Waveguide Solver

2.1

Generated by Doxygen 1.8.13

Contents

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.

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

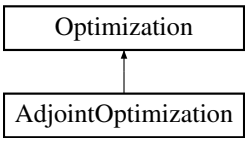
Class Documentation

4.1 AdjointOptimization Class Reference

Derived from the Optimization class, this class implements an Optimization-scheme based on an adjoint method.

`#include <AdjointOptimization.h>`

Inheritance diagram for AdjointOptimization:



Public Member Functions

- **AdjointOptimization** (Waveguide *waveguide_primal, MeshGenerator *mg, SpaceTransformation *st_primal, SpaceTransformation *st_dual, OptimizationAlgorithm< std::complex< double >> *Oa)
- std::vector< std::complex< double > > **compute_small_step** (double step)
- double **compute_big_step** (std::vector< double > step)
- void run ()

  This function is the core implementation of an optimization algorithm.

Public Attributes

- const int **type**
- Waveguide * **waveguide**
- SpaceTransformation * **primal_st**
- SpaceTransformation * **dual_st**
- MeshGenerator * **mg**
- OptimizationAlgorithm< std::complex< double > > * **oa**

4.1.1 Detailed Description

Derived from the Optimization class, this class implements an Optimization-scheme based on an adjoint method.

This method should prove to be far superior to a finite difference approach as soon as the shape has more then 2 degrees of freedom since its effort is always a total of 2 forward problems to solve.

Author

Pascal Kraft

Date

29.11.2016

Definition at line 20 of file AdjointOptimization.h.

4.1.2 Member Function Documentation

4.1.2.1 run()

```
void AdjointOptimization::run ( )  [virtual]
```

This function is the core implementation of an optimization algorithm.

Currently it is very fundamental in its technical prowess which can be improved upon in later versions. Essentially, it calculates the signal quality for a configurations and for small steps in every one of the dofs. After that, the optimization-step is estimated based on difference-quotients. Following this step, a large step is computed based upon the approximation of the gradient of the signal-quality functional and the iteration starts anew. If a decrease in quality is detected, the optimization-step is undone and the step-width is reduced. This function controls both the Waveguide- and the Waveguide-structure object.

Implements Optimization.

Definition at line 201 of file AdjointOptimization.cpp.

```
201                                {
202   Convergence_Table.set_auto_fill_mode(true);
203   bool run = true;
204   int counter = 0;
205   double quality = 0;
206
207   while (run) {
208     int small_steps = 0;
209     while (oa->perform_small_step_next(small_steps)) {
210       deallog << "Performing a small step." << std::endl;
211       double temp_step_width = oa->get_small_step_step_width(small_steps);
212       oa->pass_result_small_step(compute_small_step(temp_step_width));
213       small_steps++;
214     }
215
216     if (oa->perform_big_step_next(small_steps)) {
217       deallog << "Performing a big step." << std::endl;
218       std::vector<double> step = oa->get_big_step_configuration();
219       deallog << "Got the following big step configuration: ";
220       for (unsigned int i = 0; i < step.size(); i++) {
221         deallog << step[i] << " , ";
222       }
```

```
223       deallog << std::endl;
224       quality = compute_big_step(step);
225       oa->pass_result_big_step(
226           primal_st->evaluate_for_z(GlobalParams.M_R_ZLength / 2.0, waveguide));
227     }
228
229     counter++;
230
231     if (counter > GlobalParams.Sc_OptimizationSteps || quality > 1.0) {
232       deallog << "The optimization is shutting down after " << counter
233               << " steps. Last quality: " << 100 * quality << "%." << std::endl;
234       run = false;
235     }
236
237     if ((GlobalParams.O_C_D_ConvergenceFirst ||
238          GlobalParams.O_C_D_ConvergenceAll) &&
239         (GlobalParams.MPI_Rank == 0)) {
240       std::ofstream result_file;
241       result_file.open((solutionpath + "/convergence_rates.dat").c_str(),
242                        std::ios_base::openmode::_S_trunc);
243
244       Convergence_Table.write_text(
245           result_file,
246           dealii::TableHandler::TextOutputFormat::table_with_headers);
247       result_file.close();
248       result_file.open((solutionpath + "/convergence_rates.tex").c_str(),
249                        std::ios_base::openmode::_S_trunc);
250       Convergence_Table.write_tex(result_file);
251       result_file.close();
252
253       result_file.open((solutionpath + "/steps.dat").c_str(),
254                        std::ios_base::openmode::_S_trunc);
255       oa->WriteStepsOut(result_file);
256       result_file.close();
257     }
258   }
259 }
```

4.1.3 Member Data Documentation

4.1.3.1 type

```
const int AdjointOptimization::type
```

Initial value:

```
=
      1
```

Definition at line 22 of file AdjointOptimization.h.

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

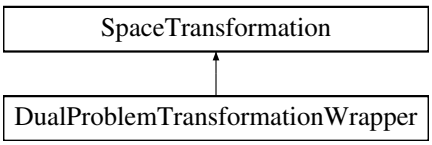
- Code/OptimizationStrategies/AdjointOptimization.h
- Code/OptimizationStrategies/AdjointOptimization.cpp

4.2 DualProblemTransformationWrapper Class Reference

If we do an adjoint computation, we need a SpaceTransformation, which has the same properties as the primal one but measures in transformed coordinates.

`#include <DualProblemTransformationWrapper.h>`

Inheritance diagram for DualProblemTransformationWrapper:



Public Member Functions

- DualProblemTransformationWrapper (SpaceTransformation *non_dual_st, int rank)

  Since this object encapsulates another Space Transformation, the construction is straight forward.
- Point< 3 > math_to_phys (Point< 3 > coord) const

  One of the core functionalities of a SpaceTransformation is to map a mathematical coordinate to a physical one (so an transformed to an untransformed coordinate).
- Point< 3 > phys_to_math (Point< 3 > coord) const

  This function does the same as math_to_phys only in the opposit direction.
- bool is_identity (Point< 3 > coord) const

  In order to test implementation, this function was added to check, if the transformation-tensor at a given coordinate is the identity or not.
- Tensor< 2, 3, std::complex< double > > **get_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, std::complex< double > > **get_Preconditioner_Tensor** (Point< 3 > &coordinate, int block) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor_For_Preconditioner** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input, int block) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor_Homogenized** (Point< 3 > &coordinate) const
- bool PML_in_X (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.
- bool PML_in_Y (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.
- bool PML_in_Z (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.
- double Preconditioner_PML_Z_Distance (Point< 3 > &p, unsigned int block) const

  This function fulfills the same purpose as those with similar names but it is supposed to be used together with Preconditioner_PML_in_Z instead of the versions without "Preconditioner".
- double PML_X_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Y_Distance (Point< 3 > &position) const

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

- double PML_Z_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- 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_Q1 (double z) const

  This member calculates the value of Q1 for a provided z-coordinate.
- double get_Q2 (double z) const

  This member calculates the value of Q2 for a provided z-coordinate.
- double get_Q3 (double z) const

  This member calculates the value of Q3 for a provided z-coordinate.
- double get_dof (int dof) const

  This is a getter for the values of degrees of freedom.
- void set_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.
- double get_free_dof (int dof) const

  This is a getter for the values of degrees of freedom.
- void set_free_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.
- std::pair< int, double > Z_to_Sector_and_local_z (double in_z) const

  Using this method unifies the usage of coordinates.
- double System_Length () const

  Returns the complete length of the computational domain.
- double Sector_Length () const

  Returns the length of one sector.
- double Layer_Length () const

  Returns the length of one layer.
- double get_r (double in_z) const

  Returns the radius for a system-coordinate;.
- 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;.
- int **Z_to_Layer** (double) const
- Vector< double > Dofs () 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 NFreeDofs () const

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

  This function returns the total number of DOFs including restrained ones.
- bool IsDofFree (int) const

  Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.
- void Print () const

  Console output of the current Waveguide Structure.
- std::complex< double > evaluate_for_z (double z_in, Waveguide *)

  Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.
- std::complex< double > **evaluate_for_z_with_sum** (double, double, Waveguide *)
- std::complex< double > **gauss_product_2D_sphere** (double z, int n, double R, double Xc, double Yc, Waveguide *in_w)

Public Attributes

- const double **XMinus**
- const double **XPlus**
- const double **YMinus**
- const double **YPlus**
- const double **ZMinus**
- const double **ZPlus**
- std::vector< Sector< 3 > > case_sectors

  This member contains all the Sectors who, as a sum, form the complete Waveguide.
- const double epsilon_K

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const double epsilon_M

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const int sectors

  Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.
- const double deltaY

  This value is initialized with the value Delta from the input-file.
- Vector< double > InitialDofs

  This vector of values saves the initial configuration.
- SpaceTransformation * **st**

4.2.1 Detailed Description

If we do an adjoint computation, we need a SpaceTransformation, which has the same properties as the primal one but measures in transformed coordinates.

This Wrapper contains the space transformation of the primal version but maps input parameters to their dual equivalent.

Essentially this class enables us to write a waveguide class which is unaware of its being primal or dual. Using this wrapper makes us compute the solution of the inverse order shape parametrization.

Author

Pascal Kraft

Date

1.12.2016

Definition at line 26 of file DualProblemTransformationWrapper.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 DualProblemTransformationWrapper()

```
DualProblemTransformationWrapper::DualProblemTransformationWrapper (
            SpaceTransformation * non_dual_st,
            int rank )
```

Since this object encapsulates another Space Transformation, the construction is straight forward.

Parameters

non_dual_st	This pointer points to the actual transformation that is being wrapped.

Definition at line 22 of file DualProblemTransformationWrapper.cpp.

```
24     : SpaceTransformation(3, inner_rank),
25       XMinus(-(GlobalParams.M_R_XLength * 0.5 - GlobalParams.M_BC_XMinus)),
26       XPlus(GlobalParams.M_R_XLength * 0.5 - GlobalParams.M_BC_XPlus),
27       YMinus(-(GlobalParams.M_R_YLength * 0.5 - GlobalParams.M_BC_YMinus)),
28       YPlus(GlobalParams.M_R_YLength * 0.5 - GlobalParams.M_BC_YPlus),
29       ZMinus(-GlobalParams.M_R_ZLength * 0.5),
30       ZPlus(GlobalParams.M_R_ZLength * 0.5),
31       epsilon_K(GlobalParams.M_W_epsilonin),
32       epsilon_M(GlobalParams.M_W_epsilonout),
33       sectors(GlobalParams.M_W_Sectors),
34       deltaY(GlobalParams.M_W_Delta) {
35   st = in_st;
36   homogenized = st->homogenized;
37 }
```

4.2.3 Member Function Documentation

4.2.3.1 Dofs()

```
Vector< double > DualProblemTransformationWrapper::Dofs ( ) 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.

Implements SpaceTransformation.

Definition at line 246 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::Dofs().

```
246                                                          {
247   return st->Dofs();
248 }
```

4.2.3.2 estimate_and_initialize()

```
void DualProblemTransformationWrapper::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 188 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::estimate_and_initialize().

```
188                                                          {
189   st->estimate_and_initialize();
190   return;
191 }
```

4.2.3.3 evaluate_for_z()

```
std::complex< double > DualProblemTransformationWrapper::evaluate_for_z (
            double ,
            Waveguide *  )  [virtual]
```

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Since I dont want to create derived classes from waveguide (which I should do eventually) I will for now include this functionality into the space transformation which is shape-sensitive. The waveguide only offers the evaluation at a point. The quadrature-rule has to be imposed by the space transformation.

Implements SpaceTransformation.

Definition at line 183 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::evaluate_for_z().

```
184                                 {
185   return st->evaluate_for_z(in_z, in_w);
186 }
```

4.2.3.4 get_dof()

```
double DualProblemTransformationWrapper::get_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 205 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::get_dof().

Referenced by HomogenousTransformationCircular::Dofs().

```
205                                                                 {
206   return st->get_dof(dof);
207 }
```

4.2.3.5 get_free_dof()

```
double DualProblemTransformationWrapper::get_free_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 213 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::get_free_dof().

```
213                                                                      {
214   return st->get_free_dof(dof);
215 }
```

4.2.3.6 get_Q1()

```
double DualProblemTransformationWrapper::get_Q1 (
            double z ) const  [virtual]
```

This member calculates the value of Q1 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q1 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 193 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::get_Q1().

```
193                                                          {
194   return st->get_Q1(z);
195 }
```

4.2.3.7 get_Q2()

```
double DualProblemTransformationWrapper::get_Q2 (
            double z ) const  [virtual]
```

This member calculates the value of Q2 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q2 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 197 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::get_Q2().

```
197                                                          {
198   return st->get_Q2(z);
199 }
```

4.2.3.8 get_Q3()

```
double DualProblemTransformationWrapper::get_Q3 (
            double z ) const  [virtual]
```

This member calculates the value of Q3 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q3 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 201 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::get_Q3().

```
201                                                          {
202   return st->get_Q3(z);
203 }
```

4.2.3.9 is_identity()

```
bool DualProblemTransformationWrapper::is_identity (
            Point< 3 > coord ) const
```

In order to test implementation, this function was added to check, if the transformation-tensor at a given coordinate is the identity or not.

Parameters

coord	This is the coordinate to test.

4.2.3.10 IsDofFree()

```
bool DualProblemTransformationWrapper::IsDofFree (
            int input ) const  [virtual]
```

Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

"restrained" means that for example the DOF represents the radius at one of the connectors (input or output) and therefore we forbid the optimization scheme to vary this value.

Implements SpaceTransformation.

Definition at line 258 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::IsDofFree().

```
258                                                                 {
259   return st->IsDofFree(input);
260 }
```

4.2.3.11 math_to_phys()

```
Point< 3 > DualProblemTransformationWrapper::math_to_phys (
            Point< 3 > coord ) const  [virtual]
```

One of the core functionalities of a SpaceTransformation is to map a mathematical coordinate to a physical one (so an transformed to an untransformed coordinate).

Parameters

coord	the coordinate to be transformed. In this class we simply pass the

Implements SpaceTransformation.

Definition at line 39 of file DualProblemTransformationWrapper.cpp.

```
39                                                                      {
40   return st->math_to_phys(coord);
41 }
```

4.2.3.12 NDofs()

```
unsigned int DualProblemTransformationWrapper::NDofs ( ) const  [virtual]
```

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

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

Implements SpaceTransformation.

Definition at line 254 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::NDofs().

Referenced by HomogenousTransformationCircular::Dofs(), HomogenousTransformationCircular::get_dof(), HomogenousTransformationCircular::get_free_dof(), HomogenousTransformationCircular::IsDofFree(), HomogenousTransformationCircular::NFreeDofs(), HomogenousTransformationCircular::set_dof(), and HomogenousTransformationCircular::set_free_dof().

```
254                                                 {
255   return st->NDofs();
256 }
```

4.2.3.13 phys_to_math()

```
Point< 3 > DualProblemTransformationWrapper::phys_to_math (
            Point< 3 > coord ) const  [virtual]
```

This function does the same as math_to_phys only in the opposit direction.

Parameters

coord	the coordinate to be transformed.

Implements SpaceTransformation.

Definition at line 43 of file DualProblemTransformationWrapper.cpp.

```
43                                                                          {
44   return st->phys_to_math(coord);
45 }
```

4.2.3.14 PML_in_X()

```
bool DualProblemTransformationWrapper::PML_in_X (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 47 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::PML_in_X().

Referenced by HomogenousTransformationCircular::PML_Z_Distance().

```
47                                                          {
48   return st->PML_in_X(p);
49 }
```

4.2.3.15 PML_in_Y()

```
bool DualProblemTransformationWrapper::PML_in_Y (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 51 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::PML_in_Y().

Referenced by HomogenousTransformationCircular::PML_Z_Distance().

```
51                                                                  {
52   return st->PML_in_Y(p);
53 }
```

4.2.3.16 PML_in_Z()

```
bool DualProblemTransformationWrapper::PML_in_Z (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 55 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::PML_in_Z().

Referenced by HomogenousTransformationCircular::PML_Z_Distance().

```
55                                                                  {
56   return st->PML_in_Z(p);
57 }
```

4.2.3.17 PML_X_Distance()

```
double DualProblemTransformationWrapper::PML_X_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 64 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::PML_X_Distance().

Referenced by HomogenousTransformationCircular::PML_Z_Distance().

```
64                                                                    {
65   return st->PML_X_Distance(p);
66 }
```

4.2.3.18 PML_Y_Distance()

```
double DualProblemTransformationWrapper::PML_Y_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 68 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::PML_Y_Distance().

Referenced by HomogenousTransformationCircular::PML_Z_Distance().

```
68                                                                          {
69   return st->PML_Y_Distance(p);
70 }
```

4.2.3.19 PML_Z_Distance()

```
double DualProblemTransformationWrapper::PML_Z_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 72 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::PML_Z_Distance().

Referenced by HomogenousTransformationCircular::PML_Z_Distance().

```
72                                                                          {
73   return st->PML_Z_Distance(p);
74 }
```

4.2.3.20 set_dof()

```
void DualProblemTransformationWrapper::set_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 209 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::set_dof().

```
209                                                                    {
210   return st->set_dof(dof, value);
211 }
```

4.2.3.21 set_free_dof()

```
void DualProblemTransformationWrapper::set_free_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 217 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::set_free_dof().

```
217                                                                         {
218   return st->set_free_dof(dof, value);
219 }
```

4.2.3.22 Z_to_Sector_and_local_z()

```
std::pair< int, double > DualProblemTransformationWrapper::Z_to_Sector_and_local_z (
            double in_z ) const
```

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 222 of file DualProblemTransformationWrapper.cpp.

References SpaceTransformation::Z_to_Sector_and_local_z().

Referenced by HomogenousTransformationCircular::get_m(), HomogenousTransformationCircular::get_Q1(), HomogenousTransformationCircular::get_Q2(), HomogenousTransformationCircular::get_Q3(), HomogenousTransformationCircular::get_r(), HomogenousTransformationCircular::get_v(), and HomogenousTransformationCircular::PML_Z_Distance().

```
222                                                                    {
223   return st->Z_to_Sector_and_local_z(in_z);
224 }
```

4.2.4 Member Data Documentation

4.2.4.1 case_sectors

```
std::vector<Sector<3> > DualProblemTransformationWrapper::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 151 of file DualProblemTransformationWrapper.h.

Referenced by HomogenousTransformationCircular::estimate_and_initialize(), HomogenousTransformationCircular::get_dof(), HomogenousTransformationCircular::get_free_dof(), HomogenousTransformationCircular::get_m(), HomogenousTransformationCircular::get_Q1(), HomogenousTransformationCircular::get_Q2(), HomogenousTransformationCircular::get_Q3(), HomogenousTransformationCircular::get_r(), HomogenousTransformationCircular::get_v(), HomogenousTransformationCircular::PML_Z_Distance(), HomogenousTransformationCircular::set_dof(), and HomogenousTransformationCircular::set_free_dof().

4.2.4.2 epsilon_K

```
const double DualProblemTransformationWrapper::epsilon_K
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value inside the core.

Definition at line 158 of file DualProblemTransformationWrapper.h.

4.2.4.3 epsilon_M

```
const double DualProblemTransformationWrapper::epsilon_M
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value outside the core.

Definition at line 164 of file DualProblemTransformationWrapper.h.

4.2.4.4 sectors

```
const int DualProblemTransformationWrapper::sectors
```

Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.

This member stores the number of Sectors the computational domain has been split into.

Definition at line 170 of file DualProblemTransformationWrapper.h.

Referenced by HomogenousTransformationCircular::estimate_and_initialize(), HomogenousTransformation↩Circular::get_dof(), HomogenousTransformationCircular::get_free_dof(), HomogenousTransformationCircular::↩NDofs(), HomogenousTransformationCircular::set_dof(), and HomogenousTransformationCircular::set_free_dof().

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

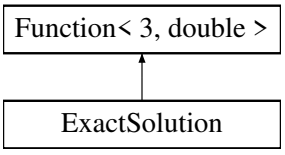
- Code/SpaceTransformations/DualProblemTransformationWrapper.h
- Code/SpaceTransformations/DualProblemTransformationWrapper.cpp

4.3 ExactSolution Class Reference

This class is derived from the Function class and can be used to estimate the L2-error for a straight waveguide.

```
#include <ExactSolution.h>
```

Inheritance diagram for ExactSolution:



Public Member Functions

- **ExactSolution** (bool in_rectangular=false, bool in_dual=false)
- double value (const Point< 3 > &p, const unsigned int component) const

  This function calculates one single component of the solution vector.
- void vector_value (const Point< 3 > &p, Vector< double > &value) const

  This function is the one that gets called from external contexts and calls the value-function to calculate the individual components.
- std::vector< std::string > **split** (std::string) const
- Tensor< 1, 3, std::complex< double > > **curl** (const Point< 3 > &in_p) const
- Tensor< 1, 3, std::complex< double > > **val** (const Point< 3 > &in_p) const

4.3.1 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 36 of file ExactSolution.h.

4.3.2 Member Function Documentation

4.3.2.1 value()

```
double ExactSolution::value (
            const Point< 3 > & p,
            const unsigned int component ) const
```

This function calculates one single component of the solution vector.

To calculate this, we do the following: We know the input on the boundary of the computational domain for $z = z_{in}$. So for a given position $p = (x, y, z)$ we calculate

$$f_c(x, y, z) = \sum_{j=0}^{N} (a_j \, \boldsymbol{\phi_j}(x, y, z_{in})) \cdot \boldsymbol{e_c} \, \mathrm{e}^{i\omega(z - z_{in})}.$$

Here, $\boldsymbol{\phi_j}$ is the j-th mode of the waveguide which is induced with the intensity a_j. $\boldsymbol{e_c}$ is the c-th unit-vector with c being the index of the component we want to compute.

Parameters

p	This value contains the position for which we want to calculate the exact solution.
component	This integer holds the index of the component we want to compute. Keep in mind that these are not coordinates in the physical sense. The components 0 to 2 are the real parts of the solution-vector and the components 3-5 are the imaginary parts.

Definition at line 13 of file ExactSolution.cpp.

```
14                                                                  {
15   Point<3, double> p = in_p;
16   if (is_dual) p[2] = -in_p[2];
17   bool zero = false;
18   if (p[0] > GlobalParams.M_R_XLength / 2.0 - GlobalParams.M_BC_XPlus)
19     zero = true;
20   if (p[0] < -GlobalParams.M_R_XLength / 2.0 + GlobalParams.M_BC_XMinus)
21     zero = true;
22   if (p[1] > GlobalParams.M_R_YLength / 2.0 - GlobalParams.M_BC_YPlus)
23     zero = true;
24   if (p[1] < -GlobalParams.M_R_YLength / 2.0 + GlobalParams.M_BC_YMinus)
25     zero = true;
26   if (p[2] > GlobalParams.M_R_ZLength / 2.0) zero = true;
27   if (zero) {
28     return 0.0;
29   }
30
31   if (is_rectangular) {
32     std::complex<double> ret_val(0.0, 0.0);
33     const double delta = abs(mesh_points[0] - mesh_points[1]);
34     const int mesh_number = mesh_points.size();
35     if (!(abs(p(1)) >= mesh_points[0] || abs(p(0)) >= mesh_points[0])) {
36       int ix = 0;
37       int iy = 0;
38       while (mesh_points[ix] > p(0) && ix < mesh_number) ix++;
39       while (mesh_points[iy] > p(1) && iy < mesh_number) iy++;
40       if (ix == 0 || iy == 0 || ix == mesh_number || iy == mesh_number) {
41         return 0.0;
42       } else {
43         double dx = (p(0) - mesh_points[ix]) / delta;
44         double dy = (p(1) - mesh_points[iy]) / delta;
45         double m1m1 = dx * dy;
46         double m1p1 = dx * (1.0 - dy);
47         double p1p1 = (1.0 - dx) * (1.0 - dy);
48         double p1m1 = (1.0 - dx) * dy;
49         switch (component % 3) {
50           case 0:
51             ret_val.real(p1p1 * vals[ix][iy].Ex.real() +
52                          p1m1 * vals[ix][iy - 1].Ex.real() +
53                          m1m1 * vals[ix - 1][iy - 1].Ex.real() +
54                          m1p1 * vals[ix - 1][iy].Ex.real());
55             ret_val.imag(p1p1 * vals[ix][iy].Ex.imag() +
56                          p1m1 * vals[ix][iy - 1].Ex.imag() +
57                          m1m1 * vals[ix - 1][iy - 1].Ex.imag() +
58                          m1p1 * vals[ix - 1][iy].Ex.imag());
59             // ret_val *= -1.0;
60             break;
61           case 1:
62             ret_val.real(p1p1 * vals[ix][iy].Ey.real() +
63                          p1m1 * vals[ix][iy - 1].Ey.real() +
64                          m1m1 * vals[ix - 1][iy - 1].Ey.real() +
65                          m1p1 * vals[ix - 1][iy].Ey.real());
66             ret_val.imag(p1p1 * vals[ix][iy].Ey.imag() +
67                          p1m1 * vals[ix][iy - 1].Ey.imag() +
68                          m1m1 * vals[ix - 1][iy - 1].Ey.imag() +
69                          m1p1 * vals[ix - 1][iy].Ey.imag());
70             break;
71           case 2:
72             ret_val.real(p1p1 * vals[ix][iy].Ez.real() +
73                          p1m1 * vals[ix][iy - 1].Ez.real() +
74                          m1m1 * vals[ix - 1][iy - 1].Ez.real() +
75                          m1p1 * vals[ix - 1][iy].Ez.real());
76             ret_val.imag(p1p1 * vals[ix][iy].Ez.imag() +
77                          p1m1 * vals[ix][iy - 1].Ez.imag() +
78                          m1m1 * vals[ix - 1][iy - 1].Ez.imag() +
79                          m1p1 * vals[ix - 1][iy].Ez.imag());
80             break;
81           default:
82             ret_val.real(0.0);
83             ret_val.imag(0.0);
84             break;
```

```
85          }
86        }
87        double n;
88        if (abs(p(0)) <= GlobalParams.M_C_Dim1In / 2.0 &&
89            abs(p(1)) <= GlobalParams.M_C_Dim2In / 2.0) {
90          n = std::sqrt(GlobalParams.M_W_epsilonin);
91        } else {
92          n = std::sqrt(GlobalParams.M_W_epsilonout);
93        }
94        double k = n * 2 * GlobalParams.C_Pi / GlobalParams.M_W_Lambda;
95        std::complex<double> phase(0.0, (p(2) - GlobalParams.Minimum_Z) * k);
96        ret_val *= std::exp(phase);
97        if (component > 2) {
98          return ret_val.imag();
99        } else {
100         return ret_val.real();
101       }
102     } else {
103       return 0.0;
104     }
105   } else {
106     return ModeMan.get_input_component(component, p, 0);
107   }
108 }
```

4.3.2.2 vector_value()

```
void ExactSolution::vector_value (
              const Point< 3 > & p,
              Vector< double > & value ) const
```

This function is the one that gets called from external contexts and calls the value-function to calculate the individual components.

The real solution looks as follows:

$$f(x,y,z) = \begin{pmatrix} \text{value}(x,y,z,0) \\ \text{value}(x,y,z,1) \\ \text{value}(x,y,z,2) \end{pmatrix} + i \begin{pmatrix} \text{value}(x,y,z,3) \\ \text{value}(x,y,z,4) \\ \text{value}(x,y,z,5) \end{pmatrix}.$$

Definition at line 110 of file ExactSolution.cpp.

Referenced by Waveguide::estimate_solution(), and Waveguide::evaluate_for_Position().

