OOFEM Input Manual

Release 1.0

Apr 08, 2021

CONTENTS

This manual describes in details the format and structure of OOFEM text input file. Input file can be prepared in any text editor or can be generated by a conversion program or FEM pre-processor.

CHAPTER

ONE

RUNNING THE CODE

The program can be executed by typing
`oofem [option [parameter]] ...`
on the command line prompt with the following command line options:

-v	Prints oofem version.
-f path	Path to oofem input file name, if not present, program interactively reads this parameter.
-r int	Restarts the analysis from given solution step. The corresponding context file (*.osf) must exist.
-rn	Turns on the equation renumbering. Default is off.
-ar int	Restarts the adaptive computation from given solution step. Requires the corresponding context file (*.osf) and domain input file (*.din) to exists. The domain input file describes the new mesh, its syntax is identical to syntax of input file, but it does not contains the output file record, job description record and analysis record.
-l int	Sets treshold for log messages (Errors=0, Warnings=1, Relevant=2, Info=3, Debug=4).
-qo path	Redirect the standard output stream (stdout) to given file.
-qe path	Redirect standard error stream (stderr) to given file.
-c	Forces the creation of context file for each solution step.
-t int	Determines the number of threads to use (requires OpenMP support compiled)
-p	Runs in parallel mode using MPI (requires MPI support compiled)

To execute OOFEM program in parallel MPI mode (indicated by the -p flag), users must know the procedure for executing/scheduling MPI jobs on the particular system(s). For instance, when using the MPICH implementation of MPI and many others, the following command initiates a program that uses eight processors:

```
mpirun –np 8 oofem –p program_options
```

CHAPTER

TWO

SYNTAX AND GENERAL RULES

Input file is composed of records. In the current implementation, each record is represented by one line in input file. The order of records in file is compulsory, and it has following structure:

1. output file record, see section *OutputFileRecord*,
2. job description record, see section *JobDescriptionRecord*,
3. analysis record, see section *AnalysisRecord*,
4. domain record, see section *DomainRecord*,
5. output manager record, see section *OutputManagerRecord*,
6. components size record, see section *ComponentsSizeRecord*,
7. node record(s), see section *NodeElementSideRecords*,
8. element record(s), see section *ElementsRecords*,
9. set record(s), see section *SetRecords*,
10. cross section record(s), see section *CrossSectionRecords*,
11. material type record(s), see section *MaterialTypeRecords*,
12. nonlocal barriers record(s), see section *NonlocalBarrierRecords*,
13. load, boundary conditions record(s), see section *LoadBoundaryInitialConditions*,
14. initial conditions record(s), see section *InitialConditions*,
15. time functions record(s), see section *TimeFunctionsRecords*.
16. optional xfem manager and associated record(s), see section *XFEMManagerRecords*

When input line begins with '#' character, then it is ignored by the parser and can serve as a comment inside input file.

The individual records consist of record keyword followed by one or more attributes. Each attribute is identified by its keyword, which can be followed by attribute value(s). Some attributes have no values. The order of attributes in the record is optional.

Sometimes, the record keyword itself can be variable, taking on a restricted range of possible values. As an example, OOFEM has element record, desribing particulat element, and record keyword determines the particular element type. In this case, the record keyword is preceded by star. We call such record keyword as entity keyword. The possible substitutions for entity keyword are typed using `Typewriter` font family. Often, some attributes are specific to particular entity keyword. Then the general format of record is described and entity specific attributes are described separately. The possible attributes are then union of general and entity specific attributes.

```
# nodal records
Node 1 coords 3 0. 0. 0.
Node 2 coords 3 0. 0. 2. dofidmask 3 1 2 3
# element record
Truss2d 1 nodes 2 1 2 crossSect 1
```

Each attribute value has a specific type, which describe its size and layout. To describe the type of an attribute, the following notation is used: `Keyword #(type)`, where type determines the attribute type and # is the placeholder for the attribute value. The possible types of attribute values are following:

- **in** - integer number.

```
val1 25
```

- **rn** - real number.

```
val2 -0.234e-3
```

- **ch** - character (usually for description of unknown type ('d' for displacement, 't' for temperature, etc.).

```
val3 t
```

- **ia** - integer array. The format of integer array is `size val(1) ... val(size)`, where size, val(1),...,val(size) are integer numbers. Values are separated by one or more spaces. As an example, consider the integer array attribute called `nodes`$= \{1, 4, 23\}$:

```
nodes 3 1 4 23
```

- **ra** - real array. The format of real array is `size val(1) ... val(size)`, where size is integer number and val(1), ..., val(size) are real numbers. Values are separated by one or more spaces. As an example, consider the real array attribute called `coords`$= \{1.0, 2.0, 3.0\}$:

```
coords 3 1.0 2.0 3.0
```

- **rm** - real matrix, format of real matrix is
  `rows columns {val(1,1) val(1,2) ...; val(2,1) ...}`, where "rows" and "columns" are integer numbers and val(1,1), ..., are real numbers. Columns are seperated by space or comma and lines by semicolon. As an example, consider the real matrix attribute called `mat1` $= \begin{bmatrix} 1.0 & -1.0 & 0.0 \\ 2.0 & 2.5 & 5.0 \end{bmatrix}$:

```
mat1 2 3 \{1.0 -1.0; 0.0 2.0; 2.5 5.0\}
```

- **dc** - dictionary. Dictionary consist of pairs, each pair has key (character type) and its associated value (integer type). Format of dictionary is `size key(1) val(1) ... key(size) val(size)`, where size is integer number, key(1),...,key(size) are single character values, and val(1), ..., val(size) are real numbers. Values are separated by one or more spaces;

```
dict1 2 a 1.0 v 0.0
```

- **rl** - range list. Range list syntax is { number1 .. numberN (start1 end1) (start2 end2)}. The enclosing brackets are compulsory. The range list represent list of integer values. Single values can be specified using single values (number1, .., NumberN). The range of values (all numbers from startI to endI including startI and endI can be specified using range value in the form (startI endI). The range is described using its start and end values enclosed in parenthesis. Any number of ranges and single values can be used to specify range list.

```
range1 { 1 7 8 (10 20) (25 30) }
```

- **et** - entity type. For example, it describes the finite element type. Possible type values are mentioned in specific sections.
- **s** - character string. The string have to be enclosed in quotes (“”) following after corresponding keyword.

```
string1 ``string example''
```

- **expr** - function expression. The expression have to be enclosed in quotes (“”). The expression is evaluated by internal parser and represent mathematical expressions as a function of certain variables. The variable names and meaning are described in specific sections. The usual arithmetic operators like -,+,*,/ are supported and their evaluation order is taken into account. The evaluation order can be changed using parenthesis. Several built-in functions are supported (sqrt, sin, cos, tan, atan, asin and acos) - these must be typed using lowercase letters and their arguments must be enclosed in parenthesis.

```
expr1 ``2.0*sin(t)/3.0''
```

The general format of record is
`[attribute1_keyword #(type)] ... [attributeXX_keyword #(type)]`

The keywords and their values are separated by one or more spaces. Please note, that a single record cooresponds to one input line in input file.

When some attribute is enclosed in brackets [ ], then it’s use is optional and often overwrites the default behavior or adds additional (but optional) information or property (for example adds a loading to node).

Example of input record.
As an example, consider the following record description:
`Particle color #(in) mass #(rn) coords #(ra) name #(s)` The following listing shows the corresponding, properly formatted, input record:

```
Particle 2 color 5 mass 0.18 coords 3 0.0 1.0 2.0 name "P1_36"
```

CHAPTER

THREE

OUTPUT AND JOB DESCRIPTION RECORDS

3.1 Output file record

This record has no keywords and contains a character string, which describes the path to output file. If the file with the same name exists, it will be overwritten.

3.2 Job description record

This record has no keywords and contains a character string, which describes the job. This description will appear in the output file.

CHAPTER

FOUR

ANALYSIS RECORD

This record describes the type of analysis, which should be performed. The analysis record can be splitted into optional meta-step input records (see below). Then certain attributes originally in analysis record can be specified independently for each meta-step. This is marked by adding "M" superscript to keyword. Then the attribute format is `Keyword^M #(type)`.

The general format of this record can be specified using

- "standard-syntax"
  `nsteps #(in) [renumber #(in)] [profileopt #(in)] attributes #(string) [ninitmodules #(in)] [nmodules #(in)] [nxfemman #(in)]`
- "meta step-syntax"
  `nmsteps #(in) [ninitmodules #(in)] [nmodules #(in)] [nxfemman #(in)]`
  immediately followed by `nmsteps` meta step records with the following syntax:
  `nsteps #(in) attributes #(string)`
  The `nmsteps` parameter determines the number of "metasteps". The meta step represent sequence of solution steps with common attributes. There is expected to be `nmsteps` subsequent metastep records. The meaning of meta step record parameters (or analysis record parameters in "standard syntax") is following:
  - `nsteps` - determines number of subsequent solution steps within metastep.
  - `renumber` - Turns out renumbering after each time step. Necessary when Dirichlet boundary conditions change during simulation. Can also be turned out by the executeable flag `-rn`.
  - `profileopt` - Nonzero value turns on the equation renumbering to optimize the profile of characteristic matrix (uses Sloan algorithm). By default, profile optimization is not performed. It will not work in parallel mode.
  - `attributes` - contains the metastep related attributes of analysis (and solver), which are valid for corresponding solution steps within meta step. If used in standard syntax, the attributes are valid for all solution step.
  - `ninitmodules` - number of initialization module records for given problem. The initialization modules are specified after meta step section (or after analysis record, if no metasteps are present). Initialization modules allow to initialize the state variables by values previously computed by external software. The available initialization modules are described in section *InitModulesSec*.
  - `nmodules` - number of export module records for given problem. The export modules are specified after initialization modules. Export modules allow to export computed data into external software for postprocessing. The available export modules are described in section *ExportModulesSec*.
  - `nxfemman` - 1 implies that an XFEM manager is created, 0 implies that no XFEM manager is created. The XFEM manager stores a list of enrichment items. The syntax of the XFEM manager record and related records is described in section *XFEMManagerRecords*.

  - `eetype` - optional error estimator type used for the problem. Used for adaptive analysis, but can also be used to compute and write error estimates to the output files. See adaptive engineering models for details.

Not all of analysis types support the metastep syntax, and if not mentioned, the standard-syntax is expected. Currently, supported analysis types are

- Linear static analysis, see section *LinearStatic*,
- Eigen value dynamic, see section *EigenValueDynamic*,
- Direct explicit nonlinear dynamics, see section *NlDEIDynamic*,
- Direct explicit (linear) dynamics, see section DEIDynamic>_,
- Implicit linear dynamic, see section *DIIDynamic*,
- Incremental **linear** static problem, see section *IncrementalLinearStatic*,
- Non-linear static analysis, see section *NonLinearStatic*.

4.1 Structural Problems

4.1.1 StaticStructural

`StaticStructural nsteps #(in) [deltat #(...)] [prescribedtimes #(...)] [stiffmode #(...)][nonlocalext #(...)][sparselinsolverparams #(...)]`

Static structural analysis. Can be used to solve linear and nonlinear static structural problems, supporting changes in boundary conditions (applied load and supports). The problem can be solved under direct load or displacement control, indirect control, or by their arbitrary combination. Note, that the individual solution steps are used to describe the history of applied incremental loading. The load cases are not supported, for each load case the new analysis has to be performed. To analyze linear static problem with multiple load combinations, please use LinearStatic solver.

By default all material nonlinearities will be taken into account, geometrical not. To include geometrically nonlinear effect one must specify level of non-linearity in element records.

The `sparselinsolverparams` parameter describes the sparse linear solver attributes and is explained in section *sparselinsolver*. The optional parameter `deltat` defines the length of time step (equal to 1.0 by default). The times corresponding to individual solution times can be specified using optional parameter `prescribedtimes`, allowing to input array of discrete solution times, the number of solution steps is then equal to the size of this array. .

4.1.2 Linear static analysis

`LinearStatics nsteps #(in) [sparselinsolverparams #(...)] [sparselinsolverparams #(...)]`

Linear static analysis. Parameter `nsteps` indicates the number of loading cases. Problem supports multiple load cases, where number of load cases correspods to number of solution steps, individual load vectors are formed in individual time-steps. However, the static system is assumed to be the same for all load cases. For each load case an auxiliary time-step is generated with time equal to load case number.

The `sparselinsolverparams` parameter describes the sparse linear solver attributes and is explained in section *sparselinsolver*.

4.1.3 LinearStability

`LinearStability nroot #(in) rtolv #(rn) [eigensolverparams #(...)]`

Solves linear stability problem. Only first `nroot` smallest eigenvalues and corresponding eigenvectors will be computed. Relative convergence tolerance is specified using `rtolv` parameter.

The `eigensolverparams` parameter describes the sparse linear solver attributes and is explained in section *eigensolverssection*.

4.1.4 EigenValueDynamic

`EigenValueDynamic nroot #(in) rtolv #(rn) [eigensolverparams #(...)]`

Represents the eigen value dynamic analysis. Only `nroot` smallest eigenvalues and corresponding eigenvectors will be computed. Relative convergence criteria is governed using `rtolv` parameter.

The `eigensolverparams` parameter describes the sparse linear solver attributes and is explained in section *eigensolverssection*.

4.1.5 NlDEIDynamic

`NlDEIDynamic nsteps #(in) dumpcoef #(rn) [deltaT #(rn)]`

Represents the direct explicit nonlinear dynamic integration. The central difference method with diagonal mass matrix is used, damping matrix is assumed to be proportional to mass matrix, $\boldsymbol{C} = \text{dumpcoef} * \boldsymbol{M}$, where $\boldsymbol{M}$ is diagonal mass matrix. Parameter `nsteps` specifies how many time steps will be analyzed. `deltaT` is time step length used for integration, which may be reduced by program in order to satisfy solution stability conditions. Parameter `reduct` is a scaling factor (smaller than 1), which is multiplied with the determined step length adjusted by the program. If `deltaT` is reduced internally, then `nsteps` is adjusted so that the total analysis time remains the same.

The parallel version has the following additional syntax:
&⟨[nonlocalext]⟩&

4.1.6 DEIDynamic

`DEIDynamic nsteps #(in) dumpcoef #(rn) [deltaT #(rn)]`

Represent the **linear** explicit integration scheme for dynamic problem solution. The central difference method with diagonal mass matrix is used, damping matrix is assumed to be proportional to mass matrix, $\boldsymbol{C} = \text{dumpcoef} * \boldsymbol{M}$, where $\boldsymbol{M}$ is diagonal mass matrix. `deltaT` is time step length used for integration, which may be reduced by program in order to satisfy solution stability conditions. Parameter `nsteps` specifies how many time steps will be analyzed.

4.1.7 DIIDynamic

`DIIDynamic nsteps #(in) deltaT #(rn) alpha #(rn) beta #(rn) Psi #(rn)`

Represents direct implicit integration of linear dynamic problems. Damping is modeled as Rayleigh damping ($\boldsymbol{c} = \text{alpha} * \boldsymbol{M} + \text{beta} * \boldsymbol{K}$). Parameter `Psi` determines integration method used, for`Psi` = 1 the Newmark and for `Psi` $\geq$ 1.37 the Wilson method will be used. Parameter `deltaT` is required time integration step length.

4.1.8 IncrementalLinearStatic

`IncrementalLinearStatic endOfTimeOfInterest #(rn) prescribedTimes #(ra)`

Represents incremental **linear** static problem. The problem is solved as series of linear solutions and is intended to be used for solving linear creep problems or incremental perfect plasticity.

Supports the changes of static scheme (applying, removing and changing boundary conditions) during the analysis.

Response is computed in times defined by `prescribedTimes` array. These times should include times, when generally the boundary conditions are changing, and in other times of interest. (For linear creep analysis, the values should be uniformly distributed on log-time scale, if no change in loading or boundary conditions). The time at the end of interested is specified using `endOfTimeOfInterest` parameter.

4.1.9 NonLinearStatic

NonLinearStatic
Non-linear static analysis. The problem can be solved under direct load or displacement control, indirect control, or by their arbitrary combination. By default all material nonlinearities will be included, geometrical not. To include geometrically nonlinear effect one must specify level of non-linearity in element records. There are two different ways, how to specify the parameters - the extended and standard syntax.

