OPTANT: Reference Guide

Delft University of Technology
Aerospace Structures & Computational Mechanics

Contents

Introduction

One of the current challenges in designing composite structures is to optimize variable stiffness laminates for e.g. stiffness or buckling load. Such an optimization requires the ability to assign different laminate properties to any point in the structure and it requires full access to the governing equations of the model, i.e. to the stiffness matrix. Especially the second requirement is often not available in commercial software packages.
OPTANT is a tool for optimization and analysis of thin-walled, composite structures which has been designed with the previous example in mind. OPTANT is object oriented (C++) and it has been fully developed using open source libraries and packages. The main goal is to perform optimization and analysis of thin walled structures using the finite element method with the capability of easily assigning variable properties to the structure and allowing full access to all parameters.

OPTANT contains all basic parts of a standard FEM tool, i.e. materials, properties, nodes, elements, single point constraints (SPC), multiple point constraints (MPC), nodal loads, pressure loads and temperature loads. It allows to perform linear static analysis and linear buckling analysis with or without prestress. A few differences with respect to many regular FEM software packages can be mentioned as well. First of all, it is possible to assign multiple properties to a single finite element, which may be used e.g. to assign a different property to each node ensuring property continuity over the element edges. Similarly, it is also possible to apply multiple temperature loads per element.
Another difference with many standard FEM tools is that all shell properties in OPTANT are composite shell properties since the main focus is on composite structures. Of course, any isotropic material is a special case of a composite material, and can be implemented as such in OPTANT.

Several open source packages are used in OPTANT, such as ARPACK++ [6] for solving the buckling eigenvalue problem and CHOLMOD [3] for factorizing the stiffness matrix. However, the most significant package is the Finite Element Interface (FEI) package from the Trilinos project [7, 10]. FEI provides an interface for e.g. assembling the global stiffness matrix, for applying loading and constraints, for solving the system of equations using third party libraries, and for accessing the solution. A few limitations/bugs in FEI were found as well, e.g. SPC cannot be implemented by reducing the global stiffness matrix in size as done for MPC, non-zero constant values in MPC equations are not handled correctly, and matrix-vector multiplication is not implemented directly. Moreover, data in FEI objects is sometimes affected by other FEI objects, leading to erroneous results. This limitation was circumvented by changing the order of certain commands in OPTANT. This is a not a very satisfying solution, but it ensures OPTANT does not suffer from this limitation.

Architecture

OPTANT is object oriented; it is written in C++. Its main function is located in `OPTANT.cpp`. One **`Domain.h`** object is created within this main function, which contains all model parameters. These model parameters are read from the input file [2]. The main function then loops over all **`Loadcases`** which are solved consecutively. A **`Solver`** object is created within this **`Loadcase`** loop, which is then initialized, prepared, and solved. After the solution has been computed, the requested output is written to the output file and the **`Solver`** object is deleted. An overview of all classes in OPTANT and their interactions is provided in Fig. 1.

The remainder of this reference guide contains the description of all classes, including class variables and functions. Note that many functions have an `int` as `return` type. Unless stated otherwise, this `int` is used to check whether an error occurred within the function. A `0` is returned if no error occurred, a `1` or `-1` is returned if an error occurred. A description of the error is usually presented at the console.

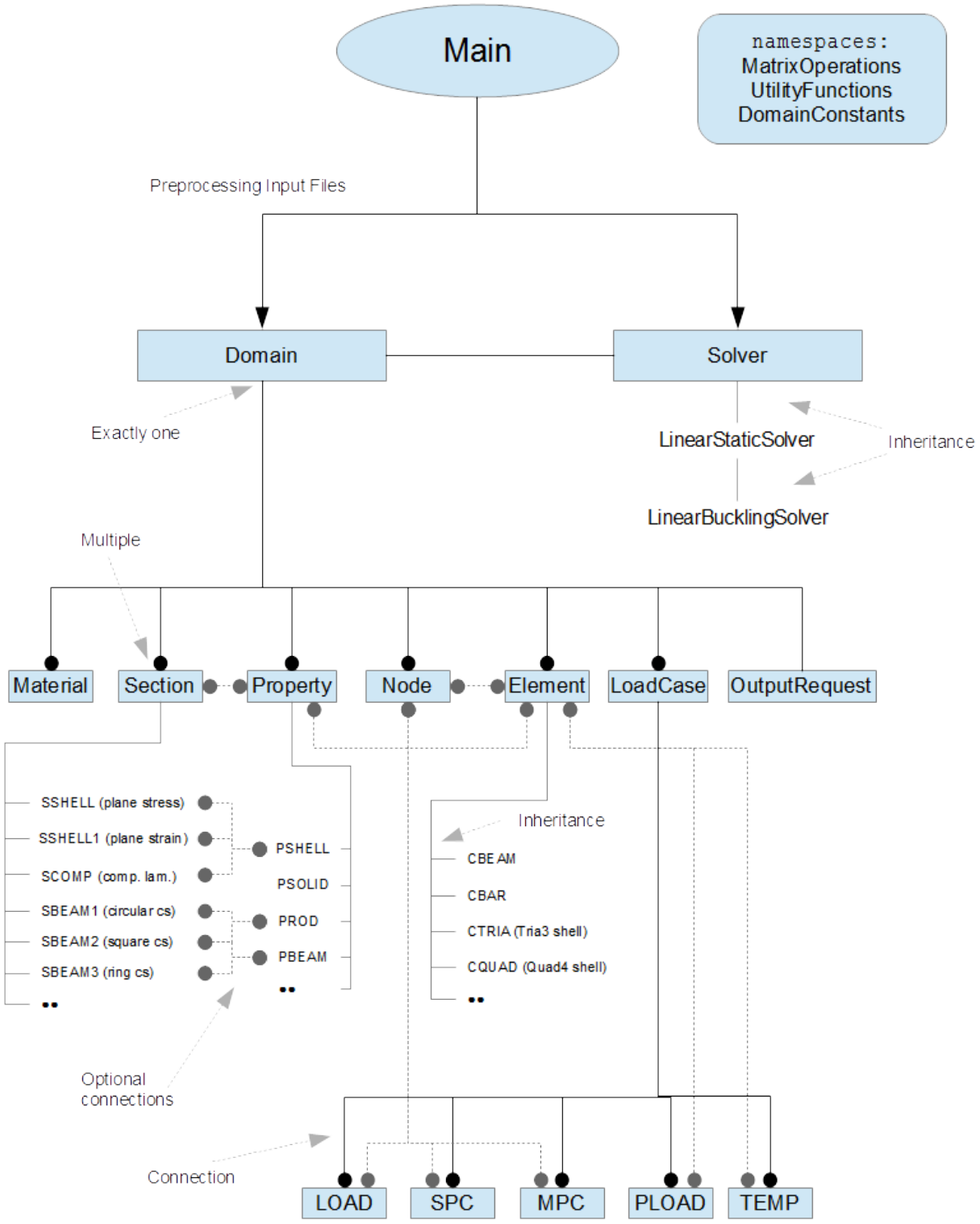

Figure 1: Overview of OPTANT classes.

DomainConstants.h

Description

The file `DomainConstants.h` contains several constant variables which are used throughout OPTANT. Therefore, it is not a class and thus no object of this type exists.

Variables

`const int`	**MAXNAMESIZE**	(=20) Size of a character array containing a name, e.g. a material name or a element card such as "TRIA".
`const int`	**MAXLINESIZE**	(=100) Size of a character array containing a line, e.g. an entire line from the input file.
`const double`	**WARNINGANGLE**	(=30deg) Critical angle at which warning message is given to the user. This angle can either be the angle between the specified normal direction and tangential plane of a shell element, or the angle between specified lateral direction and longitudinal direction of a beam element.
`const double`	**ERRORANGLE**	(=1deg) Critical angle at which an error message is given to the user.
`const int`	**DISP_FIELD_ID**	(=0) Identification number for displacement field required by the FEI package to set up the stiffness matrix.
`const int`	**DISP_FIELD_SIZE**	(=6) Size of displacement field, i.e. three translations and three rotations.
`const int`	**NODE_TYPE_ID**	(=0) Identification number for the node type required by the FEI package to set up the stiffness matrix.
`const int`	**OUTPUT_PRECISION**	(=9) Number of digits after decimal point of data (scientific notation) in the output file.
`const int`	**OUTPUT_DOUBLE_FIELD_WIDTH**	(=17) Number of characters allocated for a value of type `double` in the output file.
`const int`	**OUTPUT_INT_FIELD_WIDTH**	(=6) Number of characters allocated for a value of type `int` in the output file.

Domain.h

Description

The Domain.h class is one of the key classes in OPTANT as it contains the entire finite element model. A single object of this class is created, which has access to all model data. This model data includes the all contents of the input file [2].
The Domain.h class is closely related to the **Solver.h** class, since a Solver object requires many of the model data from Domain.h. This means many of the Domain.h functions are called by Solver objects. Especially those functions use many of the capabilities of the FEI package [10] which is used to set up the system of equations and to provide an interface to the Amesos solver [9].

Variables

NOTES:

1. The terms local ID and global ID are often used in the variable descriptions. A global ID refers to the ID as specified by the user in the input file, whereas a local ID refers to the ID used within OPTANT (usually 0,1,2,...).

2. The parameter NumTotalBlocks used in the descriptions is equivalent to the number of $ElementType definitions in the input file.

map <int,Material*> MaterialList	List containing pointers to all **Material** objects. The key-value pair <int,Material*> consists of the global material ID from the input file and the pointer to the corresponding **Material** object.
vector <Section*> SectionList	Vector containing pointers to all **Section** objects. The position of each object in this vector corresponds to its local ID.
vector <Property*> PropertyList	Vector containing pointers to all **Property** objects. The position of each object in this vector corresponds to its local ID.
vector <Node*> NodeList	Vector containing pointers to all **Node** objects. The position of each object in this vector corresponds to its local ID.
vector <Element*> ElementList	Vector containing pointers to all **Element** objects. The position of each object in this vector corresponds to its local ID.

map <int,LoadCase*>	**LoadCaseList**	List containing pointers to all **LoadCase** objects. The key-value pair `<int,LoadCase*>` consists of the global load case ID from the input file and the pointer to the corresponding **LoadCase** object.
multimap <int,SPC*>	**SPCList**	List containing pointers to all **SPC** objects. The key-value pair `<int,SPC*>` consists of the spcSetID from the input file and the pointer to the corresponding **SPC** object. Note that `multimap` is applied since multiple `SPC` objects may belong to the same spcSetID.
multimap <int,MPC*>	**MPCList**	Same as SPCList, but for **MPC** objects.
multimap <int,LOAD*>	**LOADList**	Same as SPCList, but for **LOAD** objects.
multimap <int,PLOAD*>	**PLOADList**	Same as SPCList, but for **PLOAD** objects.
multimap <int,TEMP*>	**TEMPList**	Same as SPCList, but for **TEMP** objects.
vector <int>	**NodeID_LG**	Conversion from local to global **Node** IDs. The position in the vector represents the local ID, and the vector value represents the global ID.
vector <int>	**ElementID_LG**	Same as NodeID_LG, but for **Element** objects.
vector <int>	**PropertyID_LG**	Same as NodeID_LG, but for **Property** objects.
map <int,int>	**NodeID_GL**	Conversion from global to local **Node** IDs. The key-value pair `<int,int>` consists of the global and local ID respectively.
map <int,int>	**ElementID_GL**	Same as NodeID_GL, but for **Element** objects.
map <int,int>	**PropertyID_GL**	Same as NodeID_GL, but for **Property** objects.
vector <int>	**NodalNumDOF**	Vector of length `NumTotalNodes` containing the number of DOF for each node.
vector <int>	**NodesPerElementBlock**	Vector of length `NumTotalBlocks` (see 2), containing the number of nodes per element of each block.
vector <int>	**NumElementsBlock**	Vector of length `NumTotalBlocks` (see 2), containing the number of elements for each block.
vector <int>	**ElementBlockID**	Vector of length `NumTotalElements` containing the (local) block ID of each element.
int	**ElemMatPackStorageSize**	The total number of entries in the element matrices of all **Elements** in packed storage.
vector <double>	**AllElementMatrices**	Vector of size `ElemMatPackStorageSize` containing all element matrices in packed storage. This vector is sized and filled in **BuildElementMatrices(...)**. Each **Element** object contains a pointer to its own element matrix in this vector.
vector <double>	**AllGeometricElement-Matrices**	Same as `AllElementMatrices`, but for geometric element matrices. This vector is only computed if the **Solver** is a **LinearBucklingSolver**.

fei::ParameterSet	**SolverParams**	Parameter set containing solution parameter related to the Trilinos package. `SolverParams` is created based on the `SolverParameters.dat` file, which is read by OPTANT. Its most important contribution is to specify `MUMPS` [1] as solver for the linear system.
Domain::SolType	**SolutionType**	Solution type as specified in the input file, i.e. 10 for **LinearStaticSolver** and 11 for **LinearBucklingSolver**.
int	**NumTotalElements**	Total number of **Elements**.
int	**NumLocalElements**	Total number of **Elements** on this processor (NOT USED).
int	**NumTotalNodes**	Total number of **Nodes**.
bool	**ElementMatricesBuilt**	True if element matrices have been built. This variable is used to make sure the element matrices are built only once.

Public Functions

Domain(const char* filename)

Constructor. Several variables are initialized.

filename	*Input*	Name of input file.

---

int ReadInputFile()

The input file, as specified in the constructor, is read by calling **UtilityFunction::read_input_file**.

---

int BuildElementMatrices()

All element matrices are built. First, the total number of entries in packed storage, `ElemMatPackStorageSize`, is computed. Next, the variable `ElementMatrix_` is specified for each **Element**, and finally, **BuildElementMatrix(...)** is called on all elements. The boolean `ElementMatricesBuilt` is then set to `true`.

---

int BuildGeometricElementMatrices(Solver* solver, bool makeNegative = *false* **)**

Similar to **BuildElementMatrices()**, but for geometric element matrices. A **Solver** object is required as input because the geometric matrix depends on displacements.

solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the geometric element matrices.
makeNegative	*Input*	If true, the element geometric matrices are multiplied by -1. Used for differential stiffness matrix induced by prestresses for **LinearBucklingSolver**.

---

int InitializeElementConnectivities(fei::MatrixGraph* matrixGraph)

The element (nodal) connectivities specify the location of all non-zero entries in the global

stiffness matrix. These element connectivities are initialized here. The FEI package uses connectivity blocks for each block of elements (such a block consists of all elements within a single `$ElementType` block in the input file). A block of elements is characterized mainly by the number of nodes per element.
After the connectivity blocks have been initialized for all blocks, the connectivity of each individual **Element** is initialized by iterating through the elements.

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **MPC** in the background.

---

int AssembleStiffnessMatrix(fei::MatrixGraph* matrixGraph, fei::Matrix* mat, bool useGeomStiffness = *false* **)**

All element matrices are assembled into the global matrix `mat`, which is based on the matrix graph `matrixGraph`. The matrix graph is used to get the nodal connectivities of each **Element**, which should have been initialized using **InitializeElementConnectivities(...)**.
The element matrices are obtained using **ElementMatrix(...)**, or **GeometricElementMatrix(...)** when the geometric elements are assembled

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **MPC** in the background.
mat	*Input*	Pointer to global matrix into which the element matrices are assembled.
useGeomStiffness	*Input*	True if geometric element matrices are assembled.

