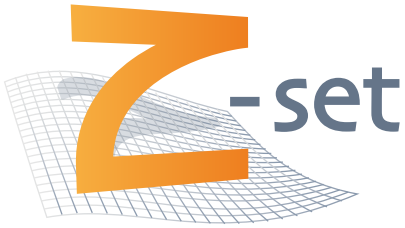
Material & Structure Analysis Suite



User commands
Version 9.1

Z-set software is distributed by

Transvalor S.A.
Centre d'affaires La Boursidière
rue de le Boursidière
Bâtiment Q, 1er étage
92350 Le Plessis-Robinson
France

http://www.zset-software.com
support@zset-software.com

Contents

Chapter 1

Introduction

Z-set user commands

Description:
This Z-set user-commands handbook covers the command syntax and some of the details for the different files involved with basic use of the FEA related functionality.

Handbook Summary:
The following list summarizes the documentation for all of Z-set. As part of our 8.2/8.3 developments, greatly expanding the software documentation is one of our primary goals.

The list below is sorted in what we feel would be an appropriate sequence for the normal user, starting with installation and reviewing capabilities, to creating input files and eventually scripting and developing add-ons to the software.

Release Notes/Zmaster The basic overview of the software, installation instructions, and documentation for the graphical user interface Zmaster on all platforms. The Zmaster manual also now covers all the base reference chapters such as environment variables, user parameters, function reference, command line programs, etc.

Examples/Training This book is essentially the "getting started" documentation for the software. The book describes Z-mat, simulation, optimization, material models, and the FEA code use. The examples cover setup of complete models, and are meant to demonstrate the capabilities with relatively simple examples.

Z-mat User commands This summarizes the command file formats and capabilities of the Z-mat interface, simulation, optimization and material files for all of Z-mat and Z-set.

Z-set User commands This summarizes the command file formats for the FEA related capabilities including meshing, FEA solution, post processing, etc.

Developer A guide to the user-extensible features of the software, including scripting, ZebFront material model development, making plugins, and the C++ programming API.

Plugins A guide to add-on features available.

Theory Theory manual covering formulations (under development).

Conventions

This page summarizes the conventions used for the Z-set input files. An overview of the general command syntax (command hierarchies) is given in the beginning of the *Examples/Training* manual.

- Running of Z-set modules generally requires that a "problem name" be given. Most input and output data files are based on this name with a variety of suffixes attached. Henceforth, *problem* will often be used to indicate the problem name given while running the commands.
- The characters `%` and `#` indicate that the rest of the line is a comment. For example:
  ```
  ***load % external problem loading
  ```
- There are no abbreviations allowed in the use of keywords. All keywords and command names must be written entirely.
- The admissible characters for the names of user variables are: `a-z, A-Z, ", +, -, *, ., =, /, _, ), (, \, ~`
- The text entry is *always* case-sensitive.
- The use of braces `[ ]` in the syntax descriptions indicates an option with a default definition.
- All parameter values used in the input files and described as "real" in the syntax descriptions must have a decimal point. All standard specifications of floating point values are accepted. Two examples of real values are: `3.0  4.2e-5`
- The use of parenthesis `()` indicates data input of real values in vector form. An example is: `(0.1 0.2 1.0)`. In most cases the size of such vectors must be compatible with the overall problem dimension.

The symbol indicates a section where the calculation is sensitive to input data or format.

Chapter 2

Mesher

Introduction

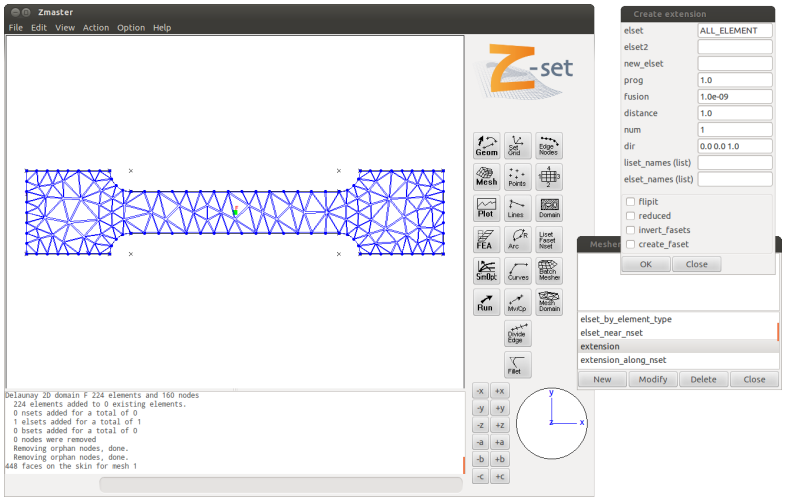
Description:
The *Mesher* module is provided in order to perform batch mesh manipulations and combinations to pre-existing meshes, and import and export Z-set format files[1]. These operations supersede the meshing utility programs previously supplied with Zebulon. The normal mode of operation is as a batch process launched with the command `Zrun -m` *prob*. The utilities are however also supported in some other applications, such as the batch mesher commands of the Zmaster *geom* and *mesh* menus.

An important thing to remember is that the ASCII file represents the objects and configuration options for a particular calculation. The computational modules are then run sequentially in order to modify or generate the mesh. This is therefore the configuration of an object system, and not the activation of repeated "database" manipulation commands.

Batch mode:
For batch operation, an ASCII input file is submitted to the Zrun program to manipulate and combine any number of meshes. The operation will generate more files depending on the options chosen. The most common use is to generate a new ASCII `.geof` or binary `.geo` file for use in a FEA calculation.

See the syntax description for `****mesher` below (page 2.4) for a detailed overview of the mesher syntax, with some examples.

Running in Zmaster:
The batch mesh operations in Zmaster operate on the current mesh (accessed under the *Mesh* command, or act on a newly generated mesh after all the Zmaster domains are generated (in the *Geom* command). This later operation happens after regenerating the mesh with the *Mesh Domain* button. The figure below shows the dialogs which get opened after selecting the *Batch Mesher Create* button in the geometry toolbar.



[1] example problems using the mesher can be found in `$Z7PATH/test/Mesher_test/INP` and in `$Z7PATH/test/Zmaster/MESHER`

The operation of the graphical interface for running batch meshers is described more completely in the Zmaster manual. Here we want to remark on the connection between the command file syntax described in this chapter and the dialog creation.

The available meshers are listed in the bottom list box of the *Mesher Create* dialog with the exact names of the `**`-level command option in this chapter. The `***` level commands are not available in Zmaster, so the operation always applies to the currently loaded mesh. In the figure above, the mesher option `**extension` (see page 2.61) is selected. Selecting either *New* or *Modify* will open up an edit dialog with the options equivalent to the `*`-level options of the selected mesher. The option data fields can then be typed directly into the dialog box entries.

User extensions:
Because there are in fact many many things one may wish to do to a mesh in the course of building a model, the interface for adding new mesh transformation commands is especially flexible and easy to program. The details of how to create a mesher are described in the separate *Z-set Developer* handbook, with example code in the distribution `$Z7PATH/User-project` directory.

****mesher

Description:

The basic syntax for meshing operations follows the same logic as other parts of the program. Meshing blocks are started with the four asterisk command `****mesher`. The mesher can load different portions of the final mesh as independent meshes. The successful mesher will therefore learn to create a series of meshing operations, modifications, and unions for a final mesh.

Note:
There is a modified syntax for the mesher which can be used by Zmaster in the `.mast` *file. The* `****master` *command accepts the sub-command* `***mesher` *which indicates that transformation commands from this chapter are to be applied every time the mesh button is pressed. One can move those commands outside the Zmaster program by changing the command to* `****mesher` *and including a* `***mesh/**open` *group.*

Syntax:
The major meshing commands are as follows:

```
****mesher
[  ***delete_file fname  ]
[  ***global_parameter  param-definitions  ]
[  ***shell shell-command  ]
[  ***function_declarations functions  ]
[  ***mesh name ]
  ....
```

`***delete_file` This allows entry of file names to delete before running the meshers. Usually this will be the new mesh file name to ensure a clean state before running.

`***function_declarations` This command allows entry of function declarations ahead of the mesher applications, so that complex pre-defined functions can be used where function selection is allowed (e.g. nsets and so on). See pages 3.144 and 6.2.

`***global_parameter` This base-level command allows setting of global parameter definitions in the mesher input to customize behavior. See the reference section in the Release/Zmaster manual.

`***mesh` This command opens a new mesh object which will be the subject of meshing operations. Use the `**open` command to load an existing mesh in for operations. Any number of `**`-level transformation commands can then be applied to the mesh before the next `***`-level command. At the occurrence of the next `***`-level command, the mesh will be saved to disk with the problem name given as a parameter to `***mesh`. Following `***mesh/**open` operations can re-open the mesh for further processing.

`***shell` This command can be used to launch external programs to generate sub-meshes or other operations. Probably the most useful purpose for this command is to run a Zmaster batch job. If the global parameter Error.HandlingShellError is set to 1, a non 0 system exit of this command will result in a Z-set error. By default Error.HandlingShellError is set to 0 and if the command invoked results in an error, Z-set will continue without raising any error.

Example:
An example of the mesher commands as output by Zmaster is as follows:

```
% in prob.mast
****mesher
  **extension
   *elset    face
   *distance 3.000000e+00
   *num      1
****return
```

These lines will make the element set `face` (which was created using one of the domain tools) to be extended into a 3d mesh every time the mesh-domains button is pressed. The mesher commands could be moved into the `.inp` file to separate the process:

```
%
% in prob.inp.. importing prob_2d.geof a,d applying an
% extension into 3d.
%
****mesher
 ***mesh prob
  **open prob_2d.geof
  **extension
   *elset    face
   *distance 3.000000e+00
   *num      1
****return
```

Example:
One feature of the batch mesher is the ability to step through multiple meshing steps in the same file. For example a multi-part mesh can be generated using a combination of Zmaster batch jobs, and mesher transformers and unions:

```
****mesher
 ***shell Zrun -B part1.mast   % generate a new, 2d part1.geof
 ***mesh  part1                % start creating the 3d, also part1.geof
  **open  part1.geof           % open the 2d created from Zmaster batch
  **extension                  % now start extensions
    *elset ALL_ELEMENT
    *new_elset solid
    *prog 1
    *distance 0.5
    *num 2
    *dir (0 0 1 )
  **nset z=0
    *plane 0. 0. 1. 0.
    *limit 0.001
 ***shell Zrun -B part2.mast   % a ***-level command closes the prev. mesh
 ***mesh part2                 % start the part2 mesh independently
  **open part2.geof            % and in the same way as above. This lets
   %                           % us generate complex meshes in 1 run
   %  etc
   %
****return
```

***mesh

Description:
As introduced before, this command indicates the start of a new mesh object. The meshing operations which follow will act on the mesh object, until the next ***-level command is reached. At that point the mesh will be saved with the given name. Use of this command invariably involves use of the **open or **import commands.

Syntax:
The basic ***mesh section syntax is summarized below:

```
***mesh [output-name]
 **export format  output-name
 **open input-name
 **open_mast mast-file.mast
 **import format  input-name
 **inp_file
 **output output-name
 **dont_save_final_mesh
 **transform
   ...
```

The mesh output name *output-name* is the problem name, and therefore does not have the .geof suffix. It is optionally included on the same line as ***mesh for convenience.

**export Write an input file in a format other than native Z-set.

**import Open a mesh file in a format other than the native Z-set .geof format. The *format* type is a keyword for one of the external mesh formats allowed.

**inp_file specify that the input file (FEA loads, BCs element formulation, time sequences, etc) are to be read and output with the mesh. This is probably desired in order to get as *full* an import or export as possible. Note: this command is provided as a convenience feature, and does not imply full compatibility between formats. It is likely the different codes options just do not have an equivalent for translation, and will be passed silently. *Be careful as well not to overwrite existing hand-edited input files!* – there are no questions before output is written.

**open Open a file defining the mesh which will be operated on. There must be only one instance of open in the ***mesh block. The file name is the problem name without the .geof suffix. Additional mesh components can be loaded using the **union command described on page 2.141.

**open_mast Open a Zmaster .mast file, running Zmaster batch on it beforehand. This open mode actually does a union, but normally it would be used in place of an **open or **import command.

**output specify the output file.

**dont_save_final_mesh disables writing the mesh, as it normally is the final stage of the meshing process.

. . . *continued*

Aside from the basic control options, any number of mesh transformation operations can be added in this section as indicated by ****transform**. The syntax and operation of these different mesher transformations are the subject of the remainder of this chapter.

Translations:
Version 9.1 is shipped with the following import/export translators:

CODE	DESCRIPTION
abaqus	Import/export an ABAQUS mesh and input file
ansys	Exports mesh and input file commands to ANSYS input file.
gfm	Import/export the GFM format from COSMOS/M with input file commands
gmsh	Import/export GMSH format. Only GMSH format's version 2 is supported for export.
ideas	Import I-deas unv format.
k	Imports/exports to LS-Dyna input file.
neu	Import FEMAP neutral format.
mesh	Import Medit/Distene mesh format.
may	Import FORGE mesh format.
fg3	Import mesh from FORGE result file.
may	Import mesh from REM3D result file.

These import functions currently only import geometrical information. Boundary conditions and real constants are mapped into mesh sets (nset, elset, etc) and given names based on their values.

```
****mesher
 ***mesh
```

Example:
A small example input follows. See the following sections for more examples demonstrating the individual operations. This example opens up a sub-mesh named `BASE1`, rotates and flips it, and creates some node/boundary sets for it.

```
****mesher
 ***mesh BASE
   **open BASE1
   **rotate
     x1 1.  0. 0.
     x3 0.  1. 0.
   **switch *axis  -z z
   **rotate  x3 0. 0. 1. x1 0.984808 -0.173648 0.0
   **translate 0. -5.5 -1.0
   **nset base_fix *plane 0. 0. 1. 0.0
   **bset base_top *plane 0. 0. 1. 1.0
****return
```

Example:
Two examples follow, the first of mesh import and the second of problem export (mesh and loads).

```
****mesher
***mesh linear_trial              % start making file linear_trial.geof
  **import gfm big_mesh.gfm       % import a Cosmos/M model
  **quad_to_lin                   % linearize the mesh
  **elset front  *func x>0.0;     % generate new sets
  **elset bottom *func y<0.0;
  **renumbering frontal_only      % frontal renumbering
****return

****mesher
 ***mesh                          % no name because we're giving output
  **export abaqus abaqus_lin.inp  % via the export command.. abaqus format
  **inp_file                      % include info from the .inp file (BCs)
  **open zebu_lin.geof            % This specifies the input name for both
****return                        % the geof and .inp files
```

**adaptation

Description:
This command is used to remesh and/or optimize a Zebulon mesh (2D, 3D or surfacic) by means of external remeshing libraries. It's main focus is on defining options for preserving FEM entities in order to repeat a computation on the new mesh. Any external remesher should be able to preserve these entities. Additional specific instructions may be required.

Syntax:
The command has the following syntax:

```
**adaptation
 *min_size  hmin
 *max_size  hmax
 [*surface_mesher surface-mesher   ]
 [*volume_mesher 2d-or-3d-mesher  ]
 *metric [  default/scalar/from_function/from_file/uniform_from_field/ ]
 [ *verbose  int-value  ]
 [ *preserve_elsets  elsets-names  ]
 [ *preserve_elsets_start_with  elsets-start-with-names  ]
 [ *preserve_bsets  bsets-names  ]
 [ *preserve_bsets_start_with  bsets-start-with-names  ]
 [ *freeze_elsets  elsets-names  ]
 [ *freeze_elsets_start_with  elsets-start-with-names  ]
 [ *freeze_bsets  bsets-names  ]
 [ *freeze_bsets_start_with  bsets-start-with-names  ]
 [ *freeze_nsets  nsets-names  ]
 [ *freeze_nsets_start_with  nsets-start-with-names  ]
 [ *freeze_fasets_geom  fasets-names  ]
 [ *freeze_fasets_geom_start_with  fasets-start-with-names  ]
 [ *ridges  lisets-names  ]
 [ *ridges_start_with  lisets-start-with-names  ]
 [ *corners  nsets-names  ]
 [ *corners_start_with  nsets-start-with-names  ]
```

`*min_size` imposes minimal edge size.

`*max_size` imposes maximal edge size.

`*surface_mesher` sets the 3D surface remesher. For now, only the option `mms` is available. Note that, some of the following options don't apply to this kind of remesher.

`*volume_mesher` sets the volume remesher. For now, only the option `mmg3d` is available for a 3D mesh and `mmg2d` for a 2D mesh. Note that, all options are automatically transferred from `adaptation` to the remesher.

`*metric default` creates the size map of the current mesh defined by the mean edge lenghts of the mesh. It has the following syntax :

```
*metric default
```

[factor *double-val*]
[min_size *double-val*]
[max_size *double-val*]
[output_file *ascii-file-name*]

where min_size and max_size are usually taken equal to the previous ones from adaptation, but can also be different. The factor is a multiplication factor of the default metric.

*metric scalar imposes a constant isotropic metric. It has the following syntax :

*metric scalar
value *double-val*
[min_size *double-val*]
[max_size *double-val*]
[output_file *ascii-file-name*]

*metric from_function creates a size map using the nodes coordinates which satisfy the given function. Remember to include a semicolon at the end of the function definition. The complete syntax is the following one :

*metric from_function
func *function;*
[min_size *double-val*]
[max_size *double-val*]
[output_file *ascii-file-name*]

*metric from_file reads a nodal size map from a file. The syntax is :

*metric from_file
file *file-name*
[min_size *double-val*]
[max_size *double-val*]
[output_file *ascii-file-name*]

where file-name defines the name of an ASCII file containing the prescribed size map for the new mesh. The ASCII file should contain the following lines :

- first line
  n_b (number of the initial mesh points : 1 integer)
- line 2 to $n_b + 1$
  x_i, y_i, z_i, d_i (definition of P_i coordinates and prescribes size : 4 double values).

Note that, if the number of prescribed sizes is different of the number of the mesh nodes, a transfer through the closest neighbour method is done.

*metric uniform_from_field creates a size map based on a uniform distribution of a given field (usually the discretization error estimate) for a target value of accuracy. The syntax is :

```
*metric uniform_from_field
 field   η
 accuracy   τ
 [min_size   double-val ]
 [max_size   double-val ]
 [output_file   ascii-file-name ]
```

The new size map is computed by the formula

$$h_{new,\Omega_E} = h_{old,\Omega_E} \frac{\tau^{\frac{1}{p}}}{\eta_{\Omega_E}^{\frac{2}{2p+d}} (\sum_{\Omega_E} \eta_{\Omega_E}^{\frac{2d}{2p+d}})^{\frac{1}{2p}}}$$

where d is the dimension, p the finite elements interpolation degree and h_{old} the default metric of the initial mesh. Note that, this metric can only be used during an adaptive procedure or when an initial solution exists, as the field η is either read from a solution file or computed by a previous computational step.

*verbose prints detailed information about the remeshing process. Takes integer values between 0 and 6. The default value is 1.

*preserve_elsets (preserve_elsets_start_with) preserves the given element sets. Set surface interpolation and nodes insertion are allowed.

*preserve_bsets (preserve_bsets_start_with) preserves the given boundary (liset/faset) sets. Interpolation and nodes insertion are allowed.

*freeze_elsets (freeze_elsets_start_with) freezes the given element sets. Surface interpolation and nodes insertion are forbidden.

*freeze_bsets (freeze_bsets_start_with) freezes the given boundary (liset/faset) sets. Interpolation and nodes insertion are forbidden.

*freeze_nsets (freeze_nsets_start_with) freezes the given node sets. The new mesh will contain the initial nodes, at their exact initial positions, and, for parallel purpose, their initial ranks. Note that, nodes insertion is allowed. Thus, for use in a boundary condition please transform the nset in bset and use the freeze_bset option.

*freeze_fasets_geom (freeze_fasets_geom_start_with) freezes the geometry of a surface (faset). Interpolation is forbidden, but nodes insertion is allowed.

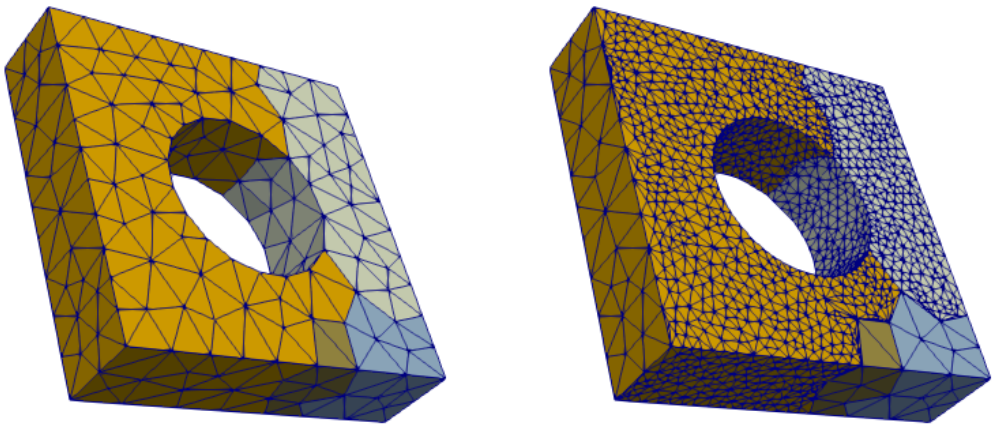
*ridges (ridges_start_with) preserves line sets (lisets). We define a ridge as a curvature line (liset) which locally splits a surface in two smooth surfaces. In the geometry optimization step, this line imposes an independent optimization on each one of its sides. It can be seen as a CAO information to be preserved.

```
****mesher
 ***mesh
  **adaptation
```

*corners (corners_start_with) preserves node sets. It is similar to a ridge, but on nodes. The geometry of each surface splitted by a corner is set to be independently optimized.

Example:

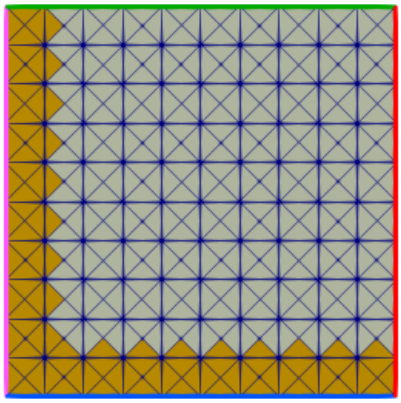
```
****mesher
 ***mesh
  **open hole_toy
  **adaptation
   *min_size .001
   *max_size 2.
   *verbose   6
   *metric scalar
    value 0.15
   *volume_mesher mmg3d
    hgrad 1.25
    hausd 0.005
   *freeze_elsets eset_overlap
   *preserve_elsets_start_with eset
   *freeze_bsets left
   *preserve_bsets right front
   *corners_start_with corn
  **output adaptation_hole_toy_remeshed.geof
****return
```

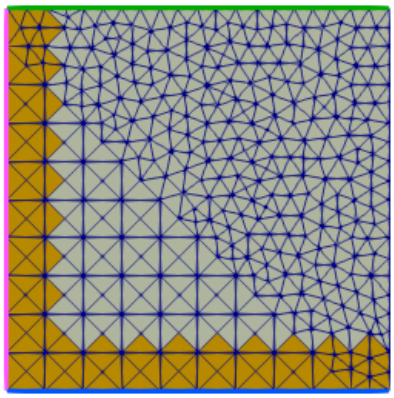


```
****mesher
```

```
                                                    ****mesher
                                                     ***mesh
                                                      **adaptation
```

```
 ***mesh
  **open carre.geof
  **adaptation
   *metric scalar
    value 0.1
   *min_size .1
   *max_size 2.
   *volume_mesher mmg2d
    hgrad 1.
   *verbose 6
   *preserve_elsets diag mix
   *freeze_elsets diag
   *preserve_bsets  left top bottom
   *freeze_nsets diag
  **output carre_remeshed.geof
****return
```





**add_element

Description:
This command is used to enter elements into a mesh "by hand." It is useful normally to utilize element numbers which are "far away" from those generated by other meshing commands. Normally Zmaster and batch mesher commands all presume a sequentially numbered mesh, starting with one. So if you know there will be less than 10000 elements, you could start your hand-created element numbers there.

It is normally very useful to write shell scripts (or command batch files) to generate the records for this type of command.

Syntax:
The `**add_element` command takes a list of element definitions, much as they are given in the `.geof` file. There are no checks if an equivalent element already exists. The element id must be unique, and the nodes with the given ids must exist before this command runs.

`**add_element`
ele-id ele-type n_1 n_2 ... n_N r_1 ...r_m
...

Any number of elements can be added. The entries must however start with an acceptable element number and element type. In the above, n_1 means node number 1 *id*, and r_1 means real constant 1 for the element (e.g. thickness). The number and type of real constant possible depends on the element type, as does the appropriate number of nodes.

A duplication of this command exists which automatically creates an element set made up of the new elements. That command has the syntax:

`**add_element_elset` *eset-name*
ele-id ele-type n_1 n_2 ... n_N r_1 ...r_m
...

Example:
Some example uses follow.

```
**add_element
 6000 c2d4  1 2 3 4
 6001 c2d4  2 5 6 7
**elset new_elem
 *elements 6000 6001
```

A shorter version would use the `**add_element_elset` variant:

```
**add_element_elset new_elem
 6000 c2d4  1 2 3 4
 6001 c2d4  2 5 6 7
```

**add_info

Description:
This command is used to assign "real-constant" information to a set of elements if that has not yet been set in other preprocessing. One command per elset/real constant value must be used. The real constant will also be added on the the end of the real constants for the elements as they exist in the given elset.

Typical applications of this command are to assign thicknesses to plane stress elements.

Syntax:
The `**add_info` command takes the following syntax:

```
**add_info
 *elset eset-name
 *info  real-value
```

Example:
An example use follows:

```
%
% set my plane stress elements thickness to 0.1 mm
%
**add_info   *elset ps_elem  *info 0.1
```

**add_node

Description:
This command is used to enter nodes into a mesh "by hand," and parallels the **add_element command. It is useful normally to utilize node numbers which are "far away" from those generated by other meshing commands.

Syntax:
The **add_node command takes a list of node definitions as they appear in the .geof file. The node dimension will be sized according to the number of coordinates entered in after the node id. Note that zero coordinates must therefore be entered.

There is checking only if the node id already exists. No check is performed for the node coordinate.

```
**add_node
  id-num   x1  x2  [ x3]
    ...
```

A duplication of this command exists which automatically creates a node set made up of the new nodes. That command has the syntax:

```
**add_node_nset nset-name
  id-num   x1  x2  [ x3]
    ...
```

Any number of nodes can be added. The entry fields are read until another **-level command is reached. The dimension of the node is assigned according to the number of entered coordinates (x_1, etc above).

Example:
An example use follows.

```
**add_node_nset some_nodes
 200 1. 0. 0.
 201 2. 0. 0.
 202 2. 1. 0.
 203 1. 1. 0.
```

**boolean_operation

Description:
This command is used to perform boolean operation between meshes (union, intersection or difference). It uses the GNU Triangular Surface library (GTS) plugin for operations on mesh skins. For volume meshes, INRIA remeshing plugins are required (see **yams_ghs3d).

Syntax:
The command has the following syntax:

```
**boolean_operation
 *operation    operation-type
 *file1    geof-filename
 *file2    geof-filename
 *output_file gts-output-filename
 [ *keep_1st    nset   ]
 [ *keep_2nd    nset   ]
 [ *dist_crit   distance   ]
 [ *surface_crit   value   ]
 [ *refinement   func(x,y,z)   ]
 [ *absolu ]
 [ *tolerance tol   ]
 [ *min_size min   ]
 [ *max_size max   ]
 [ *gradation grad   ]
 [ *optim_style opt   ]
 [ *options yams-command-line-options   ]
```

Many keywords correspond to the yams syntax and one should refer to the manual. The operation order is `file1 OPERATION file2`. For a complete description of the syntax see **yams_ghs3d documentation.

*operation *operation-type* must be `intersection`, `union` or `difference`.

*output_file file name containing GTS output after the boolean operation.

*keep_1st this will preserve the topology of specified *nset* the in first mesh.

*keep_2nd this will preserve the topology of specified *nset* the in second mesh.

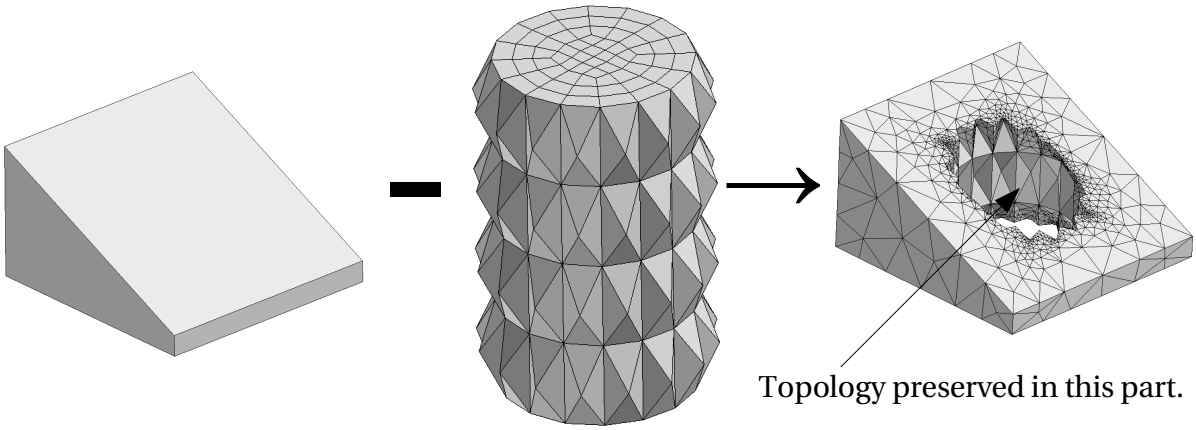
*dist_crit merge vertexes closer than the specified *distance*.

*surface_crit eliminate degenerate triangles with relative surface areas smaller than the specified *value*.

Example:

```
****mesher
 ***mesh
  **boolean_operation
```

```
****mesher
 ***mesh difference
  **boolean_operation_ghs3d
   *operation difference
   *file1 cube.geof
   *file2 volume.geof
   *output_file difference.gts
   *options        -m 100 -FEM
   *optim_style 1
   *keep_2nd skin
****return
```



Note:

The "difference" operation requires the subtracted mesh to be slightly bigger than the other one. Therefore, it may be necessary to "thicken" it slightly, using e.g. `**thicken_bset` (page 2.132) or `**porcupine` (page 2.106).

**bounding_box

Description:

This mesher is used as a utility to print the bounding box which the mesh fits in

Syntax:
There are no options with this command. The bounding box will be calculated using the current mesh configuration. That is after all preceding meshers are applied. Any number of calls to this can of course be applied.

```
**bounding_box
```

Example:
An example output from this command follows.

```
Bounding box :
  xmin=0    xmax=3
  ymin=0    ymax=3
  zmin=0    zmax=1
```

Note this is printed among the general screen messages, and stored no where else. The responses round to integer values if there is no decimal needed.

**bset

Description:

This command creates a boundary set according to various input data. The syntax is the same as for the **nset command, except an ordered boundary is produced. A typical use is to generate first an nset using the **nset command, and then simple make a copy of it in the bset format using the *use_nset option. The command can also be used to find the outer boundary of an elset.

Syntax:
The bset mesher takes the following command syntax:

```
**bset name
 ... options
```

with the following options available:

*axes switches the axes defining a cylindrical coordinate system when using the *plane selection. an example is *axes 1 3 2 where the cylinder is rotated about the 2 axis instead of the default 3-rd axis.

*elset *elset1 ... elsetN* use only the nodes which are used by the given element sets. This can be used to make a "wrap" of separate elsets, or to easily limit the selection criterion.

*function Create a bset using the nodes which satisfy the given function. Remember to include a semicolon at the end of the function definition.

*limit Limit for nodes to qualify to be accepted by a function or plane function (default is 1.e-3).

*nodes *node1...nodeN* uses the given nodes to make up the set.

*plane makes a boundary given a plane equation (4 real values). The 1st 3 values are the components of the plane normal, and the fourth is the intercept (or equiv. for cyl. coordinates).

*req_number allows the user to set the number of nodes of the *use_nset option which must match the element faces in order to create the bsets. For example if the nset given includes just the edge nodes of an element around a corner, the "wrapping" face around the corner can be added.

*surface Indicates that the set should be a bounding surface (outer) of the acceptable nodes (see also unshared_faces page 2.143).

*type cartesian | cylindrical set the type of coordinate system to be used with the *plane option.

*use_dimension *dim* sets the dimension of bset to create. Normally the bset will be a faset for 3D meshes (the max space dimension found in the mesh), and will be a liset for 2D meshes.

... *continued*

```
****mesher
 ***mesh
  **bset
```

`*use_bset` *bset-name* use the nodes in the given bset to apply the criteria given. Several entries may be given.

`*use_nset` *nset-name* make the nset named into a bset. Several entries may be given. This replaces the use of *all nodes* in the mesh to apply the selection criteria.

Example:
To create the outer "wrapping" surface (unshared) for the element set `PIPE2`:

```
**bset wrap1   *surface *elset PIPE2
```

Making an outer face at a radius of 20.

```
**bset face_r=20
 *limit 1.e-3
 *type cylindrical
 *plane 1. 0. 0. 20.
```

More examples are given for the `*nset` *command on page 2.97.*

**bset_align

Description:

This command aligns lisets (in 2D) and fasets (in 3D) such that their "normals" are aligned in the same direction sign-wise. The normals are still normal to the face.

The bset normal alignment is very important for certain boundary conditions (e.g. pressure) and contact surfaces to be correctly defined.

Syntax:

```
**bset_align
 *bsets bset1 ... bsetN
[ *normal (direction) ]
[ *towards (point) ]
[ *away_from (point) ]
[ *inwards ]
```

By default (i.e. if neither `normal`, `towards` nor `away_from` are specified), the bset are oriented outwards (or inwards, if it is specified).

`*bsets` is the list of bsets on which the reorientation is applied. The shorthand `ALL` may be used to select all bsets.

`*normal` is a global direction that all normals will be aligned with (positive scalar product)

`*towards` all normals will be directed towards this point

`*away_from` all normals will be directed away from this point

`*inwards` all normals will be directed inwards (instead of the default outwards)

These last 4 options are of course mutually exclusive. Note that the vectors must be entered using *vector notation* (c.f. page 1.3).

Note:
This mesher replaces `**faset_align` (page 2.67), which was only available in 3D.

Example:

```
**bset_align
 *bsets  top  bottom
 *normal (0. -1. 0.)   % all normals will point down

**bset_align
 *bsets exterior       % all normals will point outwards

**bset_align
 *bsets left right
 *towards (0. 0. 0.)   % all normals will point to the origin
```

**bset_to_mast

Description:
This command generates a `.mast` file fragment from an existing liset. This fragment can be added to an existing mast file, to generate a new geometry.

Syntax:
The command has the following syntax:

```
**bset_to_mast
 *liset  liset-name
 *output mast-file
```

`*liset` is the name of the liset to consider

`*output` specifies the name of the resulting mast file

Example:

```
****mesher
 ***mesh
  **dont_save_final_mesh
  **open section.geof
  **unshared_edges border
  **bset_to_mast
   *liset  border
   *output starter.mast
****return
```

**bset_to_mesh

Description:
This command creates a 2D mesh from an existing boundary set (bset). Currently, it only handles bset on the $x-y$ plane, and linear elements.

In a certain way, it does the opposite of an `**extension`.

Syntax:
The command has the following syntax:

**bset_to_mesh	
*bset_name	*bset-name*
*new_mesh_name	*resulting-geof-name*
[*keep_bset]	*liset-name(s)*

`*bset_name` is the name of the bset to consider.

`*new_mesh_name` is the name of the resulting geof file.

`*keep_bset` is a list of existing lisets to keep in the 2D mesh.

Example:

```
****mesher
 ***mesh tmp.geof
  **open tube3D.geof

  **bset_to_mesh
   *bset_name face.1
   *new_mesh_name tube_section.geof
****return
```

**build_fronts

Description:

This command creates several continuous lisets related to a given nset.

Resulting output as many as required lisets named *liset-name0*, *liset-name1*, etc.

Syntax:

```
**build_fronts
[ *elset elset-name ]
[ *nset nset-name ]
[ *bset liset-name ]
```

`*elset` restrict operation to a given elset (default is `ALL_ELEMENT`).

`*nset` nodes that must be used to create lisets.

`*bset` base name of created lisets.

**build_parallel_boundary_files

Description:
This mesh operation is used to make data files for use with boundary conditions allowing boundary set binary files to be used to apply variable conditions in a parallel computation. Normally such files are supplied at as constant values for element faces, and the binary files must be split in order to represent faces existing on parallel computation sub-domains.

Please see the ***pressure command on page 3.75 for further details for the file loading.

Syntax:
The command has the following syntax:

```
**build_parallel_boundary_files
 *file file-name
 *bset bset-name
```

*file specifies the input file (*problem*) name. The output file names will look like: *file-name*.005 for the output map 5.

*bset the name of the boundary set where these BC values apply. This set **must** be ordered exactly the same as the stored file data.

Example:
An example use follows. The first snippet is the mesher applied to a set of binary files in order to split them up correctly for a parallel computation:

```
****mesher
  ***mesh press_calc
   **open meshfile.geof
     **build_parallel_param_files
     *file thermal_calc.ctnod
     *card 200  % total number to process
   **build_parallel_boundary_files
     *file internal_pressure.bin external_pressure.bin shell_pressure.bin
     *bset internal_pressure     external_pressure     shell_pressure
****return
```

**build_parallel_param_files

Description:
This mesh operation is used to make data files for use with the ***parameter command in a parallel finite element calculation (see page 3.179). It takes a parameter binary file which would normally be used directly with the ***parameter **file command described on page 3.181 and splits it appropriately for the different sub-domains.

Syntax:
The command has the following syntax:

```
**build_parallel_param_files
 *file list-of-files
 *card nb-outputs-for-each-file
[ *ip ]
```

*file specifies either one or a list of binary input file names for reading the parameter values. These files will be read using the Ni format which defaults to big-endian. The output file name will be formatted for example *file-name*.005 (for the output map 5).

*card the number of output maps for each file specified in the *file parameters. This command assumes that the file records 1-1 the current mesh node numbering and size.

*ip indicates that the parameter values are at integration points, not nodes. All the element integration locations are assumed to map exactly the same as the calculation generating the parameter file.

**phi_psi_no_refine

Description:
This command is used to compute the two levelsets describing a planar semi-circular crack. The first levelset Φ implicitly describes the crack plane as the surface where $\Phi = 0$ whereas the second levelset Ψ implicitly describes a cylinder as the surface where $\Psi = 0$ whose intersection with the plane forms the crack.

So the crack location is defined by:

$$\begin{cases} \Phi = 0 \\ \Psi < 0 \end{cases}$$

This mesher command computes the nodal values of these two levelsets, and stores the result in two files named `phi.dat` and `psi.dat`. These two files can then be used in an XFEM analysis.

Syntax:
The command has the following syntax:

```
**phi_psi_no_refine
  *center (1. 0.5 0.5)
  *normal (0. .2 .7)
  *rho .3
  *vtk_output levelset
```

`*center` defines the crack center

`*normal` is the plane normal vector

`*rho` is the crack radius

`*vtk_output` if present, allows to export the computed levelsets into two vtk legacy files for debugging purposes using Paraview

**cfv_build

Description:
This command is used to mesh a regular shape around a crack front (the command name stands for "Crack Front Volume build"). This volume can be used to perform accurate integrals for energy release rate computations with De Lorenzi or G-theta formulations. In order to be connected to a tetrahedral mesh, "c3d5" pyramid elements are located on every face which must be connected to the rest of the mesh.

The crack front be can specified by a liset (a Catmull-Rom spline interpolation is used), or volume must be completely described by a crack front method component (i.e. `gtheta`). Warning: this method is only available for 3D meshes, and it generates linear meshes only.

Syntax:
The command has the following syntax:

```
**cfv_build
 [ *liset  liset-name propagation-direction-vector  ]
 [ *ask_crack_to crack-method-name  ]
 *rc   internal-square-length
 *ri   internal-radius
 *re   external-radius
 *nbc  number-of-square
 *nbri  number-internal-cuts
 *nbre  number-external-cuts
 [ *delta square-deformation  ]
 [ *delta_p pyramid-height  ]
 [ *connect connection-radius  ]
 [ *opening opening-angle  ]
```

Note that at least on of the command `*liset` or `*ask_crack_to` must be specified in order to select how to build the volume.

`*liset` specifies whether one liset describes the crack front. A vector is required to define the orientation of the crack.

`*ask_crack_to` specifies which *crack-method* describes the volume.

`*rc` defines the semi-edge length of the interior square of the section.

`*ri` defines the radius of the interior circle of the section.

`*re` defines the radius of the exterior circle of the section.

`*nbc` is used to define the number of sub-squares dividing interior square part of the section.

`*nbri` defines the number of radial cuts between the square part and the interior circle.

`*nbre` defines the number of radial cuts between the interior and the exterior circles.

`*delta` is used to define the square deformation.

`*delta_p` used to specify the height pyramid elements.

`*connect` used to specify volume should be connected to a hexahedral mesh: a layer of pyramidal elements is inserted.

`*opening` used to create a volume with a meshed crack open in direction *propagation-direction-vector*. Initially this crack is filled by elset "`compl`".

Various elsets are created:

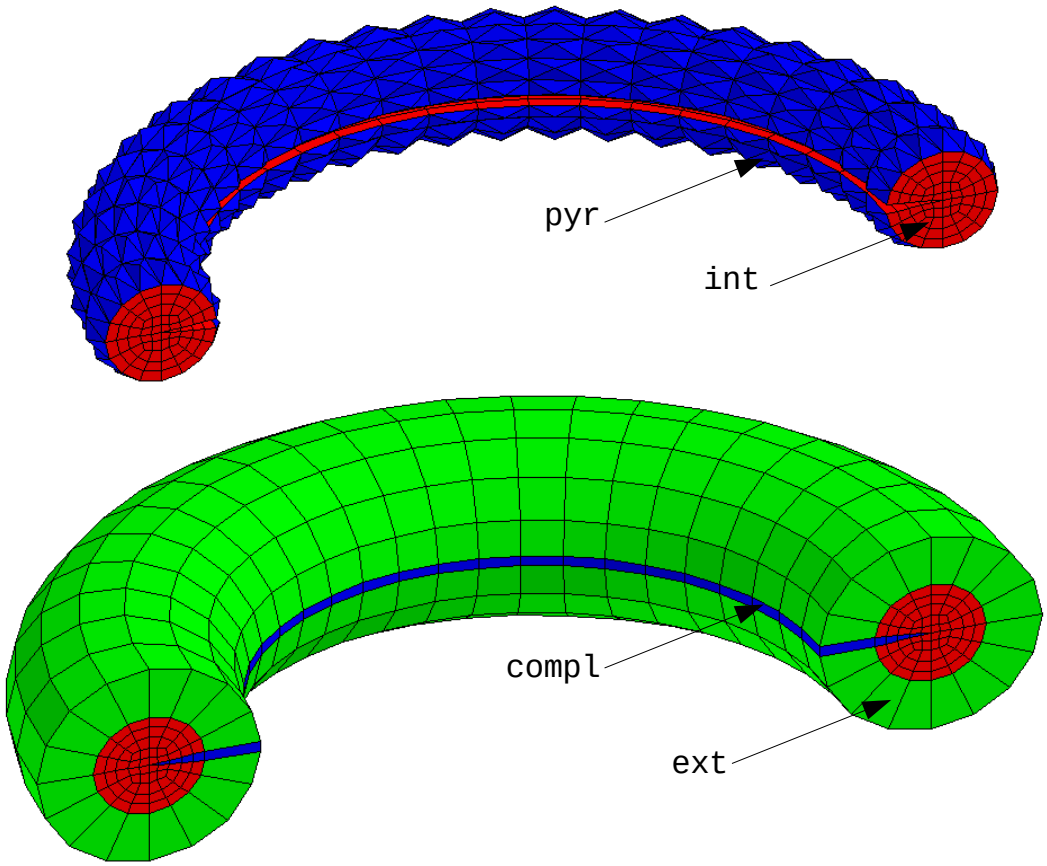
- "`int`" contains the interior part of the volume with only hexahedral elements (on which integrations should be computed).
- "`pyr`" contains the pyramidal elements used to connect the volume to tetrahedral meshes.
- "`ext`" contains external pyramidal elements if the `connect` keyword is specified.
- "`compl`" contains elements inserted inside the crack if the `opening` keyword is specified.

Various nsets are created:

- "`front`" contains the nodes located at the crack front.
- "`skin`" contains the nodes located on the skin made of pyramidal elements used to connect the volume to tetrahedral meshes.

Example:

```
****mesher
 ***mesh volume.geof
  **open crack.geof
  **cfv_build
   *liset LISET_FRONT (-1. 0. -1.)
   *rc .3
   *ri .6
   *re 1.
   *nbc 2
   *nbri 1
   *nbre 2
   *opening .1
   *delta .07
   *delta_p .2
   *connect 2.
****return
```


pyr
int
compl
ext

**check_orientation

Description:
This command verifies that all elements are correctly mapped such as to create a valid geometry (i.e. face normals all pointing out and properly connected).

Syntax:
This command takes an optional command to select the elset upon which the orientation check should be made.

```
**check_orientation
 *elset eset-name
```

`*elset` specify an element set to check and adjust so as to be correctly defined. By default the element set is `ALL_ELEMENT`.

**check_quality

Description:

This command checks the quality of a mesh, prints an histogram of element qualities and builds an elset with the "bad" elements.

Currently (Z-set 8.7) it only computes a quality for linear tetrahedrons (c3d4).

Syntax:

```
**check_quality
 *elset_to_check elset
 *elset bad-quality-elements
 *quality_threshold threshold
```

`*elset_to_check` restricts the analysis to this elset. Default is on ALL elements.

`*elset` is the name of the output elset containing all elements with a quality worse than a threshold. Default is `BAD_QUALITY_ELEMENTS`.

`*quality_threshold` modifies this threshold (default is 100).

Example:

The following is from the example mesher `Mesher_test/INP/mesh_quality.inp`:

```
****mesher
  ***mesh
   **open mesh_quality_input.geof
   **quad_to_lin
   **yams_ghs3d
    *optim_style 1
   **check_quality
    *quality_threshold 4
****return
```

**classical_renumbering

Description:
This command resets node and element IDs, from 1 to #nodes and #elements respectively.

Syntax:
The command has the following syntax:

```
**classical_renumbering
```

Note:
When a mesh is classically ordered, a node ID is equal to its rank+1, thus making some meshing or computing operations simpler and faster.

**classical_to_zstrat

Description:
This command is used to transform 3D volumic elements into "Z-strat" solid elements. Z-strat elements are quadratic in the plane and linear in their thickness. They are typically used for composite applications.

Syntax:
The command has the following syntax:

```
**classical_to_zstrat
 *axis   axis-number
[ *layered bool ]
[ *elset elset-name ]
```

`*axis` is a number defining the linear axis.

`*layered` is a boolean indicating that a layered geometry should be used (e.g. c3d12l or c3d16l instead of c3d12 and c3d16). The default value is not layered (`FALSE`).

`*elset_name` indicates that the mesher should apply only to the elements in the given elset. The default is to apply on `ALL_ELEMENT`.

Example:

```
**classical_to_zstrat
 *axis 3        % elements become linear in the z axis
 *layered TRUE  % make layered elements
 *elset E1      % apply on E1 elset
```

**cleanup_bsets

Description:
This command removes duplicated elements in the bsets.

Duplicated faces or lines may appear in a bset, when it is created with a "join" command, or if the bset is not on the boundary of the domain.

Syntax:

```
**cleanup_bsets
```

**condense_out_elset_domain

Description:
This mesher is for parallel problems, where a specific feature across the mesh is desired to be wholly contained in one domain. The mesh should already be split into domains, from which the elements in the given elset will be "condensed out" into a new domain (i.e. the original split should be $n-1$ domains, with n the total desired number). Note that doing this does not ensure that the domains will be well sized for balanced load, or that poorly conditioned partial domains with isolated small groups of elements could be created.

Syntax:
The command has the following syntax:

```
**condense_out_elset_domain
 *elset elset-name
```

`*elset` specifies a pre-existing elset name which will make up the new domain.

**compute_predefined_levelset

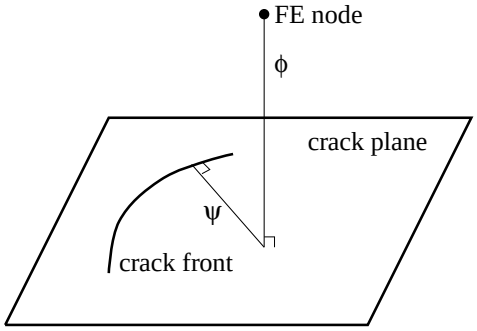
Description:

This command is used to generate levelsets (scalar field given at nodes of the FE mesh) defining cracks with classical geometries in an xfem analysis (page 3.233).

In this case the discontinuity is described by 2 levelsets ϕ and ψ, where (see figure):

- ϕ is the signed distance of a node to the crack plane,
- ψ is signed the distance to the crack front of the orthogonal projection of a node on the crack plane.

such that points for which $\phi = \psi = 0$ are located on the crack front, and positive ψ values correspond to nodes ahead of the front.

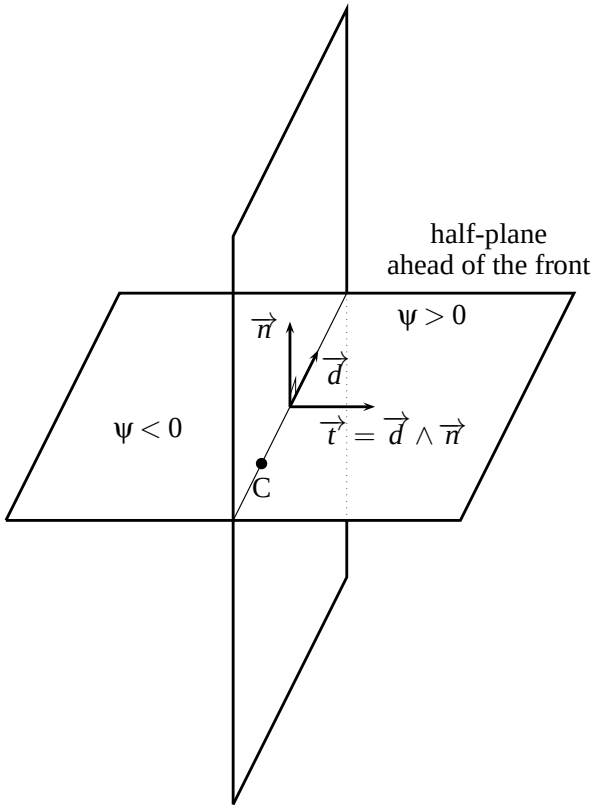


Syntax:

```
**compute_predefined_levelset
  *type (circular|elliptic|line)
[ *rho  radius  ]
  *center  c
  *normal  (n)
[ *rho2  minor_radius  ]
[ *direction  (d)  ]
[ *do_fem_output ]
[ *elset_name  name  ]
[ *elset_distance  dist  ]
[ *phi_name phi_file  ]
[ *psi_name psi_file  ]
```

- command `*type` is used to specify the crack geometry (predefined types available are: `circular`, `elliptic`, `line`),
- *radius* is the radius (`circular` cracks) or the major radius value (`elliptic` cracks). It has no effect for `line` cracks.
- *(c)* is a vector defining the position of the center for `circular` or `elliptic` cracks. In the case of `line` cracks, the coordinates of a point on the crack front is expected.
- *(n)* is a vector defining the normal to the crack plane (see figure above),

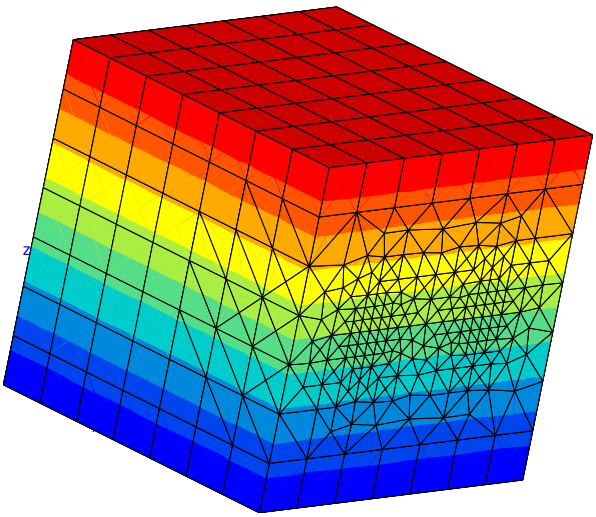
- for `elliptic` cracks, command `rho2` is needed to specify the minor radius value.
- command `*direction` defines a vector *(d)* whose meaning is the following, depending on the particular geometry:
  - the direction of the minor axis for an `elliptic` crack. Note that in this case *(d)* is expected to be orthogonal to *(n)* (otherwise an error occurs).
  - a vector parallel to the crack front for `line` cracks. The sign of this vector also defines the crack propagation direction in this case, as described on the following figure: $\psi > 0$ is obtained in direction $(t) = (d) \wedge (n)$



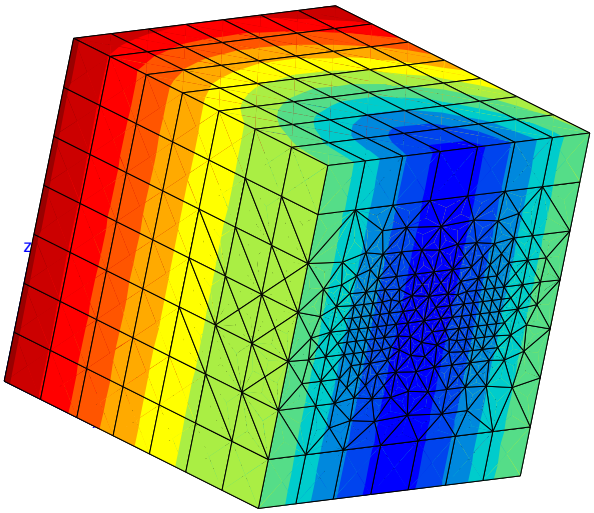
- optional command `*do_fem_output` allows the generation of FE results with (ϕ, ψ) values calculated on nodes of the input mesh (see the example for the iso-contours generated).
- command `*elset_name` creates an elset (with name *ename*) containing elements situated within a distance of less than *dist* from the crack front.
- commands `*phi_name` and `*psi_name` allows to redefine the names of the output ASCII files used to store the levelset values on the mesh nodes (default names are "`phi.dat`" and "`psi.dat`"). Those files are used to define the discontinuity by the `***xfem_crack_mode` command.

Example:

```
**compute_predefined_levelset
 *type circular
 *center (1.0 0.5 0.5)
 *normal (0. 0. 1.0)
 *rho 0.2
 *do_fem_output
```



ϕ : distance to the crack plane



ψ : distance to the crack front

**continuous_liset

Description:
This command is used to reorder a scrambled line set. It first looks for a potential line set beginning, and then puts all other segments in the right order. The result is a well ordered line set, even if the initial segment ordering is completely scrambled.

Syntax:
The command has the following syntax:

```
**continuous_liset
 *liset_name liset-name
[ *start_at_node node-id ]
[ *start_near (x,y,z) ]
```

The `*liset_name` is used to indicate the name of the source line set to be re-ordered. If this line set is a loop (hence has no "natural" beginning), the optional `start_at_node` parameter specifies where to begin the liset. Alternatively, the `start_near` parameter specifies the localization of this first node its (possibly approximate) coordinates.

**crack_2d

Description:
This command is used to insert a crack in a 2D mesh and apply 1/4 node displacements at the tip.

A duplicate of the command exists which automatically modifies the nodal positions on edges surrounding the crack tip to 1/4 node positions to simulate a singular field better. To get this option replace `crack_2d` with `crack_2d_quarter_nodes`

Syntax:
The command has the following syntax:

```
**crack_2d_quarter_nodes
 *liset liset-name
[ *node node-id   ]
[ *crack_nset nset-name    ]
[ *half ]
```

The options for this command are described below:

`*liset` give the liset name which describes the crack plane. This liset will be kept after the meshing operation.

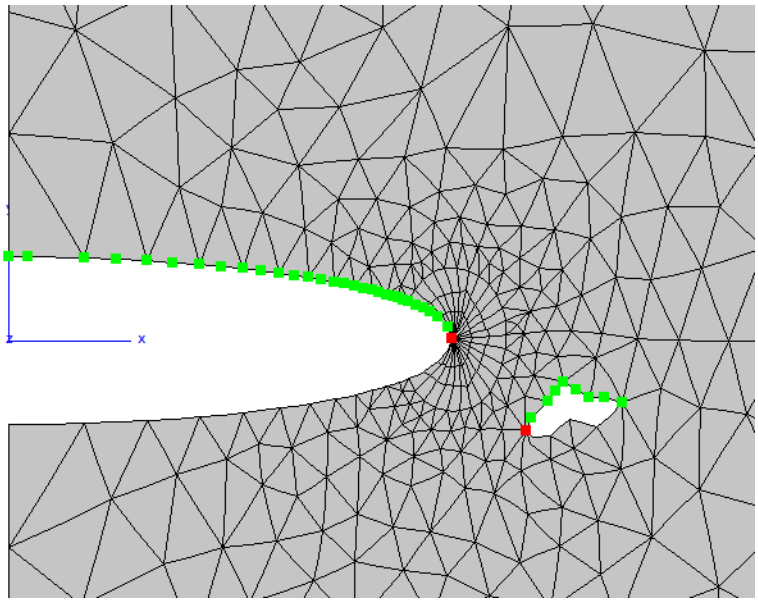
`*node` Give node ids or nset names indicating nodes which are crack tips. These nodes terminate the crack surface creation.

`*crack_nset` enter a name for a new nset to be created, which makes up the opposing surface from the crack definition liset. This lets one do contact, or other manipulations on the newly created surface.

`*half` Only make a closed crack tip at the first end of the given liset. That is for an edge-crack, make the line set originating where the crack tip is to be and leading out to the free surface.

Example:
An example use is shown below taken from the test case `Jint_test/INP/ccp_2.inp` which is shown in the figure below.

```
****mesher
 ***mesh
  **crack_2d
```



This case includes 2 cracks, one of which is open ended (the first) and one which automatically detects the crack tip nodes.

```
****mesher
 ***shell
   rm -f ccp_2.geof
   Zrun -B ccp_2.mast
 ***mesh ccp_2
  **open ccp_2.geof
  **crack_2d_quarter_nodes
   *liset crack
   *half
  **crack_2d_quarter_nodes
   *liset crk
****return
```

**crack_3d_quarter_nodes

Description:

This command is used to set 1/4 node positions for adjacent faces of a 3d crack, as defined by a line set. *It does not yet provide crack insertion via splitting face sets.*

Syntax:
The command has the following syntax:

```
**crack_3d_quarter_nodes
 *liset liset-name
```

where the *liset-name* is the name of a valid line set defining the crack tip line.

**create_interface_elements

Description:
This mesher takes a pre-existing boundary set and creates interface elements (e.g. debonding) at that location. The mesher can be used either between a boundary set and an existing node set, which assumes that bset and nset are not actually connected, or by using a boundary set only, where additional nodes will be inserted to make an interface condition.

Syntax:
The command has the following syntax:

```
 **create_interface_elements
  *boundary bset-name
[ *elset elset-name ]
[ *nset nset-name ]
[ *axi     ]
[ *reduced ]
```

`*axi` flag that the interface elements are axisymmetric. Default is not axisymmetric.

`*boundary` indicates the boundary set from which to create interface elements. There will be 1 interface element per boundary segment in the set.

`*elset` gives the name of the elset to create with the new interface elements. If this command is not included the name `interface` will be used for a new elset.

`*nset` indicates the opposing node set name. This option indicates that the mesh is already disconnected at the interface.

`*reduced` flag indicating reduced integration. Default is full integration.

**create_interface_elements_between_elsets

Description:
This mesher takes pre-existing element sets and creates interface elements (e.g. debonding/cohesive elements) at their common boundaries.

Syntax:
The command has the following syntax:

```
**create_interface_elements_between_elsets
 *elsets  elset-name1 elset-name2 [ elset-name(n)... ]
[ *axi     ]
[ *reduced ]
[ *remove_sets ]
```

`*elsets` gives the names of the elsets between which the boundary will be created (note: `elsets` and not `elset`). The keyword `ALL_ELSET` may be used to designate all elsets (except the standard elset `ALL_ELEMENT`) present in the mesh. Names for the new element sets will be automatically generated, and a new elset `EI_ALL` will be generated containing all new interface elements, as well as an elset `VOL_ALL` containing all volume elements (so that the union of `EI_ALL` with `VOL_ALL` should yield `ALL_ELEMENT`).

`*axi` flag that the interface elements are axisymmetric. Default is not axisymmetric.

`*reduced` flag indicating reduced integration. Default is full integration.

`*remove_sets` indicates that only the interface elements should be kept, and that the associated elsets generated during the meshing are thrown away. The default is to keep all elsets.

**create_interface_elset

Description:
This mesher applies interface (debonding) elements between all elements of a given elset.

Note:
Currently (Z-set 8.3 and newer), this command is implemented for linear 2d elements only (i.e. it will create `i2d4` elements).

Syntax:
The command has the following syntax:

```
**create_interface_elset
 *apply_to elset-name
 *elset_name new-elset-name
```

`*apply_to` indicates the element set which is to be broken up and have interface elements inserted at each element edge. The default element set is `ALL_ELEMENT`.

`*elset_name` The new element set containing the interface elements. The default name is `debond`.

**cut_surface

Description:

This command performs a robust surface intersection operation. It requires a 3D volume mesh that better be refined (using Distene remeshing tools) closely to a given meshed surface.

Resulting output is made of a conform surface mesh (elset `Nsurface`) decomposed in various elsets: parts of original elsets boundaries (`Nskin`) and their newly cut parts (`Nskinc`), created cut surface (`Nlip`). Some output nset are also produced: `Nlip` containing created cut surface nodes, `Nskin` skin nodes, `Nfront` front nodes (for surface containing a front liset i.e. for 3D crack insertion), `Nseg_pb` containing original volume mesh edges nodes for edges cut an even times by the cut surface (smaller original volume elements are required in this area, as cut surface curve radius must be too small there).

This fast and robust algorithm is based on the given volume elements intersection, thus a remeshing process is required before FE computations (for better surface meshing, and volume filling), on the output `Nsurface`.

Syntax:

```
 **cut_surface
[ *elset elset-name-list ]
[ *elset_to_cut elset ]
[ *nset_to_cut nset ]
[ *nset_not_to_cut nset ]
[ *surface elset-name ]
[ *tolerance tol ]
[ *front liset-name-list ]
[ *front_ini nset ]
[ *filter tol ]
[ *allow_quad ]
[ *inside ]
```

`*elset` list of elsets that should be cut by surface. If more than one is given, boundaries between the elsets will be kept during the process (to be able to keep various elsets in the output).

`*surface` is the given surface elset mesh to intersect with.

`*elset_to_cut` only this elset may be cut (default is to cut `ALL_ELEMENT`).

`*nset_to_cut` only edges contained in this nset may be cut.

`*nset_not_to_cut` edges contained in this nset are not allowed to be cut.

`*tolerance` if an edge intersection node is closer from an edge node than this distance value assume the edge node is on the cut surface. *In most cases (in fact, in all currently tested cases), the zero default value of this parameter has proved to be robust.*

`*front` list of lisets representing the extremity of the cut surface that must be accurately inserted (i.e. for 3D crack insertion).

*front_ini initial crack front, for newly cracked surface insertion (this nset represents the previous crack front), the newly surface will contain this previous front.

*filter tolerance distance used to fuse pairs of same created surface elements (i.e. front 3D crack advance, to keep only one side of previously built lip). Default value is zero (no surface will be fused), if required a usual value is 1.e-5 times the mesh characteristic length.

*allow_quad quad elements can be created during the algorithm. Default is not allowed as non-planar elements can be created an make a surface remeshing process fail.

*inside crack surface advance may not pass through the boundary of the mesh (e.g. when 2 cracks are coalescing).

**dg_transform

Description:
This command is used to transform a mesh for continuous Galerkin formulation to a mesh suitable for discontinuous Galerkin formulation.

Syntax:
The command has the following syntax:

```
 **dg_transform
[ *elset ]  elset-name
[ *bset  ]  bset-name
[ *name  ]  name-of-bset-to-create
```

`*elset` specifies elset where operations should be made.