Extended syntax

The extended syntax uses the "metastep" concept and has the following format:

`NonLinearStatic [nmsteps #(in)] nsteps #(in) [contextOutputStep #(in)] [sparselinsolverparams #(string)] [nonlinform #(in)] <[nonlocstiff #(in)]> <[nonlocalext]> <[loadbalancing]>`

This record is immediately followed by metastep records with the format described below. The analysis parameters have following meaning

- `nmsteps` - determines the number of "metasteps", default is 1.
- `nsteps` - determines number of solution steps.
- `contextOutputStep` - causes the context file to be created for every contextOutputStep-th step and when needed. Useful for postprocessing.
- The `sparselinsolverparams` parameter describes the sparse linear solver attributes and is explained in section *sparselinsolver*.
- `nonlinform` - formulation of non-linear problem. If == 1 (default), total Lagrangian formulation in undeformed original shape is used (first-order theory). If == 2, the equlibrated displacements are added to original ones and updated in each time step (second-order theory).
-

The metastep record has the following general syntax:
`nsteps #(in) [controlmode #(in)] [deltat #(rn)] [stiffmode #(in)] [refloadmode #(in)] solverParams #() [sparselinsolverparams #(string)] [donotfixload #()]`

where

- `controlmode` - determines the type of solution control used for corresponding meta step. if == 0 then indirect control will be used to control solution process (arc-length method, default). if == 1 then direct displacement or load control will be used (Newton-Raphson solver). In the later mode, one can apply the prescribed load increments as well as control displacements.
- `deltaT` - is time step length. If not specified, it is set equal to 1,0. Each solution step has associated the corresponding intrinsic time, at which the loading is generated. The `deltaT` determines the spacing between solution steps on time scale.
- `stiffMode` - If == 0 (default) then tangent stiffness will be used at new step beginning and whenever numerical method will ask for stiffness update. If == 1 the use of secant tangent will be forced. The secant stiffness will be used at new step beginning and whenever numerical method will ask for stiffness update. If == 2 then original elastic stiffness will be used during the whole solution process.
- The `refloadmode` parameter determines how the reference force load vector is obtained from given totalLoadVector and initialLoadVector. The initialLoadVector describes the part of loading which does not scale. Works only for force loading, other non-force components (temperature, prescribed displacements should always given in total values). If `refloadmode` is 0 (rlm_total, default) then the reference incremental load vector is defined as totalLoadVector assembled at given time. If `refloadmode` is 1 (rlm_inceremental) then the reference load vector is obtained as incremental load vector at given time.
- `solverParams` - parameters of solver. The solver type is determined using `controlmode`.
- The `sparselinsolverparams` parameter describes the sparse linear solver attributes and is explained in section *sparselinsolver*.
- By default, reached load at the end of metastep will be maintained in subsequent steps as fixed, non scaling load and load level will be reset to zero. This can be changed using keyword `donotfixload`, which if present, causes the loading to continue, not resetting the load level. For the indirect control the reached loading will not be fixed, however, the new reference loading vector will be assembled for the new metastep.

The direct control corresponds to `controlmode`=1 and the Newton-Raphson solver is used. Under the direct control, the total load vector assembled for specific solution step represents the load level, where equilibrium is searched. The implementation supports also displacement control - it is possible to prescribe one or more displacements by applying "quasi prescribed" boundary condition(s)[1] The load level then represents the time, where the equilibrium has been found. The Newton-Raphson solver parameters (`solverParams`) for load-control are:
`maxiter #(in) [minsteplength #(in)] [minIter #(in)] [manrmsteps #(in)] [ddm #(ia)] [ddv #(ra)] [ddltf #(in)] [linesearch #(in)] [lsearchamp #(rn)] [lsearchmaxeta #(rn)] [lsearchtol #(rn)] [nccdg #(in) ccdg1 #(ia) ccdgN #(ia) ] rtolv #(rn) [rtolf #(rn)] [rtold #(tn)] [initialGuess #(rn)]` where

- `maxiter` determines the maximum number of iterations allowed to reach equilibrium. If equilibrium is not reached, the step length (corresponding to time) is reduced.
- `minsteplength` parameter is the minimum step length allowed.
- `minIter` - minimum number of iterations which always proceed during the iterative solution.
- If `manrmsteps` parameter is nonzero, then the modified N-R scheme is used, with the stiffness updated after `manrmsteps` steps.
- `ddm` is array specifying the degrees of freedom, which displacements are controlled. Let the number of these DOFs is N. The format of `ddm` array is 2*N dofman1 idof1 dofman2 idof2 ... dofmanN idofN, where the dofmani is the number of i-th dof manager and idofi is the corresponding DOF number.

[1] Hovewer, the problem does not support the changes of static system. But it is possible to apply direct displacement control without requiring BC applied (see nrsolver documentation). Therefore it is possible to combine direct displacement control with direct load control or indirect control.

- ddv is array of relative weights of controlled displacements, the size should be equal to N. The actual value of prescribed dofs is defined as a product of its weight and the value of load time function specified using ddltf parameter (see below).
- ddltf number of load time function, which is used to evaluate the actual displacements of controlled dofs.
- linesearch nonzero value turns on line search algorithm. The lsearchtol defines tolerance (default value is 0.8), amplification factor can be specified using lsearchamp parameter (should be in interval $(1, 10)$), and parameter lsearchmaxeta defines maximum limit on the length of iterative step (allowed range is $(1.5, 15)$).
- nccdg allows to define one or more DOF groups, that are used for evaluation of convergence criteria. Each DOF is checked if it is a member of particular group and in this case its contribution is taken into account when evaluating the convergence criteria for that group. By default, if nccdg is not specified, one group containing all DOF types is created. The value of nccdg parameter defines the number of DOF type groups. For each group, the corresponding DOF types need to be specified using ccdg# parameter, where '#' should be replaced by group number (numbering starts from 1). This array contains the DofIDItem values, that identify the physical meaning of DOFs in the group. The values and their physical meaning is defined by DofIDItem enum type (see src/oofemlib/dofiditem.h for reference).
- rtolv determines relative convergence norm (both for displacement iterative change vector and for residual unbalanced force vector). Optionally, the rtolf and rtold parameters can be used to define independent relative convergence crteria for unbalanced forces and displacement iterative change. If the default convergence criteria is used, the parameters rtolv,rtolf, and rtold are real values. If the convergence criteria DOF groups are used (see bellow the description of nccdg parameter) then they should be specified as real valued arrays of nccdg size, and individual values define relative convergence criteria for each individual dof group.
- initialGuess is an optional parameter with default vaue 0, for which the first iteration of each step starts from the previously converged state and applies the prescribed displacement increments. This can lead to very high strains in elements connected to the nodes with changing prescribed displacements and the state can be far from equilibrium, which may results into slow convergence and strain localization near the boundary. If initialGuess is set to 1, the contribution of the prescribed displacement increments to the internal nodal forces is linearized and moved to the right-hand side, which often results into an initial solution closer to equilibrium. For instance, if the step is actually elastic, equilibrium is fully restored after the second iteration, while the default method may require more iterations.

The indirect solver corresponds to controlmode=0 and the CALM solver is used. The value of reference load vector is determined by refloadmode parameter mentioned above at the first step of each metastep. However, the user must ensure that the same value of reference load vector could be obtained for all solution steps of particular metastep (this is necessary for restart and adaptivity to work). The corresponding meta step solver parameters (solverParams) are:

Psi #(rn) MaxIter #(in) stepLength #(rn) [minStepLength #(in)] [initialStepLength #(rn)] [forcedInitialStepLength #(rn)] [reqIterations #(in)] [maxrestarts #(in)] [minIter #(in)] [manrmsteps #(in)] [hpcmode #(in)] [hpc #(ia)] [hpcw #(ra)] [linesearch #(in)] [lsearchamp #(rn)] [lsearchmaxeta #(rn)] [lsearchtol #(rn)] [nccdg #(in) ccdg1 #(ia) ... ccdgN #(ia)] rtolv #(rn) [rtolf #(rn)] [rtold #(rn)] [pert #(ia)] [pertw #(ra)] [rpa #(rn)] [rseed #(in)] where

- Psi - CALM Ψ control parameter. For $\Psi = 0$ displacement control is applied. For nonzero values the load control applies together with displacement control (ALM). For large Ψ load control apply.
- MaxIter - determines the maximum number of iteration allowed to reach equilibrium state. If this limit is reached, restart follows with smaller step length.
- stepLength - determines the maximum value of arc-length (step length).

- `minStepLength` - minimum step length. The step length will never be smaller. If convergence problems are encountered and step length cannot be decreased, computation terminates.
- `initialsteplength` - determines the initial step length (the arc-length). If not provided, the maximum step length (determined by `stepLength` parameter) will be used as the value of initial step length.
- `forcedInitialStepLength` - When simulation is restarted, the last predicted step length is used. Use `forcedInitialStepLength` parameter to override the value of step length. This parameter will also override the value of initial step length set by `initialsteplength` parameter.
- `reqIterations` - approximate number of iterations controlled by changing the step length.
- `maxrestarts` - maximum number of restarting computation when convergence not reached up to `MaxIter`.
- `minIter` - minimum number of iterations which always proceed during the iterative solution. `reqIterations` are set to be the same, `MaxIter` are increased if lower.
- `manrmsteps` - Forces the use of accelerated Newton Raphson method, where stiffness is updated after `manrmsteps` steps. By default, the modified NR method is used (no stiffness update).
- `hpcmode` Parameter determining the alm mode. Possible values are: 0 - (default) full ALM with quadratic constrain and all dofs, 1 - (default, if `hpc` parameter used) full ALM with quadratic constrain, taking into account only selected dofs (see `hpc` param), 2 - linearized constrain in displacements only, taking into account only selected dofs with given weight (see `hpc` and `hpcw` parameters).
- `hpc` - Special parameter for Hyper-plane control, when only selected DOFs are taken account in ALM step length condition. Important mainly for material nonlinear problems with strong localization. This array selects the degrees of freedom, which displacements are controlled. Let the number of these DOFs be N. The format of `ddm` array is 2*N dofman1 idof1 dofman2 idof2 ... dofmanN idofN, where the dofmani is the number of i-th dof manager and idofi is the corresponding DOF number.
- `hpcw` Array of DOF weights in linear constraint. The dof ordering is determined by `hpc` parameter, the size of the array should be N.
- `linesearch` nonzero value turns on line search algorithm. The `lsearchtol` defines tolerance, amplification factor can be specified using `lsearchamp` parameter (should be in interval $(1, 10)$), and parameter `lsearchmaxeta` defines maximum limit on the length of iterative step (allowed range is $(1.5, 15)$).
- `nccdg` allows to define one or more DOF groups, that are used for evaluation of convergence criteria. Each DOF is checked if it is a member of particular group and in this case its contribution is taken into account when evaluating the convergence criteria for that group. By default, if `nccdg` is not specified, one group containing all DOF types is created. The value of `nccdg` parameter defines the number of DOF type groups. For each group, the corresponding DOF types need to be specified using `ccdg#` parameter, where '#' should be replaced by group number (numbering starts from 1). This array contains the DofIDItem values, that identify the physical meaning of DOFs in the group. The values and their physical meaning is defined by DofIDItem enum type (see src/oofemlib/dofiditem.h for reference).
- `rtolv` determines relative convergence norm (both for displacement iterative change vector and for residual unbalanced force vector). Optionally, the `rtolf` and `rtold` parameters can be used to define independent relative convergence crteria for unbalanced forces and displacement iterative change. If the default convergence criteria is used, the parameters `rtolv`,`rtolf`, and `rtold` are real values. If the convergence criteria DOF groups are used (see bellow the description of `nccdg` parameter) then they should be specified as real valued arrays of `nccdg` size, and individual values define relative convergence criteria for each individual dof group.
- `pert` Array specifying DOFs that should be perturbed after the first iteration of each step. Let the number of these DOFs be M. The format of `ddm` array is 2*M dofman1 idof1 dofman2 idof2 ... dofmanN idofN, where the dofmani is the number of i-th dof manager and idofi is the corresponding DOF number.
- `pertw` Array of DOF perturbations. The dof ordering is determined by `pert` parameter, the size of the array should be M.

- `rpa` Amplitude of random perturbation that is applied to each DOF.
- `rseed` Seed for the random generator that generates random perturbations.

Standard syntax

In this case, all parameters (for analysis as well as for the solver) are supplied in analysis record. The default meta step is created for all solution steps required. Then the meta step attributes are specified within analysis record. The format of analysis record is then following

`NonLinearStatic nsteps #(in) [nonlocstiff #(in)] [contextOutputStep #(in)] [controlmode #(in)] [deltat #(rn)'] rtolv #(rn) [stiffmode #(in)] lstype #(in) smtype #(in) solverParams #() [nonlinform #(in)] <[nonlocstiff #(in)]> <[nonlocalext]> <[loadbalancing]`

The meaning of parameters is the same as for extended syntax.

Parameter `lstype` allows to select the solver for the linear system of equations. Parameter `smtype` allows to select the sparse matrix storage scheme. The scheme should be compatible with the solver type. See section *sparselinsolver* for further details.

4.1.10 Adaptive linear static

`Adaptlinearstatic nsteps #(in) [sparselinsolverparams #(...)] [meshpackage #(in)] errorestimatorparams #(...)`

Adaptive linear static analysis. Multiple loading cases are not supported. Due to linearity of a problem, the complete reanalysis from the beginning is done after adaptive remeshing. After first step the error is estimated, information about required density is generated (using mesher interface) and solution terminates. If the error criteria is not satisfied, then the new mesh and corresponding input file is generated and new analysis should be performed until the error is acceptable. Currently, the available error estimator for linear problems is Zienkiewicz-Zhu. Please note, that adaptive framework requires specific functionality provided by elements and material models. For details, see element and material model manuals.

- Parameter `nsteps` indicates the number of loading cases. Should be set to 1.
- The `sparselinsolverparams` parameter describes the sparse linear solver attributes and is explained in section *sparselinsolver*.
- The `meshpackage` parameter selects the mesh package interface, which is used to generate information about required mesh density for new remeshing. The supported interfaces are explained in section *meshpackages*. By default, the T3d interface is used.
- The `errorerestimatorparams` parameter contains the parameters of Zienkiewicz Zhu Error Estimator. These are described in section *errorestimators*.

4.1.11 Adaptive nonlinear static

`Adaptnlinearstatic Nonlinearstaticparams #() [equilmc #(in)] [meshpackage #(in)] [eetype #(in)] errorestimatorparams #(...)`

Represents Adaptive Non-LinearStatic problem. Solution is performed as a series of increments (loading or displacement). The error is estimated at the end of each load increment (after equilibrium is reached), and based on reached error, the computation continues, or the new mesh densities are generated and solution stops. Then the new discretization should be generated. The truly adaptive approach is supported, so the computation can be restarted from the last

step (see section *running-the-code*), solution is mapped to new mesh (separate solution step) and new load increment is applied. Of course, one can start the analysis from the very beginning using new mesh. Currently, the available estimators/indicators include only linear Zienkiewicz-Zhu estimator and scalar error indicator. Please note, that adaptive framework requires specific functionality provided by elements and material models. For details, see element and material model manuals.

- Set of parameters `Nonlinearstaticparams` are related to nonlinear analysis. They are described in section *NonLinearStatic*.
- Parameter `equilmc` determines, whether after mapping of primary and internal variables to new mesh the equilibrium is restored or not before new load increment is applied. The possible values are: 0 (default), when no equilibrium is restored, and 1 forcing the equilibrium to be restored before applying new step.
- The `meshpackage` parameter selects the mesh package interface, which is used to generate information about required mesh density for new remeshing. The supported interfaces are explained in section *meshpackages*. By default, the T3d interface is used.
- Parameter `eetype` determines the type of error estimator/indicator to be used. The parameters `errorestimatorparams` represent set of parameters corresponding to selected error estimator. For description, follow to section *errorestimators*.

4.1.12 Free warping analysis

`FreeWarping nsteps #(in)`

Free warping analysis computes the deplanation function of cross section with arbitrary shape. It is done by solving the Laplace's equation with automatically generated boundary conditions corresponding to the free warping problem.

This type of analysis supports only `TrWarp` elements and `WarpingCS` cross sections. One external node must be defined for each warping cross section. The coordinates of this node can be arbitrary but this node must be defined with parametr `DofIDMask 1 24` and one boundary condition which represents relative twist acting on corresponding warping cross section. No additional loads make sence in free warping analysis.

Parameter `nsteps` indicates the number of loading cases. Series of loading cases is maintained as sequence of time-steps. For each load case an auxiliary time-step is generated with time equal to load case number. Load vectors for each load case are formed as load vectors at this auxiliary time.

4.2 Transport Problems

4.2.1 Stationary transport problem

`StationaryProblem nsteps #(in) [sparselinsolverparams #(...)][exportfields #(ia)]`

Stationary transport problem. Series of loading cases is maintained as sequence of time-steps. For each load case an auxiliary time-step is generated with time equal to load case number. Load vectors for each load case are formed as load vectors at this auxiliary time. The `sparselinsolverparams` parameter describes the sparse linear solver attributes and is explained in section *sparselinsolver*.

If the present problem is used within the context of staggered-like analysis, the temperature field obtained by the solution can be exported and made available to any subsequent analyses. For example, temperature field obtained by present analysis can be taken into account in subsequent mechanical analysis. To allow this, the temperature must be "exported". This can be done by adding array `exportfields`. This array contains the field identifiers, which tell the problem to register its primary unknowns under given identifiers. See file field.h. Then the subsequent analyses can get access to exported fields and take them into account, if they support such feature.