```
111                                                                 {
112   Point<3, double> p = in_p;
113   if (is_dual) p[2] = -in_p[2];
114   bool zero = false;
115   if (p[0] > GlobalParams.M_R_XLength / 2.0 - GlobalParams.M_BC_XPlus)
116     zero = true;
117   if (p[0] < -GlobalParams.M_R_XLength / 2.0 + GlobalParams.M_BC_XMinus)
118     zero = true;
119   if (p[1] > GlobalParams.M_R_YLength / 2.0 - GlobalParams.M_BC_YPlus)
120     zero = true;
121   if (p[1] < -GlobalParams.M_R_YLength / 2.0 + GlobalParams.M_BC_YMinus)
122     zero = true;
123   if (p[2] > GlobalParams.M_R_ZLength / 2.0) zero = true;
124   if (zero) {
125     for (unsigned int i = 0; i < values.size(); i++) {
126       values[i] = 0.0;
127     }
128     return;
129   }
130   if (is_rectangular) {
131     const double delta = abs(mesh_points[0] - mesh_points[1]);
132     const int mesh_number = mesh_points.size();
133     if (!(abs(p(1)) >= mesh_points[0] || abs(p(0)) >= mesh_points[0])) {
134       int ix = 0;
135       int iy = 0;
```

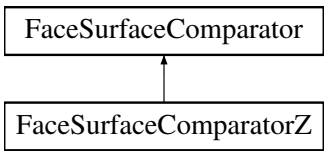
```
136       while (mesh_points[ix] > p(0) && ix < mesh_number) ix++;
137       while (mesh_points[iy] > p(1) && iy < mesh_number) iy++;
138       if (ix == 0 || iy == 0 || ix == mesh_number || iy == mesh_number) {
139         for (unsigned int i = 0; i < values.size(); i++) {
140           values[i] = 0.0;
141         }
142         return;
143       } else {
144         double dx = (p(0) - mesh_points[ix]) / delta;
145         double dy = (p(1) - mesh_points[iy]) / delta;
146         double m1m1 = dx * dy;
147         double m1p1 = dx * (1.0 - dy);
148         double p1p1 = (1.0 - dx) * (1.0 - dy);
149         double p1m1 = (1.0 - dx) * dy;
150         values[0] = p1p1 * vals[ix][iy].Ex.real() +
151                     p1m1 * vals[ix][iy - 1].Ex.real() +
152                     m1m1 * vals[ix - 1][iy - 1].Ex.real() +
153                     m1p1 * vals[ix - 1][iy].Ex.real();
154         values[1] = p1p1 * vals[ix][iy].Ey.real() +
155                     p1m1 * vals[ix][iy - 1].Ey.real() +
156                     m1m1 * vals[ix - 1][iy - 1].Ey.real() +
157                     m1p1 * vals[ix - 1][iy].Ey.real();
158         values[2] = p1p1 * vals[ix][iy].Ez.real() +
159                     p1m1 * vals[ix][iy - 1].Ez.real() +
160                     m1m1 * vals[ix - 1][iy - 1].Ez.real() +
161                     m1p1 * vals[ix - 1][iy].Ez.real();
162         values[3] = p1p1 * vals[ix][iy].Ex.imag() +
163                     p1m1 * vals[ix][iy - 1].Ex.imag() +
164                     m1m1 * vals[ix - 1][iy - 1].Ex.imag() +
165                     m1p1 * vals[ix - 1][iy].Ex.imag();
166         values[4] = p1p1 * vals[ix][iy].Ey.imag() +
167                     p1m1 * vals[ix][iy - 1].Ey.imag() +
168                     m1m1 * vals[ix - 1][iy - 1].Ey.imag() +
169                     m1p1 * vals[ix - 1][iy].Ey.imag();
170         values[5] = p1p1 * vals[ix][iy].Ez.imag() +
171                     p1m1 * vals[ix][iy - 1].Ez.imag() +
172                     m1m1 * vals[ix - 1][iy - 1].Ez.imag() +
173                     m1p1 * vals[ix - 1][iy].Ez.imag();
174         double n;
175         if (abs(p(0)) <= GlobalParams.M_C_Dim1In / 2.0 &&
176             abs(p(1)) <= GlobalParams.M_C_Dim2In / 2.0) {
177           n = std::sqrt(GlobalParams.M_W_epsilonin);
178         } else {
179           n = std::sqrt(GlobalParams.M_W_epsilonout);
180         }
181         double k = n * 2 * GlobalParams.C_Pi / GlobalParams.M_W_Lambda;
182         std::complex<double> phase(
183             0.0, -(p(2) + GlobalParams.M_R_ZLength / 2.0) * k);
184         phase = std::exp(phase);
185         for (unsigned int komp = 0; komp < 3; komp++) {
186           std::complex<double> entr(values[0 + komp], values[3 + komp]);
187           entr *= phase;
188           values[0 + komp] = entr.real();
189           values[3 + komp] = entr.imag();
190         }
191         // values[0] *= -1.0;
192         // values[3] *= -1.0;
193         return;
194       }
195     } else {
196       for (unsigned int i = 0; i < values.size(); i++) {
197         values[i] = 0.0;
198       }
199       return;
200     }
201   } else {
202     for (unsigned int c = 0; c < 6; ++c)
203       values[c] = ModeMan.get_input_component(c, p, 0);
204   }
205 }
```

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

- Code/Helpers/ExactSolution.h
- Code/Helpers/ExactSolution.cpp

4.4 FaceSurfaceComparator Class Reference

Inheritance diagram for FaceSurfaceComparator:



Public Member Functions

- virtual bool **check_face** (const dealii::parallel::distributed::Triangulation< 3, 3 >::face_iterator)

4.4.1 Detailed Description

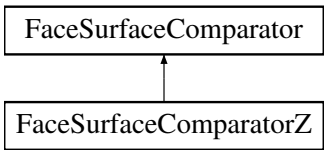
Definition at line 12 of file FaceSurfaceComparator.h.

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

- Code/HSIE/FaceSurfaceComparator.h
- Code/HSIE/FaceSurfaceComparator.cpp

4.5 FaceSurfaceComparatorZ Class Reference

Inheritance diagram for FaceSurfaceComparatorZ:



Public Member Functions

- **FaceSurfaceComparatorZ** (double in_z=0, double in_tol=0.0001)
- virtual bool **check_face** (const dealii::parallel::distributed::Triangulation< 3, 3 >::face_iterator)

4.5.1 Detailed Description

Definition at line 14 of file FaceSurfaceComparatorZ.h.

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

- Code/HSIE/FaceSurfaceComparatorZ.h
- Code/HSIE/FaceSurfaceComparatorZ.cpp

4.6 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**

4.6.1 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 11 of file GradientTable.h.

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

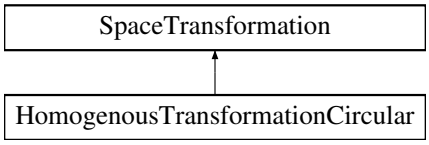
- Code/OutputGenerators/Console/GradientTable.h
- Code/OutputGenerators/Console/GradientTable.cpp

4.7 HomogenousTransformationCircular Class Reference

For this transformation we try to achieve a situation in which tensorial material properties from the coordinate transformation and PML-regions dont overlap.

`#include <HomogenousTransformationCircular.h>`

Inheritance diagram for HomogenousTransformationCircular:



Public Member Functions

- **HomogenousTransformationCircular** (int)
- Point< 3 > **math_to_phys** (Point< 3 > coord) const
- Point< 3 > **phys_to_math** (Point< 3 > coord) const
- Point< 3 > **math_to_phys_hom** (Point< 3 > coord) const
- Point< 3 > **phys_to_math_hom** (Point< 3 > coord) const
- bool **is_identity** (Point< 3 > coord) const
- Tensor< 2, 3, std::complex< double > > **get_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, std::complex< double > > **get_Preconditioner_Tensor** (Point< 3 > &coordinate, int block) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor_For_Preconditioner** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input, int block) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor_Homogenized** (Point< 3 > &coordinate) const
- bool PML_in_X (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.
- bool PML_in_Y (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.
- bool PML_in_Z (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.
- double Preconditioner_PML_Z_Distance (Point< 3 > &p, unsigned int block) const

  Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.
- double PML_X_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Y_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Z_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- 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_Q1 (double z) const

  This member calculates the value of Q1 for a provided z-coordinate.

- double get_Q2 (double z) const

  This member calculates the value of Q2 for a provided z-coordinate.

- double get_Q3 (double z) const

  This member calculates the value of Q3 for a provided z-coordinate.

- double get_dof (int dof) const

  This is a getter for the values of degrees of freedom.

- void set_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.

- double get_free_dof (int dof) const

  This is a getter for the values of degrees of freedom.

- void set_free_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.

- double System_Length () const

  Returns the complete length of the computational domain.

- double Sector_Length () const

  Returns the length of one sector.

- double Layer_Length () const

  Returns the length of one layer.

- double get_r (double in_z) const

  Returns the radius for a system-coordinate;.

- 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;.

- Vector< double > Dofs () 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 NFreeDofs () const

  This function returns the number of unrestrained degrees of freedom of the current optimization run.

- unsigned int NDofs () const

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

- bool IsDofFree (int) const

  Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

- void Print () const

  Console output of the current Waveguide Structure.

- std::complex< double > evaluate_for_z (double z_in, Waveguide *)

  Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

- std::complex< double > **evaluate_for_z_with_sum** (double, double, Waveguide *)

- std::complex< double > **gauss_product_2D_sphere_primal** (double z, int n, double R, double Xc, double Yc, Waveguide *in_w)

Public Attributes

- const double **XMinus**
- const double **XPlus**
- const double **YMinus**
- const double **YPlus**
- const double **ZMinus**
- const double **ZPlus**
- std::vector< Sector< 3 > > case_sectors

  This member contains all the Sectors who, as a sum, form the complete Waveguide.
- const double epsilon_K

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const double epsilon_M

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const int sectors

  Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.
- const double deltaY

  This value is initialized with the value Delta from the input-file.
- Vector< double > InitialDofs

  This vector of values saves the initial configuration.

4.7.1 Detailed Description

For this transformation we try to achieve a situation in which tensorial material properties from the coordinate transformation and PML-regions dont overlap.

The usage of a coordinate transformation which is identity on the domain containing our PML is a strong restriction however it ensures lower errors since the quality of the PML is harder to estimate otherwise. Also it limits us in how we model the waveguide essentially forcing us to have no bent between the wavguides-connectors.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 33 of file HomogenousTransformationCircular.h.

4.7.2 Member Function Documentation

4.7.2.1 Dofs()

```
Vector< double > HomogenousTransformationCircular::Dofs ( ) 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.

Implements SpaceTransformation.

Definition at line 586 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::get_dof(), and DualProblemTransformationWrapper::NDofs().

```
586                                                          {
587   Vector<double> ret;
588   const int total = NDofs();
589   ret.reinit(total);
590   for (int i = 0; i < total; i++) {
591     ret[i] = get_dof(i);
592   }
593   return ret;
594 }
```

4.7.2.2 estimate_and_initialize()

```
void HomogenousTransformationCircular::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 512 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, DualProblemTransformationWrapper::Sector_↩Length(), and DualProblemTransformationWrapper::sectors.

```
512                                                                {
513   case_sectors.reserve(sectors);
514   double m_0 = GlobalParams.M_W_Delta / 2.0;
515   double m_1 = -GlobalParams.M_W_Delta / 2.0;
516   double r_0 = GlobalParams.M_C_Dim1In;
517   double r_1 = GlobalParams.M_C_Dim1Out;
518   if (sectors == 1) {
519     Sector<3> temp12(true, true, -GlobalParams.M_R_ZLength / 2,
520                      GlobalParams.M_R_ZLength / 2);
521     case_sectors.push_back(temp12);
522     case_sectors[0].set_properties_force(
523         GlobalParams.M_W_Delta / 2.0, -GlobalParams.M_W_Delta / 2.0,
524         GlobalParams.M_C_Dim1In, GlobalParams.M_C_Dim1Out, 0, 0);
525   } else {
526     double length = Sector_Length();
527     Sector<3> temp(true, false, -GlobalParams.M_R_ZLength / (2.0),
```

```
528                   -GlobalParams.M_R_ZLength / 2.0 + length);
529     case_sectors.push_back(temp);
530     for (int i = 1; i < sectors; i++) {
531       Sector<3> temp2(false, false,
532                       -GlobalParams.M_R_ZLength / (2.0) + length * (1.0 * i),
533                       -GlobalParams.M_R_ZLength / (2.0) + length * (i + 1.0));
534       case_sectors.push_back(temp2);
535     }
536
537     double length_rel = 1.0 / ((double)(sectors));
538     case_sectors[0].set_properties_force(
539         m_0, InterpolationPolynomialZeroDerivative(length_rel, m_0, m_1), r_0,
540         InterpolationPolynomialZeroDerivative(length_rel, r_0, r_1), 0,
541         InterpolationPolynomialDerivative(length_rel, m_0, m_1, 0, 0));
542     for (int i = 1; i < sectors; i++) {
543       double z_l = i * length_rel;
544       double z_r = (i + 1) * length_rel;
545       case_sectors[i].set_properties_force(
546           InterpolationPolynomialZeroDerivative(z_l, m_0, m_1),
547           InterpolationPolynomialZeroDerivative(z_r, m_0, m_1),
548           InterpolationPolynomialZeroDerivative(z_l, r_0, r_1),
549           InterpolationPolynomialZeroDerivative(z_r, r_0, r_1),
550           InterpolationPolynomialDerivative(z_l, m_0, m_1, 0, 0),
551           InterpolationPolynomialDerivative(z_r, m_0, m_1, 0, 0));
552     }
553   }
554 }
```

4.7.2.3 evaluate_for_z()

```
std::complex< double > HomogenousTransformationCircular::evaluate_for_z (
            double ,
            Waveguide *  )  [virtual]
```

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Since I dont want to create derived classes from waveguide (which I should do eventually) I will for now include this functionality into the space transformation which is shape-sensitive. The waveguide only offers the evaluation at a point. The quadrature-rule has to be imposed by the space transformation.

Implements SpaceTransformation.

Definition at line 432 of file HomogenousTransformationCircular.cpp.

```
433                                 {
434   double r = GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out;
435   return gauss_product_2D_sphere(in_z, 10, r, 0, 0, in_w);
436 }
```

4.7.2.4 get_dof()

```
double HomogenousTransformationCircular::get_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 438 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, DualProblemTransformationWrapper::NDofs(), and DualProblemTransformationWrapper::sectors.

Referenced by HomogenousTransformationRectangular::Dofs().

```
438                                                                  {
439   if (dof < (int)NDofs() && dof >= 0) {
440     int sector = floor(dof / 3);
441     if (sector == sectors) {
442       return case_sectors[sector - 1].dofs_r[dof % 3];
443     } else {
444       return case_sectors[sector].dofs_l[dof % 3];
445     }
446   } else {
447     std::cout << "Critical: DOF-index out of bounds in "
448                  "HomogenousTransformationCircular::get_dof!"
449               << std::endl;
450     return 0.0;
451   }
452 }
```

4.7.2.5 get_free_dof()

```
double HomogenousTransformationCircular::get_free_dof (
             int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 454 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, DualProblemTransformationWrapper::NDofs(), and DualProblemTransformationWrapper::sectors.

```
454                                                                          {
455   int dof = in_dof + 3;
456   if (dof < (int)NDofs() - 3 && dof >= 0) {
457     int sector = floor(dof / 3);
458     if (sector == sectors) {
459       return case_sectors[sector - 1].dofs_r[dof % 3];
460     } else {
461       return case_sectors[sector].dofs_l[dof % 3];
462     }
463   } else {
464     std::cout << "Critical: DOF-index out of bounds in "
465                  "HomogenousTransformationCircular::get_free_dof!"
466               << std::endl;
467     return 0.0;
468   }
469 }
```

4.7.2.6 get_Q1()

```
double HomogenousTransformationCircular::get_Q1 (
            double z ) const  [virtual]
```

This member calculates the value of Q1 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q1 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 571 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, and DualProblemTransformationWrapper::Z_to↩_Sector_and_local_z().

```
571                                                                  {
572   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
573   return case_sectors[two.first].getQ1(two.second);
574 }
```

4.7.2.7 get_Q2()

```
double HomogenousTransformationCircular::get_Q2 (
            double z ) const  [virtual]
```

This member calculates the value of Q2 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q2 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 576 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, and DualProblemTransformationWrapper::Z_to_Sector_and_local_z().

```
576                                                                 {
577   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
578   return case_sectors[two.first].getQ2(two.second);
579 }
```

4.7.2.8 get_Q3()

```
double HomogenousTransformationCircular::get_Q3 (
            double z ) const  [virtual]
```

This member calculates the value of Q3 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q3 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 581 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, and DualProblemTransformationWrapper::Z_to_Sector_and_local_z().

```
581                                                                 {
582   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
583   return case_sectors[two.first].getQ3(two.second);
584 }
```

4.7.2.9 IsDofFree()

```
bool HomogenousTransformationCircular::IsDofFree (
            int index ) const  [virtual]
```

Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

"restrained" means that for example the DOF represents the radius at one of the connectors (input or output) and therefore we forbid the optimization scheme to vary this value.

Implements SpaceTransformation.

Definition at line 600 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::NDofs().

```
600                                                                 {
601   return index > 2 && index < (int)NDofs() - 3;
602 }
```

4.7.2.10 NDofs()

```
unsigned int HomogenousTransformationCircular::NDofs ( ) const  [virtual]
```

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

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

Implements SpaceTransformation.

Definition at line 608 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::sectors.

Referenced by HomogenousTransformationRectangular::Dofs(), HomogenousTransformationRectangular::get_↩dof(), HomogenousTransformationRectangular::get_free_dof(), HomogenousTransformationRectangular::IsDof↩Free(), HomogenousTransformationRectangular::NFreeDofs(), HomogenousTransformationRectangular::set_dof(), and HomogenousTransformationRectangular::set_free_dof().

```
608                                                             {
609   return sectors * 3 + 3;
610 }
```

4.7.2.11 PML_in_X()

```
bool HomogenousTransformationCircular::PML_in_X (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 151 of file HomogenousTransformationCircular.cpp.

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
151                                                                    {
152   return p(0) < XMinus || p(0) > XPlus;
153 }
```

4.7.2.12 PML_in_Y()

```
bool HomogenousTransformationCircular::PML_in_Y (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 155 of file HomogenousTransformationCircular.cpp.

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
155                                                                    {
156   return p(1) < YMinus || p(1) > YPlus;
157 }
```

4.7.2.13 PML_in_Z()

```
bool HomogenousTransformationCircular::PML_in_Z (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 159 of file HomogenousTransformationCircular.cpp.

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
159                                                                 {
160   return p(2) > ZPlus;
161 }
```

4.7.2.14 PML_X_Distance()

```
double HomogenousTransformationCircular::PML_X_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 170 of file HomogenousTransformationCircular.cpp.

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
170                                                                 {
171   if (p(0) > 0) {
172     return p(0) - XPlus;
173   } else {
174     return -p(0) + XMinus;
175   }
176 }
```

4.7.2.15 PML_Y_Distance()

```
double HomogenousTransformationCircular::PML_Y_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 178 of file HomogenousTransformationCircular.cpp.

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
178                                                                    {
179   if (p(1) > 0) {
180     return p(1) - YPlus;
181   } else {
182     return -p(1) + YMinus;
183   }
184 }
```

4.7.2.16 PML_Z_Distance()

```
double HomogenousTransformationCircular::PML_Z_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 186 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, Waveguide::evaluate_for_Position(), DualProblemTransformationWrapper::PML_in_X(), DualProblemTransformationWrapper::PML_in_Y(), DualProblemTransformationWrapper::PML_in_Z(), DualProblemTransformationWrapper::PML_X_Distance(), DualProblemTransformationWrapper::PML_Y_Distance(), DualProblemTransformationWrapper::PML_Z_Distance(), DualProblemTransformationWrapper::Preconditioner_PML_Z_Distance(), and DualProblemTransformationWrapper::Z_to_Sector_and_local_z().

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
186                                                                    {
187   if (p(2) < 0) {
188     return 0;
189   } else {
190     return p(2) - (GlobalParams.M_R_ZLength / 2.0);
191   }
192 }
```

4.7.2.17 Preconditioner_PML_Z_Distance()

```
double HomogenousTransformationCircular::Preconditioner_PML_Z_Distance (
            Point< 3 > & p,
            unsigned int block ) const  [virtual]
```

Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.

These Layers are not only situated at the surface of the computational domain but also inside it at the interfaces of Sectors. This function fulfills the same purpose as those with similar names but it is supposed to be used together with Preconditioner_PML_in_Z instead of the versions without "Preconditioner".

Implements SpaceTransformation.

Definition at line 163 of file HomogenousTransformationCircular.cpp.

Referenced by HomogenousTransformationRectangular::PML_Z_Distance().

```
164                                                 {
165   double width = GlobalParams.LayerThickness * 1.0;
166
167   return p(2) + GlobalParams.M_R_ZLength / 2.0 - ((double)rank) * width;
168 }
```

4.7.2.18 set_dof()

```
void HomogenousTransformationCircular::set_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 471 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, DualProblemTransformationWrapper::NDofs(), and DualProblemTransformationWrapper::sectors.

```
471                                                                     {
472   if (dof < (int)NDofs() && dof >= 0) {
473     int sector = floor(dof / 3);
474     if (sector == sectors) {
475       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
476     } else if (sector == 0) {
```

```
477       case_sectors[0].dofs_l[dof % 3] = in_val;
478     } else {
479       case_sectors[sector].dofs_l[dof % 3] = in_val;
480       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
481     }
482   } else {
483     std::cout << "Critical: DOF-index out of bounds in "
484                  "HomogenousTransformationCircular::set_dof!"
485               << std::endl;
486   }
487 }
```

4.7.2.19 set_free_dof()

```
void HomogenousTransformationCircular::set_free_dof (
             int dof,
             double value )  [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.

Implements SpaceTransformation.

Definition at line 489 of file HomogenousTransformationCircular.cpp.

References DualProblemTransformationWrapper::case_sectors, DualProblemTransformationWrapper::NDofs(), and DualProblemTransformationWrapper::sectors.

```
489                                                                    {
490   int dof = in_dof + 3;
491   if (dof < (int)NDofs() - 3 && dof >= 0) {
492     int sector = floor(dof / 3);
493     if (sector == sectors) {
494       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
495     } else if (sector == 0) {
496       case_sectors[0].dofs_l[dof % 3] = in_val;
497     } else {
498       case_sectors[sector].dofs_l[dof % 3] = in_val;
499       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
500     }
501   } else {
502     std::cout << "Critical: DOF-index out of bounds in "
503                  "HomogenousTransformationCircular::set_free_dof!"
504               << std::endl;
505   }
506 }
```

4.7.3 Member Data Documentation

4.7.3.1 case_sectors

```
std::vector<Sector<3> > HomogenousTransformationCircular::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 150 of file HomogenousTransformationCircular.h.

Referenced by HomogenousTransformationRectangular::estimate_and_initialize(), HomogenousTransformation↩ Rectangular::get_dof(), HomogenousTransformationRectangular::get_free_dof(), HomogenousTransformation↩ Rectangular::get_m(), HomogenousTransformationRectangular::get_Q1(), HomogenousTransformationRectangular↩ ::get_Q2(), HomogenousTransformationRectangular::get_Q3(), HomogenousTransformationRectangular::get_v(), HomogenousTransformationRectangular::PML_Z_Distance(), HomogenousTransformationRectangular::set_dof(), and HomogenousTransformationRectangular::set_free_dof().

4.7.3.2 epsilon_K

```
const double HomogenousTransformationCircular::epsilon_K
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value inside the core.

Definition at line 157 of file HomogenousTransformationCircular.h.

4.7.3.3 epsilon_M

```
const double HomogenousTransformationCircular::epsilon_M
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value outside the core.

Definition at line 163 of file HomogenousTransformationCircular.h.

4.7.3.4 sectors

```
const int HomogenousTransformationCircular::sectors
```

Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.

This member stores the number of Sectors the computational domain has been split into.

Definition at line 169 of file HomogenousTransformationCircular.h.

Referenced by HomogenousTransformationRectangular::estimate_and_initialize(), HomogenousTransformation↩ Rectangular::get_dof(), HomogenousTransformationRectangular::get_free_dof(), HomogenousTransformation↩ Rectangular::NDofs(), HomogenousTransformationRectangular::set_dof(), and HomogenousTransformation↩ Rectangular::set_free_dof().

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

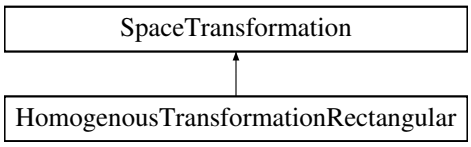
- Code/SpaceTransformations/HomogenousTransformationCircular.h
- Code/SpaceTransformations/HomogenousTransformationCircular.cpp

4.8 HomogenousTransformationRectangular Class Reference

For this transformation we try to achieve a situation in which tensorial material properties from the coordinate transformation and PML-regions dont overlap.

`#include <HomogenousTransformationRectangular.h>`

Inheritance diagram for HomogenousTransformationRectangular:



Public Member Functions

- **HomogenousTransformationRectangular** (int)
- Point< 3 > **math_to_phys** (Point< 3 > coord) const
- Point< 3 > **phys_to_math** (Point< 3 > coord) const
- bool **is_identity** (Point< 3 > coord) const
- Tensor< 2, 3, std::complex< double > > **get_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, std::complex< double > > **get_Preconditioner_Tensor** (Point< 3 > &coordinate, int block) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor_For_Preconditioner** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input, int block) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor_Homogenized** (Point< 3 > &coordinate) const
- bool PML_in_X (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.
- bool PML_in_Y (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.
- bool PML_in_Z (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.
- bool Preconditioner_PML_in_Z (Point< 3 > &p, unsigned int block) const

  Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.
- double Preconditioner_PML_Z_Distance (Point< 3 > &p, unsigned int block) const

  This function fulfills the same purpose as those with similar names but it is supposed to be used together with Preconditioner_PML_in_Z instead of the versions without "Preconditioner".
- double PML_X_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Y_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Z_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- 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_Q1 (double z) const

  This member calculates the value of Q1 for a provided z-coordinate.

- double get_Q2 (double z) const

  This member calculates the value of Q2 for a provided z-coordinate.

- double get_Q3 (double z) const

  This member calculates the value of Q3 for a provided z-coordinate.

- double get_dof (int dof) const

  This is a getter for the values of degrees of freedom.

- void set_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.

- double get_free_dof (int dof) const

  This is a getter for the values of degrees of freedom.

- void set_free_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.

- double System_Length () const

  Returns the complete length of the computational domain.

- double Sector_Length () const

  Returns the length of one sector.

- double Layer_Length () const

  Returns the length of one layer.

- double get_r (double in_z) const

  Returns the radius for a system-coordinate;.

- 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;.

- int **Z_to_Layer** (double) const
- Vector< double > Dofs () 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 NFreeDofs () const

  This function returns the number of unrestrained degrees of freedom of the current optimization run.

- unsigned int NDofs () const

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

- bool IsDofFree (int) const

  Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

- void Print () const

  Console output of the current Waveguide Structure.

- std::complex< double > evaluate_for_z (double z_in, Waveguide *)

  Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

- std::complex< double > **evaluate_for_z_with_sum** (double, double, Waveguide *)

Public Attributes

- const double **XMinus**
- const double **XPlus**
- const double **YMinus**
- const double **YPlus**
- const double **ZMinus**
- const double **ZPlus**
- std::vector< Sector< 2 > > case_sectors

  This member contains all the Sectors who, as a sum, form the complete Waveguide.
- const double epsilon_K

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const double epsilon_M

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const int sectors

  Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.
- const double deltaY

  This value is initialized with the value Delta from the input-file.
- Vector< double > InitialDofs

  This vector of values saves the initial configuration.

4.8.1 Detailed Description

For this transformation we try to achieve a situation in which tensorial material properties from the coordinate transformation and PML-regions dont overlap.

The usage of a coordinate transformation which is identity on the domain containing our PML is a strong restriction however it ensures lower errors since the quality of the PML is harder to estimate otherwise. Also it limits us in how we model the waveguide essentially forcing us to have no bent between the wavguides-connectors.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 28 of file HomogenousTransformationRectangular.h.

4.8.2 Member Function Documentation

4.8.2.1 Dofs()

```
Vector< double > HomogenousTransformationRectangular::Dofs ( ) 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.

Implements SpaceTransformation.

Definition at line 518 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::get_dof(), and HomogenousTransformationCircular::NDofs().

```
518                                                                 {
519   Vector<double> ret;
520   const int total = NDofs();
521   ret.reinit(total);
522   for (int i = 0; i < total; i++) {
523     ret[i] = get_dof(i);
524   }
525   return ret;
526 }
```

4.8.2.2 estimate_and_initialize()

```
void HomogenousTransformationRectangular::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 415 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, HomogenousTransformationCircular::Sector_↩ Length(), HomogenousTransformationCircular::sectors, and Sector< Dofs_Per_Sector >::set_properties_force().

```
415                                                                 {
416   if (GlobalParams.M_PC_Use) {
417     Sector<2> the_first(true, false, GlobalParams.sd.z[0],
418                         GlobalParams.sd.z[1]);
419     the_first.set_properties_force(GlobalParams.sd.m[0], GlobalParams.sd.m[1],
420                                    GlobalParams.sd.v[0], GlobalParams.sd.v[1]);
421     case_sectors.push_back(the_first);
422     for (int i = 1; i < GlobalParams.sd.Sectors - 2; i++) {
423       Sector<2> intermediate(false, false, GlobalParams.sd.z[i],
424                              GlobalParams.sd.z[i + 1]);
425       intermediate.set_properties_force(
426           GlobalParams.sd.m[i], GlobalParams.sd.m[i + 1], GlobalParams.sd.v[i],
427           GlobalParams.sd.v[i + 1]);
428       case_sectors.push_back(intermediate);
429     }
430     Sector<2> the_last(false, true,
```

```
431                       GlobalParams.sd.z[GlobalParams.sd.Sectors - 2],
432                       GlobalParams.sd.z[GlobalParams.sd.Sectors - 1]);
433     the_last.set_properties_force(
434         GlobalParams.sd.m[GlobalParams.sd.Sectors - 2],
435         GlobalParams.sd.m[GlobalParams.sd.Sectors - 1],
436         GlobalParams.sd.v[GlobalParams.sd.Sectors - 2],
437         GlobalParams.sd.v[GlobalParams.sd.Sectors - 1]);
438     case_sectors.push_back(the_last);
439     for (unsigned int i = 0; i < case_sectors.size(); i++) {
440       deallog << "From z: " << case_sectors[i].z_0
441               << "(m: " << case_sectors[i].get_m(0.0)
442               << " v: " << case_sectors[i].get_v(0.0) << ")" << std::endl;
443       deallog << "  To z: " << case_sectors[i].z_1
444               << "(m: " << case_sectors[i].get_m(1.0)
445               << " v: " << case_sectors[i].get_v(1.0) << ")" << std::endl;
446     }
447   } else {
448     case_sectors.reserve(sectors);
449     double m_0 = GlobalParams.M_W_Delta / 2.0;
450     double m_1 = -GlobalParams.M_W_Delta / 2.0;
451     if (sectors == 1) {
452       Sector<2> temp12(true, true, -GlobalParams.M_R_ZLength / 2.0,
453                        GlobalParams.M_R_ZLength / 2.0);
454       case_sectors.push_back(temp12);
455       case_sectors[0].set_properties_force(
456           GlobalParams.M_W_Delta / 2.0, -GlobalParams.M_W_Delta / 2.0,
457           GlobalParams.M_C_Dim1In, GlobalParams.M_C_Dim1Out, 0, 0);
458     } else {
459       double length = Sector_Length();
460       Sector<2> temp(true, false, -GlobalParams.M_R_ZLength / (2.0),
461                      -GlobalParams.M_R_ZLength / 2.0 + length);
462       case_sectors.push_back(temp);
463       for (int i = 1; i < sectors; i++) {
464         Sector<2> temp2(false, false,
465                         -GlobalParams.M_R_ZLength / (2.0) + length * (1.0 * i),
466                         -GlobalParams.M_R_ZLength / (2.0) + length * (i + 1.0));
467         case_sectors.push_back(temp2);
468       }
469
470       double length_rel = 1.0 / ((double)(sectors));
471       case_sectors[0].set_properties_force(
472           m_0, InterpolationPolynomialZeroDerivative(length_rel, m_0, m_1), 0,
473           InterpolationPolynomialDerivative(length_rel, m_0, m_1, 0, 0));
474       for (int i = 1; i < sectors; i++) {
475         double z_l = i * length_rel;
476         double z_r = (i + 1) * length_rel;
477         case_sectors[i].set_properties_force(
478             InterpolationPolynomialZeroDerivative(z_l, m_0, m_1),
479             InterpolationPolynomialZeroDerivative(z_r, m_0, m_1),
480             InterpolationPolynomialDerivative(z_l, m_0, m_1, 0, 0),
481             InterpolationPolynomialDerivative(z_r, m_0, m_1, 0, 0));
482       }
483     }
484   }
485 }
```

4.8.2.3 evaluate_for_z()

```
std::complex< double > HomogenousTransformationRectangular::evaluate_for_z (
            double ,
            Waveguide *  )  [virtual]
```

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Since I dont want to create derived classes from waveguide (which I should do eventually) I will for now include this functionality into the space transformation which is shape-sensitive. The waveguide only offers the evaluation at a point. The quadrature-rule has to be imposed by the space transformation.

