();
93       n_dofs_for_cases.push_back(temp_ndofs);
94       h_values.push_back(otherProblem->compute_h());
95       total_cells.push_back(otherProblem->n_total_cells());
96       output.push_values(temp_ndofs,numerical_error,theoretical_error);
97       otherProblem->empty_memory();
```

```
    }
    write_outputs();

     print_info("ConvergenceRun::run", "End", LoggingLevel::PRODUCTION_ONE);
}
```

The documentation for this class was generated from the following files:

- Code/Runners/ConvergenceRun.h
- Code/Runners/ConvergenceRun.cpp

13 CoreLogger Class Reference

Outputs I want:

```
#include <CoreLogger.h>
```

13.1 Detailed Description

Outputs I want:

- Timing output for all solver runs on any level.
- Convergence histories for any solver run on any level (except the lowest one maybe, bc. thats direct).
- Convergence rates
- Dof Numbers on all levels
- Memory Consumption of the direct solver

So this object mainly manages run meta-information. It needs functions that register which run the code is on (which iteration on which level etc.) There will only be one instance of this object and it will be available globally. It should use the FileLogger global instance to create files.

Definition at line 18 of file CoreLogger.h.

The documentation for this class was generated from the following file:

- Code/OutputGenerators/Console/CoreLogger.h

14 DataSeries Struct Reference

Public Attributes

- std::vector< double > **values**
- bool **is_closed**
- std::string **name**

14.1 Detailed Description

Definition at line 222 of file Types.h.

The documentation for this struct was generated from the following file:

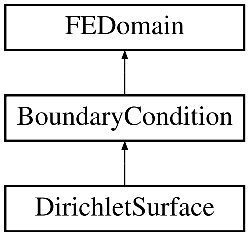
- Code/Core/Types.h

15 DirichletSurface Class Reference

This class implements dirichlet data on the given surface.

```
#include <DirichletSurface.h>
```

Inheritance diagram for DirichletSurface:



Public Member Functions

- **DirichletSurface** (unsigned int in_bid, unsigned int in_level)
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix ∗matrix, NumericVectorDistributed ∗rhs, Constraints ∗constraints) override

  Fill a system matrix.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗in_constraints) override

  Fill the sparsity pattern.
- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

  Checks if a 2D surface coordinate is on a surface of not.
- void initialize () override

  Performs initialization of datastructures.
- auto get_dof_association () -> std::vector< InterfaceDofData > override

  returns an empty array.
- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

  returns an empty array.
- std::string output_results (const dealii::Vector< ComplexNumber > &solution, std::string filename) override

  Would write output but this function has no own data to store.
- DofCount compute_n_locally_owned_dofs () override

  Computes the number of degrees of freedom that this surface owns which is 0 for dirichlet surfaces.
- DofCount compute_n_locally_active_dofs () override

There are active dofs on this surface.

- void determine_non_owned_dofs () override

  Only exists for the interface.

- auto make_constraints () -> Constraints override

  Writes the dirichlet data into a new constraint object and returns it.

Additional Inherited Members

15.1 Detailed Description

This class implements dirichlet data on the given surface.

This class is a simple derived function from the boundary condition base class. Since dirichlet constraints introduce no new degrees of freedom, the functions like fill_matrix don't do anything.

The only relevant function here is the make_constraints function which writes the dirichlet constraints into the given constraints object.

Definition at line 29 of file DirichletSurface.h.

15.2 Member Function Documentation

compute_n_locally_active_dofs()

`DofCount DirichletSurface::compute_n_locally_active_dofs ( ) [override], [virtual]`

There are active dofs on this surface.

However, Dirichlet surfaces never interact with them (Dirichlet surfaces are only active in the phase when constraints are built, bot when matrices are assembled or solutions written to an output). As a consequence, the output of this function is 0.

Returns 0. See class description.

Returns

0.

Implements FEDomain.

Definition at line 64 of file DirichletSurface.cpp.

```
64                                                             {
65     return 0;
66 }
```

compute_n_locally_owned_dofs()

`DofCount DirichletSurface::compute_n_locally_owned_dofs ( ) [override], [virtual]`

Computes the number of degrees of freedom that this surface owns which is 0 for dirichlet surfaces.

Returns 0. See class description.

Returns

0.

Implements FEDomain.

Definition at line 60 of file DirichletSurface.cpp.

```
60 {
61     return 0;
62 }
```

determine_non_owned_dofs()

`void DirichletSurface::determine_non_owned_dofs ( ) [override], [virtual]`

Only exists for the interface.

Does nothing.

The surface owns no dofs.

Implements FEDomain.

Definition at line 68 of file DirichletSurface.cpp.

```
68 {
69
70 }
```

fill_matrix()

```
void DirichletSurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints ) [override], [virtual]
```

Fill a system matrix.

See class description.

See also

DirichletSurface::make_constraints()

Parameters

matrix	only for the interface
rhs	only for the interface
constraints	only for the interface

Implements BoundaryCondition.

Definition at line 31 of file DirichletSurface.cpp.

```
                    {
    matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
   required.
   // Nothing to do here, work happens on neighbor process.
}
```

fill_sparsity_pattern()

```
void DirichletSurface::fill_sparsity_pattern (
             dealii::DynamicSparsityPattern * in_dsp,
             Constraints * in_constraints )  [override], [virtual]
```

Fill the sparsity pattern.

See class description.

See also

DirichletSurface::make_constratints()

Parameters

in_dsp	the sparsity pattern to fill
in_constraints	the constraint object to be considered when writing the sparsity pattern

Implements BoundaryCondition.

Definition at line 58 of file DirichletSurface.cpp.

```
58 { }
```

get_dof_association()

```
std::vector< InterfaceDofData > DirichletSurface::get_dof_association ( ) -> std::vector<InterfaceDofData>
[override], [virtual]
```

returns an empty array.

While this boundary condition does influence some degree of freedom values, it does not own any. Surface dofs are always owned by the interior domain and dirichlet surfaces introduce no artificial dofs like HSIE or PML. As a consequence, this object does not store any dof data at all and instead gets a vector of surface dofs from the interior when required.

Returns

The returned array is empty.

Implements BoundaryCondition.

Definition at line 44 of file DirichletSurface.cpp.

```
44                                                           {
45     std::vector<InterfaceDofData> ret;
46     return ret;
47 }
```

get_dof_association_by_boundary_id()

```
std::vector< InterfaceDofData > DirichletSurface::get_dof_association_by_boundary_id (
            BoundaryId in_boundary_id ) -> std::vector<InterfaceDofData>  [override], [virtual]
```

returns an empty array.

See function above.

See also

get_dof_association()

Parameters

in_boundary_id	NOT USED.

Returns

empty vector of InterfaceDofData type because this boundary condition has no own degrees of freedom.

Implements BoundaryCondition.

Definition at line 49 of file DirichletSurface.cpp.

```
49                                                                                    {
50     std::vector<InterfaceDofData> ret;
51     return ret;
52 }
```

initialize()

```
void DirichletSurface::initialize ( )  [override], [virtual]
```

Performs initialization of datastructures.

See the description in the base class.

Implements BoundaryCondition.

Definition at line 40 of file DirichletSurface.cpp.

```
40                                 {
41
42 }
```

is_point_at_boundary()

```
bool DirichletSurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid )  [override], [virtual]
```

Checks if a 2D surface coordinate is on a surface of not.

See the description in the base class.

Parameters

in_p	the position to be checked
in_bid	This function does NOT return the boundary the point is on. Instead, it checks if it is on the boundary provided in this argument and returns true or false

Returns

boolean indicating if the provided position is on the provided surface

Implements BoundaryCondition.

Definition at line 36 of file DirichletSurface.cpp.

```
36                                                                        {
37      return false;
38 }
```

make_constraints()

`Constraints DirichletSurface::make_constraints ( ) -> Constraints  [override], [virtual]`

Writes the dirichlet data into a new constraint object and returns it.

This is the only function on this type that does something. It projects the prescribed boundary values onto the inner domains surface and builds a AffineConstraints<ComplexNumber> object from the resulting values. The object it returns can be merged with other objects of the same type to build the global constraint object.

Returns

A constraint object representing the dirichlet data.

Reimplemented from BoundaryCondition.

Definition at line 72 of file DirichletSurface.cpp.

```
72                                                    {
73      Constraints ret(Geometry.levels[level].inner_domain->global_dof_indices);
74      dealii::IndexSet local_dof_set(Geometry.levels[level].inner_domain->n_locally_active_dofs);
75      local_dof_set.add_range(0,Geometry.levels[level].inner_domain->n_locally_active_dofs);
76      AffineConstraints<ComplexNumber> constraints_local(local_dof_set);
77
        VectorTools::project_boundary_values_curl_conforming_l2(Geometry.levels[level].inner_domain->dof_handler,
        0, *GlobalParams.source_field, b_id, constraints_local);
78      for(auto line : constraints_local.get_lines()) {
79          const unsigned int local_index = line.index;
80          const unsigned int global_index =
        Geometry.levels[level].inner_domain->global_index_mapping[local_index];
81          ret.add_line(global_index);
82          ret.set_inhomogeneity(global_index, line.inhomogeneity);
83      }
84      constraints_local.clear();
85      if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
86          for(unsigned int surf = 0; surf < 6; surf++) {
87              if(surf != b_id && !are_opposing_sites(b_id, surf)) {
88                  if(Geometry.levels[level].surface_type[surf] == SurfaceType::ABC_SURFACE) {
89                      PMLTransformedExactSolution ptes(b_id, additional_coordinate);
90
        VectorTools::project_boundary_values_curl_conforming_l2(Geometry.levels[level].surfaces[surf]->dof_handler,
        0, ptes, b_id, constraints_local);
```

```
                    for(auto line : constraints_local.get_lines()) {
                        const unsigned int local_index = line.index;
                        const unsigned int global_index =
      Geometry.levels[level].surfaces[surf]->global_index_mapping[local_index];
                        ret.add_line(global_index);
                        ret.set_inhomogeneity(global_index, line.inhomogeneity);
                    }
                    constraints_local.clear();
                }
            }
        }
    }
    return ret;
}
```

output_results()

```
std::string DirichletSurface::output_results (
            const dealii::Vector< ComplexNumber > & solution,
            std::string filename )  [override], [virtual]
```

Would write output but this function has no own data to store.

This function performs no actions. See class and base class description for details.

Parameters

solution	NOT USED.
filename	NOT USED.

Returns

Implements BoundaryCondition.

Definition at line 54 of file DirichletSurface.cpp.

```
54                                                                                          {
55     return "";
56 }
```

The documentation for this class was generated from the following files:

- Code/BoundaryCondition/DirichletSurface.h
- Code/BoundaryCondition/DirichletSurface.cpp

16 DofAssociation Struct Reference

Public Attributes

- bool **is_edge**
- DofNumber **edge_index**
- std::string **face_index**

- DofNumber **dof_index_on_hsie_surface**
- Position **base_point**
- bool **true_orientation**

16.1 Detailed Description

Definition at line 159 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

17 DofCountsStruct Struct Reference

Public Attributes

- unsigned int **hsie** = 0
- unsigned int **non_hsie** = 0
- unsigned int **total** = 0

17.1 Detailed Description

Definition at line 174 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

18 DofCouplingInformation Struct Reference

Public Attributes

- DofNumber **first_dof**
- DofNumber **second_dof**
- double **coupling_value**

18.1 Detailed Description

Definition at line 137 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

19 DofData Struct Reference

This struct is used to store data about degrees of freedom for Hardy space infinite elements. This datatype is somewhat internal and should not require additional work.

`#include <DofData.h>`

Public Member Functions

- void **set_base_dof** (unsigned int in_base_dof_index)
- **DofData** (std::string in_id)
- **DofData** (unsigned int in_id)
- auto **update_nodal_basis_flag** () -> void

Public Attributes

- DofType **type**
- int **hsie_order**
- int **inner_order**
- bool **nodal_basis**
- unsigned int **global_index**
- bool **got_base_dof_index**
- unsigned int **base_dof_index**
- std::string **base_structure_id_face**
- unsigned int **base_structure_id_non_face**
- bool **orientation** = true

19.1 Detailed Description

This struct is used to store data about degrees of freedom for Hardy space infinite elements. This datatype is somewhat internal and should not require additional work.

Definition at line 24 of file DofData.h.

The documentation for this struct was generated from the following file:

- Code/BoundaryCondition/DofData.h

20 DofIndexData Class Reference

Public Member Functions

- void **communicateSurfaceDofs** ()
- void **initialize** ()
- void **initialize_level** (unsigned int level)

Public Attributes

- bool ∗ **isSurfaceNeighbor**
- std::vector< LevelDofIndexData > **indexCountsByLevel**

20.1 Detailed Description

Definition at line 6 of file DofIndexData.h.

The documentation for this class was generated from the following files:

- Code/Hierarchy/DofIndexData.h
- Code/Hierarchy/DofIndexData.cpp

21 DofOwner Struct Reference

Public Attributes

- unsigned int **owner** = 0
- bool **is_boundary_dof** = false
- unsigned int **surface_id** = 0

21.1 Detailed Description

Definition at line 91 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

22 EdgeAngelingData Struct Reference

Public Attributes

- unsigned int **edge_index**
- bool **angled_in_x** = false
- bool **angled_in_y** = false

22.1 Detailed Description

Definition at line 74 of file Types.h.

The documentation for this struct was generated from the following file:

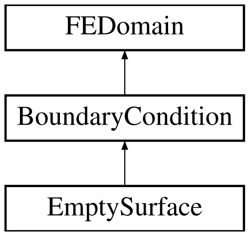
- Code/Core/Types.h

23 EmptySurface Class Reference

A surface with tangential component of the solution equals zero, i.e. specialization of the dirichlet surface.

```
#include <EmptySurface.h>
```

Inheritance diagram for EmptySurface:



Public Member Functions

- **EmptySurface** (unsigned int in_bid, unsigned int in_level)
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix *matrix, NumericVectorDistributed *rhs, Constraints *constraints) override

  Fill a system matrix.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern *in_dsp, Constraints *in_constraints) override

  Fill the sparsity pattern.
- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

  Checks if a 2D surface coordinate is on a surface of not.
- void initialize () override

  Performs initialization of datastructures.
- auto get_dof_association () -> std::vector< InterfaceDofData > override

  returns an empty array.
- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

  returns an empty array.
- std::string output_results (const dealii::Vector< ComplexNumber > &solution, std::string filename) override

  Would write output but this function has no own data to store.
- DofCount compute_n_locally_owned_dofs () override

  Computes the number of degrees of freedom that this surface owns which is 0 for empty surfaces.
- DofCount compute_n_locally_active_dofs () override

  There are active dofs on this surface.
- void determine_non_owned_dofs () override

  Only exists for the interface.

- auto make_constraints () -> Constraints override

  Writes the constraints of locally active being equal to zero into a contstrint object and returns it.

Additional Inherited Members

23.1 Detailed Description

A surface with tangential component of the solution equals zero, i.e. specialization of the dirichlet surface.

This is a DirichletSurface with a predefined soltuion to enforce - namely zero, i.e. a PEC boundary condition. It is used in the sweeping preconditioning scheme where the lower boundary dofs of all domains except the lowest in sweeping direction are set to zero to compute the rhs that acurately describes the signal propagating across the interface. The implementation is extremely simple because most functions perform no tasks at all and the make_constraints() function is a simplified version of the version in DirichletSurface. The members of this class are therefore not documented. See the documentation in the base class for more details.

See also

DirichletSurface, BoundaeyCondition

Definition at line 30 of file EmptySurface.h.

23.2 Member Function Documentation

compute_n_locally_active_dofs()

`DofCount EmptySurface::compute_n_locally_active_dofs ( ) [override], [virtual]`

There are active dofs on this surface.

However, empty surfaces never interact with them (Empty surfaces are only active in the phase when constraints are built, bot when matrices are assembled or solutions written to an output). As a consequence, the output of this function is 0.

Returns 0. See class description.

Returns

0.

Implements FEDomain.

Definition at line 63 of file EmptySurface.cpp.

```
63                                                          {
64     return 0;
65 }
```

compute_n_locally_owned_dofs()

`DofCount EmptySurface::compute_n_locally_owned_dofs ( ) [override], [virtual]`

Computes the number of degrees of freedom that this surface owns which is 0 for empty surfaces.

Returns 0. See class description.

Returns

0.

Implements FEDomain.

Definition at line 59 of file EmptySurface.cpp.

```
59                                                       {
60     return 0;
61 }
```

determine_non_owned_dofs()

`void EmptySurface::determine_non_owned_dofs ( ) [override], [virtual]`

Only exists for the interface.

Does nothing.

The surface owns no dofs.

Implements FEDomain.

Definition at line 67 of file EmptySurface.cpp.

```
67                                                {
68
69 }
```

fill_matrix()

```
void EmptySurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints ) [override], [virtual]
```

Fill a system matrix.

See class description.

See also

EmptySurface::make_constraints()

Parameters

matrix	only for the interface
rhs	only for the interface
constraints	only for the interface

Implements BoundaryCondition.

Definition at line 30 of file EmptySurface.cpp.

```
30                {
31      matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
        required.
32      // Nothing to do here, work happens on neighbor process.
33 }
```

fill_sparsity_pattern()

```
void EmptySurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * in_constraints )  [override], [virtual]
```

Fill the sparsity pattern.

See class description.

See also

EmptySurface::make_constratints()

Parameters

in_dsp	the sparsity pattern to fill
in_constraints	the constraint object to be considered when writing the sparsity pattern

Implements BoundaryCondition.

Definition at line 57 of file EmptySurface.cpp.

```
57 { }
```

get_dof_association()

```
std::vector< InterfaceDofData > EmptySurface::get_dof_association ( ) -> std::vector<InterfaceDofData>
[override], [virtual]
```

returns an empty array.

While this boundary condition does influence some degree of freedom values, it does not own any. Surface dofs are always owned by the interior domain and dirichlet surfaces introduce no artificial dofs like HSIE or PML. As a consequence, this object does not store any dof data at all and instead gets a vector of surface dofs from the interior when required.

Returns

The returned array is empty.

Implements BoundaryCondition.

Definition at line 43 of file EmptySurface.cpp.

```
43                                                                {
```

```
44     std::vector<InterfaceDofData> ret;
45     return ret;
46 }
```

get_dof_association_by_boundary_id()

```
std::vector< InterfaceDofData > EmptySurface::get_dof_association_by_boundary_id (
            BoundaryId in_boundary_id ) -> std::vector<InterfaceDofData>  [override], [virtual]
```

returns an empty array.

See function above.

See also

get_dof_association()

Parameters

in_boundary_id	NOT USED.

Returns

empty vector of InterfaceDofData type because this boundary condition has no own degrees of freedom.

Implements BoundaryCondition.

Definition at line 48 of file EmptySurface.cpp.

```
48                                                                                  {
49     std::vector<InterfaceDofData> ret;
50     return ret;
51 }
```

initialize()

```
void EmptySurface::initialize ( )  [override], [virtual]
```

Performs initialization of datastructures.

Does nothing for this version of a boundary condition.

See the description in the base class.

Implements BoundaryCondition.

Definition at line 39 of file EmptySurface.cpp.

```
39                            {
40
41 }
```

is_point_at_boundary()

```
bool EmptySurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid ) [override], [virtual]
```

Checks if a 2D surface coordinate is on a surface of not.

See the description in the base class.

Parameters

in_p	the position to be checked
in_bid	This function does NOT return the boundary the point is on. Instead, it checks if it is on the boundary provided in this argument and returns true or false

Returns

boolean indicating if the provided position is on the provided surface

Implements BoundaryCondition.

Definition at line 35 of file EmptySurface.cpp.

```
35                                                        {
36     return false;
37 }
```

make_constraints()

```
Constraints EmptySurface::make_constraints ( ) -> Constraints [override], [virtual]
```

Writes the constraints of locally active being equal to zero into a contstrint object and returns it.

This is the only function on this type that does something. It projects zero values onto the inner domains surface and builds a AffineConstraints<ComplexNumber> object from the resulting values. The object it returns can be merged with other objects of the same type to build the global constraint object.

Returns

A constraint object representing the PEC boundary data.

Reimplemented from BoundaryCondition.

Definition at line 71 of file EmptySurface.cpp.

```
71                                       {
72     Constraints ret(Geometry.levels[level].inner_domain->global_dof_indices);
73     dealii::IndexSet local_dof_set(Geometry.levels[level].inner_domain->n_locally_active_dofs);
74     local_dof_set.add_range(0,Geometry.levels[level].inner_domain->n_locally_active_dofs);
75     AffineConstraints<ComplexNumber> constraints_local(local_dof_set);
76     std::vector<InterfaceDofData> dofs =
       Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
77     for(auto line : dofs) {
78         const unsigned int local_index = line.index;
79         const unsigned int global_index =
       Geometry.levels[level].inner_domain->global_index_mapping[local_index];
80         ret.add_line(global_index);
81         ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
82     }
83     for(unsigned int surf = 0; surf < 6; surf++) {
```

```
        if(surf != b_id && !are_opposing_sites(b_id, surf)) {
            if(Geometry.levels[level].surface_type[surf] == SurfaceType::ABC_SURFACE) {
                std::vector<InterfaceDofData> dofs =
       Geometry.levels[level].surfaces[surf]->get_dof_association_by_boundary_id(b_id);
                for(unsigned int i = 0; i < dofs.size(); i++) {
                    const unsigned int local_index = dofs[i].index;
                    const unsigned int global_index =
       Geometry.levels[level].surfaces[surf]->global_index_mapping[local_index];
                    ret.add_line(global_index);
                    ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
                }
            }
        }
    }
    return ret;
}
```

output_results()

```
std::string EmptySurface::output_results (
            const dealii::Vector< ComplexNumber > & solution,
            std::string filename )  [override], [virtual]
```

Would write output but this function has no own data to store.

This function performs no actions. See class and base class description for details.

Parameters

solution	NOT USED.
filename	NOT USED.

Returns

Implements BoundaryCondition.

Definition at line 53 of file EmptySurface.cpp.

```
53                                                                              {
54     return "";
55 }
```

The documentation for this class was generated from the following files:

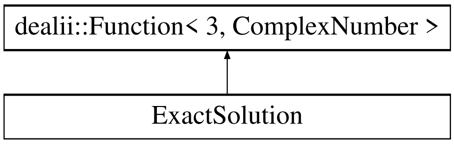
- Code/BoundaryCondition/EmptySurface.h
- Code/BoundaryCondition/EmptySurface.cpp

24 ExactSolution Class Reference

This class is derived from the Function class and can be used to estimate the L2-error for a straight waveguide. In the case of a completely cylindrical waveguide, an analytic solution is known (the modes of the input-signal themselves) and this class offers a representation of this analytical solution. If the waveguide has any other shape, this solution does not lose its value completely - it can still be used as a starting-vector for iterative solvers.

```
#include <ExactSolution.h>
```

Inheritance diagram for ExactSolution:



Public Member Functions

- ComplexNumber **value** (const Position &p, const unsigned int component) const
- void **vector_value** (const Position &p, dealii::Vector< ComplexNumber > &value) const
- dealii::Tensor< 1, 3, ComplexNumber > **curl** (const Position &in_p) const
- dealii::Tensor< 1, 3, ComplexNumber > **val** (const Position &in_p) const
- ComplexNumber **compute_phase_for_position** (const Position &in_p) const
- Position2D **get_2D_position_from_3d** (const Position &in_p) const
- J_derivative_terms **get_derivative_terms** (const Position2D &in_p) const

Static Public Member Functions

- static void **load_data** (std::string fname)

Public Attributes

- dealii::Functions::InterpolatedUniformGridData< 2 > **component_x**
- dealii::Functions::InterpolatedUniformGridData< 2 > **component_y**
- dealii::Functions::InterpolatedUniformGridData< 2 > **component_z**

Static Public Attributes

- static dealii::Table< 2, double > **data_table_x**
- static dealii::Table< 2, double > **data_table_y**
- static dealii::Table< 2, double > **data_table_z**
- static std::array< std::pair< double, double >, 2 > **ranges**
- static std::array< unsigned int, 2 > **n_intervals**

24.1 Detailed Description

This class is derived from the Function class and can be used to estimate the L2-error for a straight waveguide. In the case of a completely cylindrical waveguide, an analytic solution is known (the modes of the input-signal themselves) and this class offers a representation of this analytical solution. If the waveguide has any other shape, this solution does not lose its value completely - it can still be used as a starting-vector for iterative solvers.

The structure of this class is defined by the properties of the Function-class meaning that we have two functions:

1. virtual double value (const Point<dim> &p, const unsigned int component) calculates the value for a single component of the vector-valued return-value.
2. virtual void vector_value (const Point<dim> &p, Vector<double> &value) puts these individual components into the parameter value, which is a reference to a vector, handed over to store the result.

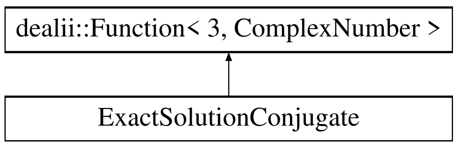
Definition at line 35 of file ExactSolution.h.

The documentation for this class was generated from the following files:

- Code/Solutions/ExactSolution.h
- Code/Solutions/ExactSolution.cpp

25 ExactSolutionConjugate Class Reference

Inheritance diagram for ExactSolutionConjugate:



Public Member Functions

- ComplexNumber **value** (const Position &p, const unsigned int component) const
- void **vector_value** (const Position &p, dealii::Vector< ComplexNumber > &value) const
- dealii::Tensor< 1, 3, ComplexNumber > **curl** (const Position &in_p) const
- dealii::Tensor< 1, 3, ComplexNumber > **val** (const Position &in_p) const

25.1 Detailed Description

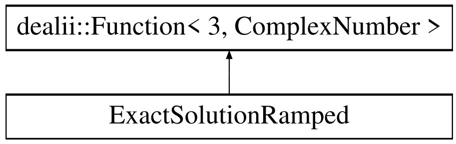
Definition at line 12 of file ExactSolutionConjugate.h.

The documentation for this class was generated from the following files:

- Code/Solutions/ExactSolutionConjugate.h
- Code/Solutions/ExactSolutionConjugate.cpp

26 ExactSolutionRamped Class Reference

Inheritance diagram for ExactSolutionRamped:



Public Member Functions

- double **get_ramping_factor_for_position** (const Position &) const
- ComplexNumber **value** (const Position &p, const unsigned int component) const
- void **vector_value** (const Position &p, dealii::Vector< ComplexNumber > &value) const
- dealii::Tensor< 1, 3, ComplexNumber > **curl** (const Position &in_p) const
- dealii::Tensor< 1, 3, ComplexNumber > **val** (const Position &in_p) const
- double **compute_ramp_for_c0** (const Position &in_p) const
- double **compute_ramp_for_c1** (const Position &in_p) const
- double **ramping_delta** (const Position &in_p) const
- double **get_ramping_factor_derivative_for_position** (const Position &in_p) const

26.1 Detailed Description

Definition at line 12 of file ExactSolutionRamped.h.

The documentation for this class was generated from the following files:

- Code/Solutions/ExactSolutionRamped.h
- Code/Solutions/ExactSolutionRamped.cpp

27 FEAdjointEvaluation Struct Reference

Public Attributes

- Position **x**
- dealii::Tensor< 1, 3, ComplexNumber > **primal_field**
- dealii::Tensor< 1, 3, ComplexNumber > **adjoint_field**
- dealii::Tensor< 1, 3, ComplexNumber > **primal_field_curl**
- dealii::Tensor< 1, 3, ComplexNumber > **adjoint_field_curl**

27.1 Detailed Description

Definition at line 233 of file Types.h.

The documentation for this struct was generated from the following file:

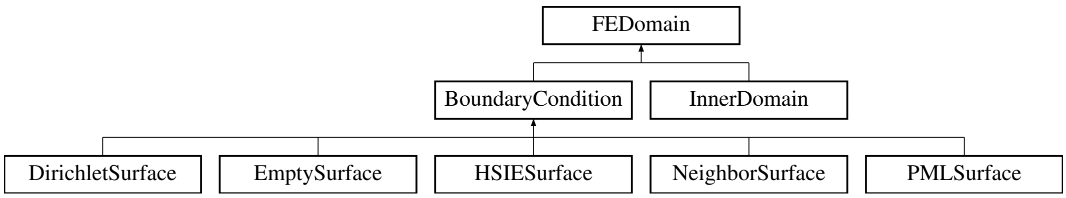
- Code/Core/Types.h

28 FEDomain Class Reference

This class is a base type for all objects that own their own dofs.

```
#include <FEDomain.h>
```

Inheritance diagram for FEDomain:



Public Member Functions

- virtual void determine_non_owned_dofs ()=0

  In derived objects, this function will check for all dofs if they are locally owned or not.
- void initialize_dof_counts (DofCount n_locally_active_dofs, DofCount n_locally_owned_dofs)

  Function for internal use.
- DofIndexVector transform_local_to_global_dofs (DofIndexVector local_index)

  Returns the global number for a local index.
- void mark_local_dofs_as_non_local (DofIndexVector indices)

  Takes an index set and marks all indices in the set as non locally owned.
- virtual bool finish_initialization (DofNumber first_own_index)

  Once all ownerships have been decided, this function numbers the locally owned dofs starting at the number provided.
- void set_non_local_dof_indices (DofIndexVector local_indices, DofIndexVector global_indices)

  For a given index vector in local and global numbering, this function stores the global indices.
- virtual DofCount compute_n_locally_owned_dofs ()=0

  Counts the number of locally owned dofs.
- virtual DofCount compute_n_locally_active_dofs ()=0

  Counts the number of locally active dofs.
- void freeze_ownership ()

  After this is called, ownership of dofs cannot be changed.
- NumericVectorLocal get_local_vector_from_global (const NumericVectorDistributed in_vector)

  For a provided vector of a global problem, this function extracts the locally active vector and returns it.
- double local_norm_of_vector (NumericVectorDistributed ∗)

  Computes the L2 norm of the contributions to the provided vector by the local object.

Public Attributes

- DofCount **n_locally_active_dofs**
- DofCount **n_locally_owned_dofs**
- dealii::IndexSet **global_dof_indices**
- DofIndexVector **global_index_mapping**
- std::vector< bool > **is_dof_owned**
- bool **is_ownership_ready**

28.1 Detailed Description

This class is a base type for all objects that own their own dofs.

For all such objects we have to manage the sets of locally active and owned dofs. This object provides an abstract interface for these tasks.

Definition at line 22 of file FEDomain.h.

28.2 Member Function Documentation

compute_n_locally_active_dofs()

`virtual DofCount FEDomain::compute_n_locally_active_dofs ( )  [pure virtual]`

Counts the number of locally active dofs.

Returns

DofCount The number of locally active dofs.

Implemented in HSIESurface, PMLSurface, InnerDomain, EmptySurface, DirichletSurface, and NeighborSurface.

compute_n_locally_owned_dofs()

`virtual DofCount FEDomain::compute_n_locally_owned_dofs ( )  [pure virtual]`

Counts the number of locally owned dofs.

Returns

DofCount The number of locally owned dofs.

Implemented in HSIESurface, PMLSurface, InnerDomain, EmptySurface, DirichletSurface, and NeighborSurface.

determine_non_owned_dofs()

`virtual void FEDomain::determine_non_owned_dofs ( ) [pure virtual]`

In derived objects, this function will check for all dofs if they are locally owned or not.

It will store the result in the vector is_dof_owned. Once this is done we can count how many new dofs this object introduces.

Implemented in HSIESurface, PMLSurface, InnerDomain, EmptySurface, DirichletSurface, and NeighborSurface.

finish_initialization()

```
bool FEDomain::finish_initialization (
            DofNumber first_own_index ) [virtual]
```

Once all ownerships have been decided, this function numbers the locally owned dofs starting at the number provided.

Parameters

first_own_index	The index the first locally owned dof should have.

Returns

true If all dofs now have a valid index.

false If there are still dofs that have no valid index

Reimplemented in HSIESurface, and PMLSurface.

Definition at line 33 of file FEDomain.cpp.

```
33                                                                {
34     if(!is_ownership_ready) {
35         std::cout « "You called finish_initialization before freeze_ownership which is not valid." «
       std::endl;
36         return false;
37     }
38     DofNumber running_index = first_own_index;
39     for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
40         if(is_dof_owned[i]) {
41             global_index_mapping[i] = running_index;
42             running_index++;
43         }
44     }
45     return true;
46 }
```

get_local_vector_from_global()

```
NumericVectorLocal FEDomain::get_local_vector_from_global (
            const NumericVectorDistributed in_vector )
```

For a provided vector of a global problem, this function extracts the locally active vector and returns it.

Parameters

in_vector	The global solution vector.

Returns

NumericVectorLocal The excerpt of the global vector in local numbering.

Definition at line 71 of file FEDomain.cpp.

```
71                                                                           {
72     NumericVectorLocal ret(n_locally_active_dofs);
73     for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
74         ret[i] = in_vector[global_index_mapping[i]];
75     }
76     return ret;
77 }
```

initialize_dof_counts()

```
void FEDomain::initialize_dof_counts (
        DofCount n_locally_active_dofs,
        DofCount n_locally_owned_dofs )
```

Function for internal use.

This sets the number of locally owned and active dofs.

Parameters

n_locally_active_dofs	The number of dofs that have support on the domain represented by this object. This is usually non-zero.
n_locally_owned_dofs	The number of dofs that are either only active on the domain represented by this object or alternatively dofs that that are shared but this object has been determined to be the owner.

Definition at line 9 of file FEDomain.cpp.

```
9                                                                               {
10     n_locally_owned_dofs = in_n_locally_owned_dofs;
11     n_locally_active_dofs = in_n_locally_active_dofs;
12     global_index_mapping.resize(n_locally_active_dofs);
13     for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
14         global_index_mapping[i] = UINT_MAX;
15         is_dof_owned.push_back(true);
16     }
17
18 }
```

local_norm_of_vector()

```
double FEDomain::local_norm_of_vector (
        NumericVectorDistributed * in_v )
```

Computes the L2 norm of the contributions to the provided vector by the local object.

Returns

double L2 norm of the local part.

Definition at line 79 of file FEDomain.cpp.

```
79                                                              {
80     double norm = 0;
81     for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
82         if(is_dof_owned[i]) {
83             norm += norm_squared(in_v->operator()(global_index_mapping[i]));
84         }
85     }
86     return std::sqrt(norm);
87 }
```

mark_local_dofs_as_non_local()

```
void FEDomain::mark_local_dofs_as_non_local (
            DofIndexVector indices )
```

Takes an index set and marks all indices in the set as non locally owned.

Parameters

indices	The set containing the dofs that are non-locally-owned.

Definition at line 65 of file FEDomain.cpp.

```
65                                                           {
66     for(unsigned int i = 0; i < in_dofs.size(); i++) {
67         is_dof_owned[in_dofs[i]] = false;
68     }
69 }
```

Referenced by PMLSurface::determine_non_owned_dofs(), and HSIESurface::determine_non_owned_dofs().

set_non_local_dof_indices()

```
void FEDomain::set_non_local_dof_indices (
            DofIndexVector local_indices,
            DofIndexVector global_indices )
```

For a given index vector in local and global numbering, this function stores the global indices.

After this call, the global index of any of the provided local indices is what was provided. The data usually comes from another boundary or process or the interior domain

Parameters

local_indices	Indices in local numbering.
global_indices	Indices in global numbering.

Definition at line 56 of file FEDomain.cpp.

```
56                                                                                   {
```

```
    if(local_indices.size() != global_indices.size()) {
        std::cout « "There was a vector size mismatch in FEDomain::set_non_local_dof_indices( " «
    local_indices.size() « " vs " « global_indices.size() « ")" « std::endl;
    }
    for(unsigned int i = 0; i < local_indices.size(); i++) {
        global_index_mapping[local_indices[i]] = global_indices[i];
    }
}
```

transform_local_to_global_dofs()

```
std::vector< DofNumber > FEDomain::transform_local_to_global_dofs (
            DofIndexVector local_index )
```

Returns the global number for a local index.

Local indices always range from zero to n_locally_active_dofs. Global indices depend on the sweeping level and many other factors.

Parameters

local_index	The local index to be transformed into global numbering

Returns

DofIndexVector

Definition at line 48 of file FEDomain.cpp.

```
48                                                                                     {
49     std::vector<DofNumber> global_dof_indices;
50     for(unsigned int i = 0; i < in_dofs.size(); i++) {
51         global_dof_indices.push_back(global_index_mapping[in_dofs[i]]);
52     }
53     return global_dof_indices;
54 }
```

Referenced by PMLSurface::fill_matrix(), PMLSurface::fill_sparsity_pattern(), InnerDomain::fill_sparsity_pattern(), HSIESurface::fill_sparsity_pattern(), and BoundaryCondition::get_global_dof_indices_by_boundary_id().

The documentation for this class was generated from the following files:

- Code/Core/FEDomain.h
- Code/Core/FEDomain.cpp

29 FEErrorStruct Struct Reference

Public Attributes

- double **L2** = 0
- double **Linfty** = 0

29.1 Detailed Description

Definition at line 228 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

30 FileLogger Class Reference

There will be one global instance of this object.

`#include <FileLogger.h>`

30.1 Detailed Description

There will be one global instance of this object.

It creates file paths and provides file names. Every IO operation will be piped through this object. The other loggers use it to persist their data.

Definition at line 14 of file FileLogger.h.

The documentation for this class was generated from the following file:

- Code/OutputGenerators/Files/FileLogger.h

31 FileMetaData Struct Reference

Public Attributes

- unsigned int **hsie_level**

31.1 Detailed Description

Definition at line 113 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

32 GeometryManager Class Reference

One object of this type is globally available to handle the geometry of the computation (what is the global computational domain, what is computed locally).

`#include <GeometryManager.h>`

Public Member Functions

- void initialize ()

  Parent of the entire initialization loop This initializes all levels of the computation.

- void initialize_inner_domain (unsigned int in_level)

    On the level in_level this builds the InnerDomain object.
- double eps_kappa_2 (Position)

    *This function computes the term epsilon_r * omega^2 at a given location.*
- double kappa_2 ()

    Like the function above but without epsilon_r.
- std::pair< double, double > compute_x_range ()

    Computes the range of the coordinate x this process is responsible for.
- std::pair< double, double > compute_y_range ()

    Same as above but for y.
- std::pair< double, double > compute_z_range ()

    Same as above but for z.
- void set_x_range (std::pair< double, double > inp_x)

    Fixes the x-range this process is working on for its inner domain.
- void set_y_range (std::pair< double, double > inp_y)

    Fixes the y-range this process is working on for its inner domain.
- void set_z_range (std::pair< double, double > inp_z)

    Fixes the z-range this process is working on for its inner domain.
- std::pair< bool, unsigned int > get_global_neighbor_for_interface (Direction dir)

    For a given direction, this function computes if there is a neighbor of this process in that direction and, if so, that process's rank.
- std::pair< bool, unsigned int > get_level_neighbor_for_interface (Direction dir, unsigned int level)

    Similar to the function above but gets the rank of the neighbor in a level communicator for the level in_level.
- bool math_coordinate_in_waveguide (Position) const

    Checks if the coordinate is in the waveguide core or not.
- dealii::Tensor< 2, 3 > get_epsilon_tensor (const Position &)

    Returns a diagonalized material tensor that does not use transformation optics.
- double get_epsilon_for_point (const Position &)

    Computes scalar \epsilon_r for the given location.
- auto **get_boundary_for_direction** (Direction) -> BoundaryId
- auto **get_direction_for_boundary_id** (BoundaryId) -> Direction
- void **validate_global_dof_indices** (unsigned int in_level)
- SurfaceType **get_surface_type** (BoundaryId b_id, unsigned int level)
- void **distribute_dofs_on_level** (unsigned int level)
- void **set_surface_types_and_properties** (unsigned int level)
- void **initialize_surfaces_on_level** (unsigned int level)
- void **initialize_level** (unsigned int level)
- void **print_level_dof_counts** (unsigned int level)

- void **perform_mpi_dof_exchange** (unsigned int level)

Public Attributes

- double **input_connector_length**
- double **output_connector_length**
- double **shape_sector_length**
- unsigned int **shape_sector_count**
- unsigned int **local_inner_dofs**
- bool **are_surface_meshes_initialized**
- double **h_x**
- double **h_y**
- double **h_z**
- std::array< unsigned int, 6 > **dofs_at_surface**
- std::array< dealii::Triangulation< 2, 2 >, 6 > **surface_meshes**
- std::array< double, 6 > **surface_extremal_coordinate**
- std::pair< double, double > **local_x_range**
- std::pair< double, double > **local_y_range**
- std::pair< double, double > **local_z_range**
- std::pair< double, double > **global_x_range**
- std::pair< double, double > **global_y_range**
- std::pair< double, double > **global_z_range**
- std::array< LevelGeometry, 4 > **levels**

32.1 Detailed Description

One object of this type is globally available to handle the geometry of the computation (what is the global computational domain, what is computed locally).

This object is one of the first to be initialized. It contains the coordinate ranges locally and globally. It also has several LevelGeometry objects in a vector. This is the core data behind the sweeping hierarchy. These level objects contain:

- the surface types for all boundaries on this level
- pointers to the boundary condition objects
- dof counting data (how many dofs exist on the level, how many dofs does this process own on this level) and also which dofs are stored where in the dof_distribution member.

This object can also determine if a coordinate is inside or outside of the waveguide and computes kappa squared required for the assembly of Maxwell's equations.

Definition at line 46 of file GeometryManager.h.

32.2 Member Function Documentation

compute_x_range()

`std::pair< double, double > GeometryManager::compute_x_range ( )`

Computes the range of the coordinate x this process is responsible for.

Since the local domains are always of the form [min_x, max_x]\times[min_y, max_y]\times[min_z, max_z], these ranges can be used to describe the local problem.

Returns

std::pair<double, double> first is the lower bound of the range, second is the upper bound.

Definition at line 214 of file GeometryManager.cpp.

```
214                                                              {
215   if (GlobalParams.Blocks_in_x_direction == 1) {
216     return std::pair<double, double>(-GlobalParams.Geometry_Size_X / 2.0, GlobalParams.Geometry_Size_X /
       2.0);
217   } else {
218     double length = GlobalParams.Geometry_Size_X / ((double) GlobalParams.Blocks_in_x_direction);
219     int block_index = GlobalParams.MPI_Rank % GlobalParams.Blocks_in_x_direction;
220     double min = -GlobalParams.Geometry_Size_X / 2.0 + block_index * length;
221     return std::pair<double, double>(min, min + length);
222   }
223 }
```

compute_y_range()

`std::pair< double, double > GeometryManager::compute_y_range ( )`

Same as above but for y.

Returns

std::pair<double, double> see above.

Definition at line 225 of file GeometryManager.cpp.

```
225                                                              {
226   if (GlobalParams.Blocks_in_y_direction == 1) {
227     return std::pair<double, double>(-GlobalParams.Geometry_Size_Y / 2.0, GlobalParams.Geometry_Size_Y /
       2.0);
228   } else {
229     double length = GlobalParams.Geometry_Size_Y / ((double) GlobalParams.Blocks_in_y_direction);
230     int block_processor_count = GlobalParams.Blocks_in_x_direction;
231     int block_index = (GlobalParams.MPI_Rank % (GlobalParams.Blocks_in_x_direction *
       GlobalParams.Blocks_in_y_direction)) / block_processor_count;
232     double min = -GlobalParams.Geometry_Size_Y / 2.0 + block_index * length;
233     return std::pair<double, double>(min, min + length);
234   }
235 }
```

compute_z_range()

`std::pair< double, double > GeometryManager::compute_z_range ( )`

Same as above but for z.

Returns

std::pair<double, double> see above.

Definition at line 237 of file GeometryManager.cpp.

```
237                                                             {
238   if (GlobalParams.Blocks_in_z_direction == 1) {
239     return std::pair<double, double>(0 + GlobalParams.global_z_shift, GlobalParams.Geometry_Size_Z +
       GlobalParams.global_z_shift);
240   } else {
241     double length = GlobalParams.Geometry_Size_Z / ((double) GlobalParams.Blocks_in_z_direction);
242     int block_processor_count = GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction;
243     int block_index = GlobalParams.MPI_Rank / block_processor_count;
244     double min = block_index * length;
245     return std::pair<double, double>(min + GlobalParams.global_z_shift, min +
       GlobalParams.global_z_shift + length);
246   }
247 }
```

eps_kappa_2()

```
double GeometryManager::eps_kappa_2 (
              Position in_p )
```

This function computes the term epsilon_r ∗ omega^2 at a given location.

This is required for the assembly of the Maxwell system.

Returns

double \epsilon_r ∗ \omega^2

Definition at line 191 of file GeometryManager.cpp.

```
191                                                      {
192   return (math_coordinate_in_waveguide(in_p)? GlobalParams.Epsilon_R_in_waveguide :
       GlobalParams.Epsilon_R_outside_waveguide) * GlobalParams.Omega * GlobalParams.Omega;
193 }
```

References math_coordinate_in_waveguide().

get_epsilon_for_point()

```
double GeometryManager::get_epsilon_for_point (
              const Position & in_p )
```

Computes scalar \epsilon_r for the given location.

Returns

double \epsilon_r of material at given location.

Definition at line 183 of file GeometryManager.cpp.

```
183                                                                        {
184   if(math_coordinate_in_waveguide(in_p)) {
185     return GlobalParams.Epsilon_R_in_waveguide;
186   } else {
187     return GlobalParams.Epsilon_R_outside_waveguide;
188   }
```

```
189 }
```

References math_coordinate_in_waveguide().

Referenced by get_epsilon_tensor().

get_epsilon_tensor()

```
dealii::Tensor< 2, 3 > GeometryManager::get_epsilon_tensor (
            const Position & in_p )
```

Returns a diagonalized material tensor that does not use transformation optics.

Artifact.

Returns

dealii::Tensor<2,3>

Definition at line 168 of file GeometryManager.cpp.

```
168                                                                        {
169   dealii::Tensor<2,3> ret;
170   const double local_epsilon = get_epsilon_for_point(in_p);
171   for(unsigned int i = 0; i < 3; i++) {
172     for(unsigned int j = 0; j < 3; j++) {
173       if(i == j) {
174         ret[i][j] = local_epsilon;
175       } else {
176         ret[i][j] = 0;
177       }
178     }
179   }
180   return ret;
181 }
```

References get_epsilon_for_point().

get_global_neighbor_for_interface()

```
std::pair< bool, unsigned int > GeometryManager::get_global_neighbor_for_interface (
            Direction dir )
```

For a given direction, this function computes if there is a neighbor of this process in that direction and, if so, that process's rank.

Parameters

dir	The direction to go to

Returns

std::pair<bool, unsigned int> first: is there a process there? second: whats its rank.

Definition at line 249 of file GeometryManager.cpp.

```
249                                                                                        {
250   std::pair<bool, unsigned int> ret(true, 0);
251   switch (in_direction) {
```

```
    case Direction::MinusX:
      if (GlobalParams.Index_in_x_direction == 0) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank - 1;
      }
      break;
    case Direction::PlusX:
      if (GlobalParams.Index_in_x_direction == GlobalParams.Blocks_in_x_direction - 1) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank + 1;
      }
      break;
    case Direction::MinusY:
      if (GlobalParams.Index_in_y_direction == 0) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank - GlobalParams.Blocks_in_x_direction;
      }
      break;
    case Direction::PlusY:
      if (GlobalParams.Index_in_y_direction == GlobalParams.Blocks_in_y_direction - 1) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank + GlobalParams.Blocks_in_x_direction;
      }
      break;
    case Direction::MinusZ:
      if (GlobalParams.Index_in_z_direction == 0) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank - (GlobalParams.Blocks_in_x_direction *
  GlobalParams.Blocks_in_y_direction);
      }
      break;
    case Direction::PlusZ:
      if (GlobalParams.Index_in_z_direction == GlobalParams.Blocks_in_z_direction - 1) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank + (GlobalParams.Blocks_in_x_direction *
  GlobalParams.Blocks_in_y_direction);
      }
      break;
  }
  return ret;
}
```

Referenced by get_level_neighbor_for_interface().

get_level_neighbor_for_interface()

```
std::pair< bool, unsigned int > GeometryManager::get_level_neighbor_for_interface (
              Direction dir,
              unsigned int level )
```

Similar to the function above but gets the rank of the neighbor in a level communicator for the level in_level.

Parameters

dir	Direction to check in
level	The level we are operating on.

Returns

std::pair<bool, unsigned int> Same as above but second returns the rank in the level communicator.

Definition at line 298 of file GeometryManager.cpp.

```
298
                    {
299   std::pair<bool, unsigned int> ret(true, 0);
300   if(level == 0) {
301     return get_global_neighbor_for_interface(in_direction);
302   }
303   if(level == 1) {
304     switch (in_direction) {
305       case Direction::MinusX:
306         if (GlobalParams.Index_in_x_direction == 0) {
307           ret.first = false;
308         } else {
309           ret.second = (GlobalParams.MPI_Rank - 1) % (GlobalParams.Blocks_in_x_direction *
       GlobalParams.Blocks_in_y_direction);
310         }
311         break;
312       case Direction::PlusX:
313         if (GlobalParams.Index_in_x_direction == GlobalParams.Blocks_in_x_direction - 1) {
314           ret.first = false;
315         } else {
316           ret.second = (GlobalParams.MPI_Rank + 1) % (GlobalParams.Blocks_in_x_direction *
       GlobalParams.Blocks_in_y_direction);
317         }
318         break;
319       case Direction::MinusY:
320         if (GlobalParams.Index_in_y_direction == 0) {
321           ret.first = false;
322         } else {
323           ret.second = (GlobalParams.MPI_Rank - GlobalParams.Blocks_in_y_direction) %
       (GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction);
324         }
325         break;
326       case Direction::PlusY:
327         if (GlobalParams.Index_in_y_direction == GlobalParams.Blocks_in_y_direction - 1) {
328           ret.first = false;
329         } else {
330           ret.second = (GlobalParams.MPI_Rank + GlobalParams.Blocks_in_y_direction) %
       (GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction);
331         }
332         break;
333       case Direction::MinusZ:
334         ret.first = false;
335         break;
336       case Direction::PlusZ:
337         ret.first = false;
338         break;
339     }
340   }
341   if(level == 2) {
342     switch (in_direction) {
343       case Direction::MinusX:
344         if (GlobalParams.Index_in_x_direction == 0) {
345           ret.first = false;
346         } else {
347           ret.second = (GlobalParams.MPI_Rank - 1) % GlobalParams.Blocks_in_x_direction;
348         }
349         break;
350       case Direction::PlusX:
351         if (GlobalParams.Index_in_x_direction == GlobalParams.Blocks_in_x_direction - 1) {
352           ret.first = false;
353         } else {
354           ret.second = (GlobalParams.MPI_Rank + 1) % GlobalParams.Blocks_in_x_direction;
355         }
356         break;
357       case Direction::MinusY:
358         ret.first = false;
359         break;
360       case Direction::PlusY:
```

```
        ret.first = false;
        break;
      case Direction::MinusZ:
        ret.first = false;
        break;
      case Direction::PlusZ:
        ret.first = false;
        break;
    }
  }
  return ret;
}
```

References get_global_neighbor_for_interface().

initialize_inner_domain()

```
void GeometryManager::initialize_inner_domain (
            unsigned int in_level )
```

On the level in_level this builds the InnerDomain object.

Parameters

in_level	The level to perform the action on.

Definition at line 72 of file GeometryManager.cpp.

```
72 {
73   levels[in_level].inner_domain = new InnerDomain(in_level);
74   levels[in_level].inner_domain->make_grid();
75   if(!are_surface_meshes_initialized) {
76     for (unsigned int side = 0; side < 6; side++) {
77       dealii::Triangulation<2, 3> temp_triangulation;
78       dealii::Triangulation<2> surf_tria;
79       Mesh tria;
80       tria.copy_triangulation(levels[in_level].inner_domain->triangulation);
81       std::set<unsigned int> b_ids;
82       b_ids.insert(side);
83       switch (side) {
84         case 0:
85           dealii::GridTools::transform(Transform_0_to_5, tria);
86           break;
87         case 1:
88           dealii::GridTools::transform(Transform_1_to_5, tria);
89           break;
90         case 2:
91           dealii::GridTools::transform(Transform_2_to_5, tria);
92           break;
93         case 3:
94           dealii::GridTools::transform(Transform_3_to_5, tria);
95           break;
96         case 4:
97           dealii::GridTools::transform(Transform_4_to_5, tria);
98           break;
99         default:
100            break;
101       }
102       dealii::GridGenerator::extract_boundary_mesh(tria, temp_triangulation, b_ids);
103       dealii::GridGenerator::flatten_triangulation(temp_triangulation, surface_meshes[side]);
104     }
105     are_surface_meshes_initialized = true;
106   }
107 }
```

kappa_2()

```
double GeometryManager::kappa_2 ( )
```

Like the function above but without epsilon_r.

Since this value is independent of the position, this function has no arguments.

Returns

double \omega^2

Definition at line 195 of file GeometryManager.cpp.

```
195                               {
196   return GlobalParams.Omega * GlobalParams.Omega;
197 }
```

math_coordinate_in_waveguide()

```
bool GeometryManager::math_coordinate_in_waveguide (
             Position in_position ) const
```

Checks if the coordinate is in the waveguide core or not.

Returns

true Location in mathematical coordinates corresponds with the interior of the waveguide.

false it does not.

Definition at line 374 of file GeometryManager.cpp.

```
374                                                                         {
375   bool in_x = std::abs(in_position[0]) <= (GlobalParams.Width_of_waveguide  / 2.0);
376   bool in_y = std::abs(in_position[1]) <= (GlobalParams.Height_of_waveguide / 2.0);
377   return in_x && in_y;
378 }
```

Referenced by eps_kappa_2(), and get_epsilon_for_point().

set_x_range()

```
void GeometryManager::set_x_range (
             std::pair< double, double > inp_x )
```

Fixes the x-range this process is working on for its inner domain.

Boundary conditions can extend beyond this value however. The idea is to use the return value of compute_x_range().

Parameters

inp_x	the x_range to use locally.

Definition at line 199 of file GeometryManager.cpp.

```
199                                                                     {
200    this->local_x_range = in_range;
201    global_x_range = std::pair<double, double>(-GlobalParams.Geometry_Size_X / 2.0,
        GlobalParams.Geometry_Size_X / 2.0);
202 }
```

set_y_range()

```
void GeometryManager::set_y_range (
              std::pair< double, double > inp_y )
```

Fixes the y-range this process is working on for its inner domain.

Boundary conditions can extend beyond this value however. The idea is to use the return value of compute_y_range().

Parameters

inp_y	the y_range to use locally.

Definition at line 204 of file GeometryManager.cpp.

```
204                                                                     {
205    this->local_y_range = in_range;
206    global_y_range = std::pair<double, double> (-GlobalParams.Geometry_Size_Y / 2.0,
        GlobalParams.Geometry_Size_Y / 2.0);
207 }
```

set_z_range()

```
void GeometryManager::set_z_range (
              std::pair< double, double > inp_z )
```

Fixes the z-range this process is working on for its inner domain.

Boundary conditions can extend beyond this value however. The idea is to use the return value of compute_z_range().

Parameters

inp_z	the z_range to use locally.

Definition at line 209 of file GeometryManager.cpp.

```
209                                                                     {
210    this->local_z_range = in_range;
211    global_z_range = std::pair<double, double>(0.0 + GlobalParams.global_z_shift ,
        GlobalParams.Geometry_Size_Z + GlobalParams.global_z_shift);
212 }
```

The documentation for this class was generated from the following files:

- Code/GlobalObjects/GeometryManager.h
- Code/GlobalObjects/GeometryManager.cpp

33 GradientTable Class Reference

The Gradient Table is an OutputGenerator, intended to write information about the shape gradient to the console upon its computation.

`#include <GradientTable.h>`

Public Member Functions

- **GradientTable** (unsigned int in_step, dealii::Vector< double > in_configuration, double in_quality, dealii::Vector< double > in_last_configuration, double in_last_quality)
- void **SetInitialQuality** (double in_quality)
- void **AddComputationResult** (int in_component, double in_step, double in_quality)
- void **AddFullStepResult** (dealii::Vector< double > in_step, double in_quality)
- void **PrintFullLine** ()
- void **PrintTable** ()
- void **WriteTableToFile** (std::string in_filename)

Public Attributes

- const int **ndofs**
- const int **nfreedofs**
- const unsigned int **GlobalStep**

33.1 Detailed Description

The Gradient Table is an OutputGenerator, intended to write information about the shape gradient to the console upon its computation.

Definition at line 12 of file GradientTable.h.

The documentation for this class was generated from the following files:

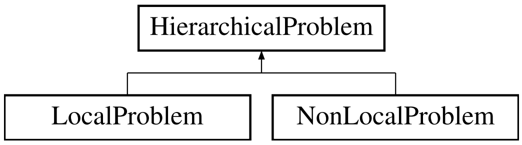
- Code/OutputGenerators/Console/GradientTable.h
- Code/OutputGenerators/Console/GradientTable.cpp

34 HierarchicalProblem Class Reference

The base class of the SweepingPreconditioner and general finite element system.

`#include <HierarchicalProblem.h>`

Inheritance diagram for HierarchicalProblem:



Public Member Functions

- HierarchicalProblem (unsigned int level, SweepingDirection direction)

  Construct a new Hierarchical Problem object Inits the level member, stores the direction of the sweep and the solve counter.

- virtual ~HierarchicalProblem ()=0

  Not implemented on this level.

- virtual void solve ()=0

  Not implemented on this level.

- virtual void solve_adjoint ()

  Not implemented on this level.

- void solve_with_timers_and_count ()

  This function calls the objects solve() method but wraps a timer computation around it.

- virtual void initialize ()=0

  Not implemented on this level, see derived classes.

- void make_constraints ()

  This function constructs all the required AffineConstraint objects.

- virtual void assemble ()=0

  Not implemented on this level, see derived classes.

- virtual void initialize_index_sets ()=0

  Not implemented on this level, see derived classes.

- void constrain_identical_dof_sets (std::vector< unsigned int > ∗set_one, std::vector< unsigned int > ∗set_two, Constraints ∗affine_constraints)

  For a given AffineConstraints object, this function adds constraints relating to numbering of dofs on two different structures.

- virtual auto reinit () -> void=0

  Not implemented on this level, see derived classes.

- auto opposing_site_bid (BoundaryId) -> BoundaryId

  For a provided boundary id this returns the opposing one The opposing sides are 0 and 1, 2 and 3, 4 and 5.

- void compute_final_rhs_mismatch ()

  Computes a vector storing the difference between the precise rhs and the approximation by the solution.

- virtual void compute_solver_factorization ()=0

  Not implemented on this level, see derived classes.

- std::string output_results (std::string in_fname_part="solution_inner_domain_level")

Basic functionality to write output files for a solution.

- virtual void reinit_rhs ()=0

  Not implemented on this level, see derived classes.

- virtual void make_sparsity_pattern ()=0

  Not implemented on this level, see derived classes.

- virtual void update_convergence_criterion (double)

  Not implemented on this level, see derived classes.

- virtual unsigned int compute_global_solve_counter ()

  Not implemented on this level, see derived classes.

- void print_solve_counter_list ()

  This function uses the return values of compute_flobal_solve_counter to create some CLI output.

- virtual void empty_memory ()

  Not implemented on this level, see derived classes.

- virtual void write_multifile_output (const std::string &filename, bool apply_coordinate_transform)=0

  Not implemented on this level, see derived classes.

- virtual std::vector< double > compute_shape_gradient ()

  Not implemented on this level, see derived classes.

Public Attributes

- SweepingDirection **sweeping_direction**
- const SweepingLevel **level**
- Constraints **constraints**
- std::array< dealii::IndexSet, 6 > **surface_index_sets**
- std::array< bool, 6 > **is_hsie_surface**
- std::vector< bool > **is_surface_locked**
- bool **is_dof_manager_set**
- bool **has_child**
- HierarchicalProblem * **child**
- dealii::SparsityPattern **sp**
- NumericVectorDistributed **solution**
- NumericVectorDistributed **direct_solution**
- NumericVectorDistributed **solution_error**
- NumericVectorDistributed **rhs**
- dealii::IndexSet **own_dofs**
- std::array< std::vector< InterfaceDofData >, 6 > **surface_dof_associations**
- dealii::PETScWrappers::MPI::SparseMatrix * **matrix**
- std::vector< std::string > **filenames**
- ResidualOutputGenerator * **residual_output**

- unsigned int **solve_counter**
- int **parent_sweeping_rank** = -1

34.1 Detailed Description

The base class of the SweepingPreconditioner and general finite element system.

Since the object should call eachother recursively but the lowest level is different than the others, we use an abstract base class and two derived types.

Definition at line 30 of file HierarchicalProblem.h.

34.2 Constructor & Destructor Documentation

HierarchicalProblem()

```
HierarchicalProblem::HierarchicalProblem (
            unsigned int level,
            SweepingDirection direction )
```

Construct a new Hierarchical Problem object Inits the level member, stores the direction of the sweep and the solve counter.

Parameters

level	Level this problem describes.
direction	The direction to sweep in. Doesnt matter for the LocalProblem.

Definition at line 17 of file HierarchicalProblem.cpp.

