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

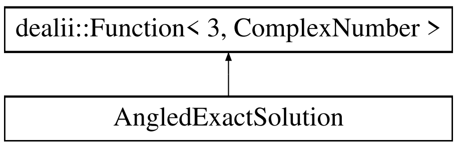
Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

Class Documentation

4.1 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

4.1.1 Detailed Description

Definition at line 12 of file AngledExactSolution.h.

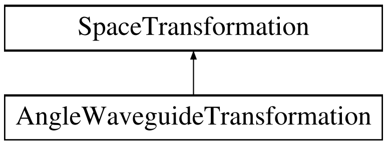
The documentation for this class was generated from the following files:

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

4.2 AngleWaveguideTransformation Class Reference

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

Inheritance diagram for AngleWaveguideTransformation:



Public Member Functions

- Position **math_to_phys** (Position coord) const
- Position **phys_to_math** (Position coord) const
- dealii::Tensor< 2, 3, double > **get_J** (Position &coordinate) override
- dealii::Tensor< 2, 3, double > **get_J_inverse** (Position &coordinate) override
- double **get_det** (Position coord) override
- dealii::Tensor< 2, 3, ComplexNumber > **get_Tensor** (Position &coordinate) override
- dealii::Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Position &coordinate) override
- 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

4.2.1 Detailed Description

Author

Pascal Kraft

Date

28.11.2016

Definition at line 20 of file AngleWaveguideTransformation.h.

4.2.2 Member Function Documentation

4.2.2.1 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 71 of file AngleWaveguideTransformation.cpp.

```
71                                                                      {
72
73 }
```

4.2.2.2 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 75 of file AngleWaveguideTransformation.cpp.

```
75                                                                              {
76   Vector<double> ret;
77   return ret;
78 }
```

4.2.2.3 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 88 of file AngleWaveguideTransformation.cpp.

```
88                                                                   {
89   return 0;
90 }
```

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

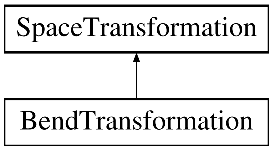
- Code/SpaceTransformations/AngleWaveguideTransformation.h
- Code/SpaceTransformations/AngleWaveguideTransformation.cpp

4.3 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
- Position **phys_to_math** (Position coord) const override
- dealii::Tensor< 2, 3, ComplexNumber > **get_Tensor** (Position &coordinate) override
- dealii::Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Position &coordinate) override
- 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

4.3.1 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.

4.3.2 Member Function Documentation

4.3.2.1 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.

```
41                                                          {
42    return;
43 }
```

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

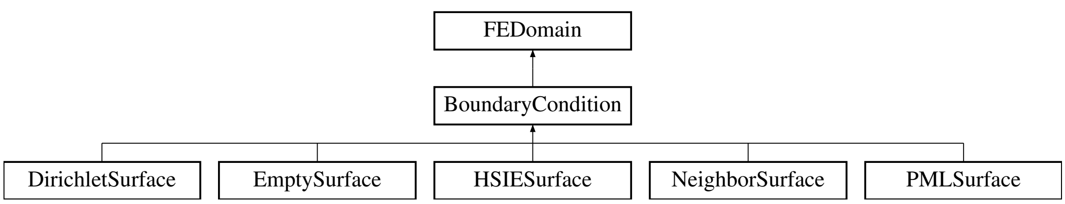
- Code/SpaceTransformations/BendTransformation.h
- Code/SpaceTransformations/BendTransformation.cpp

4.4 BoundaryCondition Class Reference

This is the base type for boundary coniditions. Some implementations are done on this level, some in the derived types.

```
#include <BoundaryCondition.h>
```

Inheritance diagram for BoundaryCondition:



Public Member Functions

- **BoundaryCondition** (unsigned int in_bid, unsigned int in_level, double in_additional_coordinate)
- virtual void initialize ()=0

  Not all data for objects of this type will be available at time of construction.
- virtual std::string output_results (const dealii::Vector< ComplexNumber > &in_solution, std::string filename)=0

  Writes output for a provided solution to a file with the provided name.
- virtual bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid)=0

  Checks if a 2D coordinate is on the a surface of the boundary methods domain.
- void set_mesh_boundary_ids ()

  If the boundary condition has its own mesh, this function iterates over the mesh and sets boundary ids on the mesh.
- auto get_boundary_ids () -> std::vector< BoundaryId >

  Returns a vector of all boundary ids associated with dofs in this domain.
- virtual auto get_dof_association () -> std::vector< InterfaceDofData >=0

  Returns a vector of all degrees of freedom shared with the inner domain.
- virtual auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData >=0

  More general version of the function above that can also handle interfaces with other boundary ids.
- virtual auto get_global_dof_indices_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< DofNumber >

  Specific version of the function above that provides the indices in the returned vector by their globally unique id instead of local numbering.
- virtual void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗constraints)=0

  If this object owns degrees of freedom, this function fills a sparsity pattern for their global indices.
- virtual void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix ∗matrix, NumericVectorDistributed ∗rhs, Constraints ∗constraints)=0

  Fills a provided matrix and right-hand side vector with the data related to the current fem system under consideration and related to this boundary condition.
- virtual void finish_dof_index_initialization ()

  Handles the communication of non-locally owned dofs and thus finishes the setup of the object.
- virtual auto make_constraints () -> Constraints

  Builds a constraint object that represents fixed values of degrees of freedom associated with this object.
- double boundary_norm (NumericVectorDistributed ∗solution)

  Computes the L2-norm of the solution passed in on the shared interface with the interior domain.
- double boundary_surface_norm (NumericVectorDistributed ∗solution, BoundaryId b_id)

  Computes the L2-norm of the solution passed in as an argument on the solution passed in as the second argument.
- virtual unsigned int cells_for_boundary_id (unsigned int boundary_id)

  Counts the number of cells associated with the boundary passed in as an argument.
- void print_dof_validation ()

  In some cases we have more then one option to validate how many dofs a domain should have.
- void force_validation ()

  Triggers the internal validation routine.
- virtual unsigned int n_cells ()

  Counts the number of cells used in the object.

Public Attributes

- const BoundaryId **b_id**
- const unsigned int **level**
- const double **additional_coordinate**
- std::vector< InterfaceDofData > **surface_dofs**
- bool **surface_dof_sorting_done**
- bool **boundary_coordinates_computed** = false
- std::array< double, 6 > **boundary_vertex_coordinates**
- DofCount **dof_counter**
- int **global_partner_mpi_rank**
- int **local_partner_mpi_rank**
- const std::vector< BoundaryId > **adjacent_boundaries**
- std::array< bool, 6 > **are_edge_dofs_owned**
- DofHandler3D **dof_handler**

4.4.1 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 30 of file BoundaryCondition.h.

4.4.2 Member Function Documentation

4.4.2.1 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.

```
96                                                                           {
97    double ret = 0;
98    for(unsigned int i = 0; i < global_index_mapping.size(); i++) {
99      ret += norm_squared(in_v->operator()(global_index_mapping[i]));
100   }
101   return std::sqrt(ret);
102 }
```

4.4.2.2 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.

```
104                                                                                           {
105   double ret = 0;
106   auto dofs = get_dof_association_by_boundary_id(in_bid);
107   for(auto it : dofs) {
108     ret += norm_squared(in_v->operator()(it.index));
109   }
110   return std::sqrt(ret);
111 }
```

References get_dof_association_by_boundary_id().

4.4.2.3 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 }
```

4.4.2.4 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, EmptySurface, DirichletSurface, PMLSurface, and NeighborSurface.

4.4.2.5 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.

4.4.2.6 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.

```
87                                                             {
88
89 }
```

4.4.2.7 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::cout « "A" « std::endl;
154           std::vector<InterfaceDofData> d = get_dof_association_by_boundary_id(surf);
155      //      std::cout « "B" « std::endl;
156           bool one_is_invalid = false;
157           unsigned int count_before = 0;
158           unsigned int count_after = 0;
159           for(unsigned int index = 0; index < d.size(); index++) {
160             if(!is_dof_owned[d[index].index]) {
161               if(global_index_mapping[d[index].index] >= Geometry.levels[level].n_total_level_dofs) {
162                 one_is_invalid = true;
163                 count_before ++;
164               }
165             }
166           }
167        //   std::cout « "C" « std::endl;
168           if(one_is_invalid) {
169             std::cout « "Forcing validation on " « b_id « " for " « surf « std::endl;
170             std::vector<unsigned int> local_indices(d.size());
171             for(unsigned int i = 0; i < d.size(); i++) {
172               local_indices[i] = d[i].index;
173             }
174             set_non_local_dof_indices(local_indices,
    Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id));
175             for(unsigned int index = 0; index < d.size(); index++) {
176               if(!is_dof_owned[d[index].index]) {
177                 if(global_index_mapping[d[index].index] >= Geometry.levels[level].n_total_level_dofs) {
178                   count_after ++;
179                 }
180               }
181             }
182             std::cout « "Count before: " « count_before « " and after: " « count_after « std::endl;
183           }
184         }
185     }
186   }
187 }
```

References get_dof_association_by_boundary_id().

4.4.2.8 get_boundary_ids()

```
std::vector< unsigned int > BoundaryCondition::get_boundary_ids ( ) -> std::vector<BoundaryId>
```

Returns a vector of all boundary ids associated with dofs in this domain.

Returns

The returned vector contains all boundary IDs that are relevant on this domain.

Definition at line 72 of file BoundaryCondition.cpp.

```
72                                                                {
73   return (Geometry.surface_meshes[b_id].get_boundary_ids());
74 }
```

4.4.2.9 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, EmptySurface, DirichletSurface, PMLSurface, and NeighborSurface.

4.4.2.10 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, EmptySurface, DirichletSurface, PMLSurface, and NeighborSurface.

Referenced by boundary_surface_norm(), force_validation(), get_global_dof_indices_by_boundary_id(), and print_dof_validation().

4.4.2.11 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.

```
76                                                                                              {
77   std::vector<InterfaceDofData> dof_data = get_dof_association_by_boundary_id(in_boundary_id);
78   std::vector<DofNumber> ret;
79   for(unsigned int i = 0; i < dof_data.size(); i++) {
80     ret.push_back(dof_data[i].index);
```

```
    }

    ret = transform_local_to_global_dofs(ret);
    return ret;
}
```

References get_dof_association_by_boundary_id().

4.4.2.12 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, EmptySurface, DirichletSurface, PMLSurface, and NeighborSurface.

4.4.2.13 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, EmptySurface, DirichletSurface, PMLSurface, and NeighborSurface.

4.4.2.14 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.

```
91                                              {
92   Constraints ret(global_dof_indices);
93   return ret;
94 }
```

4.4.2.15 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 189 of file BoundaryCondition.cpp.

```
189                                        {
190   return 0;
191 }
```

4.4.2.16 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 EmptySurface, DirichletSurface, HSIESurface, PMLSurface, and NeighborSurface.

4.4.2.17 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                                               {
```

```
  unsigned int n_invalid_dofs = 0;
  for(unsigned int i = 0; i < n_locally_active_dofs; i++) {
    if(global_index_mapping[i] >= Geometry.levels[level].n_total_level_dofs) {
      n_invalid_dofs++;
    }
  }
  if(n_invalid_dofs > 0) {
    std::cout « "On process " « GlobalParams.MPI_Rank « " surface " « b_id « " has " « n_invalid_dofs «
 " invalid dofs." « std::endl;
    for(unsigned int surf = 0; surf < 6; surf++) {
      if(surf != b_id && !are_opposing_sites(b_id, surf)) {
        unsigned int invalid_dof_count = 0;
        unsigned int owned_invalid = 0;
        auto dofs = get_dof_association_by_boundary_id(surf);
        for(auto dof:dofs) {
          if(global_index_mapping[dof.index] >= Geometry.levels[level].n_total_level_dofs) {
            invalid_dof_count++;
            if(is_dof_owned[dof.index]) {
              owned_invalid++;
            }
          }
        }
        if(invalid_dof_count > 0) {
          std::cout « "On process " « GlobalParams.MPI_Rank « " surface " « b_id « " there were "«
 invalid_dof_count « "(" « owned_invalid « ") invalid dofs towards "« surf « std::endl;
        }
      }
    }
  }
}
```

References get_dof_association_by_boundary_id().

4.4.2.18 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.

```
22                                           {
23     auto it = Geometry.surface_meshes[b_id].begin_active();
24     std::vector<double> x;
25     std::vector<double> y;
26     while(it != Geometry.surface_meshes[b_id].end()){
27       if(it->at_boundary()) {
28         for (unsigned int face = 0; face < GeometryInfo<2>::faces_per_cell; ++face) {
29           if (it->face(face)->at_boundary()) {
30             dealii::Point<2, double> c;
31             c = it->face(face)->center();
32             x.push_back(c[0]);
33             y.push_back(c[1]);
34           }
```

```
        }
      }
      ++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;
  }
}
```

Referenced by HSIESurface::HSIESurface().

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

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

4.5 BoundaryInformation Struct Reference

Public Member Functions

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

Public Attributes

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

4.5.1 Detailed Description

Definition at line 117 of file Types.h.

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

- Code/Core/Types.h

4.6 CellAngelingData Struct Reference

Public Attributes

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

4.6.1 Detailed Description

Definition at line 76 of file Types.h.

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

- Code/Core/Types.h

4.7 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**

4.7.1 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

4.8 CellwiseAssemblyDataNP Struct Reference

Public Member Functions

- **CellwiseAssemblyDataNP** (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_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**

4.8.1 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

4.9 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**

4.9.1 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

4.10 ConstraintPair Struct Reference

Public Attributes

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

4.10.1 Detailed Description

Definition at line 201 of file Types.h.

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

- Code/Core/Types.h

4.11 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** ()

4.11.1 Detailed Description

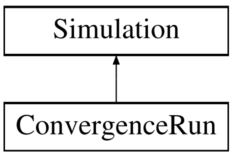
Definition at line 5 of file ConvergenceOutputGenerator.h.

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

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

4.12 ConvergenceRun Class Reference

Inheritance diagram for ConvergenceRun:



Public Member Functions

- void **prepare** () override
- void **run** () override
- void **write_outputs** ()
- void **prepare_transformed_geometry** () override
- void **set_norming_factor** ()
- double **compute_error_for_two_eval_vectors** (std::vector< std::vector< ComplexNumber >> a, std::vector< std::vector< ComplexNumber >> b)

4.12.1 Detailed Description

Definition at line 9 of file ConvergenceRun.h.

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

- Code/Runners/ConvergenceRun.h
- Code/Runners/ConvergenceRun.cpp

4.13 CoreLogger Class Reference

Outputs I want:

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

4.13.1 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

4.14 DataSeries Struct Reference

Public Attributes

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

4.14.1 Detailed Description

Definition at line 212 of file Types.h.

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

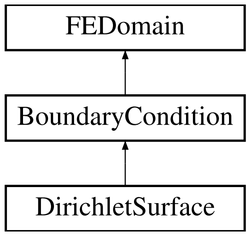
- Code/Core/Types.h

4.15 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

4.15.1 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 18 of file DirichletSurface.h.

4.15.2 Member Function Documentation

4.15.2.1 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.