4.2.2 Transient transport problem

`TransientTransport nsteps #(in) deltaT #(rn)` *or* `dTfunction #(in)` *or* `prescribedtimes #(ra) alpha #(rn) [initT #(rn)] [lumped] [keeptangent] [exportfields #(ia)]`

Nonlinear implicit integration scheme for transient transport problems. The generalized midpoint rule (sometimes called α-method) is used for time discretization, with alpha parameter, which has limits $0 \leq \alpha \leq 1$. For $\alpha = 0$ explicit Euler forward method is obtained, for $\alpha = 0.5$ implicit trapezoidal rule is recovered, which is unconditionally stable, second-order accurate in Δt, and $\alpha = 1.0$ yields implicit Euler backward method, which is unconditionally stable, and first-order accurate in Δt. `deltaT` is time step length used for integration, `nsteps` parameter specifies number of time steps to be solved. It is possible to define `dTfunction` with a number referring to corresponding time function, see section *TimeFunctionsRecords*. Variable time step is advantageous when calculating large time intervals.

The `initT` sets the initial time for integration, 0. by default. If `lumped` is set, then the stabilization of numerical algorithm using lumped capacity matrix will be used, reducing the initial oscillations. See section *StationaryTransport* for an explanation on `exportfields`.

This transport problem supports sets and changes in number of equations. It is possible to impose/remove Dirichlet boundary conditions during solution.

4.2.3 Transient transport problem - linear case - obsolete

`NonStationaryProblem nsteps #(in) deltaT #(rn) | deltaTfunction #(in) alpha #(rn) [initT #(rn)] [lumpedcapa] [sparselinsolverparams #(..)] [exportfields #(ia)] [changingProblemSize]`

Linear implicit integration scheme for transient transport problems. The generalized midpoint rule (sometimes called α-method) is used for time discretization, with alpha parameter, which has limits $0 \leq \alpha \leq 1$. For $\alpha = 0$ explicit Euler forward method is obtained, for $\alpha = 0.5$ implicit trapezoidal rule is recovered, which is unconditionally stable, second-order accurate in Δt, and $\alpha = 1.0$ yields implicit Euler backward method, which is unconditionally stable, and first-order accurate in Δt. `deltaT` is time step length used for integration, `nsteps` parameter specifies number of time steps to be solved. It is possible to define `deltaTfunction` with a number referring to corresponding time function, see section *TimeFunctionsRecords*. Variable time step is advantageous when calculating large time intervals. It is strongly suggested to use nonlinear transport solver due to stability reasons, see section *TransientTransport*.

The `initT` sets the initial time for integration, 0 by default. If `lumpedcapa` is set, then the stabilization of numerical algorithm using lumped capacity matrix will be used, reducing the initial oscillations. See section *StationaryTransport* for an explanation on `exportfields`.

This linear transport problem supports changes in number of equations. It is possible to impose/remove Dirichlet boundary conditions during solution. This feature is enabled with `changingProblemSize`, which ensures storing solution values on nodes (DoFs) directly. If the problem does not grow/decrease during solution, it is more efficient to use conventional solution strategy and the parameter should not be mentioned.

Note: This problem type **requires transport module** and it can be used only when this module is included in your oofem configuration.

4.2.4 Transient transport problem - nonlinear case - obsolete

`NlTransientTransportProblem nsteps #(in) deltaT #(rn) | deltaTfunction #(in) alpha #(rn) [initT #(rn)] [lumpedcapa #()] [nsmax #(in)] rtol #(rn) [manrmsteps #(in)] [sparselinsolverparams #(...)] [exportfields #(ia)] [changingProblemSize]`

Implicit integration scheme for transient transport problems. The generalized midpoint rule (sometimes called α-method) is used for time discretization, with alpha parameter, which has limits $0 \leq \alpha \leq 1$. For $\alpha = 0$ explicit

Euler forward method is obtained, for $\alpha = 0.5$ implicit trapezoidal rule is recovered, which is unconditionally stable, second-order accurate in Δt, and $\alpha = 1.0$ yields implicit Euler backward method, which is unconditionally stable, and first-order accurate in Δt. See matlibmanual.pdf for solution algorithm.

`deltaT` is time step length used for integration, `nsteps` parameter specifies number of time steps to be solved. For `deltaTfunction` and `initT` see section *LinearTransientTransport*. Parameter `maxiter` determines the maximum number of iterations allowed to reach equilibrium (default is 30). Norms of residual physical quantity (heat, mass) described by solution vector and the change of solution vector are determined in each iteration. The convergence is reached, when the norms are less than the value given by `rtol`. If `manrmsteps` parameter is nonzero, then the modified N-R scheme is used, with the left-hand side matrix updated after `manrmsteps` steps. `nsmax` maximum number of iterations per time step, default is 30. If `lumpedcapa` is set, then the stabilization of numerical algorithm using lumped capacity matrix will be used, reducing the initial oscillations.

See the Section *StationaryTransport* for an explanation on `exportfields`. The meaning of `changingProblemSize` is given in Section *LinearTransientTransport*.

Note: This problem type **requires transport module** and it can be used only when this module is included in your oofem configuration.

4.3 Fluid Dynamic Problems

4.3.1 Transient incompressible flow - CBS Algorithm

`CBS nsteps #(in) deltaT #() [theta1 #(in)] [theta2 #(in)] [cmflag #(in)] [scaleflag #(in) lscale #(in) uscale #(in) dscale #(in)][lstype #(in)][smtype #(in)]`

Solves the transient incompressible flow using algorithm based on Characteristics Based Split (CBS, for reference see O.C.Zienkiewics and R.L.Taylor: The Finite Element Method, 3rd volume, Butterworth-Heinemann, 2000). At present, only semi-implicit form of the algorithm is available and energy equation, yielding the temperature field, is not solved. Parameter `nsteps` determines number of solution steps. Parameter `deltaT` is time step length used for integration. This time step will be automatically adjusted to satisfy integration stability limits $\Delta t \leq \frac{h}{|\boldsymbol{u}|}$ and $\Delta t \leq \frac{h^2}{2\nu}$, if necessary. Parameters `theta1` and `theta2` are integration constants, $\theta_1, \theta_2 \in \langle \frac{1}{2}, 1 \rangle$. If `cmflag` is given a nonzero value, then consistent mass matrix will be used instead of (default) lumped one.

The characteristic equations can be solved in non-dimensional form. To enable this, the `scaleflag` should have a nonzero value, and the following parameters should be provided: `lscale`, `uscale`, and `dscale` representing typical length, velocity, and density scales.

Parameter `lstype` allows to select the solver for the linear system of equations. Parameter `smtype` allows to select the sparse matrix storage scheme. The scheme should be compatible with the solver type. See section *sparselinsolver* for further details.

4.3.2 Transient incompressible flow SUPG/PSPG Algorithm

`SUPG nsteps #(in) deltaT #(rn) rtolv #(rn) [atolv #(rn)] [stopmaxiter #(in)] [alpha #(rn)] [cmflag #(in)] [deltatltf #(in)] [miflag #(in)] [scaleflag #(in) lscale #(in) uscale #(in) dscale #(in)][lstype #(in)][smtype #(in)]`

Solves the transient incompressible flow using stabilized formulation based on SUPG and PSPG stabilization terms. The stabilization provides stability and accuracy in the solution of advection-dominated problems and permits usage of equal-order interpolation functions for velocity and pressure. Furthermore, stabilized formulation significantly improves convergence rate in iterative solution of large nonlinear systems of equations.

By changing the value α, different methods from "Generalized mid-point family" can be chosen, i.e., Forward Euler ($\alpha = 0$), Midpoint rule ($\alpha = 0.5$), Galerkin ($\alpha = 2/3$), and Backward Euler ($\alpha = 1$). Except the first one, all the

methods are implicit and require matrix inversion for solution. Some results form an energy method analysis suggest unconditional stability for $\alpha \geq 0.5$ for the generalized mid-point family. As far as accuracy is concerned, the midpoint rule is to be generally preferred.

Parameter `nsteps` determines number of solution steps. Parameter `deltaT` is time step length used for integration. Alternatively, the load time function can be used to determine time step length for particular solution step. The load time function number is determined by parameter `deltatltf` and its value evaluated for solution step number should yield the step length.

Parameters `rtolv` and `atolv` allow to specify relative and absolute errors norms for residual vector. The equilibrium iteration process will stopped when both error limits are satisfied or when the number of iteration exceeds the value given by parameter `stopmaxiter`.

If `cmflag` is given a nonzero value, then consistent mass matrix will be used instead of (default) lumped one.

The algorithm allows to solve the flow of two immiscible fluids in fixed spatial domain (currently only in 2d). This can be also used for solving free surface problems, where one of the fluids should represent air. To enable multi-fluid analysis, user should set parameter `miflag`. The supported values are described in section *materialinterfaces*. Please note, that the initial distribution of reference fluid volume should be provided as well as constitutive models for both fluids.

The characteristic equations can be solved in non-dimensional form. To enable this, the `scaleflag` should have a nonzero value, and the following parameters should be provided: `lscale`, `uscale`, and `dscale` representing typical length, velocity, and density scales.

Parameter `lstype` allows to select the solver for the linear system of equations. Parameter `smtype` allows to select the sparse matrix storage scheme. Please note that the present algorithm leads to a non-symmetric matrix. The scheme should be compatible with the solver type. See section *sparselinsolver* for further details.

4.3.3 Transient incompressible flow (PFEM Algorithm)

`PFEM nsteps #(in) deltaT #(rn) material #(in) cs #(in) pressure #(in) [mindeltat #(rn)] [maxiter #(in)] [rtolv #(rn)] [rtolp #(rn)] [alphashapecoef #(rn)] [removalratio #(rn)][scheme #(in)][lstype #(in)][smtype #(in)]`

Solves the transient incompressible flow using particle finite element method based on the Lagrangian formulation of Navier-Stokes equations.

Mesh nodes are represented by PFEMParticles (see *pfemparticles*), which can freely move and even separate from the main domain. To integrate governing equations in each solution step, a temporary mesh, built from particles, is needed. The mesh is rebuilt from scratch in each solution step to prevent large distortion of elements. Paramters `cs` and `material` assign types from cross section and material record to created elements. Thus, the problem is defined without any elements in the input file.

Mesh is generated using Delaunay triangulation and Alpha shape technique for the identification of the free surface. The parameter `alphashapecoef` should reflect initial distribution of PFEMParticles. Value approximately equal to 1,5-multiple of shortest distance of two neighboring particles has been found well. On the free surface the zero-pressure boundary condition is enforced. This must be defined in boundary condition record under the number defined by `pressure`.

Parameter `scheme` controls whether the equation system for the components of the auxiliary velocity is solved explicitly (0) or implicitly (1). The last is the default option.

Parameter `nsteps` determines number of solution steps. Parameter `deltaT` is time step length used for integration. To ensure numerical stability, step length is adapted upon mesh geometry and velocity of paricular nodes. To avoid to short time length a minimal size can be defined by `mindeltat`. Alternatively prescribing limit `removalratio` of the element edge length too close particles can be removed from solution.

Optional parameters `rtolv` and `rtolp` allow to specify relative norms for velocity and pressure difference of two subsequent iteration step. Default values are 1.e-8. By default maximal 50 iterations are performed, if not specified by `maxiter`.

Parameter `lstype` allows to select the solver for the linear system of equations. Parameter `smtype` allows to select the sparse matrix storage scheme. Please note that the present algorithm leads to a non-symmetric matrix. The scheme should be compatible with the solver type. See section *sparselinsolver* for further details.

4.4 Coupled Problems

4.4.1 Staggered Problem

`StaggeredProblem (nsteps #(in) deltaT #(rn)) | timeDefinedByProb #(in) prob1 #(s) prob2 #(s) [stepMultiplier #(rn)]`

Represent so-called staggered analysis. This can be described as an sequence of sub-problems, where the result of some sub-problem in the sequence can depend on results of previous sub-problems in sequence. Typical example is heat transfer analysis followed by mechanical analysis taking into account the temperature field generated by the heat transfer analysis. Similar analysis can be done when coupling moisture transport with concrete drying strain.

The actual implementation supports only sequence of two sub-problems. The sub-problems are described using sub-problem input files. The syntax of sub-problem input file is the same as for standalone problem. The only addition is that sub-problems should export their solution fields so that they became available for subsequent sub-problems. See the Section *StationaryTransport*.

The subproblem input files are described using `prob1` and `prob2` parameters, which are strings containing a path to sub-problem input files, the `prob1` contains input file path of the first sub-problem, which runs first for each solution step, the `prob2` contains input file path of the second sub-problem.

There are two options how to control a time step sequence. The first approach uses `timeDefinedByProb` which uses time sequence from the corresponding subproblem. The subproblem may specify arbitrary loading steps and allows high flexibility. The second approach uses the staggered problem to take control over time. Therefore any sub-problem time-stepping parameters are ignored (even if they are required by sub-problem input syntax) and only staggered-problem parameters are relevant. `deltaT` is than a time step length used for integration, `nsteps` parameter specifies number of time steps to be solved. `stepMultiplier` multiplies all times with a given constant. Default is 1.

Note: This problem type **is included in transport module** and it can be used only when this module is configured.
Note: All material models derived from StructuralMaterial base will take into account the external registered temperature field, if provided.

4.4.2 FluidStructure Problem

`FluidStructureProblem nsteps #(in) deltaT #(rn) prob1 #(s) prob2 #(s) [maxiter #(in)][rtolv #(rn)][rtolp #(rn)]`

Represents a fluid-structure analysis based on StaggeredProblem but providing iterative synchronization of sub-problems. The implementation uses the the PFEM model *pfemIncomp* for the fluid part. For the structural part a full dynamic analysis using implicit direct integration DIIDynamic *DIIDynamic* is considered.

The coupling of both phases is based on the idea of enforcing compatibility on the interface. Special fluid particle are attached to every structural node on the interface that can be hit by the fluid. These special particles have no degrees of freedom associated, so no equations are solved on them. However, their movement is fully determined by associated structural nodes. Their velocities governed by the solid part affect the fluid equation naturally.

This iterative procedure is based on the so-called Dirichlet-Neumann approach. Dirichlet boundary conditions are the prescribed velocities on the fluid side of the interface, whereas applied forces on the structural side represent the Neumann boundary conditions.

The convergence criterion is based on the difference of the pressure and velocity values on the interface from the subsequent iterative steps. Once they are smaller than prescribed tolerance, the iteration is terminated and solution can proceed to the next step.

The subproblem input files are described using `prob1` and `prob2` parameters, which are strings containing a path to sub-problem input files, the `prob1` contains input file path of the first sub-problem, which runs first for each solution step, the `prob2` contains input file path of the second sub-problem. The time step sequence is controlled by the number of steps `nsteps` and the time step length `deltaT`.

Optional parameters `rtolv` and `rtolp` allow to specify relative norms for velocity and pressure differnce of two subsequent iteration step. Default values are 1.e-3. By default maximal 50 iterations are performed, if not specified by `maxiter`.

Note: This problem type **is included in PFEM module** and it can be used only when this module is configured.

CHAPTER

FIVE

DOMAIN RECORD(S)

This set of records describes the whole domain and its type. Depending on the type of problem, there may be one or several domain records. If not indicated, one domain record is default for all problem types.

The domain type is used to resolve the default number of DOFs in node and their physical meaning. Format is following

`domain domainType`

The `domainType` can be one from the following

- The `2dPlaneStress` and `2d-Truss` modes declare two default dofs per node (u-displacement, v-displacement),
- The `3d` mode declares three default dofs per node (u-displacement, v-displacement, w-displacement),
- The `2dMindlinPlate` mode declares three default dofs per node (w-displacent, u-rotation, v-rotation). Strain vector contains κ_{xx}, κ_{yy}, κ_{xy}, γ_{xz}, γ_{yz}. Stress vector contains m_{xx}, m_{yy}, m_{xy}, q_{xz}, q_{yz}.
- The `3dShell` mode declares six default dofs per node (displacement and rotation along each axis).
- The `2dBeam` mode declares three default dofs per node (u-displacement, w-displacement, v-rotation).
- The `2dIncompFlow` mode declares three default dofs per node (u-velocity, v-velocity, and pressure). The default number of dofs per node as well as their physical meaning can be overloaded in particular dof manager record (see section *NodeElementSideRecords*).

  The further records describe particular domain components - OutputManagers, DofManagers, Elements, Cross-Section models, Material Models, Boundary and Initial Conditions and Load time functions.

5.1 Output manager record

The output manager controls output. It can filter output to specific solution steps, and within these selected steps allows also to filter output only to specific dof managers and elements. The format of output manager record is

`[tstep_all] [tstep_step #(in)] [tsteps_out #(rl)] [dofman_all] [dofman_output #(rl)] [dofman_except #(rl)] [element_all] [element_output #(rl)] [element_except #(rl)]`

To select all solution steps, in which output will be performed, use `tstep_all`. To select each `tstep_step`-nth step, use `tstep_step` parameter. In order to select only specific solution steps, the `tsteps_out`list can be specified, supplying solution step number list in which output will be done. The combination of `tstep_step` and `tsteps_out` parameters is allowed.