```
17                                                                                      :
18   level(in_own_level) {
19
20   sweeping_direction = get_sweeping_direction_for_level(in_own_level);
21   has_child = in_own_level > 0;
22   child = nullptr;
23   for(unsigned int i = 0; i < 6; i++) {
24     is_surface_locked.push_back(false);
25   }
26   solve_counter = 0;
27 }
```

34.3 Member Function Documentation

compute_final_rhs_mismatch()

```
void HierarchicalProblem::compute_final_rhs_mismatch ( )
```

Computes a vector storing the difference between the precise rhs and the approximation by the solution.

This updates a vector called rhs_mismatch by filling it with the $Ax - b$.

compute_global_solve_counter()

`virtual unsigned int HierarchicalProblem::compute_global_solve_counter ( ) [inline], [virtual]`

Not implemented on this level, see derived classes.

Returns

unsigned int

Reimplemented in NonLocalProblem, and LocalProblem.

Definition at line 180 of file HierarchicalProblem.h.

```
180                                                       {
181     return 0;
182   }
```

Referenced by print_solve_counter_list().

compute_shape_gradient()

`virtual std::vector<double> HierarchicalProblem::compute_shape_gradient ( ) [inline], [virtual]`

Not implemented on this level, see derived classes.

Returns

std::vector<double>

Reimplemented in NonLocalProblem.

Definition at line 209 of file HierarchicalProblem.h.

```
209                                                      {
210     return std::vector<double>();
211   }
```

constrain_identical_dof_sets()

```
void HierarchicalProblem::constrain_identical_dof_sets (
            std::vector< unsigned int > * set_one,
            std::vector< unsigned int > * set_two,
            Constraints * affine_constraints )
```

For a given AffineConstraints object, this function adds constraints relating to numbering of dofs on two different structures.

This function can be used to couple boundary methods together or to couple dofs from a boundary method with dofs on the inner domain.

Parameters

set_one	First index set.

Parameters

set_two	Second index set.
affine_constraints	Affine Constraint object to write the constraints into.

Definition at line 29 of file HierarchicalProblem.cpp.

```
31                                       {
32   const unsigned int n_entries = set_one->size();
33   if (n_entries != set_two->size()) {
34     print_info("HierarchicalProblem::constrain_identical_dof_sets", "There was an error in
       constrain_identical_dof_sets. No changes made.", LoggingLevel::PRODUCTION_ALL);
35   }
36
37   for (unsigned int index = 0; index < n_entries; index++) {
38     affine_constraints->add_line(set_one->operator [](index));
39     affine_constraints->add_entry(set_one->operator [](index),
40         set_two->operator [](index), ComplexNumber(-1, 0));
41   }
42 }
```

make_constraints()

`void HierarchicalProblem::make_constraints ( )`

This function constructs all the required AffineConstraint objects.

These couple the dofs in the inner domain and the boundary conditions together and is used for in-place condensation during matrix assembly.

Definition at line 53 of file HierarchicalProblem.cpp.

```
53                                             {
54   print_info("HierarchicalProblem::make_constraints", "Start");
55   IndexSet total_dofs_global(Geometry.levels[level].n_total_level_dofs);
56   total_dofs_global.add_range(0,Geometry.levels[level].n_total_level_dofs);
57   constraints.reinit(total_dofs_global);
58
59   // ABC Surfaces are least important
60   for(unsigned int surface = 0; surface < 6; surface++) {
61     if(Geometry.levels[level].surface_type[surface] == SurfaceType::ABC_SURFACE) {
62       Constraints local_constraints = Geometry.levels[level].surfaces[surface]->make_constraints();
63       constraints.merge(local_constraints, Constraints::MergeConflictBehavior::right_object_wins,true);
64     }
65   }
66
67   // Dirichlet surfaces are more important than ABC
68   for(unsigned int surface = 0; surface < 6; surface++) {
69     if(Geometry.levels[level].surface_type[surface] == SurfaceType::DIRICHLET_SURFACE) {
70       Constraints local_constraints = Geometry.levels[level].surfaces[surface]->make_constraints();
71       constraints.merge(local_constraints, Constraints::MergeConflictBehavior::right_object_wins,true);
72     }
73   }
74
75   // Open surfaces are most important
76   for(unsigned int surface = 0; surface < 6; surface++) {
77     if(Geometry.levels[level].surface_type[surface] == SurfaceType::OPEN_SURFACE) {
78       Constraints local_constraints = Geometry.levels[level].surfaces[surface]->make_constraints();
79       constraints.merge(local_constraints, Constraints::MergeConflictBehavior::right_object_wins,true);
80     }
81   }
82   constraints.close();
83
84   print_info("HierarchicalProblem::make_constraints", "End");
85 }
```

opposing_site_bid()

```
auto HierarchicalProblem::opposing_site_bid (
             BoundaryId in_bid ) -> BoundaryId
```

For a provided boundary id this returns the opposing one The opposing sides are 0 and 1, 2 and 3, 4 and 5.

This function is usually required when a function should be called when all neighboring boundaries should be iterated. In that case we iterate from 0 to 5 and exclude the one we are currently on and the opposing one.

Returns

BoundaryId The BoundaryId of the opposing side.

Definition at line 44 of file HierarchicalProblem.cpp.

```
44                                                                       {
45   if((in_bid % 2) == 0) {
46     return in_bid + 1;
47   }
48   else {
49     return in_bid - 1;
50   }
51 }
```

output_results()

```
std::string HierarchicalProblem::output_results (
             std::string in_fname_part = "solution_inner_domain_level" )
```

Basic functionality to write output files for a solution.

Parameters

in_fname_part	Core of the filename of the files.

Returns

std::string actually used filename with path which can be used to write meta data.

Definition at line 87 of file HierarchicalProblem.cpp.

```
87                                                                  {
88   GlobalTimerManager.switch_context("Output Results", level);
89   Timer timer;
90   timer.start();
91   print_info("Hierarchical::output_results()", "Start on level " + std::to_string(level));
92   std::string ret = "";
93   NumericVectorLocal in_solution(Geometry.levels[level].inner_domain->dof_handler.n_dofs());
94   for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->dof_handler.n_dofs(); i++) {
95     in_solution[i] = solution[Geometry.levels[level].inner_domain->global_index_mapping[i]];
96   }
97   std::string file_1 = Geometry.levels[level].inner_domain->output_results(in_fname_part +
       std::to_string(level) , in_solution, false);
98   ret = file_1;
99   filenames.clear();
100   filenames.push_back(file_1);
```

```

  if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
    for(unsigned int i = 0; i < 6; i++){
      if(Geometry.levels[level].surface_type[i] == SurfaceType::ABC_SURFACE){
        dealii::Vector<ComplexNumber> ds (Geometry.levels[level].surfaces[i]->dof_counter);
        for(unsigned int index = 0; index < Geometry.levels[level].surfaces[i]->dof_counter; index++) {
          ds[index] = solution[Geometry.levels[level].surfaces[i]->global_index_mapping[index]];
        }
        std::string file_2 = Geometry.levels[level].surfaces[i]->output_results(ds, "pml_domain" +
 std::to_string(level));
        filenames.push_back(file_2);
      }
    }
  }

  // End of core output
  if(level != 0) {
  //  child->output_results();
  }

  print_info("Hierarchical::output_results()", "End on level " + std::to_string(level));
  timer.stop();
  GlobalTimerManager.leave_context(level);
  return ret;
}
```

print_solve_counter_list()

`void HierarchicalProblem::print_solve_counter_list ( )`

This function uses the return values of compute_flobal_solve_counter to create some CLI output.

The function is recursive.

Definition at line 137 of file HierarchicalProblem.cpp.

```
137                                                      {
138   unsigned int n_solves_on_level = compute_global_solve_counter();
139   if(GlobalParams.MPI_Rank == 0) {
140     std::cout « "On level " « level « " there were " « n_solves_on_level « " solves." « std::endl;
141   }
142   if(level != 0) {
143     child->print_solve_counter_list();
144   }
145 }
```

References compute_global_solve_counter().

write_multifile_output()

```
virtual void HierarchicalProblem::write_multifile_output (
             const std::string & filename,
             bool apply_coordinate_transform )  [pure virtual]
```

Not implemented on this level, see derived classes.

Parameters

filename	
apply_coordinate_transform	

Implemented in LocalProblem, and NonLocalProblem.

The documentation for this class was generated from the following files:

- Code/Hierarchy/HierarchicalProblem.h
- Code/Hierarchy/HierarchicalProblem.cpp

35 HSIEPolynomial Class Reference

This class basically represents a polynomial and its derivative. It is required for the HSIE implementation.

`#include <HSIEPolynomial.h>`

Public Member Functions

- ComplexNumber evaluate (ComplexNumber x)

  Evaluates the polynomial represented by this object at the given position x.
- ComplexNumber evaluate_dx (ComplexNumber x)

  Evaluates the derivative of the polynomial represented by this object at the given position x.
- void update_derivative ()

  Updates the cached data for faster evaluation of the derivative.
- **HSIEPolynomial** (unsigned int dim, ComplexNumber k0)
- **HSIEPolynomial** (DofData &data, ComplexNumber k_0)
- **HSIEPolynomial** (std::vector< ComplexNumber > in_a, ComplexNumber k0)
- HSIEPolynomial **applyD** ()
- HSIEPolynomial **applyI** ()
- void **multiplyBy** (ComplexNumber factor)
- void **multiplyBy** (double factor)
- void **applyTplus** (ComplexNumber u_0)
- void **applyTminus** (ComplexNumber u_0)
- void **applyDerivative** ()
- void **add** (HSIEPolynomial b)

Static Public Member Functions

- static void computeDandI (unsigned int dim, ComplexNumber k_0)

  Prepares the Tensors D and I that are required for some of the computations.
- static HSIEPolynomial **PsiMinusOne** (ComplexNumber k0)
- static HSIEPolynomial **PsiJ** (int j, ComplexNumber k0)
- static HSIEPolynomial **ZeroPolynomial** ()
- static HSIEPolynomial **PhiMinusOne** (ComplexNumber k0)
- static HSIEPolynomial **PhiJ** (int j, ComplexNumber k0)

Public Attributes

- std::vector< ComplexNumber > **a**
- std::vector< ComplexNumber > **da**
- ComplexNumber **k0**

Static Public Attributes

- static bool **matricesLoaded** = false
- static dealii::FullMatrix< ComplexNumber > **D**
- static dealii::FullMatrix< ComplexNumber > **I**

35.1 Detailed Description

This class basically represents a polynomial and its derivative. It is required for the HSIE implementation.

The core data in this class is a vector a, which stores the coefficients of the polynomials and a vector da, which stores the coefficients of the derivative. Both can be evaluated for a given x with the respective functions. Additionally, there are functions to initialize a polynomial that are required by the hardy space infinite elements and some operators can be applied (like T_+ and T_-). As an important remark: The value κ_0 used in HSIE is also kept in these values because we want to be able to apply the operators D and I to one a polynomial. Since they aren't cheap to compute, I precomute them once as static members of this class. If you only intend to use evaluation, evaluation of the derivative, summation and multiplication with constants, then that value is not relevant.

See also

HSIESurface

Definition at line 31 of file HSIEPolynomial.h.

35.2 Member Function Documentation

computeDandI()

```
void HSIEPolynomial::computeDandI (
            unsigned int dim,
            ComplexNumber k_0 )  [static]
```

Prepares the Tensors D and I that are required for some of the computations.

For the defnition of D see the publication on "High order Curl-conforming Hardy spce infinite elements for exterior Maxwell problems" equation 21. D has tri-diagonal shape and represents the derivative for the Laplace-Moebius transformed shape of a function. The matrix I is the inverse of D and also gets computed in this function. These matrices are required in many places and never change. They, therefore, are only computed once and made available statically. The operator D (and I in turn) can be applied to polynomials of any degree. The computation of I, however gets more expensive the larger the maximal degree of the polynomials becomes. We therefore provide the maximal value of the dimension of polynomials.

Parameters

dim	Maximal polynomial degree of polynomials that D and I should be applied to.
k_0	This is a parameter of HSIE and also impacts D (and I).

Returns

Nothing.

Definition at line 10 of file HSIEPolynomial.cpp.

```
10                                                         {
11   HSIEPolynomial::D.reinit(dimension, dimension);
12   for (unsigned int i = 0; i < dimension; i++) {
13     for (unsigned int j = 0; j < dimension; j++) {
14       HSIEPolynomial::D.set(i, j, matrixD(i, j, k0));
15     }
16   }
17
18   HSIEPolynomial::I.copy_from(HSIEPolynomial::D);
19   HSIEPolynomial::I.invert(HSIEPolynomial::D);
20   HSIEPolynomial::matricesLoaded = true;
21 }
```

Referenced by HSIESurface::check_dof_assignment_integrity(), and HSIESurface::fill_matrix().

evaluate()

```
ComplexNumber HSIEPolynomial::evaluate (
              ComplexNumber x )
```

Evaluates the polynomial represented by this object at the given position x.

Performs the evaluation of the polynomial at x, meaning

$$f(x) = \sum_{i=0}^{D} a_i x^i .$$

Parameters

x	The poisition to evaluate the polynomial at.

Returns

The value of the polynomial at x.

Definition at line 23 of file HSIEPolynomial.cpp.

```
23                                              {
24   ComplexNumber ret(a[0]);
25   ComplexNumber x = x_in;
26   for (unsigned long i = 1; i < a.size(); i++) {
27     ret += a[i] * x;
28     x = x * x_in;
29   }
```

```
  return ret;
}
```

evaluate_dx()

`ComplexNumber HSIEPolynomial::evaluate_dx ( ComplexNumber x )`

Evaluates the derivative of the polynomial represented by this object at the given position x.

Performs the evaluation of the derivative of the polynomial at x, meaning

$$f(x) = \sum_{i=1}^{D-1} i a_i x^{i-1}.$$

Parameters

x	The poisition to evaluate the derivative at.

Returns

The value of the derivative of the polynomial at x.

Definition at line 33 of file HSIEPolynomial.cpp.

```
33                                                     {
34   ComplexNumber ret(da[0]);
35   ComplexNumber x = x_in;
36   for (unsigned long i = 1; i < da.size(); i++) {
37     ret += da[i] * x;
38     x = x * x_in;
39   }
40   return ret;
41 }
```

update_derivative()

`void HSIEPolynomial::update_derivative ( )`

Updates the cached data for faster evaluation of the derivative.

Internally, the derivative is stored as a polynomial. The cached parameters are simply ia_i. This function gets called a lot internally, so calling it yourself is likely not required.

Returns

Nothing.

Definition at line 105 of file HSIEPolynomial.cpp.

```
105                                     {
106   da = std::vector<ComplexNumber>();
107   for (unsigned int i = 1; i < a.size(); i++) {
108     da.emplace_back(i * a[i].real(), i * a[i].imag());
109   }
110 }
```

The documentation for this class was generated from the following files:

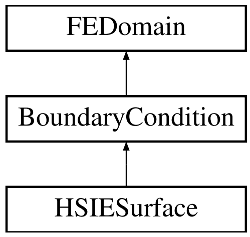
- Code/BoundaryCondition/HSIEPolynomial.h
- Code/BoundaryCondition/HSIEPolynomial.cpp

36 HSIESurface Class Reference

This class implements Hardy space infinite elements on a provided surface.

`#include <HSIESurface.h>`

Inheritance diagram for HSIESurface:



Public Member Functions

- HSIESurface (unsigned int surface, unsigned int level)

  Constructor.

- std::vector< HSIEPolynomial > build_curl_term_q (unsigned int order, const dealii::Tensor< 1, 2 > gradient)

  Builds a curl-type term required during the assembly of the system matrix for a q-type dof.

- std::vector< HSIEPolynomial > build_curl_term_nedelec (unsigned int order, const dealii::Tensor< 1, 2 > gradient_component_0, const dealii::Tensor< 1, 2 > gradient_component_1, const double value_component_0, const double value_component_1)

  Builds a curl-type term required during the assembly of the system matrix for a nedelec-type dof.

- std::vector< HSIEPolynomial > build_non_curl_term_q (unsigned int order, const double value_component)

  Builds a non-curl-type term required during the assembly of the system matrix for a q-type dof.

- std::vector< HSIEPolynomial > **build_non_curl_term_nedelec** (unsigned int, const double, const double)
- void **set_V0** (Position pos)
- auto **get_dof_data_for_cell** (CellIterator2D pointer_q, CellIterator2D pointer_n) -> DofDataVector
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix *matrix, NumericVectorDistributed *rhs, Constraints *constraints) override

  Writes all entries to the system matrix that originate from dof couplings on this surface.

- void fill_matrix_for_edge (BoundaryId other_bid, dealii::PETScWrappers::MPI::SparseMatrix *matrix, NumericVectorDistributed *rhs, Constraints *constraints)

  Not yet implemented.

- void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗in_constriants) override

  Fills a sparsity pattern for all the dofs active in this boundary condition.

- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

  Checks if a point is at an outward surface of the boundary triangulation.

- auto get_vertices_for_boundary_id (BoundaryId in_bid) -> std::vector< unsigned int >

  Get the vertices located at the provided boundary.

- auto get_n_vertices_for_boundary_id (BoundaryId in_bid) -> unsigned int

  Get the number of vertices on th eboundary with id.

- auto get_lines_for_boundary_id (BoundaryId in_bid) -> std::vector< unsigned int >

  Get the lines shared with the boundary in_bid.

- auto get_n_lines_for_boundary_id (BoundaryId in_bid) -> unsigned int

  Get the number of lines for boundary id object.

- auto compute_n_edge_dofs () -> DofCountsStruct

  Computes the number of edge dofs for this surface.

- auto compute_n_vertex_dofs () -> DofCountsStruct

  Computes the number of vertex dofs and returns them as a DofCounts object (see above).

- auto compute_n_face_dofs () -> DofCountsStruct

  Computes the number of face dofs and returns them as a Dofcounts object (see above).

- auto compute_dofs_per_edge (bool only_hsie_dofs) -> DofCount

  Computes the number of dofs per edge.

- auto compute_dofs_per_face (bool only_hsie_dofs) -> DofCount

  Computes the number of dofs on every surface face.

- auto compute_dofs_per_vertex () -> DofCount

  Computes the number of dofs on every vertex.

- void initialize () override

  Initializes the data structures.

- void initialize_dof_handlers_and_fe ()

  Part of the initialization function.

- void update_dof_counts_for_edge (CellIterator2D cell, unsigned int edge, DofCountsStruct &in_dof_counts)

  Updates the numbers of dofs for an edge.

- void update_dof_counts_for_face (CellIterator2D cell, DofCountsStruct &in_dof_counts)

  Updates the numbers of dofs for a face.

- void update_dof_counts_for_vertex (CellIterator2D cell, unsigned int edge, unsigned int vertex, DofCountsStruct &in_dof_coutns)

  Updates the dof counts for a vertex.

- void register_new_vertex_dofs (CellIterator2D cell, unsigned int edge, unsigned int vertex)

  When building the datastructures, this function adds a new dof to the list of all vertex dofs.

- void register_new_edge_dofs (CellIterator2D cell, CellIterator2D cell_2, unsigned int edge)

When building the datastructures, this function adds a new dof to the list of all edge dofs.

- void register_new_surface_dofs (CellIterator2D cell, CellIterator2D cell2)

  When building the datastructures, this function adds a new dof to the list of all face dofs.

- auto register_dof () -> DofNumber

  Increments the dof counter.

- void register_single_dof (std::string in_id, int in_hsie_order, int in_inner_order, DofType in_dof_type, DofDataVector &, unsigned int base_dof_index)

  Registers a new dof with a face base structure (first argument is string)

- void register_single_dof (unsigned int in_id, int in_hsie_order, int in_inner_order, DofType in_dof_type, DofDataVector &, unsigned int, bool orientation=true)

  Registers a new dof with a edge or vertex base structure (first argument is int)

- ComplexNumber evaluate_a (std::vector< HSIEPolynomial > &u, std::vector< HSIEPolynomial > &v, dealii::Tensor< 2, 3, double > G)

  Evaluates the function a from the publication.

- void transform_coordinates_in_place (std::vector< HSIEPolynomial > *in_vector)

  All functions for this type assume that x is the infinte direction.

- bool check_dof_assignment_integrity ()

  Checks some internal integrity conditions.

- bool check_number_of_dofs_for_cell_integrity ()

  Part of the function above.

- auto get_dof_data_for_base_dof_nedelec (DofNumber base_dof_index) -> DofDataVector

  Get the dof data for a nedelec base dof.

- auto get_dof_data_for_base_dof_q (DofNumber base_dof_index) -> DofDataVector

  Get the dof data for base dof q.

- auto get_dof_association () -> std::vector< InterfaceDofData > override

  Get the dof association vector This is a part of the boundary condition interface and returns a list of all the dofs that couple to the inner domain.

- auto undo_transform (dealii::Point< 2 >) -> Position

  Returns the 3D form of a point for a provided 2D position in the surface triangulation.

- auto undo_transform_for_shape_function (dealii::Point< 2 >) -> Position

  Transforms the 2D value of a surface dof shape function into a 3D field in the actual 3D coordinates.

- void add_surface_relevant_dof (InterfaceDofData in_index_and_orientation)

  If a new dof is active on the surface and should be returned by get_dof_association, this function adds it to the list.

- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

  Get the dof association by boundary id If two neighboring surfaces have HSIE on them, this can be used to compute on each surface which dofs are at the outside surface they share and the resulting data can be used to build the coupling terms.

- void clear_user_flags ()

  We sometimes use deal.II user flags when iterating over the triangulation.

- void set_b_id_uses_hsie (unsigned int index, bool does)

  It is usefull to know, if a neighboring surface is also using hsie.
- auto build_fad_for_cell (CellIterator2D cell) -> FaceAngelingData

  computes the face angeling data.
- void compute_extreme_vertex_coordinates ()

  This computes the coordinate ranges of the surface mesh vertices and caches the result.
- auto vertex_positions_for_ids (std::vector< unsigned int > ids) -> std::vector< Position >

  Computes all vertex positions for a set of vertex ids.
- auto line_positions_for_ids (std::vector< unsigned int > ids) -> std::vector< Position >

  Computes the positions for line ids.
- std::string output_results (const dealii::Vector< ComplexNumber > &, std::string) override

  Does nothing.
- DofCount compute_n_locally_owned_dofs () override

  Computes the number of locally owned dofs.
- DofCount compute_n_locally_active_dofs () override

  Compute the number of locally active dofs.
- void finish_dof_index_initialization () override

  This is a DofDomain via BoundaryCondition.
- void determine_non_owned_dofs () override

  Marks for every dof if it is locally owned or not.
- dealii::IndexSet compute_non_owned_dofs ()

  Returns an IndexSet with all dofs that are not locally owned.
- bool finish_initialization (DofNumber first_own_index) override

  Finishes the DofDomainInitialization.

Public Attributes

- DofDataVector **face_dof_data**
- DofDataVector **edge_dof_data**
- DofDataVector **vertex_dof_data**
- DofCount **n_edge_dofs**
- DofCount **n_face_dofs**
- DofCount **n_vertex_dofs**

36.1 Detailed Description

This class implements Hardy space infinite elements on a provided surface.

This object implements the BoundaryCondition interface. It should be considered however, that this boundary condition type is extremely complex, represented in the number of functions and lines of code it consists of. It is recommended to read the paper "High order Curl-conforming Hardy spce infinite elements for exterior Maxwell problems" for an introduction.

In many places, you will see a distinction between q and nedelec in this implementation: Infinite cells have two types of edges: finite ones and infinite ones. The finite ones are the ones on the surface. The infinite ones point in the infinite direction. The cell is basically a normal nedelec cell, but if the edge a dof is associated with, is infinite, it requires special treatment. We treat these dofs as if they were nodal elements with the center of their hat function being the base point of their inifite edge. We therefore need most computations for nodal and for edge elements.

In the assembly loop, we have to compute terms like $\langle \nabla \times u, \nabla \times v \rangle$ and $\langle u, v \rangle$.

There are NO 3D triangulations here! We only work with a 2D surface triangulation. Therefore, often when we talk about a cell, that has different properties then in objects like PMLSurface or InnerDomain, where the mesh is 3D.

For more details on this type of intinite element, see sections 4.4.4, 5.1.3 and 5.6.

Definition at line 48 of file HSIESurface.h.

36.2 Constructor & Destructor Documentation

HSIESurface()

```
HSIESurface::HSIESurface (
            unsigned int surface,
            unsigned int level )
```

Constructor.

Prepares the data structures and sets two values.

Parameters

surface	BoundaryId of the surface of the InnerDomain this condition is going to couple to.
level	the level of sweeping this object is used on.

Definition at line 18 of file HSIESurface.cpp.

```
    : BoundaryCondition(surface, in_level, Geometry.surface_extremal_coordinate[surface]),
      order(GlobalParams.HSIE_polynomial_degree),
      dof_h_q(Geometry.surface_meshes[surface]),
      Inner_Element_Order(GlobalParams.Nedelec_element_order),
      fe_nedelec(Inner_Element_Order),
      fe_q(Inner_Element_Order + 1),
      kappa(2.0 * GlobalParams.Pi / GlobalParams.Lambda) {
    dof_h_nedelec.reinit(Geometry.surface_meshes[surface]);
    dof_h_q.reinit(Geometry.surface_meshes[surface]);
    set_mesh_boundary_ids();
    dof_counter = 0;
    k0 = GlobalParams.kappa_0;
}
```

36.3 Member Function Documentation

add_surface_relevant_dof()

```
void HSIESurface::add_surface_relevant_dof (
            InterfaceDofData in_index_and_orientation )
```

If a new dof is active on the surface and should be returned by get_dof_association, this function adds it to the list.

Parameters

in_index_and_orientation	Index of the dof and point it should be sorted by.

Definition at line 889 of file HSIESurface.cpp.

```
889                                                            {
890   surface_dofs.emplace_back(dof_data);
891 }
```

build_curl_term_nedelec()

```
std::vector< HSIEPolynomial > HSIESurface::build_curl_term_nedelec (
            unsigned int order,
            const dealii::Tensor< 1, 2 > gradient_component_0,
            const dealii::Tensor< 1, 2 > gradient_component_1,
            const double value_component_0,
            const double value_component_1 )
```

Builds a curl-type term required during the assembly of the system matrix for a nedelec-type dof.

Same as above but for a nedelec dof. The computation requires two components of the gradient of the shape function and two values of the shape function. The former are provided as Tensors, the latter as individual doubles.

Parameters

order	Order of the dof we work with.
gradient_component_0	Shape function gradient component 0.
gradient_component_1	Shape function gradient component 1.
value_component_0	Value of shape function component 0.
value_component_1	Value of shape function component 1.

Returns

A three component vector containing the curl term required during assembly.

Definition at line 550 of file HSIESurface.cpp.

```
555                                        {
```

```
std::vector<HSIEPolynomial> ret;
HSIEPolynomial temp = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
temp.multiplyBy(fe_shape_gradient_component_0[1]);
temp.applyI();
HSIEPolynomial temp2 = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
temp2.multiplyBy(-1.0 * fe_shape_gradient_component_1[0]);
temp2.applyI();
temp.add(temp2);
ret.push_back(temp);

temp = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
temp.multiplyBy(-1.0 * fe_shape_value_component_1);
temp.applyDerivative();
ret.push_back(temp);

temp = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
temp.multiplyBy(fe_shape_value_component_0);
temp.applyDerivative();
ret.push_back(temp);

transform_coordinates_in_place(&ret);
return ret;
}
```

References transform_coordinates_in_place().

build_curl_term_q()

std::vector< HSIEPolynomial > HSIESurface::build_curl_term_q (
unsigned int *order*,
const dealii::Tensor< 1, 2 > *gradient*)

Builds a curl-type term required during the assembly of the system matrix for a q-type dof.

This computes the curl as a std::vetor for a monomial of given order for a shape dof, whoose projected shape function on the surface is nodal (q), and requires a local gradient value as input.

Parameters

order	Order of the dof we work with.
gradient	Local surface gradient.

Returns

A three component vector containing the curl term required during assembly.

Definition at line 595 of file HSIESurface.cpp.

```
595 {
596 std::vector<HSIEPolynomial> ret;
597 ret.push_back(HSIEPolynomial::ZeroPolynomial());
598 HSIEPolynomial temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
599 temp.multiplyBy(fe_gradient[1]);
600 ret.push_back(temp);
601 temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
602 temp.multiplyBy(-1.0 * fe_gradient[0]);
603 ret.push_back(temp);
604 transform_coordinates_in_place(&ret);
605 return ret;
606 }
```

References transform_coordinates_in_place().

build_fad_for_cell()

```
auto HSIESurface::build_fad_for_cell (
            CellIterator2D cell ) -> FaceAngelingData
```

computes the face angeling data.

Face angeling data describes if the dofs here are exactly orthogonal to the surface or if they are somehow at an angle.

Parameters

cell	The cell to compute the data for

Returns

FaceAngelingData

Definition at line 135 of file HSIESurface.cpp.

```
135                                                                         {
136   FaceAngelingData ret;
137   for(unsigned int i = 0; i < ret.size(); i++) {
138     ret[i].is_x_angled = false;
139     ret[i].is_y_angled = false;
140     ret[i].position_of_base_point = {};
141   }
142   return ret;
143 }
```

Referenced by fill_matrix().

build_non_curl_term_q()

```
std::vector< HSIEPolynomial > HSIESurface::build_non_curl_term_q (
            unsigned int order,
            const double value_component )
```

Builds a non-curl-type term required during the assembly of the system matrix for a q-type dof.

The computation requires the value of a shape function.

Parameters

order	Order of the dof we work with.
value_component	Value of shape function component.

Returns

A three component vector containing the curl term required during assembly.

Definition at line 608 of file HSIESurface.cpp.

```
609                                                                      {
610   std::vector<HSIEPolynomial> ret;
```

```
611   HSIEPolynomial temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
612   temp.multiplyBy(fe_shape_value);
613   temp = temp.applyD();
614   ret.push_back(temp);
615   ret.push_back(HSIEPolynomial::ZeroPolynomial());
616   ret.push_back(HSIEPolynomial::ZeroPolynomial());
617   transform_coordinates_in_place(&ret);
618   return ret;
619 }
```

References transform_coordinates_in_place().

check_dof_assignment_integrity()

`bool HSIESurface::check_dof_assignment_integrity ( )`

Checks some internal integrity conditions.

Returns

true Everything is fine.

false Everythin is not fine.

Definition at line 709 of file HSIESurface.cpp.

```
709                                                   {
710   HSIEPolynomial::computeDandI(order + 2, k0);
711   auto it = dof_h_nedelec.begin_active();
712   auto end = dof_h_nedelec.end();
713   auto it2 = dof_h_q.begin_active();
714   unsigned int counter = 1;
715   for (; it != end; ++it) {
716     if (it->id() != it2->id()) std::cout « "Identity failure!" « std::endl;
717     DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
718     std::vector<unsigned int> q_dofs(fe_q.dofs_per_cell);
719     std::vector<unsigned int> n_dofs(fe_nedelec.dofs_per_cell);
720     it2->get_dof_indices(q_dofs);
721     it->get_dof_indices(n_dofs);
722     std::vector<unsigned int> local_related_fe_index;
723     bool found = false;
724     for (unsigned int i = 0; i < cell_dofs.size(); i++) {
725       found = false;
726       if (cell_dofs[i].type == DofType::RAY ||
727           cell_dofs[i].type == DofType::IFFb) {
728         for (unsigned int j = 0; j < q_dofs.size(); j++) {
729           if (q_dofs[j] == cell_dofs[i].base_dof_index) {
730             local_related_fe_index.push_back(j);
731             found = true;
732           }
733         }
734       } else {
735         for (unsigned int j = 0; j < n_dofs.size(); j++) {
736           if (n_dofs[j] == cell_dofs[i].base_dof_index) {
737             local_related_fe_index.push_back(j);
738             found = true;
739           }
740         }
741       }
742       if (!found) {
743         std::cout « "Error in dof assignment integrity!" « std::endl;
744       }
745     }
746
747     if (local_related_fe_index.size() != cell_dofs.size()) {
748       std::cout « "Mismatch in cell " « counter
749                 « ": Found indices: " « local_related_fe_index.size()
750                 « " of a total " « cell_dofs.size() « std::endl;
```

```
      return false;
    }
    counter++;
    it2++;
  }

  return true;
}
```

References HSIEPolynomial::computeDandI().

check_number_of_dofs_for_cell_integrity()

`bool HSIESurface::check_number_of_dofs_for_cell_integrity ( )`

Part of the function above.

Returns

true fine

false not fine-

Definition at line 760 of file HSIESurface.cpp.

```
                                                          {
  auto it = dof_h_nedelec.begin_active();
  auto it2 = dof_h_q.begin_active();
  auto end = dof_h_nedelec.end();
  const unsigned int dofs_per_cell = 4 * compute_dofs_per_vertex() +
                                     4 * compute_dofs_per_edge(false) +
                                     compute_dofs_per_face(false);
  unsigned int counter = 0;
  for (; it != end; ++it) {
    DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
    if (cell_dofs.size() != dofs_per_cell) {
      for (unsigned int i = 0; i < 7; i++) {
        unsigned int count = 0;
        for (unsigned int j = 0; j < cell_dofs.size(); ++j) {
          if (cell_dofs[j].type == i) count++;
        }
        std::cout « cell_dofs.size() « " vs. " « dofs_per_cell « std::endl;
        std::cout « "For type " « i « " I found " « count « " dofs" « std::endl;
      }
      return false;
    }
    counter++;
    it2++;
  }
  return true;
}
```

References compute_dofs_per_edge(), compute_dofs_per_face(), and compute_dofs_per_vertex().

clear_user_flags()

`void HSIESurface::clear_user_flags ( )`

We sometimes use deal.II user flags when iterating over the triangulation.

This resets them.

Definition at line 787 of file HSIESurface.cpp.

```
{
  auto it = dof_h_nedelec.begin();
  const auto end = dof_h_nedelec.end();
  while (it != end) {
    it->clear_user_flag();
    for (unsigned int i = 0; i < 4; i++) {
      it->face(i)->clear_user_flag();
    }
    it++;
  }
}
```

compute_dofs_per_edge()

unsigned int HSIESurface::compute_dofs_per_edge (
    bool *only_hsie_dofs*) -> DofCount

Computes the number of dofs per edge.

Parameters

only_hsie_dofs	if set to true, it only computes the number of non-inner dofs, ie only the additional dofs introduced by the boundary condition.

Returns

DofCount Number of dofs.

Definition at line 330 of file HSIESurface.cpp.

```
330 {
331   unsigned int ret = 0;
332   const unsigned int INNER_REAL_DOFS_PER_LINE = fe_nedelec.dofs_per_line;
333
334   if (!only_hsie_dofs) {
335     ret += INNER_REAL_DOFS_PER_LINE;
336   }
337
338   ret += INNER_REAL_DOFS_PER_LINE * (order + 1)
339       + (INNER_REAL_DOFS_PER_LINE - 1) * (order + 2);
340
341   return ret;
342 }
```

Referenced by check_number_of_dofs_for_cell_integrity(), fill_matrix(), and update_dof_counts_for_edge().

compute_dofs_per_face()

unsigned int HSIESurface::compute_dofs_per_face (
    bool *only_hsie_dofs*) -> DofCount

Computes the number of dofs on every surface face.

Parameters

only_hsie_dofs	if set to true, it only computes the number of non-inner dofs, ie only the additional dofs introduced by the boundary condition.

Returns

DofCount

Definition at line 344 of file HSIESurface.cpp.

```
344                                                                {
345   unsigned int ret = 0;
346   const unsigned int INNER_REAL_NEDELEC_DOFS_PER_FACE =
347       fe_nedelec.dofs_per_cell -
348       dealii::GeometryInfo<2>::faces_per_cell * fe_nedelec.dofs_per_face;
349
350   ret = INNER_REAL_NEDELEC_DOFS_PER_FACE * (order + 2) * 3;
351   if (only_hsie_dofs) {
352     ret -= INNER_REAL_NEDELEC_DOFS_PER_FACE;
353   }
354   return ret;
355 }
```

Referenced by check_number_of_dofs_for_cell_integrity(), fill_matrix(), and update_dof_counts_for_face().

compute_dofs_per_vertex()

```
unsigned int HSIESurface::compute_dofs_per_vertex ( ) -> DofCount
```

Computes the number of dofs on every vertex.

All vertex dofs are automatically hardy space dofs, therefore the parameter does not exist on this fucntion.

Returns

DofCount

Definition at line 357 of file HSIESurface.cpp.

```
357                                                  {
358   unsigned int ret = order + 2;
359
360   return ret;
361 }
```

Referenced by check_number_of_dofs_for_cell_integrity(), fill_matrix(), and update_dof_counts_for_vertex().

compute_n_edge_dofs()

```
DofCountsStruct HSIESurface::compute_n_edge_dofs ( ) -> DofCountsStruct
```

Computes the number of edge dofs for this surface.

The return type contains the number of pure HSIE dofs, inner dofs active on the surface and the sum of both.

Returns

DofCountsStruct containing the dof counts.

Definition at line 267 of file HSIESurface.cpp.

```
267                                                  {
268   DoFHandler<2>::active_cell_iterator cell;
269   DoFHandler<2>::active_cell_iterator cell2;
270   DoFHandler<2>::active_cell_iterator endc;
271   endc = dof_h_nedelec.end();
```

```
  DofCountsStruct ret;
  cell2 = dof_h_q.begin_active();
  Geometry.surface_meshes[b_id].clear_user_flags();
  for (cell = dof_h_nedelec.begin_active(); cell != endc; cell++) {
    for (unsigned int edge = 0; edge < GeometryInfo<2>::lines_per_cell; edge++) {
      if (!cell->line(edge)->user_flag_set()) {
        update_dof_counts_for_edge(cell, edge, ret);
        register_new_edge_dofs(cell, cell2, edge);
        cell->line(edge)->set_user_flag();
      }
    }
    cell2++;
  }
  return ret;
}
```

Referenced by initialize().

compute_n_face_dofs()

`DofCountsStruct HSIESurface::compute_n_face_dofs ( ) -> DofCountsStruct`

Computes the number of face dofs and returns them as a Dofcounts object (see above).

Returns

DofCountsStruct The dof counts.

Definition at line 311 of file HSIESurface.cpp.

```
311                                                       {
312   std::set<std::string> touched_faces;
313   DoFHandler<2>::active_cell_iterator cell;
314   DoFHandler<2>::active_cell_iterator cell2;
315   DoFHandler<2>::active_cell_iterator endc;
316   endc = dof_h_nedelec.end();
317   DofCountsStruct ret;
318   cell2 = dof_h_q.begin_active();
319   for (cell = dof_h_nedelec.begin_active(); cell != endc; cell++) {
320     if (touched_faces.end() == touched_faces.find(cell->id().to_string())) {
321       update_dof_counts_for_face(cell, ret);
322       register_new_surface_dofs(cell, cell2);
323       touched_faces.insert(cell->id().to_string());
324     }
325     cell2++;
326   }
327   return ret;
328 }
```

References register_new_surface_dofs(), and update_dof_counts_for_face().

Referenced by initialize().

compute_n_locally_active_dofs()

`DofCount HSIESurface::compute_n_locally_active_dofs ( ) [override], [virtual]`

Compute the number of locally active dofs.

For the meaning of active, check the dealii glossary for a definition.

Returns

DofCount

Implements FEDomain.

Definition at line 964 of file HSIESurface.cpp.

```
964                                                    {
965   return dof_counter;
966 }
```

compute_n_locally_owned_dofs()

`DofCount HSIESurface::compute_n_locally_owned_dofs ( )  [override], [virtual]`

Computes the number of locally owned dofs.

For the meaning of owned, check the dealii glossary for a definition.

Returns

DofCount Number of locally owned dofs.

Implements FEDomain.

Definition at line 959 of file HSIESurface.cpp.

```
959                                                   {
960   IndexSet non_owned_dofs = compute_non_owned_dofs();
961   return dof_counter - non_owned_dofs.n_elements();
962 }
```

References compute_non_owned_dofs().

compute_n_vertex_dofs()

`DofCountsStruct HSIESurface::compute_n_vertex_dofs ( ) -> DofCountsStruct`

Computes the number of vertex dofs and returns them as a DofCounts object (see above).

Returns

DofCountsStruct The dof counts.

Definition at line 288 of file HSIESurface.cpp.

```
288                                                   {
289   std::set<unsigned int> touched_vertices;
290   DoFHandler<2>::active_cell_iterator cell;
291   DoFHandler<2>::active_cell_iterator endc;
292   endc = dof_h_q.end();
293   DofCountsStruct ret;
294   for (cell = dof_h_q.begin_active(); cell != endc; cell++) {
295     // for each edge
296     for (unsigned int vertex = 0; vertex < GeometryInfo<2>::vertices_per_cell;
297          vertex++) {
298       unsigned int idx = cell->vertex_dof_index(vertex, 0);
299       if (touched_vertices.end() == touched_vertices.find(idx)) {
300         // handle it
301         update_dof_counts_for_vertex(cell, idx, vertex, ret);
302         register_new_vertex_dofs(cell, idx, vertex);
303         // remember that it has been handled
304         touched_vertices.insert(idx);
```

```
305       }
306     }
307   }
308   return ret;
309 }
```

References register_new_vertex_dofs(), and update_dof_counts_for_vertex().

Referenced by initialize().

compute_non_owned_dofs()

`dealii::IndexSet HSIESurface::compute_non_owned_dofs ( )`

Returns an IndexSet with all dofs that are not locally owned.

All dofs that are not locally owned must retrieve their global index from somewhere else (usually the inner domain) since the owner gives the number. This function helps prepare that step.

Returns

dealii::IndexSet All the dofs that are not locally owned in a deal.II::IndexSet

Definition at line 1017 of file HSIESurface.cpp.

```
1017                                                     {
1018   IndexSet non_owned_dofs(dof_counter);
1019   for(auto it : surface_dofs) {
1020     non_owned_dofs.add_index(it.index);
1021   }
1022   for(auto surf : adjacent_boundaries) {
1023     if(Geometry.levels[level].surface_type[surf] == SurfaceType::NEIGHBOR_SURFACE) {
1024       if(surf % 2 == 0) {
1025         std::vector<InterfaceDofData> dofs_data = get_dof_association_by_boundary_id(surf);
1026         for(auto it : dofs_data) {
1027           non_owned_dofs.add_index(it.index);
1028         }
1029       }
1030     }
1031   }
1032   return non_owned_dofs;
1033 }
```

Referenced by compute_n_locally_owned_dofs(), and determine_non_owned_dofs().

determine_non_owned_dofs()

`void HSIESurface::determine_non_owned_dofs ( ) [override], [virtual]`

Marks for every dof if it is locally owned or not.

This fulfills the DofDomain interface.

Implements FEDomain.

Definition at line 995 of file HSIESurface.cpp.

```
995                                             {
996   IndexSet non_owned_dofs = compute_non_owned_dofs();
997   const unsigned int n_dofs = non_owned_dofs.n_elements();
998   std::vector<unsigned int> local_dofs(n_dofs);
999   for(unsigned int i = 0; i < n_dofs; i++) {
1000     local_dofs[i] = non_owned_dofs.nth_index_in_set(i);
```

```
  }
  mark_local_dofs_as_non_local(local_dofs);
}
```

References compute_non_owned_dofs(), and FEDomain::mark_local_dofs_as_non_local().

evaluate_a()

```
ComplexNumber HSIESurface::evaluate_a (
            std::vector< HSIEPolynomial > & u,
            std::vector< HSIEPolynomial > & v,
            dealii::Tensor< 2, 3, double > G )
```

Evaluates the function a from the publication.

See equation 7 in "High order Curl-conforming Hardy spce infinite elements for exterior Maxwell problems".

Parameters

u	Term u in the equation
v	Term v in the equation
G	Term G in the equation

Returns

ComplexNumber Value of a.

Definition at line 538 of file HSIESurface.cpp.

```
538                   {
539   ComplexNumber result(0, 0);
540   for(unsigned int i = 0; i < 3; i++) {
541     for (unsigned int j = 0; j < 3; j++) {
542       for (unsigned int k = 0; k < std::min(u[i].a.size(), v[j].a.size()); k++) {
543         result += G[i][j] * u[i].a[k] * v[j].a[k];
544       }
545     }
546   }
547   return result;
548 }
```

fill_matrix()

```
void HSIESurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints )  [override], [virtual]
```

Writes all entries to the system matrix that originate from dof couplings on this surface.

It also sets the values in the rhs and it uses the constraints object to condense the matrix entries automatically (see deal.IIs description on distribute_dofs_local_to_global with a constraint object).

Parameters

matrix	The matrix to write into.
rhs	The right hand side vector (b) in Ax = b.
constraints	These represent inhomogenous and hanging node constraints that are used to condense the matrix.

Implements BoundaryCondition.

Definition at line 145 of file HSIESurface.cpp.

```
{
 HSIEPolynomial::computeDandI(order + 2, k0);
 auto it = dof_h_nedelec.begin();
 auto end = dof_h_nedelec.end();

 QGauss<2> quadrature_formula(2);
 FEValues<2, 2> fe_q_values(fe_q, quadrature_formula,
                            update_values | update_gradients |
                                update_JxW_values | update_quadrature_points);
 FEValues<2, 2> fe_n_values(fe_nedelec, quadrature_formula,
                            update_values | update_gradients |
                                update_JxW_values | update_quadrature_points);
 std::vector<Point<2» quadrature_points;
 const unsigned int dofs_per_cell =
     GeometryInfo<2>::vertices_per_cell * compute_dofs_per_vertex() +
     GeometryInfo<2>::lines_per_cell * compute_dofs_per_edge(false) +
     compute_dofs_per_face(false);
 FullMatrix<ComplexNumber> cell_matrix(dofs_per_cell, dofs_per_cell);
 unsigned int cell_counter = 0;
 auto it2 = dof_h_q.begin();
 for (; it != end; ++it) {
   FaceAngelingData fad = build_fad_for_cell(it);
   JacobianForCell jacobian_for_cell = {fad, b_id, additional_coordinate};
   cell_matrix = 0;
   DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
   std::vector<HSIEPolynomial> polynomials;
   std::vector<unsigned int> q_dofs(fe_q.dofs_per_cell);
   std::vector<unsigned int> n_dofs(fe_nedelec.dofs_per_cell);
   it2->get_dof_indices(q_dofs);
   it->get_dof_indices(n_dofs);
   for (unsigned int i = 0; i < cell_dofs.size(); i++) {
     polynomials.push_back(HSIEPolynomial(cell_dofs[i], k0));
   }
   std::vector<unsigned int> local_related_fe_index;
   for (unsigned int i = 0; i < cell_dofs.size(); i++) {
     if (cell_dofs[i].type == DofType::RAY || cell_dofs[i].type == DofType::IFFb) {
       for (unsigned int j = 0; j < q_dofs.size(); j++) {
         if (q_dofs[j] == cell_dofs[i].base_dof_index) {
           local_related_fe_index.push_back(j);
           break;
         }
       }
     } else {
       for (unsigned int j = 0; j < n_dofs.size(); j++) {
         if (n_dofs[j] == cell_dofs[i].base_dof_index) {
           local_related_fe_index.push_back(j);
           break;
         }
       }
     }
   }

   fe_n_values.reinit(it);
   fe_q_values.reinit(it2);
   quadrature_points = fe_q_values.get_quadrature_points();
   std::vector<double> jxw_values = fe_n_values.get_JxW_values();
   std::vector<std::vector<HSIEPolynomial» contribution_value;
   std::vector<std::vector<HSIEPolynomial» contribution_curl;
```

```
    JacobianAndTensorData C_G_J;
    for (unsigned int q_point = 0; q_point < quadrature_points.size(); q_point++) {
      C_G_J = jacobian_for_cell.get_C_G_and_J(quadrature_points[q_point]);
      for (unsigned int i = 0; i < cell_dofs.size(); i++) {
        DofData &u = cell_dofs[i];
        if (cell_dofs[i].type == DofType::RAY || cell_dofs[i].type == DofType::IFFb) {
          contribution_curl.push_back(
            build_curl_term_q(u.hsie_order, fe_q_values.shape_grad(local_related_fe_index[i], q_point)));
          contribution_value.push_back(
            build_non_curl_term_q(u.hsie_order, fe_q_values.shape_value(local_related_fe_index[i], q_point)));
        } else {
          contribution_curl.push_back(
            build_curl_term_nedelec(u.hsie_order,
              fe_n_values.shape_grad_component(local_related_fe_index[i], q_point, 0),
              fe_n_values.shape_grad_component(local_related_fe_index[i], q_point, 1),
              fe_n_values.shape_value_component(local_related_fe_index[i], q_point, 0),
              fe_n_values.shape_value_component(local_related_fe_index[i], q_point, 1)));
          contribution_value.push_back(
            build_non_curl_term_nedelec(u.hsie_order,
              fe_n_values.shape_value_component(local_related_fe_index[i], q_point, 0),
              fe_n_values.shape_value_component(local_related_fe_index[i], q_point, 1)));
        }
      }

      double JxW = jxw_values[q_point];
      const double eps_kappa_2 = Geometry.eps_kappa_2(undo_transform(quadrature_points[q_point]));
      for (unsigned int i = 0; i < cell_dofs.size(); i++) {
        for (unsigned int j = 0; j < cell_dofs.size(); j++) {
          ComplexNumber part = (evaluate_a(contribution_curl[i], contribution_curl[j], C_G_J.C) - eps_kappa_2 * evaluate_a(contribution_value[i], contribution_value[j], C_G_J.G)) * JxW;
          cell_matrix[i][j] += part;
        }
      }
    }
    std::vector<unsigned int> local_indices;
    for (unsigned int i = 0; i < cell_dofs.size(); i++) {
      local_indices.push_back(cell_dofs[i].global_index);
    }
    Vector<ComplexNumber> cell_rhs(cell_dofs.size());
    cell_rhs = 0;
    local_indices = transform_local_to_global_dofs(local_indices);
    constraints->distribute_local_to_global(cell_matrix, cell_rhs, local_indices, *matrix, *rhs, true);
    it2++;
    cell_counter++;
  }
  matrix->compress(dealii::VectorOperation::add);
}
```

References build_fad_for_cell(), compute_dofs_per_edge(), compute_dofs_per_face(), compute_dofs_per_vertex(), and HSIEPolynomial::computeDandI().

fill_matrix_for_edge()

```
void HSIESurface::fill_matrix_for_edge (
            BoundaryId other_bid,
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints )
```

Not yet implemented.

When using axis parallel infinite directions, the corner and edge domains requrie additional computation of coupling terms. The function computes the coupling terms for infinite edge cells.

Parameters

other_bid	BoundaryId of the surface that shares the edge with this surface.
matrix	The matrix to write into.
rhs	The right hand side vector to write into.
constraints	These represent inhomogenous and hanging node constraints that are used to condense the matrix.

fill_sparsity_pattern()

```
void HSIESurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * in_constraints )  [override], [virtual]
```

Fills a sparsity pattern for all the dofs active in this boundary condition.

Parameters

in_dsp	The sparsit pattern to fill
in_constraints	The constraint object to be used to condense

Implements BoundaryCondition.

Definition at line 251 of file HSIESurface.cpp.

```
251
      {
252   auto it = dof_h_nedelec.begin();
253   auto end = dof_h_nedelec.end();
254   auto it2 = dof_h_q.begin();
255   for (; it != end; ++it) {
256     DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
257     std::vector<unsigned int> local_indices;
258     for (unsigned int i = 0; i < cell_dofs.size(); i++) {
259       local_indices.push_back(cell_dofs[i].global_index);
260     }
261     local_indices = transform_local_to_global_dofs(local_indices);
262     in_constraints->add_entries_local_to_global(local_indices, *in_dsp);
263     it2++;
264   }
265 }
```

References FEDomain::transform_local_to_global_dofs().

finish_dof_index_initialization()

```
void HSIESurface::finish_dof_index_initialization ( )  [override], [virtual]
```

This is a DofDomain via BoundaryCondition.

This function signifies that global dof inidices have been exchanged.

Reimplemented from BoundaryCondition.

Definition at line 968 of file HSIESurface.cpp.

```
968                                               {
969   for(BoundaryId surf:adjacent_boundaries) {
970     if(!are_edge_dofs_owned[surf] && Geometry.levels[level].surface_type[surf] !=
       SurfaceType::NEIGHBOR_SURFACE) {
971       DofIndexVector dofs_in_global_numbering =
       Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id);
972       std::vector<InterfaceDofData> local_interface_data = get_dof_association_by_boundary_id(surf);
973       DofIndexVector dofs_in_local_numbering(local_interface_data.size());
974       for(unsigned int i = 0; i < local_interface_data.size(); i++) {
975         dofs_in_local_numbering[i] = local_interface_data[i].index;
976       }
977       set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
978     }
979   }
980 
981   // Do the same for the inner interface
982   std::vector<InterfaceDofData> global_interface_data =
       Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
983   std::vector<InterfaceDofData> local_interface_data = get_dof_association();
984   DofIndexVector dofs_in_local_numbering(local_interface_data.size());
985   DofIndexVector dofs_in_global_numbering(local_interface_data.size());
986 
987   for(unsigned int i = 0; i < local_interface_data.size(); i++) {
988     dofs_in_local_numbering[i] = local_interface_data[i].index;
989     dofs_in_global_numbering[i] =
       Geometry.levels[level].inner_domain->global_index_mapping[global_interface_data[i].index];
990   }
991   set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
992 
993 }
```

finish_initialization()

```
bool HSIESurface::finish_initialization (
            DofNumber first_own_index )  [override], [virtual]
```

Finishes the DofDomainInitialization.

For each dof that is locally owned, this function sets the global index. They have a local order and the global order and indices are the same, shifted by the number of the first dof. Lets see this domain has for dofs. Three are locally owned, Number 1,2 and 4 and 3 is not locally owned and already has the global index 55. If this function is called with the number 10, the global dof indices will be 10,11,55,12.

Parameters

first_own_index	

Returns

true if all indices now have an index

false some indices (non locally owned) dont have an index yet.

Reimplemented from FEDomain.

Definition at line 1005 of file HSIESurface.cpp.

```
1005                                                {
```

```
  std::vector<InterfaceDofData> dofs =
  Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
  std::vector<InterfaceDofData> own = get_dof_association();
  std::vector<unsigned int> local_indices, global_indices;
  for(unsigned int i = 0; i < dofs.size(); i++) {
    local_indices.push_back(own[i].index);
    global_indices.push_back(dofs[i].index);
  }
  set_non_local_dof_indices(local_indices, global_indices);
  return FEDomain::finish_initialization(index);
}
```

get_dof_association()

`std::vector< InterfaceDofData > HSIESurface::get_dof_association ( ) -> std::vector<InterfaceDofData> [override], [virtual]`

Get the dof association vector This is a part of the boundary condition interface and returns a list of all the dofs that couple to the inner domain.

This is used to prepare the exchange of dof indices and to check integrity (the length of this vector has to be the same as Innerdomain->get_dof_association(boundary id of this boundary)).

Returns

> std::vector<InterfaceDofData> All the dofs that couple to the interior sorted by z, then y then x.

Implements BoundaryCondition.

Definition at line 702 of file HSIESurface.cpp.

```
702                                                        {
703   std::sort(surface_dofs.begin(), surface_dofs.end(), compareDofBaseDataAndOrientation);
704   std::vector<InterfaceDofData> ret;
705   copy(surface_dofs.begin(), surface_dofs.end(), back_inserter(ret));
706   return ret;
707 }
```

get_dof_association_by_boundary_id()

`std::vector< InterfaceDofData > HSIESurface::get_dof_association_by_boundary_id ( BoundaryId in_boundary_id ) -> std::vector<InterfaceDofData> [override], [virtual]`

Get the dof association by boundary id If two neighboring surfaces have HSIE on them, this can be used to compute on each surface which dofs are at the outside surface they share and the resulting data can be used to build the coupling terms.

Parameters

in_boundary_id	the other boundary.

Returns

std::vector<InterfaceDofData>

Implements BoundaryCondition.

Definition at line 841 of file HSIESurface.cpp.