Implements SpaceTransformation.

Definition at line 334 of file HomogenousTransformationRectangular.cpp.

```
335                                 {
336   double r = GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out;
337   return gauss_product_2D_sphere(in_z, 10, r, 0, 0, in_w);
338 }
```

4.8.2.4 get_dof()

```
double HomogenousTransformationRectangular::get_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 340 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, HomogenousTransformationCircular::NDofs(), and HomogenousTransformationCircular::sectors.

Referenced by InhomogenousTransformationCircular::Dofs().

```
340                                                                 {
341   if (dof < (int)NDofs() && dof >= 0) {
342     int sector = floor(dof / 2);
343     if (sector == sectors) {
344       return case_sectors[sector - 1].dofs_r[dof % 2];
345     } else {
346       return case_sectors[sector].dofs_l[dof % 2];
347     }
348   } else {
349     std::cout << "Critical: DOF-index out of bounds in "
350                  "HomogenousTransformationRectangular::get_dof!"
351               << std::endl;
352     return 0.0;
353   }
354 }
```

4.8.2.5 get_free_dof()

```
double HomogenousTransformationRectangular::get_free_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 356 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, HomogenousTransformationCircular::NDofs(), and HomogenousTransformationCircular::sectors.

```
356                                                                    {
357   int dof = in_dof + 2;
358   if (dof < (int)NDofs() - 2 && dof >= 0) {
359     int sector = floor(dof / 2);
360     if (sector == sectors) {
361       return case_sectors[sector - 1].dofs_r[dof % 2];
362     } else {
363       return case_sectors[sector].dofs_l[dof % 2];
364     }
365   } else {
366     std::cout << "Critical: DOF-index out of bounds in "
367                  "HomogenousTransformationRectangular::get_free_dof!"
368               << std::endl;
369     return 0.0;
370   }
371 }
```

4.8.2.6 get_Q1()

```
double HomogenousTransformationRectangular::get_Q1 (
            double z ) const  [virtual]
```

This member calculates the value of Q1 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q1 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 503 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, and SpaceTransformation::Z_to_Sector_and_↩ local_z().

```
503                                                          {
504   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
505   return case_sectors[two.first].getQ1(two.second);
506 }
```

4.8.2.7 get_Q2()

```
double HomogenousTransformationRectangular::get_Q2 (
            double z ) const  [virtual]
```

This member calculates the value of Q2 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q2 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 508 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, and SpaceTransformation::Z_to_Sector_and_↩ local_z().

```
508                                                                 {
509   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
510   return case_sectors[two.first].getQ2(two.second);
511 }
```

4.8.2.8 get_Q3()

```
double HomogenousTransformationRectangular::get_Q3 (
            double z ) const  [virtual]
```

This member calculates the value of Q3 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q3 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 513 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, and SpaceTransformation::Z_to_Sector_and_↩ local_z().

```
513                                                                    {
514   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
515   return case_sectors[two.first].getQ3(two.second);
516 }
```

4.8.2.9 IsDofFree()

```
bool HomogenousTransformationRectangular::IsDofFree (
            int index ) const  [virtual]
```

Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

"restrained" means that for example the DOF represents the radius at one of the connectors (input or output) and therefore we forbid the optimization scheme to vary this value.

Implements SpaceTransformation.

Definition at line 532 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::NDofs().

```
532                                                                    {
533   return index > 1 && index < (int)NDofs() - 1;
534 }
```

4.8.2.10 NDofs()

```
unsigned int HomogenousTransformationRectangular::NDofs ( ) const  [virtual]
```

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

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

Implements SpaceTransformation.

Definition at line 540 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::sectors.

Referenced by InhomogenousTransformationCircular::Dofs(), InhomogenousTransformationCircular::get_↩dof(), InhomogenousTransformationCircular::get_free_dof(), InhomogenousTransformationCircular::IsDof↩Free(), InhomogenousTransformationCircular::NFreeDofs(), InhomogenousTransformationCircular::set_dof(), and InhomogenousTransformationCircular::set_free_dof().

```
540                                                                    {
541   return sectors * 2 + 2;
542 }
```

4.8.2.11 PML_in_X()

```
bool HomogenousTransformationRectangular::PML_in_X (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 84 of file HomogenousTransformationRectangular.cpp.

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

```
84                                                                              {
85   return p(0) < XMinus || p(0) > XPlus;
86 }
```

4.8.2.12 PML_in_Y()

```
bool HomogenousTransformationRectangular::PML_in_Y (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 88 of file HomogenousTransformationRectangular.cpp.

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

```
88                                                                              {
89   return p(1) < YMinus || p(1) > YPlus;
90 }
```

4.8.2.13 PML_in_Z()

```
bool HomogenousTransformationRectangular::PML_in_Z (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 92 of file HomogenousTransformationRectangular.cpp.

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

```
92                                                                          {
93   return p(2) > ZPlus || p(2) < ZMinus;
94 }
```

4.8.2.14 PML_X_Distance()

```
double HomogenousTransformationRectangular::PML_X_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 102 of file HomogenousTransformationRectangular.cpp.

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

```
103                                {
104   if (p(0) > 0) {
105     return p(0) - XPlus;
106   } else {
107     return -p(0) + XMinus;
108   }
109 }
```

4.8.2.15 PML_Y_Distance()

```
double HomogenousTransformationRectangular::PML_Y_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 111 of file HomogenousTransformationRectangular.cpp.

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

```
112                                {
113   if (p(1) > 0) {
114     return p(1) - YPlus;
115   } else {
116     return -p(1) + YMinus;
117   }
118 }
```

4.8.2.16 PML_Z_Distance()

```
double HomogenousTransformationRectangular::PML_Z_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 120 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, HomogenousTransformationCircular::PML↩_in_X(), HomogenousTransformationCircular::PML_in_Y(), HomogenousTransformationCircular::PML_in_↩Z(), HomogenousTransformationCircular::PML_X_Distance(), HomogenousTransformationCircular::PML_Y↩_Distance(), HomogenousTransformationCircular::PML_Z_Distance(), HomogenousTransformationCircular::↩Preconditioner_PML_Z_Distance(), and SpaceTransformation::Z_to_Sector_and_local_z().

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

```
121                                {
122   if (p(2) < 0) {
123     return -(p(2) + (GlobalParams.M_R_ZLength / 2.0));
124   } else {
125     return p(2) - (GlobalParams.M_R_ZLength / 2.0);
126   }
127 }
```

4.8.2.17 Preconditioner_PML_in_Z()

```
bool HomogenousTransformationRectangular::Preconditioner_PML_in_Z (
            Point< 3 > & p,
            unsigned int block ) const
```

Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.

These Layers are not only situated at the surface of the computational domain but also inside it at the interfaces of Sectors.

Referenced by InhomogenousTransformationCircular::PML_Z_Distance().

4.8.2.18 set_dof()

```
void HomogenousTransformationRectangular::set_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 373 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, HomogenousTransformationCircular::NDofs(), and HomogenousTransformationCircular::sectors.

```
373                                                                          {
374   if (dof < (int)NDofs() && dof >= 0) {
375     int sector = floor(dof / 2);
376     if (sector == sectors) {
377       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
378     } else if (sector == 0) {
379       case_sectors[0].dofs_l[dof % 2] = in_val;
380     } else {
381       case_sectors[sector].dofs_l[dof % 2] = in_val;
382       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
383     }
384   } else {
385     std::cout << "Critical: DOF-index out of bounds in "
386                  "HomogenousTransformationRectangular::set_dof!"
387               << std::endl;
388   }
389 }
```

4.8.2.19 set_free_dof()

```
void HomogenousTransformationRectangular::set_free_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 391 of file HomogenousTransformationRectangular.cpp.

References HomogenousTransformationCircular::case_sectors, HomogenousTransformationCircular::NDofs(), and HomogenousTransformationCircular::sectors.

```
392                                                                    {
393   int dof = in_dof + 2;
394   if (dof < (int)NDofs() - 2 && dof >= 0) {
395     int sector = floor(dof / 2);
396     if (sector == sectors) {
397       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
398     } else if (sector == 0) {
399       case_sectors[0].dofs_l[dof % 2] = in_val;
400     } else {
401       case_sectors[sector].dofs_l[dof % 2] = in_val;
402       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
403     }
404   } else {
405     std::cout << "Critical: DOF-index out of bounds in "
406                  "HomogenousTransformationRectangular::set_free_dof!"
407               << std::endl;
408   }
409 }
```

4.8.3 Member Data Documentation

4.8.3.1 case_sectors

```
std::vector<Sector<2> > HomogenousTransformationRectangular::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 141 of file HomogenousTransformationRectangular.h.

Referenced by InhomogenousTransformationCircular::estimate_and_initialize(), InhomogenousTransformationCircular::get_dof(), InhomogenousTransformationCircular::get_free_dof(), InhomogenousTransformationCircular::get_m(), InhomogenousTransformationCircular::get_Q1(), InhomogenousTransformationCircular::get_Q2(), InhomogenousTransformationCircular::get_Q3(), InhomogenousTransformationCircular::get_r(), InhomogenousTransformationCircular::get_v(), InhomogenousTransformationCircular::PML_Z_Distance(), InhomogenousTransformationCircular::set_dof(), and InhomogenousTransformationCircular::set_free_dof().

4.8.3.2 epsilon_K

```
const double HomogenousTransformationRectangular::epsilon_K
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value inside the core.

Definition at line 148 of file HomogenousTransformationRectangular.h.

4.8.3.3 epsilon_M

```
const double HomogenousTransformationRectangular::epsilon_M
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value outside the core.

Definition at line 154 of file HomogenousTransformationRectangular.h.

4.8.3.4 sectors

```
const int HomogenousTransformationRectangular::sectors
```

Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.

This member stores the number of Sectors the computational domain has been split into.

Definition at line 160 of file HomogenousTransformationRectangular.h.

Referenced by InhomogenousTransformationCircular::estimate_and_initialize(), InhomogenousTransformation↩ Circular::get_dof(), InhomogenousTransformationCircular::get_free_dof(), InhomogenousTransformationCircular↩ ::NDofs(), InhomogenousTransformationCircular::set_dof(), and InhomogenousTransformationCircular::set_free_↩ dof().

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

- Code/SpaceTransformations/HomogenousTransformationRectangular.h
- Code/SpaceTransformations/HomogenousTransformationRectangular.cpp

4.9 HSIE_Dof_Type< hsie_order > Class Template Reference

Public Member Functions

- **HSIE_Dof_Type** (unsigned int in_type, unsigned int in_order, unsigned int q_count)
- void set_base_point (unsigned int)

  Base Points are numbered: 0: low x, low y; 1: low x, high y; 2: high x, high y; 3: high x, low y;.
- void set_base_edge (unsigned int)

  Edges are numbered: 0: y edge from point 0; 1: x edge from point 1; 2: y edge from point 2; 3: x edge from point 3;.
- std::complex< double > **eval_base** (std::vector< std::complex< double >> *in_base, std::complex< double > in_x)
- unsigned int get_type ()

  This describes the properties of a HSIE dof type.
- void **set_x_length** (double in_length)
- void **set_y_length** (double in_length)
- unsigned int **get_order** ()
- void **prepare_for_quadrature_points** (std::vector< dealii::Point< 2, double >> q_points)
- void **compute_IPsik** ()
- void **compute_dxiPsik** ()
- std::vector< std::complex< double > > **evaluate_U** (dealii::Point< 2, double >, double xi)
- std::vector< std::complex< double > > **evaluate_U_for_ACT** (dealii::Point< 2, double >, double xi)
- std::complex< double > **component_1a** (double x, double y, std::complex< double > xhi)
- std::complex< double > **component_1b** (double x, double y, std::complex< double > xhi)
- std::complex< double > **component_2a** (double x, double y, std::complex< double > xhi)
- std::complex< double > **component_2b** (double x, double y, std::complex< double > xhi)
- std::complex< double > **component_3a** (double x, double y, std::complex< double > xhi)
- std::complex< double > **component_3b** (double x, double y, std::complex< double > xhi)
- std::vector< std::complex< double > > **apply_T_plus** (std::vector< std::complex< double >>, double)
- std::vector< std::complex< double > > **apply_T_minus** (std::vector< std::complex< double >>, double)

4.9.1 Detailed Description

template<int hsie_order>
class HSIE_Dof_Type< hsie_order >

Definition at line 16 of file HSIEDofType.h.

4.9.2 Member Function Documentation

4.9.2.1 get_type()

```
template<int hsie_order>
unsigned int HSIE_Dof_Type< hsie_order >::get_type ( )  [inline]
```

This describes the properties of a HSIE dof type.

The versions are descirbed in 3.2 of "High order Curl-conforming Hardy space infinite elements for exterior Maxwell problems". Possible types are: 0. edge functions

1. surface functions
2. ray functions
3. infinite face functions type 1
4. infinite face functions type 2
5. segment functions type 1
6. segment functions type 2

Definition at line 147 of file HSIEDofType.cpp.

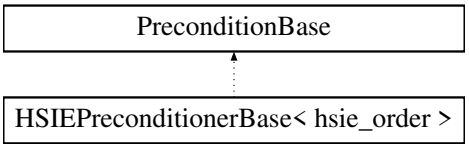
```
147                                                    {
148   return this->type;
149 }
```

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

- Code/HSIE/HSIEDofType.h
- Code/HSIE/HSIEDofType.cpp

4.10 HSIEPreconditionerBase< hsie_order > Class Template Reference

Inheritance diagram for HSIEPreconditionerBase< hsie_order >:



Public Member Functions

- HSIEPreconditionerBase (const dealii::parallel::distributed::Triangulation< 3 > *in_tria, double in_z)

  This constructor of a HSIE Predonditioner works as follows.
- unsigned int n_dofs ()

  The order of the Hardy-Space polynomials was given as a template argument to an object of this type, so no arguments are required for this function.
- unsigned int n_dofs_per_face ()

  Similar to the function n_dofs() this function doesn't currently need any arguments.
- void assemble_block ()

  This function brings the Object-member HSIEPreconditionerBase::system_matrix which is currently supposed to hold all HSIE-dofs (including the Nedelec-elements of the surface elements).
- void setup_system ()

  This must be called before assemble block since it initializes the HSIEPreconditionerBase::system_matrix.
- std::complex< double > a (HSIE_Dof_Type< hsie_order > u, HSIE_Dof_Type< hsie_order > v, bool use_curl_fomulation, dealii::Point< 2, double > x)

  This function implements the following equation:

$$a(U,V) := \frac{-2\,\mathrm{i}\,\kappa_0}{2\pi} \int_{S_0} U(z) v(\bar{z}) |\mathrm{d}z|$$

  which is called from A() which in turn is required to build the HSIEPreconditionerBase::system_matrix.
- std::complex< double > A (HSIE_Dof_Type< hsie_order > u, HSIE_Dof_Type< hsie_order > v, dealii::Tensor< 2, 3, double > G, bool use_curl_fomulation)

  This function implements the following equation:

$$A(U,V) := \int_T \sum_{i,j=1}^{3} g_{ij}(\hat{x}) a(U_i(\cdot,\hat{x}), V_j(\cdot,\hat{x})) \mathrm{d}\hat{x}$$

  where T is a Triangulation of the surface.

Public Attributes

- dealii::TrilinosWrappers::SparseMatrix system_matrix

  This matrix contains the couplings between all HSIE-dofs.

4.10.1 Detailed Description

template<int hsie_order>
class HSIEPreconditionerBase< hsie_order >

Definition at line 20 of file HSIE_Preconditioner_Base.h.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 HSIEPreconditionerBase()

```
template<int hsie_order>
HSIEPreconditionerBase< hsie_order >::HSIEPreconditionerBase (
            const dealii::parallel::distributed::Triangulation< 3 > * in_tria,
            double in_z )
```

This constructor of a HSIE Predonditioner works as follows.

It takes a given Triangulation and cuts it at a specified z-coordinate and applies infinite elements to the surface generated in this way. This class will be renamed appropriately later since it is not really a Preconditioner as much as it is a general implementation of the HSIE method which can also be employed as part of a preconditioner.

Parameters

in_tria	A handle to the triangulation which contains the surface.
in_z	the z-coordinate at which to attach the infinite Elements.

Definition at line 192 of file HSIE_Preconditioner_Base.cpp.

```
194                 {
195   FaceSurfaceComparatorZ fscz = new FaceSurfaceComparatorZ(in_z
      , 0.00001);
196   association = extract_surface_mesh_at_z(in_tria, &surf_tria, fscz);
197   surface_edges = surf_tria.n_active_lines();
198   surface_faces = surf_tria.n_active_faces();
199   surface_vertices = surf_tria.n_vertices();
200   HSIE_dofs_type_1_factor = dealii::GeometryInfo<2>::lines_per_face * 2;
201   HSIE_dofs_type_3_factor =
202       dealii::GeometryInfo<2>::lines_per_face * 2 * (hsie_order + 1);
203   HSIE_dofs_type_2_factor =
204       dealii::GeometryInfo<2>::vertices_per_face * 2 * (hsie_order + 1);
205   HSIE_degree = hsie_order;
206   fe_nedelec(GlobalParams.So_ElementOrder);
207   fe_q(1);
208   quadrature_formula(2);
209   hsie_dof_handler(surf_tria);
210   FEValues<3> fev_nedelec(fe_nedelec, quadrature_formula,
211                           update_values | update_gradients | update_JxW_values |
212                               update_quadrature_points);
213   FEValues<3> fev_q(fe_q, quadrature_formula,
214                     update_values | update_gradients | update_JxW_values |
215                         update_quadrature_points);
216 }
```

4.10.3 Member Function Documentation

4.10.3.1 a()

```
template<int hsie_order>
std::complex< double > HSIEPreconditionerBase< hsie_order >::a (
            HSIE_Dof_Type< hsie_order > u,
            HSIE_Dof_Type< hsie_order > v,
            bool use_curl_fomulation,
            dealii::Point< 2, double > x )
```

This function implements the following equation:

$$a(U,V) := \frac{-2\,\mathrm{i}\,\kappa_0}{2\pi} \int_{S_0} U(z) v(\bar{z}) |\mathrm{d}z|$$

which is called from A() which in turn is required to build the HSIEPreconditionerBase::system_matrix.

Parameters

u	is an object describing the first type of dof,
v	describes the second dof,
use_curl_formulation	simplifies the usage of this function: It causes the function not to use u_1 etc. but the terms in A()'s argument list (see equation (30) in the paper High order Curl-conforming Hardy space infinite elements for exterior Maxwell problems.)
x	is the Point in the surface triangulation where to evaluate the bilinear form.

Definition at line 303 of file HSIE_Preconditioner_Base.cpp.

```
305                                                    {}
```

4.10.3.2 A()

```
template<int hsie_order>
std::complex< double > HSIEPreconditionerBase< hsie_order >::A (
            HSIE_Dof_Type< hsie_order > u,
            HSIE_Dof_Type< hsie_order > v,
            dealii::Tensor< 2, 3, double > G,
            bool use_curl_fomulation )
```

This function implements the following equation:

$$A(U,V) := \int_T \sum_{i,j=1}^{3} g_{ij}(\hat{x}) a(U_i(\cdot,\hat{x}), V_j(\cdot,\hat{x})) \mathrm{d}\hat{x}$$

where T is a Triangulation of the surface.

For more details see the publication mentioned in the description of a().

Parameters

u	is an object describing the first type of dof,
v	describes the second dof,
G	is the $3 3$ matrix consisting of g_{ij}
use_curl_formulation	has the same purpose as in a().

Definition at line 308 of file HSIE_Preconditioner_Base.cpp.

```
310                                                    {
311   dealii::QGauss<2> quadrature_formula(2);
312   std::complex<double> ret(0, 0);
313   std::vector<dealii::Point<2>> quad_points =
314       quadrature_formula.quadrature_points;
315   for (unsigned int i = 0; i < quad_points.size(); i++) {
316     for (unsigned int j = 0; j < 3; j++) {
317       for (unsigned int k = 0; k < 3; k++) {
318         ret += quadrature_formula.weight(i) * G[j, k] *
319                a(u, v, true, quad_points[i]);
320       }
321     }
322   }
323   return ret;
324 }
```

4.10.3.3 assemble_block()

```
template<int hsie_order>
void HSIEPreconditionerBase< hsie_order >::assemble_block ( )
```

This function brings the Object-member HSIEPreconditionerBase::system_matrix which is currently supposed to hold all HSIE-dofs (including the Nedelec-elements of the surface elements).

This makes coupling the dofs to the interior via dof-constraints necessary.

4.10.3.4 n_dofs()

```
template<int hsie_order>
unsigned int HSIEPreconditionerBase< hsie_order >::n_dofs ( )
```

The order of the Hardy-Space polynomials was given as a template argument to an object of this type, so no arguments are required for this function.

Eventually a version of this function will be added which can deal with higher then lowest order elements in the interior.

Definition at line 285 of file HSIE_Preconditioner_Base.cpp.

```
285                                                    {
286   unsigned int ret = 0;
287   ret += surface_edges * HSIE_dofs_type_1_factor;
288   ret += surface_vertices * HSIE_dofs_type_2_factor;
289   ret += surface_edges * HSIE_dofs_type_3_factor;
290   return ret;
291 }
```

4.10.3.5 n_dofs_per_face()

```
template<int hsie_order>
unsigned int HSIEPreconditionerBase< hsie_order >::n_dofs_per_face ( )
```

Similar to the function n_dofs() this function doesn't currently need any arguments.

This one however returns the number of HSIE-Dofs per face, which will act as a replacement of dealii's similar functions dofs_per_face etc.

Definition at line 294 of file HSIE_Preconditioner_Base.cpp.

```
294                                                        {
295   unsigned int ret = 0;
296   ret += 4 * HSIE_dofs_type_1_factor;
297   ret += 4 * HSIE_dofs_type_2_factor;
298   ret += 4 * HSIE_dofs_type_3_factor;
299   return ret;
300 }
```

4.10.4 Member Data Documentation

4.10.4.1 system_matrix

```
template<int hsie_order>
dealii::TrilinosWrappers::SparseMatrix HSIEPreconditionerBase< hsie_order >::system_matrix
```

This matrix contains the couplings between all HSIE-dofs.

This also contains the Nedelec-Elements of the surface triangulation.

Definition at line 122 of file HSIE_Preconditioner_Base.h.

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

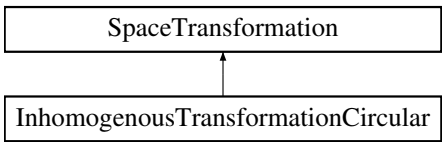
- Code/HSIE/HSIE_Preconditioner_Base.h
- Code/HSIE/HSIE_Preconditioner_Base.cpp

4.11 InhomogenousTransformationCircular Class Reference

In this case we regard a tubular waveguide and the effects on the material tensor by the space transformation and the boundary condition PML may overlap (hence inhomogenous space transformation)

`#include <InhomogenousTransformationCircular.h>`

Inheritance diagram for InhomogenousTransformationCircular:



Public Member Functions

- **InhomogenousTransformationCircular** (int)
- Point< 3 > **math_to_phys** (Point< 3 > coord) const
- Point< 3 > **phys_to_math** (Point< 3 > coord) const
- bool **is_identity** (Point< 3 > coord) const
- Tensor< 2, 3, std::complex< double > > **get_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, std::complex< double > > **get_Preconditioner_Tensor** (Point< 3 > &coordinate, int block) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor_For_Preconditioner** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input, int block) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor_Homogenized** (Point< 3 > &coordinate) const
- bool PML_in_X (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.
- bool PML_in_Y (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.
- bool PML_in_Z (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.
- bool Preconditioner_PML_in_Z (Point< 3 > &p, unsigned int block) const

  Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.
- double Preconditioner_PML_Z_Distance (Point< 3 > &p, unsigned int block) const

  This function fulfills the same purpose as those with similar names but it is supposed to be used together with Preconditioner_PML_in_Z instead of the versions without "Preconditioner".
- double PML_X_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Y_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Z_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- 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_Q1 (double z) const

  This member calculates the value of Q1 for a provided z-coordinate.
- double get_Q2 (double z) const

  This member calculates the value of Q2 for a provided z-coordinate.
- double get_Q3 (double z) const

  This member calculates the value of Q3 for a provided z-coordinate.
- double get_dof (int dof) const

  This is a getter for the values of degrees of freedom.
- void set_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.
- double get_free_dof (int dof) const

  This is a getter for the values of degrees of freedom.
- void set_free_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.
- double System_Length () const

  Using this method unifies the usage of coordinates.
- double Sector_Length () const

  Returns the length of one sector.
- double Layer_Length () const

  Returns the length of one layer.
- double get_r (double in_z) const

  Returns the radius for a system-coordinate;.
- 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;.
- int **Z_to_Layer** (double) const
- Vector< double > Dofs () 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 NFreeDofs () const

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

  This function returns the total number of DOFs including restrained ones.
- bool IsDofFree (int) const

  Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.
- void Print () const

  Console output of the current Waveguide Structure.
- std::complex< double > evaluate_for_z (double z_in, Waveguide *)

  Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.
- std::complex< double > **evaluate_for_z_with_sum** (double, double, Waveguide *)
- std::complex< double > **gauss_product_2D_sphere** (double z, int n, double R, double Xc, double Yc, Waveguide *in_w)

Public Attributes

- const double **XMinus**
- const double **XPlus**
- const double **YMinus**
- const double **YPlus**
- const double **ZMinus**
- const double **ZPlus**
- std::vector< Sector< 3 > > case_sectors

  This member contains all the Sectors who, as a sum, form the complete Waveguide.
- const double epsilon_K

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const double epsilon_M

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const int sectors

  Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.
- const double deltaY

  This value is initialized with the value Delta from the input-file.
- Vector< double > InitialDofs

  This vector of values saves the initial configuration.

4.11.1 Detailed Description

In this case we regard a tubular waveguide and the effects on the material tensor by the space transformation and the boundary condition PML may overlap (hence inhomogenous space transformation)

The usage of a coordinate transformation which is identity on the domain containing our PML is a strong restriction however it ensures lower errors since the quality of the PML is harder to estimate otherwise. Also it limits us in how we model the waveguide essentially forcing us to have no bent between the wavguides-connectors.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 28 of file InhomogenousTransformationCircular.h.

4.11.2 Member Function Documentation

4.11.2.1 Dofs()

```
Vector< double > InhomogenousTransformationCircular::Dofs ( ) 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.

Implements SpaceTransformation.

Definition at line 551 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::get_dof(), and HomogenousTransformationRectangular::N↩ Dofs().

```
551                                                                 {
552   Vector<double> ret;
553   const int total = NDofs();
554   ret.reinit(total);
555   for (int i = 0; i < total; i++) {
556     ret[i] = get_dof(i);
557   }
558   return ret;
559 }
```

4.11.2.2 estimate_and_initialize()

```
void InhomogenousTransformationCircular::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 473 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, HomogenousTransformationRectangular::↩ Sector_Length(), and HomogenousTransformationRectangular::sectors.

```
473                                                                 {
474   case_sectors.reserve(sectors);
475   double m_0 = GlobalParams.M_W_Delta / 2.0;
476   double m_1 = -GlobalParams.M_W_Delta / 2.0;
477   double r_0 = GlobalParams.M_C_Dim1In;
478   double r_1 = GlobalParams.M_C_Dim1Out;
479   if (sectors == 1) {
480     Sector<3> temp12(true, true, -GlobalParams.M_R_ZLength / 2,
481                      GlobalParams.M_R_ZLength / 2);
482     case_sectors.push_back(temp12);
483     case_sectors[0].set_properties_force(
484         GlobalParams.M_W_Delta / 2.0, -GlobalParams.M_W_Delta / 2.0,
485         GlobalParams.M_C_Dim1In, GlobalParams.M_C_Dim1Out, 0, 0);
486   } else {
```

```
487     double length = Sector_Length();
488     Sector<3> temp(true, false, -GlobalParams.M_R_ZLength / (2.0),
489                    -GlobalParams.M_R_ZLength / 2.0 + length);
490     case_sectors.push_back(temp);
491     for (int i = 1; i < sectors; i++) {
492       Sector<3> temp2(false, false,
493                       -GlobalParams.M_R_ZLength / (2.0) + length * (1.0 * i),
494                       -GlobalParams.M_R_ZLength / (2.0) + length * (i + 1.0));
495       case_sectors.push_back(temp2);
496     }
497
498     double length_rel = 1.0 / ((double)(sectors));
499     case_sectors[0].set_properties_force(
500         m_0, InterpolationPolynomialZeroDerivative(length_rel, m_0, m_1), r_0,
501         InterpolationPolynomialZeroDerivative(length_rel, r_0, r_1), 0,
502         InterpolationPolynomialDerivative(length_rel, m_0, m_1, 0, 0));
503     for (int i = 1; i < sectors; i++) {
504       double z_l = i * length_rel;
505       double z_r = (i + 1) * length_rel;
506       case_sectors[i].set_properties_force(
507           InterpolationPolynomialZeroDerivative(z_l, m_0, m_1),
508           InterpolationPolynomialZeroDerivative(z_r, m_0, m_1),
509           InterpolationPolynomialZeroDerivative(z_l, r_0, r_1),
510           InterpolationPolynomialZeroDerivative(z_r, r_0, r_1),
511           InterpolationPolynomialDerivative(z_l, m_0, m_1, 0, 0),
512           InterpolationPolynomialDerivative(z_r, m_0, m_1, 0, 0));
513     }
514   }
515
516   // for (unsigned int i = 0;  i < NFreeDofs(); ++ i) {
517   //  InitialDofs[i] = this->get_dof(i, true);
518   //}
519 }
```

4.11.2.3 evaluate_for_z()

```
std::complex< double > InhomogenousTransformationCircular::evaluate_for_z (
            double ,
            Waveguide *  )  [virtual]
```

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Since I dont want to create derived classes from waveguide (which I should do eventually) I will for now include this functionality into the space transformation which is shape-sensitive. The waveguide only offers the evaluation at a point. The quadrature-rule has to be imposed by the space transformation.

Implements SpaceTransformation.

Definition at line 392 of file InhomogenousTransformationCircular.cpp.

