Waveguide Solver

2.1

Generated by Doxygen 1.8.17

Chapter 1

Shape-Optimization of a 3D waveguide using dealii, transformation optics and the finite element method

1.1 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 dealii 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 equaations 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 fullfill the following criteria:

1. The code should be readable to starters (educational purpose),
2. The code should be maintainable (reusability),
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.

1.2 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 unreplacable. 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 dealii itself is C++. This language also forms the backbone id CUDA and manu other, relevant libraries. It is to be considered inevitable in this field. "The vhoice of this language in a wau reduces the importance of the need for a performanc implementation on the code level ∗on the functional or theoretical level this obviously has a very miimal influence on the performanc.(. Also it should be noted that there exists a very large documentation about dealii which might help the reader understand this code. Lastly dealii is basically only available on Linux since it nearly always requires a build-process which would not be possible with out enormous problems on 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.

2 Shape-Optimization of a 3D waveguide using dealii, transformation optics and the finite element method

Author

Pascal Kraft

Version

2.1

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

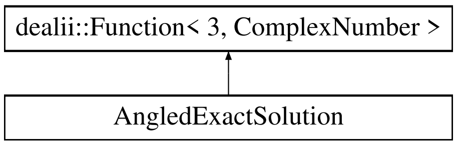
Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 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

5.1.1 Detailed Description

Definition at line 12 of file AngledExactSolution.h.

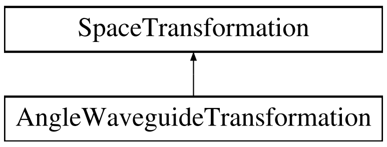
The documentation for this class was generated from the following files:

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

5.2 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

5.2.1 Detailed Description

Author

Pascal Kraft

Date

28.11.2016

Definition at line 20 of file AngleWaveguideTransformation.h.

5.2.2 Member Function Documentation

5.2.2.1 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.

```
                                                             {

}
```

5.2.2.2 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.

```
                                                         {
  if(!is_constant || !is_det_prepared) {
    det = determinant(get_J(c));
    is_det_prepared = true;
  }
  return det;
}
```

References get_J().

Referenced by get_Space_Transformation_Tensor().

5.2.2.3 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.

```
                                                                    {
  Vector<double> ret;
  return ret;
}
```

5.2.2.4 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.

```
                                                                           {
  if(!is_constant || !is_J_prepared) {
    dealii::Tensor<2, 3, double> ret;
    ret[0][0] = 1;
    ret[1][1] = 1;
    ret[2][2] = 1;
    ret[2][1] = -0.2;
    J_perm = ret;
    is_J_prepared = true;
  }
  return J_perm;
}
```

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

5.2.2.5 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.

```
                                                                                   {
  if(!is_constant || !is_J_inv_prepared) {
    dealii::Tensor<2, 3, double> ret = get_J(c);
    ret = invert(ret);
    J_inv_perm = ret;
    is_J_inv_prepared = true;
  }
  return J_inv_perm;
}
```

References get_J().

5.2.2.6 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> 33 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 92 of file AngleWaveguideTransformation.cpp.

```
                                                                          {
  Tensor<2, 3, double> ret;
  ret[0][0] = 1;
  ret[1][1] = 1;
  ret[2][2] = 1;
  return (get_J(p) * ret * transpose(get_J(p))) / get_det(p);
}
```

References get_det(), and get_J().

Referenced by get_Tensor().

5.2.2.7 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> 33 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 66 of file AngleWaveguideTransformation.cpp.

```
                                                              {
  return get_Space_Transformation_Tensor(position);
}
```

References get_Space_Transformation_Tensor().

5.2.2.8 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.

```
                                                                           {
  Position ret;
  ret[0] = coord[0];
  ret[1] = coord[1];
  ret[2] = coord[2] + GlobalParams.PML_Angle_Test*coord[1];
  return ret;
}
```

5.2.2.9 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.

```
                                                           {
  return 0;
}
```

5.2.2.10 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.

```
                                                                          {
  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:

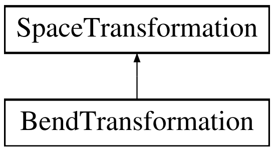
- Code/SpaceTransformations/AngleWaveguideTransformation.h
- Code/SpaceTransformations/AngleWaveguideTransformation.cpp

5.3 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

5.3.1 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.

Author

Pascal Kraft

Date

14.12.2021

Definition at line 27 of file BendTransformation.h.

5.3.2 Member Function Documentation

5.3.2.1 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.

```
                                                       {
  return;
}
```

5.3.2.2 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> 33 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 36 of file BendTransformation.cpp.

```
                                                                       {
  Tensor<2, 3, double> transformation;
  return transformation;
}
```

Referenced by get_Tensor().

5.3.2.3 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> 33 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 31 of file BendTransformation.cpp.

```
                                                    {
  return get_Space_Transformation_Tensor(position);
}
```

References get_Space_Transformation_Tensor().

5.3.2.4 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.

```
                                                            {
  Position ret;

  return ret;
}
```

5.3.2.5 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.
```
                                                             {
  Position ret;

  return ret;
}
```

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

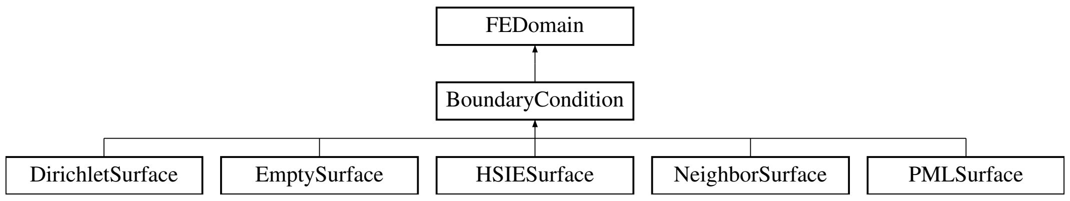
- Code/SpaceTransformations/BendTransformation.h
- Code/SpaceTransformations/BendTransformation.cpp

5.4 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< Dof↩ Number >

  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**

5.4.1 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.

5.4.2 Member Function Documentation

5.4.2.1 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.

```
                                                                          {
  double ret = 0;
  for(unsigned int i = 0; i < global_index_mapping.size(); i++) {
    ret += norm_squared(in_v->operator()(global_index_mapping[i]));
   }
   return std::sqrt(ret);
}
```

5.4.2.2 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.

```
                                                                                                    {
   double ret = 0;
   auto dofs = get_dof_association_by_boundary_id(in_bid);
   for(auto it : dofs) {
     ret += norm_squared(in_v->operator()(it.index));
   }
   return std::sqrt(ret);
}
```

References get_dof_association_by_boundary_id().

5.4.2.3 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 }
```

5.4.2.4 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.

5.4.2.5 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.

5.4.2.6 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.

```
                                                      {

}
```

5.4.2.7 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               }
177             }
178           }
179         }
180     }
181   }
182 }
```

5.4.2.8 get_boundary_ids()

```
std::vector< unsigned int > BoundaryCondition::get_boundary_ids ( ) -> std::vector<Boundary↩
Id>
```

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 }
```

5.4.2.9 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.

5.4.2.10 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().

5.4.2.11 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.

```
                                                                                                 {
  std::vector<InterfaceDofData> dof_data = get_dof_association_by_boundary_id(in_boundary_id);
  std::vector<DofNumber> ret;
  for(unsigned int i = 0; i < dof_data.size(); i++) {
    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().

5.4.2.12 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.

5.4.2.13 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.

5.4.2.14 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.

```
                                              {
  Constraints ret(global_dof_indices);
  return ret;
}
```

5.4.2.15 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.

```
                                        {
  return 0;
}
```

5.4.2.16 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.

5.4.2.17 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 }
```

5.4.2.18 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.

```
                                               {
    auto it = Geometry.surface_meshes[b_id].begin_active();
    std::vector<double> x;
    std::vector<double> y;
    while(it != Geometry.surface_meshes[b_id].end()){
      if(it->at_boundary()) {
        for (unsigned int face = 0; face < GeometryInfo<2>::faces_per_cell; ++face) {
          if (it->face(face)->at_boundary()) {
            dealii::Point<2, double> c;
            c = it->face(face)->center();
            x.push_back(c[0]);
            y.push_back(c[1]);
          }
        }
      }
      ++it;
    }
    double x_max = *max_element(x.begin(), x.end());
    double y_max = *max_element(y.begin(), y.end());
    double x_min = *min_element(x.begin(), x.end());
    double y_min = *min_element(y.begin(), y.end());
    it = Geometry.surface_meshes[b_id].begin_active();
    while(it != Geometry.surface_meshes[b_id].end()){
    if (it->at_boundary()) {
      for (unsigned int face = 0; face < dealii::GeometryInfo<2>::faces_per_cell;
          ++face) {
        Point<2, double> center;
        center = it->face(face)->center();
        if (std::abs(center[0] - x_min) < 0.0001) {
          it->face(face)->set_all_boundary_ids(
              edge_to_boundary_id[this->b_id][0]);
        }
        if (std::abs(center[0] - x_max) < 0.0001) {
          it->face(face)->set_all_boundary_ids(
              edge_to_boundary_id[this->b_id][1]);
        }
        if (std::abs(center[1] - y_min) < 0.0001) {
          it->face(face)->set_all_boundary_ids(
              edge_to_boundary_id[this->b_id][2]);
        }
        if (std::abs(center[1] - y_max) < 0.0001) {
          it->face(face)->set_all_boundary_ids(
              edge_to_boundary_id[this->b_id][3]);
        }
        }
    }
    ++it;
  }
}
```

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

- Code/BoundaryCondition/BoundaryCondition.h
- Code/BoundaryCondition/BoundaryCondition.cpp

5.5 BoundaryInformation Struct Reference

Public Member Functions

- **BoundaryInformation** (unsigned int in_coord, bool neg)

Public Attributes

- unsigned int **inner_coordinate**
- bool **negate_value**

5.5.1 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

5.6 CellAngelingData Struct Reference

Public Attributes

- EdgeAngelingData **edge_data**
- VertexAngelingData **vertex_data**

5.6.1 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

5.7 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**

5.7.1 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

5.8 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**

5.8.1 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

5.9 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**

5.9.1 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

5.10 ConstraintPair Struct Reference

Public Attributes

- unsigned int **left**
- unsigned int **right**
- bool **sign**

5.10.1 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

5.11 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** ()

5.11.1 Detailed Description

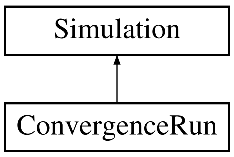
Definition at line 5 of file ConvergenceOutputGenerator.h.

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

- Code/OutputGenerators/Images/ConvergenceOutputGenerator.h
- Code/OutputGenerators/Images/ConvergenceOutputGenerator.cpp

5.12 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.

5.12.1 Detailed Description

Definition at line 9 of file ConvergenceRun.h.

5.12.2 Member Function Documentation

5.12.2.1 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.

```
                                   {
  double local = 0.0;
  for(unsigned int i = 0; i < a.size(); i++) {
    double x = std::abs(a[i][0] - b[i][0]);
    double y = std::abs(a[i][1] - b[i][1]);
    double z = std::abs(a[i][2] - b[i][2]);
    local += std::sqrt(x*x + y*y + z*z);
  }
  local /= evaluation_positions.size();
  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);
  double ret = dealii::Utilities::MPI::sum(local, MPI_COMM_WORLD);
  ret /= norming_factor;
  return ret;
}
```

5.12.2.2 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.

```
                                 {
  print_info("ConvergenceRun::prepare", "Start", LoggingLevel::DEBUG_ONE);
  GlobalParams.Cells_in_x = GlobalParams.convergence_max_cells;
  GlobalParams.Cells_in_y = GlobalParams.convergence_max_cells;
  GlobalParams.Cells_in_z = GlobalParams.convergence_max_cells;
  Geometry.initialize();
  mainProblem = new NonLocalProblem(GlobalParams.Sweeping_Level);
  mainProblem->initialize();
  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);
}
```

5.12.2.3 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.

```
                             {
    print_info("ConvergenceRun::run", "Start", LoggingLevel::PRODUCTION_ONE);
    for(unsigned int run_index = 0; run_index < GlobalParams.convergence_cell_counts.size()-1;
    run_index++) {
      GlobalParams.Cells_in_x = GlobalParams.convergence_cell_counts[run_index];
      GlobalParams.Cells_in_y = GlobalParams.convergence_cell_counts[run_index];
      GlobalParams.Cells_in_z = GlobalParams.convergence_cell_counts[run_index];
      Geometry.initialize();
      otherProblem = new NonLocalProblem(GlobalParams.Sweeping_Level);
      otherProblem->initialize();
      otherProblem->assemble();
      otherProblem->compute_solver_factorization();
      otherProblem->solve_with_timers_and_count();
      std::vector<std::vector<ComplexNumber» other_evaluations =
    otherProblem->evaluate_solution_at(evaluation_positions);
      double numerical_error = compute_error_for_two_eval_vectors(evaluation_base_problem,
    other_evaluations);
      double theoretical_error = compute_error_for_two_eval_vectors(evaluation_exact_solution,
    other_evaluations);
      numerical_errors.push_back(numerical_error);
      theoretical_errors.push_back(theoretical_error);
      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);
      print_info("ConvergenceRun::run", msg , LoggingLevel::PRODUCTION_ONE);
      unsigned int temp_ndofs = otherProblem->compute_total_number_of_dofs();
      n_dofs_for_cases.push_back(temp_ndofs);
      h_values.push_back(otherProblem->compute_h());
      total_cells.push_back(otherProblem->n_total_cells());
      output.push_values(temp_ndofs,numerical_error,theoretical_error);
      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

5.13 CoreLogger Class Reference

Outputs I want:

```
#include <CoreLogger.h>
```

5.13.1 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

5.14 DataSeries Struct Reference

Public Attributes

- std::vector< double > **values**
- bool **is_closed**
- std::string **name**

5.14.1 Detailed Description

Definition at line 222 of file Types.h.

The documentation for this struct was generated from the following file:

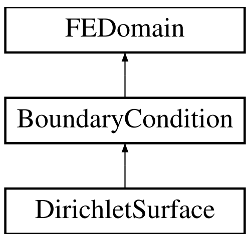
- Code/Core/Types.h

5.15 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

5.15.1 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.

5.15.2 Member Function Documentation

5.15.2.1 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.

```
                                                                 {
    return 0;
}
```

5.15.2.2 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.

```
                                                                {
    return 0;
}
```

5.15.2.3 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.

```
                                                         {

}
```

5.15.2.4 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.

```
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 }
```

5.15.2.5 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.

```
{ }
```

5.15.2.6 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.

```
                                                                         {
    std::vector<InterfaceDofData> ret;
    return ret;
}
```

5.15.2.7 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.

```
{
    std::vector<InterfaceDofData> ret;
    return ret;
}
```

5.15.2.8 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.

```
{

}
```

5.15.2.9 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.

```
{
    return false;
}
```

5.15.2.10 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.

```
                                               {
    Constraints ret(Geometry.levels[level].inner_domain->global_dof_indices);
    dealii::IndexSet local_dof_set(Geometry.levels[level].inner_domain->n_locally_active_dofs);
    local_dof_set.add_range(0,Geometry.levels[level].inner_domain->n_locally_active_dofs);
    AffineConstraints<ComplexNumber> constraints_local(local_dof_set);

    VectorTools::project_boundary_values_curl_conforming_l2(Geometry.levels[level].inner_domain->dof_handler,
    0, *GlobalParams.source_field, 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].inner_domain->global_index_mapping[local_index];
        ret.add_line(global_index);
        ret.set_inhomogeneity(global_index, line.inhomogeneity);
    }
    constraints_local.clear();
    if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
        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) {
                    PMLTransformedExactSolution ptes(b_id, additional_coordinate);

    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;
 }
```

5.15.2.11 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.

```
                                                                                                          {
    return "";
}
```

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

- Code/BoundaryCondition/DirichletSurface.h
- Code/BoundaryCondition/DirichletSurface.cpp

5.16 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**

5.16.1 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

5.17 DofCountsStruct Struct Reference

Public Attributes

- unsigned int **hsie** = 0
- unsigned int **non_hsie** = 0
- unsigned int **total** = 0

5.17.1 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

5.18 DofCouplingInformation Struct Reference

Public Attributes

- DofNumber **first_dof**
- DofNumber **second_dof**
- double **coupling_value**

5.18.1 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

5.19 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

5.19.1 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

5.20 DofIndexData Class Reference

Public Member Functions

- void **communicateSurfaceDofs** ()
- void **initialize** ()
- void **initialize_level** (unsigned int level)

Public Attributes

- bool $*$ **isSurfaceNeighbor**
- std::vector$<$ LevelDofIndexData $>$ **indexCountsByLevel**

5.20.1 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

5.21 DofOwner Struct Reference

Public Attributes

- unsigned int **owner** = 0
- bool **is_boundary_dof** = false
- unsigned int **surface_id** = 0

5.21.1 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

5.22 EdgeAngelingData Struct Reference

Public Attributes

- unsigned int **edge_index**
- bool **angled_in_x** = false
- bool **angled_in_y** = false

5.22.1 Detailed Description

Definition at line 74 of file Types.h.

The documentation for this struct was generated from the following file:

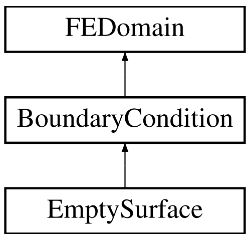
- Code/Core/Types.h

5.23 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

5.23.1 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.

5.23.2 Member Function Documentation

5.23.2.1 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 }
```

5.23.2.2 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 }
```

5.23.2.3 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 }
```

5.23.2.4 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.

```
                 {
    matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
   required.
   // Nothing to do here, work happens on neighbor process.
}
```

5.23.2.5 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.

```
{ }
```

5.23.2.6 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.

```
                                                                      {
    std::vector<InterfaceDofData> ret;
    return ret;
}
```

5.23.2.7 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.

```
                                                                                              {
    std::vector<InterfaceDofData> ret;
    return ret;
}
```

5.23.2.8 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.

```
                                  {

}
```

5.23.2.9 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.

```
                                                                  {
    return false;
}
```

5.23.2.10 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.

```
                                              {
    Constraints ret(Geometry.levels[level].inner_domain->global_dof_indices);
    dealii::IndexSet local_dof_set(Geometry.levels[level].inner_domain->n_locally_active_dofs);
    local_dof_set.add_range(0,Geometry.levels[level].inner_domain->n_locally_active_dofs);
    AffineConstraints<ComplexNumber> constraints_local(local_dof_set);
    std::vector<InterfaceDofData> dofs =
    Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
    for(auto line : dofs) {
        const unsigned int local_index = line.index;
        const unsigned int global_index =
    Geometry.levels[level].inner_domain->global_index_mapping[local_index];
        ret.add_line(global_index);
        ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
    }
    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;
}
```

5.23.2.11 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.

```
                                                                        {
    return "";
}
```

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

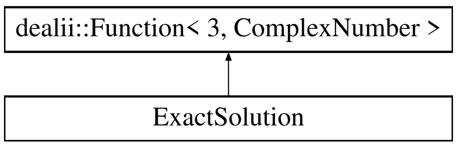
- Code/BoundaryCondition/EmptySurface.h
- Code/BoundaryCondition/EmptySurface.cpp

5.24 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**

5.24.1 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.

Author

Pascal Kraft

Date

23.11.2015

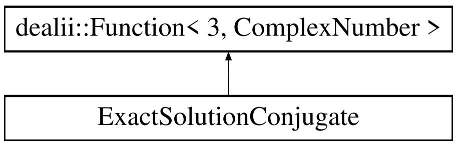
Definition at line 35 of file ExactSolution.h.

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

- Code/Solutions/ExactSolution.h
- Code/Solutions/ExactSolution.cpp

5.25 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

5.25.1 Detailed Description

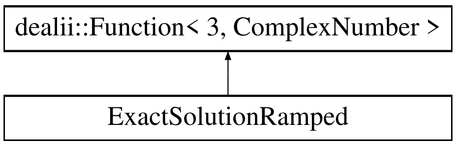
Definition at line 12 of file ExactSolutionConjugate.h.

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

- Code/Solutions/ExactSolutionConjugate.h
- Code/Solutions/ExactSolutionConjugate.cpp

5.26 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

5.26.1 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

5.27 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**

5.27.1 Detailed Description

Definition at line 233 of file Types.h.