Output manager allows also to filter output to only specific dof managers and elements. If these specific members are selected, the output happens only in selected solution steps. The `dofman_all` and `element_all` parameters select all dof managers or elements respectively. Parameter arrays `dofman_output` and `element_output` allow to select only specific members. Numbers of selected members are then contained in `dofman_output` or `element_output` lists respectively. The previously selected members can be explicitly de-selected by specifying their component numbers in `dofman_except` or `element_except` lists. A few examples:

`dofman_output {1 3}` prints nodes 1,3

`dofman_output {(1 3)}` prints nodes 1,2,3

`element_output {1 3}` prints elements 1,3

`element_output {(1 3)}` prints elements 1,2,3

`element_output {(1 3) 5 6}` prints elements 1,2,3,5,6

5.2 Components size record

This record describes the number of components in related domain. The particular records will follow immediately in input file. The general format is:

`ndofman #(in) nelem #(in) ncrosssect #(in) nmat #(in) nbc #(in) nic #(in) nltf #(in)` [`nbarrier #(in)`] where `ndofman` represents number of dof managers (e.g. nodes) and their associated records, `nelem` represents number of elements and their associated records, `ncrosssect` is number of cross sections and their records, `nmatdnMat` is number of material models and their records, `nbc` represents number of boundary conditions (including loads) and their records, `nic` parameter determines the number of initial conditions, and `nltf` represents number of time functions and their associated records. The optional parameter `nbarrier` represents the number of nonlocal barriers and their records. If not specified, no barriers are assumed.

5.3 Dof manager records

These records describe individual DofManager records (i.e. nodes or element sides (if they manage some DOFs)). The general format is following:

`DofManagerType #(in)` [`load #(ra)`] [`DofIDMask #(ia)`] [`bc #(ia)`] [`ic #(ia)`] [`doftype #(ia) masterMask #(ia)`] <[`shared`]> | <[`remote`]> | <[`null`]> <[`partitions #(ia)`]>

The order of particular records is optional, the dof manager number is determined by (`#(in)` parameter. The numbering of individual dof managers is arbitrary, it could be even non-continuous. In this context, one could think of dof manager number as a label that is assigned to individual dof manager and by which the dof manager is referenced.

By default, the nodal DOFs are determined by asking all the connected elements. Specifying additional dofs can be done using the using the `DofIDMask` array which determines their physical interpretation. Each item of `DofIDMask` array describes the physical meaning of corresponding DOF in dof manager. Currently the following values are supported: {u-displacement=1, v-displacement=2, w-displacement=3, u-rotation=4, v-rotation=5, w-rotation=6, u-velocity=7, v-velocity=8, w-velocity=9, temperature=10, pressure=11, special dofs for gradient-type constitutive models=12 and 13, mass concentration=14, special dofs for extended finite elements (XFEM)=15–30}. **It is not allowed to have two DOFs with the same physical meaning in the same DofManager.**

The applied primary (Dirichlet) boundary conditions are specified using "bc" record, while natural boundary conditions using "load" parameter.

- The size of "bc" array (primary bc) should be equal to number of DOFs in dof manager and i-th value relates to i-th DOF - the ordering and physical meaning of DOFs is determined by domain record and can be optionally specified for each dof manager individually (see next paragraph). The values of this array are corresponding boundary condition record numbers or zero, if no primary bc is applied to corresponding DOF. The compatible boundary condition type are required: primary conditions require "BoundaryCondition" records.

- The load "array" contains record numbers of natural boundary conditions that are applied. The required record type for natural condition is "NodalLoad". The actual value is the summation of all contributions, if more than one natural bc is applied. See section on boundary conditions for the syntax. Please note, that the values of natural bc for individual DOFs are specified in its record, not in dofmanager record.

By default, if "bc" and/or "load" parameters are omitted, no primary and/or natural bc are applied. Analogously, initial conditions are represented using `ic` array. The size of `ic` array should be equal to number of DOFs in dof manager. The values of this array are corresponding initial condition record numbers or zero, if no initial condition is applied to corresponding DOF (in this case zero value is assumed as value of initial condition).

Parameters `dofType` and `masterMask` allows to connect some dof manager's dofs (so-called "slave" dofs) to corresponding dof (according to their physical meaning) of another dof manager (so-called "master" dof). The master slave principle allows for example simple modeling of structure hinges, where multiple elements are connected by introducing multiple nodes (with same coordinates) sharing the same displacement dofs and each one possessing their own rotational dofs. Parameter `dofType` determines the type of (slave) dof to create. Currently supported values are 0 for master DOF, 1 for simpleSlave DOF (linked to another single master DOF), and 2 for general slave dof, that can depend on different DOFs belonging to different dof managers. If `dofType` is not specified, then by default all DOFs are created as master DOFs. If provided, masterMask is also required. The meaning of `masterMask` parameter is depending on type of particular dofManager, and will be described in corresponding sections.

Supported DofManagerType keywords are

- Node record

  `Node coords #(ra) [lcs #(ra)]`
  Represent an abstraction for finite element node. The node coordinates in space (given by global coordinate system) are described using `coords` attribute. This array contains x, y and possibly z (depends on problem under consideration) coordinate of node. By default, the coordinate system in node is global coordinate system. User defined local coordinate system in node is described using `lcs` array. This array contains six numbers, where the first three numbers represent a directional vector of the local x-axis, and the next three numbers represent a directional vector of the local y-axis. The local z-axis is determined using a vector product. A right-hand coordinate system is assumed. If user defined local coordinate system in node is specified, then the boundary conditions and applied loading are specified in this local coordinate system. The reactions and displacements are also in `lcs` system at the output.

  The node can create only master DOFs and SimpleSlave DOFs, so the allowable values of `dofType` array are in range 0,1. For the Node dof manager, the `masterMask` is the array of size equal to number of DOFs, and the i-th value determines the master dof manager, to which i-th dof is directly linked (the dof with same physical meaning are linked together). The local coordinate system in node with same linked dofs is supported, but it should be exactly the same as on master.

- Rigid arm record

  `RigidArmNode coords #(ra) master #(in) [masterMask #(ia)][lcs #(ra)]`
  Represent node connected to other node (called master) using rigid arm. Rigid arm node DOFs can be linked to master (via rigid arm transformation) or can be independent. The rigid arm node 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 can be localized directly to master related equations. The rigid arm node can not have its own boundary or initial conditions, they are determined completely from master dof conditions. Currently it is possible to map only certain dofs - see `dofType`. Linked DOFs should have dofType value equal to 2, non-linked (primary) DOFs 0.

Rigid arm node can be loaded independently of master. The node coordinates in space (given by global coordinate system) are described using `coords` field. This array contains x, y and possibly z (depends on problem under consideration) coordinate of node. The `master` parameter is the master node number, to which rigid arm node dofs are mapped. The rigid arm node and master can have arbvitrary local coordinate systems (if not specified, global one is assumed).

The optional parameter `masterMask` allows to specify how particular mapped DOF depends on master DOFs. The size of `masterMask` array should be equal to number of DOFs. For all linked DOFs (with corresponding dofType value equal to 2) the corresponding value of `masterMask` array should be 1.

The local coordinate system in rigid arm node is supported, the coordinate system in master and slave can be different. If no lcs is set, global one is assumed.the global cs applies.

- Hanging node

  `HangingNode coords #(ra) dofType #(in) [masterElement #(in)] [masterRegion #(in)]`

  Hanging node is connected to an a master element using generalized interpolation. Hanging node posses no degrees of freedom (except unlined dofs) - all values are interpolated from corresponding master elements and its DOFs. arbitrary FE mesh of concrete specimen or to facilitate the local refinement of FE mesh. The hanging nodes can be in a chain.

  The contributions of hanging node are localized directly to master related equations. The hanging node can have its own boundary or initial conditions, but only for primary unlinked DOFs. For linked DOFs, these conditions are determined completely from master DOF conditions. The local coordinate system should be same for all master nodes. The hanging node can be loaded independently of its master.

  Values of array `dofType` can have following values: 0-primary DOF, 2-linked DOF.

  The value of `masterElement` specifies the element number to which the hanging node is attached. The node can be attached to any arbitrary coordinate within the master element. The element must support the necessary interpolation classes. The same interpolation for unknowns and geometry is assumed.

  The no (or -1) value for `masterElement` is supplied, then the node will locate the element closest to its coordinate. If no (or zero) value for `masterRegion` is supplied, then all regions will be searched, otherwise only the elements in cross section with number `masterRegion`. If `masterElement` is directly supplied `masterRegion` is unused.

- Slave node

  `SlaveNode coords #(ra) dofType #(in) masterDofMan #(ia) weights #(ra)`

  Works identical to hanging node, but the weights (`weights`) are not computed from any element, but given explicitly, as well as the connected dof managers (`masterDMan`).

- Element side

  `ElementSide`
  Represents an abstraction for element side, which holds some unknowns.

- PFEMParticle

  `PFEMParticle coords #(ra)`
  Represent the particle used in PFEM analysis.

- InteractionPFEMParticle

`InteractionPFEMParticle coords #(ra) bc #(ia) coupledNode #(in)`
Represent a special particle used in the PFEM-part of the FluidStructureProblem. The particle is attached to `coupledNode` from the structural counter part. InteractionBoundaryCondition (see *interactionbc*) must be prescribed under `bc` to access the velocities from solid nodes.

5.4 Element records

These records specify a description of particular elements. The general format is following:

`ElementType #(in) mat #(in) crossSect #(in) nodes #(ia) [bodyLoads #(ia)] [boundaryLoads #(ia)] [activityltf #(in)] [lcs #(ra)] <[partitions #(ia)]> <[remote]>`

The order of element records is optional, the element number is determined by `#(in)` parameter. The numbering of individual elements is arbitrary, it could be even non-continuous. In this context, one could think of element number as a label that is assigned to individual elements and by which the element is referenced.

Element material is described by parameter `mat`, which contains corresponding material record number. Element cross section is determined by cross section with `crossSect` record number. Element dof managers (nodes, sides, etc.) defining element geometry are specified using `nodes` array.

Body load acting on element is specified using `bodyLoads` array. Components of this array are corresponding load record numbers. The loads should have the proper type (body load type), otherwise error will be generated.

Boundary load acting on element boundary is specified using `boundaryLoads` array. The format of this array is

$$2 \cdot size\ lnum(1)\ id(1)\ \ldots\ lnum(size)\ id(size),$$

where $size$ is total number of loadings applied to element, $lnum(i)$ is the applied load number, and $id(i)$ is the corresponding entity number, to which the load is applied (for example a side or a surface number). The entity numbering is element dependent and is described in element specific sections. The applied loads must be of proper type (boundary load type), otherwise error is generated.

The support for element insertion and removal during the analysis is provided. One can specify optional time function (identified by its id using `activityltf` parameter). The nonzero value of this time function indicates, whether the element is active (nonzero value, the default) or inactive (zero value) at particulat solution step. Tested for structural and transport elements. This feature allows considering temperature evolution of layered casting of concrete, where certain layers needs to be inactive before they are cast. See a corresponding example in oofem tests how to enforce hydrating material model, boundary conditions and element activity acting concurrently.

Orientation of local coordinates can be specified using `lcs` array. This array contains six numbers, where the first three numbers represent a directional vector of local x-axis, and the next three numbers represent a directional vector of local y-axis. The local z-axis is determined using the vector product. The `lcs` array on the element is particularly useful for modeling of orthotropic materials which follow the element orientation. On a beam or truss element, the `lcs` array has no effect and the 1D element orientation is aligned with the global xx component.

Available material models, their outline and corresponding parameters are described in separate **Element Library Manual.**

5.5 Set records

Sets specify regions of the geometry as a combination of volumes, surfaces, edges, and nodes. The main usage of sets are to connect regions of elements to a given cross section or apply a boundary condition, though sets can be used for

many other things as well.

`Set #(in) [elements #(ia)] [elementranges #(rl)] [allElements] [nodes #(ia)] [noderanges #(rl)][allNodes][elementboundaries #(ia)][elementedges #(ia)]`

Volumes (elements) and nodes can be specified using either a list, `elements`, `nodes`, or with a range list `elementranges`, `noderanges`. Edges `elementedges`, and surfaces `elementboundaries`, are specified in a interleaved list, every other number specifying the element, and edge/surface number (the total length of the list being twice the number of surfaces/edges). The internal numbering of edges/surfaces is available in the **Element Library Manual**.

Note that edge loads (singular loads given in "newton per length" (or equivalent), should be applied to `elementedges`, surface loads "newton per area" on `elementboundaries`, and bulk loads "newton per volume" on `elements`.

Example 1: A deadweight (gravity) load would be applied to the `elements` in a set, while a distributed line load would be applied to the midline "edge" of the beam element, thus should be applied to a `elementedges` set. In the latter case, the midline of the beam is defined as the first (and only) "edge" of the beam.

Example 2: Axisymmetric structural element analysis: A deadweight load would be applied to `elements` in a set. A external pressure would be defined as a surface load an be applied to the `elementboundaries` in a set. The element integrates the load (analytically) around the axis, so the load would still count as a surface load.

5.6 Cross section records

These records specify a cross section model descriptions. The general format is following:

`CrossSectType #(in)`

The order of particular cross section records is optional, cross section model number is determined by `#(in)` parameter. The numbering should start from one and should end at n, where n is the number of records.

The crossSectType keyword can be one from following possibilities

- Integral cross section with constant properties
  `SimpleCS [thick #(rn)][width #(rn)][area #(rn)][iy #(rn)][iz #(rn)][ik #(rn)] [shearareay #(rn)][shearareaz #(rn)]beamshearcoeff #(rn)`
  Represents integral type of cross section model. In current implementation, such cross section is described using cross section thick (`thickVal`) and width (`widthVal`). For some problems (for example 3d), the corresponding volume and cross section dimensions are determined using element geometry, and then you can omit some (or all) parameters (refer to documentation of individual elements for required cross section properties). Parameter `area` allows to set cross section area, parameters `iz`, `iz`, and `ik` represent inertia moment along y and z axis and torsion inertia moment. Parameter `beamshearcoeff` allows to set shear correction factor, or equivalent shear areas (`shearareay` and `shearareaz` parameters) can be provided. These cross section properties are assumed to be defined in local coordinate system of element.
- Integral cross section with variable properties
  `VariableCS [thick #(expr)][width #(expr)][area #(expr)][iy #(expr)][iz #(expr)][ik #(expr)][shearareay #(expr)][shearareaz #(expr)]`
  Represents integral type of cross section model, where individual cross section parameters can be expressed as an arbitrary function of global coordinates x,y,z. Similar to SimpleCS, for some problems (for example 3d), the corresponding volume and cross section dimensions are determined using element geometry, then you can omit many (or some) parameters (refer to documentation of individual elements for required cross section properties). Parameter `area` allows to set cross section area, parameters `iz`, `iz`, and `ik` represent inertia moment along y and z axis and torsion inertia moment. Parameters (`shearareay` and `shearareaz` determine shear area, which is required by beam and plate elements. All cross section properties are assumed to be defined in local coordinate system of element.

- Layered cross section

  `LayeredCS nLayers #(in) LayerMaterials #(ia) Thicks #(ra) Widths #(ra) midSurf #(rn)`

  Represents the layered cross section model, based on geometrical hypothesis, that cross sections remain planar after deformation. Number of layers is determined by `nLayers` parameter. Materials for each layer are specified by `LayerMaterials` array. For each layer is necessary to input geometrical characteristic, thick - using `Thicks` array, and width - using `Widths` array. Position of mid surface is determined by its distance from bottom of cross section using `midSurf` parameter (normal and momentum forces are then computed with regard to it's position). Elements using this cross section model must implement layered cross section extension. For information see element library manual.

- Fibered cross section

  `FiberedCS nfibers #(in) fibermaterials #(ia) thicks #(ra) widths #(ra) thick #(rn) width #(rn) fiberycentrecoords #(ra) fiberzcentrecoords #(ra)`

  Cross section represented as a set of rectangular fibers. It is based on geometrical hypothesis, that cross sections remain planar after deformation (3d generalization of layered approach for beams). Paramater `nfibers` determines the number of fibers that together form the overall cross section. The model requires to specify a material model corresponding to particular fiber using `fibermaterials` array. This array should contain for each fibre corresponding material model number (the material model specified on element level has no meaning in this particular case). **The geometry of cross section is determined from fiber dimensions and fiber positions, all input in local coordinate system of the beam (yz plane).** The thick and width of each fiber are determined using `thicks` and `widths` arrays. The overall thick and width are specified using parameters `thick` and `width`. Positions of particular fibers are specified by providing coordinates of center of each fiber using `fiberycentrecoords` array for y-coordinates and `fiberzcentrecoords` array for z-coordinates.

- Warping cross section

  `WarpingCS WarpingNode #(in)`

  Represents the cross section for Free warping analysis, see section *FreeWarping*. The `WarpingNode` parametr defines the number of external node with prescribed boundary condition which corresponds to the relative twist of warping cross section.

5.7 Material type records

These records specify a material model description. The general format is following:

`MaterialType #(in) d #(rn)`

The order of particular material records is optional, the material number is determined by `#(in)` parameter. The numbering should start from one and should end at n, where n is the number of records. Material density is compulsory parameter and it's value is given by `d` parameter.

Available material models, their outline and corresponding parameters are described in separate **Material Library Manual**.

