OOFEM Developer guide

Bořek Patzák, Martin Horák

May 21, 2026

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The aim of OOFEM project is to develop efficient and robust tool for FEM computations as well as to provide modular and extensible environment for future development.

The aim of this document is to provide the introduction to understand the OOFEM principles and internal structure, and provides commented examples how to implement new components, such as custom elements, material models, or even new problem formulations.

The OOFEM is divided into several modules. The core module, called OOFEMlib, introduces the fundamental, core classes. These classes are problem independent and they provide the common definitions and support to remaining modules. The other modules typically implement problem specific parts, such as elements, materials, solvers, etc. The OOFEMlib module is the compulsory module, that is always included in all builds. The other modules are optional and can be included or excluded from the build, depending on the user configuration.

The OOFEM package is written in C++. This document contains many examples and listings of a parts of the source code. Therefore, the ideal reader should be familiar with C++ programming language. However, any reader with an object-oriented background should follow this document, since the examples are written in a form, which is hopefully easy to read and understand.

CHAPTER

TWO

CODING STANDARDS

2.1 Naming Conventions

The names of classes, attributes, services, variables, and functions in a program serve as comments of a sort. So don't choose terse names–instead, look for names that give useful information about the meaning of the variable or function. In a OOFEM program, names should be English, like other comments.

Local variable names can be shorter, because they are used only within one context, where (presumably) comments explain their purpose.

Try to limit your use of abbreviations in symbol names. It is ok to make a few abbreviations, explain what they mean, and then use them frequently, but don't use lots of obscure abbreviations.

Please use capital letters to separate words in a name. Stick to lower case; reserve upper case for macros, and for name-prefixes that follow a uniform convention. The function or service name should always begin with lovercase letter, the first uppercase letter in function name indicates, that function is returning newly allocated pointer, which has to be dealocated. For example, you should use names like ignoreSpaceChangeFlag;

When you want to define names with constant integer values, use enum rather than *#define*. GDB knows about enumeration constants.

Use descriptive file names. The class declarations should be placed in *.h files and class implementation in corresponding *.C files. For each class, create a separate files.

2.2 Parameters and Return Values

The prefered argunent passing method for objects is by reference. Try to avoid rerturning pointers to arrays or matrices (or generally to any component), since it is not clear, whether to delocate the returned pointer or not. The most prefered way is to create local variable of vector or matrix type, pass it (using reference) to called function. The calling function is responsible to properly resize the (output) parameter and set values accordingly. The point is, that destructors are called for local variables automatically by compiler, so there is no possibility for memory leaks and the local variable can be reused for multiple calls to target function (inside loop) and therefore there is no need for repeating memory allocation and dealocation. Sometimes it may be reasonable to return pointer to constant float array or matrix (representing for example nodal coordinates), since passing output array as parametr will require array copying.

CHAPTER

THREE

GENERAL STRUCTURE

General structure is shown in Fig (*fig-generalstructure*). The key class is *EngineeringModel* representing the problem under consideration. It can contain one or more domains (abstractions for computational mesh) represented by the *Domain* class. Generally speaking, it contains the domain description, or if the program runs in parallel, then it contains the description of the domain associated with a particular processor or thread of execution. *Domain* class contains and manages lists of degree of freedom managers, elements, boundary conditions, cross sections, and material models - these describe the geometry of the problem, its constitutive properties, and applied boundary conditions. Services for accessing each of these objects are provided. *Domain* class & object provides services for reading input files and instantiating corresponding components accordingly.

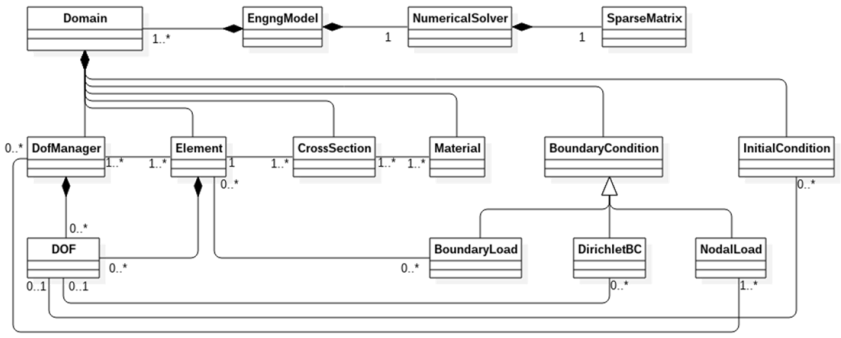


Fig. 1: OOFEM Top level structure

The *EngngModel* class is an abstraction for the problem under consideration. It represents the type of analysis to be performed. Base class declares and implements the basic general services for assembling characteristic components and services for starting the solution step and its termination. Derived classes know the form of governing equation and the physical meaning of particular components. They are responsible for forming the governing equation for each solution step, usually by summing contributions from particular elements and nodes. The underlying discrete problem is solved by suitable instance of *NumericalMethod* class, that can take advantage of efficient sparse store for characteristic matrices, represented here by *SparseMtrx* class. This part is described in detail in **`Problem representation - Engineering model`_**.

Class *Element*, is an abstraction of a finite element. Element manages its DofManagers (nodes), and has its cros section and material models defined by *CrossSection* and *Material* class instances. Element manages its Integration rules composed of individual integration points and can also have its internal DOFs.

DOF class is an abstraction for a single degree of freedom (DOF). It maintains its physical meaning and associated equation number. *DOF* is the attribute of *DofManager* class, which manages the collection of DOFs. A typical derived class is class *Node*, representing a node in a finite element mesh.

BoundaryCondition and *InitialCondition* classes are abstractions of boundary and initial conditions. They are attributes of *Domain* class and are associated with one or more DOFs. The abstract class *Load*, derived from parent *Bound-*

aryCondition class, is an abstraction for load. It is an attribute of Domain and can be associated with several dof managers or elements, according to the type of loading it represents. The class declares the basic services provided by all derived classes. Derived classes declare specific load type dependent services and implement all necessary services.

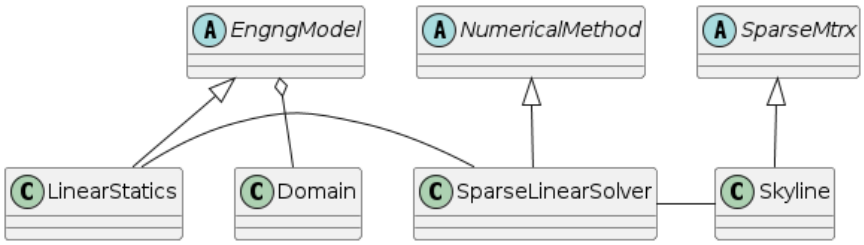
CHAPTER

FOUR

PROBLEM REPRESENTATION - ENGINEERING MODEL

In this section, we introduce in detail how the problems are represented, their discrete form assembled and finally solved.

We start with *EngngModel* class, which is an abstraction for the problem under consideration. It represents the analysis to be performed. Base class declares and implements the basic general services for assembling characteristic components and services for starting the solution step and its termination. Derived classes know the form of governing equation and the physical meaning of particular components. They are responsible for forming the governing equation for each solution step, usually by summing contributions from particular elements and nodes, and their solution using the suitable numerical method.



The solution step may represent either a time step, a load increment, or a load case. The solution steps are grouped together into so called meta steps. The meta step can be thought as a sequence of solution steps, with the same set of attributes used to drive the behavior of EngngModel. For each meta step, the *EngngModel* typically updates its control parameters according to ones defined by meta step (see updateAttributes service) and generates the sequence of the solution steps. This allows to switch to a different time increment, a different solution control, etc. If no meta step is specified, the EngngModel creates a default one for all solution steps. There are two services, where EngngModel attributes are set or updated. The first one, used for those attributes which do not vary during the solution of the problem, are initialized in *instanciateYourself* service. The *updateAttributes* method is called when specific meta step is activated, and selected attributes are updated from corresponding meta step attributes. If no meta step is introduced, a default one is created (with the attributes set to the EngngModel attributes). Then there is no difference, whether the attributes are initialized in instanciateYourself or in updateAttributes, but the preferred scheme is to read all attributes in instanciateYourself and to left updateAttributes service empty.

The basic *EngngModel* tasks are following:

- Assembling governing equations by summing contributions (typically from nodes and elements).
- Solving the problem described by governing equation(s) using the instance of a suitable numerical method. This requires to establish a mapping between numerical method-parameters and EngngModel components of governing equation. EngngModel must map each component of governing equation(s) (which has physical meaning) to the corresponding numerical component. This mapping between physical components to independent numerical components (understood by the numerical method) is important, because it allows the numerical method to be used by many EngngModels with different component meaning and allows to use different numerical methods by EngngModel. This is achieved by using the compulsory numerical component names (see further).

- Providing access to the problem solution. Services for returning unknown values according to their type and mode are provided. These services are used by DOFs to access their corresponding unknowns.
- Terminating the time step by updating the state of the problem domains (nodes and elements, including integration points).
- Managing the problem domains, meta steps, and problem input/output streams.
- Equation numbering.
- Storing and restoring problem state to/from context file.
- Managing and updating unknowns

The complete listing of *EngngModel* class declaration can be found here: https://github.com/oofem/oofem/blob/master/src/oofemlib/engngm.h. It is well commented and should be self-explanatory.

One of the key methods of *EngngModel* class is *solveYourself*, which is invoked to starts the solution of the problem. The default implementation loops over individual metasteps. For each metastep, the loop over nested solution steps is performed. The representation of solution step (*TimeStep* class instance) is created inside *giveNextStep* service and stored in EngngModel attribute *currentStep*. After some initializations, the solution step is solved by calling *solveYourselfAt* service, which performs solution for specific solution step, passed as its parameter.

The simplified implementation of *solveYourself* service is shown below:

```
void
EngngModel :: solveYourself ()
{
     int imstep, jstep;
     int smstep=1, sjstep=1;
     MetaStep* activeMStep;

     for (imstep = smstep; imstep<= nMetaSteps; imstep++) {
         activeMStep = this->giveMetaStep(imstep);
         for (jstep = sjstep; jstep <= activeMStep->giveNumberOfSteps(); jstep++) {
         this->giveNextStep();
         // update attributes according to new meta step attributes
         if (jstep == sjstep) this->updateAttributes (this->giveCurrentStep());
         this->solveYourselfAt(this->giveCurrentStep());
         this->updateYourself( this->giveCurrentStep() );
         this->terminate( this->giveCurrentStep() );

     }
}
```

The *solveYourselfAt* method typically assembles characteristic matrices and vectors and solves the problem using the suitable numerical method. The implementation should be provided by derived classes implementing specific problem. (see section ref{Engngmodelexample} for an example). After finishing the solution for the given solution step, *updateYourself* service is called to update the state of all components. Finally *terminate* service is called.

The default implementation of *updateYourself* service loops over all problem domains and calls corresponding update service for all DOF managers and elements:

```
 void
 EngngModel :: updateYourself(TimeStep *tStep)
 {
     for ( auto &domain: domainList ) {
         for ( auto &dman : domain->giveDofManagers() ) {
```

(continues on next page)

(continued from previous page)

```
            dman->updateYourself(tStep);
        }
        for ( auto &elem : domain->giveElements() ) {
            elem->updateYourself(tStep);
        }
}
```

The *terminate* ` service essentially prints the required outputs and optionally saves the context file (if required), so the solution can be restarted from this saved state later:

```
void
EngngModel :: terminate(TimeStep *tStep)
{
    exportModuleManager.doOutput(tStep);
    this->saveStepContext(tStep, CM_State | CM_Definition);
}
```

Both services are virtual, so they can be easily tailored to specific needs.

The *EngngModel* class comes with handy generic services for characteristic components assembly, that are used by derived classes to assemble the characteristic components of the problem. They essentially loop over nodes or elements (depending on the character of the requested component) of the given domain, requesting the corresponding component (determined or even evaluated by assembler class instance) and corresponding code numbers. The component contributions are assembled (using code numbers) into a target array or matrix. Here we show the simplified implementation of one of these services to assemble the characteristic matrix:

```
void EngngModel :: assemble(SparseMtrx &answer, TimeStep *tStep, const MatrixAssembler &
↪ma,
                            const UnknownNumberingScheme &s, Domain *domain)
{
    IntArray loc;
    FloatMatrix mat, R;
    int nelem = domain->giveNumberOfElements();

    for ( int ielem = 1; ielem <= nelem; ielem++ ) {
        auto element = domain->giveElement(ielem);
        ma.matrixFromElement(mat, *element, tStep);

        if ( mat.isNotEmpty() ) {
            ma.locationFromElement(loc, *element, s);
            if ( element->giveRotationMatrix(R) ) {
                mat.rotatedWith(R);
            }
            if ( answer.assemble(loc, mat) == 0 ) {
                OOFEM_ERROR("sparse matrix assemble error");
            }
        }
    }
}
```

The *assemble* service is used to assemble the characteristic matrix from elements contributions. The *MatrixAssembler* class instance is used to parametrize the *aasemble* method and it determines the element contributions. In the simple form, it can request characteristic matrix directly from element, but it can also evaluate the element contribution. The *UnknownNumberingScheme* class instance determines the unknown numbering and thus determines the code numbers

of the unknowns. The *SparseMtrx* class instance is used to store the assembled matrix.