The documentation for this struct was generated from the following file:

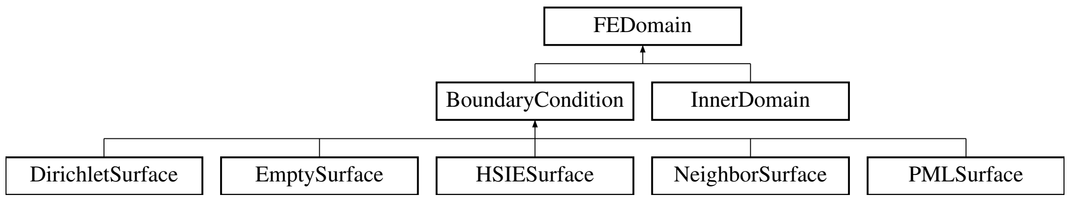
- Code/Core/Types.h

5.28 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**

5.28.1 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.

5.28.2 Member Function Documentation

5.28.2.1 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.

5.28.2.2 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.

5.28.2.3 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.

5.28.2.4 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.

```
                                                                  {
    if(!is_ownership_ready) {
        std::cout « "You called finish_initialization before freeze_ownership which is not valid." «
    std::endl;
        return false;
    }
    DofNumber running_index = first_own_index;
    for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
        if(is_dof_owned[i]) {
            global_index_mapping[i] = running_index;
            running_index++;
        }
    }
    return true;
}
```

5.28.2.5 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.

```
{
    NumericVectorLocal ret(n_locally_active_dofs);
    for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
        ret[i] = in_vector[global_index_mapping[i]];
    }
    return ret;
}
```

5.28.2.6 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.

```
{
    n_locally_owned_dofs = in_n_locally_owned_dofs;
    n_locally_active_dofs = in_n_locally_active_dofs;
    global_index_mapping.resize(n_locally_active_dofs);
    for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
        global_index_mapping[i] = UINT_MAX;
        is_dof_owned.push_back(true);
    }

}
```

5.28.2.7 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.

```
                                                                         {
    double norm = 0;
    for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
        if(is_dof_owned[i]) {
            norm += norm_squared(in_v->operator()(global_index_mapping[i]));
        }
    }
    return std::sqrt(norm);
}
```

5.28.2.8 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.

```
                                                                      {
    for(unsigned int i = 0; i < in_dofs.size(); i++) {
        is_dof_owned[in_dofs[i]] = false;
    }
}
```

Referenced by PMLSurface::determine_non_owned_dofs(), and HSIESurface::determine_non_owned_dofs().

5.28.2.9 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.

```
                                                                                    {
    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];
    }
}
```

5.28.2.10 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.

```
                                                                         {
    std::vector<DofNumber> global_dof_indices;
    for(unsigned int i = 0; i < in_dofs.size(); i++) {
        global_dof_indices.push_back(global_index_mapping[in_dofs[i]]);
    }
    return global_dof_indices;
}
```

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

5.29 FEErrorStruct Struct Reference

Public Attributes

- double **L2** = 0
- double **Linfty** = 0

5.29.1 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

5.30 FileLogger Class Reference

There will be one global instance of this object.

```
#include <FileLogger.h>
```

5.30.1 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

5.31 FileMetaData Struct Reference

Public Attributes

- unsigned int **hsie_level**

5.31.1 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

5.32 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**

5.32.1 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.

5.32.2 Member Function Documentation

5.32.2.1 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.

```
                                                             {
  if (GlobalParams.Blocks_in_x_direction == 1) {
    return std::pair<double, double>(-GlobalParams.Geometry_Size_X / 2.0, GlobalParams.Geometry_Size_X /
   2.0);
  } else {
    double length = GlobalParams.Geometry_Size_X / ((double) GlobalParams.Blocks_in_x_direction);
    int block_index = GlobalParams.MPI_Rank % GlobalParams.Blocks_in_x_direction;
    double min = -GlobalParams.Geometry_Size_X / 2.0 + block_index * length;
    return std::pair<double, double>(min, min + length);
  }
}
```

5.32.2.2 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.

```
                                                             {
  if (GlobalParams.Blocks_in_y_direction == 1) {
    return std::pair<double, double>(-GlobalParams.Geometry_Size_Y / 2.0, GlobalParams.Geometry_Size_Y /
   2.0);
  } else {
    double length = GlobalParams.Geometry_Size_Y / ((double) GlobalParams.Blocks_in_y_direction);
    int block_processor_count = GlobalParams.Blocks_in_x_direction;
    int block_index = (GlobalParams.MPI_Rank % (GlobalParams.Blocks_in_x_direction *
   GlobalParams.Blocks_in_y_direction)) / block_processor_count;
    double min = -GlobalParams.Geometry_Size_Y / 2.0 + block_index * length;
    return std::pair<double, double>(min, min + length);
  }
}
```

5.32.2.3 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.

```
                                                             {
  if (GlobalParams.Blocks_in_z_direction == 1) {
    return std::pair<double, double>(0 + GlobalParams.global_z_shift, GlobalParams.Geometry_Size_Z +
   GlobalParams.global_z_shift);
  } else {
    double length = GlobalParams.Geometry_Size_Z / ((double) GlobalParams.Blocks_in_z_direction);
    int block_processor_count = GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction;
    int block_index = GlobalParams.MPI_Rank / block_processor_count;
    double min = block_index * length;
    return std::pair<double, double>(min + GlobalParams.global_z_shift, min +
   GlobalParams.global_z_shift + length);
  }
}
```

5.32.2.4 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.

```
                                                {
  return (math_coordinate_in_waveguide(in_p)? GlobalParams.Epsilon_R_in_waveguide :
   GlobalParams.Epsilon_R_outside_waveguide) * GlobalParams.Omega * GlobalParams.Omega;
}
```

References math_coordinate_in_waveguide().

5.32.2.5 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.

```
                                                                    {
  if(math_coordinate_in_waveguide(in_p)) {
    return GlobalParams.Epsilon_R_in_waveguide;
  } else {
    return GlobalParams.Epsilon_R_outside_waveguide;
  }
}
```

References math_coordinate_in_waveguide().

Referenced by get_epsilon_tensor().

5.32.2.6 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.

```
                                                                          {
  dealii::Tensor<2,3> ret;
  const double local_epsilon = get_epsilon_for_point(in_p);
  for(unsigned int i = 0; i < 3; i++) {
    for(unsigned int j = 0; j < 3; j++) {
      if(i == j) {
        ret[i][j] = local_epsilon;
      } else {
        ret[i][j] = 0;
      }
    }
  }
  return ret;
}
```

References get_epsilon_for_point().

5.32.2.7 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.

```
                                                                                                      {
  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_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().

5.32.2.8 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.

```
                       {
  std::pair<bool, unsigned int> ret(true, 0);
  if(level == 0) {
    return get_global_neighbor_for_interface(in_direction);
  }
  if(level == 1) {
    switch (in_direction) {
      case Direction::MinusX:
        if (GlobalParams.Index_in_x_direction == 0) {
          ret.first = false;
        } else {
          ret.second = (GlobalParams.MPI_Rank - 1) % (GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction);
        }
        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) % (GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction);
        }
        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) % (GlobalParams.Blocks_in_x_direction * 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) % (GlobalParams.Blocks_in_x_direction * GlobalParams.Blocks_in_y_direction);
        }
        break;
      case Direction::MinusZ:
        ret.first = false;
        break;
      case Direction::PlusZ:
        ret.first = false;
        break;
    }
  }
  if(level == 2) {
    switch (in_direction) {
      case Direction::MinusX:
        if (GlobalParams.Index_in_x_direction == 0) {
          ret.first = false;
        } else {
          ret.second = (GlobalParams.MPI_Rank - 1) % GlobalParams.Blocks_in_x_direction;
        }
        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) % GlobalParams.Blocks_in_x_direction;
        }
        break;
      case Direction::MinusY:
        ret.first = false;
        break;
      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().

5.32.2.9 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 }
```

5.32.2.10 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 }
```

5.32.2.11 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.

```
                                                                           {
  bool in_x = std::abs(in_position[0]) <= (GlobalParams.Width_of_waveguide  / 2.0);
  bool in_y = std::abs(in_position[1]) <= (GlobalParams.Height_of_waveguide / 2.0);
  return in_x && in_y;
}
```

Referenced by eps_kappa_2(), and get_epsilon_for_point().

5.32.2.12 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.

```
                                                                 {
  this->local_x_range = in_range;
  global_x_range = std::pair<double, double>(-GlobalParams.Geometry_Size_X / 2.0,
   GlobalParams.Geometry_Size_X / 2.0);
}
```

5.32.2.13 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.

```
                                                                  {
  this->local_y_range = in_range;
  global_y_range = std::pair<double, double> (-GlobalParams.Geometry_Size_Y / 2.0,
   GlobalParams.Geometry_Size_Y / 2.0);
}
```

5.32.2.14 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.

```
                                                                  {
  this->local_z_range = in_range;
  global_z_range = std::pair<double, double>(0.0 + GlobalParams.global_z_shift ,
   GlobalParams.Geometry_Size_Z + GlobalParams.global_z_shift);
}
```

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

- Code/GlobalObjects/GeometryManager.h
- Code/GlobalObjects/GeometryManager.cpp

5.33 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**

5.33.1 Detailed Description

The Gradient Table is an OutputGenerator, intended to write information about the shape gradient to the console upon its computation.

Date

28.11.2016

Author

Pascal Kraft

Definition at line 12 of file GradientTable.h.

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

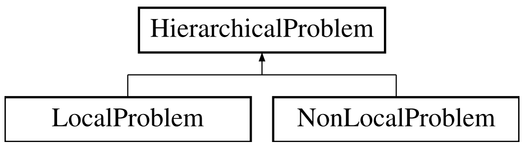
- Code/OutputGenerators/Console/GradientTable.h
- Code/OutputGenerators/Console/GradientTable.cpp

5.34 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

5.34.1 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.

5.34.2 Constructor & Destructor Documentation

5.34.2.1 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 }
```

5.34.3 Member Function Documentation

5.34.3.1 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 $Ax - b$.

5.34.3.2 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().

5.34.3.3 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    }
```

5.34.3.4 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.
set_two	Second index set.
affine_constraints	Affine Constraint object to write the constraints into.

Definition at line 29 of file HierarchicalProblem.cpp.

```
                                        {
  const unsigned int n_entries = set_one->size();
  if (n_entries != set_two->size()) {
    print_info("HierarchicalProblem::constrain_identical_dof_sets", "There was an error in
    constrain_identical_dof_sets. No changes made.", LoggingLevel::PRODUCTION_ALL);
  }

  for (unsigned int index = 0; index < n_entries; index++) {
    affine_constraints->add_line(set_one->operator [](index));
    affine_constraints->add_entry(set_one->operator [](index),
        set_two->operator [](index), ComplexNumber(-1, 0));
  }
}
```

5.34.3.5 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.

```
                                              {
  print_info("HierarchicalProblem::make_constraints", "Start");
  IndexSet total_dofs_global(Geometry.levels[level].n_total_level_dofs);
  total_dofs_global.add_range(0,Geometry.levels[level].n_total_level_dofs);
  constraints.reinit(total_dofs_global);

  // ABC Surfaces are least important
  for(unsigned int surface = 0; surface < 6; surface++) {
    if(Geometry.levels[level].surface_type[surface] == SurfaceType::ABC_SURFACE) {
      Constraints local_constraints = Geometry.levels[level].surfaces[surface]->make_constraints();
      constraints.merge(local_constraints, Constraints::MergeConflictBehavior::right_object_wins,true);
    }
  }

  // Dirichlet surfaces are more important than ABC
  for(unsigned int surface = 0; surface < 6; surface++) {
    if(Geometry.levels[level].surface_type[surface] == SurfaceType::DIRICHLET_SURFACE) {
      Constraints local_constraints = Geometry.levels[level].surfaces[surface]->make_constraints();
      constraints.merge(local_constraints, Constraints::MergeConflictBehavior::right_object_wins,true);
    }
  }

  // Open surfaces are most important
  for(unsigned int surface = 0; surface < 6; surface++) {
    if(Geometry.levels[level].surface_type[surface] == SurfaceType::OPEN_SURFACE) {
      Constraints local_constraints = Geometry.levels[level].surfaces[surface]->make_constraints();
      constraints.merge(local_constraints, Constraints::MergeConflictBehavior::right_object_wins,true);
    }
  }
  constraints.close();

  print_info("HierarchicalProblem::make_constraints", "End");
}
```

5.34.3.6 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.

```
                                                                    {
  if((in_bid % 2) == 0) {
    return in_bid + 1;
  }
  else {
    return in_bid - 1;
  }
}
```

5.34.3.7 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);
101
102   if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
103     for(unsigned int i = 0; i < 6; i++){
104       if(Geometry.levels[level].surface_type[i] == SurfaceType::ABC_SURFACE){
105         dealii::Vector<ComplexNumber> ds (Geometry.levels[level].surfaces[i]->dof_counter);
106         for(unsigned int index = 0; index < Geometry.levels[level].surfaces[i]->dof_counter; index++) {
107           ds[index] = solution[Geometry.levels[level].surfaces[i]->global_index_mapping[index]];
108         }
109         std::string file_2 = Geometry.levels[level].surfaces[i]->output_results(ds, "pml_domain" +
     std::to_string(level));
110         filenames.push_back(file_2);
111       }
112     }
113   }
114
115   // End of core output
116   if(level != 0) {
117   //  child->output_results();
118   }
119
120   print_info("Hierarchical::output_results()", "End on level " + std::to_string(level));
121   timer.stop();
122   GlobalTimerManager.leave_context(level);
123   return ret;
124 }
```

5.34.3.8 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().

5.34.3.9 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

5.35 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**

5.35.1 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_plus and T_minus). As an important remark: The value kappa_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.

5.35.2 Member Function Documentation

5.35.2.1 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.

```
                                                                               {
  HSIEPolynomial::D.reinit(dimension, dimension);
  for (unsigned int i = 0; i < dimension; i++) {
    for (unsigned int j = 0; j < dimension; j++) {
      HSIEPolynomial::D.set(i, j, matrixD(i, j, k0));
    }
  }

  HSIEPolynomial::I.copy_from(HSIEPolynomial::D);
  HSIEPolynomial::I.invert(HSIEPolynomial::D);
  HSIEPolynomial::matricesLoaded = true;
}
```

Referenced by HSIESurface::check_dof_assignment_integrity(), and HSIESurface::fill_matrix().

5.35.2.2 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.

```
                                                                  {
  ComplexNumber ret(a[0]);
  ComplexNumber x = x_in;
  for (unsigned long i = 1; i < a.size(); i++) {
    ret += a[i] * x;
    x = x * x_in;
  }
  return ret;
}
```

5.35.2.3 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.

```
                                                          {
  ComplexNumber ret(da[0]);
  ComplexNumber x = x_in;
  for (unsigned long i = 1; i < da.size(); i++) {
    ret += da[i] * x;
    x = x * x_in;
  }
  return ret;
}
```

5.35.2.4 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.

```
                                         {
  da = std::vector<ComplexNumber>();
  for (unsigned int i = 1; i < a.size(); i++) {
    da.emplace_back(i * a[i].real(), i * a[i].imag());
  }
}
```

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

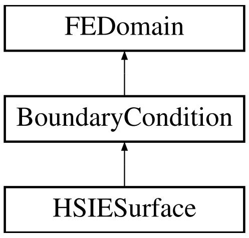
- Code/BoundaryCondition/HSIEPolynomial.h
- Code/BoundaryCondition/HSIEPolynomial.cpp

5.36 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, Dof↩ DataVector &, 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, Dof↩ DataVector &, 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**

5.36.1 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 \cref{subsec:HSIE,sub:hsieElements,sec:HSIESweeping}.

Definition at line 48 of file HSIESurface.h.

5.36.2 Constructor & Destructor Documentation

5.36.2.1 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;
}
```

5.36.3 Member Function Documentation

5.36.3.1 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 }
```

5.36.3.2 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.

```
                                                  {
  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().

5.36.3.3 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.

```
                         {
  std::vector<HSIEPolynomial> ret;
  ret.push_back(HSIEPolynomial::ZeroPolynomial());
  HSIEPolynomial temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
  temp.multiplyBy(fe_gradient[1]);
  ret.push_back(temp);
  temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
  temp.multiplyBy(-1.0 * fe_gradient[0]);
  ret.push_back(temp);
  transform_coordinates_in_place(&ret);
  return ret;
}
```

References transform_coordinates_in_place().

5.36.3.4 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.

```
                                                                 {
  FaceAngelingData ret;
  for(unsigned int i = 0; i < ret.size(); i++) {
    ret[i].is_x_angled = false;
    ret[i].is_y_angled = false;
    ret[i].position_of_base_point = {};
  }
  return ret;
}
```

Referenced by fill_matrix().

5.36.3.5 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.

```
                                                                          {
  std::vector<HSIEPolynomial> ret;
  HSIEPolynomial temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
  temp.multiplyBy(fe_shape_value);
  temp = temp.applyD();
  ret.push_back(temp);
  ret.push_back(HSIEPolynomial::ZeroPolynomial());
  ret.push_back(HSIEPolynomial::ZeroPolynomial());
  transform_coordinates_in_place(&ret);
  return ret;
}
```

References transform_coordinates_in_place().

5.36.3.6 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.

```
                                                   {
  HSIEPolynomial::computeDandI(order + 2, k0);
  auto it = dof_h_nedelec.begin_active();
  auto end = dof_h_nedelec.end();
  auto it2 = dof_h_q.begin_active();
  unsigned int counter = 1;
  for (; it != end; ++it) {
    if (it->id() != it2->id()) std::cout « "Identity failure!" « std::endl;
    DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
    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);
    std::vector<unsigned int> local_related_fe_index;
    bool found = false;
    for (unsigned int i = 0; i < cell_dofs.size(); i++) {
      found = false;
      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);
            found = true;
          }
        }
      } 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);
            found = true;
          }
        }
      }
      if (!found) {
        std::cout « "Error in dof assignment integrity!" « std::endl;
      }
    }

    if (local_related_fe_index.size() != cell_dofs.size()) {
      std::cout « "Mismatch in cell " « counter
```

```
                « ": Found indices: " « local_related_fe_index.size()
                « " of a total " « cell_dofs.size() « std::endl;
      return false;
    }
    counter++;
    it2++;
  }

  return true;
}
```

References HSIEPolynomial::computeDandI().

5.36.3.7 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().

5.36.3.8 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++;
  }
}
```

5.36.3.9 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.

```
                                                                          {
  unsigned int ret = 0;
  const unsigned int INNER_REAL_DOFS_PER_LINE = fe_nedelec.dofs_per_line;

  if (!only_hsie_dofs) {
    ret += INNER_REAL_DOFS_PER_LINE;
  }

  ret += INNER_REAL_DOFS_PER_LINE * (order + 1)
      + (INNER_REAL_DOFS_PER_LINE - 1) * (order + 2);

  return ret;
}
```

Referenced by check_number_of_dofs_for_cell_integrity(), fill_matrix(), and update_dof_counts_for_edge().

5.36.3.10 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.

```
                                                                    {
  unsigned int ret = 0;
  const unsigned int INNER_REAL_NEDELEC_DOFS_PER_FACE =
      fe_nedelec.dofs_per_cell -
      dealii::GeometryInfo<2>::faces_per_cell * fe_nedelec.dofs_per_face;

  ret = INNER_REAL_NEDELEC_DOFS_PER_FACE * (order + 2) * 3;
  if (only_hsie_dofs) {
    ret -= INNER_REAL_NEDELEC_DOFS_PER_FACE;
  }
  return ret;
}
```

Referenced by check_number_of_dofs_for_cell_integrity(), fill_matrix(), and update_dof_counts_for_face().

5.36.3.11 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.

```
                                                   {
  unsigned int ret = order + 2;

  return ret;
}
```

Referenced by check_number_of_dofs_for_cell_integrity(), fill_matrix(), and update_dof_counts_for_vertex().

5.36.3.12 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.

```
                                              {
  DoFHandler<2>::active_cell_iterator cell;
  DoFHandler<2>::active_cell_iterator cell2;
  DoFHandler<2>::active_cell_iterator endc;
  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().