5.8 Nonlocal barrier records

Nonlocal material models of integral type are based on replacement of certain suitable local quantity in local constitutive law by their nonlocal counterparts, that are obtained as weighted average over some characteristic volume. The weighted average is computed as a sum of a remote value multiplied by weight function value. The weight function typically depend on a distance between remote and receiver points and decreases with increasing distance. In some cases, it is necessary to disregard mutual interaction between some points (for example if they are on the opposite sides

of a thin notch, which prevents the nonlocal interactions to take place). The barriers are the way how to introduce these constrains. The barrier represent a curve (in 2D) or surface (in 3D). When the line connecting receiver and remote point intersects a barrier, the barriers is activated and the corresponding interaction is not taken into account.

Currently, the supported barrier types are following:

- Polyline barrier

  `polylinebarrier #(in) vertexnodes #(ia) [xcoordindx #(in)] [ycoordindx #(in)]`
  This represents a polyline barrier for 2D problems. Barrier is a polyline, defined as a sequence of nodes representing vertices. The vertices are specified using parameter `vertexnodes` array, which contains the node numbers. The optional parameters `xcoordindx` and `ycoordindx` allow to select the plane (xy, yz, or xz), where the barrier is defined. The `xcoordindx` is the first coordinate index, `ycoordindx` is the second. The default values are 1 for `xcoordindx` and 2 for `ycoordindx`, representing barrier in xy plane.

- Symmetry barrier

  `symmetrybarrier #(in) origin #(ra) normals #(ra) activemask #(ia)`
  Implementation of symmetry barier, that allows to specify up to three planes (orthogonal ones) of symmetry. This barrier allows to model the symmetry of the averaged field on the boundary without the need of modeling the other part of structure across the plane of symmetry. It is based on modifying the integration weights of source points to take into account the symmetry. The potential symmetry planes are determined by specifying orthogonal right-handed coordinate system, where axes represent the normals of corresponding symmetry planes. Parameter `origin` determines the origin of the coordinate system, the `normals` array contains three components of x-axis direction vector, followed by three components of y-axis direction vector (expressed in global coordinate system). The z-axis is determined from the orthogonality conditions. Parameter `activemask` allows to specify active symmetry planes; i-th nonzero value activates the symmetry barrier for plane with normal determined by corresponding coordinate axis (x=1, y=2, z=3).

5.9 Load and boundary conditions

These records specify description of boundary conditions. The general format is following:

`EntType #(in) loadTimeFunction #(in) [valType #(in)] [dofs #(ia)] [isImposedTimeFunction #(in)]`

The order of particular records is optional, boundary condition number is determined by `#(in)` parameter. The numbering should start from one and should end at n, where n is the number of records. Time function value (given by `loadTimeFunction` parameter) is a multiplier, using which each component (value of loading or value of boundary condition) describes its time variation. The optional parameter `valType` allows to determine the physical meaning of bc value, which is sometimes required. Supported values are (1 - temperature, 2 - force/traction, 3 - pressure, 4 - humudity, 5 - velocity, 6 - displacement). Another optional parameter `dofs` is used to determine which dofs the boundary condition should act upon. It is not relevant for all BCs..

The nonzero value of `isImposedTimeFunction` time function indicates that given boundary condition is active, zero value indicates not active boundary condition in given time (the bc does not exist). By default, the boundary condition applies at any time.

Currently, EntType keyword can be one from

- Dirichlet boundary condition

  `BoundaryCondition prescribedvalue #(rn) [d #(rn)]`

Represents boundary condition. Prescribed value is specified using `prescribedvalue` parameter. The physical meaning of value is fully determined by corresponding DOF. Optionally, the prescribed value can be specified using `d` parameter. It is introduced for compatibility reasons. If `prescribedvalue` is specified, then `d` is ignored.

- Prescribed gradient boundary condition (Dirichlet type)

  `PrescribedGradient gradient #(rm) [cCoords #(ra)]`

  Prescribes $v_i = d_{ij}(x_j - \bar{x}_j)$ or $s = d_{1j}(x_j - \bar{x}_j)$ where v_i are primary unknowns, x_j is the coordinate of the node, $\bar{x}$ is `cCoords` and d is `gradient`. The parameter `cCoords` defaults to zero. This is typical boundary condition in multiscale analysis where $d = \partial_x s$ would a macroscopic gradient at the integration point, i.e. this is a boundary condition for prolongation. It is also convenient to use when one wants to test a arbitrary specimen for shear.

- Mixed prescribed gradient / pressure boundary condition (Active type)

  `MixedGradientPressure devGradient #(ra) pressure #(rn) [cCoord #(ra)]`

  All boundary conditions of ensures that the deviatoric gradient and pressure is at least weakly fullfilled on the prescribed domain. They are used for computational homogenization of incompressible flow or elasticity problems.

- Mixed prescribed gradient / pressure boundary condition (Weakly periodic type)

  `MixedGradientPressureWeaklyPeriodic order #(rn)`

  Prescribes a periodic constant (unknown) stress tensor along the specified boundaries. For `order` set to 1, one obtains the same results as the Neumann boundary condition.

- Mixed prescribed gradient / pressure boundary condition (Neumann type)

  `MixedGradientPressureNeumann`

  Prescribes a constant (unknown) deviatoric stress tensor along the specified boundaries. Additional unknowns appears, $\boldsymbol{\sigma}_{\mathrm{dev}}$, which is handled by the boundary condition itself (no control from the input file). The input devGradient is weakly fulfilled (homogenized over the elementsides). As with the the Dirichlet type, the volumetric gradient is free. This is useful in multiscale computations of RVE's that experience incompressible behavior, typically fluid problems. In that case, the element sides should cover the entire RVE boundary. It is also convenient to use when one wants to test a arbitrary specimen for shear, with a free volumetric part (in which case the pressure is set to zero). Symmetry is not assumed, so rigid body rotations are removed, but translations need to be prescribed separately.

- Mixed prescribed gradient / pressure boundary condition (Dirichlet type)

  `MixedGradientPressureDirichlet`

  Prescribes $v_i = d_{\mathrm{dev},ij}(x_j - \bar{x}_j) + d_{\mathrm{vol}}(x_i - \bar{x}_i)$, and a pressure p. where v_i are primary unknowns, x_j is the coordinate of the node, $\bar{x}$ is `cCoords` and d_{dev} is `devGradient`. The parameter `cCoords` defaults to zero. An additional unknown appears, d_{vol}, which is handled by the boundary condition itself (no control from the input file). This unknown is in a way related to the applied pressure. This is useful in multiscale computations of RVE's that experience incompressible behavior, typically fluid problems. It is also convenient to use when one wants to test a arbitrary specimen for shear, with a free volumetric part (in which case the pressure is set to zero).

- Nodal fluxes (loads) `NodalLoad components #(ra) [cstype #(in)]` Concentrated nodal load. The components of nodal load vector are given by `components` parameter. The size of this vector corresponds to a total number of nodal DOFs, and i-th value corresponds to i-th DOF in associated dof manager. The load can be defined in global coordinate system (`cstype` = 0) or in entity - specific local coordinate system (`cstype` = 1, default).

- `PrescribedTractionPressureBC`

Represents pressure boundary condition (of Dirichlet type) due to prescribed tractions. In CBS algorithm formulation the prescribed traction boundary condition leads indirectly to pressure boundary condition in corresponding nodes. This boundary condition implements this pressure bc. The value of bc is determined from applied tractions, that should be specified on element edges/surfaces using suitable boundary loads.

- Linear constraint boundary condition

  `LinearConstraintBC weights #(ra) [weightsLtf #(ia)] dofmans #(in) dofs #(in) rhs #(rn) [rhsLtf #(in)] lhstype #(ia) rhsType #(ia)`

  This boundary condition implements a linear constraint in the form $\sum_i w_i r_i = c$, where r_i are unknowns related to DOFs determined by `dofmans` and `dofs`, the weights are determined by `weights` and `weightsLtf`. The constant is determined by `rhs` and `rhsLtf` parameters. This boundary condition is introduced as additional stationary condition using Lagrange multiplier, which is an additional degree of freedom introduced by this boundary condition.

  The individual DOFs are determined using dof manager numbers (`dofmans` array) and corresponding DOF indices (`dofs`). The weights corresponding to participating DOFs are specified using `weights` array. The weights are multiplied by value returned by load time function, associated to individual weight using optional `weightsLtf` array. By default, all weights are set to 1. The constant c is determined by `rhs` parameter and it is multiplied by the value of load time function, specified using `rhsLtf` parameter, or by 1 by default. The characteristic component, to which this boundary condition contributes must be identified using `lhstype` and `rhsType` parameters, values of which are corresponding to CharType enum. The left hand side contribution is assembled into terms identified by `lhstype`. The rhs contribution is assembled into the term identified by `rhsType` parameter. Note, that multiple values are allowed, this allows to select all variants of stifness matrix, for example. Note, that the size of `dofmans`, `dofs`, `weights`, `weightsLtf` arrays should be equal.

- InteractionBoundaryCondition

  `InteractionBoundaryCondition`

  Is a special boundary condition prescribed on InteractionPFEMParticles (see *interactionparticle* in the PFEM part of the FluidStructureProblem. This sort of particles is regarded as it would have prescribed velocities, but the values change dynamically, as the solid part deforms. The velocities are obtained from coupled structural nodes.

- Body loads
  - Volume flux (load)

    `DeadWeight components #(ra)`

    Represents dead weight loading applied on element volume (for structural elements). For transport problems, it represents the internal source, i.e. the rate of (heat) generated per unit volume. The magnitude of load for specific i-th DOF is computed as product of material density, corresponding volume and i-th member of `components` array.

  - Structural temperature load

    `StructTemperatureLoad components #(ra)`

    Represents temperature loading imposed to some elements. The members of `components` array represent the change of temperature (or change of temperature gradient) corresponding to specific element strain components. See element library manual for details.

  - Structural eigenstrain load `StructEigenstrainLoad components #(ra)`

Prescribes eigenstrain (or stress-free strain) to a structural element. The array of `components` is defined in the global coordinate system. The number of components corresponds to a material mode, e.g. plane stress has three components and 3D six. Periodic boundary conditions can be imposed using eigenstrains and master-slave nodes. Consider decomposition of strain into average and fluctuating par .. math:: boldsymbol{varepsilon}(boldsymbol{x}) = langle boldsymbol{varepsilon} rangle + boldsymbol{varepsilon}^*(boldsymbol{x})

where $\langle\boldsymbol{\varepsilon}\rangle$ can be imposed as eigenstrain over the domain and the solution gives the fluctuating part $\boldsymbol{\varepsilon}^*(\boldsymbol{x})$. Master-slave nodes have to interconnect opposing boundary nodes of a unit cell.

- Boundary loads - Constant edge fluxes (load)

```
ConstantEdgeLoad        loadType #(in)        components #(ra)
[dofexcludemask #(ia)]    [csType #(in)]    [properties #(dc)]
[propertytf #(dc)]
```

  - Constant surface fluxes (load)

```
ConstantSurfaceLoad        loadType #(in)        components #(ra)
[dofexcludemask #(ia)] [csType #(in)] [properties #(dc)] [propertytf
#(dc)]
```

    Represent constant edge/surface loads or boundary conditions. Parameter `loadType` distinguishes the type of boundary condition. Supported values are specified in bctype.h:

    * `loadType` = 2 prescribed flux input (Neumann boundary condition),
    * `loadType` = 3 uniform distributed load or the convection (Newton) BC. Parameter `components` contains the environmental values (temperature of the environment) corresponding to element unknowns, and `properties` dictionary should contain value of transfer (convection) coefficient (assumed to be a constant) under the key 'a',
    * `loadType` = 7 specifies radiative boundary condition (Stefan-Boltzmann). It requires to specify emmisivity $\varepsilon \in \langle 0, 1\rangle$, the `components` array contains the environmental values (temperature of the environment). Default units are Celsius. Optional parameter `temperOffset` = 0 can be used to calculate in Kelvin.

    If the boundary condition corresponds to distributed force load, the `components` array contains components of distributed load corresponding to element unknowns. The load is specified for all DOFs of object to which is associated. For some types of boundary conditions the zero value of load does not mean that the load is not applied (Newton's type of bc, for example). Then some mask, which allows to exclude specific dofs is necessary. The `dofexcludemask` parameter is introduced to alow this. It should have the same size as `components` array, and by default is filled with zeroes. If some value of dofExcludeMask is set to nonzero, then the corresponding componentArray is set to zero and load is not applied for this DOF. If the boundary condition corresponds to prescribed flux input, then the `components` array contains the components of prescribed input flux corresponding to element unknowns.

    The properties can vary in time. Each property can have associated time function which determines its time variation. The time functions are set up using optional `propertytf` dictionary, containing for selected properties the corresponding time function number. The time function must be registered under the same key as in `properties` dictionary. The property value is then computed by product of property value (determined by `properties`) and corresponding time function evaluated at given time. If no time function provided for particula property, a unit constant function is assumed.

    The load can be defined in global coordinate system (`csType` = 0, default) or in entity - specific local coordinate system (`csType` = 1).

  - Linear edge flux (load)

`LinearEdgeLoad loadType #(in) components #(ra) [dofexcludemask #(ia)] [csType #(in)]`

Represents linear edge load. The meanings of parameters `csType` and `loadType` are the same as for **ConstantEdgeLoad**. In `components` array are stored load components for corresponding unknowns at the beginning of edge, followed by values valid for end of edge. The load can be defined in global coordinate system (`csType` = 0, default) or in entity - specific local coordinate system (`csType` = 1).

– InteractionLoad

`InteractionLoad ndofs #(in) loadType #(in) Components #(ra) [csType #(in)] coupledparticles #(ia)`

Represents a fluid pressure induced load in the solid part of the FluidStructureProblem. The meanings of parameters `ndofs`, `csType`, and `loadType` are the same as for **LinearEdgeLoad**. In `Components` array are stored load components for corresponding unknowns at the beginning of edge (`ndofs` values), followed by values valid for end of edge (`ndofs` values). The load should be defined in global coordinate system (`csType` = 0) as it acts in normal direction of the edge. Array `coupledparticles` assign PFEMParticles from the fluid part of the problem providing fluid pressure.

5.10 Initial conditions

These records specify description of initial conditions. The general format is following:

`InitialCondition #(in) conditions #(dc)` The order of particular records is optional, load, boundary or initial condition number is determined by (`num`#)(in) parameter. The numbering should start from one and should end at n, where n is the number of records. Initial parameters are listed in `conditions` dictionary using keys followed by their initial values. Now 'v' key represents velocity and 'a' key represents acceleration.

5.11 Time functions records

These records specify description of time functions, which generally describe time variation of components during solution. The general format is following:

`TimeFunctType #(in) [initialValue #(rn)]`

The order of these records is optional, time function number is determined by `#(in)` parameter. The `initialValue` parameter allows to control the way, how increment of receiver is evaluated for the first solution step. This first solution step increment is evaluated as the difference of value of receiver at this first step and given initial value (which is by default set to zero). The increments of receiver in subsequent steps are computed as a difference between receiver evaluated at given solution step and in previous step.

The numbering should start from one and should end at n, where n is the number of records.

Currently, TimeFunctType keyword can be one from

- Constant function

  `ConstantFunction f(t) #(rn)`

  Represents the constant time function, with value `f(t)`.

- Peak function

  `PeakFunction t #(rn) f(t) #(rn)`

  Represents peak time function. If time is equal to `t` value, then the value of time function is given by `f(t)` value, otherwise zero value is returned.

- Piecewise function

  `PiecewiseLinFunction [nPoints #(in) t #(ra) f(t) #(ra)] [datafile #("string")]`

  Represents the piecewise time function. The particular time values in `t` array should be sorted according to time scale. Corresponding time function values are in `f(t)` array. Value for time, which is not present in `t` is computed using liner interpolation scheme. Number of time-value pairs is in `nPoints` parameter.

  The second alternative allows reading input data from an external ASCII file. A hash commented line (#) is skipped during reading. File name should be eclosed with " ".

- Heaviside-like time function

  `HeavisideLTF origin #(rn) value #(rn)`

  Up to time, given by parameter `origin`, the value of time function is zero. If time greater than `origin` parameter, the value is equal to parameter `value` value.

- User defined

  `UsrDefLTF f(t) #(expr) [dfdt(t) #(expr)] [d2fdt2(t) #(expr)]`

  Represents user defined time function. The expressions can depend on "t" parameter, for which actual time will be substituted and expression evaluated. The function is defined using `f(t)` parameter, and optionally, its first and second time derivatives using `dfdt(t)` and `d2fdt2(t)` parameters. The first and second derivatives may be required, this depend on type of analysis.

  Very general, but relatively slow.

5.12 Xfem manager record and associated records

This record specifies the number of enrichment items and simulation options common for all enrichment items. Functions used for enrichment (e.g. Heaviside, abs or branch functions) are not specified here, they are specified for each enrichment item separately. The same holds for the geometrical representation of each enrichment item (e.g. a polygon line or a circle). Currently, OOFEM supports XFEM simulations of cracks and material interfaces in 2D. The input format for the XFEM manager is:

`XfemManager numberofenrichmentitems #(in) numberofgppertri #(in) debugvtk #(in) vtkexport #(in) exportfields #(in)`

where `numberofenrichmentitems` represents number of enrichment items, `numberofgppertri` denotes the number of Gauss points in each subtriangle of a cut element (default 12) and `debugvtk` controls if additional debug vtk files should be written (1 activates the option, 0 is default).