```
64 {
65     return 0;
66 }
```

4.15.2.2 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.

```
60 {
61     return 0;
62 }
```

4.15.2.3 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.

```
68 {
69
70 }
```

4.15.2.4 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 }
```

4.15.2.5 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.

```
58 { }
```

4.15.2.6 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.

```
44                                                                          {
45     std::vector<InterfaceDofData> ret;
46     return ret;
47 }
```

4.15.2.7 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.

```
49                                                                                    {
50     std::vector<InterfaceDofData> ret;
51     return ret;
52 }
```

4.15.2.8 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.

```
40                                  {
41
42 }
```

4.15.2.9 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.

```
36                                                                     {
37     return false;
38 }
```

4.15.2.10 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.

```
72                                                 {
73     Constraints ret(Geometry.levels[level].inner_domain->global_dof_indices);
74     dealii::IndexSet local_dof_set(Geometry.levels[level].inner_domain->n_locally_active_dofs);
75     local_dof_set.add_range(0,Geometry.levels[level].inner_domain->n_locally_active_dofs);
76     AffineConstraints<ComplexNumber> constraints_local(local_dof_set);
77     VectorTools::project_boundary_values_curl_conforming_l2(Geometry.levels[level].inner_domain->dof_handler,
       0, *GlobalParams.source_field, b_id, constraints_local);
78     for(auto line : constraints_local.get_lines()) {
79         const unsigned int local_index = line.index;
80         const unsigned int global_index =
       Geometry.levels[level].inner_domain->global_index_mapping[local_index];
81         ret.add_line(global_index);
82         ret.set_inhomogeneity(global_index, line.inhomogeneity);
83     }
84     constraints_local.clear();
85     if(GlobalParams.BoundaryCondition == BoundaryConditionType::PML) {
86         for(unsigned int surf = 0; surf < 6; surf++) {
87             if(surf != b_id && !are_opposing_sites(b_id, surf)) {
88                 if(Geometry.levels[level].surface_type[surf] == SurfaceType::ABC_SURFACE) {
89                     PMLTransformedExactSolution ptes(b_id, additional_coordinate);
90     VectorTools::project_boundary_values_curl_conforming_l2(Geometry.levels[level].surfaces[surf]->dof_handler,
       0, ptes, b_id, constraints_local);
91                     for(auto line : constraints_local.get_lines()) {
92                         const unsigned int local_index = line.index;
93                         const unsigned int global_index =
       Geometry.levels[level].surfaces[surf]->global_index_mapping[local_index];
94                         ret.add_line(global_index);
95                         ret.set_inhomogeneity(global_index, line.inhomogeneity);
96                     }
97                     constraints_local.clear();
98                 }
99             }
100         }
101     }
102     return ret;
103 }
```

4.15.2.11 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.

```
54                                                                                {
55     return "";
56 }
```

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

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

4.16 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**

4.16.1 Detailed Description

Definition at line 149 of file Types.h.

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

- Code/Core/Types.h

4.17 DofCountsStruct Struct Reference

Public Attributes

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

4.17.1 Detailed Description

Definition at line 164 of file Types.h.

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

- Code/Core/Types.h

4.18 DofCouplingInformation Struct Reference

Public Attributes

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

4.18.1 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

4.19 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

4.19.1 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 13 of file DofData.h.

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

- Code/BoundaryCondition/DofData.h

4.20 DofIndexData Class Reference

Public Member Functions

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

Public Attributes

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

4.20.1 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

4.21 DofOwner Struct Reference

Public Attributes

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

4.21.1 Detailed Description

Definition at line 81 of file Types.h.

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

- Code/Core/Types.h

4.22 EdgeAngelingData Struct Reference

Public Attributes

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

4.22.1 Detailed Description

Definition at line 64 of file Types.h.

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

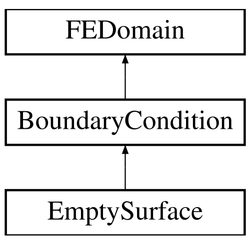
- Code/Core/Types.h

4.23 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

4.23.1 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 19 of file EmptySurface.h.

4.23.2 Member Function Documentation

4.23.2.1 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 }
```

4.23.2.2 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 }
```

4.23.2.3 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 }
```

4.23.2.4 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.

```
30                   {
31      matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
        required.
32      // Nothing to do here, work happens on neighbor process.
33 }
```

4.23.2.5 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.

```
57 { }
```

4.23.2.6 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.

```
43                                                                     {
44     std::vector<InterfaceDofData> ret;
45     return ret;
46 }
```

4.23.2.7 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.

```
48                                                                                                  {
49     std::vector<InterfaceDofData> ret;
50     return ret;
51 }
```

4.23.2.8 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.

```
39                              {
40
41 }
```

4.23.2.9 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.

```
35                                                            {
36     return false;
37 }
```

4.23.2.10 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.

```
71                               {
72      Constraints ret(Geometry.levels[level].inner_domain->global_dof_indices);
73      dealii::IndexSet local_dof_set(Geometry.levels[level].inner_domain->n_locally_active_dofs);
74      local_dof_set.add_range(0,Geometry.levels[level].inner_domain->n_locally_active_dofs);
75      AffineConstraints<ComplexNumber> constraints_local(local_dof_set);
76      std::vector<InterfaceDofData> dofs =
        Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
77      for(auto line : dofs) {
78          const unsigned int local_index = line.index;
79          const unsigned int global_index =
        Geometry.levels[level].inner_domain->global_index_mapping[local_index];
80          ret.add_line(global_index);
81          ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
82      }
83      for(unsigned int surf = 0; surf < 6; surf++) {
84          if(surf != b_id && !are_opposing_sites(b_id, surf)) {
85              if(Geometry.levels[level].surface_type[surf] == SurfaceType::ABC_SURFACE) {
86                  std::vector<InterfaceDofData> dofs =
        Geometry.levels[level].surfaces[surf]->get_dof_association_by_boundary_id(b_id);
87                  for(unsigned int i = 0; i < dofs.size(); i++) {
88                      const unsigned int local_index = dofs[i].index;
89                      const unsigned int global_index =
        Geometry.levels[level].surfaces[surf]->global_index_mapping[local_index];
90                      ret.add_line(global_index);
91                      ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
92                  }
93              }
94          }
95      }
96    return ret;
97 }
```

4.23.2.11 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.

```
53                                                                                                {
54     return "";
55 }
```

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

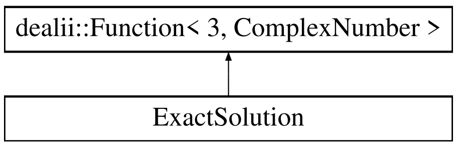
- Code/BoundaryCondition/EmptySurface.h
- Code/BoundaryCondition/EmptySurface.cpp

4.24 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

- **ExactSolution** (bool in_rectangular=false, bool in_dual=false)
- 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

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**

4.24.1 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

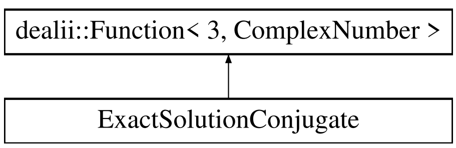
Definition at line 35 of file ExactSolution.h.

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

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

4.25 ExactSolutionConjugate Class Reference

Inheritance diagram for ExactSolutionConjugate:



Public Member Functions

- **ExactSolutionConjugate** (bool in_rectangular=false, bool in_dual=false)
- 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

4.25.1 Detailed Description

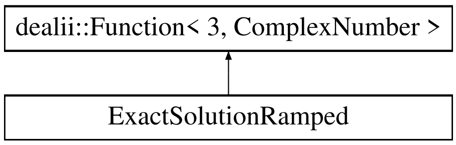
Definition at line 12 of file ExactSolutionConjugate.h.

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

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

4.26 ExactSolutionRamped Class Reference

Inheritance diagram for ExactSolutionRamped:



Public Member Functions

- **ExactSolutionRamped** (bool in_rectangular=false, bool in_dual=false)
- 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

4.26.1 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

4.27 FEAdjointEvaluation Struct Reference

Public Attributes

- Position **x**
- dealii::Tensor< 1, 3, ComplexNumber > **primal_field**
- dealii::Tensor< 1, 3, ComplexNumber > **adjoint_field**

4.27.1 Detailed Description

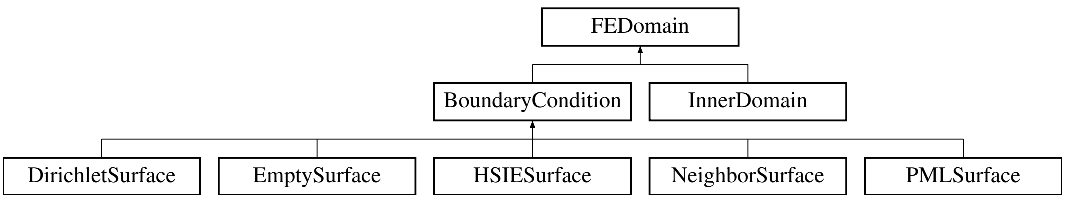
Definition at line 223 of file Types.h.

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

- Code/Core/Types.h

4.28 FEDomain Class Reference

Inheritance diagram for FEDomain:



Public Member Functions

- virtual void **determine_non_owned_dofs** ()=0
- void **initialize_dof_counts** (DofCount n_locally_active_dofs, DofCount n_locally_owned_dofs)
- DofIndexVector **transform_local_to_global_dofs** (DofIndexVector local_indices)
- void **mark_local_dofs_as_non_local** (DofIndexVector)
- virtual bool **finish_initialization** (DofNumber first_own_index)
- void **set_non_local_dof_indices** (DofIndexVector local_indices, DofIndexVector global_indices)
- virtual DofCount **compute_n_locally_owned_dofs** ()=0
- virtual DofCount **compute_n_locally_active_dofs** ()=0
- void **freeze_ownership** ()
- NumericVectorLocal **get_local_vector_from_global** (const NumericVectorDistributed in_vector)
- double **local_norm_of_vector** (NumericVectorDistributed ∗)

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**

4.28.1 Detailed Description

Definition at line 7 of file FEDomain.h.

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

- Code/Core/FEDomain.h
- Code/Core/FEDomain.cpp

4.29 FEErrorStruct Struct Reference

Public Attributes

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

4.29.1 Detailed Description

Definition at line 218 of file Types.h.

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

- Code/Core/Types.h

4.30 FileLogger Class Reference

There will be one global instance of this object.

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

4.30.1 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

4.31 FileMetaData Struct Reference

Public Attributes

- unsigned int **hsie_level**

4.31.1 Detailed Description

Definition at line 103 of file Types.h.

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

- Code/Core/Types.h

4.32 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** ()
- void **initialize_inner_domain** (unsigned int in_level)
- void **initialize_surfaces** ()
- void **perform_initialization** (unsigned int level)
- double **eps_kappa_2** (Position)
- double **kappa_2** ()
- std::pair< double, double > **compute_x_range** ()
- std::pair< double, double > **compute_y_range** ()
- std::pair< double, double > **compute_z_range** ()
- void **set_x_range** (std::pair< double, double >)
- void **set_y_range** (std::pair< double, double >)
- void **set_z_range** (std::pair< double, double >)
- std::pair< bool, unsigned int > **get_global_neighbor_for_interface** (Direction)
- std::pair< bool, unsigned int > **get_level_neighbor_for_interface** (Direction, unsigned int)
- bool **math_coordinate_in_waveguide** (Position) const
- dealii::Tensor< 2, 3 > **get_epsilon_tensor** (const Position &)
- double **get_epsilon_for_point** (const Position &)
- 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**

4.32.1 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 35 of file GeometryManager.h.

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

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

4.33 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**

4.33.1 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

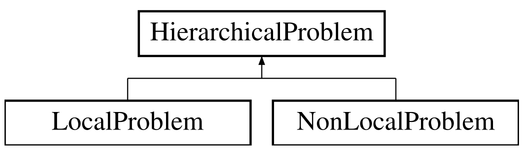
Definition at line 12 of file GradientTable.h.

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

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

4.34 HierarchicalProblem Class Reference

Inheritance diagram for HierarchicalProblem:



Public Member Functions

- **HierarchicalProblem** (unsigned int level, SweepingDirection direction)
- virtual void **solve** ()=0
- void **solve_with_timers_and_count** ()
- virtual void **initialize** ()=0
- void **make_constraints** ()
- virtual void **assemble** ()=0
- virtual void **initialize_index_sets** ()=0
- void **constrain_identical_dof_sets** (std::vector< unsigned int > *set_one, std::vector< unsigned int > *set_two, Constraints *affine_constraints)
- virtual auto **reinit** () -> void=0
- auto **opposing_site_bid** (BoundaryId) -> BoundaryId
- void **compute_final_rhs_mismatch** ()
- virtual void **compute_solver_factorization** ()=0
- std::string **output_results** (std::string in_fname_part="solution_inner_domain_level")
- virtual void **reinit_rhs** ()=0
- virtual void **make_sparsity_pattern** ()
- void **initialize_dof_counts** ()
- virtual void **update_convergence_criterion** (double)
- virtual unsigned int **compute_global_solve_counter** ()
- void **print_solve_counter_list** ()
- virtual void **empty_memory** ()
- virtual void **write_multifile_output** (const std::string &filename, bool apply_coordinate_transform)=0
- virtual std::vector< double > **compute_shape_gradient** ()

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

4.34.1 Detailed Description

Definition at line 16 of file HierarchicalProblem.h.

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

- Code/Hierarchy/HierarchicalProblem.h
- Code/Hierarchy/HierarchicalProblem.cpp

4.35 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**

4.35.1 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 20 of file HSIEPolynomial.h.

4.35.2 Member Function Documentation

4.35.2.1 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.