```
393                                 {
394   double r = GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out;
395   return gauss_product_2D_sphere(in_z, 10, r, 0, 0, in_w);
396 }
```

4.11.2.4 get_dof()

```
double InhomogenousTransformationCircular::get_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 398 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, HomogenousTransformationRectangular::N↩ Dofs(), and HomogenousTransformationRectangular::sectors.

Referenced by InhomogenousTransformationRectangular::Dofs().

```
398                                                                  {
399   if (dof < (int)NDofs() && dof >= 0) {
400     int sector = floor(dof / 3);
401     if (sector == sectors) {
402       return case_sectors[sector - 1].dofs_r[dof % 3];
403     } else {
404       return case_sectors[sector].dofs_l[dof % 3];
405     }
406   } else {
407     std::cout << "Critical: DOF-index out of bounds in "
408                  "HomogenousTransformationCircular::get_dof!"
409               << std::endl;
410     return 0.0;
411   }
412 }
```

4.11.2.5 get_free_dof()

```
double InhomogenousTransformationCircular::get_free_dof (
             int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 414 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, HomogenousTransformationRectangular::N↩ Dofs(), and HomogenousTransformationRectangular::sectors.

```
414                                                                          {
415   int dof = in_dof + 3;
416   if (dof < (int)NDofs() - 3 && dof >= 0) {
417     int sector = floor(dof / 3);
418     if (sector == sectors) {
419       return case_sectors[sector - 1].dofs_r[dof % 3];
420     } else {
421       return case_sectors[sector].dofs_l[dof % 3];
422     }
423   } else {
424     std::cout << "Critical: DOF-index out of bounds in "
425                  "HomogenousTransformationCircular::get_free_dof!"
426               << std::endl;
427     return 0.0;
428   }
429 }
```

4.11.2.6 get_Q1()

```
double InhomogenousTransformationCircular::get_Q1 (
            double z ) const  [virtual]
```

This member calculates the value of Q1 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q1 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 536 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, and SpaceTransformation::Z_to_Sector_↩ and_local_z().

```
536                                                                  {
537   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
538   return case_sectors[two.first].getQ1(two.second);
539 }
```

4.11.2.7 get_Q2()

```
double InhomogenousTransformationCircular::get_Q2 (
            double z ) const  [virtual]
```

This member calculates the value of Q2 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q2 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 541 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, and SpaceTransformation::Z_to_Sector_↩ and_local_z().

```
541                                                                 {
542   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
543   return case_sectors[two.first].getQ2(two.second);
544 }
```

4.11.2.8 get_Q3()

```
double InhomogenousTransformationCircular::get_Q3 (
            double z ) const  [virtual]
```

This member calculates the value of Q3 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q3 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 546 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, and SpaceTransformation::Z_to_Sector_↩ and_local_z().

```
546                                                                 {
547   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
548   return case_sectors[two.first].getQ3(two.second);
549 }
```

4.11.2.9 IsDofFree()

```
bool InhomogenousTransformationCircular::IsDofFree (
            int index ) const  [virtual]
```

Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

"restrained" means that for example the DOF represents the radius at one of the connectors (input or output) and therefore we forbid the optimization scheme to vary this value.

Implements SpaceTransformation.

Definition at line 565 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::NDofs().

```
565                                                                    {
566   return index > 2 && index < (int)NDofs() - 3;
567 }
```

4.11.2.10 NDofs()

```
unsigned int InhomogenousTransformationCircular::NDofs ( ) const  [virtual]
```

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

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

Implements SpaceTransformation.

Definition at line 573 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::sectors.

Referenced by InhomogenousTransformationRectangular::Dofs(), InhomogenousTransformationRectangular::get_dof(), InhomogenousTransformationRectangular::get_free_dof(), InhomogenousTransformationRectangular::IsDofFree(), InhomogenousTransformationRectangular::NFreeDofs(), InhomogenousTransformationRectangular::set_dof(), and InhomogenousTransformationRectangular::set_free_dof().

```
573                                                          {
574   return sectors * 3 + 3;
575 }
```

4.11.2.11 PML_in_X()

```
bool InhomogenousTransformationCircular::PML_in_X (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 76 of file InhomogenousTransformationCircular.cpp.

Referenced by InhomogenousTransformationRectangular::PML_Z_Distance().

```
76                                                                  {
77   return p(0) < XMinus || p(0) > XPlus;
78 }
```

4.11.2.12 PML_in_Y()

```
bool InhomogenousTransformationCircular::PML_in_Y (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 80 of file InhomogenousTransformationCircular.cpp.

Referenced by InhomogenousTransformationRectangular::PML_Z_Distance().

```
80                                                                  {
81   return p(1) < YMinus || p(1) > YPlus;
82 }
```

4.11.2.13 PML_in_Z()

```
bool InhomogenousTransformationCircular::PML_in_Z (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 84 of file InhomogenousTransformationCircular.cpp.

Referenced by InhomogenousTransformationRectangular::PML_Z_Distance().

```
84                                                                  {
85   return p(2) > ZPlus;
86 }
```

4.11.2.14 PML_X_Distance()

```
double InhomogenousTransformationCircular::PML_X_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 105 of file InhomogenousTransformationCircular.cpp.

Referenced by InhomogenousTransformationRectangular::PML_Z_Distance().

```
105                                                                 {
106   if (p(0) > 0) {
107     return p(0) - XPlus;
108   } else {
109     return -p(0) - XMinus;
110   }
111 }
```

4.11.2.15 PML_Y_Distance()

```
double InhomogenousTransformationCircular::PML_Y_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 113 of file InhomogenousTransformationCircular.cpp.

Referenced by InhomogenousTransformationRectangular::PML_Z_Distance().

```
113                                                                          {
114   if (p(1) > 0) {
115     return p(1) - YMinus;
116   } else {
117     return -p(1) - YPlus;
118   }
119 }
```

4.11.2.16 PML_Z_Distance()

```
double InhomogenousTransformationCircular::PML_Z_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 121 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, Waveguide::evaluate_for_Position(), HomogenousTransformationRectangular::PML_in_X(), HomogenousTransformationRectangular::PML_in_Y(), HomogenousTransformationRectangular::PML_in_Z(), HomogenousTransformationRectangular::PML_X_Distance(), HomogenousTransformationRectangular::PML_Y_Distance(), HomogenousTransformationRectangular::PML_Z_Distance(), HomogenousTransformationRectangular::Preconditioner_PML_in_Z(), HomogenousTransformationRectangular::Preconditioner_PML_Z_Distance(), and SpaceTransformation::Z_to_Sector_and_local_z().

Referenced by InhomogenousTransformationRectangular::PML_Z_Distance().

```
121                                                                          {
122   if (p(2) < 0) {
123     return 0;
124   } else {
125     return p(2) - (GlobalParams.M_R_ZLength / 2.0);
126   }
127 }
```

4.11.2.17 Preconditioner_PML_in_Z()

```
bool InhomogenousTransformationCircular::Preconditioner_PML_in_Z (
            Point< 3 > & p,
            unsigned int block ) const
```

Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.

These Layers are not only situated at the surface of the computational domain but also inside it at the interfaces of Sectors.

Definition at line 88 of file InhomogenousTransformationCircular.cpp.

```
89                                          {
90   if ((int)block == GlobalParams.NumberProcesses - 2) return false;
91   if ((int)block == (int)GlobalParams.MPI_Rank - 1) {
92     return true;
93   } else {
94     return false;
95   }
96 }
```

4.11.2.18 set_dof()

```
void InhomogenousTransformationCircular::set_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 431 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, HomogenousTransformationRectangular::NDofs(), and HomogenousTransformationRectangular::sectors.

```
431                                                                         {
432   if (dof < (int)NDofs() && dof >= 0) {
433     int sector = floor(dof / 3);
434     if (sector == sectors) {
435       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
436     } else if (sector == 0) {
437       case_sectors[0].dofs_l[dof % 3] = in_val;
438     } else {
439       case_sectors[sector].dofs_l[dof % 3] = in_val;
440       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
441     }
```

```
442   } else {
443     std::cout << "Critical: DOF-index out of bounds in "
444                  "HomogenousTransformationCircular::set_dof!"
445               << std::endl;
446   }
447 }
```

4.11.2.19 set_free_dof()

```
void InhomogenousTransformationCircular::set_free_dof (
              int dof,
              double value )  [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.

Implements SpaceTransformation.

Definition at line 449 of file InhomogenousTransformationCircular.cpp.

References HomogenousTransformationRectangular::case_sectors, HomogenousTransformationRectangular::N↩ Dofs(), and HomogenousTransformationRectangular::sectors.

```
450                                                                    {
451   int dof = in_dof + 3;
452   if (dof < (int)NDofs() - 3 && dof >= 0) {
453     int sector = floor(dof / 3);
454     if (sector == sectors) {
455       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
456     } else if (sector == 0) {
457       case_sectors[0].dofs_l[dof % 3] = in_val;
458     } else {
459       case_sectors[sector].dofs_l[dof % 3] = in_val;
460       case_sectors[sector - 1].dofs_r[dof % 3] = in_val;
461     }
462   } else {
463     std::cout << "Critical: DOF-index out of bounds in "
464                  "HomogenousTransformationCircular::set_free_dof!"
465               << std::endl;
466   }
467 }
```

4.11.2.20 System_Length()

```
double InhomogenousTransformationCircular::System_Length ( ) const
```

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. Returns the complete length of the computational domain.

4.11.3 Member Data Documentation

4.11.3.1 case_sectors

```
std::vector<Sector<3> > InhomogenousTransformationCircular::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 141 of file InhomogenousTransformationCircular.h.

Referenced by InhomogenousTransformationRectangular::estimate_and_initialize(), InhomogenousTransformationRectangular::get_dof(), InhomogenousTransformationRectangular::get_free_dof(), InhomogenousTransformationRectangular::get_m(), InhomogenousTransformationRectangular::get_Q1(), InhomogenousTransformationRectangular::get_Q2(), InhomogenousTransformationRectangular::get_Q3(), InhomogenousTransformationRectangular::get_v(), InhomogenousTransformationRectangular::NDofs(), InhomogenousTransformationRectangular::set_dof(), and InhomogenousTransformationRectangular::set_free_dof().

4.11.3.2 epsilon_K

```
const double InhomogenousTransformationCircular::epsilon_K
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value inside the core.

Definition at line 148 of file InhomogenousTransformationCircular.h.

4.11.3.3 epsilon_M

```
const double InhomogenousTransformationCircular::epsilon_M
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value outside the core.

Definition at line 154 of file InhomogenousTransformationCircular.h.

4.11.3.4 sectors

```
const int InhomogenousTransformationCircular::sectors
```

Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.

This member stores the number of Sectors the computational domain has been split into.

Definition at line 160 of file InhomogenousTransformationCircular.h.

Referenced by InhomogenousTransformationRectangular::estimate_and_initialize(), InhomogenousTransformation↩ Rectangular::get_dof(), InhomogenousTransformationRectangular::get_free_dof(), InhomogenousTransformation↩ Rectangular::NDofs(), InhomogenousTransformationRectangular::set_dof(), and InhomogenousTransformation↩ Rectangular::set_free_dof().

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

- Code/SpaceTransformations/InhomogenousTransformationCircular.h
- Code/SpaceTransformations/InhomogenousTransformationCircular.cpp

4.12 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 <InhomogenousTransformationRectangular.h>
```

4.12.1 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

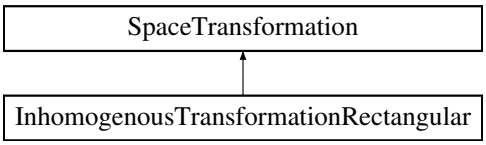
28.11.2016

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

- Code/SpaceTransformations/InhomogenousTransformationRectangular.h

4.13 InhomogenousTransformationRectangular Class Reference

Inheritance diagram for InhomogenousTransformationRectangular:



Public Member Functions

- **InhomogenousTransformationRectangular** (int)
- Point< 3 > **math_to_phys** (Point< 3 > coord) const
- Point< 3 > **phys_to_math** (Point< 3 > coord) const
- bool **is_identity** (Point< 3 > coord) const
- Tensor< 2, 3, std::complex< double > > **get_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, std::complex< double > > **get_Preconditioner_Tensor** (Point< 3 > &coordinate, int block) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input) const
- Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor_For_Preconditioner** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input, int block) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Point< 3 > &coordinate) const
- Tensor< 2, 3, double > **get_Space_Transformation_Tensor_Homogenized** (Point< 3 > &coordinate) const
- bool PML_in_X (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.
- bool PML_in_Y (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.
- bool PML_in_Z (Point< 3 > &position) const

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.
- bool Preconditioner_PML_in_Z (Point< 3 > &p, unsigned int block) const

  Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.
- double Preconditioner_PML_Z_Distance (Point< 3 > &p, unsigned int block) const

  This function fulfills the same purpose as those with similar names but it is supposed to be used together with Preconditioner_PML_in_Z instead of the versions without "Preconditioner".
- double PML_X_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Y_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- double PML_Z_Distance (Point< 3 > &position) const

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- 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_Q1 (double z) const

  This member calculates the value of Q1 for a provided z-coordinate.
- double get_Q2 (double z) const

This member calculates the value of Q2 for a provided z-coordinate.

- double get_Q3 (double z) const

  This member calculates the value of Q3 for a provided z-coordinate.
- double get_dof (int dof) const

  This is a getter for the values of degrees of freedom.
- void set_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.
- double get_free_dof (int dof) const

  This is a getter for the values of degrees of freedom.
- void set_free_dof (int dof, double value)

  This function sets the value of the dof provided to the given value.
- double System_Length () const

  Returns the complete length of the computational domain.
- double Sector_Length () const

  Returns the length of one sector.
- double Layer_Length () const

  Returns the length of one layer.
- double get_r (double in_z) const

  Returns the radius for a system-coordinate;.
- 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;.
- int **Z_to_Layer** (double) const
- Vector< double > Dofs () 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 NFreeDofs () const

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

  This function returns the total number of DOFs including restrained ones.
- bool IsDofFree (int) const

  Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.
- void Print () const

  Console output of the current Waveguide Structure.
- std::complex< double > **evaluate_for_z_with_sum** (double, double, Waveguide *)
- std::complex< double > evaluate_for_z (double z_in, Waveguide *)

  Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Public Attributes

- const double **XMinus**
- const double **XPlus**
- const double **YMinus**
- const double **YPlus**
- const double **ZMinus**
- const double **ZPlus**
- std::vector< Sector< 2 > > case_sectors

  This member contains all the Sectors who, as a sum, form the complete Waveguide.

- const double epsilon_K

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const double epsilon_M

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const int sectors

  Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.
- const double deltaY

  This value is initialized with the value Delta from the input-file.
- Vector< double > InitialDofs

  This vector of values saves the initial configuration.

4.13.1 Detailed Description

Definition at line 26 of file InhomogenousTransformationRectangular.h.

4.13.2 Member Function Documentation

4.13.2.1 Dofs()

```
Vector< double > InhomogenousTransformationRectangular::Dofs ( ) 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.

Implements SpaceTransformation.

Definition at line 445 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::get_dof(), and InhomogenousTransformationCircular::NDofs().

```
445                                                                 {
446   Vector<double> ret;
447   const int total = NDofs();
448   ret.reinit(total);
449   for (int i = 0; i < total; i++) {
450     ret[i] = get_dof(i);
451   }
452   return ret;
453 }
```

4.13.2.2 estimate_and_initialize()

```
void InhomogenousTransformationRectangular::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 342 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, InhomogenousTransformationCircular::Sector_↩ Length(), InhomogenousTransformationCircular::sectors, and Sector< Dofs_Per_Sector >::set_properties_force().

```
342                                                                    {
343   if (GlobalParams.M_PC_Use) {
344     Sector<2> the_first(true, false, GlobalParams.sd.z[0],
345                         GlobalParams.sd.z[1]);
346     the_first.set_properties_force(GlobalParams.sd.m[0], GlobalParams.sd.m[1],
347                                    GlobalParams.sd.v[0], GlobalParams.sd.v[1]);
348     case_sectors.push_back(the_first);
349     for (int i = 1; i < GlobalParams.sd.Sectors - 2; i++) {
350       Sector<2> intermediate(false, false, GlobalParams.sd.z[i],
351                              GlobalParams.sd.z[i + 1]);
352       intermediate.set_properties_force(
353           GlobalParams.sd.m[i], GlobalParams.sd.m[i + 1], GlobalParams.sd.v[i],
354           GlobalParams.sd.v[i + 1]);
355       case_sectors.push_back(intermediate);
356     }
357     Sector<2> the_last(false, true,
358                        GlobalParams.sd.z[GlobalParams.sd.Sectors - 2],
359                        GlobalParams.sd.z[GlobalParams.sd.Sectors - 1]);
360     the_last.set_properties_force(
361         GlobalParams.sd.m[GlobalParams.sd.Sectors - 2],
362         GlobalParams.sd.m[GlobalParams.sd.Sectors - 1],
363         GlobalParams.sd.v[GlobalParams.sd.Sectors - 2],
364         GlobalParams.sd.v[GlobalParams.sd.Sectors - 1]);
365     case_sectors.push_back(the_last);
366     for (unsigned int i = 0; i < case_sectors.size(); i++) {
367       deallog << "From z: " << case_sectors[i].z_0
368               << "(m: " << case_sectors[i].get_m(0.0)
369               << " v: " << case_sectors[i].get_v(0.0) << ")" << std::endl;
370       deallog << "  To z: " << case_sectors[i].z_1
371               << "(m: " << case_sectors[i].get_m(1.0)
372               << " v: " << case_sectors[i].get_v(1.0) << ")" << std::endl;
373     }
374   } else {
375     case_sectors.reserve(sectors);
376     double m_0 = GlobalParams.M_W_Delta / 2.0;
377     double m_1 = -GlobalParams.M_W_Delta / 2.0;
378     if (sectors == 1) {
379       Sector<2> temp12(true, true, -GlobalParams.M_R_ZLength / 2.0,
380                        GlobalParams.M_R_ZLength / 2.0);
381       case_sectors.push_back(temp12);
382       case_sectors[0].set_properties_force(
383           GlobalParams.M_W_Delta / 2.0, -GlobalParams.M_W_Delta / 2.0,
384           GlobalParams.M_C_Dim1In, GlobalParams.M_C_Dim1Out, 0, 0);
385     } else {
386       double length = Sector_Length();
387       Sector<2> temp(true, false, -GlobalParams.M_R_ZLength / (2.0),
388                      -GlobalParams.M_R_ZLength / 2.0 + length);
389       case_sectors.push_back(temp);
390       for (int i = 1; i < sectors; i++) {
391         Sector<2> temp2(false, false,
392                         -GlobalParams.M_R_ZLength / (2.0) + length * (1.0 * i),
393                         -GlobalParams.M_R_ZLength / (2.0) + length * (i + 1.0));
394         case_sectors.push_back(temp2);
395       }
396
397       double length_rel = 1.0 / ((double)(sectors));
398       case_sectors[0].set_properties_force(
399           m_0, InterpolationPolynomialZeroDerivative(length_rel, m_0, m_1), 0,
400           InterpolationPolynomialDerivative(length_rel, m_0, m_1, 0, 0));
```

```
401       for (int i = 1; i < sectors; i++) {
402         double z_l = i * length_rel;
403         double z_r = (i + 1) * length_rel;
404         case_sectors[i].set_properties_force(
405             InterpolationPolynomialZeroDerivative(z_l, m_0, m_1),
406             InterpolationPolynomialZeroDerivative(z_r, m_0, m_1),
407             InterpolationPolynomialDerivative(z_l, m_0, m_1, 0, 0),
408             InterpolationPolynomialDerivative(z_r, m_0, m_1, 0, 0));
409       }
410     }
411   }
412 }
```

4.13.2.3 evaluate_for_z()

```
std::complex< double > InhomogenousTransformationRectangular::evaluate_for_z (
            double ,
            Waveguide *  )  [virtual]
```

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Since I dont want to create derived classes from waveguide (which I should do eventually) I will for now include this functionality into the space transformation which is shape-sensitive. The waveguide only offers the evaluation at a point. The quadrature-rule has to be imposed by the space transformation.

Implements SpaceTransformation.

Definition at line 261 of file InhomogenousTransformationRectangular.cpp.

```
262                                 {
263   double r = GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out;
264   return gauss_product_2D_sphere(in_z, 10, r, 0, 0, in_w);
265 }
```

4.13.2.4 get_dof()

```
double InhomogenousTransformationRectangular::get_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 267 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, InhomogenousTransformationCircular::NDofs(), and InhomogenousTransformationCircular::sectors.

```
267                                                                    {
268   if (dof < (int)NDofs() && dof >= 0) {
269     int sector = floor(dof / 2);
270     if (sector == sectors) {
271       return case_sectors[sector - 1].dofs_r[dof % 2];
272     } else {
273       return case_sectors[sector].dofs_l[dof % 2];
274     }
275   } else {
276     std::cout << "Critical: DOF-index out of bounds in "
277                  "InhomogenousTransformationRectangular::get_dof!"
278               << std::endl;
279     return 0.0;
280   }
281 }
```

4.13.2.5 get_free_dof()

```
double InhomogenousTransformationRectangular::get_free_dof (
            int dof ) const  [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.

Implements SpaceTransformation.

Definition at line 283 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, InhomogenousTransformationCircular::NDofs(), and InhomogenousTransformationCircular::sectors.

```
283                                                                    {
284   int dof = in_dof + 2;
285   if (dof < (int)NDofs() - 2 && dof >= 0) {
286     int sector = floor(dof / 2);
```

```
287     if (sector == sectors) {
288       return case_sectors[sector - 1].dofs_r[dof % 2];
289     } else {
290       return case_sectors[sector].dofs_l[dof % 2];
291     }
292   } else {
293     std::cout << "Critical: DOF-index out of bounds in "
294                  "InhomogenousTransformationRectangular::get_free_dof!"
295               << std::endl;
296     return 0.0;
297   }
298 }
```

4.13.2.6 get_Q1()

```
double InhomogenousTransformationRectangular::get_Q1 (
            double z ) const  [virtual]
```

This member calculates the value of Q1 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q1 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 430 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, and SpaceTransformation::Z_to_Sector_and_↩ local_z().

```
430                                                                 {
431   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
432   return case_sectors[two.first].getQ1(two.second);
433 }
```

4.13.2.7 get_Q2()

```
double InhomogenousTransformationRectangular::get_Q2 (
            double z ) const  [virtual]
```

This member calculates the value of Q2 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q2 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 435 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, and SpaceTransformation::Z_to_Sector_and_↩ local_z().

```
435                                                                    {
436   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
437   return case_sectors[two.first].getQ2(two.second);
438 }
```

4.13.2.8 get_Q3()

```
double InhomogenousTransformationRectangular::get_Q3 (
            double z ) const  [virtual]
```

This member calculates the value of Q3 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q3 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implements SpaceTransformation.

Definition at line 440 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, and SpaceTransformation::Z_to_Sector_and_↩ local_z().

```
440                                                                    {
441   std::pair<int, double> two = Z_to_Sector_and_local_z(z_in);
442   return case_sectors[two.first].getQ3(two.second);
443 }
```

4.13.2.9 IsDofFree()

```
bool InhomogenousTransformationRectangular::IsDofFree (
            int index ) const  [virtual]
```

Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

"restrained" means that for example the DOF represents the radius at one of the connectors (input or output) and therefore we forbid the optimization scheme to vary this value.

Implements SpaceTransformation.

Definition at line 459 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::NDofs().

```
459                                                                    {
460   return index > 1 && index < (int)NDofs() - 1;
461 }
```

4.13.2.10 NDofs()

```
unsigned int InhomogenousTransformationRectangular::NDofs ( ) const  [virtual]
```

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

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

Implements SpaceTransformation.

Definition at line 467 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, InhomogenousTransformationCircular::sectors, and SpaceTransformation::Z_to_Sector_and_local_z().

```
467                                                                    {
468   return sectors * 2 + 2;
469 }
```

4.13.2.11 PML_in_X()

```
bool InhomogenousTransformationRectangular::PML_in_X (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 55 of file InhomogenousTransformationRectangular.cpp.

```
55                                                                     {
56   return p(0) < XMinus || p(0) > XPlus;
57 }
```

4.13.2.12 PML_in_Y()

```
bool InhomogenousTransformationRectangular::PML_in_Y (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 59 of file InhomogenousTransformationRectangular.cpp.

```
59                                                                    {
60   return p(1) < YMinus || p(1) > YPlus;
61 }
```

4.13.2.13 PML_in_Z()

```
bool InhomogenousTransformationRectangular::PML_in_Z (
            Point< 3 > & position ) const  [virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implements SpaceTransformation.

Definition at line 63 of file InhomogenousTransformationRectangular.cpp.

```
63                                                                    {
64   return p(2) < ZMinus || p(2) > ZPlus;
65 }
```

4.13.2.14 PML_X_Distance()

```
double InhomogenousTransformationRectangular::PML_X_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 73 of file InhomogenousTransformationRectangular.cpp.

```
74                                      {
75   if (p(0) > 0) {
76     return p(0) - XPlus;
77   } else {
78     return -p(0) + XMinus;
79   }
80 }
```

4.13.2.15 PML_Y_Distance()

```
double InhomogenousTransformationRectangular::PML_Y_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 82 of file InhomogenousTransformationRectangular.cpp.

```
83                                      {
84   if (p(1) > 0) {
85     return p(1) - YMinus;
86   } else {
87     return -p(1) + YPlus;
88   }
89 }
```

4.13.2.16 PML_Z_Distance()

```
double InhomogenousTransformationRectangular::PML_Z_Distance (
            Point< 3 > & position ) const  [virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implements SpaceTransformation.

Definition at line 91 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::PML_in_X(), InhomogenousTransformationCircular::PML_in_Y(), InhomogenousTransformationCircular::PML_in_Z(), InhomogenousTransformationCircular::PML_X_Distance(), InhomogenousTransformationCircular::PML_Y_Distance(), InhomogenousTransformationCircular::PML_Z_Distance(), and InhomogenousTransformationCircular::Preconditioner_PML_Z_Distance().

```
92                      {
93   if (p(2) < 0) {
94     return -(p(2) + (GlobalParams.M_R_ZLength / 2.0));
95   } else {
96     return p(2) - (GlobalParams.M_R_ZLength / 2.0);
97   }
98 }
```

4.13.2.17 Preconditioner_PML_in_Z()

```
bool InhomogenousTransformationRectangular::Preconditioner_PML_in_Z (
            Point< 3 > & p,
            unsigned int block ) const
```

Similar to the PML_in_Z only this function is used to generate the artificial PML used in the Preconditioner.

These Layers are not only situated at the surface of the computational domain but also inside it at the interfaces of Sectors.

4.13.2.18 set_dof()

```
void InhomogenousTransformationRectangular::set_dof (
            int dof,
            double value )  [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.

Implements SpaceTransformation.

Definition at line 300 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, InhomogenousTransformationCircular::NDofs(), and InhomogenousTransformationCircular::sectors.

```
300                                                                          {
301   if (dof < (int)NDofs() && dof >= 0) {
302     int sector = floor(dof / 2);
303     if (sector == sectors) {
304       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
305     } else if (sector == 0) {
306       case_sectors[0].dofs_l[dof % 2] = in_val;
307     } else {
308       case_sectors[sector].dofs_l[dof % 2] = in_val;
309       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
310     }
311   } else {
312     std::cout << "Critical: DOF-index out of bounds in "
313                  "InhomogenousTransformationRectangular::set_dof!"
314               << std::endl;
315   }
316 }
```

4.13.2.19 set_free_dof()

```
void InhomogenousTransformationRectangular::set_free_dof (
             int dof,
             double value )  [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.

Implements SpaceTransformation.

Definition at line 318 of file InhomogenousTransformationRectangular.cpp.

References InhomogenousTransformationCircular::case_sectors, InhomogenousTransformationCircular::NDofs(), and InhomogenousTransformationCircular::sectors.

```
319                                                                          {
320   int dof = in_dof + 2;
```

```
321   if (dof < (int)NDofs() - 2 && dof >= 0) {
322     int sector = floor(dof / 2);
323     if (sector == sectors) {
324       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
325     } else if (sector == 0) {
326       case_sectors[0].dofs_l[dof % 2] = in_val;
327     } else {
328       case_sectors[sector].dofs_l[dof % 2] = in_val;
329       case_sectors[sector - 1].dofs_r[dof % 2] = in_val;
330     }
331   } else {
332     std::cout << "Critical: DOF-index out of bounds in "
333                  "InhomogenousTransformationRectangular::set_free_dof!"
334               << std::endl;
335   }
336 }
```

4.13.3 Member Data Documentation

4.13.3.1 case_sectors

`std::vector<Sector<2> > InhomogenousTransformationRectangular::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 139 of file InhomogenousTransformationRectangular.h.

4.13.3.2 epsilon_K

`const double InhomogenousTransformationRectangular::epsilon_K`

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value inside the core.

Definition at line 146 of file InhomogenousTransformationRectangular.h.

4.13.3.3 epsilon_M

`const double InhomogenousTransformationRectangular::epsilon_M`

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value outside the core.

Definition at line 152 of file InhomogenousTransformationRectangular.h.

4.13.3.4 sectors

```
const int InhomogenousTransformationRectangular::sectors
```

Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.

This member stores the number of Sectors the computational domain has been split into.

Definition at line 158 of file InhomogenousTransformationRectangular.h.

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

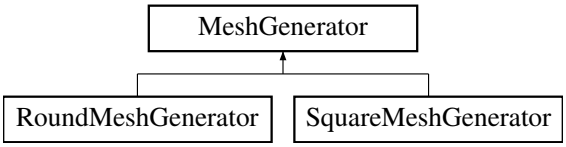
- Code/SpaceTransformations/InhomogenousTransformationRectangular.h
- Code/SpaceTransformations/InhomogenousTransformationRectangular.cpp

4.14 MeshGenerator Class Reference

This is an interface for all the mesh generators in the project describing its role and functionality.