The specification of an enrichment item may consist of several lines, see e.g. the test *sm/xFemCrackValBranch.in*. First, the enrichment item type is specified together with some optional parameters according to

`EntType #(in) enrichmentfront #(in) propagationlaw #(in)`

where `enrichmentfront` specifies an enrichment front (we may for example employ branch functions at a crack tip and Heaviside enrichment along the rest of the crack, hence the "front" of the enrichment is treated separately) and `propagationlaw` specifies a rule for crack propagation (this feature is still highly experimental though). Specification of an `enrichmentfront` and a `propagationlaw` is optional.

The next line specifies the enrichment function to be used:
`EntType #(in)`

This is followed by a line specifying the geometric description (e.g. a polygon line or a circle) according to
`EntType #(in) extra attributes` where the number and type of extra attributes to specify will vary depending on the geometry chosen, e.g. center and radius for a circle or a number of points for a polygon line.

If an enrichment front was specified previously, the type and properties of the enrichment front are specified on the next line according to
`EntType #(in) extra attributes`

If a propagation law was specified previously, it's type and properties are also specified on a separate line according to
`EntType #(in) extra attributes`

CHAPTER

SIX

APPENDIX

6.1 Sparse linear solver parameters

The sparselinsolverparams field has the following general syntax:
`[lstype #(in)] [smtype #(in)] solverParams #(string)` where parameter `lstype` allows to select the solver for the linear system of equations. Currently supported values are 0 (default) for a direct solver (ST_Direct), 1 for an Iterative Method Library (IML) solver (ST_IML), 2 for a Spooles direct solver, 3 for Petsc library family of solvers, and 4 for a DirectSparseSolver (ST_DSS). Parameter `smtype` allows to select the sparse matrix storage scheme. The scheme should be compatible with the solver type. Currently supported values (marked as "id") are summarized in table *linsolvstoragecompattable*. The 0 value is default and selects the symmetric skyline (SMT_Skyline). Possible storage formats include unsymmetric skyline (SMT_SkylineU), compressed column (SMT_CompCol), dynamically growing compressed column (SMT_DynCompCol), symmetric compressed column (SMT_SymCompCol), spooles library storage format (SMT_SpoolesMtrx), PETSc library matrix representation (SMT_PetscMtrx, a sparse serial/parallel matrix in AIJ format), and DSS compatible matrix representations (SMT_DSS). The allowed `lstype` and `smtype` combinations are summarized in the table *linsolvstoragecompattable*, together with solver parameters related to specific solver.

Table 1: Solver and storage scheme compatibility.

Storage format	`smtype` id	`lstype (id)`						
	id	Direct (0)	IML (1)	Spooles (2)	Petsc (3)	DSS (4)	MKLPardiso (6) Pardiso.org(8)	SuperLU_MT (7)
SMT_Skyline	0	•	•					
SMT_SkylineLU	1	•	•					
SMT_CompCol	2		•				•	•
SMT_DynCompCol	3		•					
SMT_SymCompCol	4		•					
SMT_DynCompRow	5		•					
SMT_SpoolesMtrx	6			•				
SMT_PetscMtrx	7				•			
SMT_DSS_sym_LDL	8							•
SMT_DSS_sym_LL	9							•
SMT_DSS_sym_LU	10							•

The solver parameters in `solverParams` depend on the solver type and are summarized in table (*sparsesolver-params*).

Table 2: Solver parameters.

Solver type	id	Solver parameters/notes
ST_Direct	0	
ST_IML	1	[`stype` #(in)] `lstol` #(rn) `lsiter` #(in)`lsprecond` #(in)
		[`precondattributes` #(string)]
		Included in OOFEM, requires to compile with USE_IML
ST_Spooles	2	[`msglvl` #(in)] [`msgfile` #(s)]
		http://www.netlib.org/linalg/spooles/spooles.2.2.html
ST_Petsc	3	See Petsc manual, for details
ST_DSS	4	Sparse direct solver, included in OOFEM
		Requires to compile with USE_DSS
ST_MKLPardiso	6	Requires Intel MKL Pardiso
ST_SuperLU_MT	7	SuperLU for shared memory machines
		http://crd-legacy.lbl.gov/ xiaoye/SuperLU/
ST_PardisoProjectOrg	8	Requires Pardiso solver(http://www.pardiso-project.org/)

In case of ST_PETSC, the user can set several run-time options, e.g., `-ksp\_type [cg, gmres, bicg, bcgs] -pc\_type [jacobi, bjacobi,none,ilu,...] -ksp\_monitor -ksp\_rtol # -ksp\_view -ksp\_converged_reason`. These options will override those that are default (PETSC KSPSetFromOptions() routine is called after any other customization routines).}

The `stype` allows to select particular iterative solver from IML library, currently supported values are 0 (default) for Conjugate-Gradient solver, 1 for GMRES solver. Parameter `lstol` represents the maximum value of residual after the final iteration and the `lsiter` is maximum number of iteration for iterative solver. The `precondattributes` parameters contains the optional preconditioner parameters. The `lsprecond` parameter determines the type of preconditioner to be used. The possible values of `lsprecond` together with supported storage schemes and their descriptions are summarized in table *precondtable*.

Table 3: Preconditioning summary.

Precond type	id	Compatible storage	Description and parameters
IML_VoidPrec	0	all	No preconditioning
IML_DiagPrec	1	all	Diagonal preconditioning
IML_ILUPrec	2	SMT_CompCol	Incomplete LU Decomposition
		SMT_DynCompCol	with no fill up
IML_ILUPrec	3	SMT_DynCompRow	Incomplete LU (ILUT) with
			fillup.
			The `precondattributes` are:
			[`droptol` #(rn)] [`partfill` #(in)].
			`droptol` dropping tolerance
			`partfill` level of fill-up
IML_ICPrec	4	SMT_SymCompCol	Incomplete Cholesky
		SMT_CompCol	with no fill up

6.2 Eigen value solvers

The eigensolverparams field has the following general syntax:
`stype #(in) [smtype #(in)] solverParams #(string)` where parameter `stype` allows to select solver type. Parameter `smtype` allows to select sparse matrix storage scheme. The scheme should be compatible with solver type. Currently supported values of `stype` are summarized in table *eigenvaluesolverparamtable*.

Table 4: Eigen Solver parameters.

Solver type	stype id	solver parameters
Subspace Iteration	0 (default)	
Inverse Iteration	1	
SLEPc solver	2	requires “smtype 7” see also SLEPc manual

There are in general two basic factors causing load imbalance between individual subdomains: (i) one comming from application nature, such as switching from linear to nonlinear response in certain regions or local adaptive refinment, and (ii) external factors, caused by resourse realocation, typical for nondedicated cluster environments, where indivudual processors are shared by different applications and users, leading to time variation in allocated processing power. The load balance recovery is achieved by repartitioning of the problem domain and transferring the work (represented typically by finite elements) from one subdomain to another. This section describes the structure and syntax of parameters related to dynamic load balancing. The corresponding part of analysis record has the following general syntax:

```
[lbflag #(in)][forcelb1 #(in)][wtp #(ia)][lbstep #(in)][relwct #(rn)][abswct
#(rn)][minwct #(rn)]
```

where the parameters have following meaning:

- `lbflag`, when set to nonzero value activates the dynamic load balancing. Default value is zero.
- `forcelb1` forces the load rebalancing after the first solution step, when set to nonzero value.
- `wtp` allows to activate optional load balancing plugins. At present, the only supported value is 1, that activates nonlocal plugin, necessary for nonlocal averaging to work properly when dynamic load balancing is active.
- `lbstep` rebalancing, if needed, is performed only every lbstep solution step. Default value is 1 (recover balance after every step, if necessary).
- `relwcr` sets relative wall-clock imbalance treshold. When achieved relative imbalance between wall clock solution time of individual processors is greater than provided treshold, the rebalancing procedure will be activated.
- `abswct` sets absolute wall-clock imbalance treshold. When achieved absolute imbalance between wall clock solution time of individual processors is greater than provided treshold, the rebalancing procedure will be activated.
- `minwct` minimum absolute imbalance to perform relative imbalance check using `relwcr` parameter, otherwise only absolute check is done. Default value is 0.

At present, the load balancing support requires ParMETIS module to be configured and compiled.

6.3 Error estimators and indicators

The currently supported values of `eetype` are in table *eetypestable*.

- EET_SEI - Represents scalar error indicator. It indicates element error based on the value of some suitable scalar value (for example damage level, plastic strain level) obtained from the element integration points and corresponding material model.
- EET_ZZEE - The implementation of Zienkiewicz Zhu Error Estimator. It requires the special element algorithms, which may not be available for all element types.

  Please note, that in the actual version, the error on the element level is evaluated using default integration rule. For example, in case of ZZ error estimator, the error (L2 or energy norm) is evaluated from the difference of computed and "recovered" stresses, which are approximated using the same interpolation functions as displacements). Therefore, in many cases, the default integration rule order is not sufficient and higher integration must be used on elements (consult element library manual and related NIP parameter).
- EET_CZZSI - The implementation of combined criteria: Zienkiewicz Zhu Error Estimator for elastic regime and scalar error indicator in non-linear regime.

Table 5: Supported error estimators and indicators.

Error estimator/indicator	`eetype`
EET_SEI	0
EET_ZZEE	1
EET_CZZSI	2

The sets of parameters (`errorestimatorparams` field) used to configure each error estimator are different

- `EET_SEI`

`[regionskipmap #(ia)] vartype #(in) minlim #(rn) maxlim #(rn) mindens #(rn) maxdens #(rn) defdens #(rn) [remeshingdensityratio #(rn)]`

  - `regionskipmap` parameter allows to skip some regions. The error is not evaluated in these regions and default mesh density is used. The size of this array should be equal to number of regions and nonzero entry indicates region to skip.
  - `vartype` parameter determines the type of internal variable to be used as error indicator. Currently supported value is 1, representing damage based indicator.
  - If the indicator value is in range given by parameters (`minlim`, `maxlim`) then the proposed mesh density is linearly interpolated within range given by parameters (`mindens`, `maxdens`). If indicator value is less than value of `minlim` parameter then value of `defdens` parameter is used as required density, if it is larger than `maxlim` then `maxdens` is used as required density.
  - `remeshingdensityratio` parameter determines the allowed ratio between proposed density and actual density. The remeshing is forced, whenever the actual ratio is smaller than this value. Default value is equal to 0.80.

- `EET_ZZEE`
  `[regionskipmap #(ia)] normtype #(in) requirederror #(rn) minelemsize #(rn)`
  - `regionskipmap` parameter allows to skip some regions. The error is not evaluated in these regions and default mesh density is used. The size of this array should be equal to number of regions and nonzero entry indicates region to skip.
  - `normtype` Allows select the type of norm used in evaluation of error. Default value is to use L2 norm (equal to 0), value equal to 1 uses the energy norm.
  - `requirederror` parameter determines the required error to obtain (in percents/100).
  - minelemsize parameter allows to set minimum limit on element size.
- EET_CZZSI - combination of parameters for EET_SEI and EET_ZZEE; the in elastic regions are driven using EET_SEI, the elastic are driven by EET_ZZEE.

6.4 Material interfaces

The material interfaces are used to represent and track the position of various interfaces on fixed grids. Typical examples include free surface, evolving interface between two materials, etc. Available representations include:

MI	miflag	Compatibility
LEPlic	0	2D triangular
LevelSet	1	2D triangular

- LEPlic- representation based on Volume-Of-Fluid approach; the initial distribution of VOF fractions should be specified for each element (see element manual)
  `[refvol #(rn)]`
  - parameter `refvol` allows to set initial volume of reference fluid, then the reference volume is computed in each step and printed, so the accuracy and mass conservation can be monitored.
- LevelSet- level set based representation
  `levelset #(ra)` OR `refmatpolyx #(ra) refmatpolyy #(ra)`
  `[lsra #(in)][rdt #(rn)][rerr #(rn)]`
  - `levelset` allows to specify the initial level set values for all nodes directly. The size should be equal to total number of nodes within the domain.

- Parameters `refmatpolyx` and `refmatpolyy` allow to initialize level set by specifying interface geometry as 2d polygon. Then polygon describes the initial zero level set, and level set values are then defined as signed distance from this polygon. Positive values are on the left side when walking along polygon. The parameter `refmatpolyx` specifies the x-coordinates of polygon vertices, parameter `refmatpolyy` y-corrdinates. Please note, that level set must be initialized, either using `levelset` parameter or using `refmatpolyx` and `refmatpolyy`.
- Parameter `lsra` allows to select level set reinitialization algorithm. Currently supported values are 0 (no re-initialization), 1 (re-initializes the level set representation by solving $d_\tau = S(\phi)(1 - |\nabla d|)$ to steady state, default), 2 (uses fast marching method to build signed distance level set representation).
- Parameters `rdt rerr` are used to control reinitialization algorithm for `lsra` = 0. `rdt` allows to change time step of integration algorithm and parameter `rerr` allows to change default error limit used to detect steady state.

6.5 Mesh generator interfaces

The mesh generator interface is responsible to provide a link to specific mesh generator. The supported values of `meshpackage` parameter are

- MPT_T3D: `meshpackage` = 0. T3d mesh interface. Default. Supports both 1d, 2d (triangles) and 3d (tetrahedras) meshes. Reliable.
- MPT_TARGE2: `meshpackage` = 1. Interface to Targe2 2D mesh generator.
- MPT_SUBDIVISION: `meshpackage`=3. Built-in subdivision algorithm. Supports triangular 2D and tetrahedral 3D meshes. Can operate in parallel mode.

6.6 Initialization modules

Initialization modules allow to initialize the state variables using data previously computed by external software. The number of initialization module records is specified in analysis record using `ninitmodules` parameter (see the initial part of section *AnalysisRecord*. The general format is the following:

`EntType initfile #(string)` The file name following the keyword "initfile" specifies the path to the file that contains the initialization data and should be given without quotes.

Currently, the only supported initialization module is

- Gauss point initialization module

`GPInitModule initfile #(string)`

- Each Gauss point is represented by one line in the initialization file.
- The Gauss points should be given in a specific order, based on the element number and the Gauss point number, in agreement with the mesh specified in later sections.

- Each line referring to a Gauss point should contain the following data:

`elnum #(in) gpnum #(in) coords #(ra) ng #(in) var_1_id #(in) values_1 #(ra) ... var_ng_id #(in) values_ng #(ra)`

- `elnum` is the element number
- `gpnum` is the Gauss point number
- `coords` are the coordinates of the Gauss point
- `ng` is the number of groups of variables that will follow
- **`var_1_id` is the identification number of variable group number** 1 (according to the definitions in internalstatetype.h)
- `values_1` are the values of variables in group number 1
- `var_ng_id` is the identification number of variable group number ng
- `values_ng` are the values of variables in group number ng

- Example:

  `37 4 3 0.02 0.04 0.05 3 52 1 0.23 62 1 0.049 1 6 0 -2.08e+07 0 0 0 0` means that Gauss point number 4 of element number 37 has coordinates $x = 0.02$, $y = 0.04$ and $z = 0.05$ and the initial values are specified for 3 groups of variables; the first group (variable ID 52) is of type IST_DamageScalar (see internalstatetype.h) and contains 1 variable (since it is a scalar) with value 0.23; the second group (ID 62) is of type IST_CumPlasticStrain and contains 1 variable with value 0.049; the third group is of type IST_StressTensor and contains 6 variables (stress components σ_x, σ_y, etc.) with values 0, -2.08e+07, 0, 0, 0, 0

6.7 Export modules

Export modules allow to export computed data into external software for post-processing. The number of export module records is specified in analysis record using `nmodules` parameter (see the initial part of section *AnalysisRecord*. The general format is the following:

`EntType [tstep_all] [tstep_step #(in)] [tsteps_out #(rl)] [subtsteps_out #(in)] [domain_all] [domain_mask #(in)]`

To select all solution steps, in which output will be performed, use `tstep_all`. To select each `tstep_step`-nth step, use `tstep_step` parameter. In order to select only specific solution steps, the `tsteps_out` list can be specified, supplying solution step number list in which output will be done. To select output for all domain of the problem the `domain_all` keyword can be used. To select only specific domains, `domain_mask` array can be used, where the values of the array specify the domain numbers to be exported. If the parameter `subtsteps_out` = 1, it turns on the export of intermediate results, for example during the substepping or individual equilibrium iterations. This option requires support from the solver.

Currently, the supported export modules are following

- VTK export, **DEPRECATED - Use VTKXML**

  `vtk [vars #(ia)] [primvars #(ia)] [cellvars #(ia)] [stype #(in)] [regionstoskip #(ia)]`

  `vtkxml [vars #(ia)] [primvars #(ia)] [cellvars #(ia)] [ipvars #(ia)] [stype #(in)] [regionsets #(ia)] [timeScale #(rn)]`

  - The vtk module is obsolete, use vtkxml instead. Vtkxml allows to export results recovered on region by region basis and has more features.