```
10                                                                    {
11   HSIEPolynomial::D.reinit(dimension, dimension);
12   for (unsigned int i = 0; i < dimension; i++) {
13     for (unsigned int j = 0; j < dimension; j++) {
14       HSIEPolynomial::D.set(i, j, matrixD(i, j, k0));
15     }
16   }
17
18   HSIEPolynomial::I.copy_from(HSIEPolynomial::D);
19   HSIEPolynomial::I.invert(HSIEPolynomial::D);
20   HSIEPolynomial::matricesLoaded = true;
21 }
```

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

4.35.2.2 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.

```
23                                                     {
24   ComplexNumber ret(a[0]);
25   ComplexNumber x = x_in;
26   for (unsigned long i = 1; i < a.size(); i++) {
27     ret += a[i] * x;
28     x = x * x_in;
29   }
30   return ret;
31 }
```

4.35.2.3 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.

```
33                                                            {
34   ComplexNumber ret(da[0]);
35   ComplexNumber x = x_in;
36   for (unsigned long i = 1; i < da.size(); i++) {
37     ret += da[i] * x;
38     x = x * x_in;
39   }
40   return ret;
41 }
```

4.35.2.4 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.

```
105                                          {
106   da = std::vector<ComplexNumber>();
107   for (unsigned int i = 1; i < a.size(); i++) {
108     da.emplace_back(i * a[i].real(), i * a[i].imag());
109   }
110 }
```

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

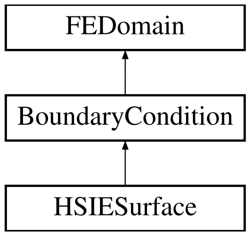
- Code/BoundaryCondition/HSIEPolynomial.h
- Code/BoundaryCondition/HSIEPolynomial.cpp

4.36 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**

4.36.1 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.

Definition at line 36 of file HSIESurface.h.

4.36.2 Constructor & Destructor Documentation

4.36.2.1 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.

```
19     : BoundaryCondition(surface, in_level, Geometry.surface_extremal_coordinate[surface]),
20       order(GlobalParams.HSIE_polynomial_degree),
21       dof_h_q(Geometry.surface_meshes[surface]),
22       Inner_Element_Order(GlobalParams.Nedelec_element_order),
23       fe_nedelec(Inner_Element_Order),
24       fe_q(Inner_Element_Order + 1),
25       kappa(2.0 * GlobalParams.Pi / GlobalParams.Lambda) {
26     dof_h_nedelec.reinit(Geometry.surface_meshes[surface]);
27     dof_h_q.reinit(Geometry.surface_meshes[surface]);
28     set_mesh_boundary_ids();
29     dof_counter = 0;
30     k0 = GlobalParams.kappa_0;
31 }
```

References BoundaryCondition::set_mesh_boundary_ids().

4.36.3 Member Function Documentation

4.36.3.1 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 }
```

Referenced by register_new_edge_dofs(), and register_new_surface_dofs().

4.36.3.2 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.

```
555                                          {
556   std::vector<HSIEPolynomial> ret;
557   HSIEPolynomial temp = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
558   temp.multiplyBy(fe_shape_gradient_component_0[1]);
559   temp.applyI();
560   HSIEPolynomial temp2 = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
561   temp2.multiplyBy(-1.0 * fe_shape_gradient_component_1[0]);
562   temp2.applyI();
563   temp.add(temp2);
564   ret.push_back(temp);
565
566   temp = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
567   temp.multiplyBy(-1.0 * fe_shape_value_component_1);
568   temp.applyDerivative();
569   ret.push_back(temp);
570
571   temp = HSIEPolynomial::PsiJ(dof_hsie_order, k0);
572   temp.multiplyBy(fe_shape_value_component_0);
573   temp.applyDerivative();
574   ret.push_back(temp);
575
576   transform_coordinates_in_place(&ret);
577   return ret;
578 }
```

References transform_coordinates_in_place().

Referenced by fill_matrix().

4.36.3.3 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.

```
595
                          {
596   std::vector<HSIEPolynomial> ret;
597   ret.push_back(HSIEPolynomial::ZeroPolynomial());
598   HSIEPolynomial temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
599   temp.multiplyBy(fe_gradient[1]);
600   ret.push_back(temp);
601   temp = HSIEPolynomial::PhiJ(dof_hsie_order, k0);
602   temp.multiplyBy(-1.0 * fe_gradient[0]);
603   ret.push_back(temp);
604   transform_coordinates_in_place(&ret);
605   return ret;
606 }
```

References transform_coordinates_in_place().

Referenced by fill_matrix().

4.36.3.4 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().

4.36.3.5 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().

Referenced by fill_matrix().

4.36.3.6 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().

4.36.3.7 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.

```
760                                                             {
761   auto it = dof_h_nedelec.begin_active();
762   auto it2 = dof_h_q.begin_active();
763   auto end = dof_h_nedelec.end();
764   const unsigned int dofs_per_cell = 4 * compute_dofs_per_vertex() +
765                                      4 * compute_dofs_per_edge(false) +
766                                      compute_dofs_per_face(false);
767   unsigned int counter = 0;
768   for (; it != end; ++it) {
769     DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
770     if (cell_dofs.size() != dofs_per_cell) {
771       for (unsigned int i = 0; i < 7; i++) {
772         unsigned int count = 0;
773         for (unsigned int j = 0; j < cell_dofs.size(); ++j) {
774           if (cell_dofs[j].type == i) count++;
775         }
776         std::cout « cell_dofs.size() « " vs. " « dofs_per_cell « std::endl;
777         std::cout « "For type " « i « " I found " « count « " dofs" « std::endl;
778       }
779       return false;
780     }
781     counter++;
782     it2++;
783   }
784   return true;
785 }
```

References compute_dofs_per_edge(), compute_dofs_per_face(), and compute_dofs_per_vertex().

4.36.3.8 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.

```
787                                    {
788   auto it = dof_h_nedelec.begin();
789   const auto end = dof_h_nedelec.end();
790   while (it != end) {
791     it->clear_user_flag();
792     for (unsigned int i = 0; i < 4; i++) {
793       it->face(i)->clear_user_flag();
794     }
795     it++;
796   }
797 }
```

4.36.3.9 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.

```
330                                                                       {
331   unsigned int ret = 0;
332   const unsigned int INNER_REAL_DOFS_PER_LINE = fe_nedelec.dofs_per_line;
333
334   if (!only_hsie_dofs) {
335     ret += INNER_REAL_DOFS_PER_LINE;
336   }
337
338   ret += INNER_REAL_DOFS_PER_LINE * (order + 1)
339       + (INNER_REAL_DOFS_PER_LINE - 1) * (order + 2);
340
341   return ret;
342 }
```

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

4.36.3.10 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.

```
344                                                                   {
345   unsigned int ret = 0;
346   const unsigned int INNER_REAL_NEDELEC_DOFS_PER_FACE =
347       fe_nedelec.dofs_per_cell -
348       dealii::GeometryInfo<2>::faces_per_cell * fe_nedelec.dofs_per_face;
349
350   ret = INNER_REAL_NEDELEC_DOFS_PER_FACE * (order + 2) * 3;
351   if (only_hsie_dofs) {
352     ret -= INNER_REAL_NEDELEC_DOFS_PER_FACE;
353   }
354   return ret;
355 }
```

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

4.36.3.11 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.

```
357                                                    {
358   unsigned int ret = order + 2;
359
360   return ret;
361 }
```

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

4.36.3.12 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.

```
267                                                      {
268   DoFHandler<2>::active_cell_iterator cell;
269   DoFHandler<2>::active_cell_iterator cell2;
270   DoFHandler<2>::active_cell_iterator endc;
271   endc = dof_h_nedelec.end();
272   DofCountsStruct ret;
273   cell2 = dof_h_q.begin_active();
274   Geometry.surface_meshes[b_id].clear_user_flags();
275   for (cell = dof_h_nedelec.begin_active(); cell != endc; cell++) {
276     for (unsigned int edge = 0; edge < GeometryInfo<2>::lines_per_cell; edge++) {
277       if (!cell->line(edge)->user_flag_set()) {
278         update_dof_counts_for_edge(cell, edge, ret);
279         register_new_edge_dofs(cell, cell2, edge);
280         cell->line(edge)->set_user_flag();
281       }
282     }
283     cell2++;
284   }
285   return ret;
286 }
```

References register_new_edge_dofs(), and update_dof_counts_for_edge().

Referenced by initialize().

4.36.3.13 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.

```
311                                                      {
312   std::set<std::string> touched_faces;
313   DoFHandler<2>::active_cell_iterator cell;
314   DoFHandler<2>::active_cell_iterator cell2;
315   DoFHandler<2>::active_cell_iterator endc;
316   endc = dof_h_nedelec.end();
317   DofCountsStruct ret;
318   cell2 = dof_h_q.begin_active();
319   for (cell = dof_h_nedelec.begin_active(); cell != endc; cell++) {
320     if (touched_faces.end() == touched_faces.find(cell->id().to_string())) {
321       update_dof_counts_for_face(cell, ret);
322       register_new_surface_dofs(cell, cell2);
323       touched_faces.insert(cell->id().to_string());
324     }
325     cell2++;
326   }
327   return ret;
328 }
```

References register_new_surface_dofs(), and update_dof_counts_for_face().

Referenced by initialize().

4.36.3.14 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;
}
```

4.36.3.15 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().

4.36.3.16 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.

```
288                                                 {
289   std::set<unsigned int> touched_vertices;
290   DoFHandler<2>::active_cell_iterator cell;
291   DoFHandler<2>::active_cell_iterator endc;
292   endc = dof_h_q.end();
293   DofCountsStruct ret;
294   for (cell = dof_h_q.begin_active(); cell != endc; cell++) {
295     // for each edge
296     for (unsigned int vertex = 0; vertex < GeometryInfo<2>::vertices_per_cell;
297          vertex++) {
298       unsigned int idx = cell->vertex_dof_index(vertex, 0);
299       if (touched_vertices.end() == touched_vertices.find(idx)) {
300         // handle it
301         update_dof_counts_for_vertex(cell, idx, vertex, ret);
302         register_new_vertex_dofs(cell, idx, vertex);
303         // remember that it has been handled
304         touched_vertices.insert(idx);
305       }
306     }
307   }
308   return ret;
309 }
```

References register_new_vertex_dofs(), and update_dof_counts_for_vertex().

Referenced by initialize().

4.36.3.17 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.

```
1017                                                 {
1018   IndexSet non_owned_dofs(dof_counter);
1019   for(auto it : surface_dofs) {
1020     non_owned_dofs.add_index(it.index);
1021   }
1022   for(auto surf : adjacent_boundaries) {
1023     if(Geometry.levels[level].surface_type[surf] == SurfaceType::NEIGHBOR_SURFACE) {
1024       if(surf % 2 == 0) {
1025         std::vector<InterfaceDofData> dofs_data = get_dof_association_by_boundary_id(surf);
1026         for(auto it : dofs_data) {
1027           non_owned_dofs.add_index(it.index);
1028         }
1029       }
1030     }
1031   }
1032   return non_owned_dofs;
1033 }
```

References get_dof_association_by_boundary_id().

Referenced by compute_n_locally_owned_dofs(), and determine_non_owned_dofs().

4.36.3.18 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().

4.36.3.19 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;
}
```

Referenced by fill_matrix().

4.36.3.20 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_curl_term_nedelec(), build_curl_term_q(), build_fad_for_cell(), build_non_curl_term_q(), compute_dofs_per_edge(), compute_dofs_per_face(), compute_dofs_per_vertex(), HSIEPolynomial::compute↩DandI(), evaluate_a(), and undo_transform().

4.36.3.21 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.

4.36.3.22 fill_sparsity_pattern()

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

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

Parameters

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

Implements BoundaryCondition.

Definition at line 251 of file HSIESurface.cpp.

```
251 {
252   auto it = dof_h_nedelec.begin();
253   auto end = dof_h_nedelec.end();
254   auto it2 = dof_h_q.begin();
255   for (; it != end; ++it) {
256     DofDataVector cell_dofs = get_dof_data_for_cell(it, it2);
257     std::vector<unsigned int> local_indices;
258     for (unsigned int i = 0; i < cell_dofs.size(); i++) {
259       local_indices.push_back(cell_dofs[i].global_index);
260     }
261     local_indices = transform_local_to_global_dofs(local_indices);
262     in_constraints->add_entries_local_to_global(local_indices, *in_dsp);
263     it2++;
264   }
265 }
```

4.36.3.23 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.

```
968                                                   {
969   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);

}
```

References get_dof_association(), and get_dof_association_by_boundary_id().

4.36.3.24 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);
}
```

References get_dof_association().

4.36.3.25 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.

```
702                                                            {
703   std::sort(surface_dofs.begin(), surface_dofs.end(), compareDofBaseDataAndOrientation);
704   std::vector<InterfaceDofData> ret;
705   copy(surface_dofs.begin(), surface_dofs.end(), back_inserter(ret));
706   return ret;
707 }
```

Referenced by finish_dof_index_initialization(), finish_initialization(), and get_dof_association_by_boundary_id().

4.36.3.26 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.