---

int InitializeConstraints(fei::MatrixGraph* matrixGraph, int globalLoadCaseID)

All **MPC** for the current load case are initialized in the matrix graph, by calling **MPC::Initialize(...)**. This initialization allows the FEI package to implement these MPC in the global stiffness matrix by reducing its size [10].
Note that also **SPC** should be initialized similar to **MPC**. However, due to a bug within FEI this can not be realized, therefore **SPC** are only applied within the stiffness matrix by setting the relevant diagonal entries equal to 1 and the relevant rows and columns equal to 0.

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **MPC** in the background.
globalLoadCaseID	*Input*	Global ID of current load case used to find which constraints are active.

---

int ApplyConstraints(fei::LinearSystem* linSys, int globalLoadCaseID)

All **SPC** for the current load case are applied to the `fei::LinearSystem`, by calling **SPC::Apply()**. The constraints are handled by FEI by modifying the stiffness matrix and force vector of the linear system.

linSys	*Input*	Pointer to the linear system, which is an FEI related object containing the stiffness matrix, displacement vector, and force vector.
globalLoadCaseID	*Input*	Global ID of current load case used to find which constraints are active.

---

int ApplyLoads(fei::MatrixGraph* matrixGraph, fei::Vector* rhsVec, int globalLoadCaseID)

All **LOAD**, **PLOAD** and **TEMP** for the current load case are applied to the force vector of the linear system. The matrix graph is e.g. used to extract element connectivities in case of element loads (**PLOAD** and **TEMP**).

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **MPC** in the background.
rhsVec	*Input*	Pointer to force vector in which the loads are applied.
globalLoadCaseID	*Input*	Global ID of current load case used to find which loads are active.

---

int ResetElementTemperatures( )

This function resets all applied temperatures for the elements by calling **ResetDeltaTemperatures()** for each element. The temperatures are reset at the end of each load case in order to prevent these temperature loads from affecting future load cases.

---

int WriteOutput(Solver* solver, const char* outputFile, int globalLoadCaseID)

Output from the linear system solution is written to the output file. The **OutputRequest** object `OutputReq_` contains several variables specifying what output data is requested.
In the implementation of this function, it is first checked whether the output file has been opened already. If this is the case, the new data is appended to the output file, otherwise a new output file is created. Next, the number of temporary files is computed based on the number of output requests. Each temporary file contains one type of output (e.g. element strains or element forces). These temporary files are then added to the output file and cleared in **HandleTempFiles(...)**. The temporary files are written by iterating through all **Nodes** or **Elements** and calling their `WriteOutput(...)` function.
Note that output for a **LinearStaticSolver** is dealt with separately in **WriteLinearBucklingOutput(...)**.

solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute and write output data.
outputFile	*Input*	Name of output file.
globalLoadCaseID	*Input*	Global ID of current load case. (NOT USED)

---

int WriteLinearBucklingOutput(LinearBucklingSolver* solver, const char* outputFile)

The procedure is very similar to **WriteOutput(...)**, but with a few differences. The number of temporary files is equal to the number of buckling modes plus one. One file is used to write the buckling loads for all modes, the other files are used to write the buckling mode shapes. This means only nodal output is generated.

solver	*Input*	Pointer to solver containing the **Arpack problem/solution** [6], which contains the buckling modes.
outputFile	*Input*	Name of output file.

Private Functions

```
int HandleTempFiles( ofstream& fout, ofstream* tempFiles, const int&
numTempFiles, const char* tempFilePath )
```

The temporary files used when writing output are closed, added to the output file, and cleared in this function.

fout	*Input*	Output stream for output file.
tempFiles	*Input*	Array of output streams for temporary files.
numTempFiles	*Input*	Number of temporary files.
tempFilePath	*Input*	Directory path of all temporary files.

Material.h

Description

The `Material.h` class is used to define the materials from the input file. Each `Material` object is characterized by a material name, material type, and an array of properties. The material name is not used in any part of the OPTANT code. The type of material determines how to interpret the specified array of properties.
Even though OPTANT has been designed specifically for composite materials, only isotropic materials are currently supported in the `Material.h` class. Composite materials should be specified by using the **PSHELL** object with the $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices as input. In a future release this should be corrected by allowing the definition of orthotropic materials, which would then be stacked into a composite laminate by means of a **Section** object.

Variables

char	**MaterialName_**	Name of the material
Material::Type	**MaterialType_**	Material type as specified in the input file, i.e. 0 for Isotropic, 1 for 2D orthotropic, 2 for 2D anisotropic and 3 for 3D orthotropic.
double	**Density_**	Density of the material. (NOT USED)
double	**ReferenceTemperature_**	Reference temperature of the material. (NOT USED)
double	**EngineeringMaterialProperties_[9]**	Engineering material properties: E_{11}, E_{22}, E_{33}, ν_{23}, ν_{13}, ν_{12}, G_{23}, G_{13}, G_{12}.
double	**MaterialStiffness_[36]**	Full 6×6 material stiffness matrix: C_{11}, C_{21}, ..., C_{61}, C_{12}, ...
double	**PlaneStressStiffness_[9]**	3×3 plane stress material stiffness matrix.
double	**PlaneStrainStiffness_[9]**	3×3 plane strain material stiffness matrix.
double	**ThermalExpansion_[6]**	6 thermal expansion coefficients: α_{11}, α_{22}, α_{33}, α_{23}, α_{13}, α_{12}.
double	**ThermalConductivity_[9]**	3×3 thermal conductivity matrix.

Public Functions

Material(const char* name, int type, double* propArray)

Constructor. The material name is initialized from the input and, depending on `type`, the `propArray` is dealt with. In this version only isotropic materials are supported, therefore, the

`propArray` is always passed to **`IsotropicMaterial(...)`**.

`name`	*Input*	Name of material name.
`type`	*Input*	Material type.
`propArray`	*Input*	Array of material properties. For isotropic materials it contains: ρ, E, ν, T_{ref}, α, λ (thermal conductivity).

Private Functions

`int IsotropicMaterial( double* propArray )`

Tranlates the properties in `propArray` into the variables of this class.

`propArray`	*Input*	Array of material properties, which contains: ρ, E, ν, T_{ref}, α, λ (thermal conductivity).

Section.h

Description

`Section.h` is a base class for several child classes for different types of beam or shell sections. These may include e.g. circular beam section, square beam section, plane stress shell section, composite shell section, etc. Especially the composite shell section is interesting, as it allows to define composites laminates using **`Material`** objects, lay-up angles and ply thicknesses.
Section object may also be used for writing output, as they allow to convert normal forces and moments into stresses. In this way the maximum stress in a beam section or the ply stresses in a composite shell section can be computed.

NOTE: `Section.h` has not been implemented!

Property.h

Description

`Property.h` is a base class for the beam property **PBEAM** and shell property **PSHELL**. A `Property` is the combination of geometric characteristics and constitutive law of a structure. For **PBEAM** this consists of parameters like e.g. EA or EI, and for **PSHELL** this consists of e.g. the $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices.
All **Elements** are characterized by at least one `Property`. For example, in a constant stiffness composite laminate all elements would have the same property, whereas they would have different properties in a variable stiffness composite laminate.

Variables

Section*	**Section_**	Pointer to **Section** for this property. (NOT USED)
Property::Type	**PropertyType_**	Property type as specified in the input file, i.e. 0 for Rod, 1 for Beam, 2 for Shell and 3 for Solid.

Public Functions

Property(Property::Type pt)

Constructor. Property type is initialized.

pt	*Input*	Property type

---

virtual int ReadFromFile(ifstream& fin, int PropOption, int NumLines, Domain& domain) = 0

Pure virtual function for reading the `Property` characteristics from the input file. This function is implemented in **PBEAM.h** and **PSHELL.h**

fin	*Input*	Input stream of input file
PropOption	*Input*	Option that indicates how this property is specified in the input file [2].
NumLines	*Input*	Number of lines in property definition in the input file [2].
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

PBEAM.h : Property.h

Description

A PBEAM.h object represents a general beam property, for which e.g. Euler-Bernoulli beam theory and Timoshenko beam theory are special cases. For example, this property allows for coupling between normal/bending loads and shear loads. A short description of the theory used for this beam property is described in the OPTANT User Manual [2].

Variables

Material*	material_	Pointer to material used for this property. This pointer may be zero based on the way the property is specified in the input file.
double	BendingStiffness_[16]	4×4 normal/bending stiffness tensor $\tilde{\boldsymbol{C}}$ [2].
double	ShearStiffness_[4]	2×2 shear stiffness tensor $\tilde{\boldsymbol{S}}$ [2].
double	CouplingStiffness_[8]	2×4 coupling stiffness tensor $\tilde{\boldsymbol{G}}$ [2].
double	BendingExpansion_[4]	4×1 normal/bending expansion vector $\tilde{\boldsymbol{c}}$ [2].
double	ShearExpansion_[2]	2×1 shear expansion vector $\tilde{\boldsymbol{g}}$ [2].

Public Functions

PBEAM()

Constructor. PropertyType_ is initialized to be a Beam property, and the variables are initialized to zero.

---

int ReadFromFile(ifstream& fin, int PropOption, int NumLines, Domain& domain)

The characteristics of the PBEAM are read from the input file and translated into the variables (BendingStiffness_, etc) of this object.

fin	*Input*	Input stream of input file
PropOption	*Input*	Option that indicates how this property is specified in the input file [2].
NumLines	*Input*	Number of lines in property definition in the input file [2].
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

PSHELL.h : Property.h

Description

A `PSHELL.h` object represents a shell property by its $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices and $\boldsymbol{a}, \boldsymbol{b}$ temperature expansion vectors. These parameters allow the specification of any composite laminate, of a plane stress or plane strain isotropic shell, etc.
The derivation and definition of $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices according to Classical Laminate Theory can be found in e.g. [8]. Based on these derivations, the $\boldsymbol{a}, \boldsymbol{b}$ are derived as follows. For each ply the strain is given as:

$$\begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \gamma_{12} \end{pmatrix} = \boldsymbol{S} \begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \tau_{12} \end{pmatrix} + \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_{12} \end{pmatrix} \Delta T \tag{1}$$

with $1, 2$ indicating the principal ply directions. Inverting this relation yields:

$$\begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \tau_{12} \end{pmatrix} = \underbrace{\boldsymbol{S}^{-1}}_{\boldsymbol{Q}} \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \gamma_{12} \end{pmatrix} - \underbrace{\boldsymbol{S}^{-1} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_{12} \end{pmatrix}}_{\boldsymbol{q}} \Delta T \tag{2}$$

in which $\boldsymbol{Q}$ is the standard plane stress stiffness matrix for an orthotropic ply material [8]. Rotating (2) to the laminate coordinates x, y gives:

$$\begin{aligned} \begin{pmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{pmatrix} &= \boldsymbol{T_\sigma}^{-1} \boldsymbol{Q} \boldsymbol{T_\varepsilon} \begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{pmatrix} - \boldsymbol{T_\sigma}^{-1} \boldsymbol{q} \Delta T \\ &= \hat{\boldsymbol{Q}} \begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{pmatrix} - \hat{\boldsymbol{q}} \Delta T \end{aligned} \tag{3}$$

with $\boldsymbol{T_\sigma}$ and $\boldsymbol{T_\varepsilon}$ being the rotation matrices for stresses and strains respectively. Similar to the $\boldsymbol{A}, \boldsymbol{B}$ matrices, the $\boldsymbol{a}, \boldsymbol{b}$ vectors are now defined as:

$$\boldsymbol{a} = \sum_{k=1}^{n_{ply}} \hat{\boldsymbol{q}}_k \left(z_k - z_{k-1} \right) \tag{4}$$

$$\boldsymbol{b} = \sum_{k=1}^{n_{ply}} \hat{\boldsymbol{q}}_k \left(z_k^2 - z_{k-1}^2 \right) \tag{5}$$

which results in the internal forces due to applied temperature $\boldsymbol{N_T}$:

$$\boldsymbol{N_T} = -\begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} \Delta T \tag{6}$$

Variables

double	**A_[6]**	$\boldsymbol{A}$ matrix of the shell in packed storage.
double	**B_[6]**	$\boldsymbol{B}$ matrix of the shell in packed storage.
double	**D_[6]**	$\boldsymbol{D}$ matrix of the shell in packed storage.
double	**a_[3]**	$\boldsymbol{a}$ vector of the shell.
double	**b_[3]**	$\boldsymbol{b}$ vector of the shell.

Public Functions

PSHELL()

Constructor. `PropertyType_` is initialized to be a Shell property, and the variables are initialized to zero.

int ReadFromFile(ifstream& fin, int PropOption, int NumLines, Domain& domain)

The characteristics of the `PSHELL` are read from the input file and translated into the variables (`A_`, etc) of this object.

fin	*Input*	Input stream of input file
PropOption	*Input*	Option that indicates how this property is specified in the input file [2].
NumLines	*Input*	Number of lines in property definition in the input file [2].
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

Node.h

Description

Each node in the input file is represented an object from the `Node.h` class. A `Node` is characterized by its local ID and its global coordinates.

Variables

double	**GlobalCoordinates_[3]**	X, Y, Z coordinates.
int	**LocalNodeID_**	Local node ID.

Public Functions

Node()

Constructor. Variables are initialized to zero.

---

int Displacements(Solver* solver, double* DOFValues)

The six DOF of this nodes are extracted from the solution of `solver` and written `DOFValues`. First the indices of the relevant DOF within the solution vector are obtained, after which the DOF are copied from the solution vector into `DOFValues`.

solver	*Input*	Pointer to solver containing the **DisplacementVector_**.
DOFValues	*Output*	Output array of nodal DOF.

---

int BucklingModeDisplacements(LinearBucklingSolver* solver, int mode, double* DOFValues)

Similar to **Displacements(...)**, but for the buckling modes. The input argument `mode` indicates the buckling modes for which the displacements are requested.

solver	*Input*	Pointer to solver containing the **Arpack problem/solution** [6], which contains the buckling modes.
mode	*Input*	Mode ID for which displacements are requested.
DOFValues	*Output*	Output array of nodal DOF.

---

int WriteOutput(Domain& domain, Solver* solver, std::ofstream* tempFiles, bool doTitle=*false*)

The nodal displacements are written in one of the temporary files, in case nodal outputs are requested by the user. The latter is checked first from the **OutputRequest** object `OutputReq_` of `domain`. If nodal outputs are requested, **Displacements(...)** is called after which the output displacements are written in the temporary file.
If `doTitle` is `true`, a title explaining the output data is written over the output displacements.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**.
tempFiles	*Input*	Array of output streams for temporary files.
doTitle	*Input*	If `true`, a specified title is printed.

---

int WriteLinearBucklingOutput(Domain& domain, LinearBucklingSolver* solver, std::ofstream* tempFiles, bool doTitle=*false* **)**

Similar to **WriteOutput(...)**, but now it is checked whether nodal buckling output is requested from the **OutputRequest** object, and **BucklingModeDisplacements(...)** is called for all buckling modes.

domain	*Input*	Pointer to the domain
solver	*Input*	Pointer to the solver containing the output
tempFiles	*Input*	Array of temporary files, to write output
doTitle	*Input*	If true, a specified title is printed

Element.h

Description

Element.h is a base class for the beam element **CBEAM**, triangular shell element **CTRIA**, and quadrilateral shell element **CQUAD**. An Element is characterized by its **Nodes**, its **Properties**, and a lateral direction (for **CBEAM**) or normal direction (for **CTRIA** and **CQUAD**).
The Element.h class has relatively many pure virtual functions for e.g. building the element matrix, building the element geometric matrix, building the element temperature load vector, etc.