`*bset` specifies bset where discontinuity should be inserted (should not work well for now).

`*name` specifies name that will be given to pairs of bset where DG interface formulation will be imposed (format will be namea0/nameb0, namea1/nameb1, ... ,nameaN/namebN).

**deform_mesh

Description:
This mesher deforms the active mesh via results taken from a previous analysis. It is a simple implementation assuming 1-1 correlation between the meshes. It can be used to load any types of displacement results, but is most commonly used to create bucking analysis imperfections via the modes of a previous eigen solution.

Syntax:
The command has the following syntax:

```
**deform_mesh
 *map solution-no
 *input_problem pb_name
 *magnitude disp
 *format fmt
```

`*format` gives the solution format to use for deforming the mesh. These can be any of the ones with a valid results database implementation (see e.g. page 2.139).

`*input_problem` The prefix name of the problem results which will be used for the deformation.

`*map` specifies the solution or map number from the results file which will be used to get the deforming displacements. It must be one listed in the `.ut` file. The default is to take the last output available.

`*magnitude` the scale factor for the displacements to be applied to the mesh. The default is 1.

Example:
An example for deforming a mesh with a previous Eigen solution follows:

```
****mesher
 ***mesh buckle.geof
  **open T_54_9.geof
  **deform_mesh
   *map 1
   *input_problem lanczos
   *magnitude 0.05
```

**delete_elset

Description:
This command is used to remove elements from the mesh by a given element set name. Elements are also removed from any other elset of which they are part. Nodes (in nsets for instance) and integration points (in ipsets) which are "orphaned" by this command will be removed as well. If elsets or ipsets become empty as a result of this, they will also be removed. The command can be used to create cavities in the mesh, which are otherwise difficult to obtain.

Syntax:
The delete elset command simply takes a list of valid elset names. There are no options with this command.

```
**delete_elset elset-name
```

Note:
If you wish to simply remove the *element set* and not the elements themselves, use the `remove_set` command (see page 2.115).

**div_quad

Description:
This mesher is used to perform a quadrilateral to triangular transformation.

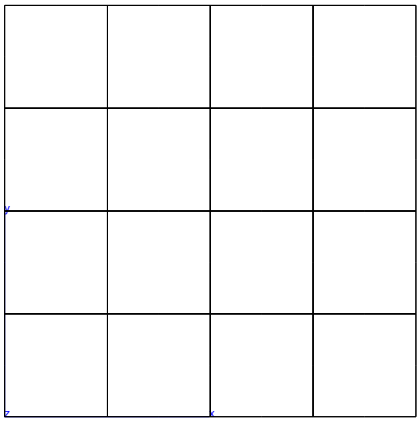
Syntax:
The command has the following syntax:

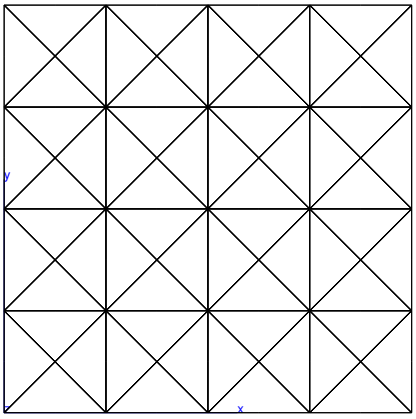
```
**div_quad
[ *elset elset-name ]
```

`*elset` gives an elset for which the division will be applied. the default elset is `ALL_ELEMENT`.

Note:
This mesher does not preserve any sets related to the elsets which are sub-divided.

Example:
The following shows the division take for a ruled mesh.





**elset

Description:
This command creates an element set according to various input data. If the element set already exists, new elements will simply be added. Elements will be included in the set only once.

Syntax:
The elset command takes a variety of options to specify the elements to be added (and each of them may be specified more than once):

```
 **elset name
[ *add_elset eset1 ... esetN ]
[ *allow_partial ]
[ *attached_to_nodes id_1 ... id_N ]
[ *attached_to_nset nset1 ... nsetN ]
[ *elements id_1 ... id_N ]
[ *func function ; ]
[ *not_in_elsets eset1 ... esetN ]
[ *remove_elset eset1 ... esetN ]
[ *sequence start_id end_id [ increment ] ]
[ *use_elsets eset1 ... esetN ]
```

where the options can be taken from the following:

`*add_elset` Adds all the elements in the named (pre-existing) element sets.

`*allow_partial` Indicates that a function describing element node positions for the elset is true if one or more of the nodes meets the given function criteria. By default, all nodes must fit the equation.

`*attached_to_nodes` indicates that all elements attached to nodes with the given ids are to be added to the elset.

`*attached_to_nset` indicates that all elements attached to nodes within the given nset names are to be added to the elset.

`*elements` Enter element ids in directly. Non-existing element ids will be silently ignored.

`*func` Enter a function which is used to select the elements. The function should be a chain of multiplied boolean expressions in the coordinate space (see example below).

`*not_in_elsets` will add the complement to the given element sets as a new elset. Not compatible with other options such as `*func` and so on.

`*remove_elset` will remove the elements of the given element sets from the list of currently added elements. This operation is executed after all other `**elset` subcommands.

... continued

```
****mesher
 ***mesh
  **elset
```

`*sequence` will add a sequence of elements from *start_id* to *end_id* with increment *increment* (Z-set 8.4 and newer). The latter can be negative, but in that case, *start_id* should be greater than *end_id*. If this is not the case, -*increment* will be used instead. If *increment* is omitted, a default value of `1` is used. Non-existing element ids generated with this command will be silently ignored.

`*use_elsets` use only the elements in the named element sets to check the given function against. The default is to use all elements in the mesh.

Example:
Some example uses follow.

```
% Shows the use of functions
  **elset front
   *func x>0.0;
  **elset bottom
   *func y<0.0;
  **elset look
   *func (y<0.0)*(z>20.0);
   *allow_partial

% Creates/adds to the elset "first_10" with elements 1 through 10
% (better use the *sequence command)
  **elset first_10
   *elements 1 2 3 4 5 6 7 8 9 10

% Creates/adds to the elset "sequence_example" with elements
% 1 4 29 30 32 34 ... 48 50 53
  **elset sequence_example
   *elements 1 4 29
   *sequence 30 50 2
   *elements 53

% Creates/adds to the elset "inverse_sequence" with elements
% 50 48 46 ... 32 30
  **elset inverse_sequence
   *sequence 50 30 -2

% Creates/adds to the elset "twice" with elements 1 through 15
  **elset twice
   *sequence 1 10
  **elset twice
   *sequence 5 15
```

**elset_by_element_type

Description:
This mesher creates one elset per element type.

Syntax:
The command has the following syntax:

```
**elset_by_element_type
```

and takes no option.

**elset_explode

Description:
This mesher is used to provide a per-elset exploded view of a mesh, which can be useful for presentation purposes. Because this mesher adds nodes between elset interfaces, the display of ctnod results files will be invalidated (including the displacements), and there is no checking to that regard[2]

Note:
By default, this mesher does not actually explode the elsets! If the shrink factor is not specified, it merely creates the nodal interfaces in order to do so later with a `**function` (page 2.68) or `**translate` (page 2.139) mesher afterwards.

Syntax:

```
**elset_explode
[ *elsets elset-list ]
[ *shrink factor ]
```

`*elsets` list of elsets used as domains to separate. By default, all elsets are used, which can lead to bad result.

`*shrink` specify the factor used separate the different parts of the mesh. The new nodes position is computed as $\underline{\mathbf{v}}' = \underline{\mathbf{v}} + f \cdot (\underline{\mathbf{e}} - \underline{\mathbf{c}})$
with $\underline{\mathbf{v}}$ and $\underline{\mathbf{v}}'$ respectively the initial and final node position, f the shrinking factor, $\underline{\mathbf{c}}$ the center of the mesh $\underline{\mathbf{e}}$ the center of the elset.

Example:
One can use this mesher after a calculation has been performed to separate the mesh, then re-assign the visualization mesh in the *problem*`.ut` file. The following splits the 2 elset problem `zsteel_rubber` found in the static test directory:

```
****mesher
***mesh toto
 **open zsteel_rubber.geof
 **elset_explode
  *elsets acier rubber
 **translate
  *elset_name acier
  *x -20.0
  *y -20.0
```

The `ut` file can be now modified to use the `toto.geof` file just created.

```
**meshfile toto.geof
**node U1 U2 Pn RU1 RU2 RPn
**integ sig11 sig22 sig33 sig12 dv F11 F22 F33 F12 F21
```

[2] actually the record sizes will be interpreted incorrectly, and the results loaded are therefore a randomized mix of values. No crash occurs however.

```
                                                    ****mesher
                                                     ***mesh
                                                      **elset_explode

**element
1 1 1 1 4.000000000000000e+00
2 1 1 2 8.000000000000000e+00
3 1 1 3 1.200000000000000e+01
4 1 1 4 1.600000000000000e+01
5 1 1 5 2.000000000000000e+01
```

And the results can be then viewed on the exploded mesh via Zmaster. When doing this make sure and turn off deformed mesh and to include either (or both) `**contour_by_element` and `**value_at_integration` output options in the calculation.

**elset_split

Description:
The **elset_split command is used to create parallel problem domains and write a *problem*.cut file based on different named elsets. Normally one would use an automated mesher such as **metis_split to do this, but the elset split ensures repeatability and fine control over the process.

Note:
Any elements in the mesh which are not assigned to a domain using the *domain sub-option will be assigned to the 1st domain.

Syntax:
The command has the following syntax:

```
**elset_split
 *no_binary
 *domain elset-name
   ...
 *domain_startswith stem
```

*no_binary indicates that no binary cut file should be written (default is to write the binary cut).

*domain enter new domain made up of the elements in the given elset name. Repeat the command for all domains.

*domain_startswith is a shorthand to automatically enroll all domains whose name starts with the given *stem*.

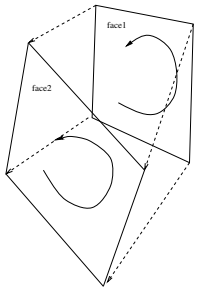
Example:
There are some examples of this command in the Parallel_test directory.

```
****mesher
 ***delete_file mpc1b_small.geof
 ***shell Zrun -B mpc1b_small.mast
 ***mesh mpc1b_small
  **open mpc1b_small.geof
  **quad_to_lin

  **elset right *add_elset R2 R4
  **elset left  *add_elset R1 R3

  **elset_split
   *domain right
   *domain left
   *no_binary
```

**extension

Description:
This command extends planar (2D or 3D shell/face) geometries into 3D. One can use either an element set or a face set which is already defined.

Note:
For extensions which are between two element sets, care must be taken to properly define the sets. Order of domain selection is critical here. The figure here shows the proper order for ruled mesh definition in Zmaster. Select the edge segments in the sense of the arrow paths when taking the same view to your mesh extension.



Syntax:

```
**extension
 *elset      eset1 | ALL_ELSET
 *elset2     eset2
 *new_elset  new_elset
 *distance   dist
 *num        num
 *prog       prog
 *dir        dir-vec
 *flipit
 *reduced
 *invert_fasets
 *remove_initial_nsets
 *create_faset
 *liset_names liset1 liset2 ...
 *elset_names elset1 elset2 ...
```

`*elset` *eset1* defines the element set or faset which will be extended. There is no limitation on the type of elements or faces contained on the face. Mixed order is not allowed however. The option `ALL_ELSET` extends each elset of the mesh[3]. If no elset is specified, the elset `ALL_ELEMENT` is taken.

`*elset2` *eset2* This optional command makes the extension between two mesh/face sets as given. **Note that the mesh topology must be exactly the same.** This command is most useful with the mapped mesh domains available in Zmaster. The degenerate case of 2 edges being the same (so the extended mesh forms a wedge) is handled. Do not use `*distance` with this option. This option is disabled when the first elset name is `ALL_ELSET`.

[3] Available in version 8.2 or newer

*new_elset *new-elset* This optional command gives the elset name for the created elements instead of being the same as the input name. This is disabled when the input elset name is ALL_ELSET.

*distance This command is used to define the extension distance, *dist* (real). Do not use it when the extension is between two meshes.

*num This command takes an integer value for *num* in order to define the number of elements which will be created in the cross section.

*prog Geometric progression of the extension.

*dir The direction along which to extend. This direction is specified in vector form and does not need to be normalized. The default is (0. 0. 1.) .

*flipit Used when the created elements are inverted.

*reduced Create reduced integration elements.

*invert_fasets In order to properly calculate the normals (and other properties) of a face set, the nodes in each element need to be ordered in a certain manner within the faset. This is very important for applying correctly many types of boundary conditions. After extension, the nodes in the newly created faset have the same order as in the parent faset, which implies that the normal points in the same direction, which may not always be the desired direction. This option allows the order within a faset to be inverted, so that the direction of the normal of the newly created faset is reversed as well.

*remove_initial_nsets Initial nsets are also extended and the suffix "-ext" is added to these new nsets names. This option allows to remove initial nsets and keep extended one with the initial names.

*create_faset The two facing sides of the extended mesh are created and named face.1 and face.2.

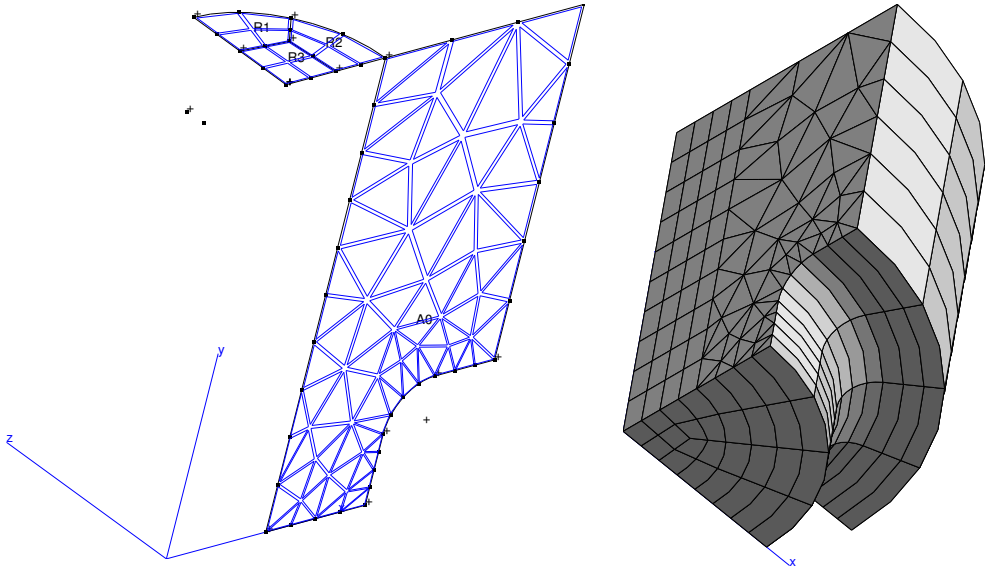
*liset_names Create the "extended" version of the given lisets. They will be suffixed with -ext.

*elset_names Create the "extended" version of the given elsets. They will be suffixed with -ext.

Example:

The following example shows the use of extensions in different directions, and making use of the 3d plane meshing in Zmaster.

```
****mesher
 ***mesh
  **extension
```




```
**extension
 *elset R1
 *dir (0. 1. 0.)
 *distance -2.0
 *num 10
**extension
 *elset R2
 *dir (0. 1. 0.)
 *distance -2.0
 *num 10
**extension
 *elset R3
 *dir (0. 1. 0.)
 *distance -2.0
 *num 10
**sweep
 *elset A0
 *angle 90.0
 *num 4
```

**extension_along_nset

Description:
This mesher is a modification of the **extension mesher in order to allow an extension path to be defined with a set of nodal points.

Syntax:
This command accepts the same commands as the **extension command (see page 2.61) and two additional commands related to the path:

```
**extension_along_nset
 *nset_to_follow nset-name
 *global_normal normal-vector
[ standard extension options ]
```

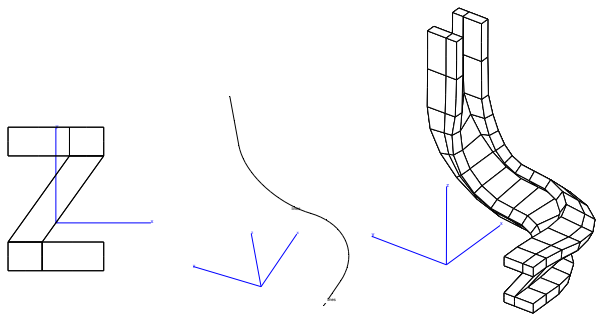
Note that many of the commands from extension, though available, are ignored for this mesher because of the pre-defined path (such as num, direction, etc).

*global_normal Input a vector allowing the initial face plane to be calculated. The tangent vector of the path is calculated based on linear segments from and to adjacent nodes.

*nset_to_follow Enter an existing nset path for the mesh extension to take place.

Example:
The following example shows extension along an nset, using a curve generated in Zmaster and meshed using line l3d2 line elements (under Ruled mesh). A boundary set called lines was also created in Zmaster which creates the nset along which the extension will take place.

```
****mesher
 ***mesh extension_nset_linear
  **open curve_linear.geof
  **union
   *add zzz.geof
   *elset zzz
   *tolerance 0.
  **extension_along_nset
   *nset_to_follow lines
   *global_normal (1. 0. 1.)
   *elset zzz
  **delete_elset lines
 ****return
```



**extract_surface

Description:
This command builds bsets corresponding to each surface of a given elset.

Syntax:
The command has the following syntax:

```
**extract_surface
[ *elset     ] elset-name
[ *criterion ] angle
```

`*elset` restricts the operation to the named elset (defaults to `ALL_ELEMENT`),

`*criterion` is the angle (in degrees) from which two adjacent elements are considered part of different surfaces (defaults to 45°).

**extrude_shell

Description:
The command **extrude_shell is used to extrude 3d shell elements into "Z-strat" solid elements.

Syntax:
The command has the following syntax:

```
**extrude_shell
[ *elset_name elset-name  ]
[ *layered  ]
[ *thickness thickness ]
```

*elset_name indicates that the mesher should apply only to the elements in the given elset. The default behavior is to use ALL_ELEMENT for the elset.

*layered is a flag setting that a layered geometry should be used (e.g. c3d12l or c3d16l instead of c3d12 and c3d16). The default is not layered.

*thickness gives an uniform thickness *thickness* to the elements. Default value is taken arbitrarily as 0.1.

**faset_align

Description:

This command aligns fasets such that their "normals" are aligned in the same direction sign-wise. The normals are still normal to the face.

The faset normal alignment is very important in order that boundary conditions (e.g. pressure) and contact surfaces are correctly defined.

> This command is now replaced by the more general **bset_align (page 2.23).

Syntax:

```
**faset_align
 *faset faset1 ... fasetN
 *normal direction
 *towards point
 *away_from point
```

The normal must be entered using *vector notation* (c.f. page 1.3). If it is not entered at all, either the option *towards or *away_from must be entered with a "target" point entered in vector form.

Example:
The following is from the example problem Straw/LIN_PLASTIC/straw.inp

```
***mesh straw
 **open straw_base.geof
 **nset fix
  *point 0. -1. 0.
 **nset R0-surf
  *func (y>=0.0)*(y<=1.0);
 **nset R1-surf
  *func (y<=0.0)*(y>=-1.0);
 **bset target
  *use_nset nodes-ext
  *elset A2
 **faset_align *faset R6.1   *normal  (0. -1.0 0. )
 **faset_align *faset target *normal  (0. -1.0 0. )
```

**function

Description:

This command modifies nodal positions according to the given function. These functions are as described in the chapter on functions and scripting (page 6.2). Function transforms can be applied to the whole structure (default), to nset(s), and to the nodes of an elset or elsets.

Syntax:

```
**function
 *func    func(x,y,z)
 *xtrans  func(x,y,z)
 *ytrans  func(x,y,z)
 *ztrans  func(x,y,z)
 *nset    nset1  ...  nsetN
 *elset   elset1  ...  elsetN
```

`*func` used for function definitions which are henceforth available. One can use a "learn" command line option as well to specify an external file with function definitions.

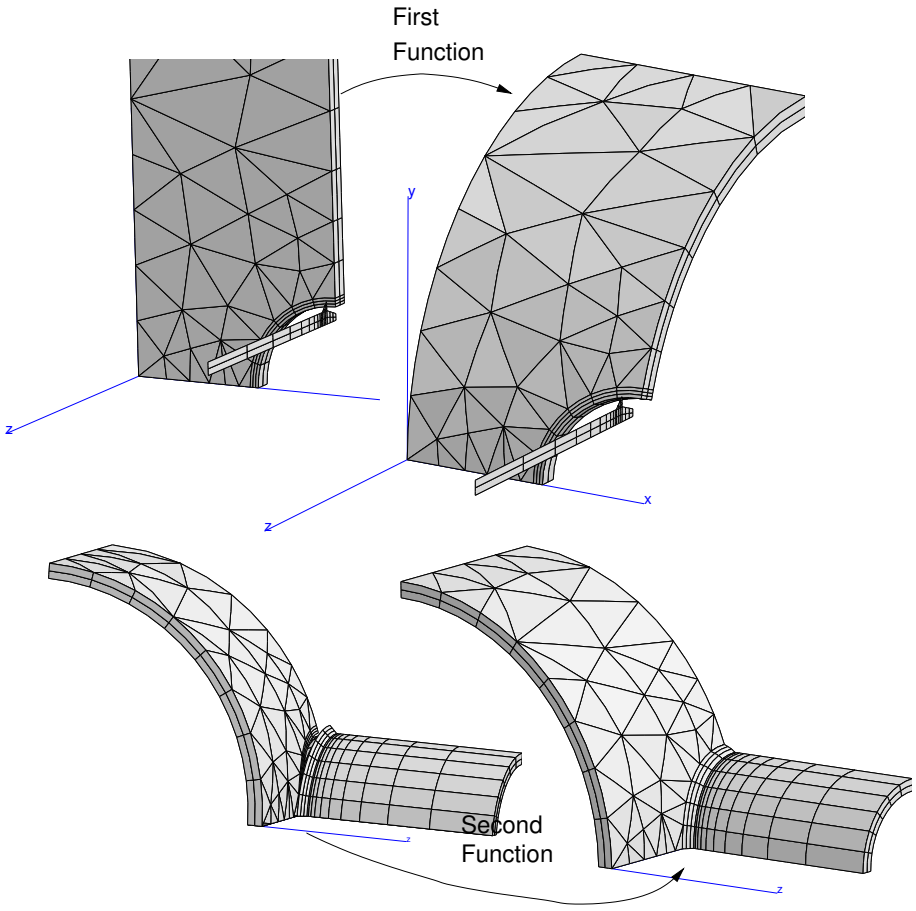
`*xtrans *ytrans *ztrans` functions which specify a coordinate reassignment in x, y, or z directions. Note that these are absolute values (trans is a bit of a misnaming), so an actual translation of 0.40 in the z direction would use `*ztrans z-0.40;`

`*nset` specify nset(s) for the function to be applied to.

`*elset` specify elset(s) for the function to be applied to.

Example:
The following are the function statements from the example
`$Z7PATH/test/Mesher_test/INP/func-xform.inp`

```
 **function
   *elset pipe1
   *function_def R:= 6.366197722 + z;
   *function_def theta:=y/6.366197722;
   *function_def Z:=R(z)*cos(theta(y))-6.366197722;
   *function_def Y:=R(z)*sin(theta(y));
   *ztrans Z(y,z);
   *ytrans Y(y,z);
...
 **function
   *elset weld
   *elset pipe2
   *function_def f1:= (z>=1.0)*1.0 + (z<1.0)*(z>0.0)*z + (z<=0.0)*0.0;
   *function_def f2:= (z>=0.2)*1.0 + (z<0.2)*(z>0.0)*(z*z/0.04);
   *ztrans f1(z)*z   + (1.-f1(z))*Z(y,z);
   *ytrans f2(z)*y   + (1.-f2(z))*Y(y,z);
```


First
Function
y
z
x
z
Second
Function

**fuse_nset

Description:

This command joins nodes forming a "mini-union" algorithm, or can also be used to "clean up" a mesh with possible gaps in nodes. This is very useful when batch mesher transforms are applied to an initially exploded arrangement of sub-meshes, which are then to be transformed together and glued. It is also an effective method to make cracks (in the absence of using **crack_2d).

Syntax:

```
 **fuse_nset
[ *nset1 nset-name ]
[ *nset2 nset-name ]
[ *tolerance tol ]
[ *cleanup_mesh  ]
[ *cleanup_nset nset ]
[ *average_locations  ]
[ *allow_mixed_fuse  ]
[ *debug  ]
```

*allow_mixed_fuse fuse nodes of different dimension.

*average_locations this indicates that the new node position for joined nodes is in the middle of their original positions. By default the node position in node set one is taken.

*cleanup_mesh this changes the mode of operation to a "clean up" mode, where all nodes closer than the tolerance are joined.

*cleanup_nset cleanup only the nodes of a given nset.

*nset1 when not running in *cleanup_mesh mode, this specifies the name of one of the two node sets which will be compared. It is required for standard use.

*nset2 enter the node set name to be compared with *nset1*.

*tolerance enter a real value for the magnitude of distance between nodes at or below which they will be considered the same (its default value is that of the global parameter Mesher.MeshFusion).

*debug create a new nset (name is debug_nset) which shows the nodes that have been fused (only the remaining ones). Useful to see what would happen is fuse_nset were not called. You can for instance make a fake call to fuse_nset with the cleanup_mesh option to see where nodes are doubled.

**geof_format

Description:
This mesher is used to convert the mesh format from ASCII based, human readable form to a binary file, and back again. Please note that the binary file will be read by Zebulon and Zmaster much faster than the formatted mesh, and will also take up less disk space.

Syntax:
The command has the following syntax:

```
**geof_format
 *format file-name
 *unformat file-name
```

The options `*format` and `*unformat` set the operation to respectively convert the current mesh file to a formatted `.geof` file, or convert the formatted file to a binary `.geo` file. The default operation is to format the mesh. The mesh will be immediately saved using the current mesh name (set after the `***mesh` command). Thus mesher commands after the `**geof_format` will not be included in that mesh file.

Note:
Alternatively, the mesh is automatically saved in binary mode if its filename ends with `.geo`, as in `***mesh mymesh.geo`.

**hexa_to_tetra

Description:
This command is used to tetrahedralize a mesh (i.e. transform all its c3d8 or c3d8r elements to c3d4).

Syntax:
The command has the following syntax:

```
**hexa_to_tetra
[ *elset ]    elset-name
```

`*elset` restricts the operation to the named elset

Note:
There is currently another mesher that has the same objective: `hexa_only_to_tetra`. The latter's main advantage is that it tries to keep a conforming mesh (i.e. cuts the faces coherently between two adjacent elements). This cutting operation is not trivial, nor always possible. So you should always check the result to see if the resulting mesh is conforming. Currently it only knows how to handle c3d8, doesn't have the `*elset` option, and doesn't update the bsets and elsets accordingly (nsets are unchanged by the cutting operation).

Syntax:
The command has the following syntax:

```
**hexa_only_to_tetra
[ *seed ]    element-rank
[ *talkative ]
```

`*seed` selects which element should be started with (default is the first element).

`*talkative` (integer) produces more output, useful if something goes wrong. 1 adds a little more output, 2 is very chatty.

Example:

```
****mesher
 ***mesh plaque_tetra.geof
  **open plaque.geof
  **hexa_only_to_tetra
   *seed 4561
   *talkative 1
****return
```

**import_abaqus_pressure

Description:
This command imports ABAQUS input data for pressures on element faces into Zebulon. This allows more robust pre-processors with ABAQUS export (e.g. Patran, Femap, etc), or existing user programs, to define variable pressures over a surface.

Syntax:
The command has the following syntax:

```
**import_abaqus_pressure
 *bset bset-name
 *ptable_file out-file-name
 *use_dimension dim
```

*bset assign the name of the new bset for the pressures to be applied to.

*ptable_file

*use_dimension

**insert_discontinuity

Description:

This command quickly duplicates nodes of a bset to create a discontinuity. If an nset is given those nodes will not be duplicates in order to represent a 3D crack front for instance.

Resulting output contains an odd number of nset named `side0`, `side1`, `side2`, `side3`, etc. linked to each pair of sides of the discontinuity (a pair is built for each discontinuous zone of the given bset).

Only for 3D meshes.

Syntax:

```
 **insert_discontinuity
[ *elset elset-name ]
[ *bset faset-name ]
[ *nset nset-name ]
```

`*elset` restrict operation to a given elset (default is `ALL_ELEMENT`).

`*bset` continuous or discontinuous faset where discontinuity must be built.

`*nset` nodes that must not be duplicated (to create a crack front for instance).

**inverse_bset

Description:
This command is used to inverse the geometric direction of line or face sets. The order of all the segments of the set is reversed, as is the nodal ordering of each segment, so the face or edge normals are inverted.

Any mix of lisets or fasets can be given to this command.

Syntax:

```
**inverse_bset
  *names l1 l2 l3
    ...
```

Note:
This command supersedes the older `**inverse_liset`.

**inverse_liset

Description:
This is used to revert the way a liset is described. It can be useful for instance for contact computations in which the target liset must be followed in an order so that impacting nodes are located on the right.

Syntax:

```
**inverse_liset
  *names l1 l2 l3
    ...
```

Any number of liset can be inverted in the same pass. Liset names have to be declared after `*names` option.

> This command is deprecated as of 8.3 in favor of `**inverse_bset`.

**join_bsets

Description:
This command is provided in order to join bsets together.

Syntax:
**join_bsets requires a new bset name, and needs one or more existing bsets to be entered which will be joined in the new set.

```
**join_bsets new-bset
  *bsets bset1 ... bsetN
[ *remove_duplicates ]
```

where *bset1* ... *bsetN* indicate the character name of pre-existing boundary sets (faset or liset).

If the remove_duplicates option is specified, boundary elements appearing more than once in the union are not repeated (the verification is not done by default, for performance reasons).

Example:
Here is a typical example:

```
**extract_surface   % detects the geometrical zones
**join_bsets pressure_zone
 *bsets surface_2 surface_3 surface_7
```

This first example doesn't require the remove_duplicates option, since extract_surface generates disjoint surfaces.

Incidentally, this mesher can also be used to remove potential duplicates in an existing bset:

```
**nset inner_surf
 *function abs(z)<1.e-3;              % picks all nodes at z=0

**bset inner_surf_with_duplicates     % when this surface is internal
 *use_nset inner_surf                 % all faces are duplicated

**join_bsets inner_surf_no_duplicates  % so we use join_bsets to remove them
 *bsets inner_surf_with_duplicates
 *remove_duplicates
**bset_align                          % and make sure orientation is correct
 *bsets inner_surf_no_duplicates
 *normal (0. 0. 1.)
```

**join_nsets

Description:
This command joins node sets. The original node sets are conserved. In the new node set the order of the nodes is the same as in the constituting node sets, and the order of the node sets within the new node set is as entered by the user. By default a check is carried out in order to prevent nodes from being added twice. This can be switched off with the `*keep_duplicates` option.

Syntax:
The `**join_nsets` command requires a new node set name and one or more existing nsets to be joined into the new node set.

```
**join_nsets new-nset
 *nsets nset1  [ nset2 ...  nsetN  ]
[ *keep_duplicates ]
```

where *nset1 ... nsetN* indicate the names of pre-existing nsets.

**lin_to_quad

Description:
Use this command to increase the integration order of elements from linear to quadratic. The command always assumes that the mid-side nodes are linearly placed between the corner nodes, so if the linear mesh approximates a curve, the quadratic mesh will in fact model the chord faceted approximation.

Syntax:
The command has the following syntax:

```
**lin_to_quad
[ *elset ] elset-name
[ *no_nset ]
```

`*elset` if specified, only the given elset is set to quadratic. The default is to run on the whole mesh.

`*no_nset` if specified, nsets are not modified, and thus only contain nodes from the original linear mesh.

Note:
In version 8.3 there is no corresponding alteration of the node sets and boundary sets which might be affected by this increase in order. The user should be careful to remove such unwanted (and in the bset case, invalid) sets by hand using the `**remove_set` command (see page 2.115).

Note:
The inverse operation `quad_to_lin`, i.e. going from quadratic to linear elements, is described on page 2.109.

**make_springs

Description:
This command is supplied as a convenient means of adding spring elements to a mesh[4]. In particular, one node springs which resist motion from the initial position can be added to nsets which is often a useful means of achieving stability in structures which are initially unstable for a quasi-static solution (e.g. soft springs added to a contact problem to allow a solution before contact begins).

Syntax:

```
 **make_springs spring-type  new-elset
[ *connect_points vector1  vector2  ]
[ *connect_nodes  node1  node2  ]
[ *nset           nset-name  ]
[ *nset_pair      nset1  nset2  ]
[ *load_pmpc_eqn  file-name  ]
[ *proximity      prox-val  ]
[ *start_ele_id   id-val  ]
```

The algorithm checks the spring geometry whether is is a one or two node spring, and uses the given options valid for that spring geometry.

`*connect_points` used for 2 node elements, this command will link the two nodes closest to the given coordinates with a spring. The command may be given as many times as required.

`*connect_nodes` used for 2 node springs, connecting by node id number rather than position. The command may be repeated.

`*nset` used for 1 node springs. Springs of the new set will be attached to each node contained in the node set.

`*nset_pair` Join nodes between the two sets with a 2 node spring.

`*proximity` sets the algorithm to do a "proximity matching" between the two given nsets (thus `*nset_pair` must be used). For each node of the 1st nset the closest node at most *prox-val* away is linked up. So not all nodes in the 1st nset are required to be attached to springs, and each will have only one spring attached. The 2nd nset may however have multiple springs attached to one node.

`*start_ele_id` uses the given id number as the starting number for the new spring elements.

[4] in Zebulon, 2 node springs are truss elements. 1 node elements are not drawn in Zmaster, and 2 node elements appear as lines

```
****mesher
 ***mesh
  **make_springs
```

Example:
A simple example adding one node springs follows.

```
  **nset spring_set
   *nodes
          % Bolts, nuts (horiz)
          136 164 163    43 70 38
          633 609 610    637 638 661

          % bolts (vert)
          1622 1623 1635 1620
          1686 1688 1699 1700
  **make_springs l3d1 SPR
   *nset spring_set
```

Example:
Here specific locations are connected with 2 node springs.

```
  **make_springs l2d2 springs
   *connect_points (0. 0. 0.) (1. 1. 1.)
   *connect_nodes  25  341
```

Example:
The last example uses proximity matching for frontal renumbering across a discontinuous mesh. The spring connectors are deleted right after doing the renumbering.

```
****mesher
 ***shell Zrun -B make_springs_renum
 ***mesh make_springs_renum
  **open make_springs_renum.geof
  **make_springs l2d2 springs
   *nset_pair face_outside face_inside
   *proximity 8.5
  **renumbering frontal_only
  **delete_elset springs
****return
```

**mesh_lin_rectangle

Description:
This command is used to quickly generate a 2D ruled linear mesh of a rectangle. See also the quadratic version, on next page.

Syntax:

```
**mesh_lin_rectangle
 *ncutx  nx
 *ncuty  ny
 *sizex  length_x
 *sizey  length_y
```

*ncutx, ncuty	Number of subdivisions along the x and y axes.
*sizex, sizey	Length of the box in each direction.

Example:

```
****mesher
 ***mesh rect.geo
  **mesh_lin_rectangle
   *ncutx 10
   *ncuty 200
   *sizex 1.
   *sizey 20.
****return
```

**mesh_quad_rectangle

Description:
This command is used to quickly generate a 2D ruled quadratic mesh of a rectangle. It shares its syntax with `mesh_lin_rectangle` (see previous page).

Syntax:

```
**mesh_quad_rectangle
 *ncutx  nx
 *ncuty  ny
 *sizex  length_x
 *sizey  length_y
```

Example:
The following example creates a mesh of a 175×3 mm plate (circular shape, in axisymmetric formulation), with square 0.1×0.1 mm elements.

```
****mesher
 ***mesh regular_quad_plate.geo

  **mesh_quad_rectangle
   *sizex 175.0e-3
   *sizey 3.0e-3
   *ncutx 1750
   *ncuty 30

  **translate *y 0.11
  **to_cax

  **rename_set
   *nsets
    x0   axis
    x1   tip
    y0   bottom
    y1   top
    x0y0 Pbot0
    x1y0 Pbot1750
    x0y1 Ptop0
    x1y1 Ptop1750

  **bset tip    *use_nset tip
  **bset top    *use_nset top
  **bset bottom *use_nset bottom
  **bset_align  *bsets ALL

****return
```

**mesh_quad_cube

Description:
This command is used to quickly mesh a cube. You can also refer to `mesh_quad_parallelepiped` on page 2.85.

Syntax:

```
**mesh_quad_cube
 *size length
[ *ncut n ]
[ *nb_nodes nb_nodes ]
```

`*size` Length of the cube sides

`*ncut` Number of segments along the cube sides

`*nb_nodes` May be specified instead of `ncut` as the target number of nodes for the final mesh

Example:

```
****mesher
 ***mesh cube
  **mesh_quad_cube
   *ncut 10
   *size 1.
****return
```

**mesh_quad_parallelepiped

Description:
This command is used to quickly generate a ruled mesh of a parallelepiped. You can also refer to `mesh_quad_cube`, page 2.84. NSETs of the 6 faces (x0, x1, y1, ...), of the 12 edges (x0y0, x0y1, x0z1, ...) and of the 8 corners (x0y0z0, x0y0z1, ...) are automatically generated by default.

Syntax:

```
**mesh_quad_parallelepiped
 *ncutx  nx
 *ncuty  ny
 *ncutz  nz
 *sizex  length_x
 *sizey  length_y
 *sizez  length_z
[ *biasx  bias_x ]
[ *biasy  bias_y ]
[ *biasz  bias_z ]
[ *min_hx  min_hx
  *max_hx  max_hx ]
[ *min_hy  min_hy
  *max_hy  max_hy ]
[ *min_hz  min_hz
  *max_hz  max_hz ]
[ *no_sets ]
```

`*ncutx, ncuty, ncutz` Number of subdivisions along the x, y and z axes.

`*sizex, sizey, sizez` Length of the box in each direction.

`*biasx, biasy, biasz` Geometric progressions parameters that drive the element size from lower to higher coordinates. Do not use bias[xyz] and min_h[xyz] together.

`*min_hx, min_hy, min_hz` An other way to set geometric progressions that drive the element size from lower to higher coordinates. Do not use bias[xyz] and min_h[xyz] together.

`*no_sets` The faces, edges and corner NSETS are not generated

Example:

```
****mesher
 ***mesh pave.geo
  **mesh_quad_parallelepiped
   *ncutx 10
   *ncuty 20
```

```
                                                    ****mesher
                                                     ***mesh
                                                      **mesh_quad_parallelepiped

   *ncutz 5
   *sizex 10.
   *sizey 20.
   *sizez 5.
****return
```

**metis_renumbering

Description:
This is a re-numbering mesher which will reduce the fill-in generated during factorization of the global matrix with sparse direct solvers (sparse_direct or sparse_dscpack). The software is based on the Metis package developed by the Computer Science Department at the University of Minnesota, and used by permission. Information on the Metis package is available on the Web at the link:

`http://www-users.cs.umn.edu/~karypis/metis/`

Note:
Since version 8.3 the nodal renumbering for sparse matrices is done automatically, so this command is no longer generally used.

Syntax:
The command has the following syntax:

```
**metis_renumbering
[ *param_files file1 ... fileN ]
```

`*param_files` enables to renumber in the same time external parameter files.

**metis_split

Description:
This is a mesher routine to build a sub-domain *problem*.cut file to be used with the parallel solver. The software is based on the Metis package developed by the Computer Science Department at the University of Minnesota, and used by permission. Information on the Metis package is available on the Web at the link:

http://www-users.cs.umn.edu/~karypis/metis/

Note:
One can also use the ONERA splitmesh program (see p.2.127).

Syntax:
The command has the following syntax:

```
 **metis_split
[ *check_domains  ]
[ *dont_check_domains  ]
[ *check_domains_iter iter  ]
[ *domains num  ]
[ *no_binary  ]
[ *no_elset  ]
```

*check_domains indicates that the domains should be checked to see if there are elements attached by less than an edge in 2d, or a face in 3d. This is important to avoid rigid body and conditioning problems, especially with triangular or tetrahedral meshes. This option is on by default; use the *dont_check_domains option to disable it.

*check_domains_iter modifies the maximum of iterations allowed in this domain verification procedure (default value: 10).

*domains specifies the number of sub-domains to be used. In the absence of this subcommand, the default value of 10 will be taken.

*no_binary suppress the binary file, which is normally used if there are both .cu (binary) and .cut (ascii) files. Useful for hand-modifying the cut files, or for verification.

*no_elset suppresses creation of additional element sets showing the domains which were created.

Example:
An example use follows. This case has a problem with some domains only tied by MPCs, so springs are created before to establish total connectivity. The springs are removed after the split is done, and then the cut file is re-written using the **write_cut_from_elsets command to take into account the deleted elements.

```
****mesher
 ***mesh calcul_parallel
  **open big_problem.geof
  **make_springs l3d2 pmpc1 *load_pmpc_eqn mpc_set_1.equ
```

```
                                                                ****mesher
                                                                 ***mesh
                                                                  **metis_split

     **make_springs l3d2 pmpc2 *load_pmpc_eqn mpc_set_1.equ
     **metis_split
      *domains 8
      *no_binary
     **delete_elset pmpc1
     **delete_elset pmpc2
     **write_cut_from_elsets
****return
```

**mmg3d

Description:

This command is used to remesh and optimize a Zebulon tetrahedral mesh by means of MMG3D remeshing library. MMG3D is part of the open source (LGPL license) MMG platform and interfaced with Zebulon through it's C language API.

Note that only linear elements are handled by `mmg3d`. A combination of the `quad_to_lin` (before `mmg3d`) and `lin_to_quad` commands (after) may then be used. Yet, the command `quad_to_lin` may degrade the initial mesh and, when `lin_to_quad` is used, the quadratic nodes are not projected on the geometry.

Syntax:
We distinguish two types of syntaxes: specific to the mmg tools (mmg3d, mmgs, mmg2d) and necessary for preserving FEM requiered mesh entities. Note that the description of the first type follows the online documentation of the mmg software (https://www.mmgtools.org/mmg-remesher-try-mmg/mmg-remesher-options). The description of the second type is given on page 2.10 and will not be discussed here.

```
**mmg3d
 *min_size  hmin
 *max_size  hmax
 [ *verbose  int-value  ]
 [ *hgrad  double-val  ]
 [ *hgradreq  double-val  ]
 [ *hausd  double-val  ]
 [ *angle_detection  degree  ]
 [ *nodetection  ]
 [ *nosurf  ]
 [ *noinsert  ]
 [ *nomove  ]
 [ *noswap  ]
 [ *octree  ]
 [ *local_parameters  local_set local_min local_max local_hausd  ]
 *metric [  default/scalar/from_function/from_file/uniform_from_field/ ]
 [ metric_options ]
 [ *preserve_elsets  elsets-names  ]
 [ *preserve_elsets_start_with  elsets-start-with-names  ]
 [ *preserve_bsets  bsets-names  ]
 [ *preserve_bsets_start_with  bsets-start-with-names  ]
 [ *freeze_elsets  elsets-names  ]
 [ *freeze_elsets_start_with  elsets-start-with-names  ]
 [ *freeze_bsets  bsets-names  ]
 [ *freeze_bsets_start_with  bsets-start-with-names  ]
 [ *freeze_nsets  nsets-names  ]
 [ *freeze_nsets_start_with  nsets-start-with-names  ]
 [ *freeze_fasets_geom  fasets-names  ]
 [ *freeze_fasets_geom_start_with  fasets-start-with-names  ]
```

```
                                                    ****mesher
                                                     ***mesh
                                                      **mmg3d

   [ *ridges  lisets-names  ]
   [ *ridges_start_with  lisets-start-with-names  ]
   [ *corners  nsets-names  ]
   [ *corners_start_with  nsets-start-with-names  ]

   [ *output_mmg_files  ]
```

`*min_size` imposes minimal edge size.

`*max_size` imposes maximal edge size.

`*verbose` prints detailed informations about the remeshing process. Takes integer values between -1 and 5, -1 being tottally mute. The default value is 1.

`*hgrad` sets a gradation value that controls the ratio between two adjacent edges ($\frac{1}{hgrad} \leq \frac{e_1}{e_2} \leq hgrad$ for two adjacent edges e_1 and e_2). By default, the gradation value equals 1.105171.

`*hgradreq` metric gradation along required edges to avoid very bad qualities when the prescribed metric doesn't match with the size of the required entity: Note that you will be impacted if you use the -nosurf option, required edges / triangles / tetrahedra and/or parallelTriangles (library mode only). When set to -1, the option is disabled.

`*hausd` controls the boundary approximation: it imposes the maximal distance between the piecewise linear representation of the boundary and the reconstructed ideal boundary. Thus, a low Hausdorff parameter leads to a refinement of the high curvature areas. The default value is set to 0.01, which is suitable for an object of size 1 in each direction.

`*angle_detection` used to modify the value for the sharp angle detection. By default it is set to 45°. This means that a sharp angle is detected at the interface of two boundary elements when the angle between their outward normals is greater than 45°. The edge or node between these normals are set as a ridge or corner.

`*nodetection` when set, it forbids the sharp angle detection.

`*nosurf` when used, it freezes the surfaces of all element groups. No surface modification is allowed.

`*noinsert` when used, it forbids nodes insertion in the new mesh generation process.

`*nomove` when used, it forbids point relocation.

`*noswap` when used, it forbids edge flipping in the new mesh generation process.

`*octree` is used by mmg3d to partition the mesh vertices and thus, to speed up the vertices insertions. Before inserting a point, mmg seeks the octree cell to which the new point will belong and checks if it is not too close from another point of this cell or of one of the neighbouring cells. By default, an octree cell may contain at most 64 vertices.

`*local_parameters` as the name suggests, it allows a local definition of the minimum, maximum and Hausdorff parameters on one boundary set (bset) or element set (elset).

```
****mesher
 ***mesh
  **mmg3d
```

*output_mmg_files when set, it creates the input and output meshes and metrics in mmg native format (.mesh and .sol).

Example:

```
****mesher
 ***mesh
  **open hole_toy
  **mmg3d
   *min_size  .01
   *max_size  2.
   *verbose   6
   *freeze_elsets eset_overlap
   *preserve_elsets_start_with eset
   *freeze_bsets left
   *preserve_bsets right front
   *corners_start_with corn
   *metric scalar
    value 0.15
   *hgrad 1.25
   *hausd 0.005
   *output_mmg_files
  **output hole_toy_remeshed.geof
****return
```

**mmg2d

Description:
This command is used to remesh and optimize a Zebulon triangular mesh by means of MMG2D remeshing library. MMG2D is part of the open source (LGPL license) MMG platform and interfaced with Zebulon through it's C language API.

Note that only linear elements are handled by `mmg2d`. A combination of the `quad_to_lin` (before `mmg2d`) and `lin_to_quad` commands (after) may then be used. Yet, the command `quad_to_lin` may degrade the initial mesh and, when `lin_to_quad` is used, the quadratic nodes are not projected on the geometry.

Syntax:
We distinguish two types of syntaxes: specific to all mmg tools (mmg3d, mmgs, mmg2d) and necessary for preserving FEM requiered mesh entities.

```
**mmg2d
 *min_size  hmin
 *max_size  hmax
 [ *verbose  int-value  ]
 [ *hgrad  double-val  ]
 [ *hausd  double-val  ]
 [ *angleDetection  degree  ]
 [ *nodetection  ]
 [ *nosurf  ]
 [ *noinsert  ]
 [ *nomove  ]
 [ *noswap  ]
 *metric [  default/scalar/from_function/from_file/uniform_from_field/ ]
 [ metric_options ]
 [ *preserve_elsets  elsets-names  ]
 [ *preserve_elsets_start_with  elsets-start-with-names  ]
 [ *preserve_bsets  bsets-names  ]
 [ *preserve_bsets_start_with  bsets-start-with-names  ]
 [ *freeze_elsets  elsets-names  ]
 [ *freeze_elsets_start_with  elsets-start-with-names  ]
 [ *freeze_bsets  bsets-names  ]
 [ *freeze_bsets_start_with  bsets-start-with-names  ]
 [ *freeze_nsets  nsets-names  ]
 [ *freeze_nsets_start_with  nsets-start-with-names  ]
 [ *corners  nsets-names  ]
 [ *corners_start_with  nsets-start-with-names  ]

 [ *output_mmg_files  ]
```

All syntaxes are identical to the ones of `**mmg3d` and the reader is invited to see their description on page 2.90

```
****mesher
 ***mesh
  **mmg2d
```

Example:

```
****mesher
 ***mesh
  **open ../GEOF/carre.geof
  **mmg2d
   *metric scalar
    value 0.1
   *min_size .1
   *max_size 2.
   *hgrad 1.
   *verbose 6
   *preserve_elsets diag mix
   *preserve_elsets_start_with eset
   *freeze_elsets diag
   *preserve_bsets  left top bottom
   *freeze_bsets left
   *freeze_nsets_start_with  x y diag
   *freeze_nsets diag
  **output carre_remeshed.geof
****return
```

**mmgs

Description:

This command is used to remesh and optimize a Zebulon triangular surface mesh by means of MMGS remeshing library, part of the open source MMG platform.

Syntaxes are are identical to the ones of `**mmg3d` (see page 2.90), except for the option `octree` which is not available.

Note that, when the freeze of a bset is asked the resulting mesh may be non-conforming and can lead to an unusable finite element mesh. Moreover, some errors were observed during the prevervation of multiple sets in the interface version (5.3.11) : some elements were not labelled.

Example:

```
****mesher
***mesh
 **open ../GEOF/skin.geof
 **mmgs
  *metric scalar
   value 0.15
  *min_size .05
  *max_size 2.
  *hgrad 1.
  *output_mmg_files
  *verbose 6
  *preserve_elsets  mix
  *preserve_bsets right  left top
 **output skin_remeshed.geof
****return
```

**modify_mesh_and_cut

Description:
This command, used for parallel computations, allows to modify the *problem*.cut and *problem*.cu files.

Syntax:

```
**modify_mesh_and_cut
 *enable_node_renumbering
 *store_node_renumbering
 *store_nodes
```

*enable_node_renumbering writes the ***renumbering option in the *problem*.cut and *problem*.cu files to enable subdomain Metis renumbering (see page 2.87) at runtime. This option should be used only with the sparse direct linear solvers.

*store_node_renumbering renumbers all subdomains using Metis renumbering and writes the new subdomain nodes order in the *problem*.cut and *problem*.cu files.

*store_nodes writes for each subdomain the list of nodes which belong to the subdomain (without any renumbering operation). This list is always written in the *problem*.cut and *problem*.cu files using the ONERA splitmesh program (see page 2.127).

Note:
Since version 8.3 the sparse solvers renumber internally so the node renumbering parts of this command are no longer of importance. The *store_nodes command will however prevent an "on the fly" calculation of the interface nodes, perhaps saving some calculation time.

**nset

Description:
This command creates a node set according to various input data. If the node set already exists, new nodes will simply be added. Nodes will be included in the set only once.

Syntax:

```
**nset name
[ *axes axis1 axis2 axis3 ]
[ *function function ; ]
[ *limit epsilon ]
[ *nodes node1 ... nodeN ]
[ *plane n1 n2 n3 intercept ]
[ *point [nearest] p1 p2 p3 ]
[ *sequence start_id end_id [ increment ] ]
[ *surface ]
[ *type cartesian | cylindrical ]
[ *use_bset bset1 ... bsetN ]
[ *use_elset eset1 ... esetN ]
[ *use_nset nset1 ... nsetN ]
```

`*axes` switches the axes defining a cylindrical coordinate system when using the `*plane` selection. an example is `*axes 1 3 2` where the cylinder is rotated about the 2 axis instead of the default 3-rd axis.

`*function` creates a nset using the nodes which satisfy the given function.

`*limit` *epsilon* specifies the precision for nodes to qualify to be accepted by a function or a plane function (because the values are never *exactly* the same due to numerical noise). The default value is 10^{-3}.

`*nodes` *node1 ... nodeN* adds specific (existing) node numbers from the mesh to the nset (probably useful with `**add_node`). Non-existing nodes will be silently ignored.

`*plane` makes an nset given a plane equation (4 real values). The first three values are the components of the plane normal, and the fourth is the intercept (or its equivalent for cylindrical coordinates).

`*point` p_1 p_2 p_3 adds existing node having the given coordinates *and* those within a radius of *epsilon*. Add the keyword `nearest` if you only want the nearest node to be added.

`*sequence` will add a sequence of nodes with ids from *start_id* to *end_id* with increment *increment* (Z-set 8.4 and newer). The latter can be negative, but in that case, *start_id* should be greater than *end_id*. If this is not the case, -*increment* will be used instead. If *increment* is omitted, a default value of `1` is used. Non-existing node ids generated with this command will be silently ignored.

`*surface` indicates that the set should be an outer bounding surface of the acceptable nodes (see also `unshared_faces` page 2.143).

`*type cartesian | cylindrical` sets the type of coordinate system to be used with the `*plane` option. Default is cartesian.

`*use_bset` *bset1 ... bsetN* apply the nset command to only those nodes in the given boundary sets.

`*use_elset` *eset1 ... esetN* apply the nset command to only those nodes in given element sets (was `*elset`).

`*use_nset` *nset1 ... nsetN* apply the nset command to only those nodes in the named node sets.

Example:
Some example uses follow.

```
**nset ring1 *elset ring1 *plane 0. 1. 0. 4.   % nodes in y=4

**nset fix   *point 20. 0. 0.                  % single point

**nset r=20                                    % the outer radius
 *limit 1.e-3                                  % of a part at R=20
 *type cylindrical                             % with some allowance for
 *plane 1. 0. 0. 20.                           % numerical errors

%
% These switch the cylindrical axes to be axis 2
%
**nset t=10      *axes 1 3 2 *type cylindrical *plane 0. 1. 0. 20.
**nset t=m10     *axes 1 3 2 *type cylindrical *plane 0. 1. 0. 0.

%
% Interpreted functions are perhaps the most useful .. see
%
**nset fix-func *function (z==0.0)*(y>=0.0);
```

**nset_intersection

Description:
This command is used to create the intersection of nsets.

Syntax:
The command has the following syntax:

```
**nset_intersection
 *nsets              nset1-name nset2-name
 *intersection_name  resulting-nset-name
```

`*nsets` is followed by the name of the nsets to intersect

`*intersection_name` is the name of the resulting nset

Example:

```
****mesher
 ***mesh tube.geof
  **open tube_tmp.geof

  **bset skin   % creates the outer skin of the tube
   *surface

  **nset_intersection  % creates the outer skin of the given domain
   *nsets zone1-volume skin
   *intersection_name zone1-skin

****return
```

**open_bset

Description:

This command takes as input a bset (faset in 3D, liset in 2D), and duplicates nodes at the corresponding interface, to insert discontinuities (contact definitions, cracks ...) in the initial mesh. In the output mesh, the following items are available:

- 2 bsets named `SIDE0` and `SIDE1`, obtained by duplicating the input bset for each side of the created discontinuity. Note that bset elements are automatically ordered such that the normal is pointing outside (a property needed if those bset are to be used for contact definitions).
- 2 elsets named `SIDE0` and `SIDE1` with elements connected to the previous bsets.
- set of nodes located at the tip of the discontinuity, and ordered in continuous lisets named `FRONT0`, `FRONT1` ... `FRONT`n, where n is problem-dependent.
- nset `FRONT` with all tip nodes in the previous lisets.

Note that this command is only allowed for linear input meshes (commands `**quad_to_lin`, `**lin_to_quad` may be used to bypass this limitation).

Also note that if the input bset contains duplicated faces, the mesher may produce erroneous results. It is recommended to verify the input bset, and/or use the `cleanup_bsets` command (see p. 2.37) to remove duplicates.

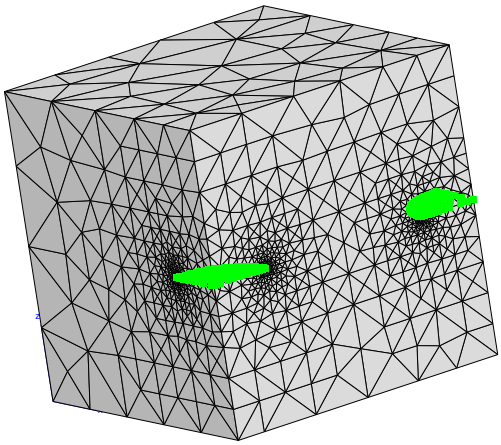
Syntax:

```
**open_bset
 *bset bname
[ *surface sname ]
[ *elset ename ]
[ *create_interface ]
```

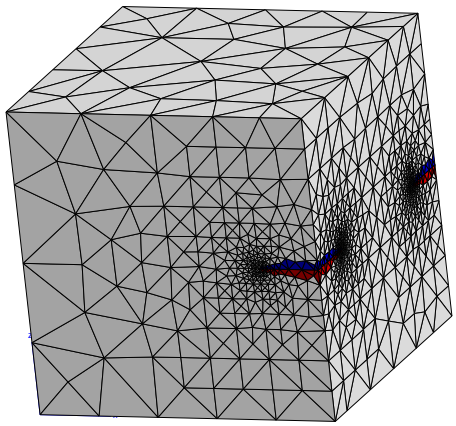
- *bname* is the name of the bset, for which nodes need to be duplicated to create a discontinuity in the input mesh,
- the optional command `*surface` takes as argument the name *sname* of an nset containing surface nodes. In this case surface nodes are removed from the front lisets (the front nodes ending points are preserved, however). You may want to use `**unshared_faces` to generate such an nset. the See examples below.
- when the optional command `*create_interface` is specified, interface elements are automatically added between both sides of the discontinuity. Corresponding interface elements are included in the output `INTERFACE` elset.

Example:

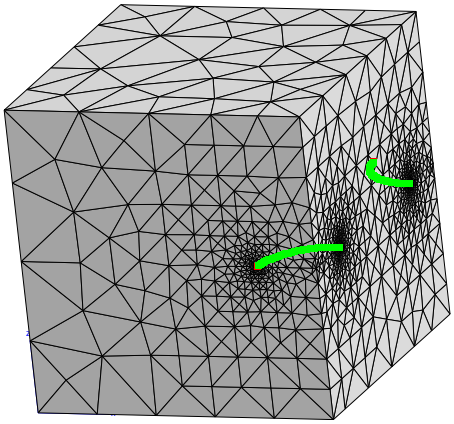
```
**open_bset
 *surface SURFACE
 *bset to_open
 *branches
```



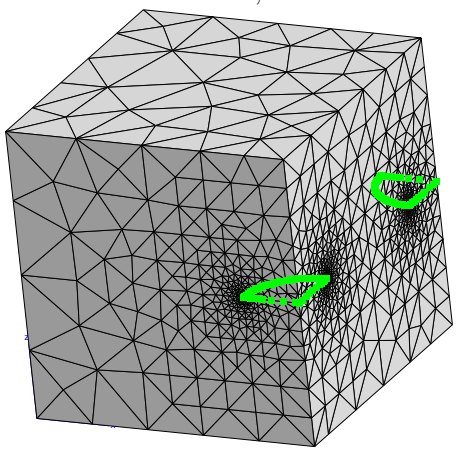
input mesh with bset to open



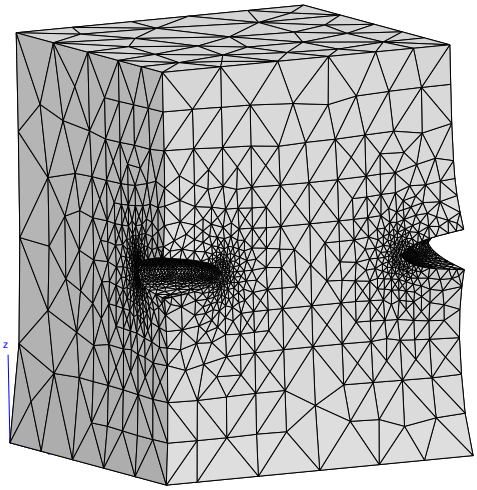
elsets `SIDE0`, `SIDE1`



2 lisets created: `FRONT0`, `FRONT1`



FRONTS without the `*surface` option



example FE results

**parallel_adaptation_x

Description:

This keyword specifies the parallel algorithm used for remeshing a distributed Zebulon mesh. For the moment, two algorithms are available in Z-set (`parallel_adaptation_1` and `parallel_adaptation_2`). Both algorithms rely on a sequential remesher, need a distributed input meshes and must be run in parallel, *e.g* with `Zrun -mpimpi -m mesher.inp`. To facilitate the transfer of finite element fields, the geometry of the interface between subdomains is preserved. Parallel algorithms share all options of `**adaptation` on page 2.10, so only specific options are detailed hereafter. To ensure the correctness of the interfaces, both algorithms are based on the communication of a thin layer of elements.

Syntax:
The command has the following syntax:

```
**parallel_adaptation_x
 [ *dump_each_step ]
 [ *skin_depth depth  ]
 [ options compatible with **adaptation ]
```

`*dump_each_step` writes produced meshes at several steps of the process. Useful in debug.

`*skin_depth` *depth* is the number of element layers used in the algorithm. The default value is 2.

Note that `parallel_adapt_x` shares the limitations of the used sequential remesher (linear elements, tetrahedra, ...). Also, `parallel_adaptation_1` and `parallel_adaptation_2`, use Graph utilities and need External/Koala library.

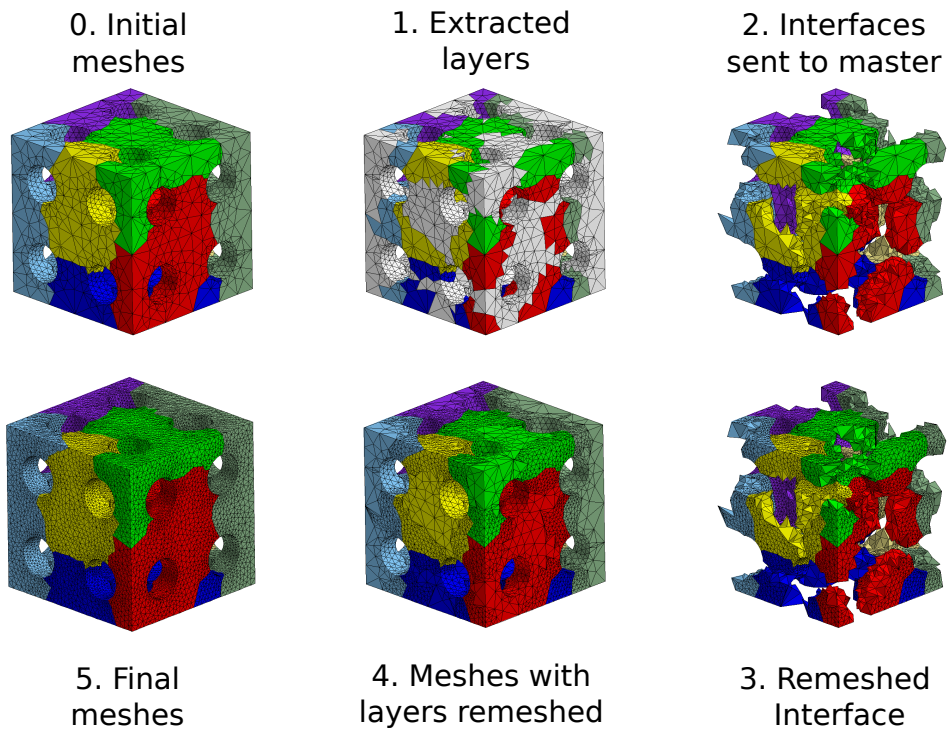
**parallel_adaptation_1

Description:

This command specifies the first parallel algorithm. It is based on a three steps process. First, all subdomains communicate to the master process a thin layer of elements connected to the interface of the subdomain decomposition. Then, the master process assembles all parts, performs a remeshing without touching the external boundary and sends back the remeshed layers to the other subdomains. Finally, all subdomains perform a volumic remeshing. The figure below illustrates the algorithm. For options, please refer to `parallel_adaptation_x`.