- The array `vars` contains identifiers for those internal variables which are to be exported. These variables will be smoothed and transfered to nodes. The id values are defined by InternalStateType enumeration, which is defined in include file "src/oofemlib/internalstatetype.h".
- The array `primvars` contains identifiers of primary variables to be exported. The possible values correspond to the values of enumerated type UnknownType, which is again defined in "src/oofemlib/unknowntype.h". Please note, that the values corresponding to enumerated type values start from zero, if not specified directly and that not all values are supported by particular material model or analysis type.
- The array `cellvars` contains identifiers of constant variables defined on an element (cell), e.g. a material number. Identifier numbers are specified in "src/oofemlib/internalstatetype.h".
- The array `ipvars` contains identifiers for those internal variables which are to be exported. These variables will be directly exported (no smoothing) as point dataset, where each point corresponds to individual integration point. A separate vtu file for these raw, point data will be created. The id values are defined by InternalStateType enumeration, which is defined in include file "src/oofemlib/internalstatetype.h".
- The parameter `stype` allows to select smoothing procedure for internal variables, which is used to compute nodal values from values in integration points. The supported values are 0 for simple nodal averageing (generally supported only by triangular and tetrahedral elements), 1 for Zienkiewicz Zhu recovery (default), and 2 for Superconvergent Patch Recovery (SPR, based on least square fitting).
- The export is done on region basis, on each region, the nodal recovery is performed independently and results are exported in a separate piece. This allows to take into account for discntinuities, or to export variables defined only by particular material model. The region volumes are defined using sets containing individual elements. By default the one region is created, containing all element in the problem domain. The optional parameter `regionsets` allows to use user-defined. The individual values refer to numbers (ids) of domain sets. Note, that regions are determined solely using elements.

  vtkxml tstep_all cellvars 1 46 vars 1 1 primvars 1 1 stype 2 regionsets 2 1 2
- `timeScale` scales time in output. In transport problem, basic units are seconds. Setting timeScale = 2.777777e-4 (=1/3600.) converts all time data in vtkXML from seconds to hours.

By default vtk and vtkxml modules perform recovery over the whole domain. The VTKXML module can operate in region-by-region mode (see `nvr` and `vrmap` parameters). In this case, the smoothing is performed only over particular virtual region, where only elements in this virtual region participate.

- Homogenization of IP quantities in the global coordinate system (such as stress, strain, damage, heat flow). Corresponding IP quantities are summed and averaged over the volume. It is possible to select region sets from which the averaging occurs. The averaging works for all domains with an extension to trusses. A truss is considered as a volume element with oriented stress and strain components along the truss axis. The transformation to global components occurs before averaging.

  `hom ists #(ia) [scale #(rn)][regionSets #(ia)][strain_energy]`
  - An integer array `ists` specifies internal state types for export which are defined in internalstatetype.h file.
  - The parameter `scale` multiplies all averaged IP quantities. `scale`=1 by default.
  - An integer array `regionSets` specifies region sets for averaging. All domain is averaged by default.
  - `strain_energy` calculates strain energy over selected elements (defined by sets) by

$$W^* = \int_V \int \sigma \mathrm{d}(\varepsilon - \varepsilon_{eig}) \mathrm{d}V$$

    where σ is the stress tensor, ε stands for the strain tensor and ε_{eig} is eigenstrain tensor (originates from temperature load or prescribed eigenstrain). Strain energy increment and total strain energy is reported in each step. The integration uses mid-point rule for stress and yields exact results for linear elastic materials.

- Gauss point export is useful if one needs to plot a certain variable (such as damage) as a function of a spatial coordinate using tools like gnuplot. It generates files with data organized in columns, each row representing one Gauss point. In this way, one can plot e.g. the damage distribution along a one-dimensional bar. `gpexportmodule [vars #(ia)] [ncoords #(in)]`
  - The array `vars` contains identifiers for those internal variables which are to be exported. The id values are defined by InternalStateType enumeration, which is defined in include file "src/oofemlib/internalstatetype.h".
  - Parameter `ncoords` specifies the number of spatial coordinates to be exported at each Gauss point. Depending on the spatial dimension of the domain, the points can have one, two or three coordinates. If `ncoords` is set to -1, only those coordinates that are actually used are exported. If `ncoords` is set to 0, no coordinates are exported. If `ncoords` is set to a positive integer, exactly `ncoords` coordinates are exported. If `ncoords` exceeds the actual number of coordinates, the actual coordinates are supplemented by zeros. For instance, if we deal with a 2D problem, the actual number of coordinates is 2. For `ncoords`=3, the two actual coordinates followed by 0 will be exported. For `ncoords`=1, only the first coordinate will be exported.

  The Gauss point export module creates a file with extension "gp" after each step for which the output is performed. This file contains a header with lines starting by the symbol #, followed by the actual data section. Each data line corresponds to one Gauss point and contains the following data:

  1. element number,
  2. material number,
  3. Gauss point number,
  4. contributing volume around Gauss point,
  5. Gauss point global coordinates (written as a real array of length `ncoords`),
  6. internal variables according to the specification in `vars` (each written as a real array of the corresponding length).

  Example: `GPExportModule 1 tstep_step 100 domain_all ncoords 2 vars 5 4 13 31 64 65`

  means that the *.gp file will be written after each 100 steps and will contain for each of the Gauss points in the entire domain its 2 coordinates and also internal variables of type 4, 13, 31, 64 and 65, which are the strain tensor, damage tensor, maximum equivalent strain level, stress work density and dissipated work density. Of course, the material model must be able to deliver such variables. The size of the strain tensor depends on the spatial dimension, and the size of the damage tensor depends on the spatial dimension and type of model (e.g., for a simple isotropic damage model it will have just 1 component while for an anisotropic damage model it may have more). The other variables in this example are scalars, but they will be written as arrays of length 1, so the actual value will always be preceded by "1" as the length of the array. Since certain internal variables have the meaning of densities (per unit volume or area, again depending on the spatial dimension), it is useful to have access to the contributing volume of the Gauss point. The product of this contributing volume and the density gives an additive contribution to the total value of the corresponding variable. This can be exploited e.g. to evaluate the total dissipated energy over the entire domain.

CHAPTER

SEVEN

EXAMPLES

7.1 Beam structure

This example for a simple beam structure gives basic overview of the input file (found under tests/sm/beam2d_1.in). Structure geometry and its constitutive and geometrical properties are shown in Fig. (*ex01*). The linear static analysis is required, the influence of shear is neglected.

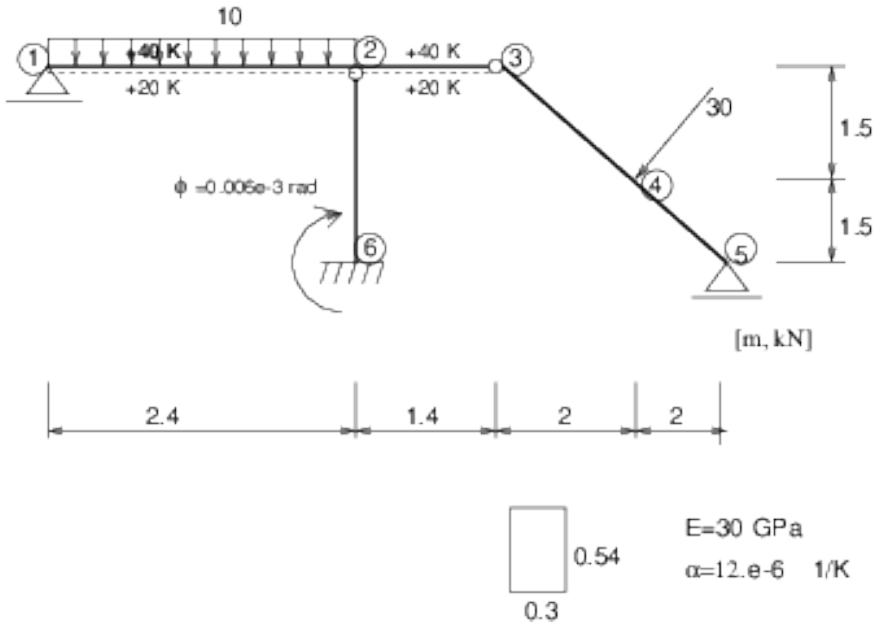

Fig. 1: Example 1 - beam2d_1.in

```
beam2d_1.out
Simple Beam Structure - linear analysis
#only momentum influence to the displacements is taken into account
#beamShearCoeff is artificially enlarged.
StaticStructural nsteps 3 nmodules 0
domain 2dBeam
OutputManager tstep_all dofman_all element_all
ndofman 6 nelem 5 ncrosssect 1 nmat 1 nbc 6 nic 0 nltf 3 nset 7
node 1 coords 3 0.  0.  0.
node 2 coords 3 2.4 0.  0.
node 3 coords 3 3.8 0.  0.
node 4 coords 3 5.8 0.  1.5
```

(continues on next page)

(continued from previous page)

```
node 5 coords 3 7.8 0.  3.0
node 6 coords 3 2.4 0.  3.0
Beam2d 1 nodes 2 1 2
Beam2d 2 nodes 2 2 3 DofsToCondense 1 6
Beam2d 3 nodes 2 3 4 DofsToCondense 1 3
Beam2d 4 nodes 2 4 5
Beam2d 5 nodes 2 6 2 DofsToCondense 1 6
SimpleCS 1 area 1.e8 Iy 0.0039366 beamShearCoeff 1.e18 thick 0.54 material 1 set 1
IsoLE 1 d 1. E 30.e6 n 0.2 tAlpha 1.2e-5
BoundaryCondition 1 loadTimeFunction 1 dofs 1 3 values 1 0.0 set 4
BoundaryCondition 2 loadTimeFunction 1 dofs 1 5 values 1 0.0 set 5
BoundaryCondition 3 loadTimeFunction 2 dofs 3 1 3 5 values 3 0.0 0.0 -0.006e-3 set 6
ConstantEdgeLoad 4 loadTimeFunction 1 Components 3 0.0 10.0 0.0 loadType 3 set 3
NodalLoad 5 loadTimeFunction 1 dofs 3 1 3 5 Components 3 -18.0 24.0 0.0 set 2
StructTemperatureLoad 6 loadTimeFunction 3 Components 2 30.0 -20.0 set 7
PeakFunction 1 t 1.0 f(t) 1.
PeakFunction 2 t 2.0 f(t) 1.
PeakFunction 3 t 3.0 f(t) 1.
Set 1 elementranges {(1 5)}
Set 2 nodes 1 4
Set 3 elementedges 2 1 1
Set 4 nodes 2 1 5
Set 5 nodes 1 3
Set 6 nodes 1 6
Set 7 elements 2 1 2
```

7.2 Plane stress example

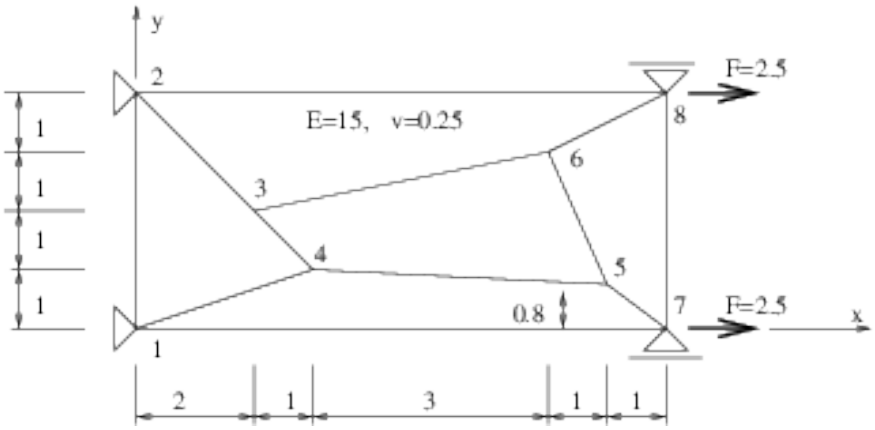

Fig. 2: Example 2

```
patch100.out
Patch test of PlaneStress2d elements -> pure compression
LinearStatic nsteps 1
domain 2dPlaneStress
OutputManager tstep_all dofman_all element_all
ndofman 8 nelem 5 ncrosssect 1 nmat 1 nbc 3 nic 0 nltf 1 nset 3
node 1 coords 3  0.0   0.0   0.0
node 2 coords 3  0.0   4.0   0.0
```

(continues on next page)

(continued from previous page)

```
node 3 coords 3  2.0   2.0   0.0
node 4 coords 3  3.0   1.0   0.0
node 5 coords 3  8.0   0.8   0.0
node 6 coords 3  7.0   3.0   0.0
node 7 coords 3  9.0   0.0   0.0
node 8 coords 3  9.0   4.0   0.0
PlaneStress2d 1 nodes 4 1 4 3 2  NIP 1
PlaneStress2d 2 nodes 4 1 7 5 4  NIP 1
PlaneStress2d 3 nodes 4 4 5 6 3  NIP 1
PlaneStress2d 4 nodes 4 3 6 8 2  NIP 1
PlaneStress2d 5 nodes 4 5 7 8 6  NIP 1
Set 1 elementranges {(1 5)}
Set 2 nodes 2 1 2
Set 3 nodes 2 7 8
SimpleCS 1 thick 1.0 width 1.0 material 1 set 1
IsoLE 1 d 0. E 15.0 n 0.25 talpha 1.0
BoundaryCondition 1 loadTimeFunction 1 dofs 2 1 2 values 1 0.0 set 2
BoundaryCondition 2 loadTimeFunction 1 dofs 1 2 values 1 0.0 set 3
NodalLoad 3 loadTimeFunction 1 dofs 2 1 2 components 2 2.5 0.0 set 3
ConstantFunction 1 f(t) 1.0
```

7.3 Examples - parallel mode

7.3.1 Node-cut example

The example shows explicit direct integration analysis of simple structure with two DOFs. The geometry and partitioning is sketched in fig.(*nodecut-ex01*).

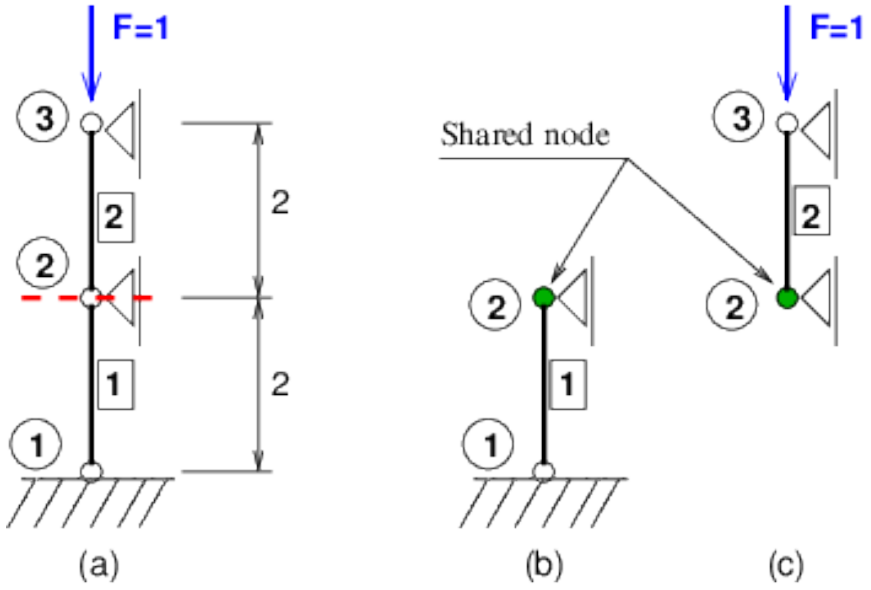

Fig. 3: Node-cut partitioning example: (a) whole geometry, (b) partition 0, (c) partition 1.

```
#
# partition 0
```

(continues on next page)

(continued from previous page)

```
#
partest.out.0
Parallel test of explicit oofem computation
#
NlDEIDynamic nsteps 3 dumpcoef 0.0 deltaT 1.0
domain 2dTruss
#
OutputManager tstep_all dofman_all element_all
ndofman 2 nelem 1 ncrosssect 1 nmat 1 nbc 3 nic 0 nltf 1 nset 4
#
Node 1 coords 3 0. 0. 0.
Node 2 coords 3 0. 0. 2. Shared partitions 1 1
Truss2d 1 nodes 2 1 2
Set 1 elements 1 1
Set 2 nodes 2 1 2
Set 3 nodes 1 1
Set 4 nodes 0
SimpleCS 1 thick 0.1 width 10.0 material 1 set 1
IsoLE 1 tAlpha 0.000012 d 10.0 E 1.0 n 0.2
BoundaryCondition 1 loadTimeFunction 1 dofs 1 1 values 1 0.0 set 2
BoundaryCondition 2 loadTimeFunction 1 dofs 1 3 values 1 0.0 set 3
NodalLoad 3 loadTimeFunction 1 dofs 2 1 3 components 2 0. 1.0 set 4
ConstantFunction 1 f(t) 1.0

#
# partition 1
#
partest.out.1
Parallel test of explicit oofem computation
#
NlDEIDynamic nsteps 3 dumpcoef 0.0 deltaT 1.0
domain 2dTruss
#
OutputManager tstep_all dofman_all element_all
ndofman 2 nelem 1 ncrosssect 1 nmat 1 nbc 3 nic 0 nltf 1 nset 4
#
Node 2 coords 3 0. 0. 2. Shared partitions 1 0
Node 3 coords 3 0. 0. 4.
Truss2d 2 nodes 2 2 3
Set 1 elements 1 2
Set 2 nodes 2 2 3
Set 3 nodes 0
Set 4 nodes 1 3
SimpleCS 1 thick 0.1 width 10.0 material 1 set 1
IsoLE 1 tAlpha 0.000012 d 10.0 E 1.0 n 0.2
BoundaryCondition 1 loadTimeFunction 1 dofs 1 1 values 1 0.0 set 2
BoundaryCondition 2 loadTimeFunction 1 dofs 1 3 values 1 0.0 set 3
NodalLoad 3 loadTimeFunction 1 dofs 2 1 3 components 2 0. 1.0 set 4
ConstantFunction 1 f(t) 1.0
```

7.3.2 Element-cut example

The example shows explicit direct integration analysis of simple structure with two DOFs. The geometry and partitioning is sketched in fig. (*nodecut-ex01*).