Variables

double*	**ElementMatrix_**	Pointer to the element matrix in packed storage in the **AllElementMatrices** vector.
double*	**GeometricElementMatrix_**	Pointer to the geometric element matrix in packed storage in the **AllGeometricElement-Matrices** vector.
int	**LocalElementID_**	Local element ID.
int	**NumProperties_**	Number of **Properties** assigned to this element.
Property**	**Properties_**	Array of **Property** pointers.
int	**NumDeltaTemperatures_**	Number of delta temperatures applied to this element.
double*	**DeltaTemperatures_**	Array of delta temperatures.
Element::Type	**ElementType_**	Element type, i.e. 0 for Rod, 1 for Beam, 2 for Shell and 3 for Solid.

Public Functions

Element(Element::Type et)

Constructor. Element type and some other variables are initialized.

et	*Input*	Element type.

~Element()

Destructor. The variable Properties_ is deleted as this is a double pointer.

---

virtual int BuildElementMatrix() = 0

Pure virtual function for building the element matrix. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

---

virtual int BuildGeometricElementMatrix(Solver* solver, bool makeNegative = *false*) = 0

Pure virtual function for building the geometric element matrix. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the geometric element matrix.
makeNegative	*Input*	If true, the element geometric matrix is multiplied by -1. Used for differential stiffness matrix induced by prestresses for **LinearBucklingSolver**.

---

virtual void ElementMatrix(double*& stiffMat, int& size = *false*) = 0

Pure virtual function for extracting the element matrix. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

stiffMat	*Output*	Pointer to location to which the element matrix is extracted.
size	*Output*	Size of the element matrix (i.e. number of rows).

---

virtual void GeometricElementMatrix(double*& geomMat, int& size = *false*) = 0

Pure virtual function for extracting the geometric element matrix. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

geomMat	*Output*	Pointer to location to which the geometric element matrix is extracted.
size	*Output*	Size of the geometric element matrix (i.e. number of rows).

---

virtual int TemperatureLoadVector(double* loadVector) = 0

Pure virtual function for computing the temperature load vector. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

loadVector	*Output*	Temperature load vector.

---

virtual int InterpolateProperties(const int nProp, double* interpMat) = 0

Pure virtual function for creating the interpolation matrix. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

nProp	*Input*	Number of properties.
interpMat	*Output*	Pointer to location of interpolation matrix.

---

virtual int NormalDirection(double* n) = 0

Pure virtual function for calculating the unit normal direction to the element. This function is implemented in **CTRIA.h** and **CQUAD.h**. Note that this function is never called for **CBEAM** elements as this function does not make sense in that case.

n	*Output*	Unit normal direction in X, Y, Z coordinates.

---

virtual int ElementArea(double& area) = 0

Pure virtual function for calculating the area of the element. This function is implemented in **CTRIA.h** and **CQUAD.h**. Note that this function is never called for **CBEAM** elements as this function does not make sense in that case.

area	*Output*	Element area.

---

void ResetDeltaTemperatures()

Resets delta temperatures by setting `NumDeltaTemperatures_` equal to zero and `DeltaTemperatures_` equal to NULL.

---

virtual int ReadFromFile(ifstream& fin, Domain& domain) = 0

Pure virtual function for reading the `Element` characteristics from the input file. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

fin	*Input*	Input stream of input file
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int ReadPropertyLines(ifstream& fin, Domain& domain, Property::Type propType)

The second line of each element definition in the input file specifies the element properties. This line is read by this function.

fin	*Input*	Input stream of input file
domain	*Input*	Reference to the **Domain.h** object for the FEM model.
propType	*Input*	Property type. This should be consistent with element type.

---

virtual int WriteOutput(Domain& domain, Solver* solver, ofstream* tempFiles, bool doTitle=*false*) = 0

Pure virtual function for writing output in the output file. This function is implemented in **CBEAM.h**, **CTRIA.h** and **CQUAD.h**.

domain	*Input*	Pointer to the domain
solver	*Input*	Pointer to the solver containing the output
tempFiles	*Input*	Array of temporary files, to write output
doTitle	*Input*	If true, a specified title is printed

CBEAM.h : Element.h

Description

The CBEAM element is a two-noded 1D element, characterized by one ore more **PBEAM** properties. The stiffness matrix and temperature force vector for this element are based on a modified Hellinger-Reissner principle, as explained in the OPTANT User Manual [2].
The beam element is given in parametric space by $\xi \in [-1, 1]$ and $d\xi = L/2dx$, with x being the coordinate along the longitudinal direction of the beam and L being the length of the beam element. The beam element stiffness matrix is a function of the **PBEAM** stiffness matrices integrated over the beam element, which gives rise to the following definitions:

$$\boldsymbol{C_{00}} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{C}} d\xi \qquad \boldsymbol{C_{01}} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{C}} \xi d\xi \qquad \boldsymbol{C_{11}} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{C}} \xi^2 d\xi \tag{7}$$

$$\boldsymbol{G_0} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{G}} d\xi \qquad \boldsymbol{G_1} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{G}} \xi d\xi \tag{8}$$

$$\bar{\boldsymbol{S}} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{S}} d\xi \tag{9}$$

$$\boldsymbol{c_0} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{c}} \Delta T d\xi \qquad \boldsymbol{c_1} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{c}} \Delta T \xi d\xi \qquad \boldsymbol{g_0} = \frac{1}{2}\int_{-1}^{1} \tilde{\boldsymbol{g}} \Delta T d\xi \tag{10}$$

These integrals are evaluated using the two-point Gauss integration rule. Note that the stiffness matrices in (7)-(10) are not constant over the beam element if multiple properties are assigned to the element. These properties are interpolated to the Gauss points using the interpolation function **InterpolateProperties(...)**.
The applied strain energy function for the beam Φ^{**} is given as:

$$\Phi^{**} = \frac{1}{2}\boldsymbol{\varepsilon}^T \tilde{\boldsymbol{C}} \boldsymbol{\varepsilon} + \boldsymbol{V}^T \tilde{\boldsymbol{G}} \boldsymbol{\varepsilon} - \frac{1}{2}\boldsymbol{V}^T \tilde{\boldsymbol{S}} \boldsymbol{V} - \boldsymbol{\varepsilon}^T \tilde{\boldsymbol{c}} \Delta T - \boldsymbol{V}^T \tilde{\boldsymbol{g}} \Delta T \tag{11}$$

The element stiffness matrix $\boldsymbol{K_e}$ and temperature force vector $\boldsymbol{F_{T,e}}$ are derived by applying the principle of variations to this expression:

$$\boldsymbol{K_e} = \frac{1}{L}\boldsymbol{B_0}^T \left(\boldsymbol{C_{00}} - \boldsymbol{C_{01}}\boldsymbol{C_{11}}^{-1}\boldsymbol{C_{01}}\right)\boldsymbol{B_0} + L\boldsymbol{B^*}^T\boldsymbol{C_s}\boldsymbol{B^*} \tag{12}$$

$$\boldsymbol{F_{T,e}} = -\boldsymbol{B_0}^T\boldsymbol{x_3} - L\boldsymbol{B_2}^T\boldsymbol{x_2} \tag{13}$$

with:

$$\boldsymbol{B^*} = \boldsymbol{G^*}\boldsymbol{B_0} + \boldsymbol{B_2} \tag{14}$$

$$\boldsymbol{C_s} = \left(\bar{\boldsymbol{S}} + \frac{L^2}{4}\left(\boldsymbol{B_3} - \frac{2}{L}\boldsymbol{G_1}\right)\boldsymbol{C_{11}}^{-1}\left(\boldsymbol{B_3} - \frac{2}{L}\boldsymbol{G_1}\right)^T\right)^{-1} \tag{15}$$

$$\boldsymbol{x_2} = -L\boldsymbol{C_s}\left(\frac{1}{2}\left(\boldsymbol{B_3} - \frac{2}{L}\boldsymbol{G_1}\right)\boldsymbol{C_{11}}^{-1}\boldsymbol{c_1} + \frac{1}{L}\boldsymbol{g_0}\right) \tag{16}$$

$$\begin{aligned}\boldsymbol{x_3} = &-L^2\boldsymbol{G^*}^T\boldsymbol{C_s}\left[\frac{1}{2}\left(\boldsymbol{B_3} - \frac{2}{L}\boldsymbol{G_1}\right)\boldsymbol{C_{11}}^{-1}\boldsymbol{c_1} + \frac{1}{L}\boldsymbol{g_0}\right] \\ &+ \boldsymbol{C_{01}}\boldsymbol{C_{11}}^{-1}\boldsymbol{c_1} - \boldsymbol{c_0}\end{aligned} \tag{17}$$

$$\boldsymbol{G^*} = \frac{1}{2}\left(\boldsymbol{B_3} - \frac{2}{L}\boldsymbol{G_1}\right)\boldsymbol{C_{11}}^{-1}\boldsymbol{C_{01}} + \frac{1}{L}\boldsymbol{G_0} \tag{18}$$

and:

$$\boldsymbol{B_0} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{19}$$

$$\boldsymbol{B_2} = \frac{1}{2L}\begin{bmatrix} 0 & -2 & 0 & 0 & 0 & -L & 0 & 2 & 0 & 0 & 0 & L \\ 0 & 0 & -2 & 0 & L & 0 & 0 & 0 & 2 & 0 & L & 0 \end{bmatrix} \tag{20}$$

$$\boldsymbol{B_3} = \frac{1}{3}\begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{21}$$

Once the nodal displacements (in element coordinates) have been computed, the shear forces $\boldsymbol{V}$ and normal force / bending moments $\boldsymbol{N}$ can be computed as:

$$\boldsymbol{V} = \boldsymbol{V_0} + \boldsymbol{V_{0,T}} \qquad = \begin{pmatrix} V_y & V_z \end{pmatrix}^T \tag{22}$$

$$\boldsymbol{N} = \boldsymbol{N_0} + \boldsymbol{N_{0,T}} + \boldsymbol{N_1}\xi \qquad = \begin{pmatrix} N_x & M_x & M_y & M_z \end{pmatrix}^T \tag{23}$$

$$\tag{24}$$

with:

$$\boldsymbol{V_0} = \boldsymbol{C_s B^* u} \tag{25}$$

$$\boldsymbol{V_{0,T}} = \boldsymbol{x_2} \tag{26}$$

$$\boldsymbol{N_0} = \left(\frac{1}{L} \left(\boldsymbol{C_{00}} - \boldsymbol{C_{01}} \boldsymbol{C_{11}}^{-1} \boldsymbol{C_{01}} \right) + L \boldsymbol{G^{*T}} \boldsymbol{C_s} \boldsymbol{G^*} \right) \boldsymbol{B_0 u} + L \boldsymbol{G^{*T}} \boldsymbol{C_s} \boldsymbol{B_2 u} \tag{27}$$

$$\boldsymbol{N_{0,T}} = \boldsymbol{x_3} \tag{28}$$

$$\boldsymbol{N_1} = \frac{3L}{2} \boldsymbol{B_3}^T \boldsymbol{V_0} \tag{29}$$

Note that the subscript T indicates the forces/moments induced by temperature loading. The element strain energy can be computed from the displacements as:

$$\phi_e = \frac{1}{2} \boldsymbol{u}^T \boldsymbol{K_e u} - \boldsymbol{u}^T \boldsymbol{F_T} \tag{30}$$

This expression is consistent with MSC Nastran, however, it is slightly inconsistent with the **CTRIA** and **CQUAD** elements. The strain energy of these two shell elements contains an additional constant value which ensures the strain energy is always equal to or greater than zero. This positiveness is preferred by the authors, however, the additional constant value has not been consistently derived for the CBEAM element.

Variables

Node*	**Nodes_[2]**	Element **Nodes**.
double	**ZDirection_[3]**	Element z direction in global coordinates with unit length. Sign convention is given in Optant User Manual [2].

Public Functions

CBEAM()

Constructor.

int BuildElementMatrix()

Element matrix, see (12), is computed. First the transformation between global coordinates X, Y, Z and the parametric coordinate ξ is computed by means of a transformation matrix Λ. The constitutive matrices as given in (7)-(10) are then computed by calling **MaterialMatrices(...)**. Subsequently, the matrix operations required to obtain the element stiffness matrix are performed. Storage allocation is minimized and the multiplication by the $\boldsymbol{B_i}$ matrices, see (19)-(21), have been hard coded to prevent multiplications by 0. To increase speed further, many of the other matrix multiplications have been hard coded as well using the **MatrixOperations** namespace.
The element matrix is stored in packed storage format.

int BuildGeometricElementMatrix(Solver* solver, bool makeNegative = *false* **)**

Currently, the geometric stiffness element for beams is set to the identity matrix, which is non-physical.
The geometric element matrix is stored in packed storage format.

solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the geometric element matrix.
makeNegative	*Input*	If true, the element geometric matrix is multiplied by -1. Used for differential stiffness matrix induced by prestresses for **LinearBucklingSolver**.

---

void ElementMatrix(double*& stiffMat, int& size)

The element matrix is translated into conventional format using **PackedToConventional(...)** and then stored in `stiffMat`.

stiffMat	*Output*	Pointer to location to which the element matrix is extracted.
size	*Output*	Size of the element matrix (=12 for `CBEAM`).

---

void GeometricElementMatrix(double*& geomMat, int& size)

The geometric element matrix is translated into conventional format using **PackedToConventional(...)** and then stored in `geomMat`.

geomMat	*Output*	Pointer to location to which the geometric element matrix is extracted.
size	*Output*	Size of the element matrix (=12 for `CBEAM`).

---

int TemperatureLoadVector(double* loadVector)

Temperature load vector, see (13), is computed. The approach is very similar to the calculation of the element stiffness matrix **BuildElementMatrix(...)**.

loadVector	*Output*	Temperature load vector.

---

int ReadFromFile(ifstream& fin, Domain& domain)

The characteristics of the `CBEAM` are read from the input file and translated into the variables (`Nodes_` and `ZDirection_`) of this object.

fin	*Input*	Input stream of input file
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int WriteOutput(Domain& domain, Solver* solver, ofstream* tempFiles, bool doTitle=*false*)

The `CBEAM` element outputs are written in the temporary files if element outputs are requested by the user. The latter is checked first from the **OutputRequest** object `OutputReq_` of `domain`. Depending on the requested output, element forces and/or element strain energy are calculated using **ElementForces(...)** and **ElementStrainEnergy(...)** respectively.
If `doTitle` is `true`, a title explaining the output data is written over the output.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**.
tempFiles	*Input*	Array of output streams for temporary files.
doTitle	*Input*	If `true`, a specified title is printed.

Private Functions

int MaterialMatrices(double* C00, double* C01, double* C11, double* G0, double* G1, double* S, double* c0, double* c1, double* g0)

The material matrices given in (7)-(10) are computed using two-point Gauss integration rule. These matrices depend on the number of **Properties** (`NumProperties_`) and applied delta temperatures (`NumDeltaTemperatures_`) assigned to this element. Therefore, two interpolation matrices are created using **InterpolateProperties(...)**, one for properties and the other for temperatures.
The matrices in (7)-(9) depend only on element properties. To compute these matrices, an outer loop is performed over all properties and inner loops over the matrix indices. The contributions of the two Gauss integration points are computed within these inner loops based on the interpolation matrix. These contributions are then added to the relevant matrix.
The vectors in (10) depend on both properties and temperatures. To compute these, two outer loops are performed which compute the contributions of properties and temperatures separately. The contributions are then added together.