```
841                                                                                         {
842   if (are_opposing_sites(b_id, in_boundary_id)) {
843     return get_dof_association();
844   }
845
846   if (in_boundary_id == b_id) {
847     std::vector<InterfaceDofData> surface_dofs_unsorted(0);
848     std::cout « "This should never be called in HSIESurface" « std::endl;
849     return surface_dofs_unsorted;
850   }
851   std::vector<InterfaceDofData> surface_dofs_unsorted;
852   std::vector<unsigned int> vertex_ids = get_vertices_for_boundary_id(in_boundary_id);
853   std::vector<unsigned int> line_ids = get_lines_for_boundary_id(in_boundary_id);
854   std::vector<Position> vertex_positions = vertex_positions_for_ids(vertex_ids);
855   std::vector<Position> line_positions = line_positions_for_ids(line_ids);
856   for(unsigned int index = 0; index < vertex_dof_data.size(); index++) {
857     DofData dof = vertex_dof_data[index];
858     for(unsigned int index_in_ids = 0; index_in_ids < vertex_ids.size(); index_in_ids++) {
859       if(vertex_ids[index_in_ids] == vertex_dof_data[index].base_structure_id_non_face) {
860         InterfaceDofData new_item;
861         new_item.index = dof.global_index;
862         new_item.base_point = vertex_positions[index_in_ids];
863         new_item.order = (dof.inner_order+1) * (dof.nodal_basis + 1);
864         surface_dofs_unsorted.push_back(new_item);
865       }
866     }
867   }
868
869   // Construct containers with base points, orientation and index
870   for(unsigned int index = 0; index < edge_dof_data.size(); index++) {
871     DofData dof = edge_dof_data[index];
872     for(unsigned int index_in_ids = 0; index_in_ids < line_ids.size(); index_in_ids++) {
873       if(line_ids[index_in_ids] == edge_dof_data[index].base_structure_id_non_face) {
874         InterfaceDofData new_item;
875         new_item.index = dof.global_index;
876         new_item.base_point = line_positions[index_in_ids];
877         new_item.order = (dof.inner_order+1) * (dof.nodal_basis + 1);
878         surface_dofs_unsorted.push_back(new_item);
879       }
880     }
881   }
882
883   // Sort the vectors.
884   std::sort(surface_dofs_unsorted.begin(), surface_dofs_unsorted.end(),
       compareDofBaseDataAndOrientation);
885
886   return surface_dofs_unsorted;
887 }
```

get_dof_data_for_base_dof_nedelec()

```
DofDataVector HSIESurface::get_dof_data_for_base_dof_nedelec (
              DofNumber base_dof_index ) -> DofDataVector
```

Get the dof data for a nedelec base dof.

All dofs on this surface are either built based on a nedelec surface dof or a q dof on the surface. For a given index from the nedelec fe this provides all dofs that are based on it.

Parameters

base_dof_index	Index of the nedelec dof for whom we search all the dofs that depend on it.

Returns

All the dofs that depend on nedelec dof number base_dof_index.

Definition at line 83 of file HSIESurface.cpp.

```
83                                                                             {
84    DofDataVector ret;
85    for (unsigned int index = 0; index < edge_dof_data.size(); index++) {
86      if ((edge_dof_data[index].base_dof_index == in_index)
87          && (edge_dof_data[index].type != DofType::RAY
88              && edge_dof_data[index].type != DofType::IFFb)) {
89        ret.push_back(edge_dof_data[index]);
90      }
91    }
92    for (unsigned int index = 0; index < vertex_dof_data.size(); index++) {
93      if ((vertex_dof_data[index].base_dof_index == in_index)
94          && (vertex_dof_data[index].type != DofType::RAY
95              && vertex_dof_data[index].type != DofType::IFFb)) {
96        ret.push_back(vertex_dof_data[index]);
97      }
98    }
99    for (unsigned int index = 0; index < face_dof_data.size(); index++) {
100     if ((face_dof_data[index].base_dof_index == in_index)
101         && (face_dof_data[index].type != DofType::RAY
102             && face_dof_data[index].type != DofType::IFFb)) {
103       ret.push_back(face_dof_data[index]);
104     }
105   }
106   return ret;
107 }
```

get_dof_data_for_base_dof_q()

```
DofDataVector HSIESurface::get_dof_data_for_base_dof_q (
              DofNumber base_dof_index ) -> DofDataVector
```

Get the dof data for base dof q.

Same as above but for q dofs.

Parameters

base_dof_index	See above.

Returns

see above.

Definition at line 109 of file HSIESurface.cpp.

```
109                                                                       {
110   DofDataVector ret;
111   for (unsigned int index = 0; index < edge_dof_data.size(); index++) {
112     if ((edge_dof_data[index].base_dof_index == in_index)
113         && (edge_dof_data[index].type == DofType::RAY
114             || edge_dof_data[index].type == DofType::IFFb)) {
115       ret.push_back(edge_dof_data[index]);
116     }
```

```
  }
  for (unsigned int index = 0; index < vertex_dof_data.size(); index++) {
    if ((vertex_dof_data[index].base_dof_index == in_index)
        && (vertex_dof_data[index].type == DofType::RAY
            || vertex_dof_data[index].type == DofType::IFFb)) {
      ret.push_back(vertex_dof_data[index]);
    }
  }
  for (unsigned int index = 0; index < face_dof_data.size(); index++) {
    if ((face_dof_data[index].base_dof_index == in_index)
        && (face_dof_data[index].type == DofType::RAY
            || face_dof_data[index].type == DofType::IFFb)) {
      ret.push_back(face_dof_data[index]);
    }
  }
  return ret;
}
```

get_lines_for_boundary_id()

```
std::vector< unsigned int > HSIESurface::get_lines_for_boundary_id (
              BoundaryId in_bid ) -> std::vector<unsigned int>
```

Get the lines shared with the boundary in_bid.

Parameters

in_bid	BoundaryID of the other boundary.

Returns

std::vector of the line ids on the boundary

Definition at line 944 of file HSIESurface.cpp.

```
944                                                                         {
945   std::vector<unsigned int> edges;
946   for(auto it = Geometry.surface_meshes[b_id].begin_active_face(); it !=
       Geometry.surface_meshes[b_id].end_face(); it++) {
947     if(is_point_at_boundary(it->center(), in_boundary_id)) {
948       edges.push_back(it->index());
949     }
950   }
951   edges.shrink_to_fit();
952   return edges;
953 }
```

get_n_lines_for_boundary_id()

```
auto HSIESurface::get_n_lines_for_boundary_id (
              BoundaryId in_bid ) -> unsigned int
```

Get the number of lines for boundary id object.

Parameters

in_bid	The other boundary.

Returns

unsigned int Count of lines on the edge shared with the other boundary

get_n_vertices_for_boundary_id()

```
auto HSIESurface::get_n_vertices_for_boundary_id (
            BoundaryId in_bid ) -> unsigned int
```

Get the number of vertices on th eboundary with id.

Parameters

in_bid	The boundary id of the other boundary

Returns

Number of dofs on the boundary

get_vertices_for_boundary_id()

```
std::vector< unsigned int > HSIESurface::get_vertices_for_boundary_id (
            BoundaryId in_bid ) -> std::vector<unsigned int>
```

Get the vertices located at the provided boundary.

Returns

std::vector<unsigned int> Indices of the vertices at the boundary

Definition at line 933 of file HSIESurface.cpp.

```
933                                                                      {
934   std::vector<unsigned int> vertices;
935   for(auto it = Geometry.surface_meshes[b_id].begin_vertex(); it !=
       Geometry.surface_meshes[b_id].end_vertex(); it++) {
936     if(is_point_at_boundary(it->center(), in_boundary_id)) {
937       vertices.push_back(it->index());
938     }
939   }
940   vertices.shrink_to_fit();
941   return vertices;
942 }
```

initialize_dof_handlers_and_fe()

```
void HSIESurface::initialize_dof_handlers_and_fe ( )
```

Part of the initialization function.

Prepares the dof handlers of q and nedelec type.

Definition at line 370 of file HSIESurface.cpp.

```
                                              {
  dof_h_q.distribute_dofs(fe_q);
  dof_h_nedelec.distribute_dofs(fe_nedelec);
}
```

Referenced by initialize().

is_point_at_boundary()

```
bool HSIESurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid )  [override], [virtual]
```

Checks if a point is at an outward surface of the boundary triangulation.

Parameters

in_p	The position to check
in_bid	The boundary id of the other surface

Returns

true if the point is located at the edge between this surface and the surface in_bid.

false if not

Implements BoundaryCondition.

Definition at line 923 of file HSIESurface.cpp.

```
923                                                                          {
924   if(!boundary_coordinates_computed) {
925     compute_extreme_vertex_coordinates();
926   }
927   if(are_opposing_sites(in_bid, b_id) || in_bid == b_id) return true;
928   Position full_position = undo_transform(in_p);
929   unsigned int component = in_bid / 2;
930   return full_position[component] == boundary_vertex_coordinates[in_bid];
931 }
```

References compute_extreme_vertex_coordinates().

line_positions_for_ids()

```
std::vector< Position > HSIESurface::line_positions_for_ids (
            std::vector< unsigned int > ids ) -> std::vector<Position>
```

Computes the positions for line ids.

Parameters

ids	The list of ids.

Returns

std::vector<Position> with the positions in same order

Definition at line 832 of file HSIESurface.cpp.

```
832                                                                                    {
833   std::vector<Position> ret(ids.size());
834   for(unsigned int line_index_in_array = 0; line_index_in_array < ids.size(); line_index_in_array++) {
835     Position p  =
       undo_transform(get_line_position_for_line_index_in_tria(&Geometry.surface_meshes[b_id],
       ids[line_index_in_array]));
836     ret[line_index_in_array] = p;
837   }
838   return ret;
839 }
```

References undo_transform().

output_results()

```
std::string HSIESurface::output_results (
            const dealii::Vector< ComplexNumber > & ,
            std::string  )  [override], [virtual]
```

Does nothing.

Fulfills the interface.

Returns

std::string filename

Implements BoundaryCondition.

Definition at line 955 of file HSIESurface.cpp.

```
955                                                                                      {
956   return "";
957 }
```

register_dof()

```
unsigned int HSIESurface::register_dof ( ) -> DofNumber
```

Increments the dof counter.

Returns

DofNumber returns the dof counter after the increment.

Definition at line 533 of file HSIESurface.cpp.

```
533                                         {
534   dof_counter++;
535   return dof_counter - 1;
536 }
```

Referenced by register_single_dof().

register_new_edge_dofs()

```
void HSIESurface::register_new_edge_dofs (
            CellIterator2D cell,
            CellIterator2D cell_2,
            unsigned int edge )
```

When building the datastructures, this function adds a new dof to the list of all edge dofs.

Parameters

cell	The cell the dof was found in, in the nedelec dof handler
cell_2	The cell the dof was found in, in the q dof handler
edge	The index of the edge it belongs to.

Definition at line 413 of file HSIESurface.cpp.

```
413
          {
414   const int max_hsie_order = order;
415   // EDGE Dofs
416   std::vector<unsigned int> local_dofs(fe_nedelec.dofs_per_line);
417   cell_nedelec->line(edge)->get_dof_indices(local_dofs);
418   bool orientation = false;
419   if(cell_nedelec->line(edge)->vertex_index(0) > cell_nedelec->line(edge)->vertex_index(1)) {
420     orientation = get_orientation(undo_transform(cell_nedelec->line(edge)->vertex(0)),
       undo_transform(cell_nedelec->line(edge)->vertex(1)));
421   } else {
422     orientation = get_orientation(undo_transform(cell_nedelec->line(edge)->vertex(1)),
       undo_transform(cell_nedelec->line(edge)->vertex(0)));
423   }
424
425   for (int inner_order = 0; inner_order < static_cast<int>(fe_nedelec.dofs_per_line); inner_order++) {
426     register_single_dof(cell_nedelec->face_index(edge), -1, inner_order + 1, DofType::EDGE,
       edge_dof_data, local_dofs[inner_order], orientation);
427     Position bp = undo_transform(cell_nedelec->face(edge)->center(false, false));
428     InterfaceDofData dof_data;
429     dof_data.index = edge_dof_data[edge_dof_data.size() - 1].global_index;
430     dof_data.order = inner_order;
431     dof_data.base_point = bp;
432     add_surface_relevant_dof(dof_data);
433   }
434
435   // INFINITE FACE Dofs Type a
436   for (int inner_order = 0; inner_order < static_cast<int>(fe_nedelec.dofs_per_line); inner_order++) {
437     for (int hsie_order = 0; hsie_order <= max_hsie_order; hsie_order++) {
438       register_single_dof(cell_nedelec->face_index(edge), hsie_order, inner_order + 1, DofType::IFFa,
       edge_dof_data, local_dofs[inner_order], orientation);
439     }
440   }
441   // INFINITE FACE Dofs Type b
442   local_dofs.clear();
443   local_dofs.resize(fe_q.dofs_per_line + 2 * fe_q.dofs_per_vertex);
444   cell_q->line(edge)->get_dof_indices(local_dofs);
445   IndexSet line_dofs(MAX_DOF_NUMBER);
446   IndexSet non_line_dofs(MAX_DOF_NUMBER);
447   for (unsigned int i = 0; i < local_dofs.size(); i++) {
448     line_dofs.add_index(local_dofs[i]);
449   }
450   for (unsigned int i = 0; i < fe_q.dofs_per_vertex; i++) {
451     non_line_dofs.add_index(cell_q->line(edge)->vertex_dof_index(0, i));
452     non_line_dofs.add_index(cell_q->line(edge)->vertex_dof_index(1, i));
453   }
454   line_dofs.subtract_set(non_line_dofs);
455   for (int inner_order = 0; inner_order < static_cast<int>(line_dofs.n_elements());
456        inner_order++) {
457     for (int hsie_order = -1; hsie_order <= max_hsie_order; hsie_order++) {
```

```
458       register_single_dof(cell_q->face_index(edge), hsie_order, inner_order, DofType::IFFb,
      edge_dof_data, line_dofs.nth_index_in_set(inner_order), orientation);
459     }
460   }
461 }
```

register_new_surface_dofs()

```
void HSIESurface::register_new_surface_dofs (
            CellIterator2D cell,
            CellIterator2D cell2 )
```

When building the datastructures, this function adds a new dof to the list of all face dofs.

Cells here are faces because the surface triangulation is 2D.

Parameters

cell	The cell the dof was found in, in the nedelec dof handler
cell_2	The cell the dof was found in, in the q dof handler
edge	The index of the edge it belongs to.

Definition at line 463 of file HSIESurface.cpp.

```
463                                                                                              {
464   const int max_hsie_order = order;
465   std::vector<unsigned int> surface_dofs(fe_nedelec.dofs_per_cell);
466   cell_nedelec->get_dof_indices(surface_dofs);
467   IndexSet surf_dofs(MAX_DOF_NUMBER);
468   IndexSet edge_dofs(MAX_DOF_NUMBER);
469   for (unsigned int i = 0; i < surface_dofs.size(); i++) {
470     surf_dofs.add_index(surface_dofs[i]);
471   }
472   for (unsigned int i = 0; i < dealii::GeometryInfo<2>::lines_per_cell; i++) {
473     std::vector<unsigned int> line_dofs(fe_nedelec.dofs_per_line);
474     cell_nedelec->line(i)->get_dof_indices(line_dofs);
475     for (unsigned int j = 0; j < line_dofs.size(); j++) {
476       edge_dofs.add_index(line_dofs[j]);
477     }
478   }
479   surf_dofs.subtract_set(edge_dofs);
480   std::string id = cell_q->id().to_string();
481   const unsigned int nedelec_dof_count = dof_h_nedelec.n_dofs();
482   dealii::Vector<ComplexNumber> vec_temp(nedelec_dof_count);
483   // SURFACE functions
484   for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
485     register_single_dof(cell_nedelec->id().to_string(), -1, inner_order, DofType::SURFACE,
      face_dof_data, surf_dofs.nth_index_in_set(inner_order));
486     Position bp = undo_transform(cell_nedelec->center());
487     InterfaceDofData dof_data;
488     dof_data.index = face_dof_data[face_dof_data.size() - 1].global_index;
489     dof_data.base_point = bp;
490     dof_data.order = inner_order;
491     add_surface_relevant_dof(dof_data);
492   }
493
494   // SEGMENT functions a
495   for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
496     for (int hsie_order = 0; hsie_order <= max_hsie_order; hsie_order++) {
497       register_single_dof(id, hsie_order, inner_order, DofType::SEGMENTa, face_dof_data,
      surf_dofs.nth_index_in_set(inner_order));
498     }
499   }
```

```

  for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
    for (int hsie_order = -1; hsie_order <= max_hsie_order; hsie_order++) {
      register_single_dof(id, hsie_order, inner_order, DofType::SEGMENTb, face_dof_data,
 surf_dofs.nth_index_in_set(inner_order));
    }
  }
}
```

References register_single_dof().

Referenced by compute_n_face_dofs().

register_new_vertex_dofs()

```
void HSIESurface::register_new_vertex_dofs (
            CellIterator2D cell,
            unsigned int edge,
            unsigned int vertex )
```

When building the datastructures, this function adds a new dof to the list of all vertex dofs.

This is always a HSIE dof that relates to an infinite edge and therefore only needs the q type dof_handler in the surface fem.

Parameters

cell	The cell the dof was found in.
edge	The index of the edge it belongs to.
vertex	The index of the vertex in the edge that the dof belongs to.

Definition at line 404 of file HSIESurface.cpp.

```
                           {
  const int max_hsie_order = order;
  for (int hsie_order = -1; hsie_order <= max_hsie_order; hsie_order++) {
    register_single_dof(cell->vertex_index(vertex), hsie_order, -1, DofType::RAY, vertex_dof_data,
 dof_index);
  }
}
```

References register_single_dof().

Referenced by compute_n_vertex_dofs().

register_single_dof() [1/2]

```
void HSIESurface::register_single_dof (
            std::string in_id,
            int in_hsie_order,
            int in_inner_order,
            DofType in_dof_type,
            DofDataVector & in_vector,
            unsigned int base_dof_index )
```

Registers a new dof with a face base structure (first argument is string)

There are several lists of the dofs that this object handles. This functions adds a single dof to those lists so it can be iterated over where necessary.

Parameters

in_id	The id of the base structures. For cells these have the type string.
in_hsie_order	Order of the hardy space polynomial.
in_inner_order	Order of the nedelec element of the dof.
in_dof_type	There are several different types of dofs. See page 13 in the publication.
base_dof_index	Index if the base dof. For example, an infinite surface dof is a combination of a hardy polynomial in the infinite direction and a surface nedelec edge dof. This number is the dof index of the nedelec edge dof.

Definition at line 508 of file HSIESurface.cpp.

```
{
   DofData dd(in_id);
   dd.global_index = register_dof();
   dd.hsie_order = in_hsie_order;
   dd.inner_order = in_inner_order;
   dd.type = in_dof_type;
   dd.set_base_dof(in_base_dof_index);
   dd.update_nodal_basis_flag();
   in_vector.push_back(dd);
}
```

References register_dof().

Referenced by register_new_surface_dofs(), and register_new_vertex_dofs().

register_single_dof() `[2/2]`

```
void HSIESurface::register_single_dof (
        unsigned int in_id,
        int in_hsie_order,
        int in_inner_order,
        DofType in_dof_type,
        DofDataVector & in_vector,
        unsigned int in_base_dof_index,
        bool orientation = true )
```

Registers a new dof with a edge or vertex base structure (first argument is int)

There are several lists of the dofs that this object handles. This functions adds a single dof to those lists so it can be iterated over where necessary.

Parameters

in_id	The id of the base structures.
in_hsie_order	Order of the hardy space polynomial.
in_inner_order	Order of the nedelec element of the dof.
in_dof_type	There are several different types of dofs. See page 13 in the publication.

Parameters

base_dof_index	Index if the base dof. For example, an infinite surface dof is a combination of a hardy polynomial in the infinite direction and a surface nedelec edge dof. This number is the dof index of the nedelec edge dof.

Definition at line 520 of file HSIESurface.cpp.

```
{
  DofData dd(in_id);
  dd.global_index = register_dof();
  dd.hsie_order = in_hsie_order;
  dd.inner_order = in_inner_order;
  dd.type = in_dof_type;
  dd.orientation = orientation;
  dd.set_base_dof(in_base_dof_index);
  dd.update_nodal_basis_flag();
  in_vector.push_back(dd);
}
```

References register_dof().

set_b_id_uses_hsie()

```
void HSIESurface::set_b_id_uses_hsie (
            unsigned int index,
            bool does )
```

It is usefull to know, if a neighboring surface is also using hsie.

Updates the local cache with the information that the neighboring boundary index uses hsie or does not

Parameters

int	index
does	if this is true, the neighbor uses hsie, if not, then not.

transform_coordinates_in_place()

```
void HSIESurface::transform_coordinates_in_place (
            std::vector< HSIEPolynomial > * in_vector )
```

All functions for this type assume that x is the infinte direction.

This transforms x to the actual infinite direction.

Parameters

in_vector	vector of length 3 that defines a field. This will be transformed to the actual coordinate system.

Definition at line 621 of file HSIESurface.cpp.

```
621 {
622   // The ray direction before transformation is x. This has to be adapted.
623   HSIEPolynomial temp = (*vector)[0];
624   switch (b_id) {
625     case 2:
626       (*vector)[0] = (*vector)[1];
627       (*vector)[1] = temp;
628       break;
629     case 3:
630       (*vector)[0] = (*vector)[1];
631       (*vector)[1] = temp;
632       break;
633     case 4:
634       (*vector)[0] = (*vector)[2];
635       (*vector)[2] = temp;
636       break;
637     case 5:
638       (*vector)[0] = (*vector)[2];
639       (*vector)[2] = temp;
640       break;
641   }
642 }
```

Referenced by build_curl_term_nedelec(), build_curl_term_q(), and build_non_curl_term_q().

undo_transform()

```
Position HSIESurface::undo_transform (
            dealii::Point< 2 > inp ) -> Position
```

Returns the 3D form of a point for a provided 2D position in the surface triangulation.

Returns

Position in 3D

Definition at line 644 of file HSIESurface.cpp.

```
644 {
645   Position ret;
646   ret[0] = inp[0];
647   ret[1] = inp[1];
648   ret[2] = additional_coordinate;
649   switch (b_id) {
650   case 0:
651     ret = Transform_5_to_0(ret);
652     break;
653   case 1:
654     ret = Transform_5_to_1(ret);
655     break;
656   case 2:
657     ret = Transform_5_to_2(ret);
658     break;
659   case 3:
660     ret = Transform_5_to_3(ret);
661     break;
662   case 4:
663     ret = Transform_5_to_4(ret);
664     break;
665   default:
666     break;
667   }
668   return ret;
669 }
```

Referenced by line_positions_for_ids(), and vertex_positions_for_ids().

undo_transform_for_shape_function()

```
Position HSIESurface::undo_transform_for_shape_function (
              dealii::Point< 2 > inp ) -> Position
```

Transforms the 2D value of a surface dof shape function into a 3D field in the actual 3D coordinates.

The input of this function has 2 components for the two dimensions of the surface triangulation. This gets transformed into the global 3D coordinate system

Returns

Position value of the shape function interpreted in 3D.

Definition at line 671 of file HSIESurface.cpp.

```
671                                                                          {
672   Position ret;
673   ret[0] = inp[0];
674   ret[1] = inp[1];
675   ret[2] = 0;
676   switch (b_id) {
677   case 0:
678     ret = Transform_5_to_0(ret);
679     break;
680   case 1:
681     ret = Transform_5_to_1(ret);
682     break;
683   case 2:
684     ret = Transform_5_to_2(ret);
685     break;
686   case 3:
687     ret = Transform_5_to_3(ret);
688     break;
689   case 4:
690     ret = Transform_5_to_4(ret);
691     break;
692   default:
693     break;
694   }
695   return ret;
696 }
```

update_dof_counts_for_edge()

```
void HSIESurface::update_dof_counts_for_edge (
              CellIterator2D cell,
              unsigned int edge,
              DofCountsStruct & in_dof_counts )
```

Updates the numbers of dofs for an edge.

Parameters

cell	Cell we are operating on
edge	index of the edge in the cell
in_dof_counts	Dof counts to be updated

Definition at line 375 of file HSIESurface.cpp.

```
{
const unsigned int dofs_per_edge_all = compute_dofs_per_edge(false);
const unsigned int dofs_per_edge_hsie = compute_dofs_per_edge(true);
in_dof_count.total += dofs_per_edge_all;
in_dof_count.hsie += dofs_per_edge_hsie;
in_dof_count.non_hsie += dofs_per_edge_all - dofs_per_edge_hsie;
}
```

References compute_dofs_per_edge().

update_dof_counts_for_face()

```
void HSIESurface::update_dof_counts_for_face (
            CellIterator2D cell,
            DofCountsStruct & in_dof_counts )
```

Updates the numbers of dofs for a face.

Parameters

cell	Cell we are operating on
in_dof_counts	Dof counts to be updated

Definition at line 385 of file HSIESurface.cpp.

```
{
const unsigned int dofs_per_face_all = compute_dofs_per_face(false);
const unsigned int dofs_per_face_hsie = compute_dofs_per_face(true);
in_dof_count.total += dofs_per_face_all;
in_dof_count.hsie += dofs_per_face_hsie;
in_dof_count.non_hsie += dofs_per_face_all - dofs_per_face_hsie;
}
```

References compute_dofs_per_face().

Referenced by compute_n_face_dofs().

update_dof_counts_for_vertex()

```
void HSIESurface::update_dof_counts_for_vertex (
            CellIterator2D cell,
            unsigned int edge,
            unsigned int vertex,
            DofCountsStruct & in_dof_coutns )
```

Updates the dof counts for a vertex.

Parameters

cell	Cell we are operating on.
edge	Index of the edge in the cell.
vertex	Index of the vertex in the edge.
in_dof_coutns	Dof counts to be updated

Definition at line 395 of file HSIESurface.cpp.

```
397                                             {
398   const unsigned int dofs_per_vertex_all = compute_dofs_per_vertex();
399
400   in_dof_count.total += dofs_per_vertex_all;
401   in_dof_count.hsie += dofs_per_vertex_all;
402 }
```

References compute_dofs_per_vertex().

Referenced by compute_n_vertex_dofs().

vertex_positions_for_ids()

```
std::vector< Position > HSIESurface::vertex_positions_for_ids (
               std::vector< unsigned int > ids ) -> std::vector<Position>
```

Computes all vertex positions for a set of vertex ids.

Parameters

ids	The list of ids.

Returns

std::vector<Position> with the positions in same order

Definition at line 823 of file HSIESurface.cpp.

```
823                                                                           {
824   std::vector<Position> ret(ids.size());
825   for(unsigned int vertex_index_in_array = 0; vertex_index_in_array < ids.size();
       vertex_index_in_array++) {
826     Position p =
       undo_transform(get_vertex_position_for_vertex_index_in_tria(&Geometry.surface_meshes[b_id],
       ids[vertex_index_in_array]));
827     ret[vertex_index_in_array] = p;
828   }
829   return ret;
830 }
```

References undo_transform().

The documentation for this class was generated from the following files:

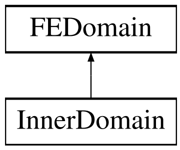
- Code/BoundaryCondition/HSIESurface.h
- Code/BoundaryCondition/HSIESurface.cpp

37 InnerDomain Class Reference

This class encapsulates all important mechanism for solving a FEM problem. In earlier versions this also included space transformation and computation of materials. Now it only includes FEM essentials and solving the system matrix.

`#include <InnerDomain.h>`

Inheritance diagram for InnerDomain:



Public Member Functions

- **InnerDomain** (unsigned int level)
- void load_exact_solution ()

  In many places it can be useful to have an interpolated exact solution for the waveguide or Hertz case.
- void make_grid ()

  This function builds the triangulation for the inner domain part on this level that is locally owned.
- void assemble_system (Constraints ∗constraints, dealii::PETScWrappers::MPI::SparseMatrix ∗matrix, NumericVectorDistributed ∗rhs)

  Main part of the system matrix assembly loop.
- std::vector< InterfaceDofData > get_surface_dof_vector_for_boundary_id (BoundaryId b_id)

  Returns a vector of all dofs active on the given surface.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_pattern, Constraints ∗constraints)

  Marks all index pairs that are non-zero in the provided matrix using the given constraints.
- void write_matrix_and_rhs_metrics (dealii::PETScWrappers::MatrixBase ∗matrix, NumericVectorDistributed ∗rhs)

  Prints some diagnostic data to the console.
- std::string output_results (std::string in_filename, NumericVectorLocal in_solution, bool apply_space_transformation)

  Generates an output file of the provided solution vector on the local domain.
- DofCount compute_n_locally_owned_dofs () override

  Fulfills FEDomain interface.
- DofCount compute_n_locally_active_dofs () override

  Fulfills FEDomain interface.
- void determine_non_owned_dofs () override

  Fulfills FEDomain interface.
- ComplexNumber compute_signal_strength (dealii::LinearAlgebra::distributed::Vector< ComplexNumber > ∗in_solution)

  Computes how strongly the fundamental mode is excited in the output waveguide in the field provided as the input.
- ComplexNumber compute_mode_strength ()

  Computes the norm of the input mode for scaling of the output signal.
- FEErrorStruct compute_errors (dealii::LinearAlgebra::distributed::Vector< ComplexNumber > ∗in_solution)

  Computes the L2 and L_infty error of the provided solution against the source field (i.e.

- std::vector< std::vector< ComplexNumber > > evaluate_at_positions (std::vector< Position > in_positions, NumericVectorLocal in_solution)

  Evaluates the provided solution (represented by in_solution) at the given positions, i.e.

- std::vector< FEAdjointEvaluation > compute_local_shape_gradient_data (NumericVectorLocal &in_solution, NumericVectorLocal &in_adjoint)

  Computes point data required to compute the shape gradient.

- Tensor< 1, 3, ComplexNumber > evaluate_J_at (Position in_p)

  Computes the forcing term J for a given position so we can use it to build a right-hand side / forcing term.

- ComplexNumber compute_kappa (NumericVectorLocal &in_solution)

  Computes the value κ.

- void **set_rhs_for_adjoint_problem** (NumericVectorLocal &in_solution, NumericVectorDistributed *in_rhs)

Public Attributes

- SquareMeshGenerator **mesh_generator**
- dealii::FE_NedelecSZ< 3 > **fe**
- dealii::Triangulation< 3 > **triangulation**
- DofHandler3D **dof_handler**
- dealii::SparsityPattern **sp**
- dealii::DataOut< 3 > **data_out**
- bool **exact_solution_is_initialized**
- NumericVectorLocal **exact_solution_interpolated**
- unsigned int **level**

37.1 Detailed Description

This class encapsulates all important mechanism for solving a FEM problem. In earlier versions this also included space transformation and computation of materials. Now it only includes FEM essentials and solving the system matrix.

Upon initialization it requires structural information about the waveguide that will be simulated. The object then continues to initialize the FEM-framework. After allocating space for all objects, the assembly-process of the system-matrix begins. Following this step, the user-selected preconditioner and solver are used to solve the system and generate outputs. This class is the core piece of the implementation.

Definition at line 88 of file InnerDomain.h.

37.2 Member Function Documentation

assemble_system()

```
void InnerDomain::assemble_system (
            Constraints * constraints,
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs )
```

Main part of the system matrix assembly loop.

Writes all contributions of the local domain to the system matrix provided as a pointer.

Parameters

constraints	All constraints on degrees of freedom.
matrix	The system matrix to be filled.
rhs	The right-hand side vector to be used.

Definition at line 297 of file InnerDomain.cpp.

```
{
  CellwiseAssemblyDataNP cell_data(&fe, &dof_handler);
  load_exact_solution();
  ExactSolution * esp;
  if(GlobalParams.Index_in_z_direction == 0 && GlobalParams.Signal_coupling_method == SignalCouplingMethod::Tapering) {
     esp = new ExactSolution();
     cell_data.set_es_pointer(esp);
  }
  for (; cell_data.cell != cell_data.end_cell; ++cell_data.cell) {
    cell_data.cell->get_dof_indices(cell_data.local_dof_indices);
    cell_data.local_dof_indices = transform_local_to_global_dofs(cell_data.local_dof_indices);
    cell_data.cell_matrix = 0;
    cell_data.cell_rhs.reinit(cell_data.dofs_per_cell);
    cell_data.cell_rhs = 0;
    cell_data.fe_values.reinit(cell_data.cell);
    cell_data.quadrature_points = cell_data.fe_values.get_quadrature_points();
    for (unsigned int q_index = 0; q_index < cell_data.n_q_points; ++q_index) {
      cell_data.prepare_for_current_q_index(q_index);
    }
    bool is_skeq_sym = true;
    for(unsigned int i = 0; i < cell_data.cell_matrix.n_rows(); i++) {
      for(unsigned int j = 0; j < i; j++) {
        if(!(std::abs(cell_data.cell_matrix[i][j] - conjugate(cell_data.cell_matrix[j][i])) < FLOATING_PRECISION)) {
          is_skeq_sym = false;
        }
      }
    }
    if(!is_skeq_sym) std::cout « "Not fulfilled!" « std::endl;
    constraints->distribute_local_to_global(cell_data.cell_matrix, cell_data.cell_rhs, cell_data.local_dof_indices,*matrix, *rhs, true);
  }
  matrix->compress(dealii::VectorOperation::add);
  rhs->compress(dealii::VectorOperation::add);
  if(GlobalParams.Index_in_z_direction == 0 && GlobalParams.Signal_coupling_method == SignalCouplingMethod::Tapering) {
    delete esp;
  }
}
```

References load_exact_solution().

compute_errors()

```
FEErrorStruct InnerDomain::compute_errors (
            dealii::LinearAlgebra::distributed::Vector< ComplexNumber > * in_solution )
```

Computes the L2 and L_infty error of the provided solution against the source field (i.e. exact solution if applicable).

Parameters

in_solution	The FE solution we want to compute the errors for.

Returns

FEErrorStruct A struct containing L2 and L_infty members.

Definition at line 526 of file InnerDomain.cpp.

```
526
        {
527   FEErrorStruct ret;
528   dealii::Vector<double> cell_vector (triangulation.n_active_cells());
529   QGauss<3> q(GlobalParams.Nedelec_element_order + 2);
530   NumericVectorLocal local_solution(n_locally_active_dofs);
531   for(unsigned int i =0 ; i < n_locally_active_dofs; i++) {
532     local_solution[i] = in_solution->operator[](global_index_mapping[i]);
533   }
534   VectorTools::integrate_difference(dof_handler, local_solution, *GlobalParams.source_field,
       cell_vector, q, dealii::VectorTools::NormType::L2_norm);
535   ret.L2 = VectorTools::compute_global_error(triangulation, cell_vector,
       dealii::VectorTools::NormType::L2_norm);
536   ret.L2 /= in_solution->l2_norm();
537   VectorTools::integrate_difference(dof_handler, local_solution, *GlobalParams.source_field,
       cell_vector, q, dealii::VectorTools::NormType::Linfty_norm);
538   ret.Linfty = VectorTools::compute_global_error(triangulation, cell_vector,
       dealii::VectorTools::NormType::Linfty_norm);
539   ret.Linfty /= in_solution->linfty_norm();
540   return ret;
541 }
```

compute_kappa()

```
ComplexNumber InnerDomain::compute_kappa (
            NumericVectorLocal & in_solution )
```

Computes the value κ.

This value is defined by

$$\kappa = \int_{\Gamma_O} \overline{\boldsymbol{E}_0} \cdot \boldsymbol{E}_p \mathrm{d}A$$

Parameters

in_solution	

Returns

ComplexNumber

Definition at line 594 of file InnerDomain.cpp.

```
594                                                                           {
595   ComplexNumber ret;
596   QGauss<2> quadrature_formula(1);
597   const FEValuesExtractors::Vector fe_field(0);
598   FEFaceValues<3> fe_values(fe, quadrature_formula, update_values | update_JxW_values |
       update_quadrature_points);
599   std::vector<unsigned int> local_dof_indices(fe.n_dofs_per_cell());
600   for (DofHandler3D::active_cell_iterator cell = dof_handler.begin_active(); cell != dof_handler.end();
       ++cell) {
601     for(unsigned int face = 0; face < 6; face++) {
602       if(std::abs(cell->face(face)->center()[2] - Geometry.global_z_range.second) < FLOATING_PRECISION)
       {
603         fe_values.reinit(cell,face);
604         double JxW;
605         auto q_points = fe_values.get_quadrature_points();
606         for(unsigned int q_index = 0; q_index < quadrature_formula.size(); q_index++) {
607           cell->get_dof_indices(local_dof_indices);
608           JxW = fe_values.get_JxW_values()[q_index];
609           Position p = q_points[q_index];
610           Tensor<1,3, ComplexNumber> E0;
611           for(unsigned int i = 0; i < 3; i++) {
612             E0[i] = GlobalParams.source_field->value(p,i);
613           }
614           for(unsigned int i = 0; i < fe.n_dofs_per_cell(); i++) {
615             // std::cout « in_solution[local_dof_indices[i]] « " and " «JxW « " and " « E0.norm() « "
       and " « fe_values[fe_field].value(i, q_index).norm() « std::endl;
616             for(unsigned int j = 0; j < 3; j++) {
617               ret += conjugate((in_solution[local_dof_indices[i]] * fe_values[fe_field].value(i,
       q_index))[j]) * E0[j] * JxW;
618             }
619           }
620         }
621       }
622     }
623   }
624   return Utilities::MPI::sum(ret, MPI_COMM_WORLD);
625 }
```

compute_local_shape_gradient_data()

```
std::vector< FEAdjointEvaluation > InnerDomain::compute_local_shape_gradient_data (
            NumericVectorLocal & in_solution,
            NumericVectorLocal & in_adjoint )
```

Computes point data required to compute the shape gradient.

To compute the shape gradient, we require at every quadrature point of the evaluation quadrature:

- The primal solution
- The curl of the primal solution
- The adjoint solution
- The curl of the adjoint solution
- The location that these values were computed at. This function computes all these values and stores them in an array. Every entry is the data for one quadrature point.

Parameters

in_solution	The solution vector from the finite element method applied to the primal problem.
in_adjoint	The solution vector of the finite element method applied to the adjoint problem.

Returns

std::vector<FEAdjointEvaluation> Vector of datasets for a quadrature of the local domain with field evaluations and curls.

Definition at line 558 of file InnerDomain.cpp.

```
558 {
559   std::vector<FEAdjointEvaluation> ret;
560   QGauss<3> quadrature_formula(1);
561   const FEValuesExtractors::Vector fe_field(0);
562   FEValues<3> fe_values(fe, quadrature_formula, update_values | update_gradients |
       update_quadrature_points);
563   std::vector<unsigned int> local_dof_indices(fe.n_dofs_per_cell());
564   for (DofHandler3D::active_cell_iterator cell = dof_handler.begin_active(); cell != dof_handler.end();
       ++cell) {
565     fe_values.reinit(cell);
566     auto q_points = fe_values.get_quadrature_points();
567     for(unsigned int q_index = 0; q_index < quadrature_formula.size(); q_index++) {
568       cell->get_dof_indices(local_dof_indices);
569       Position p = q_points[q_index];
570       FEAdjointEvaluation item;
571       item.x = p;
572       for(unsigned int i = 0; i < 3; i++) {
573         item.primal_field[i] = 0;
574         item.adjoint_field[i] = 0;
575         item.primal_field_curl[i] = 0;
576         item.adjoint_field_curl[i] = 0;
577       }
578       for(unsigned int i = 0; i < fe.n_dofs_per_cell(); i++) {
579         Tensor<1, 3, ComplexNumber> I_Val;
580         I_Val = fe_values[fe_field].value(i, q_index);
581         Tensor<1, 3, ComplexNumber> I_Curl;
582         I_Curl = fe_values[fe_field].curl(i, q_index);
583         item.primal_field += I_Val * in_solution[local_dof_indices[i]];
584         item.adjoint_field += I_Val * in_adjoint[local_dof_indices[i]];
585         item.primal_field_curl += I_Curl * in_solution[local_dof_indices[i]];
586         item.adjoint_field_curl += I_Curl * in_adjoint[local_dof_indices[i]];
587       }
588       ret.push_back(item);
589     }
590   }
591   return ret;
592 }
```

compute_mode_strength()

`ComplexNumber InnerDomain::compute_mode_strength ( )`

Computes the norm of the input mode for scaling of the output signal.

Returns

ComplexNumber

Definition at line 496 of file InnerDomain.cpp.

```
496 {
497   ComplexNumber ret(0,0);
```

```
    if(GlobalParams.Index_in_z_direction == GlobalParams.Blocks_in_z_direction - 1) {
      Vector<ComplexNumber> mode_a(3), mode_b(3);
      std::vector<Position> quadrature_points;
      for(auto cell : triangulation) {
        if(cell.at_boundary()) {
          for(unsigned int i = 0; i < 6; i++) {
            if(cell.face(i)->boundary_id() == 5) {
              quadrature_points.push_back(cell.face(i)->center());
            }
          }
        }
      }
      for(unsigned int index = 0; index < quadrature_points.size(); index++) {
        quadrature_points[index][2] = quadrature_points[index][2] - 2 * FLOATING_PRECISION;
      }
      for(unsigned int index = 0; index < quadrature_points.size(); index++) {
        GlobalParams.source_field->vector_value(quadrature_points[index], mode_a);
        for(unsigned int comp = 0; comp < 3; comp++) {
          mode_b[comp] = conjugate(mode_a[comp]);
        }
        ret += mode_a[0]*mode_b[0] + mode_a[1]*mode_b[1] + mode_a[2] * mode_b[2];
      }
      ret /= (quadrature_points.size());
      return ret;
    }
    return ret;
}
```

compute_n_locally_active_dofs()

`DofCount InnerDomain::compute_n_locally_active_dofs ( ) [override], [virtual]`

Fulfills FEDomain interface.

See definition there.

Returns

DofCount

Implements FEDomain.

Definition at line 442 of file InnerDomain.cpp.

```
{
    return dof_handler.n_dofs();
}
```

compute_n_locally_owned_dofs()

`DofCount InnerDomain::compute_n_locally_owned_dofs ( ) [override], [virtual]`

Fulfills FEDomain interface.

See definition there.

Returns

DofCount

Implements FEDomain.

Definition at line 426 of file InnerDomain.cpp.

```
                                            {
  IndexSet set_of_locally_owned_dofs(dof_handler.n_dofs());
  set_of_locally_owned_dofs.add_range(0,dof_handler.n_dofs());
  IndexSet dofs_to_remove(dof_handler.n_dofs());
  for(unsigned int surf = 0; surf < 6; surf += 2) {
    if(Geometry.levels[level].surface_type[surf] == SurfaceType::NEIGHBOR_SURFACE) {
      std::vector<InterfaceDofData> dofs = get_surface_dof_vector_for_boundary_id(surf);
      for(unsigned int i = 0; i < dofs.size(); i++) {
        dofs_to_remove.add_index(dofs[i].index);
      }
    }
  }
  set_of_locally_owned_dofs.subtract_set(dofs_to_remove);
  return set_of_locally_owned_dofs.n_elements();
}
```

compute_signal_strength()

`ComplexNumber InnerDomain::compute_signal_strength ( dealii::LinearAlgebra::distributed::Vector< ComplexNumber > * in_solution )`

Computes how strongly the fundamental mode is excited in the output waveguide in the field provided as the input.

Parameters

in_solution	The solution to check this for.

Returns

ComplexNumber The complex phase and amplitude of the fundamental mode in the solution.

Definition at line 459 of file InnerDomain.cpp.

```
459
               {
460   ComplexNumber ret(0,0);
461   if(GlobalParams.Index_in_z_direction == GlobalParams.Blocks_in_z_direction - 1) {
462     NumericVectorLocal local_solution;
463     local_solution.reinit(n_locally_active_dofs);
464     for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
465       local_solution[i] = in_solution->operator[](global_index_mapping[i]);
466     }
467     Vector<ComplexNumber> fe_evaluation(3);
468     Vector<ComplexNumber> mode(3);
469     std::vector<Position> quadrature_points;
470     for(auto cell : triangulation) {
471       if(cell.at_boundary()) {
472         for(unsigned int i = 0; i < 6; i++) {
473           if(cell.face(i)->boundary_id() == 5) {
474             quadrature_points.push_back(cell.face(i)->center());
475           }
476         }
477       }
478     }
479     for(unsigned int index = 0; index < quadrature_points.size(); index++) {
480       quadrature_points[index][2] = quadrature_points[index][2] - 2 * FLOATING_PRECISION; // This is only to make sure that even on large mesges, there are no rounding errors that lead the code to throw an error because the position isnt "inside" the mesh.
481     }
482     for(unsigned int index = 0; index < quadrature_points.size(); index++) {
483       VectorTools::point_value(dof_handler, local_solution, quadrature_points[index], fe_evaluation);
484       GlobalParams.source_field->vector_value(quadrature_points[index], mode);
485       for(unsigned int comp = 0; comp < 3; comp++) {
486         mode[comp] = conjugate(mode[comp]);
```

```
    }
    ret += fe_evaluation[0]*mode[0] + fe_evaluation[1]*mode[1] + fe_evaluation[2] * mode[2];
  }
  ret /= (quadrature_points.size());
  return ret;
 }
 return ret;
}
```

determine_non_owned_dofs()

`void InnerDomain::determine_non_owned_dofs ( )  [override], [virtual]`

Fulfills FEDomain interface.

See definition there.

Implements FEDomain.

Definition at line 446 of file InnerDomain.cpp.

```
446                                          {
447  for(unsigned int i = 0; i < 6; i += 2) {
448    if(Geometry.levels[level].surface_type[i] == SurfaceType::NEIGHBOR_SURFACE) {
449      std::vector<InterfaceDofData> dof_data = get_surface_dof_vector_for_boundary_id(i);
450      std::vector<unsigned int> local_dof_indices(dof_data.size());
451      for(unsigned int j = 0; j < dof_data.size(); j++) {
452        local_dof_indices[j] = dof_data[j].index;
453      }
454      mark_local_dofs_as_non_local(local_dof_indices);
455    }
456  }
457 }
```

evaluate_at_positions()

```
std::vector< std::vector< ComplexNumber > > InnerDomain::evaluate_at_positions (
             std::vector< Position > in_positions,
             NumericVectorLocal in_solution )
```

Evaluates the provided solution (represented by in_solution) at the given positions, i.e.

computes the E-Field at a given locations.

Parameters

in_positions	The positions we want to know the solution at.
in_solution	The solution vector from the finite element method.

Returns

std::vector<std::vector<ComplexNumber>> The vector of field evaluations.

Definition at line 543 of file InnerDomain.cpp.

```
543
                                     {
544  std::vector<std::vector<ComplexNumber» ret;
545  QGauss<3> q(GlobalParams.Nedelec_element_order + 2);
```

```
  for(unsigned int i = 0; i < in_positions.size(); i++) {
    Vector<ComplexNumber> fe_evaluation(3);
    VectorTools::point_value(dof_handler, in_solution, in_positions[i], fe_evaluation);
    std::vector<ComplexNumber> point_val;
    point_val.push_back(fe_evaluation[0]);
    point_val.push_back(fe_evaluation[1]);
    point_val.push_back(fe_evaluation[2]);
    ret.push_back(point_val);
  }
  return ret;
}
```

evaluate_J_at()

```
Tensor<1,3,ComplexNumber> InnerDomain::evaluate_J_at (
            Position in_p )
```

Computes the forcing term J for a given position so we can use it to build a right-hand side / forcing term.

Parameters

in_p	The position to evaluate J at.

Returns

Tensor<1,3,ComplexNumber> The complex vector containing the three components of J at the given location.

fill_sparsity_pattern()

```
void InnerDomain::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_pattern,
            Constraints * constraints )
```

Marks all index pairs that are non-zero in the provided matrix using the given constraints.

See the dealii documentation for more details on how this is done and why.

Parameters

in_pattern	The pattern to fill.
constraints	The constraints to consider.

Definition at line 89 of file InnerDomain.cpp.

```
89
        {
90   auto end = dof_handler.end();
91   std::vector<DofNumber> cell_dof_indices(fe.dofs_per_cell);
92   for(auto cell = dof_handler.begin_active(); cell != end; cell++) {
93     cell->get_dof_indices(cell_dof_indices);
94     cell_dof_indices = transform_local_to_global_dofs(cell_dof_indices);
95     in_constraints->add_entries_local_to_global(cell_dof_indices, *in_pattern);
96   }
```

```
97 }
```

References FEDomain::transform_local_to_global_dofs().

get_surface_dof_vector_for_boundary_id()

std::vector< InterfaceDofData > InnerDomain::get_surface_dof_vector_for_boundary_id (
    BoundaryId *b_id*)

Returns a vector of all dofs active on the given surface.

This cna be used to build the coupling of the interior with a boundary condition.

Parameters

b_id	The boundary one is interested in.

Returns

std::vector<InterfaceDofData> The vector of dofs on that surface.

Definition at line 99 of file InnerDomain.cpp.

```
99                                                                                       {
100   std::vector<InterfaceDofData> ret;
101   std::vector<types::global_dof_index> local_line_dofs(fe.dofs_per_line);
102   std::set<DofNumber> line_set;
103   std::vector<DofNumber> local_face_dofs(fe.dofs_per_face);
104   std::set<DofNumber> face_set;
105   triangulation.clear_user_flags();
106   for (auto cell : dof_handler.active_cell_iterators()) {
107     if (cell->at_boundary(b_id)) {
108       bool found_one = false;
109       for (unsigned int face = 0; face < 6; face++) {
110         if (cell->face(face)->boundary_id() == b_id && found_one) {
111           print_info("InnerDomain::get_surface_dof_vector_for_boundary_id", "There was an error!",
    LoggingLevel::PRODUCTION_ALL);
112         }
113         if (cell->face(face)->boundary_id() == b_id) {
114           found_one = true;
115           std::vector<DofNumber> face_dofs_indices(fe.dofs_per_face);
116           cell->face(face)->get_dof_indices(face_dofs_indices);
117           face_set.clear();
118           face_set.insert(face_dofs_indices.begin(), face_dofs_indices.end());
119           std::vector<InterfaceDofData> cell_dofs_and_orientations_and_points;
120           for (unsigned int i = 0; i < dealii::GeometryInfo<3>::lines_per_face; i++) {
121             std::vector<DofNumber> line_dofs(fe.dofs_per_line);
122             cell->face(face)->line(i)->get_dof_indices(line_dofs);
123             line_set.clear();
124             line_set.insert(line_dofs.begin(), line_dofs.end());
125             for(auto erase_it: line_set) {
126               face_set.erase(erase_it);
127             }
128             if(!cell->face(face)->line(i)->user_flag_set()) {
129               for (unsigned int j = 0; j < fe.dofs_per_line; j++) {
130                 InterfaceDofData new_item;
131                 new_item.index = line_dofs[j];
132                 new_item.base_point = cell->face(face)->line(i)->center();
133                 new_item.order = j;
134                 cell_dofs_and_orientations_and_points.push_back(new_item);
135               }
136               cell->face(face)->line(i)->set_user_flag();
137             }
138           }
139           unsigned int index = 0;
```

```
          for (auto item: face_set) {
            InterfaceDofData new_item;
            new_item.index = item;
            new_item.base_point = cell->face(face)->center();
            new_item.order = 0;
            cell_dofs_and_orientations_and_points.push_back(new_item);
            index++;
          }
          for (auto item: cell_dofs_and_orientations_and_points) {
            ret.push_back(item);
          }
        }
      }
    }
  }
  ret.shrink_to_fit();
  std::sort(ret.begin(), ret.end(), compareDofBaseDataAndOrientation);
  return ret;
}
```

load_exact_solution()

`void InnerDomain::load_exact_solution ( )`

In many places it can be useful to have an interpolated exact solution for the waveguide or Hertz case.

This function ensures the analytical solution is available and projects it onto the FE space to compute a solution vector.

Definition at line 51 of file InnerDomain.cpp.

```
51                                      {
52   if(!exact_solution_is_initialized) {
53     dealii::IndexSet local_indices(n_locally_active_dofs);
54     local_indices.add_range(0,n_locally_active_dofs);
55     Constraints local_constraints(local_indices);
56     local_constraints.close();
57     exact_solution_interpolated.reinit(n_locally_active_dofs);
58     VectorTools::project(dof_handler, local_constraints,
       dealii::QGauss<3>(GlobalParams.Nedelec_element_order + 2), *GlobalParams.source_field,
       exact_solution_interpolated);
59     exact_solution_is_initialized = true;
60     print_info("InnerDomain::load_exact_solution", "Norm of interpolated mode signal is " +
       std::to_string(exact_solution_interpolated.l2_norm()));
61   }
62 }
```

Referenced by assemble_system().

output_results()

```
std::string InnerDomain::output_results (
            std::string in_filename,
            NumericVectorLocal in_solution,
            bool apply_space_transformation )
```

Generates an output file of the provided solution vector on the local domain.

Parameters

in_filename	The filename to be used for the output. This will be made unique by appending process ids.
in_solution	The solution vector representing the solution on the described domain.
apply_space_transformation	If set to true, the output domain will be transformed to the physical coordinates.

Returns

std::string The actual filename used after making it unique. This can be used to write the fileset files.

Definition at line 341 of file InnerDomain.cpp.

```
341                    {
342   print_info("InnerDomain::output_results()", "Start");
343   const unsigned int n_cells = dof_handler.get_triangulation().n_active_cells();
344   unsigned int counter = 0;
345   dealii::Vector<double> eps_abs(n_cells);
346   for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
347     Position p = it->center();
348     MaterialTensor transformation;
349     if(apply_transformation) {
350       for(unsigned int i = 0 ; i < 3; i++) {
351         for(unsigned int j = 0; j < 3; j++) {
352           if(i == j) {
353             transformation[i][j] = ComplexNumber(1,0);
354           } else {
355             transformation[i][j] = ComplexNumber(0,0);
356           }
357         }
358       }
359     } else {
360       transformation = GlobalSpaceTransformation->get_Space_Transformation_Tensor(p);
361     }
362     MaterialTensor epsilon;
363     if (Geometry.math_coordinate_in_waveguide(p)) {
364       epsilon = transformation * GlobalParams.Epsilon_R_in_waveguide;
365     } else {
366       epsilon = transformation * GlobalParams.Epsilon_R_outside_waveguide;
367     }
368     eps_abs[counter] = epsilon.norm();
369     counter++;
370   }
371   if(apply_transformation) {
372     GlobalSpaceTransformation->switch_application_mode(true);
373     dealii::GridTools::transform(*GlobalSpaceTransformation, triangulation);
374   }
375   dealii::Vector<ComplexNumber> interpolated_exact_solution(in_solution.size());
376 
377   data_out.clear();
378   data_out.attach_dof_handler(dof_handler);
379   data_out.add_data_vector(in_solution, "Solution");
380 
381   data_out.add_data_vector(eps_abs, "Epsilon");
382   dealii::Vector<double> index_x(n_cells), index_y(n_cells), index_z(n_cells);
383   for(unsigned int i = 0; i < n_cells; i++) {
384     index_x[i] = GlobalParams.Index_in_x_direction;
385     index_y[i] = GlobalParams.Index_in_y_direction;
386     index_z[i] = GlobalParams.Index_in_z_direction;
387   }
388   data_out.add_data_vector(index_x, "IndexX");
389   data_out.add_data_vector(index_y, "IndexY");
390   data_out.add_data_vector(index_z, "IndexZ");
391   std::string filename = GlobalOutputManager.get_numbered_filename(in_filename, GlobalParams.MPI_Rank,
       "vtu");
```

```
  std::ofstream outputvtu(filename);

  Function<3,ComplexNumber> * esc;
  if(!apply_transformation) {
    data_out.add_data_vector(exact_solution_interpolated, "Exact_Solution");
  } else {
    esc = GlobalParams.source_field;
    dealii::IndexSet local_indices(n_locally_active_dofs);
    local_indices.add_range(0,n_locally_active_dofs);
    Constraints local_constraints(local_indices);
    local_constraints.close();
    if(GlobalParams.Point_Source_Type == 0 || GlobalParams.Point_Source_Type == 3) {
      VectorTools::project(dof_handler, local_constraints,
   dealii::QGauss<3>(GlobalParams.Nedelec_element_order + 2), *esc, interpolated_exact_solution);
      data_out.add_data_vector(interpolated_exact_solution, "Exact_Solution");
    }
  }

  dealii::Vector<ComplexNumber> error_vector(in_solution.size());
  for(unsigned int i = 0; i < in_solution.size(); i++) {
    error_vector[i] = in_solution[i] - exact_solution_interpolated[i];
  }
  data_out.add_data_vector(error_vector, "SolutionError");

  data_out.build_patches();
  data_out.write_vtu(outputvtu);
  if(apply_transformation) {
    delete esc;
    GlobalSpaceTransformation->switch_application_mode(false);
    dealii::GridTools::transform(*GlobalSpaceTransformation, triangulation);
  }
  print_info("InnerDomain::output_results()", "End");
  return filename;
}
```

write_matrix_and_rhs_metrics()

```
void InnerDomain::write_matrix_and_rhs_metrics (
            dealii::PETScWrappers::MatrixBase * matrix,
            NumericVectorDistributed * rhs )
```

Prints some diagnostic data to the console.

Parameters

matrix	
rhs	

Definition at line 334 of file InnerDomain.cpp.

```
334
                 {
335   print_info("InnerDomain::write_matrix_and_rhs_metrics", "Start", LoggingLevel::DEBUG_ALL);
336   print_info("InnerDomain::write_matrix_and_rhs", "System Matrix l_1 norm: " +
       std::to_string(matrix->l1_norm()), LoggingLevel::PRODUCTION_ALL);
337   print_info("InnerDomain::write_matrix_and_rhs", "RHS L_2 norm:  " + std::to_string(rhs->l2_norm()),
       LoggingLevel::PRODUCTION_ALL);
338   print_info("InnerDomain::write_matrix_and_rhs_metrics", "End");
339 }
```

The documentation for this class was generated from the following files:

- Code/Core/InnerDomain.h

- Code/Core/InnerDomain.cpp

38 InterfaceDofData Struct Reference

Public Member Functions

- **InterfaceDofData** (const DofNumber &in_index, const Position &in_position)

Public Attributes

- DofNumber **index**
- Position **base_point**
- unsigned int **order**

38.1 Detailed Description

Definition at line 143 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

39 J_derivative_terms Struct Reference

Public Attributes

- ComplexNumber **f**
- ComplexNumber **d_f_dyy**
- ComplexNumber **d_f_dxy**
- ComplexNumber **d_f_dx**
- ComplexNumber **h**
- ComplexNumber **d_h_dx**
- ComplexNumber **d_h_dy**
- ComplexNumber **d_h_dxx**
- ComplexNumber **d_h_dyy**
- double **beta**

39.1 Detailed Description

Definition at line 241 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

40 JacobianAndTensorData Struct Reference

Public Attributes

- dealii::Tensor< 2, 3, double > **C**
- dealii::Tensor< 2, 3, double > **G**
- dealii::Tensor< 2, 3, double > **J**

40.1 Detailed Description

Definition at line 168 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

41 JacobianForCell Class Reference

This class is only for internal use.

`#include <JacobianForCell.h>`

Public Member Functions

- JacobianForCell (FaceAngelingData &in_fad, const BoundaryId &b_id, double additional_component)

  Construct a new Jacobian For Cell object.
- void reinit_for_cell (CellIterator2D)

  Builds the base data for the provided cell.
- void reinit (FaceAngelingData &in_fad, const BoundaryId &b_id, double additional_component)

  Does the same as the constructor.
- auto get_C_G_and_J (Position2D) -> JacobianAndTensorData

  Get the C G and J tensors used in the HSIE formulation.
- std::pair< Position2D, double > split_into_triangulation_and_external_part (const Position in_point)

  For a given Cordinate in 3D, this identifies its position on the surface and the orthogonal part.
- dealii::Tensor< 2, 3, double > get_J_hat_for_position (const Position2D &position) const

  Evaluates the Jacobian at the given position.
- auto transform_to_3D_space (Position2D position) -> Position

  Takes a position on the surface and provides the 3D coordinate.

Static Public Member Functions

- static bool is_line_in_x_direction (dealii::internal::DoFHandlerImplementation::Iterators< 2, 2, false >::line_iterator line)

Checks if a edge on the HSIESurface points in the x or y direction.

- static bool is_line_in_y_direction (dealii::internal::DoFHandlerImplementation::Iterators< 2, 2, false >::line_iterator line)

  Checks if a edge on the HSIESurface points in the x or y direction.

Public Attributes

- dealii::Differentiation::SD::types::substitution_map **surface_wide_substitution_map**
- BoundaryId **boundary_id**
- double **additional_component**
- std::vector< bool > **b_ids_have_hsie**
- MathExpression **x**
- MathExpression **y**
- MathExpression **z**
- MathExpression **z0**
- dealii::Tensor< 1, 3, MathExpression > **F**
- dealii::Tensor< 2, 3, MathExpression > **J**

41.1 Detailed Description

This class is only for internal use.

The jacobian it represents is used in the HSIESurface to represent the transformation of the cell onto a cuboid. If the external direction is chosen axis-parallel, this is an identity transformation.

Definition at line 24 of file JacobianForCell.h.

41.2 Constructor & Destructor Documentation

JacobianForCell()

```
JacobianForCell::JacobianForCell (
            FaceAngelingData & in_fad,
            const BoundaryId & b_id,
            double additional_component )
```

Construct a new Jacobian For Cell object.

Parameters

in_fad	denotes which faces are angled (45 degrees) and which are not.
b_id	the boundary id of the surface the cell belongs to.
additional_component	orthogonal surface coordinate.

Definition at line 15 of file JacobianForCell.cpp.