```
#include <MeshGenerator.h>
```

Inheritance diagram for MeshGenerator:



Public Member Functions

- MeshGenerator (SpaceTransformation &in_ct)

  Since objects of this type mainly depend on the measures and the selected shape, the constructor only requires a space transformation (which determines, which points are to be considered inside the waveguide etc.)
- virtual void refine_global (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times)=0

  This function is intended to execute a global refinement of the mesh.
- virtual void refine_internal (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times)=0

  This function is intended to execute an internal refinement of the mesh.
- virtual void refine_proximity (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times, double factor)=0

  This function is intended to execute a refinement inside and near the waveguide boundary.
- virtual bool math_coordinate_in_waveguide (Point< 3 > position) const =0

  This function checks if the given coordinate is inside the waveguide or not.
- virtual bool phys_coordinate_in_waveguide (Point< 3 > position) const =0

  This function checks if the given coordinate is inside the waveguide or not.
- virtual void prepare_triangulation (parallel::distributed::Triangulation< 3 > *in_tria)=0

  This function takes a triangulation object and prepares it for the further computations.

Public Attributes

- parallel::distributed::Triangulation< 3 >::active_cell_iterator **cell**
- parallel::distributed::Triangulation< 3 >::active_cell_iterator **endc**
- SpaceTransformation * **ct**
- unsigned int **Layers**
- Point< 3 > **origin**
- Point< 3 > **p1**
- Point< 3 > **p2**
- std_cxx11::array< Tensor< 1, 3 >, 3 > **edges**
- std::vector< unsigned int > **subs**
- double **z_min**
- double **z_max**

4.14.1 Detailed Description

This is an interface for all the mesh generators in the project describing its role and functionality.

Since different shapes of waveguides (in the xz-plane) are interesting in application settings, we wish to introduce a mechanism to model this fact. Therefore all functionality related to the shape of the waveguide are encapsulated of specific objects which do all the heavy lifting. The problem si the fact that a rectangular geometry has more dofs then a square or circular one (radius versus width and height). This has implications on the space transformation and the optimization scheme and earlier versions were running the risk of getting too flawed by implementing loads of different case models over and over again. This structure leads to a higher readability of the code and reduces its error-proneness.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 32 of file MeshGenerator.h.

4.14.2 Constructor & Destructor Documentation

4.14.2.1 MeshGenerator()

```
MeshGenerator::MeshGenerator (
            SpaceTransformation & in_ct )
```

Since objects of this type mainly depend on the measures and the selected shape, the constructor only requires a space transformation (which determines, which points are to be considered inside the waveguide etc.)

Parameters

in↩ _ct	a Space Transformation which whill be stored internally for later use.

4.14.3 Member Function Documentation

4.14.3.1 math_coordinate_in_waveguide()

```
virtual bool MeshGenerator::math_coordinate_in_waveguide (
            Point< 3 > position ) const  [pure virtual]
```

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. position This value gives us the location to check for.

Implemented in SquareMeshGenerator, and RoundMeshGenerator.

Referenced by Waveguide::estimate_solution(), and RoundMeshGenerator::refine_internal().

4.14.3.2 phys_coordinate_in_waveguide()

```
virtual bool MeshGenerator::phys_coordinate_in_waveguide (
            Point< 3 > position ) const  [pure virtual]
```

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)). position This value gives us the location to check for.

Implemented in SquareMeshGenerator, and RoundMeshGenerator.

4.14.3.3 prepare_triangulation()

```
virtual void MeshGenerator::prepare_triangulation (
            parallel::distributed::Triangulation< 3 > * in_tria )  [pure virtual]
```

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.

Implemented in RoundMeshGenerator, and SquareMeshGenerator.

Referenced by Waveguide::estimate_solution().

4.14.3.4 refine_global()

```
virtual void MeshGenerator::refine_global (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times )  [pure virtual]
```

This function is intended to execute a global refinement of the mesh.

This means that every cell will be refined in every direction (effectively multiplying the number of DOFs by 8). This version is the most expensive refinement possible and should be used with caution.

Parameters

times	Number of refinement steps to be performed (gives us a multiplication of the number of degrees of freedom by 8^{times}.

Implemented in SquareMeshGenerator, and RoundMeshGenerator.

4.14.3.5 refine_internal()

```
virtual void MeshGenerator::refine_internal (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times )  [pure virtual]
```

This function is intended to execute an internal refinement of the mesh.

This means that every cell inside the waveguide will be refined in every direction. This method is rather cheap and only refines where the field is strong, however, the mesh outside the waveguide should not be too coarse to reduce numerical errors.

Parameters

times	Number of refinement steps to be performed.

Implemented in SquareMeshGenerator, and RoundMeshGenerator.

4.14.3.6 refine_proximity()

```
virtual void MeshGenerator::refine_proximity (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times,
            double factor )  [pure virtual]
```

This function is intended to execute a refinement inside and near the waveguide boundary.

Parameters

times	Number of refinement steps to be performed.

Implemented in SquareMeshGenerator, and RoundMeshGenerator.

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

- Code/MeshGenerators/MeshGenerator.h
- Code/MeshGenerators/MeshGenerator.cpp

4.15 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, dealii::Point< 3, double >, int)
- double **get_output_component** (int, dealii::Point< 3, double >, int)
- void **load** ()

4.15.1 Detailed Description

Definition at line 13 of file ModeManager.h.

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

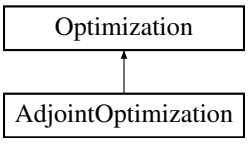
- Code/Helpers/ModeManager.h
- Code/Helpers/ModeManager.cpp

4.16 Optimization Class Reference

This class is an abstract interface to describe the general workings of an optimization scheme.

`#include <Optimization.h>`

Inheritance diagram for Optimization:



Public Member Functions

- virtual void run ()=0

  This function is the core implementation of an optimization algorithm.

Public Attributes

- const int **type** = -1
- ConditionalOStream **pout**
- const int **dofs** = 1
- const int **freedofs** = 0
- SpaceTransformation * **st**
- MeshGenerator * **mg**

4.16.1 Detailed Description

This class is an abstract interface to describe the general workings of an optimization scheme.

It is used to compute optimization steps and controls the DOFs of the ShapeTransformation.

In general there are two kinds of Optimization Schemes derived from this class. On the one hand there are finite-difference kind schemes which are based on the idea of varying the value of one shape parameter slightly, resolving the problem (which is now slightl varied compaired to the original one) and hence computing the entry of the shape gradient. Repeating this pattern for any un-restrained dof we can compute the complete gradient. The other version is based on an adjoint model where we solve the forward problem and its dual and can compute the shape gradient for all DOFs from these two results.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 35 of file Optimization.h.

4.16.2 Member Function Documentation

4.16.2.1 run()

```
virtual void Optimization::run ( )  [pure virtual]
```

This function is the core implementation of an optimization algorithm.

Currently it is very fundamental in its technical prowess which can be improved upon in later versions. Essentially, it calculates the signal quality for a configurations and for small steps in every one of the dofs. After that, the optimization-step is estimated based on difference-quotients. Following this step, a large step is computed based upon the approximation of the gradient of the signal-quality functional and the iteration starts anew. If a decrease in quality is detected, the optimization-step is undone and the step-width is reduced. This function controls both the Waveguide- and the Waveguide-structure object.

Implemented in AdjointOptimization.

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

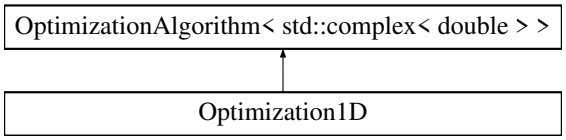
- Code/OptimizationStrategies/Optimization.h
- Code/OptimizationStrategies/Optimization.cpp

4.17 Optimization1D Class Reference

This class implements the computation of an optimization step by doing 1D optimization based on an adjoint scheme.

`#include <Optimization1D.h>`

Inheritance diagram for Optimization1D:



Public Member Functions

- bool perform_small_step_next (int small_steps_before)

  The optimization is mainly split into two kinds of steps: Full and small steps.
- double get_small_step_step_width (int small_steps_before)

  For the optimization scheme to know, which step size is appropriate, this function was included.
- bool perform_big_step_next (int small_steps_before)

  This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.
- std::vector< double > get_big_step_configuration ()

  This function computes the states that should be computed next.

Additional Inherited Members

4.17.1 Detailed Description

This class implements the computation of an optimization step by doing 1D optimization based on an adjoint scheme.

Objects of the Type OptimizationAlgorithm are used by the class OptimizationStrategy to compute the next viable configuration based on former results. Its is encapsulated in it's own class to offer comparison and easy changing between differenct schemes.

Author

Pascal Kraft

Date

9.1.2017

Definition at line 16 of file Optimization1D.h.

4.17.2 Member Function Documentation

4.17.2.1 get_big_step_configuration()

```
std::vector< double > Optimization1D::get_big_step_configuration ( )  [virtual]
```

This function computes the states that should be computed next.

If the next step will be a small step the update can be done by simply updating all dofs with a step width (or only one depending on the pattern) so this function is only used when a big step will be computed next and therefore all dofs could change differently.

Returns

This is a vector of degrees of freedom which can be used by the Optimization Strategy to update the Space Transformation.

Implements OptimizationAlgorithm< std::complex< double > >.

Definition at line 53 of file Optimization1D.cpp.

```
53                                                                  {
54   std::vector<double> ret;
55   if (residuals.size() == 0 && states.size() == 0) {
56     return ret;
57   }
58   int small_step_count = states.size();
59   int big_step_count = residuals.size();
60
61   if (big_step_count == 0 ||
62       (small_step_count != STEPS_PER_DOFS * big_step_count)) {
63     std::cout << "Warning in Optimization1D::get_big_step_configuration()"
64               << std::endl;
65   } else {
66     std::complex<double> residual = residuals[big_step_count - 1];
67     double state_red = std::abs(residual);
68     unsigned int ndofs = states[small_step_count - 1].size();
69     ret.resize(ndofs);
70     for (unsigned int i = 0; i < ndofs; i++) {
71       double max = 0;
72       int index = -1;
73       for (unsigned int j = 0; j < STEPS_PER_DOFS; j++) {
74         double magn = std::abs(
75             residual + states[small_step_count - STEPS_PER_DOFS + j][i]);
76         if (magn > max && magn > state_red) {
77           max = magn;
78           index = j;
79         }
80       }
81       if (index != -1) {
82         ret[i] = steps_widths[index];
83       } else {
84         ret[i] = 0.0;
85       }
86     }
87   }
88   return ret;
89 }
```

4.17.2.2 get_small_step_step_width()

```
double Optimization1D::get_small_step_step_width (
            int small_steps_before )  [virtual]
```

For the optimization scheme to know, which step size is appropriate, this function was included.

Parameters

small_steps_before	similar to perform_small_step_next this is the number of small steps before the current one.

Returns

double this is how much the values of the degrees of freedom should be adapted.

Implements OptimizationAlgorithm< std::complex< double > >.

Definition at line 35 of file Optimization1D.cpp.

```
35                                                                          {
36   if (small_steps_before < STEPS_PER_DOFS && small_steps_before >= 0) {
37     return steps_widths[small_steps_before];
38   } else {
39     std::cout << "Warning in Optimization1D::get_small_step_step_width(int)"
40               << std::endl;
41     return -1.0;
42   }
43 }
```

4.17.2.3 perform_big_step_next()

```
bool Optimization1D::perform_big_step_next (
              int small_steps_before )  [virtual]
```

This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.

Parameters

small_steps_before	number of small steps performed before this call.

Returns

true, if the next computation should be a big step - otherwise false.

Implements OptimizationAlgorithm< std::complex< double > >.

Definition at line 45 of file Optimization1D.cpp.

```
45                                                                  {
46   if (residuals.size() == 0 && states.size() == 0) {
47     return true;
48   }
49
50   return (small_steps_before >= STEPS_PER_DOFS);
51 }
```

4.17.2.4 perform_small_step_next()

```
bool Optimization1D::perform_small_step_next (
            int small_steps_before )  [virtual]
```

The optimization is mainly split into two kinds of steps: Full and small steps.

For FD based schemes, a small step is a computation of finite differences for all degrees of freedom which entails a lot of computation. Small here refers to the norm of the step width - not necessarily to the amount of computation required. In general this function is supposed to gather information about the target functional around the current state.

Parameters

small_steps_before	this number tells the scheme how many small steps were performed before the current request.

Returns

this is true, if the Optimization Scheme requires more small steps before a big step can be performed.

Implements OptimizationAlgorithm< std::complex< double > >.

Definition at line 23 of file Optimization1D.cpp.

```
23                                                                    {
24   if (residuals.size() == 0 && states.size() == 0) {
25     return false;
26   }
27
28   if (small_steps_before < STEPS_PER_DOFS) {
29     return true;
30   } else {
31     return false;
32   }
33 }
```

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

- Code/OptimizationAlgorithm/Optimization1D.h
- Code/OptimizationAlgorithm/Optimization1D.cpp

4.18 OptimizationAlgorithm< datatype > Class Template Reference

This class is an interface for Optimization algorithms such as CG or steepest descent.

```
#include <OptimizationAlgorithm.h>
```

Public Member Functions

- virtual void pass_result_small_step (std::vector< datatype > vec)

  A function to pass a vector of values, computed by performing a single or multiple steps with short step-width.
- virtual void pass_result_big_step (datatype input)

  Similar to pass_result_small_step but for the results of big steps.
- virtual bool perform_small_step_next (int small_steps_before)=0

  The optimization is mainly split into two kinds of steps: Full and small steps.
- virtual double get_small_step_step_width (int small_steps_before)=0

  For the optimization scheme to know, which step size is appropriate, this function was included.
- virtual bool perform_big_step_next (int small_steps_before)=0

  This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.
- virtual std::vector< double > get_big_step_configuration ()=0

  This function computes the states that should be computed next.
- void **WriteStepsOut** (std::ofstream &)

Public Attributes

- std::vector< std::vector< datatype > > **states**
- std::vector< datatype > **residuals**

4.18.1 Detailed Description

template<typename datatype>
class OptimizationAlgorithm< datatype >

This class is an interface for Optimization algorithms such as CG or steepest descent.

The derived classes take residuals and gradients, store them in a history and compute the next configuration based on the data. This functionality is encapsulated like this to enable easy exchange and comparison of convergence rates. Later on the interface will be extended to make use of output generators during runtime or at the end of the program to directly generate convergence plots. This class will also be extended to allow for restrained optimization which will become necessary at some point.

Author

Pascal Kraft

Date

29.11.2016

Definition at line 22 of file OptimizationAlgorithm.h.

4.18.2 Member Function Documentation

4.18.2.1 get_big_step_configuration()

```
template<typename datatype>
virtual std::vector<double> OptimizationAlgorithm< datatype >::get_big_step_configuration ( )
[pure virtual]
```

This function computes the states that should be computed next.

If the next step will be a small step the update can be done by simply updating all dofs with a step width (or only one depending on the pattern) so this function is only used when a big step will be computed next and therefore all dofs could change differently.

Returns

This is a vector of degrees of freedom which can be used by the Optimization Strategy to update the Space Transformation.

Implemented in OptimizationCG, OptimizationSteepestDescent, and Optimization1D.

4.18.2.2 get_small_step_step_width()

```
template<typename datatype>
virtual double OptimizationAlgorithm< datatype >::get_small_step_step_width (
            int small_steps_before )  [pure virtual]
```

For the optimization scheme to know, which step size is appropriate, this function was included.

Parameters

small_steps_before	similar to perform_small_step_next this is the number of small steps before the current one.

Returns

double this is how much the values of the degrees of freedom should be adapted.

Implemented in OptimizationCG, OptimizationSteepestDescent, and Optimization1D.

4.18.2.3 pass_result_big_step()

```
template<typename datatype>
void OptimizationAlgorithm< datatype >::pass_result_big_step (
            datatype input )  [virtual]
```

Similar to pass_result_small_step but for the results of big steps.

Since for a big step we always only perform the solution of one forward problem we also only get one change of the target functional. Therefore in this case we only pass a value, not a vector of the like.

Parameters

input	This is the value describing how much the target functional has changed upon performing the step last computed by this optimization algorithm.

Reimplemented in OptimizationCG.

Definition at line 44 of file OptimizationAlgorithm.cpp.

```
44                                                                                {
45   deallog.push("Big Step Result passed");
46   deallog << "Result:" << in_change << std::endl;
47   deallog.pop();
48   residuals.push_back(in_change);
49 }
```

4.18.2.4 pass_result_small_step()

```
template<typename datatype>
void OptimizationAlgorithm< datatype >::pass_result_small_step (
            std::vector< datatype > vec )  [virtual]
```

A function to pass a vector of values, computed by performing a single or multiple steps with short step-width.

"small" in the name references the fact, that the step width is small. In gerneral this is done whenever an accurat approximation of a gradient is saught based on linearization. This computation (especially in finite difference based approaches) can be much more costly than a big step. In a big step, one forward problem has to be solved. A small step computation based on finite differences requires $NDofs$ forward problems to be solved. This function has the purpose of passing the result of such computations to the optimization algorithm which will store it and use it to compute optimization steps in the future.

Parameters

vec	This parameter is a vector of changes of the target functional based on a change in the individual component. The components belonging to restrained degrees of freedom are set to zero.

Reimplemented in OptimizationCG.

Definition at line 13 of file OptimizationAlgorithm.cpp.

```
14                                       {
15   deallog.push("Small Step Result");
16   bool complex = std::is_same<datatype, std::complex<double>>::value;
17   if (complex) {
18     std::vector<double> values;
19     values.resize(2 * in_step_result.size());
20     for (unsigned int i = 0; i < in_step_result.size(); i++) {
21       values[2 * i] = ((std::complex<double>)in_step_result[i]).real();
22       values[2 * i + 1] = ((std::complex<double>)in_step_result[i]).imag();
23     }
24     dealii::Utilities::MPI::sum(values, MPI_COMM_WORLD, values);
25     for (unsigned int i = 0; i < in_step_result.size(); i++) {
26       ((std::complex<double>)in_step_result[i]).real(values[2 * i]);
27       ((std::complex<double>)in_step_result[i]).imag(values[2 * i + 1]);
28     }
29   } else {
```

```
30   }
31
32   for (unsigned int i = 0; i < in_step_result.size(); i++) {
33     deallog << in_step_result[i];
34     if (i != in_step_result.size() - 1) {
35       deallog << " , ";
36     }
37   }
38   deallog << std::endl;
39   deallog.pop();
40   states.push_back(in_step_result);
41 }
```

4.18.2.5 perform_big_step_next()

```
template<typename datatype>
virtual bool OptimizationAlgorithm< datatype >::perform_big_step_next (
              int small_steps_before )  [pure virtual]
```

This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.

Parameters

small_steps_before	number of small steps performed before this call.

Returns

true, if the next computation should be a big step - otherwise false.

Implemented in OptimizationCG, OptimizationSteepestDescent, and Optimization1D.

4.18.2.6 perform_small_step_next()

```
template<typename datatype>
virtual bool OptimizationAlgorithm< datatype >::perform_small_step_next (
              int small_steps_before )  [pure virtual]
```

The optimization is mainly split into two kinds of steps: Full and small steps.

For FD based schemes, a small step is a computation of finite differences for all degrees of freedom which entails a lot of computation. Small here refers to the norm of the step width - not necessarily to the amount of computation required. In general this function is supposed to gather information about the target functional around the current state.

Parameters

small_steps_before	this number tells the scheme how many small steps were performed before the current request.

Returns

this is true, if the Optimization Scheme requires more small steps before a big step can be performed.

Implemented in OptimizationCG, OptimizationSteepestDescent, and Optimization1D.

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

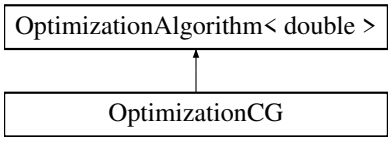
- Code/OptimizationAlgorithm/OptimizationAlgorithm.h
- Code/OptimizationAlgorithm/OptimizationAlgorithm.cpp

4.19 OptimizationCG Class Reference

This class implements the computation of an optimization step via a CG-method.

`#include <OptimizationCG.h>`

Inheritance diagram for OptimizationCG:



Public Member Functions

- virtual void pass_result_small_step (std::vector< double >)
  A function to pass a vector of values, computed by performing a single or multiple steps with short step-width.
- virtual void pass_result_big_step (double)
  Similar to pass_result_small_step but for the results of big steps.
- virtual bool perform_small_step_next (int small_steps_before)
  The optimization is mainly split into two kinds of steps: Full and small steps.
- virtual double get_small_step_step_width (int small_steps_before)
  For the optimization scheme to know, which step size is appropriate, this function was included.
- virtual bool perform_big_step_next (int small_steps_before)
  This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.
- virtual std::vector< double > get_big_step_configuration ()
  This function computes the states that should be computed next.

Additional Inherited Members

4.19.1 Detailed Description

This class implements the computation of an optimization step via a CG-method.

Objects of the Type OptimizationAlgorithm are used by the class OptimizationStrategy to compute the next viable configuration based on former results. Its is encapsulated in it's own class to offer comparison and easy changing between differenct schemes.

Author

Pascal Kraft

Date

29.11.2016

Definition at line 17 of file OptimizationCG.h.

4.19.2 Member Function Documentation

4.19.2.1 get_big_step_configuration()

```
std::vector< double > OptimizationCG::get_big_step_configuration ( )  [virtual]
```

This function computes the states that should be computed next.

If the next step will be a small step the update can be done by simply updating all dofs with a step width (or only one depending on the pattern) so this function is only used when a big step will be computed next and therefore all dofs could change differently.

Returns

This is a vector of degrees of freedom which can be used by the Optimization Strategy to update the Space Transformation.

Implements OptimizationAlgorithm< double >.

Definition at line 39 of file OptimizationCG.cpp.

```
39                                                          {
40   std::vector<double> ret;
41   // TODO: implement this function as core functionality of CG-based stepping
42   // scheme.
43   return ret;
44 }
```

4.19.2.2 get_small_step_step_width()

```
double OptimizationCG::get_small_step_step_width (
            int small_steps_before )  [virtual]
```

For the optimization scheme to know, which step size is appropriate, this function was included.

Parameters

small_steps_before	similar to perform_small_step_next this is the number of small steps before the current one.

Returns

double this is how much the values of the degrees of freedom should be adapted.

Implements OptimizationAlgorithm< double >.

Definition at line 27 of file OptimizationCG.cpp.

```
27                                                  {
28   // TODO: implement this function as core functionality of CG-based stepping
29   // scheme.
30   return 0.0;
31 }
```

4.19.2.3 pass_result_big_step()

```
void OptimizationCG::pass_result_big_step (
              double input )  [virtual]
```

Similar to pass_result_small_step but for the results of big steps.

Since for a big step we always only perform the solution of one forward problem we also only get one change of the target functional. Therefore in this case we only pass a value, not a vector of the like.

Parameters

input	This is the value describing how much the target functional has changed upon performing the step last computed by this optimization algorithm.

Reimplemented from OptimizationAlgorithm< double >.

Definition at line 15 of file OptimizationCG.cpp.

```
15                                              {
16   // TODO: implement this function as core functionality of CG-based stepping
17   // scheme.
18   return;
19 }
```

4.19.2.4 pass_result_small_step()

```
void OptimizationCG::pass_result_small_step (
              std::vector< double > vec )  [virtual]
```

A function to pass a vector of values, computed by performing a single or multiple steps with short step-width.

"small" in the name references the fact, that the step width is small. In gerneral this is done whenever an accurat approximation of a gradient is saught based on linearization. This computation (especially in finite difference based approaches) can be much more costly than a big step. In a big step, one forward problem has to be solved. A small step computation based on finite differences requires $NDofs$ forward problems to be solved. This function has the purpose of passing the result of such computations to the optimization algorithm which will store it and use it to compute optimization steps in the future.

Parameters

vec	This parameter is a vector of changes of the target functional based on a change in the individual component. The components belonging to restrained degrees of freedom are set to zero.

Reimplemented from OptimizationAlgorithm< double >.

Definition at line 10 of file OptimizationCG.cpp.

```
10                                                          {
11   // TODO: This implementation is still missing - uncertain and unimportant.
12   return;
13 }
```

4.19.2.5 perform_big_step_next()

```
bool OptimizationCG::perform_big_step_next (
            int small_steps_before )  [virtual]
```

This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.

Parameters

small_steps_before	number of small steps performed before this call.

Returns

true, if the next computation should be a big step - otherwise false.

Implements OptimizationAlgorithm< double >.

Definition at line 33 of file OptimizationCG.cpp.

```
33                                              {
34   // TODO: implement this function as core functionality of CG-based stepping
35   // scheme.
36   return false;
37 }
```

4.19.2.6 perform_small_step_next()

```
bool OptimizationCG::perform_small_step_next (
            int small_steps_before )  [virtual]
```

The optimization is mainly split into two kinds of steps: Full and small steps.

For FD based schemes, a small step is a computation of finite differences for all degrees of freedom which entails a lot of computation. Small here refers to the norm of the step width - not necessarily to the amount of computation required. In general this function is supposed to gather information about the target functional around the current state.

Parameters

small_steps_before	this number tells the scheme how many small steps were performed before the current request.

Returns

this is true, if the Optimization Scheme requires more small steps before a big step can be performed.

Implements OptimizationAlgorithm< double >.

Definition at line 21 of file OptimizationCG.cpp.

```
21                                                {
22   // TODO: implement this function as core functionality of CG-based stepping
23   // scheme.
24   return false;
25 }
```

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

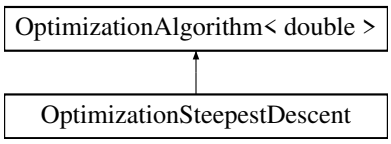
- Code/OptimizationAlgorithm/OptimizationCG.h
- Code/OptimizationAlgorithm/OptimizationCG.cpp

4.20 OptimizationSteepestDescent Class Reference

This class implements the computation of an optimization step via a Steepest-Descent-method.

`#include <OptimizationSteepestDescent.h>`

Inheritance diagram for OptimizationSteepestDescent:



Public Member Functions

- bool perform_small_step_next (int small_steps_before)

  The optimization is mainly split into two kinds of steps: Full and small steps.
- double get_small_step_step_width (int small_steps_before)

  For the optimization scheme to know, which step size is appropriate, this function was included.
- bool perform_big_step_next (int small_steps_before)

  This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.
- std::vector< double > get_big_step_configuration ()

  This function computes the states that should be computed next.

Additional Inherited Members

4.20.1 Detailed Description

This class implements the computation of an optimization step via a Steepest-Descent-method.

Objects of the Type OptimizationAlgorithm are used by the class OptimizationStrategy to compute the next viable configuration based on former results. Its is encapsulated in it's own class to offer comparison and easy changing between different schemes.

Author

Pascal Kraft

Date

29.11.2016

Definition at line 16 of file OptimizationSteepestDescent.h.

4.20.2 Member Function Documentation

4.20.2.1 get_big_step_configuration()

```
std::vector< double > OptimizationSteepestDescent::get_big_step_configuration ( )  [virtual]
```

This function computes the states that should be computed next.

If the next step will be a small step the update can be done by simply updating all dofs with a step width (or only one depending on the pattern) so this function is only used when a big step will be computed next and therefore all dofs could change differently.

Returns

This is a vector of degrees of freedom which can be used by the Optimization Strategy to update the Space Transformation.

Implements OptimizationAlgorithm< double >.

Definition at line 24 of file OptimizationSteepestDescent.cpp.

```
24                                                                          {
25   std::vector<double> ret;
26   if (residuals.size() == 0 && states.size() == 0) {
27     return ret;
28   }
29   int idx = states.size() - 1;
30   ret.resize(states[0].size());
31   for (unsigned int i = 0; i < states[0].size(); i++) {
32     ret[i] = -0.0001 * states[idx][i];
33   }
34   return ret;
35 }
```

4.20.2.2 get_small_step_step_width()

```
double OptimizationSteepestDescent::get_small_step_step_width (
            int small_steps_before )  [virtual]
```

For the optimization scheme to know, which step size is appropriate, this function was included.

Parameters

small_steps_before	similar to perform_small_step_next this is the number of small steps before the current one.

Returns

double this is how much the values of the degrees of freedom should be adapted.

Implements OptimizationAlgorithm< double >.

Definition at line 16 of file OptimizationSteepestDescent.cpp.

```
16                                                                  {
17   return GlobalParams.StepWidth;
18 }
```

4.20.2.3 perform_big_step_next()

```
bool OptimizationSteepestDescent::perform_big_step_next (
            int small_steps_before )  [virtual]
```

This functions returns true, if enough steps were performed to compute the next state to compute a full solution on.

Parameters

small_steps_before	number of small steps performed before this call.

Returns

true, if the next computation should be a big step - otherwise false.

Implements OptimizationAlgorithm< double >.

Definition at line 10 of file OptimizationSteepestDescent.cpp.

Referenced by perform_small_step_next().

```
10                                                                  {
11   int full_steps = residuals.size();
12   int small_steps = states.size();
13   return (full_steps <= small_steps);
14 }
```

4.20.2.4 perform_small_step_next()

```
bool OptimizationSteepestDescent::perform_small_step_next (
            int small_steps_before )  [virtual]
```

The optimization is mainly split into two kinds of steps: Full and small steps.

For FD based schemes, a small step is a computation of finite differences for all degrees of freedom which entails a lot of computation. Small here refers to the norm of the step width - not necessarily to the amount of computation required. In general this function is supposed to gather information about the target functional around the current state.

Parameters

small_steps_before	this number tells the scheme how many small steps were performed before the current request.

Returns

this is true, if the Optimization Scheme requires more small steps before a big step can be performed.

Implements OptimizationAlgorithm< double >.

Definition at line 20 of file OptimizationSteepestDescent.cpp.

References perform_big_step_next().