Syntax:

```
**parallel_adaptation_1
 [ and all options compatible with **parallel_adapt ]
```



**parallel_adaptation_2

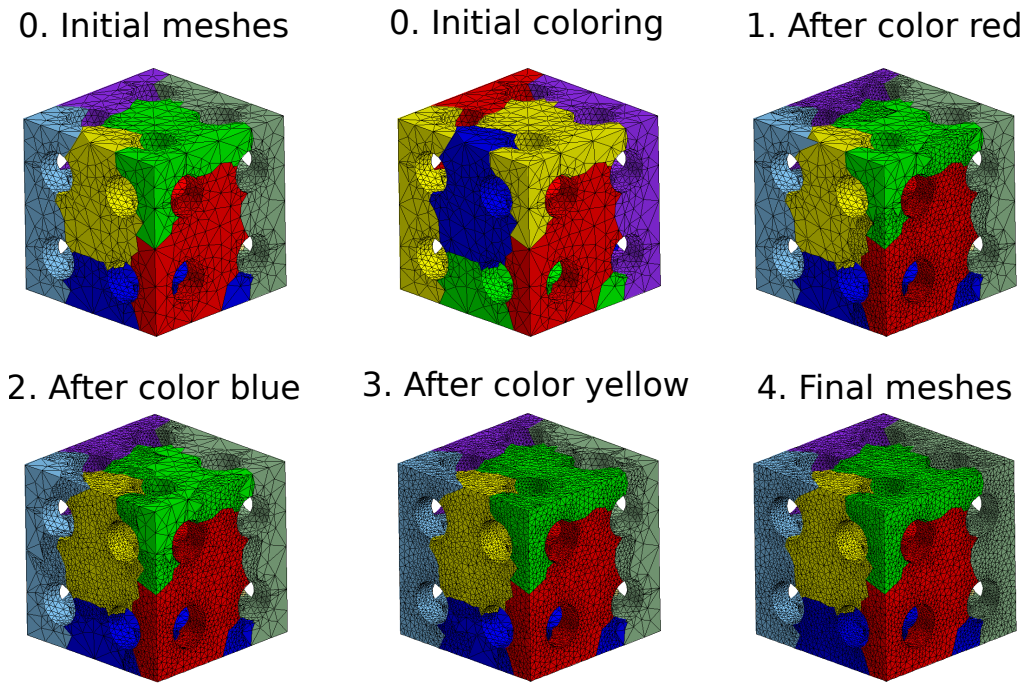
Description:

This command specifies the second parallel algorithm. It is slightly more complex than the previous algorithm but a better scalability is expected. It is based on a color per color algorithm. First, a graph coloring of the graph of domain-domain connectivity is done. Then, a loop on all colors is performed. Subdomains of the current color ask to their neighbors a thin layer of elements, perform the remeshing process, and send back to their neighbors the remeshed layers. Once all colors have been processed subdomains are remeshed in parallel with frozen interfaces. The figure below illustrates the algorithm. For other options, please refer to `parallel_adaptation_x`.

Syntax:

```
**parallel_adaptation_2
 [ *multiple_greedy_algorithm ]
 [ and all options compatible with **parallel_adapt ]
```

`*multiple_greedy_algorithm` enables the use of several greedy graph coloring algorithms in order to provide a lower number of colors.



**perturb_inside

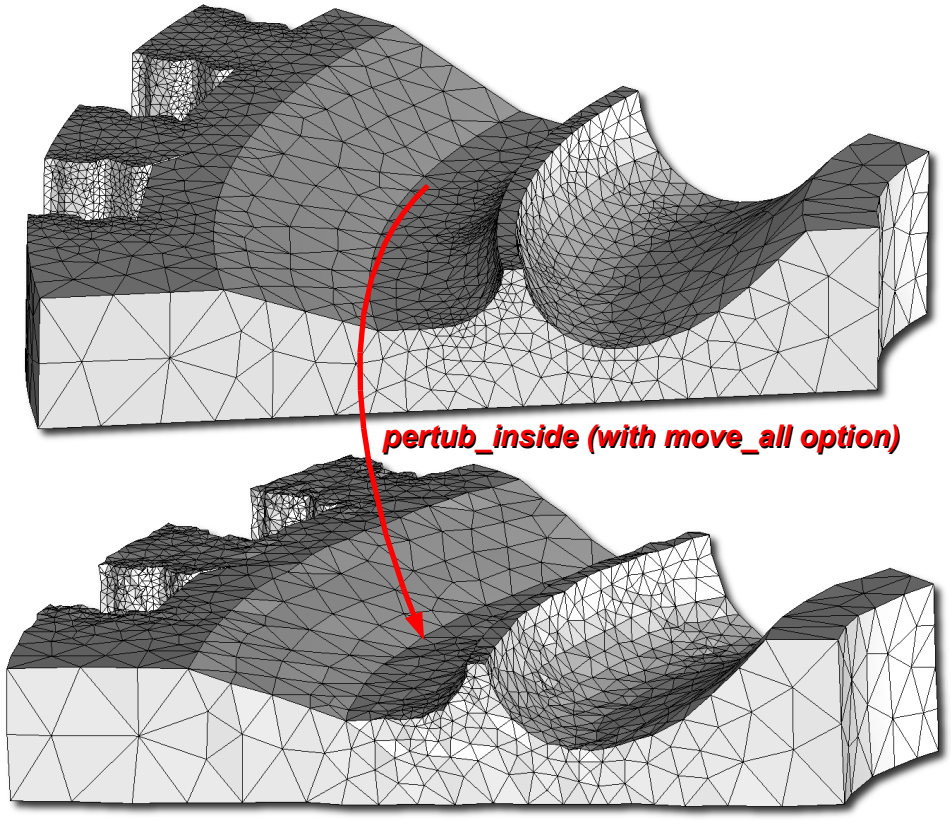
Description:
This command is used to perturb a mesh geometry with a given uniform random law and a deformation factor (works only for 3D mesh if mesh surface must be preserved, e.g. without command `move_all`).

Syntax:
The command has the following syntax:

```
**pertub_inside
 *factor      moving-factor
[ *move_all ]
```

`*factor` gives the moving factor distance.

`*move_all` impose the perturbation to be applied also on the mesh surface (works for meshes in any dimension).



**porcupine

Description:
This command builds a series of pyramids on a bset. This is generally used in conjunction with boolean operations, in order to join quadrangular mesh faces to triangular ones. See e.g. the illustration of **regularize_cfv, on page 2.18, where the middle figure shows such an example (although pyramids where built differently in that particular situation).

Syntax:
The command has the following syntax:

```
 **porcupine
[ *bset    bset-name  ]
[ *height  relative-height  ]
```

*bset is the name of the bset on which pyramids are built. If it is not specified, pyramids are built on the whole mesh skin.

*height is the height of the pyramid. It is a value relative to the perimeter of the base. The default value of $\frac{\sqrt{2}}{2}$ is chosen so that a square-based pyramid has equilateral faces.

Example:

```
****mesher
 ***mesh trench-for-diff.geof
  **open trench.geof           %  contains c3d8 elements
  **porcupine
   *height .2

% It is then used in a boolean operation
 ***mesh difference.geof
  **boolean_operation_ghs3d
   *operation difference
   *file1 earth.geof
   *file2 trench-for-diff.geof
   *output_file difference.gts
   *options        -m 100 -FEM % Memory size (MB)
   *optim_style 1
   *keep_2nd peau_commune

% And finally the ruled mesh is inserted in the hole we just created
 ***mesh earth2.geof
  **open difference.geof
  **union
   *add trench.geof
****return
```

**project_nset

Description:
This command is used to project a given nset on a specified bset. It requires bset to be made of planar faces.

Syntax:
The command has the following syntax:

```
**project_nset
 *nset            nset-to-project
 *bset            bset-to-project-on
[ *distance     ] opening-value
[ *direction    ] vector
[ *orthogonal   ] vector
```

`*distance` insert a distance between projected points and given bset (can be used to separate two surfaces).

`*direction` gives the direction vector followed during the projection (usually projection is done orthogonal to the bset surface).

`*orthogonal` is used to specify a direction on which there must be no displacement of the nset during the process.

**propag_crack

Description:
This command is used to grow 3D cracks described by an explicit mesh of the discontinuity. It verifies various conditions in order to preserve mesh's geometry.

Syntax:
The command has the following syntax:

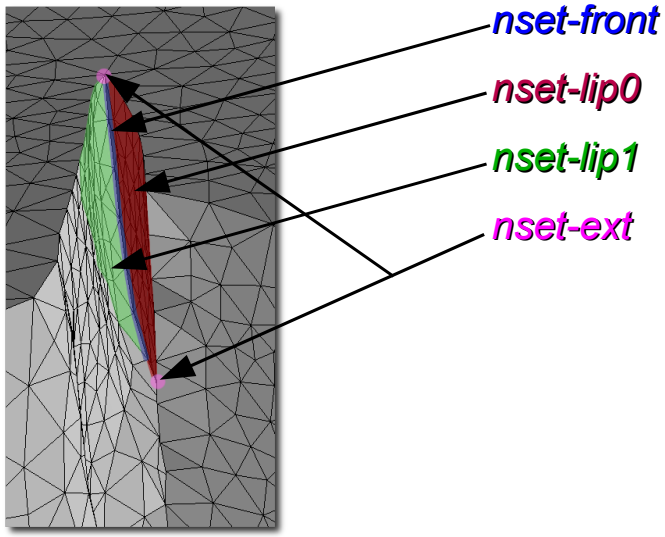
```
**propag_crack
 *crack            front-nset-name lip0-nset-name lip1-nset-name ext-nset-name
[ *ask_speed_to ] crack-component-name
[ *open         ] opening-value
[ *close        ]
```

`*crack` gives various nsets describing crack geometry: front nodes, first and second lips nodes. Last nset gives initial and final crack front nodes for opened cracks.

`*ask_speed_to` gives the name of the crack-front-component which impose the front advance.

`*open` is used to separate the lips with a given distance. This makes the remeshing process easier for the volume automatic mesher.

`*close` is used to collapse the lips (after remeshing if `open` command has been previously used).



**quad_to_lin

Description:
This command reduces the integration order of a mesh to linear. This command can be conveniently applied using the `Zquad_to_lin` script front-end.

Note:
There is also currently (Z-set 8.3 and newer) a mesher `lin_to_quad` (see page 2.79) to go from linear to quadratic. However, curved surfaces will be unrealistically modeled with facets. If there is the possibility of running with quadratic meshes, the basic meshing operations should be set to produce a quadratic mesh, and the mesh should be linearized before use.

Syntax:
The command will be activated simply with the command name.

```
**quad_to_lin
[ *param_file list-of-files ]
```

`*param_file` is an optional list of nodal parameter files, that will be linearized accordingly. Resulting files will be named with a `.lin` suffix.

**randomize

Description:
This command superimposes a random displacement to nodal positions. Such a modification may be useful to model geometric imperfections.

Syntax:

```
**randomize
 *magnitude mag
[ *nset    nset1  ...nsetN ]
[ *elset   elset1  ...elsetN ]
```

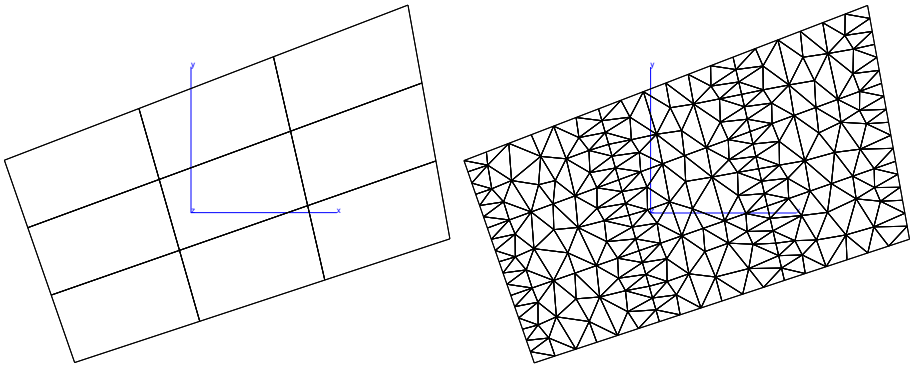
`*magnitude` enter the distance (*real* value) which is the maximum magnitude of the displacement. The random displacement is applied with randomized direction in the order of space the node is given in.

`*nset` apply the random alteration to the given node sets.

`*elset` apply the random alteration to the nodes used in the elements contained in the given element sets.

**refine_mesh_based_on_element_domains

Description:
This command allows one to generate a master file (.mast) suitable to build a refined mesh. Each element of the initial mesh provides the vertices and the edges of the new mesh. A Delaunay domain is generated for each element of the initial mesh. The following figure shows an initial mesh and the new mesh that was built with the generated master file:



Syntax:

```
 **refine_mesh_based_on_element_domains
[ *edges no_edges ]
[ *master_file filename ]
```

`*edges` *no_edges* gives the number of nodes to use for each edge of the refined mesh (default 2).

`*master_file` *filename* is the filename for the master file of the refined mesh (default `refine.mast`).

Note that this command currently (version 8.3) does not conserve the original elsets, and that it is implemented only for 2D linear elements.

If the `***mesh do_not_save` option is omitted, a .geof file will be generated. However, this new .geof file is *identical* to the one that was loaded with the `**open` command (if no other meshing operations have been carried out), and it does *not* contain the refined mesh.

Example:
An example use follows:

```
****mesher
 ***mesh do_not_save
  **open base.geof
  **refine_mesh_based_on_element_domains
   *edges 6
   *master_file fine.mast
****return
```

**regularize_cfv

Description:
This command is used to smooth crack front volume (see) after remeshing process. It imposes a same distance between each node of the front and adapt the geometry of the volume meshed around the front. It can also be used only to smooth an nset around the front using projection on the crack front geometry. It the 3D crack propagation with remeshing process this command should be used after `propag_crack` command and before any remeshing command (`yams_ghs3d`).

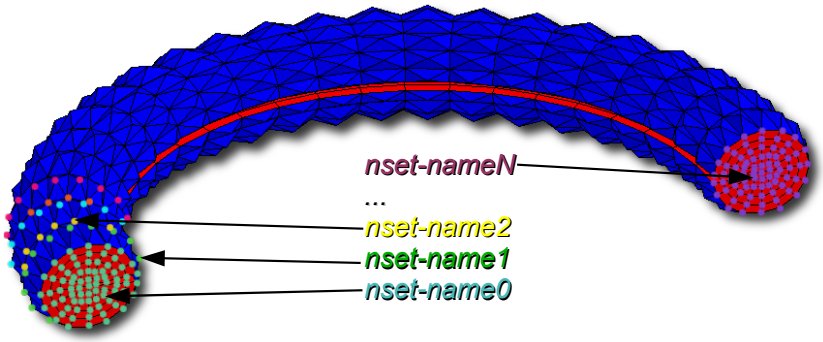
Syntax:
The command has the following syntax:

```
**regularize_cfv
 *liset       crack-front-liset
[ *cut_desc ] nset-name nset-number
[ *nset     ] nset-name
```

`*liset` gives the crack front liset.

`*cut_desc` describes the crack front volume, using such format: `nset-name0`, `nset-name1` ... `nset-name(nset-number-1)`.

`*nset` gives the name on the nset on which regularization should be done (if no CFV is given).



**remove_nodes_from_nset

Description:
This command is used to remove a set of nodes from another given node set.

Syntax:
The command has the following syntax:

```
**remove_nodes_from_nset
 *nset_name nset-name
 *nsets_to_remove nset-list
```

The `*nset_name` is used to indicate the name of the source node set (the node set where the user wants to remove nodes). The `*nsets_to_remove` command lists all node sets to be removed from the first node set.

**remove_orphans

Description:
This command removes “orphaned” nodes from the mesh (nodes which are not attached to anything).

Syntax:
There are no options or parameters for this command.

**remove_set

Description:
This command is used to delete unwanted sets from the active mesh. In contrast to the **delete_elset command (page 2.53), these commands only delete the sets given, the actual mesh entities being pointed to are left as they are.

Syntax:
The command has the following syntax:

```
**remove_set
 *nsets nset-list
 *bsets bset-list
 *elsets elset-list
 *ipsets ipset-list
 *nsets_start_with nset-prefix-list
 *bsets_start_with bset-prefix-list
 *elsets_start_with elset-prefix-list
 *ipsets_start_with ipset-prefix-list
 *null_sets
 *do_all
```

*nsets every nset which name matches exactly with one the given names will be removed.

*bsets idem for bsets.

*elsets idem for elsets.

*ipsets idem for ipsets.

*nsets_start_with every nset which name starts with one of the given prefixes will be removed.

*bsets_start_with idem for bsets.

*elsets_start_with idem for elsets.

*ipsets_start_with idem for ipsets.

*null_sets all empty sets will be removed, i.e. sets not containing any items.

*do_all delete all sets.

**rename_set

Description:
This command is used to rename sets from the active mesh.
(Available in Z8.4)

Syntax:
The command has the following syntax:

```
**rename_set
 [ *sets    old-name1 new-name1 old-name2 new-name2 ...   ]
 [ *nsets   old-name1 new-name1 ...   ]
 [ *bsets   old-name1 new-name1 ...   ]
 [ *elsets  old-name1 new-name1 ...   ]
```

Any number of pairs of names can be given to each sub-command, and any combination of them is possible. As expected, the `nsets` command renames nsets, `bsets` command renames bsets and `elsets` command renames elsets. The `sets` command tries to rename all three kind of sets (and will simply ignore nonexistent sets).

Example:

```
**rename_set
 *sets
   left  hot_boundary       % renames both the nset and the bset
   right cold_boundary
   top   contact_zone
 *nsets
   bottom fixed_border
   577    fixed_node         % will actually create a new nset
```

**renumbering

Description:
This command renumbers the mesh so that the global matrix front or bandwidth will be reduced. The algorithm is a modified Sloan renumbering scheme.

The renumbering by this command is not especially fast, and cannot directly handle disconnected mesh regions. In order to renumber across contact or MPC linked regions, use the `**make_springs` command (see page 2.80).

Note:
It is advisable to do this operation as the last mesher command before saving the mesh. Alternately on UNIX platforms the `Zrenum` command can be used.

Note:
The sparse matrix solvers should employ the metis renumbering described on page 2.87, or use built-in renumbering if available for the solver used.

Syntax:

```
**renumbering
[ *frontal ]
[ *nodal ]
[ *w1 val ]
[ *w2 val ]
[ *subdomain ]
```

`*frontal` Renumbers elements to reduce the front size in a frontal solution.

`*nodal` Renumbers nodes to reduce the bandwidth size in a banded matrix solution (skyline).

`*subdomain` Renumbers according to subdomains for optimizing parallel computations.

**resize_node

Description:
This command is used to set the nodal position to have a desired dimension. Note that since version 8.0, the code supports mixed dimension meshes, and the dimension parameter in the `.geof` file is not used (left for compatibility reasons). If two coordinates are given in the node list, the node is a 2D node, and if three coordinates are given the node is a 3D node.

In Zmaster is is OK to have 3D nodes with 2D meshes, but in the FEA solver an error will be produced during the creation of DOFs and management of the shape functions.

Syntax:
The command has the following syntax:

```
**resize_node
 *nsets nset1 ... nsetN
 *dim   dim
```

The command `*nsets` can optionally be used to specify that the dimension change only apply to those nodes. The `*dim` command specifies the dimension to change to.

**rigid_body

Description:
Some direct solvers, like `frontal` and `sparse_direct`, are able to automatically detect rigid body motions (i.e kernel) in the factorization step of the local matrices $[K]_i$ (see page **??**). But some others, like `sparse_dscpack`, can not automatically detect these rigid body motions, so they have to be indicated to it.

Using this latter solver, the `**rigid_body` command can be used to find the rigid body motions of each sub-domain, using empirical geometrically criteria, and to write them in the *problem*`.cut` and *problem*`.cu` files. It should be noticed that this command just runs in `3D` with `impose_nodal_dof` boundary conditions, in the global coordinate system.

Syntax:

```
**rigid_body
 *nset nset1 ... nsetN
 *dof  dof1  ... dofN
```

`*nset` is the list of all nset names for which an `impose_nodal_dof` boundary condition exists in the `****calcul` module. If several dofs are imposed for an nset, the name of this nset must appear several times.

`*dof` is the list of the correspondent dofs.

Note:
If the parallel computation is not a 3D one, or if other types of boundary conditions are used, the `**rigid_body` command should not be used; and the rigid body motions have to be directly specified at the end of the *problem*`.cut` file for each sub-domain using the following syntax. It should be noticed that in this case, the *problem*`.cu` has to be removed.

```
**bc sd nrb
 node_1 dof_1
 ...
 node_nrb dof_nrb
```

Where

`sd` is the number of the sub-domain for which rigid-body motions are specified.

`nbr` is the number of rigid-body motions for sub-domain *sd*.

`node_i` is a node number.

`dof_i` is the dof name for the rigid-body motion at *node_i* (e.g. U1, U2 or U3).

The information concerning nodes and dofs can be obtained by observing sub-domains, or by running the same computation using a sparse_direct linear solver and by stopping it just after sub-domain matrices factorization.

Note:
The `sparse_dscpack` direct linear solver should not be used if the rigid body motions of a sub-domain change during computation.

**rotate

Description:
This command is used to apply geometrical rotation of node coordinates within a mesh. The rotation will be specified using the same syntax as given for material rotations on page 3.159.

Syntax:

```
**rotate <ROTATION>
[ *elset_name name ]
```

Example:

```
***mesh PIPE
  **open PIPE3
  **rotate
   *elset_name right_side
    x1 1.  0. 0.
    x3 0.  1. 0.
```

Note:
See also **symmetry, page 2.131.

**scale

Description:
This command scales the coordinate values for the mesh by a specified factor. This can be used to change units (e.g. inch-mm, meter-mm, etc).

Syntax:
The syntax is simply the command name followed by a floating point scale factor:

```
**scale factor
```

An extended syntax allows different scaling factors in each direction:

```
**scale xfactor yfactor zfactor
```

**set_reduced

Description:
This command alters the degree of integration used for the element. It is a convenience feature to change quickly whether the elements are integrated fully or with reduced quadrature. With this, one does not need to alter the element type in every domain in Zmaster.

Syntax:

```
**set_reduced
[ *type reduced | normal ]
[ *elset elset1 ... elsetN ]
```

`*elset` select specific elsets to be modified. The default is to modify every element in the whole mesh.

`*type reduced` make the elements reduced integration, irrespective of their current value.

`*type normal` make the elements normal (full) integration, irrespective of their current value.

The default is to set to reduced integration. See also `set_normal` (page 2.123).

Note:
Z-set has two types of reduced quadratic tetrahedron (`c3d10`): one with 5 Gauss points (`c3d10r`), the other with 4 Gauss points (`c3d10_4`, ABAQUS provides this one). `set_reduced` generates the one with 5 Gauss points. Use the command `to_c3d10_4` (page 2.137) to generate the 4 point variant.

**set_normal

Description:
This command is the inverse of `set_reduced` (page 2.122): it forces normal (full) integration. It has the same syntax and sub-commands.

**sequential_ids

Description:
This mesher renumbers a mesh sequentially from given start values. The default start node and element is one[5].

Syntax:
The command has the following syntax:

```
**sequential_ids
 *node_start st-node
 *element_start st-node
```

[5]start numbers are available in 8.2+ only

**small

Description:
Sets nodal coordinates with absolute values smaller than a preset value to zero[6]. This may be useful for instance for adding nsets with a criterion such as

```
**nset nset_name *function (z==0);
```

which will miss points that have very small non-zero z-coordinates due to numerical noise. This command acts on all components of the position vector of the given nodes.

Syntax:

```
**small
[ *limit limiting_value ]
[ *nset  nset_name ]
```

`*limit` enter the limiting value (*real* value) below which the coordinate value will be set to zero. Negative values will have their minus chopped off. The default is `1.e-6`.

`*nset` apply to the given node set. The default is `ALL_NODE`.

[6] Available in 8.2+ only

**sort_nset

Description:
This command sorts a node set in increasing order with respect to the value of a user-specified function of the coordinates of the nodes.

Syntax:
The command has the following syntax:

```
**sort_nset
 *nset_name   nset-to-sort
 *criterion   function  ;
[ *n2_sort ]
```

`*nset_name` specifies the node set to sort. There is no default value.

`*criterion` enter a function of the coordinates in order to sort by. It is a function of two points $P_1\,(x_1, y_1, z_1)$ and $P_2\,(x_2, y_2, z_2)$ within the nset (without the z_1 and z_2 for two-dimensional meshes). It may also compare node ids (the corresponding variables are named `id1` and `id2`). The function must return whether P_1 is "greater" than P_2: criterion = -1 means $P_1 < P_2$ (no action taken), criterion = 0 means $P_1 = P_2$ (no action taken) and criterion = +1 means $P_1 > P_2$ (the order of the points will be changed). Do not forget the semicolon at the end. There is no default value.

`*n2_sort` uses a brute-force $O(n^2)$ sorting method, instead of the $O(n \log n)$ quicksort, which is used by default. This option is included because the quicksort routine is broken on some systems.

Example:

```
**sort_nset
 *nset_name left
 *criterion x1<x2; % sort nset according to increasing x. Note the ;
**sort_nset
 *nset_name radial
 *criterion (x1*x1+y1*y1+z1*z1)>(x2*x2+y2*y2+z2*z2); % sort nset according
                  % to decreasing distance from the origin (0. 0. 0.).
                  % This generates an error message for 2-D meshes.
```

Example:
This second example will reorder opposite faces `perio_x0` and `perio_x1` in lexicographic order, to e.g. ensure they have the same order before applying an `mpc2`:

```
**sort_nset
 *nset_name perio_x0
 *criterion (y1>y2) + (y1==y2)*(x1>x2);
 *nset_name perio_x1
 *criterion (y1>y2) + (y1==y2)*(x1>x2);
```

**split

Description:
This is an interface for the ONERA splitmesh program which can be used to build the subdomain information for a parallel computation. There is a different interface for 8.0 and 8.2. 8.0 requires that the splitmesh executable be installed on your system (not supplied by default), while it is integrated into 8.2.

Syntax:
For 8.0 the command has the following syntax:

```
**split
 *splitmesh_location path
 *domains num
 *mincon  int-value
 *no_elset
```

`*splitmesh_location` specify the path to the splitmesh executable program.

`*domains` enter the number of domains (separate processes) in a parallel computation.

`*no_elset` don't generate the element sets for domain visualization.

The syntax for 8.2 is the same, except that the `*splitmesh_location` is not required (or available).

Example:
An example use follows from the test `Parallel_test/INP/arm2.inp`

```
****mesher
  ***split
    **splitmesh_location /home/saturne/feyel/bin/splitmesh
    **domains 4
  ***renumbering
    **subdomain
  ***geof_format
    **unformat
****return
```

**sweep

Description:
This command extends planar (2D or 3D shell/face) geometries into 3D through a sweep rotation. One may give an arbitrary rotation axis, point to make quite arbitrary sweeps. The method works for linear/quadratic meshes in the positive or negative rotation directions. Degenerate cases are handled (inside edge is on the axis, so the inner elements become pyramids, etc).

Extension may be given with different start/end meshes. In this case, the first mesh is swept forward, while the second swept back. The mid point nodes are the interpolation between the two. Please be careful concerning the order of elements between the two meshes (domains) in this case. Their topology must be equivalent.

If no elset is given the whole mesh will be used.

Note that `sweep` is derived from `extension` (see page 2.61), thus shares most of its functionalities.

Syntax:

```
 **sweep
[ *elset  eset1 ]
[ *elset2 eset2 ]
[ *new_elset new-elset ]
[ *angle  degrees-rotation ]
[ *num    num ]
[ *prog   progression ]
[ *axis   dir-vec ]
[ *center cent-vec ]
[ *fusion dist ]
[ *create_faset ]
```

`*elset` *eset1* defines the element set or faset which will be extended. There is no limitation on the type of elements or faces contained on the face. Mixed order is not allowed however. The default is all elements.

`*elset2` *eset2* This optional command makes the extension between two mesh/face sets as given. **Note that the mesh topology must be exactly the same.** This command is most useful with the mapped mesh domains available in Zmaster. The degenerate case of 2 edges being the same (so the extended mesh forms a wedge) is handled. Do not use `*distance` with this option.

`*new_elset` *new-elset* This optional command gives the elset name for the created elements instead of being the same as the input name.

`*angle` The angle to sweep through in degrees. May be positive or negative. The default is 360 degrees.

`*num` is the number of elements in the extension direction (default is 1 element).

`*axis` Use this command to specify the rotation axis. in vector form. The default is `(0. 1. 0.)`.

`*center` Use this command to specify the rotation center in vector form. The default is `(0. 0. 0.)`.

`*fusion` specifies the distance below which 2 nodes are considered identical. Default value is that of the global parameter `Mesher.MeshFusion`.

`*create_faset` The two facing sides of the extended mesh are created and named `face.1` and `face.2`.

**switch_element

Description:
This command is used to inverse the element axis orientation. Note that the element orientation can be visualized in Zmaster.

Syntax:
The command has the following syntax:

```
**switch_element
 *elset elset-name
 *axis axis
```

The options for this command are described below:

`*elset` is the elset to which the inversion is applied

`*axis` Give the axis around which the rotation is performed

**symmetry

Description:
This command is used to do a symmetry of a nset. Its modifies nodal positions by a symmetry's transformation (central, axial or planar), described with a point and a normal. The type *line* is only valid for 2D meshes (input vectors should have 0. as z-coordinate), and the type *plane* is only valid for 3D meshes.

Syntax:
The `**symmetry` command takes the following syntax:

```
**symmetry
 *type    [point | line | plane]
 *point   origin-point
 *normal  normal-vector
[ *nset   nset ]
```

`*type` Defines the type of symmetry's axis : a point, a line or a plane.

`*nset` The nset whose you want to transform (default value: `ALL_NODE`).

`*point` This defines the origin for the type of symmetry.

`*normal` Normal's direction for the projection of the symmetry.

Example:
The following example shows the use of symmetry of the $x-z$-plane for all points:

```
**symmetry
 *type plane
 *nset ALL_NODE
 *point  (0.,0.,0.)
 *normal (0.,1.,0.)
```

**thicken_bset

Description:
This command is used to add a layer of elements on a specified bset. This is sometimes necessary before using a boolean operation (page 2.18).

Syntax:
The command has the following syntax:

```
**thicken_bset
  *bset       bset-name
  *height     height-of-elements
[ *direction  vector  ]
[ *towards    point   ]
```

`*bset` elements are attached to the specified bset. There is currently no default.

`*height` specifies the height of the newly built elements (default is 1).

`*direction` specifies a direction for the extension.

`*towards` is an alternative to `*direction`: elements are built towards the specified point (or away from, if height is negative).

Example:

```
****mesher
 ***mesh trench.geof
  **open trench_for_difference.geof
 [...]
  **thicken_bset
   *bset TheSurface
   *height -30.
   *towards (0. 0. 0.)
****return
```

**to_2d

Description:
This command transforms 3d meshes into 2d meshes. It only modifies the dimension of node location vector.

Syntax:
The command has the following syntax:

```
**to_2d
[ *s3d_to_c2d ]
```

`*s3d_to_c2d` after transformation the resulting elements will be converted into 2d elements, option supports only initial surface meshes

**to_3d

Description:
This command transforms 2d meshes into 3d meshes. It only modifies the dimension of node location vector.

Syntax:
The command has the following syntax:

```
**to_3d
```

`*c2d_to_s3d` after transformation all 2d elements will be converted into 3d surface elements

**to_cax

Description:
This command transforms 2D plane meshes into axisymmetric meshes.

Syntax:
The command has the following syntax:

```
**to_cax
```

In practice, this command simply changes the element types, from `c2d` to `cax`.

**to_c2d

Description:
This command transforms axisymmetric meshes into 2D plane meshes.

Syntax:
The command has the following syntax:

```
**to_c2d
```

In practice, this command simply changes the element types, from `cax` to `c2d`. It does the opposite of `to_cax`.

**to_c3d10_4

Description:
This command transforms `c3d10` elements (quadratic tetrahedral) to `c3d10_4` elements (reduced quadratic tetrahedral with 4 integration points).

Syntax:
The command has the following syntax:

```
**to_c3d10_4
```

In practice, this command simply changes the element name, from `c3d10` to `c3d10_4`. See also `set_reduced`, page 2.122.

**transform_fili

Description:
This command is used to correct a mesh previously imported from ABAQUS and generated by the SAFRAN FILI tool. It replaces degenerated elements with Z-set standard ones (prism and pyramid elements).

In some cases this mesher may fail on quadratic meshes, so it is recommended to work on linear meshes only (see the `lin_to_quad` mesher on page 2.79).

Syntax:
The command has the following syntax:

```
**transform_fili
[ *elset  elset-name ]
```

`*elset` specifies the elset where the operation is applied (default is to correct the whole mesh).

**translate

Description:
This command is used to apply a rigid body translation of the mesh.

Syntax:
The syntax consists of the command name followed by the 3 translation components:

```
**translate  dx  dy  dz
```

The command also has an extended syntax that provides additional functionalities:

```
**translate
 *x  dx
 *y  dy
 *z  dz
 *nset   nset-name
 *nodes  node1 ... nodeN
 *elset  elset-name
```

`*x, *y, *z` specifies the translation

`*nset` restricts the translation to the given nset

`*nodes` restricts the translation to the given node ids

`*elset` restricts the translation to the given elset

Example:

```
**translate 0. -5.5 -1.0
```

**unconnected_parts

Description:
This command creates one elset per disconnected part of an input mesh: a new list of elsets is created (part1, ..., part$N$), with N the number of disconnected parts of the input mesh.

Syntax:

```
**unconnected_parts
```

The command has no options.

Example:
An example is shown below:

```
****mesher
 ***mesh
  **open sphere_cylinder.geof
  **unconnected_parts
****return
```

**union

Description:
This command creates the union of meshes.

Syntax:

```
**union
 *add  mesh-name
[ *elset  eset-name ]
[ *merge_nset merge ]
[ *set_change_name chg-name ]   % 8.2 (obsolete)
[ *set_base_name   base-name ]  % 8.2 w(obsolete)
[ *base_name   base-name ]
[ *tolerance  val ]
[ *translate_new_meshes ]  % 8.2 (obsolete)
[ *tr trans-vector ] % 8.2 (obsolete)
[ *translation trans-vector ]
```

`*add` *mesh-name* adds the mesh in file *mesh-name*`.geof` to the active mesh.

`*elset` *eset-name* A new element set will be created with the name *eset-name* and containing all the elements added in this operation.

`*merge_nset` limit the fusion of nodes to the given node set. The node set is on the current mesh, not the newly added mesh given by *mesh-name*.

`*base_name` rename all sets from *old-name* to *base-name.old-name*.

`*tolerance` Gives the critical distance for node fusion between the two meshes. If too large, elements will collapse because node neighbors are joined, too small and there may be gaps at the interface. Use zero to join two meshes without node fusion, such as in the joining of two parts in contact. The default variable is in the global parameters, probably 1.e-3 (in the dimension of the mesh).

`*translation` Translates the new elements by *trans-vector*. *trans-vector* must be input in the form *(x y z)*

Example:
A short example follows of some union sections from the test case in
`Mesher_test/INP/PIPE.inp`

```
**union
 *add pipe-data/PIPE1.geof
 *elset PIPE1
 *tolerance 0.15
**union
 *add pipe-data/PIPE2.geof
 *elset PIPE2
 *tolerance 0.15
 *translation (0. 10. 0.)
```

**unshared_edges

Description:
This command makes an edge boundary set (liset) for the outer edges of a mesh. It operates similarly to the `**unshared_faces` for 2D meshes, and produces an edge output.

Syntax:

`**unshared_edges` *bset-name*
`*elset` *elset1 ... elsetN*

`*elset` used to specify a number of elements sets to find the unshared faces rather than the whole mesh. Note this still gives the faces which are not shared with any other element, whether in the listed elsets or not. It is just that the faces checked are a sub-set of the total mesh, and the resulting set will be on the named elsets only.

**unshared_faces

Description:
This command makes a boundary set (which can be converted to an nset using **nset**) from the outer boundary of a given element set, or of the whole mesh.

Syntax:

**unshared_faces *bset-name*
[*elsets *elset1* ... *elsetN*]

---

*elsets used to specify a number of elements sets to find the unshared faces rather than the whole mesh. Note this still gives the faces which are not shared with any other element, whether in the listed elsets or not. It is just that the faces checked are a sub-set of the total mesh, and the resulting set will be on the named elsets only.

Note:
This command is currently faster than the **bset *bset-name* *surface equivalent.

Note:
This command replaces the deprecated and ambiguous **unshared command.

**volume_to_shell

Description:
Starting with a 3D volume mesh, this command extracts the given bset and creates the corresponding surfacic 3D-shell mesh.

Syntax:
The command has the following syntax:

```
**volume_to_shell
[ *bset  bset-name ]
```

`*bset` specifies which bset to extract. The default is to extract the whole mesh skin.

**yams_ghs3d

Description:

This command can be used for remeshing an input Zebulon mesh by means of the DISTENE remeshing tools. There are basically 3 different pieces of software that may be used to build the new mesh:

- i) `Yams` is used for remeshing a surface mesh (that may be built from the skin of a 3D mesh or directly from a shell mesh) according to various criterions,
- ii) `Ghs3d` is a general purpose 3D mesher that takes as input a surface mesh and fills the volume with tetrahedra,
- iii) given a metric map, `Meshadapt` performs either surface, volume, or surface+volume adaptation at the same time. The metric map is made of the desired sizes of vertices (edges) connected to each vertex (node) of the mesh. Since this map is given on the initial mesh, several iterations may be needed in the adaptation process.

Those tools are fully interfaced for Zebulon and the `yams_ghs3d` command automatically writes out DISTENE input and reads in DISTENE output files to build the Zebulon mesh. DISTENE binaries are included in the standard Zebulon distribution (in the `$Z7PATH/PUBLIC/lib-$Z7MACHINE/Zmesh/` folder), but an optional license key is needed to run those software. `Yams`, `Ghs3d`, and `Meshadapt` manuals are also available in the `Zmesh/` folder, and the user may have a look at those if a fine tuning of the remeshing process is needed.

Note that only linear elements are handled by `yams_ghs3d`. A combination of the `quad_to_lin` (before `yams_ghs3d`) and `lin_to_quad` commands (after) may then be used to allow the use of quadratic meshes. Warning: if a computation result is available, the mesh will be deformed before remeshing.

Syntax:
The command has the following syntax:

```
**yams_ghs3d
 [ *yams_only ]
 [ *ghs_only ]
 [ *nb_iter_surf  niter-surf  ]
 [ *nb_iter_vol  niter-vol  ]
 [ *nb_iter  niter  ]
 [ *force_meshadapt ]
 [ *refinement_origin  origin-nset  ]
 [ *refinement  func(x,y,z) ;  ]
 [ *refinement_file  fname  ]
 [ *absolu ]
 [ *min_size min  ]
 [ *max_size max  ]
 [ *gradation grad  ]
 [ *tolerance tol  ]
 [ *optim_style opt  ]
```

```
[ *preserve_faset bset-name1 bset-name2 ...   ]
[ *preserve_liset bset-name1 bset-name2 ...   ]
[ *yams_options   yams-options  ]
[ *ghs3d_options   ghs3d-options  ]
[ *meshadapt_options   meshadapt-options  ]
```

`*yams_only` forces surface remeshing only (ie. performing only step i)).

`*ghs_only` forces volume remeshing only (ie. performing only step ii)).

`*nb_iter_surf` if a metric map is given, this command allows to specify the number *niter-surf* of iterations involved in step i) to satisfy a given metric map for the surface mesh (default is *niter-surf*=3).

`*nb_iter_vol` if a metric map is given, this command allows to specify the number *niter-vol* of iterations involved in step iii) to satisfy a given metric for the volume mesh (default is *niter-vol*=2),

`*nb_iter` this command can be used to define both *niter-surf* and *niter-vol* at the same time (*niter-surf*=*niter-vol*=*niter* in this case).

`*force_meshadapt` when a metric map is given as input to drive the remeshing process, this option has the effect to force the use of `Meshadapt` instead of `Yams` during step i). Note that in this case `Meshadapt` is run only 1 time during step i) (option -O1 for surface adaptation) and *nb_iter* times during step ii) (option -O3 for surface+volume adaptation),

`*refinement_origin` this command is used to define the name *origin-nset* of an nset taken as a base to calculate the metric map according to function *func(x,y,z)* (see next command),

`*refinement` this command defines the function *func(x,y,z)* used to compute the metric map (see explanations in the `Metric map calculation` paragraph). Note the only variables allowed in this function are `x`, `y` or `z` and that the definition should end with a `";"` character.

`*refinement_file` with *fname* the name of a metric map file (the file syntax is explained in the `Metric map calculation` paragraph).

`*absolu` is a `Yams` option and is only used during step i). In this case edge sizes (arguments of the `*min_size` or `*max_size` commands, and/or values given in the metric map) are *absolute* values. The default alternative is to use *relative* values (ie. sizes are scaled by size of the bounding box of the initial mesh, see the `Yams` user manual).

`*gradation` *grad* is a `Yams` only option (step i) that may be used to control the element size variation (please refer to the `Yams` user manual).

`*tolerance` *tol* is a `Yams` only option (step i) used to set the value *tol* of the maximum chordal deviation tolerance (please refer to the `Yams` user manual).

`*optim_style` *opt* is a `Yams` only option (step i) and can be used to specify the type of optimization style (coarsening, enrichment etc... please refer to the `Yams` user manual). Default is *opt*=2 for geometrical mesh enrichment.

`*preserve_XXset` used to preserve set(s) during surface remeshing. This can only be applied to set(s) located on the mesh skin and it does not mean that the *exact* set(s) topology should be kept (set(s) is/are rebuilt after the remeshing process: position and element type could have changed).

`*yams_options` , `*ghs3d_options` and `*meshadapt_options` commands are meant for advanced DISTENE tools users, and allow to specify manually the DISTENE software command line options (see user manuals in the Zmesh/ folder). Default values are in general appropriate for typical Zebulon applications.

Calculation of the metric map

The metric map is defined by a set $\mathcal{P}$ of base points with the target element sizes near those points:

$$\mathcal{P} \;=\; \{\; P_i(x_i, y_i, z_i)\;,\; d_i \quad ; \quad i = 1, ..., n_b \;\}$$

where n_b if the number of base points, P_i a particular point in $\mathcal{P}$ with coordinates (x_i, y_i, z_i) and a prescribed size of d_i.

$\mathcal{P}$ may be defined either by commands `*refinement_origin` or `*refinement_file`:

`*refinement_origin` *origin-nset* fills $\mathcal{P}$ with points corresponding to nodes in nset *origin-nset*,

`*refinement_file` *fname* defines the name of a file containing P_i, d_i. Syntax of this ASCII file is the following one:

- first line:
  n_b (number of base points: 1 integer)
- line 2 to $n_b + 1$
  $x_i \;\; y_i \;\; z_i \;\; d_i$ (definition of P_i coordinates and prescribed size: 4 double values)

Then for each node N in the current mesh, the prescribed size $d(N)$ of element edges built from node N is calculated in the following way:

- first, find closest point P^* from N in set $\mathcal{P}$:
  P^* such that $distance(P^*, N) \;=\; Min\,\{\; distance(P_i, N) \;\;,\;\; i = 1, ..., n_b \;\}$
- then, if a function *func(x,y,z)* is given by means of command `*refinement`:

  $$d(N) \;=\; func(x^*, y^*, z^*)$$

  with (x^*, y^*, z^*) the coordinates of point P^*,
- else:

  $$d(N) \;=\; d^*$$

  with d^* the prescribed size given for point P^* in $\mathcal{P}$.

- finally, if either commands `*min_size` *min* or `max_size` *max* are specified:
  if $d(N) < min$ then $d(N) = min$
  if $d(N) > max$ then $d(N) = max$

Note that the option `*absolu` is advised in this case to insure that $d(N)$ values calculated do correspond to actual edge sizes in the finite element mesh.

Example:

See tests in folder $Z7PATH/TESTS/Distene_test/INP/.

```
****mesher
 ***mesh output
  **open input.geof
  **yams_ghs3d
   *force_meshadapt
   *nb_iter 3
   *absolu
   *min_size 0.03
   *max_size 0.2
   *preserve_faset haut-ext bas-ext U1=0 U3=0
   *refinement_origin bas-ext
   *refinement x*x+y*y+z*z;
****return
```

**yams_by_elset

Description:

This command is derived from **yams_ghs3d mesher, and can be seen as a "mesher modifier". It allow to preserve elsets topology during yams_ghs3d remeshing process. This is achieved by internally providing a "color" for each named elset. The colored areas borders topology are then "protected".

Syntax:
The command has the following syntax:

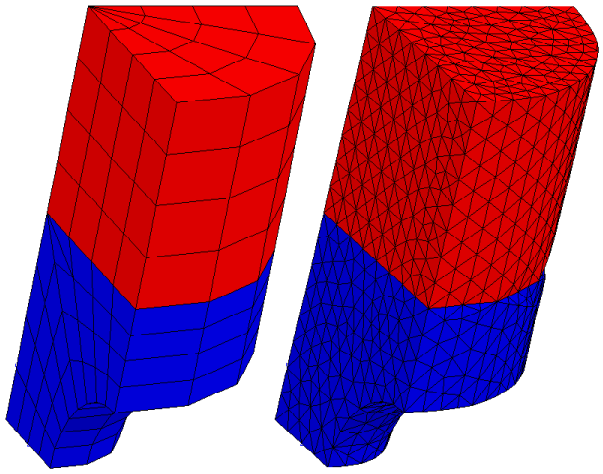
```
**yams_by_elset
  [ *elset elset-name1 elset-name2 ... ]
[ current "**yams_ghs3d" options]
```

*elset names of elsets to be preserved.

Example:

See tests in folder $Z7PATH/TESTS/Distene_test/INP/.

```
****mesher
 ***mesh cyl4
  **open cyl.geof.ref
  **yams_by_elset
   *elset center attach
  %%% yams_ghs3d options
   *force_meshadapt
   *nb_iter_surf 1
   *nb_iter 3
   *absolu
   *min_size 0.1
   *max_size 0.1
   *refinement 0.1;
****return
```



**refine_elset

Description:

This command is derived from **yams_ghs3d mesher, and can be seen as a "mesher modifier". it allow to remesh a single elset using yams_ghs3d remeshing process, without modifying neighboring element. This is achieved by extracting the target elset in a separate mesh and imposing the conservation of the interface (interface : the skin shared between the elset and the remaining of the mesh).

Syntax:
The command has the following syntax:

```
    **yams_by_elset
     [ *elset elset-name ]
     [ *material_elset material_elset_name ]
   [ current "**yams_ghs3d" options]
```

*elset name of elset to be remeshed.

*material_elset this command can be used for multi-material models. It will add all elements created by remeshing elset "elset_name" to elset "material_elset_name".

*maxsize (inherited from **yams_ghs3d) note that by default the element max size is automatically calculated as the maximum size of element at the interface of remeshed elset note that

Example:

See tests in folder $Z7PATH/TESTS/Distene_test/INP/.

```
****mesher
 ***mesh cyl3
  **open cyl.geof.ref
  **refine_elset
   *elset center
   *material_elset ALL_ELEMENT
  %%% yams_ghs3d options
   *nb_iter 3
   *absolu
   *min_size 0.03
   *preserve_faset haut-ext bas-ext U1=0 U3=0
   *refinement_origin bas-ext
   *refinement y*y;
****return
```

**remesh_from_results

Description:

This mesher allow to build a metric map (for Distene tools) based on the results of a previous computation. The metric map definition can be found in `**yams_ghs3d` section or in the Distene manuals. It can be useful to refine a mesh according to the gradient of given field or an a posteriori error estimator.

In this mesher, the metric map can be partially computed from results over a defined elset, the remaining of the map (corresponding to elset complement to the mesh) is simply filled with the actual mesh metric. We can also in this case choose a conservative remeshing based on `refine_elset` (outside elset will remain unchanged) or a less conservative remeshing based on `yams_ghs3d` (outside elset will change but keeping the same characteristic element length).

Syntax:
The command has the following syntax:

```
**remesh_from_results
  [ *result_name result-file-name ]
  [ *Z8 ]
  [ *var var-name ]
  [ *card card-number ]
  [ *power power-coefficient ]
  [ *free_interface ]
[ current "**refine_elset" options]
```

`*result_name` results file name

`*Z8` add it to specify that results are in Z8 format (.zres folder). if omitted Z7 format is assumed

`*var` name of the variable in the results database to be used for metric computation.

`*card` index of the results card to be used for metric computation.

`*power` modify the value of the variable `var` such that $var = var^{power}$

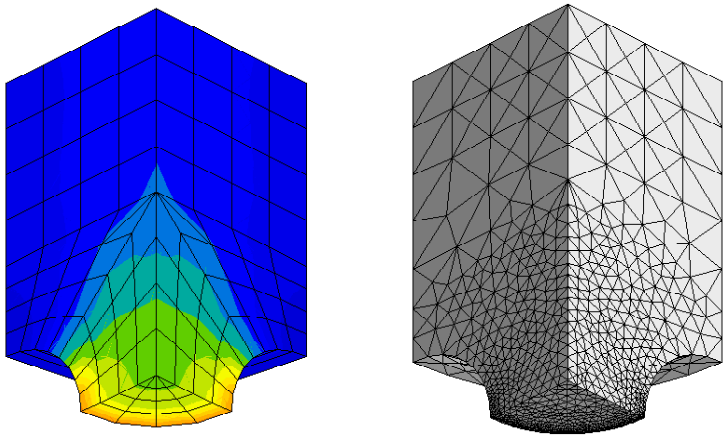
`*free_interface` if set, activate the "less conservative remeshing" otherwise the `refine_elset` strategy is used

Example:

See tests in folder $Z7PATH/TESTS/Distene_test/INP/.

```
****mesher
 ***mesh cylr4
  **remesh_from_results
```

```
                                                    ****mesher
                                                     ***mesh
                                                      **remesh_from_results

   *result_name CYL/cyl
   *var sig22
   *card 2
   *power 0.5
   *free_interface
  %%% refine_elset/yams_ghs3d options
   *elset center
   *absolu
   *min_size 0.01
   *max_size 0.15
   *nb_iter 3
   *preserve_faset haut-ext bas-ext U1=0 U3=0
****return
```



Chapter 3

Finite Element (.inp file)

Introduction

Description:
This chapter concerns the definition of finite element problem solutions with the Zebulon solver, and is indicated under the main command `****calcul`. The FEM problem definition will be given by a primary file with name *problem*`.inp`, and also several auxiliary files, as required. The prefix *problem* will be used in other input and output file names, and is henceforth called the problem name. Generally, a geometry specification file is required with the default name *problem*`.geof`, where the nodes, elements, and other geometrical information is given. It should be noted that the geometry file does *not* describe the type of element (formulation), which will be defined in the `.inp` file. Frequently a separate material file is given as well, but this information may be added to the `.inp` file just as easily.

The FEM calculation mode of Z-set reads and interprets all the non-commented instructions in the main `.inp` file between the commands `****calcul` and `****return`. Other problem modes may therefore be, in the same file, demarcated by commands starting with four asterisks.

In the file `.inp` we may give references to the names of other files for the material files, geometry files, or other data files such as external temperature fields, initial values for the material variables, etc. These files will be discussed throughout the command summary here, and in the section on file formats (page 6.10).

An overview of the functions to which the `.inp` file serves is given below:

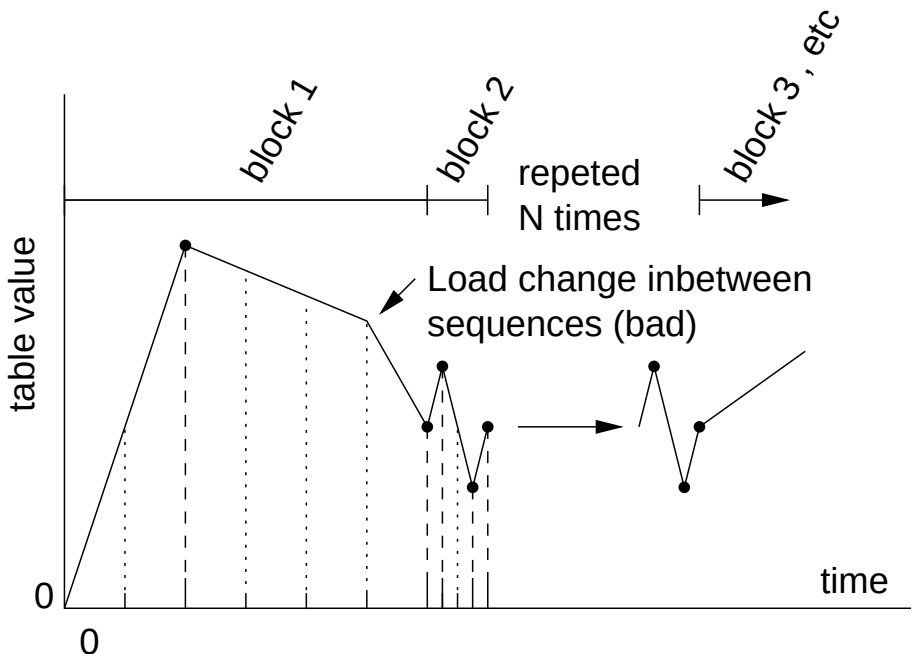
- Define the nature of the calculation: mechanical, thermal steady-state, thermal transient, etc.
- To give complementary information of the geometry or formulation of the calculation. For example, if a 2D problem is in plane stress or plane strain conditions, in finite strain formulations, etc.
- Give the boundary conditions. These include specification of the degrees of freedom and the associated forces (e.g. displacement and point forces), as well as calculated conditions such as pressure or centrifugal forces. With the exception of eigen frequency analysis, the boundary conditions must all be prescribed in the time scale of the calculation. In the case of a static mechanical calculation, the time may not have a physical significance, but is rather used as a fictitious measure of the loading processes. For example, in order to calculate a static structure in three loading cases, we can write the boundary condition evolutions as a function of dimensionless times 1, 2, and 3.
- Assign groups of elements (element sets or *elset*) a material file specifying the material law and its coefficients. Each material file is also assigned an integration method if necessary. The separation of the finite element model and numerical integration allows one to work with a small base of material files for many calculations with different numerical methods.
- Define the method(s) of global solution, and define the solution steps which must be made in a non-linear calculation. This data will include the algorithm used (Newton-

Raphson, BFGS,...) the convergence criteria, and automatic time stepping methods. The time used here defines the time measure used for all other parts of the problem definition such as the boundary conditions.

- To define tables as a function of time. Specification of numerous parameters in the calculation such as the boundary condition amplitudes will be defined by these tables.
- Specify output for whole-model fields, and directly computed curves which are generated during the calculation. Multiple output "blocks" may be specified, with different frequencies of output, etc.

Linear solution:
Linear solutions typically involve a number of "loading cases" which are normally superimposed to a structures response to many different external solicitations. There is now a special key given in the output section `**linear_solution` (see page 3.173).

Time stepping:
The success of a nonlinear calculation depends on definition of appropriate loading sequences. The following figure summarizes the basic elements of loading sequences:



This example shows the waveform desired for a particular parameter in the calculation (such as a loading level). On the x-axis is the time scale of the calculation, broken into segments (long dashed lines) and increments (dotted lines). The y-axis represents the loading level described by a hypothetical table. The sequences are put into groups any of which can be repeated to give cyclic loadings.

The function of sequences is to demarcate segments of a particular solution strategy in the linear progression of time from t_1 to t_2 with $t_2 > t_1$. During the sequence the solution parameters such as convergence criteria, method of stiffness matrix resolution, etc are fixed.

Increments which divide the sequence are used to refine the incremental solution in a linear manner (normally through steady changes in loading). The increment in time during each loading increment is therefore $\Delta t = \frac{t2 - t1}{n}$ for n increments. For nonlinear solutions, within each increment convergence may require several trial solutions which will be called *iterations*.

An alternative to the linear division is to give progressively increasing or decreasing steps, or automatic calculation of the increments. This later allows stepping as a function of the problem variables for accuracy control, as well as in function to the convergence rate, etc for divergence control.

As the loading magnitudes are given in terms of time, and not directly in terms of the segment numbers, incompatibility in the load waveforms with the sequences is available. This is schematically shown in the above figure within the second load sequence. Here the load changes at the end of the third increment of the second sequence, which is one increment away from the sequence end. This type of loading is **not** advised, and should be cautiously verified in the case of complex loadings (e.g. cyclic or otherwise lengthy).

With these terms defined, we may progress with the command summary for the FEM problem loading. To clarify their meaning however, it may be very useful to follow several of

the example problems given in the example section of the manual.

Specifying node sets:
Node sets are defined in the `.geof` file, which is created by meshing operations. In commands which require a **nset**, give the character name for the node set desired, as appears in the `.geof` file. An additional shortcut exists for the case of specifying *all* the nodes in the mesh. In this case, the keyword `ALL_NODE` may be used for the node set name.

Specifying element sets:
Element sets are defined in the `.geof` file as well, defining lists of element numbers which may be identified together. These sets will be used to assign particular element formulations to regions in the mesh, material behaviors, boundary conditions, etc. As for node sets, element sets are identified by their full name given in the `.geof` file. The entirety of elements in a mesh may be identified with the shortcut keyword `ALL_ELEMENT` to be used in the place of a user-defined set name.

Specifying magnitudes:
The magnitudes of commands (particularly boundary conditions) will be controlled using either tabular or functional values. The definition of tables and functions are treated as a major element of the user input file definition (at the `***` level). The commands to assign tables is `***table` and functions `***function`, each of which will create an object or objects with a name attribute. Giving names to the functions / tables provides a means for access in other objects which will use them to calculate magnitudes in time. The program structure is such that tables or functions may be referenced before their definition, so the `***table` and `***function` commands may appear anywhere in the valid three asterisk command level under `****calcul`.

The code now accepts pre-defined magnitude names. The most useful of these[1] is `time` which provides a magnitude equal to the current time.

[1] in fact, the only current one

Specifying command parameters:
Many of the commands (especially boundary conditions) take particular parameters to define the actual command applied. These may be character string names, integers, floating point values, vectors, etc. The program is rather strict about typing these data, in order to ensure that what is read is indeed what was intended. The following summary is thus provided to define the syntax data types:

- *character values* character values refer to alphanumeric character strings, which may include some symbol characters such as `- = +`, but not delimiting characters such as spaces or commas. The string is terminated at the next newline, space, tab, comma, or comment character `% or #`. Character values may be up to 255 characters long, but will *never* be shortened.

- *integer values* these are numeric fields separated by the standard delimiters (as in character strings), and must only be composed of the digits 0-9 with no decimal points. Decimal values found for integers will produce an error message.

- *real values* real values are used to specify decimal numbers. They may be positive, or negative, and *must* include a decimal point. Exponential notation is allowed. Some examples are `1.29  5.e+6.1 0.314159E-1`.

- *vector values* vectors define either a point in 2D or 3D space, or the components of a vector direction in space. The size of the vector should be coherent with the spatial dimension of the problem. Vectors are always surrounded by parenthesis `( )` and take only real data for their components. example: `( 1. .5 .5)`.

****calcul

Description:
This command marks the beginning of a FEM calculation definition. The Z-set program in FEM mode will search this command and interpret all the sub-commands until the termination token `****return` is reached. A keyword following the `****calcul` token will indicate the type of calculation which is to be made.

Syntax:
The calculation will be defined using the following syntax:

```
****calcul  type
      options
****return
```

The calculation *type*s which are possible are listed below:

CODE	DESCRIPTION
`mechanical`	static mechanical (no inertial effects)
`eigen`	eigen frequency analysis
`dynamic`	mechanical with inertial effects (implicit)
`explicit_mechanical`	explicit solver for structural dynamics
`thermal_steady_state`	stationary thermal calculation
`thermal_transient`	transient thermal calculation
`diffusion`	Ficks Law diffusion analysis with multiple phases
`weak_coupling`	Generalized coupled analysis

In the absence of a *type*, the default will be `mechanical`.

The basic components of the allowable input data after `****calcul` are summarized below[2]:

[2]More options may exist. Later versions of the code output all available command names with the `-H` switch.

CODE	DESCRIPTION
***mesh	used to specify the types of elements in a mesh, or give an alternate geometry file name
***restart	requests that the calculation be continued from a previous stored result
***resolution	used to declare the solution procedures including the loading sequences
***equation	declare relationships between entities (multi-point constraints) within the calculation
***impose_kinematic	imposes a geometrical evolution for problems which do not have displacement variables but do have an integration volume which changes (thermal, diffusion).
***sub_problem	used to define a sub-problem in the sequential weak coupling algorithm; may be post calculations or re-meshing operations as well.
***parameter	allows specification of externally calculated (given) parameters which may be used to alter the material characteristics during the calculation
***contact	defines surfaces of possible contact and the method of enforcement in the event there is contact
***bc	specify both the geometrical and force boundary conditions
***table	specify the tabular loading magnitudes for parameters and boundary conditions
***function	specify functional loading magnitudes for parameters and boundary conditions in terms of the time
***material	gives the information for material files as attached to element sets, local integration methods, material rotations, and initial variable values
***output	used to specify the desired output from the analysis; multiple output sections can be given to optimize the solution storage

****calcul dynamic

Description:
This option of the `****calcul` command indicates that dynamic effects should be taken into account. The solution procedure is either an implicit d-form of the Newmark time integration scheme or the α-method form Hilber, Hughes and Taylor. These methods are compatible with all mechanical element formulations. Most of the following explanations can be found in [hughes87] and [belytschko00]

The semi-discrete equation of motion (discretized in space, continuous in time) is written as:

$$\mathbf{M}\mathbf{a}(t) + \mathbf{F}_{damp}(\mathbf{v}(t)) + \mathbf{F}_{int}(\mathbf{d}(t)) = \mathbf{F}_{ext}(t) \quad (1)$$

where $\mathbf{d}(t)$, $\mathbf{v}(t) = \dot{\mathbf{d}}(t)$ and $\mathbf{a}(t) = \dot{\mathbf{v}}(t)$ are the vectors of nodal displacement, velocity and acceleration respectively. $\mathbf{M}$ is the mass matrix and $\mathbf{F}_{damp}$ and $\mathbf{F}_{ext}(t)$ are the nodal vectors of damping and external forces respectively. The following initial conditions hold:

$$\mathbf{d}(0) = \mathbf{d}_0 \quad (2)$$
$$\mathbf{v}(0) = \mathbf{v}_0 \quad (3)$$

The linear (or linearized) form of equation (1) reads:

$$\mathbf{M}\ddot{\mathbf{d}} + \mathbf{C}(\dot{\mathbf{d}}) + \mathbf{K}(\mathbf{d}) = \mathbf{F}_{ext} \quad (4)$$

where $\mathbf{C}$ and $\mathbf{K}$ are the damping and rigidity (or tangential rigidity) matrices respectively.

Newmark schemes:

The methods of the Newmark family consist in discretizing equation (4) in time in the following way:

$$\mathbf{M}\mathbf{a}^{t+\Delta t} + \mathbf{C}\mathbf{v}^{t+\Delta t} + \mathbf{K}\mathbf{d}^{t+\Delta t} = \mathbf{F}_{ext}^{t+\Delta t} \quad (5)$$
$$\mathbf{d}^{t+\Delta t} = \widetilde{\mathbf{d}}^{t+\Delta t} + \Delta t^2 \beta \mathbf{a}^{t+\Delta t} \quad (6)$$
$$\mathbf{v}^{t+\Delta t} = \widetilde{\mathbf{v}}^{t+\Delta t} + \Delta t \gamma \mathbf{a}^{t+\Delta t} \quad (7)$$

where the predictors (known from the previous increment) $\widetilde{\mathbf{d}}^{t+\Delta t}$ and $\widetilde{\mathbf{v}}^{t+\Delta t}$ are defined by:

$$\widetilde{\mathbf{d}}^{t+\Delta t} = \mathbf{d}^t + \Delta t \mathbf{v}^t + \frac{\Delta t^2}{2}(1-2\beta)\mathbf{a}^t \quad (8)$$
$$\widetilde{\mathbf{v}}^{t+\Delta t} = \mathbf{v}^t + \Delta t (1-\gamma)\mathbf{a}^t \quad (9)$$

$\mathbf{d}^t$, $\mathbf{v}^t$ and $\mathbf{a}^t$ are approximations of $\mathbf{d}(t)$, $\dot{\mathbf{d}}(t)$ and $\ddot{\mathbf{d}}(t)$ respectively. Parameters β and γ determine the stability and accuracy of the algorithm.

There are several possible implementations of the Newmark algorithm. The one used in Z-set is the d-form, meaning that the equations are solved in terms of $\mathbf{d}$ (*i.e.* not in terms of $\mathbf{a}$ or $\mathbf{v}$). The linear system to solve then reads:

$$\left(\frac{1}{\beta\Delta t^2}\mathbf{M} + \frac{\gamma}{\beta\Delta t}\mathbf{C} + K\right)\mathbf{d}^{t+\Delta t} = \mathbf{F}_{ext}^{t+\Delta t} + \left(\frac{1}{\beta\Delta t^2}\mathbf{M} + \frac{\gamma}{\beta\Delta t}\mathbf{C}\right)\widetilde{\mathbf{d}}^{t+\Delta t} - C\widetilde{\mathbf{v}}^{t+\Delta t} \quad (10)$$

Note that the choice $\beta = 0$ (corresponding to the explicit Newmark algorithm if $\mathbf{M}$ and $\mathbf{C}$ are diagonal) is not suitable for the d-form.