C00	*Output*	See (7).
C01	*Output*	See (7).
C11	*Output*	See (7).
G0	*Output*	See (8).
G1	*Output*	See (8).
S	*Output*	See (9).
c0	*Output*	See (10).
c1	*Output*	See (10).
g0	*Output*	See (10).

---

int InterpolateProperties(const int nProp, double* interpMat)

The interpolation matrix is computed in this function. Note that this interpolation matrix can be used to interpolate any parameter, not only properties.
The property at the two Gauss integration points is related to the n properties which have been assigned to the element as:

$$\begin{pmatrix} p_{GP1} \\ p_{GP2} \end{pmatrix} = \underbrace{\begin{bmatrix} N_1(\xi_1) & N_2(\xi_1) & \cdots & N_n(\xi_1) \\ N_1(\xi_2) & N_2(\xi_2) & \cdots & N_n(\xi_2) \end{bmatrix}}_{\boldsymbol{P}} \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{pmatrix} \tag{31}$$

in which $N(\xi)$ represents the shape functions. In the implementation it is assumed that the n properties are the 'nodal-like' properties, with the n 'nodes' evenly distributed over the beam element. The interpolation function N_n is then constructed to be a polynomial which equals 1 at node n and zero at all other nodes. As usual, the sum of the interpolation functions equals to 1 everywhere.

Actually, $\boldsymbol{P}^T$ is stored in `interpMat` because C++ is column based.

nProp	*Input*	Number of properties.
interpMat	*Output*	Interpolation matrix $\boldsymbol{P}^T$.

---

int ElementForces(double* elemForces, Solver* solver)

Element forces, see (22) and (23), are computed. The approach is very similar to the calculation of the element stiffness matrix **BuildElementMatrix(...)**.

elemForces	*Output*	Element forces as $[N_x, V_y, V_z, M_x, M_y, M_z]^T$.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the element forces.

---

int ElementStrainEnergy(double& elemEnergy, Solver* solver)

Element strain energy, see (30), is computed. First the element matrix is extracted using **BuildElementMatrix(...)**, the temperature load vector is computed with **TemperatureLoadVector(...)**, and the nodal displacement are extracted from `solver`. The element strain energy is then readily computed.

elemEnergy	*Output*	Element strain energy.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the element strain energy.

CTRIA.h : Element.h

Description

The CTRIA element is a three-noded 2D element, characterized by one ore more **PSHELL** properties. The implementation of this element is based on finite element templates, see [5, 4]. To construct the element stiffness matrix, the DOF are divided into membrane DOF (subscript m) and bending DOF (subscript b):

$$\boldsymbol{u_m} = \begin{pmatrix} u_x \\ u_y \\ \theta_z \end{pmatrix}; \qquad \boldsymbol{u_b} = \begin{pmatrix} u_z \\ \theta_x \\ \theta_y \end{pmatrix} \tag{32}$$

This division is convenient since finite element templates exist for membrane DOF [5] and bending DOF [4] separately. The element stiffness matrix is also divided into a pure membrane part $\boldsymbol{K_{mm}}$, a pure bending part $\boldsymbol{K_{bb}}$ and a coupling part $\boldsymbol{K_{bm}} = \boldsymbol{K_{mb}}^T$. These submatrices are defined as:

$$\boldsymbol{K_{mm}} = \boldsymbol{B_{Lm}}^T \boldsymbol{A} \boldsymbol{B_{Lm}} A + \left(\boldsymbol{B_{4m}}^T \boldsymbol{A} \boldsymbol{B_{4m}} + \boldsymbol{B_{5m}}^T \boldsymbol{A} \boldsymbol{B_{5m}} + \boldsymbol{B_{6m}}^T \boldsymbol{A} \boldsymbol{B_{6m}} \right) \frac{A}{3} \tag{33}$$

$$\boldsymbol{K_{bm}} = \boldsymbol{B_{Lb}}^T \boldsymbol{B} \boldsymbol{B_{Lm}} A + \left(\boldsymbol{B_{4b}}^T \boldsymbol{B} \boldsymbol{B_{4m}} + \boldsymbol{B_{5b}}^T \boldsymbol{B} \boldsymbol{B_{5m}} + \boldsymbol{B_{6b}}^T \boldsymbol{B} \boldsymbol{B_{6m}} \right) \frac{A}{3} \tag{34}$$

$$\boldsymbol{K_{bb}} = \boldsymbol{B_{Lb}}^T \boldsymbol{D} \boldsymbol{B_{Lb}} A + \left(\boldsymbol{B_{4b}}^T \boldsymbol{D} \boldsymbol{B_{4b}} + \boldsymbol{B_{5b}}^T \boldsymbol{D} \boldsymbol{B_{5b}} + \boldsymbol{B_{6b}}^T \boldsymbol{D} \boldsymbol{B_{6b}} \right) \frac{A}{3} \tag{35}$$

with:

$$\boldsymbol{B_{4m}} = \frac{2}{3} \sqrt{\beta_0} \boldsymbol{T_e} \boldsymbol{Q_4} \tilde{\boldsymbol{T}}_{\boldsymbol{\theta u}} \tag{36}$$

$$\boldsymbol{B_{5m}} = \frac{2}{3} \sqrt{\beta_0} \boldsymbol{T_e} \boldsymbol{Q_5} \tilde{\boldsymbol{T}}_{\boldsymbol{\theta u}} \tag{37}$$

$$\boldsymbol{B_{6m}} = \frac{2}{3} \sqrt{\beta_0} \boldsymbol{T_e} \boldsymbol{Q_6} \tilde{\boldsymbol{T}}_{\boldsymbol{\theta u}} \tag{38}$$

in which A is the element area, $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ are the material matrices interpolated to the center of the element, and the other parameters are defined either in [5] for the membrane components or in [4] for the bending components. Note that the BCIZ0 element template [4] is used for the bending part.

The entire element matrix $\boldsymbol{K_e}$ is constructed from $\boldsymbol{K_{mm}}$, $\boldsymbol{K_{bb}}$, and $\boldsymbol{K_{bm}}$ by copying their entries to the correct position int $\boldsymbol{K_e}$.

The temperature force vector is computed as:

$$\boldsymbol{F_T} = \int_A \begin{pmatrix} \boldsymbol{B_{Lm}}^T \boldsymbol{a} \\ \boldsymbol{B_{Lb}}^T \boldsymbol{b} \end{pmatrix} \Delta T dA \tag{39}$$

The integral is evaluated by numerical integration with one integration point in the center of the element. After the nodal displacements (in element coordinates) have been computed, the element strains and curvatures are obtained by:

$$\begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{pmatrix} = \boldsymbol{B_m u_m} \qquad \boldsymbol{B_m} = \boldsymbol{B_{Lm}} \tag{40}$$

$$\begin{pmatrix} \kappa_x \\ \kappa_y \\ \kappa_{xy} \end{pmatrix} = \boldsymbol{B_b u_b} \qquad \boldsymbol{B_b} = \boldsymbol{B_{Lb}} + \frac{1}{3}\left(\boldsymbol{B_{4b}} + \boldsymbol{B_{5b}} + \boldsymbol{B_{6b}}\right) \tag{41}$$

in which u_m and u_b represent the nodal membrane and bending displacement vector respectively. Note that $\boldsymbol{B_m}$ does not depend on $\boldsymbol{B_{4m}}$, $\boldsymbol{B_{5m}}$, and $\boldsymbol{B_{6m}}$, because $\boldsymbol{Q_4} + \boldsymbol{Q_5} + \boldsymbol{Q_6} = \boldsymbol{0}$ [5]. The expression for $\boldsymbol{B_b}$ has not been obtained directly from [4], but is assumed to behave similar to the membrane part.
Once the strains are known, the internal forces in the shell are computed as:

$$\boldsymbol{N} = \boldsymbol{C\varepsilon} - \boldsymbol{c}\Delta T \tag{42}$$

with:

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{B} & \boldsymbol{D} \end{bmatrix}; \qquad \boldsymbol{c} = \begin{pmatrix} \boldsymbol{a} \\ \boldsymbol{b} \end{pmatrix} \tag{43}$$

Finally, the element strain energy ϕ_e is obtained by integrating the dot product of internal forces and elastic strains over the element:

$$\begin{aligned} \phi_e &= \frac{1}{2}\int_A \boldsymbol{N}^T \boldsymbol{\varepsilon^e} dA \qquad \left(\boldsymbol{\varepsilon^e} = \boldsymbol{C}^{-1}\boldsymbol{N} = \boldsymbol{\varepsilon} - \boldsymbol{C}^{-1}\boldsymbol{c}\Delta T\right) \\ &= \frac{1}{2}\int_A \boldsymbol{N}^T\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^T\boldsymbol{c}\Delta T + \boldsymbol{c}^T\boldsymbol{C}^{-1}\boldsymbol{c}\Delta T^2 dA \qquad (44) \\ &= \frac{1}{2}\boldsymbol{u}^T\boldsymbol{K_e}\boldsymbol{u} - \boldsymbol{u}^T\boldsymbol{F_T} + \frac{1}{2}\int_A \boldsymbol{c}^T\boldsymbol{C}^{-1}\boldsymbol{c}\Delta T^2 dA \qquad (45) \end{aligned}$$

Equation (44) is used by OPTANT. Comparison of (45) with (30) shows the additional constant (independent of $\boldsymbol{u}$) value which has been added for the `CTRIA` element. This constant ensures the element strain energy is always equal to or greater than zero.

The geometric matrix for shells, required for buckling analysis, is based on the 'buckling' potential energy $\mathcal{V}$:

$$\mathcal{V} = -\frac{1}{2}\int_A \begin{pmatrix} w_{,x} \\ w_{,y} \end{pmatrix}^T \begin{bmatrix} N_x & N_{xy} \\ N_{xy} & N_y \end{bmatrix} \begin{pmatrix} w_{,x} \\ w_{,y} \end{pmatrix} dA \tag{46}$$

in which $w = u_z$ is the out of plane displacement and N_x, N_y, and N_{xy} are the in-plane forces. To construct the geometric element matrix, the following relation is assumed for w:

$$w = \bar{\boldsymbol{N}}\boldsymbol{u} \tag{47}$$

with $\bar{\boldsymbol{N}}$ representing the standard shape function matrix over a triangular element. Note that $\bar{\boldsymbol{N}}$ is a 1×18 matrix with only three non-zero elements. Applying this definition for w, the geometric element stiffness $\boldsymbol{K_{g,e}}$ is derived from (46) as:

$$\boldsymbol{K_{g,e}} = -A \begin{pmatrix} \bar{\boldsymbol{N}}_{,x} \\ \bar{\boldsymbol{N}}_{,y} \end{pmatrix}^T \begin{bmatrix} N_x & N_{xy} \\ N_{xy} & N_y \end{bmatrix} \begin{pmatrix} \bar{\boldsymbol{N}}_{,x} \\ \bar{\boldsymbol{N}}_{,y} \end{pmatrix} \tag{48}$$

Variables

Node*	**Nodes_[3]**	Element **Nodes**.
double	**MaterialDirection_[3]**	Material direction (e.g. of 0^o ply) in global coordinates with unit length.

Public Functions

CTRIA()

Constructor.

int BuildElementMatrix()

Element matrix is computed. First the transformation between global coordinates X, Y, Z and the element coordinates x, y is computed by means of a transformation matrix Λ based on the material direction. Subsequently the $\boldsymbol{B_i}$ matrices in (33)-(38) are computed by calling **FelippaTriMembrane(...)** and **FelippaTriBending(...)**. The $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices at the center of the element are then computed using **MaterialMatrices(...)**, after which the $\boldsymbol{K_{mm}}$, $\boldsymbol{K_{bb}}$, and $\boldsymbol{K_{bm}}$ matrices are constructed using (33)-(35). These submatrices are copied into the element matrix $\boldsymbol{K_e}$ at their respective positions. The C++ code performing this operation are generated by a separate MATLAB code. The element matrix $\boldsymbol{K_e}$ is finally transformed from element x, y coordinates into global X, Y, Z coordinates and converted to packed storage format using **ConventionalToPacked(...)**.
Note that storage allocation is minimized for buidling the element matrix and many of the matrix multiplications have been hard coded using the **MatrixOperations** namespace.

int BuildGeometricElementMatrix(Solver* solver, bool makeNegative = *false* **)**

Geometric element matrix, see (48), is computed. First the transformation between global coordinates X, Y, Z and the element coordinates x, y is computed by means of a transformation matrix Λ based on the material direction. Subsequently the shape function derivatives are computed and the element forces are obtained using **ElementForces(...)**. The geometric element matrix is then constructed, transformed to global coordinates, multiplied by -1 if required, and stored in packed storage format.

solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the geometric element matrix.
makeNegative	*Input*	If true, the element geometric matrix is multiplied by -1. Used for differential stiffness matrix induced by prestresses for **LinearBucklingSolver**.

---

void ElementMatrix(double*& stiffMat, int& size)

The element matrix is translated into conventional format using **PackedToConventional(...)** and then stored in `stiffMat`.

stiffMat	*Output*	Pointer to location to which the element matrix is extracted.
size	*Output*	Size of the element matrix (=18 for `CTRIA`).

---

void GeometricElementMatrix(double*& geomMat, int& size)

The geometric element matrix is translated into conventional format using **PackedToConventional(...)** and then stored in `geomMat`.

geomMat	*Output*	Pointer to location to which the geometric element matrix is extracted.
size	*Output*	Size of the element matrix (=18 for `CTRIA`).

---

int TemperatureLoadVector(double* loadVector)

Temperature load vector, see (39), is computed. The approach is very similar to the calculation of the element stiffness matrix **BuildElementMatrix(...)**. One of the differences is that only the linear parts of the $\boldsymbol{B_i}$ matrices are computed and the $\boldsymbol{a}, \boldsymbol{b}$ and ΔT at the center of the element are computed using **MaterialTemperatureMatrices(...)**.

loadVector	*Output*	Temperature load vector.

---

static int FelippaTriMembrane(double* storage, bool forStiffnessMatrix)

The $\boldsymbol{B_{Lm}}$, $\boldsymbol{B_{4m}}$, $\boldsymbol{B_{5m}}$, and $\boldsymbol{B_{6m}}$ are computed in this function based on [5]. If `forStiffnessMatrix` is `true` all of these matrices are computed, if not, only $\boldsymbol{B_{Lm}}$ is computed. This is used when in-plane element strains are computed as these only depend on $\boldsymbol{B_{Lm}}$, see (40).

storage	*Output*	Array of allocated storage used for all intermediate calculations.
forStiffnessMatrix	*Input*	If `true` all membrane $\boldsymbol{B_i}$ are computed, if `false` only $\boldsymbol{B_{Lm}}$ is computed.

---

static void FelippaTriBending(double* storage)

The $\boldsymbol{B_{Lb}}$, $\boldsymbol{B_{4b}}$, $\boldsymbol{B_{5b}}$, and $\boldsymbol{B_{6b}}$ are computed in this function based on [4].

storage	*Output*	Array of allocated storage used for all intermediate calculations.

---

int NormalDirection(double* n)

The unit normal direction to the `CTRIA` element is computed in global coordinates. This direction is determined using the right hand rule from d_{12} to d_{13}, with d_{ij} being the vector from **Node** i to

Node j.

n	*Output*	Unit normal direction vector in global X, Y, Z coordinates.