```
841                                                                                                        {
842   if (are_opposing_sites(b_id, in_boundary_id)) {
843     return get_dof_association();
844   }
845
846   if (in_boundary_id == b_id) {
847     std::vector<InterfaceDofData> surface_dofs_unsorted(0);
848     std::cout « "This should never be called in HSIESurface" « std::endl;
849     return surface_dofs_unsorted;
850   }
851   std::vector<InterfaceDofData> surface_dofs_unsorted;
852   std::vector<unsigned int> vertex_ids = get_vertices_for_boundary_id(in_boundary_id);
853   std::vector<unsigned int> line_ids = get_lines_for_boundary_id(in_boundary_id);
854   std::vector<Position> vertex_positions = vertex_positions_for_ids(vertex_ids);
855   std::vector<Position> line_positions = line_positions_for_ids(line_ids);
856   for(unsigned int index = 0; index < vertex_dof_data.size(); index++) {
857     DofData dof = vertex_dof_data[index];
858     for(unsigned int index_in_ids = 0; index_in_ids < vertex_ids.size(); index_in_ids++) {
859       if(vertex_ids[index_in_ids] == vertex_dof_data[index].base_structure_id_non_face) {
860         InterfaceDofData new_item;
861         new_item.index = dof.global_index;
862         new_item.base_point = vertex_positions[index_in_ids];
863         new_item.order = (dof.inner_order+1) * (dof.nodal_basis + 1);
864         surface_dofs_unsorted.push_back(new_item);
865       }
866     }
867   }
868
869   // Construct containers with base points, orientation and index
870   for(unsigned int index = 0; index < edge_dof_data.size(); index++) {
871     DofData dof = edge_dof_data[index];
872     for(unsigned int index_in_ids = 0; index_in_ids < line_ids.size(); index_in_ids++) {
873       if(line_ids[index_in_ids] == edge_dof_data[index].base_structure_id_non_face) {
874         InterfaceDofData new_item;
875         new_item.index = dof.global_index;
876         new_item.base_point = line_positions[index_in_ids];
877         new_item.order = (dof.inner_order+1) * (dof.nodal_basis + 1);
878         surface_dofs_unsorted.push_back(new_item);
879       }
880     }
881   }
882
883   // Sort the vectors.
884   std::sort(surface_dofs_unsorted.begin(), surface_dofs_unsorted.end(),
       compareDofBaseDataAndOrientation);
885
886   return surface_dofs_unsorted;
887 }
```

References get_dof_association(), get_lines_for_boundary_id(), get_vertices_for_boundary_id(), line_positions_for_ids(), and vertex_positions_for_ids().

Referenced by compute_non_owned_dofs(), and finish_dof_index_initialization().

4.36.3.27 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.

```
83                                                                                    {
84   DofDataVector ret;
85   for (unsigned int index = 0; index < edge_dof_data.size(); index++) {
86     if ((edge_dof_data[index].base_dof_index == in_index)
87         && (edge_dof_data[index].type != DofType::RAY
88             && edge_dof_data[index].type != DofType::IFFb)) {
89       ret.push_back(edge_dof_data[index]);
90     }
91   }
92   for (unsigned int index = 0; index < vertex_dof_data.size(); index++) {
93     if ((vertex_dof_data[index].base_dof_index == in_index)
94         && (vertex_dof_data[index].type != DofType::RAY
95             && vertex_dof_data[index].type != DofType::IFFb)) {
96       ret.push_back(vertex_dof_data[index]);
97     }
98   }
99   for (unsigned int index = 0; index < face_dof_data.size(); index++) {
100     if ((face_dof_data[index].base_dof_index == in_index)
101         && (face_dof_data[index].type != DofType::RAY
102             && face_dof_data[index].type != DofType::IFFb)) {
103       ret.push_back(face_dof_data[index]);
104     }
105   }
106   return ret;
107 }
```

4.36.3.28 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.

```
109                                                                                {
110   DofDataVector ret;
111   for (unsigned int index = 0; index < edge_dof_data.size(); index++) {
112     if ((edge_dof_data[index].base_dof_index == in_index)
113         && (edge_dof_data[index].type == DofType::RAY
114             || edge_dof_data[index].type == DofType::IFFb)) {
115       ret.push_back(edge_dof_data[index]);
116     }
117   }
118   for (unsigned int index = 0; index < vertex_dof_data.size(); index++) {
119     if ((vertex_dof_data[index].base_dof_index == in_index)
120         && (vertex_dof_data[index].type == DofType::RAY
121             || vertex_dof_data[index].type == DofType::IFFb)) {
122       ret.push_back(vertex_dof_data[index]);
123     }
124   }
125   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;
}
```

4.36.3.29 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;
}
```

References is_point_at_boundary().

Referenced by get_dof_association_by_boundary_id().

4.36.3.30 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

4.36.3.31 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

4.36.3.32 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.

```
933 {
934   std::vector<unsigned int> vertices;
935   for(auto it = Geometry.surface_meshes[b_id].begin_vertex(); it !=
       Geometry.surface_meshes[b_id].end_vertex(); it++) {
936     if(is_point_at_boundary(it->center(), in_boundary_id)) {
937       vertices.push_back(it->index());
938     }
939   }
940   vertices.shrink_to_fit();
941   return vertices;
942 }
```

References is_point_at_boundary().

Referenced by get_dof_association_by_boundary_id().

4.36.3.33 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.

```
370                                                  {
371   dof_h_q.distribute_dofs(fe_q);
372   dof_h_nedelec.distribute_dofs(fe_nedelec);
373 }
```

Referenced by initialize().

4.36.3.34 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.

```
923                                                                              {
924   if(!boundary_coordinates_computed) {
925     compute_extreme_vertex_coordinates();
926   }
927   if(are_opposing_sites(in_bid, b_id) || in_bid == b_id) return true;
928   Position full_position = undo_transform(in_p);
929   unsigned int component = in_bid / 2;
930   return full_position[component] == boundary_vertex_coordinates[in_bid];
931 }
```

References compute_extreme_vertex_coordinates(), and undo_transform().

Referenced by get_lines_for_boundary_id(), and get_vertices_for_boundary_id().

4.36.3.35 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.

```
832 {
833   std::vector<Position> ret(ids.size());
834   for(unsigned int line_index_in_array = 0; line_index_in_array < ids.size(); line_index_in_array++) {
835     Position p  =
      undo_transform(get_line_position_for_line_index_in_tria(&Geometry.surface_meshes[b_id],
      ids[line_index_in_array]));
836     ret[line_index_in_array] = p;
837   }
838   return ret;
839 }
```

References undo_transform().

Referenced by get_dof_association_by_boundary_id().

4.36.3.36 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.

```
955 {
956   return "";
957 }
```

4.36.3.37 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.

```
533 {
534   dof_counter++;
535   return dof_counter - 1;
536 }
```

Referenced by register_single_dof().

4.36.3.38 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.

```
413
           {
414   const int max_hsie_order = order;
415   // EDGE Dofs
416   std::vector<unsigned int> local_dofs(fe_nedelec.dofs_per_line);
417   cell_nedelec->line(edge)->get_dof_indices(local_dofs);
418   bool orientation = false;
419   if(cell_nedelec->line(edge)->vertex_index(0) > cell_nedelec->line(edge)->vertex_index(1)) {
420     orientation = get_orientation(undo_transform(cell_nedelec->line(edge)->vertex(0)),
       undo_transform(cell_nedelec->line(edge)->vertex(1)));
421   } else {
422     orientation = get_orientation(undo_transform(cell_nedelec->line(edge)->vertex(1)),
       undo_transform(cell_nedelec->line(edge)->vertex(0)));
423   }
424
425   for (int inner_order = 0; inner_order < static_cast<int>(fe_nedelec.dofs_per_line); inner_order++) {
426     register_single_dof(cell_nedelec->face_index(edge), -1, inner_order + 1, DofType::EDGE,
       edge_dof_data, local_dofs[inner_order], orientation);
427     Position bp = undo_transform(cell_nedelec->face(edge)->center(false, false));
428     InterfaceDofData dof_data;
429     dof_data.index = edge_dof_data[edge_dof_data.size() - 1].global_index;
430     dof_data.order = inner_order;
431     dof_data.base_point = bp;
432     add_surface_relevant_dof(dof_data);
433   }
434
435   // INFINITE FACE Dofs Type a
436   for (int inner_order = 0; inner_order < static_cast<int>(fe_nedelec.dofs_per_line); inner_order++) {
437     for (int hsie_order = 0; hsie_order <= max_hsie_order; hsie_order++) {
438       register_single_dof(cell_nedelec->face_index(edge), hsie_order, inner_order + 1, DofType::IFFa,
       edge_dof_data, local_dofs[inner_order], orientation);
439     }
440   }
441   // INFINITE FACE Dofs Type b
442   local_dofs.clear();
443   local_dofs.resize(fe_q.dofs_per_line + 2 * fe_q.dofs_per_vertex);
444   cell_q->line(edge)->get_dof_indices(local_dofs);
445   IndexSet line_dofs(MAX_DOF_NUMBER);
446   IndexSet non_line_dofs(MAX_DOF_NUMBER);
447   for (unsigned int i = 0; i < local_dofs.size(); i++) {
448     line_dofs.add_index(local_dofs[i]);
449   }
450   for (unsigned int i = 0; i < fe_q.dofs_per_vertex; i++) {
451     non_line_dofs.add_index(cell_q->line(edge)->vertex_dof_index(0, i));
452     non_line_dofs.add_index(cell_q->line(edge)->vertex_dof_index(1, i));
453   }
454   line_dofs.subtract_set(non_line_dofs);
455   for (int inner_order = 0; inner_order < static_cast<int>(line_dofs.n_elements());
456        inner_order++) {
457     for (int hsie_order = -1; hsie_order <= max_hsie_order; hsie_order++) {
458       register_single_dof(cell_q->face_index(edge), hsie_order, inner_order, DofType::IFFb,
       edge_dof_data, line_dofs.nth_index_in_set(inner_order), orientation);
459     }
460   }
461 }
```

References add_surface_relevant_dof(), register_single_dof(), and undo_transform().

Referenced by compute_n_edge_dofs().

4.36.3.39 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.

```
463                                                                                         {
464   const int max_hsie_order = order;
465   std::vector<unsigned int> surface_dofs(fe_nedelec.dofs_per_cell);
466   cell_nedelec->get_dof_indices(surface_dofs);
467   IndexSet surf_dofs(MAX_DOF_NUMBER);
468   IndexSet edge_dofs(MAX_DOF_NUMBER);
469   for (unsigned int i = 0; i < surface_dofs.size(); i++) {
470     surf_dofs.add_index(surface_dofs[i]);
471   }
472   for (unsigned int i = 0; i < dealii::GeometryInfo<2>::lines_per_cell; i++) {
473     std::vector<unsigned int> line_dofs(fe_nedelec.dofs_per_line);
474     cell_nedelec->line(i)->get_dof_indices(line_dofs);
475     for (unsigned int j = 0; j < line_dofs.size(); j++) {
476       edge_dofs.add_index(line_dofs[j]);
477     }
478   }
479   surf_dofs.subtract_set(edge_dofs);
480   std::string id = cell_q->id().to_string();
481   const unsigned int nedelec_dof_count = dof_h_nedelec.n_dofs();
482   dealii::Vector<ComplexNumber> vec_temp(nedelec_dof_count);
483   // SURFACE functions
484   for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
485     register_single_dof(cell_nedelec->id().to_string(), -1, inner_order, DofType::SURFACE,
      face_dof_data, surf_dofs.nth_index_in_set(inner_order));
486     Position bp = undo_transform(cell_nedelec->center());
487     InterfaceDofData dof_data;
488     dof_data.index = face_dof_data[face_dof_data.size() - 1].global_index;
489     dof_data.base_point = bp;
490     dof_data.order = inner_order;
491     add_surface_relevant_dof(dof_data);
492   }
493 
494   // SEGMENT functions a
495   for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
496     for (int hsie_order = 0; hsie_order <= max_hsie_order; hsie_order++) {
497       register_single_dof(id, hsie_order, inner_order, DofType::SEGMENTa, face_dof_data,
      surf_dofs.nth_index_in_set(inner_order));
498     }
499   }
500 
501   for (unsigned int inner_order = 0; inner_order < surf_dofs.n_elements(); inner_order++) {
502     for (int hsie_order = -1; hsie_order <= max_hsie_order; hsie_order++) {
503       register_single_dof(id, hsie_order, inner_order, DofType::SEGMENTb, face_dof_data,
      surf_dofs.nth_index_in_set(inner_order));
504     }
505   }
506 }
```

References add_surface_relevant_dof(), register_single_dof(), and undo_transform().

Referenced by compute_n_face_dofs().

4.36.3.40 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().

4.36.3.41 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.

Parameters

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_edge_dofs(), register_new_surface_dofs(), and register_new_vertex_dofs().

4.36.3.42 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().

4.36.3.43 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.

4.36.3.44 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.

```
621                                                                          {
622   // The ray direction before transformation is x. This has to be adapted.
623   HSIEPolynomial temp = (*vector)[0];
624   switch (b_id) {
625     case 2:
626       (*vector)[0] = (*vector)[1];
627       (*vector)[1] = temp;
628       break;
629     case 3:
630       (*vector)[0] = (*vector)[1];
631       (*vector)[1] = temp;
632       break;
633     case 4:
634       (*vector)[0] = (*vector)[2];
```

```
      (*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().

4.36.3.45 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.

```
644                                                  {
645   Position ret;
646   ret[0] = inp[0];
647   ret[1] = inp[1];
648   ret[2] = additional_coordinate;
649   switch (b_id) {
650   case 0:
651     ret = Transform_5_to_0(ret);
652     break;
653   case 1:
654     ret = Transform_5_to_1(ret);
655     break;
656   case 2:
657     ret = Transform_5_to_2(ret);
658     break;
659   case 3:
660     ret = Transform_5_to_3(ret);
661     break;
662   case 4:
663     ret = Transform_5_to_4(ret);
664     break;
665   default:
666     break;
667   }
668   return ret;
669 }
```

Referenced by compute_extreme_vertex_coordinates(), fill_matrix(), is_point_at_boundary(), line_positions_for_ids(), register_new_edge_dofs(), register_new_surface_dofs(), and vertex_positions_for_ids().

4.36.3.46 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.