α-method (HHT):

The α-method introduced by Hilber, Hughes and Taylor owns to a more general class of integration schemes called linear multisteps methods (LMS). The α-method is only slightly different from the Newmark one: the update equations (6-9) are conserved, the difference lies in the time-discrete equation which now reads:

$$\mathbf{M}\mathbf{a}^{t+\Delta t} + (1-\alpha)\mathbf{C}\mathbf{v}^{t+\Delta t} + \alpha\mathbf{C}\mathbf{v}^{t} + (1-\alpha)\mathbf{K}\mathbf{d}^{t+\Delta t} + \alpha\mathbf{K}\mathbf{d}^{t} = (1-\alpha)\mathbf{F}_{ext}^{t+\Delta t} + \alpha\mathbf{F}_{ext}^{t} \quad (11)$$

By setting $\alpha = 0$, the Newmark family of time integration methods is recovered. The α-method is usually used with the following set of parameters:

$$\gamma = \frac{1}{2} + \alpha \qquad \beta = \frac{(1+\alpha)^2}{4} \qquad \alpha \in \left[0, \frac{1}{3}\right] \quad (12)$$

The reasons of this choice will be explained in the following section.

Linear problems:

For linear problems, the behaviors of the Newmark and α-methods are well known. To select the appropriate set of parameters (α, β, γ), three points are of particular importance:

- stability (conditional or unconditional)
- order of convergence
- controllable algorithmic dissipation of the high-frequency modes

The last attribute if often desirable in structural dynamics problems. High-frequency modes are poorly approximated by the spatial finite element discretization. By employing algorithms with high-frequency dissipation, spurious high-frequency response is damped out. Properties of some classical methods are summarized in Table 1. Generally, the stability condition reads:

$$\text{unconditional} \qquad 2\beta \geq \gamma \geq \frac{1}{2} \quad (13)$$

$$\text{conditional} \qquad \gamma \geq \frac{1}{2}, \quad \beta < \frac{\gamma}{2} \quad \Rightarrow \quad \omega^h \Delta t \leq \Omega_{\text{crit}} \quad (14)$$

Where ω^h corresponds to the highest pulsation of the spatially discretized problem (h is related to the spatial discretization, *i.e.* to the size of the finite elements) and Ω_{crit} depends on the physical damping parameter. It can be shown that ω^h is bounded by the maximum element pulsation $\omega^h \leq \omega_e^h$ which increases when h decreases. As a consequence, for conditional stability, the critical time step decreases with decreasing element size. Moreover, selecting $\gamma = \frac{1}{2}$ ensures a second order accuracy but adds no damping of high-frequency modes. Selecting $\gamma > \frac{1}{2} = \frac{1}{2} + \alpha$ allows artificial (purely numerical) damping of spurious high frequency modes. This damping is maximized for $\beta = \frac{(1+\alpha)^2}{4}$. Within the Newmark framework, this choice leads to a first-order accuracy whereas a second-order accuracy is achieved with the α-method, and this is the main advantage of the α-method compared to the Newmark one. Note that damping

Method	β	γ	Stability condition	order of accuracy	
Central difference	0	$\frac{1}{2}$	$\omega^h \Delta t \leq 2$	2	not suitable for the d-form
Linear acceleration	$\frac{1}{6}$	$\frac{1}{2}$	$\omega^h \Delta t \leq 2\sqrt{3}$	2	no HF damping
Fox-Goodwin	$\frac{1}{2}$	$\frac{1}{2}$	$\omega^h \Delta t \leq \sqrt{6}$	2	no HF damping
Average acceleration (trapezoidal rule)	$\frac{1}{4}$	$\frac{1}{2}$	unconditional	2	no HF damping
Newmark modified average acceleration ($\alpha > 0$)	$\frac{(1+\alpha)^2}{4}$	$\frac{1}{2} + \alpha$	unconditional	1	HF damping proportional to α
α-method ($\alpha > 0$)	$\frac{(1+\alpha)^2}{4}$	$\frac{1}{2} + \alpha$	unconditional	2	HF damping proportional to α

Table 1: Properties of some classical methods for linear dynamics. Stability conditions are given for physically undamped problems.

is proportional to α and that low-frequency modes are affected more strongly for higher values of α.

Nonlinear problems:
The notion of stability and accuracy developed for linear dynamics are not sufficient for nonlinear problems. The use of the previous rules does not guarantee stability.

Syntax:

Dynamic calculations accept a subset of the `****-level` commands, with an additional `***init_velocity` command used to specify initial velocity conditions. The dynamics specific commands are found in the following table:

CODE	DESCRIPTION
`***resolution`	same as for the static case with specification of the time integration scheme parameters α, β and γ within the sequence definition (see **sequence)
`***init_velocity`	set up initial velocities

The command `***init_velocity` has the following syntax:

```
***init_velocity
   dof_name  elset  elset_name  value
```

By default, the initial velocity is set up to 0 everywhere.

The selection of the integration scheme (Newmark or α-method) and the parameter definition can be done within the `**sequence` block (see page 3.216) through optional commands

*alpha, *beta and *gamma. All other **sequence sub-commands are available and remain unchanged.

```
   **sequence  [ N ]
[  *alpha    val1 [ val2, valN ] ]
[  *beta     val1 [ val2, valN ] ]
[  *gamma    val1 [ val2, valN ] ]
```

These commands can be used with the following increasing level of description:

- none of these command is used. The α-method is selected with $\alpha = 0.05$, $\beta = \frac{(1+\alpha)^2}{4}$ and $\gamma = \frac{1}{2} + \alpha$.
- only *alpha is used. This selects the coefficients of the α-method with $\beta = \frac{(1+\alpha)^2}{4}$ and $\gamma = \frac{1}{2} + \alpha$.
- only *alpha and *gamma are used. This selects the coefficients of the α-method with $\beta = \frac{(\frac{1}{2}+\gamma)^2}{4}$.
- *alpha, *beta and *gamma are used. The three parameters are defined independently.

To select the Newmark integration scheme, α must be set to 0. Due to its unconditional stability and good convergence rate, it is recommended using the α-method in accordance with relations (12), even for non-linear problems.

Example:
An example implicit dynamic calculation follows. In the first sequence a Newmark average acceleration scheme is selected. In the second one, an α-method with $\alpha = 0.1$ has been chosen. Initial velocity of nset INIT1 is set up to 0.1. Note the expiring boundary condition to move and then release a load point.

```
****calcul dynamic
 ***mesh updated_lagrangian_plane_strain
 ***resolution
  **sequence
   *dtime         1.0 1.0
   *increment     10  10
   *alpha         0.  0.1
   *ratio absolu 1.e-6
   *algorithm     p1p2p3
 ***bc
  **impose_nodal_dof
      wall       U2   0.0
      wall       U1   0.0
      load  exp U2  -1.0 tab
 ***table
  **name tab
   *time    0.0   1.
   *value   0.0   1.
 ***init_velocity
```

```
    U1 elset INIT1 0.1
 ***output
  **curve dynam_ul.test
   *node_var 6 U2
 ***material
   *file  ../MAT/dynam_ul
****return
```

****calcul mechanical_explicit

Description:

This option of the ****calcul command is used to activate the explicit solver. Such a solver can be used to model and capture short time scale phenomena such as waves propagation. Explicit solvers can although be used for rough problems like crash, impact or metal sheet forming where contact occurs on large surfaces (contact is very rough from a computational point of view).

Central difference explicit schemes:
The central difference explicit scheme can be derived from the Newmark integration scheme (eq. (5)–(9) of the ****calcul dynamic section) with $\beta = 0$ and $\gamma = 0.5$. The a-form of the integration scheme then reads:

$$\left(\mathbf{M} + \frac{\Delta t}{2}\mathbf{C}\right)\mathbf{a}^{t+\Delta t} = \mathbf{F}_{ext}^{t+\Delta t} - \mathbf{C}\mathbf{v}^{t+\Delta t/2} - K\mathbf{d}^{t+\Delta t} \tag{15}$$

with

$$\mathbf{v}^{t+\Delta t/2} = \frac{1}{\Delta t}\left(\mathbf{d}^{t+\Delta t} - \mathbf{d}^{t}\right) \tag{16}$$

The following expression follows from equations (6)and (9):

$$\mathbf{a}^{t} = \frac{1}{\Delta t^2}\left(\mathbf{d}^{t+\Delta t} - 2\mathbf{d}^{t} + \mathbf{d}^{t-\Delta t}\right) \quad \text{and} \quad \mathbf{v}^{t+\Delta t/2} = \mathbf{v}^{t-\Delta t/2} + \Delta t\mathbf{a}^{t} \tag{17}$$

hence the name "central difference" scheme. Note that $\mathbf{d}^{t+\Delta t}$ and $\mathbf{v}^{t+\Delta t/2}$ are known from the previous step at time t (equations (17) and (16)) so that the right hand side of (15) is known.

In order to make this scheme explicit, a diagonalization of $\mathbf{M}$ and $\mathbf{C}$ is performed (also called a lump) so that the solution of equation (15) is trivial. Each time step is therefore solved very quickly through a simple matrix vector product and there is no linear system to solve. The basic algorithm is described below:

1. initiate $\mathbf{d}^0$ and $\mathbf{v}^0$
2. compute $\mathbf{a}^0 = \mathbf{M}^{-1}\left(\mathbf{F}_{ext}^0 - \mathbf{K}\mathbf{d}^0 - \mathbf{C}\mathbf{v}^0\right)$ and $\mathbf{v}^{1/2} = \mathbf{v}^0 + \frac{\Delta t}{2}\mathbf{a}^0$
3. enforce velocity boundary conditions
4. increment time from $t - \Delta t$ to t and compute $\mathbf{d}^t = \mathbf{d}^{t-\Delta t} + \Delta t\mathbf{v}^{t-\Delta t/2}$
5. compute $\mathbf{a}^t = \mathbf{M}^{-1}\left(\mathbf{F}_{ext}^t - \mathbf{K}\mathbf{d}^t - \mathbf{C}\mathbf{v}^{t-\Delta t/2}\right)$
6. compute $\mathbf{v}^{t+\Delta t/2} = \mathbf{v}^{t-\Delta t/2} + \Delta t\mathbf{a}^t$
7. if computation not finished, go to 3

The drawback of explicit algorithms lies in the stability condition that imposes that the time step Δt is bounded by a critical time step Δt_{crit}. The stability criterion for explicit central difference method reads (damping has no effect on stability):

$$\omega^h \Delta t \leq 2 \quad \Rightarrow \quad \Delta t_{crit} = 2/\omega^h \tag{18}$$

where ω^h is the highest natural frequency of the discretized structure. Computing ω^h is very expensive since it requires solving a large eigen values system. It can be shown that ω^h is bounded by the maximum frequency of individual elements:

$$\omega^h < \omega^h_{el} \tag{19}$$

Let us take the example of a linear beam element with stiffness $\mathbf{K}$ and lumped mass matrix $\mathbf{M}$ such that:

$$\mathbf{K} = \frac{ES}{h}\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} \qquad \mathbf{M} = \frac{\rho h S}{2}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{20}$$

where h, S, E and ρ are the element length, section, Young's modulus and mass density respectively. The non-zero solution of the eigen value problem $K - {\omega^h_{el}}^2 M$ leads to $\omega^h_{el} = 2/h\sqrt{E/\rho} = 2c/h$ where c is the wave speed. The critical time step can therefore be written:

$$\Delta t_{crit} = h/c \tag{21}$$

This corresponds to the time for the wave to go through the element. This interpretation can be verified for every kind of finite element. In practice, this last remark is used in Z-set to evaluate Δt_{crit}. The characteristic size of the element is taken as the minimum distance between two nodes of the element. The wave speed is taken to be the longitudinal wave speed (the fastest one) which reads :

$$c_L = \sqrt{\frac{\lambda + 2G}{\rho}} \tag{22}$$

where λ and G are the Lamé coefficients. An important property of finite elements applied to hyperbolic problems is that Δt_{crit} is $O(h)$ whereas it is $O(h^2)$ for parabolic problems. This makes explicit methods difficult to apply to parabolic problems such as heat conduction or diffusion in general. It is important to note that the critical time step is governed by the size of the smallest element in the mesh. If there is only one spurious small element (due to a bad meshing for instance), the computational cost would increase.

Explicit algorithms are very interesting in the sense that no linear system need to be solved. However, a major limitation lies in the $O(h)$ critical time step. Explicit methods are therefore well suited to model problems where small time steps are imposed by the physics. If there is no need for small time steps, implicit methods may be economically competitive.

Non linear problems:
Extension to non-linear problems is direct and equation (15) is replaced by:

$$(\mathbf{M} + \Delta t\mathbf{C})\,\mathbf{a}^{t+\Delta t} = \mathbf{F}^{t+\Delta t}_{ext} - \mathbf{F}^{t+\Delta t}_{int}(\mathbf{d}^{t+\Delta t}, \mathbf{v}^{t+\Delta t/2}) \tag{23}$$

In the algorithm presented above, the computation of $\mathbf{F}^{t+\Delta t}_{int}(\mathbf{d}^{t+\Delta t}, \mathbf{v}^{t+\Delta t/2})$ is performed between steps 3 and 5.

The stability condition (18) emanates from an analysis of linear equations. At this time, there is no stability theorem that covers the range of nonlinear phenomena such as contact-impact. Instability cannot be overlooked in linear problems since the solution tends to grow exponentially. However, this is not always true for nonlinear equations. A good way to detect

instabilities is to check the energy balance and make sure that no spurious energy is created. The default stability criterion used in Z-set is defined as:

$$|W_{\text{kin}} + W_{\text{int}} + W_{\text{dmp}} - W_{\text{ext}}| \leq \varepsilon \max(W_{\text{kin}}, W_{\text{int}}, W_{\text{ext}}) \tag{24}$$

where W_{kin} is the kinematic energy, W_{int} and W_{ext} are works done by internal and external forces respectively, and W_{dmp} is the energy dissipated by damping. An absolute criterion can also be used :

$$|W_{\text{kin}} + W_{\text{int}} + W_{\text{dmp}} - W_{\text{ext}}| \leq \varepsilon \tag{25}$$

ε is a small value defined with the `**sequence *ratio` command. The absolute criterion is selected by using the *absolute* option (see below). If the energy balance is not verified, the sub-step is recomputed with half time step.

Syntax:

The resolution procedure is specified by the usual way using the `***resolution` command. Specific commands are therefore added to control some parameters. Here are the different options available within the resolution block:

```
***resolution
 **sequence [ N ]
[ *ratio [absolute] val1 [ val2 ... valN ] ]
[ *beta val1 [ val2 ... valN ] ]
[ *fixed_dt val1 [ val2 ... valN ] ]
[ *max_dt val1 [ val2 ... valN ] ]
[ *min_dt val1 [ val2 ... valN ] ]
[ *damping val1 [ val2 ... valN ] ]
[ *max_successive val1 [ val2 ... valN ] ]
[**show_gauge]
```

`**sequence` has the same syntax as in `****calcul` (see page 3.216) with additional commands. Note that the time stepping is not defined by the number of increments set in `**sequence` but is either automatically computed by Z-set (by default) or explicitly specified using `*fixed_dt`. If the time increment is larger than the algorithm time step, sub-stepping occurs. The number of sub-steps performed within each sequence increment is displayed in the standard output.

`*ratio [absolute]` *value*, defines ε in (24) (or in (25) if the keyword `absolute` is used).

`*beta` *value*. If the critical time step is computed by Z-set, the effective time step is taken to be $\Delta t_{crit} \times value$. Values between 0.8 (default) and 0.9 are usual.

`*fixed_dt` *value*, specifies the time step to use. This value is not affected by `*beta`.

`*max_dt` *value*, specifies the upper bound of the time step

`*min_dt` *value*, specifies the lower bound of the time step. If the time step is lower than *value*, an error message is sent and the computation is stopped.

`*damping` *value*, add constant diagonal components in the damping matrix $\mathbf{C}$. This command is similar to the command `***explicit **damp` (see below).

*max_successive *value*, specifies that if the energy balance is not verified after *value* successive sub-steps (dividing each successive time step by 2), Z-set will stop and send an error message. The default value is set to 50 which is a relatively large value.

**show_gauge allows to display the sub-stepping evolution within an increment. A percent value is displayed in the standard output. This can be useful to get an idea on how fast the computation goes when a lot of sub-steps are performed.

Additional options are also available through the ***explicit command.

```
***explicit
 **damp [constant] value
 **every_update value
```

**damp [constant] *value*, is similar to *damping. Each **damp command is cumulative. At the moment, the only damp type is "constant" (default).

**every_update *value*, specifies that the critical time step is computed every *value* time steps. The default value is set to 20. Recomputing the critical time step is useful within a finite strain calculation, where large deformation of some elements can indeed modify Δt_{crit} (*i.e.* the time for the wave to go through the element can decrease or increase). However, this can be time consuming, please check CPU time at the end of the computation.

Example:
An example explicit dynamic calculation follows.

```
****calcul explicit_mechanical
  ***mesh plane_strain

 ***resolution
  **sequence
   *time 10. 20.
   *increment 10 30
   *fixed_dt 0.05 0.01
   *ratio   1.e-2
   *max_successive 10. 20.
   *beta 0.9 0.8

  **show_gauge

 ***explicit
  **every_update 1
  **damp constant 1.e-3

 ***bc
  **impose_nodal_dof
     left        U2   0.0
     left        U1   0.0
```

```
  **pressure
     right       0.00001  echelon

 ***table
  **name  echelon
   *time    0.0 1.e-10   10000000.
   *value   0.0 1.0      1.0

 ***material
  **elset R1
   *file  dynam_unit
****return
```

****calcul eigen

Description:
This option of the `****calcul` command indicates that eigen value solution of the natural frequencies are to be calculated. The solution of the eigen-value problem is controlled using the command `***eigen` (see page 3.119).

Eigen mode calculations accept a sub-set of the above commands, with an additional command `***eigen` used to give parameters of the eigen problem solution. The allowable commands are found in the following table:

CODE	DESCRIPTION
`***mesh`	same as above
`***eigen`	give eigen solution parameters and resolution method
`***equation`	same as above
`***bc`	same as above; only fixed conditions are allowed
`***material`	same as above
`***output`	specifies what output to save at the end of an eigen solution; this is a sub-set of the above

Example:

An example file for Eigen solution follows.

```
****calcul eigen
 ***mesh
  **elset solid small_deformation
  **elset springs spr1
 ***eigen lanczos
    6 0.01 2.
***bc
   **impose_nodal_dof
     base   U1 0.00
     base   U2 0.00
     base   U3 0.00
 ***output
 ***material
   *file building.mat
****return
```

****calcul thermal_transient

Description:
This option of the ****calcul** command indicates that the problem is transient thermal analysis.

Example:

```
****calcul thermal_transient
  ***resolution
    **sequence
      *time   10. 8010.0
      *increment 10  100
      *iteration  50 50
      *ratio 0.01 0.01
      *algorithm p1p1p1
  ***bc
    **fluconv  exte
      h  232.5
      Te 1000.0 tab1
  ***init_dof_value
      TP uniform 20.
  ***material
      *file ../MAT/TTLL_02_89
  ***table
    **name tab1
      *time  0.    50000.
      *value 1.    1.
  ***output
    **curve
      *precision 4 *small 1.e-4
      *node_var 1 TP
      *node_var 16 TP
****return
```

****calcul diffusion

Description:
This option of the ****calcul** command indicates that the problem is a diffusion one, obeying Ficks law.

CODE	DESCRIPTION
***impose_kinematic	used to allow the geometry to evolve which can be very useful for coupled analysis

****calcul weak_coupling

Description:
The weak-coupling implementation uses an iterative approach to an arbitrary coupled problem. Any number of sub-problems are solved (thus keeping the individual systems small) with the coupling taking place through appropriate transfer of results between problems. For example, a mechanical problem can transfer internal heat generation due to mechanical dissipation to a thermal problem which calculates the resulting temperature field evolution which is re-transferred to the mechanical problem to allow coefficient alteration.

Syntax:
Some special commands are of interest for the coupled problem. These define the sub-problem files (standard `.inp` files), and the method of determination for convergence of the coupled problems.

`***resolution` define the global time steps to be run; convergence parameters for the sub-problems are determined in their input files.

`***coupled_resolution` additional convergence parameters specific to the coupled solution.

`***sub_problem` specify a sub-problem to be added; these will be run in the order they are entered.

Example:
The following is the top level command file for the test
`$Z7PATH/test/Coupled_test/INP/MechTherm.inp`

```
****calcul weak_coupling
   ***resolution
    **sequence
     *time      4.0 8.0
     *increment 10 10
     *ratio     1.e-4
   ***coupled_resolution
     **iteration 2
   ***sub_problem fem MechTherm/plastic
    **transfer integ_nodeparam
     *variable q_dot
     *file     MechTherm/heat_out
    **transfer node_kinematic
     *file     MechTherm/kine_out
   ***sub_problem fem MechTherm/thermal
    **transfer node_nodeparam
     *variable TP
     *file     MechTherm/temp_out
****return
```

This section list all three stars commands available under `****calcul`, you may find an exhaustive list of these commands in the index at the end of this handbook page 8.2.

***linear_solver

Description:
This keyword specifies the solver used to solve the linear system of equations involved in the global step of the Newton-Raphson algorithm. Both direct and iterative solvers are implemented in Z-set.

Syntax:

```
***linear_solver type
```

Where *type* is a solver type, which can take the following values.

`frontal` is a direct frontal solver based on a Cholesky factorization, so matrices have to be symmetric, definite and positive. This solver is the default solver.

`sparse_direct` is a direct solver using sparse storage, i.e. only non-zero terms of the factorized matrix are stored. So it is less memory consuming than the frontal solver. This solver is based on a Crout factorization, so matrices have to be symmetric, definite, but not necessary positive.

`sparse_dscpack` is an optimized direct solver using sparse storage. It is based on a multifrontal algorithm using a Cholesky factorization, so matrices have to be symmetric definite and positive. To be very efficient, this solver uses the BLAS optimized mathematical library working on full matrices, so it is locally more memory consuming than the `sparse_direct` solver. This solver is developed by Padma Raghavan of the Pennsylvania State University, and used by permission. Information on the DSCPACK solver is available on Web at the link : `http://www.cse.psu.edu/~raghavan/Dscpack/dscpack.ps`

`mumps` is an interface to the well known MUltifrontal Massively Parallel Solver (`http://mumps.enseeiht.fr/`). It is based on a multifrontal algorithm. The current interface only uses the multithreading parallelism of MUMPS. Like `sparse_dscpack`, this solver uses the BLAS optimized mathematical library working on full matrices. It seems that MUMPS provides the best (one-core) performance in terms of computational time and the worst in terms of memory consumption. This solver can be used for symmetric and unsymmetric systems. Depending on the system to be solved, MUMPS use a Crout or LU factorization. `mumps` can solve semi-definite systems and compute associated kernel, and accepts multiple right-and-side. So `mumps` (like `dissection`) are preferred local solvers for domain decomposition methods. More information about `mumps` can be found in section `***linear_solver mumps` page (TODO).

`dissection` is an interface to the Dissection solver developed by François-Xavier Roux (Onera, Laboratoire Jacques-Louis Lions) and Atsushi Suzuki (Laboratoire Jacques-Louis Lions). It is based on a nested dissection algorithm. The Dissection Solver use two levels of shared memory parallelism. The first level is the multithreading of the optimized BLAS. The second one is controlled via pthreads, it drives the nested dissections. This solver can be used for symmetric and unsymmetric systems. For the moment, the resolution for unsymmetric matrices is not fully optimized, you may prefer `mumps` in those cases. Thanks to this two-level parallelism, `dissection` provides

a much better scalability More information about `dissection` can be found in section `***linear_solver dissection` page 3.26. Like `mumps`, `dissection` can solve semi-definite systems and compute associated kernel, and accepts multiple right-and-side. So `dissection` (like `mumps`) are preferred local solvers for domain decomposition methods.

`sparse_iterative` includes all the available iterative solvers. These solvers are less memory consuming than direct ones, because matrices have never to be assembled, but if matrices have a bad condition number, it can be very hard to achieve convergence. Two iterative solvers can be used in Z-set, a Conjugate Gradient algorithm (cg), or a Global Minimum RESidual one (gmres). The main difference between these two solvers, is the assurance to achieve convergence if matrices are non-positive using the GMRES algorithm. This solvers type needs some complementary subkeywords described in section `***linear_solver sparse_iterative` page 3.28.

Example:
The following table gives memory needed, total CPU time and number of iterations to solve using different solvers a traction problem for a cube meshed with 13824 linear elements corresponding to 46875 dofs.

	Total CPU time	Memory needed (Mb)	Number of iterations
Frontal	48.22 mm	730 Mb	
Sparse_direct	27.82 mn	489 Mb	
Mumps	xx mn	xx Mb	
Dissection	xx mn	xx Mb	
Sparse_dscpack	3.52 mn	538 Mb	
CG + lumped	1.92 mn	146 Mb	238
CG + Cholesky	2.17 mn	160 Mb	94
GMRES + lumped	5.9 mm	161 Mb	750
GMRES + Cholesky	3.12 mm	175 Mb	169

***linear_solver dissection

Description:
This keyword specifies that the dissection solver is used to solve the linear system of equations involved in the global step of the Newton-Raphson algorithm. All parameters below are optional. Default values are already set and satisfactory in most cases.

Syntax:

```
[**scaling scaling_type ]
[**ordering ordering_type ]
[**pivoting_threshold threshold ]
[**kernel_detection_all]
[**number_iterations  int ]
[**minimal_nodes_per_leaf  int  ]
[**dim_aug_kern  dim for kernel detection type ]
[**check_solution]
[**dump_operator]
```

Where

`**scaling` specifies the scaling used to increase the accuracy of the solution. *scaling* can take these three values `none` (i.e. scaling disabled), `diagonal` (i.e. scaling using diagonal coefficients of the matrix) or `kkt` (i.e. scaling based on infinite norm of rows and columns). Default value is `diagonal`.

`**ordering` specifies the renumbering used to reduce fill in. *ordering* can take these two values `tridiag` (i.e. Cuthill-McKee algorithm), `scotch` (i.e. ordering using SCOTCH software). Default value is `scotch`.

`**pivoting_threshold` specifies the threshold used to determine null pivots. Default value is set to 10^{-2}.

`**kernel_detection_all` activate the full detection of kernel. Not needed for semi-definite matrices. Must be enabled for indefinite matrices. Disabled by default.

`**number_iterations` specifies the number of nested dissections. It is by default set to -1 (i.e. automatic choice).

`**minimal_nodes_per_leaf` specifies the minimal size of dissected parts. It is by default set to 128.

`**dim_aug_kern` specifies the size of Schur complements involved for null pivots detection. Default value is 4 (i.e 2×2 Schur complements).

`**check_solution` checks the residual. Only for debugging purpose.

`**dump_operator` dumps the matrix. Only for debugging purpose.

Example:

```
                                                    ****calcul
                                                     ***linear_solver
                                                         dissection
```

```
***linear_solver dissection
  **scaling kkt
  **ordering scotch
  **pivoting_threshold 1.0e-2
```

***linear_solver sparse_iterative

Description:
This keyword specifies the iterative solver used to solve the linear system of equations involved in the global step of the Newton-Raphson algorithm.

Syntax:

```
[**precond type ]
[**full_output]
[**solver solver]
```

Where

precond specifies the *type* of pre-conditioning used to accelerate convergence. The two available types are `lumped` (e.g. diagonal) and `cholesky`. Default value is `lumped`. Using `cholesky` preconditioner can significantly reduce the number of iterations even if it needs to evaluate and factorize an supplementary matrix, unlike `lumped` preconditioner.

full_output enables to print convergence informations. Default value is `FALSE`. If `FALSE`, nothing is said about solver iterations.

solver specifies the type of iterative solver used. *solver* can take these two values `cg` (i.e. Conjugate Gradient) or `gmres` (i.e. Global Minimum RESidual). Default value is `cg`. According to the chosen solver, the following keywords are different.

Syntax:
For the Conjugate Gradient algorithm, syntax is the following:

```
[*max_iteration max_iter ]
[*precision eps ]
[*output_every_iter nb_iter]
[*output_to_file file]
[*keep_direction dir]
[*max_standing max]
[*min_iter min_iter]
[*reprojection]
```

*max_iteration *max_iter*, where *max_iter* is the maximum number of allowed iterations when solving the problem. Default value is 1000.

*precision *eps*, where *eps* is a real value defining the relative precision required for convergence when solving the problem with the CG method. Default value is *1e-08*. For a system of equation:

$$\mathbf{Kq} = \mathbf{F}$$

this relative ratio is defined by:

$$ratio = \frac{||\mathbf{F} - \mathbf{Kq}||}{||\mathbf{F}||}$$

and convergence occurs when:

$$ratio < eps$$

*output_every_iter *iter*, where *iter* is the frequency of the iteration information output. This option is only active if the keyword ***full_output is *TRUE*. Default value is 10.

*output_to_file *file*, where *file* is a character string specifying the file where iteration information are written. This option is only active if the keyword ***full_output is *TRUE*. Default value is *iterative_solver_it*.

*keep_direction *dir*, where *dir* is the the integer value specifying the number of orthogonal descent directions retained during the CG iterations. Increasing *dir* leads to faster convergence but is more memory consuming. Default value is *max_iter*+2.

*max_standing *max*, where *max* is an integer specifying the maximum number of CG iterations allowed without any significant decrease of the convergence ratio. Default value is 50.

*min_iter *min_iter*, where *min_iter* is the minimum iterations when solving the problem. Default value is 1.

*reprojection This subcommand can significantly reduce the number of CG iterations, when used in conjunction with quasi-Newton schemes of tangent matrix update (such as eeeeee or p1p1p1, see the **algorithm command). With this option the descent directions calculated during previous load increments are reused, leading to convergence in just a few iterations when the tangent matrix and the load increment stay constant over several Newton increments.

Syntax:
For the Global Minimum RESidual algorithm, syntax is the following :

```
[*max_iteration max_iter ]
[*precision eps ]
[*output_every_iter nb_iter]
[*output_to_file file]
[*krylov_space krylov_dim ]
```

Where *max_iteration, *precision, *output_every_iter, *output_to_file are the same that for CG solver.

*krylov_space *krylov_dim*, where *krylov_dim* is an integer value specifying the dimension of the krylov_space built for each cycle of the GMRES algorithm. Increasing this value leads to faster convergence.

Example:

```
***linear_solver sparse_iterative
  **full_output
  **precond cholesky
  **solver cg
   *output_to_file iterations.hist
   *precision 1.e-12
   *output_every_iter 1
   *max_iteration 1000
```

***linear_solver rigid

Description:
This keyword specifies a "wrapper" solver which encapsulate a real solver. It behaves exactly like the underlying solver, except that the kernel of the operator is computed. If requested, a boundary condition is automatically added to fix rigid body motions.

The way rigid body motions are computed is explained in the theory manual, at the linear solver chapter.

Syntax:

`[**local_solver` *type* `]`
options for the local solver
`[**create_bc]`
`[**verbose` *solver*`]`

Where

`**verbose` asks the wrapper to print detailed informations about rigid body motions found,

`**local_solver` allow to choose the underlying linear solver. Any linear solver may be used, but note that iterative one sometimes may exhibit weird behavior (especially depending on the convergence ratio used for these solvers),

`**create_bc` specifies that a boundary condition has to be added to fix all body motions. Note also that in sequential computations the presence of body motions often means that there is an error in the input file.

***auto_remesh

Description:

The ***auto_remesh section is derived from ***initialize_with_transfer and allows automatic remeshing during a computation without stopping it. This is a backbone feature for crack propagation or computations using adaptive mesh.

Basically the auto_remesh bloc of commands is divided into:

- mesher commands controlling the remesh process.
- timing commands that trigger the automatic remeshing.
- transfer commands that specify how to transfer data between original and modified meshes.

Note:

- The use of the **Z8** output database is mandatory.
- auto_remesh transfer only variables that have been saved to the output database. To transfer all variables, one should use **save_all command in ***output command bloc
- The graphical interface scripts **Zcracks** and **Zxfem** may be used to handle crack propagation problems and automatically generate input data for this command.

Syntax:

***auto_remesh takes a number of ***initialize_with_transfer controls, and some additional specific commands.

```
***auto_remesh
[**output_after_remesh]
[**no_deform_mesh]
[**each_incr]
[**frequency]
  *    ...
[**remeshing_criterion criterion ]
 **mesher_commands
        ...
% below options are inherited from initialize_with_transfer, see 3.151%
  [**quiet]
  [**reequilibrium
    *algo  algorithm
    *ratio  convergence
    *iter  max_iterations  ]
  [**skip_nodal_transfer]
  [**skip_integ_transfer]
  [**nodal_var_transfer
    [*mapping mapping_method ] ]
  [**integ_var_transfer
    [*integ_transfer transfer_method ] ]
```

**output_after_remesh save the results of the transfer to the database. With this option, Z8 database will contain two copies of the same results: the original data on the original mesh, and the transferred data on the new mesh obtained as a result of the specified mesher commands.

**no_deform_mesh use initial configuration of the old mesh to locate the nodes/IP of current mesh (if not specified, the default behavior is to use deformed mesh)

**each_incr force remeshing on each increment of current computation.

**frequency used to trigger the remeshing process, for more sub options see 3.172

**remeshing_criterion a more general remeshing trigger, The possible criterion are summarized

CODE	DESCRIPTION
at_time	
quadratic_curving_criterion	
element_shape_quality	
for_crack_propagation	
error_based	

**mesher_commands mesh manipulators commands see 2.4 for usable commands

Example:

```
***auto_remesh
%% AUTO_REMESH options
  **output_after_remesh
  **no_deform_mesh
  **frequency
   *at_time 0.02
%% INITIALIZE_WITH_TRANSFER options
  **integ_var_transfer default
   *locator bb_tree
   *integ_transfer nearest_gp_corrected
%% TRANSFORMERS
  **yams_ghs3d
   *no_deform_mesh
   *min_size  0.2
   *absolu
  **nset enc
   *use_bset enc
```

```
****calcul
 ***auto_remesh
```

This example can be found in Crack_test/INP/cube_2cracks.inp

```
***global_parameter
  Solver.OutputFormat  Z8
  Zmaster.OutputFormat Z8

***auto_remesh
 **frequency
  *d_cycle 1
  *cycle_period 2.0
  *at 2.0
 **no_deform_mesh
 **skip_integ_transfer
 **skip_nodal_transfer
 **save TO_REMESH.geo
 **drive_crack
  *mesher cube_2cracks.inp
  *geo_in TO_REMESH.geo
  *geo_out REMESHED.geo
  *advance cube_2cracks.adv
  *separate
 **open REMESHED.geo

***output
 **save_parameter
 **save_all
```

***auto_adaptation

Description:

On going work ...

This command drives an automatic mesh adaptation procedure during a computation without stopping it. The `auto_adaptation` syntax is mainly divided into:

- mesher commands controlling the remesh process.
- timing commands that trigger the automatic remeshing.
- transfer commands that specify how to transfer data between the original and the modified mesh.

Syntax:

The command has the following syntax:

```
***auto_adaptation
**remeshing_trigger   criterion
**mesher_commands
[**frequency]
[**do_transfer]
[**use_deformed_mesh]
[**do_it_again_after_remesh ]
[**output_after_remesh ]
```

`**remeshing_trigger` activates remeshing when a quantity of interest is higher than a given threshold. For now, only one criterion is available :

```
**remeshing_trigger   global_error
**threshold   double-value
**estimator_name   error-estimator-name
```

`**`*mesher_commands* can define initial mesh modifications, remeshing options and modifications of the adapted mesh . See 2.4 for available commands.

`**frequency` used to trigger the remeshing process, for more sub options see 3.172

`**do_tranfer` activates the transfer of all nodal and integration points fields computed on the initial mesh to the adapted one.

`**use_deform_mesh` deformes the mesh before transfer. By default, the initial configuration is used.

`**output_after_remesh` when used, the result of the transfer.

`**do_it_again_after_remesh` when used, the current increment is recomputed on the new mesh.

Example:

```
***auto_adaptation
  **do_it_again_after_remesh
  **output_after_remesh
  **frequency *increment 1
  **remeshing_trigger global_error
   *threshold .1
   *estimator_name relative_energy_error_hole

 **mmg3d
  *min_size  .01
  *max_size  4.
  *verbose   10
  *hgrad 2.
  *hausd 1.
  *preserve_elsets_start_with eset
  *preserve_bsets right front left
  *corners_start_with corn
  *metric uniform_from_field
   field    relative_energy_error_hole
   accuracy 0.1
  *output_mesher_files
```

***disc_error_estimation

Description:

This command drives the computation of the finite element estimated solution error due to mesh discretization. All available methods are based on the errors indicators of type Zienkiewicz and Zhu (ZZ).

Syntax:
The command has the following syntax:

```
***disc_error_estimator
**estimator   estimator-type
**name        error-estimation-given-name
[**elset       elset-name   ]
[**field       field-name   ]
[**quantity    field-type   ]
[**norm ]
[**relative_error  type   ]
[**metric   metric-type    ]
```

`**name` is the name given to the global error estimation and also to the computed field of local contributions to the global estimation error.

`**elset` denotes the element set on which the discretization error will be computed. By default, it will be computed on all elements.

`**field` denotes the name of the gradient field used to estimate the error. By default the stress field is used to compute a ZZ type error estimator.

`**quantity` defines the type of the gradient field used to estimate the discretization error. It can be a scalar field (SCALAR) or a tensorial field (TENSOR2). By default it takes the value TENSOR2.

`**norm` takes one of the values `L2` or `H1` . It defines the norm used to estimate the error : L_2 or energy norm. By default, the energy norm is used.

`**relative_error` when activated, the error estimation is given as a percentage relative to either the maximum of the error contributions or the energy of the structure. The type can either be chose to be `maximum` or `energy`.

`**metric` described in 2.10, it allows the computation of a mesh size map.

`**estimator zz2` defines an error indicator based on the Zienkiewicz and Zhu estimator of second type for linear problems. It's syntax is the following one :

```
**estimator   zz2
[**quadratic]
```

The quadratic option should be activated when used with quadratic finite elements.

`**estimator zz2_heterogeneous` defines an error indicator based on the Zienkiewicz and Zhu estimator of second type for linear problems, but adapted to heterogeneous problems. It's syntax is the following one:

**estimator *zz2_heterogeneous*
[**penalization *double-value*]

The indicator is implemented through a penalization technique, and may require an adjustement. When the penalization coefficient equals zero, the estimator is equivalent to the previous zz2 estimator applied on each heterogeneous element set.

**estimator zz2_incr defines an error indicator proposed by Boroomand and Zienkiewicz for non linear problems.

**estimator mean defines an error indicator based on the Zienkiewicz and Zhu estimator of first type for linear problems. It's syntax is the following one:

**estimator *mean*
[**quadratic]

The quadratic option should be activated when used with quadratic finite elements.

Example:

```
***disc_error_estimator
 **relative_error energy
 **field sig
 **name relative_energy_error_hole
 **elset ALL_ELEMENT
 **quantity TENSOR2
 **estimator zz2

***disc_error_estimator
 **field sig
 **name error_hole
 **elset ALL_ELEMENT
 **quantity TENSOR2
 **estimator zz2
```

***bc

Description:

This procedure and its options define the boundary conditions of a problem. The definition of boundary conditions groups both conditions acting on the degrees of liberty and those acting on the associated forces.

Syntax:

```
***bc [from t1 to t2]
 **bc-type
     bc-specific options
 **another-bc-type
     ...
```

There may be any number of sub-options defining the different conditions to impose, and also any number of `***bc` instances.

The different types of boundary conditions are selected by using different values for *bc-type*. These different BC commands are the subject of the following pages. The pages of the sub commands are sorted according to their applicability to specific problem types. The following tables are included as a quick-directory to the BCs.

If the optional arguments `from t1 to t2` are specified, all the boundary conditions declared in the current `***bc` block will be active only between times `t1` and `t2`. You may declare several `***bc` to partition the activity intervals of your boundary conditions.

General purpose BCs:
Some BCs are general-purpose, being specified in terms of any degree of freedom in the problem, and their attached location (nodal or elemental).

CODE	DESCRIPTION
impose_nodal_dof	used to directly impose any DOF value which is located at a node - p.3.49
impose_nodal_dof_rate	used to impose any DOF value in a rate form - p.3.51
impose_nodal_reaction	sets a node's associated force - p.3.51
impose_nodal_reaction_rate	rate of the nodes reaction - p.3.55
impose_element_dof	Fix DOFs which exist at element Gauss points - p.3.45
impose_element_dof_reaction	Set DOF reactions which exist at element Gauss points - p.3.46
impose_elset_dof	for DOFs defined over an element set - p.3.47
impose_elset_dof_reaction	reactions associated to DOFs defined over an element set - p.3.48
impose_nodal_dof_density	Density of a nodal dual force over element faces or edges - p.3.56
impose_nodal_energy	Fix DOF until a specified energy is reached
impose_nodal_dof_and_release	
release_nodal_dof	Time based release of fixed DOFs (can be used for crack growth) - p.3.57
submodel	used to directly impose any DOF value at sub-model boundary nodes from a master computation - p.3.58

Mechanical BCs:
There are many mechanical specific boundary conditions in Zébulon. A summary follows.

CODE	DESCRIPTION
deformation	used to impose displacements with an intermediate (strain) tensor - p.3.65
deformation_cosserat	imposes a "Cosserat" type strain
strain_gradient	displacements as $u = Er + 1/2D\ r \times r$ - p.3.83
crack_release	propagate crack (releasing nodes) according to a material variable criterion - p.3.64
radial	submit all or part of a structure to a radial expansion - p.3.76
radius	enforces "rolling" part of a structure on a radius (for example holding the radius of a joint) - p.3.77
pressure	impose a surface pressure on a liset or a faset - p.3.75
shear	impose a shear pressure on a liset (only valid in 2D) - p.3.80
impedance	impose an impedance boundary condition (only valid in dynamics) - p.3.72
hydro, hydro_finite_strain	impose a surface pressure on the deformed geometry; this simulates a fluid pressure - p.3.70 and p.3.71
curvature	Curvature on a boundary $u_i = \epsilon_{ijk} K_{jl} X_l X_k$
centrifugal	centrifugal loading for all the elements of a structure due to a rotational frequency - p.3.63
gravity	applies a uniform acceleration force to the entire structure - p.3.69
rotation	rotate nodes about a given axis - p.3.78
free_rotation	Rotation about an axis with one direction free - p.3.67.
linear_rotation	Rotation with small angle ($\theta \approx \sin\theta$) - p.3.73
linear_free_rotation	Free rotation with linear approx - p.3.68.
K_field	Impose the linear elastic crack tip solution to a node set - p.3.60
static_torsor	Impose a static torsor (resultant and momentum) on a nset. It is associated to a rigid body motion of the nset - p.3.82.

```
****calcul
 ***bc
```

Thermal BCs:
Thermal boundary conditions are provided for a variety of heat flux transfer options. Surface to surface radiation transfer is not yet possible.

CODE	DESCRIPTION
surface_heat_flux	applies a constant heat flux on a liset or a faset - p.3.84
convection_heat_flux	convective flux on a liset or a faset - p.3.85
interface_heat	imposes an inter-facial thermal resistance between two lisets or fasets - p.3.86
volumetric_heat	applies a volumetric heat flux on all the elements of a structure - p.3.87
volumetric_heat_from_parameter	applies a volumetric heat flux on all the elements of a structure. The values come from a parameter field - p.3.88
volumetric_heat_in_file	Import an internal heat generation from a mechanical problem (coupled) - p.3.89
radiation	heat flux by radiation on a liset or faset - p.3.90

Some of these keywords replace deprecated keywords as follows :

flucons	by	surface_heat_flux
fluconv	by	convection_heat_flux
fluconv_interface	by	interface_heat
fluvol	by	volumetric_heat
fluvol_in_file	by	volumetric_heat_in_file

Mass Diffusion BCs:

CODE	DESCRIPTION
surface_flux	Impose surface flux of concentration

```
****calcul
 ***bc
```

Scaling of boundary values:
The magnitude of boundary conditions are normally specified in two parts. The first is a "base value" – a real number, function or file data (see below) – which acts as a multiplicative scale factor. In the event that this value is zero, the BC will be a fixed condition, not requiring a loading table. The second part is a table name which refers to a corresponding defined table (using the `***table` command). The table is used to describe the magnitudes in time.

Table specifications may usually take more than one table name. This is very useful in the case of cyclic loadings, or other complex lengthy waveforms. The tables will be sequentially searched for valid times. That is if the first table is defined from 0. to 250., and a second from 200. to 1000., times until 250 will be calculated with the first table, while the remaining time is taken from the second table. If the time exceeds 1000 there will be an error.

Base value:
As mentioned earlier, the "base value" acts as a multiplicative scale factor for the following table. The standard base value is simply a real number. It may also be a more elaborated object, such as a function or file, which allows space-dependent boundary conditions.

The following example presents all four types of base values currently available:

```
***bc
 **impose_nodal_dof
   left   U1 0.                              % zero (the table can be omitted)
   left   U1 10. tab                         % constant scalar value
   right  U1 file       right.dat  table1    % binary file
   bottom U2 ascii_file bottom.dat table2    % ascii file
 **pressure
   top function sin(x); time                 % function
```

Note that some multi-point-constraints (see 3.126) also use this concept of base value.

Duration of application:
Normal use will involve defining the value of a boundary condition throughout the time scale of the problem. It is very important to include the value of the BC at zero time.

Boundary conditions can also be applied over a limited time during the problem, with the condition either "expiring" or coming into action after a specified period of time. This is very useful for dynamic problems where an initial movement is given and then released, or if a condition is applied to a certain point, and then continued with a different type condition. An example of the latter is to apply force control for a first sequence, followed by a rate of displacement. Note after the load is applied the absolute value of displacement is unknown (for non-linear problems) so a displacement rate must be imposed (e.g. with `impose_nodal_dof_rate`).

These cases are handled by having the BC's table defined over a limited duration of the problem time scale, and including the keyword `exp` to indicate that the expiration or activation of the BC was intentional.

```
***resolution
  **sequence
    *dtime        1.0 0.95
***bc
  **impose_nodal_dof
```

```
                                                    ****calcul
                                                     ***bc

      wall       U2   0.0
      wall       U1   0.0
      load  exp U2  -1.0 tab
***table
  **name tab
    *time    0.0   1.
    *value   0.0   1.
```

Association to geometry:
Most BCs may also be localized to certain portions of the geometry, through the use of node, element, line and face sets. Specification of these entities are discussed in the meshing chapters *file .inp: 2D meshing* and *file .inp: 3D meshing*. The following general statements can be made:

- Boundary conditions to be applied in order to directly impose the value of nodal degrees of freedom are applied to *nsets*.
- Boundary conditions which are distributed over a surface, or have a mean value for a surface (e.g. pressure) will be applied to a *liset* in 2D or a *faset* in 3D.
- Conditions distributed through the body will be applied to an *elset*.

Line or face set surface conditions are generally affected by the set's normals orientation. See for instance discussion of the pressure sign (page 3.75).

When it is desired to impose a condition to a single node or to a single element, the node or element number may always be substituted for an nset or elset name.

There are also pre-existing nsets and elsets for every node or element in the mesh. These sets are named respectively `ALL_NODE` and `ALL_ELEMENT`.

**impose_element_dof

Description:

This command is used to impose degrees of freedom which exist at element integration points. These DOFs are commonly part of mixed formulations, such as pressure-displacement, and plane-stress elements.

Syntax:

```
**impose_element_dof
  elset  dof  value  table
```

Example:
Making a 2.5 D case of planar displacements and an imposed ϵ_{33} strain. formulation.

```
***mesh plane_stress
***bc
  **impose_nodal_dof
     bottom U2 0.
     top    U2 1. time
     left   U1 0.
  **impose_element_dof 1 EZ 1.e-3 time
```

Element DOFs are also imposed for the RVE elements, with a full strain tensor being the element DOFs.

```
***bc
 **impose_element_dof
  ALL_ELEMENT E12 1. time
```

**impose_element_dof_reaction

Description:
This command imposes the conjugate reaction to an element degree of freedom.

Syntax:
The syntax takes an element set name, the name of the degree of freedom for which reaction will be fixed, a scale value, and a table name (required for non-zero *value*).

```
**impose_element_dof_reaction
  elset  dof  value  table
```

Example:

The following example apples mixed mode loading to an RVE element with the `impose_element_dof_reaction` conditions directly imposing the stress. This demonstrates the use of an element number for the element set. Note that `E22` is given to fix σ_{22}. This is because the reaction is specified by the degree of freedom name, which is the strain in RVE elements.

```
***bc
 **impose_element_dof
   1 E33 0.0
 **impose_element_dof_reaction
   1 E22 25.  time
   1 E11 400. time
```

**impose_elset_dof

Description:
The boundary condition allows one to impose the value of a degree of freedom over an element set. Principally this condition applies to the elements 2_5D (see ***mesh). The DOFs imposed by this condition must be located at element integration points (Gauss points), and not nodal unknowns.

Syntax:

```
**impose_elset_dof
  name_elset  dof_name  value  name_table
```

name_elset character name of the element set. This must be the name of a valid elset defined in the geometry file. The DOF will be imposed at every Gauss point in this element set.

dof_name The character name of the degree of freedom to be imposed. For the 2_5D elements the choices are t1 t2 t3 w1 w2 w3.

value base value which scales table values (real).

table character name or list of names for the tables to be used (see ***table).

Example:

```
**impose_elset_dof
   str t1 .01 tab1
   str t2 0.
   str t3 0.
   str w1 0.
   str w2 0.
   str w3 0.
```

**impose_elset_dof_reaction

Description:
This command allows application of a force on the group of nodes within an element set. Note that application of a force is *not* the same as application of a pressure.

Syntax:

```
**impose_elset_dof_reaction
  name_elset  dir  value  name_table
```

name_elset character name of the element set within which the reaction will be applied.

dir real value for the direction of application for the force.

value base value (real) for the condition. This value is a multiplier of the current table value thereby establishing the magnitude of the condition.

table character name for a valid loading table input with the command ***table**.

Example:

```
**impose_elset_dof_reaction
   ALL_ELEMENT  w1 1. tabmx
   ALL_ELEMENT  w2 1. tabmy
```

**impose_nodal_dof

Description:
This boundary condition imposes degrees of freedom located at nodes to defined values in time. The condition is general and therefore applies to *all* types of DOF for all types of problem.

Syntax:
The syntax required to impose nodal DOFs is:

```
**impose_nodal_dof
    nset_name  dof_name  value  table_name
```

nset_name This is the name of a valid node set (nset) which gives all the nodes where the DOF is to be imposed.

dof_name The character name of the DOF to be imposed. This must be one of the defined DOF types given by the problem, and as indexed in the appendix. DOF names are also listed in the *problem*`.ut` reference file.

value This real value acts as a multiplier on the current applied load table value. If the value is zero, no table will be required.

table_name A character name for the loading table which describes the DOF value in time. A corresponding table must therefore be given elsewhere (see the procedure `***table`).

Example:
Supposing that a mesh has been created with a node set composed of a single node, node 1. This example displaces that node a fixed amount in the directions u_1 and u_2. The u_1 magnitude will be 1.2 times the table value specified by `table3` while u_2 will be only 0.2 times that value. Using the same table assures synchronization between the loadings.

```
***bc
 **impose_nodal_dof node1 U1 1.2 table3
 **impose_nodal_dof node1 U2 0.2 table3
```

Note that a structure will normally have more displacement conditions than the fixing of a single node. Other conditions such as pressure and force are also likely.

The same command may be used to specify the temperature in a thermal problem.

```
**impose_nodal_dof node1 TP 200.0 tab1
```

Note:
Some variants exist, which allow the *value* to vary in space:

**impose_nodal_dof

nset_name *dof_name* `function` *function(x,y,z);* *table_name*

nset_name *dof_name* `file` *binary_file* *table_name*

nset_name *dof_name* `ascii_file` *ascii_file* *table_name*

The `function` variant allows the DOF value to depend on nodal coordinates. The function may also depend on time; this is however discouraged, because in that case incremental values are not properly computed. The time dependency should rather be specified through the table.

Both `file` variants read the value of each DOF from a file; thus the file should contain as many entries as nodes in the nset, values being ordered as in the nset. Note that the binary form expects entries as "floats" (not "doubles").

**impose_nodal_dof_rate

Description:
This boundary condition imposes degrees of freedom located at nodes on a rate basis. The loading table will be scaled by the base value at any time to give the loading rate.

Syntax:
The syntax required to impose nodal DOF rates is:

```
**impose_nodal_dof_rate
   nset_name  dof_name  value  table_name
```

nset_name This is the name of a valid node set (nset) which gives all the nodes where the DOF is to be imposed.

dof_name The character name of the DOF to be imposed. This must be one of the defined DOF types given by the problem, and as indexed in the appendix. DOF names are also listed in the *problem*`.ut` reference file.

value This real value acts as a multiplier on the current applied load table value. If the value is zero, no table will be required.

table_name A character name for the loading table which describes the DOF value in time. A corresponding table must therefore be given elsewhere (see the procedure `***table`).

Example:
This example is for an axisymmetric specimen with applied hydrostatic pressure of 52 MPa, followed by strain rate controlled compression cycling along one axis, while the hydrostatic pressure is maintained. Note the use of zero in `*dtime` to apply step loading of the strain rate. Expiring (`exp`) is also applied to the DOF rate condition because the DOF is not active at all times.

```
***bc
  **pressure
    press     -1. ptab
    top   exp -1. ptab2
  **impose_nodal_dof
    y=0       U2  0.0
    x=0       U1  0.0
  **impose_nodal_dof_rate
    y=1  exp U2  1.0 seg
***table
  **name ptab
    *time  0.0  100. 5000.
    *value 0.0  52.0 52.0
  **name ptab2
    *time  0.0  100.
    *value 0.0  52.0
  **name seg
```

```
                                                    ****calcul
                                                     ***bc
                                                      **impose_nodal_dof_rate

*dtime  100.0
         0.0  3.1579e+01
         0.0  7.8947e+01
         0.0  5.5263e+01
*value   0.0
        -3.8e-05 -3.8e-05
         3.8e-05  3.8e-05
        -3.8e-05 -3.8e-05
```

**impose_nodal_dof_and_release

Description:
This boundary condition imposes degrees of freedom located at nodes to defined values in time. The DOFs are imposed until a specified time; they are then progressively relaxed to a null flux BC.

Syntax:
The syntax of this boundary condition is:

```
**impose_nodal_dof_and_release
  nset_name  dof_name  t_ini  t_release  value  table_name
```

nset_name is the name of a valid node set (nset) where the DOF is imposed.

t_ini is the time until which the DOF is imposed. At *t_ini*, the BC automatically switches to impose the equivalent force (nodal reactions) on the given nodes.

t_release is the time at which the force imposed on constrained nodes is set back to 0. The time interval *t_ini t_release* is used to progressively shed this force from its current value back to 0.

dof_name is the name of the DOF to be imposed. This must be one of the defined DOF types given by the problem. DOF names are also listed in the *problem*`.ut` reference file.

value this real value acts as a multiplier on the current applied load table value. If the value is zero, no table will be required. It may also be any valid *basic value* (see p.3.43 for more details).

table_name a character name for the loading table which describes the DOF evolution in time. A corresponding table must therefore be given elsewhere (see ***table, p.3.227).

Example:
Supposing that a mesh has been created with two node sets composed of a single node, node 1 and node 2. This example sets the displacement of specified nodes to 0 in the direction u_1 until the current time is 100s. Then, this constraint is released in 0.5 s. Such an example can be used to simulate crack propagation using a node releasing approach.

```
***bc
 **impose_nodal_dof_and_release node1 U1 100.0 100.5 0.
 **impose_nodal_dof_and_release node2 U1 100.0 100.5 0.
```

**impose_nodal_reaction

Description:
This option sets the "force" reactions which act on a node set. Note that applying a force to a node set is very different than applying a pressure (using the BC `**pressure`). The condition is general so it may be used for all types of problem. For the simple mechanical case, the nodal reactions are in units of force.

Syntax:

`**impose_nodal_reaction` *nset dir val table*

---

nset Character name of the node set where the condition is applied.

dir Direction in which the reaction is applied. This will be the DOF name whose associated reaction is affected. See the appendix for DOF names.

val Real number for the multiplicative base value of the condition.

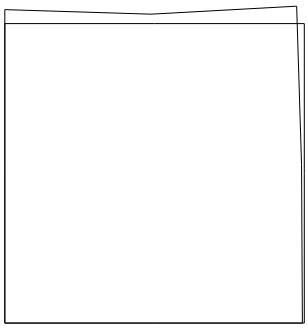
table Character table name of a valid table, or a list of character table names.

Example:
The current example demonstrates the incorrect use of this option to impose a surface pressure. The difference is a common misconception of boundary conditions in the FEM. The problem uses a plane-stress eight node square element with fixed `U1` on the left border and fixed `U2` on the bottom. Material behavior is linear elastic.

The top node set is defined using the right top, middle top, and left top nodes in sequence. As the square has unit sides, one could suppose that the three forces applied should be 1/3 the stress. The BC is therefore applied as:

```
***bc
  **impose_nodal_reaction
    top U2  0.3333 tab
# **pressure
#   press  1. tab
```

which results in the following distorted deformation:



Proper application of a pressure requires knowledge of the surface integration along the boundary. The options `**pressure` and `**shear` must therefore be used.

**impose_nodal_reaction_rate

Description:
This is an associated reaction version of `**impose_nodal_dof_rate`. The condition is useful when the initial reaction state is unknown, which can be generated with a mix of displacement and reaction BCs using the `exp` keyword.

Syntax:
The syntax is analogous to `**impose_nodal_dof_rate`.

```
**impose_nodal_reaction_rate
    nset_name  dof_name  value  table_name
```

**impose_nodal_dof_density

Description:
The command imposes the flux (associated reaction) density across a surface (nset or faset). It is a general boundary condition, which can mean imposed pressure in the X-Y-Z directions for mechanical problems. The condition `**fluconv` is a special case of this condition as well.

Note that the `**pressure` command adjusts itself for `updated` elements according to the surface normal, and for 3D meshes there is no `**shear` option because the face tangent is not fixed. This option differs because it is related to the density of the DOF reactions and therefore rests in their coordinate space.

Syntax:

```
**impose_nodal_dof_density
  bset  dof  value  table
```

where *bset* is a boundary set name (liset or faset), *dof* is a degree of freedom name (e.g. `U1`), *value* is a scaling value (decimal point number), and *table* is the name of a loading table input elsewhere.

Example:
Pressure defined in the U1-U2 directions.

```
***bc
 **impose_nodal_dof
   left   U1 0.0
   bottom U2 0.0
   fix    U3 0.0
 **impose_nodal_dof_density
   face.2 U1 0.25  tab1
   face.2 U2 0.25  tab2
```

**release_nodal_dof

Description:
This command allows a time-based progressive release of a fixed condition. The crack length is given by the loading table of the condition. Using a function or table of values one can specify complex crack growth rates. For example crack growth measurements can be input as a table, and the crack propagated as actually seen. Then loading parameters such as ΔK or maximum crack tip opening can be correlated to the growth.

Syntax:

```
**release_nodal_dof
 nset  dof  base-val  table
```

Note:
The crack length is measured as the distance sum of distances between nodes with the initial tip being at the first node of the given node set (i.e. its the arc length s of the line drawn by the node set). Curved crack paths are therefore possible. The node set should be ordered beforehand.

**submodel

Description:
Assume that the current computation is running on a part of a bigger problem already solved. we call *submodel* the current part and *Master* the bigger problem. This boundary condition imposes degrees of freedom located at boundary nodes in a submodel to values computed from the Master. The submodel boundary nodes have not to match with Master's one, this BC is able to locate them in the Master mesh and interpolate their values from Master's dofs. The condition is general and therefore applies to *all* types of DOF for all types of problem.

Syntax:
The syntax of submodel is:

```
**submodel
 [*format format ]
 *global_problem  Master_name
 *dofs  dof_1_name  dof_2_name  ...  dof_n_name
 *driven_nsets  nset_name
```

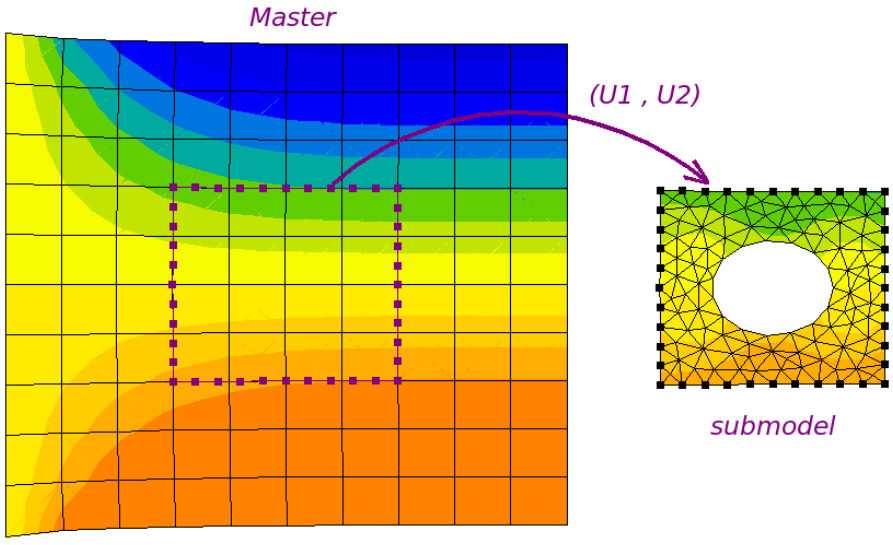
format This is the results format of the *Master* computation. If `*format` is omitted, the default Z7 format is assumed.

Master_name The name of the *Master* computation.

dof_i_name The name of the DOF to be imposed. This must be one of the defined DOF types given by the *Master* and *submodel* problems. DOF names are also listed in the *problem*`.ut` reference file.

nset_name This is the name of a valid node set (nset) which gives all the *submodel* nodes where the DOFs are to be imposed.

Example:
This example can be seen in test database, we use the displacement *U1, U2* on a simple square to drive a submodel with a hole.



```
                                                    ****calcul
                                                     ***bc
                                                      **submodel

***bc
  **submodel
    *format Z7
    *global_problem master
    *dofs U1 U2
    *driven_nsets external
```

**K_field

Description:
This boundary condition is used to impose on a `NSET` the displacement field given by the linear solution at a crack tip. The crack is loaded under mode I or II. This boundary condition is limited to 2D plane problems. It will work on axisymmetric problems , however its physical meaning will be uncertain.

The displacement field is given by:

Mode I:

$$u_1 = \frac{K_I}{2\mu}\sqrt{\frac{r}{2\pi}}\cos\frac{\theta}{2}\left(\kappa - 1 + 2\sin^2\frac{\theta}{2}\right)$$
$$u_2 = \frac{K_I}{2\mu}\sqrt{\frac{r}{2\pi}}\sin\frac{\theta}{2}\left(\kappa + 1 - 2\cos^2\frac{\theta}{2}\right)$$

Mode II:

$$u_1 = \frac{K_{II}}{2\mu}\sqrt{\frac{r}{2\pi}}\sin\frac{\theta}{2}\left(\kappa + 1 + 2\cos^2\frac{\theta}{2}\right)$$
$$u_2 = -\frac{K_{II}}{2\mu}\sqrt{\frac{r}{2\pi}}\cos\frac{\theta}{2}\left(\kappa - 1 - 2\sin^2\frac{\theta}{2}\right)$$

with

$$\kappa = \begin{array}{cc} 3-4\nu & \text{plane strain} \\ (3-\nu)/(1-\nu) & \text{plane stress} \end{array}$$

where ν is the Poisson's ratio. Let M be the point where the displacement field is computed and T the crack tip. $r = ||\vec{MT}||$, $\theta = (\vec{Ox}_1, \vec{MT})$.

