Software Dokumentation

Beschreibung der Funktionen und Parameter

amr-Paket der FEDDLib

Implementierung einer parallelen adaptiven Gitterverfeinerung

Generated by Doxygen 1.8.13

Contents

Chapter 1

Class Documentation

1.1 FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO > Class Template Reference

Public Member Functions

- AdaptiveMeshRefinement (string problemType, ParameterListPtr_Type parameterListAll)

  Initializing problem with the kind of problem we are solving for determining the correct error estimation. ParameterListAll delivers all necessary information (i.e. dim, feType). This constructor is used if no exact solutions are known.
- AdaptiveMeshRefinement (string problemType, ParameterListPtr_Type parameterListAll, Func_Type exactSolFunc)

  Initializing problem with the kind of problem we are solving for determining the correct error estimation. ParameterListAll delivers all necessary information (i.e. dim, feType).
- AdaptiveMeshRefinement (string problemType, ParameterListPtr_Type parameterListAll, Func_Type exactSolFuncU, Func_Type exactSolFuncP)

  Initializing problem with the kind of problem we are solving for determining the correct error estimation. ParameterListAll delivers all necessary information (i.e. dim, feType).
- DomainPtr_Type globalAlgorithm (DomainPtr_Type domainP1, DomainPtr_Type domainP12, BlockMultiVectorConstPtr_Type solution, ProblemPtr_Type problem, RhsFunc_Type rhsFunc)

  Global Algorithm of Mesh Refinement.
- DomainPtr_Type refineArea (DomainPtr_Type domainP1, vec2D_dbl_Type area, int level)

  Initializing problem if only a certain area should be refined.
- MultiVectorConstPtr_Type calcExactSolution ()

  Calculating exact solution for velocity if possible with exactSolFunc_.
- MultiVectorConstPtr_Type calcExactSolutionP ()

  Calculating exact solution for pressure if possible with exactSolPFunc_.
- void identifyProblem (BlockMultiVectorConstPtr_Type valuesSolution)

Identifying the problem with respect to the degrees of freedom and whether we calculate pressure. By telling how many blocks the MultiVector has, we can tell whether we calculate pressure or not. Depending on the numbers of entries within the solution vector, we can tell how many degreesOfFreedom (dofs) we have.

- void calcErrorNorms (MultiVectorConstPtr_Type exactSolution, MultiVectorConstPtr_Type solutionP12, MultiVectorConstPtr_Type exactSolutionP)

  Calculating error norms. If the exact solution is unknown we use approxmated error norm and error indicators.

- void initExporter (ParameterListPtr_Type parameterListAll)

  ParaViewExporter initiation. ParameterListAll contains most settings for ExporterParaView. ExporterParaViewAMR is an extension of ExportParaView and as such uses its setup.

- void exportSolution (MeshUnstrPtr_Type mesh, MultiVectorConstPtr_Type exportSolutionMv, MultiVectorConstPtr_Type errorValues, MultiVectorConstPtr_Type exactSolutionMv, MultiVectorConstPtr_Type exportSolutionPMv, MultiVectorConstPtr_Type exactSolutionPMv)

  ParaViewExporter export of solutions and other error values on current mesh.

- void exportError (MeshUnstrPtr_Type mesh, MultiVectorConstPtr_Type errorElConst, MultiVectorConstPtr_Type errorElConstH1, MultiVectorConstPtr_Type difH1Eta, MultiVectorConstPtr_Type vecDecompositionConst)

  ParaViewExporter export of solutions and other error values on current mesh.

- void writeRefinementInfo ()

  Writing refinement information at the end of mesh refinement.

1.1.1 Constructor & Destructor Documentation

1.1.1.1 AdaptiveMeshRefinement() [1/3]

```
template<class SC , class LO , class GO , class NO >
FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::AdaptiveMeshRefinement
(
            string problemType,
            ParameterListPtr_Type parameterListAll )
```

Initializing problem with the kind of problem we are solving for determining the correct error estimation. ParameterListAll delivers all necessary information (i.e. dim, feType). This constructor is used if no exact solutions are known.

Parameters

in	*problemType*	Laplace, Stokes, NavierStokes.
in	*parameterListAll*	Parameterlist as used as input parametersProblem.xml.

1.1.1.2 AdaptiveMeshRefinement() [2/3]

```
template<class SC , class LO , class GO , class NO >
FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::AdaptiveMeshRefinement
(
            string problemType,
            ParameterListPtr_Type parameterListAll,
            Func_Type exactSolFunc )
```

Initializing problem with the kind of problem we are solving for determining the correct error estimation. ParameterListAll delivers all necessary information (i.e. dim, feType).

Parameters

in	*problemType*	Laplace, Stokes, NavierStokes.
in	*paramerterListAll*	Parameterlist as used as input parametersProblem.xml.
in	*exactSolFun*	Exact solution function.

1.1.1.3 AdaptiveMeshRefinement() [3/3]

```
template<class SC , class LO , class GO , class NO >
FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::AdaptiveMeshRefinement
(
            string problemType,
            ParameterListPtr_Type parameterListAll,
            Func_Type exactSolFuncU,
            Func_Type exactSolFuncP )
```

Initializing problem with the kind of problem we are solving for determining the correct error estimation. ParameterListAll delivers all necessary information (i.e. dim, feType).

Parameters

in	*problemType*	Laplace, Stokes, NavierStokes.
in	*paramerterListAll*	Parameterlist as used as input parametersProblem.xml.
in	*exactSolFunU*	Exact solution for velocity u.
in	*exactSolFunP*	Exact solution for velocity p.

1.1.2 Member Function Documentation

1.1.2.1 calcErrorNorms()

```
template<class SC , class LO , class GO , class NO >
void FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::calcErrorNorms (
```

```
            MultiVectorConstPtr_Type exactSolution,
            MultiVectorConstPtr_Type solutionP12,
            MultiVectorConstPtr_Type exactSolutionP )
```

Calculating error norms. If the exact solution is unknown we use approxmated error norm and error indicators.