```
671                                                                              {
672   Position ret;
673   ret[0] = inp[0];
674   ret[1] = inp[1];
675   ret[2] = 0;
676   switch (b_id) {
677   case 0:
678     ret = Transform_5_to_0(ret);
679     break;
680   case 1:
681     ret = Transform_5_to_1(ret);
682     break;
683   case 2:
684     ret = Transform_5_to_2(ret);
685     break;
686   case 3:
687     ret = Transform_5_to_3(ret);
688     break;
689   case 4:
690     ret = Transform_5_to_4(ret);
691     break;
692   default:
693     break;
694   }
695   return ret;
696 }
```

4.36.3.47 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.

```
377                                       {
378   const unsigned int dofs_per_edge_all = compute_dofs_per_edge(false);
379   const unsigned int dofs_per_edge_hsie = compute_dofs_per_edge(true);
380   in_dof_count.total += dofs_per_edge_all;
381   in_dof_count.hsie += dofs_per_edge_hsie;
382   in_dof_count.non_hsie += dofs_per_edge_all - dofs_per_edge_hsie;
383 }
```

References compute_dofs_per_edge().

Referenced by compute_n_edge_dofs().

4.36.3.48 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().

4.36.3.49 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().

4.36.3.50 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.

```
823                                                                          {
824   std::vector<Position> ret(ids.size());
825   for(unsigned int vertex_index_in_array = 0; vertex_index_in_array < ids.size();
       vertex_index_in_array++) {
826     Position p =
       undo_transform(get_vertex_position_for_vertex_index_in_tria(&Geometry.surface_meshes[b_id],
       ids[vertex_index_in_array]));
827     ret[vertex_index_in_array] = p;
828   }
829   return ret;
830 }
```

References undo_transform().

Referenced by get_dof_association_by_boundary_id().

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

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

4.37 InhomogenousTransformationRectangle 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 (hence inhomogenous space transformation)

```
#include <PredefinedShapeTransformation.h>
```

4.37.1 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 (hence inhomogenous space transformation)

If this kind of boundary condition works stably we will also be able to deal with more general settings (which might for example incorporate angles in between the output and input connector.

Author

Pascal Kraft

Date

28.11.2016

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

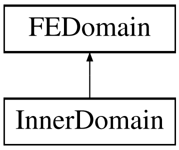
- Code/SpaceTransformations/PredefinedShapeTransformation.h

4.38 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** ()
- void **evaluate** ()
- void **store** ()
- void **make_grid** ()
- void **setup_system** ()
- void **assemble_system** (Constraints ∗constraints, dealii::PETScWrappers::MPI::SparseMatrix ∗matrix, NumericVectorDistributed ∗rhs)
- void **Compute_Dof_Numbers** ()
- void **solution_evaluation** (Position position, double ∗solution) const
- void **adjoint_solution_evaluation** (Position position, double ∗solution) const
- std::vector< InterfaceDofData > **get_surface_dof_vector_for_boundary_id** (BoundaryId b_id)
- void **fill_sparsity_pattern** (dealii::DynamicSparsityPattern ∗in_pattern, Constraints ∗constraints)
- void **write_matrix_and_rhs_metrics** (dealii::PETScWrappers::MatrixBase ∗matrix, NumericVectorDistributed ∗rhs)
- std::string **output_results** (std::string in_filename, NumericVectorLocal in_solution, bool apply_space_transformation)
- void **fill_rhs_vector** (NumericVectorDistributed in_vec, unsigned int level)
- DofCount **compute_n_locally_owned_dofs** () override
- DofCount **compute_n_locally_active_dofs** () override
- void **determine_non_owned_dofs** () override
- ComplexNumber **compute_signal_strength** (dealii::LinearAlgebra::distributed::Vector< ComplexNumber > ∗in_solution)
- ComplexNumber **compute_mode_strength** ()
- FEErrorStruct **compute_errors** (dealii::LinearAlgebra::distributed::Vector< ComplexNumber > ∗in_solution)
- std::vector< std::vector< ComplexNumber > > **evaluate_at_positions** (std::vector< Position > in_positions, NumericVectorLocal in_solution)
- std::vector< FEAdjointEvaluation > **compute_local_shape_gradient_data** (NumericVectorLocal &in_solution)

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**

4.38.1 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.

Author

Pascal Kraft

Date

03.07.2016

Definition at line 80 of file InnerDomain.h.

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

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

4.39 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**

4.39.1 Detailed Description

Definition at line 133 of file Types.h.

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

- Code/Core/Types.h

4.40 JacobianAndTensorData Struct Reference

Public Attributes

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

4.40.1 Detailed Description

Definition at line 158 of file Types.h.

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

- Code/Core/Types.h

4.41 JacobianForCell Class Reference

Public Member Functions

- **JacobianForCell** (FaceAngelingData &in_fad, const BoundaryId &b_id, double additional_component)
- void **reinit_for_cell** (CellIterator2D)
- void **reinit** (FaceAngelingData &in_fad, const BoundaryId &b_id, double additional_component)
- auto **get_C_G_and_J** (Position2D) -> JacobianAndTensorData
- std::pair< Position2D, double > **split_into_triangulation_and_external_part** (const Position in_point)
- dealii::Tensor< 2, 3, double > **get_J_hat_for_position** (const Position2D &) const
- auto **transform_to_3D_space** (Position2D) -> Position

Static Public Member Functions

- static bool **is_line_in_x_direction** (dealii::internal::DoFHandlerImplementation::Iterators< 2, 2, false >↩ ::line_iterator line)
- static bool **is_line_in_y_direction** (dealii::internal::DoFHandlerImplementation::Iterators< 2, 2, false >↩ ::line_iterator line)

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**

4.41.1 Detailed Description

Definition at line 8 of file JacobianForCell.h.

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

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

4.42 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)

4.42.1 Detailed Description

Definition at line 8 of file LaguerreFunction.h.

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

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

4.43 LevelDofIndexData Class Reference

4.43.1 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

4.44 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**

4.44.1 Detailed Description

Definition at line 170 of file Types.h.

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

- Code/Core/Types.h

4.45 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**

4.45.1 Detailed Description

Definition at line 25 of file GeometryManager.h.

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

- Code/GlobalObjects/GeometryManager.h

4.46 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**

4.46.1 Detailed Description

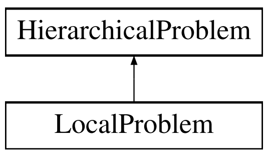
Definition at line 54 of file Types.h.

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

- Code/Core/Types.h

4.47 LocalProblem Class Reference

Inheritance diagram for LocalProblem:



Public Member Functions

- void **solve** () override
- void **initialize** () override
- void **assemble** () override
- void **initialize_index_sets** () override
- void **validate** ()
- auto **reinit** () -> void override
- auto **reinit_rhs** () -> void override
- dealii::IndexSet **compute_interface_dof_set** (BoundaryId interface_id)
- void **compute_solver_factorization** () override
- double **compute_L2_error** ()
- double **compute_error** ()
- unsigned int **compute_global_solve_counter** () override
- void **empty_memory** () override
- void **write_multifile_output** (const std::string &in_filename, bool transform=false) override

Public Attributes

- SolverControl **sc**
- dealii::PETScWrappers::SparseDirectMUMPS **solver**

4.47.1 Detailed Description

Definition at line 13 of file LocalProblem.h.

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

- Code/Hierarchy/LocalProblem.h
- Code/Hierarchy/LocalProblem.cpp

4.48 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** ()

4.48.1 Detailed Description

Definition at line 5 of file ModeManager.h.

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

- Code/GlobalObjects/ModeManager.h
- Code/GlobalObjects/ModeManager.cpp

4.49 MPICommunicator Class Reference

Public Member Functions

- std::pair< bool, unsigned int > **get_neighbor_for_interface** (Direction in_direction)
- void **initialize** ()
- void **destroy_comms** ()

Public Attributes

- std::vector< MPI_Comm > **communicators_by_level**
- std::vector< unsigned int > **rank_on_level**

4.49.1 Detailed Description

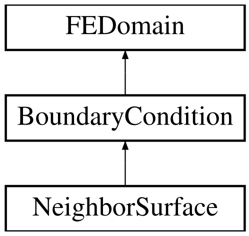
Definition at line 7 of file MPICommunicator.h.

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

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

4.50 NeighborSurface Class Reference

Inheritance diagram for NeighborSurface:



Public Member Functions

- **NeighborSurface** (unsigned int in_bid, unsigned int in_level)
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix ∗, NumericVectorDistributed ∗rhs, Constraints ∗constraints) override

    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.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗in_constriants) override

    If this object owns degrees of freedom, this function fills a sparsity pattern for their global indices.
- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

    Checks if a 2D coordinate is on the a surface of the boundary methods domain.
- void initialize () override

    Not all data for objects of this type will be available at time of construction.
- void **set_mesh_boundary_ids** ()
- auto get_dof_association () -> std::vector< InterfaceDofData > override

    Returns a vector of all degrees of freedom shared with the inner domain.
- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

    More general version of the function above that can also handle interfaces with other boundary ids.
- std::string output_results (const dealii::Vector< ComplexNumber > &, std::string) override

    Writes output for a provided solution to a file with the provided name.
- DofCount **compute_n_locally_owned_dofs** () override
- DofCount **compute_n_locally_active_dofs** () override
- void **determine_non_owned_dofs** () override
- void finish_dof_index_initialization () override

    Handles the communication of non-locally owned dofs and thus finishes the setup of the object.
- void **distribute_dof_indices** ()
- void **send** ()
- void **receive** ()
- void **prepare_dofs** ()

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**

4.50.1 Detailed Description

Definition at line 8 of file NeighborSurface.h.

4.50.2 Member Function Documentation

4.50.2.1 fill_matrix()

```
void NeighborSurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints )  [override], [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.

Implements BoundaryCondition.

Definition at line 31 of file NeighborSurface.cpp.

```
31                   {
32     matrix->compress(dealii::VectorOperation::add); // <-- this operation is collective and therefore
       required.
33     // Nothing to do here, work happens on neighbor process.
34 }
```

4.50.2.2 fill_sparsity_pattern()

```
void NeighborSurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * constraints )  [override], [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

Implements BoundaryCondition.

Definition at line 68 of file NeighborSurface.cpp.

```
68                                                                                          {
69 }
```

4.50.2.3 finish_dof_index_initialization()

```
void NeighborSurface::finish_dof_index_initialization ( )  [override], [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 from BoundaryCondition.

Definition at line 83 of file NeighborSurface.cpp.
```
83                                                          {
84     prepare_dofs();
85     if(is_lower_interface) {
86         receive();
87     } else {
88         send();
89     }
90 }
```

4.50.2.4 get_dof_association()

`std::vector< InterfaceDofData > NeighborSurface::get_dof_association ( ) -> std::vector< InterfaceDofData > [override], [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.

Implements BoundaryCondition.

Definition at line 44 of file NeighborSurface.cpp.
```
44                                                                     {
45     std::vector<InterfaceDofData> dof_indices =
       Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
46     for(unsigned int i = 0; i < dof_indices.size(); i++) {
47         dof_indices[i].index = inner_dofs[i];
48     }
49     return dof_indices;
50 }
```

4.50.2.5 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]`

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.

Implements BoundaryCondition.

Definition at line 52 of file NeighborSurface.cpp.

```
52
       {
53     std::vector<InterfaceDofData> own_dof_indices;
54     for(unsigned int i = 0; i < boundary_dofs[in_boundary_id].size(); i++) {
55         InterfaceDofData idd;
56         idd.order = 0;
57         idd.base_point = {0,0,0};
58         idd.index = boundary_dofs[in_boundary_id][i];
59         own_dof_indices.push_back(idd);
60     }
61     return own_dof_indices;
62 }
```

4.50.2.6 initialize()

`void NeighborSurface::initialize ( ) [override], [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.

Implements BoundaryCondition.

Definition at line 40 of file NeighborSurface.cpp.

```
40                                   {
41
42 }
```

4.50.2.7 is_point_at_boundary()

```
bool NeighborSurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid ) [override], [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.

Implements BoundaryCondition.

Definition at line 36 of file NeighborSurface.cpp.

```
36                                                                    {
37     return false;
38 }
```

4.50.2.8 output_results()

```
std::string NeighborSurface::output_results (
            const dealii::Vector< ComplexNumber > & in_solution,
            std::string filename ) [override], [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.

Implements BoundaryCondition.

Definition at line 64 of file NeighborSurface.cpp.

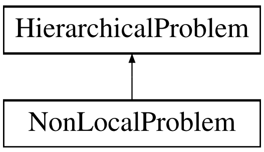
```
64                                                                                    {
65     return "";
66 }
```

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

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

4.51 NonLocalProblem Class Reference

Inheritance diagram for NonLocalProblem:



Public Member Functions

- **NonLocalProblem** (unsigned int)
- void **prepare_sweeping_data** ()
- void **assemble** () override
- void **solve** () override
- void **apply_sweep** (Vec x_in, Vec x_out)
- void **init_solver_and_preconditioner** ()
- void **initialize** () override
- void **initialize_index_sets** () override
- void **reinit** () override
- void **compute_solver_factorization** () override
- void **reinit_rhs** () override
- void **S_inv** (NumericVectorDistributed ∗src, NumericVectorDistributed ∗dst)

- auto **set_x_out_from_u** (Vec x_out) -> void
- std::string **output_results** ()
- void **write_multifile_output** (const std::string &filename, bool apply_coordinate_transform) override
- void **communicate_external_dsp** (DynamicSparsityPattern ∗in_dsp)
- void **make_sparsity_pattern** () override
- void **set_u_from_vec_object** (Vec in_v)
- void **set_vector_from_child_solution** (NumericVectorDistributed ∗)
- void **set_child_rhs_from_vector** (NumericVectorDistributed ∗)
- void **print_vector_norm** (NumericVectorDistributed ∗, std::string marker)
- void **perform_downward_sweep** ()
- void **perform_upward_sweep** ()
- void **complex_pml_domain_matching** (BoundaryId in_bid)
- void **register_dof_copy_pair** (DofNumber own_index, DofNumber child_index)
- ComplexNumber **compute_signal_strength_of_solution** ()
- void **update_shared_solution_vector** ()
- FEErrorStruct **compute_global_errors** (dealii::LinearAlgebra::distributed::Vector< ComplexNumber > ∗in_solution)
- void **update_convergence_criterion** (double last_residual) override
- unsigned int **compute_global_solve_counter** () override
- void **reinit_all_vectors** ()
- unsigned int **n_total_cells** ()
- double **compute_h** ()
- unsigned int **compute_total_number_of_dofs** ()
- std::vector< std::vector< ComplexNumber > > **evaluate_solution_at** (std::vector< Position >)
- void **empty_memory** () override
- std::vector< double > **compute_shape_gradient** () override

Additional Inherited Members

4.51.1 Detailed Description

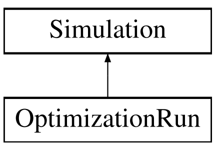
Definition at line 14 of file NonLocalProblem.h.