Syntax:
The syntax is as follows:

`**K_field` *nset center young poisson* `plane_state mode` *value* [*table*]
nset name of the `NSET`
center crack tip position (2D vector)
young Young modulus (double)
poisson Poisson coefficient (double)
`plane_state plane_stress` or `plane_strain`
`mode` specifies loading mode: `I` or `II`
value basic boundary condition value (double). This represents the basic value of $K_{I,II}/2\mu$.
table table name

Example:

```
                                                    ****calcul
                                                     ***bc
                                                      **K_field

**K_field
   border  % nset name
   (0.,0.) % crack tip position
   0.3     % Poisson's ratio
   plane_stress
   II      % mode II
   10.     % basic value
   Ktab    % table name
```

**T_field

Description:
This boundary condition is used to impose on a `NSET` the displacement field given by the linear solution at a crack tip of and applied T–field.

Under plane strain conditions, the displacement field is given by:

$$u_1 = T\frac{1-\nu^2}{E}R\cos\theta$$
$$u_2 = -T\frac{\nu(1-\nu)}{E}R\sin\theta$$

where ν is the Poisson's ratio, E the Young modulus.

Syntax:
The syntax is as follows:

`**T_field` *nset center young poisson value* [*table*]
nset name of the `NSET`
center crack tip position (2D vector)
poisson Poisson coefficient (double)
young Young module (double)
value basic boundary condition value (double). This represents the basic value of T.
table table name

Example:

**centrifugal

Description:

The centrifugal boundary condition is used to apply body forces $\rho\omega^2 r$ to a defined set of elements (`elset`) due to rotational acceleration, with ρ the volumetric mass, ω the rotational rate and r the distance to the rotational axis.

Syntax:

```
**centrifugal [updated] [square]
  elset_name (v) dir value  table  ...  [tableN]
```

elset_name Character name for the set of elements which are subjected to the centrifugal loading.

updated Uses the current configuration rather than the initial configuration for determining the distance r to the rotational axis as well as the direction of the centrifugal force.

square Applies a force proportional to the rotational rate ω instead of ω^2.

v Vector giving a point position through which the rotational axis passes. Axisymmetric problems require that the origin is input as `(0.0 0.0)`.

dir Direction of the rotational axis (`d1`, `d2`, or `d3`). Axisymmetric 2D problems are required to use the `d2` axis direction. Planar 2D geometries require that the axis is `d3`.

value Base value (real) which scales a table value to determine the applied rotational rate ω.

table Character name for a valid loading table or tables which, after multiplication with *value*, will describe the time evolution of the rotational rate.

To apply a centrifugal force, it is required to give the volumetric mass ρ for the material in the material file. This is specified with the `***coefficient` command.

Units required depend of course on the consistency of units within the problem. For example, if the dimensional unit is millimeter, the forces are given in Newton, and the rotational rate ω is in radians per second, the volumetric mass must be in tons (metric)/mm^3.

Example:

```
**centrifugal
   str (0.0 0.0) d2 1.e7 tab1
```

**crack_release

Description:
This BC is used to propagate a crack (release DOFs) according to node-extrapolated material values (note there can be some significant error due to the extrapolation error, especially with the high variable gradients found around crack tips).

Syntax:

```
**crack_release
  nset  dof  var-value-list
```

where *nset* is the node set defining the ligament through which the crack propagates, and *dof* is the fixed DOF name (e.g. `U2`). Following these entries, a list of variable-value combinations are to be entered to specify the crack growth criterion.

Example:
For a plasticity problem a maximum allowable plastic strain equivalent of 20% could be used (remember the name of p - here `epcum` depends on the material law used).

```
**crack_release
  middle U2 epcum 0.20
```

Or one could use a secondary material variable such as the von Mises stress as a ultimate tensile strength.

```
**crack_release
  ligament U2 sig::mises 520.0
  ligament U1 sig::mises 520.0
```

**deformation

Description:
This boundary condition imposes the displacement vector $\vec{v}$ of a node by the following formula:

$$\vec{u} = \boldsymbol{\epsilon}\vec{r}$$

where $\boldsymbol{\epsilon}$ is a symmetric tensor, and $\vec{r}$ the position vector of the node considered. The origin of $\vec{r}$ may be a fixed point or a node of the mesh.

For the case where the origin is a fixed point the node set considered must be given, as well as the position of the origin point in the form of a vector, and the components of the tensor $\boldsymbol{\epsilon}$. In 1D ϵ_{11} and ϵ_{22} may be imposed; in 2D ϵ_{12} may be added; in 3D the remaining ϵ_{33}, ϵ_{13}, and ϵ_{23} terms may be added. The tensor components are respectively named `E11`, `E22`, `E12`, `E33`, `E13`, and `E23`. If a component is not defined its value is set to zero. The values may be set by using a table for all of the components, or a table for each individual component (only for non-zero components).

Syntax:

**deformation [*node*]
nset_name (vec | *num_node)*
compos_name value [*table*]... *compos_name value*
[*table*]

`node` is an optional keyword indicating that the origin of $\vec{r}$ is a node of the mesh.

nset_name Character name of the nset group upon which the displacement will be imposed.

vec Position of the origin point in the form of a vector. This is in the case where the center is not a node of the mesh. Recall the vector form is 2 or three real values enclosed in parenthesis such as `( 1. 0. 0. )`.

num_node Integer number of a node in the case where the `node` keyword was given.

compos_name value Character string giving the name of a component of the $\boldsymbol{\epsilon}$ tensor as defined above, followed by a real giving the components value. Any number of these components may be given in any order thereby specifying the deformation tensor.

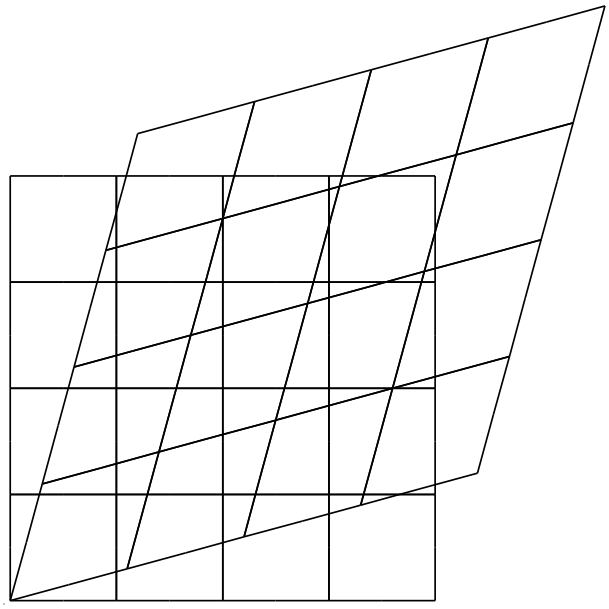
table character name of the table(s) used to describe the field amplitude. See the command `***table`.

Example:
This example shows a simple uniform deformation field to an initially square multi-element mesh. The required BC input data is the following (see test `deform` in `Z7`test/Static_test/):

```
**deformation
  boundary (0. 0.) E22 0.1  E11 0.1 E12 .3 tab
```

```
****calcul
 ***bc
  **deformation
```

where `tab` is a unit table definition given after. The resulting deformed structure is displayed below. This type of condition is useful for testing the mechanical behavior of materials under complex strain loadings.



Another imposed strain field is given below:

```
**deformation node
   nsetdef 2 E11 2. E33 4. tab
```

**free_rotation

Description:

This rotation condition allows a boundary to be rotated about one of the coordinate axes (2 displacements are imposed), while the DOFs along the third axis remains free. The usefulness of this condition is obviously only for 3D problems.

Syntax:
The syntax is the following:

```
**free_rotation
  nset  dir  [origin]  value  table
```

nset Name of the node set upon which the condition is active.

dir an axis about which the node set will rotate (`d1` `d2` or `d3`). The DOF acting along the given direction is free in the absence of other conditions acting on it.

origin specifies an arbitrary point on the rotation axis (use vector notation, see 1.3); it defaults to (0. 0. 0.).

value Base value (scaling factor, decimal number).

table Name of a valid loading table.

**linear_free_rotation

Description:
A linearized version of the `**free_rotation` condition for use in linear small deformation analysis. See page 3.73 for more discussion on this.

**gravity

Description:
The gravity condition applies a body force due to uniform acceleration such as gravity.

To apply a gravitational force, it is required to give the volumetric mass for the material in the material file. This is specified with the `***coefficient` command.

Syntax:
The gravitational BC is specified with the following syntax:

`**gravity` *elset_name* `dir` *value* [*table*]

---

elset_name Character name for the pre-defined element set where the gravitational acceleration is applied.

dir Direction of the gravitational force (`d1`, `d2` or `d3`). Axisymmetric 2D problems require that the direction be `d2`. Plane 2D geometries require that the direction is `d1` or `d2`.

value Real base value scaling the current table magnitude to determine the acceleration magnitude.

table Character name for a valid loading table or tables which will describe the acceleration in time.

Units required depend of course on the consistency of units within the problem. For example, if the dimensional unit is millimeter, the forces are given in Newtons, and the gravitational acceleration is in mm/sec/sec, the volumetric mass must be in tons (metric)/mm/mm/mm.

Example:

```
**gravity
    str d2 -20.0 tab1
```

**hydro

Description:

This boundary condition is used to impose a surface pressure calculated based on the updated (deformed) geometry, i.e. based on the geometry at the *end* of each increment (compare this to `***bc **pressure` at page 3.75). Hydro pressure will be defined according to the geometrical normal of a line set in 2D or a face set in 3D.

The pressure imposed on a body is applied in the direction of the surface's normal[3]. For instance to model a solid submerged in fluid, the pressure's value should be negative if the mesh uses the convention of outwards normals, and positive if normals point inwards.

See also `**hydro_finite_strain`, on next page.

Syntax:

`**hydro` *bset value* [*table*]

---

bset This is the character name of a boundary set, i.e. a line set for 2D problems or a face set in 3D problems.

value Real value for the base or scaling value of the hydrostatic pressure. The table value is scaled by this factor to calculate the pressure at a given time.

table Character name for a valid loading table or tables which will describe the pressure in time.

Example:

```
**hydro face1 -100.0 tab1
```

[3]You may verify a bset's normals in Zmaster, and change them with the mesher `bset_align` (p. 2.23).

**hydro_finite_strain

Description:

This boundary condition is similar to the `**hydro` boundary condition: it applies a surface pressure calculated on the updated (deformed) geometry. Both conditions apply the same loading forces.

`hydro_finite_strain` computes the exact (non-symmetric) tangent matrix by perturbation, whereas `hydro` only computes a (symmetric) approximation of the tangent matrix. Both conditions add non-linearities to the problem. Thus, `hydro_finite_strain` has a better convergence rate, at the expense of additional computational cost (especially noticeable if this condition is the only one that adds non-symmetric terms to the tangent matrix).

Syntax:

`**hydro_finite_strain` *bset value* [*table*]

---

bset This is the character name of a boundary set, i.e. a line set for 2D problems or a face set in 3D problems.

value Basic value for the base or scaling value of the hydrostatic pressure. The table value is scaled by this factor to calculate the pressure at a given time.

table Character name for a valid loading table or tables which will describe the pressure in time.

Example:

The following example is taken from `$Z7TEST/Hyperelastic_test/INP/ring-axi.inp`:

```
***resolution
  % [...]
 **no_symmetrize   % mandatory for hydro_finite_strain

***bc
 **hydro_finite_strain
   upstream   -40.1e5 time
   tip        -40.0e5 time
   downstream -40.0e5 time
```

**impedance

Description:

The impedance boundary condition can be useful, for example, for modeling the presence behind an interface of a material having different wave propagation properties. It can be used to tune the part of reflection and transmission of an incident wave on an interface. The characteristic impedance Z of a medium is a material property defined as :

$$Z = \rho c$$

where ρ is the density of the medium and c is the longitudinal wave speed.

The impedance bc is a Robin condition that links the stress on the boundary with the velocity :

$$\underline{\underline{\sigma}}\vec{n} = Z\,(\vec{v}.\vec{n})\,\vec{n} \quad \text{on } \Gamma_i$$

Where $\vec{n}$ is the unitary outward normal vector to the boundary Γ_i. $\underline{\underline{\sigma}}$ and $\vec{v}$ are the stress tensor and the velocity respectively. Numerically, impedance is handled through additional terms in the damping matrix (see calcul dynamic page 3.9).

A classical situation where an impedance bc is needed is the hopkinson experiment. An input bar impacts a sample from a side, a wave is generated and propagates though the sample in contact with an output bar on the other side. To simulate the sample/output bar interface without meshing the output bar, an impedance bc can be used with Z equal to the output bar characteristic impedance.

Here are two special cases to illustrate the role of the impedance in the reflection/transmission behavior of a wave at the interface between two media. Lets take two media having impedances Z_1 and Z_2 respectively and lets consider a wave coming from material 1:

- if $Z_2/Z_1 = 0$ (medium 2 is void for instance), the wave is totally reflected
- if $Z_2/Z_1 = 1$, the wave is totally transmitted to medium 2

Syntax:

```
**impedance bset value
```

Where *bset* is the name of the bset where the impedance bc is applied and *value* is the impedance factor.

Example:

```
***bc
 **impedance
   interface  40.e6
```

**linear_rotation

Description:
This condition is a linearized version of the `**rotation` boundary condition (page 3.78). For small deformation problems (not updated), this condition should be used.

This version uses the approximation that angles will be small, so $\sin\theta \approx \theta$. Large rotations will of course result in an expansion of the boundary.

Syntax:
The syntax is the same as for `**rotation`. For 2D problems we have:

```
**linear_rotation [ node ]
  nset_name ((origin) | num_node )
  angle [ table ]
```

For 3D problems, the syntax is slightly changed:

```
**linear_rotation [ node ]
  nset_name [ num_node ] (axe)
  [ (origin) ] angle [ table ]
```

See the discussion on page 3.78 for more detail on the input structure.

**positive_displacement

Description:
This boundary condition is used to ensure $U.v \geq 0$, where v is the direction along which the positive displacement must be verified. This is very similar to a contact problem, and it works by creating a node-to-rigid contact element, as described in the PhD thesis of M. Barboteu.

"Contact, frottement et techniques de calcul parallèle", M. Barboteu, Université de Montpellier, France, March 1st, 1999 (see pages 24–26).

The additional DOFs are named `PD`.

Note that one has to use the extended convergence info, since this BC involves additional degrees of freedom which are forces (and not displacement). This boundary conditions only works in 2D.

Syntax:
The syntax required to `positive_displacement` is:

```
**positive_displacement
  nset_name  dir
```

nset_name is the name of the nset on which the boundary condition is to be imposed.

dir is the direction along which the positive displacement is imposed.

Example:
This example assumes that a mesh with nsets `ligament`, `lip` and `top` have previously been created. It simulates a cracked plate under tension. The crack lip is denoted as `lip`. This positive displacement is imposed on the nset `lip` to avoid the crack to close and subsequent mesh inter-penetration in the crack lip region.

```
 ***mesh plane_stress
[...]
 ***resolution
  **sequence
   *time 600.
   *increment 1000
   *algorithm p1p2p3
   *ratio (U:.05   absolu EZ:1.e-6   PD:1.e-8)

  ***bc
   **impose_nodal_dof
     ligament U2 0.
   **pressure
     top 400. load_cycle
   **positive_displacement lip 0. 1.
```

In this example, the DOF `PD` is attached to the positive displacement BC, and `EZ` appears because of the `plane_stress` formulation.

**pressure

Description:

The pressure mechanical boundary condition is used to impose a pressure normal to a surface defined by a `liset` (in 2D) or a `faset` (in 3D).

The pressure imposed on a body is applied in the direction of the surface's normal[4]. For instance to model a solid submerged in fluid, the pressure's value should be negative if the mesh uses the convention of outwards normals, and positive if normals point inwards.

Pressure is calculated on the *initial geometry* for small deformation/small displacement formulations, and on the *updated* (deformed) geometry at the *start* of the increment for small deformation/large displacement formulations. Therefore, in *large deformation* problems this option may not be appropriate for modeling the fluid pressure on a surface. The `**hydro` and `**hydro_finite_strain` commands (pages 3.70 and 3.71) may be used to apply a pressure on the deformed surface to simulate hydrostatic fluid pressure.

Syntax:

```
**pressure
  liset value table ... [tableN]
  faset value table ... [tableN]
```

liset Character name of the line set upon which pressure is applied in 2D problems.

faset Character name of the face set upon which pressure is applied in 3D problems.

value Base value (real number, function or file) which scales a table value to determine the pressure magnitude.

table Character name for a valid loading table or tables which will describe the pressure in time.

[4]You may verify a bset's normals in Zmaster, and change them with the mesher `bset_align` (p. 2.23).

**radial

Description:
This boundary condition allows imposing radial displacements which simulate the expansion or contraction around a group of nodes. An example is the shrinking or expanding of tubes.

Syntax:

```
**radial
  nset_name origin direction value [table]
```

nset_name Character name of the nset where the condition is applied.

origin Vector form for the origin where the expansion axis passes. A vector form will give the two or three coordinates necessary in parenthesis. An example is `(1. 0. 1.)`.

direction Direction of the axis of expansion. These directions must coincide with the problem coordinate axis, and are defined with the tokens `d1`, `d2`, and `d3`.

value Real number giving the base value for the conditions magnitude. A positive value results in expansion and negative contraction.

table Name of a table describing the load magnitude through time (see `***table`).

Example:

```
**radial
   str (0.0 0.0) d3 3.0 tab1
```

In 2D the expansion axis is necessarily `d3`.

**radius

Description:
The radius condition imposes an indentation profile of constant radius. The condition is valid only in 3D mechanical problems with the radius profile in the U2 direction.

The radius profile is applied as the inverse of the radius during deformation. This allows a zero radius (flat surface) to be deformed in sequence to the given radius. The surface is therefore tightened into the radial form. The exact expression used to calculate the radius at the increment i between the start of a sequence s and the end e is:

$$\frac{1}{R_i} = \frac{1}{R_s} + \frac{i}{N_{inc}} \left(\frac{1}{R_s} - \frac{1}{R_e} \right)$$

with N_{inc} the number of increments in the sequence.

If the radius value is positive, the center of the deformation profile will be located in the negative z space. The inverse statement is of course true as well.

Syntax:
The radial condition takes the following syntax:

```
**radius   nset_name value [table]
```

nset_name The character name of the node set upon which the condition is applied. This node set must be defined in the plane $z = 0$.

value Base value (real) for the radius. The base value is a multiplicative scale for the table values at all times.

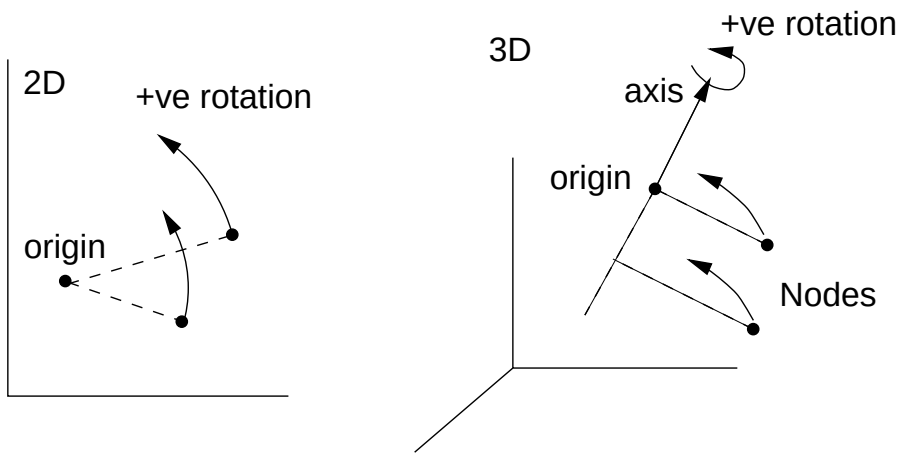
table Character name for the table which describes the condition's magnitude.

Example:

```
**radius
   surface  100.0 tab1
```

**rotation

Description:
This condition rotates all or part of a structure about (1) a fixed point, (2) a node of the mesh. This condition functions in 2D or 3D mechanical problems. For the 2D case, the rotation is always about the third direction axis.



Syntax:
The syntax for 2D problems is the following:

```
**rotation [ node ]
  nset_name ((origin) | num_node )
  angle [ table ]
```

For 3D problems, the syntax is slightly changed:

```
**rotation [ node ]
  nset_name [ num_node ] (axe)
  [ (origin) ] angle [ table ]
```

`node` Optional keyword which specifies that the condition's center of rotation will be defined about a node of the mesh.

nset_name Name of the node set upon which the condition is active.

origin Origin of rotation from which the displacement field is calculated. This point is given using vector syntax.

num_node Integer node number defining the center of rotation in the event that the `node` keyword was given.

angle Real value in degrees acting as the multiplicative factor for the condition magnitude. At any time, the rotation angle will be this value times the given table value.

table Name of a valid loading table.

```
****calcul
 ***bc
  **rotation
```

axe Vector giving the axis of rotation. Note that the Cartesian axes are defined by X: `(1. 0. 0.)`, Y: `(0. 1. 0.)`, Z: `(0. 0. 1.)`.

Example:
Several examples of the rotation syntax are given below. These each give the corresponding test name user the `Static_test` directory.

```
% ztole3m
**rotation bord (0. 0.) 360.0 tab1

% hook_ld2
**rotation node sommet 1 180. tab

% cuberot
**rotation rot (1. 0. 0.) (0. 0. 0.) 360. table
```

**shear

Description:
The shear BC imposes a tangential pressure along a pre-defined line set in an analogous fashion as in the **pressure command. it is impossible to impose a shear along a face set, thereby limiting its applicability to 2D problems.

Syntax:

```
**shear
  liset value table ... [tableN]
  ...
```

liset Character name of the line set where the shear pressure is applied.

value Real base or scaling value for the condition.

table Character name for a valid loading table or tables which will describe the pressure in time.

Example:

This example is presented as a useful template for testing material behavior in shear. The single square element is defined as follows:

```
***geometry
**node  8  2
  1  0.0000000  0.0000000
  2  1.0000000  0.0000000
  3  1.0000000  1.0000000
  4  0.0000000  1.0000000
  5  0.5000000  0.0000000
  6  1.0000000  0.5000000
  7  0.5000000  1.0000000
  8  0.0000000  0.5000000
**element  1
  1   c2d8   1 5 2 6 3 7 4 8
***group
**liset s1
  quad         1     5     2
  quad         3     7     4
**liset s2
  quad         2     6     3
  quad         4     8     1
*nset bottom
  1 5 2
**nset corner
  1
***return
```

```
****calcul
 ***bc
  **shear
```

A shear stress loading may be applied with the following boundary conditions in the input file:

```
***bc
  **shear
    s1     exp  1. tab
    s2     exp -1. tab
  **impose_nodal_dof
    bottom  U2 0.0
    corner  U1 0.0
```

**static_torsor

Description:
The static_torsor BC imposes a static torsor (resultant and momentum) on a nset. This means that the sum of the action of every nodal forces of the nset is equivalent to the specified static torsor. It is an ill-posed problem since we add $3N$ unknowns (nodal forces in the three directions) and only 6 equations (3 equations for the resultant and 3 for the momentum). This BC is therefore associated to a rigid body motion of the nodes of the nset through a **mpc_rb relationship (see page 3.139). Note that the user doesn't have to define the **mpc_rb relationship, it is done automatically.

When a **mpc_rb is applied, only master 6 dofs over every dofs of the nset remain in the linear system. A small system of equations is then solved where the 6 remaining external forces associated to the 6 master dofs are computed.

Syntax:

```
**static_torsor
 *nset          nset
 *resultant     vector
 *momentum      vector
 *point         vector
 *table         value table_name
```

Where *nset* is the name of the nset where the torsor is applied. *resultant and *momentum define the resultant and the momentum at the point defined by the *point command. *value* and *table_name* define the multiplicative factor of the torsor.

Example:

```
***bc
 **static_torsor
  *nset       top
  *resultant  (5.    1.   0.)
  *momentum   (0.01  0.0  0.)
  *point      (1.    0.5  0.5)
  *table      1.  time
```

**strain_gradient

Description:
The syntax is as follow :

Syntax:

```
**strain_gradient
  nset  dof  value  table
```

**surface_heat_flux

Description:
The **surface_heat_flux condition is used to impose a surface heat flux on a line set in 2D or a face set in 3D (i.e. a Neumann boundary condition). In 3D and SI units, it should be given in $W.m^{-2}$.

Z-set's convention is chosen such that a positive flux heats the material body, i.e. it is an inflow flux. In other words, it is integrated assuming an inward normal (note that the actual liset or bset alignment is not taken into account).
This keyword replaces the now deprecated keywords **flucons and **surface_heat.

Syntax:

```
**surface_heat_flux
  liset value table
  faset value table
```

liset Character name for the line set upon which the heat flux is applied in 2D.

faset Character name for the face set upon which the heat flux is applied in 3D.

value Base value (see 3.43) for the inflow heat flux. This value scales the current table value to obtain the heat flux magnitude.

table Character name of a predefined loading table or list of tables (see the option ***table p. 3.227). The table value determines the exterior temperature.

Example:

```
**surface_heat_flux
  myliset   -10. table1

**surface_heat_flux
  bottom  function  exp(x)*sin(y)*cos(y+z) ; table2
```

**convection_heat_flux

Description:
The **convection_heat_flux BC imposes a convective heat flux on a line set in 2D or a face set in 3D:

$$q_c = a(T - T_e)$$

where $q_c = -K\underline{\nabla}\mathbf{T}.\underline{\mathbf{n}}$ is the outflow flux, a the convection coefficient, T_e the exterior temperature and $\underline{\mathbf{n}}$ the outward normal (note that the actual liset or bset alignment is not taken into account).
This keyword replaces the now deprecated keyword **fluconv.

Syntax:

```
**convection_heat_flux group
  h  a
  Te Te table
```

group Character name for the group of line sets in 2D or face sets in 3D where the flux condition is to be applied.

a Real value for the convection coefficient.

Te Real value for the exterior temperature T_e.

table Character name of the table describing the flux value in time (see ***table).

The coefficient a may depend on the temperature for this condition. The syntax required for a convection heat flux condition with this dependence is given below:

```
**convection_heat_flux group
  h temperature
  h0  T0
  h1  T1
  h2  T2
  ...
  Te  Te table
```

The keyword **convection_heat_flux and the two following lines must be repeated as many times as there are convective flux boundary conditions.

Example:

```
**convection_heat_flux myliset
  h   100.0
  Te  -10. table1
```

**interface_heat

Description:
The **interface_heat BC option is used to impose an inter-facial heat transfer with thermal resistance between two line set boundaries in 2D, or two face set boundaries in 3D. The inter-facial flux will be calculated as:

$$\text{flux} = a(T_1 - T_2)$$

where T_1 and T_2 are the temperature at each side of the interface.

The two interfaces must be in geometrical correspondence, which may be summarized as follows:

- A line or face at one side of the interface must correspond to exactly one line or face on the other side of the interface.
- The nodes in the lines or faces in correspondence must be paired exactly one to one. For this to be true, it is necessary that the distance between two corresponding nodes be inferior than the distance ϵ. In result of this requirement is the fact that all nodes which are *not* in correspondence have a separation distance *greater* than ϵ.

This keyword replaces the now deprecated keyword **fluconv_interface.

Syntax:

```
**interface_heat  group1  group2  ε
  h  a
```

group1 Character name for the first line set group in 2D, or face set group in 3D.

group2 Character name for the second line set group in 2D, or face set group in 3D.

ϵ Real value for the critical distance between nodes.

a Base value (real) for the value of the coefficient a.

The coefficient a may depend on the temperature for this condition. The syntax required for an inter-facial convection resistance condition with this dependence is given below:

```
**interface_heat group
   h    temperature
   h0   T0
   h1   T1
   h2   T2
   ...
   Te   Te table
```

The keyword **interface_heat and the two following lines must be repeated as many times as there are convective flux boundary conditions.

Example:

```
**interface_heat liset1 liset2 0.001
   h   100.0
```

**volumetric_heat

Description:
This BC is used to impose a volumetric heat flux within a given element set. In 3D and SI units, it should be given in $W.m^{-3}$.
This keyword replaces the now deprecated keyword `**fluvol`.

Syntax:

```
**volumetric_heat
  elset value table
```

elset Character name for the element set (`elset`) within which the heat generation is applied.

value Real base value for the heat flux. This value scales the current table value to obtain the heat flux magnitude.

table Character name of a pre-defined loading table or list of tables (see the option `***table` p. 3.227). The table value describes the magnitude of the flux in time.

Example:

```
**volumetric_heat
    str  20.0 tab1
```

**volumetric_heat_from_parameter

Description:
This BC is used to impose a volumetric heat flux within a given element set. The time-space values are taken from a parameter field.

Syntax:

```
**volumetric_heat_from_parameter
  elset value parameter_name
```

elset Character name for the element set (`elset`) within which the heat generation is applied.

value Real base value for the heat flux. This value scales the given parameter field to obtain the heat flux magnitude.

parameter_name Name of the parameter.

Example:

```
**volumetric_heat_from_parameter
    str  20.0 param1
```

**volumetric_heat_in_file

Description:
The `**volumetric_heat_in_file` BC is used to impose a volumetric heat flux within a given element set, using input from a file.
This keyword replaces the now deprecated keyword `**fluvol_in_file`.

Syntax:

```
**volumetric_heat_in_file
  elset value file-prefix
```

elset Character name for the element set (`elset`) within which the heat generation is applied.

value Real base value for the heat flux. This value scales the current table value to obtain the heat flux magnitude.

file-prefix pre-fix for the input file. The first file of concern is *file-prefix*`.catalog`, which lists the initial and final times for the file inputs (presumably from another time step of a mechanical problem, see below). Binary files for the beginning of the problem, start of an increment and end of an increment are: *file-prefix*`.first` *file-prefix*`.initial` *file-prefix*`.final` The current increment should be within the time bounds given in the catalog file.

Example:

```
***sub_problem fem MechTherm/plastic
 **transfer integ_nodeparam
  *variable q_dot
  *file     heat_out
```

In `thermal.inp` the heat is imported using the current BC:

```
 **volumetric_heat_in_file
   ALL_ELEMENT 1.0 heat_out
```

An example of what's in the catalog file is:

```
Time_ini 7.600000
Time  8.000000
```

**radiation

Description:
This BC for thermal problems applies a radiation heat flux on a line set (`liset`) in 2D, or a face set (`faset`) in 3D problems. The expression for the heat flux is:

$$\text{flux} = a(T^4 - Te^4)$$

Syntax:

`**radiation` *group a Te table*

---

group Character name of the previously defined line set in 2D or face set in 3D.

a Real value for the coefficient a.

Te Real value for the exterior temperature, T_e, scale factor. This value will be a multiplicative scale of the current table value to determine the external temperature.

table Character name of a pre-defined table name (see `***table`). This table describes the magnitude of the external temperature

The coefficient a is constant. There is currently no provision for temperature or time dependence in this parameter. We recall that the coefficient is the product of the two terms:

$$a = \sigma\epsilon$$

with σ the Stefan's constant ($5.73\ 10^{-8}\ \ W/m^2\ K^4$) and ϵ the grey body constant (no dimension).

The formula of radiation is not valid except for temperatures defined in on the absolute scale (Kelvin). This requires therefore that the units throughout the problem be defined consistently in Kelvin when radiation boundary conditions are applied.

Example:

```
**radiation myliset 1.0e8  273.0 table1
```

***coupled_resolution

Description:
This command is used to specify special controls relating to the convergence of weak coupled problems.

Syntax:

```
***coupled_resolution
 **iteration iter
```

Example:
An example control for a coupled problem follows (from `$Z7PATH/test/Coupled_test/INP/MechTherm.inp`):

```
***resolution
 **sequence
  *time      4.0 8.0
  *increment 10 10
  *ratio     1.e-4
***coupled_resolution
  **iteration 2
```

***compute_G_by_gth

Description:
This command is used to calculate the energy release rate G along a 3D crack front using the the energetic G-θ method. Various options may be activated to:

- smooth the crack front by defining the number of points of the spline curve used to build the variational problem solved to obtain the energy release rate (G) and crack virtual extension (θ) at each control point of the underlying spline,
- output results in terms of conventional stress-intensity factors (SIF) K_I, K_{II}, K_{III} instead of the default G values,
- define propagation laws based on the previous SIF calculations, that will generate crack advance values that may be used to drive crack propagation by means of the `***auto_remesh` command,
- define various crack bifurcation criteria for out-of-plane crack propagations.

Note that this command may be applied either to conforming cracks FE models (ie. the crack is explicitly introduced in the mesh, and the crack front is defined by a liset) or xfem ones (in this case the discontinuity is defined by means of level-sets in a `***xfem_crack_mode` block of commands, see 3.233). Note also, that the input for this command may be built automatically by the interactive `Zcracks` and `Zxfem` commands.

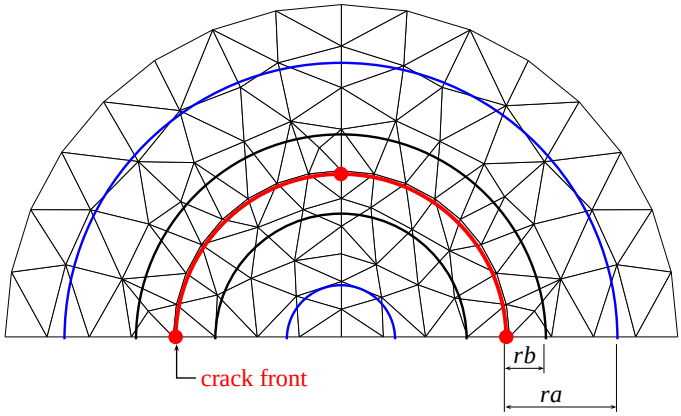
Syntax:
The syntax is as follows:

```
***compute_G_by_gth component-name
[ **crack_front  liset-name ] | [ **xfem ]
  **nbnodes  nbnodes
[ **theta ra [ rb ] ] | [ **elem_radius  nbe ]
[ **elset  elset_name  ]
[ **exclude_BE ]
[ **lip  lipname  ]
[ **compute_K ]
[ **compute_Ki ]
[ **fatigue  preload period  ]
[ **behavior  btype  ]
[ **Delta_N  deltan  ]
[ **h  hmax  ]
[**optim_dir prec scan ]
[ **vectorial ]
```

The different *option-keys* are

- *liset-name* is the liset (line set of nodes) defining the crack front for conforming mesh modelizations. This liset is usually automatically created in the FE mesh by the `Zcracks` interactive script, and maintained during propagation analysis.
- alternatively the command `**xfem` should be used for xfem models.

- as defined on the next figure, the parameter *nbnodes* (**nbnodes command) defines the number of points on the spline curve built internally to smooth out the crack front. Note that a negative values can be given for the *nbnodes* parameter, in which case the number of spline control points is calculated as a fraction of the total number of crack front nodes (number of nodes in liset *liset-name* in the conforming mode).



red: crack front
blue: *G* integration domain frontier (defined by a distance *ra* to the crack front)
black: interior domain eliminated during *G* integration (defined by the *rb* radius)

● spline control points used to smooth the crack front (number of points defined by the **nbpoints *nb* command

- the **theta command defines the size of the integration domain used to build the G-θ variational problem. As shown on the figure, this domain has the shape of a ring, centered around the crack front, where *ra* is the radius of the circular section of the ring. The optional parameter *rb* may also be entered, and correspond to the size of an inner volume, close to the crack front: in this case, the mechanical fields on the first rows of integration points/elements near the front are not considered precise enough, and are eliminated from the integration domain. Note that in practice, the outer radius *ra* of the integration volume, should be chosen to stay within the volume (ie. it should not reach the outer free surface) to avoid erroneous $G-\theta$ solutions.

- the command **elem_radius is an alternative way to specify the size of the integration domain. In this case the outer radius *ra* is calculated to correspond to *nbe* times the average size of crack front elements.

- the optional command **elset can be used to speed up the G-θ volume definition, ie. only candidate elements in elset *elset_name* will be tested for addition in the volume integration.

- in general, values near free surfaces (at both ends of opened crack fronts) are less precise than inside the volume along the crack front. The optional command **exclude_BE may be used to reset those values, and calculate SIFs at the front ends by extrapolation of values obtained on inner points.

- command *lip is not used in the xfem mode. *lipname* is a name of an nset in the FE mesh, containing nodes on the crack lips. In the conforming mode this nset is used to

calculate the tangential plane at each crack front point, an the orientation of the front normals at this point.

- when the optional **compute_K command is specified, the default G energy release rate output, is replaced by the mode I K_I stress intensity factor. This K_I value is obtained from G using the following equation:

$$K_I = \sqrt{\frac{E\,G}{1-\nu^2}}$$

Note that in order to evaluate the previous equation, the $G-\theta$ module needs to retrieve the elasticity coefficients of elements close to the current crack front node, and that the influence of external parameters (eg. temperature) on those parameters is accounted for. Note also, that when this option is used, the propagation law behavior automatically takes K values as input, instead of the default G energy release rate, such that coefficients of those laws should be modified accordingly.

- the optional **compute_Ki command activates the calculation of the K_I, K_{II}, K_{II} stress intensity factors by means of an interaction integral computation based on the Westergaard analytical solution. This choice has an impact on the branching criterion for out of plane propagation, that automatically switches to a modal mixity angle α calculated from those stress intensity factors instead of the default $Gmax$ based criterion. In this case, α is computed using the following equation:

$$\alpha = 2\ arctan\left(\frac{-K_{II}/K_I + \sqrt{(K_{II}/K_I)^2+8}}{4}\right)$$

- command **fatigue *preload period* is mandatory for fatigue crack propagation, and defines the cycle period (parameter *period*) and a preloading initial time value at the beginning of the calculation during which crack propagation is ignored (parameter *preload*). Depending on the above definitions, crack propagation will occur at times $t = preload + n_{cyc}\ period$, where n_{cyc} is the cycle number.

- command **behavior *btype* defines the model *btype* selected to compute crack advance, that will drive crack propagation and automatic remeshing procedures (by means of the ***auto_remesh command). Currently the only model available is the Paris law for fatigue crack propagation:

$$\theta = \frac{da}{dN} = C\,(\Delta G)^m$$

where ΔG is the G variation during the cycle (or K if the **compute_K keyword is activated) computed at each point along the crack front.

- command **Delta_N *dn* defines a multiplication factor *dn* applied to the crack advance θ calculated with the propagation law specified by the **behavior command. This option may be important to insure that a significant crack advance will occur at each simulated cycle in the FE analysis, and that automatic remeshing is indeed pertinent.

- command **h *hmax* is another way to control the value of the multiplication factor *dn* applied to the θ values. When a positive value is given, *hmax* is the target crack advance when remeshing occurs, and the multiplication *dn* is calculated accordingly:

$$dn \quad such\ that \quad hmax \ = \ \max_i \left\{ dn \ \frac{\theta_i}{\theta_{max}} \right\}$$

  where θ_i is the crack advance at point i on the crack front, and θ_{max} the max value of θ_i on all crack front points. When a negative values is specified, *hmax* defines a target crack advance of *hmax* times the mean size of elements along the crack front.

- the command **optim_dir *prec scan* triggers out-of-plane propagation. Without this keyword, the crack advance direction always lie in a plane corresponding to the initial crack definition: in the xfem mode this plane is characterized by the first levelset definition, and in the conforming mode the **lip nset allows this tangential plane calculation at each crack front point.

  When a G_{max} bifurcation criterion is used (which happens when no keyword **compute_Ki has been specified, see above), 2 methods are available to compute the angle α_{max} (at each crack front point) for which $G = G_{max}$.

  First one (default case) is a dichotomy search where parameter *prec* is the precision, and *scan* the angle variation used during the scanning of angle values needed to initiate the dichotomy algorithm. Angle values should be entered in degrees, and a typical definition is:

  ```
  **optim_dir 1. 15.
  ```

  for a precision of 1 degree in the calculation of the modal mixity angle, and a scan of values every 15 degree to calculate the initial dichotomy interval.

  The second method is activated with the **vectorial keyword, and is based on the calculation of G in 2 orthogonal directions. The first direction, along the tangential crack direction, gives a so-called G_I value. Then a second G_{II} value is calculated in a direction orthogonal to this tangential plane. The branching angle α for the new propagation direction is then defined as: $\alpha \ = \ arctan\left(\frac{G_{II}}{G_I}\right)$.

Example:

```
% Conforming mode example:
***compute_G_by_gth FRONT0
 % G-theta problem definition
 **crack_front FRONT0
 **nbnodes -8
 **elem_radius 3
 **elset NEW
 **lip lip
 % SIF output options
 **compute_K
 **compute_Ki
 **excludeBE
```

```
****calcul
 ***compute_G_by_gth
```

```
 % crack propagation law definition
 **fatigue 0.000000e+00 2.000000e+00
 **behavior paris
  C 1.000000e-06
  m 3.000000e+00
 **Delta_N 100
 **h -2.0
 % out-of-plane propagation
 **optim_dir 1.000000e+00 1.500000e+01
```

Example:

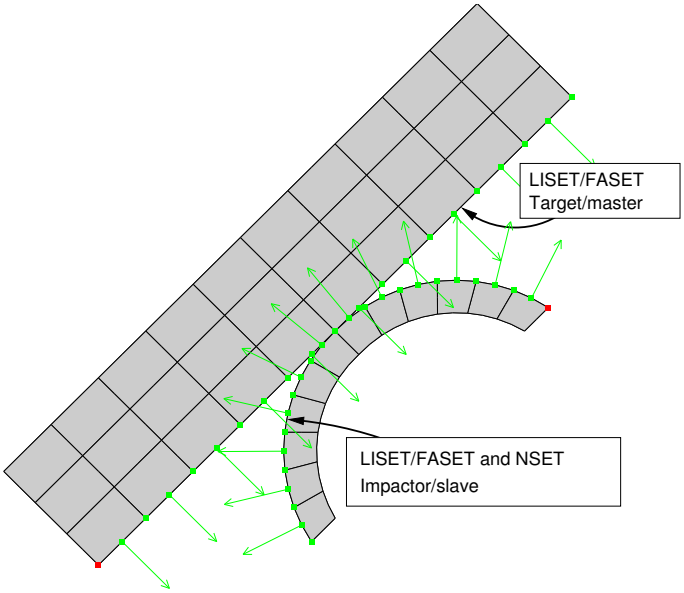
```
% XFEM mode example (note, no out-of plane propagation)
***compute_G_by_gth gtheta_a
   **xfem
   **theta 0.3 0.0
   **nbnodes 3
   **behavior paris
     C 335.298
     m 3.74
   **Delta_N 2000
```

***contact

Description:

The ***contact section defines zones of contact in static and dynamic mechanical problems. Typically, applying contact is one of the more troublesome aspects of nonlinear finite element analysis, because of increased discontinuities and sometimes unavoidable over- and underconstraints. This is above the obvious increase in modeling and input file complexity. After Z-set version 8.2, major updates to the contact system have been made which hopefully increase the robustness and quality of contact mechanics. The algorithms of Z-set are particularly good at avoiding global convergence issues with chatter, and in performing well when there are large differences in the relative stiffnesses of the contacting parts.

Contact is enforced with a group of *contact zones*. Zones are modeled by a node set acting as the *impactor* (*slave*) surface which may come in contact with a *target* (*master*) surface area. Each zone may have a different contact model (law) with or without frictional behaviors, and can include a variety of control parameters. The contact condition is defined in terms of a contact *stress*, but can alternatively be defined directly in terms of nodal reaction *forces*.[5]



To define contact zones, one must identify target/master surface areas, and corresponding impact/slave sets of nodes. Under normal use, i.e. stress-based contact, a node set (*nset*) is required for the impactor side, and a boundary set (*liset* or *faset*) is required for *both* target and impactor.[6] The boundary sets *must* be set up so that the normals point outward (see page 6.7), which can be verified graphically in Zmaster using the `Bset normals` option of the `Mesh Bset...` command. An example view of a properly set up two-dimensional example is given in the above image, where the target nset is not visible because it coincides with its corresponding liset/faset (as it should).

[5] Note that for second order interpolation functions the magnitude and even direction of the nodal reactions is not intuitive. The fact is that treating nodal reactions directly assumes a discrete force acting over a zero area, which of course results in a singular stress in a continuum. Thus for these higher interpolations the only real choice is to deal with stresses distributed over an area.

[6] This is in contrast to older versions where a bset was required for the target only. A facility is available to automatically generate the impactor bset from its nset, but in general it is preferable to set this up ahead of time. One can use the ***mesher **bset command (page 2.21) with the *use_nset sub-command.

Note that if contact is occurring on only one node, or if there is a zero radius corner in contact, the contact stress will still be distributed over the adjacent boundaries. If the purpose of this modeling is to impose a boundary condition, or if the frictional behavior is not important, it is possible to use the `**force_basis` option, or to define the impactor boundary set with a dot set. In the latter case, each node has unit area and does not have any additional connectivity.

Solution method:
Contact introduces an additional system of constraints on the displacement field. These are implicitly defined in terms of the contact stresses. The contact strategy in Z-set consists in solving a sub-problem for the contact stresses and relative displacements between surfaces, from which the additional global constraints are computed (a process very similar to a sub-structuring problem). The nonlinear contact solution is thus done by running "sub-space iterations" depending on the chosen contact law, and a *flexibility matrix* found from the global stiffness. The flexibility matrix is a (possibly) full matrix describing the deformation of each node on a contacting surface due to a unit force on each other node on the contact surface. Note that these terms will only be zero if the nodes are on separate bodies. The contact-modified solution intervenes in the global iterative procedure in several steps:

1. An estimate of the global field is made *without* enforcing the contact in the current iteration, by applying the classic Newton method using the last converged residual. This residual consists of both the internal reactions and the added contact reactions from the last iteration.

2. The flexibility matrix is computed from the global stiffness matrix by using repeated back solves of unit loads over all contact nodes. Because back solves are being made, it is very advantageous to have all the contact nodes numbered at the end of the global unknowns. However, this will in general increase the total bandwidth of the problem. Very often the computation of the flexibility matrix will be the most important part of the solution time of a contact problem. Currently the sparse matrix solvers include a compromise renumbering scheme for best performance.

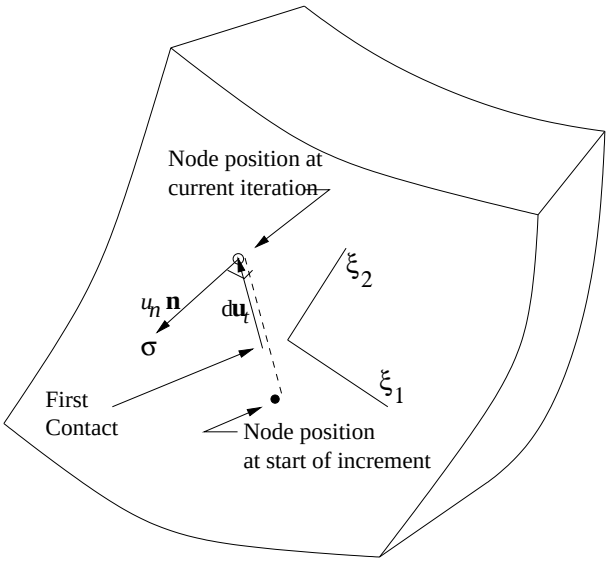
3. The next step is then proceeding with the "inner" iterations by using this flexibility matrix to solve for the contact stresses and displacements. There are both direct solvers (involving the inverse of the flexibility matrix) and block diagonal iterative solvers in case that direct solution becomes too costly. The solver can be chosen with the `**solve_method` option. By default, the currently implemented contact laws use a direct solver. Running in verbose mode (either using the `-v` command line switch, the `**verbose` output option, or by using the `**verbose_contact` contact option) will show a great deal of additional output on the progress of the local contact solution. This option should be enabled if there are any difficulties with the contact solution.

4. The last stage updates the set of global residual equations to include the contact reactions, which will in turn be used for the next dof prediction in the global Newton iterations.

The contact algorithm outlined above has many advantages, at the cost of computing the flexibility matrix. Firstly the method isolates the severe nonlinearities involved in solving the contact state from the global material and geometry nonlinearities. In particular, the effect of "chattering" is almost totally isolated to the local contact iterations, where it is very much more efficiently treated. In addition, the local system defining the contact is nonsymmetric

in the case of sliding friction, but because these terms are solved only in the local iterations, the global matrix can remain symmetric even under high friction (as long as other equations have led to a symmetric tangent before the contact has been considered).

Contact behavior:
The contact behavior is defined in terms of the quantities schematically represented in the following figure:



As shown, there is a *normal mode* of contact (to be enforced to null within a given limit), and a *shear mode* with a displacement vector defined in local coordinate axes (ξ_1, ξ_2) of the face element. The following variable definitions are used in the descriptions of the contact laws:

σ the normal contact stress (scalar), defined with compressive values being positive.

$\boldsymbol{\tau}$ the tangent stress vector.

u_n the normal displacement (scalar) between potentially contacting surfaces. Values greater than zero have a gap between the surfaces and values less than zero indicate a penetration.

$d\mathbf{u}_t$ the incremental tangent displacement (vector), measuring the projected slip on the element face during the increment.

k_n, k_t intrinsic contact stiffnesses in the normal and tangent directions (units of stress/length), as estimated by the code from the global matrix.

μ (not drawn, dimensionless scalar) the friction coefficient as a fraction of the normal stress.

With these variable definitions we describe the following contact models:

`normal contact` with the following auxiliary variables $\hat{\sigma} = \sigma - k_n u_n$ and $\hat{\boldsymbol{\tau}} = \boldsymbol{\tau} - k_t d\mathbf{u}_t$,

the normal contact law is expressed as

if $\hat{\sigma} > 0$
then
$k_n u_n = 0$
if $|\hat{\boldsymbol{\tau}}| < \mu\sigma$
then $k_t d\mathbf{u}_t = 0$
else $\boldsymbol{\tau} = \mu\sigma\mathbf{n}_t$ with $\mathbf{n}_t = \hat{\boldsymbol{\tau}}/|\hat{\boldsymbol{\tau}}|$
else $\sigma = 0, \boldsymbol{\tau} = 0$

The normal and tangential stiffnesses $k_n > 0$ and $k_t > 0$ are normalizing parameters which are automatically set by the program.

`penalty contact` is identical to normal contact, except that it allows for the contact constraints to be violated using the penalty parameters in the normal and tangential directions γ_n and γ_t. *The larger the penalty, the more the constraint is satisfied.* Numerically, these penalty parameters tend to make the local contact matrix diagonally dominant and can condition the matrix under situations where other constraints conflict with the contact constraints. When the penalty parameters approach infinity, the penalty contact model behaves exactly as the standard normal contact. In addition, in problems with *dynamic* impact, it is useful to include a term β which effectively damps any oscillations introduced by severe impact, while introducing only a small penalty under less impulsive conditions. Defining the auxiliary forces $\hat{\sigma} = \sigma - \left[\gamma_n^{-1}\sigma + k_n(u_n + \beta\gamma_n^{-1}\, du_n)\right]$ and $\hat{\boldsymbol{\tau}} = \boldsymbol{\tau} - \left[\gamma_t^{-1}\, d\boldsymbol{\tau} + k_t\, d\mathbf{u}_t\right]$, the penalty contact model is expressed as

if $\hat{\sigma} > 0$
then
$\gamma_n^{-1}\sigma + k_n(u_n + \beta\gamma_n^{-1}\, du_n) = 0$
if $|\hat{\boldsymbol{\tau}}| < \mu\sigma$
then $\gamma_t^{-1}\, d\boldsymbol{\tau} + k_t\, d\mathbf{u}_t = 0$
else $\boldsymbol{\tau} = \mu\sigma\mathbf{n}_t$ with $\mathbf{n}_t = \hat{\boldsymbol{\tau}}/|\hat{\boldsymbol{\tau}}|$
else $\sigma = 0, \boldsymbol{\tau} = 0$

`isotropic coulomb` is almost identical to penalty contact, with the exception that the friction stress is limited by a user-defined maximum shear strength τ_{max}. Thus, the *normal* contact model is a subset of the *penalty* contact model, and the *penalty* contact model is a subset of the *isotropic Coulomb* contact model. Defining the auxiliary forces $\hat{\sigma} = \sigma - \left[\gamma_n^{-1}\sigma + k_n(u_n + \beta\gamma_n^{-1}\, du_n)\right]$ and $\hat{\boldsymbol{\tau}} = \boldsymbol{\tau} - \left[\gamma_t^{-1}\, d\boldsymbol{\tau} + k_t\, d\mathbf{u}_t\right]$, the isotropic

Coulomb contact model is expressed as:

if $\hat{\sigma} > 0$
then
$\gamma_n^{-1}\sigma + k_n(u_n + \beta\gamma_n^{-1}\, du_n) = 0$
if $|\hat{\boldsymbol{\tau}}| < \min(\mu\sigma, \tau_{max})$
then $\gamma_t^{-1}\, d\boldsymbol{\tau} + k_t\, d\mathbf{u}_t = 0$
else $\boldsymbol{\tau} = \min(\mu\sigma, \tau_{max})\, \mathbf{n}_t \quad$ with $\quad \mathbf{n}_t = \hat{\boldsymbol{\tau}}/|\hat{\boldsymbol{\tau}}|$
else $\sigma = 0, \boldsymbol{\tau} = 0$

`orthotropic coulomb` is a generalization of the isotropic coulomb model.

Syntax:
Contact takes a number of main control commands, and sub-blocks defining the different zones and behavior models. One can alternately define the contact model to apply to all zones, or zone-by-zone.

The **soft_param and **penalty_param commands are no longer supported in versions 8.2 and newer. This is now specified using the **zone *behavior_coef command, as described below for each specific contact model. A warning will be issued, but these commands will be ignored.

```
***contact [ contact model ]
  **zone [ contact model ]
    zone subcommands
[ **conv precision max_iter [ force_precision ] ]
[ **stable eps [ precondition|damp ] ]
[ **init_d_stress [ sequence ] ]
[ **force_basis ]
[ **solve_method direct|iterative ]
[ **limit_activity t1 t2 ]
[ **spy_node node [ node ] ... ]
[ **verbose_contact [ filename ] ]
```

**zone defines a contact zone, as well as the law defining its contact model. Current (as of version 8.2) contact models are normal, penalty, coulomb and ortho_coulomb. The keyword soft is deprecated and automatically converted to penalty instead. More than one zone may be specified. If multiple zones exist which all use the same contact model, then the contact model may be specified immediately after the ***contact command. Available sub-options are described on page 3.106.

**conv defines the convergence parameters. The first (real) parameter *precision* defines a non-dimensional tolerance defining the *reduction* of the contact residual relative to the residual at the beginning of the inner contact iterations (i.e. the global problem residual *without* the additional contact influences of this global iteration included). A moderate relative value is therefore quite acceptable, even for problems needing a high *absolute final* residual.

The default value of 1.e-5 is quite small, so in most cases this command will be useful. A recommended value of 1.e-2 indicates that the contact convergence is two orders of magnitude smaller than the current global convergence state. Unless it is believed that the influence of contact is very very significantly altering the global behavior, this value should be ok.

The second (integer) parameter is the number of iterations before contact divergence will be signaled (default 200). The last optional (real) parameter *force_precision* indicates the desired magnitude of the contact residual relative to the magnitude of contact stress (default 1.e-5). Only one of the two convergence criteria needs to be satisfied in order to achieve convergence.

Unlike previous versions, the contact solution algorithm in 8.2 always assigns the initial guess of the contact stresses based on the contact stress determined from the previous global iteration. As the global iteration converges, the initial absolute error in the local

contact iteration will also decrease. It is recommended that *force_precision* be set only as tight as the expected convergence rate of the global iteration. When the global iterations converge, the absolute error in the contact law will be satisfied to the same degree as the global iterations.

**stable takes one real value *eps* (default 1.e-2) and the choice between precondition (default) or damping. If the static model contains multiple bodies and at least one of these bodies has a rigid body mode (if the contact constraints are not considered), then this command is absolutely required. For dynamic problems it is not needed. The command instructs Z-set to add a term on the diagonal of the stiffness matrix of the contact nodes, in effect adding a grounded incremental spring to each node involved in contact. The value of *eps* then denotes the relative magnitude of the spring constant.

The specifier precondition instructs Z-set to add *only* the incremental spring to the stiffness matrix. It does *not* add the force of the spring to the external forces. This option stabilizes the contact solution algorithm, but in no way affects the converged solution.

With the specifier damping, not only the incremental spring is added to the stiffness matrix, but also the force of the incremental spring is added to the external forces. This actually affects the converged solution. This specifier can be conveniently used to avoid quasi-static singularity problems when a free body is not initially in contact in the initial loading stages. A small value relative to the problem stiffness is recommended, as is a convergence study to ensure that this numerical technique does not significantly alter the structural behavior.

**init_d_stress causes the algorithm to make an initial guess for the stress *increments* based on the previous solution, thereby possibly accelerating convergence. This has advantages especially for monotonic loading paths. For non-monotonic loading paths, the option sequence disables this guess for the first increment of each sequence, so that the contact algorithm is not confused too much by sudden changes in loading direction. This command is somewhat similar in spirit to the ***resolution **init_d_dof command (see page 3.214). Without this command, the initial guess for the stress *increment* is set to zero, that is, the *initial* guess of the contact stress at the *end* of the *current* increment is assigned a value equal to the contact stress at the end of the *previous* increment.

**force_basis sets the algorithm based on forces instead of stresses. Synonyms: **lumped and **forced_based.

**solve_method specifies the type of solver used for the contact solution. The option direct (default value for all currently implemented contact laws) uses the recommended direct solver. A second option iterative uses less memory, but requires more iterations to converge.

**limit_activity only activates the contact conditions between times t_1 and t_2 (taken as two real parameters).

**spy_node causes Z-set to print out more detailed information of the contact stresses at the specified list of nodes. These nodes must lie in the node sets given under **zone impactor.

**verbose_contact gives additional information about the current state of all contact nodes involved. This information is written to an output file *filename*, or in absence of a user-supplied filename to a file called *problem.contact_info*.

Advice when problems occur:
The following hints may help out with common issues of setting up contact problems.

- Use the **verbose_contact contact option to examine the progress details.
- Check the target (master) surface normals to make sure that they point outward from the target body. This can be easily checked in Zmaster's Mesh mode, under the Bsets.. popup to observe bset normals. In the case that some normals are inverted, one can use the inverse_liset (page 2.76) or faset_align mesher commands (page 2.67).
- Check to see that the target is the stiffer and/or less refined surface.
- Check the contact behavior. All contact models from versions before 8.3 are deprecated. The preferred "standard" model is coulomb which includes penalty parameters for "soft" behavior in the case of overconstraint, can optionally capped friction behavior.
- Check if any of the bodies are unconstrained in any of the degrees of freedom if not for contact. In this case, the global matrix will be singular and will no doubt cause problems during the first prediction of new displacement fields. In this case use the **stable precondition command with a small value (e.g. 1.e-4). This command adds a small value to the global matrix pivots for the contact nodes, stabilizing the global matrix conditioning, but not actually altering the residual computation (i.e. not changing the physics at all). Of course a large stabilization value makes the global matrix less correct, and eventually convergence will be inhibited by this parameter. This method is only needed for static problems.
- Check to see if a body is underconstrained in some dofs, and may not always be in contact. If this is the case one can use the **stable damp command which will include *incrementally* defined springs on the contact nodes with a given stiffness. Please be certain (by doing an appropriate convergence study) that the value of this parameter does not actually alter the results.
- Check to see if there are impactor nodes near the point of expected contact. If the slave surface is too coarse or the warning distance is too large, the contact may not be properly detected. In particular, if the impactor is too coarse there may be cases where the impactor does not penetrate the target, but the target does penetrate the impactor. The definition of the master/slave relationship of contact zones allows the master nodes to penetrate the slave.
- Check to see if there are sharp convex points in the contact zone. This can cause problems because the surface normals vary tremendously on the master surface, or by redistributing the contact stress erroneously around corners of slave surfaces.
- Check the integration order of the elements. Generally underintegrated elements will produce less oscillations (spatial and temporal) in the solution. This is especially the case under highly hydrostatic constraint or in locations where there is a rapid change in the

contact condition (e.g. Hertzian contact problems). This will be especially important for isochoric plasticity and viscoplasticity problems, or otherwise close to incompressible elements.

- Check for odd convergence and ‘chattering” when the contact residual rocks back and forth between two residual values. Sometimes there is a problem with a node alternating between sticking and sliding contact states. In such an instance increase the slipping penalty parameter.

**zone

Description:

Contact zones define different contact pairs (impactor/target), zone by zone control parameters, and contact material behaviors.

Syntax:
The zone syntax is summarized below:

```
**zone [ contact model ]
 *impactor impactor_zone
 *target target_zone
 *behavior_coef
  key-value-pairs
[ *dont_check_bcs ]
[ *variable_friction file [ position ] ]
[ *warning_distance warning_distance ]
```

`**zone` specifies a contact zone with its specific contact model. Details of each model can be found in the following pages. Currently (as of Z-set 8.2), valid choices are `normal`, `penalty`, `coulomb` and `ortho_coulomb`. Multiple `**zone` commands may exist, each with its own contact model. If all contact zones have the same contact model, this model may be specified immediately after the `***contact` command, instead of here. There is no default behavior. Parameters of the contact model are given with the `*behavior_coef` command.

`*impactor` (or alternatively `*slave`) needs the nset *impactor_zone*. The associated bset (liset or faset) has to exist as well. The bset has to be oriented with normals pointing away from the interior of the associated body.

`*target` (or alternatively `*master`) takes the bset *target_zone*. Again, this bset *must* be oriented properly.

`*behavior_coef` sets the parameters of the specific contact model. Details can be found in the following pages.

`*dont_check_bcs` specifies that the code will *not* check whether contact conditions are compatible with other boundary conditions that may exist at the same node, as it does by default. With the default behavior, the code will try to fix any incompatibilities, or if that is not possible, the contact condition will be ignored. A warning message will be given if this is the case. This option disables the verification for the current zone.

`*variable_friction` (or `*file`) specifies a variable friction coefficient μ, depending on the current and/or cumulated value of tangential slip. This variable friction cannot (yet) depend on any external parameter, such as temperature. A precise description of this behavior can be found in the material manual under `***behavior variable_friction`. Not implemented for the `ortho_coulomb` contact model.

`*warning_distance` specifies the maximum distance *warning_distance* for which the algorithm will try to detect if contact occurs between target and impactor of the current zone. In other words, if the distance between target and impactor is *larger* than this

warning distance, the algorithm will *not* try to detect contact. This distance should be chosen larger than the decrease in distance between target and impactor during the increment of interest, otherwise there is a risk of penetration without the contact algorithm coming into action.

The subcommands `*friction` and `*gap` are now deprecated, as they are replaced with their equivalents in the `*behavior_coef` subcommand. Z-set will issue a warning, and in the case of `*friction` automatically convert to the new format. The `*gap` command will be entirely ignored.

**zone normal

Description:

This is the normal contact model as described a few pages back. For the meaning of its coefficient the reader is referred to that description.

Syntax:
The normal behavior zones take all the standard **zone commands, with the following specific behavior coefficient:

```
**zone normal
[  other zone subcommands ]
 *behavior_coef
  friction mu
```

The default value is $mu = 0.$

**zone penalty

Description:

This is the penalty contact model as described a few pages back. For the meaning of its coefficients the reader is referred to that description.

Syntax:
The penalty behavior zones take all the standard `**zone` commands, with the following specific behavior coefficients:

```
 **zone penalty
[  other zone subcommands ]
  *behavior_coef
   friction mu
[  penalty_normal gamma_n ]
[  penalty_slip gamma_t ]
[  damp_normal beta ]
```

Default values are *mu* = `0.`, *gamma_n* = `1.e12`, *gamma_t* = `1.e2` and *beta* = `1.`

**zone coulomb

Description:

This is the "standard" contact model for contact with Coulomb friction, as described a few pages back. For the meaning of its coefficients the reader is referred to that description. Note that this contact model effectively contains the normal and penalty contact models.

Syntax:
The Coulomb behavior zones take all the standard `**zone` commands, with the following specific behavior coefficients:

```
 **zone coulomb
[  other zone subcommands ]
  *behavior_coef
   friction mu
[  max_shear tau_max ]
[  penalty_normal gamma_n ]
[  penalty_slip gamma_t ]
[  damp_normal beta ]
```

Default values are *mu* = `0.`, *tau_max* = `1.e12`, *gamma_n* = `1.e12`, *gamma_t* = `1.e2` and *beta* = `1.`

**zone ortho_coulomb

Description:

This is the orthotropic Coulomb contact model. It is similar to the isotropic Coulomb model, except that some parameters now have two be supplied with two values instead of one, and that the initial material orientation (vector) with respect to the master or slave surface now has to be specified.