5.36.3.13 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.

```
                                              {
  std::set<std::string> touched_faces;
  DoFHandler<2>::active_cell_iterator cell;
  DoFHandler<2>::active_cell_iterator cell2;
  DoFHandler<2>::active_cell_iterator endc;
  endc = dof_h_nedelec.end();
  DofCountsStruct ret;
  cell2 = dof_h_q.begin_active();
  for (cell = dof_h_nedelec.begin_active(); cell != endc; cell++) {
    if (touched_faces.end() == touched_faces.find(cell->id().to_string())) {
      update_dof_counts_for_face(cell, ret);
      register_new_surface_dofs(cell, cell2);
      touched_faces.insert(cell->id().to_string());
    }
    cell2++;
  }
  return ret;
}
```

References register_new_surface_dofs(), and update_dof_counts_for_face().

Referenced by initialize().

5.36.3.14 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.

```
                                                        {
  return dof_counter;
}
```

5.36.3.15 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.

```
                                                       {
  IndexSet non_owned_dofs = compute_non_owned_dofs();
  return dof_counter - non_owned_dofs.n_elements();
}
```

References compute_non_owned_dofs().

5.36.3.16 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.

```
                                                    {
  std::set<unsigned int> touched_vertices;
  DoFHandler<2>::active_cell_iterator cell;
  DoFHandler<2>::active_cell_iterator endc;
  endc = dof_h_q.end();
  DofCountsStruct ret;
  for (cell = dof_h_q.begin_active(); cell != endc; cell++) {
    // for each edge
    for (unsigned int vertex = 0; vertex < GeometryInfo<2>::vertices_per_cell;
         vertex++) {
      unsigned int idx = cell->vertex_dof_index(vertex, 0);
      if (touched_vertices.end() == touched_vertices.find(idx)) {
        // handle it
        update_dof_counts_for_vertex(cell, idx, vertex, ret);
        register_new_vertex_dofs(cell, idx, vertex);
        // remember that it has been handled
        touched_vertices.insert(idx);
      }
    }
  }
  return ret;
}
```

References register_new_vertex_dofs(), and update_dof_counts_for_vertex().

Referenced by initialize().

5.36.3.17 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.

```
                                                    {
  IndexSet non_owned_dofs(dof_counter);
  for(auto it : surface_dofs) {
    non_owned_dofs.add_index(it.index);
  }
  for(auto surf : adjacent_boundaries) {
    if(Geometry.levels[level].surface_type[surf] == SurfaceType::NEIGHBOR_SURFACE) {
      if(surf % 2 == 0) {
        std::vector<InterfaceDofData> dofs_data = get_dof_association_by_boundary_id(surf);
        for(auto it : dofs_data) {
          non_owned_dofs.add_index(it.index);
        }
      }
    }
  }
  return non_owned_dofs;
}
```

Referenced by compute_n_locally_owned_dofs(), and determine_non_owned_dofs().

5.36.3.18 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.

```
                                             {
  IndexSet non_owned_dofs = compute_non_owned_dofs();
  const unsigned int n_dofs = non_owned_dofs.n_elements();
  std::vector<unsigned int> local_dofs(n_dofs);
  for(unsigned int i = 0; i < n_dofs; i++) {
     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().

5.36.3.19 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.

```

                        {
  ComplexNumber result(0, 0);
  for(unsigned int i = 0; i < 3; i++) {
    for (unsigned int j = 0; j < 3; j++) {
      for (unsigned int k = 0; k < std::min(u[i].a.size(), v[j].a.size()); k++) {
        result += G[i][j] * u[i].a[k] * v[j].a[k];
      }
    }
  }
  return result;
}
```

5.36.3.20 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().

5.36.3.21 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.

5.36.3.22 fill_sparsity_pattern()

```
void HSIESurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * in_constriants )  [override], [virtual]
```

Fills a sparsity pattern for all the dofs active in this boundary condition.

Parameters

in_dsp	The sparsit pattern to fill
in_constriants	The constraint object to be used to condense

Implements BoundaryCondition.

Definition at line 251 of file HSIESurface.cpp.

```
    {
  auto it = dof_h_nedelec.begin();
  auto end = dof_h_nedelec.end();
  auto it2 = dof_h_q.begin();
  for (; it != end; ++it) {
    DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
    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);
    }
    local_indices = transform_local_to_global_dofs(local_indices);
    in_constraints->add_entries_local_to_global(local_indices, *in_dsp);
    it2++;
  }
}
```

References FEDomain::transform_local_to_global_dofs().

5.36.3.23 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.

```
                                                    {
  for(BoundaryId surf:adjacent_boundaries) {
    if(!are_edge_dofs_owned[surf] && Geometry.levels[level].surface_type[surf] !=
   SurfaceType::NEIGHBOR_SURFACE) {
      DofIndexVector dofs_in_global_numbering =
   Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id);
      std::vector<InterfaceDofData> local_interface_data = get_dof_association_by_boundary_id(surf);
      DofIndexVector dofs_in_local_numbering(local_interface_data.size());
      for(unsigned int i = 0; i < local_interface_data.size(); i++) {
        dofs_in_local_numbering[i] = local_interface_data[i].index;
      }
      set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
    }
  }

  // Do the same for the inner interface
  std::vector<InterfaceDofData> global_interface_data =
   Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
  std::vector<InterfaceDofData> local_interface_data = get_dof_association();
  DofIndexVector dofs_in_local_numbering(local_interface_data.size());
  DofIndexVector dofs_in_global_numbering(local_interface_data.size());

  for(unsigned int i = 0; i < local_interface_data.size(); i++) {
    dofs_in_local_numbering[i] = local_interface_data[i].index;
    dofs_in_global_numbering[i] =
   Geometry.levels[level].inner_domain->global_index_mapping[global_interface_data[i].index];
  }
  set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);

}
```

5.36.3.24 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.

```
                                                    {
  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);
}
```

5.36.3.25 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.

```
                                                             {
  std::sort(surface_dofs.begin(), surface_dofs.end(), compareDofBaseDataAndOrientation);
  std::vector<InterfaceDofData> ret;
  copy(surface_dofs.begin(), surface_dofs.end(), back_inserter(ret));
  return ret;
}
```

5.36.3.26 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.

```
                                                                                                       {
  if (are_opposing_sites(b_id, in_boundary_id)) {
    return get_dof_association();
  }

  if (in_boundary_id == b_id) {
    std::vector<InterfaceDofData> surface_dofs_unsorted(0);
    std::cout « "This should never be called in HSIESurface" « std::endl;
    return surface_dofs_unsorted;
  }
  std::vector<InterfaceDofData> surface_dofs_unsorted;
  std::vector<unsigned int> vertex_ids = get_vertices_for_boundary_id(in_boundary_id);
  std::vector<unsigned int> line_ids = get_lines_for_boundary_id(in_boundary_id);
  std::vector<Position> vertex_positions = vertex_positions_for_ids(vertex_ids);
  std::vector<Position> line_positions = line_positions_for_ids(line_ids);
  for(unsigned int index = 0; index < vertex_dof_data.size(); index++) {
    DofData dof = vertex_dof_data[index];
    for(unsigned int index_in_ids = 0; index_in_ids < vertex_ids.size(); index_in_ids++) {
      if(vertex_ids[index_in_ids] == vertex_dof_data[index].base_structure_id_non_face) {
        InterfaceDofData new_item;
        new_item.index = dof.global_index;
        new_item.base_point = vertex_positions[index_in_ids];
        new_item.order = (dof.inner_order+1) * (dof.nodal_basis + 1);
        surface_dofs_unsorted.push_back(new_item);
      }
    }
  }

  // Construct containers with base points, orientation and index
  for(unsigned int index = 0; index < edge_dof_data.size(); index++) {
    DofData dof = edge_dof_data[index];
    for(unsigned int index_in_ids = 0; index_in_ids < line_ids.size(); index_in_ids++) {
      if(line_ids[index_in_ids] == edge_dof_data[index].base_structure_id_non_face) {
        InterfaceDofData new_item;
        new_item.index = dof.global_index;
        new_item.base_point = line_positions[index_in_ids];
        new_item.order = (dof.inner_order+1) * (dof.nodal_basis + 1);
        surface_dofs_unsorted.push_back(new_item);
      }
    }
  }

  // Sort the vectors.
  std::sort(surface_dofs_unsorted.begin(), surface_dofs_unsorted.end(),
   compareDofBaseDataAndOrientation);

  return surface_dofs_unsorted;
}
```

5.36.3.27 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.

```
                                                                                    {
  DofDataVector ret;
  for (unsigned int index = 0; index < edge_dof_data.size(); index++) {
    if ((edge_dof_data[index].base_dof_index == in_index)
        && (edge_dof_data[index].type != DofType::RAY
            && edge_dof_data[index].type != DofType::IFFb)) {
      ret.push_back(edge_dof_data[index]);
    }
  }
  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;
}
```

5.36.3.28 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.

```
                                                                          {
  DofDataVector ret;
  for (unsigned int index = 0; index < edge_dof_data.size(); index++) {
    if ((edge_dof_data[index].base_dof_index == in_index)
        && (edge_dof_data[index].type == DofType::RAY
            || edge_dof_data[index].type == DofType::IFFb)) {
      ret.push_back(edge_dof_data[index]);
    }
  }
  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;
}
```

5.36.3.29 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.

```
                                                                                        {
  std::vector<unsigned int> edges;
  for(auto it = Geometry.surface_meshes[b_id].begin_active_face(); it !=
   Geometry.surface_meshes[b_id].end_face(); it++) {
    if(is_point_at_boundary(it->center(), in_boundary_id)) {
      edges.push_back(it->index());
    }
  }
  edges.shrink_to_fit();
  return edges;
}
```

5.36.3.30 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

5.36.3.31 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

5.36.3.32 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.

```
                                                                                 {
  std::vector<unsigned int> vertices;
  for(auto it = Geometry.surface_meshes[b_id].begin_vertex(); it !=
   Geometry.surface_meshes[b_id].end_vertex(); it++) {
    if(is_point_at_boundary(it->center(), in_boundary_id)) {
      vertices.push_back(it->index());
    }
  }
  vertices.shrink_to_fit();
  return vertices;
}
```

5.36.3.33 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().

5.36.3.34 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.

```
                                                                       {
  if(!boundary_coordinates_computed) {
    compute_extreme_vertex_coordinates();
  }
  if(are_opposing_sites(in_bid, b_id) || in_bid == b_id) return true;
  Position full_position = undo_transform(in_p);
  unsigned int component = in_bid / 2;
  return full_position[component] == boundary_vertex_coordinates[in_bid];
}
```

References compute_extreme_vertex_coordinates().

5.36.3.35 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.

```
                                                                                   {
  std::vector<Position> ret(ids.size());
  for(unsigned int line_index_in_array = 0; line_index_in_array < ids.size(); line_index_in_array++) {
    Position p  =
   undo_transform(get_line_position_for_line_index_in_tria(&Geometry.surface_meshes[b_id],
   ids[line_index_in_array]));
    ret[line_index_in_array] = p;
  }
  return ret;
}
```

References undo_transform().

5.36.3.36 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.

```
                                                                                          {
  return "";
}
```

5.36.3.37 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.

```
                                         {
  dof_counter++;
  return dof_counter - 1;
}
```

Referenced by register_single_dof().

5.36.3.38 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.

```
{
  const int max_hsie_order = order;
  // EDGE Dofs
  std::vector<unsigned int> local_dofs(fe_nedelec.dofs_per_line);
  cell_nedelec->line(edge)->get_dof_indices(local_dofs);
  bool orientation = false;
  if(cell_nedelec->line(edge)->vertex_index(0) > cell_nedelec->line(edge)->vertex_index(1)) {
    orientation = get_orientation(undo_transform(cell_nedelec->line(edge)->vertex(0)),
   undo_transform(cell_nedelec->line(edge)->vertex(1)));
  } else {
    orientation = get_orientation(undo_transform(cell_nedelec->line(edge)->vertex(1)),
   undo_transform(cell_nedelec->line(edge)->vertex(0)));
  }

  for (int inner_order = 0; inner_order < static_cast<int>(fe_nedelec.dofs_per_line); inner_order++) {
    register_single_dof(cell_nedelec->face_index(edge), -1, inner_order + 1, DofType::EDGE,
   edge_dof_data, local_dofs[inner_order], orientation);
    Position bp = undo_transform(cell_nedelec->face(edge)->center(false, false));
    InterfaceDofData dof_data;
    dof_data.index = edge_dof_data[edge_dof_data.size() - 1].global_index;
    dof_data.order = inner_order;
    dof_data.base_point = bp;
    add_surface_relevant_dof(dof_data);
  }

  // INFINITE FACE Dofs Type a
  for (int inner_order = 0; inner_order < static_cast<int>(fe_nedelec.dofs_per_line); inner_order++) {
    for (int hsie_order = 0; hsie_order <= max_hsie_order; hsie_order++) {
      register_single_dof(cell_nedelec->face_index(edge), hsie_order, inner_order + 1, DofType::IFFa,
   edge_dof_data, local_dofs[inner_order], orientation);
    }
  }
  // INFINITE FACE Dofs Type b
  local_dofs.clear();
  local_dofs.resize(fe_q.dofs_per_line + 2 * fe_q.dofs_per_vertex);
  cell_q->line(edge)->get_dof_indices(local_dofs);
  IndexSet line_dofs(MAX_DOF_NUMBER);
  IndexSet non_line_dofs(MAX_DOF_NUMBER);
  for (unsigned int i = 0; i < local_dofs.size(); i++) {
    line_dofs.add_index(local_dofs[i]);
  }
  for (unsigned int i = 0; i < fe_q.dofs_per_vertex; i++) {
    non_line_dofs.add_index(cell_q->line(edge)->vertex_dof_index(0, i));
    non_line_dofs.add_index(cell_q->line(edge)->vertex_dof_index(1, i));
  }
  line_dofs.subtract_set(non_line_dofs);
  for (int inner_order = 0; inner_order < static_cast<int>(line_dofs.n_elements());
       inner_order++) {
    for (int hsie_order = -1; hsie_order <= max_hsie_order; hsie_order++) {
      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);
    }
  }
}
```

5.36.3.39 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.

```
{
  const int max_hsie_order = order;
  std::vector<unsigned int> surface_dofs(fe_nedelec.dofs_per_cell);
  cell_nedelec->get_dof_indices(surface_dofs);
  IndexSet surf_dofs(MAX_DOF_NUMBER);
  IndexSet edge_dofs(MAX_DOF_NUMBER);
  for (unsigned int i = 0; i < surface_dofs.size(); i++) {
    surf_dofs.add_index(surface_dofs[i]);
  }
  for (unsigned int i = 0; i < dealii::GeometryInfo<2>::lines_per_cell; i++) {
    std::vector<unsigned int> line_dofs(fe_nedelec.dofs_per_line);
    cell_nedelec->line(i)->get_dof_indices(line_dofs);
    for (unsigned int j = 0; j < line_dofs.size(); j++) {
      edge_dofs.add_index(line_dofs[j]);
    }
  }
  surf_dofs.subtract_set(edge_dofs);
  std::string id = cell_q->id().to_string();
  const unsigned int nedelec_dof_count = dof_h_nedelec.n_dofs();
  dealii::Vector<ComplexNumber> vec_temp(nedelec_dof_count);
  // SURFACE functions
  for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
    register_single_dof(cell_nedelec->id().to_string(), -1, inner_order, DofType::SURFACE,
 face_dof_data, surf_dofs.nth_index_in_set(inner_order));
    Position bp = undo_transform(cell_nedelec->center());
    InterfaceDofData dof_data;
    dof_data.index = face_dof_data[face_dof_data.size() - 1].global_index;
    dof_data.base_point = bp;
    dof_data.order = inner_order;
    add_surface_relevant_dof(dof_data);
  }

  // SEGMENT functions a
  for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
    for (int hsie_order = 0; hsie_order <= max_hsie_order; hsie_order++) {
      register_single_dof(id, hsie_order, inner_order, DofType::SEGMENTa, face_dof_data,
 surf_dofs.nth_index_in_set(inner_order));
    }
  }

  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().

5.36.3.40 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().

5.36.3.41 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().

5.36.3.42 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.
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().

5.36.3.43 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.

5.36.3.44 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.

```
                                                                                   {
  // The ray direction before transformation is x. This has to be adapted.
  HSIEPolynomial temp = (*vector)[0];
  switch (b_id) {
    case 2:
      (*vector)[0] = (*vector)[1];
      (*vector)[1] = temp;
      break;
    case 3:
      (*vector)[0] = (*vector)[1];
      (*vector)[1] = temp;
      break;
    case 4:
      (*vector)[0] = (*vector)[2];
      (*vector)[2] = temp;
      break;
    case 5:
      (*vector)[0] = (*vector)[2];
      (*vector)[2] = temp;
      break;
  }
}
```

Referenced by build_curl_term_nedelec(), build_curl_term_q(), and build_non_curl_term_q().

5.36.3.45 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.

```
{
  Position ret;
  ret[0] = inp[0];
  ret[1] = inp[1];
  ret[2] = additional_coordinate;
  switch (b_id) {
  case 0:
    ret = Transform_5_to_0(ret);
    break;
  case 1:
    ret = Transform_5_to_1(ret);
    break;
  case 2:
    ret = Transform_5_to_2(ret);
    break;
  case 3:
    ret = Transform_5_to_3(ret);
    break;
  case 4:
    ret = Transform_5_to_4(ret);
    break;
  default:
    break;
  }
  return ret;
}
```

Referenced by line_positions_for_ids(), and vertex_positions_for_ids().

5.36.3.46 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.

```
{
  Position ret;
  ret[0] = inp[0];
  ret[1] = inp[1];
  ret[2] = 0;
  switch (b_id) {
  case 0:
    ret = Transform_5_to_0(ret);
    break;
  case 1:
    ret = Transform_5_to_1(ret);
    break;
  case 2:
    ret = Transform_5_to_2(ret);
    break;
  case 3:
    ret = Transform_5_to_3(ret);
    break;
  case 4:
    ret = Transform_5_to_4(ret);
    break;
  default:
    break;
  }
  return ret;
}
```

5.36.3.47 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().

5.36.3.48 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().

5.36.3.49 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.

```
                                                    {
  const unsigned int dofs_per_vertex_all = compute_dofs_per_vertex();

  in_dof_count.total += dofs_per_vertex_all;
  in_dof_count.hsie += dofs_per_vertex_all;
}
```

References compute_dofs_per_vertex().

Referenced by compute_n_vertex_dofs().

5.36.3.50 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.

```
                                                                                 {
  std::vector<Position> ret(ids.size());
  for(unsigned int vertex_index_in_array = 0; vertex_index_in_array < ids.size();
   vertex_index_in_array++) {
    Position p =
   undo_transform(get_vertex_position_for_vertex_index_in_tria(&Geometry.surface_meshes[b_id],
   ids[vertex_index_in_array]));
    ret[vertex_index_in_array] = p;
  }
  return ret;
```

```
830 }
```

References undo_transform().

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

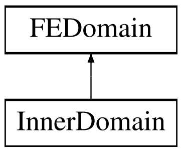
- Code/BoundaryCondition/HSIESurface.h
- Code/BoundaryCondition/HSIESurface.cpp

5.37 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**

5.37.1 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.

5.37.2 Member Function Documentation

5.37.2.1 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().

5.37.2.2 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.

```
    {
FEErrorStruct ret;
dealii::Vector<double> cell_vector (triangulation.n_active_cells());
QGauss<3> q(GlobalParams.Nedelec_element_order + 2);
NumericVectorLocal local_solution(n_locally_active_dofs);
for(unsigned int i =0 ; i < n_locally_active_dofs; i++) {
  local_solution[i] = in_solution->operator[](global_index_mapping[i]);
}
VectorTools::integrate_difference(dof_handler, local_solution, *GlobalParams.source_field,
 cell_vector, q, dealii::VectorTools::NormType::L2_norm);
ret.L2 = VectorTools::compute_global_error(triangulation, cell_vector,
 dealii::VectorTools::NormType::L2_norm);
ret.L2 /= in_solution->l2_norm();
VectorTools::integrate_difference(dof_handler, local_solution, *GlobalParams.source_field,
 cell_vector, q, dealii::VectorTools::NormType::Linfty_norm);
ret.Linfty = VectorTools::compute_global_error(triangulation, cell_vector,
 dealii::VectorTools::NormType::Linfty_norm);
ret.Linfty /= in_solution->linfty_norm();
return ret;
}
```

5.37.2.3 compute_kappa()

```
ComplexNumber InnerDomain::compute_kappa (
              NumericVectorLocal & in_solution )
```

Computes the value κ.