Parameters

in	*exactSolution*	If known, otherwise a dummy vector with all zeros is input.
in	*solutionP12*	Finite element solution u_h and maybe p_h.
in	*exactSolutionP*	If known, otherwise a vector with all zeros is input.

1.1.2.2 exportError()

```
template<class SC , class LO , class GO , class NO >
void FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::exportError (
            MeshUnstrPtr_Type mesh,
            MultiVectorConstPtr_Type errorElConst,
            MultiVectorConstPtr_Type errorElConstH1,
            MultiVectorConstPtr_Type difH1Eta,
            MultiVectorConstPtr_Type vecDecompositionConst )
```

ParaViewExporter export of solutions and other error values on current mesh.

Parameters

in	*mesh*	The current mesh on which refinement is performed.
in	*errorElConst*	Estimated error eta_T.
in	*errorElConstH1*	H1 error elmentwise.
in	*difH1Eta*	|eta_T-| u_h|_T| difference between estimated error an H1-error.
in	*vecDecompositionConst*	Information of distribution of elements among processors.

1.1.2.3 exportSolution()

```
template<class SC , class LO , class GO , class NO >
void FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::exportSolution (
            MeshUnstrPtr_Type mesh,
            MultiVectorConstPtr_Type exportSolutionMv,
            MultiVectorConstPtr_Type errorValues,
            MultiVectorConstPtr_Type exactSolutionMv,
```

```
            MultiVectorConstPtr_Type exportSolutionPMv,
            MultiVectorConstPtr_Type exactSolutionPMv )
```

ParaViewExporter export of solutions and other error values on current mesh.

Parameters

in	*mesh*	The current mesh on which refinement is performed.
in	*exportSolutionMv*	Export vector of velocity fe solution.
in	*exportSolutionPMv*	Export vector of pressure fe solution.
in	*errorValue*	|u-u_h| on domainP12.
in	*exactSolutionMv*	Export vector of exact velocity solution.
in	*exactSolutionPMv*	Export vector of exact pressure solution.

1.1.2.4 globalAlgorithm()

```
template<class SC , class LO , class GO , class NO >
AdaptiveMeshRefinement< SC, LO, GO, NO >::DomainPtr_Type FEDD::Adaptive←
MeshRefinement< SC, LO, GO, NO >::globalAlgorithm (
            DomainPtr_Type domainP1,
            DomainPtr_Type domainP12,
            BlockMultiVectorConstPtr_Type solution,
            ProblemPtr_Type problem,
            RhsFunc_Type rhsFunc )
```

Global Algorithm of Mesh Refinement.

Given domains and solutions depending on problem a global mesh refinement algorithm and error estimation is performed. For example if to solve simple laplace problem, we have only one solution to put in, if to estimate error for Navier-Stokes equation we need pressure and velocity solutions.

Parameters

in	*domainP1*	Domain with P_1 discretization, always neccesary as refinement is performed on P_1 mesh.
in	*domainP12*	Domain with P_1 or P_2 discretization if available, otherwise input domainP1.
in	*solution*	Solution of problem on P_1 or P_2 discretization, can contain velocity and pressure solution.
in	*problem*	The problem itself as the problemPtr. Contains Laplace, Stokes or Navier-Stokes problem.
in	*rhs*	Right hand side function from the pde system. Necessary for error estimation.

1.1.2.5 identifyProblem()

```
template<class SC , class LO , class GO , class NO >
void FEDD::AdaptiveMeshRefinement< SC, LO, GO, NO >::identifyProblem (
            BlockMultiVectorConstPtr_Type valuesSolution )
```

Identifying the problem with respect to the degrees of freedom and whether we calculate pressure. By telling how many blocks the MultiVector has, we can tell whether we calculate pressure or not. Depending on the numbers of entries within the solution vector, we can tell how many degreesOfFreedom (dofs) we have.

Parameters

in	*valuesSolution*	Block that contains solution for velocity and as the case may be pressure.

1.1.2.6 refineArea()

```
template<class SC , class LO , class GO , class NO >
AdaptiveMeshRefinement< SC, LO, GO, NO >::DomainPtr_Type FEDD::Adaptive←
MeshRefinement< SC, LO, GO, NO >::refineArea (
            DomainPtr_Type domainP1,
            vec2D_dbl_Type area,
            int level )
```

Initializing problem if only a certain area should be refined.

Parameters

in	*domainP1*	P_1 Domain
in	*area*	Area that is suppose to be refined. If is a vector defining the area as follows: row1:[x_0,x_1] x-limits, row2: [y_0,y_1] y-limits, row3: [z_0,z_1] z-limits.
in	*level*	intensity of refinement. Number of levels equals number of performed refinements.

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

- AdaptiveMeshRefinement_decl.hpp
- AdaptiveMeshRefinement_def.hpp

1.2 FEDD::ErrorEstimation< SC, LO, GO, NO > Class Template Reference

Public Member Functions

- ErrorEstimation (int dim, string problemType)

ErrorEstimation is initiated with the dimension and problemType, as it is necessary to fit errorEstimation to the problem at hand.