```
15                {
16   reinit(in_fad, in_bid, in_additional_component);
17 }
```

References reinit().

41.3 Member Function Documentation

get_C_G_and_J()

```
JacobianAndTensorData JacobianForCell::get_C_G_and_J (
            Position2D in_p ) -> JacobianAndTensorData
```

Get the C G and J tensors used in the HSIE formulation.

See also

HSIESurface

Returns

JacobianAndTensorData

Definition at line 66 of file JacobianForCell.cpp.

```
66                                                              {
67   JacobianAndTensorData ret;
68   ret.J = get_J_hat_for_position(in_p);
69   const double J_norm = ret.J.norm();
70   const Tensor<2,3,double> J_inverse = invert(ret.J);
71   ret.G = J_norm * J_inverse * transpose(J_inverse);
72   ret.C = (1.0 / J_norm) * transpose(ret.J) * ret.J;
73   return ret;
74 }
```

References get_J_hat_for_position().

get_J_hat_for_position()

```
dealii::Tensor< 2, 3, double > JacobianForCell::get_J_hat_for_position (
            const Position2D & position ) const
```

Evaluates the Jacobian at the given position.

Parameters

position	2D coordinate to evaluate the jacobian at.

Returns

dealii::Tensor<2,3,double>

Definition at line 52 of file JacobianForCell.cpp.

```
52                                                                                                       {
```

```
  dealii::Tensor<2,3,double> ret;
  dealii::Differentiation::SD::types::substitution_map substitution_map;
  substitution_map[x] = MathExpression(position[0]);
  substitution_map[y] = MathExpression(position[1]);
  substitution_map[z] = MathExpression(additional_component);
  for(unsigned int i = 0; i < 3; i++){
    for(unsigned int j = 0; j < 3; j++){
      ret[i][j] = J[i][j].substitute_and_evaluate<double>(substitution_map);
    }
  }
  return ret;
}
```

Referenced by get_C_G_and_J().

is_line_in_x_direction()

```
static bool JacobianForCell::is_line_in_x_direction (
            dealii::internal::DoFHandlerImplementation::Iterators< 2, 2, false >::line_iterator
line ) [static]
```

Checks if a edge on the HSIESurface points in the x or y direction.

Parameters

line	An iterator pointing to a line in a surface triangualtion.

Returns

true the line points in the x-direction

false the line does not point in the x-direction

is_line_in_y_direction()

```
static bool JacobianForCell::is_line_in_y_direction (
            dealii::internal::DoFHandlerImplementation::Iterators< 2, 2, false >::line_iterator
line ) [static]
```

Checks if a edge on the HSIESurface points in the x or y direction.

Parameters

line	An iterator pointing to a line in a surface triangualtion.

Returns

true the line points in the y-direction

false the line does not point in the y-direction

reinit()

```
void JacobianForCell::reinit (
            FaceAngelingData & in_fad,
            const BoundaryId & b_id,
            double additional_component )
```

Does the same as the constructor.

Parameters

in_fad	denotes which faces are angled (45 degrees) and which are not.
b_id	the boundary id of the surface the cell belongs to.
additional_component	orthogonal surface coordinate.

Definition at line 19 of file JacobianForCell.cpp.

```
19 {
20   x                    = {"x"};
21   y                    = {"y"};
22   z                    = {"z"};
23   z0                   = {"z0"};
24   boundary_id          = in_bid;
25   additional_component = in_additional_component;
26   bool all_straight = true;
27   for(unsigned int i = 0; i < 4; i++) {
28     if(!in_fad[i].is_x_angled || !in_fad[i].is_y_angled) {
29       all_straight = false;
30     }
31   }
32   if(all_straight) {
33     F[0]                   = x;
34     F[1]                   = y;
35     F[2]                   = (z-z0);
36   } else {
37     F[0]                   = x;
38     F[1]                   = y;
39     F[2]                   = (z-z0);
40   }
41   surface_wide_substitution_map[z0]  = MathExpression(in_additional_component);
42   for(unsigned int i = 0; i <3; i++) {
43     F[i] = F[i].substitute(surface_wide_substitution_map);
44   }
45   for(unsigned int i = 0; i < 3; i++) {
46     J[i][0] = F[i].differentiate(x);
47     J[i][1] = F[i].differentiate(y);
48     J[i][2] = F[i].differentiate(z);
49   }
50 }
```

Referenced by JacobianForCell().

reinit_for_cell()

```
void JacobianForCell::reinit_for_cell (
            CellIterator2D  )
```

Builds the base data for the provided cell.

split_into_triangulation_and_external_part()

```
std::pair< Position2D, double > JacobianForCell::split_into_triangulation_and_external_part (
            const Position in_point )
```

For a given Cordinate in 3D, this identifies its position on the surface and the orthogonal part.

Parameters

in_point	The position in 3D

Returns

std::pair<Position2D,double> Th cordinate in 2D and the orthogonal part

Definition at line 99 of file JacobianForCell.cpp.

```
99
        {
100   Position temp = in_point;
101   if (boundary_id == 0) {
102     temp = Transform_0_to_5(in_point);
103   }
104   if (boundary_id == 1) {
105     temp = Transform_1_to_5(in_point);
106   }
107   if (boundary_id == 2) {
108     temp = Transform_2_to_5(in_point);
109   }
110   if (boundary_id == 3) {
111     temp = Transform_3_to_5(in_point);
112   }
113   if (boundary_id == 4) {
114     temp = Transform_4_to_5(in_point);
115   }
116   return {{temp[0], temp[1]}, temp[2]};
117 }
```

transform_to_3D_space()

```
Position JacobianForCell::transform_to_3D_space (
            Position2D position ) -> Position
```

Takes a position on the surface and provides the 3D coordinate.

Parameters

position	location on the surface

Returns

Position

Definition at line 76 of file JacobianForCell.cpp.

```
76                                                                   {
77   Position ret= {in_position[0], in_position[1], additional_component};
78   if (boundary_id == 0) {
79     return Transform_5_to_0(ret);
80   }
81   if (boundary_id == 1) {
82     return Transform_5_to_1(ret);
83   }
84   if (boundary_id == 2) {
85     return Transform_5_to_2(ret);
86   }
87   if (boundary_id == 3) {
88     return Transform_5_to_3(ret);
89   }
90   if (boundary_id == 4) {
91     return Transform_5_to_4(ret);
92   }
93   if (boundary_id == 5) {
94     return ret;
95   }
96   return ret;
97 }
```

The documentation for this class was generated from the following files:

- Code/BoundaryCondition/JacobianForCell.h
- Code/BoundaryCondition/JacobianForCell.cpp

42 LaguerreFunction Class Reference

Static Public Member Functions

- static double **evaluate** (unsigned int n, unsigned int m, double x)
- static double **factorial** (unsigned int n)
- static unsigned int **binomial_coefficient** (unsigned int n, unsigned int k)

42.1 Detailed Description

Definition at line 20 of file LaguerreFunction.h.

The documentation for this class was generated from the following files:

- Code/BoundaryCondition/LaguerreFunction.h
- Code/BoundaryCondition/LaguerreFunction.cpp

43 LaguerreFunctions Class Reference

`#include <LaguerreFunction.h>`

43.1 Detailed Description

These is not currently being used. It will be used in a complex scaled infinite element once that is implemented. Since it is not currently used, this is not documented.

The documentation for this class was generated from the following file:

- Code/BoundaryCondition/LaguerreFunction.h

44 LevelDofIndexData Class Reference

44.1 Detailed Description

Definition at line 2 of file LevelDofIndexData.h.

The documentation for this class was generated from the following files:

- Code/Hierarchy/LevelDofIndexData.h
- Code/Hierarchy/LevelDofIndexData.cpp

45 LevelDofOwnershipData Struct Reference

Public Member Functions

- **LevelDofOwnershipData** (unsigned int in_global)

Public Attributes

- unsigned int **global_dofs**
- unsigned int **owned_dofs**
- dealii::IndexSet **locally_owned_dofs**
- dealii::IndexSet **input_dofs**
- dealii::IndexSet **output_dofs**
- dealii::IndexSet **locally_relevant_dofs**

45.1 Detailed Description

Definition at line 180 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

46 LevelGeometry Struct Reference

Public Attributes

- std::array< SurfaceType, 6 > **surface_type**
- CubeSurfaceTruncationState **is_surface_truncated**
- std::array< std::shared_ptr< BoundaryCondition >, 6 > **surfaces**
- std::vector< dealii::IndexSet > **dof_distribution**
- DofNumber **n_local_dofs**
- DofNumber **n_total_level_dofs**
- InnerDomain ∗ **inner_domain**

46.1 Detailed Description

Definition at line 36 of file GeometryManager.h.

The documentation for this struct was generated from the following file:

- Code/GlobalObjects/GeometryManager.h

47 LocalMatrixPart Struct Reference

Public Attributes

- dealii::AffineConstraints< ComplexNumber > **constraints**
- dealii::SparsityPattern **sp**
- dealii::SparseMatrix< ComplexNumber > **matrix**
- unsigned int **n_dofs**
- dealii::IndexSet **lower_sweeping_dofs**
- dealii::IndexSet **upper_sweeping_dofs**
- dealii::IndexSet **local_dofs**

47.1 Detailed Description

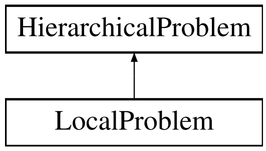
Definition at line 64 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

48 LocalProblem Class Reference

Inheritance diagram for LocalProblem:



Public Member Functions

- LocalProblem ()

    Construct a new LocalProblem object This initializes the local solver object and the matrix (not its sparsity pattern).

- ~LocalProblem () override

    Deletes the system matrix.

- void solve () override

    Calls the direct sovler.

- void initialize () override

    Calls the reinitialization of the data structures.

- void assemble () override

    Assembles the local problem (inner domain and boundary methods).

- void initialize_index_sets () override

    For local problems this is relatively simple because all locally active dofs are also locally owned.

- void validate ()

    This function only outputs some diagnostic data about the system matrix.

- auto reinit () -> void override

    Reinitializes the data structures (solution vector, builds constraints, makes sparsity pattern, reinits the matrix).

- auto reinit_rhs () -> void override

    Reinits the right hand side vector.

- dealii::IndexSet compute_interface_dof_set (BoundaryId interface_id)

    Computes the interface dofs index set for all the dofs on a surface of the inner domain.

- void compute_solver_factorization () override

    This level uses a direct solver (MUMPS) and this function computes the LDL^T factorization it uses internally.

- double compute_L2_error ()

    Computes the L2 error of the solution that was computed last compaired to the exact solution of the problem.

- double compute_error ()

    Computes the L2 error and runs a time measurement around it.

- unsigned int compute_global_solve_counter () override

    All LocalProblem objects add up how often they have called their solver.

- void empty_memory () override

Frees up some memory from datastructures that are only required during the solution process to slim down the memory consumption after solving has terminated.

- void write_multifile_output (const std::string &in_filename, bool transform=false) override

  Writes output of the solution of this problem including the boundary conditions and also provides a meta-file that can be used in Paraview to load all output by opening one file.

- void make_sparsity_pattern () override

  Not implemented on this level, see derived classes.

Public Attributes

- SolverControl **sc**
- dealii::PETScWrappers::SparseDirectMUMPS **solver**

48.1 Detailed Description

Definition at line 13 of file LocalProblem.h.

48.2 Constructor & Destructor Documentation

LocalProblem()

`LocalProblem::LocalProblem ( )`

Construct a new LocalProblem object This initializes the local solver object and the matrix (not its sparsity pattern).

It also copies the set of locally owned dofs.

Definition at line 41 of file LocalProblem.cpp.

```
41                              :
42   HierarchicalProblem(0, SweepingDirection::Z),
43   sc(),
44   solver(sc, MPI_COMM_SELF) {
45     solver.set_symmetric_mode(true);
46     print_info("Local Problem", "Done building base problem. Preparing matrix.");
47     matrix = new dealii::PETScWrappers::MPI::SparseMatrix();
48     for(unsigned int i = 0; i < 6; i++) Geometry.levels[0].is_surface_truncated[i] = true;
49     own_dofs = Geometry.levels[0].dof_distribution[0];
50 }
```

48.3 Member Function Documentation

compute_error()

`double LocalProblem::compute_error ( )`

Computes the L2 error and runs a time measurement around it.

Returns

double returns the error value.

Definition at line 168 of file LocalProblem.cpp.

```
168                                    {
169   Timer timer;
170   timer.start ();
171   double error = compute_L2_error();
172   timer.stop ();
173   print_info("LocalProblem::compute_error", "L2 Error: " + std::to_string(error) + " (computed in " +
       std::to_string(timer.cpu_time()) + "s)");
174   return error;
175 }
```

References compute_L2_error().

compute_global_solve_counter()

`unsigned int LocalProblem::compute_global_solve_counter ( )  [override], [virtual]`

All LocalProblem objects add up how often they have called their solver.

Returns

unsigned int Number of solver runs on the lowest level.

Reimplemented from HierarchicalProblem.

Definition at line 194 of file LocalProblem.cpp.

```
194                                                        {
195   return Utilities::MPI::sum(solve_counter, MPI_COMM_WORLD);
196 }
```

compute_interface_dof_set()

```
dealii::IndexSet LocalProblem::compute_interface_dof_set (
               BoundaryId interface_id )
```

Computes the interface dofs index set for all the dofs on a surface of the inner domain.

Parameters

interface_id	

Returns

dealii::IndexSet

Definition at line 56 of file LocalProblem.cpp.

```
56                                                                            {
57   BoundaryId opposing_interface_id = opposing_Boundary_Id(interface_id);
58   dealii::IndexSet ret(Geometry.levels[0].n_local_dofs);
59   for(unsigned int i = 0; i < 6; i++) {
60     if( i == interface_id) {
```

```
    std::vector<InterfaceDofData> current =
  Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(interface_id);
    for(unsigned int j = 0; j < current.size(); j++) {
      ret.add_index(current[j].index);
    }
  } else {
    if(i != opposing_interface_id && Geometry.levels[0].is_surface_truncated[i]) {
      std::vector<InterfaceDofData> current =
  Geometry.levels[0].surfaces[i]->get_dof_association_by_boundary_id(i);
      for(unsigned int j = 0; j < current.size(); j++) {
        ret.add_index(current[j].index);
      }
    }
  }
}
return ret;
}
```

compute_L2_error()

`double LocalProblem::compute_L2_error ( )`

Computes the L2 error of the solution that was computed last compaired to the exact solution of the problem.

Keep in mind that the "exact solution" for the waveguide case is a mode propagating on a straight waveguide, which is not applicable for a bent waveguide.

Returns

double Error value.

Definition at line 177 of file LocalProblem.cpp.

```
177                                      {
178   NumericVectorLocal solution_inner(Geometry.levels[level].inner_domain->n_locally_active_dofs);
179   for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->n_locally_active_dofs; i++) {
180     solution_inner[i] = solution(i);
181   }
182   dealii::Vector<double>
    cellwise_error(Geometry.levels[level].inner_domain->triangulation.n_active_cells());
183   dealii::VectorTools::integrate_difference(
184     MappingQGeneric<3>(1),
185     Geometry.levels[level].inner_domain->dof_handler,
186     solution_inner,
187     *GlobalParams.source_field,
188     cellwise_error,
189     dealii::QGauss<3>(GlobalParams.Nedelec_element_order + 2),
190     dealii::VectorTools::NormType::L2_norm );
191   return dealii::VectorTools::compute_global_error(Geometry.levels[level].inner_domain->triangulation,
    cellwise_error, dealii::VectorTools::NormType::L2_norm);
192 }
```

Referenced by compute_error().

compute_solver_factorization()

`void LocalProblem::compute_solver_factorization ( )  [override], [virtual]`

This level uses a direct solver (MUMPS) and this function computes the LDL^T factorization it uses internally.

The solve function of LocalProblem objects are called sequentially in the sweeping preconditioner. The factorization only has to be computed once but that step is expensive. By providing this function we can call it in parallel on all LocalProblems resulting in perfect parallelization of the effort.

Implements HierarchicalProblem.

Definition at line 158 of file LocalProblem.cpp.

```
158                                                    {
159    Timer timer1;
160    print_info("LocalProblem::compute_solver_factorization", "Begin solver factorization: ",
        LoggingLevel::PRODUCTION_ONE);
161    timer1.start();
162    solve();
163    timer1.stop();
164    solution = 0;
165    print_info("LocalProblem::compute_solver_factorization", "Walltime: " +
        std::to_string(timer1.wall_time()) , LoggingLevel::PRODUCTION_ONE);
166 }
```

write_multifile_output()

```
void LocalProblem::write_multifile_output (
              const std::string & in_filename,
              bool transform = false )  [override], [virtual]
```

Writes output of the solution of this problem including the boundary conditions and also provides a meta-file that can be used in Paraview to load all output by opening one file.

Parameters

in_filename	Name to use for the output file
transform	If set to true, the output will be in the physical coordinate system.

Implements HierarchicalProblem.

Definition at line 203 of file LocalProblem.cpp.

```
203                                                                                  {
204    NumericVectorLocal local_solution(Geometry.levels[0].inner_domain->n_locally_active_dofs);
205    std::vector<std::string> generated_files;
206    for(unsigned int i = 0; i < Geometry.levels[0].inner_domain->n_locally_active_dofs; i++) {
207      local_solution[i] = solution[i];
208    }
209
210    std::string file_1 = Geometry.levels[0].inner_domain->output_results(in_filename + "0" ,
        local_solution, false);
211    generated_files.push_back(file_1);
212    if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
213      for (unsigned int surf = 0; surf < 6; surf++) {
214        if(Geometry.levels[0].surface_type[surf] == SurfaceType::ABC_SURFACE){
215          dealii::Vector<ComplexNumber> ds (Geometry.levels[0].surfaces[surf]->n_locally_active_dofs);
216          for(unsigned int index = 0; index < Geometry.levels[0].surfaces[surf]->n_locally_active_dofs;
        index++) {
217            ds[index] = solution[Geometry.levels[0].surfaces[surf]->global_index_mapping[index]];
218          }
219          std::string file_2 = Geometry.levels[0].surfaces[surf]->output_results(ds, in_filename +
        "_pml0");
220          generated_files.push_back(file_2);
221        }
222      }
223    }
224
225    std::string filename = GlobalOutputManager.get_full_filename("_" + in_filename + ".pvtu");
```

```
  std::ofstream outputvtu(filename);
  for(unsigned int i = 0; i < generated_files.size(); i++) {
    generated_files[i] = "../" + generated_files[i];
  }
  Geometry.levels[0].inner_domain->data_out.write_pvtu_record(outputvtu, generated_files);
}
```

The documentation for this class was generated from the following files:

- Code/Hierarchy/LocalProblem.h
- Code/Hierarchy/LocalProblem.cpp

49 ModeManager Class Reference

Public Member Functions

- void **prepare_mode_in** ()
- void **prepare_mode_out** ()
- int **number_modes_in** ()
- int **number_modes_out** ()
- double **get_input_component** (int, Position, int)
- double **get_output_component** (int, Position, int)
- void **load** ()

49.1 Detailed Description

Definition at line 16 of file ModeManager.h.

The documentation for this class was generated from the following files:

- Code/GlobalObjects/ModeManager.h
- Code/GlobalObjects/ModeManager.cpp

50 MPICommunicator Class Reference

Utility class that provides additional information about the MPI setup on the level.

`#include <MPICommunicator.h>`

Public Member Functions

- std::pair< bool, unsigned int > get_neighbor_for_interface (Direction in_direction)

  Get the neighbor for interface For the provided surface, this function computes the MPI rank of the neighbor and if it exists.
- void initialize ()

  Initializes this object by computing the level communicators.
- void destroy_comms ()

This is used to free up some space and is just in general a good practice.

Public Attributes

- std::vector< MPI_Comm > **communicators_by_level**
- std::vector< unsigned int > **rank_on_level**

50.1 Detailed Description

Utility class that provides additional information about the MPI setup on the level.

This object wraps all information about communicators on all levels, i.e.which MPI_COMM to use on which level, ranks of this process on all levels and provides some useful functions like computing the neightbor MPI ranks by interface id.

Definition at line 20 of file MPICommunicator.h.

50.2 Member Function Documentation

get_neighbor_for_interface()

```
std::pair< bool, unsigned int > MPICommunicator::get_neighbor_for_interface (
              Direction in_direction )
```

Get the neighbor for interface For the provided surface, this function computes the MPI rank of the neighbor and if it exists.

Parameters

in_direction	The direction to check in.

Returns

std::pair<bool, unsigned int> First is true, if there is a neighbor in this direction. Second is the global MPI_rank of the neighbor.

Definition at line 53 of file MPICommunicator.cpp.

```
                              {
    std::pair<bool, unsigned int> ret(true, 0);
    switch (in_direction) {
    case Direction::MinusX:
      if (GlobalParams.Index_in_x_direction == 0) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank - 1;
      }
      break;
    case Direction::PlusX:
      if (GlobalParams.Index_in_x_direction
          == GlobalParams.Blocks_in_x_direction - 1) {
        ret.first = false;
      } else {
        ret.second = GlobalParams.MPI_Rank + 1;
```

```
    }
    break;
  case Direction::MinusY:
    if (GlobalParams.Index_in_y_direction == 0) {
      ret.first = false;
    } else {
      ret.second = GlobalParams.MPI_Rank - GlobalParams.Blocks_in_y_direction;
    }
    break;
  case Direction::PlusY:
    if (GlobalParams.Index_in_y_direction == GlobalParams.Blocks_in_y_direction - 1) {
      ret.first = false;
    } else {
      ret.second = GlobalParams.MPI_Rank + GlobalParams.Blocks_in_y_direction;
    }
    break;
  case Direction::MinusZ:
    if (GlobalParams.Index_in_z_direction == 0) {
      ret.first = false;
    } else {
      ret.second = GlobalParams.MPI_Rank - (GlobalParams.Blocks_in_x_direction *
    GlobalParams.Blocks_in_y_direction);
    }
    break;
  case Direction::PlusZ:
    if (GlobalParams.Index_in_z_direction
        == GlobalParams.Blocks_in_z_direction - 1) {
      ret.first = false;
    } else {
      ret.second = GlobalParams.MPI_Rank
          + (GlobalParams.Blocks_in_x_direction
              * GlobalParams.Blocks_in_y_direction);
    }
    break;
  }
  return ret;
}
```

The documentation for this class was generated from the following files:

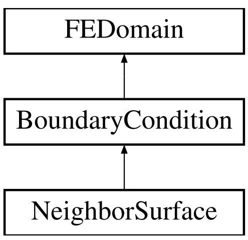
- Code/Hierarchy/MPICommunicator.h
- Code/Hierarchy/MPICommunicator.cpp

51 NeighborSurface Class Reference

For non-local problem, these interfaces are ones, that connect two inner domains and handle the communication between the two as well as the adjacent boundaries. This matrix has no effect for the assembly of system matrices since these boundaries have no own dofs. This object mainly communicates dof indices during the initialization phase.

`#include <NeighborSurface.h>`

Inheritance diagram for NeighborSurface:



Public Member Functions

- **NeighborSurface** (unsigned int in_bid, unsigned int in_level)
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix ∗matrix, NumericVectorDistributed ∗rhs, Constraints ∗constraints) override

  Does nothing, only fulfills the interface.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗in_constriants) override

  Does nothing, only fulfills the interface.
- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

  Does nothing, alwys returns false since this function is only there to fulfill the interface of boundary condition.
- void initialize () override

  Initializes the datastructures.
- void set_mesh_boundary_ids ()

  sets boundary ids on the surface triangulation.
- auto get_dof_association () -> std::vector< InterfaceDofData > override

  Fulfills the boundary condition interface.
- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

  Fulfills the boundary condition interface.
- std::string output_results (const dealii::Vector< ComplexNumber > &solution, std::string filename) override

  Does nothing in this class.
- DofCount compute_n_locally_owned_dofs () override

  Computes the number of locally owned dofs.
- DofCount compute_n_locally_active_dofs () override

  Computes the number of locally active dofs.
- void determine_non_owned_dofs () override

  Prepares internal datastructures for dof numbering On this class, however, this function does nothing since objects of this type own no dofs.
- void finish_dof_index_initialization () override

  Interfaces of this type always have a neighbor.
- void distribute_dof_indices ()

  Distributes the dofs indices to the inner domain and all neighbors.
- void send ()

  Sends the own dofs to the partner process.
- void receive ()

  Receives the dof numbers from the partner process.
- void prepare_dofs ()

Before the dofs can be exchanged, the boundary has to determine which the local dofs actually are.

Public Attributes

- const bool **is_lower_interface**
- std::array< std::set< unsigned int >, 6 > **edge_ids_by_boundary_id**
- std::array< std::set< unsigned int >, 6 > **face_ids_by_boundary_id**
- std::array< std::vector< InterfaceDofData >, 6 > **dof_indices_by_boundary_id**
- std::array< std::vector< unsigned int >, 6 > **boundary_dofs**
- std::vector< unsigned int > **inner_dofs**
- std::vector< unsigned int > **global_indices**
- unsigned int **n_dofs**
- bool **dofs_prepared**

51.1 Detailed Description

For non-local problem, these interfaces are ones, that connect two inner domains and handle the communication between the two as well as the adjacent boundaries. This matrix has no effect for the assembly of system matrices since these boundaries have no own dofs. This object mainly communicates dof indices during the initialization phase.

Definition at line 25 of file NeighborSurface.h.

51.2 Member Function Documentation

compute_n_locally_active_dofs()

`DofCount NeighborSurface::compute_n_locally_active_dofs ( )  [override], [virtual]`

Computes the number of locally active dofs.

Returns

DofCount number of locally active dofs.

Implements FEDomain.

Definition at line 75 of file NeighborSurface.cpp.

```
75                                                   {
76     return 0;
77 }
```

compute_n_locally_owned_dofs()

`DofCount NeighborSurface::compute_n_locally_owned_dofs ( )  [override], [virtual]` Computes the number of locally owned dofs.

Returns

DofCount number of locally owned dofs.

Implements FEDomain.

Definition at line 71 of file NeighborSurface.cpp.
```
71                                                      {
72     return 0;
73 }
```

fill_matrix()

```
void NeighborSurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints )  [override], [virtual]
```

Does nothing, only fulfills the interface.

Parameters

matrix	Matrix to fill.
rhs	Rhs to fill.
constraints	Constraints to condense.

Implements BoundaryCondition.

Definition at line 31 of file NeighborSurface.cpp.
```
31
                  {
32     matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
       required.
33     // Nothing to do here, work happens on neighbor process.
34 }
```

fill_sparsity_pattern()

```
void NeighborSurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * in_constriants )  [override], [virtual]
```

Does nothing, only fulfills the interface.

Parameters

in_dsp	Sparsity pattern to use
in_constriants	Constraints to use

Implements BoundaryCondition.

Definition at line 68 of file NeighborSurface.cpp.

```
68                                                                                                   {
69 }
```

finish_dof_index_initialization()

```
void NeighborSurface::finish_dof_index_initialization ( )  [override], [virtual]
```

Interfaces of this type always have a neighbor.

This function exchanges the data. For example, for normal sweeping in z direction, if there are 2 blocks, 0 and 1 then they share one interface. Surface 5 on 0 and 4 at 1. On both these Surfaces, there are boundary conditions of type neighbor and block 1 needs to number the surface dofs with the same numbers as 0 does so the matrix they assemble together. To fulfill this purpose, they retreive the local numbering of the surface dofs from the inner domain and then exchange it, or, more precisely it is sent up. The lower process sends this data to the higher, because the lower process owns the dofs.

Reimplemented from BoundaryCondition.

Definition at line 83 of file NeighborSurface.cpp.

```
83                                                          {
84      prepare_dofs();
85      if(is_lower_interface) {
86          receive();
87      } else {
88          send();
89      }
90 }
```

References prepare_dofs(), receive(), and send().

get_dof_association()

```
std::vector< InterfaceDofData > NeighborSurface::get_dof_association ( ) -> std::vector<InterfaceDofData>
[override], [virtual]
```

Fulfills the boundary condition interface.

For NeighborSurface this function returns the return value from InnerDomain::get_dof_association.

Returns

std::vector<InterfaceDofData> a vector of dofs at the interface.

Implements BoundaryCondition.

Definition at line 44 of file NeighborSurface.cpp.

```
44                                                                        {
45      std::vector<InterfaceDofData> dof_indices =
        Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
46      for(unsigned int i = 0; i < dof_indices.size(); i++) {
47          dof_indices[i].index = inner_dofs[i];
48      }
49      return dof_indices;
50 }
```

get_dof_association_by_boundary_id()

```
std::vector< InterfaceDofData > NeighborSurface::get_dof_association_by_boundary_id (
            BoundaryId in_boundary_id ) -> std::vector<InterfaceDofData>  [override], [virtual]
```

Fulfills the boundary condition interface.

This function returns either the surface dofs from the inner domain or one of the adjacent interfaces to this one.

Parameters

in_boundary_id	Boundary to search on.

Returns

std::vector<InterfaceDofData> Vector of all the dofs at the surface

Implements BoundaryCondition.

Definition at line 52 of file NeighborSurface.cpp.

```
52
      {
53      std::vector<InterfaceDofData> own_dof_indices;
54      for(unsigned int i = 0; i < boundary_dofs[in_boundary_id].size(); i++) {
55          InterfaceDofData idd;
56          idd.order = 0;
57          idd.base_point = {0,0,0};
58          idd.index = boundary_dofs[in_boundary_id][i];
59          own_dof_indices.push_back(idd);
60      }
61      return own_dof_indices;
62 }
```

is_point_at_boundary()

```
bool NeighborSurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid )  [override], [virtual]
```

Does nothing, alwys returns false since this function is only there to fulfill the interface of boundary condition.

Parameters

in_p	
in_bid	

Returns

true

false

Implements BoundaryCondition.

Definition at line 36 of file NeighborSurface.cpp.

```
36                                                         {
37     return false;
38 }
```

output_results()

```
std::string NeighborSurface::output_results (
            const dealii::Vector< ComplexNumber > & solution,
            std::string filename )  [override], [virtual]
```

Does nothing in this class.

Parameters

solution	The solution to be evaluated
filename	The name of the file to write the solution to

Returns

std::string filename

Implements BoundaryCondition.

Definition at line 64 of file NeighborSurface.cpp.

```
64                                                                             {
65     return "";
66 }
```

prepare_dofs()

```
void NeighborSurface::prepare_dofs ( )
```

Before the dofs can be exchanged, the boundary has to determine which the local dofs actually are.

Not all dofs on the surface are necessariy locally owned by the inner domain - they could belong to another prcess via another surface for example. This is an important action during the distribution of dof indices.

Definition at line 117 of file NeighborSurface.cpp.

```
117                                 {
118     // For the lower process, i.e. the one where this layer is an upper boundary, I set the correct
       indices here. For the other process, I use the same code, but basically only calculate n_dofs,
       because the value arrays will be filled later by receive.
119     std::vector<InterfaceDofData> temp =
       Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
120     n_dofs = 0;
121     inner_dofs.resize(temp.size());
```

```
    for(unsigned int i = 0; i < temp.size(); i++) {
        inner_dofs[i] = Geometry.levels[level].inner_domain->global_index_mapping[temp[i].index];
    }
    n_dofs += temp.size();
    for(unsigned int surf = 0; surf < 6; surf++) {
        if(surf != b_id && !are_opposing_sites(surf, b_id)) {
            if(Geometry.levels[level].surface_type[surf] == SurfaceType::ABC_SURFACE) {
                boundary_dofs[surf] =
   Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id);
                n_dofs += boundary_dofs[surf].size();
            }
        }
    }
    global_indices.resize(n_dofs);
    dofs_prepared = true;
}
```

Referenced by finish_dof_index_initialization().

The documentation for this class was generated from the following files:

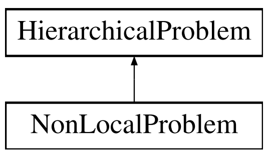
- Code/BoundaryCondition/NeighborSurface.h
- Code/BoundaryCondition/NeighborSurface.cpp

52 NonLocalProblem Class Reference

The NonLocalProblem class is part of the sweeping preconditioner hierarchy.

`#include <NonLocalProblem.h>`

Inheritance diagram for NonLocalProblem:



Public Member Functions

- NonLocalProblem (unsigned int level)

  Construct a new Non Local Problem object using a level value as input.
- ~NonLocalProblem () override

  Destroy the Non Local Problem object This means deleting the matrix and locally owned dofs index array as well as the KSP object in PETSC.
- void prepare_sweeping_data ()

  Computes some basic information about the sweep like the number of processes in the sweeping direction as well as the own index in that direction.
- void assemble () override

  Calls assemble on the InnerProblem and the boundary methods.
- void solve () override

  Solves using a GMRES solver with a sweeping preconditioner.
- void solve_adjoint () override

Similar to solve() but uses the adjoint solution for the output of the solution.

- void apply_sweep (Vec x_in, Vec x_out)

  Cor function of the sweeping preconditioner.

- void init_solver_and_preconditioner ()

  Prepares the PETSC objects required for the computation.

- void initialize () override

  Recursive.

- void initialize_index_sets () override

  Part of the initialization hierarchy.

- void reinit () override

  Builds constraints and sparsity pattern, then initializes the matrix and some cached data for faster data access.

- void compute_solver_factorization () override

  Recursive.

- void reinit_rhs () override

  Prepare the data structure which stores the right hand side vector.

- void S_inv (NumericVectorDistributed ∗src, NumericVectorDistributed ∗dst)

  Applies the operator S^{-1} to the provided src vector and returns the result in dst.

- auto set_x_out_from_u (Vec x_out) -> void

  Set the x out from u object We use different data types for computation in our own code then the somewhat clunky PETSC data types.

- std::string output_results ()

  Writes output files about the run on this level.

- void write_multifile_output (const std::string &filename, bool apply_coordinate_transform) override

  Generates actual output files about the current levels solution.

- void communicate_external_dsp (DynamicSparsityPattern ∗in_dsp)

  Exchange non-zero entries of the system matrix across neighboring processes.

- void make_sparsity_pattern () override

  Determines the non-zero entries of the system matrix and prepares a sparsity pattern object that stores this information for efficient memory allocation of the matrices.

- void set_u_from_vec_object (Vec in_v)

  Turns the input PETSC vector, the sweeping preconditioner should be applied to into a data structure that works well in deal.II.

- void set_vector_from_child_solution (NumericVectorDistributed ∗vec)

  Copies the solution of a child solver run up one hierarchy level.

- void set_child_rhs_from_vector (NumericVectorDistributed ∗)

  Copies a rhs vector down to the child vector befor calling solve on it.

- void print_vector_norm (NumericVectorDistributed ∗vec, std::string marker)

  Outputs the L2 norm of a provided vector.

- void perform_downward_sweep ()

  Performs the first half of the sweeping preconditioner.
- void perform_upward_sweep ()

  Performs the second half of the sweeping preconditioner.
- void complex_pml_domain_matching (BoundaryId in_bid)

  PML domains are sometimes different across the hierarchy.
- void register_dof_copy_pair (DofNumber own_index, DofNumber child_index)

  Used by complex_pml_domain_matching to register a degree of freedom that has the index own_index on this level and child_index in the child.
- ComplexNumber compute_signal_strength_of_solution ()

  Computes how strong the signal is on the output connector.
- void update_shared_solution_vector ()

  Not all locally active dofs (dofs that couple to locally owned ones) are locally owned.
- FEErrorStruct compute_global_errors (dealii::LinearAlgebra::distributed::Vector< ComplexNumber > *in_solution)

  Computes the L2 error of the provided vector solution agains a theoretical solution of the current problem.
- void update_convergence_criterion (double last_residual) override

  To be able to abort early on child solvers, we need to store the current residual on the current level.
- unsigned int compute_global_solve_counter () override

  Adds up the number of solver calls on the current level.
- void reinit_all_vectors ()

  Reinits all vectors on the current vector.
- unsigned int n_total_cells ()

  Computes the number of cells of the local part of the current problem and then adds these valus for all processes in the current sweep.
- double compute_h ()

  Computes the mesh constant of the local level problem.
- unsigned int compute_total_number_of_dofs ()

  Computes the total number of dofs on the current level (not only the locally owned part).
- std::vector< std::vector< ComplexNumber > > evaluate_solution_at (std::vector< Position > locations)

  Computes the E-field evaluation at all the positions in the input vector and returns a vector of the same length with the values.
- void empty_memory () override

  Reduces the memory consumption of local data structures to save memory once compuataions are done.
- std::vector< double > compute_shape_gradient () override

  Computes the shape gradient contributions of this process.
- void set_rhs_for_adjoint_problem ()

  Set the rhs for the computation of the adjoint state.

Additional Inherited Members

52.1 Detailed Description

The NonLocalProblem class is part of the sweeping preconditioner hierarchy.

It assembles a system-matrix and right-hand side and solves it using a GMRES solver. It also handles all the communication required to perform that task and assembles sparsity patterns.

Definition at line 32 of file NonLocalProblem.h.

52.2 Constructor & Destructor Documentation

NonLocalProblem()

```
NonLocalProblem::NonLocalProblem (
            unsigned int level )
```

Construct a new Non Local Problem object using a level value as input.

The constructor of this class actually performs several tasks: Initialize the solver control (which performs convergence tests), start initialization of the remaining objects of the sweeping hierarchy (Nonlocal-Problem(3) calls NonlocalProblem(2) calls NonLocalProblem(1) calls LocalProeblm()). Additionally, it determines the correct sweeping direction and initializes cached values of neighbors and the matrix. Next it prepares the locally active dof set and builds an output object for residuals of the own GMRES solver.

Parameters

level	

Definition at line 89 of file NonLocalProblem.cpp.

```
89                                                      :
90   HierarchicalProblem(level, static_cast<SweepingDirection> (2 + GlobalParams.Sweeping_Level - level)),
91   sc(GlobalParams.GMRES_max_steps, GlobalParams.Solver_Precision, true, true)
92 {
93   sweeping_direction = get_sweeping_direction_for_level(level);
94   if(level > 1) {
95     child = new NonLocalProblem(level - 1);
96   } else {
97     child = new LocalProblem();
98   }
99
100   prepare_sweeping_data();
101
102   matrix = new dealii::PETScWrappers::MPI::SparseMatrix();
103
104   locally_active_dofs = dealii::IndexSet(Geometry.levels[level].n_total_level_dofs);
105   for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->global_index_mapping.size(); i++) {
106     locally_active_dofs.add_index(Geometry.levels[level].inner_domain->global_index_mapping[i]);
107   }
108
109   for(unsigned int surf = 0; surf < 6; surf++) {
110     Geometry.levels[level].surfaces[surf]->print_dof_validation();
111   }
112   for(unsigned int surf = 0; surf < 6; surf++) {
113     for(unsigned int i = 0; i < Geometry.levels[level].surfaces[surf]->global_index_mapping.size(); i++)
      {
```

```
      unsigned int global_index = Geometry.levels[level].surfaces[surf]->global_index_mapping[i];
      locally_active_dofs.add_index(global_index);
    }
  }
  n_locally_active_dofs = locally_active_dofs.n_elements();
  residual_output = new ResidualOutputGenerator("ConvergenceHistoryLevel"+std::to_string(level),
   "Convergence History on level " + std::to_string(level), total_rank_in_sweep, level ,
   parent_sweeping_rank);
}
```

52.3 Member Function Documentation

apply_sweep()

```
void NonLocalProblem::apply_sweep (
            Vec x_in,
            Vec x_out )
```

Cor function of the sweeping preconditioner.

Applies the preconditioner to an input vector and returns the result in the second argument

This function has been refactored to be easier to read. This formulation is in line with the algorithm formulations in the dissertation documents.

Parameters

x_in	The vector the preconditioner should be applied to.
x_out	The vector storing the result.

Definition at line 313 of file NonLocalProblem.cpp.

```
313                                                     {
314   set_u_from_vec_object(b_in);
315   perform_downward_sweep();
316   perform_upward_sweep();
317   set_x_out_from_u(u_out);
318 }
```

References perform_downward_sweep(), perform_upward_sweep(), set_u_from_vec_object(), and set_x_out_from_u(

assemble()

```
void NonLocalProblem::assemble ( ) [override], [virtual]
```

Calls assemble on the InnerProblem and the boundary methods.

Steps: First reset the system matrix and rhs to zero (for the optimization cases). Then start a timer. Call assemble_systeem on the InnerDomain and fill_matrix on the boundary contributions. Then stop the timer. Finally compress the datastructures and update the PETSC ksp object to recognize the new operator.

Implements HierarchicalProblem.

Definition at line 238 of file NonLocalProblem.cpp.

```
238                             {
239   matrix->operator=(0);
240   rhs = 0;
241   matrix->compress(dealii::VectorOperation::insert);
242   rhs.compress(dealii::VectorOperation::insert);
243   print_info("NonLocalProblem::assemble", "Begin assembly");
244   GlobalTimerManager.switch_context("Assemble", level);
245   Timer timer;
246   timer.start();
247   Geometry.levels[level].inner_domain->assemble_system(&constraints, matrix, &rhs);
248   print_info("NonLocalProblem::assemble", "Inner assembly done. Assembling boundary method
       contributions.");
249   for(unsigned int i = 0; i< 6; i++) {
250       Geometry.levels[level].surfaces[i]->fill_matrix(matrix, &rhs, &constraints);
251   }
252   timer.stop();
253   print_info("NonLocalProblem::assemble", "Compress matrix.");
254   matrix->compress(dealii::VectorOperation::add);
255   rhs.compress(dealii::VectorOperation::add);
256   print_info("NonLocalProblem::assemble", "Assemble child.");
257   child->assemble();
258   print_info("NonLocalProblem::assemble", "Compress vectors.");
259   solution.compress(dealii::VectorOperation::add);
260   rhs.compress(VectorOperation::add);
261   // constraints.distribute(solution);
262   print_info("NonLocalProblem::assemble", "End assembly.");
263   KSPSetOperators(ksp, *matrix, *matrix);
264   GlobalTimerManager.leave_context(level);
265 }
```

Referenced by OptimizationRun::solve_main_problem().

communicate_external_dsp()

```
void NonLocalProblem::communicate_external_dsp (
              DynamicSparsityPattern * in_dsp )
```

Exchange non-zero entries of the system matrix across neighboring processes.

This is an important function and reasonably complex. However, it mainly handles the exchange of data in the sparsity pattern and is not mathematical in nature.

Parameters

in_dsp	The dsp to fill.

Definition at line 587 of file NonLocalProblem.cpp.

```
587                                                                                {
588   std::vector<std::vector<unsigned int» rows, cols;
589   for(unsigned int i = 0; i < n_procs_in_sweep; i++) {
590     rows.emplace_back();
591     cols.emplace_back();
592   }
593   for(auto it = in_dsp->begin(); it != in_dsp->end(); it++) {
594     if(!own_dofs.is_element(it->row())) {
595       for(unsigned int proc = 0; proc < n_procs_in_sweep; proc++) {
596         if(Geometry.levels[level].dof_distribution[proc].is_element(it->row())) {
597           rows[proc].push_back(it->row());
598           cols[proc].push_back(it->column());
599         }
600       }
601     }
602   }
```

```
std::vector<unsigned int> entries_by_proc;
entries_by_proc.resize(n_procs_in_sweep);
for(unsigned int i = 0; i < n_procs_in_sweep; i++) {
  entries_by_proc[i] = rows[i].size();
}
std::vector<unsigned int> recv_buffer;
recv_buffer.resize(n_procs_in_sweep);
MPI_Alltoall(entries_by_proc.data(), 1, MPI_UNSIGNED, recv_buffer.data(), 1, MPI_UNSIGNED,
 GlobalMPI.communicators_by_level[level]);
MPI_Status recv_status;
unsigned int receiving_neighbors = 0;
std::vector<std::vector<unsigned int» received_rows;
std::vector<std::vector<unsigned int» received_cols;
unsigned int sent_neighbors = 0;
std::vector<std::vector<unsigned int» sent_rows;
std::vector<std::vector<unsigned int» sent_cols;
for(unsigned int other_proc = 0; other_proc < n_procs_in_sweep; other_proc++) {
  if(other_proc != total_rank_in_sweep) {
    if(recv_buffer[other_proc] != 0 || entries_by_proc[other_proc] != 0) {
      if(entries_by_proc[other_proc] > 0) {
        const unsigned int n_loc_dofs = entries_by_proc[other_proc];
        sent_rows.emplace_back(n_loc_dofs);
        sent_cols.emplace_back(n_loc_dofs);
        for(unsigned int i = 0; i < n_loc_dofs; i++) {
          sent_rows[sent_neighbors][i] = rows[other_proc][i];
          sent_cols[sent_neighbors][i] = cols[other_proc][i];
        }
        sent_neighbors++;
      }
    }
  }
}
sent_neighbors = 0;
for(unsigned int other_proc = 0; other_proc < n_procs_in_sweep; other_proc++) {
  if(other_proc != total_rank_in_sweep) {
    if(recv_buffer[other_proc] != 0 || entries_by_proc[other_proc] != 0) {
      if(total_rank_in_sweep < other_proc) {
        // Send then receive
        if(entries_by_proc[other_proc] > 0) {
          const unsigned int n_loc_dofs = entries_by_proc[other_proc];
          MPI_Send(sent_rows[sent_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc,
 GlobalParams.MPI_Rank, GlobalMPI.communicators_by_level[level]);
          MPI_Send(sent_cols[sent_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc,
 GlobalParams.MPI_Rank, GlobalMPI.communicators_by_level[level]);
          sent_neighbors++;
        }
        // receive part
        if(recv_buffer[other_proc] > 0) {
          // There is something to receive
          const unsigned int n_loc_dofs = recv_buffer[other_proc];
          received_rows.emplace_back(n_loc_dofs);
          received_cols.emplace_back(n_loc_dofs);
          MPI_Recv(received_rows[receiving_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc,
 MPI_ANY_TAG, GlobalMPI.communicators_by_level[level], &recv_status);
          MPI_Recv(received_cols[receiving_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc,
 MPI_ANY_TAG, GlobalMPI.communicators_by_level[level], &recv_status);
          receiving_neighbors ++;
        }
      } else {
        // Receive then send
        if(recv_buffer[other_proc] > 0) {
          // There is something to receive
          const unsigned int n_loc_dofs = recv_buffer[other_proc];
          received_rows.emplace_back(n_loc_dofs);
          received_cols.emplace_back(n_loc_dofs);
          MPI_Recv(received_rows[receiving_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc,
 MPI_ANY_TAG, GlobalMPI.communicators_by_level[level], &recv_status);
          MPI_Recv(received_cols[receiving_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc,
 MPI_ANY_TAG, GlobalMPI.communicators_by_level[level], &recv_status);
          receiving_neighbors ++;
        }

        if(entries_by_proc[other_proc] > 0) {
          const unsigned int n_loc_dofs = entries_by_proc[other_proc];
```

```
          MPI_Send(sent_rows[sent_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc, GlobalParams.MPI_Rank, GlobalMPI.communicators_by_level[level]);
          MPI_Send(sent_cols[sent_neighbors].data(), n_loc_dofs, MPI_UNSIGNED, other_proc, GlobalParams.MPI_Rank, GlobalMPI.communicators_by_level[level]);
          sent_neighbors++;
        }
      }
    }
  }
}
for(unsigned int j = 0; j < receiving_neighbors; j++) {
  for(unsigned int i = 0; i < received_cols[j].size(); i++) {
    in_dsp->add(received_rows[j][i], received_cols[j][i]);
  }
}
}
```

complex_pml_domain_matching()

```
void NonLocalProblem::complex_pml_domain_matching (
            BoundaryId in_bid )
```

PML domains are sometimes different across the hierarchy.

Whenever we copy a vector up or down we have to match the indices correctly.

This function prepares index pairs across the hierarchy that reference the same dof on different levels. It only performs this task for one boundary and builds the mapping for dofs on the current level and the immediate child.

The data is stored in the vector_copy_own_indices, vector_copy_child_indices and vector_copy_array. These datastructures are always used when we call functions like set_child_rhs_from_vector.

Parameters

in_bid	The surface to perform this task on.

Definition at line 418 of file NonLocalProblem.cpp.

```
418                                                                     {
419   // always more dofs on the lower level
420   dealii::IndexSet lower_is (Geometry.levels[level-1].n_total_level_dofs);
421   dealii::IndexSet upper_is (Geometry.levels[level].n_total_level_dofs);
422   auto higher_cell = Geometry.levels[level].surfaces[in_bid]->dof_handler.begin();
423   auto lower_cell = Geometry.levels[level-1].surfaces[in_bid]->dof_handler.begin();
424   auto higher_end = Geometry.levels[level].surfaces[in_bid]->dof_handler.end();
425   auto lower_end = Geometry.levels[level-1].surfaces[in_bid]->dof_handler.end();
426   while(higher_cell != higher_end) {
427     bool found = true;
428     // first find the same cell in the child
429     if(! ((higher_cell->center() - lower_cell->center()).norm() < FLOATING_PRECISION)) {
430       while((higher_cell->center() - lower_cell->center()).norm() > FLOATING_PRECISION && lower_cell != lower_end) {
431         lower_cell++;
432       }
433       if(lower_cell == lower_end) {
434         lower_cell = Geometry.levels[level-1].surfaces[in_bid]->dof_handler.begin();
435       }
436       while((higher_cell->center() - lower_cell->center()).norm() > FLOATING_PRECISION && lower_cell != lower_end) {
437         lower_cell++;
438       }
439       if(lower_cell == lower_end) {
440         found = false;
```

```
        std::cout « "ERROR IN COMPLEX PML DOMAIN MATCHING" « std::endl;
      }
    }
    if(found) {
      // lower_cell and higher_cell point to the same cell on two different levels. Match the dofs.
      const unsigned int n_dofs_per_cell =
    Geometry.levels[level].surfaces[in_bid]->dof_handler.get_fe().dofs_per_cell;
      std::vector<DofNumber> lower_dofs(n_dofs_per_cell);
      std::vector<DofNumber> upper_dofs(n_dofs_per_cell);
      lower_cell->get_dof_indices(lower_dofs);
      std::sort(lower_dofs.begin(), lower_dofs.end());
      higher_cell->get_dof_indices(upper_dofs);
      std::sort(upper_dofs.begin(), upper_dofs.end());
      for(unsigned int i = 0; i < n_dofs_per_cell; i++) {
        if(Geometry.levels[level].surfaces[in_bid]->is_dof_owned[upper_dofs[i]] &&
    Geometry.levels[level-1].surfaces[in_bid]->is_dof_owned[lower_dofs[i]]) {

    lower_is.add_index(Geometry.levels[level-1].surfaces[in_bid]->global_index_mapping[lower_dofs[i]]);

    upper_is.add_index(Geometry.levels[level].surfaces[in_bid]->global_index_mapping[upper_dofs[i]]);
        }
      }
    }
    lower_cell++;
    higher_cell++;
  }
  for(unsigned int i = 0; i < upper_is.n_elements(); i++) {
    register_dof_copy_pair(upper_is.nth_index_in_set(i), lower_is.nth_index_in_set(i));
  }
}
```

compute_global_errors()

FEErrorStruct NonLocalProblem::compute_global_errors (
    dealii::LinearAlgebra::distributed::Vector< ComplexNumber > * *in_solution*)

Computes the L2 error of the provided vector solution agains a theoretical solution of the current problem.

Parameters

in_solution	The solution vector.

Returns

FEErrorStruct A structure containing the L2 error.

Definition at line 796 of file NonLocalProblem.cpp.

```
796
                  {
797   FEErrorStruct errors = Geometry.levels[level].inner_domain->compute_errors(in_solution);
798   FEErrorStruct ret;
799   ret.L2 = Utilities::MPI::sum(errors.L2, GlobalMPI.communicators_by_level[level]);
800   ret.Linfty = Utilities::MPI::max(errors.Linfty, GlobalMPI.communicators_by_level[level]);
801   return ret;
802 }
```

compute_global_solve_counter()

unsigned int NonLocalProblem::compute_global_solve_counter ( ) [override], [virtual] Adds up the number of solver calls on the current level.

Returns

unsigned int How often the solver was called on this level.

Reimplemented from HierarchicalProblem.

Definition at line 817 of file NonLocalProblem.cpp.

```
817                                                    {
818   unsigned int contribution = 0;
819   if(total_rank_in_sweep == 0) {
820     contribution = solve_counter;
821   }
822   return Utilities::MPI::sum(contribution, MPI_COMM_WORLD);
823 }
```

compute_h()

`double NonLocalProblem::compute_h ( )`

Computes the mesh constant of the local level problem.

Returns

double Mesh size constant for the triangulation.

Definition at line 834 of file NonLocalProblem.cpp.

```
834                                {
835   double temp = Geometry.h_x;
836   temp = std::max(temp, Geometry.h_y);
837   temp = std::max(temp, Geometry.h_z);
838   return temp;
839 }
```

compute_shape_gradient()

`std::vector< double > NonLocalProblem::compute_shape_gradient ( )  [override], [virtual]`

Computes the shape gradient contributions of this process.

The i-th entry in this vector is the derivative of the loss functional by the i-th degree of freedom of the shape.

Returns

std::vector<double>

Reimplemented from HierarchicalProblem.

Definition at line 861 of file NonLocalProblem.cpp.

```
861                                                   {
862   print_info("NonLocalProblem::compute_shape_gradient", "Start");
863   const unsigned int n_shape_dofs = GlobalSpaceTransformation->n_free_dofs();
864   std::vector<double> ret(n_shape_dofs);
865   for(unsigned int i = 0; i < n_shape_dofs; i++) {
866     ret[i] = 0;
867   }
868
869   std::vector<FEAdjointEvaluation> field_evaluations;
```

```
  Timer timer1;
  timer1.start();

  NumericVectorLocal local_solution(Geometry.levels[level].inner_domain->n_locally_active_dofs);
  NumericVectorLocal local_adjoint(Geometry.levels[level].inner_domain->n_locally_active_dofs);
  update_shared_solution_vector();

  for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->n_locally_active_dofs; i++) {
    local_solution[i] = shared_solution[Geometry.levels[level].inner_domain->global_index_mapping[i]];
    local_adjoint[i] = shared_adjoint[Geometry.levels[level].inner_domain->global_index_mapping[i]];
  }

  field_evaluations = Geometry.levels[level].inner_domain->compute_local_shape_gradient_data(local_solution, local_adjoint);

  timer1.stop();
  print_info("NonLocalProblem::compute_shape_gradient", "Walltime: " + std::to_string(timer1.wall_time()) , LoggingLevel::PRODUCTION_ONE);

  // Now, I have the evalaution and the adjoint field stored for a set of positions in the array field_evaluations.
  for(unsigned int i = 0; i < field_evaluations.size(); i++) {
    for(unsigned int j = 0; j < n_shape_dofs; j++) {
      Tensor<2, 3, ComplexNumber> local_step_tensor = GlobalSpaceTransformation->get_Tensor_for_step(field_evaluations[i].x, j, 0.01);
      Tensor<2, 3, ComplexNumber> local_inverse_step_tensor = GlobalSpaceTransformation->get_inverse_Tensor_for_step(field_evaluations[i].x, j, 0.01);
      Tensor<1,3,ComplexNumber> local_adj = field_evaluations[i].adjoint_field;
      Tensor<1,3,ComplexNumber> local_adj_curl = field_evaluations[i].adjoint_field_curl;
      for(unsigned int k = 0; k < 3; k++) {
        local_adj[k].imag(- local_adj[k].imag());
        local_adj_curl[k].imag(- local_adj_curl[k].imag());
      }
      ComplexNumber change = (field_evaluations[i].primal_field_curl * local_inverse_step_tensor * local_adj_curl) - Geometry.eps_kappa_2(field_evaluations[i].x) * (field_evaluations[i].primal_field * local_step_tensor) * local_adj;
      const double delta = change.real();
      ret[j] += delta;
    }
  }
  for(unsigned int i = 0; i <n_shape_dofs; i++) {
    ret[i] = dealii::Utilities::MPI::sum(ret[i], MPI_COMM_WORLD);
  }
  print_info("NonLocalProblem::compute_shape_gradient", "End");
  return ret;
}
```

compute_signal_strength_of_solution()

`ComplexNumber NonLocalProblem::compute_signal_strength_of_solution ( )`

Computes how strong the signal is on the output connector.

Returns

ComplexNumber Phase and amplitude of the signal.

Definition at line 785 of file NonLocalProblem.cpp.

```
{
  print_info("NonLocalProblem::compute_signal_strength_of_solution", "Start");
  update_shared_solution_vector();
  ComplexNumber integral = Geometry.levels[level].inner_domain->compute_signal_strength(&shared_solution);
  ComplexNumber base = Geometry.levels[level].inner_domain->compute_mode_strength();
  ComplexNumber integral_sum = dealii::Utilities::MPI::sum(integral, GlobalMPI.communicators_by_level[level]);
```

```
791   ComplexNumber mode_sum = dealii::Utilities::MPI::sum(base, GlobalMPI.communicators_by_level[level]);
792   print_info("NonLocalProblem::compute_signal_strength_of_solution", "End");
793   return integral_sum / mode_sum;
794 }
```

Referenced by OptimizationRun::perform_step().

compute_solver_factorization()

`void NonLocalProblem::compute_solver_factorization ( ) [override], [virtual]`

Recursive.

This function only propagates to the child. On the lowest level (which is a LocalProblem), this will prepare the direct solver factorization.

Implements HierarchicalProblem.

Definition at line 485 of file NonLocalProblem.cpp.