4.1 Numerial Method Interface

The *EngngModel* needs to solve the underlying discrete problem. This is done by the suitable instance of *NumericalMethod* class. The design attempts to separate the problem formulation from the numerical solution of the problem, and the data storage format.

Derived classes from *NumericalMethod* class are supposed to declare the interface for specific problem type (like solution of linear system). The interface usually consist in declaring virtual abstract function *solve*, with parameters corresponding to problem under consideration. The data are specified using parameters passed to *solve* method (so called mapping of physical components to their numerical counterpart). The parameters of numerical method either are passed to *solve* method or are set by *instanciateYourself* service from input file. The *solve* method shoud return value of *NM_Status* type.

Many problems require updating components during the solution. To keep definition and implementation of numerical method independent on particular problem, the *EngngModel* must also provide service for updating mapped components components, if this is necessary. This is provided by *EngngModel::updateComponent* method. This method is invoked by numerical method, when the update of some components during solution is needed (for example in the Newton Raphson algorithm for the solution of non-linear equations, stiffness or internal force vector need to be updated during the solution process).

The derived classes from class{Numerical method} are supposed to declare the interface for specific problem type (like solution of linear system). It should be pointed out, that all numerical methods solving the same numerical problem have to use the same genaral interface - this is enforced by introducing the abstract class representing family of numerical method for solving specific problem and declaring the common interface.

There are typically multiple numerical methods for solving the same problem. The *NumericalMethod* can implement the solution itself, but it can also implement an interface to external numerical libraries (like PETSc, Trilinos, etc.).

This concept is further enhanced by the introduction of a base abstract class *SparseMatrix* representing sparse matrix storage. This class only declares the basic required services provided by all sparse matrix implementations (like assembly of contribution, multiplication by a vector, possible factorization, etc). The implementation is left on derived classes. Numerical methods are then implemented only using basic services declared by the *Sparse Matrix* class. Thus, numerical method class instances will work with any sparse matrix representation, even added in the future, without changing any code.

As an example, the declaration of the *SparseLinearSystem* class is shown below. This class is an abstraction for all numerical methods solving sparse linear system of equations.

```
1  class SparseLinearSystemNM : public NumericalMethod
2  {
3  public:
4      /// Constructor.
5      SparseLinearSystemNM(Domain * d, EngngModel * m);
6      /// Destructor.
7      virtual ~SparseLinearSystemNM();
8
9      /**
10     * Solves the given sparse linear system of equations @f$ A\cdot x=b @f$.
11     * @param A Coefficient matrix.
12     * @param b Right hand side.
13     * @param x Solution array.
14     * @return Status of the solver.
15     */
16     virtual ConvergedReason solve(SparseMtrx &A, FloatArray &b, FloatArray &x) = 0;
```

(continues on next page)

(continued from previous page)

```
    /**
    * Solves the given sparse linear system of equations @f$ A\cdot X=B @f$.
    * Default implementation calls solve multiple times.
    * @param A Coefficient matrix.
    * @param B Right hand side.
    * @param X Solution matrix.
    * @return Status of the solver.
    */
    virtual ConvergedReason solve(SparseMtrx &A, FloatMatrix &B, FloatMatrix &X);
};
```

To summarize, the natural independence of the problem formulation, numerical solution of the problem, and data storage format have been obtained, which leads to a modular and extensible structure.

4.2 Domain class

The computational grid is represented by *Domain* class. It manages all components of the FEM discrete model. These include dof managers (nodes, element sides possessing DOFs), elements, material and cross section models, boundary and initial conditions, time functions, and so on. For every component type *Domain* maintains the component list and provides the corresponding access services.

The basic services provided by Domain are the following:

- Reading its description from input and creating corresponding objects. This task includes the reading and parsing the particular mesh input records, creating the corresponding components representations (objects) of appropriate type, initializing these components using their *instanciteFromString* methods and storing them into corresponding list.
- Provides services for accessing its particular components. The services returning the total number of particular domain components and particular component access methods based on component number are provided.

The domain also manages instances of *SpatialLocalizer* and *connectivityTable* classes to serve the connectivity and spatial localization related services (finding elements shared by the node, finding the closest node search, finding the element containing given point, etc.).

For complete definition of *Domain* class interface, please go to https://github.com/oofem/oofem/blob/master/src/oofemlib/domain.h

4.3 Example - Linear Static Analysis implementation

In this section, the example of the implementation of linear static analysis will be given. The linear static analysis is a typical example of a structural engineering problem. In this particular case, the analysis is time independent, meta steps are not used (default one is created) and time steps are used to distinguish load cases (different load vectors).

The class definition includes the declaration of characteristic components of the problem - the stiffness matrix and load and displacement vectors. Two additional variables are used to store the linear solver type and the sparse matrix type, which can be selected by the user. Finally, the reference to suitable instance of *SparseLinearSystemNM* class is stored in the *nMethod* attribute.

The following services are declared/implemented: - *solveYourselfAt* for solving the solution step, responsible for forming the stiffness matrix and load vector, and calling the numerical method to solve the problem, - *giveUnknownComponent* providing access to problem unknowns (displacements), - context i/o services for serializing and deserializing the state of the problem (*saveContext* and *restoreContext* services), - solver parameter initialization (*initializeFrom*) and consistency checking (*checkConsistency*).

```
class LinearStatic : public StructuralEngngModel
{
protected:
    std :: unique_ptr< SparseMtrx > stiffnessMatrix;
    FloatArray loadVector;
    FloatArray displacementVector;

    LinSystSolverType solverType;
    SparseMtrxType sparseMtrxType;
    /// Numerical method used to solve the problem.
    std :: unique_ptr< SparseLinearSystemNM > nMethod;

    int initFlag;
    EModelDefaultEquationNumbering equationNumbering;

public:
    LinearStatic(int i, EngngModel *master = nullptr);
    virtual ~LinearStatic();

    void solveYourself() override;
    void solveYourselfAt(TimeStep *tStep) override;

    double giveUnknownComponent(ValueModeType type, TimeStep *tStep, Domain *d, Dof␣
↪*dof) override;
    void saveContext(DataStream &stream, ContextMode mode) override;
    void restoreContext(DataStream &stream, ContextMode mode) override;

    void updateDomainLinks() override;

    TimeStep *giveNextStep() override;
    NumericalMethod *giveNumericalMethod(MetaStep *mStep) override;

    void initializeFrom(InputRecord &ir) override;

    virtual UnknownNumberingScheme &giveEquationNumbering() { return equationNumbering;␣
↪}

    // identification
    virtual const char *giveInputRecordName() const { return _IFT_LinearStatic_Name; }
    const char *giveClassName() const override { return "LinearStatic"; }
};
```

The implementation of the class{LinearStatic} class and its methods follows. First, we start with *initializeFrom* method, responsible for reading user input parameters. The input record is represented by instance of *InputRecord* class, which allows for key-value lookup. The *LinearStatic* reads the solver type and sparse matrix type from the input record and stores them in the corresponding attributes. The error handling is achieved using exceptions thrown by *IR_GIVE_OPTIONAL_FIELD* macros, which is defined in the *InputRecord* class.

```
void
LinearStatic :: initializeFrom(InputRecord &ir)
{
    // call parent class
    StructuralEngngModel :: initializeFrom(ir);
```

(continues on next page)

(continued from previous page)

```

    int val = 0;
    IR_GIVE_OPTIONAL_FIELD(ir, val, _IFT_EngngModel_lstype);
    solverType = ( LinSystSolverType ) val;

    val = 0;
    IR_GIVE_OPTIONAL_FIELD(ir, val, _IFT_EngngModel_smtype);
    sparseMtrxType = ( SparseMtrxType ) val;

}
```

The *giveNumericalMethod* method is responsible for allocating and returning the suitable instance numerical method. The method uses class factory to create the instance of the sparse linear solver.

```
NumericalMethod *LinearStatic :: giveNumericalMethod(MetaStep *mStep)
{
    if ( !nMethod ) {
        nMethod = classFactory.createSparseLinSolver(solverType, this->giveDomain(1),␣
↪this);
    }
    if ( !nMethod ) {
        OOFEM_ERROR("linear solver creation failed for lstype %d", solverType);
    }
    return nMethod.get();
}
```

The *giveUnknownComponent* method provides access to problem unknowns, in our case to displacement vector. the unknown component of the problem. The method is called by the numerical method to access the unknowns (displacements) of the problem.

```
double LinearStatic :: giveUnknownComponent(ValueModeType mode, TimeStep *tStep, Domain␣
↪*d, Dof *dof)
{
    // get DOF equation number
    int eq = dof->__giveEquationNumber();

    if ( tStep != this->giveCurrentStep() ) {
        OOFEM_ERROR("unknown time step encountered");
        return 0.;
    }

    switch ( mode ) {
    case VM_Total:
    case VM_Incremental:
        if ( displacementVector.isNotEmpty() ) {
            return displacementVector.at(eq);
        } else {
            return 0.;
        }

    default:
        OOFEM_ERROR("Unknown is of undefined type for this problem");
```

(continues on next page)

(continued from previous page)

```
    }

    return 0.;
}
```

Finally, we provide the implementation of the *solveYourselfAt* method, which is responsible for solving the problem at the given time step representing load-case. The method first assembles the stiffness matrix and load vector, and then calls the numerical method to solve the problem.

```
void LinearStatic :: solveYourselfAt(TimeStep *tStep)
{
    // initFlag is used to avoid assembling stiffness matrix for each load-case
    if ( initFlag ) {
        OOFEM_LOG_DEBUG("Assembling stiffness matrix\n");
        stiffnessMatrix = classFactory.createSparseMtrx(sparseMtrxType);
        if ( !stiffnessMatrix ) {
            OOFEM_ERROR("sparse matrix creation failed");
        }

        stiffnessMatrix->buildInternalStructure( this, 1, this->giveEquationNumbering()␣
↪);
        // use TangentAssembler to assemble the stiffness matrix
        this->assemble( *stiffnessMatrix, tStep, TangentAssembler(TangentStiffness),
                        this->giveEquationNumbering(), this->giveDomain(1) );
        initFlag = 0;
    }
    // allocate space for displacementVector
    displacementVector.resize( this->giveNumberOfDomainEquations( 1, this->
↪giveEquationNumbering() ) );
    displacementVector.zero();
    loadVector.resize( this->giveNumberOfDomainEquations( 1, this->
↪giveEquationNumbering() ) );
    loadVector.zero();

    OOFEM_LOG_DEBUG("Assembling load\n");
    this->assembleVector( loadVector, tStep, ExternalForceAssembler(), VM_Total,
                          this->giveEquationNumbering(), this->giveDomain(1) );

    // assemble internal part of load vector (forces induced by prescribed␣
↪displacements, etc.)
    FloatArray internalForces( this->giveNumberOfDomainEquations( 1, this->
↪giveEquationNumbering() ) );
    internalForces.zero();
    this->assembleVector( internalForces, tStep, InternalForceAssembler(), VM_Total,
                          this->giveEquationNumbering(), this->giveDomain(1) );

    loadVector.subtract(internalForces);

    OOFEM_LOG_INFO("\n\nSolving ...\n\n");

    this->giveNumericalMethod( this->giveMetaStep( tStep->giveMetaStepNumber() ) );
    ConvergedReason s = nMethod->solve(*stiffnessMatrix, loadVector,␣
```

(continues on next page)

(continued from previous page)

```
↪displacementVector);
    if ( s != CR_CONVERGED ) {
        OOFEM_ERROR("No success in solving system.");
    }
}
```

Please refer to full source code of the *LinearStatic* class in the OOFEM source code repository for more details: https://github.com/oofem/oofem/blob/master/src/sm/EngineeringModels/linearstatic.C

CHAPTER

FIVE

DOFMANAGERS, DOFS, BOUNDARY AND INITIAL CONDITIONS

5.1 Degrees of Freedom

Abstract class *Dof* is provided and it represents abstraction for Degree Of Freedom (DOF) in finite element mesh. DOFs are possessed by DofManagers (i.e, nodes, element sides or whatever) and one *Dof* instance can belong to only one *DofManager* instance. *Dof* maintain its physical meaning and reference to related DofManager (reference to DofManager which possess particular DOF). To describe physical meaning of particular Dof, special enum type *DofId* has been introduced (see src/oofemlib/cltypes.h). This type is more descriptive than UnknownType, which determines physical meaning for unknowns generally (displacement or temperature). *DofId* type has to distinguish between DOFs representing displacement, but in different directions, since only some of those may be required by particular elements.