---

int ElementArea(double& area)

The area of the CTRIA element is computed as the absolute value, divided by two, of the cross product between d_{12} and d_{13}. d_{ij} represents the vector from **Node** i to **Node** j.

area	*Output*	Element area.

---

int ReadFromFile(ifstream& fin, Domain& domain)

The characteristics of the CTRIA are read from the input file and translated into the variables (Nodes_ and MaterialDirection_) of this object.

fin	*Input*	Input stream of input file
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int WriteOutput(Domain& domain, Solver* solver, ofstream* tempFiles, bool doTitle=*false*)

The CTRIA element outputs are written in the temporary files if element outputs are requested by the user. The latter is checked first from the **OutputRequest** object OutputReq_ of domain. Depending on the requested output, element strains, element forces and/or element strain energy are calculated using **ElementStrains(...)**, **ElementForces(...)** and **ElementStrainEnergy(...)** respectively.
If doTitle is true, a title explaining the output data is written over the output..

domain	*Input*	Reference to the **Domain.h** object for the FEM model.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**.
tempFiles	*Input*	Array of output streams for temporary files.
doTitle	*Input*	If true, a specified title is printed.

Private Functions

---

int VerifyMaterialDirection(int GlobalElemID)

This function checks whether the material direction specified by the user is (too close to) the tangential direction of the CTRIA element. This check is performed by computing the angle between the specified material direction and the normal direction of the element. If the angle is too small, a warning or error message is provided to the user. The critical angles are defined in **DomainConstants.h**.

GlobalElemID	*Input*	Global element ID, used in the potential warning or error message.

---

int MaterialMatrices(double* Am, double* Bm, double* Dm)

The $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices at the center of the CTRIA element are computed by interpolating the

PSHELL properties assigned to this element. The interpolation matrix is computed using **InterpolateProperties(...)**, after which the contribution of each property is computed by looping over all properties. Finally the matrices are converted to conventional storage format and stored.

Am	*Output*	$\boldsymbol{A}$ matrix at center of element in conventional storage format.
Bm	*Output*	$\boldsymbol{B}$ matrix at center of element in conventional storage format.
Dm	*Output*	$\boldsymbol{D}$ matrix at center of element in conventional storage format.

---

int MaterialTemperatureMatrices(double* am, double* bm, double& dTemp)

The $\boldsymbol{a}, \boldsymbol{b}$ vectors and ΔT at the center of the CTRIA element are computed by interpolating the **PSHELL** properties and ΔT values assigned to this element respectively. Interpolation matrices are computed for both properties and applied temperatures using **InterpolateProperties(...)**. The contribution of each property (applied temperature) is computed by looping over all properties (applied temperatures).

am	*Output*	$\boldsymbol{a}$ vector at center of element.
bm	*Output*	$\boldsymbol{b}$ vector at center of element.
dTemp	*Output*	ΔT at center of element.

---

int InterpolateProperties(const int nProp, double* interpMat)

The interpolation matrix, similar to (31), is computed in this function. Note that this interpolation matrix can be used to interpolate any parameter, not only properties.
Since the CTRIA has only one integration point, the interpolation matrix $\boldsymbol{P}$, see (31), is a row vector of size n. The interpolation is such that all assigned properties get equal weight, and consequently, all entries in $\boldsymbol{P}$ are equal to $1/n$. This means the interpolation is simply the average of all properties.

nProp	*Input*	Number of properties.
interpMat	*Output*	Interpolation matrix $\boldsymbol{P}^T$.

---

int ElementStrains(double* elemStrains, Solver* solver)

Element strains, see (40) and (41), are computed. The approach is very similar to the first part of calculating the element stiffness matrix **BuildElementMatrix(...)**, i.e. up to computation of the $\boldsymbol{B_i}$ matrices. Note that **FelippaTriMembrane** is called with argument `forStiffnessMatrix = false`, since only $\boldsymbol{B_{Lm}}$ is required to compute in-plane strains.

elemStrains	*Output*	Element strains as $[\varepsilon_x, \varepsilon_y, \gamma_{xy}, \kappa_x, \kappa_y, \kappa_{xy}]^T$.
solver	*Input*	Pointer to solver containing the **DisplacementVector**_, which is required to compute the element strains.

---

int ElementForces(double* elemForces, double* elemStrains)

Element forces, see (42), are computed. First the material matrices are computed using **MaterialMatrices(...)** and **MaterialTemperatureMatrices(...)**, after which (42) is applied.

elemForces	*Output*	Element forces as $[N_x, N_y, N_{xy}, M_x, M_y, M_{xy}]^T$.

elemStrains	*Input*	Element strains as $[\varepsilon_x, \varepsilon_y, \gamma_{xy}, \kappa_x, \kappa_y, \kappa_{xy}]^T$.

---

int ElementStrainEnergy(double& elemEnergy, double* elemForces, double* elemStrains)

Element strain energy, see (44), is computed. The first term in the integral of (44) is always calculated, the other terms only if $\Delta T \neq 0$ (which is computed using **MaterialTemperatureMatrices(...)**).

elemEnergy	*Output*	Element strain energy.
elemForces	*Input*	Element forces as $[N_x, N_y, N_{xy}, M_x, M_y, M_{xy}]^T$.
elemStrains	*Input*	Element strains as $[\varepsilon_x, \varepsilon_y, \gamma_{xy}, \kappa_x, \kappa_y, \kappa_{xy}]^T$.

CQUAD.h : Element.h

Description

The CQUAD element is a four-noded 2D element, characterized by one ore more **PSHELL** properties. The implementation of this element is based on splitting each CQUAD element into four triangles as shown in Fig. 2.

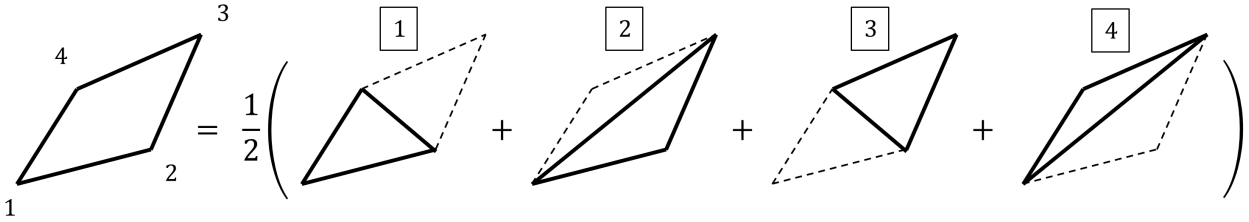

Figure 2: CQUAD element consisting of four triangular elements.

Each triangle is treated as if it were a CTRIA element, however, no CTRIA.h objects are created explicitly in the implementation. The element strains and element forces are computed at the centres of the triangles, i.e. not at the standard two-point Gauss integration points.

Variables

Node*	**Nodes_[4]**	Element **Nodes**.
double	**MaterialDirection_[3]**	Material direction (e.g. of 0^o ply) in global coordinates with unit length.

Public Functions

CQUAD()

Constructor.

---

int BuildElementMatrix()

Element matrix is computed. A loop over the four triangles, see Fig. 2, is performed in which the same procedure, as described in **CTRIA::BuildElementMatrix()**, is performed for each triangle. Note that properties are interpolated to the center of each triangle outside the triangle

loop, since the interpolation matrix contains all four triangles, see **InterpolateProperties(...)**.
The element matrix of each triangle (size 18×18) is assembled into the CQUAD element matrix (size 24×24) at the end of the triangle loop. This assembly is different for each triangle, and the C++ code performing this task has been generated by a separate MATLAB code. Note that the assembly is such that the triangle element matrices in conventional storage format are directly assembled into the CQUAD element matrix in packed storage format.

---

int BuildGeometricElementMatrix(Solver* solver, bool makeNegative = *false* **)**

Geometric element matrix is computed using the same approach as for the CQUAD element matrix, **BuildElementMatrix()**. The procedure described in **CTRIA::BuildGeometricElementMatrix** is performed for each triangle within the triangle loop. The element forces are computed outside this triangle loop because the element force vector contains the element forces at the centre of each triangle, see **ElementForces(...)**.

solver	*Input*	Pointer to solver containing the **DisplacementVector**_, which is required to compute the geometric element matrix.
makeNegative	*Input*	If true, the element geometric matrix is multiplied by -1. Used for differential stiffness matrix induced by prestresses for **LinearBucklingSolver**.

---

void ElementMatrix(double*& stiffMat, int& size)

The element matrix is translated into conventional format using **PackedToConventional(...)** and then stored in stiffMat.

stiffMat	*Output*	Pointer to location to which the element matrix is extracted.
size	*Output*	Size of the element matrix (=24 for CQUAD).

---

void GeometricElementMatrix(double*& geomMat, int& size)

The geometric element matrix is translated into conventional format using **PackedToConventional(...)** and then stored in geomMat.

geomMat	*Output*	Pointer to location to which the geometric element matrix is extracted.
size	*Output*	Size of the element matrix (=24 for CQUAD).

---

int TemperatureLoadVector(double* loadVector)

Temperature load vector is computed by looping over the four triangles, see Fig. 2, and applying the procedure as described in **CTRIA::TemperatureLoadVector(...)** for each triangle. Note that properties and applied temperatures are interpolated to the center of each triangle outside the triangle loop, since the interpolation matrix contains all four triangles, see **InterpolateProperties(...)**.
The temperature force vector of each triangle is assembled into the CQUAD temperature force vector. The C++ code used for the assembly has been generated by a separate MATLAB code.

loadVector	*Output*	Temperature load vector.

---

int NormalDirection(double* n)

The unit normal direction to the CQUAD element is computed in global coordinates. This direction is determined using the right hand rule from d_{13} to d_{24}, with d_{ij} being the vector from **Node** i to **Node** j, see Fig. 2. Note that a CQUAD element is curved in general, which means that the normal direction is not constant over the element. The vector computed in this function can be considered to be an 'average' normal direction of the element.

n	*Output*	Unit normal direction vector in global X, Y, Z coordinates.

---

int ElementArea(double& area)

The area of the CQUAD element is computed by adding the area of each triangle, see Fig. 2, and divide by two. The procedure to compute the area of the triangles is the same as described in **CTRIA::ElementArea(...)**.

area	*Output*	Element area.

---

int ReadFromFile(ifstream& fin, Domain& domain)

The characteristics of the CQUAD are read from the input file and translated into the variables (Nodes_ and MaterialDirection_) of this object.

fin	*Input*	Input stream of input file
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int WriteOutput(Domain& domain, Solver* solver, ofstream* tempFiles, bool doTitle=*false*)

The CQUAD element outputs are written in the temporary files if element outputs are requested by the user. The latter is checked first from the **OutputRequest** object OutputReq_ of domain. Depending on the requested output, element strains, element forces and/or element strain energy are calculated using **ElementStrains(...)**, **ElementForces(...)** and **ElementStrainEnergy(...)** respectively.
If doTitle is true, a title explaining the output data is written over the output..

domain	*Input*	Reference to the **Domain.h** object for the FEM model.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**.
tempFiles	*Input*	Array of output streams for temporary files.
doTitle	*Input*	If true, a specified title is printed.

Private Functions

---

int VerifyMaterialDirection(int GlobalElemID)

This function checks whether the material direction specified by the user is (too close to) the tangential direction of one of the four triangles, see Fig. 2, of the CQUAD element. This check is performed by computing the angle between the specified material direction and the normal direction of the particular triangle. If the angle is too small, a warning or error message is

provided to the user. The critical angles are defined in **DomainConstants.h**.

GlobalElemID	*Input*	Global element ID, used in the potential warning or error message.

---

int MaterialMatrices(double* Am, double* Bm, double* Dm, Property properties, const int& nProp, double* InterpP, const int& triaID)**

The $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{D}$ matrices at the center of triangle `triaID`, see Fig. 2, are computed by interpolating the **PSHELL** properties assigned to this element. This function requires more input parameters than **CTRIA::MaterialMatrices(...)** because the interpolation matrices are created outside the triangle loop in e.g. **BuildElementMatrix()**. However, the procedure to do the interpolation is very similar to **CTRIA::MaterialMatrices(...)**.

Am	*Output*	$\boldsymbol{A}$ matrix at center of triangle `triaID` in conventional storage format.
Bm	*Output*	$\boldsymbol{B}$ matrix at center of triangle `triaID` in conventional storage format.
Dm	*Output*	$\boldsymbol{D}$ matrix at center of triangle `triaID` in conventional storage format.
properties	*Input*	Array of pointers to **Properties** assigned to this element.
nProp	*Input*	Number of **Properties** assigned to this element.
InterpP	*Input*	Property interpolation matrix $\boldsymbol{P}^T$.
triaID	*Input*	Triangle ID, see Fig. 2.

---

int MaterialTemperatureMatrices(double* am, double* bm, double& dTemp Property properties, const int& nProp, double* InterpP, double* temp, const int& nTemp, double* InterpT, const int& triaID,)**

The $\boldsymbol{a}, \boldsymbol{b}$ vectors and ΔT at the center of triangle `triaID`, see Fig. 2, are computed by interpolating the **PSHELL** properties and ΔT values assigned to this element respectively. This function requires more input parameters than **CTRIA::MaterialTemperatureMatrices(...)** because the interpolation matrices are created outside the triangle loop in e.g. **BuildElementMatrix()**. However, the procedure to do the interpolation is very similar to **CTRIA::MaterialTemperatureMatrices(...)**.

am	*Output*	$\boldsymbol{a}$ vector at center of triangle `triaID`.
bm	*Output*	$\boldsymbol{b}$ vector at center of triangle `triaID`.
dTemp	*Output*	ΔT at center of triangle `triaID`.
properties	*Input*	Array of pointers to **Properties** assigned to this element.
nProp	*Input*	Number of **Properties** assigned to this element.
InterpP	*Input*	Property interpolation matrix $\boldsymbol{P}^T$.
temp	*Input*	Array of applied ΔT's.
nTemp	*Input*	Number of applied ΔT's.
InterpT	*Input*	Temperature interpolation matrix $\boldsymbol{P_T}^T$.
triaID	*Input*	Triangle ID, see Fig. 2.

---

int InterpolateProperties(const int nProp, double* interpMat)

The interpolation matrix, similar to (31), is computed in this function. Note that this interpolation matrix can be used to interpolate any parameter, not only properties.
The interpolation matrix $\boldsymbol{P}$ for the `CQUAD` element is a $4 \times n$ matrix, in which each row represents one of the four triangles. These rows are the same as described in **CTRIA::InterpolateProperties(...)**.

nProp	*Input*	Number of properties.
interpMat	*Output*	Interpolation matrix $\boldsymbol{P}^T$.

---

int ElementStrains(double* elemStrains, Solver* solver)

Element strains at the center of each triangle, see Fig. 2, are computed as well as the average of the four triangles. The element strains in one triangle is computed as described in **CTRIA::ElementStrains(...)**.
The output of this function is a 6×5 matrix, in which the first four columns contain the strains at the center of the triangles and the fifth column is the average of the first four columns.

elemStrains	*Output*	6×5 element strain matrix, with each column as $[\varepsilon_x, \varepsilon_y, \gamma_{xy}, \kappa_x, \kappa_y, \kappa_{xy}]^T$.
solver	*Input*	Pointer to solver containing the **DisplacementVector_**, which is required to compute the element strains.