Syntax:

The orthotropic Coulomb behavior zones take all the standard `**zone` commands, with the following behavior coefficient key-value pairs:

```
 **zone ortho_coulomb
[  other zone subcommands ]
  *behavior_coef
   friction mu_1 mu_2
[  max_shear taumax_1 taumax_2  ]
[  penalty_normal gamma_n ]
[  penalty_slip gamma_s1 gamma_s2  ]
[  damp_normal beta ]
[  direction nx ny nz ]
[  master | slave ]
```

Default values are *mu_1* = *mu_2* = `0.`, *taumax_1* = *taumax_2* = `1.e12`, *gamma_n* = `1.e12`, *gamma_s1* = *gamma_s2* = `1.e2`, *beta* = `1.`, (*nx ny nz*) = (`1.  0.  0.`), and the last keyword has default value `slave`. Note that this behavior is currently not compatible with the `*variable_friction` zone subcommand.

***continuum_contact

Description:

***continuum_contact is an alternative to ***contact, and is based on a penalty method to enforce the contact constraints. Compared to ***contact that solves a local problem restricted to nodes/degrees of freedom entering contact, in this formulation, contact elements are dynamically added to the global FE equilibrium system, depending on the contact state at the current Newton iteration.

An important implication of this is that since all non-linearities are treated at the same time (i.e. contact, but also material or geometric non-linearities), a larger number of equilibrium iterations are generally needed by this method compared to the previous one. This may become a major cause of divergence in case of so-called *contact fluttering* (new contact elements entering or leaving the non-linear system at each equilibrium iteration). A special procedure has been devised to alleviate this particular problem, and can be triggered by the **stabilize_behavior command.

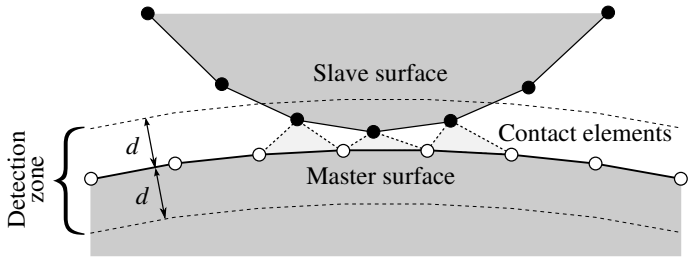
On the other hand, large CPU times (mainly involved in the calculation of the flexibility matrix, see page 3.97) may be needed to build the local contact problem with the ***contact algorithm. As a result, the present formulation is in general more efficient, as soon as the number of nodes entering contact becomes large. Also, another major advantage of ***continuum_contact, is that it's the only contact formulation that supports Z-set's domain-decomposition parallel solvers, thus extending the field of HPC to large contact problems.

Contact elements and definition of the warning zone:

Definition of contacting parts in the FE mesh is similar to the one used in the ***contact. An arbitrary number of contact zones can be added, each zone being characterized by:

- a slave (impactor) nset,
- a master (target) bset,
- a *warning distance*, defining a critical value of the distance to the master surface below which contact constraints should be added for nodes belonging to the slave nset.

As shown in the next figure, when the altitude of a given slave node is below the warning distance, a contact element is automatically created that links this slave node to nodes of the closest bset element in the master surface definition.



The contact penalty formulation:

In the penalty method, contact constraints are only enforced approximately and some degree of penetration is allowed, depending on the value used for the p_n penalty parameter.

Large values decrease the amount of penetration allowable, but can lead to convergence problems, such that choosing an optimal value for p_n is a tradeoff between precision/efficiency requirements. An automatic choice of this parameter, based on the stiffness of the underlying master surface, can be activated by means of the `*automatic_penalty` command.

Current limitations and use with quadratic elements:

- in the current release (8.6), `***continuum_contact` is limited to frictionless contact only.
- specific issues related to the handling of contact at middle nodes of quadratic elements are not treated explicitly in the formulation. The only solution currently available for quadratic meshes is to use the `*add_mpc` option, that automatically adds relationships at middle nodes, to tie them linearly to corner nodes of the corresponding edge of second-order elements.
- `***continuum_contact` is not yet supported for dynamic problems.

Stabilization of the material behavior:

Command `*stabilize_behavior` greatly improves convergence in the general case where both material and contact non-linearities are included. When defined, convergence of the equilibrium problem for each increment, is split-up in 2 successive phases:

- severe non-linear iterations: during this initial phase, material non-linearities are deactivated (i.e. the material behavior is linearized around the state obtained at the end of the last converged increment), and only contact non-linearities are active.
- standard equilibrium iterations: once a *preliminary stabilized* contact state has been obtained at the end of the previous phase, material non-linearities are released, and conventional equilibrium iterations proceed until convergence. Note that the contact non-linearities are included, such that the contact state may indeed change during this phase. However, smaller contact-state variations are expected compared to the case where the first phase is not done.

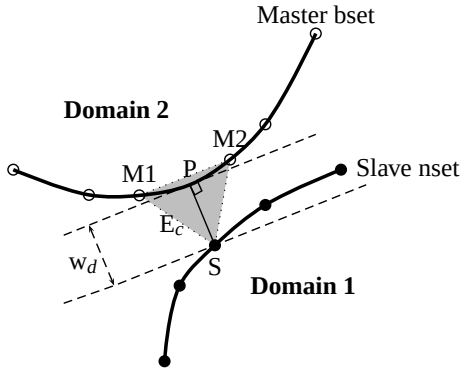
Experience has shown that convergence problems associated with large contact-state fluctuations interacting in an harmful way with a strongly non-linear material response in the contact zone, are alleviated, and even completely removed, by the use of this `*stabilize_behavior` procedure.

Use with parallel domain-decomposition solvers:

This contact formulation is now fully supported by the `generic_dd` (or `feti`) parallel solvers, meaning that the occurrence of contact between sub-domains is automatically accounted for. In this case, new elements are added to the sub-domain containing the master surface, while the slave nodes of the impactor sub-domains are automatically added to the list of nodes of the master domain, and to the corresponding interfaces.

For example, in the case of the figure hereafter, where contact is defined between slave nodes of Domain 1 and a master surface included in Domain 2, steps involved in the parallel implementation are the following one:

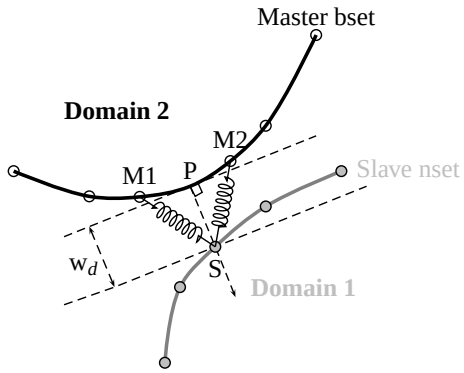
- Parallel contact detection: slave node S of Domain 1 is the warning zone of the master surface of Domain 2 (distance $PS < w_d$, where P is the orthogonal projection of S on the master surface, and w_d is the warning distance).
- Update of the master subdomain definition: Slave node S is added to list of nodes of Domain 2, while a new contact element (S, M_1, M_2) is also automatically created in this domain.
- Management of the interfaces between domains: Node S is now shared by both domains, and corresponding degrees of freedoms are added to the interface definition.



Some specific issues related to the use of domain-decomposition solvers then need some further description:

- as shown in the figure above, if a node (S) of the slave nset is within the warning zone of the master bset defined in another domain, new nodes/elements are added to the master domain (**Domain 1** in the figure). In the event that no contact at node S is detected during equilibrium iterations (positive altitude, no penetration of the master surface), the stiffness associated to this contact element is zero, such that S is not connected to other elements in **Domain 2**, giving rise to null pivots during tangent matrix factorization. To avoid this problem, as shown in the figure below, a small stiffness is automatically added, the value of which is defined by the `Parallel.Contact.Stabilize` global parameter (default value is 1.e-3). The stiffness of those *stabilization springs* can be redefined by using the `***global_parameter` command, or directly at the command line level using the `-s` switch:

```
$ Zrun -s Parallel.Contact.Stabilize 1.e-1 ...
```



- the `generic_dd` solver supports many options (choice of primal/dual treatment for user-specified interface degrees of freedom), and preconditioning methods for the iterative resolution of the interface problem. Particular options may or may not be well adapted to the handling of those contact problems, in particular because of a poor conditioning of the global stiffness due to the penalty method. Experience has shown that a full dual treatment of the interface problem associated to the use of a Dirichlet preconditioner is the most robust combination. Typical input commands of `generic_dd` are included hereafter for reference:

```
***linear_solver generic_dd
 **local_solver mumps *no_option
 **dof_kind
  *dof all dual
 **iterative_solver nncg
   precision 1.e-06
   max_iteration 1000
 **precond full
 **scaling stiffness
 **projector_schur none
 **projector_scaling stiffness
```

Syntax:

```
***continuum_contact
  **zone penalty
   *master mname
   *slave sname
   *warning_distance  dist
[  *penalty_normal  pn  ]
[  *automatic_penalty ]
[  *auto_factor  fact  ]
[  *keep_master ]
[  *add_mpc [ middle | corner ] [ full | everywhere ] ]
[  *stabilize [ everything ]  stab  ]
[  *damp ]
[  *remove  ctip  ]
[  **zone penalty
     ...                ]
[  **stabilize_behavior ]
```

`**zone` this command starts the definition of a contact zone. Note that as many zones as needed may be added to describe contact between various sub-components in the global mesh.

`*master` defines the master target bset. *mname* is a `liset` for 2D problems, or a `faset` in 3D. Care must be taken that the normal to this bset is pointing outwards the interior of the associated body.

*slave defines the name *sname* of an nset containing impactor slave nodes for this particular zone.

*warning_distance defines the critical distance between a slave node and the closest point on the master surface, below which a contact element is created for this particular slave. Note that if the slave node is not found in contact during equilibrium iterations (positive altitude, no penetration), null stiffness/nodal forces will be calculated for the corresponding contact element, such that setting a too large value for *dist* does not change the results. However, the size of the global stiffness matrix is increased by those useless elements, and for efficiency reasons *dist* should not be taken too big.

*penalty_normal this optional command allows to define the value of the penalty parameter *pn* used to enforce non-penetration of slave nodes inside the master surface. As discussed above, setting manually an optimal value to *pn* may be cumbersome, and using the *automatic_penalty command is a practical alternative, especially when the slave surface is not homogeneous, such that using the exact same value for all the contact zone is not suitable.

*automatic_penalty with this option a value of the penalty coefficient is automatically computed for each slave node, based on the stiffness of the underlying master surface at this particular node.

*auto_factor this optional command allows to modify the heuristic used in the previous *automatic_penalty method, where *pn* is defined by the following equation:

$$pn \;=\; fact \;.\; stiffness\_bset$$

with $stiffness\_bset$ is the average of the stiffness at nodes of the closest bset element. Experience has shown that the default value of $fact = 0.1$ is a good choice, such that using of *auto_factor to redefine *fact* is scarcely needed.

*keep_master in the default mode, all bset elements within the warning zone of a given slave node are selected as contacting candidates at the beginning of the increment. Then, during equilibrium iterations, a contact element is created only with the closest one in this list of candidate, a choice that may change from one iteration to the other. If command *keep_master is defined, the closest element at the beginning of the increment is retained during all iterations. This command may be used to speed-up convergence in case of small-sliding.

*add_mpc this command should be used for contact between quadratic FE meshes. In this case, contact conditions are only enforced for a subset of the slave nodes of quadratic elements, i.e. *corner nodes* when corner is defined (default) or alternatively *middle nodes* when middle is used. In all cases, linear relationships are automatically added to middle nodes in order that they stay aligned with corner nodes of the corresponding element edge. Optional subcommand all adds those MPCs at all the middle nodes of the slave nset regardless of the contact state. If everywhere is used instead of all, this procedure is extended to all middle nodes of the master surface.

*remove this command typically takes as argument the name *ctip* of an nset containing nodes at the crack tip, and may for example be used in **Zcracks** simulations when

contact between the crack lips is defined. In this case one side of the crack is defined as *master* and the other one as *slave*, while the crack tip is indeed common to both sides, such that enforcing contact conditions for those nodes has no sense an could inhibit convergence.

`*stabilize` this command is intended to remove rigid-body movements associated with the definition of a contacting component whose motion is not completely prescribed by boundary conditions, and is only restrained by contact with other components. In this case springs with a stiffness corresponding to the *stab* argument are added:

- at the slave node for each contact element (*grounding* of the slave node)
- between the slave and the master nodes of each contact element.

When the option `everything` is added, not only the slave node, but also the master nodes are *grounded*. The stiffness *damp* should be chosen significantly smaller than the stiffness of the contacting bodies, but large enough to remove rigid motions. Since adding those *stabilization springs*, only change the global stiffness matrix, setting a too large value for *damp* can degrade convergence.

`*damp` when this option is used in conjunction with `*stabilize`, corresponding internal forces are also added to the problem residual. This strategy improves convergence, but the quality of the solution is now altered for too large *stab* values.

`*stabilize_behavior` as described previously this command may be used to improve convergence when both material and contact non-linearities are included at the same time.

Example:

```
***continuum_contact
 **zone penalty
  *slave imp
  *master tar
  *warning_distance 1.0
  *automatic_penalty
```

***dimension

Description:
Because of numerical noise, Z-set sometimes has to make decisions about when a quantity is very small, small, large, or huge. For instance, when a typical time increment during a calculation is of the order of 10 s, output is not written to a file at $t = 1000$ s (if so requested through the `***output` command) when the increment ends at $t = 999.9999999$ s.

In order to remedy this kind of problem, Z-set has predefined typical values that occur very often, and defines `small` as "multiplier $\times$ typical value". The example above will pass with the default values: the default value for `time` is 1.0 and the default multiplier associated with `small` is 10^{-6} (so `small` $= 1.0 \times 10^{-6} = 10^{-6}$), and since

$$1000 - \texttt{small} = 999.9999990 < 999.9999999 < 1000.000001 = 1000 + \texttt{small},$$

output will be written. However, for certain other cases these default values need to be modified, for instance for impact problems where time increments may come down to the order of 10^{-7} s or less. This can be done through the `***dimension` command.

The `**dimension` command lets the user modify the typical values for `stress`, `deformation`, `displacement`, `time` and `undimensional`. The multipliers associated to `tiny`, `small`, `large` and `huge` may also be changed. Default values are listed below.

Syntax:
The syntax is as follows:

```
***dimension
[ **unit unit  typical ]
[ **size size  multiplier ]
```

unit may be `stress`, `deformation`, `displacement`, `time` or `undimensional`.

typical specifies the typical order of magnitude (a positive `double` value) that will occur for the quantity *unit*. The default values are 100. for `stress`, 10^{-6} for `deformation`, 0.1 for `displacement`, 1. for `time` and 1. for `undimensional`. Note: this does *not* specify the actual units (despite the name of the `**unit` command that suggests otherwise). For example, giving `**unit time 1.0e-6` does *not* mean that all times are measured in microseconds.

size may be `tiny`, `small`, `large` or `huge`.

multiplier gives the multiplier (a positive `double`) associated with the *size* keyword. Default values are 10^{-12} for `tiny`, 10^{-6} for `small`, 10^{6} for `large` and 10^{12} for `huge`. Their respective values should satisfy `tiny` $<$ `small` $< 1.0 <$ `large` $<$ `huge`. Note that the multiplier given here affects *all* units.

Example:
The following example may be useful for impact problems, where very small time increments often occur. In absence of these commands, output will not be generated at the proper instants for time increments smaller than 10^{-6} s.

```
***dimension
 **unit time 1.0e-6
```

***eigen

Description:
The procedure `***eigen` allows to specify the modes and resonant frequencies to extract in eigen-value problems.

The eigen frequencies are normalized in the following manner:

$$Max(over\ the\ nodes)(\sqrt{U_1^2\ +\ U_2^2\ +\ U_3^2}) = 1$$

And the associated energy for a frequency is calculated as:

$$E = \frac{1}{2} U.K.U \qquad \text{with } U \text{ normalized as given above}$$

The frequency values are stored with their associated energies in an ASCII text file named *problem*`.eigen_info`.

There are currently multiple methods implemented to extract eigen values: `spectra`, `lanczos` and `inverse_vector_iteration` (this is the default method). `spectra` is the most robust and the only method that allows to compute free modes (no kinematic constraints on the structure), or semi-free modes (some rigid body modes are still possible). If needed, it is also the only one that can compute negative eigen values.

The syntax is presented for each method separately below (the spectra solver is further detailed in the devel manual).

Syntax:

```
***eigen spectra
[ *type type ]
[ *nb_modes nb_modes ]
[ *nb_sub nb_sub ]
[ *tol tol ]
[ *output_eigen_not_freq ]
[ *clean_rbm ]
[ *output_rbm ]
[**use_lumped_mass ]
```

`*type` is the type of eigen problem to solve. It can be either `basic` or `generalized`. The latter is the default option and solve the generalized eigen problem $K.U = \lambda M.U$ (smallest eigenvalues). `basic` can be used to compute solutions of $K.U = \lambda U$.

`*nb_modes` is the number of eigen values (and eigen modes) to compute. Default is 10.

`*nb_sub` is the size of the Krylov subspace for the eigen value extraction algorithm. Default is 4`nb_modes` (which is relatively high for accuracy, 2`nb_modes`+1 may be enough if `nb_modes` is sufficiently large for the system).

`*tol` is the precision parameter for the calculated eigenvalues.

`*output_eigen_not_freq` indicates that output must be given in terms of eigen values λ_i, not in terms of eigen frequencies f_i, with $f_i = \frac{\sqrt{\lambda_i}}{2\pi}$.

*clean_rbm indicates that rigid body modes are to be computed (based on geometry only) and removed for the evaluation of the Krylov subspace. This option cannot be used if the mass matrix is lumped, and is generally unecessary when tpye is basic.

*output_rbm indicates that if rigid body modes are computed, they must be output as eigen modes with vanishing eigen value and energy. Only works with the previous options.

**use_lumped_mass is not specific to spectra, but must necessarily come after the spectra options (any other two stars options for ***eigen as well).

Example:
(Full tests are available in External/Spectra/Test/ and are available on request for any user).
Compute the 10 smallest eigen values (outputs eigen frequencies) of the generalized eigen value problem.

```
***eigen spectra
```

Compute the 25 smallest eigen values of the basic problem (subspace size is 100).

```
***eigen spectra
  *type basic
  *nb_modes 25
  *output_eigen_not_freq
```

Compute the 10 smallest eigen values (outputs eigen frequencies) of the generalized eigen value problem using a lumped mass matrix.

```
***eigen spectra
  *type generalized
 **use_lumped_mass
```

Compute the 40 smallest eigen values (outputs eigen frequencies) of the generalized eigen value problem using a lumped mass matrix with a subspace of size 100. Also computes rigid body modes and removes them from the analysis but outputs them in the results files.

```
***eigen spectra
  *type generalized
  *nb_modes 40
  *nb_sub   100
  *clean_rbm
  *output_rbm
```

Syntax:

***eigen *lanczos*
nb_freq freq_max nb_iter nb_sub

nb_freq is the number of eigen frequencies to find.

freq_max is the maximum value of the frequencies to search for.

nb_iter number of iteration for the eigen value search by QR decomposition. Default is 8.

nb_sub size of the Krylov subspace. Default is 2nb_freq if nb_freq< 8, else it is 2nb_freq+8.

Example:
For this example the calculation will search for 8 frequencies and then stop. The calculation will stop also if the last frequency extracted is greater than 10000 (*freq_max*).

```
***eigen lanczos
   8 10000.
```

Syntax:

***eigen *inverse_vector_iteration*
nb_freq d_freq freq_max

nb_freq is the number of eigen frequencies to find.

d_freq is the search interval for the resonant frequencies. If several frequencies are located in an interval smaller than *d_freq* the calculation may fail to find all the frequencies. lanczos method does not need this parameter.

freq_max is the maximum value of the frequencies to search for.

Example:
For this example the calculation will search for 8 frequencies and then stop. The calculation will stop also if the last frequency extracted is greater than 10000 (*freq_max*).

```
***eigen
   8 100. 10000.
```

The extraction of the resonant modes is made with the method of inverse powers and with a shifting of frequencies. After having obtained a frequency f, the next frequency is searched about $f + \Delta f$ where Δf is given by the parameter *d_freq*. The choice of this frequency is therefore very important.

- If *d_freq* is small the calculation may fall back on a frequency already calculated. In this case the search will be shifted to $f + 2\Delta f$ and so on until a new frequency is found. This succession of searches will inevitably add to the cost of the calculation.
- If *d_freq* is large the search process risks to omit intermediate resonant frequencies.
- Lastly if it is desired to have the first resonant frequency it will be necessary to give a small value for the parameter *d_freq*. If the value is small enough the calculation will be sure to not skip the first frequency. Searches for subsequent frequencies will then be very slow however.

***elastic_energy

This command is now obsolete, please have a look at ***energy_monitoring.

Description:
This option indicates that a calculation of the global elastic strain energy should be made and output. This calculation gives the following energy calculation for the structure:

$$\frac{1}{2}\int_V \tilde{\sigma} : D^{-1} : \tilde{\sigma} \quad dV$$

where D is the elastic matrix extracted from the the material behavior.

The result of this the elastic energy is stored in a formatted ASCII file named *problem*.elastic_energy. The output is sequential, with one line per increment of the calculation.

Syntax:

```
***elastic_energy elset-name
[ **frequency frequency ]
[ **precision precision ]
```

elset-name is a character name for the element set over which the strain energy is calculated. The predefined element set ALL_ELEMENT may be used to declare the entire structure.

frequency may be used to specify when the energy is computed (by default, it is made at each increment). This option has the same syntax as output frequency (see p.3.174).

precision is an integer specifying how many significant digits are used in the output file (default is 6).

Example:
From $Z7TEST/Energy_test/INP/elastic_energy1.inp

```
***elastic_energy ALL_ELEMENT
```

***energy_monitoring

Description:
This command can be used to monitor the evolution over time of the energies involved in the model (work of external forces, kinetic energy, energy dissipated by damage or plasticity, etc). It is also possible to monitor the evolution of linear momentum and angular momentum. An ASCII file is written at the end of the simulation, each row corresponds to an increment. This command processes in two ways depending on the considered energy :

- the kinetic energy (W_{kin}), the internal energy (W_{int}) and the work of external forces (W_{ext}) are computed from the global nodal vectors. W_{int} and W_{ext} are integrated through a trapezoidal rule:

$$\begin{aligned} W_{int}^{n+1} &= W_{int}^{n} + \frac{1}{2}\Delta U^T \left(F_{int}^{n} + F_{int}^{n+1}\right) \\ W_{ext}^{n+1} &= W_{ext}^{n} + \frac{1}{2}\Delta U^T \left(F_{ext}^{n} + F_{ext}^{n+1}\right) \end{aligned}$$

  where ΔU is the variation of displacement over the current increment (*i.e.* $U^{n+1} - U^n$). The kinetic energy is computed by:

$$W_{kin}^{n+1} = \frac{1}{2} V^T M V$$

  where M is the mass matrix and V is the nodal velocity vector.

- all other energies are computed by integrating dedicated internal variables over the time increment and over the whole domain. These variables have to be defined and updated within the material model, there are expected to be increments of energy densities (*i.e.* not energy rate neither cumulated energies).

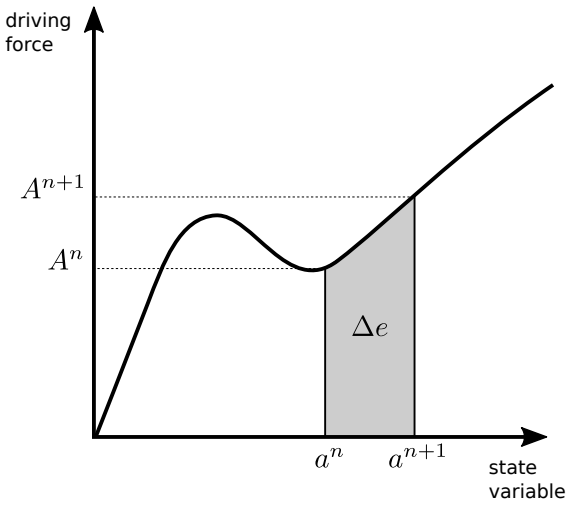
  For instance, if one is interested in monitoring the dissipation due to a mechanism modelized with the state variable a associated to the driving force A, classical thermodynamics of material states that (if a and A are tensors):

$$\dot{e} = A : \dot{a}$$

  This requires the energy increment to be computed through a trapezoidal rule within the material model:

$$\Delta e = \frac{1}{2}\Delta a (A^{n+1} + A^n)$$

  where Δe is the increment of energy density that has to be defined by the user within the material model as an internal variable (*VAR_AUX*).



The purpose of ***energy_monitoring is to perform the integration over the domain (following the integration rule associated to each element) :

$$\Delta E = \int_\Omega \Delta e \, d\Omega \tag{26}$$

$$E^{n+1} = E^n + \Delta E \tag{27}$$

Note that if $A \equiv \sigma$ and $a \equiv \varepsilon$, the resulting energy should be equal to W_{int}.

Also note that if $A \equiv \sigma$ and $a \equiv \varepsilon_{el}$ (elastic strain), the resulting energy should be equal to the elastic energy.

It is also possible to compute linear and angular momenta. This is can be a useful guideline to check the validity of simulations, especially in the case of finite rotations in dynamics (it is well know that the Newmark integration scheme does not properly conserve angular momentum).

- the linear momentum is defined as:

$$\vec{p} = \int_\Omega \rho \vec{v} \, d\Omega \tag{28}$$

where $\vec{v}$ is the velocity field and ρ is the local density. It is computed following the integration rule of the elements.

- the angular momentum is defined as:

$$\vec{L_O} = \int_\Omega \rho \, O\vec{M} \times \vec{v} \, d\Omega \tag{29}$$

where O is an arbitrary fixed point and $O\vec{M}$ is the position vector. It is also computed following the integration rule of the elements.

Syntax:

```
***energy_monitoring
[ **file_name file_name ]
 **energy
  *energy_density energy_density_name
  *name  name
[  *elset elset_name ]
[ **momentum
  *center  (momentum_point) ]
```

file_name is the name of the output file. It is optional, if not given the output file is named *problem_name-energy_monitoring.dat*.

energy_density_name `energy_density_name` is the name of the internal variable (`VAR_AUX`) defined in the internal model as the energy density increment.

name is the name of the current energy as it will appear in the header of the output file.

elset_name is the name of the elset where to apply the integration. Default is `ALL_ELEMENT`.

momentum_point is a vector defining the position where the angular momentum is computed

One can use as many `**energy` blocks as desired. Note that time, W_{int}, W_{ext} and W_{kin} are always computed as the first four columns of the output file.

Example:

```
***energy_monitoring
 **file_name toto.energy

 % energy dissipated through damage
 **energy
  *name E_elastic
  *elset ALL_ELEMENT
  *energy_density damage_energy

 **momentum
  *center (0. 0. 0.)
```

***equation

Description:
This option is used to add linear relationships (equations) between different degrees of freedom in the problem. This allows the standard set of linear equations to include constraint conditions (MPC = multi-point constraint). In the current version of the code, these supplemental relationships are resolved by the direct elimination of degrees of freedom.

Syntax:

```
***equation
 **MPC_type
    ...
```

The relationship types which are currently defined are summarized in the following table:

CODE	DESCRIPTION
**free	impose an arbitrary relation between a single DOF and any number of others (p. 3.127)
**mpc1	sets nodal DOFs of a given type to be equal within a node set (p. 3.128)
**mpc2	sets two groups of nodes on a paired basis. A linear relation is enforced between the ith DOF in one node set and the ith node on the other (p. 3.129)
**mpc2x	is similar to mpc2, and provides an extended syntax (p. 3.130)
**mpc2_dof_elset	Sets a condition similar to **mpc2 between two DOFs belonging to ELSETs (p. 3.131)
**mpc3	models sectorial symmetry of the 3-axis (p. 3.132)
**mpc4	sets the DOFs of an node set equal to a surface function (line set in 2D or face set in 3D); this condition allows incompatible meshes to be attached at the surface (p. 3.133)
**mpc2d3d	combines mixed dimension meshes (p. 3.135)
**mpc_periodic	periodic boundary conditions (p. 3.136)
**nul_div_u	this procedure imposes a constant volume in the interior of a line set in 2D or a face set in 3D (p. 3.137)
**mpc_rb	imposes a rigid body motion on a nset (p. 3.139)

**free

Description:
This MPC imposes any relation between a single DOF and other DOFs. There are multiple ways to define the DOFs.

The equation applied is:

$$s = c_1 m_1 + \ldots c_n m_n + C$$

Syntax:
This command has a slightly free form, non-standard format.

```
**free
 slave is
   master1  coef1
     ...
   masterN  coefN
   coefficient
```

The terms above *slave* and *master#* are some special definitions of DOFs specific to this MPC condition. The syntax is given below. The *coef#* values are real number for the c_i terms above. And *coefficient* is a real value for the coefficient C. Note that the `is` above is a necessary keyword.

`node:`*id*`:XX` nodal dof, *id* is the node id, `XX` the dof type.

```
node:120:U2.
```

`element:`*id*`:`*rk*`:XX` element dof, *id* is the element id, *rk* the rank of the dof in the element (counted from 1 to the number of element dof of type `XX`), `XX` the dof type.

```
element:25:3:EZ.
```

`elset:`*name*`:XX` elset dof, *name* is the elset name, `XX` the dof type.

```
elset:metal:E33
```

**mpc1

Description:
This equation type imposes that a given degree of freedom at all the nodes of a valid node set are equal in value.

Syntax:

```
**mpc1 nset_name dof_name
```

nset_name Character name for a valid node set.

dof_name Character name for a DOF keyword. The keywords available for different problem types are described in the chapter *DOF*.

Example:
The following example could be used to assure that the top surface of a flat topped structure remains flat during deformation.

```
%  from cisap.inp
***equation
  **mpc1 top U2
```

**mpc2

Description:
This equation type indicates that groups of nodes are tied on a paired basis. The i^{th} DOF in one node set will be equal to the i^{th} node on the other factored by a coefficient, plus a possible translation.

Alternatively, one can specify element sets to link DOFs belonging to elements. Note that node set names are search first, which could lead to some problems if element sets and node sets have similar names.

Syntax:
There are two possible syntaxes. A standard one and an extended one. The standard syntax is used to simply apply a factor between two degrees of freedom; the default value of *ratio* is equal to 1, and the relationship is such as *dof2_in_name2* = *ratio* . *dof1_in_name1*

```
**mpc2 name1 dof1 name2 dof2 [ratio]
```

The extended one is used to introduce the affine relationship
dof2_in_name2 = *ratio* . *dof1_in_name1* + *translation*:

```
**mpc2   name1 dof1 name2 dof2
  *ratio  ratio
  *translation  translation
  *inversion
```

The `*inversion` keyword may be used to prescribe an inversion of the nset order.

Example:
This could be used to model in 2D a sectorial symmetry following an angle of 45 degrees. The nodes of the bisection line of the 1 and 2 axis should then be imposed to have the same displacement following the axis 1 and 2.

This condition also allows modeling of axisymmetric conditions (symmetry in relation to a point) by taking two symmetrical node sets in relation to a point and tying similar typed DOFs of the same type with a coefficient of -1.

```
% from aube.inp
***equation
 **mpc2 n_fuite U1 n_fuite U2  0.4431622
```

On the other hand, the following syntax allows the user to impose a shift between two lips of a crack.

```
% from tsqu1.inp
***equation
 **mpc2  lipleft  U1  lipright U1  *ratio 1. *translation 0.15
 **mpc2  lipleft  U2  lipright U2  *ratio 1. *translation -0.1
```

**mpc2x

Description:
This equation type indicates that groups of nodes are tied on a paired basis. The i^{th} DOF in one node set will be equal to the i^{th} node in the other factored by a coefficient, plus a possible translation:

$$\text{DOF}_Y^B = \alpha \cdot \text{DOF}_X^A + \tau$$

It generalizes the standard mpc2, by allowing both the translation and the ratio to be given by a basic value (i.e. the product of a value and a time-dependent table, see p.3.43 for more details).

Syntax:

```
**mpc2x
  nsetA dofX
  nsetB dofY
[ *translation value table ]
[ *ratio       value table ]
[ *cumulative ]
[ *inversion  ]
```

nsetA nsetB character name of the two paired node sets.

translation value[7] and table specifying the translation τ.

ratio value[7] and table specifying the proportionality coefficient α.

cumulative allows subsequent invocations of this mpc to be cumulated. It is otherwise forbidden, to avoid accidentally applying twice an MPC on the same DOF.

inversion may be used to prescribe an inversion of the nset order.

Example:
The first example (from `$Z7TEST/Hyperelastic_test/INP/ring-3-degrees-mpc.inp`) prescribes $U_\theta = 0$ in a sectorial mesh:

```
**mpc2x     % U3 = z/x * U1 (ensures U_theta = 0)
  face.2 U1
  face.2 U3
 *ratio function z/x ; tab_constant
```

The second example imposes the relative slide of both faces of a crack, where the translation is stored in an file (one value per node in the nset):

```
**mpc2x
  faceA U1
  faceB U1
 *translation ascii_file slide_U1.dat tab_crack
```

[7] As for boundary conditions, the value may either be a real value, a function or a file.

**mpc2_dof_elset

Description:
The syntax is the same as for `mpc2` but the condition is used to link DOFs belonging to elements sets.

Example:
This could be used in the case of periodic elements. Here in the case of large strains periodic elements:

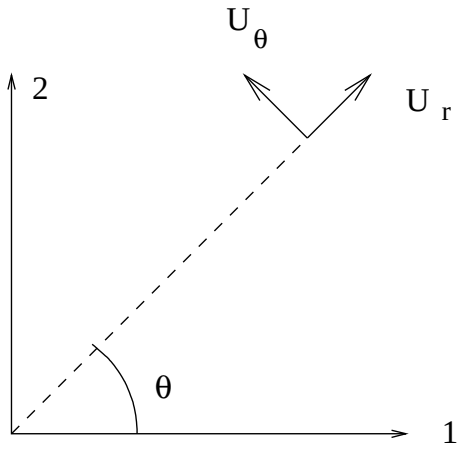
```
**mpc2_dof_elset ALL_ELEMENT E12 ALL_ELEMENT E21
```

**mpc3

Description:
The mpc3 equation type models sectorial symmetry about the 3rd axis.

The following relation is applied to all the nodes of the given node set:

$$U_\theta = 0.0 \Longleftrightarrow U_2 = tg(\theta).U_1$$



Syntax:
For this relationship it is required to give a node set name situated on the symmetry plane, and the angle of the sector in degrees.

```
**mpc3 nset_name theta
```

nset_name Character name for the node set in the plane of symmetry.

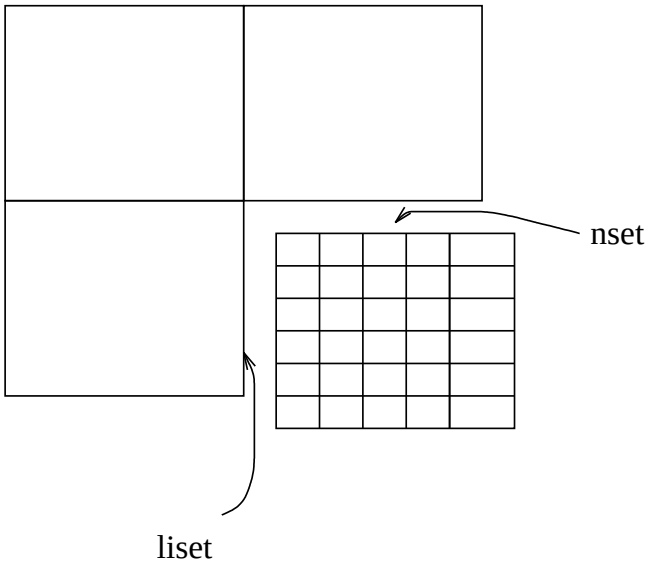
theta Real value for the sectorial angle.

Example:

```
% from zcentrifuge3.inp
***equation
    **mpc1 haut U3
    **mpc3 p0 1.0
```

**mpc4

Description:
The `mpc4` relationship is used to interface incompatible meshes. The DOFs of a given node set will be set to the DOF value on a corresponding boundary. This boundary will generally be taken on the coarser mesh, and is entered as a line set in 2D or a face set in 3D problems. Satisfaction of the `mpc4` condition is made by DOF elimination on the node set.



The position of each node in the node set is first searched in the corresponding line or face set. Two cases may present themselves during this search process. The first is that the node has no projected position on the surface in which case no constraint is applied to that node. The second case is that the node does have a projected position on the surface. In this case the DOF type will be verified for existence on the surface and then subjected to the following relation:

$$U_{\text{nset nodes}} = \sum_{\text{surface nodes}} N_i U_i$$

If the DOF is not found on the surface during the verification no condition will be placed on that DOF.

It may be noted here that the global solution size (front) will be reduced if the numbering scheme is such that the element numbers attached to the node set are lower than those attached to the line or face sets.

... continued

Syntax:
There are two possible syntaxes. A standard one and an extended one. The standard syntax is:

```
**mpc4   nset liset (faset)
```

and the extended one is:

```
**mpc4
  *nset   nset
  *bset   liset (faset)
  *delta   (dx,dy) ((dx,dy,dz))
```

The `*delta` keyword may be used to prescribe a fixed spatial offset between the nset and the liset. This is useful for instance to impose periodic conditions on a unit cell which exhibits incompatible pair sides (e.g. for a 2D square unit cell, left and right sides do not have the same number of nodes).

**mpc2d3d

Description:
This MPC condition is used to "glue" a 3D mesh to a 2D mesh for mixed dimension analysis. The masters are the DOFs of the 2D mesh controlling the 3D slave DOFs. DOFs are attached (equal) if they have the same x_1 and x_2 coordinates.

Syntax:

```
**mpc2d3d
  nset2d  nset3d  dof  dx
```

nset2d the name of a node set on the 2D mesh.

nset3d the name of a node set on the 3D mesh.

dof the DOF type to fix (e.g. `U1`, `U2`, ...).

dx a real value for the distance criteria for node position equivalence.

**mpc_periodic

Description:
This procedure enables one to prescribe an average deformation with periodic boundary conditions, according to

$$u = Ex + v(x)$$

where $v(x)$ is periodic; the components are denoted $E11, E22, E12, E21$ in the 2D case and $E11, E22, E12, E21, E33, E13, E31, E23, E32$ in the 3D case. The actual value of the prescribed average deformation is equal to the product of the basic value i and the value in the table *tab*. Concerning the *nsets* it is important to distinguish faces, edges and corners.

Syntax:

```
**mpc_periodic [inv]
dof nset1 nset2 component1 value1 component2 value2 ... tab
```

**nul_div_u

Description:
This relationship imposes a constant volume inside a surface (line set in 2D or face set in 3D) which is linearized by increment. The linearization cumulates an error of the same order as the incremental deformations. The condition may be strictly respected by demanding error correction, but only when the volume is calculated as follows: In 2D:

$$\int x\,dy \quad \text{or} \quad \int y\,dx$$

In axisymmetric:

$$\int 2\pi r z\,dr$$

In 3D:

$$\iint x\,dydz \quad \text{or} \quad \iint y\,dzdx \quad \text{or} \quad \iint z\,dxdy$$

Notes:
The surface must be closed, or at the least its limits must have fixed displacements.

This procedure may be introduced in the middle of a calculation by using `***restart`. If the keyword `correction` is not present after `***restart` the volume is maintained constant from the instant of restart. In the contrary case the volume is maintained constant, but by taking the reference volume as the volume at time $t = 0$.

Syntax:

`**nul_div_u` *surf_name* [`correction` *dir*]

---

surf_name Character name of the line set in 2D or the face set in 3D.

`correction` Keyword indicating that the corrections are to be activated.

dir Direction indicating the manner which the volume is calculated: In 2D plane geometries:

$$\begin{array}{lll} dir\text{=}\mathbf{d1} & \text{if} & V = \int x\,dy \\ dir\text{=}\mathbf{d2} & \text{if} & V = \int y\,dx \end{array} \tag{30}$$

In 2D axisymmetric geometry:

$$dir\text{=}\mathbf{d2} \quad \text{if} \quad V = \int 2\pi r z\,dr$$

In 3D geometry:

$$\begin{array}{lll} dir\text{=}\mathbf{d1} & \text{if} & V = \iint x\,dydz \\ dir\text{=}\mathbf{d2} & \text{if} & V = \iint y\,dzdx \\ dir\text{=}\mathbf{d3} & \text{if} & V = \iint z\,dxdy \end{array} \tag{31}$$

```
****calcul
 ***equation
  **nul_div_u
```

Example:

```
***equation
 **mpc1 bottom U1
 **mpc2 bottom U1 top U2
 **mpc3 nset1  20.
 **mpc4 nset1  surface1
 **nul_div_u  faset1
```

**mpc_rb

Description:
This command imposes a rigid body motion relationship between all nodes of a nset. Mpc can only define linear relationships between dofs, so that `mpc_rb` is only valid under the small displacement assumption. The relationship is reactualized at every increment, so that `mpc_rb` can be reasonably used with finite transformation if the increments are small enough.

In 3D, 6 "well chosen" master dofs are sufficient to define the rigid body motion (*i.e.* 3 translation components and 3 rotation components). They are automatically selected so that the three rotation parameters can be determined.

Under the small displacement assumption, the displacement $\vec{U}_i$ of every node i (at location O_i) of the nset can be expressed as:

$$\vec{U}_i = \vec{U}_m + O_i\vec{O}_m \wedge \vec{\theta}$$

where $\vec{U}_m$ and $\vec{\theta}$ are functions of the 6 master dofs.

Syntax:

`**mpc_rb`	*nset*

***feti

Description:
This command allows input of parameters used by the FETI iterative parallel solver, and therefore may only be included in parallel mode (execution with switch `-PP` as described in the chapter about *Parallel computing* at page **??**).

As the FETI solver is based upon a Conjugate Gradient (CG) method to solve the interface problem arising from sub-domain decomposition, those parameters are mainly conventional input parameters associated with an iterative CG method.

Syntax:

```
    ***feti
     **projector        proj
     **precision        eps
   [  **max_iteration   iter   ]
   [  **max_standing    max    ]
   [  **precond         type   ]
   [  **keep_direction  dir    ]
   [  **reprojection]
   [  **init_lp]
```

`**projector` *proj*
After subdomain decomposition the local problem defined on each subdomain may be singular. To avoid rigid-body motions the FETI solver uses a *projected* CG algorithm (see page **??** for reference). Projection of the gradient requires solving a linear system of equation whose dimension is the total number of rigid-body motions in the subdomains. The string argument *proj* specifies the method used to solve this system of equation, and can be one of the following:

CODE	DESCRIPTION
`direct`	Direct Gauss solver
`iterative`	Iterative Conjugate Gradient solver

Direct solver is faster, and this option is strongly advised unless small pivots problems arise during resolution in which case the *iterative* option may be more efficient.

`**precision` *eps*, where *eps* is a real value defining the relative precision required for convergence when solving the interface problem with the CG method. For a system of equation:

$$\mathbf{Kq} = \mathbf{F}$$

this relative ratio is defined by:

$$ratio = \frac{||\mathbf{F} - \mathbf{Kq}||}{||\mathbf{F}||}$$

and convergence occurs when:

$$ratio < eps$$

**max_iteration *iter*, where *iter* is the maximum iterations when solving the interface problem (default is 100).

**max_standing *max*, where *max* is an integer specifying the maximum number of CG iterations allowed without any significant decrease of the convergence ratio.

**precond *type* This optional keyword is used to specify the type of pre-conditioning used to accelerate CG convergence. Until now, the only pre-conditioner available is `lumped`. Pre-conditioning is strongly advised and can significantly reduce the number of FETI iterations.

**keep_direction *dir*, where *dir* is the the integer value of the number of orthogonal descent directions retained during the CG iterations. Increasing *dir* leads to faster convergence but is more memory consuming. Default value is *iter*+1.

**reprojection This subcommand can significantly reduce the number of FETI iterations, when used in conjunction with quasi-Newton schemes of tangent matrix update (such as `eeeeee` or `p1p1p1`, see the `**algorithm` command). With this option the descent directions calculated during previous load increments are reused, leading to convergence in just a few iterations when the tangent matrix and the load increment stay constant over several Newton increments.

**init_lp When a new resolution is asked for (new iteration, or new increment), this option allows to reuse the previous interface force field, instead of starting from a zero force field (which is the default option). This option has to be used together with the `**reprojection` option.

Example:

```
***feti
   **projector direct
   **keep_direction 150
   **precision 0.00000001
   **max_iteration 1000
   **precond lumped
```

***file_management

Description:
This command allows the user to adjust Zebulon's use of external temporary files.

Syntax:

```
***file_management
[ **keep_temporary_files ]
[ **max_nb_dof num ]
[ **z7_tmp_dir path ]
```

Example:
The following is an example of user controlled file management.

```
***file_management
 **keep_temporary_files
 **max_nb_dof 1
 **z7_tmp_dir ./
```

On a Linux OS, some graphical monitoring of the in-core memory and temporary disk storage usage is available.

```
Zrun -monitor myprob >& /dev/null &
Zplot_stats STAT.4390
```

***fluid_structure_interface

Description:
The fluid structure interface option is for eigen value extraction problems to model fluid filled cavities and their effect on dynamic response.

Syntax:

```
***fluid_structure_interface
 **interface
  *surface_name surf-name
  *fluid_mass_vol value
  *free_surface nset-name
```

Example:

```
***fluid_structure_interface
 **interface
  *surface_name interf1
  *fluid_mass_vol 1.0e-20
```

***function_declaration

Description:

This command allows input of function declarations ahead of other three stars commands. Complex pre-defined functions can thus be learned, and re-used later in the input file.

Syntax:
The syntax is the following:

```
***function_declaration
   func_name := definition ;
```

Note that ":=" for assignment and semi-colon at the end are **required**. The function's arguments are discovered as they appear in the declaration, and used in that order (see examples below, and page 6.2 for a general description of functions).

Note:

The command `***function_declaration` is also allowed in `****mesher` and `****post-processing`, and such definitions are not shared between `****`-instances[8]. A behavior has access to functions learned in the `****calcul`, but for convenience and readability it can also host its own definitions in a `**functions` section in the `***behavior` material file.

Example:
The following example is from `Thermal_test/INP/fluconv_analytical.inp`:

```
****calcul thermal_steady_state
 ***function_declarations
    T   := x^2 * exp(y) * cos(z) ;
    LT  := 2 * exp(y) * cos(z) ;
    Q1  := 2 * x * exp(y) * cos(z) ;
    Q2  := T(x,y,z) ;
    Q3  := x^2 * exp(y) * sin(z) ;
    Text:= x^2 * exp(y) * (cos(z) + sin(z)) ;
% ...
 ***bc
  **convection_heat_flux z0
    h 1.
    Te function  Text(x,y,z) ; time
  **surface_heat_flux
    x0 function -Q1(x,y,z) ; time
    x1 function  Q1(x,y,z) ; time
% ...
  **volumetric_heat
    ALL_ELEMENT function -LT(y,z) ; time
```

[8] The `-learn` command line switch makes it possible to share function definitions between different computations, see 6.2.

***global_parameter

Description:
This command is used to set-up, directly in the input file, non-default values for Z-set global parameters. Those parameters are used quite extensively in the code, and provide a means to customize various software operations.

Alternatively, global parameters values may be defined:

- by passing values after the `-s` switch at the command line level:

  `$ Zrun -s` *name value* `...`

  where *name* is the name of the global parameter and *value* the value specified by the user for the current run of `Z-set`.

- by adding definitions in the `zsetrc` file, as described in the `"Adjustable parameters"` section of the `"Getting started & Zmaster"` manual (`$ Zman intro`).

Note that all available global parameters and their current values are given as output by the `-H` switch:

```
$ Zrun -H
...
 Known global parameters:
   Base.ClockFormat                        direct
   Base.EnableMsgFile                      1
   Base.MessageSuffix                      .msg
...
```

and that contents of the `zsetrc` file can be managed interactively by the `Zmaster "zsetrc Editor"` menu item.

Example:
Switch to Z8 output format for the current calculation:

```
 ***global_parameter
    Solver.OutputFormat Z8
    Zmaster.OutputFormat Z8
```

***global_bifurcation

Description:

This procedure and its options define the rate boundary value problems for which the Hill's uniqueness criterion (uniqueness in terms of velocity fields) is performed [Hill58]. For elastic material in a finite deformation framework this also informs on the equilibrium's stability.

The standard bifurcation analysis consists in the evaluation of the smallest eigenvalues of the global stiffness matrix after kinematic conditions (like Dirichlet BCs and/or MPCs) are taken into account. It may capture buckling, necking or localization. When the smallest eigenvalue becomes negative, it is known that a bifurcation point has been bypassed.

In a more general way, the uniqueness of various rate boundary value problems can be analyzed thanks to this tool. This is the concept of "weakened stability analysis" and "weakened problem" introduced in [Ph.D. thesis - AL KOTOB 2019]. This approach relies the eigen value analysis of the global stiffness matrix after apply a given set of $\{DBC\}$ (Dirichlet Boundary Conditions) that differ from the incremental problem's $\{DBC\}$. This may be useful to analyze the uniqueness of the response of a single member in a contact problem.

The equation to solve for each rate boundary value problem is given by:

$$\left(\left[K^{\{DBC\}}\right] - \lambda\,[I]\right)\{X\} = \{0\}$$

where $\left[K^{\{DBC\}}\right]$ is the global tangent matrix modified by the set of Dirichlet Boundary Conditions $\{DBC\}$.

The eigenvalues are outputed in the *problem*`.msg` file during the computation and in the *problem*-`CASENAME.eigen` file if the option is specified. The eigen vectors associated to the smallest eigenvalues are saved in the *problem*`.node` file if the option is specified.

To visualize these modes as displacement field use the global parameter Zmaster.deformed_by_Ufields 1 (default value is 0).

Syntax:

```
  ***global_bifurcation
[  **stop_if_unstable [num_case nb_incr] ]
[  **standard [n_mode n_subspace]  ]
[
  **change_bcs [max_number_of_cases]
    *file_for_bcs BC_FILE_NAME
    *case [n_mode n_subspace]
     [read_bcs bc_n1 bc_n2 ...]
     [read_mpc mpc_n1 mpc_n2 ...]
  [ *case [n_mode n_subspace]
     [read_bcs bc_n1 bc_n2 ...]
     [read_mpc mpc_n1 mpc_n2 ...]  ... ]
                                      ]
[ ***output
   **extra global_bifurcation[_2D]  ]
```

`**stop_if_unstable` option indicates that the incremental problem should be interrupted if the smallest eigenvalue of the *num_case* rate boundary value problem is negative

for *nb_incr* consecutive increments. Default is: *num_case*= 1 and *nb_incr*= 1 (which means as soon a one eigenvalue becomes negative for the first rate boundary value problem specified). Note that *num_case* refers to the order in which the rate boundary value prblems were declared. This option can be declared multiple times to set various interruption counters for the different rate boundary value problems.

**standard this command chooses indicates that the rate boundary value problem to analyze is the one specified in the incremental problem. If only ****global_bifurcation* is specified, this option is activated by default. *n_mode* and *n_subspace* are respectively the number of eigenvalues to compute and the number of vectors to build the subspace for the eigenvalue extraction. Default is *n_mode*= 10 and *n_subspace*= 4∗*n_mode*. *n_mode* must be specified if one wants to specify *n_subspace*.

**change_bcs indicates that the rate boundary value problem is to be given by bcs other than the ones used for the incremental problem. If this option is activated, then the ***standard* option must be specified if one needs to run the standard analysis too.

```
****calcul
  ...
 ***bc % bcs specified to solve incremental problem
 ...
 ***equation  % mpc specified to solve incremental problem
 ...
 ...
 ***global_bifurcation
  **stop_if_unstable 1 2
  **stop_if_unstable 3 1
  **standard 2 20
  **change_bcs
   *file_for_bcs bifurcation.bcs
   *case 2 20
   read_bcs 1

   *case 2  20
   read_bcs 2
   read_mpc 2

 ***output
  **extra global_bifurcation
****return

/in file bifurcation.bcs/
***bc
 **impose_nodal_dof
    top U1 0.
    top U2 0.
    top U3 0.
***return
```

```
                                                    ****calcul
                                                     ***global_bifurcation

***bc
 **impose_nodal_dof
    bottom U1 0.
    bottom U2 0.
    bottom U3 0.
***return

***equation
  **mpc1 top U1
***return

***equation
  **mpc1 top U2
***return

***equation
  **mpc1 top U3
***return
```

***impose_kinematic

Description:
This command imposes a kinematic configuration (displacements) for a problem which does not have any kinematic variables (e.g. thermal or diffusion problems). It is commonly used for coupled problems.

Syntax:
The syntax is:

```
***impose_kinematic type
  *file fname
```

Example:

```
***impose_kinematic from_transfer
   *file MechTherm/kine_out
```

***init_dof_value

Description:
This procedure imposes the initial values of specified degrees of liberty of the problem (at $t = 0$). The values may be either uniform either defined by nset or elset or read from an initializing binary file. This file may be for example the output from a previous calculation (see the example).

Syntax:
The syntax used to initialize the problem DOFs is:

```
***init_dof_value
[   dof_name elset elset_name  value  ]
[   dof_name nset nset_name  value  ]
[   dof_name uniform value  ]
[   dof_name file    file_name  position  ]
```

where *dof_name* indicates replacement with the character name of the desired DOFs (see appendix). The keywords `elset` and `nset` indicate the type of set. These keywords require a set's name *elset_name* or *nset_name* followed by a real value The keywords `uniform` and `file` indicate the method upon which the values will be loaded. Specifying `uniform` requires a real value for *value* which is the absolute value of the DOF. File storage set by the `file` keyword requires a character name for the file, *file_name*, and an integer value for the record position, *position*. The DOF values will be taken from this file position in sequence using single precision 4 byte floating point format.

Example:
This example shows how to load a data file generated from a small C++ program. This source is as follows:

```
#include <fstream.h>
main()
{ fstream out("U2.dat",ios::out);
  float x=1.;
  for(int i=0;i<8;i++) out.write(&x,sizeof(float));
  out.close();
}
```

which sets a uniform value of 1 (for example).

The input for this example could be:

```
***init_dof_value
  U1 uniform 2.
  U2 file U2.dat 568
```

which will set all the u_1 nodal displacements to 2, and all the u_2 displacements to 1.

***initialize_with_transfer

Description:

The ***initialize_with_transfer section allows to initialize a computation from the results of a previous one which have the same material variables. This feature is mainly used when one needs to modify the geometry of the problem and continue the computation.

Syntax:
***initialize_with_transfer takes a number of main control commands, and sub-blocks defining the results fields transfer method.

```
***initialize_with_transfer
[**format  result_format ]
 **old_problem  result_file_name
 **map  result_card_num
[**use_deformed_mesh]
[**dont_use_deformed_mesh]
[**put_nodes_back]
[**quiet]
[**initial_time time ]
[**auto_resume]
[**reequilibrium]
  *algo  algorithm
  *ratio  convergence
  *iter  max_iterations
[**skip_nodal_transfer]
[**skip_integ_transfer]
[**nodal_var_transfer]
  [*mapping mapping_method ]
[**integ_var_transfer]
  [*integ_transfer transfer_method ]
```

**format defines the imported results format. If this option is omitted, the default Z7 format is assumed.

**old_problem defines the name of the results database to transfer to current computation.

**map defines the results map to transfer.

**use_deformed_mesh use deformed configuration of the old mesh to locate the nodes/IP of current mesh (assumed TRUE if not specified)

**dont_use_deformed_mesh use initial configuration of the old mesh to locate the nodes/IP of current mesh (use this to disable previous)

**put_nodes_back when a deformed mesh is used (use_deformed_mesh) this option allows to retrieve the initial non deformed configuration on the current mesh by subtracting the transferred displacement

**quiet run silently without extra messages (default is a more verbose mode).

**initial_time set the initial time for the current computation. If this option is omitted, the default initial time is 0.

**auto_resume set the initial time for the current computation to be the last time step available in loaded results

**reequilibrium run a "zero delta time" increment of computation at the end of transfer to retrieve equilibrium. This command has the same syntax as **sequence (see page 3.216)

**skip_nodal_transfer disable the transfer of nodal variables (assumed FALSE if not specified)

**skip_integ_transfer disable the transfer of IP variables (assumed FALSE if not specified)

**nodal_var_transfer choose the method and its parameters for transferring nodal variables (use default nodal transfer options if not specified)

*mapping allow to transfer nodal variables between meshes of different dimensions (no mapping if not specified). The possible mapping methods are summarized:

CODE	DESCRIPTION
2d_3d	the results from 2D computation are mapped to 3D mesh assuming extrusion along z direction
axi_2d	the results from 1D axisymmetric computation are mapped to 2D mesh assuming revolution around y direction
axi_3d	the results from 2D axisymmetric computation are mapped to 3D mesh assuming revolution around y direction

**integ_var_transfer choose the method and its parameters for transferring IP variables (if not specified, IP variables are extrapolated to the nodes then transferred as nodal variables by nodal interpolation, finally they are interpolated back to integration points)

*integ_transfer The possible transfer methods are summarized:

CODE	DESCRIPTION
nearest_gp	for each integration point in the current mesh, locate the nearest one in the loaded initialization mesh and simply copy the value to transfer
nearest_gp_corrected	in this mode, we add a correction step after nearest_gp, where the correction try to enforce the compatibility between transferred DOF and the IP variables. To achieve that, a local integration of the behavior is done using the difference between the transferred gradient and the one recomputed from transferred DOF
moving_least_square	two types of moving least square interpolations are provided: linear and quadratic (which is the default one). To choose an interpolation type add : linear_2D, linear_3D, default_2D, default_3D

Example:

This example can be found in Transfer_test/INP/TransferQuadra.2.inp

```
***initialize_with_transfer
  **old_problem TransferQuadra.1.ut
  **quiet
  **use_deformed_mesh
  **format Z7
  **put_nodes_back
    *to_file xx.geof
  **auto_resume
```

The same example modified to use moving least square

```
***initialize_with_transfer
  **old_problem TransferQuadra.1.ut
  **quiet
  **use_deformed_mesh
  **format Z7
  **put_nodes_back
    *to_file xx.geof
  **auto_resume
  **integ_var_transfer default
   *integ_transfer moving_least_square default_2D
```

***make_restart_file

Description:
Allows "restart" files to be created at certain pre-defined moments in addition to the default backup file (`.rst`).

Syntax:

```
***make_restart_file when
```

CODE	DESCRIPTION
`always`	save at each increment
`end_of_sequence`	save at the end of each sequence
`end_of_cycle`	save at the end of cycles

There is no default for *when*. The restart files will have the name *pb_name*`.rst#` where `#` is the incremental number of the restart file.

***matrix_storage

Description:
This command specifies which global matrix solver to use.

Syntax:
This command is a one-liner with no sub-commands. The command takes a parameter which must be a keyword for one of the available solvers.

```
***matrix_storage type
```

The possible solver keywords are:

CODE	DESCRIPTION
frontal	Standard frontal method where the solution is made by marching across the element "front," adding and eliminating degrees of freedom; This is the default
skyline	Traditional skyline method
sparse_direct	Direct (inversion) of a sparse storage matrix[9]
sparse_iterative	Iterative sparse storage matrix

***material

Description:
This command marks the definition of the materials in a structure to be studied. The behavior of each material is defined in a file with special syntax (see the chapter *Material Behavior*). The purpose of this command is therefore to define the material file names, associate these files to different element sets, and specify other global applications on top of a material model such as rotation of material coordinates or give local integration methods.

Syntax:

```
***material
[ **elset  nom ]
    *file  nom  [ num ]
[   *integration    ]
[   *rotation       ]
[   *var_mat_ini    ]
```

The sub-procedure `**elset` is used when there is more than one material. It is necessary then to give the name of the element set considered (*elset*). In the absence of the `**elset` command, the following options with one asterisk will be applied automatically to the ensemble of elements.

The sub-procedure `*file` is used to specify the file name for a material file in absolute or relative path names. An optional integer parameter is now available after the file name, giving the number of material declaration in a particular file. The second example shows this type of syntax.

Example:
A simple example of the `***material` syntax is:

```
***material
  *file    mat
```

For a structure with multiple materials in different element sets, an example is:

```
***material
 **elset E1  *file E.inp 1
 **elset E2  *file E.inp 2
```

The program will search the first instance of `***behavior` in the input file `E.inp` and assign that material to element set `E1`, while element set `E2` will have the material defined after the second `***behavior` in the input file. Please use caution when playing around with instances like this - **Always Verify Your Materials**.

***integration**

Description:
This option determines the local integration method for a material behavior.

Syntax:

`*integration` *method params*

The allowable methods are summarized in the table below:

CODE	DESCRIPTION
`runge_kutta`	explicit integration with automated time steps based on integration error
`theta_method_a`	implicit generalized midpoint integration; this method normally supplies the best tangent matrix
`theta_auto_a`	automatic time stepping in the implicit θ-method
`theta_method_b`	implicit integration by trapezoidal rule

`runge_kutta` The Runge-Kutta method implements a second order explicit integration with automatic time stepping. Variables are normalized to allow varied variable magnitudes in "stiff" sets of equations. The method takes two real parameters. These are the convergence criteria followed by a minimum value for normalization. Standard RK error calculation for each integrated variable will be normalized by either the increment of the variable or this second parameter, whichever is greater. the resulting error is compared with the first parameter.

The Runge-Kutta integration with the `gen_evp` material behavior provides a tangent matrix in models with a single inelastic deformation. This matrix is however not consistent with the integration scheme, and thus yields less than optimal global convergence. The explicit integration also performs poorly in heavily time-dependent problems such as viscoplasticity. However, some complex models are only implemented with this method.

`theta_method_a` The θ-A method is the standard integration for the majority of material laws requiring integration.

$$x(t+\Delta t) - x(t) = \dot{x}\left(t+\theta\Delta t\right)\Delta t$$

This method requires 3 parameters to describe the convergence. These are first the θ value (real) followed by the residual required for convergence (real) and the maximum number of local iterations in the integration (integer).

The value for θ must be greater than zero and less than one. It is **strongly** advised to use θ values of 1 for time independent (plastic) materials, and 1/2 for time dependent (viscoplastic) problems. Time independent plasticity will normally show strong oscillations about the solution for values of θ less than 1.

Reasonable values of convergence range from 10^{-6} to 10^{-10}. Values which are too large usually lead to poor global convergence. Too small values will not converge due to

numerical round-off (10^{-12} is about the limit). Convergence will rarely take more than 25 iterations, and should not take more than 50. If this is the case, there may be some error in the integration (make a bug report), or the material parameters are excessive (damage laws may provoke this). If the local iterations are greater than 50 it is probably better to reduce the global iterations or use automatic time stepping (global or local).

The default integration is dependent on the material law used. Most behaviors modeling plastic or viscoplastic materials use a default of the θ-method with $theta = 0.5$,eta = 1.e-9 and $max\_iteration = 200$.

Example:

```
% plasticity or large deformation
*integration theta_method_a 1.0 1.e-9 50

% difficult viscoplastic case
*integration theta_method_a 0.5 1.e-6 100

% complex law
*integration runge_kutta 1.e-3 1.e-3
```

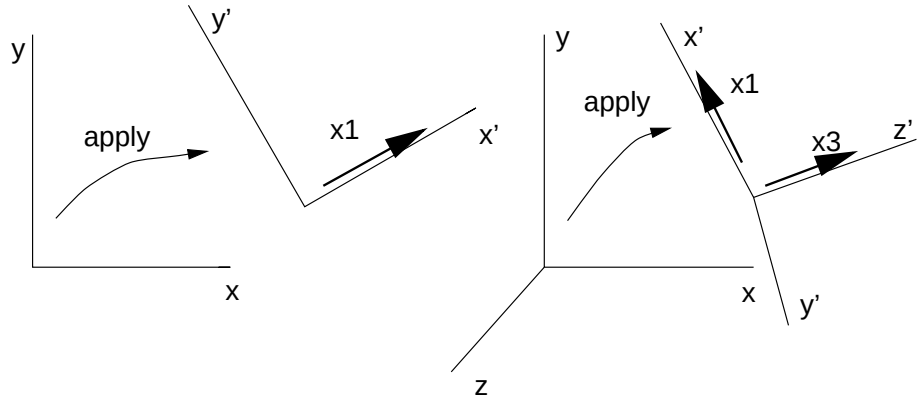
***rotation**

Description:

This material option is used to change a coordinate systems by rotation. It is used here to simplify specification of some materials (anisotropy, etc), but the syntax is general. Other applications using the rotation object include specification of grain orientations for polycrystals (see page 3.2), and mesh rotations (see page 2.120).

There are currently three methods for specifying a rotation. These are by vectors of the rotated coordinate axes in the global coordinate system, by Euler angles (used for crystal orientation for example), or the Rodrigues' representation.