DOF can be subjected to boundary (BC) or initial (IC) condition. Method for obtaining DOF unknown value is provided. If no IC condition has been given, zero value IC is assumed otherwise when needed.

Dof class generally supports changes of static system during computation. This feature generally leads to equation renumbering. Then because equation number associated to dof may change, it may become extremely complicated to ask EngngModel for unknown from previous time step (because equation number may have been changed). To overcome this problem, derived class will implement so called unknown dictionary, which is updated after finishing each time step and where unknowns for particular dof are stored. Dof then uses this dictionary for requests for unknowns instead of asking EngngModel instance for unknowns. Unknowns in dof dictionary are updated by EngngModel automatically (if EngngModel supports changes of static system) after finishing time step.

The base class *Dof* declares many services necessary for DOF handling. Some of them are abstract, they have to be implemented by derived classes representing specific DOF types. This basic DOF interface includes the following services

- Services for requesting corresponding unknowns (*giveUnknown*) and the prescribed boundary values (*giveBcValue*).
- Methods for requesting associated equation number (*giveEquationNumber*).
- Services for requesting DOF properties (*hasBc*, *hasIc*, *giveDofID*, *giveDofIDName*, *giveUnknownType*, *isPrimaryDof*).
- Services for dof output (*printOutputAt*, *printStaticOutputAt*, *printDynamicOutputAt*),
- updating the receiver after new equilibrium is reached (*updateYourself*, *updateUnknownsDictionary*),
- services for storing/restoring context (*saveContext*, *restoreContext*).

The derived classes implement specific types of DOFs. The most common types are provided with OOFEMlim. Supported are so-called MasterDofs, representing true DOFs with their own equation number, SlaveDOFs, representing single DOF linked to another single DOF, Rigid Arm DOFs allowing to model rigid arms (single dof linked to one or more other DOFs). Their brief description follows:

- *MasterDof* - class representing "master" degree of freedom. Master is degree of freedom, which has its related unknown and corresponding equation number. It maintain its equation number.

- *SlaveDof* - class representing "slave" degree of freedom. SlaveDOF is linked to some master dof (link slave-slave is not allowed). Slaves can be used to implement duplicated joints (by specifying some DOF in node as masters and some as slaves linked to DOFs in other node (with same or possibly different coordinates). SlaveDOF share the same equation number with master. Almost all operation as request for BC or IC or equation number are simply forwarded to master. Other functions (which change internal state) like updateYourself, updateUnknownsDictionary, askNewEquationNumber or context storing/restoring functions are empty functions, relying on fact that same function will be called also for master. From this point of view, one can see slave dof as link to other dof.
- *RigidArmSlaveDof* - class representing "rigid arm slave" degree of freedom. This DOF is linked to some master DOFs by linear combination (link slave-slave is not allowed) using rigid arm. Implemented using nodal transformation. The Rigid Arm Slave dof represent DOF, which is directly related to one or more master DOFs. Therefore the rigid arm slave's equation number is undefined. Similarly, rigid arm slave cannot have own boundary or initial conditions - these are entirely determined using master boundary or initial conditions. The transformation for DOFs and load is not orthogonal - the inverse transformation can not be constructed by transposition. Because of time consuming inversion, methods can generally compute both transformations for DOFs as well as load components. It is important to ensure (on input) that both slave dofManager and master dofManager are using the same local coordinate system. In future releases, this can be checked using checkConsistency function, where this check could be performed.

5.2 Dof Managers

The *DofManager* class is the base class for all DOF managers. *DofManager* is an abstraction for object possessing degrees of freedom (DOFs). Dof managers (respectively derived classes, representing nodes or sides) are usually shared by several elements and are maintained by corresponding domain. The elements keep the logical reference to corresponding dof managers (they store their numbers, not the physical links). The base class declares the variable storing total number of DOFs, the *dofArray`array representing the list of DOFs managed by dof manager, and `loadArray* list for storing the applied loadings references. The *DofManager* declares (and implements some of them) following services:

- DOF management methods. The methods for requesting the total number of DOFs (*giveNumberOfDofs*), requesting particular DOFs (*giveDof*), assembling location array of code numbers (*giveLocationArray* and *giveCompleteLocationArray*),
- methods for DOF selection based on their physical meaning (*giveDofArray* and *findDofWithDofId*), and services for requesting unknowns related to dof manager DOFs (*giveUnknownVector* and *givePrescribedUnknownVector*).
- Transformation functions. The governing equations can be assembled not only in global coordinate system, but also in user-defined local coordinate system of each dof manager. Following methods introduce necessary transformation methods, allowing DOFs to be expressed in their own local c.s. or to be dependent on other DOFs on other dofManager (to implement slave or rigid arm nodes etc.). The methods for computing transformation matrices from global c.s to receiver's defined coordinate system are declared (*computeDofTransformation* and *computeLoadTransformation*). The *requiresTransformation* indicates whether dofManager requires the transformation from global c.s. to dof manager specific coordinate system.
- Load/boundary condition management functions. The two methods are provided for handling applied loading - the service for computing the load vector of receiver in given time (*computeLoadVectorAt*) and service providing the list of applied loads (*giveLoadArray*).
- Context related services for storing/restoring the receiver state to context (*saveContext* and *restoreContext* services) are implemented.
- Instantiating service *initializeFrom*.
- Miscellaneous services for receiver printing, identification, etc.

In the OOFEMlib the common specialized dof managers are defined and implemented. Those currently provided are:

- *Node* class. Class implementing node in finite element mesh. Node manages its position in space, and if specified, its local coordinate system in node. If local coordinate system is defined, all equilibrium equations are assembled in this system and therefore all DOFs and applied boundary and initial conditions apply in this local coordinate system. By default, global coordinate system is assumed in each node.
- *RigidArmNode* class. Class implementing node connected to other node (master) using rigid arm in finite element mesh. Rigid arm node supports not only slave DOFs mapped to master but also some DOFs can be primary DOFs. The *masterDofMask* attribute is introduced allowing to distinguish between primary and mapped (slave) DOFs. The primary DOFs can have their own boundary and initial conditions. The introduction of rigid arm connected nodes allows to avoid very stiff elements used for modelling the rigid-arm connection. The rigid arm node maps its DOFs to master DOFs using simple transformations (small rotations are assumed). Therefore, the contribution to rigid arm node are localized directly to master related equations. The rigid arm node slave (mapped DOFs can not have its own boundary or initial conditions, they are determined completely from master dof conditions. The local coordinate system in slave is not supported in current implementation, the global lcs applies. On the other hand, rigid arm node can be loaded independently of master. The transformation for DOFs and load is not orthogonal - the inverse transformation can not be constructed by transposition. Because of time consuming inversion, methods can generally compute both transformations for DOFs as well as loads.
- *ElementSide* class - representing finite element side possessing some DOFs.

5.3 Boundary Conditions

The library introduces the base abstract class *GeneralBoundaryCondition* for all boundary conditions (both primary and secondary). Boundary condition is an attribute of the domain (it belongs to). The other system components subjected to boundary conditions keep reference to corresponding boundary conditions, like elements, nodes, and DOFs.

The base class only declares itself as a base class of all boundary conditions, and declares only very basic services. It introduces 'loadTimeFunction' as an attribute of each boundary condition. 'loadTimeFunction' represent time variation, its value is dependent on time step. The value (or the components) of a boundary condition (load) will be the product of its value by the value of the associated load time function at given time step. The meaning of boundary condition components is dependent on particular boundary condition type, and should be defined in derived classes documentation. This base class introduces also two general services for requesting boundary condition physical meaning and boundary condition geometrical character (point-wise, acting on element body or edge and so on).

Derived classes should represent the base classes for particular boundary condition type (like force load, or boundary condition prescribed directly on some DOF) and should declare the basic common interface. For example, the *Load* is derived from *GeneralBoundaryCondition* and represent base class for all boundary conditions representing load. The following derived classes are provided by OOFEMlib

- *Load* class - base abstract class for all loads. Declares the attribute *componentArray* to store load components and method for evaluating the component values array at given time (component array multiplied with load time function value) is provided.
- *NodalLoad* - implementation of a concentrated load/flux (force, moment,...) that acts directly on a dof manager (node or element side, if it has associated DOFs). This load could not be applied on an element. A nodal load is usually attribute of one or more nodes or element sides.
- *BoundaryLoad* - abstract base class representing a boundary load (force, momentum, ...) that acts directly on a boundary of some finite element (on element side, face, ...). Boundary load is usually attribute of one or more finite elements. This base class only declares the common services to all derived classes. Derived classes must implement abstract services and possibly may customize existing. Boundary load is represented by its geometry (determined by its type - linear, quadratic load) and values (it is assumed, that user will supply all necessary values for each dof). The load can generally be specified in global space or can be related to local entity space (related to edge, surface). If load is specified in global space then its values are evaluated at points, which is characterized by global coordinates. If load is specified in entity space, then point is characterized by entity isoparametric coordinates.

- *BodyLoad* - Class representing base class for all element body load, acting over whole element volume (e.g., the dead weight).
- *BoundaryCondition* - class representing Dirichlet boundary condition (primary boundary condition). This boundary condition is usually attribute of one or more degrees of freedom (DOF). The type of unknown (physical meaning) is fully determined by corresponding DOF, to which given BC is associated. Boundary condition can change its value in time using its inherited *loadTimeFunction* attribute. It can also switch itself on or off depending on nonzero value of introduced isImposedTimeFunction load time function. Please note, that previous option must be supported by particular engineering model (because equation renumbering is necessary, and for incremental solution schemes DOFs unknown dictionaries must be used). See particular engineering model documentation for details.

5.4 Initial Conditions

The *InitialCondition* - class implementing general initial condition. Initial condition is usually attribute of one or more degrees of freedom (DOFs). One particular DOF (with its physical meaning - for example displacement) can have associated only single initial condition. Initial condition thus must represent several initial conditions for particular DOF (for example velocity and acceleration of unknown can be prescribed using single initial condition instance). These multiple entries are distinguished by their *CharTypeMode* ` value. The CharTypeMode value is also used as key in *initialValueDictionary*. Initial conditions apply and should be taken into account only in one particular time step, which number is determined from engineering model *giveNumberOfTimeStepWhenIcApply* ` service. If in this time step both boundary condition on unknown and also initial condition on value of this unknown (TotalMode CharTypeMode) are prescribed, then always value reported by boundary condition is taken into account.

CHAPTER

SIX

ELEMENTS

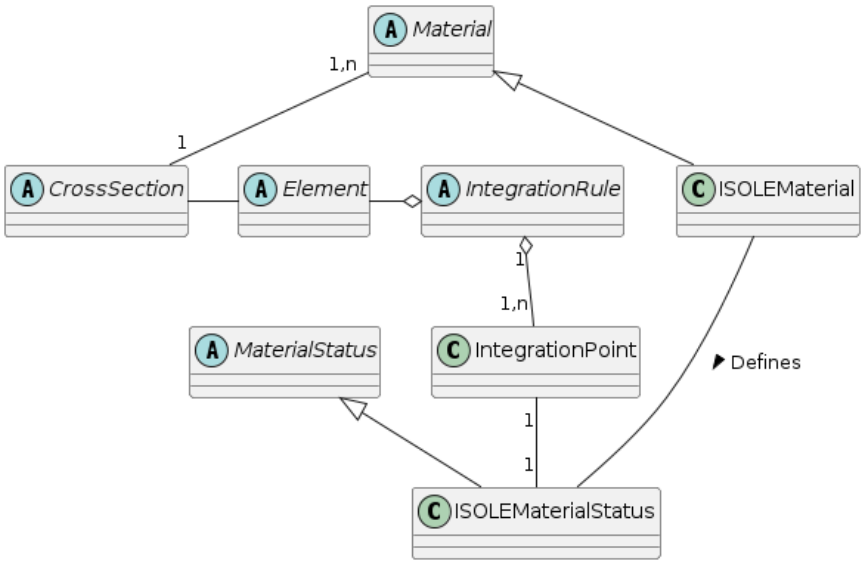
In this frame, the following base classes are introduced:

- Class *Element*, which is an abstraction of a finite element. It declares common general services, provided by all elements. Derived classes are the base classes for specific analysis types (structural analysis, thermal analysis). They declare and implement necessary services for specific analysis.
- Class *IntegrationPoint* is an abstraction for the integration point of the finite element. For historical reasons the *IntegrationPoint* is an alias to *GaussPoint* class. It maintains its coordinates and integration weight. Any integration point can generally contain any number of other integration points - called slaves. The *IntegrationPoint* instance containing slaves is called master. Slaves are, for example, introduced by a layered cross section model, where they represent integration points for each layer, or can be introduced at material model level, where they may represent, for example, micro-planes. Slave integration points are hidden from elements. The *IntegrationPoint* contains associated material status to store state variables (the reasons for introducing this feature will be explained later).
- *CrossSection* class is an abstraction for cross section. Its main role is to hide from an element all details concerning the cross section description and implementation. By cross section description is meant, for example, an integral cross section model, layered cross section model or fibered model. Elements do not communicate directly with material, instead they always use a *CrossSection*, which performs all necessary integration over its volume and invokes necessary material class services. Note, that for some problems, the use of cross section is not necessary, and then the elements can communicate directly with material model. However, for some problems (for example structural analysis) the introduction of cross section is natural.
- *Material* class is the base class for all constitutive models. Derived classes should be the base analysis-specific classes, which declare required analysis specific services (for example structural material class declares services for the stiffness computation and services for the real stress evaluation). Similarly to cross section representation, the material model analysis specific interface, defined in terms of general services, allows the use of any material model, even that added in the future, without modifying any code, because all material models implement the same interface.