```
20                                                          {
21   return !perform_big_step_next(0);
22 }
```

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

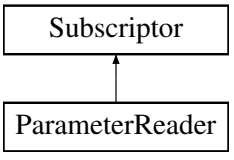
- Code/OptimizationAlgorithm/OptimizationSteepestDescent.h
- Code/OptimizationAlgorithm/OptimizationSteepestDescent.cpp

4.21 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 (ParameterHandler &prmhandler)

  Deal Offers the ParameterHandler object wich contains all of the parsing-functionality.
- void read_parameters (const std::string inputfile)

  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.

4.21.1 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.

4.21.2 Constructor & Destructor Documentation

4.21.2.1 ParameterReader()

```
ParameterReader::ParameterReader (
            ParameterHandler & prmhandler )
```

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 8 of file ParameterReader.cpp.

```
9     : prm(prmhandler) {}
```

4.21.3 Member Function Documentation

4.21.3.1 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 11 of file ParameterReader.cpp.

Referenced by read_parameters().

```
11                                      {
12   prm.enter_subsection("Output");
13   prm.enter_subsection("Optimization");
14   prm.enter_subsection("Gnuplot");
15   prm.declare_entry(
16       "Optimization History Live", "false", Patterns::Bool(),
17       "Currently not implemented. This will open an X-window at runtime and "
18       "show the current shape to allow for full tracking of the current "
19       "procedure while it runs (to be able to abort as early as possible.");
20   prm.declare_entry(
21       "Optimization History Shapes", "true", Patterns::Bool(),
22       "If this value is set to 'true', after every step a plot and a data file "
23       "of the current shape will be generated. The plot shows single (tubular) "
24       "or multiple (rectangular) crossections of the current waveguide shape.");
25   prm.declare_entry("Optimization History", "true", Patterns::Bool(),
26                     "If this value is set to 'true', a plot and data file will "
27                     "be generated logging the values of the signal quality "
28                     "after every step of the current optimization scheme.");
29   prm.leave_subsection();
30   prm.enter_subsection("VTK");
31   prm.enter_subsection("TransformationWeights");
32   prm.declare_entry(
33       "TransformationWeightsAll", "false", Patterns::Bool(),
34       "If this is enabled, a .vtk file is generated in every step logging the "
35       "norm of the transformation tensor as 3D data.");
36   prm.declare_entry(
37       "TransformationWeightsFirst", "false", Patterns::Bool(),
38       "If this is enabled, a .vtk file is generated in the first step logging "
39       "the norm of the transformation tensor as 3D data.");
40   prm.declare_entry(
41       "TransformationWeightsLast", "false", Patterns::Bool(),
42       "If this is enabled, a .vtk file is generated in the last step logging "
43       "the norm of the transformation tensor as 3D data.");
44   prm.leave_subsection();
45   prm.enter_subsection("Solution");
46   prm.declare_entry("SolutionAll", "false", Patterns::Bool(),
47                     "If this is enabled, a .vtk file is generated in every "
```

```
48                   "step logging the solution as 3D data.");
49   prm.declare_entry("SolutionFirst", "true", Patterns::Bool(),
50                   "If this is enabled, a .vtk file is generated in the first "
51                   "step logging the solution as 3D data.");
52   prm.declare_entry("SolutionLast", "true", Patterns::Bool(),
53                   "If this is enabled, a .vtk file is generated in the last "
54                   "step logging the solution as 3D data.");
55   prm.leave_subsection();
56   prm.leave_subsection();
57   prm.leave_subsection();
58   prm.enter_subsection("Convergence");
59   prm.enter_subsection("DataFiles");
60   prm.declare_entry("ConvergenceFirst", "false", Patterns::Bool(),
61                   "This causes the code to generate a datafile with the data "
62                   "of the system matrix solution process of the first step.");
63   prm.declare_entry("ConvergenceLast", "false", Patterns::Bool(),
64                   "This causes the code to generate a datafile with the data "
65                   "of the system matrix solution process of the last step.");
66   prm.declare_entry("ConvergenceAll", "false", Patterns::Bool(),
67                   "This causes the code to generate a datafile with the data "
68                   "of the system matrix solution process of all steps.");
69   prm.leave_subsection();
70   prm.enter_subsection("Plots");
71   prm.declare_entry("ConvergenceFirst", "false", Patterns::Bool(),
72                   "This causes the code to generate a plot of the data of "
73                   "the system matrix solution process of the first step.");
74   prm.declare_entry("ConvergenceLast", "false", Patterns::Bool(),
75                   "This causes the code to generate a plot of the data of "
76                   "the system matrix solution process of the last step.");
77   prm.declare_entry("ConvergenceAll", "false", Patterns::Bool(),
78                   "This causes the code to generate a plot of the data of "
79                   "the system matrix solution process of all steps.");
80   prm.leave_subsection();
81   prm.leave_subsection();
82   prm.enter_subsection("General");
83   prm.declare_entry("SummaryFile", "true", Patterns::Bool(),
84                   "This generates an output of the simulation (mathematical "
85                   "terms. Convergence rates, residuals, parameters etc.)");
86   prm.declare_entry("LogFile", "true", Patterns::Bool(),
87                   "This generates a general log (Code steps, warnings, "
88                   "errors, diagnostics)");
89   prm.leave_subsection();
90   prm.leave_subsection();
91   prm.enter_subsection("Measures");
92   prm.enter_subsection("PredefinedCases");
93   prm.declare_entry("ComputeCase", "false", Patterns::Bool(),
94                   "Do you want to compute a predefined case?");
95   prm.declare_entry("SelectCase", "0", Patterns::Integer(0, 72),
96                   "Which Case to compute (see numbers in hdf5 files).");
97   prm.leave_subsection();
98   prm.enter_subsection("Connectors");
99   prm.declare_entry("Shape", "Circle", Patterns::Selection("Circle|Rectangle"),
100                    "Describes the shape of the input connector.");
101   prm.declare_entry(
102       "Dimension1 In", "2.0", Patterns::Double(0),
103       "First dimension of the input connector. For a circular waveguide this "
104       "is the radius. For a rectangular waveguide this is the width.");
105   prm.declare_entry(
106       "Dimension2 In", "2.0", Patterns::Double(0),
107       "Second dimension of the input connector. For a circular waveguide this "
108       "has no meaning. For a rectangular waveguide this is the height.");
109   prm.declare_entry(
110       "Dimension1 Out", "2.0", Patterns::Double(0),
111       "First dimension of the output connector. For a circular waveguide this "
112       "is the radius. For a rectangular waveguide this is the width.");
113   prm.declare_entry(
114       "Dimension2 Out", "2.0", Patterns::Double(0),
115       "Second dimension of the output connector. For a circular waveguide this "
116       "has no meaning. For a rectangular waveguide this is the height.");
117   prm.leave_subsection();
118   prm.enter_subsection("Region");
119   prm.declare_entry(
120       "XLength", "10.0", Patterns::Double(0),
121       "Length of the system in x-Direction (Connectors lie in the XY-plane and "
122       "the offset lies in the y-direction. Measured in micrometres");
123   prm.declare_entry(
124       "YLength", "10.0", Patterns::Double(0),
125       "Length of the system in y-Direction (Connectors lie in the XY-plane and "
126       "the offset lies in the y-direction. Measured in micrometres");
127   prm.declare_entry(
128       "ZLength", "6.0", Patterns::Double(0),
129       "Length of the system in z-Direction (Connectors lie in the XY-plane and "
130       "the offset lies in the y-direction. Measured in micrometres");
131   prm.leave_subsection();
132   prm.enter_subsection("Waveguide");
133   prm.declare_entry(
134       "Delta", "1.0", Patterns::Double(0),
```

```
135       "Offset between the two connectors measured in micrometres.");
136   prm.declare_entry(
137       "epsilon in", "2.21", Patterns::Double(0),
138       "Material-Property of the optical fiber (optical thickness).");
139   prm.declare_entry(
140       "epsilon out", "2.2", Patterns::Double(0),
141       "Material-Property of environment of the fiber (optical thickness).");
142   prm.declare_entry("Lambda", "5.6328", Patterns::Double(0),
143                     "Vacuum-wavelength of the incoming wave.");
144   prm.declare_entry("Sectors", "2", Patterns::Integer(1),
145                     "Number of Sectors used for Modelling of the Waveguide.");
146   prm.leave_subsection();
147   prm.enter_subsection("Boundary Conditions");
148   prm.declare_entry("Type", "PML", Patterns::Selection("PML|HSIE"),
149                     "The way the output-connector is modeled. HSIE uses the "
150                     "Hardy-space infinite element for setting boundary "
151                     "conditions but isn't implemented yet.");
152   prm.declare_entry("ZMinus", "5", Patterns::Double(0),
153                     "Thickness of the additional range on the negative z axis "
154                     "for better wave coupling");
155   prm.declare_entry(
156       "ZPlus", "1", Patterns::Double(0),
157       "Thickness of the PML area on the side of the output connector.");
158   prm.declare_entry(
159       "XMinus", "1.0", Patterns::Double(0),
160       "Thickness of the PML on the negative X-axis. Measured in micrometers");
161   prm.declare_entry(
162       "XPlus", "1.0", Patterns::Double(0),
163       "Thickness of the PML on the positive X-axis. Measured in micrometers");
164   prm.declare_entry(
165       "YMinus", "1.0", Patterns::Double(0),
166       "Thickness of the PML on the negative Y-axis. Measured in micrometers");
167   prm.declare_entry(
168       "YPlus", "1.0", Patterns::Double(0),
169       "Thickness of the PML on the positive Y-axis. Measured in micrometers");
170   prm.declare_entry("KappaXMax", "10.0", Patterns::Double(0),
171                     "PML Tuning Parameter");
172   prm.declare_entry("KappaYMax", "10.0", Patterns::Double(0),
173                     "PML Tuning Parameter");
174   prm.declare_entry("KappaZMax", "10.0", Patterns::Double(0),
175                     "PML Tuning Parameter");
176   prm.declare_entry("SigmaXMax", "10.0", Patterns::Double(0),
177                     "PML Tuning Parameter");
178   prm.declare_entry("SigmaYMax", "10.0", Patterns::Double(0),
179                     "PML Tuning Parameter");
180   prm.declare_entry("SigmaZMax", "10.0", Patterns::Double(0),
181                     "PML Tuning Parameter");
182   prm.declare_entry(
183       "DampeningExponentM", "3", Patterns::Integer(3),
184       "Dampening Exponent M for the intensety of dampening in the PML region.");
185   prm.leave_subsection();
186   prm.leave_subsection();
187   prm.enter_subsection("Schema");
188   prm.declare_entry(
189       "Homogeneity", "false", Patterns::Bool(),
190       "If this is enabled, a space transformation is used which is equal the "
191       "identity on the PML-region for the dampening along the x and y axis.");
192   prm.declare_entry("Optimization Schema", "Adjoint",
193                     Patterns::Selection("Adjoint|FD"),
194                     "If this is set to adjoint, the shape gradient will be "
195                     "computed by means of an adjoint based method. If it is "
196                     "set to FD, finite differences are use.");
197   prm.declare_entry("Optimization Steps", "10", Patterns::Integer(1),
198                     "Number of Optimization steps to be performed.");
199   prm.declare_entry(
200       "Stepping Method", "Steepest",
201       Patterns::Selection("Steepest|CG|LineSearch"),
202       "Method of step computation. Steepest uses steepest descent. CG uses a "
203       "conjugate gradient method to compute the next step. Line Search only "
204       "works based on an adjoint optimization setting, where searches can be "
205       "performed cheaply.");
206   // prm.declare_entry("Step Width", "Adjoint",
207   // Patterns::Selection("Adjoint|Experimental"), "This parameter descibes the
208   // scheme used to compute the next step width. This can be adjoint (if an
209   // adjoint schema is used, which causees the computation of multiple shape
210   // hradient with differing step widths in the parameters. This would be too
211   // costly for FD and ist therefore not available in that mode. An experimental
212   // approach can be used which tries to use information from the gradient and a
213   // seperate step width control to compute the step. ");
214   prm.leave_subsection();
215   prm.enter_subsection("Solver");
216   prm.declare_entry(
217       "Solver", "GMRES", Patterns::Selection("GMRES|UMFPACK|MINRES"),
218       "Which Solver to use for the solution of the system matrix");
219   prm.declare_entry("GMRESSteps", "30", Patterns::Integer(1),
220                     "Steps until restart of Krylow subspace generation");
221   prm.declare_entry("Preconditioner", "Sweeping",
```

```
222                   Patterns::Selection(
223                       "Sweeping|FastSweeping|HSIESweeping|HSIEFastSweeping"),
224                   "Which preconditioner to use");
225   prm.declare_entry("PreconditionerDampening", "0.0", Patterns::Double(0),
226                   "Dampening for the preconditioner to acclelerate "
227                   "convergence. Typically 0 (off) or 1.0");
228   prm.declare_entry("Steps", "30", Patterns::Integer(1),
229                   "Number of Steps the Solver is supposed to do.");
230   prm.declare_entry("Precision", "1e-6", Patterns::Double(0),
231                   "Minimal error value, the solver is supposed to accept as "
232                   "correct solution.");
233   prm.leave_subsection();
234   prm.enter_subsection("Constants");
235   prm.declare_entry(
236       "AllOne", "true", Patterns::Bool(),
237       "If this is set to true, EpsilonZero and MuZero are set to 1.");
238   prm.declare_entry("EpsilonZero", "8.854e-18", Patterns::Double(0),
239                   "Physical constant Epsilon zero. The standard value is "
240                   "measured in micrometers.");
241   prm.declare_entry("MuZero", "1.257e-12", Patterns::Double(0),
242                   "Physical constant Mu zero. The standard value is measured "
243                   "in micrometers.");
244   prm.declare_entry("Pi", "3.14159265", Patterns::Double(0),
245                   "Mathematical constant Pi.");
246   prm.leave_subsection();
247   prm.enter_subsection("Refinement");
248   prm.declare_entry("Global", "1", Patterns::Integer(0),
249                   "Global refinement-steps.");
250   prm.declare_entry("SemiGlobal", "1", Patterns::Integer(0),
251                   "Semi-Global refinement-steps (close to the "
252                   "Waveguide-boundary and inside).");
253   prm.declare_entry("Internal", "1", Patterns::Integer(0),
254                   "Internal refinement-steps.");
255   prm.leave_subsection();
256 }
```

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

- Code/Helpers/ParameterReader.h
- Code/Helpers/ParameterReader.cpp

4.22 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 Attributes

- bool **O_O_G_HistoryLive**
- bool **O_O_G_HistoryShapes**
- bool **O_O_G_History**
- bool **O_O_V_T_TransformationWeightsAll**
- bool **O_O_V_T_TransformationWeightsFirst**
- bool **O_O_V_T_TransformationWeightsLast**
- bool **O_O_V_S_SolutionAll**
- bool **O_O_V_S_SolutionFirst**
- bool **O_O_V_S_SolutionLast**
- bool **O_C_D_ConvergenceAll**
- bool **O_C_D_ConvergenceFirst**
- bool **O_C_D_ConvergenceLast**
- bool **O_C_P_ConvergenceAll**

- bool **O_C_P_ConvergenceFirst**
- bool **O_C_P_ConvergenceLast**
- bool **O_G_Summary**
- bool **O_G_Log**
- ConnectorType **M_C_Shape**
- double **M_C_Dim1In**
- double **M_C_Dim2In**
- double **M_C_Dim1Out**
- double **M_C_Dim2Out**
- double **M_R_XLength**
- double **M_R_YLength**
- double **M_R_ZLength**
- double **M_W_Delta**
- double **M_W_epsilonin**
- double **M_W_epsilonout**
- int **M_W_Sectors**
- double **M_W_Lambda**
- BoundaryConditionType **M_BC_Type**
- double **M_BC_Zminus**
- double **M_BC_Zplus**
- double **M_BC_XMinus**
- double **M_BC_XPlus**
- double **M_BC_YMinus**
- double **M_BC_YPlus**
- double **M_BC_KappaXMax**
- double **M_BC_KappaZMax**
- double **M_BC_KappaYMax**
- double **M_BC_SigmaXMax**
- double **M_BC_SigmaZMax**
- double **M_BC_SigmaYMax**
- double **M_BC_DampeningExponent**
- bool **Sc_Homogeneity**
- OptimizationSchema **Sc_Schema**
- int **Sc_OptimizationSteps**
- SteppingMethod **Sc_SteppingMethod**
- SolverOptions **So_Solver**
- PreconditionerOptions **So_Preconditioner**
- double **So_PreconditionerDampening**
- int **So_ElementOrder**
- int **So_RestartSteps**
- int **So_TotalSteps**
- double **So_Precision**
- bool **C_AllOne**
- double **C_Mu**
- double **C_Epsilon**
- double **C_Pi**
- double **C_c**
- double **C_f0**
- double **C_k0**
- double **C_omega**
- int **R_Global**
- int **R_Local**
- int **R_Interior**
- double **LayerThickness**
- double **SectorThickness**

- unsigned int **MPI_Rank**
- MPI_Comm **MPIC_World**
- bool **PMLLayer**
- double **LayersPerSector**
- int **NumberProcesses**
- bool **Head** = false
- double **SystemLength**
- double **Maximum_Z**
- double **Minimum_Z**
- double **Phys_V**
- double **Phys_SpotRadius**
- double **StepWidth**
- bool **M_PC_Use**
- int **M_PC_Case**
- ShapeDescription **sd**

4.22.1 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 77 of file Parameters.h.

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

- Code/Helpers/Parameters.h

4.23 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

- std::complex< double > **Ex**
- std::complex< double > **Ey**
- std::complex< double > **Ez**

4.23.1 Detailed Description

Definition at line 6 of file PointVal.h.

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

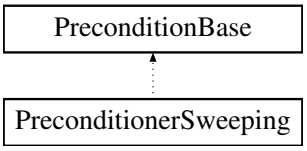
- Code/Helpers/PointVal.h
- Code/Helpers/PointVal.cpp

4.24 PreconditionerSweeping Class Reference

This class implements the DealII preconditioner interface and offers a sweeping preconditioning mechanism.

`#include <PreconditionerSweeping.h>`

Inheritance diagram for PreconditionerSweeping:



Public Member Functions

- PreconditionerSweeping (MPI_Comm in_mpi_comm, int in_own, int in_others, int in_above, int bandwidth, dealii::IndexSet locally_owned, dealii::IndexSet ∗in_fixed_dofs, int rank, bool fast)

  This constructor is the only one that should be used at this time.
- void Hinv (const dealii::Vector< double > &src, dealii::Vector< double > &dst) const

  For the application of the preconditioner we require the application of the inverse of H.
- void LowerProduct (const dealii::Vector< double > &src, dealii::Vector< double > &dst) const

  Cases in which we require multiplications with $A(E_{i+1}, E_i)$, are where this function is used.
- void UpperProduct (const dealii::Vector< double > &src, dealii::Vector< double > &dst) const

  Cases in which we require multiplications with $A(E_i, E_{i+1})$, are where this function is used.
- virtual void vmult (dealii::TrilinosWrappers::MPI::BlockVector &dst, const dealii::TrilinosWrappers::MPI::↩ BlockVector &src) const

  In order to be called by the iterative solver, this function has to be overloaded.
- virtual void vmult_fast (dealii::TrilinosWrappers::MPI::BlockVector &dst, const dealii::TrilinosWrappers::MPI↩ ::BlockVector &src) const
- virtual void **vmult_slow** (dealii::TrilinosWrappers::MPI::BlockVector &dst, const dealii::TrilinosWrappers::↩ MPI::BlockVector &src) const
- void **Prepare** (dealii::TrilinosWrappers::MPI::BlockVector &src)
- void **init** (dealii::SolverControl in_sc, dealii::TrilinosWrappers::SparseMatrix ∗, dealii::TrilinosWrappers::↩ SparseMatrix ∗)

Public Attributes

- dealii::TrilinosWrappers::SparseMatrix ∗ **matrix**
- dealii::SparseMatrix< double > ∗ **prec_matrix_upper**
- dealii::SparseMatrix< double > ∗ **prec_matrix_lower**

4.24.1 Detailed Description

This class implements the DealII preconditioner interface and offers a sweeping preconditioning mechanism.

Details can be found in the paper A sweeping preconditioner for time-harmonic Maxwell's equations with finite elements. The general idea is as follows: Let Ω be the computational domain (internally) truncated by an absorbing boundary condition. This domain can be split into layers along a direction (in our case z and triangulated. We therefore have a triangulation spread across multiple processes. We chose the splitting such, that the degrees of freedom are ordered process-wise. Let K be the number of Layers and $T_i \quad \imath \in \{1, \dots, K\}$ the parts of the triangulation. For a PML function

$$\sigma_i(\xi) = \begin{cases} \theta\left(\frac{-1+(i-1)l-\xi}{l} * \right)^2 & \xi \in [-1+(i-2)l, -1+(i-1)l] \\ 0, & \xi \in [-1+*(i-1)l, 1-l] \\ \Theta\left(\frac{\xi-1+l}{l}\right)^2, & \xi \in *[1-l, 1] \end{cases}$$

we now regard the problem

$$\nabla \times *\tilde{\mu}_{r,i}^{-1} \nabla \times \boldsymbol{E} - *\kappa^2 \tilde{\epsilon}_{r,i} \boldsymbol{E} = 0 \quad \text{in } *\operatorname{int}(\mathrm{T}_{i-1} \cup \mathrm{T}_i$$

where

Described in words: We put a PML into the neighboring block of the block we want to precondition and setup the system matrix for this smaller problem. This matrix we can then invert and name the Operator H_i^{-1}. We then define the operator

$$S(\boldsymbol{v}) = P_{0,n_i} H_i^{-1}(\boldsymbol{v}*, \boldsymbol{0}))$$

where P_{0,n_i} describes the extraction of the first n_i components. For the one block which has no neighbor the inverse of the block of the system matrix can be used. The inversion does not have to be performed numerically - a decomposition (performed by UMFPACK or MUMPS) is sufficient.

Date

28.11.2016

Author

Pascal Kraft

Definition at line 59 of file PreconditionerSweeping.h.

4.24.2 Constructor & Destructor Documentation

4.24.2.1 PreconditionerSweeping()

```
PreconditionerSweeping::PreconditionerSweeping (
            MPI_Comm in_mpi_comm,
            int in_own,
            int in_others,
            int in_above,
            int bandwidth,
            dealii::IndexSet locally_owned,
            dealii::IndexSet * in_fixed_dofs,
            int rank,
            bool fast )
```

This constructor is the only one that should be used at this time.

Parameters

in_own	This is the number of degrees of freedom that the current process has to deal with (owned).
in_others	This is the number of degrees of freedom that the process below has. Every process has to deal with one other process. The other neighbor only contacts it for a multiplication with its own matrix block - in this case, no objects of unknown size are concerned. However: for the one process that does require more contact needs a vector tp be initialized. This vectors size is this int.
bandwidth	The number of dofs per line on average is required for the construction of matrices.
locally_owned	The degrees of freedom associated with the current process. Required for vector and matrix construction.
dampening_parameter	If set to zero, no dampening is used. Otherwise, dampening according to (3.10) in the sweeping preconditioner paper is used. Thi solves the equation $$\nabla \times \tilde{\mu}^{-1} \nabla \times \boldsymbol{E} - (\kappa + i\alpha)^2 \tilde{\epsilon} \boldsymbol{E} = 0$$

Definition at line 27 of file PreconditionerSweeping.cpp.

```
32                                                                                {
33   locally_owned_dofs = in_locally_owned_dofs;
34   own = in_own;
35   others = in_others;
36   IndexSet elements(own + others);
37   elements.add_range(0, own + others);
38   indices = new int[in_locally_owned_dofs.n_elements()];
39   sweepable = in_locally_owned_dofs.n_elements();
40   for (unsigned int i = 0; i < sweepable; i++) {
41     indices[i] = in_locally_owned_dofs.nth_index_in_set(i);
42   }
43   fixed_dofs = in_fixed_dofs;
44   rank = in_rank;
45   bandwidth = in_bandwidth;
46   mpi_comm = in_mpi_comm;
47   above = in_above;
48   prec_matrix_lower = 0;
49   prec_matrix_upper = 0;
50   matrix = 0;
51   fast = in_fast;
52 }
```

4.24.3 Member Function Documentation

4.24.3.1 Hinv()

```
void PreconditionerSweeping::Hinv (
            const dealii::Vector< double > & src,
            dealii::Vector< double > & dst ) const
```

For the application of the preconditioner we require the application of the inverse of H.

This is implemented in this function. (The mathematical usage is included in lines 2, 6 and 13 and indirectly in every use of the Operator S.

Parameters

src	This is the vector to be multiplied by H_i^{-1}.
dst	This is the vector to store the result in.

Definition at line 213 of file PreconditionerSweeping.cpp.

```
214                                                                    {
215   dealii::Vector<double> inputb(own + others);
216   for (int i = 0; i < own; i++) {
217     inputb[i] = src(i);
218   }
219
220   solver->solve(inputb);
221
222   for (int i = 0; i < own; i++) {
223     dst[i] = inputb[i];
224   }
225 }
```

4.24.3.2 LowerProduct()

```
void PreconditionerSweeping::LowerProduct (
              const dealii::Vector< double > & src,
              dealii::Vector< double > & dst ) const
```

Cases in which we require multiplications with $A(E_{i+1}, E_i)$, are where this function is used.

See algorithm lines 2 and 4.

Parameters

src	This is the vector to be multiplied by $A(E_{i+1}, E_i)$.
dst	This is the vector to store the result in.

Definition at line 312 of file PreconditionerSweeping.cpp.

```
313                                                                    {
314   if ((int)rank == 0) {
315     std::cout << "ERROR!" << std::endl;
316   }
317
318   prec_matrix_lower->vmult(dst, src);
319 }
```

4.24.3.3 UpperProduct()

```
void PreconditionerSweeping::UpperProduct (
              const dealii::Vector< double > & src,
              dealii::Vector< double > & dst ) const
```

Cases in which we require multiplications with $A(E_i, E_{i+1})$, are where this function is used.

See algorithm lines 11 and 13.

Parameters

src	This is the vector to be multiplied by $A(E_i, E_{i+1})$.
dst	This is the vector to store the result in.

Definition at line 303 of file PreconditionerSweeping.cpp.

```
304                                                                          {
305   if ((int)rank + 1 == GlobalParams.NumberProcesses) {
306     std::cout << "ERROR!" << std::endl;
307   }
308
309   prec_matrix_upper->vmult(dst, src);
310 }
```

4.24.3.4 vmult()

```
void PreconditionerSweeping::vmult (
              dealii::TrilinosWrappers::MPI::BlockVector & dst,
              const dealii::TrilinosWrappers::MPI::BlockVector & src ) const  [virtual]
```

In order to be called by the iterative solver, this function has to be overloaded.

It gets called from GMRES and is the core function which contains the implementation. For a description of the interface, see the implementation in the base class.

Parameters

dst	The vector to store the result in.
src	The vector to be multiplied by the approximate inverse.

Definition at line 62 of file PreconditionerSweeping.cpp.

```
64                                                  {
65   if (fast) {
66     this->vmult_fast(dst, src);
67   } else {
68     this->vmult_slow(dst, src);
69   }
70 }
```

4.24.3.5 vmult_fast()

```
void PreconditionerSweeping::vmult_fast (
              dealii::TrilinosWrappers::MPI::BlockVector & dst,
              const dealii::TrilinosWrappers::MPI::BlockVector & src ) const  [virtual]
```

fast version starting here. if (rank == 0) { MPI_Send(&input[0], own, MPI_DOUBLE, rank + 1, 0, mpi_comm); } else { MPI_Recv(& recv_buffer_above[0], above, MPI_DOUBLE, rank-1, 0, mpi_comm, MPI_STATUS_IGNORE); if(rank < GlobalParams.NumberProcesses-1) MPI_Send(&input[0], own, MPI_DOUBLE, rank + 1, 0, mpi_comm); }

if(rank > 0) { LowerProduct(recv_buffer_above, temp_own); if(rank == GlobalParams.NumberProcesses-1) { solver->solve(temp_own); input -= temp_own; } else { Hinv(temp_own, temp_own_2); input -= temp_own_2; } }

for(int i = 0; i < own; i++){ if(! fixed_dofs->is_element(indices[i])){ dst[indices[i]] = input[i]; } }

Definition at line 72 of file PreconditionerSweeping.cpp.