This value is defined by

$$\kappa = \int_{\Gamma_O} \overline{\boldsymbol{E}_0} \cdot \boldsymbol{E}_p A$$

Parameters

in_solution	

Returns

ComplexNumber

Definition at line 594 of file InnerDomain.cpp.

```
                                                                        {
ComplexNumber ret;
QGauss<2> quadrature_formula(1);
const FEValuesExtractors::Vector fe_field(0);
FEFaceValues<3> fe_values(fe, quadrature_formula, update_values | update_JxW_values |
 update_quadrature_points);
std::vector<unsigned int> local_dof_indices(fe.n_dofs_per_cell());
for (DofHandler3D::active_cell_iterator cell = dof_handler.begin_active(); cell != dof_handler.end();
 ++cell) {
    for(unsigned int face = 0; face < 6; face++) {
      if(std::abs(cell->face(face)->center()[2] - Geometry.global_z_range.second) < FLOATING_PRECISION)
    {
        fe_values.reinit(cell,face);
        double JxW;
        auto q_points = fe_values.get_quadrature_points();
        for(unsigned int q_index = 0; q_index < quadrature_formula.size(); q_index++) {
          cell->get_dof_indices(local_dof_indices);
          JxW = fe_values.get_JxW_values()[q_index];
          Position p = q_points[q_index];
          Tensor<1,3, ComplexNumber> E0;
          for(unsigned int i = 0; i < 3; i++) {
            E0[i] = GlobalParams.source_field->value(p,i);
          }
          for(unsigned int i = 0; i < fe.n_dofs_per_cell(); i++) {
            // 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;
            for(unsigned int j = 0; j < 3; j++) {
              ret += conjugate((in_solution[local_dof_indices[i]] * fe_values[fe_field].value(i,
    q_index))[j]) * E0[j] * JxW;
            }
          }
        }
      }
    }
  }
  return Utilities::MPI::sum(ret, MPI_COMM_WORLD);
}
```

5.37.2.4 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.

```
                                                                                {
  std::vector<FEAdjointEvaluation> ret;
  QGauss<3> quadrature_formula(1);
  const FEValuesExtractors::Vector fe_field(0);
  FEValues<3> fe_values(fe, quadrature_formula, update_values | update_gradients |
   update_quadrature_points);
  std::vector<unsigned int> local_dof_indices(fe.n_dofs_per_cell());
  for (DofHandler3D::active_cell_iterator cell = dof_handler.begin_active(); cell != dof_handler.end();
   ++cell) {
    fe_values.reinit(cell);
    auto q_points = fe_values.get_quadrature_points();
    for(unsigned int q_index = 0; q_index < quadrature_formula.size(); q_index++) {
      cell->get_dof_indices(local_dof_indices);
      Position p = q_points[q_index];
      FEAdjointEvaluation item;
      item.x = p;
      for(unsigned int i = 0; i < 3; i++) {
        item.primal_field[i] = 0;
        item.adjoint_field[i] = 0;
        item.primal_field_curl[i] = 0;
        item.adjoint_field_curl[i] = 0;
      }
      for(unsigned int i = 0; i < fe.n_dofs_per_cell(); i++) {
        Tensor<1, 3, ComplexNumber> I_Val;
        I_Val = fe_values[fe_field].value(i, q_index);
        Tensor<1, 3, ComplexNumber> I_Curl;
        I_Curl = fe_values[fe_field].curl(i, q_index);
        item.primal_field += I_Val * in_solution[local_dof_indices[i]];
        item.adjoint_field += I_Val * in_adjoint[local_dof_indices[i]];
        item.primal_field_curl += I_Curl * in_solution[local_dof_indices[i]];
        item.adjoint_field_curl += I_Curl * in_adjoint[local_dof_indices[i]];
      }
      ret.push_back(item);
    }
  }
  return ret;
}
```

5.37.2.5 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.

```
                                                {
  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;
}
```

5.37.2.6 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.

```
442                                                         {
443   return dof_handler.n_dofs();
444 }
```

5.37.2.7 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.

```
426                                                        {
427   IndexSet set_of_locally_owned_dofs(dof_handler.n_dofs());
428   set_of_locally_owned_dofs.add_range(0,dof_handler.n_dofs());
429   IndexSet dofs_to_remove(dof_handler.n_dofs());
430   for(unsigned int surf = 0; surf < 6; surf += 2) {
431     if(Geometry.levels[level].surface_type[surf] == SurfaceType::NEIGHBOR_SURFACE) {
432       std::vector<InterfaceDofData> dofs = get_surface_dof_vector_for_boundary_id(surf);
433       for(unsigned int i = 0; i < dofs.size(); i++) {
434         dofs_to_remove.add_index(dofs[i].index);
435       }
436     }
437   }
438   set_of_locally_owned_dofs.subtract_set(dofs_to_remove);
439   return set_of_locally_owned_dofs.n_elements();
440 }
```

5.37.2.8 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.

```
                  {
  ComplexNumber ret(0,0);
  if(GlobalParams.Index_in_z_direction == GlobalParams.Blocks_in_z_direction - 1) {
    NumericVectorLocal local_solution;
    local_solution.reinit(n_locally_active_dofs);
    for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
      local_solution[i] = in_solution->operator[](global_index_mapping[i]);
    }
    Vector<ComplexNumber> fe_evaluation(3);
    Vector<ComplexNumber> mode(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; // 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.
    }
    for(unsigned int index = 0; index < quadrature_points.size(); index++) {
      VectorTools::point_value(dof_handler, local_solution, quadrature_points[index], fe_evaluation);
      GlobalParams.source_field->vector_value(quadrature_points[index], mode);
      for(unsigned int comp = 0; comp < 3; comp++) {
        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;
}
```

5.37.2.9 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.

```
                                            {
  for(unsigned int i = 0; i < 6; i += 2) {
    if(Geometry.levels[level].surface_type[i] == SurfaceType::NEIGHBOR_SURFACE) {
      std::vector<InterfaceDofData> dof_data = get_surface_dof_vector_for_boundary_id(i);
      std::vector<unsigned int> local_dof_indices(dof_data.size());
      for(unsigned int j = 0; j < dof_data.size(); j++) {
        local_dof_indices[j] = dof_data[j].index;
      }
      mark_local_dofs_as_non_local(local_dof_indices);
    }
  }
}
```

5.37.2.10 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.

```
                                  {
  std::vector<std::vector<ComplexNumber» ret;
  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;
}
```

5.37.2.11 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.

5.37.2.12 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.

```
          {
  auto end = dof_handler.end();
  std::vector<DofNumber> cell_dof_indices(fe.dofs_per_cell);
  for(auto cell = dof_handler.begin_active(); cell != end; cell++) {
    cell->get_dof_indices(cell_dof_indices);
    cell_dof_indices = transform_local_to_global_dofs(cell_dof_indices);
    in_constraints->add_entries_local_to_global(cell_dof_indices, *in_pattern);
  }
}
```

References FEDomain::transform_local_to_global_dofs().

5.37.2.13 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.

```
                                                                                      {
  std::vector<InterfaceDofData> ret;
  std::vector<types::global_dof_index> local_line_dofs(fe.dofs_per_line);
  std::set<DofNumber> line_set;
  std::vector<DofNumber> local_face_dofs(fe.dofs_per_face);
  std::set<DofNumber> face_set;
  triangulation.clear_user_flags();
  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;
140           for (auto item: face_set) {
141             InterfaceDofData new_item;
142             new_item.index = item;
143             new_item.base_point = cell->face(face)->center();
144             new_item.order = 0;
145             cell_dofs_and_orientations_and_points.push_back(new_item);
146             index++;
147           }
148           for (auto item: cell_dofs_and_orientations_and_points) {
149             ret.push_back(item);
150           }
151         }
152       }
153     }
154   }
155   ret.shrink_to_fit();
156   std::sort(ret.begin(), ret.end(), compareDofBaseDataAndOrientation);
157   return ret;
158 }
```

5.37.2.14 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;
```

```
    print_info("InnerDomain::load_exact_solution", "Norm of interpolated mode signal is " +
    std::to_string(exact_solution_interpolated.l2_norm()));
  }
}
```

Referenced by assemble_system().

5.37.2.15 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.

```
                  {
  print_info("InnerDomain::output_results()", "Start");
  const unsigned int n_cells = dof_handler.get_triangulation().n_active_cells();
  unsigned int counter = 0;
  dealii::Vector<double> eps_abs(n_cells);
  for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
    Position p = it->center();
    MaterialTensor transformation;
    if(apply_transformation) {
      for(unsigned int i = 0 ; i < 3; i++) {
        for(unsigned int j = 0; j < 3; j++) {
          if(i == j) {
            transformation[i][j] = ComplexNumber(1,0);
          } else {
            transformation[i][j] = ComplexNumber(0,0);
          }
        }
      }
    } else {
      transformation = GlobalSpaceTransformation->get_Space_Transformation_Tensor(p);
    }
    MaterialTensor epsilon;
    if (Geometry.math_coordinate_in_waveguide(p)) {
      epsilon = transformation * GlobalParams.Epsilon_R_in_waveguide;
    } else {
      epsilon = transformation * GlobalParams.Epsilon_R_outside_waveguide;
    }
    eps_abs[counter] = epsilon.norm();
    counter++;
  }
  if(apply_transformation) {
    GlobalSpaceTransformation->switch_application_mode(true);
    dealii::GridTools::transform(*GlobalSpaceTransformation, triangulation);
  }
  dealii::Vector<ComplexNumber> interpolated_exact_solution(in_solution.size());

```

```
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");
392   std::ofstream outputvtu(filename);
393 
394   Function<3,ComplexNumber> * esc;
395   if(!apply_transformation) {
396     data_out.add_data_vector(exact_solution_interpolated, "Exact_Solution");
397   } else {
398     esc = GlobalParams.source_field;
399     dealii::IndexSet local_indices(n_locally_active_dofs);
400     local_indices.add_range(0,n_locally_active_dofs);
401     Constraints local_constraints(local_indices);
402     local_constraints.close();
403     if(GlobalParams.Point_Source_Type == 0 || GlobalParams.Point_Source_Type == 3) {
404       VectorTools::project(dof_handler, local_constraints,
       dealii::QGauss<3>(GlobalParams.Nedelec_element_order + 2), *esc, interpolated_exact_solution);
405       data_out.add_data_vector(interpolated_exact_solution, "Exact_Solution");
406     }
407   }
408 
409   dealii::Vector<ComplexNumber> error_vector(in_solution.size());
410   for(unsigned int i = 0; i < in_solution.size(); i++) {
411     error_vector[i] = in_solution[i] - exact_solution_interpolated[i];
412   }
413   data_out.add_data_vector(error_vector, "SolutionError");
414 
415   data_out.build_patches();
416   data_out.write_vtu(outputvtu);
417   if(apply_transformation) {
418     delete esc;
419     GlobalSpaceTransformation->switch_application_mode(false);
420     dealii::GridTools::transform(*GlobalSpaceTransformation, triangulation);
421   }
422   print_info("InnerDomain::output_results()", "End");
423   return filename;
424 }
```

5.37.2.16 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);
```

```
  print_info("InnerDomain::write_matrix_and_rhs_metrics", "End");
}
```

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

- Code/Core/InnerDomain.h
- Code/Core/InnerDomain.cpp

5.38 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**

5.38.1 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

5.39 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**

5.39.1 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

5.40 JacobianAndTensorData Struct Reference

Public Attributes

- dealii::Tensor< 2, 3, double > **C**
- dealii::Tensor< 2, 3, double > **G**
- dealii::Tensor< 2, 3, double > **J**

5.40.1 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

5.41 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**

5.41.1 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.

5.41.2 Constructor & Destructor Documentation

5.41.2.1 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.

```
                {
  reinit(in_fad, in_bid, in_additional_component);
}
```

References reinit().

5.41.3 Member Function Documentation

5.41.3.1 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.

```
                                                                       {
  JacobianAndTensorData ret;
  ret.J = get_J_hat_for_position(in_p);
  const double J_norm = ret.J.norm();
  const Tensor<2,3,double> J_inverse = invert(ret.J);
  ret.G = J_norm * J_inverse * transpose(J_inverse);
  ret.C = (1.0 / J_norm) * transpose(ret.J) * ret.J;
  return ret;
}
```

References get_J_hat_for_position().

5.41.3.2 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.

```
{
  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().

5.41.3.3 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

5.41.3.4 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

5.41.3.5 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.

```
           {
  x                    = {"x"};
  y                    = {"y"};
  z                    = {"z"};
  z0                   = {"z0"};
  boundary_id          = in_bid;
  additional_component = in_additional_component;
  bool all_straight = true;
  for(unsigned int i = 0; i < 4; i++) {
    if(!in_fad[i].is_x_angled || !in_fad[i].is_y_angled) {
      all_straight = false;
    }
  }
  if(all_straight) {
    F[0]                   = x;
    F[1]                   = y;
    F[2]                   = (z-z0);
  } else {
    F[0]                   = x;
    F[1]                   = y;
    F[2]                   = (z-z0);
  }
  surface_wide_substitution_map[z0]   = MathExpression(in_additional_component);
  for(unsigned int i = 0; i <3; i++) {
    F[i] = F[i].substitute(surface_wide_substitution_map);
  }
  for(unsigned int i = 0; i < 3; i++) {
    J[i][0] = F[i].differentiate(x);
    J[i][1] = F[i].differentiate(y);
    J[i][2] = F[i].differentiate(z);
  }
}
```

Referenced by JacobianForCell().

5.41.3.6 reinit_for_cell()

```
void JacobianForCell::reinit_for_cell (
              CellIterator2D  )
```

Builds the base data for the provided cell.

5.41.3.7 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.

```
      {
  Position temp = in_point;
  if (boundary_id == 0) {
    temp = Transform_0_to_5(in_point);
  }
  if (boundary_id == 1) {
    temp = Transform_1_to_5(in_point);
  }
  if (boundary_id == 2) {
    temp = Transform_2_to_5(in_point);
  }
  if (boundary_id == 3) {
    temp = Transform_3_to_5(in_point);
  }
  if (boundary_id == 4) {
    temp = Transform_4_to_5(in_point);
  }
  return {{temp[0], temp[1]}, temp[2]};
}
```

5.41.3.8 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.

```
                                                                    {
  Position ret= {in_position[0], in_position[1], additional_component};
  if (boundary_id == 0) {
    return Transform_5_to_0(ret);
  }
  if (boundary_id == 1) {
    return Transform_5_to_1(ret);
  }
  if (boundary_id == 2) {
    return Transform_5_to_2(ret);
  }
  if (boundary_id == 3) {
    return Transform_5_to_3(ret);
  }
  if (boundary_id == 4) {
    return Transform_5_to_4(ret);
  }
  if (boundary_id == 5) {
    return ret;
  }
  return ret;
}
```

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

- Code/BoundaryCondition/JacobianForCell.h
- Code/BoundaryCondition/JacobianForCell.cpp

5.42 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)

5.42.1 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

5.43 LaguerreFunctions Class Reference

```
#include <LaguerreFunction.h>
```

5.43.1 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

5.44 LevelDofIndexData Class Reference

5.44.1 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

5.45 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**

5.45.1 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

5.46 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**

5.46.1 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

5.47 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**

5.47.1 Detailed Description

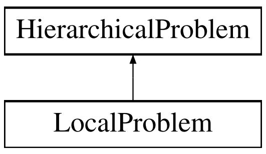
Definition at line 64 of file Types.h.

The documentation for this struct was generated from the following file:

- Code/Core/Types.h

5.48 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**

5.48.1 Detailed Description

Definition at line 13 of file LocalProblem.h.

5.48.2 Constructor & Destructor Documentation

5.48.2.1 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.

```
                            :
  HierarchicalProblem(0, SweepingDirection::Z),
  sc(),
  solver(sc, MPI_COMM_SELF) {
    solver.set_symmetric_mode(true);
    print_info("Local Problem", "Done building base problem. Preparing matrix.");
    matrix = new dealii::PETScWrappers::MPI::SparseMatrix();
    for(unsigned int i = 0; i < 6; i++) Geometry.levels[0].is_surface_truncated[i] = true;
    own_dofs = Geometry.levels[0].dof_distribution[0];
}
```

5.48.3 Member Function Documentation

5.48.3.1 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.

```
                                    {
  Timer timer;
  timer.start ();
  double error = compute_L2_error();
  timer.stop ();
  print_info("LocalProblem::compute_error", "L2 Error: " + std::to_string(error) + " (computed in " +
   std::to_string(timer.cpu_time()) + "s)");
  return error;
}
```

References compute_L2_error().

5.48.3.2 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.

```
                                                       {
  return Utilities::MPI::sum(solve_counter, MPI_COMM_WORLD);
}
```

5.48.3.3 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.

```
                                                                              {
  BoundaryId opposing_interface_id = opposing_Boundary_Id(interface_id);
  dealii::IndexSet ret(Geometry.levels[0].n_local_dofs);
  for(unsigned int i = 0; i < 6; i++) {
    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;
}
```

5.48.3.4 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.

```
                                         {
  NumericVectorLocal solution_inner(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++) {
    solution_inner[i] = solution(i);
  }
  dealii::Vector<double>
   cellwise_error(Geometry.levels[level].inner_domain->triangulation.n_active_cells());
  dealii::VectorTools::integrate_difference(
    MappingQGeneric<3>(1),
    Geometry.levels[level].inner_domain->dof_handler,
    solution_inner,
    *GlobalParams.source_field,
    cellwise_error,
    dealii::QGauss<3>(GlobalParams.Nedelec_element_order + 2),
    dealii::VectorTools::NormType::L2_norm );
  return dealii::VectorTools::compute_global_error(Geometry.levels[level].inner_domain->triangulation,
   cellwise_error, dealii::VectorTools::NormType::L2_norm);
}
```

Referenced by compute_error().

5.48.3.5 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.

```
                                                    {
  Timer timer1;
  print_info("LocalProblem::compute_solver_factorization", "Begin solver factorization: ",
   LoggingLevel::PRODUCTION_ONE);
  timer1.start();
  solve();
  timer1.stop();
  solution = 0;
  print_info("LocalProblem::compute_solver_factorization", "Walltime: " +
   std::to_string(timer1.wall_time()) , LoggingLevel::PRODUCTION_ONE);
}
```

5.48.3.6 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.

```
                                                                           {
  NumericVectorLocal local_solution(Geometry.levels[0].inner_domain->n_locally_active_dofs);
  std::vector<std::string> generated_files;
  for(unsigned int i = 0; i < Geometry.levels[0].inner_domain->n_locally_active_dofs; i++) {
    local_solution[i] = solution[i];
  }

  std::string file_1 = Geometry.levels[0].inner_domain->output_results(in_filename + "0" ,
   local_solution, false);
  generated_files.push_back(file_1);
  if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
    for (unsigned int surf = 0; surf < 6; surf++) {
      if(Geometry.levels[0].surface_type[surf] == SurfaceType::ABC_SURFACE){
        dealii::Vector<ComplexNumber> ds (Geometry.levels[0].surfaces[surf]->n_locally_active_dofs);
        for(unsigned int index = 0; index < Geometry.levels[0].surfaces[surf]->n_locally_active_dofs;
   index++) {
          ds[index] = solution[Geometry.levels[0].surfaces[surf]->global_index_mapping[index]];
        }
        std::string file_2 = Geometry.levels[0].surfaces[surf]->output_results(ds, in_filename +
   "_pml0");
        generated_files.push_back(file_2);
      }
    }
  }

  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

5.49 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** ()

5.49.1 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

5.50 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**

5.50.1 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.

5.50.2 Member Function Documentation

5.50.2.1 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:

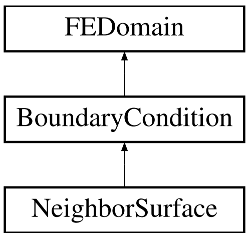
- Code/Hierarchy/MPICommunicator.h
- Code/Hierarchy/MPICommunicator.cpp

5.51 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**

5.51.1 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.

5.51.2 Member Function Documentation

5.51.2.1 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.

```
                                                            {
    return 0;
}
```

5.51.2.2 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.

```
                                                                       {
    return 0;
}
```

5.51.2.3 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.

```
                       {
    matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
    required.
    // Nothing to do here, work happens on neighbor process.
}
```

5.51.2.4 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_constraints	Constraints to use

Implements BoundaryCondition.