One of the most important goals of OOFEM is its extensibility. In the case of extension of the material library, the analyst is facing a key problem. Every material model must store its unique state variables in every integration point. The amount, type, and meaning of these history variables vary for each material model. Therefore, it is not possible to efficiently match all needs and to reflect them in the integration point data structure. The suggested remedy is the following:

The *IntegrationPoint* class is equipped with the possibility to have associated a *MaterialStatus* class instance. When a new material model is implemented, the analyst has also to declare and implement a related material status derived from the base *MaterialStatus* class. This status contains all necessary state variables and related data access and modification services. The *IntegrationPoint* provides services for inserting and accessing its related status. For every *IntegrationPoint*, the corresponding material creates unique copy of its material status and associates it with that integration point. Because the *IntegrationPoint* is a compulsory parameter of all *Material* class methods, the state variables are always accessible.

:: TODO Add a figure of the element-material frame.



OOFEM Top level structure

In Fig (*fig-elementmaterialframe*), the material - element frame is depicted in more detail, although still simplified.

6.1 *Element* class

This class is the base class for all FE elements. It is also the abstract class, declaring some services, which have to be implemented by the derived classes. The main purpose of this class is to define a common interface and attributes, provided by all individual element implementations. The *Element* ` class does neither declare nor implement any method related to specific analysis. These services are to be declared and possibly implemented by derived classes. The direct child of *Element* class are assumed to be the base classes for particular analysis or problem type. They typically introduce the general services required for a specific analysis purpose - like evaluation of stiffness or mass matrices for structural analysis, or evaluation of capacity and conductivity matrices for heat transfer analysis. Usually they also provide generic implementation of these services.

The *Element* class is derived from parent *FEMComponent*, leke many other classes. It inherits the *FEMComponent* ability to keep its number and reference to the domain, it belongs to, its error and warning reporting services. Also the *FEMComponent* class introduces several abstract services. The most important are

- *initializeFrom* for object initialization from a given record,
- *saveContext* and *restoreContext* methods for storing and restoring object state to/from a stream

The attributes defined by the *Element* class include the arrays used to keep its list of nodes and sides, variables to store its material and cross section number, lists of applied body and boundary loads, list of integration rules and array storing its code numbers.

The following important services are declared/introduced by the *Element*:

- services for component management - include services for accesing element's nodes, material and cross section (*giveDofManager*, *giveNode*, *giveNumberOfDofManagers*, *giveNumberOfNodes*, *giveMaterial*, *giveCrossSection*).
- services related to code numbers management:
  - *giveLocationArray* returns the element location array. This location array is obtained by appending code-numbers of element nodes (according to the node numbering), followed by code numbers of element sides.

The ordering of DOFs for the particular node/side is specified using a node/side DOF mask, which is obtained/defined by *giveNodeDofIDMask* or *giveSideDofIDMask* services. Please note, that this local DOF ordering must be taken into account when assembling various local characteristic vectors and matrices. Once the element location array is assembled, it is cached and reused to avoid time consuming assembly. Some engineering models may support dynamic changes of the static system (generally, of boundary conditions) during analysis, then these models use *invalidateLocationArray* function to invalidate location array after finishing time step, to force new equation numbering.

  - *computeNumberOfDofs* - computes or simply returns total number of element's local DOFs. Must be implemented by particular element.
  - *giveDofManDofIDMask* service return DOF mask for corresponding dof manager (node or side). This mask defines the DOFs which are used by element at the given node/side. The mask influences the code number ordering for the particular node. Code numbers are ordered according to the node order and DOFs belonging to the particular node are ordered according to this mask. If element requests DOFs using a node mask which are not in the node then error is generated. This masking allows node to be shared by different elements with different DOFs in the same node/side. Element's local code numbers are extracted from the node/side using this mask. These services must be implemented (overloaded) by particular element implementations.

- services for requesting the so-called characteristic components: *giveCharacteristicMatrix* and *giveCharacteristicVector*. The component requested is identified by parameter of type *CharType* (see cltypes.h). These are general methods for obtaining various element contributions to the global problem. These member functions have to be overloaded by derived analysis-specific classes in order to invoke the proper method according to the type of requested component.
- services related to the solution step update and termination. These services are used to update the internal variables at element's integration points prior to reached state *updateYourself* and *updateInternalState*. Similar service for the internal state initialization is also declared (*initializeYourself*) and re-initialization to previous equilibrium state (see *initForNewStep*).
- services for accessing local element's unknowns from corresponding DOFs. These include methods for requesting local element vector of unknowns (*computeVectorOf*) and local element vector of prescribed unknowns (*computeVectorOfPrescribed*).
- Services for handling transformations between element local coordinate system and coordinate system used in nodes (possibly different from global coordinate system).
- Othetr miscellaneous services. Their detailed description can be found in *Element* class definition in *sr/oofemlib/element.h*.

6.2 Analysis specific element classes

The direct child classes of the *Element* class are supposed to be (but not need to be) base classes for particular problem types. For example, the *StructuralElement* class is the base class for all structural elements. It declares all necessary services required by structural analysis (for example methods computing stiffness matrices, load, strain and stress vectors etc.). This class may provide general implementations of some of these services if possible, implemented using some low-level virtual functions (like computing element shape functions), which are declared, but their implementation is left on derived classes, which implement specific elements.

6.3 Structural element

To proceed, lets take *StructuralElement* class as an example. This class is derived from the *Element* class. The basic tasks of the structural element is to compute its contributions to global equilibrium equations (mass and stiffness matrices, various load vectors (due to boundary conditions, force loading, thermal loading, etc.) and computing the corresponding strains and stresses from nodal displacements. Therefore the corresponding virtual services for computing

these contributions are declared. These standard contributions can be computed by numerical integration of appropriate terms, which typically depend on element interpolation or material model, over the element volume. Therefore, it is possible to provide general implementations of these services, provided that the corresponding methods for computing interpolation dependent terms and material terms are implemented, and corresponding integration rules are initialized.

This concept will be demonstrated on service computing stiffness matrix. Since element stiffness matrix contributes to the global equilibrium, the stiffness will be requested (by engineering model) using *giveCharacteristicMatrix* service.

```
void
StructuralElement ::  giveCharacteristicMatrix (FloatMatrix& answer,
                    CharType mtrx, TimeStep *tStep)
//
// returns characteristic matrix of receiver according to mtrx
//
{
if (mtrx == TangentStiffnessMatrix)
    this -> computeStiffnessMatrix(answer, TangentStiffness, tStep);
else if (mtrx == SecantStiffnessMatrix)
    this -> computeStiffnessMatrix(answer, SecantStiffness, tStep);
else if (mtrx == MassMatrix)
    this -> computeMassMatrix(answer, tStep);
else if
....
}
```

The first parameter is the matrix to be computed, the mtrx parameter determines the type of contribution and the last parameter, time step, represents solution step. Focusing only on material nonlinearity, the element stiffness matrix can be evaluated using well-known formula

$$K = \int_V B^T D B \, dV,$$

where B is the so-called geometrical matrix, containing derivatives of shape functions and D is the material stiffness matrix. If D is symmetric (which is usually the case) then element stiffness is symmetric, too. The numerical integration is used to evaluate the integral. For numerical integration, we will use *IntegrationRule* class instance. The integration rules for a specific element are created during element initialization and are stored in *integrationRulesArray* attribute, defined/introduced by the parent *Element* class. In order to implement the stiffness evaluation, the methods for computing geometrical matrix B and material stiffness matrix D are declared (as virtual), but not implemented. They have to be implemented by specific elements, because they know their interpolation and material mode details. The implementation of *computeStiffnessMatrix* ` is as follows:

```
void
StructuralElement :: computeStiffnessMatrix (FloatMatrix& answer,
                    MatResponseMode rMode, TimeStep* tStep)
// Computes numerically the stiffness matrix of the receiver.
{
int j;
double      dV ;
FloatMatrix d, bj, dbj;
GaussPoint  *gp ;
IntegrationRule* iRule;

// give reference to integration rule
iRule = integrationRulesArray[giveDefaultIntegrationRule()];

```

(continues on next page)

(continued from previous page)

```
// loop over integration points
for (j=0 ; j < iRule->getNumberOfIntegrationPoints() ; j++) {
    gp = iRule->getIntegrationPoint(j) ;
    // compute geometrical matrix of particular element
    this -> computeBmatrixAt(gp, bj) ;
    //compute material stiffness
    this -> computeConstitutiveMatrixAt(d, rMode, gp, tStep);
    // compute jacobian
    dV = this -> computeVolumeAround(gp) ;
    // evaluate stiffness
    dbj.beProductOf (d, bj) ;
    answer.plusProductSymmUpper(bj,dbj,dV) ;
}
answer.symmetrized() ;
return  ;
}
```

Inside the integration loop, only the upper half of the element stiffness is computed in element local coordinate system. Then, the lower part of the stiffness is initialized from the upper part (answer.symmetrized()). The other element contributions can be computed using similar procedures. In general, different integration rules can be used for evaluation of different element contributions. For example, the support for the reduced integration of some terms of the stiffness matrix can be easily supported.

The element strain and stress vectors are to be computed using *computeStrainVector* and *computeStressVector* services. The element strain vector can be evaluated using

$$\varepsilon = Bu,$$

where B is the geometrical matrix and u is element local displacement vector.

```
void
StructuralElement :: computeStrainVector (FloatArray& answer,
                    GaussPoint* gp, TimeStep* stepN)
// Computes the vector containing the strains
// at the Gauss point gp of the receiver,
// at time step stepN. The nature of these strains depends
// on the element's type.
{
FloatMatrix b;
FloatArray  u ;

this -> computeBmatrixAt(gp, b) ;
// compute vector of element's unknowns
this -> computeVectorOf(DisplacementVector,
                        UnknownMode_Total,stepN,u) ;
// transform global unknowns into element local c.s.
if (this->updateRotationMatrix())
    u.rotatedWith(this->rotationMatrix,'n') ;
answer.beProductOf (b, u) ;
return ;
}
```

The stress evaluation on the element level is rather simple, since the stress evaluation from a given strain increment and actual state (kept within the integration point) is done at the cross section and material model levels:

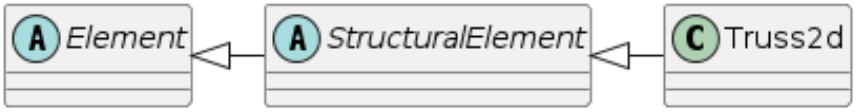
```
void
StructuralElement :: computeStressVector (FloatArray& answer,
                    GaussPoint* gp, TimeStep* stepN)
// Computes the vector containing the stresses
// at the Gauss point gp of the receiver, at time step stepN.
// The nature of these stresses depends on the element's type.
{
FloatArray Epsilon ;
StructuralCrossSection* cs = (StructuralCrossSection*)
                            this->giveCrossSection();
Material *mat = this->giveMaterial();

this->computeStrainVector (Epsilon, gp,stepN) ;
// ask cross section model for real stresses
// for given strain increment
cs -> giveRealStresses (answer, ReducedForm, gp, Epsilon, stepN);
return  ;
}
```

For further reference see src/sm/Elements/structuralelement.h and src/sm/Elements/structuralelement.C files located in your source oofem directory.