- MultiVectorPtr_Type estimateError (MeshUnstrPtr_Type inputMeshP12, MeshUnstrPtr_Type inputMeshP1, BlockMultiVectorConstPtr_Type valuesSolution, RhsFunc_Type rhsFunc, string FEType)

  Main Function for a posteriori error estimation.
- void identifyProblem (BlockMultiVectorConstPtr_Type valuesSolution)

  Identifying the problem with respect to the degrees of freedom and whether we calculate pressure. By telling how many block the MultiVector has, we can tell whether we calculate pressure or not. Depending on the numbers of entries within the solution vector, we can tell how many degreesOfFreedom (dofs) we have.
- void makeRepeatedSolution (BlockMultiVectorConstPtr_Type valuesSolution)

  We split the solution from the BlockMultiVector valuesSolution into one or more seperate blocks, where the blocks represent the different dimensions.
- vec3D_dbl_Type calcNPhi (string phiDerivative, int dofsSol, string FEType)

  Function that calculates the jump part for nabla u or p.
- vec_dbl_Type calculateJump ()

  Part of the error estimator that calculates the jump part of the estimation. What kind of jump is calculated depends on the problemType we have at hand.
- vec2D_dbl_Type gradPhi (int dim, int intFE, vec_dbl_Type &p)

  Calcutlating the gradient of phi depending on quad points p.
- vec_dbl_Type phi (int dim, int intFE, vec_dbl_Type &p)

  Calcutlating phi depending on quad points p.
- MultiVectorPtr_Type determineCoarseningError (MeshUnstrPtr_Type mesh_k, MeshUnstrPtr_Type mesh_k_m, MultiVectorPtr_Type errorElementMv_k, string distribution, string markingStrategy, double theta)

  DetermineCoarseningError is the essential part of the mesh coarsening process.
- double determineResElement (FiniteElement element, RhsFunc_Type rhsFunc)

  Function that that determines | | u_h + f | | _(L2(T)), | | u_h + f - p_h | | _T or | | u_h + f - p_h - (u_h)u_h| | _T for an Element T.
- double determineDivU (FiniteElement element)

  Function that that determines | | div(u) | | _T for a Element T.
- vec2D_dbl_Type getQuadValues (int dim, string FEType, string Type, vec_dbl_Type &QuadW, FiniteElement surface)

  Returns neccesary quadrature Values. Is distinguishes between needing Element or Surface information.
- void markElements (MultiVectorPtr_Type errorElementMv, double theta, string strategy, MeshUnstrPtr_Type meshUnstr)

  Function that marks the elements for refinement.
- vec_dbl_Type determineVolTet (ElementsPtr_Type elements, vec2D_dbl_ptr_Type points)

  function, that determines volume of tetrahedra.
- vec_dbl_Type calcDiamTriangles (ElementsPtr_Type elements, vec2D_dbl_ptr_Type points, vec_dbl_Type &areaTriangles, vec_dbl_Type &rho_T, vec_dbl_Type &C_T)

  Calculating the diameter of triangles.

- vec_dbl_Type calcDiamTriangles3D (SurfaceElementsPtr_Type surface↩ TriangleElements, vec2D_dbl_ptr_Type points, vec_dbl_Type &areaTriangles, vec_dbl_Type &rho_T, vec_dbl_Type &C_T)

  Calculating the diameter of triangles.
- vec_dbl_Type calcDiamTetraeder (ElementsPtr_Type elements, vec2D_dbl_ptr↩ _Type points, vec_dbl_Type volTet)

  Calculating the circumdiameter of tetraeder.
- vec_dbl_Type calcRhoTetraeder (ElementsPtr_Type elements, Surface↩ ElementsPtr_Type surfaceTriangleElements, vec_dbl_Type volTet, vec_dbl_Type areaTriangles)

  Calcutlating the incircumdiameter of tetrahedra.
- vec_dbl_Type determineAreaTriangles (ElementsPtr_Type elements, Edge↩ ElementsPtr_Type edgeElements, SurfaceElementsPtr_Type surfaceElements, vec2D_dbl_ptr_Type points)

  Calculating the area of the triangle elements of tetrahedra.
- void buildTriangleMap ()

  Build Surface Map. Contrary to building the edge map, building the surface map is somewhat simpler as elementsOfSurfaceGlobal and elementsOfSurfaceLocal already exist. Via elementsOfSurface global each surface can be uniquely determined by the two elements it connects.
- void updateElementsOfSurfaceLocalAndGlobal (EdgeElementsPtr_Type edge↩ Elements, SurfaceElementsPtr_Type surfaceTriangleElements)

  UpdateElementsOfSurfaceLocalAndGlobal is performed here instead of in mesh↩ Refinement, as the information is only needed in case of error estimation. Equivalent function as updateElementsOfEdgeGlobal but with the important destinction, that it runs without communication and only relies on local information.
- void tagArea (MeshUnstrPtr_Type meshUnstr, vec2D_dbl_Type area)

  Tags only a certain Area for refinement and is independent of any error estimation.

1.2.1 Constructor & Destructor Documentation

1.2.1.1 ErrorEstimation()

```
template<class SC , class LO , class GO , class NO >
FEDD::ErrorEstimation< SC, LO, GO, NO >::ErrorEstimation (
            int dim,
            string problemType )
```

ErrorEstimation is initiated with the dimension and problemType, as it is necessary to fit errorEstimation to the problem at hand.

Parameters

in	*dim*	Dimension of problem.
in	*problemType*	Type of problem. Choose between Laplace, Stokes and NavierStokes

1.2.2 Member Function Documentation

1.2.2.1 buildTriangleMap()

```
template<class SC , class LO , class GO , class NO >
void FEDD::ErrorEstimation< SC, LO, GO, NO >::buildTriangleMap ( )
```

Build Surface Map. Contrary to building the edge map, building the surface map is somewhat simpler as elementsOfSurfaceGlobal and elementsOfSurfaceLocal already exist. Via elementsOfSurface global each surface can be uniquely determined by the two elements it connects.

The surfacemap is only used for error estimation. Thus it is only build here and not in the refinementFactory.

1.2.2.2 calcDiamTetraeder()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::calcDiamTetraeder
(
            ElementsPtr_Type elements,
            vec2D_dbl_ptr_Type points,
            vec_dbl_Type volTet )
```

Calculating the circumdiameter of tetraeder.

Parameters

in	*elements*	Elements.
in	*points*	Points.
in	*volTet*	Volume of tetrahedra.
out	*diamElements*	Uncircumdiameter of tetrahedra.

1.2.2.3 calcDiamTriangles()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::calcDiamTriangles
(
            ElementsPtr_Type elements,
            vec2D_dbl_ptr_Type points,
            vec_dbl_Type & areaTriangles,
            vec_dbl_Type & rho_T,
            vec_dbl_Type & C_T )
```

Calculating the diameter of triangles.