```
485                                                       {
486   child->compute_solver_factorization();
487 }
```

References HierarchicalProblem::compute_solver_factorization().

Referenced by OptimizationRun::solve_main_problem().

compute_total_number_of_dofs()

`unsigned int NonLocalProblem::compute_total_number_of_dofs ( )`

Computes the total number of dofs on the current level (not only the locally owned part).

Returns

unsigned int Number of dofs on this level.

Definition at line 841 of file NonLocalProblem.cpp.

```
841                                                               {
842   return Geometry.levels[level].n_total_level_dofs;
843 }
```

empty_memory()

`void NonLocalProblem::empty_memory ( ) [override], [virtual]`

Reduces the memory consumption of local data structures to save memory once compuataions are done.

This deletes, among other things, the factorization in direct solvers.

Reimplemented from HierarchicalProblem.

Definition at line 854 of file NonLocalProblem.cpp.

```
854                                     {
855   matrix->clear();
```

```
856   KSPReset(ksp);
857   child->empty_memory();
858 }
```

References HierarchicalProblem::empty_memory().

evaluate_solution_at()

```
std::vector< std::vector< ComplexNumber > > NonLocalProblem::evaluate_solution_at (
            std::vector< Position > locations )
```

Computes the E-field evaluation at all the positions in the input vector and returns a vector of the same length with the values.

Parameters

locations	A vector containing a set of positions that must be part of the local triangulation.

Returns

std::vector<std::vector<ComplexNumber>> Vector of e-field evaluations for the provided locations.

Definition at line 845 of file NonLocalProblem.cpp.

```
845                                                                                                  {
846   NumericVectorLocal local_solution(Geometry.levels[level].inner_domain->n_locally_active_dofs);
847   update_shared_solution_vector();
848   for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->n_locally_active_dofs; i++) {
849     local_solution[i] = shared_solution[Geometry.levels[level].inner_domain->global_index_mapping[i]];
850   }
851   return Geometry.levels[level].inner_domain->evaluate_at_positions(positions, local_solution);
852 }
```

init_solver_and_preconditioner()

```
void NonLocalProblem::init_solver_and_preconditioner ( )
```

Prepares the PETSC objects required for the computation.

This code relies on PETSC to perform the computationally expensive tasks. We use itterative solvers from this library. This function sets up the Krylov Space wrapper for the solvers (KSP) which is default for PETSC applications and also provides the preconditioner to the object. The NonLocalProblem object contains all required functions for the evalutaion of the preconditioner and the constructed preconditioner object (PC) simply references those (In detail: A Batch-Preconditioner is initialized which is a way of wrapping a function call and providing it as a preconditioner). Additionally, it sets the operator used in the solver to the system matrix constructed for the NonLocalProblem. In the next step it provides the individual solver with necessary data depending on its type. For example: For GMRES we set the restart parameter and the preconditioner side.

Definition at line 157 of file NonLocalProblem.cpp.

```
157                                               {
158   // dealii::PETScWrappers::PreconditionNone pc_none;
159   // pc_none.initialize(*matrix);
160   KSPCreate(GlobalMPI.communicators_by_level[level], &ksp);
```

```
  KSPGetPC(ksp, &pc);
  KSPSetOperators(ksp, *matrix, *matrix);
  if(GlobalParams.solver_type == SolverOptions::MINRES) {
    KSPSetType(ksp, KSPMINRES);
  }
  if(GlobalParams.solver_type == SolverOptions::GMRES) {
    KSPSetType(ksp, KSPGMRES);
    KSPGMRESSetRestart(ksp, GlobalParams.GMRES_max_steps);
    KSPSetPCSide(ksp, PCSide::PC_RIGHT);
  }
  if(GlobalParams.solver_type == SolverOptions::TFQMR) {
    KSPSetType(ksp, KSPTFQMR);
  }
  if(GlobalParams.solver_type == SolverOptions::BICGS) {
    KSPSetType(ksp, KSPBCGS);
  }
  if(GlobalParams.solver_type == SolverOptions::PCONLY) {
    KSPSetType(ksp, KSPRICHARDSON);
  }
  if(GlobalParams.solver_type == SolverOptions::S_CG) {
    KSPSetType(ksp, KSPCG);
  }

  PCSetType(pc, PCSHELL);
  pc_create(&shell, this);
  PCShellSetApply(pc,pc_apply);
  PCShellSetContext(pc, (void*) &shell);
  KSPSetPC(ksp, pc);
  // KSPSetConvergenceTest(ksp, &convergence_test, reinterpret_cast<void *>(&sc), nullptr);

  KSPMonitorSet(ksp, MonitorError, this, nullptr);
  KSPSetUp(ksp);
  KSPSetTolerances(ksp, 1e-10, GlobalParams.Solver_Precision, 1000, GlobalParams.GMRES_max_steps);
}
```

initialize()

`void NonLocalProblem::initialize ( ) [override], [virtual]`

Recursive.

Prepares all datastructures.

At the point of this function call, the NonLocalProblem object can access the dof distribution on the current level and we can therefore prepare vectors and matrices as well as sparsity patterns. The function also calls itself on the child level.

Implements HierarchicalProblem.

Definition at line 468 of file NonLocalProblem.cpp.

```
468                                {
469   GlobalTimerManager.switch_context("Initialize", level);
470   child->initialize();
471   initialize_index_sets();
472   reinit_all_vectors();
473   reinit();
474   init_solver_and_preconditioner();
475   GlobalTimerManager.leave_context(level);
476 }
```

initialize_index_sets()

`void NonLocalProblem::initialize_index_sets ( ) [override], [virtual]`

Part of the initialization hierarchy.

Sets the locally cached values of the owned dofs and prepares a petsc index array for efficient extraction of dof values from vectors.

Implements HierarchicalProblem.

Definition at line 478 of file NonLocalProblem.cpp.

```
478                                          {
479   own_dofs = Geometry.levels[level].dof_distribution[total_rank_in_sweep];
480   locally_owned_dofs_index_array = new PetscInt[own_dofs.n_elements()];
481   get_petsc_index_array_from_index_set(locally_owned_dofs_index_array, own_dofs);
482
483 }
```

n_total_cells()

`unsigned int NonLocalProblem::n_total_cells ( )`

Computes the number of cells of the local part of the current problem and then adds these valus for all processes in the current sweep.

Returns

unsigned int Number of cells on this level.

Definition at line 825 of file NonLocalProblem.cpp.

```
825                                          {
826   unsigned int local = Geometry.levels[level].inner_domain->triangulation.n_active_cells();
827   for(unsigned int i = 0; i < 6; i++) {
828     local += Geometry.levels[level].surfaces[i]->n_cells();
829   }
830   unsigned int ret = dealii::Utilities::MPI::sum(local, MPI_COMM_WORLD);
831   return ret;
832 }
```

output_results()

`std::string NonLocalProblem::output_results ( )`

Writes output files about the run on this level.

This calls another function which performs the actual writing of the output. This function mainly generates a vector of all locally active dofs (they might be stored on another process) and makes it available locally. It also logs signal strength and solver data.

Returns

std::string empty string in this case.

Definition at line 489 of file NonLocalProblem.cpp.

```
489                                        {
490   print_info("NonLocalProblem", "Start output results on level" + std::to_string(level));
491   print_solve_counter_list();
```

```
  update_shared_solution_vector();
  FEErrorStruct errors = compute_global_errors(&shared_solution);
  print_info("NonLocalProblem::output_results", "Errors: L2 = " + std::to_string(errors.L2) + " and
   Linfty  = " + std::to_string(errors.Linfty));
  write_multifile_output("solution", false);
  ComplexNumber signal_strength = compute_signal_strength_of_solution();
  print_info("NonLocalProblem::output_results", "Signal strength: " +
   std::to_string(std::abs(signal_strength)));
  if(GlobalParams.Output_transformed_solution) {
    write_multifile_output("transformed_solution", true);
  }
  print_info("NonLocalProblem", "End output results on level" + std::to_string(level));
  return "";
}
```

perform_downward_sweep()

`void NonLocalProblem::perform_downward_sweep ( )`

Performs the first half of the sweeping preconditioner.

The code looks more bloated than in the pseudo-code algorithm but most of it is just vector storage management.

Definition at line 729 of file NonLocalProblem.cpp.

```
729                                             {
730   for(int i = n_blocks_in_sweeping_direction - 1; i >= 0; i--) {
731     if((int)index_in_sweeping_direction == i) {
732       S_inv(&u, &dist_vector_1);
733     } else {
734       for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
735         dist_vector_1[own_dofs.nth_index_in_set(j)] = 0;
736       }
737     }
738     dist_vector_1.compress(VectorOperation::insert);
739     matrix->vmult(dist_vector_2, dist_vector_1);
740     if((int)index_in_sweeping_direction == i-1) {
741       for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
742         const unsigned int index = own_dofs.nth_index_in_set(j);
743         ComplexNumber current_value(u(index).real(), u(index).imag());
744         ComplexNumber delta(dist_vector_2[index].real(), dist_vector_2[index].imag());
745         u[index] = current_value - delta;
746       }
747     }
748     if((int)index_in_sweeping_direction == i) {
749       for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
750         const unsigned int index = own_dofs.nth_index_in_set(j);
751         u[index] = (ComplexNumber) dist_vector_1[index];
752       }
753     }
754     u.compress(VectorOperation::insert);
755   }
756 }
```

References S_inv().

Referenced by apply_sweep().

perform_upward_sweep()

`void NonLocalProblem::perform_upward_sweep ( )`

Performs the second half of the sweeping preconditioner.

The code looks more bloated than in the pseudo-code algorithm but most of it is just vector storage management.

Definition at line 758 of file NonLocalProblem.cpp.

```
758                               {
759   for(unsigned int i = 0; i < n_blocks_in_sweeping_direction-1; i++) {
760     if(index_in_sweeping_direction == i) {
761       for(unsigned int index = 0; index < own_dofs.n_elements(); index++) {
762         dist_vector_1[own_dofs.nth_index_in_set(index)] = (ComplexNumber) u[own_dofs.nth_index_in_set(index)];
763       }
764     } else {
765       for(unsigned int index = 0; index < own_dofs.n_elements(); index++) {
766         dist_vector_1[own_dofs.nth_index_in_set(index)] = 0;
767       }
768     }
769     dist_vector_1.compress(VectorOperation::insert);
770     matrix->Tvmult(dist_vector_2, dist_vector_1);
771 
772     if(index_in_sweeping_direction == i+1) {
773       S_inv(&dist_vector_2, &dist_vector_3);
774       for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
775         const unsigned int index = own_dofs.nth_index_in_set(j);
776         ComplexNumber current_value = u(index);
777         ComplexNumber delta = dist_vector_3[index];
778         u[index] = current_value - delta;
779       }
780     }
781     u.compress(VectorOperation::insert);
782   }
783 }
```

Referenced by apply_sweep().

print_vector_norm()

```
void NonLocalProblem::print_vector_norm (
              NumericVectorDistributed * vec,
              std::string marker )
```

Outputs the L2 norm of a provided vector.

Parameters

vec	The vector to measure
marker	A string marker that will be part of the output so it can be identified in the logs.

Definition at line 712 of file NonLocalProblem.cpp.

```
712                                                                       {
713   in_v->extract_subvector_to(vector_copy_own_indices, vector_copy_array);
714   double local_norm = 0.0;
715   double max = 0;
716   for(unsigned int i = 0; i < vector_copy_array.size(); i++) {
717     double local = std::abs(vector_copy_array[i])*std::abs(vector_copy_array[i]);
718     if(local > max) {
719       max = local;
720     }
721     local_norm += local;
722   }
723   local_norm = dealii::Utilities::MPI::sum(local_norm, GlobalMPI.communicators_by_level[level]);
```

```
if(GlobalParams.MPI_Rank == 0) {
  std::cout « marker « ": " « std::sqrt(local_norm) « std::endl;
}
}
```

register_dof_copy_pair()

```
void NonLocalProblem::register_dof_copy_pair (
            DofNumber own_index,
            DofNumber child_index )
```

Used by complex_pml_domain_matching to register a degree of freedom that has the index own_index on this level and child_index in the child.

Whenever a vector is copied between the child and this, the dof child_index on the child and own_index on this will have the same value.

Parameters

own_index	Index on this.
child_index	Index on the child.

Definition at line 412 of file NonLocalProblem.cpp.

```
412                                                                                  {
413   vector_copy_own_indices.push_back(own_index);
414   vector_copy_child_indeces.push_back(child_index);
415   vector_copy_array.push_back(ComplexNumber(0.0, 0.0));
416 }
```

reinit()

```
void NonLocalProblem::reinit ( )  [override], [virtual]
```

Builds constraints and sparsity pattern, then initializes the matrix and some cached data for faster data access.

Matrix initialization is a complex step for large runs because large memory consumtion is expected.

Implements HierarchicalProblem.

Definition at line 355 of file NonLocalProblem.cpp.

```
355                             {
356   print_info("Nonlocal reinit", "Reinit starting for level " + std::to_string(level));
357   MPI_Barrier(MPI_COMM_WORLD);
358   GlobalTimerManager.switch_context("Reinit", level);
359
360   make_constraints();
361
362   make_sparsity_pattern();
363   MPI_Barrier(MPI_COMM_WORLD);
364
365   if(GlobalParams.MPI_Rank == 0) std::cout « "Start reinit of rhs vector." « std::endl;
366
367   reinit_rhs();
368   MPI_Barrier(MPI_COMM_WORLD);
369
```

```
  if(GlobalParams.MPI_Rank == 0) std::cout « "Start reinit of system matrix." « std::endl;

  matrix->reinit(Geometry.levels[level].dof_distribution[total_rank_in_sweep],
   Geometry.levels[level].dof_distribution[total_rank_in_sweep], sp,
   GlobalMPI.communicators_by_level[level]);

  MPI_Barrier(MPI_COMM_WORLD);

  if(GlobalParams.MPI_Rank == 0) print_info("Nonlocal reinit", "Matrix initialized");

  for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->n_locally_active_dofs; i++) {
    if(Geometry.levels[level].inner_domain->is_dof_owned[i] &&
   Geometry.levels[level-1].inner_domain->is_dof_owned[i]) {
      vector_copy_own_indices.push_back(Geometry.levels[level].inner_domain->global_index_mapping[i]);

   vector_copy_child_indeces.push_back(Geometry.levels[level-1].inner_domain->global_index_mapping[i]);
      vector_copy_array.push_back(ComplexNumber(0.0, 0.0));
    }
  }
  for(unsigned int surf = 0; surf < 6; surf++) {
    if(Geometry.levels[level].surface_type[surf] == Geometry.levels[level-1].surface_type[surf]) {
      if(Geometry.levels[level].surfaces[surf]->dof_counter !=
   Geometry.levels[level-1].surfaces[surf]->dof_counter) {
        complex_pml_domain_matching(surf);
      } else {
        for(unsigned int i = 0; i < Geometry.levels[level].surfaces[surf]->n_locally_active_dofs; i++) {
          if(Geometry.levels[level].surfaces[surf]->is_dof_owned[i] &&
   Geometry.levels[level-1].surfaces[surf]->is_dof_owned[i]) {
            register_dof_copy_pair(Geometry.levels[level].surfaces[surf]->global_index_mapping[i],
   Geometry.levels[level-1].surfaces[surf]->global_index_mapping[i]);
          }
        }
      }
    }
  }
  GlobalTimerManager.leave_context(level);
  print_info("Nonlocal reinit", "Reinit done for level " + std::to_string(level));
}
```

S_inv()

```
void NonLocalProblem::S_inv (
            NumericVectorDistributed * src,
            NumericVectorDistributed * dst )
```

Applies the operator S^{-1} to the provided src vector and returns the result in dst.

This is the function call in the preconditioner that calls the solver of the child problem.

Parameters

src	The vector the child solver should be applied to.
dst	The vector to store the result in.

Definition at line 335 of file NonLocalProblem.cpp.

```
335                                                                                                 {
336   set_child_rhs_from_vector(src);
337   child->solve_with_timers_and_count();
338   set_vector_from_child_solution(dst);
339 }
```

References set_child_rhs_from_vector(), set_vector_from_child_solution(), and HierarchicalProblem::solve_with_time

Referenced by perform_downward_sweep().

set_u_from_vec_object()

```
void NonLocalProblem::set_u_from_vec_object (
            Vec in_v )
```

Turns the input PETSC vector, the sweeping preconditioner should be applied to into a data structure that works well in deal.II.

Parameters

in_v	The vector.

Definition at line 700 of file NonLocalProblem.cpp.

```
700                                                     {
701   const unsigned int n_loc_dofs = own_dofs.n_elements();
702   const ComplexNumber * pointer;
703   VecGetArrayRead(in_v, &pointer);
704   for(unsigned int i = 0; i < n_loc_dofs; i++) {
705     u[own_dofs.nth_index_in_set(i)] = *pointer;
706     pointer++;
707   }
708   VecRestoreArrayRead(in_v, &pointer);
709   u.compress(dealii::VectorOperation::insert);
710 }
```

Referenced by apply_sweep().

set_vector_from_child_solution()

```
void NonLocalProblem::set_vector_from_child_solution (
            NumericVectorDistributed * vec )
```

Copies the solution of a child solver run up one hierarchy level.

Parameters

vec	The vector to store the child solution in on this level.

Definition at line 341 of file NonLocalProblem.cpp.

```
341                                                                                      {
342   child->solution.extract_subvector_to(vector_copy_child_indeces, vector_copy_array);
343   in_u->set(vector_copy_own_indices, vector_copy_array);
344   //in_u->compress(VectorOperation::insert);
345 }
```

Referenced by S_inv().

set_x_out_from_u()

```
void NonLocalProblem::set_x_out_from_u (
            Vec x_out ) -> void
```

Set the x out from u object We use different data types for computation in our own code then the somewhat clunky PETSC data types.

Therefore, once we are done computing the output vector of the sweeping preconditioner application to an input vector in our own data-type, we have to update the provided output vector, which is a PETSC data structure. This function performs no math only copying of the vector to the appropirate output format.

Parameters

x_out	

Definition at line 320 of file NonLocalProblem.cpp.

```
320                                              {
321   ComplexNumber * values = new ComplexNumber[own_dofs.n_elements()];
322
323   u.extract_subvector_to(vector_copy_own_indices, vector_copy_array);
324
325   for(unsigned int i = 0; i < own_dofs.n_elements(); i++) {
326     values[i] = vector_copy_array[i];
327   }
328
329   VecSetValues(x_out, own_dofs.n_elements(), locally_owned_dofs_index_array, values, INSERT_VALUES);
330   VecAssemblyBegin(x_out);
331   VecAssemblyEnd(x_out);
332   delete[] values;
333 }
```

Referenced by apply_sweep().

solve()

```
void NonLocalProblem::solve ( )  [override], [virtual]
```

Solves using a GMRES solver with a sweeping preconditioner.

The Sweeping preconditioner is also implemented in this class and calls on the child object for the next level. The included direct solver call can only occur if it is hard-coded to do so or the parameter use_direct_solver was set. This is only intended for debugging use. The function also uses a timer and generates output on the main stream of the application.

Implements HierarchicalProblem.

Definition at line 278 of file NonLocalProblem.cpp.

```
278                              {
279   is_shared_solution_up_to_date = false;
280   std::chrono::steady_clock::time_point time_begin;
281   std::chrono::steady_clock::time_point time_end;
282   if(level == GlobalParams.Sweeping_Level) {
283     print_vector_norm(&rhs, "RHS");
284     time_begin = std::chrono::steady_clock::now();
285   }
286
287   bool run_itterative_solver = !GlobalParams.solve_directly;
288
289   if(run_itterative_solver) {
```

```
    residual_output->new_series("Run " + std::to_string(solve_counter + 1));
    // Solve with sweeping

    PetscErrorCode ierr = KSPSolve(ksp, rhs, solution);
    residual_output->close_current_series();
    if(ierr != 0) {
      std::cout « "Error code from Petsc: " « std::to_string(ierr) « std::endl;
    }

  } else {
    // Solve Directly for reference
    SolverControl sc;
    dealii::PETScWrappers::SparseDirectMUMPS direct_solver(sc, GlobalMPI.communicators_by_level[level]);
    direct_solver.solve(*matrix, solution, rhs);
  }

  if(level == GlobalParams.Sweeping_Level) {
    time_end = std::chrono::steady_clock::now();
    print_info("NonlocalProblem::solve", "Solving took " +
   std::to_string(std::chrono::duration_cast<std::chrono::seconds>(time_end - time_begin).count()) +
   "[s]");
  }

}
```

update_convergence_criterion()

```
void NonLocalProblem::update_convergence_criterion (
              double last_residual )  [override], [virtual]
```

To be able to abort early on child solvers, we need to store the current residual on the current level.

This value can then be accessed by a child solver to determine its abort condition.

Parameters

last_residual	Latest computed local residual.

Reimplemented from HierarchicalProblem.

Definition at line 804 of file NonLocalProblem.cpp.

```
804                                                              {
805   if(GlobalParams.use_relative_convergence_criterion) {
806     double base_value = last_residual;
807     if(last_residual > 1.0) {
808       base_value = 1.0;
809     }
810     double new_abort_limit = base_value * GlobalParams.relative_convergence_criterion;
811     new_abort_limit = std::max(new_abort_limit, GlobalParams.Solver_Precision);
812     KSPSetTolerances(ksp, 1e-10,new_abort_limit , 1000, GlobalParams.GMRES_max_steps);
813     // std::cout « "Setting level " « level « " convergence criterion to " « new_abort_limit «
       std::endl;
814   }
815 }
```

update_shared_solution_vector()

```
void NonLocalProblem::update_shared_solution_vector ( )
```

Not all locally active dofs (dofs that couple to locally owned ones) are locally owned.

For output operations we need to access all these values from local memory. This function gathers all non-locally-owned dof values and stores them in a purely local vector.

Definition at line 505 of file NonLocalProblem.cpp.

```
505                                                      {
506   if(! is_shared_solution_up_to_date) {
507     shared_solution.reinit(own_dofs, locally_active_dofs, GlobalMPI.communicators_by_level[level]);
508     for(unsigned int i= 0; i < own_dofs.n_elements(); i++) {
509       shared_solution[own_dofs.nth_index_in_set(i)] = solution[own_dofs.nth_index_in_set(i)];
510     }
511     shared_solution.update_ghost_values();
512     is_shared_solution_up_to_date = true;
513   }
514   if(has_adjoint) {
515     shared_adjoint.reinit(own_dofs, locally_active_dofs, GlobalMPI.communicators_by_level[level]);
516     for(unsigned int i= 0; i < own_dofs.n_elements(); i++) {
517       shared_adjoint[own_dofs.nth_index_in_set(i)] = adjoint_state[own_dofs.nth_index_in_set(i)];
518     }
519     shared_adjoint.update_ghost_values();
520   }
521 }
```

Referenced by set_rhs_for_adjoint_problem(), and write_multifile_output().

write_multifile_output()

```
void NonLocalProblem::write_multifile_output (
              const std::string & filename,
              bool apply_coordinate_transform )  [override], [virtual]
```

Generates actual output files about the current levels solution.

For a given filename this function writes the vtu and vtk output files for the inner domain and the boundary methods (if they are PML). It keeps track of all the generated files and generates a header file for Paraview which loads all the individual files. If the input flaf transformed is true, it does the same for the solution in the physical coordinate sysytem.

Parameters

filename	Base part of the output file names.
apply_coordinate_transform	if true, the output will be in transformed coordinates.

Implements HierarchicalProblem.

Definition at line 523 of file NonLocalProblem.cpp.

```
523                                                                                           {
524   update_shared_solution_vector();
525   if(GlobalParams.MPI_Rank == 0 && !GlobalParams.solve_directly) {
526     residual_output->run_gnuplot();
527     if(level > 1) {
528       child->residual_output->run_gnuplot();
529       if(level == 3) {
530         child->child->residual_output->run_gnuplot();
531       }
532     }
533   }
534   std::vector<std::string> generated_files;
535 
536   NumericVectorLocal local_solution(Geometry.levels[level].inner_domain->n_locally_active_dofs);
```

```
  for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->n_locally_active_dofs; i++) {
    local_solution[i] = shared_solution[Geometry.levels[level].inner_domain->global_index_mapping[i]];
  }

  std::string file_1 = Geometry.levels[level].inner_domain->output_results(in_filename +
   std::to_string(level) , local_solution, transform);
  generated_files.push_back(file_1);
  if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML && !transform) {
    for (unsigned int surf = 0; surf < 6; surf++) {
      if(Geometry.levels[level].surface_type[surf] == SurfaceType::ABC_SURFACE){
        dealii::Vector<ComplexNumber> ds (Geometry.levels[level].surfaces[surf]->n_locally_active_dofs);
        for(unsigned int index = 0; index <
   Geometry.levels[level].surfaces[surf]->n_locally_active_dofs; index++) {
          ds[index] =
   shared_solution[Geometry.levels[level].surfaces[surf]->global_index_mapping[index]];
        }
        std::string file_2 = Geometry.levels[level].surfaces[surf]->output_results(ds, in_filename +
   "_pml" + std::to_string(level));
        generated_files.push_back(file_2);
      }
    }
  }
  std::vector<std::vector<std::string» all_files =
   dealii::Utilities::MPI::gather(GlobalMPI.communicators_by_level[level], generated_files);
  if(GlobalParams.MPI_Rank == 0) {
    std::vector<std::string> flattened_filenames;
    for(unsigned int i = 0; i < all_files.size(); i++) {
      for(unsigned int j = 0; j < all_files[i].size(); j++) {
        flattened_filenames.push_back(all_files[i][j]);
      }
    }
    std::string filename = GlobalOutputManager.get_full_filename("_" + in_filename + ".pvtu");
    std::ofstream outputvtu(filename);
    for(unsigned int i = 0; i < flattened_filenames.size(); i++) {
      flattened_filenames[i] = "../" + flattened_filenames[i];
    }
    Geometry.levels[level].inner_domain->data_out.write_pvtu_record(outputvtu, flattened_filenames);
  }
}
```

References update_shared_solution_vector().

The documentation for this class was generated from the following files:

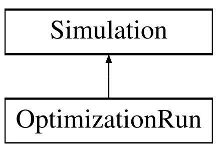
- Code/Hierarchy/NonLocalProblem.h
- Code/Hierarchy/NonLocalProblem.cpp

53 OptimizationRun Class Reference

This runner performs a shape optimization run based on adjoint based shape optimization.

`#include <OptimizationRun.h>`

Inheritance diagram for OptimizationRun:



Public Member Functions

- OptimizationRun ()

  Computes the number of free shape dofs for this configuration.
- void prepare () override

  Prepares the object by constructing the solver hierarchy.
- void run () override

  Calls the BFGS solver and writes output.
- void prepare_transformed_geometry () override

  Not required / implemented for this runner.

Static Public Member Functions

- static double perform_step (const dealii::Vector< double > &x, dealii::Vector< double > &g)

  This function is called by the BFGS solver.
- static void solve_main_problem ()

  Assembles and solves forward and adjoint problem.
- static void set_shape_dofs (const dealii::Vector< double > in_shape_dofs)

  This function updates the stored shape configuration for a provided vector of dof values.

53.1 Detailed Description

This runner performs a shape optimization run based on adjoint based shape optimization.

It is therefore one of the runner types that solves multiple forward problems.

Definition at line 22 of file OptimizationRun.h.

53.2 Constructor & Destructor Documentation

OptimizationRun()

`OptimizationRun::OptimizationRun ( )`

Computes the number of free shape dofs for this configuration.

Also inits the step counter to 0.

Definition at line 29 of file OptimizationRun.cpp.

```
29                              :
30   n_free_dofs(GlobalSpaceTransformation->n_free_dofs())
31   {
32   function_pointer = &OptimizationRun::perform_step;
33   OptimizationRun::step_counter = 0;
34 }
```

53.3 Member Function Documentation

perform_step()

```
double OptimizationRun::perform_step (
            const dealii::Vector< double > & x,
            dealii::Vector< double > & g )  [static]
```

This function is called by the BFGS solver.

It gives the next state and requests the shape gradient and the loss functional for that configuration in return.

In the function we set the provided values in x as the new shape parameter values. Then we solve the forward and adjoint state and compute the shape gradient. We push the values into the input argument g which stores the gradient components and compute the loss functional which we return. Additionaly we increment the step counter.

Parameters

x	New shape configuration to compute.
g	Return argument to write the gradient to.

Returns

double The evaluation of the loss functional for the given shape parametrization.

Definition at line 84 of file OptimizationRun.cpp.

```
84                                                                                       {
85    std::vector<double> x_vec(x.size());
86    for(unsigned int i = 0; i < x.size(); i++) {
87      x_vec[i] = x[i];
88    }
89    OptimizationRun::shape_dofs.push_back(x_vec);
90    OptimizationRun::set_shape_dofs(x);
91    OptimizationRun::solve_main_problem();
92    double loss_functional_evaluation = -std::abs(mainProblem->compute_signal_strength_of_solution());
93    print_info("OptimizationRun::perform_step", "Loss functional in step " +
      std::to_string(OptimizationRun::step_counter) + ": " + std::to_string(loss_functional_evaluation));
94    std::vector<double> shape_grad = mainProblem->compute_shape_gradient();
95    OptimizationRun::shape_gradients.push_back(shape_grad);
96    std::string msg = "Shape gradient: ( ";
97    for(unsigned int i = 0; i < g.size(); i++) {
98      g[i] = shape_grad[i];
99      msg += std::to_string(g[i]);
100     if(i < g.size() -1) {
101       msg += ", ";
102     } else {
103       msg += ")";
104     }
105   }
106   print_info("OptimizationRun::perform_step", msg);
107   OptimizationRun::step_counter += 1;
108   return loss_functional_evaluation;
109 }
```

References NonLocalProblem::compute_signal_strength_of_solution(), set_shape_dofs(), and solve_main_problem().

run()

```
void OptimizationRun::run ( )  [override], [virtual]
```

Calls the BFGS solver and writes output.

First prepare the vector of shape parameters for the start configuration. Then we call the BFGS solver to perform the shape optimization and give it a handle to this object for the update handler.

Implements Simulation.

Definition at line 49 of file OptimizationRun.cpp.

```
49                          {
50    print_info("OptimizationRun::run", "Start", LoggingLevel::PRODUCTION_ONE);
51    const unsigned int n_shape_dofs = n_free_dofs;
52    dealii::Vector<double> shape_dofs(n_shape_dofs);
53    OptimizationRun::step_counter = 0;
54    for(unsigned int i = 0; i < n_shape_dofs; i++) {
55      shape_dofs[i] = GlobalSpaceTransformation->get_free_dof(i);
56      if(GlobalParams.MPI_Rank == 0) {
57        std::cout « "Shape dof " « i « ": " « shape_dofs[i] « std::endl;
58      }
59    }
60    dealii::SolverControl sc(GlobalParams.optimization_n_shape_steps,
        GlobalParams.optimization_residual_tolerance, true, true);
61    dealii::SolverBFGS<dealii::Vector<double» solver(sc);
62    try{
63      solver.solve(function_pointer, shape_dofs);
64    } catch(dealii::StandardExceptions::ExcMessage & e) {
65      print_info("OptimizationRun::run", "Optimization terminated.");
66    }
67
68    GlobalTimerManager.write_output();
69    OptimizationRun::mainProblem->output_results();
70    print_info("OptimizationRun::run", "End", LoggingLevel::PRODUCTION_ONE);
71 }
```

set_shape_dofs()

```
void OptimizationRun::set_shape_dofs (
              const dealii::Vector< double > in_shape_dofs )  [static]
```

This function updates the stored shape configuration for a provided vector of dof values.

Parameters

in_shape_dofs	

Definition at line 111 of file OptimizationRun.cpp.

```
111                                                                   {
112    std::string msg = "( ";
113    for(unsigned int i = 0; i < in_shape_dofs.size(); i++) {
114      msg += std::to_string(in_shape_dofs[i]);
115      if(i != in_shape_dofs.size() - 1) {
116        msg += ", ";
117      } else {
118        msg += ")";
119      }
120    }
121    print_info("OptimizationRun::set_shape_dofs", msg);
122
123    for(unsigned int i = 0; i < in_shape_dofs.size(); i++) {
124      GlobalSpaceTransformation->set_free_dof(i, in_shape_dofs[i]);
```

```
  }

}
```

Referenced by perform_step().

The documentation for this class was generated from the following files:

- Code/Runners/OptimizationRun.h
- Code/Runners/OptimizationRun.cpp

54 OutputManager Class Reference

Whenever we write output, we require filenames.

```
#include <OutputManager.h>
```

Public Member Functions

- void initialize ()

  Ensures the output directory exists and writes some basic output files like the run description.
- std::string get_full_filename (std::string filename)

  Creates a full filename that can be used with an std::ofstream based on a core part provided as an argument.
- std::string get_numbered_filename (std::string filename, unsigned int number, std::string extension)

  Gives a full filename with relative path for a provided core part, identifier and extension.
- void write_log_ling (std::string in_line)

  Writes a line of output to the processes output text file.
- void write_run_description (std::string git_commit_hash)

  Generates a file in the output folder with some core data about the run.

Public Attributes

- std::string **base_path**
- unsigned int **run_number**
- std::string **output_folder_path**
- std::ofstream **log_stream**

54.1 Detailed Description

Whenever we write output, we require filenames.

This object wraps the functionality of generating unique filenames for each process, boundary etc.

Definition at line 24 of file OutputManager.h.

54.2 Member Function Documentation

get_full_filename()

```
std::string OutputManager::get_full_filename (
            std::string filename )
```

Creates a full filename that can be used with an std::ofstream based on a core part provided as an argument.

Parameters

filename	The core bit of the full path (in Solutions/run356/solution.vtk this would be solution.vtk)-

Returns

std::string The full filename with relative path.

Definition at line 53 of file OutputManager.cpp.

```
53                                                          {
54     return output_folder_path + "/" + filename;
55 }
```

Referenced by get_numbered_filename(), and write_run_description().

get_numbered_filename()

```
std::string OutputManager::get_numbered_filename (
            std::string filename,
            unsigned int number,
            std::string extension )
```

Gives a full filename with relative path for a provided core part, identifier and extension.

This can be used whenever we know that multiple processes will call the same output method and provide the rank on every process to make sure the processess dont interfere with eachothers files.

Parameters

filename	Main part of the filename
number	Unique bit to differentiate between processes or boundary conditions or levels.
extension	File extension to be appended at the end

Returns

std::string Fully qualified filename to use for the generation of output.

Definition at line 85 of file OutputManager.cpp.

```
85
      {
```

```
86     return get_full_filename(filename) + std::to_string(number) + '.' + extension;
87 }
```

References get_full_filename().

write_log_ling()

```
void OutputManager::write_log_ling (
            std::string in_line )
```

Writes a line of output to the processes output text file.

Parameters

in_line	The text to be written to the log.

Definition at line 89 of file OutputManager.cpp.

```
89                                                        {
90     log_stream « in_line « std::endl;
91 }
```

write_run_description()

```
void OutputManager::write_run_description (
            std::string git_commit_hash )
```

Generates a file in the output folder with some core data about the run.

Parameters

git_commit_hash	This git hash will be included in the output to describe in which state the code was.

Definition at line 57 of file OutputManager.cpp.

```
57                                                                     {
58     std::string filename = get_full_filename("run_description.txt");
59     std::ofstream out(filename);
60     out « "Number of processes: \t" « GlobalParams.NumberProcesses « std::endl;
61     out « "Sweeping level: " « GlobalParams.Sweeping_Level « std::endl;
62     out « "Truncation Method: " « ((GlobalParams.BoundaryCondition == BoundaryConditionType::HSIE)?
       "HSIE" : "PML") « std::endl;
63     out « "Signal input method: " « (GlobalParams.use_tapered_input_signal ? "Taper" : "Dirichlet") «
       std::endl;
64     out « "Set 0 on input interface: " « (GlobalParams.prescribe_0_on_input_side ? "true" : "false") «
       std::endl;
65     out « "Use predefined shape: " « (GlobalParams.Use_Predefined_Shape ? "true" : "false") « std::endl;
66     if(GlobalParams.Use_Predefined_Shape) {
67         out « "Predefined Shape Number: " « GlobalParams.Number_of_Predefined_Shape « std::endl;
68     }
69     out « "Block Counts: [" « GlobalParams.Blocks_in_x_direction « "x" «
       GlobalParams.Blocks_in_y_direction « "x" « GlobalParams.Blocks_in_z_direction « "]" « std::endl;
70     out « "Global cell count x: " « GlobalParams.Blocks_in_x_direction * GlobalParams.Cells_in_x «
       std::endl;
71     out « "Global cell count y: " « GlobalParams.Blocks_in_y_direction * GlobalParams.Cells_in_y «
       std::endl;
```

```
    out « "Global cell count z: " « GlobalParams.Blocks_in_z_direction * GlobalParams.Cells_in_z « std::endl;
    out « "Number of PML cell layers: " « GlobalParams.PML_N_Layers « std::endl;
    out « "Use relative convergence limiter: " « (GlobalParams.use_relative_convergence_criterion ? "true" : "false") « std::endl;
    if(GlobalParams.use_relative_convergence_criterion) {
        out « "Relative convergence limit: " « GlobalParams.relative_convergence_criterion « std::endl;
    }
    out « "Global x range: " « Geometry.global_x_range.first « " to " « Geometry.global_x_range.second «std::endl;
    out « "Global y range: " « Geometry.global_y_range.first « " to " « Geometry.global_y_range.second «std::endl;
    out « "Global z range: " « Geometry.global_z_range.first « " to " « Geometry.global_z_range.second «std::endl;
    out « "Git commit hash: " « git_commit_hash « std::endl;
    out.close();
}
```

References get_full_filename().

The documentation for this class was generated from the following files:

- Code/GlobalObjects/OutputManager.h
- Code/GlobalObjects/OutputManager.cpp

55 ParameterOverride Class Reference

An object used to interpret command line arguments of type –override.

`#include <ParameterOverride.h>`

Public Member Functions

- bool read (std::string)

    Checks if the provided override string is valid and if so parses it.
- void perform_on (Parameters &in_p)

    Performs the parsed overrides on the provided parameter object.
- bool validate (std::string in_arg)

    Checks if the provided override string is a valid set of parameters and values.

Public Attributes

- bool **has_overrides**

55.1 Detailed Description

An object used to interpret command line arguments of type –override.

This is usefull when we re-run the same code and only want to vary one or few parameter values. Without this object type we would need paramter files for all combinations. With this type, we define the overrides and create base parameter files for all the other parameters.

Definition at line 22 of file ParameterOverride.h.

55.2 Member Function Documentation

perform_on()

```
void ParameterOverride::perform_on (
            Parameters & in_p )
```

Performs the parsed overrides on the provided parameter object.

Parameters

in_p	The parameter object to be updated (in place)

Definition at line 38 of file ParameterOverride.cpp.

```
38                                                           {
39      for(unsigned int i = 0; i < overrides.size(); i++) {
40          if(overrides[i].first == "n_pml_cells") {
41              print_info("ParameterOverride", "Replacing pml_n_cells with " + overrides[i].second);
42              in_parameters.PML_N_Layers = std::stoi(overrides[i].second);
43          }
44          if(overrides[i].first == "pml_sigma_max") {
45              print_info("ParameterOverride", "Replacing pml_sigma_max with " + overrides[i].second);
46              in_parameters.PML_Sigma_Max = std::stod(overrides[i].second);
47          }
48          if(overrides[i].first == "pml_order") {
49              print_info("ParameterOverride", "Replacing pml_order with " + overrides[i].second);
50              in_parameters.PML_skaling_order = std::stoi(overrides[i].second);
51          }
52          if(overrides[i].first == "solver_type") {
53              print_info("ParameterOverride", "Replacing iterative solver with " + overrides[i].second);
54              in_parameters.solver_type = solver_option(overrides[i].second);
55          }
56          if(overrides[i].first == "geometry_size_z") {
57              print_info("ParameterOverride", "Replacing geometry size z with " + overrides[i].second);
58              in_parameters.Geometry_Size_Z = stod(overrides[i].second);
59          }
60          if(overrides[i].first == "processes_in_z") {
61              print_info("ParameterOverride", "Replacing number of processes in z with " +
        overrides[i].second);
62              in_parameters.Blocks_in_z_direction = stoi(overrides[i].second);
63          }
64          if(overrides[i].first == "predefined_case_number") {
65              print_info("ParameterOverride", "Replacing predefined case number with " +
        overrides[i].second);
66              in_parameters.Number_of_Predefined_Shape = stoi(overrides[i].second);
67          }
68          if(overrides[i].first == "system_length") {
69              print_info("ParameterOverride", "Replacing system length with " + overrides[i].second);
70              in_parameters.Geometry_Size_Z = std::stod(overrides[i].second);
71          }
72      }
73 }
```

read()

```
bool ParameterOverride::read (
            std::string in_string )
```

Checks if the provided override string is valid and if so parses it.

Returns

true The input was valid and parsing it was successful.

false There was an error

Definition at line 8 of file ParameterOverride.cpp.

```
8                                                   {
9     if(!validate(in_string)) {
10         return false;
11     }
12     std::vector<std::string> blocks = split(in_string, ";");
13     for(unsigned int i = 0; i < blocks.size(); i++) {
14         std::vector<std::string> line_split = split(blocks[i], "=");
15         overrides.push_back(std::pair<std::string, std::string>(line_split[0], line_split[1]));
16         has_overrides = true;
17     }
18     return true;
19 }
```

References validate().

validate()

```
bool ParameterOverride::validate (
              std::string in_arg )
```

Checks if the provided override string is a valid set of parameters and values.

Parameters

in_arg	The parameter value of the override argument passed to the main application.

Returns

true This can be used as an override

false There was an error

Definition at line 21 of file ParameterOverride.cpp.

```
21                                                       {
22     if(in_string.size() < 4) {
23         return false;
24     }
25     if (in_string.find('=') == std::string::npos) {
26         return false;
27     }
28     std::vector<std::string> blocks = split(in_string, ";");
29     for(unsigned int i = 0; i < blocks.size(); i++) {
30         std::vector<std::string> line_split = split(blocks[i], "=");
31         if(line_split.size() != 2) {
32             return false;
33         }
34     }
35     return true;
36 }
```

Referenced by read().

The documentation for this class was generated from the following files:

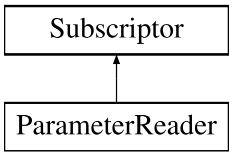
- Code/Helpers/ParameterOverride.h
- Code/Helpers/ParameterOverride.cpp

56 ParameterReader Class Reference

This class is used to gather all the information from the input file and store it in a static object available to all processes.

```
#include <ParameterReader.h>
```

Inheritance diagram for ParameterReader:



Public Member Functions

- ParameterReader ()

  Deal Offers the ParameterHandler object wich contains all of the parsing-functionality.
- Parameters read_parameters (const std::string run_file, const std::string case_file)

  This member calls the read_input_from_xml()-function of the contained ParameterHandler and this replaces the default values with the values in the input file.
- void declare_parameters ()

  In this function, we add all values descriptions to the parameter-handler.

56.1 Detailed Description

This class is used to gather all the information from the input file and store it in a static object available to all processes.

The ParameterReader is a very useful tool. It uses a deal-function to read a xml-file and parse the contents to specific variables. These variables have default values used in their declaration. The members of this class do two things:

1. declare the variables. This includes setting a data-type for them and a default value should none be provided in the input file. Furthermore there can be restrictions like maximum or minimum values etc.
2. call an external function to parse an input-file.

After creating an object of this type and calling both declare() and read(), this object contains all the information from the input file and can be used in the code without dealing with persistence.

Definition at line 40 of file ParameterReader.h.

56.2 Constructor & Destructor Documentation

ParameterReader()

```
ParameterReader::ParameterReader ( )
```

Deal Offers the ParameterHandler object wich contains all of the parsing-functionality.

An object of that type is included in this one. This constructor simply uses a copy-constructor to initialize it.

Definition at line 6 of file ParameterReader.cpp.

```
6 { }
```

56.3 Member Function Documentation

declare_parameters()

```
void ParameterReader::declare_parameters ( )
```

In this function, we add all values descriptions to the parameter-handler.

This includes

1. a default value,
2. a data-type,
3. possible restrictions (greater than zero etc.),
4. a description, which is displayed in deals ParameterGUI-tool,
5. a hierarchical structure to order the variables.

Deals Parameter-GUI can be installed at build-time of the library and offers a great and easy way to edit the input file. It displays appropriate input-methods depending on the type, so, for example, in case of a selection from three different values (i.e. the name of a solver that has to either be GMRES, MINRES or UMFPACK) it displays a dropdown containing all the options.

Definition at line 8 of file ParameterReader.cpp.

```
8                                      {
9     run_prm.enter_subsection("Run parameters");
10     {
11         run_prm.declare_entry("solver precision" , "1e-6", Patterns::Double(), "Absolute precision for
       solver convergence.");
12         run_prm.declare_entry("GMRES restart after" , "30", Patterns::Integer(), "Number of steps until
       GMRES restarts.");
13         run_prm.declare_entry("GMRES maximum steps" , "30", Patterns::Integer(), "Number of maximum GMRES
       steps until failure.");
14         run_prm.declare_entry("use relative convergence criterion", "true", Patterns::Bool(), "If this is
       set to false, lower level sweeping will ignore higher level current residual.");
15         run_prm.declare_entry("relative convergence criterion", "1e-2", Patterns::Double(), "The factor
       by which a lower level convergence criterion is computed.");
16         run_prm.declare_entry("solve directly", "false", Patterns::Bool(), "If this is set to true, GMRES
       will be replaced by a direct solver.");
```

```
    run_prm.declare_entry("kappa angle" , "1.0", Patterns::Double(), "Phase of the complex value kappa with norm 1 that is used in HSIEs.");
    run_prm.declare_entry("processes in x" , "1", Patterns::Integer(), "Number of processes in x-direction.");
    run_prm.declare_entry("processes in y" , "1", Patterns::Integer(), "Number of processes in y-direction.");
    run_prm.declare_entry("processes in z" , "1", Patterns::Integer(), "Number of processes in z-direction.");
    run_prm.declare_entry("sweeping level" , "1", Patterns::Integer(), "Hierarchy level to be used. 1: normal sweeping. 2: two level hierarchy, i.e sweeping in sweeping. 3: three level sweeping, i.e. sweeping in sweeping in swepping.");
    run_prm.declare_entry("cell count x" , "20", Patterns::Integer(), "Number of cells a single process has in x-direction.");
    run_prm.declare_entry("cell count y" , "20", Patterns::Integer(), "Number of cells a single process has in y-direction.");
    run_prm.declare_entry("cell count z" , "20", Patterns::Integer(), "Number of cells a single process has in z-direction.");
    run_prm.declare_entry("output transformed solution", "false", Patterns::Bool(), "If set to true, both the solution in mathematical and in physical coordinates will be written as outputs.");
    run_prm.declare_entry("Logging Level", "Production One", Patterns::Selection("Production One|Production All|Debug One|Debug All"), "Specifies which messages should be printed and by whom.");
    run_prm.declare_entry("solver type", "GMRES", Patterns::Selection("GMRES|MINRES|TFQMR|BICGS|CG|PCONLY"), "Choose the itterative solver to use.");
}
run_prm.leave_subsection();

case_prm.enter_subsection("Case parameters");
{
    case_prm.declare_entry("source type", "0", Patterns::Integer(), "PointSourceField is 0: empty, 1: cos()cos(), 2: Hertz Dipole, 3: Waveguide");
    case_prm.declare_entry("transformation type", "Waveguide Transformation", Patterns::Selection("Waveguide Transformation|Angle Waveguide Transformation|Bend Transformation"), "Inhomogenous Waveguide Transformation is used for straight waveguide cases and the predefined cases. Angle Waveguide Transformation is a PML test. Bend Transformation is an example for a 90 degree bend.");
    case_prm.declare_entry("geometry size x", "5.0", Patterns::Double(), "Size of the computational domain in x-direction.");
    case_prm.declare_entry("geometry size y", "5.0", Patterns::Double(), "Size of the computational domain in y-direction.");
    case_prm.declare_entry("geometry size z", "5.0", Patterns::Double(), "Size of the computational domain in z-direction.");
    case_prm.declare_entry("epsilon in", "2.3409", Patterns::Double(), "Epsilon r inside the material.");
    case_prm.declare_entry("epsilon out", "1.8496", Patterns::Double(), "Epsilon r outside the material.");
    case_prm.declare_entry("epsilon effective", "2.1588449", Patterns::Double(), "Epsilon r outside the material.");
    case_prm.declare_entry("mu in", "1.0", Patterns::Double(), "Mu r inside the material.");
    case_prm.declare_entry("mu out", "1.0", Patterns::Double(), "Mu r outside the material.");
    case_prm.declare_entry("fem order" , "0", Patterns::Integer(), "Degree of nedelec elements in the interior.");
    case_prm.declare_entry("signal amplitude", "1.0", Patterns::Double(), "Amplitude of the input signal or PointSourceField");
    case_prm.declare_entry("width of waveguide", "2.0", Patterns::Double(), "Width of the Waveguide core.");
    case_prm.declare_entry("height of waveguide", "1.8", Patterns::Double(), "Height of the Waveguide core.");
    case_prm.declare_entry("Enable Parameter Run", "false", Patterns::Bool(), "For a series of Local solves, this can be set to true");
    case_prm.declare_entry("Kappa 0 Real", "1", Patterns::Double(), "Real part of kappa_0 for HSIE.");
    case_prm.declare_entry("Kappa 0 Imaginary", "1", Patterns::Double(), "Imaginary part of kappa_0 for HSIE.");
    case_prm.declare_entry("PML sigma max", "10.0", Patterns::Double(), "Parameter Sigma Max for all PML layers.");
    case_prm.declare_entry("HSIE polynomial degree" , "4", Patterns::Integer(), "Polynomial degree of the Hardy-space polynomials for HSIE surfaces.");
    case_prm.declare_entry("Min HSIE Order", "1", Patterns::Integer(), "Minimal HSIE Element order for parameter run.");
    case_prm.declare_entry("Max HSIE Order", "21", Patterns::Integer(), "Maximal HSIE Element order for parameter run.");
    case_prm.declare_entry("Boundary Method", "HSIE", Patterns::Selection("HSIE|PML"), "Choose the boundary element method (options are PML and HSIE).");
    case_prm.declare_entry("PML thickness", "1.0", Patterns::Double(), "Thickness of PML layers.");
```

```
        case_prm.declare_entry("PML skaling order", "3", Patterns::Integer(), "PML skaling order is the exponent with wich the imaginary part grows towards the outer boundary.");
        case_prm.declare_entry("PML n layers", "8", Patterns::Integer(), "Number of cell layers used in the PML medium.");
        case_prm.declare_entry("PML Test Angle", "0.2", Patterns::Double(), "For the angeling test, this is a in z' = z - a * y.");
        case_prm.declare_entry("Input Signal Method", "Dirichlet", Patterns::Selection("Dirichlet|Taper"), "Taper uses a tapered exact solution to build a right hand side. Dirichlet applies dirichlet boundary values.");
        case_prm.declare_entry("Signal tapering type", "C1", Patterns::Selection("C0|C1"), "Tapering type for signal input");
        case_prm.declare_entry("Prescribe input zero", "false", Patterns::Bool(), "If this is set to true, there will be a dirichlet zero condition enforced on the global input interface (Process index z: 0, boundary id: 4).");
        case_prm.declare_entry("Predefined case number", "1", Patterns::Integer(), "Number in [1,35] that describes the predefined shape to use.");
        case_prm.declare_entry("Use predefined shape", "false", Patterns::Bool(), "If set to true, the geometry for the predefined case from 'Predefined case number' will be used.");
        case_prm.declare_entry("Number of shape sectors", "5", Patterns::Integer(), "Number of sectors for the shape approximation");
        case_prm.declare_entry("perform convergence test", "false", Patterns::Bool(), "If true, the code will perform a cnovergence run on a sequence of meshes.");
        case_prm.declare_entry("convergence sequence cell count", "1,2,4,8,10,14,16,20", Patterns::List(Patterns::Integer()), "The sequence of cell counts in each direction to be used for convergence analysis.");
        case_prm.declare_entry("global z shift", "0", Patterns::Double(), "Shifts the global geometry to remove the center of the dipole for convergence studies.");
        case_prm.declare_entry("Optimization Algorithm", "BFGS", Patterns::Selection("BFGS|Steepest"), "The algorithm to compute the next parametrization in an optimization run.");
        case_prm.declare_entry("Initialize Shape Dofs Randomly", "false", Patterns::Bool(), "If set to true, the shape dofs are initialized to random values.");
        case_prm.declare_entry("perform optimization", "false", Patterns::Bool(), "If true, the code will perform shape optimization.");
        case_prm.declare_entry("vertical waveguide displacement", "0", Patterns::Double(), "The delta of the waveguide core at the input and output interfaces.");
        case_prm.declare_entry("constant waveguide height", "true", Patterns::Bool(), "If false, the waveguide shape will be subject to optimization in the y direction.");
        case_prm.declare_entry("constant waveguide width", "true", Patterns::Bool(), "If false, the waveguide shape will be subject to optimization in the x direction.");
    }
    case_prm.leave_subsection();
}
```

The documentation for this class was generated from the following files:

- Code/Helpers/ParameterReader.h
- Code/Helpers/ParameterReader.cpp

57 Parameters Class Reference

This structure contains all information contained in the input file and some values that can simply be computed from it.

`#include <Parameters.h>`

Public Member Functions

- auto **complete_data** () -> void
- auto **check_validity** () -> bool

Public Attributes

- ShapeDescription **sd**
- double **Solver_Precision** = 1e-6
- unsigned int **GMRES_Steps_before_restart** = 30
- unsigned int **GMRES_max_steps** = 100
- unsigned int **MPI_Rank**
- unsigned int **NumberProcesses**
- double **Amplitude_of_input_signal** = 1.0
- bool **Output_transformed_solution** = false
- double **Width_of_waveguide** = 1.8
- double **Height_of_waveguide** = 2.0
- double **Horizontal_displacement_of_waveguide** = 0
- double **Vertical_displacement_of_waveguide** = 0
- double **Epsilon_R_in_waveguide** = 2.3409
- double **Epsilon_R_outside_waveguide** = 1.8496
- double **Epsilon_R_effective** = 2.1588449
- double **Mu_R_in_waveguide** = 1.0
- double **Mu_R_outside_waveguide** = 1.0
- unsigned int **HSIE_polynomial_degree** = 5
- bool **Perform_Optimization** = false
- unsigned int **optimization_n_shape_steps** = 15
- double **optimization_residual_tolerance** = 1.e-10
- double **kappa_0_angle** = 1.0
- ComplexNumber **kappa_0**
- unsigned int **Nedelec_element_order** = 0
- unsigned int **Blocks_in_z_direction** = 1
- unsigned int **Blocks_in_x_direction** = 1
- unsigned int **Blocks_in_y_direction** = 1
- unsigned int **Index_in_x_direction**
- unsigned int **Index_in_y_direction**
- unsigned int **Index_in_z_direction**
- unsigned int **Cells_in_x** = 20
- unsigned int **Cells_in_y** = 20
- unsigned int **Cells_in_z** = 20
- int **current_run_number** = 0
- double **Geometry_Size_X** = 5
- double **Geometry_Size_Y** = 5
- double **Geometry_Size_Z** = 5
- unsigned int **Number_of_sectors** = 1

- double **Sector_thickness**
- double **Sector_padding**
- double **Pi** = 3.141592653589793238462
- double **Omega** = 1.0
- double **Lambda** = 1.55
- double **Waveguide_value_V** = 1.0
- bool **Use_Predefined_Shape** = false
- unsigned int **Number_of_Predefined_Shape** = 1
- unsigned int **Point_Source_Type** = 0
- unsigned int **Sweeping_Level** = 1
- LoggingLevel **Logging_Level** = LoggingLevel::DEBUG_ALL
- dealii::Function< 3, ComplexNumber > * **source_field**
- bool **Enable_Parameter_Run** = false
- unsigned int **N_Kappa_0_Steps** = 20
- unsigned int **Min_HSIE_Order** = 1
- unsigned int **Max_HSIE_Order** = 10
- double **PML_Sigma_Max** = 5.0
- unsigned int **PML_N_Layers** = 8
- double **PML_thickness** = 1.0
- double **PML_Angle_Test** = 0.2
- unsigned int **PML_skaling_order** = 3
- BoundaryConditionType **BoundaryCondition** = BoundaryConditionType::HSIE
- bool **use_tapered_input_signal** = false
- double **tapering_min_z** = 0.0
- double **tapering_max_z** = 1.0
- SolverOptions **solver_type** = SolverOptions::GMRES
- SignalTaperingType **Signal_tapering_type** = SignalTaperingType::C1
- SignalCouplingMethod **Signal_coupling_method** = SignalCouplingMethod::Tapering
- double **tapering_z_min** = 0
- double **tapering_t_max** = 1
- bool **prescribe_0_on_input_side** = false
- bool **use_relative_convergence_criterion** = false
- double **relative_convergence_criterion** = 0.01
- bool **Perform_Convergence_Test** = false
- unsigned int **convergence_max_cells** = 20
- TransformationType **transformation_type** = TransformationType::WavegeuideTransformationType
- std::vector< unsigned int > **convergence_cell_counts**
- double **global_z_shift** = 0
- bool **solve_directly** = false
- SteppingMethod **optimization_stepping_method** = SteppingMethod::BFGS

- bool **keep_waveguide_height_constant** = true
- bool **keep_waveguide_width_constant** = true
- bool **randomly_initialize_shape_dofs** = false

57.1 Detailed Description

This structure contains all information contained in the input file and some values that can simply be computed from it.

In the application, static Variable of this type makes the input parameters available globally.

Definition at line 29 of file Parameters.h.

The documentation for this class was generated from the following files:

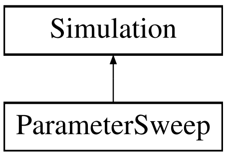
- Code/Helpers/Parameters.h
- Code/Helpers/Parameters.cpp

58 ParameterSweep Class Reference

The Parameter run performs multiple forward runs for a sweep across a parameter value, i.e multiple computations for different domain sizes or similar.

```
#include <ParameterSweep.h>
```

Inheritance diagram for ParameterSweep:



Public Member Functions

- void prepare () override

  In derived classes, this function sets up all that is required to perform the core functionality, i.e.
- void run () override

  Run the core computation.
- void prepare_transformed_geometry () override

  If a representation of the solution in the physical coordinates is required, this function provides it.

58.1 Detailed Description

The Parameter run performs multiple forward runs for a sweep across a parameter value, i.e multiple computations for different domain sizes or similar.

This is not really required anymore because there is now an implementation of parameter overrides which does the same but is parallelizable. The class is not documented for this reason but the code is simple.

Definition at line 23 of file ParameterSweep.h.

58.2 Member Function Documentation

prepare()

```
void ParameterSweep::prepare ( )  [override], [virtual]
```

In derived classes, this function sets up all that is required to perform the core functionality, i.e.

construct problems types.

Implements Simulation.

Definition at line 18 of file ParameterSweep.cpp.