6.4 Example: 2D Truss element

In this section, we provide a simple, but complete example of a two-dimensional truss element implementation. The element is derived from the *StructuralElement* class and is called *Truss2d*:



The definition of the *Truss2d* class is as follows (see also sm/src/Elements/Bars/Truss2d.h):

```
class Truss2d : public StructuralElement {
protected :
    double        length ;
    double        pitch ;
public :
    Truss2d (int,Domain*) ;    // constructor
    ~Truss2d ()   {}           // empty destructor
    // mass matrix coputations
    void computeLumpedMassMatrix (FloatMatrix& answer,
                                  TimeStep* tStep) override;
    // general mass service overloaded
    void computeMassMatrix (FloatMatrix& answer, TimeStep* tStep) override
        {computeLumpedMassMatrix(answer, tStep);}
    // DOF management
    virtual int           computeNumberOfDofs (EquationID ut) override {return 4;}
    virtual void          giveDofManDofIDMask  (int inode, EquationID, IntArray& )␣
↪const override;

```

(continues on next page)

(continued from previous page)

```
    double computeVolumeAround (GaussPoint*) override;
    // definition & identification
    char* giveClassName (char* s) const override
         { return strcpy(s,"Truss2d") ;}
    classType giveClassID () const override { return Truss2dClass; }

    IRResultType initializeFrom (InputRecord* ir) override;
protected:
    // computes geometrical matrix
    void computeBmatrixAt (GaussPoint*, FloatMatrix&,
                           int=1, int=ALL_STRAINS) override;
    // computes interpolation matrix
    void computeNmatrixAt (GaussPoint*, FloatMatrix&) override;
    // initialize element's integration rules
    void computeGaussPoints () override;
    // transformation from global->local c.s.
    int           computeGtoLRotationMatrix (FloatMatrix&) override;

    double giveLength () ;
    double givePitch () ;
} ;
```

The *Truss2d* class declares two attributes, the element *length* ` and element *pitch*, defined as angle between global x-axis and element x-axis. They can be computed from coordinates of element nodes, but they are used at different places of implementation and precomputing then can save some proccesing time. We define the constructor and destructor of the *Truss2d* class. Next we define methods to compute characteristic contributions of the element. Note that default implementation of characteristic matrix evaluation (stiffness nad mass matrix) is already provided by parent *StructuralElement* class. We just need to implement the methods for computing geometrical and interpolation matrices. However in this case, the mass matrix is going to be computed using the lumped mass matrix method, and we need to overload the default implementation.

We also need to implement the methods for computing the number of element DOFs (*computeNumberOfDofs*), method to return element DOFs for specific node (*giveDofManDofIDMask*). Also, the method to evaluate the volume associated to given integration point (*computeVolumeAround*), method to initialize element from input record (*initializeFrom*). Finally, in the protected section, there is a declaration of methods for computing the geometrical (*computeBmatrixAt*) and interpolation matrices (*computeNmatrixAt*), as well as the method to initialize the element's integration rules (*computeGaussPoints*) and method to compute the element transformation matrix (*computeGtoLRotationMatrix*). Note than all these methods overload/specialize/define methods declared in the parent *StructuralElement* or *Element* classes. Finally, we declare two methods to compute the element length and pitch.

The implementation of the *Truss2d* class is as follows (see also sm/src/Elements/Bars/Truss2d.C, but note this implementation supports geometrical nonlinearity and is more general), wih some minor methods ommited for brevity: We start with the services for computing interpolation and geometrical matrices. Note that the implementation of the service{computeNmatrixAt} is not necessary for the current purpose (it would be required by the default mass matrix computation), but it is added for completeness. The both methods compute response at a given integration point, which is passed as a parameter. The service{computeBmatrixAt} has two additional parameters which determine the range of strain components for which response is assembled. This has something to do with support for reduced/selective integration and is not important in presented case.

The ordering of element unknowns is following: $r_e = \{u1, y1, u2, y2\}^T$, where u1,u2 are displacements in x-direction and y1,y2 are displacements in y-direction and indices indicate element nodes. This ordering is defined by element node numbering and element nodal unknowns (determined by *giveDofManDofIDMask* method). The vector of unknowns (and also element code numbers) is appended from nodal contributions. The element uses linear interpolation functions, so the element shape functions are linear functions of the local coordinate. The element has 2 nodes, so the element

has 4 DOFs. The interpolation functions are defined as follows:

$$N_1 = \frac{1-\xi}{2}, \quad N_2 = \frac{1+\xi}{2},$$

and their derivatives:

$$\frac{\partial N_1}{\partial \xi} = -\frac{1}{2}, \quad \frac{\partial N_2}{\partial \xi} = \frac{1}{2}.$$

So that the interpolation and geometrical matrices are defined as follows:

$$\{u, v\}^T = N^e r_e, \ \{\varepsilon\} = B^e r_e^l$$

$$B^e = \left[-\tfrac{1}{L} \quad 0 \quad \tfrac{1}{L} \quad 0\right],$$
$$N^e = \begin{bmatrix} N_1 & 0 & N_2 & 0 \\ 0 & N_1 & 0 & N_2 \end{bmatrix},$$

The transformation matrix from unknowns in global coordinate system to element local coordinate system is defined as follows:

$$\cos\theta = (x_2 - x_1)/l, \ \sin\theta = (y_2 - y_1)/l,$$

$$\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{bmatrix}^l = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & \cos\theta & \sin\theta \\ 0 & 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{bmatrix}^g,$$

where θ is the pitch of the element.

Note

Recently, the interpolation classes have been added, that can significantly facilitate the element implementation. They provide shape functions, their derivatives, and transformation Jacobian out of the box.

```
void
Truss2d ::   giveDofManDofIDMask  (int inode, EquationID, IntArray& answer) const {
// returns DofId mask array for inode element node.
// DofId mask array determines the dof ordering requested from node.
// DofId mask array contains the DofID constants (defined in cltypes.h)
// describing physical meaning of particular DOFs.
//IntArray* answer = new IntArray (2);
answer.resize (2);

answer.at(1) = D_u;
answer.at(2) = D_w;

return ;
}

void
Truss2d :: computeNmatrixAt (GaussPoint* aGaussPoint,
                             FloatMatrix& answer)
// Returns the displacement interpolation matrix {N}
```

(continues on next page)

(continued from previous page)

```
// of the receiver, evaluated at aGaussPoint.
{
    double       ksi,n1,n2 ;

    ksi = aGaussPoint -> giveCoordinate(1) ;
    n1  = (1. - ksi) * 0.5 ;
    n2  = (1. + ksi) * 0.5 ;

    answer.resize (2,4);
    answer.zero();

    answer.at(1,1) = n1 ;
    answer.at(1,3) = n2 ;
    answer.at(2,2) = n1 ;
    answer.at(2,4) = n2 ;

    return  ;
}


void
Truss2d :: computeBmatrixAt (GaussPoint* aGaussPoint,
                             FloatMatrix& answer, int li, int ui)
//
// Returns linear part of geometrical
// equations of the receiver at gp.
// Returns the linear part of the B matrix
//
{
double coeff,l;

answer.resize(1,4);
l = this->giveLength();
coeff = 1.0/l;

answer.at(1,1) =-coeff;
answer.at(1,2) = 0.0;
answer.at(1,3) = coeff;
answer.at(1,4) = 0.0;

return;
```

Next, the following two functions compute the basic geometric characteristics of a bar element - its length and pitch, defined as the angle between global x-axis and the local element x-axis (oriented from node1 to node2).

```
double  Truss2d :: giveLength ()
// Returns the length of the receiver.
{
    double dx,dz ;
    Node   *nodeA,*nodeB ;

    if (length == 0.) {
```

(continues on next page)

(continued from previous page)

```
    nodeA = this->giveNode(1) ;
    nodeB = this->giveNode(2) ;
    dx    = nodeB->giveCoordinate(1)-nodeA->giveCoordinate(1);
    dz    = nodeB->giveCoordinate(3)-nodeA->giveCoordinate(3);
    length= sqrt(dx*dx + dz*dz) ;}

    return length ;
}


double  Truss2d :: givePitch ()
// Returns the pitch of the receiver.
{
    double xA,xB,zA,zB ;
    Node   *nodeA,*nodeB ;

    if (pitch == 10.) {  // 10. : dummy initialization value
    nodeA  = this -> giveNode(1) ;
    nodeB  = this -> giveNode(2) ;
    xA     = nodeA->giveCoordinate(1) ;
    xB     = nodeB->giveCoordinate(1) ;
    zA     = nodeA->giveCoordinate(3) ;
    zB     = nodeB->giveCoordinate(3) ;
    pitch  = atan2(zB-zA,xB-xA) ;}

    return pitch ;
}
```

When an element is created, the default constructor is called. To initialize the element, according to its record in the input file, the *initializeFrom* is immediately called after element creation. The element implementation should first call the parent implementation to ensure that attributes declared at parent level are initialized properly. Then the element has to initialize attributes declared by itself and also to set up its integration rules. In our example, special method *computeGaussPoints* is called to initialize integration rules. in this case, only one integration rule is created. It is of type *GaussIntegrationRule*, indicating that the Gaussian integration is used. Once integration rule is created, its integration points are created to represent line integral, with 1 integration point. Integration points will be associated to element under consideration and will have 1D material mode (which determines the type of material model response):

```
IRResultType
Truss2d :: initializeFrom (InputRecord* ir)
{
    this->NLStructuralElement :: initializeFrom (ir);
    this -> computeGaussPoints();
    return IRRT_OK;
}

void  Truss2d :: computeGaussPoints ()
    // Sets up the array of Gauss Points of the receiver.
{

numberOfIntegrationRules = 1 ;
integrationRulesArray = new IntegrationRule*;
integrationRulesArray[0] = new GaussIntegrationRule (1,domain, 1, 2);
```

(continues on next page)

(continued from previous page)

```
integrationRulesArray[0]->
    setUpIntegrationPoints (_Line, 1, this, _1dMat);

}
```

Next, we present method calculating lumped mass matrix:

```
void
Truss2d :: computeLumpedMassMatrix (FloatMatrix& answer, TimeStep* tStep)
    // Returns the lumped mass matrix of the receiver. This expression is
    // valid in both local and global axes.
{
    Material* mat ;
    double    halfMass ;

    mat        = this -> giveMaterial() ;
    halfMass   = mat->give('d') *
                 this->giveCrossSection()->give('A') *
                 this->giveLength() / 2.;

    answer.resize (4,4) ; answer.zero();
    answer . at(1,1) = halfMass ;
    answer . at(2,2) = halfMass ;
    answer . at(3,3) = halfMass ;
    answer . at(4,4) = halfMass ;

    if (this->updateRotationMatrix())
    answer.rotatedWith(*this->rotationMatrix) ;
    return  ;
}
```

Finally, the following method computes the part of the element transformation matrix, corresponding to transformation between global and element local coordinate systems. This method is called from the *updateRotationMatrix* service, implemented at the *StructuralElement* level, which computes the element transformation matrix, taking into account further transformations (nodal coordinate system, for example).

```
int
Truss2d ::  computeGtoLRotationMatrix (FloatMatrix& rotationMatrix)
// computes the rotation matrix of the receiver.
// r(local) = T * r(global)
{
double sine,cosine ;

sine           = sin (this->givePitch()) ;
cosine         = cos (pitch) ;

rotationMatrix.resize(4,4);
rotationMatrix . at(1,1) =  cosine ;
rotationMatrix . at(1,2) =  sine   ;
rotationMatrix . at(2,1) = -sine   ;
rotationMatrix . at(2,2) =  cosine ;
rotationMatrix . at(3,3) =  cosine ;
```

(continues on next page)

(continued from previous page)

```
17 rotationMatrix . at(3,4) =  sine   ;
18 rotationMatrix . at(4,3) = -sine   ;
19 rotationMatrix . at(4,4) =  cosine ;
20
21 return 1 ;
22 }
```

CHAPTER

SEVEN

MATERIAL MODELS

The base class for all material models is the *Material* class, derived from the *FEMComponent* parent. It declares analysis independent part of the material interface. The analysis specific part of the interface should be introduced by derived classes, which are supposed to represent the base classes for specific problem. The typical example is *StructuralMaterial* class which declares the services of material model necessary for the structural analysis. The generic services declared or implemented at top *Material* level include

- Material status related services. The material model has to be able to create instance of corresponding material status, where the history variables are stored. This is done by invoking *CreateStatus*. The status corresponding to a given integration point can be requested using the *giveStatus* service.
- Services for integration point update and initialization. There are generally two sets of history variables kept in corresponding material statuses for each integration point. One set is referring to previous equilibrated state, the second one to the actual state during the solution (more precisely to the achieved local-equilibrium state), which may not correspond to the global equilibrium state. The methods are provided to update the actual state as equilibrated (*updateYourself* service) and for initialization of actual state to previous equilibrium (*initTempStatus*). The implementation of these services simply extract the corresponding status of a given integration point and calls the corresponding service of material status.
- Services for testing the material model capabilities (*testMaterialExtension*, *hasMaterialModeCapability*, and *hasNonLinearBehaviour* services).
- Services for requesting internal variables and properties.