Parameters

in	*elements*	Elements
in	*points*	Points
out	*diamElements*	Diameter of triangles (Uncirclediameter).
out	*rho_T*	Incirclediameter.
out	*C_T*	Shape parameter.

1.2.2.4 calcDiamTriangles3D()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::calcDiamTriangles3D
(
            SurfaceElementsPtr_Type surfaceTriangleElements,
            vec2D_dbl_ptr_Type points,
            vec_dbl_Type & areaTriangles,
            vec_dbl_Type & rho_T,
            vec_dbl_Type & C_T )
```

Calculating the diameter of triangles.

Parameters

in	*elements*	Elements.
in	*points*	Points.
out	*diamElements*	Diameter of triangles.

1.2.2.5 calcNPhi()

```
template<class SC , class LO , class GO , class NO >
vec3D_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::calcNPhi (
            string phiDerivative,
            int dofsSol,
            string FEType )
```

Function that calculates the jump part for nabla u or p.

Parameters

in	*phiDerivative*	phiDerivative is either 'Gradient' or 'None' and what kind of jump is calculated depends on the problemType we have at hand. If phiDerivative is 'Gradient' the nabla u jump part is caluculated and if its 'None' then the pressure jump.

Parameters

in	*dofsSol*	Degree of freedom of the caluclated jump part. The pressure dof is always 1 whereas velocity dof can vary depending on the problem.
in	*FEType*	Finite element type of the calculated jump part.

1.2.2.6 calcRhoTetraeder()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::calcRhoTetraeder
(
            ElementsPtr_Type elements,
            SurfaceElementsPtr_Type surfaceTriangleElements,
            vec_dbl_Type volTet,
            vec_dbl_Type areaTriangles )
```

Calcutlating the incircumdiameter of tetrahedra.

Parameters

in	*elements*	Elements.
in	*surfaceTriangleElements*	TriangleElements.
in	*volTet*	Volume of tetrahedra.
in	*areaTriangles*	Area of faces of tetrahedra.
out	*rhoElements*	Incircumdiameter of tetrahedra.

1.2.2.7 determineAreaTriangles()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::determineArea↩
Triangles (
            ElementsPtr_Type elements,
            EdgeElementsPtr_Type edgeElements,
            SurfaceElementsPtr_Type surfaceElements,
            vec2D_dbl_ptr_Type points )
```

Calculating the area of the triangle elements of tetrahedra.

Parameters

in	*elements*	Elements.
in	*edgeElements*	Edges.
in	*surfaceElements*	Triangle Elements.
in	*points*	Points.
out	*areaTriangles*	Area of triangles.

1.2.2.8 determineCoarseningError()

```
template<class SC , class LO , class GO , class NO >
ErrorEstimation< SC, LO, GO, NO >::MultiVectorPtr_Type FEDD::Error←
Estimation< SC, LO, GO, NO >::determineCoarseningError (
            MeshUnstrPtr_Type mesh_k,
            MeshUnstrPtr_Type mesh_k_m,
            MultiVectorPtr_Type errorElementMv_k,
            string distribution,
            string markingStrategy,
            double theta )
```

DetermineCoarseningError is the essential part of the mesh coarsening process.

Instead of calulating an error for mesh level k, we redestribute it to lower mesh levels and refining those. We execute this function with an estimated error from level k. With this calculated error, we mark the elements according to that error and refine afterwards. If we decide to coarsen a certain mesh level, we take that level, look at the k-m level and refine that to the point where we are at the same level we wanted to perform the coarsening on.

Parameters

in	*mesh_k*	Current mesh of level k.
in	*mesh_k_m*	Mesh of refinement level k-m.
in	*errorElementMv←_k*	The error estimation of mesh level k.
in	*distribution*	Either 'forwards' or 'backwards'. We determine the error estimate in level k-m with redistributing backwards. if we are in level k-m we calculate the k-m+1 mesh level error estimation via redistributing the k-m error forward.
in	*markingStrategy*	The strategy with which element are marked.
in	*theta*	Marking threshold.

1.2.2.9 determineDivU()

```
template<class SC , class LO , class GO , class NO >
double FEDD::ErrorEstimation< SC, LO, GO, NO >::determineDivU (
            FiniteElement element )
```

Function that that determines | | div(u) | |_T for a Element T.

Parameters

in	*element*	FiniteElement element where | |div(u)| |_T is calculated on.
out	*divElement*	Divergence of u_h on element

1.2.2.10 determineResElement()

```
template<class SC , class LO , class GO , class NO >
double FEDD::ErrorEstimation< SC, LO, GO, NO >::determineResElement (
            FiniteElement element,
            RhsFunc_Type rhsFunc )
```

Function that that determines | | u_h + f | | _(L2(T)), | | u_h + f - p_h | | _T or | | u_h + f - p_h - (u_h)u_h| | _T for an Element T.

Parameters

in	*element*	FiniteElement element where | |div(u)| |_T is calculated on.
in	*rhsFunc*	The right hand side function of the pde.
out	*resElement*	Residual of element according to problemType

1.2.2.11 determineVolTet()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::determineVolTet (
            ElementsPtr_Type elements,
            vec2D_dbl_ptr_Type points )
```

function, that determines volume of tetrahedra.

Parameters

in	*elements*	Elements.
in	*edgeElements*	Edges.
in	*points*	Points.
out	*volumeTetrahedra*	Volume of tetrahedras

1.2.2.12 estimateError()

```
template<class SC , class LO , class GO , class NO >
ErrorEstimation< SC, LO, GO, NO >::MultiVectorPtr_Type FEDD::Error↩
Estimation< SC, LO, GO, NO >::estimateError (
            MeshUnstrPtr_Type inputMeshP12,
            MeshUnstrPtr_Type inputMeshP1,
            BlockMultiVectorConstPtr_Type valuesSolution,
            RhsFunc_Type rhsFunc,
            string FETypeV )
```

Main Function for a posteriori error estimation. Depending on the problem the the error estimation is calculated accordingly.