```
74                                                  {
75   dealii::Vector<double> recv_buffer_above(above);
76   dealii::Vector<double> recv_buffer_below(others);
77   dealii::Vector<double> temp_own(own);
78   dealii::Vector<double> temp_own_2(own);
79   dealii::Vector<double> input(own);
80   for (unsigned int i = 0; i < sweepable; i++) {
81     input[i] = src[indices[i]];
82   }
83
84   if (rank == GlobalParams.NumberProcesses - 1) {
85     solver->solve(input);
86   } else {
87     if (rank > 0) {
88       Hinv(input, temp_own);
89     }
90   }
91   if (rank == GlobalParams.NumberProcesses - 1) {
92     MPI_Send(&input[0], own, MPI_DOUBLE, rank - 1, 0, mpi_comm);
93   } else {
94     MPI_Recv(&recv_buffer_below[0], others, MPI_DOUBLE, rank + 1, 0, mpi_comm,
95              MPI_STATUS_IGNORE);
96     if (rank > 0)
97       MPI_Send(&temp_own[0], own, MPI_DOUBLE, rank - 1, 0, mpi_comm);
98   }
99
100   if (rank < GlobalParams.NumberProcesses - 1) {
101     UpperProduct(recv_buffer_below, temp_own);
102     input -= temp_own;
103   }
104
105   if (rank < GlobalParams.NumberProcesses - 1) {
106     for (int i = 0; i < own; i++) {
107       temp_own[i] = input[i];
108     }
109     Hinv(temp_own, input);
110   }
111
112   /**
113    * fast version starting here.
114     if ( rank == 0) {
115       MPI_Send(&input[0], own, MPI_DOUBLE, rank + 1, 0, mpi_comm);
116     } else {
117       MPI_Recv(& recv_buffer_above[0], above, MPI_DOUBLE, rank-1, 0, mpi_comm,
118    MPI_STATUS_IGNORE); if(rank < GlobalParams.NumberProcesses-1)
119    MPI_Send(&input[0], own, MPI_DOUBLE, rank + 1, 0, mpi_comm);
120     }
121
122     if(rank > 0) {
123       LowerProduct(recv_buffer_above, temp_own);
124       if(rank == GlobalParams.NumberProcesses-1) {
125         solver->solve(temp_own);
126         input -= temp_own;
127       } else {
128         Hinv(temp_own, temp_own_2);
129         input -= temp_own_2;
130       }
131     }
132
133     for(int i = 0; i < own; i++ ){
134       if(! fixed_dofs->is_element(indices[i])){
135         dst[indices[i]] = input[i];
136       }
137     }
138     **/
139   if (rank == 0) {
140     MPI_Send(&input[0], own, MPI_DOUBLE, rank + 1, 0, mpi_comm);
141   } else {
142     MPI_Recv(&recv_buffer_above[0], above, MPI_DOUBLE, rank - 1, 0, mpi_comm,
143              MPI_STATUS_IGNORE);
144     LowerProduct(recv_buffer_above, temp_own);
145     Hinv(temp_own, temp_own_2);
146     input -= temp_own_2;
147     if ((int)rank + 1 < GlobalParams.NumberProcesses) {
148       MPI_Send(&input[0], own, MPI_DOUBLE, rank + 1, 0, mpi_comm);
149     }
```

```
150   }
151
152   for (int i = 0; i < own; i++) {
153     if (!fixed_dofs->is_element(indices[i])) {
154       dst[indices[i]] = input[i];
155     }
156   }
157 }
```

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

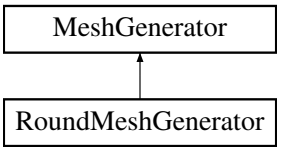
- Code/Core/PreconditionerSweeping.h
- Code/Core/PreconditionerSweeping.cpp

4.25 RoundMeshGenerator Class Reference

This class generates meshes, that are used to discretize a rectangular Waveguide.

`#include <RoundMeshGenerator.h>`

Inheritance diagram for RoundMeshGenerator:



Public Member Functions

- **RoundMeshGenerator** (SpaceTransformation *in_ct)
- void refine_global (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times)

  This function is intended to execute a global refinement of the mesh.
- void refine_internal (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times)

  This function is intended to execute an internal refinement of the mesh.
- void refine_proximity (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times, double factor)

  This function is intended to execute a refinement inside and near the waveguide boundary.
- bool math_coordinate_in_waveguide (Point< 3 > position) const

  This function checks if the given coordinate is inside the waveguide or not.
- bool phys_coordinate_in_waveguide (Point< 3 > position) const

  This function checks if the given coordinate is inside the waveguide or not.
- void set_boundary_ids (parallel::distributed::Triangulation< 3 > &tria) const

  This function is a helper during distributed mesh generation.
- void prepare_triangulation (parallel::distributed::Triangulation< 3 > *in_tria)

  This function takes a triangulation object and prepares it for the further computations.

Additional Inherited Members

4.25.1 Detailed Description

This class generates meshes, that are used to discretize a rectangular Waveguide.

It is derived from MeshGenerator.

This Generator creates a mesh around a cylindrical waveguide. It should be used in conjunction with a Space↩ Transformation, which uses a circular shape of the waveguide and an appropriate distribution of the DOFs.

Author

Pascal Kraft

Date

28.11.2016

Definition at line 21 of file RoundMeshGenerator.h.

4.25.2 Member Function Documentation

4.25.2.1 math_coordinate_in_waveguide()

```
bool RoundMeshGenerator::math_coordinate_in_waveguide (
            Point< 3 > position ) const  [virtual]
```

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. position This value gives us the location to check for.

Implements MeshGenerator.

Definition at line 243 of file RoundMeshGenerator.cpp.

Referenced by SquareMeshGenerator::refine_internal().

```
244                                       {
245   return Distance2D(in_position) <
246          (GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out) / 2.0;
247 }
```

4.25.2.2 phys_coordinate_in_waveguide()

```
bool RoundMeshGenerator::phys_coordinate_in_waveguide (
            Point< 3 > position ) const  [virtual]
```

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)). position This value gives us the location to check for.

Implements MeshGenerator.

Definition at line 249 of file RoundMeshGenerator.cpp.

References SpaceTransformation::get_m(), and SpaceTransformation::get_r().

```
250                                       {
251   Point<3, double> temp = in_position;
252   temp[1] -= ct->get_m(in_position[2]);
253   double r = ct->get_r(in_position[2]);
254   return (abs(temp[0]) < r && abs(temp[1]) < r);
255 }
```

4.25.2.3 prepare_triangulation()

```
void RoundMeshGenerator::prepare_triangulation (
            parallel::distributed::Triangulation< 3 > * in_tria )  [virtual]
```

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.

mesh_info(∗in_tria, "Output"+std::to_string(GlobalParams.MPI_Rank)+".vtk");

Implements MeshGenerator.

Definition at line 107 of file RoundMeshGenerator.cpp.

References set_boundary_ids().

```
108                                                {
109   deallog.push("RoundMeshGenerator:prepare_triangulation");
110   deallog << "Starting Mesh preparation" << std::endl;
111
112   const std_cxx11::array<Tensor<1, 3>, 3> edges2(edges);
113
114   GridGenerator::subdivided_parallelepiped<3, 3>(*in_tria, origin, edges2, subs,
115                                                  false);
```

```
116
117   in_tria->repartition();
118
119   in_tria->signals.post_refinement.connect(
120       std_cxx11::bind(&RoundMeshGenerator::set_boundary_ids,
121                       std_cxx11::cref(*this), std_cxx11::ref(*in_tria)));
122
123   in_tria->refine_global(3);
124
125   in_tria->set_all_manifold_ids(0);
126
127   GridTools::transform(&Triangulation_Stretch_to_circle, *in_tria);
128
129   unsigned int man = 1;
130
131   in_tria->set_manifold(man, round_description);
132
133   in_tria->set_all_manifold_ids(0);
134   cell = in_tria->begin_active();
135
136   endc = in_tria->end();
137   for (; cell != endc; ++cell) {
138     if (Distance2D(cell->center()) < 0.25) {
139       cell->set_all_manifold_ids(1);
140       cell->set_manifold_id(1);
141     }
142   }
143
144   in_tria->set_manifold(man, round_description);
145
146   in_tria->set_all_manifold_ids(0);
147   cell = in_tria->begin_active();
148   endc = in_tria->end();
149   for (; cell != endc; ++cell) {
150     if (Distance2D(cell->center()) < 0.25) {
151       cell->set_all_manifold_ids(1);
152       cell->set_manifold_id(1);
153     }
154   }
155
156   in_tria->set_manifold(man, round_description);
157   parallel::distributed::Triangulation<3>::active_cell_iterator
158       cell = in_tria->begin_active(),
159       endc = in_tria->end();
160
161   double len = 2.0 / Layers;
162
163   cell = in_tria->begin_active();
164   for (; cell != endc; ++cell) {
165     int temp = (int)std::floor((cell->center(true, false)[2] + 1.0) / len);
166     if (temp >= (int)Layers || temp < 0)
167       std::cout << "Critical Error in Mesh partitioning. See make_grid! "
168                    "Solvers might not work."
169                 << std::endl;
170   }
171
172   GridTools::transform(&Triangulation_Stretch_X, *in_tria);
173   GridTools::transform(&Triangulation_Stretch_Y, *in_tria);
174   GridTools::transform(&Triangulation_Stretch_Computational_Radius, *in_tria);
175
176   if (GlobalParams.R_Global > 0) {
177     in_tria->refine_global(GlobalParams.R_Global);
178   }
179
180   double MaxDistFromBoundary =
181       (GlobalParams.M_C_Dim1Out + GlobalParams.M_C_Dim1In) * 1.4 / 2.0;
182   for (int i = 0; i < GlobalParams.R_Local; i++) {
183     cell = in_tria->begin_active();
184     for (; cell != endc; ++cell) {
185       if (std::abs(Distance2D(cell->center(true, false)) -
186                    (GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out) / 2.0) <
187           MaxDistFromBoundary) {
188         cell->set_refine_flag();
189       }
190     }
191     in_tria->execute_coarsening_and_refinement();
192     MaxDistFromBoundary =
193         (MaxDistFromBoundary +
194          ((GlobalParams.M_C_Dim1Out + GlobalParams.M_C_Dim1In) / 2.0)) /
195         2.0;
196   }
197
198   for (int i = 0; i < GlobalParams.R_Interior; i++) {
199     cell = in_tria->begin_active();
200     for (; cell != endc; ++cell) {
201       if (Distance2D(cell->center(true, false)) <
202           (GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out) / 2.0) {
```

```
203         cell->set_refine_flag();
204       }
205     }
206     in_tria->execute_coarsening_and_refinement();
207   }
208
209   // mesh_info(triangulation, solutionpath + "/grid" +
210   // static_cast<std::ostringstream*>( &(std::ostringstream() <<
211   // GlobalParams.MPI_Rank) )->str() + ".vtk");
212
213   GridTools::transform(&Triangulation_Stretch_Z, *in_tria);
214
215   GridTools::transform(&Triangulation_Shift_Z, *in_tria);
216
217   z_min = 10000000.0;
218   z_max = -10000000.0;
219   cell = in_tria->begin_active();
220   endc = in_tria->end();
221
222   for (; cell != endc; ++cell) {
223     if (cell->is_locally_owned()) {
224       for (int face = 0; face < 6; face++) {
225         z_min = std::min(z_min, cell->face(face)->center()[2]);
226         z_max = std::max(z_max, cell->face(face)->center()[2]);
227       }
228     }
229   }
230
231   cell = in_tria->begin_active();
232   endc = in_tria->end();
233
234   /// mesh_info(*in_tria,
235   /// "Output"+std::to_string(GlobalParams.MPI_Rank)+".vtk");
236
237   set_boundary_ids(*in_tria);
238
239   deallog << "Done" << std::endl;
240   deallog.pop();
241 }
```

4.25.2.4 refine_global()

```
void RoundMeshGenerator::refine_global (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times )  [virtual]
```

This function is intended to execute a global refinement of the mesh.

This means that every cell will be refined in every direction (effectively multiplying the number of DOFs by 8). This version is the most expensive refinement possible and should be used with caution.

Parameters

times	Number of refinement steps to be performed (gives us a multiplication of the number of degrees of freedom by 8^{times}.

Implements MeshGenerator.

Definition at line 257 of file RoundMeshGenerator.cpp.

```
258                                                                    {
259   in_tria->refine_global(times);
260 }
```

4.25.2.5 refine_internal()

```
void RoundMeshGenerator::refine_internal (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times ) [virtual]
```

This function is intended to execute an internal refinement of the mesh.

This means that every cell inside the waveguide will be refined in every direction. This method is rather cheap and only refines where the field is strong, however, the mesh outside the waveguide should not be too coarse to reduce numerical errors.

Parameters

times	Number of refinement steps to be performed.

Implements MeshGenerator.

Definition at line 278 of file RoundMeshGenerator.cpp.

References MeshGenerator::math_coordinate_in_waveguide().

```
279                                                                     {
280   for (unsigned int i = 0; i < times; i++) {
281     cell = in_tria->begin_active();
282     for (; cell != endc; ++cell) {
283       if (math_coordinate_in_waveguide(cell->center())) {
284         cell->set_refine_flag();
285       }
286     }
287     in_tria->execute_coarsening_and_refinement();
288   }
289 }
```

4.25.2.6 refine_proximity()

```
void RoundMeshGenerator::refine_proximity (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times,
            double factor ) [virtual]
```

This function is intended to execute a refinement inside and near the waveguide boundary.

Parameters

times	Number of refinement steps to be performed.

Implements MeshGenerator.

Definition at line 262 of file RoundMeshGenerator.cpp.

```
264                 {
```

```
265   for (unsigned int t = 0; t < times; t++) {
266     double R = (GlobalParams.M_C_Dim1Out + GlobalParams.M_C_Dim1In) *
267                (1.0 + factor) / 2.0;
268     cell = in_tria->begin_active();
269     for (; cell != endc; ++cell) {
270       if (Distance2D(cell->center(true, false)) < R) {
271         cell->set_refine_flag();
272       }
273     }
274     in_tria->execute_coarsening_and_refinement();
275   }
276 }
```

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

- Code/MeshGenerators/RoundMeshGenerator.h
- Code/MeshGenerators/RoundMeshGenerator.cpp

4.26 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.
- 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, double, double, double)

  This function is used during the optimization-operation to update the properties of the space-transformation.
- void **set_properties** (double, double, double, double, double, double)
- void set_properties_force (double, double, double, double)

  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, double, double, double, double, double)
- 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.

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**
- double * **dofs_l**
- double * **dofs_r**
- unsigned int * **derivative**
- bool * **zero_derivative**

4.26.1 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 19 of file Sector.h.

4.26.2 Constructor & Destructor Documentation

4.26.2.1 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 20 of file Sector.cpp.

```
22     : left(in_left),
23       right(in_right),
24       boundary(in_left && in_right),
25       z_0(in_z_0),
26       z_1(in_z_1) {
27   dofs_l = new double[Dofs_Per_Sector];
28   dofs_r = new double[Dofs_Per_Sector];
29   derivative = new unsigned int[Dofs_Per_Sector];
30   zero_derivative = new bool[Dofs_Per_Sector];
31   if (Dofs_Per_Sector == 3) {
32     zero_derivative[0] = true;
33     zero_derivative[1] = false;
34     zero_derivative[2] = true;
35     derivative[0] = 0;
36     derivative[1] = 2;
37     derivative[2] = 0;
38   }
39   if (Dofs_Per_Sector == 2) {
40     zero_derivative[0] = false;
41     zero_derivative[1] = true;
42     derivative[0] = 1;
43     derivative[1] = 0;
44   }
45
46   for (unsigned int i = 0; i < Dofs_Per_Sector; i++) {
47     dofs_l[i] = 0;
48     dofs_r[i] = 0;
49   }
50   NInternalBoundaryDofs = 0;
51   LowestDof = 0;
52   NActiveCells = 0;
53   NDofs = Dofs_Per_Sector;
54 }
```

4.26.3 Member Function Documentation

4.26.3.1 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 158 of file Sector.cpp.

```
158                                                                    {
159   if (i > 0 && i < NDofs) {
160     if (z < 0.0) z = 0.0;
161     if (z > 1.0) z = 1.0;
162     if (zero_derivative[i]) {
163       return InterpolationPolynomialZeroDerivative(z, dofs_l[i], dofs_r[i]);
164     } else {
165       return InterpolationPolynomial(z, dofs_l[i], dofs_r[i],
166                                      dofs_l[derivative[i]],
167                                      dofs_r[derivative[i]]);
168     }
169   } else {
170     std::cout << "There seems to be an error in Sector::get_dof. i > 0 && i < "
171                  "dofs_per_sector false."
172               << std::endl;
173     return 0;
174   }
175 }
```

4.26.3.2 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 190 of file Sector.cpp.

```
190                                                {
191   if (z < 0.0) z = 0.0;
192   if (z > 1.0) z = 1.0;
```

```
193   if (Dofs_Per_Sector == 2) {
194     return InterpolationPolynomial(z, dofs_l[0], dofs_r[0], dofs_l[1],
195                                    dofs_r[1]);
196   } else {
197     return InterpolationPolynomial(z, dofs_l[1], dofs_r[1], dofs_l[2],
198                                    dofs_r[2]);
199   }
200 }
```

4.26.3.3 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 178 of file Sector.cpp.

```
178                                                    {
179   if (z < 0.0) z = 0.0;
180   if (z > 1.0) z = 1.0;
181   if (Dofs_Per_Sector < 3) {
182     deallog << "Error in Sector: Acces to radius dof without existence."
183             << std::endl;
184     return 0;
185   }
186   return InterpolationPolynomialZeroDerivative(z, dofs_l[0], dofs_r[0]);
187 }
```

4.26.3.4 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 203 of file Sector.cpp.

```
203                                                    {
204   if (z < 0.0) z = 0.0;
```

```
205   if (z > 1.0) z = 1.0;
206   if (Dofs_Per_Sector == 2) {
207     return InterpolationPolynomialZeroDerivative(z, dofs_l[1], dofs_r[1]);
208   } else {
209     return InterpolationPolynomialZeroDerivative(z, dofs_l[2], dofs_r[2]);
210   }
211 }
```

4.26.3.5 getLowestDof()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getLowestDof ( ) const
```

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 420 of file Sector.cpp.

```
420                                                    {
421   return LowestDof;
422 }
```

4.26.3.6 getNActiveCells()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getNActiveCells ( ) const
```

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 435 of file Sector.cpp.

```
435                                                      {
436   return NActiveCells;
437 }
```

4.26.3.7 getNDofs()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::getNDofs ( ) const
```

This function returns the number of dofs which are part of this sector.

The same remarks as for getLowestDof() apply.

Definition at line 425 of file Sector.cpp.

```
425                                                 {
426   return NDofs;
427 }
```

4.26.3.8 getNInternalBoundaryDofs()

```
template<unsigned int Dofs_Per_Sector>
unsigned int Sector< Dofs_Per_Sector >::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.

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 430 of file Sector.cpp.

```
430                                                         {
431   return NInternalBoundaryDofs;
432 }
```

4.26.3.9 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 214 of file Sector.cpp.

```
214                                                 {
215   return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
216               z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
217 }
```

4.26.3.10 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 220 of file Sector.cpp.

```
220                                                 {
221   return 1 / (dofs_l[0] + z * z * z * (2 * dofs_l[0] - 2 * dofs_r[0]) -
222               z * z * (3 * dofs_l[0] - 3 * dofs_r[0]));
223 }
```

4.26.3.11 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 226 of file Sector.cpp.

```
226                                                 {
227   return 0.0;
228 }
```

4.26.3.12 set_properties()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::set_properties (
            double ,
            double ,
            double ,
            double  )
```

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 defrees of freedom if left is true and it cannot edit the right degrees of freedom if right is true

Definition at line 127 of file Sector.cpp.

Referenced by Sector< 3 >::Sector(), and Sector< 3 >::set_properties().

```
127                                                                     {
128   std::cout << "The code does not work for this number of dofs per Sector."
129             << std::endl;
130   return;
131 }
```

4.26.3.13 setLowestDof()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setLowestDof (
            unsigned int in_lowestdof )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 440 of file Sector.cpp.

```
440                                                          {
441   LowestDof = in_lowestdof;
442 }
```

4.26.3.14 setNActiveCells()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setNActiveCells (
            unsigned int in_nactivecells )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 456 of file Sector.cpp.

```
456                                                             {
457   NActiveCells = in_nactivecells;
458 }
```

4.26.3.15 setNDofs()

```
template<unsigned int Dofs_Per_Sector>
void Sector< Dofs_Per_Sector >::setNDofs (
            unsigned int in_ndofs )
```

Setter for the value that the getter should return.

Called after Dof-reordering.

Definition at line 445 of file Sector.cpp.

```
445                                                      {
446   NDofs = in_ndofs;
447 }
```

4.26.3.16 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 450 of file Sector.cpp.

```
451                                             {
452   NInternalBoundaryDofs = in_ninternalboundarydofs;
453 }
```

4.26.3.17 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 411 of file Sector.cpp.

Referenced by Sector< 3 >::getQ3().

```
412                                     {
413   Tensor<2, 3, double> ret;
414   std::cout << "The code does not work for you Sector specification."
415             << Dimension << std::endl;
416   return ret;
417 }
```

4.26.4 Member Data Documentation

4.26.4.1 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 60 of file Sector.h.

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

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

4.27 ShapeDescription Class Reference

Public Member Functions

- void **SetByString** (std::string)

Public Attributes

- int **Sectors**
- std::vector< double > **m**
- std::vector< double > **v**
- std::vector< double > **z**

4.27.1 Detailed Description

Definition at line 14 of file ShapeDescription.h.

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

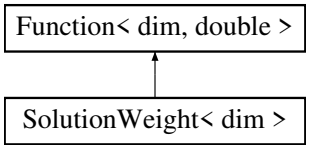
- Code/Helpers/ShapeDescription.h
- Code/Helpers/ShapeDescription.cpp

4.28 SolutionWeight< dim > Class Template Reference

This function has internal usage to execute a function only on the interior of the Waveguide.

```
#include <SolutionWeight.h>
```

Inheritance diagram for SolutionWeight< dim >:



Public Member Functions

- SolutionWeight ()

  The whole class contains no specific data.
- virtual double value (const Point< dim > &p, const unsigned int component) const

  This function returns 1, if the given component is inside the Waveguide and 0 otherwise.
- virtual void vector_value (const Point< dim > &p, Vector< double > &value) const

  This function gets called by the framework and calls value(const Point<dim> &p, const unsigned int component) for all components and stores the results in value.

4.28.1 Detailed Description

template<int dim>
class SolutionWeight< dim >

This function has internal usage to execute a function only on the interior of the Waveguide.

SolutionWeight is a class, that has been derived from the class Function and which can be used to generate handles to functions, that return specific values. The pattern of passing object built from classes derived from Function is a commonly used one in Deal.II. This function offers a weight for locations inside the waveguide. In order to integrate or calculate the L2-norm inside the Waveguide, this is needed. The value of the function to be used is multiplied with this weighing-function which returns 1 for any point inside the Waveguide and 0 outside of it. Mathematically, this function can be represented by

$$f : \mathbb{R}^{dim} \to \{1, 0\},\ \boldsymbol{x} \mapsto \begin{cases} 1 & \text{for } \boldsymbol{x} \in \Omega_W \\ 0 & \text{else} \end{cases}$$

, where Ω_W is the set of all points contained in the Waveguide.

Definition at line 40 of file SolutionWeight.h.

4.28.2 Constructor & Destructor Documentation

4.28.2.1 SolutionWeight()

```
template<int dim>
SolutionWeight< dim >::SolutionWeight ( )
```

The whole class contains no specific data.

The only information it needs stem from the Parameters object parsed from the input file. This object gets initialized inside this constructor.

Definition at line 29 of file SolutionWeight.cpp.

```
29 : Function<dim>(6) {}
```

4.28.3 Member Function Documentation

4.28.3.1 value()

```
template<int dim>
double SolutionWeight< dim >::value (
            const Point< dim > & p,
            const unsigned int component ) const  [virtual]
```

This function returns 1, if the given component is inside the Waveguide and 0 otherwise.

Since this method is intended also for vector-valued functions , this method also has to account for a component of the result which for scalar functions is 0.

Parameters

p	This is the location in which the test should be executed.
component	Determines which component is to be evaluated. In this case that information does not have any further meaning.

Definition at line 8 of file SolutionWeight.cpp.

```
9                                                                    {
10   double value = Distance2D(p);
11   if (value < (GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out) / 2.0) {
12     if (p[2] < GlobalParams.M_R_ZLength / 2.0) {
13       return 1.0;
14     } else {
15       return 0.0;
16     }
17   } else
18     return 0.0;
19 }
```

4.28.3.2 vector_value()

```
template<int dim>
void SolutionWeight< dim >::vector_value (
            const Point< dim > & p,
            Vector< double > & value ) const  [virtual]
```

This function gets called by the framework and calls value(const Point<dim> &p, const unsigned int component) for all components and stores the results in value.

Parameters

p	The location is given here and gets passed along to the individual value-calls.
value	This is a vector which returns the results in place. It is a reference that is edited in value(const Point<dim> &p, const unsigned int component).

Definition at line 22 of file SolutionWeight.cpp.

```
23                                                                    {
24   for (unsigned int c = 0; c < 6; ++c)
25     values[c] = SolutionWeight<dim>::value(p, c);
26 }
```

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

- Code/Core/SolutionWeight.h
- Code/Core/SolutionWeight.cpp

4.29 SpaceTransformation Class Reference

The SpaceTransformation class encapsulates the coordinate transformation used in the simulation.

`#include <SpaceTransformation.h>`

Inheritance diagram for SpaceTransformation:

Public Member Functions

- **SpaceTransformation** (int, int)
- virtual Point< 3 > **math_to_phys** (Point< 3 > coord) const =0
- virtual Point< 3 > **phys_to_math** (Point< 3 > coord) const =0
- bool **is_identity** (Point< 3 > coord) const
- virtual Tensor< 2, 3, std::complex< double > > **get_Tensor** (Point< 3 > &coordinate) const =0
- virtual Tensor< 2, 3, std::complex< double > > **get_Preconditioner_Tensor** (Point< 3 > &coordinate, int block) const =0
- virtual Tensor< 2, 3, double > **get_Space_Transformation_Tensor** (Point< 3 > &coordinate) const =0
- virtual Tensor< 2, 3, double > **get_Space_Transformation_Tensor_Homogenized** (Point< 3 > &coordinate) const =0
- virtual Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input) const =0
- virtual Tensor< 2, 3, std::complex< double > > **Apply_PML_To_Tensor_For_Preconditioner** (Point< 3 > &coordinate, Tensor< 2, 3, double > Tensor_input, int block) const =0
- virtual Tensor< 2, 3, std::complex< double > > **get_Tensor_for_step** (Point< 3 > &coordinate, unsigned int dof, double step_width)
- virtual bool PML_in_X (Point< 3 > &position) const =0

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.
- virtual bool PML_in_Y (Point< 3 > &position) const =0

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.
- virtual bool PML_in_Z (Point< 3 > &position) const =0

  This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.
- virtual double Preconditioner_PML_Z_Distance (Point< 3 > &p, unsigned int block) const =0

This function fulfills the same purpose as those with similar names but it is supposed to be used together with Preconditioner_PML_in_Z instead of the versions without "Preconditioner".

- virtual double PML_X_Distance (Point< 3 > &position) const =0

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- virtual double PML_Y_Distance (Point< 3 > &position) const =0

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- virtual double PML_Z_Distance (Point< 3 > &position) const =0

  This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.
- 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_Q1 (double z) const =0

  This member calculates the value of Q1 for a provided z-coordinate.
- virtual double get_Q2 (double z) const =0

  This member calculates the value of Q2 for a provided z-coordinate.
- virtual double get_Q3 (double z) const =0

  This member calculates the value of Q3 for a provided z-coordinate.
- virtual double get_dof (int dof) const =0

  This is a getter for the values of degrees of freedom.
- virtual void set_dof (int dof, double value)=0

  This function sets the value of the dof provided to the given value.
- virtual std::pair< double, double > **dof_support** (unsigned int index) const
- bool **point_in_dof_support** (Point< 3 > location, unsigned int dof_index) const
- virtual double get_free_dof (int dof) const =0

  This is a getter for the values of degrees of freedom.
- virtual void set_free_dof (int dof, double value)=0

  This function sets the value of the dof provided to the given value.
- std::pair< int, double > Z_to_Sector_and_local_z (double in_z) const

  Using this method unifies the usage of coordinates.
- double Sector_Length () const

  Returns the length of one sector.
- virtual double get_r (double in_z) const =0

  Returns the radius for a system-coordinate;.
- virtual double get_m (double in_z) const =0

  Returns the shift for a system-coordinate;.
- virtual double get_v (double in_z) const =0

  Returns the tilt for a system-coordinate;.
- int Z_to_Layer (double) const

  This Method writes a comprehensive description of the current structure to the console.
- virtual Vector< double > Dofs () const =0

  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 NFreeDofs () const =0

  This function returns the number of unrestrained degrees of freedom of the current optimization run.
- virtual unsigned int NDofs () const =0

  This function returns the total number of DOFs including restrained ones.
- virtual bool IsDofFree (int) const =0

  Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.
- virtual void Print () const =0

  Console output of the current Waveguide Structure.
- virtual std::complex< double > evaluate_for_z (double, Waveguide *)=0

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

- std::complex< double > **evaluate_for_z_with_sum** (double, double, Waveguide *)
- std::complex< double > **gauss_product_2D_sphere** (double z, int n, double R, double Xc, double Yc, Waveguide *in_w)

Public Attributes

- bool **homogenized** = false
- const unsigned int **dofs_per_layer**
- const unsigned int **boundary_dofs_in**
- const unsigned int **boundary_dofs_out**
- const double epsilon_K

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const double epsilon_M

  The material-property ϵ_r has a different value inside and outside of the waveguides core.
- const int sectors

  Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.
- const double deltaY

  This value is initialized with the value Delta from the input-file.
- Vector< double > InitialDofs

  This vector of values saves the initial configuration.
- double InitialQuality

  This vector of values saves the initial configuration.
- const int **rank**

4.29.1 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 3x3 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 29 of file SpaceTransformation.h.

4.29.2 Member Function Documentation

4.29.2.1 Dofs()

```
virtual Vector<double> SpaceTransformation::Dofs ( ) const  [pure 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.

Implemented in DualProblemTransformationWrapper, InhomogenousTransformationCircular, HomogenousTransformationCircular, HomogenousTransformationRectangular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::Dofs().

4.29.2.2 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 DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::estimate_and_initialize().

4.29.2.3 evaluate_for_z()

```
virtual std::complex<double> SpaceTransformation::evaluate_for_z (
            double ,
            Waveguide *  )  [pure virtual]
```

Since the Wavegudie itself may be circular or rectangular now, the evaluation routines should be moved to a point in the code where this information is included in the code.

Since I dont want to create derived classes from waveguide (which I should do eventually) I will for now include this functionality into the space transformation which is shape-sensitive. The waveguide only offers the evaluation at a point. The quadrature-rule has to be imposed by the space transformation.

Implemented in DualProblemTransformationWrapper, InhomogenousTransformationCircular, HomogenousTransformationCircular, HomogenousTransformationRectangular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::evaluate_for_z().

4.29.2.4 get_dof()

```
virtual double SpaceTransformation::get_dof (
            int dof ) const  [pure 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.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::get_dof(), and Z_to_Layer().