```
18                            {
19   print_info("ParameterSweep::prepare", "Start", LoggingLevel::DEBUG_ONE);
20   if(GlobalParams.Point_Source_Type == 0) {
21     rmProblem = new RectangularMode();
22   }
23   print_info("ParameterSweep::prepare", "End", LoggingLevel::DEBUG_ONE);
24 }
```

The documentation for this class was generated from the following files:

- Code/Runners/ParameterSweep.h
- Code/Runners/ParameterSweep.cpp

59 PMLMeshTransformation Class Reference

Generating the basic mesh for a PML domain is simple because it is an axis parallel cuboid. This functions shifts and stretches the domain to the correct proportions.

`#include <PMLMeshTransformation.h>`

Public Member Functions

- **PMLMeshTransformation** (std::pair< double, double > in_x_range, std::pair< double, double > in_y_range, std::pair< double, double > in_z_range, double in_base_coordinate, unsigned int in_outward_direction, std::array< bool, 6 > in_transform_coordinate)
- Position operator() (const Position &in_p) const

  Transforms a coordinate of the unit cube onto the actual sizes provided in the constructor of this object.
- Position undo_transform (const Position &in_p)

  Inverse operation of operator().

Public Attributes

- std::pair< double, double > **default_x_range**
- std::pair< double, double > **default_y_range**
- std::pair< double, double > **default_z_range**
- double **base_coordinate_for_transformed_direction**
- unsigned int **outward_direction**
- std::array< bool, 6 > **transform_coordinate**

59.1 Detailed Description

Generating the basic mesh for a PML domain is simple because it is an axis parallel cuboid. This functions shifts and stretches the domain to the correct proportions.

Specifically, the implementation is done in the operator() function. Choosing this nomenclature, the function is compatible with deal.IIs interface for a coordinate transformation and an object of this type can be used directy in the GridTools::transform function.

Definition at line 25 of file PMLMeshTransformation.h.

59.2 Member Function Documentation

operator()()

```
Position PMLMeshTransformation::operator() (
            const Position & in_p ) const
```

Transforms a coordinate of the unit cube onto the actual sizes provided in the constructor of this object.

Parameters

in_p	The coordinate to be transformed.

Returns

Position The transformed coordinated.

Definition at line 27 of file PMLMeshTransformation.cpp.

```
27                                                              {
28     Position ret = in_p;
29     double extension_factor = std::abs(in_p[outward_direction] -
       base_coordinate_for_transformed_direction);
30     if(outward_direction != 0) {
31         if(std::abs(in_p[0] - default_x_range.first ) < FLOATING_PRECISION && transform_coordinate[0])
       ret[0] -= extension_factor;
32         if(std::abs(in_p[0] - default_x_range.second) < FLOATING_PRECISION && transform_coordinate[1])
       ret[0] += extension_factor;
33     }
34     if(outward_direction != 1) {
35         if(std::abs(in_p[1] - default_y_range.first ) < FLOATING_PRECISION && transform_coordinate[2])
       ret[1] -= extension_factor;
```

```
        if(std::abs(in_p[1] - default_y_range.second) < FLOATING_PRECISION && transform_coordinate[3])
    ret[1] += extension_factor;
    }
    if(outward_direction != 2) {
        if(std::abs(in_p[2] - default_z_range.first ) < FLOATING_PRECISION && transform_coordinate[4])
    ret[2] -= extension_factor;
        if(std::abs(in_p[2] - default_z_range.second) < FLOATING_PRECISION && transform_coordinate[5])
    ret[2] += extension_factor;
    }
    return ret;
}
```

undo_transform()

```
Position PMLMeshTransformation::undo_transform (
            const Position & in_p )
```

Inverse operation of operator().

Parameters

in_p	The coordinate on which to undo the transformation

Returns

Position The coordinate before operator() was applied to it.

Definition at line 45 of file PMLMeshTransformation.cpp.

```
45                                                                     {
46     Position ret = in_p;
47     if(in_p[0] < default_x_range.first)  ret[0] = default_x_range.first;
48     if(in_p[0] > default_x_range.second) ret[0] = default_x_range.second;
49     if(in_p[1] < default_y_range.first)  ret[1] = default_y_range.first;
50     if(in_p[1] > default_y_range.second) ret[1] = default_y_range.second;
51     if(in_p[2] < default_z_range.first)  ret[2] = default_z_range.first;
52     if(in_p[2] > default_z_range.second) ret[2] = default_z_range.second;
53     return ret;
54 }
```

The documentation for this class was generated from the following files:

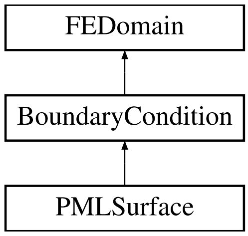
- Code/BoundaryCondition/PMLMeshTransformation.h
- Code/BoundaryCondition/PMLMeshTransformation.cpp

60 PMLSurface Class Reference

An implementation of a UPML method.

`#include <PMLSurface.h>`

Inheritance diagram for PMLSurface:



Public Member Functions

- **PMLSurface** (unsigned int in_bid, unsigned int in_level)
- bool is_point_at_boundary (Position, BoundaryId)

  Checks if the provided coordinate is at the provided boundary.
- auto make_constraints () -> Constraints override

  For this method we use PEC boundary conditions on the outside of the PML domain.
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix *matrix, NumericVectorDistributed *rhs, Constraints *constraints) override

  Writes the FE system of this PML domain to a provided system matrix and rhs vector using the constraints.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern *in_dsp, Constraints *in_constriants) override

  Sets the locations of actually coupling dofs to non-zero in a sparsity pattern so we know to reserve memory for it.
- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

  Checks if a 2D position of the surface mesh is also at another boundary, i.e.
- bool is_position_at_boundary (const Position in_p, const BoundaryId in_bid)

  This function and the next are used to color the surfaces of the PML domain.
- bool is_position_at_extended_boundary (const Position in_p, const BoundaryId in_bid)

  This function and the previous one a re used to color the surfaces of the PML domain.
- void initialize () override

  Initializes the data structures to reserve memory.
- void set_mesh_boundary_ids ()

  Set the mesh boundary ids by checking if faces and edges are at certain boundaries.
- void prepare_mesh ()

  Builds the mesh of the PML domain and corner/edge connecting domains.
- auto cells_for_boundary_id (unsigned int boundary_id) -> unsigned int override

  Counts the number of cells at a boundary id.
- void init_fe ()

  Initializes all the parts of the finite element loop like the dof handler and the finite element object that provides shape functions.
- auto fraction_of_pml_direction (Position) -> std::array< double, 3 >

  Computes the fraction of the PML thickness of the provided position for the computation of sigma for all three space directions.

- auto get_pml_tensor_epsilon (Position in_p) -> dealii::Tensor< 2, 3, ComplexNumber >

  Get the PML material tensor ϵ_p for a given position.
- auto get_pml_tensor_mu (Position in_p) -> dealii::Tensor< 2, 3, ComplexNumber >

  Get the PML material tensor μ_p for a given position.
- auto get_pml_tensor (Position) -> dealii::Tensor< 2, 3, ComplexNumber >

  Internal function that computes the purely geometric transformation tensor.
- auto get_dof_association () -> std::vector< InterfaceDofData > override

  Get the degrees of freedom associated with the interface to the inner domain.
- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

  Get the degrees of freedom associated with either the inner domain or another boundary conditions domain.
- void compute_coordinate_ranges (dealii::Triangulation< 3 > ∗in_tria)

  Internal function to compute the coordinate ranges of the domain occupied by this UPML domain.
- void set_boundary_ids ()

  Color the mesh surfaces.
- void fix_apply_negative_Jacobian_transformation (dealii::Triangulation< 3 > ∗in_tria)

  Inverts vertex and edge orders to switch the sign of the cell volumes.
- std::string output_results (const dealii::Vector< ComplexNumber > &solution_vector, std::string filename) override

  Writes an output file for paraview of the solution provided projected onto the local mesh.
- void validate_meshes ()

  Performs basic tests on the meshes to check if they are valid.
- DofCount compute_n_locally_owned_dofs () override

  Counts the locally owned dofs.
- DofCount compute_n_locally_active_dofs () override

  Counts the locally active dofs.
- void finish_dof_index_initialization () override

  Iterates over all surfaces of the PML domain and sets the dof indices if the surface is not locally owned.
- void determine_non_owned_dofs () override

  Marks all non-owned dofs in the is_dof_locally_owned array.
- dealii::IndexSet compute_non_owned_dofs ()

  Generates an dealii::IndexSet of all non locally owned dofs.
- bool finish_initialization (DofNumber first_own_index) override

  Given a first index, this function numbers the owned dofs starting at that number.
- bool mg_process_edge (dealii::Triangulation< 3 > ∗return_pointer, BoundaryId b_id)

  Checks if the PML requires an extension domain towards the boundary with BoundaryId b_id and, if so, creates a mesh of that extension and provides it in the pointer argument.
- bool mg_process_corner (dealii::Triangulation< 3 > ∗current_list, BoundaryId first_bid, BoundaryId second_bid)

Same as above but for edges.

- bool extend_mesh_in_direction (BoundaryId in_bid)

  Check if an extension domain is required towards the boundary in_bid.

- void prepare_dof_associations ()

  Caches the association of dofs with the surfaces so it can be accessed cheaper in the future.

- unsigned int n_cells () override

  Counts the number of local cells.

Public Attributes

- std::pair< double, double > **x_range**
- std::pair< double, double > **y_range**
- std::pair< double, double > **z_range**
- double **non_pml_layer_thickness**
- dealii::Triangulation< 3 > **triangulation**

60.1 Detailed Description

An implementation of a UPML method.

This is one of the core objects in the entire implementation. For an explanation of the PML method, please read section 4.4.3.

This object assembles matrix blocks for our system that act as an absorbing boundary condition. In essence, the object builds a mesh for the PML domain and uses Nedelec elements on it to compute the matrix entries. Additionally, it can fill a sparsity pattern with the information about which dofs couple to which and it also manages its own dofs, i.e. the ones that aren't also dofs on the inner domain. The object additionally sets the PEC boundary conditions section 4.4.1 on the outside boundary of the PML domain. If a neighboring boundary condition also uses PML, this object is capable of building a connecting mesh of the corner or edge domains to couple the systems together. The method can use either a constant value for the imaginary part of the material tensor or a ramping value of arbitrary order.

Mesh geometry: The inner domain is a cube of say 10x10x10 cells. This method primarily builds an extension of that geometry in one direction. We choose one boundary (specified by b_id) and connect an additional domain with the inner domain. This additional domain shares the same cell counts in the surface tangential directions and has a specified thickness, which is a global parameter. Lets assume this thickness is 5. If the b_id is 5, i.e. this PML surface is handling the +z surface of the inner domain, then the PML domain will have 10x10x5 cells (the 5 in the third component because z direction). An important point is the following: If say boundary id 3 (+y) also uses PML and has such an extension, then we need to somehow couple the dofs of the PML domain for b_id 5 facing towards +y and the boundary dofs of the PML domain for b_id 3 facing towards +z together. To facilitate this, we introduce a connecting domain, an edge domain. This edge domain will have 10x5x5 cells. The same problem arises if we add another PML domain on the surface for b_id 1 (+x). All three PML domains discussed so far share a corner which we have to discretize by 5x5x5 cells.

To be able to easily ectract the boundary degrees of freedom, we rely on coloring, i.e. a cached value on each edge indicating to which surface it belongs. This can then be used to quickly retrieve dof indices for boundaries. To make this possible, we itterate over the mesh and check for relevant structures (cell, face and edge centers) if they are located at the relevant surfaces. Also: We want all dofs to be owend by one process / object. As a consequence, the connecting corner and edge domains are assembled by one side, not shared. Edge and corner domains are always owned by the boundary condition with the higher b_id (this makes sense in cobination with sweeping). If a mesh is extended in a direction, we use the method is_position_at_extended_boundary, otherwise we use is_position_at_boundary.

The shape of these PML domains can be seen in the output generated by this code since the solution on PML domains is written to seperate output files.

As a special implementation detail it should be noted, that the cell layer touching the inner domain does not use the material tensor with imaginary part. It is instead treated as normal computational domain.

Definition at line 44 of file PMLSurface.h.

60.2 Member Function Documentation

cells_for_boundary_id()

```
unsigned int PMLSurface::cells_for_boundary_id (
            unsigned int boundary_id ) -> unsigned int  [override], [virtual]
```

Counts the number of cells at a boundary id.

Parameters

boundary_id	The boundary to search on.

Returns

unsigned int The number of cells.

Reimplemented from BoundaryCondition.

Definition at line 127 of file PMLSurface.cpp.

```
127                                                                          {
128     unsigned int ret = 0;
129     for(auto it = triangulation.begin(); it!= triangulation.end(); it++) {
130       if(it->at_boundary()) {
131         for(unsigned int i = 0; i < 6; i++) {
132           if(it->face(i)->boundary_id() == in_boundary_id) {
133             ret++;
134           }
135         }
136       }
137     }
138     return ret;
139 }
```

compute_coordinate_ranges()

```
void PMLSurface::compute_coordinate_ranges (
            dealii::Triangulation< 3 > * in_tria )
```

Internal function to compute the coordinate ranges of the domain occupied by this UPML domain.

Parameters

in_tria	

Definition at line 486 of file PMLSurface.cpp.

```
486 {
487   x_range.first = 100000.0;
488   y_range.first = 100000.0;
489   z_range.first = 100000.0;
490   x_range.second = -100000.0;
491   y_range.second = -100000.0;
492   z_range.second = -100000.0;
493   for(auto it = in_tria->begin(); it != in_tria->end(); it++) {
494     for(unsigned int i = 0; i < 6; i++) {
495       if(it->face(i)->at_boundary()) {
496         Position p = it->face(i)->center();
497         if(p[0] < x_range.first) {
498           x_range.first = p[0];
499         }
500         if(p[0] > x_range.second) {
501           x_range.second = p[0];
502         }
503         if(p[1] < y_range.first) {
504           y_range.first = p[1];
505         }
506         if(p[1] > y_range.second) {
507           y_range.second = p[1];
508         }
509         if(p[2] < z_range.first) {
510           z_range.first = p[2];
511         }
512         if(p[2] > z_range.second) {
513           z_range.second = p[2];
514         }
515       }
516     }
517   }
518 }
```

compute_n_locally_active_dofs()

`DofCount PMLSurface::compute_n_locally_active_dofs ( )  [override], [virtual]`

Counts the locally active dofs.

Returns

DofCount the number of the dofs that are locally active.

Implements FEDomain.

Definition at line 674 of file PMLSurface.cpp.

```
674 {
675   return dof_counter;
676 }
```

compute_n_locally_owned_dofs()

`DofCount PMLSurface::compute_n_locally_owned_dofs ( )  [override], [virtual]` Counts the locally owned dofs.

Returns

DofCount the number of the dofs that are locally owned.

Implements FEDomain.

Definition at line 669 of file PMLSurface.cpp.

```
669                                                     {
670   IndexSet non_owned_dofs = compute_non_owned_dofs();
671   return dof_counter - non_owned_dofs.n_elements();
672 }
```

References compute_non_owned_dofs().

compute_non_owned_dofs()

```
dealii::IndexSet PMLSurface::compute_non_owned_dofs ( )
```

Generates an dealii::IndexSet of all non locally owned dofs.

Returns

dealii::IndexSet The IndexSet of non-owned dofs.

Definition at line 745 of file PMLSurface.cpp.

```
745                                                     {
746   IndexSet non_owned_dofs(dof_counter);
747   std::vector<unsigned int> non_locally_owned_surfaces;
748   for(auto surf : adjacent_boundaries) {
749     if(!are_edge_dofs_owned[surf]) {
750       non_locally_owned_surfaces.push_back(surf);
751     }
752   }
753   non_locally_owned_surfaces.push_back(inner_boundary_id);
754
755   std::vector<unsigned int> local_indices(fe_nedelec.dofs_per_face);
756   // The non owned surfaces are the one towards the inner domain and the surfaces 0,1 and 2 if they are
       false in the input.
757   for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
758     for(unsigned int face = 0; face < 6; face++) {
759       if(it->face(face)->at_boundary()) {
760         for(auto surf: non_locally_owned_surfaces) {
761           if(it->face(face)->boundary_id() == surf) {
762             it->face(face)->get_dof_indices(local_indices);
763             for(unsigned int i = 0; i < fe_nedelec.dofs_per_face; i++) {
764               non_owned_dofs.add_index(local_indices[i]);
765             }
766           }
767         }
768       }
769     }
770   }
771   return non_owned_dofs;
772 }
```

Referenced by compute_n_locally_owned_dofs(), and determine_non_owned_dofs().

extend_mesh_in_direction()

```
bool PMLSurface::extend_mesh_in_direction (
            BoundaryId in_bid )
```

Check if an extension domain is required towards the boundary in_bid.

Parameters

in_bid	The other boundary to be checked towards

Returns

true

false

Definition at line 520 of file PMLSurface.cpp.

```
520                                                 {
521   if(Geometry.levels[level].surface_type[in_bid] != SurfaceType::ABC_SURFACE) {
522     return false;
523   }
524   if(b_id == 4 || b_id == 5) {
525     return true;
526   }
527   if(b_id == 0 || b_id == 1) {
528     return false;
529   }
530   if(b_id == 2 || b_id == 3) {
531     return in_bid < b_id;
532   }
533   return false;
534 }
```

fill_matrix()

```
void PMLSurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints )  [override], [virtual]
```

Writes the FE system of this PML domain to a provided system matrix and rhs vector using the constraints.

This is part of the default assembly cycle of dealii.

Parameters

matrix	The sytem matrix to write into.
rhs	The right-hand side vector (rhs) to write into.
constraints	The constraints to consider while writing.

Implements BoundaryCondition.

Definition at line 458 of file PMLSurface.cpp.

```
458
                            {
```

```
  CellwiseAssemblyDataPML cell_data(&fe_nedelec, &dof_handler);
  for (; cell_data.cell != cell_data.end_cell; ++cell_data.cell) {
    cell_data.cell->get_dof_indices(cell_data.local_dof_indices);
    cell_data.local_dof_indices = transform_local_to_global_dofs(cell_data.local_dof_indices);
    cell_data.cell_rhs.reinit(cell_data.dofs_per_cell, false);
    cell_data.fe_values.reinit(cell_data.cell);
    cell_data.quadrature_points = cell_data.fe_values.get_quadrature_points();
    std::vector<types::global_dof_index> input_dofs(fe_nedelec.dofs_per_line);
    IndexSet input_dofs_local_set(fe_nedelec.dofs_per_cell);
    std::vector<Position> input_dof_centers(fe_nedelec.dofs_per_cell);
    std::vector<Tensor<1, 3, double» input_dof_dirs(fe_nedelec.dofs_per_cell);
    cell_data.cell_matrix = 0;
    for (unsigned int q_index = 0; q_index < cell_data.n_q_points; ++q_index) {
      Position pos = cell_data.get_position_for_q_index(q_index);
      dealii::Tensor<2,3,ComplexNumber> epsilon = get_pml_tensor_epsilon(pos);
      dealii::Tensor<2,3,double> J = GlobalSpaceTransformation->get_J(pos);

      epsilon = J * epsilon * transpose(J) / GlobalSpaceTransformation->get_det(pos);
      dealii::Tensor<2,3,ComplexNumber> mu = get_pml_tensor_mu(pos);
      mu = invert(J * mu * transpose(J) / GlobalSpaceTransformation->get_det(pos));
      cell_data.prepare_for_current_q_index(q_index, epsilon, mu);
    }
    constraints->distribute_local_to_global(cell_data.cell_matrix, cell_data.cell_rhs,
   cell_data.local_dof_indices,*matrix, *rhs, true);
  }
  matrix->compress(dealii::VectorOperation::add);
}
```

References get_pml_tensor_epsilon(), and FEDomain::transform_local_to_global_dofs().

fill_sparsity_pattern()

```
void PMLSurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * in_constriants )  [override], [virtual]
```

Sets the locations of actually coupling dofs to non-zero in a sparsity pattern so we know to reserve memory for it.

The function also uses a provided constraints object to make this operation more efficient. If, for example, a dof is set to zero, we don't need to store values in the system matrix row and column relating to it.

This is part of the default assembly cycle of dealii.

Parameters

in_dsp	The sparsity pattern to fill with the entries.
in_constriants	Constraints to consider.

Implements BoundaryCondition.

Definition at line 449 of file PMLSurface.cpp.

```
449
     {
450   std::vector<unsigned int> local_indices(fe_nedelec.dofs_per_cell);
451   for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
452     it->get_dof_indices(local_indices);
453     local_indices = transform_local_to_global_dofs(local_indices);
454     in_constraints->add_entries_local_to_global(local_indices, *in_dsp);
455   }
456 }
```

References FEDomain::transform_local_to_global_dofs().

finish_dof_index_initialization()

```
void PMLSurface::finish_dof_index_initialization ( ) [override], [virtual]
```

Iterates over all surfaces of the PML domain and sets the dof indices if the surface is not locally owned.

This function should be nilpotent and only called during setup. It is purely internal and not mathematically relevant.

Reimplemented from BoundaryCondition.

Definition at line 678 of file PMLSurface.cpp.

```
678                                                    {
679   for(unsigned int surf = 0; surf < 6; surf++) {
680     if(surf != b_id && !are_opposing_sites(surf, b_id)) {
681       if(!are_edge_dofs_owned[surf] && Geometry.levels[level].surface_type[surf] !=
       SurfaceType::NEIGHBOR_SURFACE) {
682         DofIndexVector dofs_in_global_numbering =
       Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id);
683         std::vector<InterfaceDofData> local_interface_data = get_dof_association_by_boundary_id(surf);
684         DofIndexVector dofs_in_local_numbering(local_interface_data.size());
685         for(unsigned int i = 0; i < local_interface_data.size(); i++) {
686           dofs_in_local_numbering[i] = local_interface_data[i].index;
687         }
688         set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
689       }
690     }
691   }
692   // Do the same for the inner interface
693   std::vector<InterfaceDofData> global_interface_data =
       Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
694   std::vector<InterfaceDofData> local_interface_data =
       get_dof_association_by_boundary_id(inner_boundary_id);
695   DofIndexVector dofs_in_local_numbering(local_interface_data.size());
696   DofIndexVector dofs_in_global_numbering(local_interface_data.size());
697
698   for(unsigned int i = 0; i < local_interface_data.size(); i++) {
699     dofs_in_local_numbering[i] = local_interface_data[i].index;
700     dofs_in_global_numbering[i] =
       Geometry.levels[level].inner_domain->global_index_mapping[global_interface_data[i].index];
701   }
702   set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
703 }
```

finish_initialization()

```
bool PMLSurface::finish_initialization (
              DofNumber first_own_index ) [override], [virtual]
```

Given a first index, this function numbers the owned dofs starting at that number.

Parameters

first_own_index	The first locally owned index will receive this index.

Returns

true If all indices now have a valid global index.

false There are still indices that are not numbered.

Reimplemented from FEDomain.

Definition at line 716 of file PMLSurface.cpp.

```
716                                                     {
717   std::vector<InterfaceDofData> dofs =
       Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
718   std::vector<InterfaceDofData> own = get_dof_association();
719   std::vector<unsigned int> local_indices, global_indices;
720   if(own.size() != dofs.size()) {
721     std::cout « "Size mismatch in finish initialization: " « own.size() « " != " « dofs.size() «
       std::endl;
722   }
723   for(unsigned int i = 0; i < dofs.size(); i++) {
724     local_indices.push_back(own[i].index);
725     global_indices.push_back(dofs[i].index);
726   }
727   set_non_local_dof_indices(local_indices, global_indices);
728   return FEDomain::finish_initialization(index);
729 }
```

fix_apply_negative_Jacobian_transformation()

```
void PMLSurface::fix_apply_negative_Jacobian_transformation (
              dealii::Triangulation< 3 > * in_tria )
```

Inverts vertex and edge orders to switch the sign of the cell volumes.

Currently, this should not be required.

Parameters

in_tria	The triangulation to perform the operation on.

Definition at line 619 of file PMLSurface.cpp.

```
619                                                                             {
620   double min_z_before = min_z_center_in_triangulation(*in_tria);
621   GridTools::transform(invert_z, *in_tria);
622   double min_z_after = min_z_center_in_triangulation(*in_tria);
623   Tensor<1,3> shift;
624   shift[0] = 0;
625   shift[1] = 0;
626   shift[2] = min_z_before - min_z_after;
627   GridTools::shift(shift, *in_tria);
628 }
```

fraction_of_pml_direction()

```
std::array< double, 3 > PMLSurface::fraction_of_pml_direction (
              Position in_p ) -> std::array<double, 3>
```

Computes the fraction of the PML thickness of the provided position for the computation of sigma for all three space directions.

As described in section 4.4.3, we can ramp up the value of sigma as we approach the outer boundary to reduce the effect of reflections by a profile like eq. (4.45). In this equation, this function computes z/d for all three directions and returns them.

Returns

std::array<double, 3>

Definition at line 328 of file PMLSurface.cpp.

```
328                                                                         {
329   std::array<double, 3> ret;
330   for(unsigned int i = 0; i < 3; i++) {
331     std::pair<double, double> range;
332     switch (i)
333     {
334       case 0:
335         range = Geometry.local_x_range;
336         break;
337       case 1:
338         range = Geometry.local_y_range;
339         break;
340       case 2:
341         range = Geometry.local_z_range;
342         break;
343       default:
344         break;
345     }
346     ret[i] = 0;
347     if(in_p[i] < lower_pml_ranges[i].first) {
348       ret[i] = std::abs(in_p[i] - lower_pml_ranges[i].first) / effective_pml_thickness;
349     }
350     if(in_p[i] > upper_pml_ranges[i].first) {
351       ret[i] = std::abs(in_p[i] - upper_pml_ranges[i].first) / effective_pml_thickness;
352     }
353   }
354   return ret;
355 }
```

Referenced by get_pml_tensor().

get_dof_association()

`std::vector< InterfaceDofData > PMLSurface::get_dof_association ( ) -> std::vector<InterfaceDofData> [override], [virtual]`

Get the degrees of freedom associated with the interface to the inner domain.

Returns

std::vector<InterfaceDofData> Vector of all dofs and their base points.

Implements BoundaryCondition.

Definition at line 324 of file PMLSurface.cpp.

```
324                                                         {
325     return get_dof_association_by_boundary_id(inner_boundary_id);
326 }
```

References get_dof_association_by_boundary_id().

get_dof_association_by_boundary_id()

```
std::vector< InterfaceDofData > PMLSurface::get_dof_association_by_boundary_id (
              BoundaryId in_boundary_id ) -> std::vector<InterfaceDofData>  [override], [virtual]
```

Get the degrees of freedom associated with either the inner domain or another boundary conditions domain.

Parameters

in_boundary_id	The other boundary id. If this is b_id, this returns the same as get_dof_association().

Returns

std::vector<InterfaceDofData> Vector of all dofs and their base points.

Implements BoundaryCondition.

Definition at line 320 of file PMLSurface.cpp.

```
320                                                                                                    {
321   return dof_associations[in_bid];
322 }
```

Referenced by get_dof_association().

get_pml_tensor()

```
dealii::Tensor< 2, 3, ComplexNumber > PMLSurface::get_pml_tensor (
              Position in_p ) -> dealii::Tensor<2,3,ComplexNumber>
```

Internal function that computes the purely geometric transformation tensor.

Returns

dealii::Tensor<2,3,ComplexNumber>

Definition at line 368 of file PMLSurface.cpp.

```
368                                                                              {
369   dealii::Tensor<2,3,ComplexNumber> ret;
370   const std::array<double, 3> fractions = fraction_of_pml_direction(in_p);
371   ComplexNumber sx = {1 , std::pow(fractions[0], GlobalParams.PML_skaling_order) *
       GlobalParams.PML_Sigma_Max};
```

```
ComplexNumber sy = {1 , std::pow(fractions[1], GlobalParams.PML_skaling_order) *
 GlobalParams.PML_Sigma_Max};
ComplexNumber sz = {1 , std::pow(fractions[2], GlobalParams.PML_skaling_order) *
 GlobalParams.PML_Sigma_Max};
for(unsigned int i = 0; i < 3; i++) {
    for(unsigned int j = 0; j < 3; j++) {
        ret[i][j] = 0;
    }
}
ret[0][0] = sy*sz/sx;
ret[1][1] = sx*sz/sy;
ret[2][2] = sx*sy/sz;
return ret;
}
```

References fraction_of_pml_direction().

Referenced by get_pml_tensor_epsilon(), and get_pml_tensor_mu().

get_pml_tensor_epsilon()

```
dealii::Tensor< 2, 3, ComplexNumber > PMLSurface::get_pml_tensor_epsilon (
              Position in_p ) -> dealii::Tensor<2,3,ComplexNumber>
```

Get the PML material tensor ϵ_p for a given position.

This is ϵ_p in **??**.

Parameters

in_p	The location to compute the material tensor at

Returns

dealii::Tensor<2,3,ComplexNumber> The material tensor ϵ_p for a UPML medium at a given location.

Definition at line 357 of file PMLSurface.cpp.

```
357                                                                               {
358     dealii::Tensor<2,3,ComplexNumber> ret = get_pml_tensor(in_p);
359     ret *= Geometry.get_epsilon_for_point(in_p);
360     return ret;
361 }
```

References get_pml_tensor().

Referenced by fill_matrix(), and output_results().

get_pml_tensor_mu()

```
dealii::Tensor< 2, 3, ComplexNumber > PMLSurface::get_pml_tensor_mu (
              Position in_p ) -> dealii::Tensor<2,3,ComplexNumber>
```

Get the PML material tensor μ_p for a given position.

This is μ_p in **??**.

Parameters

in_p	The location to compute the material tensor at

Returns

dealii::Tensor<2,3,ComplexNumber> The material tensor μ_p for a UPML medium at a given location.

Definition at line 363 of file PMLSurface.cpp.

```
363                                                             {
364     dealii::Tensor<2,3,ComplexNumber> ret = get_pml_tensor(in_p);
365     return ret;
366 }
```

References get_pml_tensor().

init_fe()

`void PMLSurface::init_fe ( )`

Initializes all the parts of the finite element loop like the dof handler and the finite element object that provides shape functions.

See the deal.ii documentation on this since it is oriented on their structure of fe computations.

Definition at line 141 of file PMLSurface.cpp.

```
141                         {
142     dof_handler.reinit(triangulation);
143     dof_handler.distribute_dofs(fe_nedelec);
144     dof_counter = dof_handler.n_dofs();
145 }
```

Referenced by initialize().

initialize()

`void PMLSurface::initialize ( ) [override], [virtual]`

Initializes the data structures to reserve memory.

This function is part of the default dealii assembly loop.

Implements BoundaryCondition.

Definition at line 247 of file PMLSurface.cpp.

```
247                            {
248   prepare_mesh();
249   init_fe();
250   prepare_dof_associations();
251 }
```

References init_fe(), prepare_dof_associations(), and prepare_mesh().

is_point_at_boundary() [1/2]

```
bool PMLSurface::is_point_at_boundary (
            Position ,
            BoundaryId  )
```

Checks if the provided coordinate is at the provided boundary.

Returns

true if the point is at that boundary.

false if not.

is_point_at_boundary() [2/2]

```
bool PMLSurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid )  [override], [virtual]
```

Checks if a 2D position of the surface mesh is also at another boundary, i.e.

an edge of the inner domain.

Parameters

in_p	The position to check for.
in_bid	The boundary Id we check for.

Returns

true If the provided position is at that boundary id.

false If not.

Implements BoundaryCondition.

Definition at line 224 of file PMLSurface.cpp.

```
{
  return false;
}
```

is_position_at_boundary()

```
bool PMLSurface::is_position_at_boundary (
            const Position in_p,
            const BoundaryId in_bid )
```

This function and the next are used to color the surfaces of the PML domain.

See the class description for details.

Parameters

in_p	
in_bid	

Returns

true

false

Definition at line 147 of file PMLSurface.cpp.

```
147                                                                                  {
148   switch (in_bid)
149   {
150     case 0:
151       if(std::abs(in_p[0] - x_range.first) < FLOATING_PRECISION) return true;
152       break;
153     case 1:
154       if(std::abs(in_p[0] - x_range.second) < FLOATING_PRECISION) return true;
155       break;
156     case 2:
157       if(std::abs(in_p[1] - y_range.first) < FLOATING_PRECISION) return true;
158       break;
159     case 3:
160       if(std::abs(in_p[1] - y_range.second) < FLOATING_PRECISION) return true;
161       break;
162     case 4:
163       if(std::abs(in_p[2] - z_range.first) < FLOATING_PRECISION) return true;
164       break;
165     case 5:
166       if(std::abs(in_p[2] - z_range.second) < FLOATING_PRECISION) return true;
167       break;
168   }
169   return false;
170 }
```

is_position_at_extended_boundary()

```
bool PMLSurface::is_position_at_extended_boundary (
              const Position in_p,
              const BoundaryId in_bid )
```

This function and the previous one a re used to color the surfaces of the PML domain.

See the class description for details.

Parameters

in_p	
in_bid	

Returns

true

false

Definition at line 172 of file PMLSurface.cpp.

```
172                                                                                    {
173   if(std::abs(in_p[b_id / 2] - surface_coordinate) < FLOATING_PRECISION) {
174 
175     switch(b_id / 2) {
176       case 0:
177         return false;
178         break;
179       case 1:
180         if((in_bid / 2) == 0) {
181           if(in_p[0] < Geometry.local_x_range.first && in_bid == 0) {
182 
183             return true;
184           }
185           if(in_p[0] > Geometry.local_x_range.second && in_bid == 1) {
186             return true;
187           }
188           return false;
189         } else {
190           return false;
191         }
192         break;
193       case 2:
194         if(in_bid == 3) {
195           return in_p[1] > Geometry.local_y_range.second;
196         }
197         if(in_bid == 2) {
198           return in_p[1] < Geometry.local_y_range.first;
199         }
200         if(in_bid == 1) {
201           bool not_y = in_p[1] <= Geometry.local_y_range.second && in_p[1] >=
       Geometry.local_y_range.first;
202           if(not_y) {
203             return in_p[0] > Geometry.local_x_range.second;
204           } else {
205             return false;
206           }
207         }
208         if(in_bid == 0) {
209           bool not_y = in_p[1] <= Geometry.local_y_range.second && in_p[1] >=
       Geometry.local_y_range.first;
210           if(not_y) {
211             return in_p[0] < Geometry.local_x_range.first;
212           } else {
213             return false;
214           }
215         }
216         break;
217     }
218     return false;
219   } else {
220     return b_id == in_bid;
221   }
222 }
```

make_constraints()

`Constraints PMLSurface::make_constraints ( ) -> Constraints  [override], [virtual]`

For this method we use PEC boundary conditions on the outside of the PML domain.

This function writes the dof constraints representing those PEC constraints to an empty Affine Constraints object and returns it.

As described in section 4.4.3, we apply PEC boundary conditions, i.e. dirichlet zero values for the tangential trace on the surface of the PML domain that is facing outward. The affined constraints object we build here can be used to condense the system matrix to set the constrained dofs to the right value.

Returns

Constraints The constraint object to be used anywhere in the code to condense a system or to update vector values.

Reimplemented from BoundaryCondition.

Definition at line 731 of file PMLSurface.cpp.

```
731                                              {
732   IndexSet global_indices = IndexSet(Geometry.levels[level].n_total_level_dofs);
733   global_indices.add_range(0, Geometry.levels[level].n_total_level_dofs);
734   Constraints ret(global_indices);
735   std::vector<InterfaceDofData> dofs = get_dof_association_by_boundary_id(outer_boundary_id);
736   for(auto dof : dofs) {
737         const unsigned int local_index = dof.index;
738         const unsigned int global_index = global_index_mapping[local_index];
739         ret.add_line(global_index);
740         ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
741     }
742   return ret;
743 }
```

mg_process_corner()

```
bool PMLSurface::mg_process_corner (
            dealii::Triangulation< 3 > * current_list,
            BoundaryId first_bid,
            BoundaryId second_bid )
```

Same as above but for edges.

Therefore requires two boundary ids.

Parameters

return_pointer	The pointer to be used to store the extension triangulation in.
first_bid	
second_bid	

Returns

true This corner requires an extension domain, i.e. there are PML boundaries on the other two boundaries and the extension is locally owned.

false Either no domain is required or it is not locally owned.

Definition at line 836 of file PMLSurface.cpp.

```
836
          {
837   if(b_id == 4 || b_id == 5) {
838     bool generate_this_part = Geometry.levels[level].is_surface_truncated[first_bid] &&
     Geometry.levels[level].is_surface_truncated[second_bid];
839     if(generate_this_part) {
840       // Do the generation.
841
     GridGenerator::subdivided_hyper_cube(*tria,GlobalParams.PML_N_Layers,0,GlobalParams.PML_thickness);
842       dealii::Tensor<1,3> shift;
843       bool lower_x = first_bid == 0 || second_bid == 0;
844       bool lower_y = first_bid == 2 || second_bid == 2;
845       if(lower_x) {
```

```
      shift[0] = - GlobalParams.PML_thickness + Geometry.local_x_range.first;
    } else {
      shift[0] = Geometry.local_x_range.second;
    }
    if(lower_y) {
      shift[1] = - GlobalParams.PML_thickness + Geometry.local_y_range.first;
    } else {
      shift[1] = Geometry.local_y_range.second;
    }
    if(b_id == 4) {
      shift[2] = - GlobalParams.PML_thickness + Geometry.local_z_range.first;
    } else {
      shift[2] = Geometry.local_z_range.second;
    }
    dealii::GridTools::shift(shift, *tria);
    return true;
  }
}
return false;
}
```

mg_process_edge()

```
bool PMLSurface::mg_process_edge (
        dealii::Triangulation< 3 > * return_pointer,
        BoundaryId b_id )
```

Checks if the PML requires an extension domain towards the boundary with BoundaryId b_id and, if so, creates a mesh of that extension and provides it in the pointer argument.

Parameters

return_pointer	The pointer to be used to store the extension triangulation in.
b_id	The boundary toward which we are checking for an extension

Returns

true The PML domain requires extension here and the extension is stored in return_pointer

false No extension is required.

Definition at line 774 of file PMLSurface.cpp.

```
774 {
775   // This line checks if the domain even exists
776   bool domain_exists = Geometry.levels[level].is_surface_truncated[other_bid];
777   // the next step checks if this boundary generates it. For b_id 4 and 5, this is always the case. For 2 and 3 it is only true if the other b_id
778   bool is_owned = false;
779   if(b_id == 4 || b_id == 5) {
780     is_owned = true;
781   }
782   if(b_id == 2 || b_id == 3) {
783     is_owned = (other_bid == 0 || other_bid == 1);
784   }
785   if(domain_exists && is_owned) {
786     std::vector<unsigned int> subdivisions(3);
787     Position p1, p2;
788     if(b_id / 2 != 0 && other_bid /2 != 0) {
789       subdivisions[0] = GlobalParams.Cells_in_x;
790       p1[0] = Geometry.local_x_range.first;
791       p2[0] = Geometry.local_x_range.second;
792     } else {
```

```
      subdivisions[0] = GlobalParams.PML_N_Layers;
      if(b_id == 0 || other_bid == 0) {
        p1[0] = Geometry.local_x_range.first - GlobalParams.PML_thickness;
        p2[0] = Geometry.local_x_range.first;
      } else {
        p1[0] = Geometry.local_x_range.second;
        p2[0] = Geometry.local_x_range.second + GlobalParams.PML_thickness;
      }
    }
    if(b_id / 2 != 1 && other_bid /2 != 1) {
      subdivisions[1] = GlobalParams.Cells_in_y;
      p1[1] = Geometry.local_y_range.first;
      p2[1] = Geometry.local_y_range.second;
    } else {
      subdivisions[1] = GlobalParams.PML_N_Layers;
      if(b_id == 2 || other_bid == 2) {
        p1[1] = Geometry.local_y_range.first - GlobalParams.PML_thickness;
        p2[1] = Geometry.local_y_range.first;
      } else {
        p1[1] = Geometry.local_y_range.second;
        p2[1] = Geometry.local_y_range.second + GlobalParams.PML_thickness;
      }
    }
    if(b_id / 2 != 2 && other_bid /2 != 2) {
      subdivisions[2] = GlobalParams.Cells_in_z;
      p1[2] = Geometry.local_z_range.first;
      p2[2] = Geometry.local_z_range.second;
    } else {
      subdivisions[2] = GlobalParams.PML_N_Layers;
      if(b_id == 4 || other_bid == 4) {
        p1[2] = Geometry.local_z_range.first - GlobalParams.PML_thickness;
        p2[2] = Geometry.local_z_range.first;
      } else {
        p1[2] = Geometry.local_z_range.second;
        p2[2] = Geometry.local_z_range.second + GlobalParams.PML_thickness;
      }
    }
    dealii::GridGenerator::subdivided_hyper_rectangle(*tria,subdivisions, p1, p2);
    return true;
  }
  return false;
}
```

n_cells()

`unsigned int PMLSurface::n_cells ( )  [override], [virtual]`

Counts the number of local cells.

Returns

unsigned int

Reimplemented from BoundaryCondition.

Definition at line 867 of file PMLSurface.cpp.

```
867                                {
868   return triangulation.n_active_cells();
869 }
```

Referenced by output_results().

output_results()

```
std::string PMLSurface::output_results (
            const dealii::Vector< ComplexNumber > & solution_vector,
            std::string filename )  [override], [virtual]
```

Writes an output file for paraview of the solution provided projected onto the local mesh.

Parameters

solution_vector	The fe solution vector to be used.
filename	Fragment of the filename to be used (this will be extended by process and boundary ids for uniqueness)

Returns

The filename of the generated file

Implements BoundaryCondition.

Definition at line 630 of file PMLSurface.cpp.

```
630                                                                                            {
631   dealii::DataOut<3> data_out;
632   data_out.attach_dof_handler(dof_handler);
633
634   dealii::Vector<ComplexNumber> zero = dealii::Vector<ComplexNumber>(in_data.size());
635   for(unsigned int i = 0; i < in_data.size(); i++) {
636     zero[i] = 0;
637   }
638
639   const unsigned int n_cells = dof_handler.get_triangulation().n_cells();
640   dealii::Vector<double> eps_abs(n_cells);
641   unsigned int counter = 0;
642   for(auto it = dof_handler.begin(); it != dof_handler.end(); it++) {
643     Position p = it->center();
644     MaterialTensor epsilon = get_pml_tensor_epsilon(p);
645     eps_abs[counter] = epsilon.norm();
646     counter++;
647   }
648
649   data_out.add_data_vector(in_data, "Solution");
650   data_out.add_data_vector(eps_abs, "Epsilon");
651    dealii::Vector<double> index_x(n_cells), index_y(n_cells), index_z(n_cells);
652   for(unsigned int i = 0; i < n_cells; i++) {
653     index_x[i] = GlobalParams.Index_in_x_direction;
654     index_y[i] = GlobalParams.Index_in_y_direction;
655     index_z[i] = GlobalParams.Index_in_z_direction;
656   }
657   data_out.add_data_vector(index_x, "IndexX");
```

```
  data_out.add_data_vector(index_y, "IndexY");
  data_out.add_data_vector(index_z, "IndexZ");
  data_out.add_data_vector(zero, "Exact_Solution");
  data_out.add_data_vector(zero, "SolutionError");
  const std::string filename = GlobalOutputManager.get_numbered_filename(in_filename + "-" +
   std::to_string(b_id) + "-", GlobalParams.MPI_Rank, "vtu");
  std::ofstream outputvtu(filename);
  data_out.build_patches();
  data_out.write_vtu(outputvtu);
  return filename;
}
```

References get_pml_tensor_epsilon(), and n_cells().

set_boundary_ids()

`void PMLSurface::set_boundary_ids ( )`

Color the mesh surfaces.

This function updates the local mesh to set the boundary ids of all outside faces.

Definition at line 536 of file PMLSurface.cpp.

```
536                               {
537   std::array<unsigned int, 6> countrers;
538   for(unsigned int i= 0; i < 6; i++) {
539     countrers[i] = 0;
540   }
541   // first set all to outer_boundary_id
542   for(auto it = triangulation.begin(); it != triangulation.end(); it++) {
543     for(unsigned int face = 0; face < 6; face ++) {
544       if(it->face(face)->at_boundary()) {
545         it->face(face)->set_all_boundary_ids(outer_boundary_id);
546         countrers[outer_boundary_id]++;
547       }
548     }
549   }
550   // then locate all the faces connecting to the inner domain
551   for(auto it = triangulation.begin(); it != triangulation.end(); it++) {
552     for(unsigned int face = 0; face < 6; face ++) {
553       if(it->face(face)->at_boundary()) {
554         Position p = it->face(face)->center();
555         // Have to use outer_boundary_id here because direction 4 of the pml (-z) is at the boundary 5
       of the inner domain (+z)
556         bool is_located_properly = std::abs(p[b_id/2] -
       get_surface_coordinate_for_bid(outer_boundary_id)) < FLOATING_PRECISION;
557         if((b_id / 2) != 0) {
558           is_located_properly &= p[0] > Geometry.local_x_range.first + FLOATING_PRECISION;
559           is_located_properly &= p[0] < Geometry.local_x_range.second - FLOATING_PRECISION;
560         }
561         if((b_id / 2) != 1) {
562           is_located_properly &= p[1] > Geometry.local_y_range.first + FLOATING_PRECISION;
563           is_located_properly &= p[1] < Geometry.local_y_range.second - FLOATING_PRECISION;
564         }
565         if((b_id / 2) != 2) {
566           is_located_properly &= p[2] > Geometry.local_z_range.first + FLOATING_PRECISION;
567           is_located_properly &= p[2] < Geometry.local_z_range.second - FLOATING_PRECISION;
568         }
569         if(is_located_properly) {
570           it->face(face)->set_all_boundary_ids(inner_boundary_id);
571           countrers[inner_boundary_id]++;
572         }
573       }
574     }
575   }
576   // then check all of the other boundary ids.
577   for(auto it = triangulation.begin(); it != triangulation.end(); it++) {
578     for(unsigned int face = 0; face < 6; face ++) {
```

```
    if(it->face(face)->at_boundary()) {
      Position p = it->face(face)->center();
      for(unsigned int i = 0; i < 6; i++) {
        if(i != b_id && !are_opposing_sites(i,b_id)) {
          bool is_at_boundary = false;
          if(extend_mesh_in_direction(i)) {
            is_at_boundary = is_position_at_extended_boundary(p,i);
          } else {
            is_at_boundary = is_position_at_boundary(p,i);
          }
          if(is_at_boundary) {
            it->face(face)->set_all_boundary_ids(i);
            countrers[i]++;
          }
        }
      }
    }
  }
}
//std::cout « "On " « GlobalParams.MPI_Rank « " and " « b_id « " inner " « inner_boundary_id « " and outer " « outer_boundary_id « " and ["«countrers[0]« (extend_mesh_in_direction(0)? "*": "")«","«countrers[1]« (extend_mesh_in_direction(1)? "*": "")«","«countrers[2]« (extend_mesh_in_direction(2)? "*": "")«","«countrers[3]« (extend_mesh_in_direction(3)? "*": "")«","«countrers[4]«","«countrers[5]«"]"«std::endl;
}
```

set_mesh_boundary_ids()

`void PMLSurface::set_mesh_boundary_ids ( )`

Set the mesh boundary ids by checking if faces and edges are at certain boundaries.

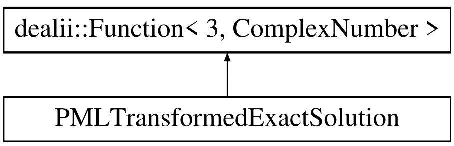
After this is called, we can retrieve dofs by boundary id.

The documentation for this class was generated from the following files:

- Code/BoundaryCondition/PMLSurface.h
- Code/BoundaryCondition/PMLSurface.cpp

61 PMLTransformedExactSolution Class Reference

Inheritance diagram for PMLTransformedExactSolution:



Public Member Functions

- **PMLTransformedExactSolution** (BoundaryId in_main_id, double in_additional_coordinate)
- std::vector< std::string > **split** (std::string) const
- ComplexNumber **value** (const Position &p, const unsigned int component) const
- void **vector_value** (const Position &p, dealii::Vector< ComplexNumber > &value) const

- dealii::Tensor< 1, 3, ComplexNumber > curl (const Position &in_p) const
- dealii::Tensor< 1, 3, ComplexNumber > **val** (const Position &in_p) const
- std::array< double, 3 > **fraction_of_pml_direction** (const Position &in_p) const
- double **compute_scaling_factor** (const Position &in_p) const

61.1 Detailed Description

Definition at line 12 of file PMLTransformedExactSolution.h.

61.2 Member Function Documentation

curl()

```
dealii::Tensor< 1, 3, ComplexNumber > PMLTransformedExactSolution::curl (
            const Position & in_p ) const
```

NumericVectorLocal curls = base_solution->curl(in_p); double scaling_factor = compute_scaling_factor(in_p); for(unsigned int i = 0; i < 3; i++) { ret[i] *= scaling_factor; }

Definition at line 48 of file PMLTransformedExactSolution.cpp.

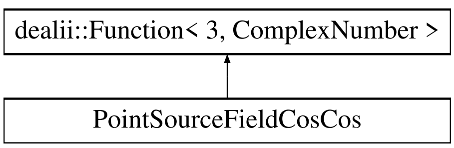
```
48                                                                             {
49   dealii::Tensor<1, 3, ComplexNumber> ret;
50   /**
51   NumericVectorLocal curls = base_solution->curl(in_p);
52   double scaling_factor = compute_scaling_factor(in_p);
53   for(unsigned int i = 0; i < 3; i++) {
54     ret[i] *= scaling_factor;
55   }
56   **/
57   return ret;
58 }
```

The documentation for this class was generated from the following files:

- Code/Solutions/PMLTransformedExactSolution.h
- Code/Solutions/PMLTransformedExactSolution.cpp

62 PointSourceFieldCosCos Class Reference

Inheritance diagram for PointSourceFieldCosCos:



Public Member Functions

- ComplexNumber **value** (const Position &p, const unsigned int component=0) const override
- void **vector_value** (const Position &p, NumericVectorLocal &vec) const override
- void **vector_curl** (const Position &p, NumericVectorLocal &vec)

62.1 Detailed Description

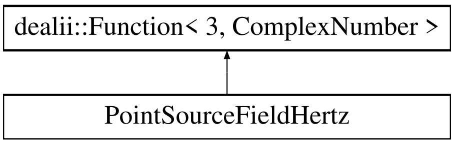
Definition at line 30 of file PointSourceField.h.

The documentation for this class was generated from the following files:

- Code/Helpers/PointSourceField.h
- Code/Helpers/PointSourceField.cpp

63 PointSourceFieldHertz Class Reference

Inheritance diagram for PointSourceFieldHertz:



Public Member Functions

- **PointSourceFieldHertz** (double in_k=1.0)
- void **set_cell_diameter** (double diameter)
- ComplexNumber **value** (const Position &p, const unsigned int component=0) const override
- void **vector_value** (const Position &p, NumericVectorLocal &vec) const override
- void **vector_curl** (const Position &p, NumericVectorLocal &vec)

Public Attributes

- double **k** = 1
- const ComplexNumber **ik**
- double **cell_diameter** = 0.01

63.1 Detailed Description

Definition at line 17 of file PointSourceField.h.

The documentation for this class was generated from the following files:

- Code/Helpers/PointSourceField.h

- Code/Helpers/PointSourceField.cpp

64 PointVal Class Reference

Old class that was used for the interpolation of input signals.

```
#include <PointVal.h>
```

Public Member Functions

- **PointVal** (double, double, double, double, double, double)
- void **set** (double, double, double, double, double, double)
- void **rescale** (double)

Public Attributes

- ComplexNumber **Ex**
- ComplexNumber **Ey**
- ComplexNumber **Ez**

64.1 Detailed Description

Old class that was used for the interpolation of input signals.

Definition at line 20 of file PointVal.h.

The documentation for this class was generated from the following files:

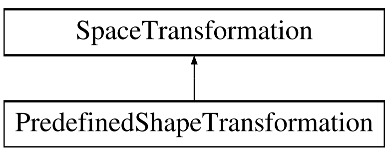
- Code/Helpers/PointVal.h
- Code/Helpers/PointVal.cpp

65 PredefinedShapeTransformation Class Reference

This class is used to describe the hump examples.

```
#include <PredefinedShapeTransformation.h>
```

Inheritance diagram for PredefinedShapeTransformation:



Public Member Functions

- Position math_to_phys (Position coord) const

  Transforms a coordinate in the mathematical coord system to physical ones.
- Position phys_to_math (Position coord) const

  Transforms a coordinate in the physical coord system to mathematical ones.
- dealii::Tensor< 2, 3, ComplexNumber > get_Tensor (Position &coordinate)

  Get the transformation tensor at a given location.
- dealii::Tensor< 2, 3, double > get_Space_Transformation_Tensor (Position &coordinate)

  Get the real part of the transformation tensor at a given location.
- Tensor< 2, 3, double > get_J (Position &) override

  Compute the Jacobian of the current transformation at a given location.
- Tensor< 2, 3, double > get_J_inverse (Position &) override

  Compute the Jacobian of the current transformation at a given location and invert it.
- void estimate_and_initialize ()

  At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.
- double get_m (double in_z) const

  Returns the shift for a system-coordinate;.
- double get_v (double in_z) const

  Returns the tilt for a system-coordinate;.
- void Print () const

  Console output of the current Waveguide Structure.

Public Attributes

- std::vector< Sector< 2 > > case_sectors

  This member contains all the Sectors who, as a sum, form the complete Waveguide.

65.1 Detailed Description

This class is used to describe the hump examples.

Definition at line 18 of file PredefinedShapeTransformation.h.

65.2 Member Function Documentation

estimate_and_initialize()

`void PredefinedShapeTransformation::estimate_and_initialize ( ) [virtual]`

At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.

Therefore the values for the degrees of freedom need to be estimated. This function sets all variables to appropiate values and estimates an appropriate shape based on averages and a polynomial interpolation of the boundary conditions on the shape.

Implements SpaceTransformation.

Definition at line 52 of file PredefinedShapeTransformation.cpp.

```
52                                                               {
53    print_info("PredefinedShapeTransformation::estimate_and_initialize", "Start");
54    Sector<2> the_first(true, false, GlobalParams.sd.z[0], GlobalParams.sd.z[1]);
55    the_first.set_properties_force(GlobalParams.sd.m[0], GlobalParams.sd.m[1],
56                                   GlobalParams.sd.v[0], GlobalParams.sd.v[1]);
57    case_sectors.push_back(the_first);
58    for (int i = 1; i < GlobalParams.sd.Sectors - 2; i++) {
59        Sector<2> intermediate(false, false, GlobalParams.sd.z[i], GlobalParams.sd.z[i + 1]);
60        intermediate.set_properties_force(
61            GlobalParams.sd.m[i], GlobalParams.sd.m[i + 1], GlobalParams.sd.v[i],
62            GlobalParams.sd.v[i + 1]);
63        case_sectors.push_back(intermediate);
64    }
65    Sector<2> the_last(false, true,
66                       GlobalParams.sd.z[GlobalParams.sd.Sectors - 2],
67                       GlobalParams.sd.z[GlobalParams.sd.Sectors - 1]);
68    the_last.set_properties_force(
69        GlobalParams.sd.m[GlobalParams.sd.Sectors - 2],
70        GlobalParams.sd.m[GlobalParams.sd.Sectors - 1],
71        GlobalParams.sd.v[GlobalParams.sd.Sectors - 2],
72        GlobalParams.sd.v[GlobalParams.sd.Sectors - 1]);
73    case_sectors.push_back(the_last);
74    if(GlobalParams.MPI_Rank == 0) {
75      for (unsigned int i = 0; i < case_sectors.size(); i++) {
76        std::string msg_lower = "Layer at z: " + std::to_string(case_sectors[i].z_0) + "(m: " +
      std::to_string(case_sectors[i].get_m(0.0)) + " v: " + std::to_string(case_sectors[i].get_v(0.0)) +
      ")";
77        print_info("PredefinedShapeTransformation::estimate_and_initialize", msg_lower);
78      }
79      std::string msg_last = "Layer at z: " + std::to_string(case_sectors[case_sectors.size()-1].z_1) +
      "(m: " + std::to_string(case_sectors[case_sectors.size()-1].get_m(1.0)) + " v: " +
      std::to_string(case_sectors[case_sectors.size()-1].get_v(1.0)) + ")";
80
81    }
82    print_info("PredefinedShapeTransformation::estimate_and_initialize", "End");
83 }
```

get_J()

```
Tensor< 2, 3, double > PredefinedShapeTransformation::get_J (
              Position & ) [override], [virtual]
```

Compute the Jacobian of the current transformation at a given location.

Returns

Tensor<2,3,double> Jacobian matrix at the given location.

Reimplemented from SpaceTransformation.

Definition at line 109 of file PredefinedShapeTransformation.cpp.

```
109                                                                 {
110   Tensor<2,3,double> ret = I;
111   ret[1][2] = - get_v(in_p[2]);
112   return ret;
```

```
113 }
```

References get_v().

Referenced by get_J_inverse(), and get_Space_Transformation_Tensor().

get_J_inverse()

```
Tensor< 2, 3, double > PredefinedShapeTransformation::get_J_inverse (
            Position & ) [override], [virtual]
```

Compute the Jacobian of the current transformation at a given location and invert it.

Returns

Tensor<2,3,double> Inverse of the jacobian matrix at the given location.

Reimplemented from SpaceTransformation.

Definition at line 115 of file PredefinedShapeTransformation.cpp.

```
115                                                                  {
116    Tensor<2,3,double> ret = get_J(in_p);
117    return invert(ret);
118 }
```

References get_J().

get_Space_Transformation_Tensor()

```
Tensor< 2, 3, double > PredefinedShapeTransformation::get_Space_Transformation_Tensor (
            Position & ) [virtual]
```

Get the real part of the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 100 of file PredefinedShapeTransformation.cpp.

```
100                                                                    {
101    Tensor<2, 3, double> J_loc = get_J(position);
102    Tensor<2, 3, double> ret;
103    ret[0][0] = 1;
104    ret[1][1] = 1;
105    ret[2][2] = 1;
106    return (J_loc * ret * transpose(J_loc)) / determinant(J_loc);
107 }
```

References get_J().

Referenced by get_Tensor().

get_Tensor()

```
Tensor< 2, 3, ComplexNumber > PredefinedShapeTransformation::get_Tensor (
            Position & ) [virtual]
```

Get the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 47 of file PredefinedShapeTransformation.cpp.

```
47                                                                  {
48   return get_Space_Transformation_Tensor(position);
49 }
```

References get_Space_Transformation_Tensor().

math_to_phys()

```
Position PredefinedShapeTransformation::math_to_phys (
            Position coord ) const [virtual]
```

Transforms a coordinate in the mathematical coord system to physical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the mathematical system

Returns

Position Coordinate in the physical system

Implements SpaceTransformation.

Definition at line 26 of file PredefinedShapeTransformation.cpp.