Parameters

in	*inputMeshP1*	The P1 Mesh that is used for later refinement.
in	*inputMeshP12*	The possible P2 Mesh, if one of the solutions is of P2 Discretisation, otherwise both meshes are P1.
in	*solution*	Solution of the PDE in BlockMultiVector Format (Block 0: Velocity, Block 1: Pressure).
in	*rhs*	The right hand side function of the pde.
in	*FETypeV*	Finite element type as the maximum FEType for the Velocity, pressure is assumed to be P1 always.

1.2.2.13 getQuadValues()

```
template<class SC , class LO , class GO , class NO >
vec2D_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::getQuadValues (
            int dim,
            string FEType,
            string Type,
            vec_dbl_Type & QuadW,
            FiniteElement surface )
```

Returns neccesary quadrature Values. Is distinguishes between needing Element or Surface information.

Parameters

in	*dim*	Dimension for which the quadrature points are needed.
in	*FEType*	Finite element type for which the quadrature points are needed.
in	*Type*	Type of quadrature points are need. Either 'Element' if you integrate over an element or 'Surface' if you need to integrate over a surface (i.e. for calculating the jump)
in	*QuadW*	Vector to be filled with the quadrature weights accordingly
in	*FiniteElement*	surface for which you need the quadrature points in case if 'Surface' type, as it is needed for figuring out the quadrature points
out	*QuadPts*	Quadrature points
out	*QuadW*	Quadrature weights Keep in mind that elementwise quadPoint are defined on reference element whereas surface quadPoints are defined on the input surface, which is typically not the reference Element.

1.2.2.14 gradPhi()

```
template<class SC , class LO , class GO , class NO >
vec2D_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::gradPhi (
            int dim,
            int intFE,
            vec_dbl_Type & p )
```

Calcutlating the gradient of phi depending on quad points p.

Parameters

in	*dim*	Dimension.
in	*intFE*	Integer value of discretisation P1 (1) or P2(2).
in	*p*	Quadpoints or other points for which phi is suppose to be evaluated.
out	*gradPhi*	Gradient of phi with evaluated values p.

1.2.2.15 identifyProblem()

```
template<class SC , class LO , class GO , class NO >
void FEDD::ErrorEstimation< SC, LO, GO, NO >::identifyProblem (
            BlockMultiVectorConstPtr_Type valuesSolution )
```

Identifying the problem with respect to the degrees of freedom and whether we calculate pressure. By telling how many block the MultiVector has, we can tell whether we calculate pressure or not. Depending on the numbers of entries within the solution vector, we can tell how many degreesOfFreedom (dofs) we have.

Parameters

in	*valuesSolution*	Blocks that contains solution for velocity and as the case may be pressure.

1.2.2.16 makeRepeatedSolution()

```
template<class SC , class LO , class GO , class NO >
void FEDD::ErrorEstimation< SC, LO, GO, NO >::makeRepeatedSolution (
            BlockMultiVectorConstPtr_Type valuesSolution )
```

We split the solution from the BlockMultiVector valuesSolution into one or more seperate blocks, where the blocks represent the different dimensions.

Parameters

in	*valuesSolution*	Block that contains solution for velocity and as the case may be pressure.

1.2.2.17 markElements()

```
template<class SC , class LO , class GO , class NO >
void FEDD::ErrorEstimation< SC, LO, GO, NO >::markElements (
            MultiVectorPtr_Type errorElementMv,
            double theta,
            string strategy,
            MeshUnstrPtr_Type meshP1 )
```

Function that marks the elements for refinement.

Parameters

in	*errorElementMv*	MultiVector that contains the estimated error for each element.
in	*theta*	Parameter determining for marking strategies.
in	*markingStrategy*	Strategy with which the elements are marked. Implemented Strategies 'Doerfler' or 'Maximum'.
in	*meshP1*	P1 mesh which is used for later refinement and has to be the one beeing marked.

!! it is essential that the meshP1 mesh inserted here is the mesh that will be used for mesh refinement, as it contains the elementwise-information determining refinement. !!

1.2.2.18 phi()

```
template<class SC , class LO , class GO , class NO >
vec_dbl_Type FEDD::ErrorEstimation< SC, LO, GO, NO >::phi (
            int dim,
            int intFE,
            vec_dbl_Type & p )
```

Calcutlating phi depending on quad points p.

Parameters

in	*dim.*	
in	*intFE*	integer value of discretisation P1 (1) or P2(2).
in	*p*	Quadpoints or other points for which phi is to be evaluated.
out	*phi*	Phi with evaluated values p.

1.2.2.19 tagArea()

```
template<class SC , class LO , class GO , class NO >
void FEDD::ErrorEstimation< SC, LO, GO, NO >::tagArea (
            MeshUnstrPtr_Type inputMeshP1,
            vec2D_dbl_Type area )
```

Tags only a certain Area for refinement and is independent of any error estimation.

Parameters

in	*inputMeshP1*	The P1 Mesh that is used for later refinement.
in	*area*	Area that is suppose to be refined. If is a vector defining the area as follows: row1:[x_0,x_1] x-limits, row2: [y_0,y_1] y-limits, row3: [z_0,z_1] z-limits .

1.2.2.20 updateElementsOfSurfaceLocalAndGlobal()

```
template<class SC , class LO , class GO , class NO >
void FEDD::ErrorEstimation< SC, LO, GO, NO >::updateElementsOfSurface↩
LocalAndGlobal (
            EdgeElementsPtr_Type edgeElements,
            SurfaceElementsPtr_Type surfaceTriangleElements )
```

UpdateElementsOfSurfaceLocalAndGlobal is performed here instead of in mesh↩ Refinement, as the information is only needed in case of error estimation. Equivalent function as updateElementsOfEdgeGlobal but with the important destinction, that it runs without communication and only relies on local information.