7.1 Analysis specific material model classes

The direct derived classes of the *Material* class, such as *StructuralMaterial* are supposed to declare the analysis specific part of the material model interface, which is required (and assumed) by the corresponding element class (*StructuralElement*) and cross section class (*StructuralCrossSection*).

To store all necessary history variables of the material model, so called status concept is adopted. The material status can be thought as container of all necessary history variables. Usually two kinds of these variables are stored. The temporary ones refer to the actual state of an integration point, but do not necessary correspond to the global equilibrium. These are changing during global equilibrium search iteration. The non-temporary variables are related to the previously converged state. For each material model, the corresponding twin status has to be defined, and the unique instance for each integration point has to be created and associated with it. The integration point provides the services for accessing corresponding status. All material statuses, related to particular material models, have to be derived from the base *MaterialStatus* class. This class declares the basic status interface. The two most important services are:

- *initTempStatus* intended to initialize the temporary internal variables according to variables related to previously reached equilibrium state,
- *updateYourself* designed to update the equilibrium-like history variables according to temporary variables, when the new global equilibrium has been reached. The derived classes should also define methods for accessing the corresponding history variables.

7.2 Structural Material class - Example

The structural (or mechanical) constitutive model should generally support several so-called material models, corresponding to various modeling assumptions (plane-stress, plane-strain, or 1D-behavior, for example). The concept of multiple material modes is generally supported by *Material* class, which provides the services for testing the model capabilities. It is generally assumed, that results obtained from constitutive model services are formed according to material mode. This mode is attribute of each integration point, which is compulsory parameter of all material services. For computational convenience, the so-called full and reduced formats of stress/strains vectors are introduced, corresponding to material modes. The full format includes all components, even if they are zero due to stress/strain mode nature. In the reduced format, only generally nonzero components are stored. (Full format should be used only if absolutely necessary, to avoid wasting of space. For example, it is used by output routines to print results in general form). Methods for converting vectors between full and reduced format are provided. If possible, all computations should be performed in reduced space.

```
The convention used to construct reduced strain/stress vectors is as follows:
If in a particular mode a particular stress component is zero, the corresponding strain␣
↪is not computed
and not stored in reduced vector, and in full vector there is zero value on␣
↪corresponding position.
On the other hand, if zero strain component is imposed, then this condition must be␣
↪taken into
account in geometrical relations (at element level), and corresponding components are␣
↪included
in stress/strain reduced vectors.
```

Generally, the following major tasks are declared by *StructuralMaterial* or inherited from *Material* class:

- Computing the real stress vector (tensor) at an integration point for a given strain increment and updating its temporary state corresponding to the local equilibrium, but not necessarily to the global equilibrium (see *giveRealStressVector*). The parameters include the total strain vector and the corresponding integration point. The total strain is defined as strain computed directly from the displacement field at a given time. The stress independent parts (temperature, eigen strains) should be subtracted and the corresponding load-history variables (stored in the corresponding status) can be used. The temporary history variables in the status should be updated according to the newly reached state. The temporary history variables are moved into equilibrium history variables just after the global structure equilibrium has been reached by the iteration process.
- Updating the integration point state (final state), when the global equilibrium has been reached.
- Returning material stiffness and/or flexibility matrices for a given material mode. The general methods computing the response for the specific material mode are provided, based on converting 3D stiffness or compliance matrix to that corresponding to the specific material mode. But, if it is possible to compute stiffness or compliance matrix directly for the specific mode, then these general methods should be overloaded.
- Storing/restoring integration point state to/from a stream.
- Requesting internal variables, their types and properties (*giveIPValue*, *giveIPValueSize*, *giveIPValueType*, and *giveIntVarCompFullIndx* services).
- Returning material properties.

Structural material services should not be called directly by elements. Instead, the elements should pass their requests to the corresponding cross section model, that performs all necessary integration over its volume and invokes corresponding material model services.

The *StructuralMaterial* class comes with definition of associated material status - *StructuralMaterialStatus*. This is only an abstract class. For every instance of *StructuralMaterial* class there should be a twin statu class, which defines and maintains all necessary state variables. The *StructuralMaterialStatus* only introduces attributes common to all structural analysis material models - the strain and stress vectors (referring both to temporary state, corresponding to

the local equilibrium and non-temporary ones, corresponding to the global equilibrium). The corresponding services for accessing, setting, initializing, and updating these attributes are provided.

7.3 Isotropic Damage Model - Example

In this section, the implementation of ann isotropic damage model will be described. To cover the various models based on isotropic damage concept, a base class *IsotropicDamageMaterial* is defined first, declaring the necessary services and providing the implementation of them, which are general. The derived classes then only implement a particular damage-evolution law.

The isotropic damage models are based on the simplifying assumption that the stiffness degradation is isotropic, i.e., stiffness moduli corresponding to different directions decrease proportionally and independently of direction of loading. Consequently, the damaged stiffness matrix is expressed as

$$D = (1 - \omega)D_e,$$

where D_e is elastic stiffness matrix of the undamaged material and ω is the damage parameter. Initially, ω is set to zero, representing the virgin undamaged material, and the response is linear-elastic. As the material undergoes the deformation, the initiation and propagation of microdefects decreases the stiffness, which is represented by the growth of the damage parameter ω. For $\omega = 1$, the stiffness completely disappears.

In the present context, the D matrix represents the secant stiffness that relates the total strain to the total stress

$$\sigma = D\varepsilon = (1 - \omega)D_e\varepsilon.$$

Similarly to the theory of plasticity, a loading function \$f is introduced. In the damage theory, it is natural to work in the strain space and therefore the loading function is depending on the strain and on an additional parameter κ, describing the evolution of the damage. Physically, κ is a scalar measure of the largest strain level ever reached. The loading function usually has the form

$$f(\varepsilon, \kappa) = \tilde{\varepsilon}(\varepsilon) - \kappa,$$

where $\tilde{\varepsilon}$ is the equivalent strain, i.e., the scalar measure of the strain level. Damage can grow only if current state reaches the boundary of elastic domain ($f = 0$). This is expressed by the following loading/unloading conditions

$$f \leq 0, \quad \dot{\kappa} \geq 0, \quad \dot{\kappa} f = 0.$$

It remains to link the variable κ to the damage parameter ω. As both κ and ω grow monotonically, it is convenient to postulate an explicit evolution law

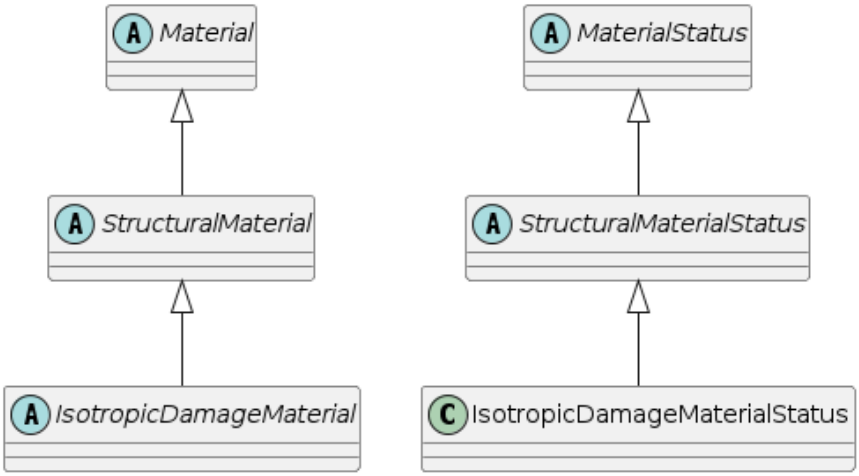
$$\omega = g(\kappa).$$

The important advantage of this explicit formulation is that the stress corresponding to the given strain can be evaluated directly, without the need to solve the nonlinear system of equations. For the given strain, the corresponding stress is computed simply by evaluating the current equivalent strain, updating the maximum previously reached equivalent strain value κ and the damage parameter and reducing the effective stress according to $\sigma = (1 - \omega)D_e\varepsilon$.

This general framework for computing stresses and stiffness matrix is common for all material models of this type. Therefore, it is natural to introduce the base class for all isotropic-based damage models which provides the general implementation for the stress and stiffness matrix evaluation algorithms. The particular models then only provide their equivalent strain and damage evolution law definitions. The base class only declares the virtual services for computing equivalent strain and corresponding damage. The implementation of common services uses these virtual functions, but they are only declared at *IsotropicDamageMaterial* class level and have to be implemented by the derived classes.

Together with the material model, the corresponding status has to be defined, containing all necessary history variables. For the isotropic-based damage models, the only history variable is the value of the largest strain level ever

reached (κ). In addition, the corresponding damage level $omega$ will be stored. This is not necessary because damage can be always computed from corresponding κ. The *IsotropicDamageMaterialStatus* class is derived from *StructuralMaterialStatus* class. The base class represents the base material status class for all structural statuses. At *StructuralMaterialStatus* level, the attributes common to all structural analysis material models - the strain and stress vectors (both the temporary and non-temporary) are introduced. The corresponding services for accessing, setting, initializing, and updating these attributes are provided. Therefore, only the κ and ω parameters are introduced (both the temporary and non-temporary). The corresponding services for manipulating these attributes are added and services for context initialization, update, and store/restore operations are overloaded, to handle the history parameters properly.



The declaration of the *IsotropicDamageMaterialStatus* class follows:

```
class IsotropicDamageMaterialStatus : public StructuralMaterialStatus {
protected:
/// scalar measure of the largest strain level ever reached in material
double kappa;
/// non-equilibrated scalar measure of the largest strain level
double tempKappa;
/// damage level of material
double damage;
/// non-equilibrated damage level of material
double tempDamage;

public:
/// Constructor
IsotropicDamageMaterialStatus (int n, Domain*d, GaussPoint* g) ;
/// Destructor
~IsotropicDamageMaterialStatus ();

/// Prints the receiver state to stream
void   printOutputAt (FILE *file, TimeStep* tStep) ;

/// Returns the last equilibrated scalar measure
/// of the largest strain level
double giveKappa () {return kappa;}
/// Returns the temp. scalar measure of the
/// largest strain level
double giveTempKappa () {return tempKappa;}
```

(continues on next page)

(continued from previous page)

```
/// Sets the temp scalar measure of the largest
/// strain level to given value
void   setTempKappa (double newKappa) { tempKappa = newKappa;}
/// Returns the last equilibrated damage level
double giveDamage () {return damage;}
/// Returns the temp. damage level
double giveTempDamage () {return tempDamage;}
/// Sets the temp damage level to given value
void   setTempDamage (double newDamage) { tempDamage = newDamage;}


// definition
char*                giveClassName (char* s) const
    { return strcpy(s,"IsotropicDamageMaterialModelStatus") ;}
classType            giveClassID () const
    { return IsotropicDamageMaterialStatusClass; }

/**
    Initializes the temporary internal variables,
    describing the current state according to
    previously reached equilibrium internal variables.
*/
virtual void initTempStatus ();
/**
    Update equilibrium history variables
    according to temp-variables.
    Invoked, after new equilibrium state has been reached.
*/
virtual void updateYourself(TimeStep*);

// saves current context(state) into stream
/**
    Stores context of receiver into given stream.
    Only non-temp internal history variables are stored.
    @param stream stream where to write data
    @param obj pointer to integration point, which invokes this method
    @return nonzero if o.k.
*/
contextIOResultType    saveContext (FILE* stream, void *obj = NULL);
/**
    Restores context of receiver from given stream.
    @param stream stream where to read data
    @param obj pointer to integration point, which invokes this method
    @return nonzero if o.k.
*/
contextIOResultType    restoreContext(FILE* stream, void *obj = NULL);
};
```

The base *IsotropicDamageMaterial* class is derived from the *StructuralMaterial* class. The generic methods defined by *StructuralMaterial* require only to implement two fundamental services:

- *give3dMaterialStiffnessMatrix* for evaluating the material stiffness matrix in full, 3D mode,
- *giveRealStressVector_3d* for computing the real stress vector in 3D mode.