```
26                                                                            {
27   Position ret;
28   std::pair<int, double> sec = Z_to_Sector_and_local_z(coord[2]);
29   double m = case_sectors[sec.first].get_m(sec.second);
30   ret[0] = coord[0];
31   ret[1] = coord[1] + m;
32   ret[2] = coord[2];
33   return ret;
34 }
```

References case_sectors, and SpaceTransformation::Z_to_Sector_and_local_z().

phys_to_math()

```
Position PredefinedShapeTransformation::phys_to_math (
            Position coord ) const [virtual]
```

Transforms a coordinate in the physical coord system to mathematical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the physical system

Returns

Position Coordinate in the mathematical system

Implements SpaceTransformation.

Definition at line 36 of file PredefinedShapeTransformation.cpp.

```
36                                                                {
37   Position ret;
38   std::pair<int, double> sec = Z_to_Sector_and_local_z(coord[2]);
39   double m = case_sectors[sec.first].get_m(sec.second);
40   ret[0] = coord[0];
41   ret[1] = coord[1] - m;
42   ret[2] = coord[2];
43   return ret;
44 }
```

References case_sectors, and SpaceTransformation::Z_to_Sector_and_local_z().

65.3 Member Data Documentation

case_sectors

`std::vector<Sector<2> > PredefinedShapeTransformation::case_sectors`

This member contains all the Sectors who, as a sum, form the complete Waveguide.

These Sectors are a partition of the simulated domain.

Definition at line 41 of file PredefinedShapeTransformation.h.

Referenced by get_m(), get_v(), math_to_phys(), and phys_to_math().

The documentation for this class was generated from the following files:

- Code/SpaceTransformations/PredefinedShapeTransformation.h
- Code/SpaceTransformations/PredefinedShapeTransformation.cpp

66 RayAngelingData Struct Reference

Public Attributes

- bool **is_x_angled** = false
- bool **is_y_angled** = false
- Position2D **position_of_base_point**

66.1 Detailed Description

Definition at line 121 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

67 RectangularMode Class Reference

Legacy code.

`#include <RectangularMode.h>`

Public Member Functions

- void **assemble_system** ()
- void **make_mesh** ()
- void **make_boundary_conditions** ()
- void **output_solution** ()
- void **run** ()
- void solve ()
- void **SortDofsDownstream** ()
- IndexSet **get_dofs_for_boundary_id** (types::boundary_id)
- std::vector< InterfaceDofData > **get_surface_dof_vector_for_boundary_id** (unsigned int b_id)

Static Public Member Functions

- static auto **compute_epsilon_for_Position** (Position in_position) -> double

Public Attributes

- double **beta**
- unsigned int **n_dofs_total**
- unsigned int **n_eigenfunctions** = 1
- std::vector< ComplexNumber > **eigenvalues**
- std::vector< PETScWrappers::MPI::Vector > **eigenfunctions**
- std::vector< DofNumber > **surface_first_dofs**
- std::array< std::shared_ptr< HSIESurface >, 4 > **surfaces**
- dealii::FE_NedelecSZ< 3 > **fe**
- Constraints **constraints**
- Constraints **periodic_constraints**
- Triangulation< 3 > **triangulation**

- DoFHandler< 3 > **dof_handler**
- SparsityPattern **sp**
- PETScWrappers::SparseMatrix **mass_matrix**
- PETScWrappers::SparseMatrix **stiffness_matrix**
- NumericVectorDistributed **rhs**
- NumericVectorDistributed **solution**
- const double **layer_thickness**
- const double **lambda**

67.1 Detailed Description

Legacy code.

This object was intended to become a mode solver but numerical results have shown that an exact computation is not required. It is simpler to use provided mode profiles that are computed offline.

Definition at line 61 of file RectangularMode.h.

67.2 Member Function Documentation

solve()

```
void RectangularMode::solve ( )
```

eigensolver.solve(stiffness_matrix, mass_matrix, eigenvalues, eigenfunctions, n_eigenfunctions);

Definition at line 281 of file RectangularMode.cpp.

```
281                         {
282    print_info("RectangularProblem::solve", "Start");
283    dealii::SolverControl                   solver_control(n_dofs_total, 1e-6);
284    // dealii::SLEPcWrappers::SolverKrylovSchur eigensolver(solver_control);
285    IndexSet own_dofs(n_dofs_total);
286    own_dofs.add_range(0, n_dofs_total);
287    eigenfunctions.resize(n_eigenfunctions);
288    for (unsigned int i = 0; i < n_eigenfunctions; ++i)
289      eigenfunctions[i].reinit(own_dofs, MPI_COMM_SELF);
290    eigenvalues.resize(n_eigenfunctions);
291    // eigensolver.set_which_eigenpairs(EPS_SMALLEST_MAGNITUDE);
292    // eigensolver.set_problem_type(EPS_GNHEP);
293    print_info("RectangularProblem::solve", "Starting solution for a system with " +
        std::to_string(n_dofs_total) + " degrees of freedom.");
294    /**
295    eigensolver.solve(stiffness_matrix,
296                      mass_matrix,
297                      eigenvalues,
298                      eigenfunctions,
299                      n_eigenfunctions);
300                      **/
301    for(unsigned int i =0 ; i < n_eigenfunctions; i++) {
302      // constraints.distribute(eigenfunctions[0]);
303      eigenfunctions[i] /= eigenfunctions[i].linfty_norm();
304    }
305    print_info("RectangularProblem::solve", "End");
306 }
```

The documentation for this class was generated from the following files:

- Code/ModalComputations/RectangularMode.h
- Code/ModalComputations/RectangularMode.cpp

68 ResidualOutputGenerator Class Reference

Public Member Functions

- **ResidualOutputGenerator** (std::string in_name, std::string in_title, unsigned int in_rank_in_sweep, unsigned int in_level, int in_parent_sweeping_rank)
- void **push_value** (double value)
- void **close_current_series** ()
- void **new_series** (std::string name)
- void **write_gnuplot_file** ()
- void **run_gnuplot** ()
- void **write_residual_statement_to_console** ()

68.1 Detailed Description

Definition at line 5 of file ResidualOutputGenerator.h.

The documentation for this class was generated from the following files:

- Code/OutputGenerators/Images/ResidualOutputGenerator.h
- Code/OutputGenerators/Images/ResidualOutputGenerator.cpp

69 SampleShellPC Struct Reference

Public Attributes

- NonLocalProblem ∗ **parent**

69.1 Detailed Description

Definition at line 214 of file HierarchicalProblem.h.

The documentation for this struct was generated from the following file:

- Code/Hierarchy/HierarchicalProblem.h

70 Sector< Dofs_Per_Sector > Class Template Reference

Sectors are used, to split the computational domain into chunks, whose degrees of freedom are likely coupled.

```
#include <Sector.h>
```

Public Member Functions

- Sector (bool in_left, bool in_right, double in_z_0, double in_z_1)

  Constructor of the Sector class, that takes all important properties as an input property.
- dealii::Tensor< 2, 3, double > TransformationTensorInternal (double in_x, double in_y, double in_z) const

  This method gets called from the WaveguideStructure object used in the simulation.
- void set_properties (double m_0, double m_1, double r_0, double r_1)

  This function is used during the optimization-operation to update the properties of the space-transformation.
- void **set_properties** (double m_0, double m_1, double r_0, double r_1, double v_0, double v_1)
- void set_properties_force (double m_0, double m_1, double r_0, double r_1)

  This function is the same as set_properties with the difference of being able to change the values of the input- and output boundary.
- void **set_properties_force** (double m_0, double m_1, double r_0, double r_1, double v_0, double v_1)
- double getQ1 (double) const

  The values of Q1, Q2 and Q3 are needed to compute the solution in real coordinates from the one in trnsformed coordinates.
- double getQ2 (double) const

  The values of Q1, Q2 and Q3 are needed to compute the solution in real coordinates from the one in transformed coordinates.
- double getQ3 (double) const

  The values of Q1, Q2 and Q3 are needed to compute the solution in real coordinates from the one in transformed coordinates.
- unsigned int getLowestDof () const

  This function returns the number of the lowest degree of freedom associated with this Sector.
- unsigned int getNDofs () const

  This function returns the number of dofs which are part of this sector.
- unsigned int getNInternalBoundaryDofs () const

  In order to set appropriate boundary conditions it makes sense to determine, which degrees are associated with an edge which is part of an interface to another sector.
- unsigned int getNActiveCells () const

  This function can be used to query the number of cells in a Sector / subdomain.
- void setLowestDof (unsigned int)

  Setter for the value that the getter should return.
- void setNDofs (unsigned int)

  Setter for the value that the getter should return.
- void setNInternalBoundaryDofs (unsigned int)

  Setter for the value that the getter should return.
- void setNActiveCells (unsigned int)

  Setter for the value that the getter should return.

- double get_dof (unsigned int i, double z) const

  This function returns the value of a specified dof at a given internal position.
- double get_r (double z) const

  Get an interpolation of the radius for a coordinate z.
- double get_v (double z) const

  Get an interpolation of the tilt for a coordinate z.
- double get_m (double z) const

  Get an interpolation of the shift for a coordinate z.
- void **set_properties** (double, double, double, double)
- void **set_properties** (double in_m_0, double in_m_1, double in_r_0, double in_r_1, double in_v_0, double in_v_1)
- void **set_properties_force** (double, double, double, double)
- void **set_properties_force** (double in_m_0, double in_m_1, double in_r_0, double in_r_1, double in_v_0, double in_v_1)
- Tensor< 2, 3, double > **TransformationTensorInternal** (double in_x, double in_y, double z) const

Public Attributes

- const bool left

  This value describes, if this Sector is at the left (small z) end of the computational domain.
- const bool right

  This value describes, if this Sector is at the right (large z) end of the computational domain.
- const bool boundary

  This value is true, if either left or right are true.
- const double **z_0**
- const double z_1

  The objects created from this class are supposed to hand back the material properties which include the space-transformation Tensors.
- unsigned int **LowestDof**
- unsigned int **NDofs**
- unsigned int **NInternalBoundaryDofs**
- unsigned int **NActiveCells**
- std::vector< double > **dofs_l**
- std::vector< double > **dofs_r**
- std::vector< unsigned int > **derivative**
- std::vector< bool > **zero_derivative**

70.1 Detailed Description

template<unsigned int Dofs_Per_Sector>
class Sector< Dofs_Per_Sector >

Sectors are used, to split the computational domain into chunks, whose degrees of freedom are likely coupled.

The interfaces between Sectors lie in the xy-plane and they are ordered by their z-value.

Definition at line 25 of file Sector.h.

70.2 Constructor & Destructor Documentation

Sector()

```
template<unsigned int Dofs_Per_Sector>
Sector< Dofs_Per_Sector >::Sector (
            bool in_left,
            bool in_right,
            double in_z_0,
            double in_z_1 )
```

Constructor of the Sector class, that takes all important properties as an input property.

Parameters

in_left	stores if the sector is at the left end. It is used to initialize the according variable.
in_right	stores if the sector is at the right end. It is used to initialize the according variable.
in_z_0	stores the z-coordinate of the left surface-plain. It is used to initialize the according variable.
in_z_1	stores the z-coordinate of the right surface-plain. It is used to initialize the according variable.

Definition at line 12 of file Sector.cpp.

```
    : left(in_left),
      right(in_right),
      boundary(in_left && in_right),
      z_0(in_z_0),
      z_1(in_z_1) {
  dofs_l.resize(Dofs_Per_Sector);
  dofs_r.resize(Dofs_Per_Sector);
  derivative.resize(Dofs_Per_Sector);
  zero_derivative.resize(Dofs_Per_Sector);
  if (Dofs_Per_Sector == 3) {
    zero_derivative[0] = true;
    zero_derivative[1] = false;
    zero_derivative[2] = true;
    derivative      [0] = 0;
    derivative      [1] = 2;
    derivative      [2] = 0;
  }
  if (Dofs_Per_Sector == 2) {
    zero_derivative[0] = false;
    zero_derivative[1] = true;
```

```
    derivative     [0] = 1;
    derivative     [1] = 0;
  }

  for (unsigned int i = 0; i < Dofs_Per_Sector; i++) {
    dofs_l[i] = 0;
    dofs_r[i] = 0;
  }
  NInternalBoundaryDofs = 0;
  LowestDof = 0;
  NActiveCells = 0;
  NDofs = Dofs_Per_Sector;
}
```

70.3 Member Function Documentation

get_dof()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::get_dof (
              unsigned int i,
              double z ) const
```

This function returns the value of a specified dof at a given internal position.

Parameters

i	index of the dof. This class has a template argument specifying the number of dofs per sector. This argument has to be less or equal.
z	this is a relative value for interpolation with $z \in [0, 1]$. If $z = 0$ the values for the lower end of the sector are returned. If $z = 1$ the values for the upper end of the sector are returned. In between the values are interpolated according to the rules for the specific dof.

Definition at line 146 of file Sector.cpp.

```
146                                                                       {
147   if (i > 0 && i < NDofs) {
148     if (z < 0.0) z = 0.0;
149     if (z > 1.0) z = 1.0;
150     if (zero_derivative[i]) {
151       return InterpolationPolynomialZeroDerivative(z, dofs_l[i], dofs_r[i]);
152     } else {
153       return InterpolationPolynomial(z, dofs_l[i], dofs_r[i],
154                                      dofs_l[derivative[i]],
155                                      dofs_r[derivative[i]]);
156     }
157   } else {
158     print_info("Sector<Dofs_Per_Sector>::get_dof", "There seems to be an error in Sector::get_dof. i > 0
     && i < dofs_per_sector false.", LoggingLevel::PRODUCTION_ALL);
159     return 0;
160   }
161 }
```

get_m()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::get_m (
            double z ) const
```

Get an interpolation of the shift for a coordinate z.

Parameters

double	z is the $z \in [0, 1]$ coordinate for the interpolation.

Definition at line 175 of file Sector.cpp.

```
175                                                    {
176   if (z < 0.0) z = 0.0;
177   if (z > 1.0) z = 1.0;
178   if (Dofs_Per_Sector == 2) {
179     return InterpolationPolynomial(z, dofs_l[0], dofs_r[0], dofs_l[1],
180                                    dofs_r[1]);
181   } else {
182     return InterpolationPolynomial(z, dofs_l[1], dofs_r[1], dofs_l[2],
183                                    dofs_r[2]);
184   }
185 }
```

get_r()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::get_r (
            double z ) const
```

Get an interpolation of the radius for a coordinate z.

Parameters

double	z is the $z \in [0, 1]$ coordinate for the interpolation.

Definition at line 164 of file Sector.cpp.

```
164                                                    {
165   if (z < 0.0) z = 0.0;
166   if (z > 1.0) z = 1.0;
167   if (Dofs_Per_Sector < 3) {
168     print_info("Sector<Dofs_Per_Sector>::get_r", "Error in Sector: Access to radius dof without
      existence.", LoggingLevel::PRODUCTION_ALL);
169     return 0;
170   }
171   return InterpolationPolynomialZeroDerivative(z, dofs_l[0], dofs_r[0]);
172 }
```

get_v()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::get_v (
              double z ) const
```

Get an interpolation of the tilt for a coordinate z.

Parameters

double	z is the $z \in [0, 1]$ coordinate for the interpolation.

Definition at line 188 of file Sector.cpp.

```
188                                                    {
189   if (z < 0.0) z = 0.0;
190   if (z > 1.0) z = 1.0;
191   if (Dofs_Per_Sector == 2) {
192     return InterpolationPolynomialZeroDerivative(z, dofs_l[1], dofs_r[1]);
193   } else {
194     return InterpolationPolynomialZeroDerivative(z, dofs_l[2], dofs_r[2]);
195   }
196 }
```

getLowestDof()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getLowestDof
```

This function returns the number of the lowest degree of freedom associated with this Sector.

Keep in mind, that the degrees of freedom associated with edges on the lower (small z) interface are not included since this functionality is supposed to help in the block-structure generation and those dofs are part of the neighboring block.

Definition at line 397 of file Sector.cpp.

```
397                                                         {
398   return LowestDof;
399 }
```

getNActiveCells()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getNActiveCells
```

This function can be used to query the number of cells in a Sector / subdomain.

In this case there are no problems with interface-dofs. Every cell belongs to exactly one sector (the problem arises from the fact, that one edge can (and most of the time will) belong to more then one cell).

Definition at line 412 of file Sector.cpp.

```
412                                                            {
413   return NActiveCells;
414 }
```

getNDofs()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getNDofs
```

This function returns the number of dofs which are part of this sector.

The same remarks as for getLowestDof() apply.

Definition at line 402 of file Sector.cpp.

```
402                                              {
403   return NDofs;
404 }
```

getNInternalBoundaryDofs()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getNInternalBoundaryDofs
```

In order to set appropriate boundary conditions it makes sense to determine, which degrees are associated with an edge which is part of an interface to another sector.

Due to the reordering of dofs this is especially easy since the dofs on the interface are those in the interval

$$[\,\mathrm{LowestDof} + \mathrm{NDofs} - \mathrm{NInternalBoundaryDofs}\,,\ \mathrm{LowestDof} + \mathrm{NDofs}\,]$$

Definition at line 407 of file Sector.cpp.

```
407                                                               {
408   return NInternalBoundaryDofs;
409 }
```

getQ1()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::getQ1 (
            double z ) const
```

The values of Q1, Q2 and Q3 are needed to compute the solution in real coordinates from the one in trnsformed coordinates.

This function returnes Q1 for a given position and the current transformation.

Definition at line 199 of file Sector.cpp.

```
199                                             {
200   return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
201                z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
202 }
```

getQ2()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::getQ2 (
            double z ) const
```

The values of Q1, Q2 and Q3 are needed to compute the solution in real coordinates from the one in transformed coordinates.

This function returnes Q2 for a given position and the current transformation.

Definition at line 205 of file Sector.cpp.

```
205                                             {
206   return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
207               z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
208 }
```

getQ3()

```
template<unsigned int Dofs_Per_Sector>
double Sector< Dofs_Per_Sector >::getQ3 (
            double  ) const
```

The values of Q1, Q2 and Q3 are needed to compute the solution in real coordinates from the one in transformed coordinates.

This function returnes Q3 for a given position and the current transformation.

Definition at line 211 of file Sector.cpp.

```
211                                           {
212   return 0.0;
213 }
```

set_properties()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::set_properties (
            double m_0,
            double m_1,
            double r_0,
            double r_1 )
```

This function is used during the optimization-operation to update the properties of the space-transformation.

However, to ensure, that the boundary-conditions remain intact, this function cannot edit the left degrees of freedom if left is true and it cannot edit the right degrees of freedom if right is true

Definition at line 119 of file Sector.cpp.

```
119                                                                    {
120   print_info("Sector<Dofs_Per_Sector>::set_properties", "The code does not work for this number of dofs
       per Sector.", LoggingLevel::PRODUCTION_ALL);
121   return;
122 }
```

setLowestDof()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setLowestDof (
            unsigned int inLowestDOF )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 417 of file Sector.cpp.

```
417                                                                       {
418   LowestDof = inLowestDOF;
419 }
```

setNActiveCells()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setNActiveCells (
            unsigned int inNumberOfActiveCells )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 433 of file Sector.cpp.

```
434                                           {
435   NActiveCells = inNumberOfActiveCells;
436 }
```

setNDofs()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setNDofs (
            unsigned int inNumberOfDOFs )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 422 of file Sector.cpp.

```
422                                                                      {
423   NDofs = inNumberOfDOFs;
424 }
```

setNInternalBoundaryDofs()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setNInternalBoundaryDofs (
            unsigned int in_ninternalboundarydofs )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 427 of file Sector.cpp.
```
                                      {
  NInternalBoundaryDofs = in_ninternalboundarydofs;
}
```

TransformationTensorInternal()

```
template<unsigned int Dimension>
Tensor< 2, 3, double > Sector< Dimension >::TransformationTensorInternal (
            double in_x,
            double in_y,
            double in_z ) const
```

This method gets called from the WaveguideStructure object used in the simulation.

This is where the Waveguide object gets the material Tensors to build the system-matrix. This method returns a complex-values Matrix containing the system-tensors μ^{-1} and ϵ.

Parameters

in_x	x-coordinate of the point, for which the Tensor should be calculated.
in_y	y-coordinate of the point, for which the Tensor should be calculated.
in_z	z-coordinate of the point, for which the Tensor should be calculated.

Definition at line 389 of file Sector.cpp.
```
                               {
  Tensor<2, 3, double> ret;
  print_info("Sector<Dimension>::TransformationTensorInternal", "The code does not work for you Sector
   specification." + std::to_string(Dimension), LoggingLevel::PRODUCTION_ALL);
  return ret;
}
```

70.4 Member Data Documentation

z_1

```
template<unsigned int Dofs_Per_Sector>
const double Sector< Dofs_Per_Sector >::z_1
```

The objects created from this class are supposed to hand back the material properties which include the space-transformation Tensors.

For this to be possible, the Sector has to be able to transform from global coordinates to coordinates that are scaled inside the Sector. For this purpose, the z_0 and z_1 variables store the z-coordinate of both, the left and right surface.

Definition at line 66 of file Sector.h.

The documentation for this class was generated from the following files:

- Code/Core/Sector.h
- Code/Core/Sector.cpp

71 ShapeDescription Class Reference

Public Member Functions

- void **SetByString** (std::string)
- void **SetStraight** ()

Public Attributes

- int **Sectors**
- std::vector< double > **m**
- std::vector< double > **v**
- std::vector< double > **z**

71.1 Detailed Description

Definition at line 17 of file ShapeDescription.h.

The documentation for this class was generated from the following files:

- Code/Helpers/ShapeDescription.h
- Code/Helpers/ShapeDescription.cpp

72 ShapeFunction Class Reference

These objects are used in the shape optimization code.

`#include <ShapeFunction.h>`

Public Member Functions

- ShapeFunction (double in_z_min, double in_z_max, unsigned int in_n_sectors, bool in_bad_init=false)

  Construct a new Shape Function object These functions a parametrized by the z coordinate.
- double evaluate_at (double z) const

  Evaluates the shape function for a given z-coordinate.
- double evaluate_derivative_at (double z) const

  Evaluates the shape function derivative for a given z-coordinate.
- void set_constraints (double in_f_0, double in_f_1, double in_df_0, double in_df_1)

  Sets the default constraints for these types of function.
- void update_constrained_values ()

We only store the derivative values and the values of the function at the lower and upper limit.

- void set_free_values (std::vector< double > in_dof_values)

  Set the free dof values.
- unsigned int get_n_dofs () const

  Get the number of degrees of freedom of this object.
- unsigned int get_n_free_dofs () const

  Get the number of unconstrained degrees of freedom of this object.
- double get_dof_value (unsigned int index) const

  Get the value of a dof.
- double get_free_dof_value (unsigned int index) const

  Same as get_dof_value but in free dof numbering, so index 0 is the first free dof and the last one is the last free dof.
- void initialize ()

  Sets up the object by computing initial values for the shape dofs based on the boundary constraints.
- void set_free_dof_value (unsigned int index, double value)

  Set the value of the index-th free dof to value.
- void print ()

  Prints some cosmetic output about a shape function.

Static Public Member Functions

- static unsigned int compute_n_dofs (unsigned int in_n_sectors)

  For a provided number of sectors, this provides the number of degrees of freedom the function will have.
- static unsigned int compute_n_free_dofs (unsigned int in_n_sectors)

  Computes how many unconstrained dofs a shape function will have (static).

Public Attributes

- const unsigned int **n_free_dofs**
- const unsigned int **n_dofs**

72.1 Detailed Description

These objects are used in the shape optimization code.

They have a certain number of degrees of freedom and are used for the description of coordinate transformations. These functions are described in the optimization chapter of the dissertation document.

Definition at line 18 of file ShapeFunction.h.

72.2 Constructor & Destructor Documentation

ShapeFunction()

```
ShapeFunction::ShapeFunction (
            double in_z_min,
            double in_z_max,
            unsigned int in_n_sectors,
            bool in_bad_init = false )
```

Construct a new Shape Function object These functions a parametrized by the z coordinate.

Therefore, the constructor requires the z-range. Additionally we need the number of sectors. Per sector, there is an additional degree of freedom. The bad init flag triggers a bad initialization of the values such that an optimization algorithm has some space for optimization.

Parameters

in_z_min	Lower end-point of the range.
in_z_max	Upper end-point of the range.
in_n_sectors	Number of sectors.
in_bad_init	Bad-init flag triggers 0-initialization to give optimization some play.

Definition at line 22 of file ShapeFunction.cpp.

```
22
      :
23 sector_length((in_z_max - in_z_min) / (2*(double)in_n_sectors)),
24 n_free_dofs(ShapeFunction::compute_n_free_dofs(in_n_sectors)),
25 n_dofs(ShapeFunction::compute_n_dofs(in_n_sectors))
26 {
27     dof_values.resize(n_dofs);
28     for(unsigned int i = 0; i < n_dofs; i++) {
29         dof_values[i] = 0;
30     }
31     z_min = in_z_min;
32     z_max = in_z_max/2.0;
33     bad_init = in_bad_init;
34 }
```

72.3 Member Function Documentation

compute_n_dofs()

```
unsigned int ShapeFunction::compute_n_dofs (
            unsigned int in_n_sectors )  [static]
```

For a provided number of sectors, this provides the number of degrees of freedom the function will have.

See the chapter in the dissertation for details.

Parameters

in_n_sectors	Number of sectors of the function.

Returns

unsigned int Number of degrees of freedom of the shape function.

Definition at line 17 of file ShapeFunction.cpp.

```
17                                                                    {
18     return in_n_sectors+3;
19 }
```

Referenced by compute_n_free_dofs().

compute_n_free_dofs()

```
unsigned int ShapeFunction::compute_n_free_dofs (
            unsigned int in_n_sectors )  [static]
```

Computes how many unconstrained dofs a shape function will have (static).

See the chapter in the dissertation for details.

Parameters

in_n_sectors	Number of sectors of the function.

Returns

unsigned int Number of degrees of unconstrained degrees of freedom of the shape function.

Definition at line 8 of file ShapeFunction.cpp.

```
8                                                                        {
9      int ret = ShapeFunction::compute_n_dofs(in_n_sectors);
10      ret -= 5;
11      if(ret < 0) {
12          std::cout « "The shape function is underdetermined. Add more sectors." « std::endl;
13      }
14      return std::abs(ret);
15 }
```

References compute_n_dofs().

evaluate_at()

```
double ShapeFunction::evaluate_at (
            double z ) const
```

Evaluates the shape function for a given z-coordinate.

Parameters

z	z-coordinate to evaluate the function at.

Returns

double function value at that z-coordinate.

Definition at line 36 of file ShapeFunction.cpp.

```
36                                                  {
37     if(z <= z_min) {
38         return dof_values[0];
39     }
40     if(z > z_max) {
41         return evaluate_at(2*z_max - z);
42     }
43     double ret = dof_values[0];
44     double z_temp = z_min;
45     unsigned int index = 1;
46     while(z_temp + sector_length < z+FLOATING_PRECISION) {
47         ret += 0.5 * sector_length * (dof_values[index + 1] - dof_values[index]);
48         ret += sector_length * dof_values[index];
49         index++;
50         z_temp += sector_length;
51     }
52     double delta_z = z - z_temp;
53     if(std::abs(delta_z) <= FLOATING_PRECISION) {
54         return ret;
55     }
56     ret += 0.5 * delta_z * (dof_values[index + 1] - dof_values[index]) * (delta_z/sector_length);
57     ret += delta_z * dof_values[index];
58     return ret;
59 }
```

Referenced by print(), and update_constrained_values().

evaluate_derivative_at()

```
double ShapeFunction::evaluate_derivative_at (
              double z ) const
```

Evaluates the shape function derivative for a given z-coordinate.

Parameters

z	z-coordinate to evaluate the derivative of the function at.

Returns

double derivative of the function at provided z-coordinate.

Definition at line 61 of file ShapeFunction.cpp.

```
61                                                              {
62     if(z <= z_min) {
63         return dof_values[1];
64     }
65     if(z > z_max) {
66         return - evaluate_derivative_at(2*z_max - z);
67     }
68     unsigned int index = 1;
69     double z_temp = z_min;
70     while(z_temp + sector_length < z) {
71         index ++;
72         z_temp += sector_length;
73     }
74     double delta_z = z - z_temp;
75     if(std::abs(delta_z) < FLOATING_PRECISION) {
76         return dof_values[index];
```

```
    } else {
        return dof_values[index] + (dof_values[index + 1] - dof_values[index]) * ( delta_z /
    sector_length );
    }
}
```

get_dof_value()

```
double ShapeFunction::get_dof_value (
            unsigned int index ) const
```

Get the value of a dof.

Parameters

index	The index of the dof.

Returns

double The value of the dof.

Definition at line 138 of file ShapeFunction.cpp.

```
138                                                            {
139     return dof_values[index];
140 }
```

Referenced by WaveguideTransformation::get_dof_values().

get_free_dof_value()

```
double ShapeFunction::get_free_dof_value (
            unsigned int index ) const
```

Same as get_dof_value but in free dof numbering, so index 0 is the first free dof and the last one is the last free dof.

Parameters

index	Index of the free dof to query for.

Returns

double Value of that dof.

Definition at line 141 of file ShapeFunction.cpp.

```
141                                                            {
142     return dof_values[index + 2];
143 }
```

get_n_dofs()

`unsigned int ShapeFunction::get_n_dofs ( ) const`

Get the number of degrees of freedom of this object.

Returns

unsigned int Number of dofs.

Definition at line 132 of file ShapeFunction.cpp.

```
132                                           {
133     return n_dofs;
134 }
```

Referenced by WaveguideTransformation::get_dof_values().

get_n_free_dofs()

`unsigned int ShapeFunction::get_n_free_dofs ( ) const`

Get the number of unconstrained degrees of freedom of this object.

This is the number of dofs that can be varied during the optimization.

Returns

unsigned int Number of free dofs.

Definition at line 135 of file ShapeFunction.cpp.

```
135                                                {
136     return n_free_dofs;
137 }
```

set_constraints()

```
void ShapeFunction::set_constraints (
            double in_f_0,
            double in_f_1,
            double in_df_0,
            double in_df_1 )
```

Sets the default constraints for these types of function.

The constraints are usually function value and derivative at the upper and lower boundary, i.e. for $z = z_{min}$ and $z = z_{max}$.

Parameters

in_f_0	$f(z_{min})$
in_f_1	$f(z_{max})$
in_df_0	$\frac{\partial f}{\partial z}(z_{min})$
in_df_1	$\frac{\partial f}{\partial z}(z_{max})$

Definition at line 82 of file ShapeFunction.cpp.

```
82                                                                                          {
83      f_0 = in_f_0;
84      df_0 = in_df_0;
85      f_1 = in_f_1;
86      df_1 = in_df_1;
87      update_constrained_values();
88 }
```

References update_constrained_values().

Referenced by WaveguideTransformation::estimate_and_initialize().

set_free_dof_value()

```
void ShapeFunction::set_free_dof_value (
              unsigned int index,
              double value )
```

Set the value of the index-th free dof to value.

Parameters

index	Index of the dof.
value	Value of the dof.

Definition at line 145 of file ShapeFunction.cpp.

```
145                                                                        {
146     if(index < n_free_dofs) {
147         dof_values[index + 2] = value;
148         update_constrained_values();
149     } else {
150         std::cout « "You tried to write to a constrained dof of a shape function." « std::endl;
151     }
152 }
```

References update_constrained_values().

set_free_values()

```
void ShapeFunction::set_free_values (
            std::vector< double > in_dof_values )
```

Set the free dof values.

This function gets called by the optimization method.

Parameters

in_dof_values	The values to set.

Definition at line 99 of file ShapeFunction.cpp.

```
99                                                          {
100     if(in_dof_values.size() != n_free_dofs) {
101         std::cout << "Provided wrong number of degrees of freedom." << std::endl;
102     }
103     for(unsigned int i = 0; i < in_dof_values.size(); i++) {
104         dof_values[2 + i] = in_dof_values[i];
105     }
106     update_constrained_values();
107 }
```

References update_constrained_values().

update_constrained_values()

```
void ShapeFunction::update_constrained_values ( )
```

We only store the derivative values and the values of the function at the lower and upper limit.

During the computation we only consider the derivatives for the shape gradient. Of these values the highest and lowest index are constrained directly (typically to zero) and an additional constraint is computed based on the difference between the function value at input and output.

Definition at line 90 of file ShapeFunction.cpp.

```
90                                          {
91      dof_values[0] = f_0;
92      dof_values[1] = df_0;
93      dof_values[n_dofs-2] = df_1;
94      dof_values[n_dofs-1] = f_1;
95      double f_y_min2 = evaluate_at(z_max - sector_length - sector_length);
96      dof_values[n_dofs-3] = (dof_values[n_dofs - 1] - f_y_min2) / sector_length - (0.5 *
        (dof_values[n_dofs - 4] + dof_values[n_dofs - 2]));
97 }
```

References evaluate_at().

Referenced by set_constraints(), set_free_dof_value(), and set_free_values().

The documentation for this class was generated from the following files:

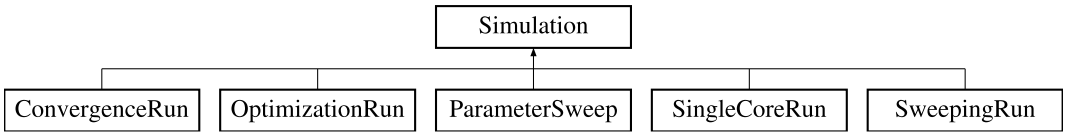
- Code/Optimization/ShapeFunction.h
- Code/Optimization/ShapeFunction.cpp

73 Simulation Class Reference

This base class is very important and abstract.

```
#include <Simulation.h>
```

Inheritance diagram for Simulation:



Public Member Functions

- virtual void prepare ()=0

  In derived classes, this function sets up all that is required to perform the core functionality, i.e.
- virtual void run ()=0

  Run the core computation.
- virtual void prepare_transformed_geometry ()=0

  If a representation of the solution in the physical coordinates is required, this function provides it.
- void create_output_directory ()

  Create a output directory to store the computational results in.

73.1 Detailed Description

This base class is very important and abstract.

While the HierarchicalProblem types perform the computation of an E-field solution to a problem, these classes are the reason why we do so. The derived classes handle default experiments for the sweeping preconditioners, convergence studies or shape optimization.

Definition at line 23 of file Simulation.h.

73.2 Member Function Documentation

prepare()

```
virtual void Simulation::prepare ( )  [pure virtual]
```

In derived classes, this function sets up all that is required to perform the core functionality, i.e.

construct problems types.

Implemented in OptimizationRun, ConvergenceRun, SweepingRun, SingleCoreRun, and ParameterSweep.

The documentation for this class was generated from the following files:

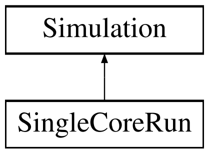
- Code/Runners/Simulation.h
- Code/Runners/Simulation.cpp

74 SingleCoreRun Class Reference

In cases in which a single core is enough to solve the problem, this runner can be used.

`#include <SingleCoreRun.h>`

Inheritance diagram for SingleCoreRun:



Public Member Functions

- void prepare () override

  Prepares the mainProblem, which in this case is cheap because it is completely local.
- void run () override

  Computes the solution.
- void prepare_transformed_geometry () override

  Not required / not implemented.

74.1 Detailed Description

In cases in which a single core is enough to solve the problem, this runner can be used.

It is the only one that constructs the mainProblem member to be a Local instead of a NonLocal problem.

Definition at line 21 of file SingleCoreRun.h.

The documentation for this class was generated from the following files:

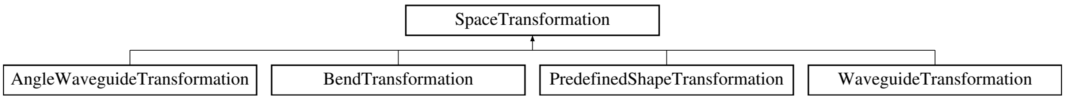
- Code/Runners/SingleCoreRun.h
- Code/Runners/SingleCoreRun.cpp

75 SpaceTransformation Class Reference

The SpaceTransformation class encapsulates the coordinate transformation used in the simulation.

`#include <SpaceTransformation.h>`

Inheritance diagram for SpaceTransformation:



Public Member Functions

- virtual Position math_to_phys (Position coord) const =0

  Transforms a coordinate in the mathematical coord system to physical ones.
- virtual Position phys_to_math (Position coord) const =0

  Transforms a coordinate in the physical coord system to mathematical ones.
- virtual double get_det (Position)

  Get the determinant of the transformation matrix at a provided location.
- virtual Tensor< 2, 3, double > get_J (Position &)

  Compute the Jacobian of the current transformation at a given location.
- virtual Tensor< 2, 3, double > get_J_inverse (Position &)

  Compute the Jacobian of the current transformation at a given location and invert it.
- virtual Tensor< 2, 3, ComplexNumber > get_Tensor (Position &)=0

  Get the transformation tensor at a given location.
- virtual Tensor< 2, 3, double > get_Space_Transformation_Tensor (Position &)=0

  Get the real part of the transformation tensor at a given location.
- Tensor< 2, 3, ComplexNumber > get_Tensor_for_step (Position &coordinate, unsigned int dof, double step_width)

  For adjoint based optimization we require a tensor describing the change of the material tensor at a given location if the requested dof is changed by step_width.
- Tensor< 2, 3, ComplexNumber > get_inverse_Tensor_for_step (Position &coordinate, unsigned int dof, double step_width)

  Same as the function above but returns the inverse.
- void switch_application_mode (bool apply_math_to_physical)

  This function can be used in the dealii::transform function by applying the operator() function.
- virtual void estimate_and_initialize ()=0

  At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.
- virtual double get_dof (int) const

  This is a getter for the values of degrees of freedom.
- virtual double get_free_dof (int) const

  This is a getter for the values of degrees of freedom.
- virtual void set_free_dof (int, double)

  This function sets the value of the dof provided to the given value.
- virtual std::pair< int, double > Z_to_Sector_and_local_z (double in_z) const

  Using this method unifies the usage of coordinates.

- virtual Vector< double > get_dof_values () const

  Other objects can use this function to retrieve an array of the current values of the degrees of freedom of the functional we are optimizing.
- virtual unsigned int n_free_dofs () const

  This function returns the number of unrestrained degrees of freedom of the current optimization run.
- virtual unsigned int n_dofs () const

  This function returns the total number of DOFs including restrained ones.
- virtual void Print () const =0

  Console output of the current Waveguide Structure.
- Position operator() (Position) const

  Applies either math_to_phys or phys_to_math depending on the current transformation mode.

Public Attributes

- bool **apply_math_to_phys** = true

75.1 Detailed Description

The SpaceTransformation class encapsulates the coordinate transformation used in the simulation.

Two important decisions have to be made in the computation: Which shape should be used for the waveguide? This can either be rectangular or tubular. Should the coordinate-transformation always be equal to identity in any domain where PML is applied? (yes or no). However, the space transformation is the only information required to compute the Tensor g which is a 3×3 matrix whilch (multiplied by the material value of the untransfomred coordinate either inside or outside the waveguide) gives us the value of ϵ and μ. From this class we derive several different classes which then specify the interface specified in this class.

Definition at line 35 of file SpaceTransformation.h.

75.2 Member Function Documentation

estimate_and_initialize()

```
virtual void SpaceTransformation::estimate_and_initialize ( )  [pure virtual]
```

At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.

Therefore the values for the degrees of freedom need to be estimated. This function sets all variables to appropiate values and estimates an appropriate shape based on averages and a polynomial interpolation of the boundary conditions on the shape.

Implemented in WaveguideTransformation, BendTransformation, AngleWaveguideTransformation, and PredefinedShapeTransformation.

get_det()

```
virtual double SpaceTransformation::get_det (
            Position  )  [inline], [virtual]
```

Get the determinant of the transformation matrix at a provided location.

Returns

double determinant of J.

Reimplemented in AngleWaveguideTransformation.

Definition at line 63 of file SpaceTransformation.h.

```
63                                  {
64     return 1.0;
65   }
```

get_dof()

```
virtual double SpaceTransformation::get_dof (
            int   ) const  [inline], [virtual]
```

This is a getter for the values of degrees of freedom.

A getter-setter interface was introduced since the values are estimated automatically during the optimization and non-physical systems should be excluded from the domain of possible cases.

Parameters

dof	The index of the degree of freedom to be retrieved from the structure of the modelled waveguide.

Returns

This function returns the value of the requested degree of freedom. Should this dof not exist, 0 will be returned.

Reimplemented in WaveguideTransformation.

Definition at line 156 of file SpaceTransformation.h.

```
156                                  {
157     return 0;
158   };
```

Referenced by get_inverse_Tensor_for_step(), and get_Tensor_for_step().

get_dof_values()

```
virtual Vector<double> SpaceTransformation::get_dof_values ( ) const  [inline], [virtual]
```

Other objects can use this function to retrieve an array of the current values of the degrees of freedom of the functional we are optimizing.

This also includes restrained degrees of freedom and other functions can be used to determine this property. This has to be done because in different cases the number of restrained degrees of freedom can vary and we want no logic about this in other functions.

Reimplemented in WaveguideTransformation, and AngleWaveguideTransformation.

Definition at line 197 of file SpaceTransformation.h.

```
197                                         {
198     Vector<double> ret;
199     return ret;
200   };
```

get_free_dof()

```
virtual double SpaceTransformation::get_free_dof (
            int   ) const  [inline], [virtual]
```

This is a getter for the values of degrees of freedom.

A getter-setter interface was introduced since the values are estimated automatically during the optimization and non-physical systems should be excluded from the domain of possible cases.

Parameters

dof	The index of the degree of freedom to be retrieved from the structure of the modelled waveguide.

Returns

This function returns the value of the requested degree of freedom. Should this dof not exist, 0 will be returnd.

Reimplemented in WaveguideTransformation.

Definition at line 169 of file SpaceTransformation.h.

```
169 { return 0.0; };
```

get_inverse_Tensor_for_step()

```
Tensor< 2, 3, ComplexNumber > SpaceTransformation::get_inverse_Tensor_for_step (
            Position & coordinate,
            unsigned int dof,
            double step_width )
```

Same as the function above but returns the inverse.

Parameters

coordinate	Location to compute the tensor.

Parameters

dof	The index of the dof to be updated.
step_width	The step_width for the step.

Returns

Tensor<2, 3, ComplexNumber>

Definition at line 45 of file SpaceTransformation.cpp.

```
45
                                       {
46    double old_value = get_dof(dof);
47    Tensor<2, 3, double> trafo1 = invert(get_Space_Transformation_Tensor(coordinate));
48
49    set_free_dof(dof, old_value + step_width);
50    Tensor<2, 3, double> trafo2 = invert(get_Space_Transformation_Tensor(coordinate));
51
52    set_free_dof(dof, old_value);
53    return trafo2 - trafo1;
54 }
```

References get_dof(), get_Space_Transformation_Tensor(), and set_free_dof().

get_J()

```
virtual Tensor<2,3,double> SpaceTransformation::get_J (
                Position & )  [inline], [virtual]
```

Compute the Jacobian of the current transformation at a given location.

Returns

Tensor<2,3,double> Jacobian matrix at the given location.

Reimplemented in AngleWaveguideTransformation, WaveguideTransformation, and PredefinedShapeTransformation.

Definition at line 72 of file SpaceTransformation.h.

```
72                                                  {
73      Tensor<2,3,double> ret;
74      ret[0][0] = 1;
75      ret[1][1] = 1;
76      ret[2][2] = 1;
77      return ret;
78    }
```

get_J_inverse()

```
virtual Tensor<2,3,double> SpaceTransformation::get_J_inverse (
                Position & )  [inline], [virtual]
```

Compute the Jacobian of the current transformation at a given location and invert it.

Returns

Tensor<2,3,double> Inverse of the jacobian matrix at the given location.

Reimplemented in AngleWaveguideTransformation, WaveguideTransformation, and PredefinedShapeTransformation.

Definition at line 85 of file SpaceTransformation.h.

```
85                                                           {
86      Tensor<2,3,double> ret;
87      ret[0][0] = 1;
88      ret[1][1] = 1;
89      ret[2][2] = 1;
90      return ret;
91    }
```

get_Space_Transformation_Tensor()

```
virtual Tensor<2, 3, double> SpaceTransformation::get_Space_Transformation_Tensor (
            Position & ) [pure virtual]
```

Get the real part of the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 real valued tensor for a given locations.

Implemented in WaveguideTransformation, AngleWaveguideTransformation, BendTransformation, and PredefinedShapeTransformation.

Referenced by get_inverse_Tensor_for_step(), and get_Tensor_for_step().

get_Tensor()

```
virtual Tensor<2, 3, ComplexNumber> SpaceTransformation::get_Tensor (
            Position & ) [pure virtual]
```

Get the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 complex valued tensor for a given locations.

Implemented in WaveguideTransformation, AngleWaveguideTransformation, BendTransformation, and PredefinedShapeTransformation.

get_Tensor_for_step()

```
Tensor< 2, 3, ComplexNumber > SpaceTransformation::get_Tensor_for_step (
            Position & coordinate,
            unsigned int dof,
            double step_width )
```

For adjoint based optimization we require a tensor describing the change of the material tensor at a given location if the requested dof is changed by step_width.

The function basically computes the transformation tensor for the current parameter values and then updates the parametrization in the dof-th component by step_width and computes the material tensor. It then computes the difference of the two and returns it.

Parameters

coordinate	Location to compute the difference tensor.
dof	The index of the dof to be updated.
step_width	The step_width for the step.

Returns

Tensor<2, 3, ComplexNumber>

Definition at line 34 of file SpaceTransformation.cpp.

```
34                               {
35    double old_value = get_dof(dof);
36    Tensor<2, 3, double> trafo1 = get_Space_Transformation_Tensor(coordinate);
37
38    set_free_dof(dof, old_value + step_width);
39    Tensor<2, 3, double> trafo2 = get_Space_Transformation_Tensor(coordinate);
40
41    set_free_dof(dof, old_value);
42    return trafo2 - trafo1;
43 }
```

References get_dof(), get_Space_Transformation_Tensor(), and set_free_dof().

math_to_phys()

```
virtual Position SpaceTransformation::math_to_phys (
            Position coord ) const  [pure virtual]
```

Transforms a coordinate in the mathematical coord system to physical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the mathematical system

Returns

Position Coordinate in the physical system

Implemented in WaveguideTransformation, BendTransformation, AngleWaveguideTransformation, and PredefinedShapeTransformation.

Referenced by operator()().

n_dofs()

```
virtual unsigned int SpaceTransformation::n_dofs ( ) const  [inline], [virtual]
```

This function returns the total number of DOFs including restrained ones.

This is the lenght of the array returned by Dofs().

Reimplemented in WaveguideTransformation, and AngleWaveguideTransformation.

Definition at line 214 of file SpaceTransformation.h.

```
214                                             {
215     return 0;
216   }
```

operator()()

```
Position SpaceTransformation::operator() (
             Position in_p ) const
```

Applies either math_to_phys or phys_to_math depending on the current transformation mode.

This can be used in the dealii::transform() function.

Returns

Position Location to be transformed.

Definition at line 56 of file SpaceTransformation.cpp.

```
56                                                            {
57   return math_to_phys(in_p);
58 }
```

References math_to_phys().

phys_to_math()

```
virtual Position SpaceTransformation::phys_to_math (
             Position coord ) const  [pure virtual]
```

Transforms a coordinate in the physical coord system to mathematical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the physical system

Returns

Position Coordinate in the mathematical system

Implemented in WaveguideTransformation, BendTransformation, AngleWaveguideTransformation, and PredefinedShapeTransformation.

set_free_dof()

```
virtual void SpaceTransformation::set_free_dof (
            int ,
            double  ) [inline], [virtual]
```

This function sets the value of the dof provided to the given value.

It is important to consider, that some dofs are non-writable (i.e. the values of the degrees of freedom on the boundary, like the radius of the input-connector cannot be changed).

Parameters

dof	The index of the parameter to be changed.
value	The value, the dof should be set to.

Reimplemented in WaveguideTransformation.

Definition at line 178 of file SpaceTransformation.h.

```
178 {return;};
```

Referenced by get_inverse_Tensor_for_step(), and get_Tensor_for_step().

switch_application_mode()

```
void SpaceTransformation::switch_application_mode (
            bool apply_math_to_physical )
```

This function can be used in the dealii::transform function by applying the operator() function.

To make it possible to apply both the math_to_phys as well as the phys_to_math transformation we have this function which switches the operation mode.

Parameters

apply_math_to_physical	If this is true, the transformation will now transform from math to phys. Phys to math otherwise.

Definition at line 60 of file SpaceTransformation.cpp.

```
60                                                              {
61   apply_math_to_phys = appl_math_to_phys;
62 }
```

Z_to_Sector_and_local_z()

```
std::pair< int, double > SpaceTransformation::Z_to_Sector_and_local_z (
            double in_z ) const  [virtual]
```

Using this method unifies the usage of coordinates.

This function takes a global z coordinate (in the computational domain) and returns both a Sector-Index and an internal z coordinate indicating which sector this coordinate belongs to and how far along in the sector it is located.

Parameters

double	in_z global system z coordinate for the transformation.

Definition at line 9 of file SpaceTransformation.cpp.

```
9                                                                                 {
10   std::pair<int, double> ret;
11   ret.first = 0;
12   ret.second = 0.0;
13   if (in_z <= Geometry.global_z_range.first) {
14     ret.first = 0;
15     ret.second = 0.0;
16   } else if (in_z < Geometry.global_z_range.second && in_z > Geometry.global_z_range.first) {
17     ret.first = floor( (in_z + Geometry.global_z_range.first) / (GlobalParams.Sector_thickness));
18     ret.second = (in_z + Geometry.global_z_range.first -  (ret.first * GlobalParams.Sector_thickness)) /
       (GlobalParams.Sector_thickness);
19   } else if (in_z >= Geometry.global_z_range.second) {
20     ret.first = GlobalParams.Number_of_sectors - 1;
21     ret.second = 1.0;
22   }
23
24   if (ret.second < 0 || ret.second > 1){
25     std::cout « "Global ranges: " « Geometry.global_z_range.first « " to " «
       Geometry.global_z_range.second « std::endl;
26     std::cout « "Details " « GlobalParams.Sector_thickness « ", " « floor( (in_z +
       Geometry.global_z_range.first) / (GlobalParams.Sector_thickness)) « " and " « (in_z +
       Geometry.global_z_range.first) / (GlobalParams.Sector_thickness) « std::endl;
27     std::cout « "In an erroneous call: ret.first: " « ret.first « " ret.second: " « ret.second « " and
       in_z: " « in_z « " located in sector " « ret.first « " and " « GlobalParams.Sector_thickness «
       std::endl;
28   }
29   return ret;
30 }
```

Referenced by PredefinedShapeTransformation::get_m(), PredefinedShapeTransformation::get_v(), PredefinedShapeTransformation::math_to_phys(), and PredefinedShapeTransformation::phys_to_math().

The documentation for this class was generated from the following files:

- Code/SpaceTransformations/SpaceTransformation.h
- Code/SpaceTransformations/SpaceTransformation.cpp

76 SquareMeshGenerator Class Reference

This class generates meshes, that are used to discretize a rectangular Waveguide.

`#include <SquareMeshGenerator.h>`

Public Member Functions

- bool math_coordinate_in_waveguide (Position position) const

  This function checks if the given coordinate is inside the waveguide or not.
- bool phys_coordinate_in_waveguide (Position position) const

  This function checks if the given coordinate is inside the waveguide or not.
- void prepare_triangulation (dealii::Triangulation< 3 > ∗in_tria)

  This function takes a triangulation object and prepares it for the further computations.
- unsigned int **getDominantComponentAndDirection** (Position in_dir) const
- void **set_boundary_ids** (dealii::Triangulation< 3 > &) const
- void **refine_triangulation_iteratively** (dealii::Triangulation< 3, 3 > ∗)
- bool **check_and_mark_one_cell_for_refinement** (dealii::Triangulation< 3 >::active_cell_iterator)

Public Attributes

- dealii::Triangulation< 3 >::active_cell_iterator **cell**
- dealii::Triangulation< 3 >::active_cell_iterator **endc**

76.1 Detailed Description

This class generates meshes, that are used to discretize a rectangular Waveguide.

Important: This is legacy code. This is currently not required.

The original intention of this project was to model tubular (or cylindrical) waveguides. The motivation behind this thought was the fact, that for this case the modes are known analytically. In applications however modes can be computed numerically and other shapes are easier to fabricate. For example square or rectangular waveguides can be printed in 3D on the scales we currently compute while tubular waveguides on that scale are not yet feasible.

Definition at line 33 of file SquareMeshGenerator.h.

76.2 Member Function Documentation

math_coordinate_in_waveguide()

```
bool SquareMeshGenerator::math_coordinate_in_waveguide (
            Position position ) const
```

This function checks if the given coordinate is inside the waveguide or not.

The naming convention of physical and mathematical system find application. In this version, the waveguide has been transformed and the check for a tubal waveguide for example only checks if the radius of a given vector is below the average of input and output radius. \params position This value gives us the location to check for.

phys_coordinate_in_waveguide()

```
bool SquareMeshGenerator::phys_coordinate_in_waveguide (
            Position position ) const
```

This function checks if the given coordinate is inside the waveguide or not.

The naming convention of physical and mathematical system find application. In this version, the waveguide is bent. If we are using a space transformation f then this function is equal to math_coordinate_in_waveguide(f(x,y,z)). \params position This value gives us the location to check for.

prepare_triangulation()

```
void SquareMeshGenerator::prepare_triangulation (
            dealii::Triangulation< 3 > * in_tria )
```

This function takes a triangulation object and prepares it for the further computations.

It is intended to encapsulate all related work and is explicitely not const.

Parameters

in_tria	The triangulation that is supposed to be prepared. All further information is derived from the parameter file and not given by parameters.

Definition at line 85 of file SquareMeshGenerator.cpp.

```
85                                                                  {
86   GridGenerator::hyper_cube(*in_tria, -1.0, 1.0, false);
87   GridTools::transform(&Triangulation_Shit_To_Local_Geometry, *in_tria);
88   set_boundary_ids(*in_tria);
89
90   in_tria->signals.post_refinement.connect(
91       std::bind(&SquareMeshGenerator::set_boundary_ids,
92           std::cref(*this), std::ref(*in_tria)));
93
94   refine_triangulation_iteratively(in_tria);
95
96   set_boundary_ids(*in_tria);
97 }
```

The documentation for this class was generated from the following files:

- Code/MeshGenerators/SquareMeshGenerator.h
- Code/MeshGenerators/SquareMeshGenerator.cpp

77 SurfaceCellData Struct Reference

Public Attributes

- std::vector< DofNumber > **dof_numbers**
- Position **surface_face_center**

77.1 Detailed Description

Definition at line 217 of file Types.h.

The documentation for this struct was generated from the following file:

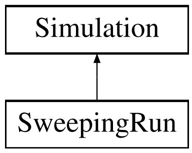
- Code/Core/Types.h

78 SweepingRun Class Reference

This runner constructs a single non-local problem and solves it.

```
#include <SweepingRun.h>
```

Inheritance diagram for SweepingRun:



Public Member Functions

- void prepare () override

  Prepare the solver hierarchy for the parameters provided in the input fields.
- void run () override

  Solve the non-local problem.
- void prepare_transformed_geometry () override

  Not required / Not implemented.

78.1 Detailed Description

This runner constructs a single non-local problem and solves it.

This is mainly used for work on the sweeping preconditioner since it enables a single run and result output.

Definition at line 22 of file SweepingRun.h.

The documentation for this class was generated from the following files:

- Code/Runners/SweepingRun.h
- Code/Runners/SweepingRun.cpp

79 TimerManager Class Reference

A class that stores timers for later output.

```
#include <TimerManager.h>
```

Public Member Functions

- void initialize ()

  Preoares the internal datastructures.
- void switch_context (std::string context, unsigned int level)

  After this point, the timers will count towards the new section.
- void write_output ()

  Writes an output file containing all the timer information about all levels and sections.
- void leave_context (unsigned int level)

  End contribution to the current context on the provided level.

Public Attributes

- std::vector< dealii::TimerOutput > **timer_outputs**
- std::vector< std::string > **filenames**
- std::vector< std::ofstream $*$ > **filestreams**
- unsigned int **level_count**

79.1 Detailed Description

A class that stores timers for later output.

It uses sections to compute all times of similar type, like all solve calls on a certain level or all assembly work. The object computes timing individually for every level.

Definition at line 22 of file TimerManager.h.

79.2 Member Function Documentation

leave_context()

```
void TimerManager::leave_context (
            unsigned int level )
```

End contribution to the current context on the provided level.

Parameters

level	The HSIE sweeping level whose timing measurements we want to switch to another context. If we get done with assembly work on level two and want to switch to solving, we would call leave_context(2) followed by enter_context("solve", 2).

Definition at line 33 of file TimerManager.cpp.

```
33                                     {
34     timer_outputs[level].leave_subsection();
35 }
```

switch_context()

```
void TimerManager::switch_context (
            std::string context,
            unsigned int level )
```

After this point, the timers will count towards the new section.

Parameters

context	Name of the section to switch to.
level	The level we are currently on.

Definition at line 29 of file TimerManager.cpp.

```
29                                                   {
30     timer_outputs[level].enter_subsection(context);
31 }
```

The documentation for this class was generated from the following files:

- Code/GlobalObjects/TimerManager.h
- Code/GlobalObjects/TimerManager.cpp

80 VertexAngelingData Struct Reference

Public Attributes

- unsigned int **vertex_index**
- bool **angled_in_x** = false
- bool **angled_in_y** = false

80.1 Detailed Description

Definition at line 80 of file Types.h.

The documentation for this struct was generated from the following file:

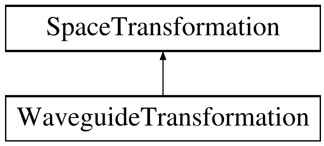
- Code/Core/Types.h

81 WaveguideTransformation Class Reference

In this case we regard a rectangular waveguide and the effects on the material tensor by the space transformation and the boundary condition PML may overlap.

```
#include <WaveguideTransformation.h>
```

Inheritance diagram for WaveguideTransformation:



Public Member Functions

- Position math_to_phys (Position coord) const override

  Transforms a coordinate in the mathematical coord system to physical ones.
- Position phys_to_math (Position coord) const override

  Transforms a coordinate in the physical coord system to mathematical ones.
- dealii::Tensor< 2, 3, ComplexNumber > get_Tensor (Position &coordinate) override

  Get the transformation tensor at a given location.
- dealii::Tensor< 2, 3, double > get_Space_Transformation_Tensor (Position &coordinate) override

  Get the real part of the transformation tensor at a given location.
- Tensor< 2, 3, double > get_J (Position &) override

  Compute the Jacobian of the current transformation at a given location.
- Tensor< 2, 3, double > get_J_inverse (Position &) override

  Compute the Jacobian of the current transformation at a given location and invert it.
- void estimate_and_initialize () override

  At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.
- double get_dof (int dof) const override

  This is a getter for the values of degrees of freedom.
- double get_free_dof (int dof) const override

  This is a getter for the values of degrees of freedom.
- void set_free_dof (int dof, double value) override

  This function sets the value of the dof provided to the given value.
- Vector< double > get_dof_values () const override

  Other objects can use this function to retrieve an array of the current values of the degrees of freedom of the functional we are optimizing.
- unsigned int n_free_dofs () const override

  This function returns the number of unrestrained degrees of freedom of the current optimization run.
- unsigned int n_dofs () const override

  This function returns the total number of DOFs including restrained ones.
- void Print () const override

  Console output of the current Waveguide Structure.