Parameters

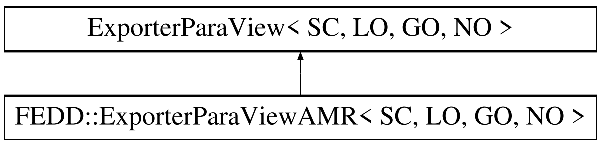
in	*edgeElements*	Edes.
in	*surfaceTriangleElements*	Triangle elements.

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

- ErrorEstimation_decl.hpp
- ErrorEstimation_def.hpp

1.3 FEDD::ExporterParaViewAMR< SC, LO, GO, NO > Class Template Reference

Inheritance diagram for FEDD::ExporterParaViewAMR< SC, LO, GO, NO >:

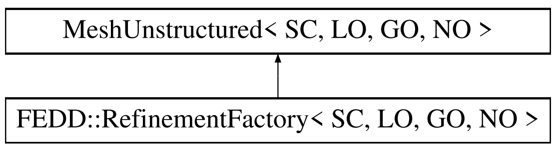


The documentation for this class was generated from the following files:

- ExporterParaViewAMR_decl.hpp
- ExporterParaViewAMR_def.hpp

1.4 FEDD::RefinementFactory< SC, LO, GO, NO > Class Template Reference

Inheritance diagram for FEDD::RefinementFactory< SC, LO, GO, NO >:



Public Member Functions

- RefinementFactory (CommConstPtr_Type comm, int volumeID=10)

  Initiating RefinementFactory via MeshUnstructured.
- RefinementFactory (CommConstPtr_Type comm, int volumeID, string refinement↩ Restriction, int refinement3DDiagonal=0, int restrictionLayer_=2)

  Initiating RefinementFactory via MeshUnstructured with additional information for mesh refinement.
- void refineMesh (MeshUnstrPtr_Type meshP1, int iteration, MeshUnstrPtr_Type outputMesh, string refinementMode)

  Main function of RefinementFactory, performs one complete mesh refinement, according to red-green refinement (Verfuerth) or tetrahedral grid refinement (Bey).
- void assignEdgeFlags (MeshUnstrPtr_Type meshP1, EdgeElementsPtr_Type edgeElements)

  Not all edges are marked with a flag in the beginning. In order to set the correct flags to new points we assign the edge flag of the edge they originated from, similar to the function determineEdgeFlagP2New, but this function uses the edgeMap.
- void refineRegular (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int i, SurfaceElementsPtr_Type surfaceTriangleElements)

  2D and 3D regular refinement. Chosen by error estimator or otherwise elements are refined regular by connecting edge midpoints.
- void refineGreen (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int i)

  2D green refinement: refining the element according to green scheme - connecting node on refined edge with the opposite node.
- void refineBlue (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int i)

  2D blue refinement: refining element according to blue refinement scheme - connecting nodes of shorter edge with midpoint of longer tagged edge and connect that with opposite corner
- void refineRed (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int i)

  2D red refinement: refining the element red by connecting all tagged edges midpoints. one element is refined into 4.
- void refineType1 (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int indexElement, SurfaceElementsPtr_Type surfaceTriangleElements)

  3D Type(1) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955
- void refineType2 (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int indexElement, SurfaceElementsPtr_Type surfaceTriangleElements)

  3D Type(2) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955
- void refineType3 (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int indexElement, SurfaceElementsPtr_Type surfaceTriangleElements)

  3D Type(3) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955
- void refineType4 (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int indexElement, SurfaceElementsPtr_Type surfaceTriangleElements)

3D Type(4) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955

- void addMidpoint (EdgeElementsPtr_Type edgeElements, int i)

  Adding a Midpoint on an edge.
- int determineLongestEdge (EdgeElementsPtr_Type edgeElements, vec_int_Type edgeVec, vec2D_dbl_ptr_Type points)

  Eetermine longest edge in triangle.
- void buildEdgeMap (MapConstPtr_Type mapGlobalProc, MapConstPtr_Type mapProc)

  Building edgeMap after refinement.
- void buildNodeMap (EdgeElementsPtr_Type edgeElements, MapConstPtr_Type mapGlobalProc, MapConstPtr_Type mapProc, int newPoints, int newPoints↩ Repeated)

  Building nodemap after refinement.
- void updateElementsOfEdgesLocalAndGlobal (int maxRank, MapConstPtr_Type edgeMap)

  Updating ElementsOfEdgesLocal and ElementsOfEdgesGlobal.
- vec_bool_Type checkInterfaceSurface (EdgeElementsPtr_Type edgeElements, vec_int_Type originFlag, vec_int_Type edgeNumbers, int indexElement)

  Checking if surfaces are part of the interface. Done by checking if all edges of a triangle are part of the interface and if both elements connected to the surface are on different processors.
- void refinementRestrictions (MeshUnstrPtr_Type meshP1, ElementsPtr_Type elements, EdgeElementsPtr_Type edgeElements, SurfaceElementsPtr_Type surfaceTriangleElements, int &newPoints, int &newPointsCommon, vec_GO↩ _Type &globalInterfaceIDsTagged, MapConstPtr_Type mapInterfaceEdges, int &newElements)

  Refinement Restrictions.
- void refineMeshRegIreg (ElementsPtr_Type elements, EdgeElementsPtr_Type edgeElements, int &newElements, MapConstPtr_Type edgeMap, Surface↩ ElementsPtr_Type surfaceTriangleElements)

  Refinement performed according to the set of rules determined by Bey or Verfürth.
- void buildSurfaceTriangleElements (ElementsPtr_Type elements, Edge↩ ElementsPtr_Type edgeElements, SurfaceElementsPtr_Type surfaceTriangle↩ Elements, MapConstPtr_Type edgeMap, MapConstPtr_Type elementMap)

  Building surface triangle elements, as they are not originally part of the mesh information provided by mesh partitioner.
- void bisectEdges (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int indexElement, SurfaceElementsPtr_Type surfaceTriangleElements, string mode="default")

  2D and 3D that bisects the edges of tagged Elements. Chosen by error estimator or otherwise elements are refined regular by connecting edge midpoints.
- void bisectElement3 (EdgeElementsPtr_Type edgeElements, ElementsPtr_Type elements, int indexElementp)

  2D refinement by bisection of tagged Elements with three tagged Edges.