The first case is displayed in the following figure:



For the material rotation of this section, the material gradient will be rotated (rotation is applied) before being integrated by the material behavior. For small deformation mechanics, this would be a rotation of the strain tensor.

$$\boldsymbol{\epsilon}'_{tot} = \mathbf{R}^T \boldsymbol{\epsilon}_{tot} \mathbf{R}$$

The material behavior then solves for the flux in terms of the new gradient, which is $\boldsymbol{\epsilon}'_{tot} \rightarrow \boldsymbol{\sigma}'$ for the mechanical problem. Afterwards the flux is rotated to the global coordinates again:

$$\boldsymbol{\sigma} = \mathbf{R} \boldsymbol{\sigma}' \mathbf{R}^T$$

Note that *only* the flux is rotated back to the global coordinates, but not the other quantities such as internal variables. For some material behaviors this can be modified, see for instance the `**global_output` option of the `***gen_evp` behavior in the Z-mat manual.

Syntax:

For rotations specified using coordinate axes:

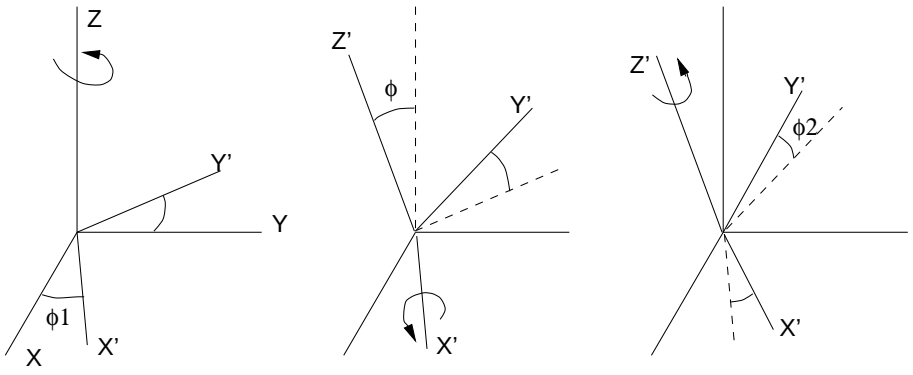
```
*rotation
[ x1 x* y* [z*] ]
[ x2 x* y* [z*] ]
[ x3 x* y* [z*] ]
```

The arguments `x1, x2, x3` indicate the components of direction vectors for the transformed coordinate frame. Exactly one direction is required in 2D problems, and two directions are required in 3D. The order of definition is not important. The local coordinate system may

be assembled with any of the geometrical axes. The input vectors will also be normalized by the program to automatically make unit vectors.

Using the notation here that *t1* is the first direction vector defined, and *t2* is the second (for 3D problems), direction vectors of the coordinate system are defined as follows: The first vector is collinear to *t1*. The second vector is a vector in the plane defined by *t1*,*t2* and is perpendicular to *t1*. The third direction will always be calculated using the vector product of the first two vectors. The *t* vectors will replaced by those given by you using the `x1`, `x2`, `x3` choices.

Rotation by giving Euler angles is similar. The significance of the three angles is given in the following figure:



Syntax:
For rotations specified using Euler angles:

```
*rotation  ϕ1  ϕ  ϕ2
```

Using the Rodrigues' representation, a rotation can be fully defined by a single vector $\underline{V}$. The axis of rotation $\underline{n}$ and the angle θ are then:

$$\underline{n} = (n_1, n_2, n_3) = \frac{\underline{V}}{||\underline{V}||}$$

$$\theta = 2 \arctan\left(||\underline{V}||\right)$$

An efficient way to compute the corresponding rotation matrix $\mathbf{R}$ is given by the Rodrigues' formula:

$$\mathbf{R} = \mathbf{I} + \sin(\theta)\mathbf{K} + (1 - \cos(\theta))\mathbf{K}^2$$

with

$$\mathbf{K} = \begin{bmatrix} 0 & -n_3 & n_2 \\ n_3 & 0 & -n_1 \\ -n_2 & n_1 & 0 \end{bmatrix}$$

Syntax:
For rotation specified using Rodrigues' vectors:

```
*rotation rod  V1  V2  V3
```

***var_mat_ini**

Description:
This option allows one to give the initial values for internal or auxiliary material variables. This option allows specifying any of the material variables, and thus depends on the individual material model. Common uses are to give the initial porosity or damage for initially porous or pre-damaged materials.

Syntax:
The syntax for this option requires the name of an internal variable. The variable name is followed by a real number for the initial value to be applied, or by the keyword `function` and a function specification. The variable name can also be followed by either a character name for a binary data file, or the keyword `ascii_file` followed by a character name for an ascii data file. Finally, it is also possible to initialize with a script, introduced by the `z7p` keyword.

Data files must furnish values for all the integration points stored as `float` variables in ascii or binary format. Ascii data files may also contain comments. Functions may only depend on coordinates x, y and z of the integration points. Any coordinate present in the function, but not at the integration point (typically a function depending on z for a two-dimensional mesh, as in the example below), will have a value of zero, so care should be taken in those cases.

The syntax is summarized below:

`*var_mat_ini` *variable_name* *variable_value*
`*var_mat_ini` *variable_name* `function` *function* `;`
`*var_mat_ini` *variable_name* `[ ascii_file ]` *file_name*
`*var_mat_ini` *variable_name* `z7p` *script_name*

Example:
Some syntax examples follow:

```
*var_mat_ini epcum .015

*var_mat_ini
  porosity    0.1
  grain_size  0.001
  epcum       epcum.dat

*var_mat_ini
  porosity    function    x*x+(y-2.)-2000.*cos(z);

*var_mat_ini
  grain_size  ascii_file  grain_size_distribution.dat

% from $Z7TEST/Program_test/Material_test/INP/bending.inp
  *var_mat_ini  eel11 z7p eel11-SC.z7p
```

***mesh

Description:
Under the mesh description command we may specify the formulation used to satisfy the governing equations of the problem, including assumptions of a quasi-geometrical nature such as plane stress or plane strain conditions. The formulations possible in mechanical problems include classical small deformations, Total-Lagrangian, Updated-Lagrangian, or special cases such as the Cosserat continuum. Thermal problems do not have any alternate formulations.

The procedure also allows specifying a geometry file different than *prob*`.geof`.

Syntax:
The syntax for this command is summarized below:

```
***mesh [ element_type ]
 **file  filename
 **elset elset-name  [ element_type ]
  *section type
   ...
 **local_frame type
   ...
 **import type  fname
 **predefined type
```

The argument *element_type* defines the type of element (its formulation) which will be used throughout the element set in question. The different types possible are described on the following pages. In order to simply apply the element formulation to the whole structure, fill in the *element_type* on the same line as the `***mesh` keyword. In order to define different element types for different regions of the mesh, put the *element_type* keys after `**elset` sub-commands.

Note that plane element geometries in 2D require a specification of element formulation in order to distinguish plane stress or plane strain. The default element type only applies to axisymmetric geometries in 2D. Element names with `plane_strain` will enforce the $\epsilon_{33} = 0$ condition, while elements with `plane_stress` enforce $\sigma_{33} = 0$[10].

`**elset` *elset-name* sets the element formulation and possibly other characteristics for a given element set name. When used, no element formulation should be given after the `***mesh` keyword.

- `*section` *type* this elset sub-command assigns section properties to a shell mesh. See the separate pages following which describe the particular sections available (starting at page 3.169).

`**file` Indicates that the mesh is in a file named other than *problem*`.geof`.

`**import` Indicates that the mesh file is in a separate file, and in a format other than native Z-set. It is normally recommended to use the batch mesher to translate the mesh before running the analysis (see page 2.7).

[10] Plane stress is imposed in Z-set using additional degrees of freedom, and not at the material behavior level, although this exists too with some of the behaviors (notably `gen_evp`; see page 3.2). For plane stress behaviors, one must choose ironically `plane_strain` element formulations.

**local_frame Assigns a local frame to the degrees of freedom attached to an elset (see page following at 3.170).

**predefined *type* Predefined elements may be used to quickly perform calculations on certain meshes. These are limited currently to single elements which can be used to quickly test and verify material behaviors. The predefined type *type* may be chosen from the following elements: cax8, cax8r, c2d8, c2d8r, cax4, cax4r, c2d4, c2d4r, rve1d, rve2d, rve3d, c3d20, and c3d20r. One can use Zrun -H to see the installed predefined elements.

The elements will be defined in the positive quadrant in a rectangle or cube form with its limits at 1. Two dimensional elements have liset and nsets defined as bottom, right, top, and left.

Caution

In each calculation there may exist several different element types, and several different materials. Unfortunately, the dynamic structure of the program limits the ability to do exhaustive compatibility verification. Caution must therefore be given to the consistency of options within a calculation. The verification procedure will be improved in the next version of the code.

Example:
The following is an example of the "simple" use of ***mesh to assign the whole problem to be a plane stress case, with a single predefined mesh as a 8 nodes 2D element.

```
***mesh plane_stress
   **predefined c2d8
```

The second example is of a mesh stored in a fully qualified path name file, and two element sets with different element formulations.

```
***mesh
  **file  /home/user/meshes/big_mesh.geof
  **elset rubber_part total_lagrangian
  **elset stiff       small_deformation
```

Mechanical calculations:
The following are brief descriptions of the mechanical element formulations available.

- `spr` Spring (or pinned truss) elements for one or 2 node geometries (`l2d1 l2d2 l3d1 l3d2`). This element requires that a valid spring material behavior be assigned to all such elements (c.f. 3.2). The two node element realigns with the line between those two nodes, and is therefore non-linear.

- `linear_spring` A linearized version of the spring element, so no non-linearities are introduced.

- `small_deformation plane_strain small_deformation_plane_strain` Mechanical elements in small deformation formulations. The integration volume is that of the initial structure, and significant rotations may cause erroneous calculation of the strain. The default 2D geometry is axisymmetric. Choosing the `plane_strain` option will enforce the $\epsilon_{33} = 0$ condition.

- `plane_stress` Plane stress 2D element formulations. In order to be compatible with all material behaviors, this element adds degrees of freedom to each Gauss integration point which represent the strain ϵ_{33}. These DOFs are used to enforce zero surface pressure and therefore $\sigma_{33} = 0$. Relationships may however be defined using the MPC command in order to specify generalized plane strain conditions. Setting the `EZ` (ϵ_{33}) DOF variable to be uniform over an elset yields zero overall applied pressure, but allows variation in the $\sigma_{33} = 0$ field. The 3-direction strain will of course be uniform.

- `small_deformation_updated plane_strain_updated plane_stress_updated` Identical to the non `_updated` formulations but using the geometry at the end of an increment as the integration volume. This produces non-linear structural behavior even with linear behavior models.

- `small_deformation_select_int small_deformation_select_int_updated` Special version of small deformation elements for **linear** interpolation to solve the problem of strong local variations or oscillations in the stress fields (esp. pressure terms).

- `cb_shell cb_shell_updated_lagrangian` Shell elements formulated using the so-called "continuum based shell" assumption: the mechanical response of the shell element is computed using an underlying 3D volumic element whose geometrical configuration is dynamically constructed at runtime by the mother shell element. See for instance the corresponding chapter in the book: *Nonlinear Finite Elements for Continua and Structures*, T. Belytschko, W.-K. Liu and B. Moran, Wiley.

  This element type uses 5 degrees of freedom per node: three displacements `U1`, `U2`, `U3`, and two rotations `W1` and `W2`. Thus the rotation is assumed to be continuous along the shell.

  Note that you must also give the initial thickness of the shell for this elset, using the following syntax:

```
**elset shell_part cb_shell
  *thickness 0.1
```

- `total_lagrangian` `total_lagrangian_plane_strain` Elements formulated for large displacement using Total-Lagrangian assumptions. These elements do not have any straining under rigid body motions.

- `total_lagrangian_mixte_u_p` `total_lagrangian_plane_strain_mixte_u_p` Incompressible elements in large displacements. The formulation is a mixed pressure-displacement Total-Lagrangian method with degrees of freedom for the pressure at certain nodes.

- `total_lagrangian_mixte_u_ps` `total_lagrangian_plane_strain_mixte_u_ps` Incompressible elements in large displacements. The formulation is a mixed pressure-displacement Total-Lagrangian method with a single pressure degree of freedom per element.

- `updated_lagrangian` `updated_lagrangian_plane_strain` `updated_lagrangian_plane_stress` Updated Lagrangian element formulation for finite strain calculations. Must be used in the finite strain case for materials with internal variables such as metal plasticity or porous plastic materials. This element type *must* be used in conjunction with behaviors modified for updated Lagrangian methods (see the command `***behavior`).

- `periodic` `periodic_plane_strain` periodic elements in axisymmetric, 3D, or plane strain. These elements allow one to impose the mean stress values for a periodic cell. Note that the displacement field that is saved by default is the *total* displacement field and *not* only its the periodic part. You can override this choice and save only the periodic part by appending after the `periodic` keyword: `periodic_info **periodic_displacement_field`.

- `smallw` `smallw_updated` Continuous mechanical element small displacement, small rotation - 3D only. This element combined with an appropriate behavior such as a smallw crystal can be used to model texture evolution in a crystal for moderate deformations.

- `2_5D` `2_5D_updated` two and one half dimension elements. These use a 3D material law with a 2D geometry. There are six degrees of freedom per Gauss point (element DOFs) $[t_1\ t_2\ t_3]$ and $[w_1\ w_2\ w_3]$. The structures displacement field is:

$$\vec{u}\,(x,y,z) = \vec{u}_0\,(x,y,z) + \vec{u}_1\,(x,y)$$

$$\begin{aligned} u^x &= u_1^x + z t_1 - w_3\,(y - Y_0)\,z \\ u^y &= u_1^y + z t_2 + w_3\,(x - X_0)\,z \\ u^z &= u_1^z + z t_3 + w_1\,(y - Y_0)\,z - w_2\,(x - X_0)\,z \end{aligned}$$

  With this element, one can use full 3D material behaviors, and fix out of plane degrees of freedom. The values of X_0 and Y_0 are modified through the `***specials` command.

- `2_5D periodic` It is a small deformation periodic 2_5D element. These use a 3D material law with a periodic 2D geometry The displacement field is searched for in the form

$$\underline{\mathbf{u}}(x,y,z) = \underset{\sim}{\mathbf{E}} \cdot \underline{\mathbf{x}} + z\underline{\mathbf{t}} + z\underline{\mathbf{w}} \wedge (\underline{\mathbf{x}} - \underline{\mathbf{x}}_0) + \underline{\mathbf{U}}(x,y)$$

with $\underline{\mathbf{x}}_0(x_0, y_0, 0)$ and E_{11}, E_{22}, E_{12} are the only non-vanishing components of $\underset{\sim}{\mathbf{E}}$. The nodal dof are U_1, U_2. The element dof are $E_{11}, E_{22}, E_{12}, t_1, t_2, t_3, w_1, w_2, w_3$. The previous equation is now written in components:

$$\begin{aligned} u_1 &= E_{11}x + E_{12}y + zt_1 - w_3 z(y - y_0) + U_1(x, y) \\ u_2 &= E_{12}x + E_{22}y + zt_2 + w_3 z(x - x_0) + U_2(x, y) \\ u_3 &= zt_3 - w_2 z(x - x_0) + w_1 z(y - y_0) \end{aligned}$$

The gradient of the displacement field is:

$$\begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} \\ u_{2,1} & u_{2,2} & u_{2,3} \\ u_{3,1} & u_{3,2} & u_{3,3} \end{bmatrix} = \begin{bmatrix} E_{11} + U_{1,1} & E_{12} + U_{1,2} - w_3 z & t_1 - w_3(y - y_0) \\ E_{12} + U_{2,1} + w_3 z & E_{22} + U_{2,2} & t_2 + w_3(x - x_0) \\ -w_2 z & w_1 z & t_3 + w_1(y - y_0) - w_2(x - x_0) \end{bmatrix}$$

The associated strain field is:

$$\begin{bmatrix} \varepsilon_{11} & \varepsilon_{12} & \varepsilon_{31} \\ - & \varepsilon_{22} & \varepsilon_{23} \\ - & - & \varepsilon_{33} \end{bmatrix} = \begin{bmatrix} E_{11} + U_{1,1} & E_{12} + (U_{1,2} + U_{2,1})/2 & (t_1 - w_3(y - y_0) - w_2 z)/2 \\ - & E_{22} + U_{2,2} & (t_2 + w_3(x - x_0) + w_1 z)/2 \\ - & - & t_3 + w_1(y - y_0) - w_2(x - x_0) \end{bmatrix}$$

- `pressure` Fluid interface mesh pressure elements.
- `cosserat` `cosserat_plane_strain` `cosserat_plane_stress` Mechanical elements modified for the Cosserat continuum. This method adds DOFs at each node for an independent micro-polar rotation named `W3`. This method allows limitation of localization under shear deformation in strain-softening materials. The method uses a characteristic material length which must be input in the material definition, and the behavior must be modified for the Cosserat formulation (see the `***behavior` command).

```
****calcul
 ***mesh
```

Thermal calculations:
For thermal calculations, the type of element is completely defined by the geometry given in the `.geof` file. The `***mesh` command should thus only be used if an alternate geometry file name is to be used.

***section uniform**

Description:
This command assigns a uniform section of 1D elements to integrate the shell thickness. This command is a shortcut for having a separate mesh file for the section, which is possible too.

Syntax:

```
**elset elset-name  shell-formulation
 *section uniform num-elem  ele-type  thickness
```

Example:
Here's a short example of a shell structure with 2 different section properties.

```
***mesh
 **elset rib mindlin_shell
   *section uniform 1 c1d3 .4
 **elset flange mindlin_shell
   *section uniform 1 c1d3 .25
```

**local_frame

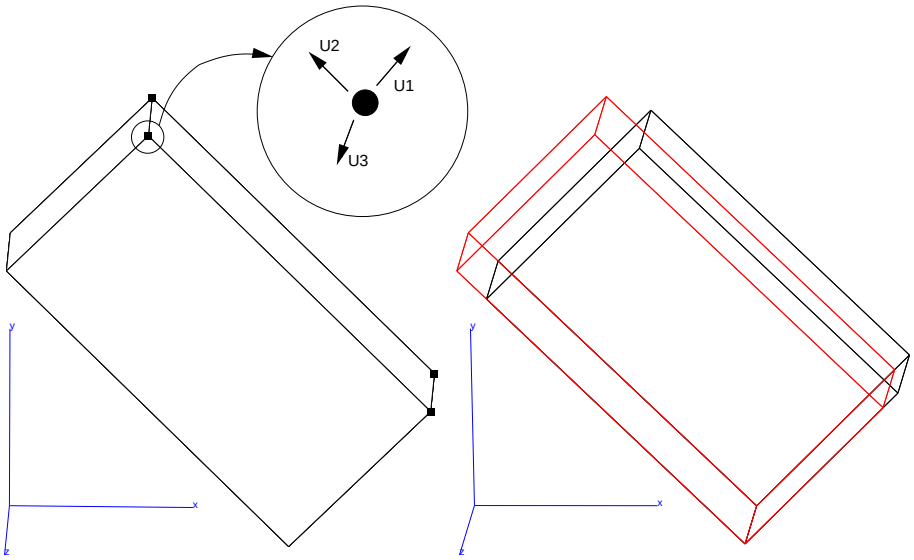
Description:
This command assigns a local coordinate system to the degrees of freedom attached to nodes of a given node set. The command is not compatible with every element formulation, and thus care should be used with this option. When in doubt, ask your supplier for additional advice.

Syntax:

```
**local_frame nset
 *euler Euler rotation definition
 *cartesian Cartesian rotation definition
 *cylindrical
   center
 [ axis ]
```

Example:
The following example shows the use of this option to make a sliding boundary condition at an orientation away from the coordinate axes.

```
***mesh small_deformation
 **local_frame top-ext
  *euler   45.0 0. 0.
 **local_frame bottom-ext
  *cartesian x1  .05 .05 0. x3 0. 0. 1.
```



Another example is to impose a cylindrical coordinate system. This type linearizes the curvature, and is thus valid only for very small deformations.

```
***mesh plane_strain
 **local_frame outer
  *cylindrical
   (2. 1.)
```

***sub_problem

Description:
This command is used to define a sub-problem in a weakly coupled analysis, and specify data to be transferred in order to interface with other sub-problems.

Syntax:

```
***sub_problem type  prob-name
 **transfer type
  *variable var-name
  *file     out-file
...
```

The available transfer types are described below. In version 8.0 the transfers generate a separate set of files for each exported variable[11], with names *out-file*`.catalog` *out-file*`.first` *out-file*`.initial` and *out-file*`.final`.

`integ_integparam` Make an integration variable (grad, flux, vint, vaux) and exported variable for use as an `*ip` type external parameter in another sub-problem.

`integ_nodeparam` Make an integration variable (grad, flux, vint, vaux) and exported variable for use as an `*node` type external parameter in another sub-problem. These variables are extrapolated, and will therefore have some error. The extrapolation is also *averaged* between elements, so it will not account for discontinuous materials well.

`node_nodeparam` Export one of the problems nodal variables (degree of freedom) for use as a node based external parameter in another problem (e.g. temperature).

`node_kinematic` Export nodal displacements in a way which is suitable for the `***impose_kinematic` option (see page 3.149).

[11] these will be combined into a "transfer" database in the next release

***output

Description:
This command allows management of the output data files. By default, the code will only store the nodal DOF and associated reaction values in the file *problem*`.node`. In order to have the values of the material behavior variables the user is required to specify some of the sub-procedures below.

The version 7 of Z-set allows multiple output specifications, so the user may optimize disk usage for a problem. Normally this is very useful when the structure is to be output only at certain critical points during the calculation, while data plots may be output more frequently using the `**curve` sub-option. Many of these curves may be specified with different output files, thus automating the post-processing stage of the analysis. Older versions of the code required saving the *entire* calculation with the frequency desired for the curves even if their information was rather specific. Use of output data files for the curves also allows automation of their generation using standard scripts with plotting software.

Syntax:

```
***output  [ name ]
[   **linear_solution  ]
[   **contour  ]
[   **no_contour  ]
[   **contour_by_element  ]
[   **no_contour_by_element  ]
[   **value_at_integration  ]
[   **no_value_at_integration  ]
[   **save_in_material_frame  ]
[   **component name1 name2  ]
[   **test tname  ]
[     *plot  ]
[     *precision digit  ]
[     *small small  ]
[     *gauss_var elem_id pg_id var1 var2 ...  ]
[     *node_var node_id dof_name1 dof_name2 ...  ]
[     *point_var (x y z) var1 var2 ...  ]
[     *nset_var nset dof_name1 dof_name2 ...  ]
[     *liset_var liset var1 var2 ...  ]
[     *node_extrapolated node_id var1 var2 ...  ]
[     *element_node_var node_id var1 var2 ...  ]
[   **curve tname ]
[   **frequency  ]
[     *dtime dtime  ]
[     *at time1 time2 ...  ]
[     *increment nb_increment  ]
[     *cycle cycle1 cycle2 ...  ]
[     *period start stop  ]
[   **store_global_matrix matrix_type when filename  ]
```

```
[   **verbose  ]
[   **no_defaults  ]
[   **output_first  ]
[   **save_parameter [list of parameters' names] ]
[   **reaction | **no_reaction ]
[   **ele      | **no_ele ]
[   **node     | **no_node ]
```

The optional character *name* may be used to specify the output file names. In the event of more than one `***output` specification, the use of names will prevent overwriting of the same files for each output. This is essential to avoid, as the storage records are referenced using the `.ut` directory which would be overwritten for two like-named outputs. The default name will be the problem name *problem* which was given when running the Z-set program.

Names of the integration point variables are furnished for each material behavior. The names of the DOFs are given in the table on page 6.4.

The sub-procedures are:

`**linear_solution` This option indicates that the solution should be post processed with the intention to view linear summations of different load cases. The time steps are thus considered to be "load cases". In fact, this command only adds the keyword `**linear_solution` to the *problem*`.ut` file header, which can be edited by hand as well.

`**value_at_integration` This option indicates that default variables held at the integration points are to be stored in the file *problem*`.integ`. In each material behavior there will be a special set of variables which are stored by default (documented in the sections for each behavior). When the problem has more than one material specified, the union of all variable groups for all materials is stored at each integration point. If the value is not actually defined for a material at a certain integration point a null value will be stored[12].

`**no_value_at_integration` This option allows to disable storage of the default variables held at the integration points in the file *problem*`.integ`.

`**save_in_material_frame` This option indicates that default variables held at the integration points are to be stored in the file *problem*`.integ`. By default they are stored in their local material coordinate system (if a `*rotation` or a `*pre_problem_treatment` is defined), but not the stress and strain fields `sig` and `eto`. With this option the latter are not only saved in the global coordinate frame, but also in the material coordinate system, in additional variables `sig_lf` and `eto_lf`. This option is very useful for composite materials.

`**contour` The contour option indicates that the integration point (material) variables will be stored at the nodal positions in the file *problem*`.ctnod`. Material variables are extrapolated to the nodal positions for all the elements. Each node has therefore a number of extrapolated values for each material variable to be stored, equal to the number of elements connected to that node. The recorded output value at the node will be the arithmetic mean of the individual element extrapolations. Data stored in this way will incorrectly smooth stress discontinuities across material boundaries. Such

[12] More intelligent strategies for outputting variable values will be implemented in future versions of the code

output is however better in the single material case because it assures continuity of variable fields from one element to another.

**no_contour This option allows to disable storage of the integration point (material) at the nodal positions in the file *problem*.ctnod.

**contour_by_element This output option causes the material variables to be saved at nodal locations by extrapolation, element by element. The data is stored in a binary file named *problem*.ctele. Element extrapolation leaves the discontinuities at boundaries, and will thus represent the multi-material interfaces correctly. Each node will have a number of values stored in the output file corresponding to each attached element.

The magnitude of discontinuity as observed using this method of extrapolation may often be used as a measure of the mesh quality. Better meshes should show small differences element to element in the material variable values.

**no_contour_by_element This option allows to disable extrapolation and storage of the material variables at nodal locations, element by element, in the file *problem*.ctele.

**component This option allows direct specification of the material variables to be stored. This allows a reduction of the output file sizes by limiting the material variables output, or outputting of secondary variables not normally stored by the default output.

The code does not verify that the variables exist, as that would pose problems for multi-material problems having a defined variable in one element set only. The stored value for undefined variables will always be zero. Note that if a variable is forgotten with this option it will be required to re-run the entire problem to obtain its value. It is also essential to leave this option untouched when using the ***restart options.

Syntax is the following:

```
**component
  name1 name2 ...
```

where *name#* are the variable names to be output. One may see all the variable names defined by each material law by running Z-set with the verbose -v command line switch, or using the **verbose option (see below).

**save_parameter This option indicates that we wish to store the parameters defined in the calculation[13]. The parameters are either stored in the file *problem*.node or *problem*.integ, depending on their base location. These parameters can then be examined together with the calculation results in Zmaster, or used in post-processings.

**frequency This option adjusts the frequency of output storage for all the options within an ***output block. The option allows a variety of definitions in problem time or in increments. If a given output time does not correspond directly to a discrete calculation time, the next solution after the requested time will be output. The problem is always stored at the end of the calculation.

The sub-options are described below:

*dtime sets the output in by increment of time. This option takes a single real value for the increment of time between each output.

[13] See also **save_coefficients in the Materials Manual, that saves material parameters.

*at specifies the exact times for output. This option takes a list of real values for the output times.

*increment outputs at every specified number of increments. The option takes a single integer which defines the increments between outputs. For example *increment 3 will store increments 3,6,9, etc.

*cycle outputs only during the specified cycles numbers. Any number of integer cycle numbers may be entered after the keyword.

*period outputs only during the specified time interval

**test is used to output specific data in an ASCII text file. Giving an optional character name after the **test option will define the name of the output file for all the variables specified after this option. By default, the filename will be the output name (as given after ***output or the problem name in the absence of an output name), suffixed by .test.

The output data will be controlled by a variety of options:

*plot specifies that the output file is to be stored in column format (recommended). This allows direct plotting of the output data. The variable names will be listed at the head of the file with the # character before (a standard comment character).

*precision gives the precision for the output variables (number of digits after the decimal). All variables are output in exponential form. The default precision is 6 decimal places.

*small gives the value below which the result must be considered zero. Default value is 10^{-9}.

*gauss_var Specifies output of Gauss point data. The syntax requires the element number (integer) followed by the local Gauss point number (integer - numbering is described in the appendix). After the Gauss point specification an unlimited list of character names for the material variables to be output is given. These names may be any of the valid material names as viewed while running with the -v command line switch.

*node_var This option specifies that nodal quantities are to be output. The syntax requires a node number (integer, or alternatively the name of a nset containing a single node) followed by a list of character names for the nodal variables to be output. The variables may be DOF names (such as U1, U2, etc) or DOF reactions prefixed with an R (such as RU1, RU2, etc), or coordinates (X, Y, Z).

*point_var Output data at a point defined by its coordinates. This option is especially useful for computations with remeshing (where node_var is not appropriate).

*nset_var Stores nodal *reactions* as summed over a node set. The syntax requires a node set character name to be given followed by a list of the nodal DOFs from which the reaction is to be taken. Note that the R is not given here. An example is *nset_var top U2 which will output the applied load (F_2) over the surface defined by the node set top.

*liset_var Output data along a line set. Data is output in separate files for each output time, so it is **strongly** suggested to use the **frequency option. The syntax takes a liset name, and a list of variables to be output on the liset. The

standard definitions `X`, `Y` and `Z` may be used to output the coordinates. Any other nodal variable is possible to output on the liset. If material variables are demanded, they will be output using an averaged extrapolation of the adjacent elements. Example: `*liset_var outside Y U1 U2 RU1`

`*node_extrapolated` Material variables will be extrapolated to the nodal coordinates. The syntax takes the node number, followed by a list of integration variables.

```
*node_extrapolated 22 sig22
```

`*element_node_var` Material variables will be extrapolated to nodes using the shape function of a single element. This gives the extrapolated value valid on a material interface where there are may be variable discontinuities.

```
*element_node_var  14 22 sig22
```

Note that mixing of the `*-_var` options may be given in order to force sequential output of all the variables.

`**curve` This option is identical to `**test` with the `*plot` sub-option automatically set.

`**store_global_matrix` allows storage of the global matrix associated to the structure's mesh in a file. The matrix may thus be saved at different points in the calculation and re-used for the calculation of resonant eigen frequency analysis of the pre-loaded structure (see `***preload`). The syntax is the following:

```
**store_global_matrix
matrix_name when filename
[...]
```

where *matrix_name* is the name of the matrix to be stored, *when* is the time at which the matrix is to be saved, and *filename* is the filename to be used for the matrix. Currently the only value for *matrix_name* is `stiffness_matrix` which stores the global problem stiffness, or `conduction_matrix` for thermal problems. Such triplets may be repeated, in order to write the matrix at different times.

`**verbose` Specifies verbose screen output during the calculation. As the screen output is stored in the file *problem*`.msg` this option may create a very large file for long (many increments) problems. It has been found that the `.msg` file may rapidly become larger than the primary output files so verbose output is not normally set. This option is equivalent to using the `-v` command line switch.

The verbose option is especially useful in problem setup, however, as it causes output of the local material variables to the screen.

`**no_defaults` Cancels all the default parameters. If used alone, this totally cancels the output.

`**reaction, ele, node` Enables (resp. disables for the `no_` prefixed options) the storage of these values in the results files.

```
                                                    ****calcul
                                                     ***output
```

Example:

```
  ***output
     **test
      *precision 4
      *small 1.e-6
       *gauss_var 1 1 evi11 sig22
       *node_var  53 U1
     **frequency
       *dtime 1.5
       *increment 10
     **store_global_matrix
       stiffness_matrix 10.0 disk.stif

***output
   **frequency
     *at_time  30.   60.
               270.  300.
               1770. 1800.
               3570. 3600.
               7170. 7200.
   **contour
   **value_at_integration
   **save_parameter

 ***output
    **no_defaults
    **curve q1.charge
       *nset_var top U2
    **curve q1.dv9
       *node_var 2 U1
       *node_var 25 U1
    **curve q1.stress_strain
      *gauss_var 1 1 eto22 sig22 epcum
```

```
****calcul
 ***output
  **test
   *J
```

**test

*J

Description:

This command is used for calculating the J integral or ΔJ using a line set only. The command will work in 2D plane stress and plane strain only. Any material may be used, and any standard mechanical element formulation.

The integral is evaluated by performing direct integration of the contour integral

$$J = \int_{\Gamma} \left[W n_x - \sigma_{ij} n_j \frac{\partial \mathbf{u}}{\partial x} \right] d\Gamma$$

The calculation will be negative if the path is oriented clockwise. Also, remember that if there is a plane of symmetry about the crack plane, and you only model this one half, the associated J integral will be one half of its real value.

Syntax:

The syntax for this output command is the following:

```
*J liset_name [ time-ini time-end ]
  ...
```

Each contour desired should be entered as a separate `*J` command.

Adding the two optional real values *time-ini* and *time-end* will cause the J integral to be evaluated as the ΔJ integral. Note that this is not an incremental version of J as the LEFM ΔK is used, but rather an evaluation of J over a path with residual stresses.

Caution

Because the J integral is computed along a boundary, it is using extrapolated Gauss point data to the nodes, which is then again re-interpolated to the intermediate points on the line. This can accumulate some error, especially if the mesh is too coarse. One should verify the stability of J calculations with respect to path (path independence), and the mesh size.

Also, because this is a `***output` option, it is subject to the `**frequency` command. As the J integral is integrated over time, for non-linear problems it's calculation requires a sufficient number of time steps.

Example:

```
***output
 **contour
 **test
  *J path1
  *J path2

***output
 **contour
 **test
  *J path1 0.  10.
  *J path1 20. 30.
```

***parameter

Description:
Using the coefficient mechanism, material behavior may depend on both internal variables calculated by the material law, and external user specified parameters which are essentially problem loading. The coefficient syntax is discussed in the `***behavior` chapter in the Z-mat manual.

Use of the parameter mechanism may also intervene directly in the material law such as thermal deformations which depend on a temperature parameter. This temperature field may be the output of previous calculations to provide immediate capability of un-coupled or weakly coupled multi-domain solutions. Other non-standard parameters are also permissible allowing user data to be used in the calculation. One example is specifying a distribution of the density, and then making the elastic modulus depend on this density.

There are several methods of specifying the parameters within a geometry.

Syntax:
The baseline syntax for external parameters follows.

```
***parameter type parameter_name
   [ *node | *ip ]
      ...
```

type a `**`-level sub-command for the type of external parameter input which is desired. The different types are described in the pages which follow, including their specific values. In the absence of a *type* (input starts directly with `*`-level commands, or data entry fields, the type `**file` is taken as the default.

parameter_name Character name of the parameter. The naming of parameters is inherently open in the code. However, the use of certain names have particular signification for different options in the material behavior or global loading. An important example is the name `temperature` which is searched by the thermal strain material objects.

`*node` indicates that the parameter exists at nodal locations, and will be interpolated to integration points. Record sizes are the number of nodes. This is the default.

`*ip` indicates that the parameter exists at integration points. The record size will be the number of integration points in the mesh.

It is imperative that the parameters be defined at all times from $t = 0.$ to the end time of the problem. Between the times for which the parameter is defined through *file*, *function* or *uniform* the value will be linearly interpolated. It is therefore desirable that the times of parameter definition correspond to the end sequence times. Uniform values may be applied in an analogous manner as the boundary conditions with the use of tables.
. . . *continued*

```
****calcul
 ***parameter
```

Examples:
Two simple examples are shown below.

```
***parameter temperature
   0. uniform 125.
   1. uniform 175.
   2. uniform 175.
   3. uniform 125.

***parameter humidity
  *ip
   0. uniform  0.25
   1. uniform  0.25
   2. uniform  0.75
   3. uniform  0.25
```

**file

Description:
This is the default parameter type. It allows mixed use of uniform parameter values, parameter values specified by a function and binary file records containing the values of the parameter field. This parameter type is very similar to the `ascii_file` type (see page 3.184), but binary files generally load faster, which is important for very large parameter files. Note: some command options are available only in Z-set version 8.4 or newer.

Syntax:

```
 **file param-name
[ *node ]
[ *ip ]
[ *dtime ]
[ *rec_size  size ]
[ *table_file tablefile name ]
[ *cycle_conversion  start  end  period ]
   time  uniform  value
   time  function func;
   time  file     filename  record
```

`**file` takes *param-name* (a character string) as argument for specifying the name of the parameter. This name will be used to reference the parameter values elsewhere in the problem.

`*node` specifies that the parameter is applied at nodes. This is the default.

`*ip` specifies that the parameter is applied at integration points.

`*dtime` converts the absolute *time* values given in the table below to time increments.

`*table_file` takes as argument the character string *table_file name* as the name of the file from which the table containing the list of times or time increments with their associated parameter types (*uniform*, *function* or *file*) can be read. This is an alternative to directly specifying the table here (see below), and can be useful for automatic table generation by scripts.

`*rec_size` enter an integer defining the record size. This value should be the number of nodes for `*node` type parameters, or the total number of integration points for `*ip` type parameters.

`*cycle_conversion` real values for *start* time of cycles, *end* time, and cycle *period* are to be input. This command generates a variable `time` which can be used in function records to define the time.

. . . *continued*

```
****calcul
 ***parameter
  **file
```

The table specifying parameter values contain the following elements:

time a time value for the given parameter data. Parameter values during the calculation will be found using linear interpolation between given times. Do not forget to specify a value for time = 0.0. If the `*dtime` command is given, the time value becomes a time increment value. The *time* values may also be specified as a `FUNCTION`. This is especially useful in conjunction with the `*cycle_conversion` command (see example below).

value a uniform parameter value.

func a `function` specified in the standard Z-set manner. Do not forget the ; at the end.

filename name of the binary file to use. Different files can be given for different time values.

record record number in the file. These are numbered from zero (0 is the first record).

File format:
The format for the parameter files consists of an ordered list of float (single precision floating point) values in big endian format. The ordering corresponds to the nodal numbering in the *problem*`.geof` file. Each record therefore comprises only this list of float values for each node. The record numbers are noted to be the record number (number of the block storing the entire nodal data set) and not the file position as would be read in C or Fortran programming.

Examples:
The example below shows the use of mixed parameter types in order to impose a variable temperature field.

```
***parameter
 **file temperature
  *node
  *rec_size 8
   0.    uniform 40.
   1.    file  ../DATA/thermal_field.node 1
   2.    file  ../DATA/thermal_field.node 2
   3.    uniform 120.
```

... continued

```
****calcul
 ***parameter
  **file
```

Here is a relatively complex example using a function to repeat values for many cycles. The *time* values are now specified using a function.

```
***parameter
 **file temperature
  *node
  *rec_size 757
  *cycle_conversion 1. 1.e20 60.0
   0.                              uniform  23.0
   function 1.0+      cycle*60.0; file     mechanical.data  20
   function 1.0+ 10.0+cycle*60.0; file     mechanical.data  5
   function 1.0+ 14.0+cycle*60.0; file     mechanical.data  6
   function 1.0+ 18.0+cycle*60.0; file     mechanical.data  7

   ...

   function 1.0+ 56.0+cycle*60.0; file     mechanical.data  19
   function 1.0+ 60.0+cycle*60.0; file     mechanical.data  20
```

**ascii_file

Description:
This parameter type allows an easy input of parameter data with an ASCII file. The use of ASCII allows simple generation of the table values, for example by using scripts. Just as for the binary file parameter type (see page 3.181), it allows mixed use of uniform parameter values, parameter values specified by a function and binary file records containing the values of the parameter field. Note: some command options are available only in Z-set version 8.4 or newer.

Syntax:

```
 **ascii_file param-name
[ *node ]
[ *ip ]
[ *dtime ]
[ *table_file tablefile name ]
[ *cycle_conversion  start  end  period ]
[ *rec_size size ]
   time  uniform  value
   time  function func;
   time  file     filename  record  column
```

ascii_file takes *param-name* (a character string) as argument for specifying the name of the parameter. This name will be used to reference the parameter values elsewhere in the problem.

*node specifies that the parameter is applied at nodes. This is the default.

*ip specifies that the parameter is applied at integration points.

*dtime converts the absolute *time* values given in the table below to time increments.

*table_file takes as argument the character string *table_file name* as the name of the file from which the table containing the list of times or time increments with their associated parameter types (*uniform*, *function* or *file*) can be read. This is an alternative to directly specifying the table here (see below), and can be useful for automatic table generation by scripts.

*rec_size specifies the *size* of each data entry record. For FEA problems and *node type parameters, this is the number of nodes (can be found in the *problem*.geof file, or in Zmaster). For *ip type parameters, this is the number of integration points.

*cycle_conversion real values for *start* time of cycles, *end* time, and cycle *period* are to be input. This command generates a variable time which can be used in function records to define the time.

. . . *continued*

The table specifying parameter values contain the following elements:

time a time value for the given parameter data. parameter values during the calculation will be found using linear interpolation between given times. Do not forget to specify a value for time = 0.0. If the `*dtime` command is given, the time value becomes a time increment value. The *time* values may also be specified as a function. This is especially useful in conjunction with the `*cycle_conversion` command (see the example given in the `***parameter **file` section on page 3.181).

value a uniform parameter value.

func a function specified in the standard Z-set manner. Do not forget the ; at the end.

filename name of the ASCII file to use. Different files can be given for different time values.

record record number in the file. These are numbered from zero (0 is the first record).

column column number for the data. Column number starts with 1.

Example:
The following is an example of a variable Young's modulus imposed with an `ascii_file` parameter.

```
***parameter
 **ascii_file yng
  *rec_size 20
  *node
   0.0   file param_as_young.inp 0 2
   1.e20 file param_as_young.inp 1 2
```

The file `param_as_young.inp` of the example above might look like this :

```
 1.   200000.0   % first line in file (rec 0)
 2.   195000.
 3.   198000.
...
 20.  199000.    % end of 0th rec
 1.   220000.0   % start 1st record
...
```

The parameter is then applied in the constitutive law:

```
***behavior gen_evp
 **elasticity isotropic
   young = yng      % young assigned to field parameter
   poisson 0.25
 **save_coefficients
   young
```

If the parameter is to be applied at integration points, the data file must look like this :

```
                                                    ****calcul
                                                     ***parameter
                                                      **ascii_file

  1.   200000.
  1.   195000.
  1.   198000.
  2.   211000.
  2.   205000.
  2.   203000.
 ...
```

where the fist columns indicates the element number. In this example, the elements have 3 integration points. `rec_size` hes to be set to the total number of integration points.

**from_results

Description:
The **from_results command indicates that the parameter will be imported from a previous computation. It can of course be imported from a Z-set computation[14] but can also be imported from other software (ABAQUS odb for example). Z-set then does the necessary time interpolations. In case of non-conforming meshes, where space interpolation is also necessary, use the **from_results_with_transfer syntax instead (see next page).

Syntax:

```
**from_results param-name
 *database format file_base
[ *initial_value initial_value ]
[ *remap old_name new_name ]
[ *ip ]
```

param-name a character string name for the parameter. This name will be used to reference the parameter values elsewhere in the problem.

format is the format of the imported file. For example, use Z7 to import Z-set results or odb for ABAQUS. See page 2.8 for a complete list of known formats.

file_base is the filename prefix of the results files.

initial_value is a constant value to be used at $t = 0$.

remap : this command allows to rename a field named *old_name* in the results file to *new_name* in the current computation (actually *new_name* must be */param-name*).

Example:
The following is an example of temperature imported from a previous Z-set computation:

```
***parameter
 **from_results temperature
  *database Z7 specimen_thermal
  *remap  TP  temperature
```

[14] and thus gives an alternate and simpler syntax than **results, page 3.190

**from_results_with_transfer

Description:
The **from_results_with_transfer command indicates that the parameter will be imported from a previous computation. It is similar to **from_results and additionally allows for non-conforming meshes by doing the proper space interpolations.

Syntax:

```
**from_results_with_transfer param-name
 *database format file_base
[ *initial_value initial_value ]
[ *remap old_name new_name ]
[ *ip ]
[ *locator] locator-type
[ *integ_transfer] transfer-type
```

Most options are shared with **from_results (refer to the previous page); specific ones are:

locator-type is the localization method; the default (bb_tree) is currently optimal.

transfer-type is the method used to transfer between integration points. The default is currently nearest_gp. See page 3.153 for a list of available methods.

Example:
The following is an example of temperature imported from a previous ABAQUS computation:

```
***parameter
 **from_results_with_transfer temperature
  *database odb thermal_model
```

**function

Description:
This parameter type is specified as a function of (t, x, y, z).

Note:
If the function only depends on time, you may also use the `**table` syntax (see page 3.191).

Syntax:

```
 **function param-name
  *function function(t,x,y,z);
  *tables   table-names
[ *node | *ip ]
[ *time_independent ]
```

param-name a character name for the parameter. This name will be used to reference the parameter values elsewhere in the problem.

function a (t, x, y, z) function (do not forget the terminating ;)

table-names a (list of) table name referring to the `***table` section. If more than one name is given, the first valid one is used (this validity being decided at each increment).

`*node | *ip` specifies whether this parameter is based at nodes (the default) or at integration points.

`*time_independent` this switch declares the parameter as time independent. The parameter is thus only computed once at the beginning of the computation. This can be necessary when random numbers are used in the function.

The parameter's value will be computed as the product between *function* and *table*.

Example:

The following is an example of a temperature gradient along x:

```
***parameter
 **function temperature
  *function 300. + 600. * x ;  % gradient between 300 and 900 degrees
  *node
  *tables plateau              % time dependence is given here

***output                      % adding this command will let you
 **save_parameter              % check the parameter's value in Zmaster
```

**results

Description:
The **results command indicates that the parameter will be read in from a Zebulon FEA results file (**changing is a deprecated alias for the **results parameter).

Syntax:
The syntax required is:

```
 **results param-name
[ *node ]
[ *ip ]
  *file_base fname
```

param-name a character string name for the parameter. This name will be used to reference the parameter values elsewhere in the problem.

*file_base enter *fname* a file name prefix for the results files. The first file of concern is *file-prefix*.catalog, which lists the initial and final times for the file inputs (presumably from another time step of a mechanical problem, see below). Binary files for the beginning of the problem, start of an increment and end of an increment are: *file-prefix*.first *file-prefix*.initial *file-prefix*.final The current increment should be within the time bounds given in the catalog file.

Example:
The following is an example of temperature values imposed with a table parameter.

```
***parameter
  **results temperature
   *node
   *file_base MechTherm/temp_out
```

**table

Description:
This type of external parameter is used for uniform values which use entries from ***table (see page 3.227) to determine the parameter value through time. The input is rather like a boundary condition.

Syntax:
The syntax required is:

```
**table param-name
[ *node ]
[ *ip ]
   value  table
```

param-name a character string name for the parameter. This name will be used to reference the parameter values elsewhere in the problem.

value a scaling value (base value - decimal input).

table a table name entered in a ***table section.

Example:
The following is an example of temperature values imposed with a table parameter.

```
***parameter
 **table temperature
    1.0 temp_tab
```

**z7p

Description:
This parameter type is specified as a z7p external script.

Note:
If the z7p function only depends on time, you may also use the **table syntax (see page 3.191).

Syntax:

```
**z7p param-name
 *z7p external.z7p
 *tables   table-names
[ *node | *ip ]
[ *time_independent ]
```

param-name a character name for the parameter. This name will be used to reference the parameter values elsewhere in the problem.

z7p a z7p script file where a function `double compute(x,y,z,t)` is defined

table-names a (list of) table name referring to the ***table section. If more than one name is given, the first valid one is used (this validity being decided at each increment).

*node | *ip specifies whether this parameter is based at nodes (the default) or at integration points.

*time_independent this switch declares the parameter as time independent. The parameter is thus only computed once at the beginning of the computation. This can be necessary when random numbers are used in the function.

The parameter's value will be computed as the product between *function* and *table*.

Example:

The following is an example of a temperature gradient along x:

```
***parameter
 **z7p temperature
  *z7p external.z7p;  % external script
  *node
  *tables plateau              % time dependence is given here

***output                      % adding this command will let you
 **save_parameter              % check the parameter's value in Zmaster
```

The associated z7p script is as follow :

```
                                                          ****calcul
                                                           ***parameter
                                                            **z7p

double compute(double x, double y, double z, double t){
   double ret;
   if( x < 0.5 ){
      ret = 300. + 600. * x;
   }
   else{
      ret = 300. - 600. * x;
   }
   return ret;
}
```

***post_increment

Description:
This command starts reading of any number of post-increment calculations which are to be processed as additional output, or modifications based on the converged solution.

Syntax:

```
***post_increment
...
```

The post-increment sub-options depend on the installed modules, and so may be expanded by using "plug-ins." As of Z8.0, the following models are included.

CODE	DESCRIPTION
parks	Parks method for Stress Intensity Factor
j_integral_lorenzi	a method implementing the J–integral with virtual crack extension
non_local	A "weak" non–local model which can be applied to any state variable

Example:

```
***post_increment
    **parks
      perturb elset rg1
      tip_radius 55.
      da (1.e-3 0.)
      da (1.e-5 0.)
    **j_integral_lorenzi
      perturb elset rg1
      tip_radius 55.
      da (1.e-3 0.)
    **j_integral_lorenzi
      perturb elset next
      tip_radius 55.
      da (1.e-3 0.)
```

**i_integral

Description:
This option calculates the T stress using the method of Chen etal. [Chen01]. and DeLorenzi [Hors85]. The implementation is similar to the `**j_integral_lorenzi` in its use of a virtual crack extension.

Syntax:

`**i_integral`	
`perturb (elset | tip)`	*(name1 | node_num)*
`da (dax day)`	*to indicate the crack direction (and not the crack growth direction !).*
`tip nset`	*indicates the node corresponding to the tip*
`young E`	*gives the Young's modulus*
`poisson nu`	*gives the Poisson's ratio*
`[factor f]`	*indicates a multiplicative factor to be used in the case of symmetries. Default value is 1.*

The T—stress calculations works with small deformation plane stress and plane strain elements. Like for the deLorenzi or Parks methods, the syntax `perturb elset next` will search the next elset surrounding the crack tip. This syntax can be repeated.

**j_integral_lorenzi

Description:
This option calculates the J integral using the method of Horst and DeLorenzi [Hors85]. The implementation is analogous to the **parks (page 3.199) in its use of a virtual crack extension. This method is more reliable (accurate, insensitive to mesh density, and path independent) for small deformation analysis than the output method *J described on page 3.178. It is also valid for axisymmetric or 3D analysis where *J is not. However, unlike that method the **j_integral_lorenzi is limited to small deformation, and can not calculate ΔJ.

Remember that is the case of linear elasticity:

$$J = \begin{array}{ll} \frac{K^2}{E}(1-\nu^2) & \text{plane strain} \\ \frac{K^2}{E} & \text{plane stress} \end{array}$$

Syntax:

```
**j_integral_lorenzi
  perturb (elset | tip) (name1 | node_num)
      continuing with as many lines as G calculations
  [ tip_radius val ]
  da (dax day)
```

Refer to the **parks option for a complete description of these options.

**non_local

Description:
This post increment method provides a "smoothing" of specified material variables in order to constrain the gradient in those variables to be below a given limit. This is an important method in eliminating localization when stiffness drops according to those parameters (i.e. damage or strain softening). This is a "weak" method in that it allows localization during the solution convergence of a given increment.

Let d be the non–local variable ; Δd the local time increment ; $\Delta d_{\rm nl}$ the non–local time increment.

$$\Delta d_{\rm nl}\left(\vec{X}\right) = \int_V \Delta d(\vec{x})\psi(\vec{x}-\vec{X})\,dx$$

$$\psi(\vec{x}) = \frac{1}{\lambda^N \pi^{N/2}} \exp\left(-||x||^2/\lambda^2\right)$$

where $N = 1, 2, 3$ is the space dimension.

$$\int_{-\infty}^{+\infty} \cdots \int_{-\infty}^{+\infty} \frac{1}{\lambda^N \pi^{N/2}} \exp\left(-(x_1^2 + \cdots + x_N^2)/\lambda^2\right)\, dx_1 \ldots dx_N = 1$$

Using the FE method integrals are discretized using Gauss quadrature. So that

$$\Delta d_{\rm nl}(\vec{x}_i) = \sum_j \Delta d(\vec{x}_j)\psi(\vec{x}_j - \vec{x}_i)V_j = M_{ij}\Delta d(\vec{x}_j)$$

and V_j the volume associated to Gauss point j. where x_i, x_j are the positions of the Gauss points (G number of Gauss points).

Due to numerical integration and other limitations (optimizing memory) it is necessary to correct the M matrix which should fulfill the conditions

$$[\mathrm{c}_1]\ \sum_j M_{ij} = 1 \quad \text{and} \quad [\mathrm{c}_2]\ \sum_i M_{ij} = 1$$

Several correction methods can be proposed ($M \to M'$):

`average_line` $M'_{ij} = M_{ij}/\sum_i M_{ij}$. [c$_2$] is not fulfilled.

`average_column` $M'_{ij} = M_{ij}/\sum_j M_{ij}$. [c$_1$] is not fulfilled.

`diagonal_line` $M'_{ii} = M_{ii} + (1 - \sum_i M_{ij})$ and $M'_{ij} = M_{ij}$ for $i \neq j$. [c$_2$] is not fulfilled.

`diagonal_column` $M'_{ii} = M_{ii} + (1 - \sum_j M_{ij})$ and $M'_{ij} = M_{ij}$ for $i \neq j$. [c$_1$] is not fulfilled.

`iterative` Line i is multiply by a scalar l_i and column j is multiply by a scalar c_j. Conditions [c$_1$] and c[c$_2$]can be fulfilled simultaneously. There are $2 \times G$ equations

$$\sum_i l_i c_j M_{ij} = c_j \sum_i l_i M_{ij} = 1$$

$$\sum_j l_i c_j M_{ij} = l_i \sum_j c_j M_{ij} = 1$$

Coefficient can be found iteratively:

$$c_j^{k+1} = \frac{1}{\sum_i l_i^k M_{Ia}} = \frac{1}{M^T l^k | \, j}$$

$$l_i^{k+1} = \frac{1}{\sum_j c_j^k M_{ij}} = \frac{1}{M c^k | \, i}$$

with $l_i^0 = 1$ and $c_j^0 = 1$.

Syntax:
The command syntax is described below, with a few sub-commands.

```
**non_local
 *lambda lam
 *variable var1 ... varN
 *elset elset-name
 *cut_off val
 *normalize type
```

`*lambda` takes a real value for λ

`*variable` list the material variables which are to be the subject of non-local.

`*elset` apply the method to the given elset name.

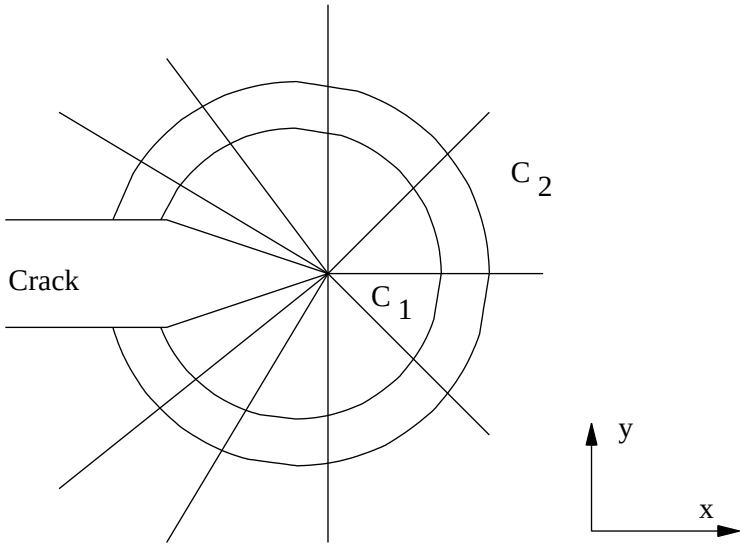
`*cut_off` specify a real value for the cut-off.

`*normalize` replace *type* with one of the keywords described above (`average_line`, etc).

**parks

Description:
The **parks option indicates that the crack tip energy release rate G is to be calculated by the perturbation method (c.f. Parks [Park74]). The method is only available in 2D calculations.

Parks' method advances the crack length a small amount in order to evaluate the energy release per crack length unit. The obtained stress intensity factor taken from G is thus a "global" measure of the mixed mode I and II cracking.



It is necessary to define a translated internal zone (in the interior of the contour C_1). The perturbed zone by the translation is that constituted by the first arrangement of elements about the translated contour (contour between C_1 and C_2). Beyond the contour C_2 nothing will be modified.

The internal translated contour is fabricated from either an element set or a single node defining the crack tip. These geometrical groups must be defined in the *problem*.geof file with the meshing commands described in the leading chapters of this manual. For the case of axisymmetric calculations, it is necessary to define the radius of the crack tip point.

Note:
The results are stored in the sequential formatted file *problem*.parks with one line per increment calculated. Be careful that for for structures exploiting symmetry (one half the crack is modeled) the value of G must be multiplied by a factor of 2, while K must be multiplied by the square root of two.

Syntax:

```
**parks
  perturb (elset | tip) (name1 | node_num)
      continuing with as many lines as G calculations
  [ tip_radius val ]
  da (dax day)
```

name1 the character name of an elset composed of the translation elements.

node_num integer corresponding to the crack tip.

val real value designating the radius of the crack tip. This option only applies to axisymmetric geometries.

da vector describing the virtual advancement of the crack tip.

dax real value for the advancement of the crack along the x axis only. This value may be negative.

day real value for the advancement of the crack along the y axis only. This value may be negative.

Example:

```
***post_increment
    **parks
      perturb elset inter
      da (1.e-2 0.)
      da (1.e-4 0.)
      da (1.e-6 0.)
      da (1.e-8 0.)
      da (1.e-10 0.)
      da (1.e-12 0.)
      da (1.e-16 0.)
      da (1.e-20 0.)
      da (1.e-30 0.)
      da (1.e-100 0.)
```

***pre_problem

Description:
This procedure and its options are executed at the beginning of the problem, before doing anything.

Syntax:

```
***pre_problem
 **first-pre-problem-type
     first-pre-problem-specific options
 **another-first-pre-problem-type
     ...
```

There may be any number of sub-options defining the different pre_problem to treat, and also any number of ***pre_problem instances.

The different pre_problems commands are the subject of the following pages. The following tables are included as a quick-directory to the pre_problems.

General purpose pre_problems:

CODE	DESCRIPTION
**init_z7p_rotations	used to impose a material rotation to a particular elset using a z7p program
**layer_orientation	used to define the stack sequence when a layer element (c3d16l for instance) is used.

**init_z7p_rotations

Description:
The goal is there to compute a material frame (material rotation) for the given elset. For all the integration points of this elset, the positions if each points is given to a t should then return a rotation matrix which will be used as the material for that integration point. This is very useful for composite materials.

Syntax:

```
**init_z7p_rotations
  *elset_name  name_elset
  *script  name_script
```

name_elset character name of the element set. This must be the name of a valid `elset` defined in the geometry file. The rotation will be imposed at every Gauss point in this element set.

name_script The character name of the Zlanguage script which defined the material frame as a function of the gauss point positions.

Example:
The following example apply a rotation to all gauss points of the elset called elbow. The rotation is around the `z` axis, such that the center is `cx=27.` and `cy=10.`.

```
***pre_problem
   **init_z7p_rotations
      *script rotation.z7p
      *elset_name elbow
```

The script called *rotation.z7p* is given as follows :

```
void initialize()
{
  global double cx,cy;

  cx=27.; cy=10.;
}

void apply()
{
// X,Y,Z : position of the current integration point
// to be filled by the end-user : FRAME
        double angle;
        angle = atan((X-cx)/(cy-Y));
        angle = angle*180./pi;
        FRAME.set_euler_angles_3D(0.,0.,angle);
}
```

**layer_orientation

Description:
The goal is there to define the stack sequence when a layer element (c3d16l for instance) is used.

Syntax:

```
**layer_orientation
  *elset  name_elset
  *stack_sequence  number_of_layer
   frame_1
   frame_2
   ...
  [*elset_rotation  frame]
  [*element_rotation_script/  name_script]/
```

name_elset character name of the element set. This must be the name of a valid `elset` defined in the geometry file.

number_of_layer (integer) gives the number of layers of the element defined in *name_elset*. This number must be consistent with the one defined in `***mesh`.

stack_sequence used to indicate the type of stack, Define the rotation from the laboratory coordinate system (in which the material behavior is defined) to the coordinate system of the ply. Use Euler angle (see page 3.156) or a around the y axis.

**elset_rotation frame* These rotations are defined using Euler angles. It gives the rotations to change the coordinate system from the global coordinates to the element coordinates (see example)

**element_rotation_script* name_script/ gives the rotations to change the coordinate system from the global coordinates to the element coordinates using a Zlanguage script (see page 3.202).

Example:
First example :
The following example define a laminate composite (0,+45). The material is defined in the global coordinate system such as the fiber direction is z.

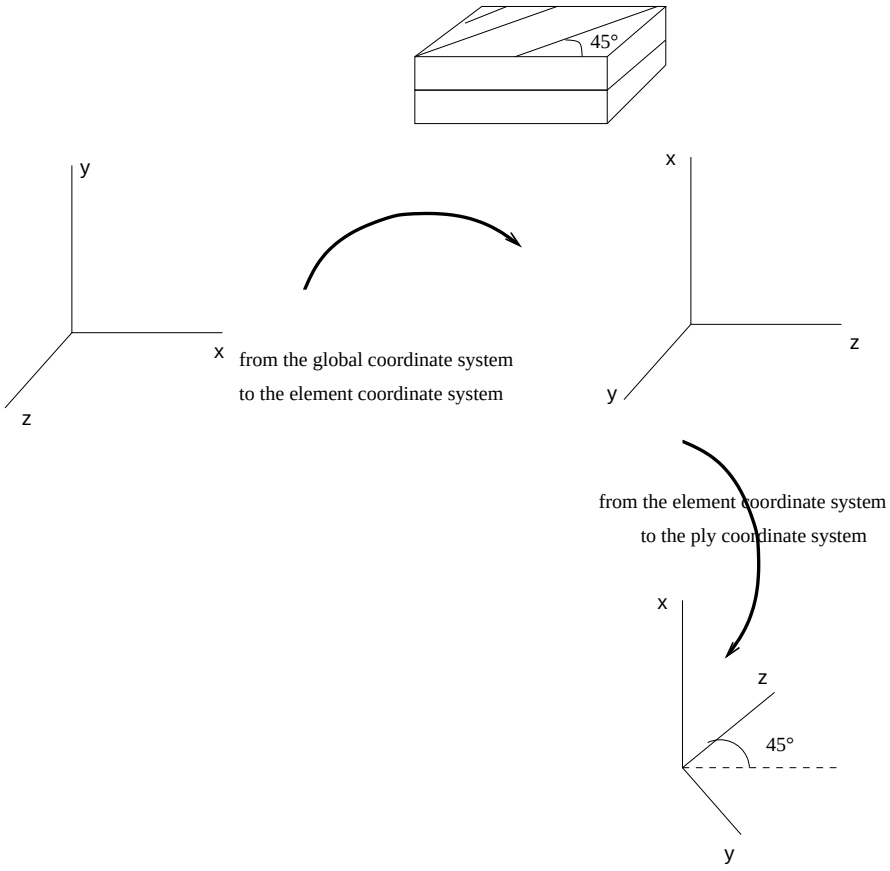
```
***pre_problem
 **layer_orientation
   *elset ALL_ELEMENT
  *stack_sequence y_axis 2
   0. 45.
   *elset_rotation
   0. 0. 0.
```

For a material define such as 1 is the direction of fibers, 2 and 3 are equivalent

```
***behavior linear_elastic
         **elasticity orthotropic
                  y1111 1.63339e+05
                  y2222 1.02212e+04
                  y3333 1.02212e+04
```

Second example :
The next figure shows an example such as the laminate composite is defined by a the (0,+45) stack sequence, and the elset is rotated at 90 around the z axis.

```
***pre_problem
 **layer_orientation
   *elset ALL_ELEMENT
  *stack_sequence y_axis 2
   0. 45.
   *elset_rotation
   90. 0. 0.
```



***random_distribution

Description:
This command creates a random distribution in the structure, which can be used to create spatially randomized parameters such as material coefficients.

Syntax:

```
***random_distribution
 **name name
 **type dimen type
 **law  law
 **load load-file
```

**name gives a character name to the distribution. This user assigned name will be used to reference the random value at a particular point in space.

**type specify *dimen* (real) for the space dimension and a distribution *type*. The later is currently restricted to be `cellular`.

**law *law* define the type of random law to use. *law* should be a keyword defining the type. In the standard distribution, there are the following types available.