- std::pair< ResponsibleComponent, unsigned int > **map_free_dof_index** (unsigned int) const
- std::pair< ResponsibleComponent, unsigned int > **map_dof_index** (unsigned int) const

Additional Inherited Members

81.1 Detailed Description

In this case we regard a rectangular waveguide and the effects on the material tensor by the space transformation and the boundary condition PML may overlap.

The waveguide transformation is a variable y-shift of the coordinate system and uses a shape-function to describe the shape.

For the non-documented members see the documentation in the base class SpaceTransformation.

Definition at line 38 of file WaveguideTransformation.h.

81.2 Member Function Documentation

estimate_and_initialize()

```
void WaveguideTransformation::estimate_and_initialize ( )  [override], [virtual]
```

At the beginning (before the first solution of a system) only the boundary conditions for the shape of the waveguide are known.

Therefore the values for the degrees of freedom need to be estimated. This function sets all variables to appropiate values and estimates an appropriate shape based on averages and a polynomial interpolation of the boundary conditions on the shape.

Implements SpaceTransformation.

Definition at line 129 of file WaveguideTransformation.cpp.

```
129                                                     {
130   vertical_shift.set_constraints(0, GlobalParams.Vertical_displacement_of_waveguide, 0,0);
131   vertical_shift.initialize();
132   if(!GlobalParams.keep_waveguide_height_constant) {
133     waveguide_height.set_constraints(1, 1, 0,0);
134     waveguide_height.initialize();
135   }
136   if(!GlobalParams.keep_waveguide_width_constant) {
137     waveguide_width.set_constraints(1, 1, 0,0);
138     waveguide_height.initialize();
139   }
140 }
```

References ShapeFunction::set_constraints().

get_dof()

```
double WaveguideTransformation::get_dof (
              int dof ) const  [override], [virtual]
```

This is a getter for the values of degrees of freedom.

A getter-setter interface was introduced since the values are estimated automatically during the optimization and non-physical systems should be excluded from the domain of possible cases.

Parameters

dof	The index of the degree of freedom to be retrieved from the structure of the modelled waveguide.

Returns

This function returns the value of the requested degree of freedom. Should this dof not exist, 0 will be returnd.

Reimplemented from SpaceTransformation.

Definition at line 68 of file WaveguideTransformation.cpp.

```
                                                    {
  std::pair<ResponsibleComponent, unsigned int> comp = map_dof_index(index);
  switch (comp.first)
  {
    case VerticalDisplacementComponent:
      return vertical_shift.get_dof_value(comp.second);
      break;
    case WaveguideHeightComponent:
      return waveguide_height.get_dof_value(comp.second);
      break;
    case WaveguideWidthComponent:
      return waveguide_width.get_dof_value(comp.second);
      break;
    default:
      break;
  }
  return 0.0;
}
```

get_dof_values()

`Vector< double > WaveguideTransformation::get_dof_values ( ) const [override], [virtual]`

Other objects can use this function to retrieve an array of the current values of the degrees of freedom of the functional we are optimizing.

This also includes restrained degrees of freedom and other functions can be used to determine this property. This has to be done because in different cases the number of restrained degrees of freedom can vary and we want no logic about this in other functions.

Reimplemented from SpaceTransformation.

Definition at line 142 of file WaveguideTransformation.cpp.

```
                                                               {
  Vector<double> ret(n_dofs());
  unsigned int total_counter = 0;
  for(unsigned int i = 0; i < vertical_shift.get_n_dofs(); i++) {
    ret[total_counter] = vertical_shift.get_dof_value(i);
    total_counter ++;
  }
  if(!GlobalParams.keep_waveguide_height_constant) {
    for(unsigned int i = 0; i < waveguide_height.get_n_dofs(); i++) {
      ret[total_counter] = waveguide_height.get_dof_value(i);
```

```
      total_counter ++;
    }
  }
  if(!GlobalParams.keep_waveguide_width_constant) {
    for(unsigned int i = 0; i < waveguide_width.get_n_dofs(); i++) {
      ret[total_counter] = waveguide_width.get_dof_value(i);
      total_counter ++;
    }
  }
  return ret;
}
```

References ShapeFunction::get_dof_value(), ShapeFunction::get_n_dofs(), and n_dofs().

get_free_dof()

```
double WaveguideTransformation::get_free_dof (
              int dof ) const  [override], [virtual]
```

This is a getter for the values of degrees of freedom.

A getter-setter interface was introduced since the values are estimated automatically during the optimization and non-physical systems should be excluded from the domain of possible cases.

Parameters

dof	The index of the degree of freedom to be retrieved from the structure of the modelled waveguide.

Returns

This function returns the value of the requested degree of freedom. Should this dof not exist, 0 will be returnd.

Reimplemented from SpaceTransformation.

Definition at line 87 of file WaveguideTransformation.cpp.

```
87                                                         {
88   std::pair<ResponsibleComponent, unsigned int> comp = map_free_dof_index(index);
89   switch (comp.first)
90   {
91     case VerticalDisplacementComponent:
92       return vertical_shift.get_free_dof_value(comp.second);
93       break;
94     case WaveguideHeightComponent:
95       return waveguide_height.get_free_dof_value(comp.second);
96       break;
97     case WaveguideWidthComponent:
98       return waveguide_width.get_free_dof_value(comp.second);
99       break;
100     default:
101       break;
102   }
103   return 0.0;
104 }
```

get_J()

```
Tensor< 2, 3, double > WaveguideTransformation::get_J (
            Position & ) [override], [virtual]
```

Compute the Jacobian of the current transformation at a given location.

Returns

Tensor<2,3,double> Jacobian matrix at the given location.

Reimplemented from SpaceTransformation.

Definition at line 202 of file WaveguideTransformation.cpp.

```
202                                                              {
203   Tensor<2,3,double> ret = I;
204   const double z = in_p[2];
205   if(GlobalParams.keep_waveguide_height_constant && GlobalParams.keep_waveguide_width_constant) {
206     // Only shift down vertically
207     ret[1][2] = -vertical_shift.evaluate_derivative_at(z);
208   } else {
209     const double y = in_p[1];
210     const double h = waveguide_height.evaluate_at(z);
211     const double dh = waveguide_height.evaluate_derivative_at(z);
212     const double dm = vertical_shift.evaluate_derivative_at(z);
213     if(GlobalParams.keep_waveguide_width_constant) {
214       // Vertical shift and vertical stretching of the waveguide (variable height)
215       // f(y) = (y / waveguide_height.evaluate_at(z)) - vertical_shift.evaluate_at(z);
216       ret[1][2] = - dm - y * dh / h*h;
217     } else {
218       // Vertical shift, vertical stretching and horizontal stretching of the waveguide (variable height and width)
219       const double w = waveguide_width.evaluate_at(z);
220       const double dw = waveguide_width.evaluate_derivative_at(z);
221       const double x = in_p[0];
222       ret[0][2] = - x  * dw / (w*w);
223       ret[1][2] = - dm - y * dh / h*h;
224     }
225   }
226 
227   return ret;
228 }
```

Referenced by get_J_inverse(), and get_Space_Transformation_Tensor().

get_J_inverse()

```
Tensor< 2, 3, double > WaveguideTransformation::get_J_inverse (
            Position & ) [override], [virtual]
```

Compute the Jacobian of the current transformation at a given location and invert it.

Returns

Tensor<2,3,double> Inverse of the jacobian matrix at the given location.

Reimplemented from SpaceTransformation.

Definition at line 230 of file WaveguideTransformation.cpp.

```
230                                                                      {
231   Tensor<2,3,double> ret = get_J(in_p);
232   return invert(ret);
233 }
```

References get_J().

get_Space_Transformation_Tensor()

```
Tensor< 2, 3, double > WaveguideTransformation::get_Space_Transformation_Tensor (
              Position &  )  [override], [virtual]
```

Get the real part of the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 192 of file WaveguideTransformation.cpp.

```
192                                                                             {
193   Tensor<2, 3, double> J_loc = get_J(position);
194
195   Tensor<2, 3, double> ret;
196   ret[0][0] = 1;
197   ret[1][1] = 1;
198   ret[2][2] = 1;
199   return (J_loc * ret * transpose(J_loc)) / determinant(J_loc);
200 }
```

References get_J().

Referenced by get_Tensor().

get_Tensor()

```
Tensor< 2, 3, ComplexNumber > WaveguideTransformation::get_Tensor (
              Position &  )  [override], [virtual]
```

Get the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 3×3 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 64 of file WaveguideTransformation.cpp.

```
64                                                           {
65   return get_Space_Transformation_Tensor(position);
66 }
```

References get_Space_Transformation_Tensor().

math_to_phys()

```
Position WaveguideTransformation::math_to_phys (
              Position coord ) const  [override], [virtual]
```

Transforms a coordinate in the mathematical coord system to physical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the mathematical system

Returns

Position Coordinate in the physical system

Implements SpaceTransformation.

Definition at line 31 of file WaveguideTransformation.cpp.

```
31                                                                    {
32   Position ret;
33   if(GlobalParams.keep_waveguide_width_constant) {
34     ret[0] = coord[0];
35   } else {
36     ret[0] = coord[0] * waveguide_width.evaluate_at(coord[2]);
37   }
38   if(GlobalParams.keep_waveguide_height_constant) {
39     ret[1] = coord[1] + vertical_shift.evaluate_at(coord[2]);
40   } else {
41     ret[1] = (coord[1] + vertical_shift.evaluate_at(coord[2])) * waveguide_height.evaluate_at(coord[2]);
42   }
43   ret[2] = coord[2];
44   return ret;
45 }
```

n_dofs()

`unsigned int WaveguideTransformation::n_dofs ( ) const  [override], [virtual]`

This function returns the total number of DOFs including restrained ones.

This is the lenght of the array returned by Dofs().

Reimplemented from SpaceTransformation.

Definition at line 180 of file WaveguideTransformation.cpp.

```
180                                                      {
181   unsigned int ret = vertical_shift.n_dofs;
182   if(!GlobalParams.keep_waveguide_height_constant) {
183     ret += waveguide_height.n_dofs;
184   }
185   if(!GlobalParams.keep_waveguide_width_constant) {
186     ret += waveguide_width.n_dofs;
187   }
188   return ret;
189 }
```

Referenced by get_dof_values().

phys_to_math()

```
Position WaveguideTransformation::phys_to_math (
              Position coord ) const  [override], [virtual]
```

Transforms a coordinate in the physical coord system to mathematical ones.

The implementations in the derived classes are crucial to understand the transformation.

Parameters

coord	Coordinate in the physical system

Returns

Position Coordinate in the mathematical system

Implements SpaceTransformation.

Definition at line 47 of file WaveguideTransformation.cpp.

```
47                                                                        {
48   Position ret;
49   if(GlobalParams.keep_waveguide_width_constant) {
50     ret[0] = coord[0];
51   } else {
52     ret[0] = coord[0] / waveguide_width.evaluate_at(coord[2]);
53   }
54   if(GlobalParams.keep_waveguide_height_constant) {
55     ret[1] = coord[1] - vertical_shift.evaluate_at(coord[2]);
56   } else {
57     ret[1] = (coord[1] / waveguide_height.evaluate_at(coord[2])) - vertical_shift.evaluate_at(coord[2]);
58   }
59   ret[2] = coord[2];
60   return ret;
61 }
```

set_free_dof()

```
void WaveguideTransformation::set_free_dof (
              int dof,
              double value )  [override], [virtual]
```

This function sets the value of the dof provided to the given value.

It is important to consider, that some dofs are non-writable (i.e. the values of the degrees of freedom on the boundary, like the radius of the input-connector cannot be changed).

Parameters

dof	The index of the parameter to be changed.
value	The value, the dof should be set to.

Reimplemented from SpaceTransformation.

Definition at line 106 of file WaveguideTransformation.cpp.

```
106                                                                       {
107    std::pair<ResponsibleComponent, unsigned int> comp = map_free_dof_index(index);
108    switch (comp.first)
109    {
110      case VerticalDisplacementComponent:
111        vertical_shift.set_free_dof_value(comp.second, value);
112        return;
113        break;
```

```
    case WaveguideHeightComponent:
      waveguide_height.set_free_dof_value(comp.second, value);
      return;
      break;
    case WaveguideWidthComponent:
      waveguide_width.set_free_dof_value(comp.second, value);
      return;
      break;
    default:
      break;
  }
  std::cout « "There was an error setting a free dof value." « std::endl;
  return;
}
```

The documentation for this class was generated from the following files:

- Code/SpaceTransformations/WaveguideTransformation.h
- Code/SpaceTransformations/WaveguideTransformation.cpp

File Documentation

82 Code/BoundaryCondition/BoundaryCondition.h File Reference

Contains the BoundaryCondition base type which serves as the abstract base class for all boundary conditions.

```
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <vector>
#include "../Core/Types.h"
#include "./HSIEPolynomial.h"
#include "../Core/FEDomain.h"
```

Classes

- class BoundaryCondition

  This is the base type for boundary coniditions. Some implementations are done on this level, some in the derived types.

82.1 Detailed Description

Contains the BoundaryCondition base type which serves as the abstract base class for all boundary conditions.

83 Code/BoundaryCondition/DirichletSurface.h File Reference

Contains the implementation of Dirichlet tangential data on a boundary.

```
#include "../Core/Types.h"
#include "./BoundaryCondition.h"
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/lac/affine_constraints.h>
```

Classes

- class DirichletSurface

  This class implements dirichlet data on the given surface.

83.1 Detailed Description

Contains the implementation of Dirichlet tangential data on a boundary.

84 Code/BoundaryCondition/DofData.h File Reference

Contains an internal data type.

```
#include "../Core/Enums.h"
#include <string>
```

Classes

- struct DofData

  This struct is used to store data about degrees of freedom for Hardy space infinite elements. This datatype is somewhat internal and should not require additional work.

84.1 Detailed Description

Contains an internal data type.

85 Code/BoundaryCondition/EmptySurface.h File Reference

Contains the implementation of an empty surface, i.e. dirichlet zero trace.

```
#include "../Core/Types.h"
#include "./BoundaryCondition.h"
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/lac/affine_constraints.h>
```

Classes

- class EmptySurface

  A surface with tangential component of the solution equals zero, i.e. specialization of the dirichlet surface.

85.1 Detailed Description

Contains the implementation of an empty surface, i.e. dirichlet zero trace.

86 Code/BoundaryCondition/HSIEPolynomial.h File Reference

Contains the implementation of a Hardy polynomial which is required for the Hardy Space infinite elements.

```
#include <deal.II/lac/full_matrix.h>
#include "DofData.h"
#include "../Core/Types.h"
```

Classes

- class HSIEPolynomial

  This class basically represents a polynomial and its derivative. It is required for the HSIE implementation.

86.1 Detailed Description

Contains the implementation of a Hardy polynomial which is required for the Hardy Space infinite elements.

87 Code/BoundaryCondition/HSIESurface.h File Reference

Implementation of a boundary condition based on Hardy Space infinite elements.

```
#include "../Core/Types.h"
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_nedelec.h>
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/tria.h>
#include "DofData.h"
#include "HSIEPolynomial.h"
#include "../Helpers/Parameters.h"
#include "./BoundaryCondition.h"
```

Classes

- class HSIESurface

  This class implements Hardy space infinite elements on a provided surface.

87.1 Detailed Description

Implementation of a boundary condition based on Hardy Space infinite elements.

88 Code/BoundaryCondition/JacobianForCell.h File Reference

An internal datatype.

```
#include <deal.II/base/tensor.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/differentiation/sd/symengine_number_types.h>
#include "../Core/Types.h"
```

Classes

- class JacobianForCell

  This class is only for internal use.

88.1 Detailed Description

An internal datatype.

89 Code/BoundaryCondition/LaguerreFunction.h File Reference

An implementation of Laguerre functions which is not currently being used.

Classes

- class LaguerreFunction

89.1 Detailed Description

An implementation of Laguerre functions which is not currently being used.

90 Code/BoundaryCondition/NeighborSurface.h File Reference

An implementation of a surface that handles the communication with a neighboring process.

```
#include "../Core/Types.h"
#include "./BoundaryCondition.h"
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/lac/affine_constraints.h>
```

Classes

- class NeighborSurface

    For non-local problem, these interfaces are ones, that connect two inner domains and handle the communication between the two as well as the adjacent boundaries. This matrix has no effect for the assembly of system matrices since these boundaries have no own dofs. This object mainly communicates dof indices during the initialization phase.

90.1 Detailed Description

An implementation of a surface that handles the communication with a neighboring process.

91 Code/BoundaryCondition/PMLMeshTransformation.h File Reference

Coordinate transformation for PML domains.

```
#include <utility>
#include "../Core/Types.h"
```

Classes

- class PMLMeshTransformation

    Generating the basic mesh for a PML domain is simple because it is an axis parallel cuboid. This functions shifts and stretches the domain to the correct proportions.

91.1 Detailed Description

Coordinate transformation for PML domains.

92 Code/BoundaryCondition/PMLSurface.h File Reference

Implementation of the PML Surface class.

```
#include "../Core/Types.h"
#include "./BoundaryCondition.h"
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/lac/affine_constraints.h>
#include "./PMLMeshTransformation.h"
```

Classes

- class PMLSurface

An implementation of a UPML method.

92.1 Detailed Description

Implementation of the PML Surface class.

93 Code/Core/Enums.h File Reference

All the enums used in this project.

Enumerations

- enum **SweepingDirection** { **X** = 0, **Y** = 1, **Z** = 2 }
- enum **DofType** {
  EDGE, **SURFACE**, **RAY**, **IFFa**,
  IFFb, **SEGMENTa**, **SEGMENTb** }
- enum **Direction** {
  MinusX = 0, **PlusX** = 1, **MinusY** = 2, **PlusY** = 3,
  MinusZ = 4, **PlusZ** = 5 }
- enum **ConnectorType** { **Circle**, **Rectangle** }
- enum **BoundaryConditionType** { **PML**, **HSIE** }
- enum **Evaluation_Domain** { **CIRCLE_CLOSE**, **CIRCLE_MAX**, **RECTANGLE_INNER** }
- enum **SurfaceType** { **OPEN_SURFACE**, **NEIGHBOR_SURFACE**, **ABC_SURFACE**, **DIRICHLET_SURFACE** }
- enum **Evaluation_Metric** { **FUNDAMENTAL_MODE_EXCITATION**, **POYNTING_TYPE_ENERGY** }
- enum **SpecialCase** {
  none, **reference_bond_nr_0**, **reference_bond_nr_1**, **reference_bond_nr_2**,
  reference_bond_nr_40, **reference_bond_nr_41**, **reference_bond_nr_42**, **reference_bond_nr_43**,
  reference_bond_nr_44, **reference_bond_nr_45**, **reference_bond_nr_46**, **reference_bond_nr_47**,
  reference_bond_nr_48, **reference_bond_nr_49**, **reference_bond_nr_50**, **reference_bond_nr_51**,
  reference_bond_nr_52, **reference_bond_nr_53**, **reference_bond_nr_54**, **reference_bond_nr_55**,
  reference_bond_nr_56, **reference_bond_nr_57**, **reference_bond_nr_58**, **reference_bond_nr_59**,
  reference_bond_nr_60, **reference_bond_nr_61**, **reference_bond_nr_62**, **reference_bond_nr_63**,
  reference_bond_nr_64, **reference_bond_nr_65**, **reference_bond_nr_66**, **reference_bond_nr_67**,
  reference_bond_nr_68, **reference_bond_nr_69**, **reference_bond_nr_70**, **reference_bond_nr_71**,
  reference_bond_nr_72 }
- enum **OptimizationSchema** { **FD**, **Adjoint** }
- enum **SolverOptions** {
  GMRES, **MINRES**, **BICGS**, **TFQMR**,
  PCONLY, **S_CG** }
- enum **PreconditionerOptions** { **Sweeping**, **FastSweeping**, **HSIESweeping**, **HSIEFastSweeping** }
- enum **SteppingMethod** { **Steepest**, **BFGS** }

- enum **TransformationType** { **WavegeuideTransformationType**, **AngleWaveguideTransformationType**, **BendTransformationType**, **PredefinedShapeTransformationType** }

93.1 Detailed Description

All the enums used in this project.

94 Code/Core/FEDomain.h File Reference

A base class for all objects that have either locally owned or active dofs.

```
#include <deal.II/base/index_set.h>
#include <climits>
#include "../Core/Types.h"
```

Classes

- class FEDomain

  This class is a base type for all objects that own their own dofs.

94.1 Detailed Description

A base class for all objects that have either locally owned or active dofs.

95 Code/Core/InnerDomain.h File Reference

Contains the implementation of the inner domain which handles the part of the computational domain that is locally owned.

```
#include <sys/stat.h>
#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>
#include <sstream>
#include <deal.II/base/function.h>
#include <deal.II/base/index_set.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/multithread_info.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/point.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_management.h>
#include <deal.II/base/timer.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_handler.h>
```

```
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/filtered_iterator.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_gmres.h>
#include <deal.II/lac/sparse_direct.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/lac/petsc_vector.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include <deal.II/lac/la_parallel_vector.h>
#include "../Core/Types.h"
#include "../Solutions/ExactSolution.h"
#include "../GlobalObjects/ModeManager.h"
#include "../Helpers/ParameterReader.h"
#include "../Helpers/Parameters.h"
#include "../Helpers/staticfunctions.h"
#include "./Sector.h"
#include "../MeshGenerators/SquareMeshGenerator.h"
#include "../Core/Enums.h"
#include <deal.II/base/convergence_table.h>
#include <deal.II/base/table_handler.h>
#include "../GlobalObjects/GlobalObjects.h"
#include "./FEDomain.h"
```

Classes

- class InnerDomain

  This class encapsulates all important mechanism for solving a FEM problem. In earlier versions this also included space transformation and computation of materials. Now it only includes FEM essentials and solving the system matrix.

95.1 Detailed Description

Contains the implementation of the inner domain which handles the part of the computational domain that is locally owned.

96 Code/Core/Sector.h File Reference

Contains the header of the Sector class.

```
#include <deal.II/base/tensor.h>
```

Classes

- class Sector< Dofs_Per_Sector >

  Sectors are used, to split the computational domain into chunks, whose degrees of freedom are likely coupled.

96.1 Detailed Description

Contains the header of the Sector class.

97 Code/Core/Types.h File Reference

This file contains all type declarations used in this project.

```
#include <array>
#include <vector>
#include <complex>
#include <deal.II/base/point.h>
#include <deal.II/differentiation/sd/symengine_number_types.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include <deal.II/lac/petsc_vector.h>
#include <deal.II/base/index_set.h>
#include "../BoundaryCondition/DofData.h"
```

Classes

- struct LocalMatrixPart
- struct EdgeAngelingData
- struct VertexAngelingData
- struct CellAngelingData

- struct DofOwner
- struct FileMetaData
- struct RayAngelingData
- struct BoundaryInformation
- struct DofCouplingInformation
- struct InterfaceDofData
- struct DofAssociation
- struct JacobianAndTensorData
- struct DofCountsStruct
- struct LevelDofOwnershipData
- struct ConstraintPair
- struct SurfaceCellData
- struct DataSeries
- struct FEErrorStruct
- struct FEAdjointEvaluation
- struct J_derivative_terms

Typedefs

- using **EFieldComponent** = std::complex< double >
- using **EFieldValue** = std::array< EFieldComponent, 3 >
- using **DofCount** = unsigned int
- using **Position** = dealii::Point< 3, double >
- using **Position2D** = dealii::Point< 2, double >
- using **DofNumber** = unsigned int
- using **DofSortingData** = std::pair< DofNumber, Position >
- using **NumericVectorLocal** = dealii::Vector< EFieldComponent >
- using **NumericVectorDistributed** = dealii::PETScWrappers::MPI::Vector
- using **SparseComplexMatrix** = dealii::PETScWrappers::MPI::SparseMatrix
- using **SweepingLevel** = unsigned int
- using **HSIEElementOrder** = unsigned int
- using **NedelecElementOrder** = unsigned int
- using **BoundaryId** = unsigned int
- using **ComplexNumber** = std::complex< double >
- using **DofHandler2D** = dealii::DoFHandler< 2 >
- using **DofHandler3D** = dealii::DoFHandler< 3 >
- using **CellIterator2D** = DofHandler2D::active_cell_iterator
- using **CellIterator3D** = DofHandler3D::active_cell_iterator
- using **DofDataVector** = std::vector< DofData >
- using **MathExpression** = dealii::Differentiation::SD::Expression

- using **Mesh** = dealii::Triangulation< 3 >
- using **MaterialTensor** = dealii::Tensor< 2, 3, ComplexNumber >
- using **FaceAngelingData** = std::array< RayAngelingData, 4 >
- using **CubeSurfaceTruncationState** = std::array< bool, 6 >
- using **DofFieldTrace** = std::vector< ComplexNumber >
- using **Constraints** = dealii::AffineConstraints< ComplexNumber >
- using **DofIndexVector** = std::vector< DofNumber >

Enumerations

- enum **SignalTaperingType** { **C1**, **C0** }
- enum **SignalCouplingMethod** { **Tapering**, **Dirichlet** }
- enum **FileType** { **ConvergenceCSV**, **ParaviewVTU**, **TexReport**, **MetaText** }
- enum **LoggerEntryType** { **ConvergenceHistoryEntry**, **FinalConvergenceStep**, **SolverMetaData** }
- enum **LoggingLevel** { **DEBUG_ALL**, **DEBUG_ONE**, **PRODUCTION_ALL**, **PRODUCTION_ONE** }

Variables

- const double **FLOATING_PRECISION** = 0.00001
- const std::vector< std::vector< unsigned int > > **edge_to_boundary_id**

97.1 Detailed Description

This file contains all type declarations used in this project.

97.2 Variable Documentation

edge_to_boundary_id

```
const std::vector<std::vector<unsigned int> > edge_to_boundary_id
```

Initial value:

```
= {
    {4,5,2,3}, {5,4,2,3}, {0,1,4,5}, {0,1,5,4}, {1,0,2,3}, {0,1,2,3}
}
```

Definition at line 60 of file Types.h.

98 Code/GlobalObjects/GeometryManager.h File Reference

Contains the GeometryManager header, which handles the distribution of the computational domain onto processes and most of the initialization.

```
#include <deal.II/base/index_set.h>
#include "../Core/Types.h"
#include "../BoundaryCondition/BoundaryCondition.h"
#include <memory>
#include <utility>
#include "../Core/Enums.h"
```

Classes

- struct LevelGeometry
- class GeometryManager

  One object of this type is globally available to handle the geometry of the computation (what is the global computational domain, what is computed locally).

98.1 Detailed Description

Contains the GeometryManager header, which handles the distribution of the computational domain onto processes and most of the initialization.

99 Code/GlobalObjects/GlobalObjects.h File Reference

Contains the declaration of some global objects that contain the parameter values as well as some values derived from them, like the geometry and information about other processes.

```
#include "../Helpers/Parameters.h"
#include "GeometryManager.h"
#include "../Hierarchy/MPICommunicator.h"
#include "ModeManager.h"
#include "OutputManager.h"
#include "TimerManager.h"
#include "../SpaceTransformations/SpaceTransformation.h"
```

Functions

- void **initialize_global_variables** (const std::string run_file, const std::string case_file, std::string override_data="")

Variables

- Parameters **GlobalParams**

- GeometryManager **Geometry**
- MPICommunicator **GlobalMPI**
- ModeManager **GlobalModeManager**
- OutputManager **GlobalOutputManager**
- TimerManager **GlobalTimerManager**
- SpaceTransformation * **GlobalSpaceTransformation**

99.1 Detailed Description

Contains the declaration of some global objects that contain the parameter values as well as some values derived from them, like the geometry and information about other processes.

100 Code/GlobalObjects/ModeManager.h File Reference

Not currently in use.

```
#include <deal.II/base/point.h>
```

Classes

- class ModeManager

100.1 Detailed Description

Not currently in use.

101 Code/GlobalObjects/OutputManager.h File Reference

Creates filenames and manages file system paths.

```
#include "../Core/Types.h"
#include <sys/stat.h>
#include <iostream>
#include <fstream>
```

Classes

- class OutputManager

  Whenever we write output, we require filenames.

101.1 Detailed Description

Creates filenames and manages file system paths.

102 Code/GlobalObjects/TimerManager.h File Reference

Implementation of a handler for multiple timers with names that can gernerate output.

```
#include <deal.II/base/timer.h>
#include <array>
```

Classes

- class TimerManager

  A class that stores timers for later output.

102.1 Detailed Description

Implementation of a handler for multiple timers with names that can gernerate output.

103 Code/Helpers/ParameterOverride.h File Reference

A utility class that overrides certain parameters from an input file.

```
#include <string>
#include "Parameters.h"
```

Classes

- class ParameterOverride

  An object used to interpret command line arguments of type –override.

103.1 Detailed Description

A utility class that overrides certain parameters from an input file.

104 Code/Helpers/ParameterReader.h File Reference

Contains the parameter reader header. This object parses the parameter files.

```
#include <deal.II/base/parameter_handler.h>
#include "../Core/InnerDomain.h"
```

Classes

- class ParameterReader

  This class is used to gather all the information from the input file and store it in a static object available to all processes.

104.1 Detailed Description

Contains the parameter reader header. This object parses the parameter files.

105 Code/Helpers/Parameters.h File Reference

A struct containing all provided parameter values and some computed values based on it (like MPI rank etc.)

```
#include <mpi.h>
#include <string>
#include "ShapeDescription.h"
#include "../Core/Types.h"
#include "../Core/Enums.h"
```

Classes

- class Parameters

  This structure contains all information contained in the input file and some values that can simply be computed from it.

105.1 Detailed Description

A struct containing all provided parameter values and some computed values based on it (like MPI rank etc.)

106 Code/Helpers/PointSourceField.h File Reference

Some implementations of fields that can be used in the code for forcing or error computation.

```
#include <deal.II/base/function.h>
#include "../Core/Types.h"
```

Classes

- class PointSourceFieldHertz
- class PointSourceFieldCosCos

106.1 Detailed Description

Some implementations of fields that can be used in the code for forcing or error computation.

107 Code/Helpers/PointVal.h File Reference

Not currently used.

```
#include "../Core/Types.h"
```

Classes

- class PointVal

  Old class that was used for the interpolation of input signals.

107.1 Detailed Description

Not currently used.

108 Code/Helpers/ShapeDescription.h File Reference

An object used to wrap the description of the prescribed waveguide shapes.

```
#include <string>
#include <vector>
```

Classes

- class ShapeDescription

108.1 Detailed Description

An object used to wrap the description of the prescribed waveguide shapes.

109 Code/Helpers/staticfunctions.h File Reference

This is an important file since it contains all the utility functions used anywhere in the code.

```
#include <deal.II/base/index_set.h>
#include <deal.II/base/point.h>
#include <deal.II/base/tensor.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/dofs/dof_handler.h>
```

```
#include <deal.II/lac/affine_constraints.h>
#include <fstream>
#include "./Parameters.h"
#include "./ParameterOverride.h"
#include "../Core/Types.h"
```

Functions

- Tensor< 1, 3, double > crossproduct (Tensor< 1, 3, double >, Tensor< 1, 3, double >)

    For given vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^3$*, this function calculates the following crossproduct:*
- std::string **exec** (const char ∗cmd)
- ComplexNumber **matrixD** (int in_row, int in_column, ComplexNumber in_k0)
- std::pair< DofNumber, DofNumber > **get_max_and_min_dof_for_interface_data** (std::vector< InterfaceDofData > in_data)
- bool **comparePositions** (Position p1, Position p2)
- bool **compareDofBaseData** (std::pair< DofNumber, Position > c1, std::pair< DofNumber, Position > c2)
- bool **compareDofBaseDataAndOrientation** (InterfaceDofData, InterfaceDofData)
- bool **compareSurfaceCellData** (SurfaceCellData c1, SurfaceCellData c2)
- bool **compareDofDataByGlobalIndex** (InterfaceDofData, InterfaceDofData)
- bool **areDofsClose** (const InterfaceDofData &a, const InterfaceDofData &b)
- bool **compareFEAdjointEvals** (const FEAdjointEvaluation field_a, const FEAdjointEvaluation field_b)
- double **dotproduct** (Tensor< 1, 3, double >, Tensor< 1, 3, double >)
- void **mesh_info** (Triangulation< 3 > ∗, std::string)
- template<int dim>
  void **mesh_info** (const Triangulation< dim >)
- Parameters **GetParameters** (const std::string run_file, const std::string case_file, ParameterOverride &in_po)
- Position **Triangulation_Shit_To_Local_Geometry** (const Position &p)
- Position **Transform_4_to_5** (const Position &p)
- Position **Transform_3_to_5** (const Position &p)
- Position **Transform_2_to_5** (const Position &p)
- Position **Transform_1_to_5** (const Position &p)
- Position **Transform_0_to_5** (const Position &p)
- Position **Transform_5_to_4** (const Position &p)
- Position **Transform_5_to_3** (const Position &p)
- Position **Transform_5_to_2** (const Position &p)
- Position **Transform_5_to_1** (const Position &p)
- Position **Transform_5_to_0** (const Position &p)
- bool **file_exists** (const std::string &name)

- double **Distance2D** (const Position &, const Position &=Position())
- double **Distance3D** (const Position &, const Position &=Position())
- std::vector< types::global_dof_index > **Add_Zero_Restraint** (AffineConstraints< double > ∗, DoFHandler< 3 >::active_cell_iterator &, unsigned int, unsigned int, unsigned int, bool, IndexSet)
- void **add_vector_of_indices** (IndexSet ∗, std::vector< types::global_dof_index >)
- double **hmax_for_cell_center** (Position)
- double **InterpolationPolynomial** (double, double, double, double, double)
- double **InterpolationPolynomialDerivative** (double, double, double, double, double)
- double **InterpolationPolynomialZeroDerivative** (double, double, double)
- double **sigma** (double, double, double)
- auto **compute_center_of_triangulation** (const Mesh ∗) -> Position
- bool **get_orientation** (const Position &vertex_1, const Position &vertex_2)
- NumericVectorLocal **crossproduct** (const NumericVectorLocal &u, const NumericVectorLocal &v)
- Position **crossproduct** (const Position &u, const Position &v)
- void **multiply_in_place** (const ComplexNumber factor_1, NumericVectorLocal &factor_2)
- void **print_info** (const std::string &label, const std::string &message, LoggingLevel level=LoggingLevel::DEBUG_O
- void **print_info** (const std::string &label, const unsigned int message, LoggingLevel level=LoggingLevel::DEBUG_O
- void **print_info** (const std::string &label, const std::vector< unsigned int > &message, LoggingLevel level=LoggingLevel::DEBUG_ONE)
- void **print_info** (const std::string &label, const std::array< bool, 6 > &message, LoggingLevel level=LoggingLevel::DEBUG_ONE)
- bool **is_visible_message_in_current_logging_level** (LoggingLevel level=LoggingLevel::DEBUG_ONE)
- void **write_print_message** (const std::string &label, const std::string &message)
- BoundaryId **opposing_Boundary_Id** (BoundaryId b_id)
- bool **are_opposing_sites** (BoundaryId a, BoundaryId b)
- std::vector< DofCouplingInformation > **get_coupling_information** (std::vector< InterfaceDofData > &dofs_interface_1, std::vector< InterfaceDofData > &dofs_interface_2)
- Position **deal_vector_to_position** (NumericVectorLocal &inp)
- auto **get_affine_constraints_for_InterfaceData** (std::vector< InterfaceDofData > &dofs_interface_1, std::vector< InterfaceDofData > &dofs_interface_2, const unsigned int max_dof) -> Constraints
- void **shift_interface_dof_data** (std::vector< InterfaceDofData > ∗dofs_interface_1, unsigned int shift)
- dealii::Triangulation< 3 > **reforge_triangulation** (dealii::Triangulation< 3 > ∗original_triangulation)
- ComplexNumber **conjugate** (const ComplexNumber &in_number)
- bool **is_absorbing_boundary** (SurfaceType in_st)
- double **norm_squared** (const ComplexNumber in_c)
- bool **are_edge_dofs_locally_owned** (BoundaryId self, BoundaryId other, unsigned int in_level)
- std::vector< BoundaryId > **get_adjacent_boundary_ids** (BoundaryId self)
- SweepingDirection **get_sweeping_direction_for_level** (unsigned int in_level)
- int **generate_tag** (unsigned int global_rank_sender, unsigned int receiver, unsigned int level)

- std::vector< std::string > **split** (std::string str, std::string token)
- SolverOptions **solver_option** (std::string in_name)
- std::vector< double > **fe_evals_to_double** (const std::vector< FEAdjointEvaluation > &inp)
- std::vector< FEAdjointEvaluation > **fe_evals_from_double** (const std::vector< double > &inp)
- Position **adjoint_position_transformation** (const Position in_p)
- dealii::Tensor< 1, 3, ComplexNumber > **adjoint_field_transformation** (const dealii::Tensor< 1, 3, ComplexNumber > in_field)

Variables

- std::string **solutionpath**
- std::ofstream **log_stream**
- std::string **constraints_filename**
- std::string **assemble_filename**
- std::string **precondition_filename**
- std::string **solver_filename**
- std::string **total_filename**
- int **StepsR**
- int **StepsPhi**
- int **alert_counter**
- std::string **input_file_name**

109.1 Detailed Description

This is an important file since it contains all the utility functions used anywhere in the code.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

109.2 Function Documentation

crossproduct()

```
Tensor<1, 3, double> crossproduct (
            Tensor< 1, 3, double > ,
            Tensor< 1, 3, double >  )
```

For given vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^3$, this function calculates the following crossproduct:

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$

Definition at line 224 of file staticfunctions.cpp.

```
                                                   {
    Tensor<1, 3, double> ret;
    ret[0] = a[1] * b[2] - a[2] * b[1];
    ret[1] = a[2] * b[0] - a[0] * b[2];
    ret[2] = a[0] * b[1] - a[1] * b[0];
    return ret;
}
```

110 Code/Hierarchy/HierarchicalProblem.h File Reference

This class contains a forward declaration of LocalProblem and NonLocalProblem and the class HierarchicalProblem.

```
#include "../Core/Types.h"
#include "../Helpers/Parameters.h"
#include "DofIndexData.h"
#include <deal.II/base/index_set.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include <deal.II/lac/petsc_vector.h>
#include "../Core/FEDomain.h"
#include "../OutputGenerators/Images/ResidualOutputGenerator.h"
```

Classes

- class HierarchicalProblem

    The base class of the SweepingPreconditioner and general finite element system.

- struct SampleShellPC

110.1 Detailed Description

This class contains a forward declaration of LocalProblem and NonLocalProblem and the class HierarchicalProblem.

111 Code/Hierarchy/MPICommunicator.h File Reference

This class stores the implementation of the MPICommunicator type.

```
#include <mpi.h>
#include <vector>
#include "../Core/Enums.h"
```

Classes

- class MPICommunicator

  Utility class that provides additional information about the MPI setup on the level.

111.1 Detailed Description

This class stores the implementation of the MPICommunicator type.

112 Code/Hierarchy/NonLocalProblem.h File Reference

This file includes the class NonLocalProblem which is the essential class for the hierarchical sweeping preconditioner.

```
#include "../Core/Types.h"
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <mpi.h>
#include <complex>
#include "HierarchicalProblem.h"
#include "./LocalProblem.h"
#include <deal.II/lac/solver_control.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include "../Core/Enums.h"
```

Classes

- class NonLocalProblem

  The NonLocalProblem class is part of the sweeping preconditioner hierarchy.

112.1 Detailed Description

This file includes the class NonLocalProblem which is the essential class for the hierarchical sweeping preconditioner.

113 Code/MeshGenerators/SquareMeshGenerator.h File Reference

```
#include <deal.II/base/point.h>
#include <deal.II/grid/tria.h>
#include <array>
#include <vector>
#include "./SquareMeshGenerator.h"
#include "../Core/Types.h"
```

Classes

- class SquareMeshGenerator

  This class generates meshes, that are used to discretize a rectangular Waveguide.

114 Code/ModalComputations/RectangularMode.h File Reference

This is no longer active code.

```
#include <deal.II/base/function.h>
#include <deal.II/base/index_set.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/multithread_info.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/point.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_management.h>
#include <deal.II/base/timer.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_nedelec_sz.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/grid/filtered_iterator.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_gmres.h>
```

```
#include <deal.II/lac/sparse_direct.h>
#include <deal.II/lac/sparsity_pattern.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/lac/petsc_vector.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include <deal.II/lac/la_parallel_vector.h>
#include "../Core/Types.h"
#include "../BoundaryCondition/HSIESurface.h"
```

Classes

- class RectangularMode

    Legacy code.

114.1 Detailed Description

This is no longer active code.

115 Code/Optimization/ShapeFunction.h File Reference

Stores the implementation of the ShapeFunction Class.

```
#include <vector>
```

Classes

- class ShapeFunction

    These objects are used in the shape optimization code.

115.1 Detailed Description

Stores the implementation of the ShapeFunction Class.

116 Code/Runners/OptimizationRun.h File Reference

Contains the Optimization Runner which performs shape optimization type computations.

```
#include "../GlobalObjects/GeometryManager.h"
#include "../Helpers/Parameters.h"
#include "../Hierarchy/NonLocalProblem.h"
#include <functional>
```

Classes

- class OptimizationRun

  This runner performs a shape optimization run based on adjoint based shape optimization.

116.1 Detailed Description

Contains the Optimization Runner which performs shape optimization type computations.

117 Code/Runners/ParameterSweep.h File Reference

Contains the parameter sweep runner which is somewhat deprecated.

```
#include "./Simulation.h"
#include "../GlobalObjects/GeometryManager.h"
#include "../Helpers/Parameters.h"
#include "../Hierarchy/NonLocalProblem.h"
#include "../ModalComputations/RectangularMode.h"
```

Classes

- class ParameterSweep

  The Parameter run performs multiple forward runs for a sweep across a parameter value, i.e multiple computations for different domain sizes or similar.

117.1 Detailed Description

Contains the parameter sweep runner which is somewhat deprecated.

118 Code/Runners/Simulation.h File Reference

Base class of the simulation runners.

```
#include "../GlobalObjects/GeometryManager.h"
#include "../Helpers/Parameters.h"
#include "../Hierarchy/NonLocalProblem.h"
#include "../ModalComputations/RectangularMode.h"
```

Classes

- class Simulation

  This base class is very important and abstract.

118.1 Detailed Description

Base class of the simulation runners.

119 Code/Runners/SingleCoreRun.h File Reference

This is deprecated. It is supposed to be used for minature examples that rely on only a Local Problem instead of an object hierarchy.

```
#include "../GlobalObjects/GeometryManager.h"
#include "../Helpers/Parameters.h"
#include "../Hierarchy/NonLocalProblem.h"
#include "../ModalComputations/RectangularMode.h"
```

Classes

- class SingleCoreRun

    In cases in which a single core is enough to solve the problem, this runner can be used.

119.1 Detailed Description

This is deprecated. It is supposed to be used for minature examples that rely on only a Local Problem instead of an object hierarchy.

120 Code/Runners/SweepingRun.h File Reference

Default Runner for sweeping preconditioner runs.

```
#include "../GlobalObjects/GeometryManager.h"
#include "../Helpers/Parameters.h"
#include "../Hierarchy/NonLocalProblem.h"
#include "../ModalComputations/RectangularMode.h"
```

Classes

- class SweepingRun

    This runner constructs a single non-local problem and solves it.

120.1 Detailed Description

Default Runner for sweeping preconditioner runs.

121 Code/SpaceTransformations/WaveguideTransformation.h File Reference

Contains the implementation of the Waveguide Transformation.

```
#include <deal.II/base/point.h>
#include <deal.II/base/tensor.h>
#include <deal.II/lac/vector.h>
#include <math.h>
#include <vector>
#include "../Core/InnerDomain.h"
#include "../Optimization/ShapeFunction.h"
#include "../Core/Sector.h"
#include "SpaceTransformation.h"
```

Classes

- class WaveguideTransformation

  In this case we regard a rectangular waveguide and the effects on the material tensor by the space transformation and the boundary condition PML may overlap.

Enumerations

- enum **ResponsibleComponent** { **VerticalDisplacementComponent**, **WaveguideHeightComponent**, **WaveguideWidthComponent** }

121.1 Detailed Description

Contains the implementation of the Waveguide Transformation.

Bibliography

[Mon92] Peter Monk. “A finite element method for approximating the time-harmonic Maxwell equations”. In: *Numer. Math.* 63.2 (1992), pp. 243–261. ISSN: 0029-599X.

[Bil+18] Muhammad Rodlin Billah et al. “Hybrid integration of silicon photonics circuits and InP lasers by photonic wire bonding”. In: *Optica* 5.7 (July 2018), pp. 876–883. DOI: 10.1364/OPTICA.5.000876. URL: http://www.osapublishing.org/optica/abstract.cfm?URI=optica-5-7-876.

[Neg+18] F. Negredo et al. “Fast and reliable method to estimate losses of single-mode waveguides with an arbitrary 2D trajectory”. In: *Journal of the Optical Society of America A* 35.6 (June 2018), p. 1063. DOI: 10.1364/josaa.35.001063. URL: https://doi.org/10.1364/josaa.35.001063.

[Ott17] Julian Ott. “Halfspace Matching: a Domain Decomposition Method for Scattering by 2D Open Waveguides”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2017. 138 pp. DOI: 10.5445/IR/1000070898.

[Gan+10] Justyna K. Gansel et al. “Gold helix photonic metamaterials: A numerical parameter study”. In: *Opt. Express* 18.2 (Jan. 2010), pp. 1059–1069. DOI: 10.1364/OE.18.001059. URL: http://opg.optica.org/oe/abstract.cfm?URI=oe-18-2-1059.

[Kar+10] Matthias Karl et al. “Reversed pyramids as novel optical micro-cavities”. In: *Superlattices and Microstructures* 47 (Jan. 2010), pp. 83–86. DOI: 10.1016/j.spmi.2009.07.011.

[HZH09] Jin Hu, Xiaoming Zhou, and Gengkai Hu. “Design method for electromagnetic cloak with arbitrary shapes based on Laplace’s equation”. In: *Opt. Express* 17.3 (Feb. 2009), pp. 1308–1320. DOI: 10.1364/OE.17.001308. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-17-3-1308.

[LZ16] Matti Lassas and Ting Zhou. “The Blow-up of Electromagnetic Fields in 3-Dimensional Invisibility Cloaking for Maxwell’s Equations”. In: *SIAM Journal on Applied Mathematics* 76.2 (2016), pp. 457–478. DOI: 10.1137/15M103964X. URL: http://dx.doi.org/10.1137/15M103964X.

[LGG15] Yan Liu, Boris Gralak, and Sebastien Guenneau. “Finite Element Analysis of Electromagnetic Waves in Two-Dimensional Transformed Bianisotropic Media”. In: *Optics Express* (2015). URL: http://arxiv.org/abs/1512.05450.

[Rah+08] Marco Rahm et al. “Design of electromagnetic cloaks and concentrators using form-invariant coordinate transformations of Maxwell’s equations”. In: *Photonics and Nanostructures - Fundamentals and Applications* 6.1 (2008), pp. 87–95. ISSN: 1569-4410. DOI: 10.1016/j.photonics.2007.07.013. URL: http://www.sciencedirect.com/science/article/pii/S1569441007000375.

[KH14] Andreas Kirsch and Frank Hettlich. *The Mathematical Theory of Time-Harmonic Maxwell’s Equations - Expansion-, Integral-, and Variational Methods*. Berlin, Heidelberg: Springer, 2014. ISBN: 978-3-319-11086-8.

[Ged96] S. D. Gedney. “An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices”. In: *IEEE Transactions on Antennas and Propagation* 44.12 (Dec. 1996), pp. 1630–1639. ISSN: 0018-926X. DOI: 10.1109/8.546249.

[WP96] A. J. Ward and J. B. Pendry. "Refraction and geometry in Maxwell's equations". In: *Journal of Modern Optics* 43.4 (1996), pp. 773–793. DOI: 10.1080/09500349608232782.

[NZG10] André Nicolet, Frédéric Zolla, and Christophe Geuzaine. "Transformation Optics, Generalized Cloaking and Superlenses". In: *IEEE Transactions on Magnetics* 46.8 (Aug. 2010), pp. 2975–2981. ISSN: 0018-9464. DOI: 10.1109/tmag.2010.2043073.

[Ber94] Jean-Pierre Berenger. "A perfectly matched layer for the absorption of electromagnetic waves". In: *Journal of Computational Physics* 114.2 (1994), pp. 185–200. ISSN: 0021-9991. DOI: 10.1006/jcph.1994.1159. URL: http://www.sciencedirect.com/science/article/pii/S0021999184711594.

[Zsc+05] Lin Werner Zschiedrich et al. "JCMmode: an adaptive finite element solver for the computation of leaky modes". In: *Integrated Optics: Devices, Materials, and Technologies IX*. Ed. by Yakov Sidorin and Christoph A. Waechter. Vol. 5728. International Society for Optics and Photonics. SPIE, 2005, pp. 192–202. DOI: 10.1117/12.590372. URL: https://doi.org/10.1117/12.590372.

[TH05] Allen Taflove and Susan C. Hagness. *Computational electrodynamics : the finite-difference time-domain method*. 3. ed. Artech House antennas and propagation library. Boston [u.a.]: Artech House, 2005. ISBN: 1580538320; 9781580538329. URL: https://swbplus.bsz-bw.de/bsz121631346inh.htm.

[Sch15] Andreas Schulz. "Numerical Analysis of the Electro-Magnetic Perfectly Matched Layer in a Discontinuous Galerkin Discretization". PhD thesis. 2015. DOI: 10.5445/IR/1000047785.

[NS11] Lothar Nannen and Achim Schädle. "Hardy space infinite elements for Helmholtz-type problems with unbounded inhomogeneities". In: *Wave Motion* 48.2 (2011), pp. 116–129. ISSN: 0165-2125. DOI: 10.1016/j.wavemoti.2010.09.004. URL: http://www.sciencedirect.com/science/article/pii/S016521251000082X.

[Nan+11] L. Nannen et al. "High order Curl-conforming Hardy space infinite elements for exterior Maxwell problems". In: *ArXiv e-prints* (2011). URL: http://adsabs.harvard.edu/abs/2011arXiv1103.2288N.

[Dur70] "Pure and Applied Mathematics". In: *Theory of Hp Spaces*. Ed. by Peter L. Duren. Vol. 38. Pure and Applied Mathematics. Elsevier, 1970, pp. 260–261. DOI: https://doi.org/10.1016/S0079-8169(13)60002-1. URL: https://www.sciencedirect.com/science/article/pii/S0079816913600021.

[Hof14] K. Hoffman. *Banach Spaces of Analytic Functions*. Dover Publications, 2014. ISBN: 9780486149967. URL: https://books.google.de/books?id=YUP%5C_AwAAQBAJ.

[Sch02] Frank Schmidt. "A New Approach to Coupled Interior-Exterior Helmholtz-Type Problems: Theory and Algorithms". In: 2002.

[HN09] Thorsten Hohage and Lothar Nannen. "Hardy Space Infinite Elements for Scattering and Resonance Problems". In: *SIAM Journal on Numerical Analysis* 47.2 (2009), pp. 972–996. DOI: 10.1137/070708044. URL: https://doi.org/10.1137/070708044.

[Pom+07] Jan Pomplun et al. "Finite element simulation of radiation losses in photonic crystal fibers". In: *physica status solidi (a)* 204.11 (Nov. 2007), pp. 3822–3837. ISSN: 1862-6319. DOI: 10.1002/pssa.200776414.

[Agh+08] Y. Ould Agha et al. "On the use of PML for the computation of leaky modes". In: *COMPEL - The international journal for computation and mathematics in electrical and electronic engineering* 27.1 (Jan. 2008). Ed. by Patrick Dular, pp. 95–109. DOI: 10.1108/03321640810836672. URL: https://doi.org/10.1108/03321640810836672.

[Iva+09] Alyona Ivanova et al. "Variational Effective Index Method for 3D Vectorial Scattering Problems in Photonics: TE Polarization". Undefined. In: *PIERS 2009 Moscow Proceedings*. The Electromagnetics Academy, 2009, pp. 1038–1042. ISBN: 978-1-934142-10-3.

[ISH13] Alyona Ivanova, Remco Stoffer, and Manfred Hammer. *A variational mode solver for optical waveguides based on quasi-analytical vectorial slab mode expansion*. 2013. DOI: 10.48550/ARXIV.1307.1315. URL: https://arxiv.org/abs/1307.1315.

[Ham22] M. Hammer. *Mode solver for 2-D multilayer waveguides, variational effective index approximation*. 2022. URL: https://www.siio.eu/eims.html.

[Zag06] Sabine Zaglmayr. "High Order Finite Element Methods for Electromagnetic Field Computation". PhD thesis. 2006. URL: https://www.applied.math.tugraz.at/~zaglmayr/pub/szthesis.pdf.

[BW76] K.J. Bathe and E.L. Wilson. *Numerical Methods in Finite Element Analysis*. Prentice-Hall civil engineering and engineering mechanics series. Prentice-Hall, 1976. ISBN: 9780136271901.

[TEY12] Paul Tsuji, Bjorn Engquist, and Lexing Ying. "A sweeping preconditioner for time-harmonic Maxwell's equations with finite elements". In: *Journal of Computational Physics* 231.9 (2012), pp. 3770–3783. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2012.01.025.

[Dav04] Timothy A Davis. "A column pre-ordering strategy for the unsymmetric-pattern multifrontal method". In: *ACM Transactions on Mathematical Software (TOMS)* 30.2 (2004), pp. 165–195.

[Ame+01] P.R. Amestoy et al. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[Bur+06] Sven Burger et al. "Benchmark of FEM, Waveguide and FDTD Algorithms for Rigorous Mask Simulation". In: *Proceedings of SPIE - The International Society for Optical Engineering* 5992 (Jan. 2006). DOI: 10.1117/12.631696.

[SG04] Olaf Schenk and Klaus Gärtner. "Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO". In: *Future Gener. Comput. Syst.* 20.3 (Apr. 2004), pp. 475–487. ISSN: 0167-739X. DOI: 10.1016/j.future.2003.07.011. URL: https://doi.org/10.1016/j.future.2003.07.011.

[SS86] Youcef Saad and Martin H Schultz. "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems". In: *SIAM J. Sci. Stat. Comput.* 7.3 (July 1986), pp. 856–869. ISSN: 0196-5204.

[Kel] C. T. Kelley. "2. Conjugate Gradient Iteration". In: *Iterative Methods for Linear and Nonlinear Equations*, pp. 11–31. DOI: 10.1137/1.9781611970944.ch2. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611970944.ch2.

[BH96] Rudolf Beck and Ralf Hiptmair. *Multilevel Solution of the Time-Harmonic Maxwell's Equations Based on Edge Elements*. eng. Tech. rep. SC-96-51. Takustr. 7, 14195 Berlin: ZIB, 1996.

[GZ16] Martin J. Gander and Hui Zhang. *A Class of Iterative Solvers for the Helmholtz Equation: Factorizations, Sweeping Preconditioners, Source Transfer, Single Layer Potentials, Polarized Traces, and Optimized Schwarz Methods*. 2016.

[GZ18] Martin J. Gander and Hui Zhang. "Restrictions on the Use of Sweeping Type Preconditioners for Helmholtz Problems". In: *Domain Decomposition Methods in Science and Engineering XXIV*. Ed. by Petter E. Bjørstad et al. Springer International Publishing, 2018, pp. 321–332.

[NW06] Jorge Nocedal and Stephen J. Wright, eds. *Numerical Optimization*. Springer-Verlag, 2006. DOI: 10.1007/b98874.

[Fan10] Haw-ren Fang. "Stability analysis of block LDL^T factorization for symmetric indefinite matrices". In: *IMA Journal of Numerical Analysis* 31.2 (Apr. 2010), pp. 528–555. ISSN: 0272-4979. DOI: 10.1093/imanum/drp053. eprint: https://academic.oup.com/imajna/article-pdf/31/2/528/1930495/drp053.pdf. URL: https://doi.org/10.1093/imanum/drp053.

[EY11] Björn Engquist and Lexing Ying. "Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation". In: *Communications on Pure and Applied Mathematics* 64.5 (2011), pp. 697–735. DOI: 10.1002/cpa.20358. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.20358. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.20358.

[Bla+02] L Susan Blackford et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[WD99] R. Clint Whaley and Jack Dongarra. "Automatically Tuned Linear Algebra Software". In: *Ninth SIAM Conference on Parallel Processing for Scientific Computing*. CD-ROM Proceedings. 1999.

[09] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009. ISBN: 630813-054US.

[NW22] Lothar Nannen and Markus Wess. "Complex-scaled infinite elements for resonance problems in heterogeneous open systems". In: *Advances in Computational Mathematics* 48.2 (Feb. 2022). DOI: 10.1007/s10444-021-09923-1. URL: https://doi.org/10.1007/s10444-021-09923-1.

[Wes20] Markus Wess. "Frequency-dependent complex-scaled infinite elements for exterior Helmholtz resonance problems". PhD thesis. 2020. DOI: 10.34726/HSS.2020.78903. URL: https://repositum.tuwien.at/handle/20.500.12708/15095.

[Wal14] É. Walter. *Numerical Methods and Optimization: A Consumer Guide*. Springer International Publishing, 2014. ISBN: 9783319076713. URL: https://www.springer.com/de/book/9783319076706.

[Sem+15] Johannes Semmler et al. "Shape Optimization in Electromagnetic Applications". In: *New Trends in Shape Optimization*. Ed. by Aldo Pratelli and Günter Leugering. Cham: Springer International Publishing, 2015, pp. 251–269. ISBN: 978-3-319-17563-8. DOI: 10.1007/978-3-319-17563-8_11. URL: https://doi.org/10.1007/978-3-319-17563-8_11.

[Lal+13] Christopher M. Lalau-Keraly et al. "Adjoint shape optimization applied to electromagnetic design". In: *Opt. Express* 21.18 (Sept. 2013), pp. 21693–21701. DOI: 10.1364/OE.21.021693.

[Fle87] Roger Fletcher. *Practical Methods of Optimization*. Second. John Wiley & Sons, 1987.

[Dav91] W. Davidon. "Variable Metric Method for Minimization". In: *SIAM Journal on Optimization* 1.1 (1991), pp. 1–17. DOI: 10.1137/0801001.

[Arn+21] Daniel Arndt et al. "The `deal.II` Library, Version 9.3". In: *Journal of Numerical Mathematics* 29.3 (2021), pp. 171–186. DOI: 10.1515/jnma-2021-0081. URL: https://dealii.org/deal93-preprint.pdf.

[KL17] R.M. Kynch and P.D. Ledger. "Resolving the sign conflict problem for hp-hexahedral Nédélecelements with application to eddy current problems". In: *Computers and Structures* 181 (2017), pp. 41–54. DOI: 10.1016/j.compstruc.2016.05.021. URL: https://doi.org/10.1016/j.compstruc.2016.05.021.