Definition at line 68 of file NeighborSurface.cpp.

```
                                                                                            {
}
```

5.51.2.5 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.

```
                                                     {
    prepare_dofs();
    if(is_lower_interface) {
        receive();
    } else {
        send();
    }
}
```

References prepare_dofs(), receive(), and send().

5.51.2.6 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.

```
                                                                  {
    std::vector<InterfaceDofData> dof_indices =
    Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
    for(unsigned int i = 0; i < dof_indices.size(); i++) {
        dof_indices[i].index = inner_dofs[i];
    }
    return dof_indices;
}
```

5.51.2.7 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.

```
{
    std::vector<InterfaceDofData> own_dof_indices;
    for(unsigned int i = 0; i < boundary_dofs[in_boundary_id].size(); i++) {
        InterfaceDofData idd;
        idd.order = 0;
        idd.base_point = {0,0,0};
        idd.index = boundary_dofs[in_boundary_id][i];
        own_dof_indices.push_back(idd);
    }
    return own_dof_indices;
}
```

5.51.2.8 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.

```
                                                                   {
    return false;
}
```

5.51.2.9 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.

```
                                                                                           {
    return "";
}
```

5.51.2.10 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.

```
                                   {
     // 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.
     std::vector<InterfaceDofData> temp =
    Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
     n_dofs = 0;
     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:

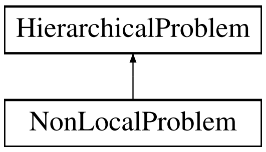
- Code/BoundaryCondition/NeighborSurface.h
- Code/BoundaryCondition/NeighborSurface.cpp

5.52 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

5.52.1 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.

5.52.2 Constructor & Destructor Documentation

5.52.2.1 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 (NonlocalProblem(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.

```
                                                    :
  HierarchicalProblem(level, static_cast<SweepingDirection> (2 + GlobalParams.Sweeping_Level - level)),
  sc(GlobalParams.GMRES_max_steps, GlobalParams.Solver_Precision, true, true)
{
  sweeping_direction = get_sweeping_direction_for_level(level);
  if(level > 1) {
    child = new NonLocalProblem(level - 1);
  } else {
    child = new LocalProblem();
  }

  prepare_sweeping_data();

  matrix = new dealii::PETScWrappers::MPI::SparseMatrix();

  locally_active_dofs = dealii::IndexSet(Geometry.levels[level].n_total_level_dofs);
  for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->global_index_mapping.size(); i++) {
    locally_active_dofs.add_index(Geometry.levels[level].inner_domain->global_index_mapping[i]);
  }

  for(unsigned int surf = 0; surf < 6; surf++) {
    Geometry.levels[level].surfaces[surf]->print_dof_validation();
  }
  for(unsigned int surf = 0; surf < 6; surf++) {
    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);
}
```

5.52.3 Member Function Documentation

5.52.3.1 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.

```
                                                 {
  set_u_from_vec_object(b_in);
  perform_downward_sweep();
  perform_upward_sweep();
  set_x_out_from_u(u_out);
}
```

References perform_downward_sweep(), perform_upward_sweep(), set_u_from_vec_object(), and set_x_out_↩
from_u().

5.52.3.2 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.

```
                                {
  matrix->operator=(0);
  rhs = 0;
  matrix->compress(dealii::VectorOperation::insert);
  rhs.compress(dealii::VectorOperation::insert);
  print_info("NonLocalProblem::assemble", "Begin assembly");
  GlobalTimerManager.switch_context("Assemble", level);
  Timer timer;
  timer.start();
  Geometry.levels[level].inner_domain->assemble_system(&constraints, matrix, &rhs);
  print_info("NonLocalProblem::assemble", "Inner assembly done. Assembling boundary method
   contributions.");
  for(unsigned int i = 0; i< 6; i++) {
      Geometry.levels[level].surfaces[i]->fill_matrix(matrix, &rhs, &constraints);
  }
  timer.stop();
  print_info("NonLocalProblem::assemble", "Compress matrix.");
  matrix->compress(dealii::VectorOperation::add);
  rhs.compress(dealii::VectorOperation::add);
  print_info("NonLocalProblem::assemble", "Assemble child.");
  child->assemble();
  print_info("NonLocalProblem::assemble", "Compress vectors.");
  solution.compress(dealii::VectorOperation::add);
  rhs.compress(VectorOperation::add);
  // constraints.distribute(solution);
  print_info("NonLocalProblem::assemble", "End assembly.");
  KSPSetOperators(ksp, *matrix, *matrix);
  GlobalTimerManager.leave_context(level);
}
```

Referenced by OptimizationRun::solve_main_problem().

5.52.3.3 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.

```
{
  std::vector<std::vector<unsigned int» rows, cols;
  for(unsigned int i = 0; i < n_procs_in_sweep; i++) {
    rows.emplace_back();
    cols.emplace_back();
  }
  for(auto it = in_dsp->begin(); it != in_dsp->end(); it++) {
    if(!own_dofs.is_element(it->row())) {
      for(unsigned int proc = 0; proc < n_procs_in_sweep; proc++) {
        if(Geometry.levels[level].dof_distribution[proc].is_element(it->row())) {
          rows[proc].push_back(it->row());
          cols[proc].push_back(it->column());
        }
      }
    }
  }
  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]);
    }
  }
}
```

5.52.3.4 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.

```
                                                                   {
  // always more dofs on the lower level
  dealii::IndexSet lower_is (Geometry.levels[level-1].n_total_level_dofs);
  dealii::IndexSet upper_is (Geometry.levels[level].n_total_level_dofs);
  auto higher_cell = Geometry.levels[level].surfaces[in_bid]->dof_handler.begin();
  auto lower_cell = Geometry.levels[level-1].surfaces[in_bid]->dof_handler.begin();
  auto higher_end = Geometry.levels[level].surfaces[in_bid]->dof_handler.end();
  auto lower_end = Geometry.levels[level-1].surfaces[in_bid]->dof_handler.end();
  while(higher_cell != higher_end) {
    bool found = true;
    // first find the same cell in the child
    if(! ((higher_cell->center() - lower_cell->center()).norm() < FLOATING_PRECISION)) {
      while((higher_cell->center() - lower_cell->center()).norm() > FLOATING_PRECISION && lower_cell != lower_end) {
        lower_cell++;
      }
      if(lower_cell == lower_end) {
        lower_cell = Geometry.levels[level-1].surfaces[in_bid]->dof_handler.begin();
      }
      while((higher_cell->center() - lower_cell->center()).norm() > FLOATING_PRECISION && lower_cell !=
     lower_end) {
        lower_cell++;
      }
      if(lower_cell == lower_end) {
        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));
  }
}
```

5.52.3.5 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.

```
                      {
  FEErrorStruct errors = Geometry.levels[level].inner_domain->compute_errors(in_solution);
  FEErrorStruct ret;
  ret.L2 = Utilities::MPI::sum(errors.L2, GlobalMPI.communicators_by_level[level]);
  ret.Linfty = Utilities::MPI::max(errors.Linfty, GlobalMPI.communicators_by_level[level]);
  return ret;
}
```

5.52.3.6 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.

```
                                                            {
  unsigned int contribution = 0;
  if(total_rank_in_sweep == 0) {
    contribution = solve_counter;
  }
  return Utilities::MPI::sum(contribution, MPI_COMM_WORLD);
}
```

5.52.3.7 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.

```
                                    {
  double temp = Geometry.h_x;
  temp = std::max(temp, Geometry.h_y);
  temp = std::max(temp, Geometry.h_z);
  return temp;
}
```

5.52.3.8 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.

```
                                                                 {
  print_info("NonLocalProblem::compute_shape_gradient", "Start");
  const unsigned int n_shape_dofs = GlobalSpaceTransformation->n_free_dofs();
  std::vector<double> ret(n_shape_dofs);
  for(unsigned int i = 0; i < n_shape_dofs; i++) {
    ret[i] = 0;
  }

  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;
}
```

5.52.3.9 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]);
  ComplexNumber mode_sum = dealii::Utilities::MPI::sum(base, GlobalMPI.communicators_by_level[level]);
  print_info("NonLocalProblem::compute_signal_strength_of_solution", "End");
  return integral_sum / mode_sum;
}
```

Referenced by OptimizationRun::perform_step().

5.52.3.10 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.

```
                                                 {
  child->compute_solver_factorization();
}
```

References HierarchicalProblem::compute_solver_factorization().

Referenced by OptimizationRun::solve_main_problem().

5.52.3.11 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.

```
                                                          {
  return Geometry.levels[level].n_total_level_dofs;
}
```

5.52.3.12 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.

```
                                   {
  matrix->clear();
  KSPReset(ksp);
  child->empty_memory();
}
```

References HierarchicalProblem::empty_memory().

5.52.3.13 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.

```
                                                                                                  {
  NumericVectorLocal local_solution(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]];
  }
  return Geometry.levels[level].inner_domain->evaluate_at_positions(positions, local_solution);
}
```

5.52.3.14 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 iterative 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.

```
                                                      {
  // dealii::PETScWrappers::PreconditionNone pc_none;
  // pc_none.initialize(*matrix);
  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);
}
```

5.52.3.15 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.

```
                                  {
  GlobalTimerManager.switch_context("Initialize", level);
  child->initialize();
  initialize_index_sets();
  reinit_all_vectors();
  reinit();
  init_solver_and_preconditioner();
  GlobalTimerManager.leave_context(level);
}
```

5.52.3.16 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.

```
                                                {
  own_dofs = Geometry.levels[level].dof_distribution[total_rank_in_sweep];
  locally_owned_dofs_index_array = new PetscInt[own_dofs.n_elements()];
  get_petsc_index_array_from_index_set(locally_owned_dofs_index_array, own_dofs);

}
```

5.52.3.17 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.

```
                                                {
  unsigned int local = Geometry.levels[level].inner_domain->triangulation.n_active_cells();
  for(unsigned int i = 0; i < 6; i++) {
    local += Geometry.levels[level].surfaces[i]->n_cells();
  }
  unsigned int ret = dealii::Utilities::MPI::sum(local, MPI_COMM_WORLD);
  return ret;
}
```

5.52.3.18 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.

```
                                            {
  print_info("NonLocalProblem", "Start output results on level" + std::to_string(level));
  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 "";
}
```

5.52.3.19 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.

```
                                              {
  for(int i = n_blocks_in_sweeping_direction - 1; i >= 0; i--) {
    if((int)index_in_sweeping_direction == i) {
      S_inv(&u, &dist_vector_1);
    } else {
      for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
        dist_vector_1[own_dofs.nth_index_in_set(j)] = 0;
      }
    }
    dist_vector_1.compress(VectorOperation::insert);
    matrix->vmult(dist_vector_2, dist_vector_1);
    if((int)index_in_sweeping_direction == i-1) {
      for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
        const unsigned int index = own_dofs.nth_index_in_set(j);
        ComplexNumber current_value(u(index).real(), u(index).imag());
        ComplexNumber delta(dist_vector_2[index].real(), dist_vector_2[index].imag());
        u[index] = current_value - delta;
      }
    }
    if((int)index_in_sweeping_direction == i) {
      for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
        const unsigned int index = own_dofs.nth_index_in_set(j);
        u[index] = (ComplexNumber) dist_vector_1[index];
      }
    }
    u.compress(VectorOperation::insert);
  }
}
```

References S_inv().

Referenced by apply_sweep().

5.52.3.20 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.

```
                                            {
  for(unsigned int i = 0; i < n_blocks_in_sweeping_direction-1; i++) {
    if(index_in_sweeping_direction == i) {
      for(unsigned int index = 0; index < own_dofs.n_elements(); index++) {
        dist_vector_1[own_dofs.nth_index_in_set(index)] = (ComplexNumber)
       u[own_dofs.nth_index_in_set(index)];
      }
    } else {
      for(unsigned int index = 0; index < own_dofs.n_elements(); index++) {
        dist_vector_1[own_dofs.nth_index_in_set(index)] = 0;
      }
    }
    dist_vector_1.compress(VectorOperation::insert);
    matrix->Tvmult(dist_vector_2, dist_vector_1);

    if(index_in_sweeping_direction == i+1) {
      S_inv(&dist_vector_2, &dist_vector_3);
      for(unsigned int j = 0; j < own_dofs.n_elements(); j++) {
        const unsigned int index = own_dofs.nth_index_in_set(j);
        ComplexNumber current_value = u(index);
        ComplexNumber delta = dist_vector_3[index];
        u[index] = current_value - delta;
      }
    }
    u.compress(VectorOperation::insert);
  }
}
```

Referenced by apply_sweep().

5.52.3.21 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.

```
                                                                                    {
  in_v->extract_subvector_to(vector_copy_own_indices, vector_copy_array);
  double local_norm = 0.0;
  double max = 0;
  for(unsigned int i = 0; i < vector_copy_array.size(); i++) {
    double local = std::abs(vector_copy_array[i])*std::abs(vector_copy_array[i]);
    if(local > max) {
      max = local;
    }
    local_norm += local;
  }
  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;
  }
}
```

5.52.3.22 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.

```
                                                                                          {
  vector_copy_own_indices.push_back(own_index);
  vector_copy_child_indeces.push_back(child_index);
  vector_copy_array.push_back(ComplexNumber(0.0, 0.0));
}
```

5.52.3.23 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.

```
                             {
  print_info("Nonlocal reinit", "Reinit starting for level " + std::to_string(level));
  MPI_Barrier(MPI_COMM_WORLD);
  GlobalTimerManager.switch_context("Reinit", level);

  make_constraints();

  make_sparsity_pattern();
  MPI_Barrier(MPI_COMM_WORLD);

  if(GlobalParams.MPI_Rank == 0) std::cout « "Start reinit of rhs vector." « std::endl;

  reinit_rhs();
  MPI_Barrier(MPI_COMM_WORLD);

  if(GlobalParams.MPI_Rank == 0) std::cout « "Start reinit of system matrix." « std::endl;

```

```
372   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]);
373
374   MPI_Barrier(MPI_COMM_WORLD);
375
376   if(GlobalParams.MPI_Rank == 0) print_info("Nonlocal reinit", "Matrix initialized");
377
378   for(unsigned int i = 0; i < Geometry.levels[level].inner_domain->n_locally_active_dofs; i++) {
379     if(Geometry.levels[level].inner_domain->is_dof_owned[i] &&
       Geometry.levels[level-1].inner_domain->is_dof_owned[i]) {
380       vector_copy_own_indices.push_back(Geometry.levels[level].inner_domain->global_index_mapping[i]);
381
       vector_copy_child_indeces.push_back(Geometry.levels[level-1].inner_domain->global_index_mapping[i]);
382       vector_copy_array.push_back(ComplexNumber(0.0, 0.0));
383     }
384   }
385   for(unsigned int surf = 0; surf < 6; surf++) {
386     if(Geometry.levels[level].surface_type[surf] == Geometry.levels[level-1].surface_type[surf]) {
387       if(Geometry.levels[level].surfaces[surf]->dof_counter !=
       Geometry.levels[level-1].surfaces[surf]->dof_counter) {
388         complex_pml_domain_matching(surf);
389       } else {
390         for(unsigned int i = 0; i < Geometry.levels[level].surfaces[surf]->n_locally_active_dofs; i++) {
391           if(Geometry.levels[level].surfaces[surf]->is_dof_owned[i] &&
       Geometry.levels[level-1].surfaces[surf]->is_dof_owned[i]) {
392             register_dof_copy_pair(Geometry.levels[level].surfaces[surf]->global_index_mapping[i],
       Geometry.levels[level-1].surfaces[surf]->global_index_mapping[i]);
393           }
394         }
395       }
396     }
397   }
398   GlobalTimerManager.leave_context(level);
399   print_info("Nonlocal reinit", "Reinit done for level " + std::to_string(level));
400 }
```

5.52.3.24 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↩_timers_and_count().

Referenced by perform_downward_sweep().

5.52.3.25 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.

```
                                                    {
  const unsigned int n_loc_dofs = own_dofs.n_elements();
  const ComplexNumber * pointer;
  VecGetArrayRead(in_v, &pointer);
  for(unsigned int i = 0; i < n_loc_dofs; i++) {
    u[own_dofs.nth_index_in_set(i)] = *pointer;
    pointer++;
  }
  VecRestoreArrayRead(in_v, &pointer);
  u.compress(dealii::VectorOperation::insert);
}
```

Referenced by apply_sweep().

5.52.3.26 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.

```
                                                                                  {
  child->solution.extract_subvector_to(vector_copy_child_indeces, vector_copy_array);
  in_u->set(vector_copy_own_indices, vector_copy_array);
  //in_u->compress(VectorOperation::insert);
}
```

Referenced by S_inv().

5.52.3.27 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.

```
                                               {
  ComplexNumber * values = new ComplexNumber[own_dofs.n_elements()];

  u.extract_subvector_to(vector_copy_own_indices, vector_copy_array);

  for(unsigned int i = 0; i < own_dofs.n_elements(); i++) {
    values[i] = vector_copy_array[i];
  }

  VecSetValues(x_out, own_dofs.n_elements(), locally_owned_dofs_index_array, values, INSERT_VALUES);
  VecAssemblyBegin(x_out);
  VecAssemblyEnd(x_out);
  delete[] values;
}
```

Referenced by apply_sweep().

5.52.3.28 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.

```
                           {
  is_shared_solution_up_to_date = false;
  std::chrono::steady_clock::time_point time_begin;
  std::chrono::steady_clock::time_point time_end;
  if(level == GlobalParams.Sweeping_Level) {
    print_vector_norm(&rhs, "RHS");
    time_begin = std::chrono::steady_clock::now();
  }

  bool run_itterative_solver = !GlobalParams.solve_directly;

  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]");
  }

}
```

5.52.3.29 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.

```
                                                                         {
  if(GlobalParams.use_relative_convergence_criterion) {
    double base_value = last_residual;
    if(last_residual > 1.0) {
      base_value = 1.0;
    }
    double new_abort_limit = base_value * GlobalParams.relative_convergence_criterion;
    new_abort_limit = std::max(new_abort_limit, GlobalParams.Solver_Precision);
    KSPSetTolerances(ksp, 1e-10,new_abort_limit , 1000, GlobalParams.GMRES_max_steps);
    // std::cout « "Setting level " « level « " convergence criterion to " « new_abort_limit «
   std::endl;
  }
}
```

5.52.3.30 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.

```
                                                         {
  if(! is_shared_solution_up_to_date) {
    shared_solution.reinit(own_dofs, locally_active_dofs, GlobalMPI.communicators_by_level[level]);
    for(unsigned int i= 0; i < own_dofs.n_elements(); i++) {
      shared_solution[own_dofs.nth_index_in_set(i)] = solution[own_dofs.nth_index_in_set(i)];
    }
    shared_solution.update_ghost_values();
    is_shared_solution_up_to_date = true;
  }
  if(has_adjoint) {
    shared_adjoint.reinit(own_dofs, locally_active_dofs, GlobalMPI.communicators_by_level[level]);
    for(unsigned int i= 0; i < own_dofs.n_elements(); i++) {
      shared_adjoint[own_dofs.nth_index_in_set(i)] = adjoint_state[own_dofs.nth_index_in_set(i)];
    }
    shared_adjoint.update_ghost_values();
  }
}
```

Referenced by set_rhs_for_adjoint_problem(), and write_multifile_output().

5.52.3.31 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.

```
                                                                                         {
  update_shared_solution_vector();
  if(GlobalParams.MPI_Rank == 0 && !GlobalParams.solve_directly) {
    residual_output->run_gnuplot();
    if(level > 1) {
      child->residual_output->run_gnuplot();
      if(level == 3) {
        child->child->residual_output->run_gnuplot();
      }
    }
  }
  std::vector<std::string> generated_files;

  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:

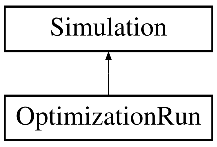
- Code/Hierarchy/NonLocalProblem.h
- Code/Hierarchy/NonLocalProblem.cpp

5.53 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.

5.53.1 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.

5.53.2 Constructor & Destructor Documentation

5.53.2.1 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.