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

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

4.52 OptimizationRun Class Reference

Inheritance diagram for OptimizationRun:



Public Member Functions

- void **prepare** () override
- void **run** () override
- void **prepare_transformed_geometry** () override
- double **compute_step** ()

Static Public Member Functions

- static double **perform_step** (const dealii::Vector$<$ double $>$ &x, dealii::Vector$<$ double $>$ &g)
- static void **solve_main_problem** ()
- static void **set_shape_dofs** (const dealii::Vector$<$ double $>$ in_shape_dofs)

4.52.1 Detailed Description

Definition at line 8 of file OptimizationRun.h.

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

- Code/Runners/OptimizationRun.h
- Code/Runners/OptimizationRun.cpp

4.53 OutputManager Class Reference

Public Member Functions

- void **initialize** ()
- std::string **get_full_filename** (std::string filename)
- std::string **get_numbered_filename** (std::string filename, unsigned int number, std::string extension)
- void **write_log_ling** (std::string in_line)
- void **write_run_description** (std::string git_commit_hash)

Public Attributes

- std::string **base_path**
- unsigned int **run_number**
- std::string **output_folder_path**
- std::ofstream **log_stream**

4.53.1 Detailed Description

Definition at line 8 of file OutputManager.h.

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

- Code/GlobalObjects/OutputManager.h
- Code/GlobalObjects/OutputManager.cpp

4.54 ParameterOverride Class Reference

Public Member Functions

- bool **read** (std::string)
- void **perform_on** (Parameters &)
- bool **validate** (std::string)

Public Attributes

- bool **has_overrides**

4.54.1 Detailed Description

Definition at line 5 of file ParameterOverride.h.

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

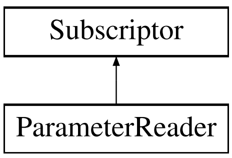
- Code/Helpers/ParameterOverride.h
- Code/Helpers/ParameterOverride.cpp

4.55 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.

4.55.1 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 29 of file ParameterReader.h.

4.55.2 Constructor & Destructor Documentation

4.55.2.1 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 { }
```

4.55.3 Member Function Documentation

4.55.3.1 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|Jump"), "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

4.56 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::Jump
- 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

4.56.1 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

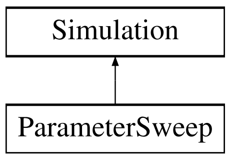
Definition at line 18 of file Parameters.h.

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

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

4.57 ParameterSweep Class Reference

Inheritance diagram for ParameterSweep:



Public Member Functions

- void **prepare** () override
- void **run** () override
- void **prepare_transformed_geometry** () override

4.57.1 Detailed Description

Definition at line 9 of file ParameterSweep.h.

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

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

4.58 PMLMeshTransformation Struct Reference

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
- Position **undo_transform** (const Position &in_p)

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**

4.58.1 Detailed Description

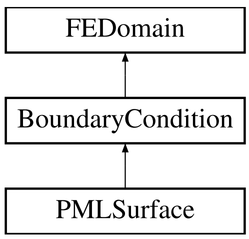
Definition at line 6 of file PMLMeshTransformation.h.

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

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

4.59 PMLSurface Class Reference

Inheritance diagram for PMLSurface:



Public Member Functions

- **PMLSurface** (unsigned int in_bid, unsigned int in_level)
- bool **is_point_at_boundary** (Position, BoundaryId)
- auto make_constraints () -> Constraints override

  Builds a constraint object that represents fixed values of degrees of freedom associated with this object.
- void fill_matrix (dealii::PETScWrappers::MPI::SparseMatrix ∗, NumericVectorDistributed ∗rhs, Constraints ∗constraints) override

  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.
- void fill_sparsity_pattern (dealii::DynamicSparsityPattern ∗in_dsp, Constraints ∗in_constriants) override

  If this object owns degrees of freedom, this function fills a sparsity pattern for their global indices.
- bool is_point_at_boundary (Position2D in_p, BoundaryId in_bid) override

  Checks if a 2D coordinate is on the a surface of the boundary methods domain.
- bool **is_position_at_boundary** (const Position in_p, const BoundaryId in_bid)
- bool **is_position_at_extended_boundary** (const Position in_p, const BoundaryId in_bid)
- void initialize () override

  Not all data for objects of this type will be available at time of construction.
- void **set_mesh_boundary_ids** ()

- void **prepare_mesh** ()
- auto cells_for_boundary_id (unsigned int boundary_id) -> unsigned int override

  Counts the number of cells associated with the boundary passed in as an argument.
- void **init_fe** ()
- auto **fraction_of_pml_direction** (Position) -> std::array< double, 3 >
- auto **get_pml_tensor_epsilon** (Position) -> dealii::Tensor< 2, 3, ComplexNumber >
- auto **get_pml_tensor_mu** (Position) -> dealii::Tensor< 2, 3, ComplexNumber >
- auto **get_pml_tensor** (Position) -> dealii::Tensor< 2, 3, ComplexNumber >
- auto get_dof_association () -> std::vector< InterfaceDofData > override

  Returns a vector of all degrees of freedom shared with the inner domain.
- auto get_dof_association_by_boundary_id (BoundaryId in_boundary_id) -> std::vector< InterfaceDofData > override

  More general version of the function above that can also handle interfaces with other boundary ids.
- void **compute_coordinate_ranges** (dealii::Triangulation< 3 > *in_tria)
- void **set_boundary_ids** ()
- void **fix_apply_negative_Jacobian_transformation** (dealii::Triangulation< 3 > *in_tria)
- std::string output_results (const dealii::Vector< ComplexNumber > &, std::string) override

  Writes output for a provided solution to a file with the provided name.
- void **validate_meshes** ()
- DofCount **compute_n_locally_owned_dofs** () override
- DofCount **compute_n_locally_active_dofs** () override
- void finish_dof_index_initialization () override

  Handles the communication of non-locally owned dofs and thus finishes the setup of the object.
- void **determine_non_owned_dofs** () override
- dealii::IndexSet **compute_non_owned_dofs** ()
- bool **finish_initialization** (DofNumber first_own_index) override
- bool **mg_process_edge** (dealii::Triangulation< 3 > *current_list, BoundaryId b_id)
- bool **mg_process_corner** (dealii::Triangulation< 3 > *current_list, BoundaryId first_bid, BoundaryId second_bid)
- bool **extend_mesh_in_direction** (BoundaryId in_bid)
- void **prepare_dof_associations** ()
- unsigned int n_cells () override

  Counts the number of cells used in the object.

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**

4.59.1 Detailed Description

Definition at line 9 of file PMLSurface.h.

4.59.2 Member Function Documentation

4.59.2.1 cells_for_boundary_id()

```
unsigned int PMLSurface::cells_for_boundary_id (
            unsigned int boundary_id ) -> unsigned int  [override], [virtual]
```

Counts the number of cells 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 from BoundaryCondition.

Definition at line 127 of file PMLSurface.cpp.

```
127                                                                                 {
128     unsigned int ret = 0;
129     for(auto it = triangulation.begin(); it!= triangulation.end(); it++) {
130       if(it->at_boundary()) {
131         for(unsigned int i = 0; i < 6; i++) {
132           if(it->face(i)->boundary_id() == in_boundary_id) {
133             ret++;
134           }
135         }
136       }
137     }
138     return ret;
139 }
```

4.59.2.2 fill_matrix()

```
void PMLSurface::fill_matrix (
            dealii::PETScWrappers::MPI::SparseMatrix * matrix,
            NumericVectorDistributed * rhs,
            Constraints * constraints )  [override], [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.

Implements BoundaryCondition.

Definition at line 458 of file PMLSurface.cpp.

```
458
                               {
459   CellwiseAssemblyDataPML cell_data(&fe_nedelec, &dof_handler);
460   for (; cell_data.cell != cell_data.end_cell; ++cell_data.cell) {
461     cell_data.cell->get_dof_indices(cell_data.local_dof_indices);
462     cell_data.local_dof_indices = transform_local_to_global_dofs(cell_data.local_dof_indices);
463     cell_data.cell_rhs.reinit(cell_data.dofs_per_cell, false);
464     cell_data.fe_values.reinit(cell_data.cell);
465     cell_data.quadrature_points = cell_data.fe_values.get_quadrature_points();
466     std::vector<types::global_dof_index> input_dofs(fe_nedelec.dofs_per_line);
467     IndexSet input_dofs_local_set(fe_nedelec.dofs_per_cell);
468     std::vector<Position> input_dof_centers(fe_nedelec.dofs_per_cell);
469     std::vector<Tensor<1, 3, double» input_dof_dirs(fe_nedelec.dofs_per_cell);
470     cell_data.cell_matrix = 0;
471     for (unsigned int q_index = 0; q_index < cell_data.n_q_points; ++q_index) {
472       Position pos = cell_data.get_position_for_q_index(q_index);
473       dealii::Tensor<2,3,ComplexNumber> epsilon = get_pml_tensor_epsilon(pos);
474       dealii::Tensor<2,3,double> J = GlobalSpaceTransformation->get_J(pos);
475
476       epsilon = J * epsilon * transpose(J) / GlobalSpaceTransformation->get_det(pos);
477       dealii::Tensor<2,3,ComplexNumber> mu = get_pml_tensor_mu(pos);
478       mu = invert(J * mu * transpose(J) / GlobalSpaceTransformation->get_det(pos));
479       cell_data.prepare_for_current_q_index(q_index, epsilon, mu);
480     }
481     constraints->distribute_local_to_global(cell_data.cell_matrix, cell_data.cell_rhs,
     cell_data.local_dof_indices,*matrix, *rhs, true);
482   }
483   matrix->compress(dealii::VectorOperation::add);
484 }
```

4.59.2.3 fill_sparsity_pattern()

```
void PMLSurface::fill_sparsity_pattern (
            dealii::DynamicSparsityPattern * in_dsp,
            Constraints * constraints )  [override], [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

Implements BoundaryCondition.

Definition at line 449 of file PMLSurface.cpp.

```
449
     {
450   std::vector<unsigned int> local_indices(fe_nedelec.dofs_per_cell);
451   for(auto it = dof_handler.begin_active(); it != dof_handler.end(); it++) {
452     it->get_dof_indices(local_indices);
453     local_indices = transform_local_to_global_dofs(local_indices);
454     in_constraints->add_entries_local_to_global(local_indices, *in_dsp);
455   }
456 }
```

4.59.2.4 finish_dof_index_initialization()

`void PMLSurface::finish_dof_index_initialization ( )  [override], [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 from BoundaryCondition.

Definition at line 678 of file PMLSurface.cpp.

```
678                                                {
679   for(unsigned int surf = 0; surf < 6; surf++) {
680     if(surf != b_id && !are_opposing_sites(surf, b_id)) {
681       if(!are_edge_dofs_owned[surf] && Geometry.levels[level].surface_type[surf] !=
     SurfaceType::NEIGHBOR_SURFACE) {
682         DofIndexVector dofs_in_global_numbering =
     Geometry.levels[level].surfaces[surf]->get_global_dof_indices_by_boundary_id(b_id);
683         std::vector<InterfaceDofData> local_interface_data = get_dof_association_by_boundary_id(surf);
684         DofIndexVector dofs_in_local_numbering(local_interface_data.size());
685         for(unsigned int i = 0; i < local_interface_data.size(); i++) {
686           dofs_in_local_numbering[i] = local_interface_data[i].index;
687         }
688         set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
689       }
690     }
691   }
692   // Do the same for the inner interface
693   std::vector<InterfaceDofData> global_interface_data =
     Geometry.levels[level].inner_domain->get_surface_dof_vector_for_boundary_id(b_id);
694   std::vector<InterfaceDofData> local_interface_data =
     get_dof_association_by_boundary_id(inner_boundary_id);
695   DofIndexVector dofs_in_local_numbering(local_interface_data.size());
696   DofIndexVector dofs_in_global_numbering(local_interface_data.size());
697
698   for(unsigned int i = 0; i < local_interface_data.size(); i++) {
699     dofs_in_local_numbering[i] = local_interface_data[i].index;
700     dofs_in_global_numbering[i] =
     Geometry.levels[level].inner_domain->global_index_mapping[global_interface_data[i].index];
701   }
702   set_non_local_dof_indices(dofs_in_local_numbering, dofs_in_global_numbering);
703 }
```

References get_dof_association_by_boundary_id().

4.59.2.5 get_dof_association()

std::vector< InterfaceDofData > PMLSurface::get_dof_association ( ) -> std::vector< InterfaceDofData > [override], [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.

Implements BoundaryCondition.

Definition at line 324 of file PMLSurface.cpp.

```
324                                                          {
325     return get_dof_association_by_boundary_id(inner_boundary_id);
326 }
```

References get_dof_association_by_boundary_id().

4.59.2.6 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]

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.

Implements BoundaryCondition.

Definition at line 320 of file PMLSurface.cpp.
```
                                                                                                 {
  return dof_associations[in_bid];
}
```

Referenced by finish_dof_index_initialization(), get_dof_association(), and make_constraints().

4.59.2.7 initialize()

```
void PMLSurface::initialize ( )  [override], [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.

Implements BoundaryCondition.

Definition at line 247 of file PMLSurface.cpp.
```
                          {
  prepare_mesh();
  init_fe();
  prepare_dof_associations();
}
```

4.59.2.8 is_point_at_boundary()

```
bool PMLSurface::is_point_at_boundary (
            Position2D in_p,
            BoundaryId in_bid )  [override], [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.

Implements BoundaryCondition.

Definition at line 224 of file PMLSurface.cpp.