---

int ElementForces(double* elemForces, double* elemStrains)

Element forces at the center of each triangle, see Fig. 2, are computed as well as the average of the four triangles. The element forces in one triangle is computed as described in **CTRIA::ElementForces(...)**.
The output of this function is a 6×5 matrix, in which the first four columns contain the forces at the center of the triangles and the fifth column is the average of the first four columns.

elemForces	*Output*	6×5 element force matrix, with each column as $[N_x, N_y, N_{xy}, M_x, M_y, M_{xy}]^T$.
elemStrains	*Input*	6×5 element strain matrix, with each column as $[\varepsilon_x, \varepsilon_y, \gamma_{xy}, \kappa_x, \kappa_y, \kappa_{xy}]^T$.

---

int ElementStrainEnergy(double& elemEnergy, double* elemForces, double* elemStrains)

Element strain energy in the CQUAD element is computed by looping over the four triangles, see Fig. 2. The strain energy contribution of one triangle is computed as described in **CTRIA::ElementStrainEnergy(...)**.

elemEnergy	*Output*	Element strain energy.
elemForces	*Input*	6×5 element force matrix, with each column as $[N_x, N_y, N_{xy}, M_x, M_y, M_{xy}]^T$.
elemStrains	*Input*	6×5 element strain matrix, with each column as $[\varepsilon_x, \varepsilon_y, \gamma_{xy}, \kappa_x, \kappa_y, \kappa_{xy}]^T$.

SPC.h

Description

Objects from the SPC.h class are used to apply single point constraints to the FEM model. An SPC is characterized by a **Node**, one or more constrained DOF, and the prescribed value of the constraint.

Variables

Node*	**Nodes_**	**Node** at which SPC is applied.
int	**PrescribedDOF_[6]**	Array in which each entry represents a nodal DOF. A value of 1 in the array means the corresponding DOF is constrained, a value of 0 means the corresponding DOF is free.
double	**PrescribedValue_**	Prescribed value of constrained DOF.

Public Functions

SPC(Node* spcNode, const char* preDOF, double preVal)

Constructor. The variables of the SPC.h object are initialized with the input arguments. The character array preDOF contains the prescribed DOF as characters between '1' and '6'. This character array is transformed into the format of the class variable PrescribedDOF_

spcNode	*Input*	**Node** at which SPC is applied.
preDOF	*Input*	Character array containing the prescribed DOF as written in the input file [2].
preVal	*Input*	Prescribed value of constrained DOF.

int Initialize(fei::MatrixGraph* matrixGraph)

The SPC is initialized in the matrix graph, similar to **MPC** constraints, using the initSlaveConstraint(...) function from the FEI package [10]. However, due to a limitation of the FEI package, this function cannot be used to initialize the SPC object in the matrix graph. Therefore, this function is empty in the implementation!

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The matrixGraph also handles all initialized **MPC** in the background.

int Apply(fei::LinearSystem* linSys)

The SPC is applied to the linear system linSys. Within the FEI package, this means that the rows and columns of the constrained DOF in the global stiffness matrix are set to 0, except for the diagonal terms which are set to 1. The entry of the constrained DOF in the force vector is set to PrescribedValue_.
The implementation of this function in OPTANT exists of a loop over the prescribed DOF of this SPC. The constraints are then applied using the linSys->loadEssentialBCs(...) function from the FEI package.

linSys	*Input*	Pointer to the linear system, which is an FEI related object containing the stiffness matrix, displacement vector, and force vector.

MPC.h

Description

Multiple point constraints are applied to the FEM model using `MPC.h` objects. An `MPC` relates a slave DOF to one or more master DOF and a constant value as:

$$u_S + C_0 = \sum_i w_{M,i} u_{M,i} \tag{49}$$

in which, u_S is the slave DOF, C_0 is the constant value, and $w_{M,i}$ are the weights for the master DOF $u_{M,i}$. Note that the implementation of the constant C_0 is not performed correctly within the FEI package, which is why the user should always use $C_0 = 0$.

Variables

int	**NumNodes_**	Total number of DOF participating in (49), i.e. one plus number of master DOF.
Node**	**MPCNodes_**	Array of length `NumNodes_` containing pointers to the **Nodes** related to the participating DOF.
int*	**PrescribedDOF_**	Array of length `NumNodes_` containing the nodal DOF (0-5) of all participating DOF.
double*	**Weights_**	Array of length `NumNodes_` containing the weights w, see (49), of all participating DOF. Note that the weight of the slave dof equals -1.
double	**RHSConstant_**	Value of C_0, see (49).
int*	**nodeIDs_**	Array of length `NumNodes_` containing the local node IDs of all **Nodes** in `MPCNodes_`.
vector <int>	**uniqueNodeIDs_**	Array of length `uniqueNodeIDsCounter_` containing the unique local node IDs of `nodeIDs_`.
unordered_map <int,int>	**nodeMap_**	Map which relates each node ID in `nodeIDs_` (key) to its index in `uniqueNodeIDs_` (value).
int	**uniqueNodeIDsCounter_**	Number of unique local node IDs.

Public Functions

MPC(int numNodes, Node mpcNodes, int* prescribedDOF, double* weights, double rhsConstant)**

Constructor. All variables are initialized.

numNodes	*Input*	Initialization of `NumNodes_`.
mpcNodes	*Input*	Initialization of `MPCNodes_`.
prescribedDOF	*Input*	Initialization of `PrescribedDOF_`.
weights	*Input*	Initialization of `Weights_`.
rhsConstant	*Input*	Initialization of `RHSConstant_`.

~MPC()

Destructor. The variable `MPCNodes_` is deleted as this is a double pointer.

---

int Initialize(fei::MatrixGraph* matrixGraph)

The `MPC` is applied to the matrix graph. Within the FEI package, this means that the stiffness matrix and force vector based on this matrix graph are reduced in size to incorporate the `MPC` [10]. After solving the linear system, the slave DOF are automatically computed based on the applied `MPC`. This also means that accessing the solution is the same with or without `MPC` applied. The OPTANT implementation of this function starts by creating an alternative `weights` vector of size (`uniqueNodeIDsCounter_ * 6`), which contains an entry for all 6 DOF of each participating **Node** in the `MPC`. The entries from `Weights_` are substituted at their respective positions in `weights`. Then, this vector is used together with other class variables to initialize the `MPC` using the `matrixGraph ->initSlaveConstraint(...)` function from the FEI package.

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **MPC** in the background.

LOAD.h

Description

A LOAD.h object represents a nodal load applied to the FEM model; it is characterized by a **Node** and force vector.

Variables

Node*	**Node_**	**Node** at which LOAD is applied.
double	**LoadVector_[6]**	Load vector as $[f_X, f_Y, f_Z, m_X, m_Y, m_Z]^T$

Public Functions

LOAD(Node* node, double* loadvec)

Constructor. All variables are initialized.

node	*Input*	Initialization of Node_.
loadvec	*Input*	Initialization of LoadVector_.

---

int Initialize(Domain* domain)

EMPTY.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int Apply(fei::MatrixGraph* matrixGraph, fei::Vector* rhsVec)

The LOAD is applied to the global force vector rhsVec. The matrix graph is used to find the indices in the vector related to Node_. LoadVector_ is the summed into rhsVec using the rhsVec->sumIn(...) function from the FEI package.

matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The matrixGraph also handles all initialized **MPC** in the background.
rhsVec	*Input*	Pointer to force vector in which the loads are applied.

PLOAD.h

Description

An object of the PLOAD.h class represents a pressure load acting on one or more **Elements**. A PLOAD object is characterized by the pressure and subjected elements.

Variables

int	**NumPressElements_**	Number of **Elements** subjected to this PLOAD.
Element**	**PressElements_**	Array of pointers to **Elements** subjected to this PLOAD.
double	**Pressure_**	Pressure.

Public Functions

PLOAD(int numPressElements, Element pressElem, double pressure)**

Constructor. All variables are initialized.

numPressElements	*Input*	Initialization of NumPressElements_.
pressElem	*Input*	Initialization of PressElements_.
pressure	*Input*	Initialization of Pressure_.

~PLOAD()

Destructor. The variable PressElements_ is deleted as this is a double pointer.

---

int Initialize(Domain* domain)

EMPTY.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int Apply(Domain* domain, fei::MatrixGraph* matrixGraph, fei::Vector* rhsVec)

The PLOAD is applied to the global force vector rhsVec. A loop is performed over all **Elements** in PressElements_. Within this loop it is first verified whether the element is a shell element.

A warning is provided to the user if this is not the case. If the verification has succeeded, the normal direction and the area of the element are computed using **`NormalDirection(...)`** and **`ElementArea(...)`**. The pressure force is evenly distributed over all element **`Nodes`**. Each nodal force is summed into `rhsVec` using the `rhsVec->sumIn(...)` from the FEI package. Note that the matrix graph is used to find the indices in `rhsVec` related to each `Node_`.

`domain`	*Input*	Reference to the **`Domain.h`** object for the FEM model.
`matrixGraph`	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **`MPC`** in the background.
`rhsVec`	*Input*	Pointer to force vector in which the loads are applied.

TEMP.h

Description

A TEMP object is used to apply a temperature loading to an **Element** in the FEM model using an applied change in temperature ΔT. One TEMP object is used to subject one element to one or more applied ΔT. If more than one value for ΔT are specified, the applied temperature is interpolated within the element using **InterpolateProperties(...)**.

Variables

Element*	**Element_**	**Element** to which TEMP load is applied.
int	**NumDeltaTemperatures_**	Number of applied ΔT values for this TEMP load.
double*	**DeltaTemperatures_**	Array of applied ΔT values.

Public Functions

TEMP(Element* element, int NumTemperatures, double* deltaTemp)

Constructor. All variables are initialized.

element	*Input*	Initialization of Element_.
NumTemperatures	*Input*	Initialization of NumTemperatures.
deltaTemp	*Input*	Initialization of DeltaTemperatures_.

---

int Initialize(Domain* domain)

EMPTY.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int Apply(Domain* domain, fei::MatrixGraph* matrixGraph, fei::Vector* rhsVec)

The TEMP is applied to the global force vector rhsVec. First the **NumDeltaTemperatures_** and **DeltaTemperatures_** variable of Element_ are set equal to the corresponding variables of this TEMP object. The temperature load vector is then computed using **Element_->TemperatureLoadVector(...)**. A loop over all **Nodes** of Element_ is then performed in which the corresponding nodal temperature force is summed into rhsVec using the

`rhsVec->sumIn(...)` from the FEI package. Note that the matrix graph is used to find the indices in `rhsVec` related to each `Node_`.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.
matrixGraph	*Input*	Pointer to the matrix graph, which is FEI related object representing the location of all non-zero entries in a matrix. The `matrixGraph` also handles all initialized **MPC** in the background.
rhsVec	*Input*	Pointer to force vector in which the loads are applied.

LoadCase.h

Description

A LoadCase object defines a combination of sets of **SPC**, **MPC**, **LOAD**, **PLOAD** and **TEMP** objects, which together form one load case to be solved for. Each **SPC**, **MPC**, etc., belongs to a set as specified in the input file [2] and as stored in the variables SPCList, MPCList, etc., of the Domain.h object. A LoadCase object is simply characterized by the set ID for each of these constraint/loading types. Note that a set ID is set equal to 0 if the corresponding constraint/loading type is not used in this LoadCase.
Each LoadCase is solved consecutively, and the outputs are all written in the same output file. The number of LoadCase objects is thus the same as the total number of **Solvers** created.

Variables

int	**SPCSetID_**	Global ID of activated **SPC** set.
int	**MPCSetID_**	Global ID of activated **MPC** set.
int	**LOADSetID_**	Global ID of activated **LOAD** set.
int	**PLOADSetID_**	Global ID of activated **PLOAD** set.
int	**TEMPSetID_**	Global ID of activated **TEMP** set.

Public Functions

LoadCase(int spc, int mpc, int load, int pload, int temp)

Constructor. All variables are initialized.

spc	*Input*	Initialization of SPCSetID_.
mpc	*Input*	Initialization of MPCSetID_.
load	*Input*	Initialization of LOADSetID_.
pload	*Input*	Initialization of PLOADSetID_.
temp	*Input*	Initialization of TEMPSetID_.

---

int Initialize(Domain* domain)

EMPTY.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.

---

int Apply(Domain* domain)

EMPTY.

domain	*Input*	Reference to the **Domain.h** object for the FEM model.

Solver.h

Description

The Solver.h class is one of the key classes in OPTANT together with the **Domain.h** class. Solver.h is a base class for **LinearStaticSolver**, which itself is a base class for **LinearBucklingSolver**. A **LinearStaticSolver** or **LinearBucklingSolver** is created for each **LoadCase** in the main loop of OPTANT.
A Solver object is characterized by a pointer to the **Domain** object, by the global **LoadCase** ID, and by high level parameters required by the FEI package. These parameters are all used by the base classes of Solver.h.

Variables

Domain*	**DomainPtr_**	Pointer to the **Domain.h** object for the FEM model.
int	**GlobalLoadCaseID_**	Global **LoadCase** ID.
const char*	**OutputFile_**	Name of output file.
bool	**verbose_**	True for only one of the processors. Used to execute certain tasks (e.g. writing output or **solving** linear buckling problem) on one processor only.
fei::SharedPtr <fei::Factory>	**Factory_**	Pointer to the fei::Factory object, which is an FEI related object containing all other FEI related objects. Its purpose is to handle e.g. multiple processors.
fei::SharedPtr <fei:: VectorSpace>	**NodeSpace_**	Pointer to the vector space, which is an FEI related object representing the displacement space of the model. The vectorSpace contains the definition of e.g. the node type and its size. This vectorSpace is used to construct the fei::MatrixGraph object.
fei::SharedPtr <fei:: MatrixGraph>	**MatrixGraph_**	Pointer to the matrix graph, which is an FEI related object representing the location of all non-zero entries in a matrix. The matrixGraph also handles all initialized **MPC** in the background.

Public Functions

Solver(Domain* domPtr, int globalLoadCaseID, const char* outputFile, bool verbose)

Constructor. Some variables are initialized.

domPtr	*Input*	Initialization of DomainPtr_.
globalLoadCaseID	*Input*	Initialization of GlobalLoadCaseID_.
outputFile	*Input*	Initialization of OutputFile_.
verbose	*Input*	Initialization of verbose_.

---

virtual ~Solver()

Virtual destructor. The factory is reset using Factory_.reset() from the FEI package.

---

virtual int Initialize(MPI_Comm& comm)

Virtual function for initializing the Solver, which is achieved in two steps. The first step is to create and initialize Factory_. The initialization of Factory_ consists of assigning Amesos [9] as solver and assigning the parameters in SolverParams of the **Domain.h** object.
The second step is to create and initialize NodeSpace_ and MatrixGraph_. The solver parameters in SolverParams are assigned to both variables, and the node type is defined in NodeSpace_ based on the variables in **DomainConstants.h**. The element connectivities and constraints are initialized to MatrixGraph_ using **DomainPtr_-> InitializeElementConnectivities(...)** and **DomainPtr_-> InitializeConstraints(...)** respectively.
Note that everything in this function is done on all processes, except for initializing the element connectivities and constraints.
This function is also implemented in **LinearStaticSolver.h** and **LinearBucklingSolver.h**

comm	*Input*	Basic object to handle communication among processes.