1.4.1 Constructor & Destructor Documentation

1.4.1.1 RefinementFactory() [1/2]

```
template<class SC , class LO , class GO , class NO >
FEDD::RefinementFactory< SC, LO, GO, NO >::RefinementFactory (
            CommConstPtr_Type comm,
            int volumeID = 10 )
```

Initiating RefinementFactory via MeshUnstructured.

Parameters

in	*comm*	CommPtr.
in	*volumeID*	The flag ID of triangles in 2D or tetrahedra in 3D. Usually 10.

1.4.1.2 RefinementFactory() [2/2]

```
template<class SC , class LO , class GO , class NO >
FEDD::RefinementFactory< SC, LO, GO, NO >::RefinementFactory (
            CommConstPtr_Type comm,
            int volumeID,
            string refinementRestriction,
            int refinement3DDiagonal = 0,
            int restrictionLayer = 2 )
```

Initiating RefinementFactory via MeshUnstructured with additional information for mesh refinement.

Parameters

in	*comm*	CommPtr
in	*volumeID*	The flag ID of triangles in 2D or tetrahedra in 3D. Usually 10.
in	*refinementRestriction*	Restriction for repeated refinement steps.
in	*refinement3DDiagonal*	3D diagonal pick for regular refinement

1.4.2 Member Function Documentation

1.4.2.1 addMidpoint()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::addMidpoint (
            EdgeElementsPtr_Type edgeElements,
            int edgeID )
```

Adding a Midpoint on an edge.

Parameters

in	*edgeElements*	Edges.
in	*edgeID*	Edge ids where the midpoints is added.

1.4.2.2 assignEdgeFlags()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::assignEdgeFlags (
            MeshUnstrPtr_Type meshP1,
            EdgeElementsPtr_Type edgeElements )
```

Not all edges are marked with a flag in the beginning. In order to set the correct flags to new points we assign the edge flag of the edge they originated from, similar to the function determineEdgeFlagP2New, but this function uses the edgeMap.

Parameters

in	*meshP1*	InputMesh, always P1.
in	*edgeElements*	Edges that receive flags.

1.4.2.3 bisectEdges()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::bisectEdges (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement,
            SurfaceElementsPtr_Type surfaceTriangleElements,
            string mode = "default" )
```

2D and 3D that bisects the edges of tagged Elements. Chosen by error estimator or otherwise elements are refined regular by connecting edge midpoints.

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.4 bisectElement3()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::bisectElement3 (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement )
```

2D refinement by bisection of tagged Elements with three tagged Edges.

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.5 buildEdgeMap()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::buildEdgeMap (
            MapConstPtr_Type mapGlobalProc,
            MapConstPtr_Type mapProc )
```

Building edgeMap after refinement.

Parameters

in	*mapGlobalProc*	Map of global processor numbers
in	*mapProc*	Map of local processor number

1.4.2.6 buildNodeMap()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::buildNodeMap (
            EdgeElementsPtr_Type edgeElements,
            MapConstPtr_Type mapGlobalProc,
            MapConstPtr_Type mapProc,
            int newPoints,
            int newPointsRepeated )
```

Building nodemap after refinement.

Parameters

in	*edgeElements*	Edges.
in	*mapGlobalProc*	Map of global processor numbers.
in	*mapProc*	Map of local processor numbers.
in	*newPoints*	Number of new points per refinement iteration.
in	*newPointsRepeated*	Number of new repeated points per refinement iteration.

1.4.2.7 buildSurfaceTriangleElements()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::buildSurfaceTriangle↩
Elements (
            ElementsPtr_Type elements,
            EdgeElementsPtr_Type edgeElements,
            SurfaceElementsPtr_Type surfaceTriangleElements,
            MapConstPtr_Type edgeMap,
            MapConstPtr_Type elementMap )
```

Building surface triangle elements, as they are not originally part of the mesh information provided by mesh partitioner.

Parameters

in	*elements*	Elements.
in	*edgeElements*	Edges.
in	*surfaceTriangleElements*	Pointer which will be filled with surfaceTriangleElements.
in	*edgeMap*	Global Mapping of edges
in	*elementMap*	Global Mapping of elements.

1.4.2.8 checkInterfaceSurface()

```
template<class SC , class LO , class GO , class NO >
vec_bool_Type FEDD::RefinementFactory< SC, LO, GO, NO >::checkInterface↩
Surface (
            EdgeElementsPtr_Type edgeElements,
            vec_int_Type originFlag,
            vec_int_Type edgeNumbers,
            int indexElement )
```

Checking if surfaces are part of the interface. Done by checking if all edges of a triangle are part of the interface and if both elements connected to the surface are on different processors.

Parameters

in	*edgeElements*	Edges.
in	*originFlag*	Flags of surfaces of indexElement.
in	*edgeNumbers*	Numbers of inserted surfaces edges.
in	*indexElement*	Index of element in question.

1.4.2.9 determineLongestEdge()

```
template<class SC , class LO , class GO , class NO >
int FEDD::RefinementFactory< SC, LO, GO, NO >::determineLongestEdge (
            EdgeElementsPtr_Type edgeElements,
            vec_int_Type edgeVec,
            vec2D_dbl_ptr_Type points )
```

Eetermine longest edge in triangle.

Parameters

in	*edgeElements*	Edges.
in	*edgeVec*	Vector with edge ids of triangle.
in	*points*	Points.
out	*Local*	edgeID of the longest edge.

1.4.2.10 refineBlue()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineBlue (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement )
```

2D blue refinement: refining element according to blue refinement scheme - connecting nodes of shorter edge with midpoint of longer tagged edge and connect that with opposite corner

Parameters

in	*edgeElements*	Edges
in	*elements*	Elements.
in	*indexELement*	Element in question.