```
                                   :
    n_free_dofs(GlobalSpaceTransformation->n_free_dofs())
    {
    function_pointer = &OptimizationRun::perform_step;
    OptimizationRun::step_counter = 0;
  }
```

5.53.3 Member Function Documentation

5.53.3.1 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.

```
                                                                                            {
    std::vector<double> x_vec(x.size());
    for(unsigned int i = 0; i < x.size(); i++) {
      x_vec[i] = x[i];
    }
    OptimizationRun::shape_dofs.push_back(x_vec);
    OptimizationRun::set_shape_dofs(x);
    OptimizationRun::solve_main_problem();
    double loss_functional_evaluation = -std::abs(mainProblem->compute_signal_strength_of_solution());
    print_info("OptimizationRun::perform_step", "Loss functional in step " +
      std::to_string(OptimizationRun::step_counter) + ": " + std::to_string(loss_functional_evaluation));
    std::vector<double> shape_grad = mainProblem->compute_shape_gradient();
    OptimizationRun::shape_gradients.push_back(shape_grad);
    std::string msg = "Shape gradient: ( ";
    for(unsigned int i = 0; i < g.size(); i++) {
      g[i] = shape_grad[i];
      msg += std::to_string(g[i]);
      if(i < g.size() -1) {
        msg += ", ";
      } else {
        msg += ")";
      }
    }
    print_info("OptimizationRun::perform_step", msg);
    OptimizationRun::step_counter += 1;
    return loss_functional_evaluation;
}
```

References NonLocalProblem::compute_signal_strength_of_solution(), set_shape_dofs(), and solve_main_↩ problem().

5.53.3.2 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.

```
                             {
    print_info("OptimizationRun::run", "Start", LoggingLevel::PRODUCTION_ONE);
    const unsigned int n_shape_dofs = n_free_dofs;
    dealii::Vector<double> shape_dofs(n_shape_dofs);
    OptimizationRun::step_counter = 0;
    for(unsigned int i = 0; i < n_shape_dofs; i++) {
      shape_dofs[i] = GlobalSpaceTransformation->get_free_dof(i);
      if(GlobalParams.MPI_Rank == 0) {
        std::cout « "Shape dof " « i « ": " « shape_dofs[i] « std::endl;
      }
    }
    dealii::SolverControl sc(GlobalParams.optimization_n_shape_steps,
      GlobalParams.optimization_residual_tolerance, true, true);
```

```
  dealii::SolverBFGS<dealii::Vector<double>> solver(sc);
  try{
    solver.solve(function_pointer, shape_dofs);
  } catch(dealii::StandardExceptions::ExcMessage & e) {
    print_info("OptimizationRun::run", "Optimization terminated.");
  }

  GlobalTimerManager.write_output();
  OptimizationRun::mainProblem->output_results();
  print_info("OptimizationRun::run", "End", LoggingLevel::PRODUCTION_ONE);
}
```

5.53.3.3 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.

```
                                                                                {
  std::string msg = "( ";
  for(unsigned int i = 0; i < in_shape_dofs.size(); i++) {
    msg += std::to_string(in_shape_dofs[i]);
    if(i != in_shape_dofs.size() - 1) {
      msg += ", ";
    } else {
      msg += ")";
    }
  }
  print_info("OptimizationRun::set_shape_dofs", msg);

  for(unsigned int i = 0; i < in_shape_dofs.size(); i++) {
    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

5.54 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**

5.54.1 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.

5.54.2 Member Function Documentation

5.54.2.1 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.

```
                                                        {
    return output_folder_path + "/" + filename;
}
```

Referenced by get_numbered_filename(), and write_run_description().

5.54.2.2 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.

```
       {
    return get_full_filename(filename) + std::to_string(number) + '.' + extension;
}
```

References get_full_filename().

5.54.2.3 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.

```
                                        {
    log_stream « in_line « std::endl;
}
```

5.54.2.4 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.

```
                                                             {
    std::string filename = get_full_filename("run_description.txt");
    std::ofstream out(filename);
    out « "Number of processes: \t" « GlobalParams.NumberProcesses « std::endl;
    out « "Sweeping level: " « GlobalParams.Sweeping_Level « std::endl;
    out « "Truncation Method: " « ((GlobalParams.BoundaryCondition == BoundaryConditionType::HSIE)?
    "HSIE" : "PML") « std::endl;
    out « "Signal input method: " « (GlobalParams.use_tapered_input_signal ? "Taper" : "Dirichlet") «
    std::endl;
    out « "Set 0 on input interface: " « (GlobalParams.prescribe_0_on_input_side ? "true" : "false") «
    std::endl;
    out « "Use predefined shape: " « (GlobalParams.Use_Predefined_Shape ? "true" : "false") « std::endl;
    if(GlobalParams.Use_Predefined_Shape) {
        out « "Predefined Shape Number: " « GlobalParams.Number_of_Predefined_Shape « std::endl;
    }
    out « "Block Counts: [" « GlobalParams.Blocks_in_x_direction « "x" «
    GlobalParams.Blocks_in_y_direction « "x" « GlobalParams.Blocks_in_z_direction « "]" « std::endl;
    out « "Global cell count x: " « GlobalParams.Blocks_in_x_direction * GlobalParams.Cells_in_x «
    std::endl;
    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

5.55 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**

5.55.1 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.

5.55.2 Member Function Documentation

5.55.2.1 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 }
```

5.55.2.2 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().

5.55.2.3 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.

```
                                                        {
       if(in_string.size() < 4) {
           return false;
       }
       if (in_string.find('=') == std::string::npos) {
           return false;
       }
       std::vector<std::string> blocks = split(in_string, ";");
       for(unsigned int i = 0; i < blocks.size(); i++) {
           std::vector<std::string> line_split = split(blocks[i], "=");
           if(line_split.size() != 2) {
               return false;
           }
       }
       return true;
   }
```

Referenced by read().

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

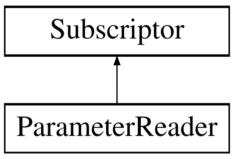
- Code/Helpers/ParameterOverride.h
- Code/Helpers/ParameterOverride.cpp

5.56 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.

5.56.1 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.

Author

Pascal Kraft

Date

23.11.2015

Definition at line 40 of file ParameterReader.h.

5.56.2 Constructor & Destructor Documentation

5.56.2.1 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 { }
```

5.56.3 Member Function Documentation

5.56.3.1 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.");
17         run_prm.declare_entry("kappa angle" , "1.0", Patterns::Double(), "Phase of the complex value kappa with norm 1 that is used in HSIEs.");
18         run_prm.declare_entry("processes in x" , "1", Patterns::Integer(), "Number of processes in x-direction.");
19         run_prm.declare_entry("processes in y" , "1", Patterns::Integer(), "Number of processes in y-direction.");
20         run_prm.declare_entry("processes in z" , "1", Patterns::Integer(), "Number of processes in z-direction.");
21         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 sweeping.");
22         run_prm.declare_entry("cell count x" , "20", Patterns::Integer(), "Number of cells a single process has in x-direction.");
23         run_prm.declare_entry("cell count y" , "20", Patterns::Integer(), "Number of cells a single process has in y-direction.");
24         run_prm.declare_entry("cell count z" , "20", Patterns::Integer(), "Number of cells a single process has in z-direction.");
25         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.");
26         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.");
27         run_prm.declare_entry("solver type", "GMRES", Patterns::Selection("GMRES|MINRES|TFQMR|BICGS|CG|PCONLY"), "Choose the itterative solver to use.");
28     }
29     run_prm.leave_subsection();
30 
31     case_prm.enter_subsection("Case parameters");
32     {
33         case_prm.declare_entry("source type", "0", Patterns::Integer(), "PointSourceField is 0: empty, 1: cos()cos(), 2: Hertz Dipole, 3: Waveguide");
34         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

5.57 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

5.57.1 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.

Author

: Pascal Kraft

Date

: 28.11.2016

Definition at line 29 of file Parameters.h.

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

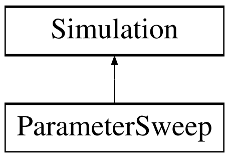
- Code/Helpers/Parameters.h
- Code/Helpers/Parameters.cpp

5.58 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.

5.58.1 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.

5.58.2 Member Function Documentation

5.58.2.1 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.

```
                              {
  print_info("ParameterSweep::prepare", "Start", LoggingLevel::DEBUG_ONE);
  if(GlobalParams.Point_Source_Type == 0) {
    rmProblem = new RectangularMode();
  }
  print_info("ParameterSweep::prepare", "End", LoggingLevel::DEBUG_ONE);
}
```

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

- Code/Runners/ParameterSweep.h
- Code/Runners/ParameterSweep.cpp

5.59 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**

5.59.1 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.

5.59.2 Member Function Documentation

5.59.2.1 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;
36          if(std::abs(in_p[1] - default_y_range.second) < FLOATING_PRECISION && transform_coordinate[3])
        ret[1] += extension_factor;
37      }
38      if(outward_direction != 2) {
39          if(std::abs(in_p[2] - default_z_range.first ) < FLOATING_PRECISION && transform_coordinate[4])
        ret[2] -= extension_factor;
40          if(std::abs(in_p[2] - default_z_range.second) < FLOATING_PRECISION && transform_coordinate[5])
        ret[2] += extension_factor;
41      }
42      return ret;
43 }
```

5.59.2.2 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.

```
                                                                     {
       Position ret = in_p;
       if(in_p[0] < default_x_range.first)  ret[0] = default_x_range.first;
       if(in_p[0] > default_x_range.second) ret[0] = default_x_range.second;
       if(in_p[1] < default_y_range.first)  ret[1] = default_y_range.first;
       if(in_p[1] > default_y_range.second) ret[1] = default_y_range.second;
       if(in_p[2] < default_z_range.first)  ret[2] = default_z_range.first;
       if(in_p[2] > default_z_range.second) ret[2] = default_z_range.second;
       return ret;
   }
```

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

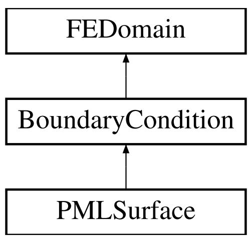
- Code/BoundaryCondition/PMLMeshTransformation.h
- Code/BoundaryCondition/PMLMeshTransformation.cpp

5.60 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 \epsilon_p for a given position.
- auto get_pml_tensor_mu (Position in_p) -> dealii::Tensor< 2, 3, ComplexNumber >

  Get the PML material tensor \mu_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, Boundary↩ Id 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**

5.60.1 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 \cref{subsec:PML}.

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 \cref{subsec:PEC} 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.

5.60.2 Member Function Documentation

5.60.2.1 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.

```
                                                                      {
    unsigned int ret = 0;
    for(auto it = triangulation.begin(); it!= triangulation.end(); it++) {
      if(it->at_boundary()) {
        for(unsigned int i = 0; i < 6; i++) {
          if(it->face(i)->boundary_id() == in_boundary_id) {
            ret++;
          }
        }
      }
    }
    return ret;
}
```

5.60.2.2 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.

```
                                                                       {
  x_range.first = 100000.0;
  y_range.first = 100000.0;
  z_range.first = 100000.0;
  x_range.second = -100000.0;
  y_range.second = -100000.0;
  z_range.second = -100000.0;
  for(auto it = in_tria->begin(); it != in_tria->end(); it++) {
    for(unsigned int i = 0; i < 6; i++) {
      if(it->face(i)->at_boundary()) {
        Position p = it->face(i)->center();
        if(p[0] < x_range.first) {
          x_range.first = p[0];
        }
        if(p[0] > x_range.second) {
          x_range.second = p[0];
        }
        if(p[1] < y_range.first) {
          y_range.first = p[1];
        }
        if(p[1] > y_range.second) {
          y_range.second = p[1];
        }
        if(p[2] < z_range.first) {
          z_range.first = p[2];
        }
        if(p[2] > z_range.second) {
          z_range.second = p[2];
        }
      }
    }
  }
}
```

5.60.2.3 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.

```
                                                       {
  return dof_counter;
}
```

5.60.2.4 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.

```
                                                      {
  IndexSet non_owned_dofs = compute_non_owned_dofs();
  return dof_counter - non_owned_dofs.n_elements();
}
```

References compute_non_owned_dofs().

5.60.2.5 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.

```
                                                    {
  IndexSet non_owned_dofs(dof_counter);
  std::vector<unsigned int> non_locally_owned_surfaces;
  for(auto surf : adjacent_boundaries) {
    if(!are_edge_dofs_owned[surf]) {
      non_locally_owned_surfaces.push_back(surf);
    }
  }
  non_locally_owned_surfaces.push_back(inner_boundary_id);

  std::vector<unsigned int> local_indices(fe_nedelec.dofs_per_face);
  // 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.
  for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
    for(unsigned int face = 0; face < 6; face++) {
      if(it->face(face)->at_boundary()) {
        for(auto surf: non_locally_owned_surfaces) {
          if(it->face(face)->boundary_id() == surf) {
            it->face(face)->get_dof_indices(local_indices);
            for(unsigned int i = 0; i < fe_nedelec.dofs_per_face; i++) {
              non_owned_dofs.add_index(local_indices[i]);
            }
          }
        }
      }
    }
  }
  return non_owned_dofs;
}
```

Referenced by compute_n_locally_owned_dofs(), and determine_non_owned_dofs().

5.60.2.6 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.

```
                                                            {
  if(Geometry.levels[level].surface_type[in_bid] != SurfaceType::ABC_SURFACE) {
    return false;
  }
  if(b_id == 4 || b_id == 5) {
    return true;
  }
  if(b_id == 0 || b_id == 1) {
    return false;
  }
  if(b_id == 2 || b_id == 3) {
    return in_bid < b_id;
  }
  return false;
}
```

5.60.2.7 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.

```
                              {
  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().

5.60.2.8 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.

```
{
  std::vector<unsigned int> local_indices(fe_nedelec.dofs_per_cell);
  for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
    it->get_dof_indices(local_indices);
    local_indices = transform_local_to_global_dofs(local_indices);
    in_constraints->add_entries_local_to_global(local_indices, *in_dsp);
  }
}
```

References FEDomain::transform_local_to_global_dofs().

5.60.2.9 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.

```
                                                 {
  for(unsigned int surf = 0; surf < 6; surf++) {
    if(surf != b_id && !are_opposing_sites(surf, b_id)) {
      if(!are_edge_dofs_owned[surf] && Geometry.levels[level].surface_type[surf] !=
 SurfaceType::NEIGHBOR_SURFACE) {
        DofIndexVector dofs_in_global_numbering =
 Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id);
        std::vector<InterfaceDofData> local_interface_data = get_dof_association_by_boundary_id(surf);
        DofIndexVector dofs_in_local_numbering(local_interface_data.size());
        for(unsigned int i = 0; i < local_interface_data.size(); i++) {
          dofs_in_local_numbering[i] = local_interface_data[i].index;
        }
        set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
      }
    }
  }
  // Do the same for the inner interface
  std::vector<InterfaceDofData> global_interface_data =
 Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
  std::vector<InterfaceDofData> local_interface_data =
 get_dof_association_by_boundary_id(inner_boundary_id);
  DofIndexVector dofs_in_local_numbering(local_interface_data.size());
  DofIndexVector dofs_in_global_numbering(local_interface_data.size());

  for(unsigned int i = 0; i < local_interface_data.size(); i++) {
    dofs_in_local_numbering[i] = local_interface_data[i].index;
    dofs_in_global_numbering[i] =
 Geometry.levels[level].inner_domain->global_index_mapping[global_interface_data[i].index];
  }
  set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
}
```

5.60.2.10 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.

```
                                                     {
  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;
  if(own.size() != dofs.size()) {
    std::cout « "Size mismatch in finish initialization: " « own.size() « " != " « dofs.size() «
   std::endl;
  }
  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);
}
```

5.60.2.11 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.

```
                                                                                              {
  double min_z_before = min_z_center_in_triangulation(*in_tria);
  GridTools::transform(invert_z, *in_tria);
  double min_z_after = min_z_center_in_triangulation(*in_tria);
  Tensor<1,3> shift;
  shift[0] = 0;
  shift[1] = 0;
  shift[2] = min_z_before - min_z_after;
  GridTools::shift(shift, *in_tria);
}
```

5.60.2.12 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 \cref{subsec:PML}, we can ramp up the value of sigma as we approach the outer boundary to reduce the effect of reflections by a profile like \cref{eqn:PMLIncrease}. 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.

```
                                                                        {
  std::array<double, 3> ret;
  for(unsigned int i = 0; i < 3; i++) {
    std::pair<double, double> range;
    switch (i)
    {
      case 0:
        range = Geometry.local_x_range;
        break;
      case 1:
        range = Geometry.local_y_range;
        break;
      case 2:
        range = Geometry.local_z_range;
        break;
      default:
        break;
    }
    ret[i] = 0;
    if(in_p[i] < lower_pml_ranges[i].first) {
      ret[i] = std::abs(in_p[i] - lower_pml_ranges[i].first) / effective_pml_thickness;
    }
    if(in_p[i] > upper_pml_ranges[i].first) {
      ret[i] = std::abs(in_p[i] - upper_pml_ranges[i].first) / effective_pml_thickness;
    }
  }
  return ret;
}
```

Referenced by get_pml_tensor().

5.60.2.13 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.

```
                                                        {
    return get_dof_association_by_boundary_id(inner_boundary_id);
}
```

References get_dof_association_by_boundary_id().

5.60.2.14 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.

```
                                                                                                      {
  return dof_associations[in_bid];
}
```

Referenced by get_dof_association().

5.60.2.15 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.

```
                                                                          {
  dealii::Tensor<2,3,ComplexNumber> ret;
  const std::array<double, 3> fractions = fraction_of_pml_direction(in_p);
  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().

5.60.2.16 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 \epsilon_p for a given position.

This is \epsilon_p in \cref{sec:PML}.

Parameters

in_p	The location to compute the material tensor at

Returns

dealii::Tensor<2,3,ComplexNumber> The material tensor \epsilon_p for a UPML medium at a given location.

Definition at line 357 of file PMLSurface.cpp.

```
                                                                                  {
    dealii::Tensor<2,3,ComplexNumber> ret = get_pml_tensor(in_p);
    ret *= Geometry.get_epsilon_for_point(in_p);
    return ret;
}
```

References get_pml_tensor().

Referenced by fill_matrix(), and output_results().

5.60.2.17 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 \mu_p for a given position.

This is \mu_p in \cref{sec:PML}.

Parameters

in_p	The location to compute the material tensor at

Returns

dealii::Tensor<2,3,ComplexNumber> The material tensor \mu_p for a UPML medium at a given location.

Definition at line 363 of file PMLSurface.cpp.

```
                                                                                        {
       dealii::Tensor<2,3,ComplexNumber> ret = get_pml_tensor(in_p);
       return ret;
 }
```

References get_pml_tensor().

5.60.2.18 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.

```
                             {
       dof_handler.reinit(triangulation);
       dof_handler.distribute_dofs(fe_nedelec);
       dof_counter = dof_handler.n_dofs();
 }
```

Referenced by initialize().

5.60.2.19 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.

```
                                {
    prepare_mesh();
    init_fe();
    prepare_dof_associations();
 }
```

References init_fe(), prepare_dof_associations(), and prepare_mesh().

5.60.2.20 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.

5.60.2.21 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;
}
```

5.60.2.22 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.

```
                                                                                 {
  switch (in_bid)
  {
    case 0:
      if(std::abs(in_p[0] - x_range.first) < FLOATING_PRECISION) return true;
      break;
    case 1:
      if(std::abs(in_p[0] - x_range.second) < FLOATING_PRECISION) return true;
      break;
    case 2:
      if(std::abs(in_p[1] - y_range.first) < FLOATING_PRECISION) return true;
      break;
    case 3:
      if(std::abs(in_p[1] - y_range.second) < FLOATING_PRECISION) return true;
      break;
    case 4:
      if(std::abs(in_p[2] - z_range.first) < FLOATING_PRECISION) return true;
      break;
    case 5:
      if(std::abs(in_p[2] - z_range.second) < FLOATING_PRECISION) return true;
      break;
  }
  return false;
}
```

5.60.2.23 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.