---

virtual int Prepare() = 0

Pure virtual function for preparing the Solver. This function is implemented in **LinearStaticSolver.h** and **LinearBucklingSolver.h**

---

virtual int Solve() = 0

Pure virtual function for solving the Solver. This function is implemented in **LinearStaticSolver.h** and **LinearBucklingSolver.h**

---

virtual int WriteOutput(bool writeLoadCaseInfo = *true* **)**

Virtual function for writing output for the Solver. The general information about the **LoadCase** of this Solver is written to the output file.
This function is also implemented in **LinearStaticSolver.h** and **LinearBucklingSolver.h**

writeLoadCaseInfo	*Input*	If true, general info about the load case is written to the output file. This variable is only used in the child classes of Solver.h.

LinearStaticSolver.h : Solver.h

Description

A `LinearStaticSolver` object is used to solve a general linear static FEM problem of the type:

$$\boldsymbol{K}\boldsymbol{u} = \boldsymbol{F} \tag{50}$$

An object of this class is characterized by the stiffness matrix $\boldsymbol{K}$, the displacement vector $\boldsymbol{u}$, the force vector $\boldsymbol{F}$, the linear system containing (50), and the `fei::Solver` solving the linear system.

Variables

fei::SharedPtr <fei::Matrix>	**StiffnesMatrix_**	Pointer to global stiffness matrix $\boldsymbol{K}$.
fei::SharedPtr <fei::Vector>	**DisplacementVector_**	Pointer to displacement vector $\boldsymbol{u}$.
fei::SharedPtr <fei::Vector>	**ForceVector_**	Pointer to force vector $\boldsymbol{F}$.
fei::SharedPtr <fei::LinearSystem>	**LinSys_**	Pointer to linear system in (50).
fei::SharedPtr <fei::Solver>	**Solver_**	Pointer to solver of (50).

Public Functions

LinearStaticSolver(Domain* domPtr, int globalLoadCaseID, const char* outputFile, bool verbose)

Constructor.

domPtr	*Input*	Initialization of `DomainPtr_` of base **Solver** object.
globalLoadCaseID	*Input*	Initialization of `GlobalLoadCaseID_` of base **Solver** object.
outputFile	*Input*	Initialization of `OutputFile_` of base **Solver** object.
verbose	*Input*	Initialization of `verbose_` of base **Solver** object.

---

virtual ~LinearStaticSolver()

Virtual destructor. The solver is reset using `Solver_.reset()` from the FEI package.

---

virtual int Initialize(MPI_Comm& comm)

Solver::Initialize(comm) is called. This function is also implemented in **LinearBucklingSolver.h**.

comm	*Input*	Basic object to handle communication among processes.

---

virtual int Prepare()

Virtual function for preparing the `LinearStaticSolver`, which is achieved in two steps. The first step is to compute the element matrices using **DomainPtr_->BuildElementMatrices()**. These computations are performed on only one processor, and only if **ElementMatricesBuilt** is `false`.
The second step is to set up the linear system of equation. In this step the `StiffnesMatrix_`, `DisplacementVector_`, `ForceVector_`, and `LinSys_` are created on all processes and the first three are assigned to `LinSys_`. The stiffness matrix is assembled using **DomainPtr_-> AssembleStiffnessMatrix(...)** and the loads and constraints are applied to the linear system using **DomainPtr_-> ApplyConstraints(...)** and **DomainPtr_-> ApplyLoads(...)** respectively. This assembly and application of constraints/loads is performed on only one processor after which the FEI package handles sharing it over all processors. The latter is accomplished by calling `LinSys_-> loadComplete()` on all processes.
This function is also implemented in **LinearBucklingSolver.h**.

---

virtual int Solve()

Virtual function for solving the `LinearStaticSolver`. First the solver name (`MUMPS`) is extracted from `SolverParams` of the **Domain.h** object. The `fei::Solver Solver_` is then created based on this solver name. Finally the linear system is solved using `Solver_-> solve(...)` from the FEI package.
This function is also implemented in **LinearBucklingSolver.h**.

---

virtual int WriteOutput(bool writeLoadCaseInfo = *true* **)**

Virtual function for writing output of the `LinearStaticSolver`. First **Solver::WriteOutput(...)** is called if `writeLoadCaseInfo` is `true`. Second, **DomainPtr_->WriteOutput(...)** is called on only one of the processes.
This function is also implemented in **LinearBucklingSolver.h**.

writeLoadCaseInfo	*Input*	If `true`, general info about the load case is written to the output file.

LinearBucklingSolver.h : LinearStaticSolver.h

Description

A `LinearBucklingSolver` object is used to solve a general linear buckling FEM problem, which is characterized by the eigenvalue problem:

$$[(\boldsymbol{K} + \boldsymbol{K_d}) - \lambda \boldsymbol{K_g}]\, \boldsymbol{u} = \boldsymbol{0} \tag{51}$$

in which $\boldsymbol{K}$ is the stiffness matrix, $\boldsymbol{K_d}$ is the differential stiffness matrix due to prestresses, and $\boldsymbol{K_g}$ is the geometric stiffness matrix. $\boldsymbol{K_d}$ is computed as $-\boldsymbol{K_{g,prestress}}$, i.e. it is the geometric stiffness of the prestresses multiplied by minus one.
Existing algorithms for solving eigenvalue problems are most efficient in computing the largest eigenvalues, whereas the lowest eigenvalues are of interest in linear buckling analysis. Therefore, (52) is solved instead of (51):

$$\left[\boldsymbol{K_g} - \frac{1}{\lambda}\left(\boldsymbol{K} + \boldsymbol{K_d}\right)\right] \boldsymbol{u} = \boldsymbol{0} \tag{52}$$

ARPACK++ [6] is used by OPTANT to solve the eigenvalue problem. Solving (52) directly using this package resulted in erroneous results or in non-converged results. Therefore, OPTANT actually solves the standard eigenvalue problem:

$$\left[\left(\boldsymbol{K} + \boldsymbol{K_d}\right)^{-1} \boldsymbol{K_g} - \frac{1}{\lambda}\boldsymbol{I}\right] \boldsymbol{u} = \boldsymbol{0} \tag{53}$$

This eigenvalue problem results in a different normalization of the eigenvalues and, theoretically, in poorer convergence than (52).
The most likely cause of the issue with solving (52) is a bug within ARPACK++, since the implementation in OPTANT is almost identical for (52) and (53). The only difference is the function **`ArpackOperations::OpB(...)`**, which has been verified extensively.

The geometric stiffness matrix $\boldsymbol{K_g}$ is determined by solving the **`LinearStaticSolver`** first and computing the element geometric matrices from the resulting normal forces. The differential stiffness matrix $\boldsymbol{K_d}$ is only computed if a prestress **`LoadCase`** has been specified in the input file [2]. In this case, an additional **`LinearStaticSolver`** is created, in which the prestress **`LoadCase`** is solved. The normal forces resulting from this solve are used to computed $\boldsymbol{K_d}$.
A `LinearBucklingSolver` object is mainly characterized by the number of requested

buckling modes, the geometric stiffness matrix $\boldsymbol{K_g}$, the prestress **LinearStaticSolver**, and the ARPACK eigenvalue problem. Note that if the prestress load case ID is the same as the buckling load case ID, no calculations are performed and OPTANT immediately skips to the next **LoadCase**.

Variables

int	**nModes_**	Number of requested buckling modes.
fei::SharedPtr <fei::Matrix>	**GeomStiffnesMatrix_**	Pointer to global geometric stiffness matrix $\boldsymbol{K_g}$.
fei::SharedPtr <fei::Vector>	**GeomDisplacementVector_**	Pointer to 'geometric' displacement vector. Only used to construct `GeomLinSys_`.
fei::SharedPtr <fei::Vector>	**GeomForceVector_**	Pointer to 'geometric' force vector. Used to obtain random initial vector for buckling analysis with all **SPC** and **MPC** satisfied.
fei::SharedPtr <fei::LinearSystem>	**GeomLinSys_**	Pointer to 'geometric' linear system. Used to apply all **SPC** and **MPC** to `GeomForceVector_`.
Solver*	**psSolver_**	Pointer to presstress **Solver**.
ARSymStdEig <double,Arpack-Operations>*	**ProbSymStdEigB_**	Pointer to ARPACK++ eigenvalue problem. This object contains all eigenvalues and eigenvectors after the linear buckling problem has been solved.
int	**nconvStdB_**	Number of converged eigenmodes of `ProbSymStdEigB_`, which is an output of solving the linear buckling problem.

Public Functions

LinearBucklingSolver(Domain* domPtr, int globalLoadCaseID, int gPSLoadCaseID, const char* outputFile, bool verbose)

Constructor. If `gPSLoadCaseID` is non-zero, `psSolver_` is set to a new **LinearStaticSolver** object, otherwise `psSolver_` is set to `NULL`.

domPtr	*Input*	Initialization of `DomainPtr_` of base **Solver** object.
globalLoadCaseID	*Input*	Initialization of `GlobalLoadCaseID_` of base **Solver** object.
gPSLoadCaseID	*Input*	Global **LoadCase** ID of presstress load case. `gPSLoadCaseID` equals 0 if no prestress is applied.
outputFile	*Input*	Initialization of `OutputFile_` of base **Solver** object.
verbose	*Input*	Initialization of `verbose_` of base **Solver** object.

~LinearBucklingSolver()

Destructor. `psSolver_` is deleted if it is not a `NULL`-pointer.

---

int Initialize(MPI_Comm& comm)

Solver::Initialize(comm) is called and, if `psSolver_` is not a `NULL`-pointer, **psSolver_**

->Initialize(comm) is also called.

comm	*Input*	Basic object to handle communication among processes.

---

int Prepare()

The LinearBucklingSolver is prepared in several steps. First of all, **Solver::WriteOutput()** is called to write general load case input to the output file. This step is already performed in this preparation function to handle the presstress load case properly.
If psSolver_ is not a NULL-pointer, the next step is to perform **psSolver_-> Prepare()**, **psSolver_-> Solve()** and **psSolver_-> WriteOutput(false)**. The geometric element matrices for the prestress output are then computed using **DomainPtr_-> BuildGeometricElementMatrices (psSolver_,true)**. These matrices are the element $\boldsymbol{K_{d,e}}$ matrices of (51). The applied temperatures at all **Elements** are reset using **DomainPtr_-> ResetElementTemperatures()**.
Before assembling the $\boldsymbol{K_{d,e}}$ matrices into **StiffnesMatrix_**, first **LinearStaticSolver:: Prepare()** and **LinearStaticSolver:: Solve()** have to be performed to solve the linear static problem of the buckling case forces.
Next, if psSolver_ is not a NULL-pointer, the $\boldsymbol{K_{d,e}}$ matrices are assembled into **StiffnesMatrix_**. This means that $\boldsymbol{K} + \boldsymbol{K_d}$, which is required for buckling analysis, is now stored in **StiffnesMatrix_**. The constraints have to be applied to **LinSys_** again to ensure that the updated StiffnesMatrix_ incorporates the constraints correctly.
The geometric element matrices resulting from the **LinearStaticSolver:: Solve()**, are now computed. The 'geometric' linear system GeomLinSys_ is then set up similarly to **LinearStaticSolver:: Prepare()**. The 'geometric' force vector is created using **InitialVector(...)**, and the **SPC** and **MPC** are applied to this force vector by calling GeomLinSys_-> loadComplete().

---

int Solve()

The LinearBucklingSolver is solved in a few steps. First the total number of DOF (master and slave) is computed and shared among all processes. An **ArpackOperations** object is then created, in which the operation of multiplying by $(\boldsymbol{K} + \boldsymbol{K_d})^{-1}\boldsymbol{K_g}$ is defined in **OpBiA(...)**.
The eigenvalue problem is solved on only one processor since ARPACK++ is a serial code. An ARPACK++ symmetric standard eigenvalue problem ARSymStdEig [6] is created using GeomForceVector_ as initial vector.
After the eigenvalue problem has been solved it is checked whether all eigenvalues are positive. If this is not the case, the $\boldsymbol{K} + \boldsymbol{K_d}$ matrix was not positive definite and OPTANT will provide an error message to the user. Physically this means that the prestresses already caused buckling.

---

int WriteOutput(bool writeLoadCaseInfo = *true* **)**

The output of the LinearBucklingSolver is written to the output file. Note that the general information about the **LoadCase** and the results of the prestress load case have already been written to the output file in **Prepare()**. Therefore, only **DomainPtr_-> WriteLinearBucklingOutput(...)** is called in this function.

writeLoadCaseInfo	*Input*	If true, general info about the load case is written to the output file.

Private Functions

`void InitialVector( int n, double* initVec )`

An initial vector of length `n` is created in this function and stored in `initVec`. A normal distribution with a mean of 0 and standard deviation of 1 is applied in this function.

`n`	*Input*	Length of vector.
`initVec`	*Output*	Random initial vector.

ArpackOperations.h

Description

The ArpackOperations.h class is essential for solving the linear bucking problem of (52). For brevity and to be consistent with the notation of ARPACK++, this equation is rewritten as:

$$\left(\boldsymbol{A} - \tilde{\lambda}\boldsymbol{B}\right)\boldsymbol{x} = \boldsymbol{0} \tag{54}$$

with:

$$\boldsymbol{A} = \boldsymbol{K_g} \tag{55}$$
$$\boldsymbol{B} = \boldsymbol{K} + \boldsymbol{K_d} \tag{56}$$
$$\tilde{\lambda} = \frac{1}{\lambda} \tag{57}$$
$$\boldsymbol{x} = \boldsymbol{u} \tag{58}$$

ARPACK++ requires two operations to solve the eigenvalue problem, which are **OpB(double*v, double*w)** and **OpBiA(double*v, double*w)**. These function multiply v by $\boldsymbol{B}$ and $\boldsymbol{B}^{-1}\boldsymbol{A}$ respectively and store the result in w.
To perform the multiplication by $\boldsymbol{B}^{-1}\boldsymbol{A}$ the Cholesky decomposition of $\boldsymbol{B}$ is required. This decomposition is performed using the CHOLMOD package [3]. CHOLMOD requires the $\boldsymbol{B}$ matrix to be converted into Compressed Column Storage (CCS) format, which cannot be achieved from the fei::Matrix format directly. Therefore, the fei::Matrix is first converted into an Epetra_CrsMatrix [1], which is then converted into CCS format.

An ArpackOperations object is created in each **LinearBucklingSolver**. Such an object is mainly characterized by the global stiffness matrix, the global geometric matrix, and the total number of DOF.

Variables

fei::SharedPtr <fei::Matrix>	**StiffnessMatrix_**	Pointer to global stiffness matrix $\boldsymbol{K} + \boldsymbol{K_d} = \boldsymbol{B}$.
fei::SharedPtr <fei::Matrix>	**GeometricStiffnessMatrix_**	Pointer to global geometric stiffness matrix $\boldsymbol{K_g} = \boldsymbol{A}$.
fei::SharedPtr <fei::Vector>	**x_**	Pointer to 'displacement'-like vector used to store intermediate results.