4.29.2.5 get_free_dof()

```
virtual double SpaceTransformation::get_free_dof (
            int dof ) const  [pure 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.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::get_free_dof().

4.29.2.6 get_Q1()

```
virtual double SpaceTransformation::get_Q1 (
            double z ) const  [pure virtual]
```

This member calculates the value of Q1 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q1 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::get_Q1().

4.29.2.7 get_Q2()

```
virtual double SpaceTransformation::get_Q2 (
            double z ) const  [pure virtual]
```

This member calculates the value of Q2 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q2 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::get_Q2().

4.29.2.8 get_Q3()

```
virtual double SpaceTransformation::get_Q3 (
            double z ) const  [pure virtual]
```

This member calculates the value of Q3 for a provided z-coordinate.

This value is used in the transformation of the solution-vector in transformed coordinates (solution of the system-matrix) to real coordinates (physical field).

Parameters

z	The value of Q3 is independent of x and y. Therefore only a z-coordinate is provided in a call to the function.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::get_Q3().

4.29.2.9 IsDofFree()

```
virtual bool SpaceTransformation::IsDofFree (
            int  ) const  [pure virtual]
```

Since Dofs() also returns restrained degrees of freedom, this function can be applied to determine if a degree of freedom is indeed free or restrained.

"restrained" means that for example the DOF represents the radius at one of the connectors (input or output) and therefore we forbid the optimization scheme to vary this value.

Implemented in DualProblemTransformationWrapper, InhomogenousTransformationCircular, HomogenousTransformationCircular, HomogenousTransformationRectangular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::IsDofFree(), and Waveguide::run().

4.29.2.10 NDofs()

```
virtual unsigned int SpaceTransformation::NDofs ( ) const  [pure virtual]
```

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

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

Implemented in DualProblemTransformationWrapper, InhomogenousTransformationCircular, HomogenousTransformationCircular, HomogenousTransformationRectangular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::NDofs(), and Waveguide::run().

4.29.2.11 PML_in_X()

```
virtual bool SpaceTransformation::PML_in_X (
            Point< 3 > & position ) const  [pure virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the x-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::PML_in_X().

4.29.2.12 PML_in_Y()

```
virtual bool SpaceTransformation::PML_in_Y (
            Point< 3 > & position ) const  [pure virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the y-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::PML_in_Y().

4.29.2.13 PML_in_Z()

```
virtual bool SpaceTransformation::PML_in_Z (
            Point< 3 > & position ) const  [pure virtual]
```

This function is used to determine, if a system-coordinate belongs to a PML-region for the PML that limits the computational domain along the z-axis.

Since there are 3 blocks of PML-type material, there are 3 functions.

Parameters

position	Stores the position in which to test for presence of a PML-Material.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::PML_in_Z().

4.29.2.14 PML_X_Distance()

```
virtual double SpaceTransformation::PML_X_Distance (
            Point< 3 > & position ) const  [pure virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::PML_X_Distance().

4.29.2.15 PML_Y_Distance()

```
virtual double SpaceTransformation::PML_Y_Distance (
            Point< 3 > & position ) const  [pure virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::PML_Y_Distance().

4.29.2.16 PML_Z_Distance()

```
virtual double SpaceTransformation::PML_Z_Distance (
            Point< 3 > & position ) const  [pure virtual]
```

This function calculates for a given point, its distance to a PML-boundary limiting the computational domain.

This function is used merely to make code more readable. There is a function for every one of the dimensions since the normal vectors of PML-regions in this implementation are the coordinate-axis. This value is set to zero outside the PML and positive inside both PML-domains (only one for the z-direction).

Parameters

position	Stores the position from which to calculate the distance to the PML-surface.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::PML_Z_Distance().

4.29.2.17 set_dof()

```
virtual void SpaceTransformation::set_dof (
            int dof,
            double value )  [pure 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.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::set_dof(), and Z_to_Layer().

4.29.2.18 set_free_dof()

```
virtual void SpaceTransformation::set_free_dof (
            int dof,
            double value )  [pure 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.

Implemented in DualProblemTransformationWrapper, HomogenousTransformationCircular, HomogenousTransformationRectangular, InhomogenousTransformationCircular, and InhomogenousTransformationRectangular.

Referenced by DualProblemTransformationWrapper::set_free_dof().

4.29.2.19 Z_to_Sector_and_local_z()

```
std::pair< int, double > SpaceTransformation::Z_to_Sector_and_local_z (
            double in_z ) const
```

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 13 of file SpaceTransformation.cpp.

References deltaY, epsilon_K, epsilon_M, InitialQuality, and sectors.

Referenced by InhomogenousTransformationRectangular::get_m(), HomogenousTransformationRectangular::get_m(), InhomogenousTransformationCircular::get_m(), InhomogenousTransformationRectangular::get_Q1(), HomogenousTransformationRectangular::get_Q1(), InhomogenousTransformationCircular::get_Q1(), InhomogenousTransformationRectangular::get_Q2(), HomogenousTransformationRectangular::get_Q2(), InhomogenousTransformationCircular::get_Q2(), InhomogenousTransformationRectangular::get_Q3(), HomogenousTransformationRectangular::get_Q3(), InhomogenousTransformationCircular::get_Q3(), InhomogenousTransformationCircular::get_r(), InhomogenousTransformationRectangular::get_v(), HomogenousTransformationRectangular::get_v(), InhomogenousTransformationCircular::get_v(), InhomogenousTransformationRectangular::NDofs(), InhomogenousTransformationCircular::PML_Z_Distance(), HomogenousTransformationRectangular::PML_Z_Distance(), and DualProblemTransformationWrapper::Z_to_Sector_and_local_z().

```
14                        {
15   std::pair<int, double> ret;
16   ret.first = 0;
17   ret.second = 0.0;
18   if (in_z <= -GlobalParams.M_R_ZLength / 2.0) {
19     ret.first = 0;
20     ret.second = 0.0;
21   } else if (abs(in_z) < GlobalParams.M_R_ZLength / 2.0) {
22     ret.first = floor((in_z + GlobalParams.M_R_ZLength / 2.0) /
23                       (GlobalParams.SectorThickness));
24     ret.second = (in_z + GlobalParams.M_R_ZLength / 2.0 -
25                   (ret.first * GlobalParams.SectorThickness)) /
26                  (GlobalParams.SectorThickness);
27   } else if (in_z >= GlobalParams.M_R_ZLength / 2.0) {
28     ret.first = sectors - 1;
29     ret.second = 1.0;
30   }
31   return ret;
32 }
```

4.29.3 Member Data Documentation

4.29.3.1 epsilon_K

```
const double SpaceTransformation::epsilon_K
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value inside the core.

Definition at line 143 of file SpaceTransformation.h.

Referenced by Z_to_Sector_and_local_z().

4.29.3.2 epsilon_M

```
const double SpaceTransformation::epsilon_M
```

The material-property ϵ_r has a different value inside and outside of the waveguides core.

This variable stores its value outside the core.

Definition at line 149 of file SpaceTransformation.h.

Referenced by Z_to_Sector_and_local_z().

4.29.3.3 sectors

```
const int SpaceTransformation::sectors
```

Since the computational domain is split into subdomains (called sectors), it is important to keep track of the amount of subdomains.

This member stores the number of Sectors the computational domain has been split into.

Definition at line 155 of file SpaceTransformation.h.

Referenced by Z_to_Sector_and_local_z().

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

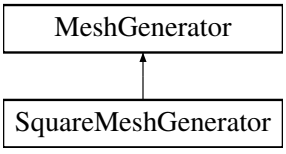
- Code/SpaceTransformations/SpaceTransformation.h
- Code/SpaceTransformations/SpaceTransformation.cpp

4.30 SquareMeshGenerator Class Reference

This class generates meshes, that are used to discretize a rectangular Waveguide.

```
#include <SquareMeshGenerator.h>
```

Inheritance diagram for SquareMeshGenerator:



Public Member Functions

- **SquareMeshGenerator** (SpaceTransformation *st)
- void refine_global (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times)

  This function is intended to execute a global refinement of the mesh.
- void refine_internal (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times)

  This function is intended to execute an internal refinement of the mesh.
- void refine_proximity (parallel::distributed::Triangulation< 3 > *in_tria, unsigned int times, double factor)

  This function is intended to execute a refinement inside and near the waveguide boundary.
- bool math_coordinate_in_waveguide (Point< 3 > position) const

  This function checks if the given coordinate is inside the waveguide or not.
- bool phys_coordinate_in_waveguide (Point< 3 > position) const

  This function checks if the given coordinate is inside the waveguide or not.
- void prepare_triangulation (parallel::distributed::Triangulation< 3 > *in_tria)

  This function takes a triangulation object and prepares it for the further computations.
- void **set_boundary_ids** (parallel::distributed::Triangulation< 3 > &tria) const

Public Attributes

- parallel::distributed::Triangulation< 3 >::active_cell_iterator **cell**
- parallel::distributed::Triangulation< 3 >::active_cell_iterator **endc**

4.30.1 Detailed Description

This class generates meshes, that are used to discretize a rectangular Waveguide.

It is derived from MeshGenerator.

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 24 of file SquareMeshGenerator.h.

4.30.2 Member Function Documentation

4.30.2.1 math_coordinate_in_waveguide()

```
bool SquareMeshGenerator::math_coordinate_in_waveguide (
            Point< 3 > position ) const  [virtual]
```

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. position This value gives us the location to check for.

Implements MeshGenerator.

Definition at line 191 of file SquareMeshGenerator.cpp.

```
192                                         {
193   return std::abs(in_position[0]) <
194              (GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out) / 2.0 &&
195          std::abs(in_position[1]) <
196              (GlobalParams.M_C_Dim2In + GlobalParams.M_C_Dim2Out) / 2.0;
197 }
```

4.30.2.2 phys_coordinate_in_waveguide()

```
bool SquareMeshGenerator::phys_coordinate_in_waveguide (
            Point< 3 > position ) const  [virtual]
```

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)). position This value gives us the location to check for.

Implements MeshGenerator.

Definition at line 199 of file SquareMeshGenerator.cpp.

References SpaceTransformation::get_m(), and SpaceTransformation::get_r().

```
200                                         {
201   std::cout
202       << "NOT IMPLEMENTED: SquareMeshGenerator::phys_coordinate_in_waveguide"
203       << std::endl;
204   Point<3, double> temp = in_position;
205   temp[1] -= ct->get_m(in_position[2]);
206   double r = ct->get_r(in_position[2]);
207   return (abs(temp[0]) < r && abs(temp[1]) < r);
208   return false;
209 }
```

4.30.2.3 prepare_triangulation()

```
void SquareMeshGenerator::prepare_triangulation (
            parallel::distributed::Triangulation< 3 > * in_tria )  [virtual]
```

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.

Implements MeshGenerator.

Definition at line 93 of file SquareMeshGenerator.cpp.

References RoundMeshGenerator::set_boundary_ids().

```
94                                                {
95   deallog.push("SquareMeshGenerator:prepare_triangulation");
96   deallog << "Starting Mesh preparation" << std::endl;
97
98   const std_cxx11::array<Tensor<1, 3>, 3> edges2(edges);
99
100   GridGenerator::subdivided_parallelepiped<3, 3>(*in_tria, origin, edges2, subs,
101                                                  false);
102
103   in_tria->repartition();
104
105   in_tria->signals.post_refinement.connect(
106       std_cxx11::bind(&SquareMeshGenerator::set_boundary_ids,
107                       std_cxx11::cref(*this), std_cxx11::ref(*in_tria)));
108
109   in_tria->refine_global(3);
110
111   GridTools::transform(&Triangulation_Stretch_Computational_Rectangle,
112                        *in_tria);
113
114   parallel::distributed::Triangulation<3>::active_cell_iterator
115
116       cell = in_tria->begin_active(),
117       endc = in_tria->end();
118
119   double len = 2.0 / Layers;
120
121   cell = in_tria->begin_active();
122   for (; cell != endc; ++cell) {
123     int temp = (int)std::floor((cell->center()[2] + 1.0) / len);
124     if (temp >= (int)Layers || temp < 0)
125       std::cout << "Critical Error in Mesh partitioning. See make_grid! "
126                    "Solvers might not work."
127                 << std::endl;
128   }
129
130   if (GlobalParams.R_Global > 0) {
131     in_tria->refine_global(GlobalParams.R_Global);
132   }
133
134   double MaxDistX =
135       (GlobalParams.M_C_Dim1Out + GlobalParams.M_C_Dim1In) * 1.4 / 2.0;
136   double MaxDistY =
137       (GlobalParams.M_C_Dim2Out + GlobalParams.M_C_Dim2In) * 1.4 / 2.0;
138   for (int i = 0; i < GlobalParams.R_Local; i++) {
139     cell = in_tria->begin_active();
140     for (; cell != endc; ++cell) {
141       if (std::abs(cell->center()[0]) < MaxDistX &&
142           std::abs(cell->center()[1]) < MaxDistY) {
143         cell->set_refine_flag();
144       }
145     }
146     in_tria->execute_coarsening_and_refinement();
147     MaxDistX = (GlobalParams.M_C_Dim1Out + GlobalParams.M_C_Dim1In) * 1.4 / 2.0;
148     MaxDistY = (GlobalParams.M_C_Dim2Out + GlobalParams.M_C_Dim2In) * 1.4 / 2.0;
149   }
150
151   for (int i = 0; i < GlobalParams.R_Interior; i++) {
152     cell = in_tria->begin_active();
153     for (; cell != endc; ++cell) {
154       if (std::abs(cell->center()[0]) <
155               (GlobalParams.M_C_Dim1In + GlobalParams.M_C_Dim1Out) / 2.0 &&
156           std::abs(cell->center()[1]) <
157               (GlobalParams.M_C_Dim2In + GlobalParams.M_C_Dim2Out) / 2.0) {
158         cell->set_refine_flag();
159       }
160     }
161     in_tria->execute_coarsening_and_refinement();
```

```
162   }
163
164   GridTools::transform(&Triangulation_Stretch_Z, *in_tria);
165
166   GridTools::transform(&Triangulation_Shift_Z, *in_tria);
167
168   z_min = 10000000.0;
169   z_max = -10000000.0;
170   cell = in_tria->begin_active();
171   endc = in_tria->end();
172
173   for (; cell != endc; ++cell) {
174     if (cell->is_locally_owned()) {
175       for (int face = 0; face < 6; face++) {
176         z_min = std::min(z_min, cell->face(face)->center()[2]);
177         z_max = std::max(z_max, cell->face(face)->center()[2]);
178       }
179     }
180   }
181
182   cell = in_tria->begin_active();
183   endc = in_tria->end();
184
185   set_boundary_ids(*in_tria);
186
187   deallog << "Done" << std::endl;
188   deallog.pop();
189 }
```

4.30.2.4 refine_global()

```
void SquareMeshGenerator::refine_global (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times )  [virtual]
```

This function is intended to execute a global refinement of the mesh.

This means that every cell will be refined in every direction (effectively multiplying the number of DOFs by 8). This version is the most expensive refinement possible and should be used with caution.

Parameters

times	Number of refinement steps to be performed (gives us a multiplication of the number of degrees of freedom by 8^{times}.

Implements MeshGenerator.

Definition at line 211 of file SquareMeshGenerator.cpp.

```
212                                                                    {
213   in_tria->refine_global(times);
214 }
```

4.30.2.5 refine_internal()

```
void SquareMeshGenerator::refine_internal (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times )  [virtual]
```

This function is intended to execute an internal refinement of the mesh.

This means that every cell inside the waveguide will be refined in every direction. This method is rather cheap and only refines where the field is strong, however, the mesh outside the waveguide should not be too coarse to reduce numerical errors.

Parameters

times	Number of refinement steps to be performed.

Implements MeshGenerator.

Definition at line 234 of file SquareMeshGenerator.cpp.

References RoundMeshGenerator::math_coordinate_in_waveguide().

```
235                                                                    {
236   for (unsigned int i = 0; i < times; i++) {
237     cell = in_tria->begin_active();
238     for (; cell != endc; ++cell) {
239       if (math_coordinate_in_waveguide(cell->center())) {
240         cell->set_refine_flag();
241       }
242     }
243     in_tria->execute_coarsening_and_refinement();
244   }
245 }
```

4.30.2.6 refine_proximity()

```
void SquareMeshGenerator::refine_proximity (
            parallel::distributed::Triangulation< 3 > * in_tria,
            unsigned int times,
            double factor )  [virtual]
```

This function is intended to execute a refinement inside and near the waveguide boundary.

Parameters

times	Number of refinement steps to be performed.

Implements MeshGenerator.

Definition at line 216 of file SquareMeshGenerator.cpp.

```
218                   {
219   double X = (GlobalParams.M_C_Dim1Out + GlobalParams.M_C_Dim1In) *
220              (1.0 + factor) / 2.0;
221   double Y = (GlobalParams.M_C_Dim2Out + GlobalParams.M_C_Dim2In) *
222              (1.0 + factor) / 2.0;
223   for (unsigned int i = 0; i < times; i++) {
224     cell = in_tria->begin_active();
225     for (; cell != endc; ++cell) {
226       if (std::abs(cell->center()[0]) < X && std::abs(cell->center()[1]) < Y) {
227         cell->set_refine_flag();
228       }
229     }
```

```
230     in_tria->execute_coarsening_and_refinement();
231   }
232 }
```

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

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

4.31 tagGSPHERE Struct Reference

Public Attributes

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

4.31.1 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

4.32 Waveguide Class Reference

This class encapsulates all important mechanism for solving a FEM problem.

```
#include <Waveguide.h>
```

Public Member Functions

- Waveguide (MPI_Comm in_mpi_comm, MeshGenerator *in_mg, SpaceTransformation *in_st)

  This is the constructor that should be used to initialize objects of this type.
- void run ()

  This method as well as the rerun() method, are used by the optimization-algorithm to use and reuse the Waveguide-object.
- void assemble_part ()

  The assemble_part function is a part of the assemble_system() functionality.
- std::complex< double > evaluate_for_Position (double x, double y, double z)

  To compute the output quality of the signal and it's transmition along the waveguid-axis, this function performs a comparison of the fundamental mode of a waveguide and the actual situation.
- double evaluate_for_z (double)

  To compute the output quality of the signal and it's transmition along the waveguid-axis, this function performs a comparison of the fundamental mode of a waveguide and the actual situation.
- void evaluate ()

  This function has the purpose of filling the qualities array in every process with the appropriate Values from the other ones.
- void store ()

  The storage has the following purpose: Regarding the optimization-process there are two kinds of runs.
- void estimate_solution ()

  This function is currently not in use.
- std::vector< std::complex< double > > **assemble_adjoint_local_contribution** (double stepwidth)
- void **switch_to_primal** (SpaceTransformation *primal_st)
- void **switch_to_dual** (SpaceTransformation *dual_st)
- Point< 3, double > **transform_coordinate** (const Point< 3, double >)

Public Attributes

- double * **qualities**
- MeshGenerator * **mg**

4.32.1 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 assemblation-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 102 of file Waveguide.h.

4.32.2 Constructor & Destructor Documentation

4.32.2.1 Waveguide()

```
Waveguide::Waveguide (
             MPI_Comm in_mpi_comm,
             MeshGenerator * in_mg,
             SpaceTransformation * in_st )
```

This is the constructor that should be used to initialize objects of this type.

Parameters

param	This is a reference to a parsed form of the input-file.
structure	This parameter gives a reference to the structure of the real Waveguide. This is necessary since during matrix-assembly it is required to call a function which generates the transformation tensor which is purely structure-dependent.

Definition at line 44 of file Waveguide.cpp.

```
46     : fe(FE_Nedelec<3>(GlobalParams.So_ElementOrder), 2),
47       triangulation(in_mpi_comm,
48                     parallel::distributed::Triangulation<3>::MeshSmoothing(
49                         parallel::distributed::Triangulation<3>::none),
50                     parallel::distributed::Triangulation<
51                         3>::Settings::no_automatic_repartitioning),
52       even(Utilities::MPI::this_mpi_process(in_mpi_comm) % 2 == 0),
53       rank(Utilities::MPI::this_mpi_process(in_mpi_comm)),
54       real(0),
55       imag(3),
56       solver_control(GlobalParams.So_TotalSteps, GlobalParams.So_Precision,
57                      true, true),
58       dof_handler(triangulation),
59       run_number(0),
60       condition_file_counter(0),
61       eigenvalue_file_counter(0),
62       Layers(GlobalParams.NumberProcesses),
63       Dofs_Below_Subdomain(Layers),
64       Block_Sizes(Layers),
65       is_stored(false),
66       Sectors(GlobalParams.M_W_Sectors),
67       minimum_local_z(2.0 * GlobalParams.M_R_ZLength),
68       maximum_local_z(-2.0 * GlobalParams.M_R_ZLength),
69       pout(std::cout, rank == 0),
70       timer(in_mpi_comm, pout, TimerOutput::OutputFrequency::summary,
71             TimerOutput::wall_times),
72       es(GlobalParams.M_C_Shape == ConnectorType::Rectangle) {
73   mg = in_mg;
74   st = in_st;
75   mpi_comm = in_mpi_comm;
76   solution = NULL;
77   is_stored = false;
78   solver_control.log_frequency(10);
79   const int number = Layers - 1;
80   qualities = new double[number];
81   execute_recomputation = false;
82   mkdir((solutionpath + "/" + "primal").c_str(), ACCESSPERMS);
83   mkdir((solutionpath + "/" + "dual").c_str(), ACCESSPERMS);
84 }
```

4.32.3 Member Function Documentation

4.32.3.1 assemble_part()

```
void Waveguide::assemble_part ( )
```

The assemble_part function is a part of the assemble_system() functionality.

It builds a part of the system-matrix. assemble_system() creates the global system matrix. After splitting the degrees of freedom into several blocks, this method takes one block (identified by the integer passed as an argument) and calculates all matrix-entries that reference it.

Date

25.10.2018

Author

Pascal Kraft

4.32.3.2 estimate_solution()

```
void Waveguide::estimate_solution ( )
```

This function is currently not in use.

It is supposed to create a useful input-vector for the first step of the iteration. However currently this is not used, since current cases simply use a zero-vector for the first step and previous solutions in the subsequent steps.

Definition at line 112 of file Waveguide.cpp.

References MeshGenerator::math_coordinate_in_waveguide(), MeshGenerator::prepare_triangulation(), and ExactSolution::vector_value().

```
112                                {
113   MPI_Barrier(mpi_comm);
114   deallog.push("estimate_solution");
115   deallog << "Starting solution estimation..." << std::endl;
116   DoFHandler<3>::active_cell_iterator cell, endc;
117   deallog << "Lambda: " << GlobalParams.M_W_Lambda << std::endl;
118   unsigned int min_dof = locally_owned_dofs.nth_index_in_set(0);
119   unsigned int max_dof =
120       locally_owned_dofs.nth_index_in_set(locally_owned_dofs.n_elements() - 1);
121   cell = dof_handler.begin_active(), endc = dof_handler.end();
122   for (; cell != endc; ++cell) {
123     if (cell->is_locally_owned()) {
124       for (unsigned int i = 0; i < GeometryInfo<3>::faces_per_cell; i++) {
125         std::vector<types::global_dof_index> local_dof_indices(
126             fe.dofs_per_line);
127         for (unsigned int j = 0; j < GeometryInfo<3>::lines_per_face; j++) {
128           ((cell->face(i))->line(j))->get_dof_indices(local_dof_indices);
129           Tensor<1, 3, double> ptemp =
130               ((cell->face(i))->line(j))->center(true, false);
131           if (std::abs(ptemp[2] - GlobalParams.Minimum_Z) > 0.0001) {
132             Point<3, double> p(ptemp[0], ptemp[1], ptemp[2]);
133             Tensor<1, 3, double> dtemp = ((cell->face(i))->line(j))->vertex(0) -
134                                          ((cell->face(i))->line(j))->vertex(1);
135             dtemp = dtemp / dtemp.norm();
136             Point<3, double> direction(dtemp[0], dtemp[1], dtemp[2]);
137             Vector<double> val(6);
138             es.vector_value(p, val);
139             double a = direction(0) * val(0) + direction(1) * val(1) +
140                        direction(2) * val(2);
141             double b = direction(0) * val(3) + direction(1) * val(4) +
142                        direction(2) * val(5);
```

```
143             if (local_dof_indices[0] >= min_dof &&
144                 local_dof_indices[0] < max_dof) {
145               EstimatedSolution[local_dof_indices[0]] = a;
146             }
147             if (local_dof_indices[1] >= min_dof &&
148                 local_dof_indices[1] < max_dof) {
149               EstimatedSolution[local_dof_indices[1]] = b;
150             }
151           }
152         }
153       }
154     }
155   }
156   MPI_Barrier(mpi_comm);
157   EstimatedSolution.compress(VectorOperation::insert);
158   deallog << "Done." << std::endl;
159   deallog.pop();
160 }
```

4.32.3.3 evaluate()

```
void Waveguide::evaluate ( )
```

This function has the purpose of filling the qualities array in every process with the appropriate Values from the other ones.

Now it will become necessary to build an optimization-scheme ontop, which can handle this information on process one and then distribute a new shape to the others. The function will use the Waveguide-Property execute_rebuild to signal a need for recomputation.

4.32.3.4 evaluate_for_Position()

```
std::complex< double > Waveguide::evaluate_for_Position (
              double x,
              double y,
              double z )
```

To compute the output quality of the signal and it's transmition along the waveguid-axis, this function performs a comparison of the fundamental mode of a waveguide and the actual situation.

For this purpose we integrate the product of the two functions over a cross-section of the waveguide in transformed coordinates. To perform this action we need to use numeric integration so the integral is decomposed into a sum over local evaluations. For this to be possible this function can be handed x,y and z coordinates and returns the according value.

Parameters

x	gives the x-coordinate.
y	gives the y-coordinate.
z	gives the z-coordinate.

Definition at line 88 of file Waveguide.cpp.

References ExactSolution::vector_value().

Referenced by InhomogenousTransformationCircular::PML_Z_Distance(), HomogenousTransformationCircular::↩ PML_Z_Distance(), and SpaceTransformation::Z_to_Layer().

```
89                                                                    {
90   dealii::Point<3, double> position(x, y, z);
91   Vector<double> result(6);
92   Vector<double> mode(6);
93   if (primal) {
94     VectorTools::point_value(dof_handler, primal_with_relevant, position,
95                              result);
96   } else {
97     VectorTools::point_value(dof_handler, primal_solution, position, result);
98   }
99   position[2] = GlobalParams.Minimum_Z;
100   this->es.vector_value(position, mode);
101
102   std::complex<double> c1(result(0), result(3));
103   std::complex<double> c2(result(1), result(4));
104   std::complex<double> c3(result(2), result(5));
105   std::complex<double> m1(mode(0), mode(3));
106   std::complex<double> m2(mode(1), mode(4));
107   std::complex<double> m3(mode(2), mode(5));
108
109   return m1 * c1 + m2 * c2 + m3 * c3;
110 }
```

4.32.3.5 evaluate_for_z()

```
double Waveguide::evaluate_for_z (
              double  )
```

To compute the output quality of the signal and it's transmition along the waveguid-axis, this function performs a comparison of the fundamental mode of a waveguide and the actual situation.

For this purpose we integrate the product of the two functions over a cross-section of the waveguide in transformed coordinates. To perform this action we need to use numeric integration so the integral is decomposed into a sum over local evaluations. For this to be possible this function can be handed x,y and z coordinates and returns the according value.

Parameters

x	gives the x-coordinate.
y	gives the y-coordinate.
z	gives the z-coordinate.

4.32.3.6 run()

```
void Waveguide::run ( )
```

This method as well as the rerun() method, are used by the optimization-algorithm to use and reuse the Waveguide-object.

Since the system-matrix consumes a lot of memory it makes sense to reuse it, rather then creating a new one for every optimization step. All properties of position[2]the object have to be created properly for this function to work.

Definition at line 1630 of file Waveguide.cpp.

References SpaceTransformation::IsDofFree(), and SpaceTransformation::NDofs().

```
1630                    {
1631   deallog.push("Waveguide_" + path_prefix + "_run");
1632
1633   if (run_number == 0) {
1634     deallog << "Setting up the mesh..." << std::endl;
1635     timer.enter_subsection("Setup Mesh");
1636     make_grid();
1637     timer.leave_subsection();
1638
1639     Compute_Dof_Numbers();
1640
1641     deallog << "Setting up FEM..." << std::endl;
1642     timer.enter_subsection("Setup FEM");
1643     setup_system();
1644     timer.leave_subsection();
1645
1646     timer.enter_subsection("Reset");
1647     timer.leave_subsection();
1648
1649   } else {
1650     Prepare_Boundary_Constraints();
1651
1652     timer.enter_subsection("Reset");
1653     reinit_all();
1654     timer.leave_subsection();
1655   }
1656
1657   deallog.push("Assembly");
1658   deallog << "Assembling the system..." << std::endl;
1659   timer.enter_subsection("Assemble");
1660   assemble_system();
1661   timer.leave_subsection();
1662   deallog.pop();
1663
1664   deallog.push("Solving");
1665   deallog << "Solving the system..." << std::endl;
1666   timer.enter_subsection("Solve");
1667   solve();
1668   timer.leave_subsection();
1669   deallog.pop();
1670
1671   timer.enter_subsection("Evaluate");
1672   timer.leave_subsection();
1673
1674   timer.print_summary();
1675
1676   deallog << "Writing outputs..." << std::endl;
1677   timer.reset();
1678
1679   output_results(false);
1680
1681   deallog.pop();
1682   run_number++;
1683 }
```

4.32.3.7 store()

```
void Waveguide::store ( )
```

The storage has the following purpose: Regarding the optimization-process there are two kinds of runs.

The first one, taking place with no knowledge of appropriate starting values for the degrees of freedom, and the following steps, in which the prior results can be used to estimate appropriate starting values for iterative solvers as well as the preconditioner. This function switches the behaviour in the following way: Once it is called, it stores the current solution in a run-independent variable, making it available for later runs. Also it sets a flag, indicating, that prior solutions are now available for usage in the solution process.

Definition at line 1457 of file Waveguide.cpp.

```
1457                       {
1458   reinit_storage();
1459   // storage.reinit(dof_handler.n_dofs());
1460   if (primal) {
1461     storage = primal_solution;
1462   } else {
1463     storage = dual_solution;
1464   }
1465   is_stored = true;
1466 }
```

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

- Code/Core/Waveguide.h
- Code/Core/Waveguide.cpp

Index