```
224                                                     {
225   return false;
226 }
```

4.59.2.9 make_constraints()

`Constraints PMLSurface::make_constraints ( ) -> Constraints [override], [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 from BoundaryCondition.

Definition at line 731 of file PMLSurface.cpp.

```
731                                              {
732   IndexSet global_indices = IndexSet(Geometry.levels[level].n_total_level_dofs);
733   global_indices.add_range(0, Geometry.levels[level].n_total_level_dofs);
734   Constraints ret(global_indices);
735   std::vector<InterfaceDofData> dofs = get_dof_association_by_boundary_id(outer_boundary_id);
736   for(auto dof : dofs) {
737         const unsigned int local_index = dof.index;
738         const unsigned int global_index = global_index_mapping[local_index];
739         ret.add_line(global_index);
740         ret.set_inhomogeneity(global_index, ComplexNumber(0,0));
741     }
742   return ret;
743 }
```

References get_dof_association_by_boundary_id().

4.59.2.10 n_cells()

```
unsigned int PMLSurface::n_cells ( ) [override], [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 from BoundaryCondition.

Definition at line 867 of file PMLSurface.cpp.

```
867                                    {
868   return triangulation.n_active_cells();
869 }
```

Referenced by output_results().

4.59.2.11 output_results()

```
std::string PMLSurface::output_results (
            const dealii::Vector< ComplexNumber > & in_solution,
            std::string filename ) [override], [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.

Implements BoundaryCondition.

Definition at line 630 of file PMLSurface.cpp.

```
630                                                                                                    {
631   dealii::DataOut<3> data_out;
632   data_out.attach_dof_handler(dof_handler);
633
634   dealii::Vector<ComplexNumber> zero = dealii::Vector<ComplexNumber>(in_data.size());
635   for(unsigned int i = 0; i < in_data.size(); i++) {
636     zero[i] = 0;
637   }
638
639   const unsigned int n_cells = dof_handler.get_triangulation().n_cells();
640   dealii::Vector<double> eps_abs(n_cells);
641   unsigned int counter = 0;
642   for(auto it = dof_handler.begin(); it != dof_handler.end(); it++) {
643     Position p = it->center();
644     MaterialTensor epsilon = get_pml_tensor_epsilon(p);
645     eps_abs[counter] = epsilon.norm();
646     counter++;
647   }
648
649   data_out.add_data_vector(in_data, "Solution");
650   data_out.add_data_vector(eps_abs, "Epsilon");
651    dealii::Vector<double> index_x(n_cells), index_y(n_cells), index_z(n_cells);
652   for(unsigned int i = 0; i < n_cells; i++) {
653     index_x[i] = GlobalParams.Index_in_x_direction;
654     index_y[i] = GlobalParams.Index_in_y_direction;
655     index_z[i] = GlobalParams.Index_in_z_direction;
656   }
657   data_out.add_data_vector(index_x, "IndexX");
658   data_out.add_data_vector(index_y, "IndexY");
659   data_out.add_data_vector(index_z, "IndexZ");
660   data_out.add_data_vector(zero, "Exact_Solution");
661   data_out.add_data_vector(zero, "SolutionError");
662   const std::string filename = GlobalOutputManager.get_numbered_filename(in_filename + "-" +
       std::to_string(b_id) + "-", GlobalParams.MPI_Rank, "vtu");
663   std::ofstream outputvtu(filename);
664   data_out.build_patches();
665   data_out.write_vtu(outputvtu);
666   return filename;
667 }
```

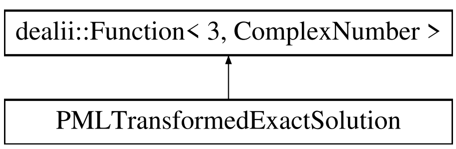
References n_cells().

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

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

4.60 PMLTransformedExactSolution Class Reference

Inheritance diagram for PMLTransformedExactSolution:



Public Member Functions

- **PMLTransformedExactSolution** (BoundaryId in_main_id, double in_additional_coordinate)
- std::vector< std::string > **split** (std::string) const
- ComplexNumber **value** (const Position &p, const unsigned int component) const
- void **vector_value** (const Position &p, dealii::Vector< ComplexNumber > &value) const
- dealii::Tensor< 1, 3, ComplexNumber > curl (const Position &in_p) const
- dealii::Tensor< 1, 3, ComplexNumber > **val** (const Position &in_p) const
- std::array< double, 3 > **fraction_of_pml_direction** (const Position &in_p) const
- double **compute_scaling_factor** (const Position &in_p) const

4.60.1 Detailed Description

Definition at line 12 of file PMLTransformedExactSolution.h.

4.60.2 Member Function Documentation

4.60.2.1 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.

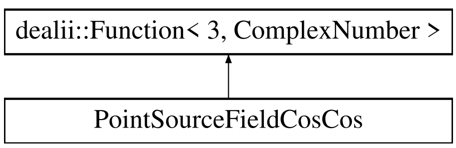
```
48                                                                    {
49   dealii::Tensor<1, 3, ComplexNumber> ret;
50   /**
51   NumericVectorLocal curls = base_solution->curl(in_p);
52   double scaling_factor = compute_scaling_factor(in_p);
53   for(unsigned int i = 0; i < 3; i++) {
54     ret[i] *= scaling_factor;
55   }
56   **/
57   return ret;
58 }
```

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

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

4.61 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)

4.61.1 Detailed Description

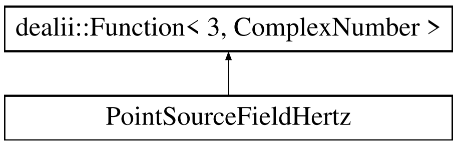
Definition at line 19 of file PointSourceField.h.

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

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

4.62 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

4.62.1 Detailed Description

Definition at line 6 of file PointSourceField.h.

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

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

4.63 PointVal Class Reference

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**

4.63.1 Detailed Description

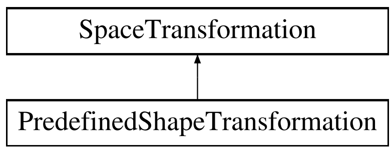
Definition at line 4 of file PointVal.h.

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

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

4.64 PredefinedShapeTransformation Class Reference

Inheritance diagram for PredefinedShapeTransformation:



Public Member Functions

- Position **math_to_phys** (Position coord) const
- Position **phys_to_math** (Position coord) const
- dealii::Tensor< 2, 3, ComplexNumber > **get_Tensor** (Position &coordinate)
- dealii::Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Position &coordinate)
- Tensor< 2, 3, double > **get_J** (Position &) override
- Tensor< 2, 3, double > **get_J_inverse** (Position &) override
- 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.

4.64.1 Detailed Description

Definition at line 24 of file PredefinedShapeTransformation.h.

4.64.2 Member Function Documentation

4.64.2.1 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 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 53 of file PredefinedShapeTransformation.cpp.

```
53                                                              {
54   print_info("PredefinedShapeTransformation::estimate_and_initialize", "Start");
55   Sector<2> the_first(true, false, GlobalParams.sd.z[0], GlobalParams.sd.z[1]);
56   the_first.set_properties_force(GlobalParams.sd.m[0], GlobalParams.sd.m[1],
57                                  GlobalParams.sd.v[0], GlobalParams.sd.v[1]);
58   case_sectors.push_back(the_first);
59   for (int i = 1; i < GlobalParams.sd.Sectors - 2; i++) {
60       Sector<2> intermediate(false, false, GlobalParams.sd.z[i], GlobalParams.sd.z[i + 1]);
61       intermediate.set_properties_force(
62           GlobalParams.sd.m[i], GlobalParams.sd.m[i + 1], GlobalParams.sd.v[i],
63           GlobalParams.sd.v[i + 1]);
64       case_sectors.push_back(intermediate);
65   }
66   Sector<2> the_last(false, true,
67                       GlobalParams.sd.z[GlobalParams.sd.Sectors - 2],
68                       GlobalParams.sd.z[GlobalParams.sd.Sectors - 1]);
69   the_last.set_properties_force(
70       GlobalParams.sd.m[GlobalParams.sd.Sectors - 2],
71       GlobalParams.sd.m[GlobalParams.sd.Sectors - 1],
72       GlobalParams.sd.v[GlobalParams.sd.Sectors - 2],
73       GlobalParams.sd.v[GlobalParams.sd.Sectors - 1]);
74   case_sectors.push_back(the_last);
75   if(GlobalParams.MPI_Rank == 0) {
76     for (unsigned int i = 0; i < case_sectors.size(); i++) {
77       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)) +
     ")";
78       print_info("PredefinedShapeTransformation::estimate_and_initialize", msg_lower);
79     }
80     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)) + ")";
81
82   }
83   print_info("PredefinedShapeTransformation::estimate_and_initialize", "End");
84 }
```

References case_sectors, get_m(), get_v(), and Sector< Dofs_Per_Sector >::set_properties_force().

4.64.3 Member Data Documentation

4.64.3.1 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 47 of file PredefinedShapeTransformation.h.

Referenced by estimate_and_initialize(), get_m(), and get_v().

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

- Code/SpaceTransformations/PredefinedShapeTransformation.h
- Code/SpaceTransformations/PredefinedShapeTransformation.cpp

4.65 RayAngelingData Struct Reference

Public Attributes

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

4.65.1 Detailed Description

Definition at line 111 of file Types.h.

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

- Code/Core/Types.h

4.66 RectangularMode Class Reference

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**

4.66.1 Detailed Description

Definition at line 47 of file RectangularMode.h.

4.66.2 Member Function Documentation

4.66.2.1 solve()

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

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

Definition at line 281 of file RectangularMode.cpp.

```
281                            {
282   print_info("RectangularProblem::solve", "Start");
283   dealii::SolverControl                  solver_control(n_dofs_total, 1e-6);
284   // dealii::SLEPcWrappers::SolverKrylovSchur eigensolver(solver_control);
285   IndexSet own_dofs(n_dofs_total);
286   own_dofs.add_range(0, n_dofs_total);
287   eigenfunctions.resize(n_eigenfunctions);
288   for (unsigned int i = 0; i < n_eigenfunctions; ++i)
289     eigenfunctions[i].reinit(own_dofs, MPI_COMM_SELF);
290   eigenvalues.resize(n_eigenfunctions);
291   // eigensolver.set_which_eigenpairs(EPS_SMALLEST_MAGNITUDE);
292   // eigensolver.set_problem_type(EPS_GNHEP);
293   print_info("RectangularProblem::solve", "Starting solution for a system with " +
       std::to_string(n_dofs_total) + " degrees of freedom.");
```

```
/**
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

4.67 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** ()

4.67.1 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

4.68 SampleShellPC Struct Reference

Public Attributes

- NonLocalProblem * **parent**

4.68.1 Detailed Description

Definition at line 71 of file HierarchicalProblem.h.

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

- Code/Hierarchy/HierarchicalProblem.h

4.69 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**

4.69.1 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 14 of file Sector.h.

4.69.2 Constructor & Destructor Documentation

4.69.2.1 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;
}
```

4.69.3 Member Function Documentation

4.69.3.1 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.

```
146                                                                          {
147   if (i > 0 && i < NDofs) {
148     if (z < 0.0) z = 0.0;
149     if (z > 1.0) z = 1.0;
150     if (zero_derivative[i]) {
151       return InterpolationPolynomialZeroDerivative(z, dofs_l[i], dofs_r[i]);
152     } else {
153       return InterpolationPolynomial(z, dofs_l[i], dofs_r[i],
154                                      dofs_l[derivative[i]],
155                                      dofs_r[derivative[i]]);
156     }
157   } else {
158     print_info("Sector<Dofs_Per_Sector>::get_dof", "There seems to be an error in Sector::get_dof. i > 0 && i < dofs_per_sector false.", LoggingLevel::PRODUCTION_ALL);
159     return 0;
160   }
161 }
```

4.69.3.2 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.

```
175                                                      {
176   if (z < 0.0) z = 0.0;
177   if (z > 1.0) z = 1.0;
178   if (Dofs_Per_Sector == 2) {
179     return InterpolationPolynomial(z, dofs_l[0], dofs_r[0], dofs_l[1],
180                                    dofs_r[1]);
181   } else {
182     return InterpolationPolynomial(z, dofs_l[1], dofs_r[1], dofs_l[2],
183                                    dofs_r[2]);
184   }
185 }
```

4.69.3.3 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.

```
164                                              {
165   if (z < 0.0) z = 0.0;
166   if (z > 1.0) z = 1.0;
167   if (Dofs_Per_Sector < 3) {
168     print_info("Sector<Dofs_Per_Sector>::get_r", "Error in Sector: Access to radius dof without
       existence.", LoggingLevel::PRODUCTION_ALL);
169     return 0;
170   }
171   return InterpolationPolynomialZeroDerivative(z, dofs_l[0], dofs_r[0]);
172 }
```

4.69.3.4 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.

```
188                                              {
189   if (z < 0.0) z = 0.0;
190   if (z > 1.0) z = 1.0;
191   if (Dofs_Per_Sector == 2) {
192     return InterpolationPolynomialZeroDerivative(z, dofs_l[1], dofs_r[1]);
193   } else {
194     return InterpolationPolynomialZeroDerivative(z, dofs_l[2], dofs_r[2]);
195   }
196 }
```

4.69.3.5 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.

```
397                                                          {
398   return LowestDof;
399 }
```

4.69.3.6 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.

```
412                                                            {
413   return NActiveCells;
414 }
```

4.69.3.7 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.

```
402                                                       {
403   return NDofs;
404 }
```

4.69.3.8 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.

```
407                                                                   {
408   return NInternalBoundaryDofs;
409 }
```

4.69.3.9 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.

```
199                                              {
200   return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
201             z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
202 }
```

4.69.3.10 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.

```
205                                              {
206   return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
207             z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
208 }
```

4.69.3.11 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.

```
211                                            {
212   return 0.0;
213 }
```

4.69.3.12 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.

```
119                                                                         {
120   print_info("Sector<Dofs_Per_Sector>::set_properties", "The code does not work for this number of dofs
       per Sector.", LoggingLevel::PRODUCTION_ALL);
121   return;
122 }
```

4.69.3.13 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.