```
{
  if(std::abs(in_p[b_id / 2] - surface_coordinate) < FLOATING_PRECISION) {

    switch(b_id / 2) {
      case 0:
        return false;
        break;
      case 1:
        if((in_bid / 2) == 0) {
          if(in_p[0] < Geometry.local_x_range.first && in_bid == 0) {

            return true;
          }
          if(in_p[0] > Geometry.local_x_range.second && in_bid == 1) {
            return true;
          }
          return false;
        } else {
          return false;
        }
        break;
      case 2:
        if(in_bid == 3) {
          return in_p[1] > Geometry.local_y_range.second;
        }
        if(in_bid == 2) {
          return in_p[1] < Geometry.local_y_range.first;
        }
        if(in_bid == 1) {
          bool not_y = in_p[1] <= Geometry.local_y_range.second && in_p[1] >=
   Geometry.local_y_range.first;
          if(not_y) {
            return in_p[0] > Geometry.local_x_range.second;
          } else {
            return false;
          }
        }
        if(in_bid == 0) {
          bool not_y = in_p[1] <= Geometry.local_y_range.second && in_p[1] >=
   Geometry.local_y_range.first;
          if(not_y) {
            return in_p[0] < Geometry.local_x_range.first;
          } else {
            return false;
          }
        }
        break;
    }
    return false;
  } else {
    return b_id == in_bid;
  }
}
```

5.60.2.24 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 \cref{subsec:PML}, 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.

```
                                                {
  IndexSet global_indices = IndexSet(Geometry.levels[level].n_total_level_dofs);
  global_indices.add_range(0, Geometry.levels[level].n_total_level_dofs);
  Constraints ret(global_indices);
  std::vector<InterfaceDofData> dofs = get_dof_association_by_boundary_id(outer_boundary_id);
  for(auto dof : dofs) {
        const unsigned int local_index = dof.index;
        const unsigned int global_index = global_index_mapping[local_index];
        ret.add_line(global_index);
        ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
    }
  return ret;
}
```

5.60.2.25 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.

```
            {
  if(b_id == 4 || b_id == 5) {
    bool generate_this_part = Geometry.levels[level].is_surface_truncated[first_bid] &&
   Geometry.levels[level].is_surface_truncated[second_bid];
    if(generate_this_part) {
      // Do the generation.
   GridGenerator::subdivided_hyper_cube(*tria,GlobalParams.PML_N_Layers,0,GlobalParams.PML_thickness);
      dealii::Tensor<1,3> shift;
      bool lower_x = first_bid == 0 || second_bid == 0;
      bool lower_y = first_bid == 2 || second_bid == 2;
      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;
}
```

5.60.2.26 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.

```
                                                                    {
  // This line checks if the domain even exists
  bool domain_exists = Geometry.levels[level].is_surface_truncated[other_bid];
  // 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
  bool is_owned = false;
  if(b_id == 4 || b_id == 5) {
    is_owned = true;
  }
  if(b_id == 2 || b_id == 3) {
    is_owned = (other_bid == 0 || other_bid == 1);
  }
  if(domain_exists && is_owned) {
    std::vector<unsigned int> subdivisions(3);
    Position p1, p2;
    if(b_id / 2 != 0 && other_bid /2 != 0) {
      subdivisions[0] = GlobalParams.Cells_in_x;
      p1[0] = Geometry.local_x_range.first;
      p2[0] = Geometry.local_x_range.second;
    } 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;
}
```

5.60.2.27 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.

```
                                   {
  return triangulation.n_active_cells();
}
```

Referenced by output_results().

5.60.2.28 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.

```
{
  dealii::DataOut<3> data_out;
  data_out.attach_dof_handler(dof_handler);

  dealii::Vector<ComplexNumber> zero = dealii::Vector<ComplexNumber>(in_data.size());
  for(unsigned int i = 0; i < in_data.size(); i++) {
    zero[i] = 0;
  }

  const unsigned int n_cells = dof_handler.get_triangulation().n_cells();
  dealii::Vector<double> eps_abs(n_cells);
  unsigned int counter = 0;
  for(auto it = dof_handler.begin(); it != dof_handler.end(); it++) {
    Position p = it->center();
    MaterialTensor epsilon = get_pml_tensor_epsilon(p);
    eps_abs[counter] = epsilon.norm();
    counter++;
  }

  data_out.add_data_vector(in_data, "Solution");
  data_out.add_data_vector(eps_abs, "Epsilon");
   dealii::Vector<double> index_x(n_cells), index_y(n_cells), index_z(n_cells);
  for(unsigned int i = 0; i < n_cells; i++) {
    index_x[i] = GlobalParams.Index_in_x_direction;
    index_y[i] = GlobalParams.Index_in_y_direction;
    index_z[i] = GlobalParams.Index_in_z_direction;
  }
  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().

5.60.2.29 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.

```
{
  std::array<unsigned int, 6> countrers;
  for(unsigned int i= 0; i < 6; i++) {
    countrers[i] = 0;
  }
  // first set all to outer_boundary_id
  for(auto it = triangulation.begin(); it != triangulation.end(); it++) {
    for(unsigned int face = 0; face < 6; face ++) {
      if(it->face(face)->at_boundary()) {
        it->face(face)->set_all_boundary_ids(outer_boundary_id);
        countrers[outer_boundary_id]++;
      }
    }
  }
  // then locate all the faces connecting to the inner domain
  for(auto it = triangulation.begin(); it != triangulation.end(); it++) {
    for(unsigned int face = 0; face < 6; face ++) {
      if(it->face(face)->at_boundary()) {
        Position p = it->face(face)->center();
        // Have to use outer_boundary_id here because direction 4 of the pml (-z) is at the boundary 5
   of the inner domain (+z)
        bool is_located_properly = std::abs(p[b_id/2] -
   get_surface_coordinate_for_bid(outer_boundary_id)) < FLOATING_PRECISION;
        if((b_id / 2) != 0) {
          is_located_properly &= p[0] > Geometry.local_x_range.first + FLOATING_PRECISION;
          is_located_properly &= p[0] < Geometry.local_x_range.second - FLOATING_PRECISION;
        }
        if((b_id / 2) != 1) {
          is_located_properly &= p[1] > Geometry.local_y_range.first + FLOATING_PRECISION;
          is_located_properly &= p[1] < Geometry.local_y_range.second - FLOATING_PRECISION;
        }
        if((b_id / 2) != 2) {
          is_located_properly &= p[2] > Geometry.local_z_range.first + FLOATING_PRECISION;
          is_located_properly &= p[2] < Geometry.local_z_range.second - FLOATING_PRECISION;
        }
        if(is_located_properly) {
          it->face(face)->set_all_boundary_ids(inner_boundary_id);
          countrers[inner_boundary_id]++;
        }
      }
    }
  }
  // then check all of the other boundary ids.
  for(auto it = triangulation.begin(); it != triangulation.end(); it++) {
    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;
}
```

5.60.2.30 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.

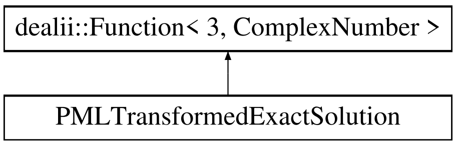
After this is called, we can retrieve dofs by boundary id.

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

- Code/BoundaryCondition/PMLSurface.h
- Code/BoundaryCondition/PMLSurface.cpp

5.61 PMLTransformedExactSolution Class Reference

Inheritance diagram for PMLTransformedExactSolution:


dealii::Function< 3, ComplexNumber >
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

5.61.1 Detailed Description

Definition at line 12 of file PMLTransformedExactSolution.h.

5.61.2 Member Function Documentation

5.61.2.1 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.

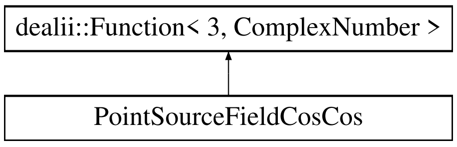
```
                                                                                              {
    dealii::Tensor<1, 3, ComplexNumber> ret;
    /**
    NumericVectorLocal curls = base_solution->curl(in_p);
    double scaling_factor = compute_scaling_factor(in_p);
    for(unsigned int i = 0; i < 3; i++) {
      ret[i] *= scaling_factor;
    }
    **/
    return ret;
  }
```

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

- Code/Solutions/PMLTransformedExactSolution.h
- Code/Solutions/PMLTransformedExactSolution.cpp

5.62 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)

5.62.1 Detailed Description

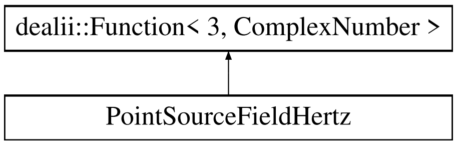
Definition at line 30 of file PointSourceField.h.

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

- Code/Helpers/PointSourceField.h
- Code/Helpers/PointSourceField.cpp

5.63 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

5.63.1 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

5.64 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**

5.64.1 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:

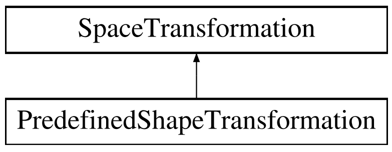
- Code/Helpers/PointVal.h
- Code/Helpers/PointVal.cpp

5.65 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.

5.65.1 Detailed Description

This class is used to describe the hump examples.

Definition at line 18 of file PredefinedShapeTransformation.h.

5.65.2 Member Function Documentation

5.65.2.1 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 appropriate 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.

```
                                                              {
  print_info("PredefinedShapeTransformation::estimate_and_initialize", "Start");
  Sector<2> the_first(true, false, GlobalParams.sd.z[0], GlobalParams.sd.z[1]);
  the_first.set_properties_force(GlobalParams.sd.m[0], GlobalParams.sd.m[1],
                                 GlobalParams.sd.v[0], GlobalParams.sd.v[1]);
  case_sectors.push_back(the_first);
  for (int i = 1; i < GlobalParams.sd.Sectors - 2; i++) {
      Sector<2> intermediate(false, false, GlobalParams.sd.z[i], GlobalParams.sd.z[i + 1]);
      intermediate.set_properties_force(
          GlobalParams.sd.m[i], GlobalParams.sd.m[i + 1], GlobalParams.sd.v[i],
          GlobalParams.sd.v[i + 1]);
      case_sectors.push_back(intermediate);
  }
  Sector<2> the_last(false, true,
                     GlobalParams.sd.z[GlobalParams.sd.Sectors - 2],
                     GlobalParams.sd.z[GlobalParams.sd.Sectors - 1]);
  the_last.set_properties_force(
      GlobalParams.sd.m[GlobalParams.sd.Sectors - 2],
      GlobalParams.sd.m[GlobalParams.sd.Sectors - 1],
      GlobalParams.sd.v[GlobalParams.sd.Sectors - 2],
      GlobalParams.sd.v[GlobalParams.sd.Sectors - 1]);
  case_sectors.push_back(the_last);
  if(GlobalParams.MPI_Rank == 0) {
    for (unsigned int i = 0; i < case_sectors.size(); i++) {
      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)) +
    ")";
      print_info("PredefinedShapeTransformation::estimate_and_initialize", msg_lower);
    }
    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)) + ")";

  }
  print_info("PredefinedShapeTransformation::estimate_and_initialize", "End");
}
```

5.65.2.2 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.

```
                                                                       {
  Tensor<2,3,double> ret = I;
  ret[1][2] = - get_v(in_p[2]);
  return ret;
}
```

References get_v().

Referenced by get_J_inverse(), and get_Space_Transformation_Tensor().

5.65.2.3 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.

```
                                                                        {
  Tensor<2,3,double> ret = get_J(in_p);
  return invert(ret);
}
```

References get_J().

5.65.2.4 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> 33 real valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 100 of file PredefinedShapeTransformation.cpp.

```
                                                                          {
  Tensor<2, 3, double> J_loc = get_J(position);
  Tensor<2, 3, double> ret;
  ret[0][0] = 1;
  ret[1][1] = 1;
  ret[2][2] = 1;
  return (J_loc * ret * transpose(J_loc)) / determinant(J_loc);
}
```

References get_J().

Referenced by get_Tensor().

5.65.2.5 get_Tensor()

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

Get the transformation tensor at a given location.

Returns

Tensor<2, 3, ComplexNumber> 33 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 47 of file PredefinedShapeTransformation.cpp.

```
                                                                  {
    return get_Space_Transformation_Tensor(position);
}
```

References get_Space_Transformation_Tensor().

5.65.2.6 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.

```
                                                                              {
    Position ret;
    std::pair<int, double> sec = Z_to_Sector_and_local_z(coord[2]);
    double m = case_sectors[sec.first].get_m(sec.second);
    ret[0] = coord[0];
    ret[1] = coord[1] + m;
    ret[2] = coord[2];
    return ret;
}
```

References case_sectors, and SpaceTransformation::Z_to_Sector_and_local_z().

5.65.2.7 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.

```
                                                                          {
    Position ret;
    std::pair<int, double> sec = Z_to_Sector_and_local_z(coord[2]);
    double m = case_sectors[sec.first].get_m(sec.second);
    ret[0] = coord[0];
    ret[1] = coord[1] - m;
    ret[2] = coord[2];
    return ret;
}
```

References case_sectors, and SpaceTransformation::Z_to_Sector_and_local_z().

5.65.3 Member Data Documentation

5.65.3.1 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

5.66 RayAngelingData Struct Reference

Public Attributes

- bool **is_x_angled** = false
- bool **is_y_angled** = false
- Position2D **position_of_base_point**

5.66.1 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

5.67 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**

5.67.1 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.

5.67.2 Member Function Documentation

5.67.2.1 solve()

```
void RectangularMode::solve ( )
```

eigensolver.solve(stiffness_matrix, mass_matrix, eigenvalues, eigenfunctions, n_eigenfunctions);

Definition at line 281 of file RectangularMode.cpp.

```
                           {
print_info("RectangularProblem::solve", "Start");
dealii::SolverControl                   solver_control(n_dofs_total, 1e-6);
// dealii::SLEPcWrappers::SolverKrylovSchur eigensolver(solver_control);
IndexSet own_dofs(n_dofs_total);
own_dofs.add_range(0, n_dofs_total);
eigenfunctions.resize(n_eigenfunctions);
for (unsigned int i = 0; i < n_eigenfunctions; ++i)
  eigenfunctions[i].reinit(own_dofs, MPI_COMM_SELF);
eigenvalues.resize(n_eigenfunctions);
// eigensolver.set_which_eigenpairs(EPS_SMALLEST_MAGNITUDE);
// eigensolver.set_problem_type(EPS_GNHEP);
print_info("RectangularProblem::solve", "Starting solution for a system with " +
 std::to_string(n_dofs_total) + " degrees of freedom.");
/**
```

```
  eigensolver.solve(stiffness_matrix,
                    mass_matrix,
                    eigenvalues,
                    eigenfunctions,
                    n_eigenfunctions);
                    **/
  for(unsigned int i =0 ; i < n_eigenfunctions; i++) {
    // constraints.distribute(eigenfunctions[0]);
    eigenfunctions[i] /= eigenfunctions[i].linfty_norm();
  }
  print_info("RectangularProblem::solve", "End");
}
```

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

- Code/ModalComputations/RectangularMode.h
- Code/ModalComputations/RectangularMode.cpp

5.68 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** ()

5.68.1 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

5.69 SampleShellPC Struct Reference

Public Attributes

- NonLocalProblem ∗ **parent**

5.69.1 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

5.70 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**

5.70.1 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.

Author

Pascal Kraft

Date

17.12.2015

Definition at line 25 of file Sector.h.

5.70.2 Constructor & Destructor Documentation

5.70.2.1 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;
}
```

5.70.3 Member Function Documentation

5.70.3.1 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.

```
                                                                          {
  if (i > 0 && i < NDofs) {
    if (z < 0.0) z = 0.0;
    if (z > 1.0) z = 1.0;
    if (zero_derivative[i]) {
      return InterpolationPolynomialZeroDerivative(z, dofs_l[i], dofs_r[i]);
    } else {
      return InterpolationPolynomial(z, dofs_l[i], dofs_r[i],
                                     dofs_l[derivative[i]],
                                     dofs_r[derivative[i]]);
    }
  } else {
    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);
    return 0;
  }
}
```

5.70.3.2 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.

```
                                                      {
  if (z < 0.0) z = 0.0;
  if (z > 1.0) z = 1.0;
  if (Dofs_Per_Sector == 2) {
    return InterpolationPolynomial(z, dofs_l[0], dofs_r[0], dofs_l[1],
                                   dofs_r[1]);
  } else {
    return InterpolationPolynomial(z, dofs_l[1], dofs_r[1], dofs_l[2],
                                   dofs_r[2]);
  }
}
```

5.70.3.3 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.

```
                                                    {
  if (z < 0.0) z = 0.0;
  if (z > 1.0) z = 1.0;
  if (Dofs_Per_Sector < 3) {
    print_info("Sector<Dofs_Per_Sector>::get_r", "Error in Sector: Access to radius dof without
   existence.", LoggingLevel::PRODUCTION_ALL);
    return 0;
  }
  return InterpolationPolynomialZeroDerivative(z, dofs_l[0], dofs_r[0]);
}
```

5.70.3.4 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.

```
                                                    {
  if (z < 0.0) z = 0.0;
  if (z > 1.0) z = 1.0;
  if (Dofs_Per_Sector == 2) {
    return InterpolationPolynomialZeroDerivative(z, dofs_l[1], dofs_r[1]);
  } else {
    return InterpolationPolynomialZeroDerivative(z, dofs_l[2], dofs_r[2]);
  }
}
```

5.70.3.5 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.

```
                                                          {
  return LowestDof;
}
```

5.70.3.6 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.

```
                                                             {
  return NActiveCells;
}
```

5.70.3.7 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.

```
                                                      {
  return NDofs;
}
```

5.70.3.8 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.

```
                                                                      {
  return NInternalBoundaryDofs;
}
```

5.70.3.9 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.

```
                                                    {
  return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
              z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
}
```

5.70.3.10 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.

```
                                                    {
  return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
              z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
}
```

5.70.3.11 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.

```
                                                  {
  return 0.0;
}
```

5.70.3.12 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.

```
                                                                            {
  print_info("Sector<Dofs_Per_Sector>::set_properties", "The code does not work for this number of dofs
   per Sector.", LoggingLevel::PRODUCTION_ALL);
  return;
}
```

5.70.3.13 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.

```
                                                                    {
  LowestDof = inLowestDOF;
}
```

5.70.3.14 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.

```
                                          {
  NActiveCells = inNumberOfActiveCells;
}
```

5.70.3.15 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.

```
                                                                    {
  NDofs = inNumberOfDOFs;
}
```

5.70.3.16 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;
}
```

5.70.3.17 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;
}
```

5.70.4 Member Data Documentation

5.70.4.1 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

5.71 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**

5.71.1 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

5.72 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**

5.72.1 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.

5.72.2 Constructor & Destructor Documentation

5.72.2.1 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 }
```

5.72.3 Member Function Documentation

5.72.3.1 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.

```
                                                                       {
    return in_n_sectors+3;
}
```

Referenced by compute_n_free_dofs().

5.72.3.2 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.

```
                                                                           {
   int ret = ShapeFunction::compute_n_dofs(in_n_sectors);
    ret -= 5;
    if(ret < 0) {
        std::cout « "The shape function is underdetermined. Add more sectors." « std::endl;
    }
    return std::abs(ret);
}
```

References compute_n_dofs().

5.72.3.3 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.

```
                                                    {
    if(z <= z_min) {
        return dof_values[0];
    }
    if(z > z_max) {
        return evaluate_at(2*z_max - z);
    }
    double ret = dof_values[0];
    double z_temp = z_min;
    unsigned int index = 1;
    while(z_temp + sector_length < z+FLOATING_PRECISION) {
        ret += 0.5 * sector_length * (dof_values[index + 1] - dof_values[index]);
        ret += sector_length * dof_values[index];
        index++;
        z_temp += sector_length;
    }
    double delta_z = z - z_temp;
    if(std::abs(delta_z) <= FLOATING_PRECISION) {
        return ret;
    }
    ret += 0.5 * delta_z * (dof_values[index + 1] - dof_values[index]) * (delta_z/sector_length);
    ret += delta_z * dof_values[index];
    return ret;
}
```

Referenced by print(), and update_constrained_values().