The remaining material modes are derived from these two basic services. However, the derived classes can reimplement (overload) these services, when desired. In our case, it is possible to obtain reduced stress and stiffness in more efficient way, than done in generic way on *StructuralMaterial* level and which requires matrix inversion. Therefore, the services for reduced modes are overloaded.

```
class IsotropicDamageMaterial : public StructuralMaterial
{
protected:
    /// Coefficient of thermal dilatation.
    double tempDillatCoeff = 0.;

    /// Maximum limit on omega. The purpose is elimination of a too compliant material which may cause convergence problems. Set to something like 0.99 if needed.
    double maxOmega = 0.999999;

    /// Indicator of the type of permanent strain formulation (0 = standard damage with no permanent strain)
    int permStrain = 0;

    /// Reference to bulk (undamaged) material
    LinearElasticMaterial *linearElasticMaterial = nullptr;
    /**
    * Variable controlling type of loading/unloading law, default set to idm_strainLevel
    * defines the two two possibilities:
    * - idm_strainLevelCR the unloading takes place, when strain level is smaller than the largest level ever reached;
    * - idm_damageLevelCR the unloading takes place, when damage level is smaller than the largest damage ever  reached;
    */
    enum loaUnloCriterium { idm_strainLevelCR, idm_damageLevelCR } llcriteria = idm_strainLevelCR;

public:
    /// Constructor
    IsotropicDamageMaterial(int n, Domain *d);
    /// Destructor
    virtual ~IsotropicDamageMaterial();

    bool hasMaterialModeCapability(MaterialMode mode) const override;
    const char *giveClassName() const override { return "IsotropicDamageMaterial"; }

    /// Returns reference to undamaged (bulk) material
    LinearElasticMaterial *giveLinearElasticMaterial() { return linearElasticMaterial; }

    FloatMatrixF<6,6> give3dMaterialStiffnessMatrix(MatResponseMode mode, GaussPoint *gp, TimeStep *tStep) const override;

    void giveRealStressVector(FloatArray &answer, GaussPoint *gp,
                              const FloatArray &reducedStrain, TimeStep *tStep) override;

    FloatArrayF<6> giveRealStressVector_3d(const FloatArrayF<6> &strain, GaussPoint *gp, TimeStep *tStep) const override
    {
```

(continues on next page)

(continued from previous page)

```
42          FloatArray answer;
43          const_cast<IsotropicDamageMaterial*>(this)->giveRealStressVector(answer, gp,␣
↪strain, tStep);
44          return answer;
45      }
46      FloatArrayF<4> giveRealStressVector_PlaneStrain( const FloatArrayF<4> &strain,␣
↪GaussPoint *gp, TimeStep *tStep) const override
47      {
48          FloatArray answer;
49          const_cast<IsotropicDamageMaterial*>(this)->giveRealStressVector(answer, gp,␣
↪strain, tStep);
50          return answer;
51      }
52      FloatArray giveRealStressVector_StressControl(const FloatArray &strain, const␣
↪IntArray &strainControl, GaussPoint *gp, TimeStep *tStep) const override
53      {
54          FloatArray answer;
55          const_cast<IsotropicDamageMaterial*>(this)->giveRealStressVector(answer, gp,␣
↪strain, tStep);
56          return answer;
57      }
58      FloatArrayF<3> giveRealStressVector_PlaneStress(const FloatArrayF<3> &strain,␣
↪GaussPoint *gp, TimeStep *tStep) const override
59      {
60          FloatArray answer;
61          const_cast<IsotropicDamageMaterial*>(this)->giveRealStressVector(answer, gp,␣
↪strain, tStep);
62          return answer;
63      }
64      FloatArrayF<1> giveRealStressVector_1d(const FloatArrayF<1> &strain, GaussPoint *gp,
↪ TimeStep *tStep) const override
65      {
66          FloatArray answer;
67          const_cast<IsotropicDamageMaterial*>(this)->giveRealStressVector(answer, gp,␣
↪strain, tStep);
68          return answer;
69      }
70
71      int giveIPValue(FloatArray &answer, GaussPoint *gp, InternalStateType type,␣
↪TimeStep *tStep) override;
72
73      FloatArrayF<6> giveThermalDilatationVector(GaussPoint *gp, TimeStep *tStep) const␣
↪override;
74      virtual double evaluatePermanentStrain(double kappa, double omega) const { return 0.
↪; }
75
76      /**
77      * Returns the value of material property 'aProperty'. Property must be identified
78      * by unique int id. Integration point also passed to allow for materials with␣
↪spatially
79      * varying properties
80      * @param aProperty ID of property requested.
```

(continues on next page)

(continued from previous page)

```
     * @param gp Integration point,
     * @return Property value.
     */
    double give(int aProperty, GaussPoint *gp) const override;
    /**
     * Computes the equivalent strain measure from given strain vector (full form).
     * @param[out] kappa Return parameter, containing the corresponding equivalent strain.
     * @param strain Total strain vector in full form.
     * @param gp Integration point.
     * @param tStep Time step.
     */
    virtual double computeEquivalentStrain(const FloatArray &strain, GaussPoint *gp, TimeStep *tStep) const = 0;
    /**Computes derivative of the equivalent strain with regards to strain
     * @param[out] answer Contains the resulting derivative.
     * @param strain Strain vector.
     * @param gp Integration point.
     * @param tStep Time step.
     */
    virtual void computeEta(FloatArray &answer, const FloatArray &strain, GaussPoint *gp, TimeStep *tStep) const { OOFEM_ERROR("not implemented"); }
    /**
     * Computes the value of damage parameter omega, based on given value of equivalent strain.
     * @param[out] omega Contains result.
     * @param kappa Equivalent strain measure.
     * @param strain Total strain in full form.
     * @param gp Integration point.
     */
    virtual double computeDamageParam(double kappa, const FloatArray &strain, GaussPoint *gp) const = 0;

    void initializeFrom(InputRecord &ir) override;
    void giveInputRecord(DynamicInputRecord &input) override;

    MaterialStatus *CreateStatus(GaussPoint *gp) const override { return new IsotropicDamageMaterialStatus(gp); }

    FloatMatrixF<1,1> give1dStressStiffMtrx(MatResponseMode mmode, GaussPoint *gp,
                                            TimeStep *tStep) const override;

    void saveContext(DataStream &stream, ContextMode mode) override;
    void restoreContext(DataStream &stream, ContextMode mode) override;

protected:
    /**
     * Abstract service allowing to perform some initialization, when damage first appear.
     * @param kappa Scalar measure of strain level.
     * @param totalStrainVector Current total strain vector.
     * @param gp Integration point.
```

(continues on next page)

(continued from previous page)

```
    */
    virtual void initDamaged(double kappa, FloatArray &totalStrainVector, GaussPoint␣
↪*gp) const { }

    /**
    * Returns the value of derivative of damage function
    * wrt damage-driving variable kappa corresponding
    * to a given value of the  kappa, depending on
    * the type of selected damage law.
    * @param kappa Equivalent strain measure.
    * @param gp Integration point.
    */
    virtual double damageFunctionPrime(double kappa, GaussPoint *gp) const {
        OOFEM_ERROR("not implemented");
        return 0;
    }

    FloatMatrixF<3,3> givePlaneStressStiffMtrx(MatResponseMode mmode, GaussPoint *gp,
↪TimeStep *tStep) const override;
    FloatMatrixF<4,4> givePlaneStrainStiffMtrx(MatResponseMode mmode, GaussPoint *gp,␣
↪TimeStep *tStep) const override;

};
```

Finally we present the implementation of the selected methods of *IsotropicDamageMaterial* class. Let us start with the implementation of the *give3dMaterialStiffnessMatrix* method, which computes the 3D material stiffness, respecting the previous loading history described by state variable:

```
FloatMatrixF<6,6>
IsotropicDamageMaterial :: give3dMaterialStiffnessMatrix(MatResponseMode mode,
                                                         GaussPoint *gp,
                                                         TimeStep *tStep) const
//
// computes full constitutive matrix for case of gp stress-strain state.
//
{
    auto status = static_cast< IsotropicDamageMaterialStatus * >( this->giveStatus(gp)␣
↪);
    double tempDamage;
    if ( mode == ElasticStiffness ) {
        tempDamage = 0.0;
    } else {
        tempDamage = status->giveTempDamage();
        tempDamage = min(tempDamage, maxOmega);
    }

    auto d = this->linearElasticMaterial->give3dMaterialStiffnessMatrix(mode, gp,␣
↪tStep);
    return d * (1.0 - tempDamage);
    //TODO - correction for tangent mode
}
```

The *giveRealStressesVector* method computes the real stress respecting the previous loading history (described by

state variables) and the given strain vector. It computes a new locally consistent (equilibrated) state, which is stored in temporary variables of the corresponding status (attribute of the integration point).

```
void
IsotropicDamageMaterial :: giveRealStressVector(FloatArray &answer, GaussPoint *gp,
                                                const FloatArray &totalStrain,
                                                TimeStep *tStep)
//
// returns real stress vector in 3d stress space of receiver according to
// previous level of stress and current
// strain increment, the only way, how to correctly update gp records
//
{
    IsotropicDamageMaterialStatus *status = static_cast< IsotropicDamageMaterialStatus␣
↪* >( this->giveStatus(gp) );
    LinearElasticMaterial *lmat = this->giveLinearElasticMaterial();
    FloatArray reducedTotalStrainVector;
    FloatMatrix de;
    double f, equivStrain, tempKappa = 0.0, omega = 0.0;

    this->initTempStatus(gp);

    // subtract stress-independent part
    // note: eigenStrains (temperature) are present in strains stored in gp
    // therefore it is necessary to subtract always the total eigen strain value
    this->giveStressDependentPartOfStrainVector(reducedTotalStrainVector, gp,␣
↪totalStrain, tStep, VM_Total);

    // compute equivalent strain
    equivStrain = this->computeEquivalentStrain(reducedTotalStrainVector, gp, tStep);

    if ( llcriteria == idm_strainLevelCR ) {
        // compute value of loading function if strainLevel crit apply
        f = equivStrain - status->giveKappa();

        if ( f <= 0.0 ) {
            // damage does not grow
            tempKappa = status->giveKappa();
            omega     = status->giveDamage();
        } else {
            // damage grows
            tempKappa = equivStrain;
            this->initDamaged(tempKappa, reducedTotalStrainVector, gp);
            // evaluate damage parameter
            omega = this->computeDamageParam(tempKappa, reducedTotalStrainVector, gp);
        }
    } else if ( llcriteria == idm_damageLevelCR ) {
        // evaluate damage parameter first
        tempKappa = equivStrain;
        this->initDamaged(tempKappa, reducedTotalStrainVector, gp);
        omega = this->computeDamageParam(tempKappa, reducedTotalStrainVector, gp);
        if ( omega < status->giveDamage() ) {
            // unloading takes place
```

(continues on next page)

(continued from previous page)

```
            omega = status->giveDamage();
        }
    } else {
        OOFEM_ERROR("unsupported loading/unloading criterion");
    }

    // get material stiffness from bulk material
    lmat->giveStiffnessMatrix(de, SecantStiffness, gp, tStep);

    // damage deactivation in compression for 1D model
    if ( ( reducedTotalStrainVector.giveSize() > 1 ) || ( reducedTotalStrainVector.
↪at(1) > 0. ) ) {
        //emj
        de.times(1.0 - omega);
    }

    answer.beProductOf(de, reducedTotalStrainVector);

    // update gp status
    status->letTempStrainVectorBe(totalStrain);
    status->letTempStressVectorBe(answer);
    status->setTempKappa(tempKappa);
    status->setTempDamage(omega);
}
```

The above general class provides the general framework. The derived classes have to implement only services *computeEquivalentStrain* and *computeDamageParam*, which are only related to particular material model under consideration. For full reference, please refer to the src/sm/Materials/isotropicdamagematerial.h and src/sm/Materials/isotropicdamagematerial.C files.

For example of simple Isotropic Damage Model, please refer to the src/sm/Materials/ConcreteMaterials/idm1.h and src/sm/Materials/ConcreteMaterials/idm1.C files.

CHAPTER

EIGHT

UTILITY CLASSES

8.1 Vectors and Matrices

The OOFEMlib provides the abstraction for integer vectors (*IntArray*) and for real vectors and matrices (*FloatArray* and *FloatMatrix* classes). The usual arithmetic operations (like vector and matrix additions, subtractions, matrix vector multiplication, matrix inverse, solution of linear system, finding eigenvalues and eigenvectors) are provided. However, the usual math operators ('+','*') are not overloaded, user has to call specific routines. The vector and matrices provide both 0-based and 1-based component access, they allow for dynamic resize with optional chunk. In fact, the current implementation of *resize* only grows the receiver, the possible request for shrink does not cause the reallocation of memory, since allocated memory is kept for future possible resize. If resize operation wants to force allocation, then *hardResize* method should be invoked instead. The preferred argument passing is by reference, even for function or procedure return values. The called function performs resize and fills up the return parameter(s). This is motivated by aim to avoid memory allocation/de-allocation problems. The programer should always avoid to return pointers to newly allocated arrays or matrices. See section **`Coding Standards`_** for details.