```
417                                                                 {
418   LowestDof = inLowestDOF;
419 }
```

4.69.3.14 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.

```
434                                       {
435   NActiveCells = inNumberOfActiveCells;
436 }
```

4.69.3.15 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.

```
422                                                            {
423   NDofs = inNumberOfDOFs;
424 }
```

4.69.3.16 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.

```
428                                        {
429   NInternalBoundaryDofs = in_ninternalboundarydofs;
430 }
```

4.69.3.17 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;
}
```

4.69.4 Member Data Documentation

4.69.4.1 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 55 of file Sector.h.

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

- Code/Core/Sector.h
- Code/Core/Sector.cpp

4.70 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**

4.70.1 Detailed Description

Definition at line 6 of file ShapeDescription.h.

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

- Code/Helpers/ShapeDescription.h
- Code/Helpers/ShapeDescription.cpp

4.71 ShapeFunction Class Reference

Public Member Functions

- **ShapeFunction** (double in_z_min, double in_z_max, unsigned int in_n_sectors)
- double **evaluate_at** (double z) const
- double **evaluate_derivative_at** (double z) const
- void **set_constraints** (double in_f_0, double in_f_1, double in_df_0, double in_df_1)
- void **update_constrained_values** ()
- void **set_free_values** (std::vector< double > in_dof_values)
- unsigned int **get_n_dofs** () const
- unsigned int **get_n_free_dofs** () const
- double **get_dof_value** (unsigned int index) const
- double **get_free_dof_value** (unsigned int index) const
- void **initialize** ()
- void **set_free_dof_value** (unsigned int index, double value)

Static Public Member Functions

- static unsigned int **compute_n_dofs** (unsigned int in_n_sectors)
- static unsigned int **compute_n_free_dofs** (unsigned int in_n_sectors)

Public Attributes

- const unsigned int **n_free_dofs**
- const unsigned int **n_dofs**

4.71.1 Detailed Description

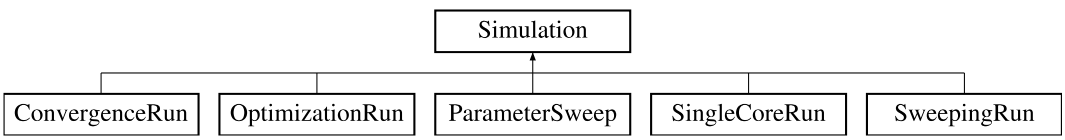
Definition at line 5 of file ShapeFunction.h.

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

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

4.72 Simulation Class Reference

Inheritance diagram for Simulation:



Public Member Functions

- virtual void **prepare** ()=0
- virtual void **run** ()=0
- virtual void **prepare_transformed_geometry** ()=0
- void **create_output_directory** ()

4.72.1 Detailed Description

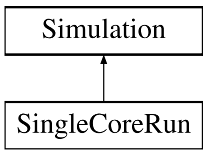
Definition at line 8 of file Simulation.h.

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

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

4.73 SingleCoreRun Class Reference

Inheritance diagram for SingleCoreRun:



Public Member Functions

- void **prepare** () override
- void **run** () override
- void **prepare_transformed_geometry** () override

4.73.1 Detailed Description

Definition at line 8 of file SingleCoreRun.h.

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

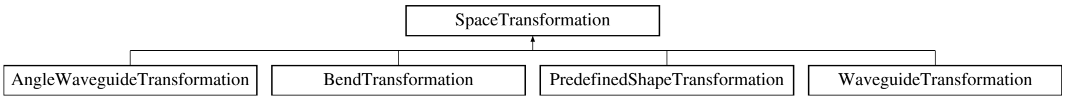
- Code/Runners/SingleCoreRun.h
- Code/Runners/SingleCoreRun.cpp

4.74 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
- virtual Position **phys_to_math** (Position coord) const =0
- virtual double **get_det** (Position)
- virtual Tensor< 2, 3, double > **get_J** (Position &)
- virtual Tensor< 2, 3, double > **get_J_inverse** (Position &)
- virtual Tensor< 2, 3, ComplexNumber > **get_Tensor** (Position &)=0
- virtual Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Position &)=0
- virtual Tensor< 2, 3, ComplexNumber > **get_Tensor_for_step** (Position &coordinate, unsigned int dof, double step_width)
- void **switch_application_mode** (bool apply_math_to_physical)
- 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

Public Attributes

- bool **apply_math_to_phys** = true

4.74.1 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.

4.74.2 Member Function Documentation

4.74.2.1 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, PredefinedShapeTransformation, and AngleWaveguideTransformatio

4.74.2.2 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 93 of file SpaceTransformation.h.

```
93                                   {
94     return 0;
95   };
```

4.74.2.3 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 134 of file SpaceTransformation.h.

```
134                                        {
135     Vector<double> ret;
136     return ret;
137   };
```

4.74.2.4 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 106 of file SpaceTransformation.h.

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

4.74.2.5 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 151 of file SpaceTransformation.h.

```
151                                         {
152     return 0;
153   }
```

4.74.2.6 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 115 of file SpaceTransformation.h.

```
115 {return;};
```

4.74.2.7 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 10 of file SpaceTransformation.cpp.

```
10                                                                                 {
11   std::pair<int, double> ret;
12   ret.first = 0;
13   ret.second = 0.0;
14   if (in_z <= Geometry.global_z_range.first) {
15     ret.first = 0;
16     ret.second = 0.0;
17   } else if (in_z < Geometry.global_z_range.second && in_z > Geometry.global_z_range.first) {
18     ret.first = floor( (in_z + Geometry.global_z_range.first) / (GlobalParams.Sector_thickness));
19     ret.second = (in_z + Geometry.global_z_range.first -  (ret.first * GlobalParams.Sector_thickness)) /
       (GlobalParams.Sector_thickness);
20   } else if (in_z >= Geometry.global_z_range.second) {
21     ret.first = GlobalParams.Number_of_sectors - 1;
22     ret.second = 1.0;
23   }
24
25   if (ret.second < 0 || ret.second > 1){
26     std::cout « "Global ranges: " « Geometry.global_z_range.first « " to " «
       Geometry.global_z_range.second « std::endl;
27     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;
28     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;
29   }
30   return ret;
31 }
```

Referenced by PredefinedShapeTransformation::get_m(), and PredefinedShapeTransformation::get_v().

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

- Code/SpaceTransformations/SpaceTransformation.h
- Code/SpaceTransformations/SpaceTransformation.cpp

4.75 SquareMeshGenerator Class Reference

This class generates meshes, that are used to discretize a rectangular Waveguide. It is derived from Mesh↩ Generator.

`#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**

4.75.1 Detailed Description

This class generates meshes, that are used to discretize a rectangular Waveguide. It is derived from Mesh↩ Generator.

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 23 of file SquareMeshGenerator.h.

4.75.2 Member Function Documentation

4.75.2.1 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.

4.75.2.2 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.

4.75.2.3 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.

```
85                                                                        {
86   GridGenerator::hyper_cube(*in_tria, -1.0, 1.0, false);
87   GridTools::transform(&Triangulation_Shit_To_Local_Geometry, *in_tria);
88   set_boundary_ids(*in_tria);
89
90   in_tria->signals.post_refinement.connect(
91       std::bind(&SquareMeshGenerator::set_boundary_ids,
92           std::cref(*this), std::ref(*in_tria)));
93
94   refine_triangulation_iteratively(in_tria);
95
96   set_boundary_ids(*in_tria);
97 }
```

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

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

4.76 SurfaceCellData Struct Reference

Public Attributes

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

4.76.1 Detailed Description

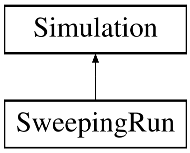
Definition at line 207 of file Types.h.

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

- Code/Core/Types.h

4.77 SweepingRun Class Reference

Inheritance diagram for SweepingRun:



Public Member Functions

- void **prepare** () override
- void **run** () override
- void **prepare_transformed_geometry** () override

4.77.1 Detailed Description

Definition at line 8 of file SweepingRun.h.

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

- Code/Runners/SweepingRun.h
- Code/Runners/SweepingRun.cpp

4.78 tagGSPHERE Struct Reference

Public Attributes

- int **n**
- double ∗ **r**
- double ∗ **t**
- double ∗ **q**
- double ∗ **A**
- double **B**

4.78.1 Detailed Description

Definition at line 23796 of file QuadratureFormulaCircle.cpp.

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

- Code/Helpers/QuadratureFormulaCircle.cpp

4.79 TimerManager Class Reference

Public Member Functions

- void **initialize** ()
- void **switch_context** (std::string context, unsigned int level)
- void **write_output** ()
- void **leave_context** (unsigned int level)

Public Attributes

- std::vector$<$ dealii::TimerOutput $>$ **timer_outputs**
- std::vector$<$ std::string $>$ **filenames**
- std::vector$<$ std::ofstream $*$ $>$ **filestreams**
- unsigned int **level_count**

4.79.1 Detailed Description

Definition at line 6 of file TimerManager.h.

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

- Code/GlobalObjects/TimerManager.h
- Code/GlobalObjects/TimerManager.cpp

4.80 VertexAngelingData Struct Reference

Public Attributes

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

4.80.1 Detailed Description

Definition at line 70 of file Types.h.

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

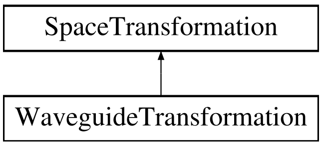
- Code/Core/Types.h

4.81 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
- Position **phys_to_math** (Position coord) const override
- dealii::Tensor< 2, 3, ComplexNumber > **get_Tensor** (Position &coordinate) override
- dealii::Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Position &coordinate) override
- Tensor< 2, 3, double > **get_J** (Position &) override
- Tensor< 2, 3, double > **get_J_inverse** (Position &) override
- 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

4.81.1 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.

If this kind of boundary condition works stably we will also be able to deal with more general settings (which might for example incorporate angles in between the output and input connector.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 29 of file WaveguideTransformation.h.

4.81.2 Member Function Documentation

4.81.2.1 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 130 of file WaveguideTransformation.cpp.

```
130                                                    {
131    vertical_shift.set_constraints(0, GlobalParams.Vertical_displacement_of_waveguide, 0,0);
132    vertical_shift.initialize();
133    if(!GlobalParams.keep_waveguide_height_constant) {
134      waveguide_height.set_constraints(1, 1, 0,0);
135      waveguide_height.initialize();
136    }
137    if(!GlobalParams.keep_waveguide_width_constant) {
138      waveguide_width.set_constraints(1, 1, 0,0);
139      waveguide_height.initialize();
140    }
141  }
```

4.81.2.2 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 69 of file WaveguideTransformation.cpp.

```
69                                                          {
70   std::pair<ResponsibleComponent, unsigned int> comp = map_dof_index(index);
71   switch (comp.first)
72   {
73     case VerticalDisplacementComponent:
74       return vertical_shift.get_dof_value(comp.second);
75       break;
76     case WaveguideHeightComponent:
77       return waveguide_height.get_dof_value(comp.second);
78       break;
79     case WaveguideWidthComponent:
80       return waveguide_width.get_dof_value(comp.second);
81       break;
82     default:
83       break;
84   }
85   return 0.0;
86 }
```

4.81.2.3 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 143 of file WaveguideTransformation.cpp.

```
143                                                              {
144   Vector<double> ret(n_dofs());
145   unsigned int total_counter = 0;
146   for(unsigned int i = 0; i < vertical_shift.get_n_dofs(); i++) {
147     ret[total_counter] = vertical_shift.get_dof_value(i);
148     total_counter ++;
149   }
150   if(!GlobalParams.keep_waveguide_height_constant) {
151     for(unsigned int i = 0; i < waveguide_height.get_n_dofs(); i++) {
152       ret[total_counter] = waveguide_height.get_dof_value(i);
153       total_counter ++;
154     }
155   }
156   if(!GlobalParams.keep_waveguide_width_constant) {
157     for(unsigned int i = 0; i < waveguide_width.get_n_dofs(); i++) {
158       ret[total_counter] = waveguide_width.get_dof_value(i);
159       total_counter ++;
160     }
161   }
162   return ret;
163 }
```

References n_dofs().

4.81.2.4 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 88 of file WaveguideTransformation.cpp.

```
88                                                  {
89   std::pair<ResponsibleComponent, unsigned int> comp = map_free_dof_index(index);
90   switch (comp.first)
91   {
92     case VerticalDisplacementComponent:
93       return vertical_shift.get_free_dof_value(comp.second);
94       break;
95     case WaveguideHeightComponent:
96       return waveguide_height.get_free_dof_value(comp.second);
97       break;
98     case WaveguideWidthComponent:
99       return waveguide_width.get_free_dof_value(comp.second);
100      break;
101    default:
102      break;
103  }
104  return 0.0;
105 }
```

4.81.2.5 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 181 of file WaveguideTransformation.cpp.

```
181                                                   {
182  unsigned int ret = vertical_shift.n_dofs;
183  if(!GlobalParams.keep_waveguide_height_constant) {
184    ret += waveguide_height.n_dofs;
185  }
186  if(!GlobalParams.keep_waveguide_width_constant) {
187    ret += waveguide_width.n_dofs;
188  }
189  return ret;
190 }
```

Referenced by get_dof_values().

4.81.2.6 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 107 of file WaveguideTransformation.cpp.

```
107                                                        {
108    std::pair<ResponsibleComponent, unsigned int> comp = map_free_dof_index(index);
109    switch (comp.first)
110    {
111      case VerticalDisplacementComponent:
112        vertical_shift.set_free_dof_value(comp.second, value);
113        return;
114        break;
115      case WaveguideHeightComponent:
116        waveguide_height.set_free_dof_value(comp.second, value);
117        return;
118        break;
119      case WaveguideWidthComponent:
120        waveguide_width.set_free_dof_value(comp.second, value);
121        return;
122        break;
123      default:
124        break;
125    }
126    std::cout « "There was an error setting a free dof value." « std::endl;
127    return;
128  }
```

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

- Code/SpaceTransformations/WaveguideTransformation.h
- Code/SpaceTransformations/WaveguideTransformation.cpp

Index