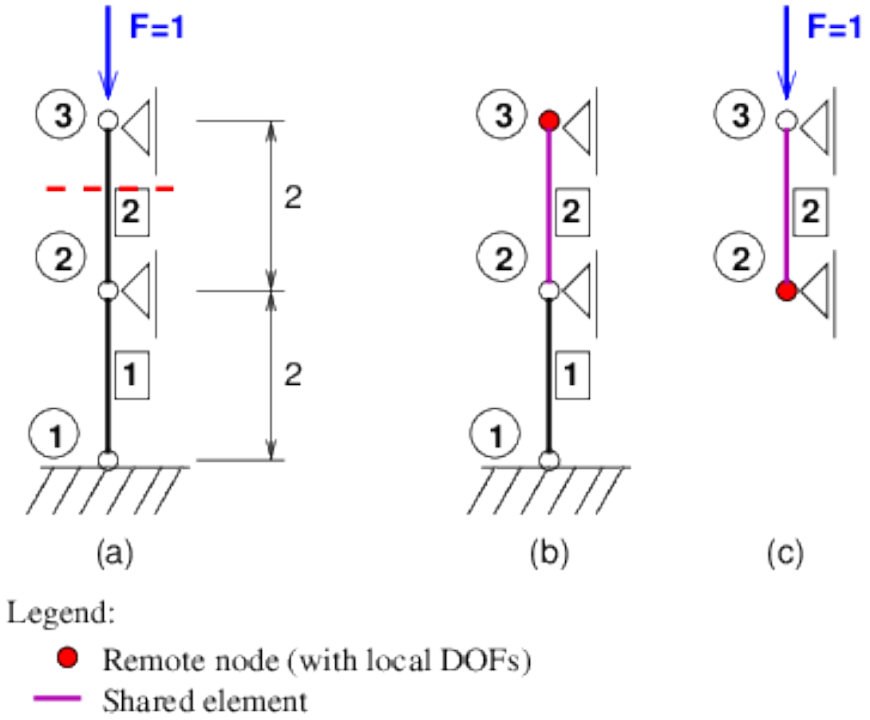

Fig. 4: Element-cut partitioning example: (a) whole geometry, (b) partition 0, (c) partition 1.

```
#
# partition 0
#
partest2.out.0
Parallel test of explicit oofem computation
#
NlDEIDynamic nsteps 5 dumpcoef 0.0 deltaT 1.0
domain 2dTruss
#
OutputManager tstep_all dofman_all element_all
ndofman 3 nelem 2 ncrosssect 1 nmat 1 nbc 3 nic 0 nltf 1 nset 4
#
Node 1 coords 3 0. 0. 0.
Node 2 coords 3 0. 0. 2.
Node 3 coords 3 0. 0. 4. Remote partitions 1 1
Truss2d 1 nodes 2 1 2
Truss2d 2 nodes 2 2 3
Set 1 elements 2 1 2
Set 2 nodes 3 1 2 3
Set 3 nodes 1 1
Set 4 nodes 1 3
SimpleCS 1 thick 0.1 width 10.0 material 1 set 1
IsoLE 1 tAlpha 0.000012 d 10.0 E 1.0 n 0.2
BoundaryCondition 1 loadTimeFunction 1 dofs 1 1 values 1 0.0 set 2
BoundaryCondition 2 loadTimeFunction 1 dofs 1 3 values 1 0.0 set 3
NodalLoad 3 loadTimeFunction 1 dofs 2 1 3 components 2 0. 1.0 set 4
ConstantFunction 1 f(t) 1.0
```

(continues on next page)

(continued from previous page)

```
#
# partition 1
#
partest2.out.1
Parallel test of explicit oofem computation
#
NlDEIDynamic nsteps 5 dumpcoef 0.0 deltaT 1.0
domain 2dTruss
#
OutputManager tstep_all dofman_all element_all
ndofman 2 nelem 1 ncrosssect 1 nmat 1 nbc 3 nic 0 nltf 1 nset 4
#
Node 2 coords 3 0. 0. 2 Remote partitions 1 0
Node 3 coords 3 0. 0. 4
Truss2d 2 nodes 2 2 3
Set 1 elements 1 2
Set 2 nodes 2 2 3
Set 3 nodes 0
Set 4 nodes 1 3
SimpleCS 1 thick 0.1 width 10.0 material 1 set 1
IsoLE 1 tAlpha 0.000012 d 10.0 E 1.0 n 0.2
BoundaryCondition 1 loadTimeFunction 1 dofs 1 1 values 1 0.0 set 2
BoundaryCondition 2 loadTimeFunction 1 dofs 1 3 values 1 0.0 set 3
NodalLoad 3 loadTimeFunction 1 dofs 2 1 3 components 2 0. 1.0 set 4
ConstantFunction 1 f(t) 1.0
```

7.4 Figures

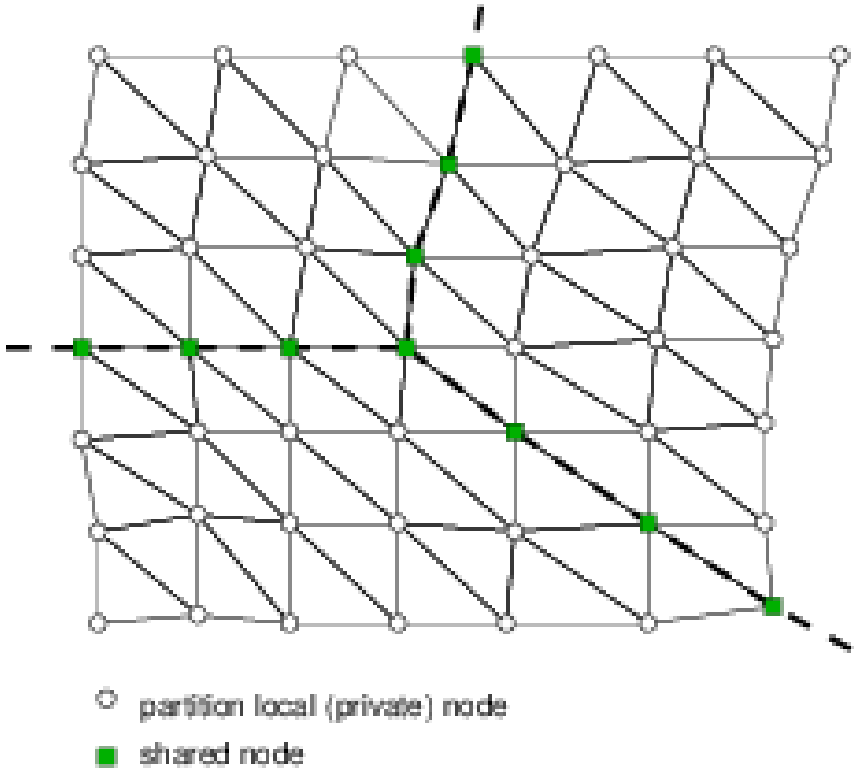

Fig. 5: Node-cut partitioning.

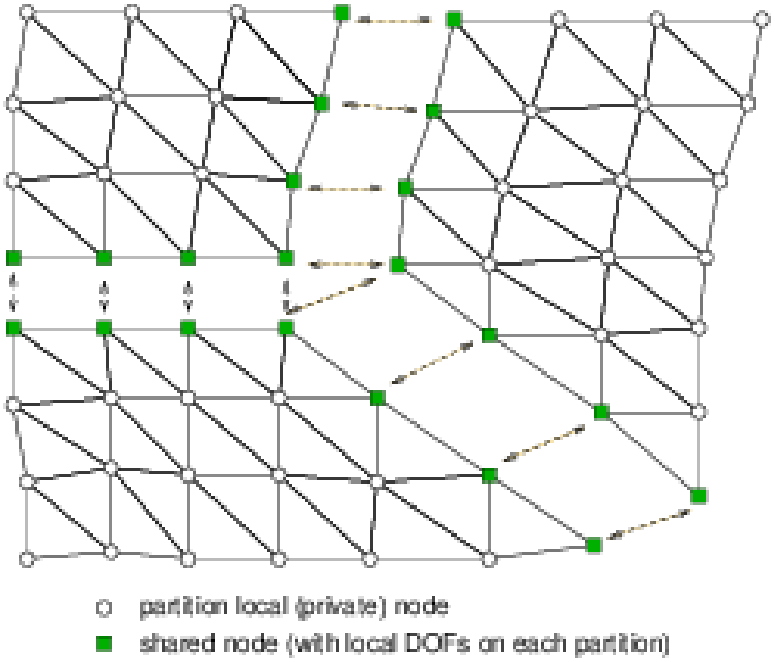

Fig. 6: Node-cut partitioning - local constitutive mode.

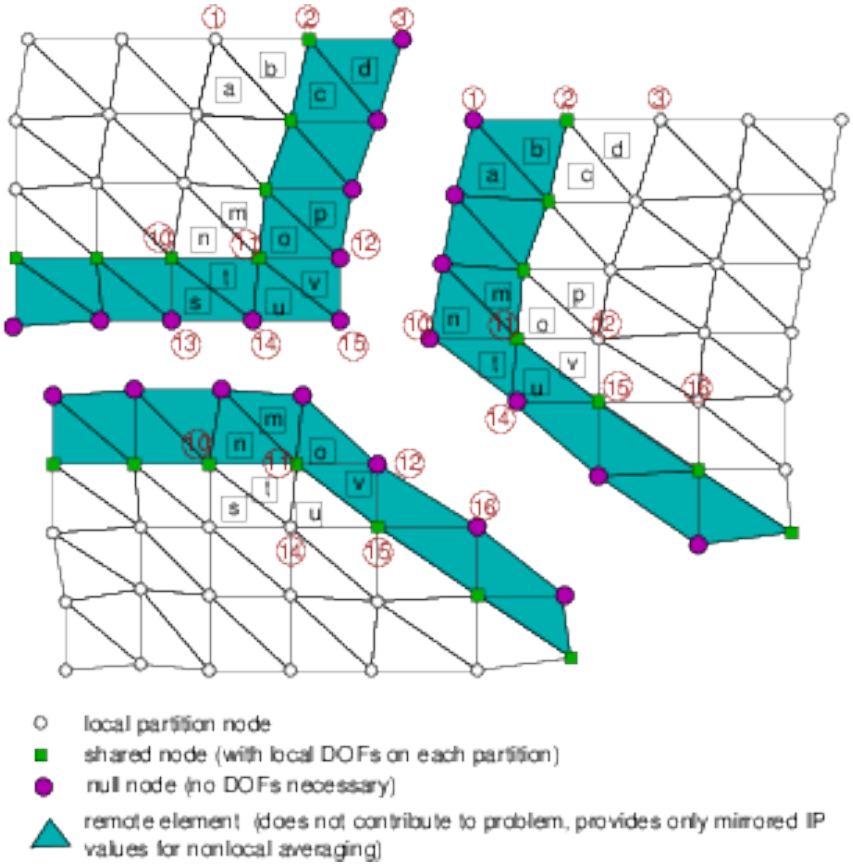

Fig. 7: Node-cut partitioning - nonlocal constitutive mode.

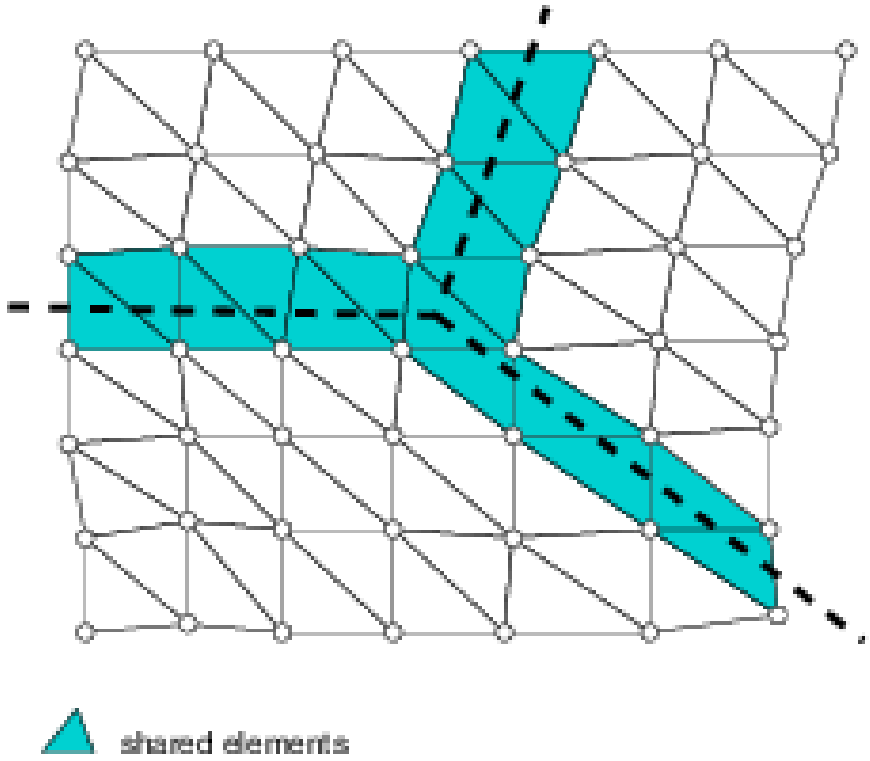

Fig. 8: Element-cut partitioning.

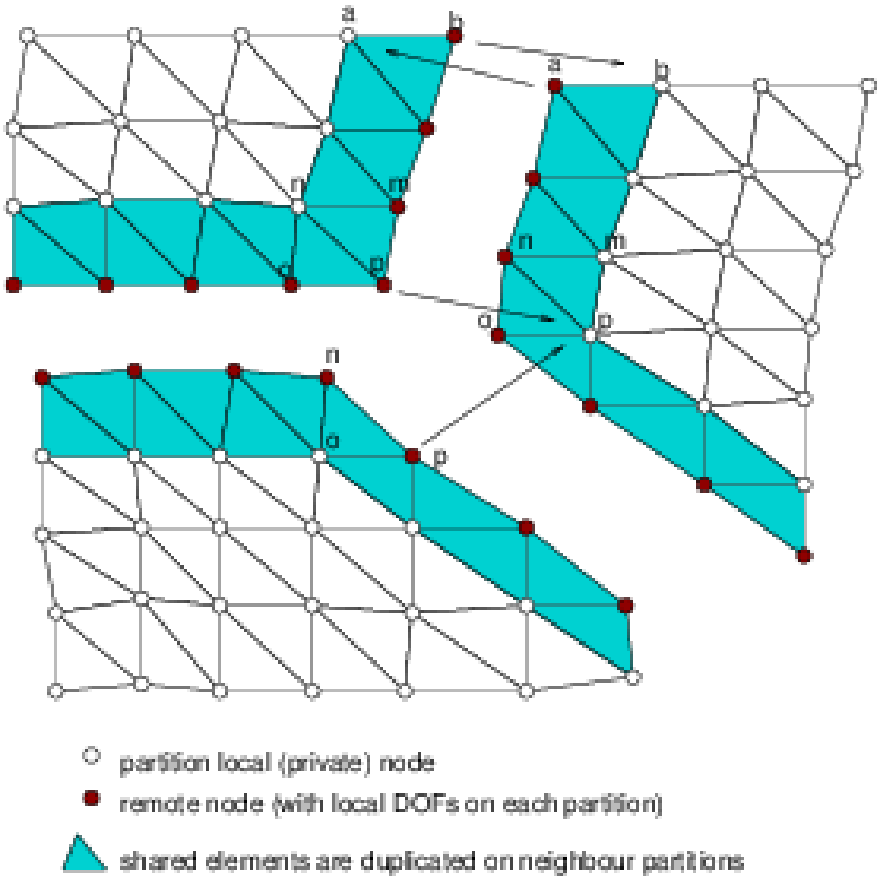

Fig. 9: Element-cut partitioning, local constitutive mode.

CHAPTER

EIGHT

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 wisit www.oofem.org