8.2 Solution steps

The class representing solution step is *TimeStep*. It may represent either time step, load increment, or load case depending on current Engineering model used.

Solution step maintain the reference to corresponding Engineering model class instance. It maintain also its "intrinsic time" and corresponding time increment. The meaning of these values is dependent on current Engineering model used. The time may represent either current time, load increment number or load case number. See corresponding EngngModel reference for details.

Some components (typically integration points real stresses or integration points non-local values) are computationally very demanding. Because in typical code, there are number of requests for same value during the computation process, it may be efficient to store these values and compute them only once. The principal problem is to recognize, when is necessary to re-compute these stored values to reflect newly reached state. This cannot be determined form solution step "time", because solution step may represent for example load increment, inside which typically many iterations are needed to reach convergence. For this purpose, a concept of solution state counters is provided. Whenever the solution state changes, the engineering model updates the solution state counter. The solution state counter is guaranteed to grow up smoothly (it newer decreases) during solution process. Other components of program (integration points) can then store their computationally expensive values but have to store also corresponding solution state counter value valid when these were computed. Then their can easily check for difference between freezed solution state counter for their value with current solution state requested from solution step and recompute the values if necessary.

8.3 Load Time Functions

Abstract base class representing load time function. Classes derived from Load class typically describe load from spatial point of view. The purpose of introducing load time function is to express variation of some components in time. Load time function typically belongs to domain and is attribute of one or more loads. Generally load time function is real function of time ($y = f(t)$).

CHAPTER

NINE

XFEM MODULE

An entity subject to XFEM modeling is described by an *EnrichmentItem*. Such an entity can be e.g. a crack or a material interface. The *EnrichmentItem* is responsible for the geometrical description as well as the enrichment functions. The *EnrichmentItem* is also responsible for the necessary updates if the interface is moving (e.g. if a crack is propagating). The *EnrichmentItem* has different data members to fulfill these requirements:

- The *EnrichmentItem`has an `EnrichmentDomain* that is responsible for the geometrical description. The *EnrichmentDomain* can be e.g. a polygon line or a list of node numbers.
- The *EnrichmentItem* has an *EnrichmentFunction* that describes the enrichment along the crack or interface.
- The *EnrichmentItem* has an *EnrichmentFront* that specifies a special treatment of the ends of the interface if desired. It may e.g. apply branch functions within a specified radius around the crack tips.
- The *EnrichmentItem* has a *PropagationLaw* that describes how the interface evolves in time. It may e.g. propagate a crack in a specified direction.

The enrichment functions are represented by classes derived from the pure virtual *EnrichmentFunction* class. In particular, the following methods have to be overloaded by all subclasses of *EnrichmentFunction*:

- *evaluateEnrFuncAt*: Computes the value of the enrichment function in a given point for a given level set value.
- *evaluateEnrFuncDerivAt*: Computes the gradient of the enrichment function in a given point for a given level set value.
- *giveJump*: Returns the jump of the enrichment function when the level set field changes sign, e.g. 1 for Heaviside, 2 for Sign and 0 for abs enrichment. This method is needed for a generic implementation of cohesive zones.

The enrichment items are stored in an array in the *XfemManager* class, that also provides initialization and access to individual enrichment items. The purpose of the *XfemManager* is to encapsulate XFEM specific functionality. This avoids to a certain extent the need to modify and penetrate the original code.

When a certain node in the problem domain is subjected to enrichment, the corresponding enrichment item has to introduce additional degree(s) of freedom in that node. Since several enrichment items can be active in a node, all DOFs are assigned with a label, that allows to distinguish between them (in traditional FE codes labels are also used to distinguish physical meaning of particular DOF representing displacement along x,y,z axes, rotations, etc.). The *XfemManager* class maintains a pool of available labels and is responsible for their assignment to individual enrichment items.

Each enrichment item keeps its assigned DOF labels (provided by *XfemManager*). These are used by individual enrichment items to introduce and later identify related DOFs in individual nodes. The physical meaning of these DOFs depends on the enrichment item. For example, a single DOF is needed to represent slip along a slip line and two DOFs are required for a displacement discontinuity in 2D modeled with Heaviside enrichment.

If the enrichment items evolve in time, the geometrical description needs to be updated and new enriched DOFs may need to be created. This is done by the *updateGeometry* method in the *XfemManager* class. The sub-triangulation of cut elements may have to be updated as a result of the geometrical update (e.g. previously uncut elements may be cut by a crack and therefore need to be subdivided). The solver class *XFEMStatic* handles this problem by creating a new

domain with a new sub-triangulation and mapping state variables as well as primary variables from the old domain to the new domain.

The sub-triangulation of cut elements as well as computation of B-matrices for 2D XFEM elements are to a large extent encapsulated in the *XfemElementInterface* class. Methods needed for cohesive zones are also implemented here. The 2D elements with XFEM functionality are therefore derived from the standard *Element* base class, as well as from the *XfemElementInterface*.

When a cut element has been subdivided by the *XfemElementInterface*, a *PatchIntegrationRule* class is used to represent the integration scheme, where Gauss integration is employed on individual triangles. When initializing the integration points, their local coordinates (related to the triangle geometry) are mapped into parent element coordinates and the corresponding integration weights are scaled accordingly. There is no need to store individual patch geometries, as the parent element geometry is used to correctly evaluate transformation Jacobians (we may however wish to store the them for debugging or visualization).

If a cohesive crack model is employed, additional integration points along the discontinuity are needed. These are stored separately, see *XfemElementInterface* class.

The following test cases involve the XFEM module:

- sm/xFemCrackVal.in
- sm/xFemCrackValBranch.in
- sm/xfemCohesiveZone1.in
- benchmark/xfem01.in
- sm/plasticRemap1.in

CHAPTER

TEN

IGA MODULE

In the traditional FEA context, *Element* class is parent class for finite elements. It maintains list of nodes, boundary conditions (loads), integration rules, it keeps links to its interpolation and associated material model and provides general services for accessing its attributes, methods for returning its code numbers, and abstract services for the evaluation of characteristic components (e.g. stiffness matrix, load vector, etc.). In the proposed IGA module, the individual NURBS (or B-spline) patches are represented by classes derived from the base *IGAElement* class which is in turn derived from *Element* class. As a consequence, the B-spline patch is represented and treated as a single entity, which returns its contributions that are localized into global system of characteristic equations. This is important also in the sense of slightly different handling of primary unknowns, compared to the standard FEA. Moreover, it is quite natural because in a general case, degrees of freedom (DOFs) on all nodes (control points) of the patch may interact with each other.

One of the fundamental issues, that has to be addressed at the element level, is the integration. The integration, usually the Gaussian integration, is performed over the nonzero knot spans. Individual integration points are represented by *GaussPoint* class. They maintain their spatial coordinates and corresponding integration weights. The individual integration points are set up and grouped together by base *IntegrationRule* class. For example, *GaussIntegrationRule* class, derived from *IntegrationRule* base, implements Gauss quadrature. An important feature is that an element may have several integration rules. This is very useful for the implementation of the reduced or selective integration, for example. The concept of multiple integration rules per element is extended in the present context of the IGA. The *IGA_IntegrationElement* class is introduced to represent the integration rule for individual nonzero knot spans. Since it is derived from *GaussIntegrationRule*, it inherits the capability to set up Gauss integration points. The reason for creating a new class is to introduce the new attribute *knotSpan*, where the corresponding knot span is stored. This information is used in interpolation class to evaluate corresponding basis functions and their derivatives at a given integration point. Note that generally, the active knot span can be determined for each integration point on the fly whenever it is needed, but in our approach this information is stored to save the computational time. The *IGA_IntegrationElement* instances are set up for each nonzero knot span. The element integration then consists of a loop over individual nonzero knot spans, i.e. the loop over *IGA_IntegrationElements* and by inner loop over the individual integration points of the particular knot span.

The individual IGA elements are derived from *IGAElement* class, further derived from the base *Element*. The purpose of *IGAElement* class is to provide general method for initialization of individual integration rules on nonzero knot spans (represented by *IGA_IntegrationElement* class). The integration rules provide methods to create individual integration points in parametric space. An efficient implementation requires to map coordinates of individual integration points with parametric coordinates related to knot spans directly to knot vector based space. Specific elements are derived from the base *Element* (or *IGAElement* class), that delivers the generic element part and also from one or more classes implementing problem-specific functionality, such as *BSplinePlaneStressElement*. In the presented approach, *BSplinePlaneStressElement* is derived from *StructuralElementEvaluator*, which is an abstract base class, that defines the interface for structural analysis, which includes methods to evaluate mass and stiffness matrices, load vectors, etc. Some of the methods are already implemented at this level, such as stiffness matrix evaluation, based on declared abstract services (evaluation of strain-displacement matrix, etc.), which have to be implemented by derived classes. The example of the evaluation of the element stiffness matrix, which can be used by both classical and IGA based elements, is presented using symbolic code:

```
StructuralElementEvaluator::computeStiffnessMatrix() {
  loop on all integration rules of the element:
    loop on all Gauss points of the IntegrationRule:
      B = this->computeStrainDisplacementMatrix(gp);
      D = this->computeConstitutiveMatrix(gp);
      dV = this->computeVolumeAround(gp);
      stiffnessMatrix->add(product of B^T_D_B_dV);
}
```

In the structural analysis context, classes derived from *StructuralElementEvaluator* implement desired functionality for specific types of structural analyzes (plane-stress, plane-strain, full 3D, etc). Provided that the element defines its interpolation (B-spline basis functions), it is possible to evaluate remaining abstract methods from *StructuralElementEvaluator* interface. Thus, when a new element is defined, it has to create its own interpolation and should be derived from *Element* class, which delivers the general basic element functionality and from one or more evaluators, implementing analysis-specific functionality. Such a design, based on decoupled representation of element interpolation and problem specific evaluators, has several advantages. It allows to define problem specific methods only once for several elements with different geometry and interpolation and allows straightforward implementation of elements for multi-physics simulations.

```
PlaneStressStructuralElementEvaluator::
computeStrainDisplacementMatrix(IntegrationPoint gp) {
FEInterpolation interp = gp->giveElement()->giveInterpolation();
interp->evalShapeFunctDerivatives (der, gp);

answer.resize(3, nnode*2);              // 2 DOFs per each node
answer.zero();

for i=1:nnode
    // epsilon_x
    answer.at(1, i*2-1) = der.at(i, 1);   // dN(i)/dx
    // epsilon_y
    answer.at(2, i*2)   = der.at(i, 2);   // dN(i)/dy
    // shear strain
    answer.at(3, i*2-1) = der.at(i, 2);   // dN(i)/dy
    answer.at(3, i*2)   = der.at(i, 1);   // dN(i)/dx
}
```

The description of the element interpolation is encapsulated into a class derived from *FEInterpolation* class which defines the abstract interface in terms of services that evaluate shape functions, their derivatives, jacobian matrix, etc. at given point. In the frame of presented work, the *BSplineInterpolation* class has been implemented. Each finite element has to set up its interpolation and provide access to it. This is enforced by general *Element* interface, that requires to define the method for accessing element interpolation. The abstract *FEInterpolation* class interface is essential, as it allows to implement problem specific element methods already at the top level (like the evaluation of element interpolation or strain-displacement matrices). An efficient implementation should profit from the locality of individual interpolation functions which have limited support over several consecutive knot spans. Therefore methods declared by *FEInterpolation* class evaluating values of interpolation functions or their derivatives return the values only for those, that are nonzero in actual knot span. This enables to compute characteristic element contributions on a knot span basis efficiently. For each individual knot span, the contributions are computed only for generally nonzero shape functions and then are localized into element contribution. The mask of nonzero shape functions for individual knot spans can be evaluated using *giveKnotBasisFuncMask* service declared by *FEInterpolation* and implemented by *BSplineInterpolation*.

CHAPTER

ELEVEN

ABOUT

This manual is part of OOFEM documentation project. OOFEM is open source finite element solver which has been originally developped at Department of Mechanics of Faculty of Civil Engineering, Czech Technical University in Prague, Czech Republic.

For more information about oofem, please visit https://www.oofem.org

11.1 Legal notice

This notice must be preserved on all partial or complete copies of this manual. All modifications to this manual, translations or derivative work based on this manual must be first approved in writing by the author. If part of this manual is distributed, a notice how to obtain the full version must be included.

```
The OOFEM is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Copyright (C) 1995-2024 Bořek Patzák

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search