1.4.2.11 refineGreen()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineGreen (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement )
```

2D green refinement: refining the element according to green scheme - connecting node on refined edge with the opposite node.

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.

1.4.2.12 refinementRestrictions()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refinementRestrictions
(
            MeshUnstrPtr_Type meshP1,
            ElementsPtr_Type elements,
            EdgeElementsPtr_Type edgeElements,
            SurfaceElementsPtr_Type surfaceTriangleElements,
            int & newPoints,
            int & newPointsCommon,
            vec_GO_Type & globalInterfaceIDsTagged,
            MapConstPtr_Type mapInterfaceEdges,
            int & newElements )
```

Refinement Restrictions.

In 2D we can add some Restrictions to the Mesh Refinement: Bisection: this will keep the regularity of the Mesh by only refining whith a irregular strategy when the longest edge is involved. If not we add a node to the longest edge, whereby the irregular refinement strategy is changed. GreenTags: this will only check tagged green Elements, if its irregular refinement tag from the previous refinement is 'green' and if so not refine it green again but add a node to the longest edge and thus refine it blue. In the 3D Case we simply never refine an element irregularly twice, this strategy is called simply 'Bey'. If an element is refined regular, its refinement tag changes from eventually 'irregular' to regular. If those elements should still not be refined irregular we use the strategy 'BeyIrregular'.

Furthermore if there is no fitting irrregular refinement strategy (Type(1)-Type(4) don't fit) we refine regular instead.

Parameters

in	*meshP1*	P_1 Mesh.
in	*elements*	Element.
in	*edgeElements*	Edges.
in	*iteration*	Current iteration.
in	*newPoints*	Number of new unique points originating from restrictions.
in	*newPointsCommon*	Number of new repeated points originating from restrictions.
in	*globalInterfaceIDsTagged*	List of global IDs of tagged interface edges.
in	*mapInterfaceEdges*	Map of interface edges.
in	*restriction*	The kind of restriction we want to apply.
in	*newElements*	Number of new elements orginating from restrictions.

1.4.2.13 refineMesh()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineMesh (
            MeshUnstrPtr_Type meshP1,
            int iteration,
            MeshUnstrPtr_Type outputMesh,
            string refinementMode )
```

Main function of RefinementFactory, performs one complete mesh refinement, according to red-green refinement (Verfuerth) or tetrahedral grid refinement (Bey).

Parameters

in	*meshP1*	InputMesh P1.
in	*iteration*	Current Iteration.
in	*refinementMode*	In 2D we can choose between 'Regular' (red-green) or 'Bisection. In 3D only 'Regular' is possible, which is also the default mode.
out	*outputMesh*	Refined mesh.

1.4.2.14 refineMeshRegIreg()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineMeshRegIreg (
            ElementsPtr_Type elements,
```

```
EdgeElementsPtr_Type edgeElements,
int & newElements,
MapConstPtr_Type edgeMap,
SurfaceElementsPtr_Type surfaceTriangleElements )
```

Refinement performed according to the set of rules determined by Bey or Verfürth.

Parameters

in	*elements*	Elements.
in	*edgeElements*	Edges.
in	*newElements*	Number of new elements originating from refinement.
in	*edgeMap*	Map of global edge ids.
in	*surfaceTriangleElements*	Triangle elements (3D case).

1.4.2.15 refineRed()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineRed (
        EdgeElementsPtr_Type edgeElements,
        ElementsPtr_Type elements,
        int indexElement )
```

2D red refinement: refining the element red by connecting all tagged edges midpoints. one element is refined into 4.

Parameters

in	*edgeElements*	Edges
in	*elements*	Elements.
in	*indexELement*	Element in question.

1.4.2.16 refineRegular()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineRegular (
        EdgeElementsPtr_Type edgeElements,
        ElementsPtr_Type elements,
        int indexElement,
        SurfaceElementsPtr_Type surfaceTriangleElements )
```

2D and 3D regular refinement. Chosen by error estimator or otherwise elements are refined regular by connecting edge midpoints.
3D regular refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.17 refineType1()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineType1 (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement,
            SurfaceElementsPtr_Type surfaceTriangleElements )
```

3D Type(1) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.18 refineType2()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineType2 (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement,
            SurfaceElementsPtr_Type surfaceTriangleElements )
```

3D Type(2) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.19 refineType3()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineType3 (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement,
            SurfaceElementsPtr_Type surfaceTriangleElements )
```

3D Type(3) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.20 refineType4()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::refineType4 (
            EdgeElementsPtr_Type edgeElements,
            ElementsPtr_Type elements,
            int indexElement,
            SurfaceElementsPtr_Type surfaceTriangleElements )
```

3D Type(4) refinement as defined in "Tetrahedral Grid Refinement" by J. Bey 'Algorithm Regular Refinement' in Computing, Springer Verlag 1955

Parameters

in	*edgeElements*	Edges.
in	*elements*	Elements.
in	*indexELement*	Element in question.
in	*surfaceTriangleElements*	Triangle elements.

1.4.2.21 updateElementsOfEdgesLocalAndGlobal()

```
template<class SC , class LO , class GO , class NO >
void FEDD::RefinementFactory< SC, LO, GO, NO >::updateElementsOfEdges←↩
LocalAndGlobal (
            int maxRank,
            MapConstPtr_Type edgeMap )
```

Updating ElementsOfEdgesLocal and ElementsOfEdgesGlobal.

Parameters

in	*maxRank*	The maximal processor rank.
in	*edgeMap*	Map of global edge ids.

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

- RefinementFactory_decl.hpp
- RefinementFactory_def.hpp