[1] Epetra is another package from the Trilinos Project

fei::SharedPtr <fei::Vector>	y_	Pointer to 'force'-like vector used to store intermediate results.
fei::SharedPtr <fei::Factory>	Factory_	Pointer to the fei::Factory object, which is an FEI related object containing all other FEI related objects. Its purpose is to handle e.g. multiple processors.
Epetra_CrsMatrix*	EpetraStiffnessMatrix_	Pointer to global stiffness matrix in sparse Epetra format.
Epetra_CrsMatrix*	EpetraGeometricStiffnessMatrix_	Pointer to global geometric stiffness matrix in sparse Epetra format. (NOT USED!!)
Epetra_MultiVector*	EpetraSolution_	Pointer to solution multi-vector in sparse Epetra format. The Epetra_MultiVector may contain more than one vector, however, in the implementation it always contains only one vector.
Epetra_MultiVector*	EpetraForce_	Pointer to force vector in sparse Epetra format. The Epetra_MultiVector may contain more than one vector, however, in the implementation it always contains only one vector.
cholmod_common	Chol_	The cholmod_common object contains the parameter, statistics, and workspace for the CHOLMOD routines [3].
cholmod_factor*	CholFactorization_	Cholesky factorization of the global stiffness matrix.
cholmod_dense*	CholSolution_	Solution vector of linear system solved using CholFactorization_.
cholmod_dense*	CholForce_	Force vector of linear system solved using CholFactorization_.
bool	StiffMatIsPosDef_	If true, the stiffness matrix is positive definite, if false it is not.
int	ndof_	Total number of DOF (masters and slaves).
bool	mpcPresent_	If true, **MPC** are present. This parameter is determined within the constructor of an ArpackOperations.h object.

Public Functions

ArpackOperations(fei::SharedPtr <fei::Factory> fact, fei::SharedPtr <fei::Matrix> stiffMat, fei::SharedPtr <fei::Matrix> geomStiffMat, int ndof, bool verbose)

Constructor. Some variables are initialized based on the input arguments. x_ and y_ are created based on the matrix graph of StiffnessMatrix_ and CHOLMOD is started.
The factorization of the stiffness matrix is performed on only one processor. First EpetraStiffnessMatrix_, EpetraGeometricStiffnessMatrix_, EpetraSolution_, and EpetraForce_ are constructed based on StiffnessMatrix_, GeometricStiffnessMatrix_, x_, and y_ respectively. The size of the Epetra matrices and vectors is equal to the reduced size (i.e. with slave DOF removed) of the corresponding FEI matrices and vectors. Consequently, if the size of the Epetra matrices equals ndof_, then no **MPC** are present and mpcPresent_ is set to false.
Next, the EpetraStiffnessMatrix_ is converted into CCS format, after which it is factorized in CholFactorization_. Finally, storage space is allocated for CholSolution_ and CholForce_.

fact	*Input*	Initialization of `Factory_`.
stiffMat	*Input*	Initialization of `StiffnessMatrix_`.
geomStiffMat	*Input*	Initialization of `GeometricStiffnessMatrix_`.
ndof	*Input*	Initialization of `ndof_`.
verbose	*Input*	`True` for only one of the processors. Used to perform Cholesky decomposition on one processor only.

~ArpackOperations()

Destructor. CHOLMOD is finished.

---

void OpB(double* v, double* w)

Multiplication by `StiffnessMatrix_`. The first step is to copy `v` into `x_`, after which `x_` is converted into `EpetraSolution_`. Next, `EpetraForce_` is computed as the matrix product of `EpetraStiffnessMatrix_` and `EpetraSolution_`, and it is then converted into `y_`. Finally, `y_` is copied into `w`.
This procedure is relatively cumbersome, however, necessary as the FEI package does not provide for matrix-vector multiplication. Note that `w` always satisfies the prescribed **MPC**.

v	*Input*	Input vector $\boldsymbol{v}$.
w	*Output*	Output vector $\boldsymbol{w} = \boldsymbol{B}\boldsymbol{v}$.

---

void OpA(double* v)

Multiplication by `GeometricStiffnessMatrix_`. Procedure is the similar to **OpB(...)**, but with `EpetraGeometricStiffnessMatrix_` instead of `EpetraStiffnessMatrix_`, and conversion into `y_` is final task, i.e. `y_` is not converted int a double array.

v	*Input*	Input vector $\boldsymbol{v}$.

---

void OpBiA(double* v, double* w)

Multiplication by $\boldsymbol{B}^{-1}\boldsymbol{A}$. First **OpA(...)** is called which results in `EpetraForce_` to be equal to $\boldsymbol{A}\boldsymbol{v}$. Next `EpetraForce_` is converted into `CholForce_`, after which the $\boldsymbol{B}\boldsymbol{w} = \boldsymbol{A}\boldsymbol{v}$ is solved for $\boldsymbol{w}$ (i.e. `CholSolution_`) using `CholFactorization_`.
If no **MPC** are present (i.e. `mpcPresent_=false`), `CholSolution_` contains all DOF, and so it is immediately copied into `w`. However, if **MPC** are present, `CholSolution_` does not contain the slave DOF. In this case `CholSolution_` is converted into `EpetraForce_`, which is converted into `y_`. Finally, `y_` is then copied into `w`. This procedure is relatively cumbersome, however, necessary as the FEI package does not provide for matrix-vector multiplication and for easy conversion to CCS format. Note that `w` always satisfies the prescribed **MPC**.

v	*Input*	Input vector $\boldsymbol{v}$.
w	*Output*	Output vector $\boldsymbol{w} = \boldsymbol{B}^{-1}\boldsymbol{A}\boldsymbol{v}$.

---

void set_Epetra_MultiVector(fei::Vector* feivec, Epetra_MultiVector* epvec)

This function converts an `Epetra_MultiVector*` into an `fei::Vector*`. The procedure

utilizes some specific tools from the FEI package. The function is largely based on the `get_Epetra_MultiVector` function from the FEI package.

feivec	*Input*	Pointer to `fei::Vector` to be converted.
epvec	*Output*	Converted vector as `Epetra_MultiVector*`.

OutputRequest.h

Description

The OutputRequest.h class contains the requested output by the user. These requests are specified by the user in the file OutputRequests.dat. Only one OutputRequest object is created, which is contained in the **Domain.h** class. An OutputRequest is mainly characterized by severable boolean variables indicating whether a specific output is requested or not.

Variables

bool	**OutputFileAlreadyOpened_**	If true, the output file has already been opened. In this case new output is appended to this file.
bool	**PrintModelInfo_**	If true, general information about the model is printed in the output file.
bool	**PrintNodalOutput_**	If true, nodal output (i.e. displacements) is printed in the output file.
bool	**PrintElementOutput_**	If true, element outputs (i.e. strains, forces and strain energy) are printed in the output file if the are also requested individually.
bool	**PrintElementStrains_**	If true and PrintElementOutput_=true, element strains are printed in the output file.
bool	**PrintElementForces_**	If true and PrintElementOutput_=true, element forces are printed in the output file.
bool	**PrintElementStrainEnergy_**	If true and PrintElementOutput_=true, element strain energies are printed in the output file.
bool	**PrintNodalBucklingOutput_**	If true, nodal buckling outputs (i.e. buckling modes and buckling loads) are printed in the output file.
int	**numNodalOutputs_**	Total number of different nodal outputs. Used to determine the number of required temporary files for writing the output.
int	**numElementOutputs_**	Total number of different element outputs. Used to determine the number of required temporary files for writing the output.

Public Functions

OutputRequest()

Constructor. All boolean variables are initialized to be `false`.

`void SetParameters( fei::ParameterSet& params )`

The variables of this class are set based on the `fei::Parameterset` which is obtained from the user file `OutputRequests.dat`.

`params`	*Input*	Parameter set containing output requests based on `OutputRequests.dat`.

`void PrintModelInfo( Domain* domain, ofstream& fout )`

General information about the model is printed to the output file. This information is not intended for the users of OPTANT, but instead, it can be used by the developer e.g. to check whether the input file has been correctly imported into OPTANT or to check if the all **`Elements`** are aligned in the heap memory.

`domain`	*Input*	Reference to the **`Domain.h`** object for the FEM model.
`fout`	*Input*	Output stream for output file.

UtilityFunctions.h

Description

The main purpose of the `UtilityFunctions` namespace is to read the OPTANT input model [2] and other user specified parameters. In addition, it also contains a few functions to print arrays or matrices of numbers to the output file or to the console.
In order to better understand the functions in `UtilityFunctions`, it is convenient to consider the execute command line for OPTANT in the `Makefile`:

```
mpirun -np $(NPROC) ./$(MAINFILE) -inp $(INPUTFILE) -sol $(SOLVERFILE)
-req $(REQUESTFILE)
```

The parameters in this command line are described below:

`mpirun`	Indicates that multiple processes are used.
`np`	Number of processes.
`./$(MAINFILE)`	Execute command of compiled OPTANT file.
`-inp`	Key which is followed by name of input file.
`$(INPUTFILE)`	Name of input file.
`-sol`	Key which is followed by name of solver parameters file.
`$(SOLVERFILE)`	Name of solver parameters file.
`-req`	Key which is followed by name of output requests file.
`$(REQUESTFILE)`	Name of output requests file.

Namespace Functions

`int getFileName( int argc, char** argv, const char* argTag, char* destination )`

This function finds the file name, which is followed by the key `argTag` in the execute command line. First the location of the key `argTag` (`-inp`, `-sol` or `-req`) in the execute command line is obtained using **`whichArg(...)`**. The file name is then obtained from the next position in `argv` and it is stored in `destination`.

`argc`	*Input*	Number of arguments in the execute command line.
`argv`	*Input*	Array of character pointers containing all arguments of the command line.
`argTag`	*Input*	Key that precedes the wanted file name in the execute command line.
`destination`	*Output*	Output containing the file name.

int read_input_file(const char* filename, Domain& domain)

The input file is read. This function is divided into several other functions, each of which reads one entity in the input file (i.e. **Materials**, **Properies**, **Nodes**, etc.). The implementation of these functions contains several checks whether the format specified in [2] is used in the input file. Warnings or errors are provided to the user if this is not the case.
Many variables of the **Domain.h** object are specified in this read function.

filename	*Input*	Name of input file.
domain	*Input*	Reference to the **Domain.h** object for the FEM model.

int read_param_file(const char* filename, MPI_COMM comm, vector<string>& file_contents)

This function reads a parameter file (e.g. `SolverParams.dat` or `OutputRequests.dat`) into a vector of string. The implementation of this function has been obtained from the FEI package documents.

filename	*Input*	Name of the file.
comm	*Input*	Basic object to handle communication among processes.
file_contents	*Output*	`String` vector containing the file parameters.

int whichArg(int argc, const char*const* argv, const char* findarg)

This function returns the index of `findarg` in `argv`. The index is found by looping through `argv` and comparing each entry with `findarg`.

argc	*Input*	Number of arguments in the execute command line.
argv	*Input*	Array of character pointers containing all arguments of the command line.
findarg	*Input*	Argument to be found.

void PrintArray(double* array, int size, ofstream& fout)

Print array as row vector to output file.

array	*Input*	Array to be printed.
size	*Input*	Size of the array to be printed.
fout	*Input*	Output stream for output file.

void PrintMatrix(double* array, int nrow, int ncol, ofstream& fout)

Print matrix to output file. Note that C++ is column based, whereas printing the matrix is 'row based'. This inconsistency is handled by using two loops, one loop over the rows and one loop over the columns.

array	*Input*	Matrix to be printed.
nrow	*Input*	Number of matrix rows.
ncol	*Input*	Number of matrix columns

fout	*Input*	Output stream for output file.

MatrixOperations.h

Description

The MatrixOperations namespace contains many functions performing hard-coded operations, such as Cholesky decomposition, matrix inversion, matrix-matrix multiplication, matrix-vector multiplication, etc. These operations are mostly applied to small matrices and to matrices with specific properties such as symmetric matrices or triangular matrices.
In all MatrixOperations functions it is attempted to minimize required storage by allowing to overwrite the input by the output. A few of these functions are described below.

Namespace Functions

---

void dpotrf4(const double* A, double* L)

Cholesky decomposition of a 4×4 symmetric positive definite matrix A is computed and stored in L. The output L may be stored on the location of A.

A	*Input*	Input matrix $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$.
L	*Output*	Cholesky decomposition $\boldsymbol{L}$.

---

void Add(const double alpha, const double beta, const double* A, const double* B, double* C, const int M)

Add two matrices as $\boldsymbol{C} = \alpha\boldsymbol{A} + \beta\boldsymbol{B}$. C may be stored on the position of both A and B.

alpha	*Input*	α.
beta	*Input*	β.
A	*Input*	$\boldsymbol{A}$.
B	*Input*	$\boldsymbol{B}$.
C	*Output*	$\boldsymbol{C}$.
M	*Input*	Total size of the matrices, i.e. $M = n_{row}n_{col}$.

---

void Scale(const double alpha, const double beta, const double* A, double* C, const int M)

Scale a matrix as $\boldsymbol{C} = \alpha\boldsymbol{A} + \beta$. C may be stored on the position of A.

alpha	*Input*	α.
beta	*Input*	β.

A	*Input*	$\boldsymbol{A}$.
C	*Output*	$\boldsymbol{C}$.
M	*Input*	Total size of the matrices, i.e. $M = n_{row} n_{col}$.

```
double Dot( const double* A, const double* B, const int M )
```

Return dot product of vectors `A` and `B` of length `M`.

A	*Input*	Vector to apply in dot product.
B	*Input*	Vector to apply in dot product.
M	*Input*	Length of `A` and `B`.

```
void Cross( const double* A, const double* B, double* C )
```

Store cross product of two 3×1 vectors: $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$.

A	*Input*	3×1 vector to apply in cross product.
B	*Input*	3×1 vector to apply in cross product.
C	*Output*	3×1 cross product of `A` and `B`.

```
void ConventionalToPacked( const double* C, double* P, const int n )
```

Converts $n \times n$ matrix from conventional storage format $\boldsymbol{C}$ to packed storage format $\boldsymbol{P}$.

C	*Input*	Matrix in conventional storage format.
P	*Output*	Matrix in packed storage format.
n	*Input*	Size of square matrix.

```
void PackedToConventional( const double* P, double* C, const int n )
```

Converts $n \times n$ matrix from packed storage format $\boldsymbol{P}$ to conventional storage format $\boldsymbol{C}$.

P	*Input*	Matrix in packed storage format.
C	*Output*	Matrix in conventional storage format.
n	*Input*	Size of square matrix.

PCH_OPTANT.h

Description

The goal `PCH_OPTANT.h` is to increase compile speed of OPTANT using precompiled headers. All OPTANT classes are included in `PCH_OPTANT.h` and several type definitions are specified. When compiling OPTANT, this `PCH_OPTANT.h` is compiled first resulting in a precompiled headers file. After that, OPTANT is compiled using this precompiled headers file, which reduces compile time by approximately a factor of two. Note that this compile sequence is defined in `Makefile`.

Bibliography

[1] Multifrontal massively parallel solver (mumps 4.10.0) users guide.

[2] Optant: User manual.

[3] Devis, T. A. User guide for cholmod: a sparse cholesky factorization and modification package.

[4] Felippa, C. A. Computational mechanics for the twenty-first century. Civil-Comp press, Edinburgh, UK, UK, 2000, ch. Recent Advances in Finite Element Templates, pp. 71–98.

[5] Felippa, C. A. A study of optimal membrane triangles with drilling freedoms. *Computer Methods in Applied Mechanics and Engineering 192*, 1618 (2003), 2125 – 2168.

[6] Gomes, F. M., and Sorensen, D. C. Arpack++, an object-oriented version of arpack eigenvalue package.

[7] Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A. An overview of trilinos.

[8] Kassapoglou, C. *Design and Analysis of Composite Structures*. 2010.

[9] Sala, M. Amesos 2.0 reference guide.

[10] Williams, A. B. Finite element interface to linear solvers (fei) version 2.9: Guide and reference manual.