5.72.3.4 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.

```
                                                            {
      if(z <= z_min) {
          return dof_values[1];
      }
      if(z > z_max) {
          return - evaluate_derivative_at(2*z_max - z);
      }
      unsigned int index = 1;
      double z_temp = z_min;
      while(z_temp + sector_length < z) {
          index ++;
          z_temp += sector_length;
      }
      double delta_z = z - z_temp;
      if(std::abs(delta_z) < FLOATING_PRECISION) {
          return dof_values[index];
      } else {
          return dof_values[index] + (dof_values[index + 1] - dof_values[index]) * ( delta_z /
      sector_length );
      }
  }
```

5.72.3.5 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.

```
                                                            {
    return dof_values[index];
 }
```

Referenced by WaveguideTransformation::get_dof_values().

5.72.3.6 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.

```
                                                                  {
    return dof_values[index + 2];
}
```

5.72.3.7 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.

```
                                            {
    return n_dofs;
}
```

Referenced by WaveguideTransformation::get_dof_values().

5.72.3.8 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.

```
                                                 {
    return n_free_dofs;
}
```

5.72.3.9 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.

```
                                                                                                        {
    f_0 = in_f_0;
    df_0 = in_df_0;
    f_1 = in_f_1;
    df_1 = in_df_1;
    update_constrained_values();
}
```

References update_constrained_values().

Referenced by WaveguideTransformation::estimate_and_initialize().

5.72.3.10 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.

```
                                                                          {
    if(index < n_free_dofs) {
        dof_values[index + 2] = value;
        update_constrained_values();
    } else {
        std::cout « "You tried to write to a constrained dof of a shape function." « std::endl;
    }
}
```

References update_constrained_values().

5.72.3.11 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.

```
                                                                      {
    if(in_dof_values.size() != n_free_dofs) {
        std::cout « "Provided wrong number of degrees of freedom." « std::endl;
    }
    for(unsigned int i = 0; i < in_dof_values.size(); i++) {
        dof_values[2 + i] = in_dof_values[i];
    }
    update_constrained_values();
}
```

References update_constrained_values().

5.72.3.12 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.

```
                                                     {
    dof_values[0] = f_0;
    dof_values[1] = df_0;
    dof_values[n_dofs-2] = df_1;
    dof_values[n_dofs-1] = f_1;
    double f_y_min2 = evaluate_at(z_max - sector_length - sector_length);
    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]));
}
```

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:

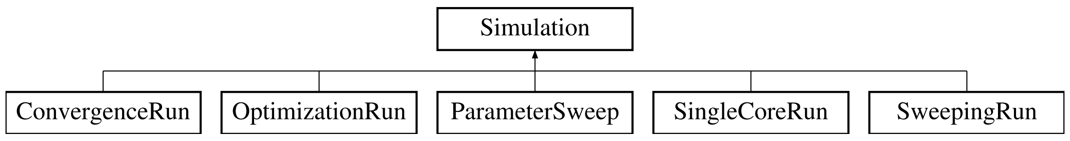
- Code/Optimization/ShapeFunction.h
- Code/Optimization/ShapeFunction.cpp

5.73 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.

5.73.1 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.

5.73.2 Member Function Documentation

5.73.2.1 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:

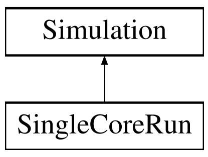
- Code/Runners/Simulation.h
- Code/Runners/Simulation.cpp

5.74 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.

5.74.1 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:

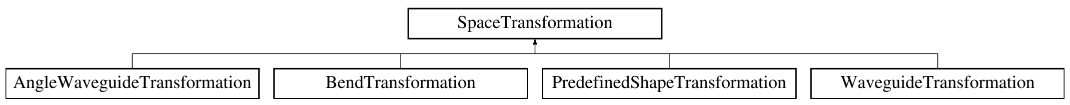
- Code/Runners/SingleCoreRun.h
- Code/Runners/SingleCoreRun.cpp

5.75 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

5.75.1 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.

Author

Pascal Kraft

Date

17.12.2015

Definition at line 35 of file SpaceTransformation.h.

5.75.2 Member Function Documentation

5.75.2.1 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 PredefinedShapeTransformatio

5.75.2.2 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.

```
                                  {
     return 1.0;
   }
```

5.75.2.3 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.

```
                                    {
    return 0;
  };
```

Referenced by get_inverse_Tensor_for_step(), and get_Tensor_for_step().

5.75.2.4 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.

```
                                                {
    Vector<double> ret;
    return ret;
  };
```

5.75.2.5 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.

```
{ return 0.0; };
```

5.75.2.6 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.
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.

```

                                    {
  double old_value = get_dof(dof);
  Tensor<2, 3, double> trafo1 = invert(get_Space_Transformation_Tensor(coordinate));

  set_free_dof(dof, old_value + step_width);
  Tensor<2, 3, double> trafo2 = invert(get_Space_Transformation_Tensor(coordinate));

  set_free_dof(dof, old_value);
  return trafo2 - trafo1;
}
```

References get_dof(), get_Space_Transformation_Tensor(), and set_free_dof().

5.75.2.7 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.

```
                                          {
    Tensor<2,3,double> ret;
    ret[0][0] = 1;
    ret[1][1] = 1;
    ret[2][2] = 1;
    return ret;
  }
```

5.75.2.8 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.

```
                                                {
    Tensor<2,3,double> ret;
    ret[0][0] = 1;
    ret[1][1] = 1;
    ret[2][2] = 1;
    return ret;
  }
```

5.75.2.9 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> 33 real valued tensor for a given locations.

Implemented in WaveguideTransformation, AngleWaveguideTransformation, BendTransformation, and PredefinedShapeTransformatio

Referenced by get_inverse_Tensor_for_step(), and get_Tensor_for_step().

5.75.2.10 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> 33 complex valued tensor for a given locations.

Implemented in WaveguideTransformation, AngleWaveguideTransformation, BendTransformation, and PredefinedShapeTransformatio

5.75.2.11 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.

```
                          {
  double old_value = get_dof(dof);
  Tensor<2, 3, double> trafo1 = get_Space_Transformation_Tensor(coordinate);

  set_free_dof(dof, old_value + step_width);
  Tensor<2, 3, double> trafo2 = get_Space_Transformation_Tensor(coordinate);

  set_free_dof(dof, old_value);
  return trafo2 - trafo1;
}
```

References get_dof(), get_Space_Transformation_Tensor(), and set_free_dof().

5.75.2.12 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 PredefinedShapeTransformatio

Referenced by operator()().

5.75.2.13 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.
```
                                         {
    return 0;
  }
```

5.75.2.14 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.
```
                                                                {
  return math_to_phys(in_p);
}
```

References math_to_phys().

5.75.2.15 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 PredefinedShapeTransformatio

5.75.2.16 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.

```
{return;};
```

Referenced by get_inverse_Tensor_for_step(), and get_Tensor_for_step().

5.75.2.17 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.

```
                                                                  {
    apply_math_to_phys = appl_math_to_phys;
}
```

5.75.2.18 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.

```
                                                                                    {
  std::pair<int, double> ret;
  ret.first = 0;
  ret.second = 0.0;
  if (in_z <= Geometry.global_z_range.first) {
    ret.first = 0;
    ret.second = 0.0;
  } else if (in_z < Geometry.global_z_range.second && in_z > Geometry.global_z_range.first) {
    ret.first = floor( (in_z + Geometry.global_z_range.first) / (GlobalParams.Sector_thickness));
    ret.second = (in_z + Geometry.global_z_range.first -  (ret.first * GlobalParams.Sector_thickness)) /
    (GlobalParams.Sector_thickness);
  } else if (in_z >= Geometry.global_z_range.second) {
    ret.first = GlobalParams.Number_of_sectors - 1;
    ret.second = 1.0;
  }

  if (ret.second < 0 || ret.second > 1){
    std::cout « "Global ranges: " « Geometry.global_z_range.first « " to " «
    Geometry.global_z_range.second « std::endl;
    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;
    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;
  }
  return ret;
}
```

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

5.76 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**

5.76.1 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.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 33 of file SquareMeshGenerator.h.

5.76.2 Member Function Documentation

5.76.2.1 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.

5.76.2.2 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.

5.76.2.3 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.

```
                                                                                {
  GridGenerator::hyper_cube(*in_tria, -1.0, 1.0, false);
  GridTools::transform(&Triangulation_Shit_To_Local_Geometry, *in_tria);
  set_boundary_ids(*in_tria);

  in_tria->signals.post_refinement.connect(
      std::bind(&SquareMeshGenerator::set_boundary_ids,
          std::cref(*this), std::ref(*in_tria)));

  refine_triangulation_iteratively(in_tria);

  set_boundary_ids(*in_tria);
}
```

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

- Code/MeshGenerators/SquareMeshGenerator.h
- Code/MeshGenerators/SquareMeshGenerator.cpp

5.77 SurfaceCellData Struct Reference

Public Attributes

- std::vector< DofNumber > **dof_numbers**
- Position **surface_face_center**

5.77.1 Detailed Description

Definition at line 217 of file Types.h.

The documentation for this struct was generated from the following file:

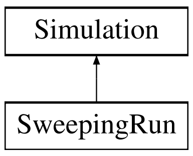
- Code/Core/Types.h

5.78 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.

5.78.1 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

5.79 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**

5.79.1 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.

5.79.2 Member Function Documentation

5.79.2.1 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 }
```

5.79.2.2 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

5.80 VertexAngelingData Struct Reference

Public Attributes

- unsigned int **vertex_index**
- bool **angled_in_x** = false
- bool **angled_in_y** = false

5.80.1 Detailed Description

Definition at line 80 of file Types.h.

The documentation for this struct was generated from the following file:

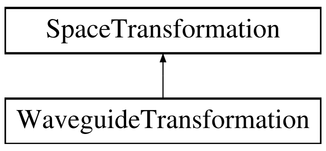
- Code/Core/Types.h

5.81 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

5.81.1 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.

5.81.2 Member Function Documentation

5.81.2.1 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.

```
                                                       {
  vertical_shift.set_constraints(0, GlobalParams.Vertical_displacement_of_waveguide, 0,0);
  vertical_shift.initialize();
  if(!GlobalParams.keep_waveguide_height_constant) {
    waveguide_height.set_constraints(1, 1, 0,0);
    waveguide_height.initialize();
  }
  if(!GlobalParams.keep_waveguide_width_constant) {
    waveguide_width.set_constraints(1, 1, 0,0);
    waveguide_height.initialize();
  }
}
```

References ShapeFunction::set_constraints().

5.81.2.2 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;
}
```

5.81.2.3 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().

5.81.2.4 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.

```
                                                                    {
  std::pair<ResponsibleComponent, unsigned int> comp = map_free_dof_index(index);
  switch (comp.first)
  {
    case VerticalDisplacementComponent:
      return vertical_shift.get_free_dof_value(comp.second);
      break;
    case WaveguideHeightComponent:
      return waveguide_height.get_free_dof_value(comp.second);
      break;
    case WaveguideWidthComponent:
      return waveguide_width.get_free_dof_value(comp.second);
      break;
    default:
      break;
  }
  return 0.0;
}
```

5.81.2.5 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.

```
                                                                          {
  Tensor<2,3,double> ret = I;
  const double z = in_p[2];
  if(GlobalParams.keep_waveguide_height_constant && GlobalParams.keep_waveguide_width_constant) {
    // Only shift down vertically
    ret[1][2] = -vertical_shift.evaluate_derivative_at(z);
  } else {
    const double y = in_p[1];
    const double h = waveguide_height.evaluate_at(z);
    const double dh = waveguide_height.evaluate_derivative_at(z);
    const double dm = vertical_shift.evaluate_derivative_at(z);
    if(GlobalParams.keep_waveguide_width_constant) {
      // Vertical shift and vertical stretching of the waveguide (variable height)
      // f(y) = (y / waveguide_height.evaluate_at(z)) - vertical_shift.evaluate_at(z);
      ret[1][2] = - dm - y * dh / h*h;
    } else {
      // Vertical shift, vertical stretching and horizontal stretching of the waveguide (variable height
     and width)
      const double w = waveguide_width.evaluate_at(z);
      const double dw = waveguide_width.evaluate_derivative_at(z);
      const double x = in_p[0];
      ret[0][2] = - x  * dw / (w*w);
      ret[1][2] = - dm - y * dh / h*h;
    }
  }

  return ret;
}
```

Referenced by get_J_inverse(), and get_Space_Transformation_Tensor().

5.81.2.6 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().

5.81.2.7 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> 33 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().

5.81.2.8 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> 33 complex valued tensor for a given locations.

Implements SpaceTransformation.

Definition at line 64 of file WaveguideTransformation.cpp.

```
{
  return get_Space_Transformation_Tensor(position);
}
```

References get_Space_Transformation_Tensor().

5.81.2.9 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.

```
{
  Position ret;
  if(GlobalParams.keep_waveguide_width_constant) {
    ret[0] = coord[0];
  } else {
    ret[0] = coord[0] * waveguide_width.evaluate_at(coord[2]);
  }
  if(GlobalParams.keep_waveguide_height_constant) {
    ret[1] = coord[1] + vertical_shift.evaluate_at(coord[2]);
  } else {
    ret[1] = (coord[1] + vertical_shift.evaluate_at(coord[2])) * waveguide_height.evaluate_at(coord[2]);
  }
  ret[2] = coord[2];
  return ret;
}
```

5.81.2.10 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.

```
                                                       {
  unsigned int ret = vertical_shift.n_dofs;
  if(!GlobalParams.keep_waveguide_height_constant) {
    ret += waveguide_height.n_dofs;
  }
  if(!GlobalParams.keep_waveguide_width_constant) {
    ret += waveguide_width.n_dofs;
  }
  return ret;
}
```

Referenced by get_dof_values().

5.81.2.11 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.

```
                                                                          {
  Position ret;
  if(GlobalParams.keep_waveguide_width_constant) {
    ret[0] = coord[0];
  } else {
    ret[0] = coord[0] / waveguide_width.evaluate_at(coord[2]);
  }
  if(GlobalParams.keep_waveguide_height_constant) {
    ret[1] = coord[1] - vertical_shift.evaluate_at(coord[2]);
  } else {
    ret[1] = (coord[1] / waveguide_height.evaluate_at(coord[2])) - vertical_shift.evaluate_at(coord[2]);
  }
  ret[2] = coord[2];
  return ret;
}
```

5.81.2.12 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.

```
                                                                    {
      std::pair<ResponsibleComponent, unsigned int> comp = map_free_dof_index(index);
      switch (comp.first)
      {
        case VerticalDisplacementComponent:
          vertical_shift.set_free_dof_value(comp.second, value);
          return;
          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

Chapter 6

File Documentation

6.1 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.

6.1.1 Detailed Description

Contains the BoundaryCondition base type which serves as the abstract base class for all boundary conditions.

Author

Pascal Kraft (`kraft.pascal@gmail.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.2 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.

6.2.1 Detailed Description

Contains the implementation of Dirichlet tangential data on a boundary.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.3 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.

6.3.1 Detailed Description

Contains an internal data type.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.4 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.

6.4.1 Detailed Description

Contains the implementation of an empty surface, i.e. dirichlet zero trace.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.5 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.

6.5.1 Detailed Description

Contains the implementation of a Hardy polynomial which is required for the Hardy Space infinite elements.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.6 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.

6.6.1 Detailed Description

Implementation of a boundary condition based on Hardy Space infinite elements.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.7 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.

6.7.1 Detailed Description

An internal datatype.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.8 Code/BoundaryCondition/LaguerreFunction.h File Reference

An implementation of Laguerre functions which is not currently being used.

Classes

- class LaguerreFunction

6.8.1 Detailed Description

An implementation of Laguerre functions which is not currently being used.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.9 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.

6.9.1 Detailed Description

An implementation of a surface that handles the communication with a neighboring process.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.10 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.

6.10.1 Detailed Description

Coordinate transformation for PML domains.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.11 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.

6.11.1 Detailed Description

Implementation of the PML Surface class.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.12 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_SURF**↩
  ACE }
- 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** }

6.12.1 Detailed Description

All the enums used in this project.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.13 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.

6.13.1 Detailed Description

A base class for all objects that have either locally owned or active dofs.

Author

Pascal Kraft (kraft.pascal@gmail.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.14 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.

6.14.1 Detailed Description

Contains the implementation of the inner domain which handles the part of the computational domain that is locally owned.

Author

Pascal Kraft (`kraft.pascal@gmail.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.15 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.

6.15.1 Detailed Description

Contains the header of the Sector class.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.16 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**

6.16.1 Detailed Description

This file contains all type declarations used in this project.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.16.2 Variable Documentation

6.16.2.1 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.

6.17 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).

6.17.1 Detailed Description

Contains the GeometryManager header, which handles the distribution of the computational domain onto processes and most of the initialization.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.18 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**

6.18.1 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.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.19 Code/GlobalObjects/ModeManager.h File Reference

Not currently in use.

```
#include <deal.II/base/point.h>
```

Classes

- class ModeManager

6.19.1 Detailed Description

Not currently in use.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.20 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.

6.20.1 Detailed Description

Creates filenames and manages file system paths.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.21 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.

6.21.1 Detailed Description

Implementation of a handler for multiple timers with names that can gernerate output.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.22 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.

6.22.1 Detailed Description

A utility class that overrides certain parameters from an input file.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.23 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.

6.23.1 Detailed Description

Contains the parameter reader header. This object parses the parameter files.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.24 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.

6.24.1 Detailed Description

A struct containing all provided parameter values and some computed values based on it (like MPI rank etc.)

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.25 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

6.25.1 Detailed Description

Some implementations of fields that can be used in the code for forcing or error computation.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.26 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.

6.26.1 Detailed Description

Not currently used.

Author

your name (`you@domain.com`)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.27 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

6.27.1 Detailed Description

An object used to wrap the description of the prescribed waveguide shapes.

Author

your name (you@domain.com)

Version

0.1

Date

2022-03-22

Copyright

Copyright (c) 2022

6.28 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 > ∗, DoF↩Handler< 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::D↩EBUG_ONE)
- void **print_info** (const std::string &label, const unsigned int message, LoggingLevel level=LoggingLevel::↩DEBUG_ONE)
- 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**

6.28.1 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

6.28.2 Function Documentation

6.28.2.1 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;
}
```

6.29 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

6.29.1 Detailed Description

This class contains a forward declaration of LocalProblem and NonLocalProblem and the class HierarchicalProblem.

Author

Pascal Kraft

Version

0.1

Date

2022-04-21

Copyright

Copyright (c) 2022

6.30 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.

6.30.1 Detailed Description

This class stores the implementation of the MPICommunicator type.

Author

Pascal Kraft

Version

0.1

Date

2022-04-21

Copyright

Copyright (c) 2022

6.31 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.

6.31.1 Detailed Description

This file includes the class NonLocalProblem which is the essential class for the hierarchical sweeping preconditioner.

Author

Pascal Kraft

Version

0.1

Date

2022-04-21

Copyright

Copyright (c) 2022

6.32 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.

6.32.1 Detailed Description

Author

Pascal Kraft

Version

0.1

Date

2022-04-21

Copyright

Copyright (c) 2022

6.33 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.

6.33.1 Detailed Description

This is no longer active code.

Author

Pascal Kraft

Version

0.1

Date

2022-04-21

Copyright

Copyright (c) 2022

6.34 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.

6.34.1 Detailed Description

Stores the implementation of the ShapeFunction Class.

Author

Pascal Kraft

Version

0.1

Date

2022-04-21

Copyright

Copyright (c) 2022

6.35 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.

6.35.1 Detailed Description

Contains the Optimization Runner which performs shape optimization type computations.

Author

Pascal Kraft

Version

0.1

Date

2022-04-22

Copyright

Copyright (c) 2022

6.36 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.

6.36.1 Detailed Description

Contains the parameter sweep runner which is somewhat deprecated.

Author

Pascal Kraft

Version

0.1

Date

2022-04-22

Copyright

Copyright (c) 2022

6.37 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.

6.37.1 Detailed Description

Base class of the simulation runners.

Author

Pascal Kraft

Version

0.1

Date

2022-04-22

Copyright

Copyright (c) 2022

6.38 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.

6.38.1 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.

Author

Pascal Kraft

Version

0.1

Date

2022-04-22

Copyright

Copyright (c) 2022

6.39 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.

6.39.1 Detailed Description

Default Runner for sweeping preconditioner runs.

Author

Pascal Kraft

Version

0.1

Date

2022-04-22

Copyright

Copyright (c) 2022

6.40 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** }

6.40.1 Detailed Description

Contains the implementation of the Waveguide Transformation.

Author

Pascal Kraft

Version

0.1

Date

2022-04-22

Copyright

Copyright (c) 2022