CODE	DESCRIPTION
`uniform`	Uniform law $F(x) = x$
`weibull`	Weibull type statistical distribution $F(x) = 1 - \exp\left[-((x-A)/b)^C\right]$
`discrete`	User gives $F(x)$
`normal`	$F(x) = erf(x)$

These laws are described in more detail in the following pages.

**load is used to load the random field from a pre-existing file. This is used for example for verification tests where it is desired that the distribution be the same every time the case is run. One can use this to preserve interesting random distribution cases for more than one run.

... continued

Example:
An example of random distribution being used for randomized coefficients follows.

```
***random_distribution
   **name Erand
   **type 2 cellular
     *start (0. 0.)
     *end   (1. 1.)
```

```
      *cell  (.2 .2)
    **load random.data
  ....

  ***behavior linear_elastic
   **elasticity
       young random Erand
       poisson 0.3
  ***return
```

The random distribution can also be used to initialize material state variables.

```
***random_distribution
  **name eelrand
  **type 2 cellular
    *start (0. 0.)
    *end   (1. 1.)
    *cell  (.2 .2)
  **load random_iv.data
***material
    *file ../MAT/random_iv.mat
    *var_mat_ini eel11 random eelrand
****return
```

***resolution

Description:
This procedure is used to fix the calculation parameters and method of resolution for the global matrix problem $\mathbf{Ku} = \mathbf{R}$. Except for the simplest linear problems, at least some of the procedures sub-options will be required. Non-linear problems are best solved with some substantial customization of these parameters.

Syntax:
Specification of the method of resolution requires a definition of the form:

```
***resolution type
```

which will be followed by a number of options of the ** command level. The possible types of resolution are summarized in the following table:

CODE	DESCRIPTION
newton	resolution of type Newton-Raphson, or modified Newton
bfgs	resolution of type BFGS quasi-Newton
riks	a Riks solution to pass by local instabilities or snap-throughs

In the absence of *type* the default `newton` will be substituted.

This `***resolution` command has a variety of sub-procedures to define load sequencing:

`**sequence` Define loading sequences.

`**cycles` Define cycles of loading sequences.

`**automatic_time` Define automatic time stepping.

`**init_d_dof` Accelerate convergence by pre-initializing the DOF increment.

`**no_symmetrize` Use the non-symmetric solver.

`**max_divergence` Specify the maximum number of divergences allowable before the problem stops.

`**skip_cycles` Use the cycle skip algorithm.

`**use_lumped_mass` Use the lumped mass matrix (only available for ****calcul dynamics)

***resolution newton

Description:
The default algorithm is the `newton` method, which can be used as a full updated Newton-Raphson algorithm, or as a modified Newton-Raphson method depending on the `*algorithm` choice in the sequence definitions.

Note that the finite element method calculates the following residual:

$$\mathbf{R}^i = \mathbf{F}_{ext} - \mathbf{F}^i_{int}$$

where $\mathbf{F}_{ext}$ are terms of the discretized weak form of the problem variational statement due to externally applied forces, $\mathbf{F}^i_{int}$ are the terms due to "internal forces" (e.g. the $\int_V \delta\boldsymbol{\epsilon} : \boldsymbol{\sigma} dV$ virtual work term in mechanics), and $\mathbf{R}^i$ is the residual imbalance at an iteration i due to the non-linearity of the problem. Convergence of a loading step is achieved when a measure of this residual falls below a desired magnitude.

A truncated Taylor series is used to find the adjustment to the problem variables (degrees of freedom):

$$\mathbf{R}^{i+1}\left(\mathbf{q}^i + \Delta\mathbf{q}\right) = \mathbf{R}^i\left(\mathbf{q}^i\right) + \mathbf{K}\left(\mathbf{q}^i\right)\Delta\mathbf{q}$$

$$\Delta\mathbf{q}^{i+1} = -\mathbf{K}^{-1}\mathbf{R}^i \qquad \mathbf{q}^{i+1} = \mathbf{q}^i + \Delta\mathbf{q}^{i+1}$$

Where $\mathbf{K}$ is a stiffness matrix. Note that The stiffness can be an approximate measure of the real stiffness at a give time, and the solution can still be found. Typically however, any approximations to the stiffness other than the real algorithmic consistent tangent will result in increased iterations, and a reduction in the convergence "radius."

Syntax:
The Newton-Raphson method provides a quadratic convergence: the residual of the problem to be solved can not *theoretically* increase. Because real problems usually do not fall into the Newton-Raphson assumptions, one may observe that the residual increases during iterations. In this case, use the `**line_search` option to activate the line search procedure. The idea behind it is that if the residual increases, it means that the magnitude of the current Newton-Raphson correction is too large: a search is done along the descent direction to find the largest magnitude so that the residual decreases (one show that this optimum value always exists).

```
***resolution newton
  **line_search
```

***resolution bfgs

Description:
The BFGS algorithm is a quasi-Newton method of matrix updating which can be used to efficiently solve large non-linear problems. The algorithm consists of two methods of convergence acceleration [Matt79]. The first is the updating process which corrects an approximation of the inverted non-linear tangent with the history of trail solutions and their errors. The second is a relaxation of the displacement increment to be compatible with the "direction" of the solution. The algorithm requires an initial calculation of the matrix inverse. Further iterations with the BFGS algorithm do *not* however require additional inversions of the rigidity matrix, making the method very efficient for large non-linear structures.

Given an inverted matrix $\mathbf{K}_{ini}^{-1}$ which could have been calculated at the beginning of an increment, or just one time at the start of the problem (inverted elastic behavior stiffness), one updates the inverse through the course of iterations as follows:

$$\mathbf{K}_N^{-1} = \left[\prod_{j=1}^{m} \mathbf{A}_j\right] \mathbf{K}_{ini}^{-1} \left[\prod_{j=1}^{m} \mathbf{A}_j^T\right]$$

$$\mathbf{A}_i = \mathbf{1} + \mathbf{w}\mathbf{v}^T$$

which shows that the storage requirements are simply that of two vectors of size N (the number of degrees of freedom in the problem). The vectors $\mathbf{v}$ and $\mathbf{w}$ are calculated using the changes in displacement and changes in residual between updates.

Syntax:
The BFGS algorithm allows the following syntax to adjust the solution process:

```
***resolution bfgs [ iter ][ no_optimize ]
```

The parameter *iter* is an integer specifying the number of iterations which will performed before the BFGS updates begin. The optional token `no_optimize` indicates that the optimization (line search) procedure is not desired. The line search adds evaluations of the material law per iteration which may be very expensive for complex material behaviors.
Caution

The contact solutions are not compatible with the BFGS algorithm. Also, the use of BFGS requires matrix updating of the types `eeeee`, `p1p1p1` or `p1p2p2`. Use of `p1p1p1` requires the number of iterations *iter* to be greater than one, and `p1p2p2` requires *iter* to be greater than two.

***resolution riks

Description:
The `riks` algorithm is a method for which local instabilities can be passed in a quasi-static analysis, such as a bucking situation of a roof collapse. Note that these "snap-though" type problems are in reality dynamic events which can also, perhaps more appropriately, be solved using implicit dynamics.

The method assumes proportional loading as well. So if the load at a point is not merely scaled through time (without change in direction) the algorithm can be used successfully.

Syntax:
The syntax for Riks allows a number of keywords and keyword parameter pairs to be entered after the `***resolution` command.

```
***resolution riks option-keys
```

Example:

```
***resolution riks
    opt    4     % optimal number of iteration
    max    30    % maximum number of iteration
    div    2.    % divergence division factor
    maxDs  0.3   %
    Plus         % Keep going positive..
```

**automatic_time

Description:
This procedure sets automatic calculation of the time stepping (increments) within a sequence. The method will attempt to optimize time steps based on user adjustable convergence and accuracy parameters, and also provide divergence control. The optimization parameters are currently limited to the following:

- optimal number of iterations per increment.
- maximum variation in material variables (e.g. max allowable plastic flow).

The step calculation is made by first examining the integrated material variables on which control has been set. The estimation of the next step size will then be:

$$\Delta t_{i+1} = \Delta t_i \frac{\Delta v}{\Delta v_i}$$

with Δt_i the last increment of time (just solved), Δt_{i+1} the new estimated time step, Δv_i the internal variable increment just achieved, and Δv the desired internal variable increment. This value is calculated for each of the controlled internal variables. The new time increment to be used for the next increment will be the minimum of these Δt_{i+1} values. It is remarked that control may be placed on any of the internal variables existing in the problem. This includes variables defined for only a portion of the structure in a multi-material problem. The problem variables may be confirmed at run-time with the `-v` command line switch or the `**verbose` output command.

If the time stepping is also controlled by the number of desired iterations, an additional estimation of time step is made as follows:

$$\Delta t_{i+1} = \Delta t_i \sqrt{\frac{nc}{ni}}$$

with ni the number of iterations for the last convergence, and nc the desired number of iterations. All other variables are as described above. The next time increment will be taken as the minimum value calculated from the internal variable control and the iteration control.

As the time step is determined from the previous time step, it remains to define the initial time step in a sequence. This initial step may be given using two different methods. 1) The default method uses the number of increments in the sequence as defined under the `**sequence` command (the `*increment` sub-command). This method gives the standard $\Delta t_0 = \Delta t_{sequence}/\#$ increments. 2) Explicitly enter the first time step with the `first_dtime` sub-command described below.

Divergence control is available when the maximum number of allowable iterations is passed within any of the defined increments. The sub-command `*divergence` will allow specifying a dividing factor used to reduce the time step. The maximum number of successive divergences is also adjustable.

If the increment of all controlled internal variables is less than the desired amount, or if the convergence is achieved in less than the desired number of iterations, the time step will be increased by a user defined factor (`*security`) in order to accelerate the solution.

Syntax:
The following syntax is used to define the automatic time-stepping in a calculation:

```
**automatic_time [ type ] ( var1 val1 [ var2 val2... ] | global iter | mandatory)
[  *divergence  div [ times  ] ]
[  *security    ratio ]
[  *max_dtime   max_dtime  [  max_dtime2 ... ] ]
[  *min_dtime   min_dtime  [  min_dtime2 ... ] ]
[  *first_dtime first_dtime  [  first_dtime2 ... ] ]
```

CODE	DESCRIPTION
standard	Automatic time assuming that the loading path is basically continuous. If mandatory is specified the indicated incremental value is considered to be mandatory: if the increment of the specified variable is too large, one considers that a divergence occurred, and the automatic time will reduce the time step.
by_sequence	The algorithm is initialized to the user's increment input for each sequence; recommended for cyclic loading for example
divergence_control	Automatic time only to control divergence; the input increments will be used except when a divergence occurs

*divergence *div [times]* Divergence control taking a real value for the devising factor *div* and an integer for the number of allowable divergences *times*. The default value for *times* is one.

*first_dtime *first_dtime* Set the first time step in *all* sequences with the real value *first_dtime*. This option is advised only for monotonic or other simple loading paths. A list of *first_dtimes* may also be given, corresponding to the first dtime of each sequence. The last is implicitly repeated, if necessary.

*max_dtime *max_dtime* Set the maximum allowable time step to the real value *max_dtime*. In the case of overestimating the number of iterations or change in the internal variables, this option will limit the acceleration of solution.

*min_dtime *min_dtime* Set the minimum time step to the real value *min_dtime*. In the event where the number of iterations or the change in internal variables are grossly underestimated, this option will limit the time step to a minimum value. It prevents extreme reduction of the time step which will effectively limit the calculation from advancing further.

*security *ratio* Defines the multiplicative factor used to increase the time step in the event of good convergence or small changes of the internal variables. The parameter *ratio* is a real value specifying the factor to be used.

If automatic_time exhibits strange behavior with small or large time steps, try to use ***dimension instructions 3.118.

Example:
A very frequent use of this command follows:

```
**automatic_time global 3
 *divergence  2.0  10
 *security    1.2
```

Frequently it is a good idea to keep the allowable iterations small (**sequence *iteration command), and let the automatic time reduce the step size more readily.

Some other variations to control accuracy based on changes in material variables follow.

```
**automatic_time evcum 0.001 eel11 0.001 global 5

**automatic_time f 0.01 p 0.01
    *divergence  2.0   50
    *security    1.2
    *max_dtime   0.03
    *min_dtime   0.001
```

**init_d_dof

Description:
This procedure sets the initial increment DOF values for iteration $i+1$ as a function of the previous solution i:

$$\Delta \mathrm{DOF}_{i+1} = \mathrm{ratio} \times \frac{\Delta t_{i+1}}{\Delta t_i} \times \Delta \mathrm{DOF}_i$$

The objective of this method is to accelerate convergence during relatively stable loading paths. The assumption is that the next solution will not vary greatly from the previous one. This method is therefore dangerous to convergence if the loading changes rapidly, such as a newly unloaded section of the structure.

Syntax:

```
**init_d_dof [ ratio ][ sequence ]
```

`ratio` is a real value acting as a scaling factor, as shown above. The default value is `1.0`.

`sequence` indicates that the initialization is only applied within each sequence, based on the first solution of that sequence. In absence of this option the initialization is carried out for the entire calculation, even across sequence boundaries.

Example:

```
**init_d_dof 0.9 sequence
```

**max_divergence

Description:
This procedure allows a given value to be set for the maximum allowable ratio of two successive convergence ratios. The calculation will be terminated if this critical ratio is exceeded.

Syntax:

```
**max_divergence value % value is a double
```

Example:

```
**max_divergence 1.5
```

**sequence

Description:

This command writes the sequences or blocks of time steps for the problem[15]. Solution requires specification of discrete times for which the governing equations are to be evaluated. The program allows an unlimited number of **sequence or **cycles (see next command page) instances in order to create complex loading histories. There is really no distinction between the sequence and the cycle commands; the cycle command automatically generates a number of repeated sequence groups to form cycles.

As discussed in the introduction of ****calcul (page 3.7), the sequences are broken up into increments which are the most fundamental unit of time stepping.

Syntax:

Sequences are created as necessary according to the sub-options of the **sequence command. Normally, the sequences are defined by end times or incremental times of the sequences. The syntax for these definitions is the following:

```
  **sequence  [ N ]
[  *algorithm   algo1  [ algo2, algoN ] ]
[  *increment   inc1, inc2, ..., incN ]
[  *iteration   iter1, iter2, ..., iterN ]
[  *limit_dof   vari, vali1,...valin [ , varj, valj1,..., valjN ] ]
[  *time        time1, time2,..., timeN ]
[  *dtime       dtime1, dtime2,..., dtimeN ]
[  *ratio       [ absolu ] ratio1, ratio2, ..., ratioN  ]
```

The integer value *N* defines the number of sequences which will be run, even if there are more defined in the sub-commands. This allows the run-time of a problem to be limited quickly, without deleting or commenting information which may be useful later on. In the absence of this value, the number of sequences will be determined based on the number of time or dtime values input.

All the other options are used for specifying particular parameters which correspond to the sequences. If the number of values given after an option is less than the number of sequences of the current block (the last **sequence command), the last value given will be repeated for all the subsequent sequences. If one or several of the sub commands are absent, the default value will apply to all the sequences defined.

*algorithm String values substituted for *algo1* to *algoN* are the algorithm solution methods for each segment of the loading. The values for these methods are summarized:

[15] Sequences are extensively discussed in the examples manual

CODE	DESCRIPTION
eeeeee	Newton "elastic" method. Matrix is not re-calculated during iterations
EEEEEE	Linear method. No iterations, matrix calculated only once (once after time step or Newmark coefficients changes in dynamics). The computation of internal forces is made through the K.U multiplication which is much faster than the classical element integration procedure. Only nodal fields are post-treated, no element fields are available. This can be useful to treat linear dynamics problems in a very fast way.
p1p1p1	Modified Newton algorithm with the matrix calculated and inverted the first and second iterations
p1p2p2	Modified Newton algorithm with the matrix calculated and inverted the first, second, and third iterations
p1p2p3	Newton-Raphson algorithm where the matrix is always updated during iterations

`*increment` The integer values *inc1, inc2, ..., incN* are used to define the number of increments in corresponding sequences. In the absence of this option, all sequences are calculated in a single loading increment.

`*iteration` *iter1, ..., iterN*. Integer values used to limit the maximum number of iterations in a given increment, valid during a sequence. The default number of iterations is 10. These values may be used to adjust the divergence control, or limit the CPU consumption of a poorly converging problem.

`*limit_dof` *vari, vali1,...valin*, ... This option takes a DOF character name (`U1`, `U2`, etc) and a list of real values corresponding the given segments. Any number of DOF name / list of limiting values may be input sequentially to put limits on the different DOFs of the problem. Real numbers input are limiting values for any iteration of the degree of freedom. In the course of iterations, we will have available $\Delta\mathbf{u}_i$ at iteration $i+1$ from the solution of the linear system, $d\mathbf{u}_i = (\mathbf{K}_i)^{-1}\,\mathbf{R}_i$. In place of the standard updating of the DOF values, $\Delta\mathbf{u}_{i+1} = \Delta\mathbf{u}_i + d\mathbf{u}_i$ we have $\Delta\mathbf{u}_{i+1} = \Delta\mathbf{u}_i + \beta d\mathbf{u}_i$ where β is a limiting factor:

$$\beta = \min\left(\frac{\text{limitation value}}{\max_{DOF}(d\mathbf{u}_i)}, \beta\right)$$

with $0 < \beta < 1$.

This option allows the calculation to pass points where the algorithm would predict instability, at the cost of increased number of iterations. Excessive trial delta values in the DOFs may occur when the problem tangent matrix displays softening behavior.

`*time` *time1, time2,..., timeN* specifies the absolute times at the end of the sequences. There are as many values as the number of sequences. The first sequence of the problem is at

$t = 0$, or the last time given in the previous **sequence command. This initial value must be less than the first time *time1* given here. In the absence of a **time* option, the times will be assigned to the sequence numbers.

*dtime *dtime1, dtime2,..., dtimeN* specifies the increment of time taken over a sequence. This method of specifying the sequence times is useful if a "time shiftable" block of sequences is desired. An example is a block of loadings to be subjected to different pre-loadings, or loading after a series of cycles, where additional computations would be necessary to assure that the times are well-posed. Options *dtime and *time are mutually exclusive.

*ratio [absolu] *ratio1, ratio2, ..., ratioN* This option is for specifying the maximum global residual for convergence in a sequence. The real values *ratio1*, etc correspond to the sequences defined. A single given value will be repeated for all the sequences.

The residual calculation will be calculated according to the following formula:

$$\text{ratio} = \frac{\sqrt{\sum_{\text{iddl}} \left(R^{\text{ext}}_{\text{iddl}} - R^{\text{int}}_{\text{iddl}}\right)^2}}{\sqrt{\sum_{\text{iddl}} {R^{\text{ext}}}^2}} \qquad \times 0.01 \text{ ratio}$$

$$\text{ratio} = \sqrt{\sum_{\text{iddl}} \left(R^{\text{ext}}_{\text{iddl}} - R^{\text{int}}_{\text{iddl}}\right)^2} \quad \text{ratio absolu}$$

where the option absolu indicates non-normalized ratio calculation.

Proposition de modif de la doc de *ratio

*ratio [norme_force|absolu|automatic] *ratio1, ratio2, ..., ratioN* This option is for specifying a global convergence criterion in a sequence. The real values *ratio1*, etc correspond to the sequences defined. A single given value will be repeated for all the sequences.

The norme_force keyword (default) means that convergence is achieved if:

$$\frac{\left\|R^{\text{int}} - R^{\text{ext}}\right\|_2}{\left\|R^{\text{ext}}\right\|_2} < 0.01 \times \text{ratio}$$

where R^{int} and R^{ext} are the internal and external nodal forces vectors. $R^{\text{int}} - R^{\text{ext}}$ is therefore the equilibrium residual. The absolute keyword means that convergence is achieved if:

$$\left\|R^{\text{int}} - R^{\text{ext}}\right\|_\infty < \text{ratio}$$

The automatic keyword means that norme_force is selected excepted if $||R^{\text{ext}}|| = 0$ in which case the absolute criterion is taken.

There is an extended syntax for ratio which is of the form :

*ratio (crit_type1 dof_type1:v1 crit_type2 dof_type2:v2 ...)

This extended syntax is useful for multifield problems. The global dof vector is split into sub-vectors corresponding to each specified dof_type. The specified criterion (crit_type can be norme_force, absolu or automatic) is then applied on each sub-vector and global convergence is achieved if all criteria are met. dof_type can be any dof type such as U1, U2, TP, ... or any class of dof (i.e. dof names with the same prefix) such as U which concatenates U1, U2 and U3 in the same sub-vector.

```
****calcul
 ***resolution
  **sequence
```

Example:

```
  **sequence
   *time          1.0    5.0
   *increment     10
   *iteration     20
   *ratio absolu  1.0
   *algorithm     p1p2p3
   *limit_dof     U1 .1 .1  U2 .02 .03
```

In this example, `d` is a dof coming from a multifield element.

```
***resolution
 **sequence
  *time 1.
  *increment 20
  *iteration 5
  *algorithm p1p2p3
  *ratio (automatic U:1.e-3 absolu d:1.e-6)

 %
 % from ab429.inp making 3 sequences
 %
 **sequence
   *time       3.   33.  50.
   *ratio      1.    3.   3.
   *iteration  50   30   30
   *increment  10   15   34
   *algorithm  p1p2p3
   *limit_dof
      U1   0.2   0.5   0.5
      U2   0.2   0.5   0.5
      PR  10.0  10.0  10.0
```

**cycles

Description:
The cycles option is essentially the same as the **sequence option except that it repeats the block of sequences a given number of times. Normally, use of the **cycles option will be employed after a series of pre-loading sequences, in order to attain the initial loading state of the cycles. Therefore, cyclic loadings will usually require a **sequence definition for the pre-load, followed by a **cycles definition for the cycles themselves. Unless the end segment of the cycles will return to zero loading, additional **segment definitions may be given to relax the structure to an unloaded state.

An example loading is a pre-load in tension, followed by tension-torsion cycles, and finally a relaxation of the initial tensile load. Many more cases may be imagined.

As the calculation time scale is the principal measure of the solution, it is important to understand the method of time scale construction from a cycles definition. The times given for this option are always relative to the beginning of the cycle. This defines a local cycle-based time scale, which may be repeated.

Syntax:
The syntax for cyclic loading is the following:

```
  **cycles  [ N ]
[  *algorithm   algo1 [ algo2, algoN] ]
[  *increment   inc1, inc2, ..., incN]
[  *iteration   iter1, iter2, ..., iterN]
[  *limit_dof   vari, vali1,...valin [ , varj, valj1,..., valjN] ]
[  *time        time1, time2,..., timeN]
[  *dtime       dtime1, dtime2,..., dtimeN]
[  *ratio       [ absolu ] ratio1, ratio2, ..., ratioN]
```

the integer N is the number of cycles to be created.

Except for the time scale alteration for the cyclic sequences defined with the *time or *dtime options, all sub-commands for this command are equivalent to the **sequence command.

Example:
An example loading gives 5 cycles with the waveform 10s-300s-10s after a pre-load of 1s.

```
**sequence
  *time      1.
**cycles 5
  *dtime     10. 300. 10.
  *increment 5   10   1
  *algorithm p1p2p3
```

This example assumes the first one second loading segment will be a linear solution. The first segment of the first cycle (from 1 second to 11 seconds on the "global" clock) may be a loading with non-linear behavior, so 5 increments are given. The following rest time has 10 increments, also assuming non-linear behavior. The final segment from times 311 seconds to 321 seconds is assumed linear and will be solved in one increment. The same increments will

```
****calcul
 ***resolution
  **cycles
```

then be repeated 4 more times. A corresponding loading waveform is given in the `***table` command.

**skip_cycles

Description:
The skip cycles command is used for cyclic calculations to extrapolate the problem variables and thereby reduce the number of cycles actually calculated. Because there can be many time steps per cycle, this command has the possibility of greatly reducing the calculation time.

Syntax:

```
**skip_cycles [ type ]
 *precision val
 *during_cycles cyc-st  cyc-end
 *use_sequence seq-num
 *extrapolate_from cyc1  cyc2  cyc3
 *check_with_component var-name
```

Example:

For example we give a viscoplastic problem with 300 cycles and several sequences per cycle. The option `*use_sequence` was used to specify the beginning of sequence 1 because the material response is pretty much elastic there throughout the cycling. The response changes rapidly before cycle 4, so `*during_cycles` was used to start the cycle skip after that point.

```
**cycles 300
 *time      15.0 30. 40.0 50.0
 *increment 20   4   5    4
 *ratio absolu 1.e-3
 *algorithm p1p2p3
**skip_cycles polynomial_extrapolation
 *precision     0.2
 *trim          0.8
 *order         2
 *error_skip    2
 *during_cycles 4   290
 *use_sequence 1
 *extrapolate_from 1 2 3
 *check_with_component evcum
```

**use_lumped_mass

Description:
This procedure replaces the mass matrix by the lumped one currently used with explicit algorithm (see ****calcul mechanical_explicit in 3.14). Only available for ****calcul dynamics.

***restart

Description:
Manages the restart of an interrupted or unfinished calculation. The program automatically saves a restart file (filename *prob*`.rst`) at the end of each increment. If special restart saves are desired at different stages of the calculation, use the `***make_restart_file` option. Saving a calculation's `.rst` file may be used to repeatedly calculate a second part of a calculation without repeating the initial portion. The option may also be used to change the parameters of the calculation at intermediate points in the time-scale, although with significant user work.

Syntax:

```
 ***restart
[ **file name-of-restart-file ]
```

The default restart file name is *prob*`.rst` ; you may specify a different name with the `**file` option.

Drastically changing the boundary conditions or other parameters in the `.inp` file and re-starting the calculation may lead to poor results.

***auto_restart

Description:
This option allows automatic restart of a problem, if a restart file exists. The only difference with `***restart` is that no error message is output if no restart file is available.

***shell

Description:

This command launches a sub-shell to execute external programs.

Syntax:

```
***shell
    shell-command
***shell
[ **frequency frequency ]
[ **init_problem ]
     a-first-shell-command
     another-shell-command
[ **start_increment ]
     a-first-shell-command
     another-shell-command
...
```

In the first form, the shell commands are executed when the *.inp* file is read. In the second form, the shell commands are executed at the given entry points in the problem, with the optional frequency.

Currently, the following entry points are available (see the Problem Component description in the devel manual):

- `init_problem`
- `start_increment`
- `end_increment`
- `start_iteration`
- `end_iteration`
- `manage_restart`
- `mesh_changed`
- `end_problem`

***table

Description:
The procedure ***table** indicates a loading table definition in time. The table will be available in the other commands of the **.inp** file such as the boundary conditions or the external parameter definitions. Loading tables are currently only defined as a series of time-magnitude points which are linearly interpolated to intermediate times.

The tables are named with arbitrary, user-defined character strings which are used in the syntax of other commands to indicate which table to use. There are no conditions on the number or size of the tables.

Syntax:
Loading tables are assembled through the use of the following sub-commands:

```
***table
 **name    name
  *time    t_1 ... t_n
  *value   v_1 ... v_n
 **cycle   name  t_ini   t_end
  *time    t_1 ... t_n
  *value   v_1 ... v_n
**function  name FUNCTION
**file      name file_name ctime cvalue
```

**name Specifies that a simple table is to be created with name as given directly following the **name keyword. A table defined as such is assembled with the *time and *value options.

*time indicates that a list of real values will follow giving the individual time points of the table. These time values are in absolute time measured from the beginning of the problem. The current version of the code requires in most applications that the table be able to give a value for all valid times in the problem. This is noted to begin with the initial time $t = 0$. Most tables will therefore be defined starting with this initial point.

*dtime This option serves the same function as the *time command, but is defined in incremental form. The option is useful for complex loadings where it is easier to think in incremental form, or as the only way to currently model step loading (using segments of $\Delta t = 0$).

*value This option indicates that a series of real numbers will be given to define the table values. It is necessary that there be exactly one value for each time specified with the *time command. As the time definitions are given from the beginning of the problem, there must be a corresponding initial value for time $t = 0$. This value will normally be zero in the case of boundary condition loading, as the problem may not begin in a pre-deformed state through standard boundary conditions (use a restart procedure with the pre-deformed structure stored in the restart file). If one table is succeeded by another, it is necessary to ensure the continuity from one table to another.

**cycle *name* t_{ini} t_{end} In order to generate cyclic loadings it is convenient to simply specify a load cycle to be repeated a number of times. This option does exactly that with the initial time for the cycles given by t_{ini} (a real value). The cycles will continue repeated until the time t_{end} (real) regardless if that time is at the end point or an intermediate point in the cycle. The cycle is defined as a simple table above using the *time and *value options, only that the time will be in a local cycle scale. The start of the cycle (first at time t_{ini}) is at the local time zero. The zero time point must therefore *always* be given for the cycle table format. In contrast with the standard table, the corresponding initial value is not usually zero, as some pre-loading may have been applied before the cycles begin. This command is examined more fully in the examples below.

**function This allows to generate tables defined by functions depending on time. *name* specifies the table name followed by a FUNCTION object. The variable time MUST be the only argument of the function.

**file This allows to generate tables defined by a file (column format). *name* specifies the table name. *file_name* specifies the file name. *ctime* specifies the column of *file_name* representing the time and *cvalue* the column of *file_name* representing the table value (the first column is numbered 1).

Example:

1. Suppose that a U2 displacement is to be applied to the top surface of a structure. This surface will have a node set defined by the user in the geometry description with the name top. The loading profile is to be:

at $t = 1$, $u = 10.$
at $t = 5$, $u = -10.$
at $t > 5$, $u = -10.$

On a face set named inter a pressure is also to be applied. The pressure value will be zero until $t = 1$, after which there is a ramp loading in the pressure until $t = 7$ at which point the pressure is 100.

The loading sequences will correspond to the points of load change:

```
**sequence
  *time 1. 5. 7.
```

Declaration of the displacement and pressure boundary conditions are written:

```
 ***bc
   **impose_nodal_dof
       haut U2 1. tab1
   **pressure inter 100. tab2
```

Note the two different table names tab1 and tab2 because the loading waveforms are different. The table definitions may be written as:

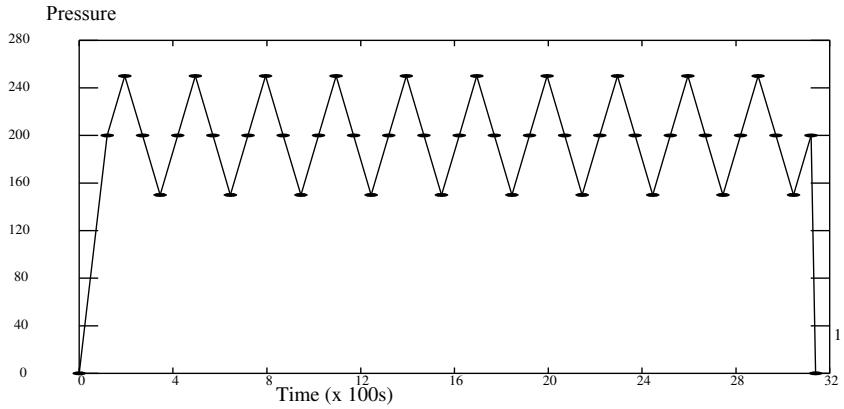
```
                                                    ****calcul
                                                     ***table
```

```
***table
  **name tab1
    *time  0.  1.   5.   7.
    *value 0. 10. -10. -10.
  **name tab2
    *time  0. 1. 7.
    *value 0. 0. 1.
```

2. Cyclic loading is given as a second example. This case represents the calculation of a structure under a pre-loaded pressure of 200 MPa followed by a cycling of the pressure about the mean value of 200 MPa with a cyclic magnitude of 100 MPa. After 10 cycles the pressure is returned to zero to observe the residual stress field.

```
***resolution
   **sequence
    *time       120.
    *increment  5
   **cycles 10
    *dtime      75. 75. 75. 75.
    *increment  10
    *algorithm  p1p2p3
   **sequence
    *dtime      20.
    *increment  5
  ***bc
   **pressure
      top 1.0 tab1 cycs tab2

  ***table
   **name tab1
    *time      0.0  120.
    *value     0.0  200.
   **cycle cycs 120. 3120.
    *time      0.0   75. 150. 225. 300.
    *value   200.0  250. 200. 150. 200.
   **name tab2
    *time   3120.0  3140.
    *value   200.0    0.
```

```
****calcul
 ***table
```



***function

Description:
This command is similar to `***table` in the preceding section, but defines the magnitude as a function of time (see page 6.2 for a discussion of functions).

The `***function` command can only be defined in terms of the variable `time`.

Syntax:
The syntax is the following:

`***function` *name* *func_def*;

Note that semi-colon is **required**. The name has the same significance as in the `***table` command. Each function requires its own `***function` declaration.

Example:

```
***function  xtab cos(time);
***function  ytab sin(time);

***function  funny time + 3.0*exp(-8.2*time);

***function  constant 1.0;
```

The last example will give a warning, but works.

***specials

Description:
The command ***specials allows one to specify certain special boundary conditions. Currently, the command may only be used to declare mesh conditions which exist in the mesh, and is thus could be thought of as a "mesh modifier".

Syntax:

```
   ***specials
[  **mesh     ]
[   *specify  ]
```

**mesh is used to specify special conditions which exist in the mesh.

*specify Introduces the specified parameters to the mesh. The only option currently available for this command is X0_Y0_for_E2_5 which fixes the origin of moments applied to 2.5D elements (generalized plane strain). The moments are applied throughout the thickness of an elset. The syntax requires an element set name follow the keyword, and also real values for the center on the axes 1 and 2. This option is associated to the BC command **impose_elset_dof.

Example:

```
***specials
 **mesh
  *specify X0_Y0_for_E2_5 eprouvette 1. 1.
```

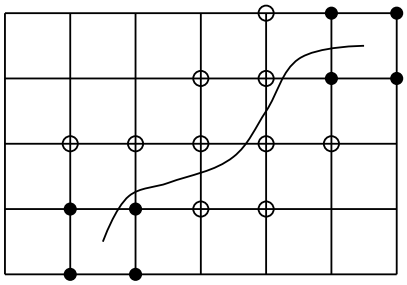
***xfem_crack_mode

Description:
This command allows the use of XFEM enriched elements, in order to model the influence of a discontinuity (crack) on mechanical fields, without the need to explicitly introduce this crack in the FE mesh. Note that the graphical interactive **Zxfem** script can be used to handle Xfem models, and automatically generate input data for this command.

In this context the crack is defined by 2 levelsets, ϕ (signed distance to the crack plane) and ψ (signed distance to the front of the projection on the crack plane). The `**compute_predefined_levelset` mesher command can be used to generate such levelsets for basic crack geometries (see page 2.39. Let I be a set containing all nodes in the FE mesh, we then define 2 subsets J and K of I (see figure below):

- $J \subset I$ correspond to nodes in elements E completely cut by by the crack (ϕ is changing sign on E, but sign of ϕ is constant),
- $K \subset I$ correspond to nodes in elements that include the crack front (both ϕ and psi are changing sign on E).

  - J set: nodes $\in$ elements cut by the crack
  - K set: nodes $\in$ elements containing the crack front



Enriched shape functions are then defined in the following way:

$$u^E(x) \;=\; \sum_{i \in I \cap E} u_i \;\phi_i(x) \;+\; \sum_{i \in L \cap E} a_i \;\phi_i(x)\; H_i(x) \;+\; \sum_{i \in K \cap E} \phi_i(x) \left[\sum_{j=1}^{4} b_i^j \; F_i^j(x) \right]$$

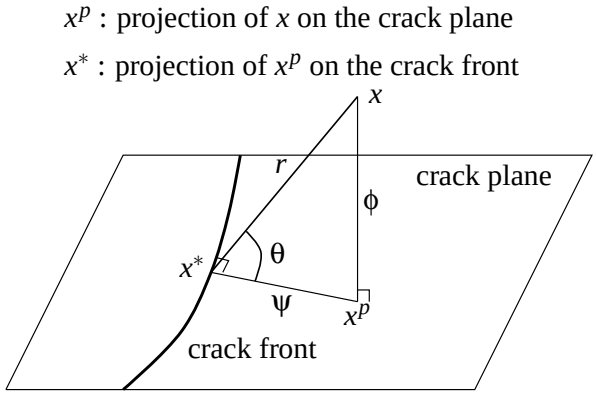
where:

- $\phi_i(x)$ denote the shape functions of classical (ie. not enriched) finite elements and u_i the conventional displacement degrees of freedom ($i = 1, N_E$, where N_E is the number of nodes of element E)
- $H_i(x)$ are shape functions taking a value of 1 on one side of the crack, and -1 on the other. They are used to model the discontinuity of displacement fields across the crack. The corresponding a_i degrees of freedom are called `Xh` in results files.
- the $F_i^j(x)$ $(j = 1, ..., 4)$ shape functions account for classical elastic analytical solutions around a crack tip:

$$(F^1, F^2, F^3, F^4) =$$

$$\left(\sqrt{(r)}\ sin\left(\frac{\theta}{2}\right)\ ,\ \sqrt{(r)}\ cos\left(\frac{\theta}{2}\right)\ ,\ \sqrt{(r)}\ sin\left(\frac{\theta}{2}\right)\ sin(\theta)\ ,\ \sqrt{(r)}\ cos\left(\frac{\theta}{2}\right)\ sin(\theta)\right)$$

where (r, θ) are polar coordinates defined around the crack front as defined on the next figure:



- the $b_i^j (j = 1; 4)$ degrees of freedom associated to the F_i^j shape functions will be called `Xa`, `Xb`, `Xc`, `Xd` in results files.

Note that the basic enrichment scheme described above is the so-called *topological enrichment* mode. A major problem associated with this mode, is that the influence of the refined shape functions used for nodes in the K set ($\sqrt{r}$ shape functions), is decreasing when mesh size near the crack front is refined, which has a negative effect on the rate of convergence. If this is indeed a problem, *geometrical enrichment* can be used, where all nodes within a target user-defined distance of the crack front are automatically added to the K set, and are given full $\sqrt{r}$ type of enrichment.

In the current Zébulon release (8.5), some restrictions on the use of this method still exist, that will be removed in the next versions:

- only 3D linear elements are currently officially supported,
- extensive capabilities do exist to use this scheme within the context of crack propagation (with automatic remeshing if needed), but those options are still under development and will not be addressed here.

Syntax:
The syntax is as follows:

```
[ ***xfem_gtheta ]
  ***xfem_crack_mode
  [ **elset  ename ]
    **discontinuity 3d_levelset
     ( no_option |
       (
         [ *psi_file fpsi ]
         [ *phi_file fphi ]
         [ *fit_to_vertice eps ]
```

```
      )
    )
  [ **geometrical_radius  rad ]
  [  *set_mpc ]
```

where:

- keyword `***xfem_gtheta` is mandatory for connection with the `***compute_G_by_gth` command (computation of 3D stress intensity factors by means of the $G-\theta$ method 3.92).
- optional command `**elset` *ename* may be used to specify the name of an elset containing candidate elements of the enrichment process. By default, all elements in the FE mesh are concerned, and are indeed enriched when cut by the crack or located within the geometrical radius definition.
- commands `*phi_file`, `*psi_file` allow to specify the name of ASCII files defining the ϕ, ψ levelsets on nodes of the FE mesh (see 2.39 for the mesher command used to create those files). Default names for those files are "`phi.dat`", "`psi.dat`".
- the optional command `*fit_to_vertices` may be used to define the critical distance value *eps*, that controls if the crack does cut an element (ie. the element needs to be split), or simply pass through one of its corner nodes (vertice). Note that *eps* is a *relative* value, scaled against the size of the element edge cut by the discontinuity (default value is *eps*=0.05, ie. 5% of the length of the corresponding element edge).
- command `**geometrical_radius` specifies the size *rad* of the domain used for geometrical enrichment: int his case, all elements with nodes within a distance of less than *rad* of the crack front will be enriched with $\sqrt{r}$ shape functions. This option enhance the rate of convergence of the xfem method (mesh size sensitivity), but should be used with caution since the size of the problem is strongly increased (12 additional dof by nodes concerned in 3D). Default mode is topological enrichment only (*rad*=0).
- when geometrical enrichment is activated, the optional `*set_mpc` may be used to automatically add linear relationship between the geometrically enriched zone and the rest of the mesh, in order to insure continuity of displacements, and improve the rate of convergence.

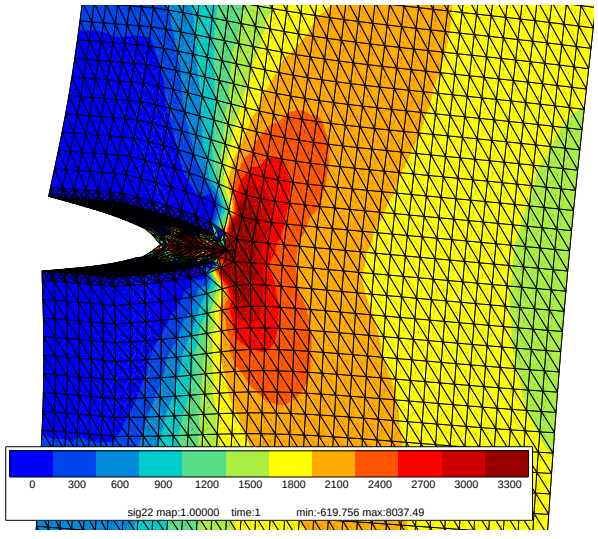
Example:

```
% Basic example : calculation with xfem enriched elements
****calcul
 % activation of the Z8 output format mandatory
 % for Zmaster handling of XFEM results
 ***global_parameter
  Solver.OutputFormat     Z8
  Zmaster.OutputFormat    Z8
 ...
 % levelset definition in default phi.dat, psi.dat files
```

```
 ***xfem_crack_mode
  **discontinuity 3d_levelset no_option
 ...
 % special xfem output requests allowed in the Z8
 % output mode
 ***output
   **xfem_split_integ
   **xfem_split_node
   **xfem_split_contour
 ...
****return
```

Example:

```
% use with the g-theta module
****calcul
 ...
 % mandatory to link xfem with gtheta
 ***xfem_gtheta
 ***xfem_crack_mode
  **discontinuity 3d_levelset
 ...
 % gtheta commands for SIF calculation
 % (note the **xfem keyword)
 ***compute_G_by_gth gtheta_a
    **xfem
    **behavior paris
 ...
****return
```



Example of Zmaster capabilities to handle XFEM results and draw the solution on a *cracked* mesh as if the discontinuity was explicitly introduced in the model

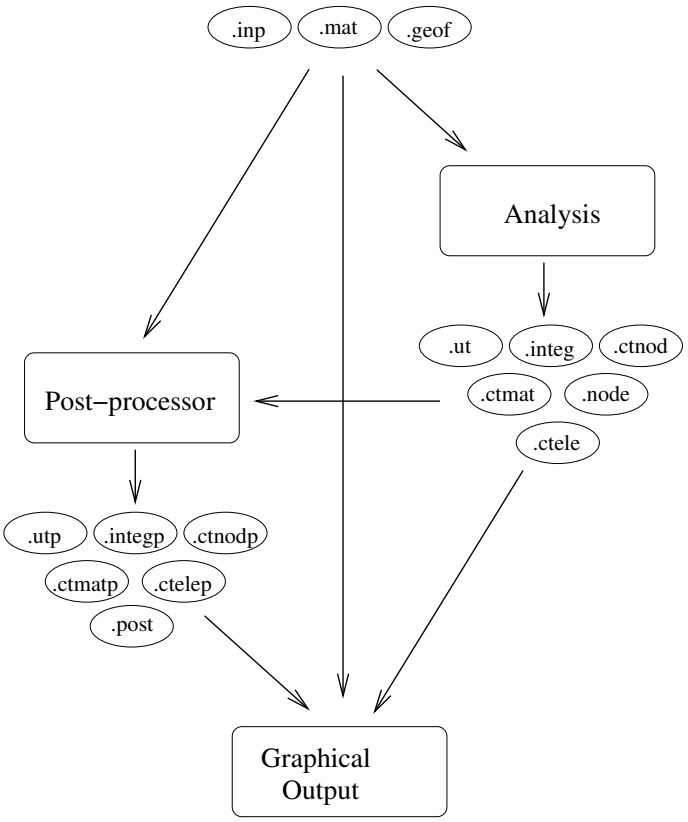
Chapter 4

Post calculations

Introduction to Post Computations

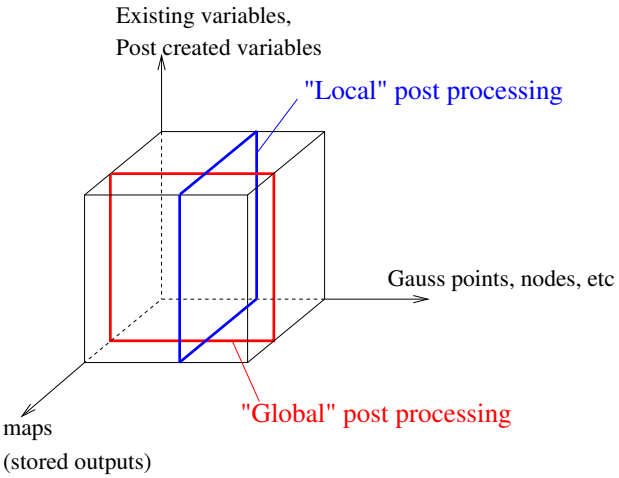
This mode of operation is used to apply post calculations on results obtained by the finite element method, and with simulations. Because the data input for post processing are in the calculation results files, the file name resulting from the post-processor are constructed from the original name, by adding `p` to the different file name extensions. For example, the treatment of integration points in the problem *problem*`.integ` will produce a file named *problem*`.integp`. A file *problem*`.utp` is automatically generated in order to allow graphical visualization of the new variables.

All of the results (calculation and post-calculations) are simultaneously available in the Zmaster graphical program, and the Zmaster batch program.



Two basic types of operations are defined:

- A post treatment so-called *local* which is used to evaluate a criterion at nodal or Gauss point locations throughout the mesh, or in a part of it. The calculation is based only on the data contained at the point(s) of concern.
- Post calculations which are *global* in that they are used to evaluate a criterion over the whole structure, or in a sub-part of it, at a given instant. The calculation therefore normally accounts for data distributed in space.


Existing variables,
Post created variables
"Local" post processing
Gauss points, nodes, etc
"Global" post processing
maps
(stored outputs)

Local post-processing:

The local post computations is used for example to predict the component life of a structure using simple monotonic loading (e.g. Rice-Tracey models), under creep or fatigue (low or high cycle models). It can also used used to determine some derived data relative to the local history at each point, such as the maximum stress temperature or temperature during the loading history.

According to the users choice, the computation will be made at integration (Gauss) points (from the file *problem*`.integ`) or at nodal points from data originally saved at the nodal points (files *problem*`.node` and *problem*`.ctnod`).

The local post treatments give the user the ability to make connections between Z-set computations and other codes as well. This can be achieved for example using the `format` command to create formatted ASCII text output, or using and external program or script to calculate a particular user post computation. Note also that the post-processor is fully extensible using the Z-set plug-ins.

Global post-processing:

Global post processing is applied when, for example, one wishes to calculate the mean values of a variable in the entire structure. The majority of the time, the structure's geometry must be known, and an integration volume is defined. Another example is a Weibull criterion for brittle fracture prediction.

The majority of global post computations produce output which is reduced to a scalar value for each time step of the calculation. These values are stored with some description in an ASCII file *problem*`.post`. Nevertheless, some of the global post processors produce local results (all the time using a non-local algorithm), which will be stored in the appropriate file (node or integration values).

General rules:

For the two types of post-calculation, the user defines the *context* operation: groups of elements, of nodes, or of Gauss points on which to calculate. The output numbers may also be taken into account. The context can be redefines during the calculation. It is important to remember that a given computation or criterion will be applied to the last context which was defined. The individual computations will be applied one after another in the order of appearance in the input file. One can alternate arbitrarily local and global post computations.

The variables for which the post computation is applied can be:

- FEA output: their names appear in the file *problem*`.ut`
- Post-computation output calculated in the course of the same execution., That is, once a post computation is applied, the new variables generated by that application are henceforth added to the list of available variables.

Recursive computations are available using the newly generated variables in post-computations. An example would be to calculate an equivalent strain as the combination of strains at each point, and then calculate the maximum of these equivalent strains. The global results (stored in the file *problem*`.post`) are not however re-usable.

The data source:

In version 8.3 and greater Z-post has the capability to run off of different data source formats than the default Zebulon solver files. Notably there is the possibility to import from ABAQUS, ANSYS Z-sim and arbitrary ASCII files. With this mesher commands can be applied to the results files as necessary to allow new selections to be available. The commands to do that import are described in the section for `***data_source` on page 4.14. An example showing typical use follows.

```
****post_processing
  ***data_source rst                    % load the ansys results
   **open t-base_model_fillet.rst       % and write a native
   **write_geof                         % mesh file (GEOF format)
   **elset LOOK                         % with some set generation
    *elements
      2205 2163 311 325 339 353 381 367 395 409 423 437 479 521
   **nset edge1
    *nodes
    2396 2397 1 2394 2395 2393 2392 2390 2391 2389 2388 2386 2387
    2385 2384 2382 2383 2381 2380 2379 2373
   **bset edge1
    *use_nset edge1
    *use_dimension 2
  ***precision 3
  %--------------------------------------
  ***local_post_processing              % here we copy the displacements
     **output_number 1-999              % so the post generated views
     **file node                        % can be deformed
     **output_to_node
     **nset ALL_NODE
     **process copy *list_var U1 U2 U3
  ***local_post_processing              %
   **output_number 1-999                % Ansys results are contours
   **file ctele   **elset ALL_ELEMENT   % by element (CTELE)
   **material_file  post1.inp           % where to find coefs.. (here we
                                        % use the same physical file)
   **process transform_frame
    *tensor_variables sig
    *output_variables sigm              % "sigma-material"
    *use_element_rotations
  ***return
```

The data output:

Z-post has the capability to output to a number of different output formats, with multiple selections of localized output formats from your posts.

In every case, the standard "-p" files will also be written because these files are used for access to intermediate variables in the course of several posts relying on each other. An example use of this post processing with an ansys input and abaqus output follows:

```
****post_processing
  ***data_source rst                    % load ansys rst format
    **open ansys_model.rst              % in the file ansys_model.rst

  ***data_output odb
    *problem_name abaqus_post_results   % makes abaqus_post_results.odb file
    *elset outer_surface                % make odb of this elset only
  ***local_post_processing
    ...
```

Please note that the data outputting also consist of a format for zebulon files. This output format has following additional features which may be preferable in some cases:

- the data is written to disk only in a single large write, without seek calls which can cause network latency.
- the output can be specified to a list of elsets, from which a reduced mesh is given. this data output can therefore be used to reduce storage size.
- can be used to write to a "zebulon" calculation format (non-p suffixed files) and therefore can form the basis of a subsequent series of post calculations.

Interface with Zmaster:

The post calculations has been embedded in Zmaster in addition to their use by generating additional output files. That is, the post process computations can be run directly on the selected data in Zmaster. After adding a post computation in the Zmaster environment, the `.mast` file needs to be re-saved.

Material coefficients:

Many of the post computation models use coefficients to specify parameters which vary for different materials. There are 2 ways of specifying these coefficients, each of which may be more convenient for different situations.

The first method is to have the material data separated from the process definition. This allows re-use of the coefficients for multiple computations by creating a repository of material values. An example input with two creep entries follows.

```
  **material_file creep.inp
  **process creep
    *var sig
  **process creep
    *var sig
    *express_life_as time
```

Both creep process computations would then use the creep entry in a material file `creep.inp`:

```
 ***post_processing_data
  **process creep
    r 5.
    A 1500.
    S0 0.
***return
```

Another method is to simply put the coefficients in-line with the process command. This is obviously easier for "one-off" post processing:

```
 **process creep
    *var sig
    *model_coef
      r     5.0
      A  1500.0
      S0    0.0
```

Pointers and stacked post computations:

Like all of the Z-set packages, the post processing software is build on objects which perform specific tasks. In the post case however there are a large number of cases where certain calculations such as a thermo-mechanical fatigue analysis employing plastic range, oxidation, and creep damages will need to re-use other computations which themselves can act as stand-alone computations. This is to say, very often a post computation will require pointers to other post input files defining those additional procedures. The following is a detailed example of such input (from the test `lcf.inp` in `Post_test/INP`

The analysis is defined by the "primary" section, which will be the first instance of `****post_processing` in the input file (or N-th instance in the case of an `-N` input switch).

```
****post_processing
    ***local_post_processing
       **file integ
       **elset ALL_ELEMENT

       **material_file ../MAT/test_simple
       **process LCF             % LCF auto creates a stress range making NC_S NF_S
         *mode NLC_ONERA         % creates NR_NLC_ONERA
         *fatigue fatigue_S 2    % this means use fatigue_S post
         *creep   creep     2    % and creep post in post .inp file #2

       **material_file ../MAT/test_with_a
       **process LCF             % there is already NC_S NF_S, new are NC_S_n1 NF_S_n1
         *mode NLC_ONERA         % creates NR_NLC_ONERA_n1
         *fatigue fatigue_S 3    % look in 3rd ****post_processing segment
         *creep   creep     3

       **material_file ../MAT/test_simple_norm
       **process LCF             % now NC_S_n2 NF_S_n2
         *mode NLC_ONERA         % NR_NLC_ONERA_n2
         *fatigue fatigue_S 4
         *creep   creep     4

       **material_file ../MAT/test_with_a_norm
       **process LCF             % and finally NC_S_n3 NF_S_n3
         *mode NLC_ONERA         % NR_NLC_ONERA_n2
         *fatigue fatigue_S 5
         *creep   creep     5

    ***global_post_processing    % average values are taken for the validation
       **output_number 1         % purposes. One can look at a field in Zmaster
       **process average
         *list_var NC_S NF_S NF_S_n1 NF_S_n2 NF_S_n3
       **process average
         *list_var NR_NLC_ONERA NR_NLC_ONERA_n1 NR_NLC_ONERA_n2 NR_NLC_ONERA_n3
****return
```

Note that the active material file is being switched throughout the input structure, and each calculation uses a pointer for the `*fatigue` and `*creep` parts. That is, the `LCF` model is a method of combining fatigue and creep damages (with for example the nonlinear combination

method of ONERA), and the ways that those damages are calculated is defined elsewhere. In this sense we are following the "object-oriented" sense found in all of Z-set, and the damages are being treated as abstractions. They become concrete through the additional objects instanced in the different pointed to data files.

For this example, the following sections are included in the same input file, after the above "primary" section.

```
%section 2
test_simple
****post_processing
    **process fatigue_S
      *var sig
      *mode simple
    **process creep
      *var sig
****return

%section 3
test_with_a
****post_processing
    **process fatigue_S
      *var sig
      *mode with_a
    **process creep
      *var sig
****return

%section 4
test_simple_norm
****post_processing
    **process fatigue_S
      *var sig
      *mode simple
      *normalized_coeff
    **process creep
      *var sig
****return

%section 5
test_with_a_norm
****post_processing
    **process fatigue_S
      *mode with_a
      *var sig
      *normalized_coeff
    **process creep
      *var sig
****return
```

. . . continued

For the material files, there will be a succession of **process sections with the types corresponding to the types of calculation in the input file. For example, in the file ../MAT/test_simple referenced above we have:

```
%
% fatigue coefficients for the formula without a, and
% non normed coefficients
%
***post_processing_data
 **process fatigue_S      % coefficients relating to the
   M          14782.065   % fatigue_S defined in %section 2
   beta       2.5
   sigma_l    40.5
   sigma_u   170.
   b1        0.0003
   b2        0.0003
**process creep          % coefficients relating to
   A 452.                % creep defined in %section 2
   S0 0.
   r 10.
   k 30.
**process LCF            % coefficients controlling the LCF
   a   0.1               % from section 1, mode NLC_ONERA
   k   30.
   beta  2.5
   beta  2.5
***return
```

In the following LCF sections, there is a similar coefficient file with similar structure. These different files get ready because of the sequence of **material_file options swapping the effective material file name as the main input is read.

****post_processing

Description:
This command marks the start of an input section for the post processor. More than one section of this type can be in the same input file *problem*`.inp`. By default, it is the first section which is executed, and commands are read until the next command which begins with 4 asterisks.

The execution command is (see options for Zrun in `Zman intro`):

```
Zrun -pp prob
```

In order to execute other post processing sections in the same file, use the `-N` option. For example:

```
Zrun -N 3 -pp prob
```

will execute the third section of `****post_processing` in the file `prob.inp`.

Syntax:
A post processing file will generally include `***` star commands which adjust the operation of the post computations, and different sections of`***local_post_processing` and `***global_post_processing` sections defining the operations to be performed.

```
****post_processing
 ***precision num
 ***data_source source-type
 ***data_output output-type
 ***global_parameter
 ***suppress_p_on_post_files
 ***post_file_prefix prefix
 ***local_post_processing
    ...
 ***global_post_processing
    ...
****return
```

Commands which are essentially flags or adjustments to the `****post_processing` process follow:

`***global_parameter` After this option one can include global parameter statements as if they were in a `zsetrc` file. Please see the Z-set/Release Notes manual for more information under the Reference/Adjustable parameters section.

`***mesh` This option is used to re-map element types in the mesh to a different (post-processable) type. The command takes a series of *elset-name* and *elem-type* pairs to do the mapping.

`***post_file_prefix` used to set what the output files should be named as. This option makes the output look like another problem from the original input problem.

`***precision` set the precision for formatted output of real values.

***suppress_p_on_post_files this option indicates that the post data files should not have a suffix -p in the set of files used. This option can be used to stack multiple post computations together when combined with the ***post_file_prefix command. One can also use the ***data_output command to a similar effect.

The following additional commands will often be used in multiple instances, and are all discussed in their own documentation sections. The local and post processing commands follow other *** options because they both have many sub-options for all the computation types.

***data_source allows the user to ask the post-processor to read its input data from one of many different source formats (see documentation in following sections).

***data_output more than one instance of this option allows outputting results to different formats.

***global_post_processing Start a sequence of parameters and commands defining the type of global post treatment to be applied.

***local_post_processing Start a sequence of parameters and commands defining the type of local post treatment to be applied.

Example:
A complete post processing file follows to show the basic structure of the input options all together.

```
****post_processing
    ***precision 6
    ***local_post_processing
       **file integ
       **material_file creep.inp
       **elset ALL_ELEMENT
       **output_number 1-100
       **process mises
         *var sig
       **process eigen2
         *var sig
       **process trace
         *var sig
    ***global_post_processing
       **process average
         *list_var sigmises sigp1 sigp2 sigp3 sigii
    ***local_post_processing
       **output_number 1-100
       **process creep
         *var sig
       **process creep
         *var sig
         *express_life_as time
    ***global_post_processing
       **output_number 1
       **process average
         *list_var NC_S TC_S
****return
```

***data_source

Description:
This procedure is used to set the source of data for post processing. The selection should come before any other ***-level commands.

Syntax:

```
***data_source type
[ **open file ]
   ...
 possible mesher commands
```

The following data source types are currently available.

CODE	DESCRIPTION
Z7	standard Zebulon results
d3plot	LS-DYNA binary format
fin	ABAQUS .fin ASCII file format
odb	ABAQUS .odb database format (for launching use Zodb script instead of Zrun)
ideas	I-DEAS format
neu	FEMAP neutral format
rst	ANSYS .rst file (Unix or win64 binary)
sim	Z-set simulator format
t16	MARC t16 results format
fg3	FORGE results format
in3	REM3D results format
ascii	ASCII file input

When using the ***data_source option one gets the chance to add additional meshing operations before the post processing begins. A particularly useful application of this is to add node or element sets which can then be used to specify the location for post processing.

Note:
For ABAQUS FIL formats we recommend using the POSITION=AVERAGED AT NODES *EL FILE option. Please note that the ODB format is much more robust and the preferred method. The ODB format is completely implemented for all node/element node/integration point data formats.

Note:
When using the ***data_source option the post processing will generally pass silently if a requested field variable is available in the results database, but not with the specified location

(e.g. element nodes `ctele`, integration points `integ`, etc). In those cases the data used will most likely be null (in some instances an automatic interpolation or extrapolation is used).

Note:
For FORGE and REM3D formats an alternative method to translate results into Z-set format is to use `****forge` utility, as described on page 5.5.

Example:
A typical example follows. For import examples there are numerous `.pst` files in the `Z-mat` test directory and the `Zansys` directory. Also for Zebulon, Z-sim, and ASCII data sources look in the `Post_test` directory.

```
****post_processing
  ***data_source Z7
    **open plast3_util
  ***precision 6
  ***global_post_processing
   **file node
   **output_number 1-999
   **nset ALL_NODE
   **process curve plast3_util.test
    *precision   3
    *node look   U2
    *nset press  RU2
    *node look   eto22 sig22 epcum
****return
```

. . . continued

Example:
Output from Z-sim or even ASCII files can be used for post processing (even with 1D data). An example of of loading the binary results from the simulator follows:

```
****post_processing
  ***data_source simulator
    **dimension 2          % sets the pb dimension
    **file_name sim_pb     % data discribed in sim_pb.uti
  ***local_post_processing
    **file integ           % sim data is always integ
    **elset ALL_ELEMENT    % always need a location
    **process format
      *file sig.txt
      *list_var sig11 sig22 sig33 sig12
****return
```

An example load from ascii file follows:

```
****post_processing
  ***data_source ascii
    **dimension 2
    **file_name result.dat
  ***local_post_processing
    **file integ
    **elset ALL_ELEMENT
    **process format
      *file sig.txt
      *list_var sig11 sig22 sig33 sig12
****return
```

In this case, the post-processor expects `result.dat` to be an ascii file in column order containing a first line which describes the contents of all columns. This file could contain, for instance:

```
# time sig11 sig22 sig33 sig12
0. 0. 0. 0. 0.
10. 0. 112. 0. 0.
```

There are several validation examples in `test`Post_test/INP/ for ascii and simulation data sourcing. Abaqus fil and odb formats are used int the `.pst` files under the `Z-mat` directories.

***data_output

Description:
This procedure is used to add a destination output of the post processing data of the following commands. The selection should normally come at the start of the ***-level commands, except for a ***data_source command which should always be first.

Syntax:

```
***data_output type
 *problem_name name
 *elset eset-name
 type-specific-options
```

*problem_name is used to specify the root-name of the problem which would have been run in the native code. Usually the output file will be this name appended with a dot suffix of the file type.

*elset specifies that the output problem is to be a sub-mesh made up of the given element set elements. This can be used as a general results file reduction capability with

The following data source types are currently available.

CODE	DESCRIPTION
odb	ABAQUS odb database format (for launching use command switch Zodb -pp instead of Zrun -pp)
rst	ANSYS rst database format
fg3	FORGE fg3 database format

Currently there are no additional *type-specific* options.

. . . *continued*

Example:
A typical example follows. The data output capability has test cases in the `test/Z-mat/zebu_interface` with a full assortment of stacked input and output runs are done.

```
****post_processing
  ***data_source Z7
    **open zpost_matsim.utp

  ***data_output odb
     *problem_name lifetime
     *elset MANIFOLD

  ***local_post_processing
     **output_number 1-9999
     **file node
     **nset manifold_nodes
     **material_file post_coefs.mat

   **process neu_sehitoglu_evi
    *total_strain_for_range

   **process copy        % for validation between input/output
    *list_var evcum
    *out_var  evcum
    *last_only
****return
```

***local_post_processing

Description:
This procedure is used to define the local post computations to apply at nodes or Gauss points. We recall that the local post computations are performed across all stored time points at each location. These operations are therefore primarily temporal, whereas a global post is used for spatial operations such as averaging over volumes. The global operations *can* however generate similar fields as local computations.

Syntax:
The following syntax summary applies:

```
***local_post_processing
[ **duplicate | **no_duplicate ]
[ **elset eset ]
[ **file file-key ]
[ **ipset ipset ]
[ **material_file fname ]
[ **nset nset ]
[ **output_number out-num-list ]
[ **at t1 t2 ... tN ]
[ **output_to_node|**no_output_to_node ]
[ **packet_size size ]
[ **process type ]
   ...
```

The sub options define geometrical groups of concern, time period, material files, and the post treatments to apply. Any number of ****process** commands can be added in a *****local_post_processing** section, and they will all normally have their own sub commands and parameters to enter. The following table summarizes the function of each of these commands, and the pages which follow give the detailed command input syntax.

****duplicate** switches if we allow post processors to create variables with the same name or if the duplicate names are appended with **_n#** in the ****no_duplicate** case (default).

****elset** define the element set of concern.

****file** define the *type* of file where the data will be read. (*problem*.**integ**, *problem*.**node**...).

****ipset** define the Gauss point group of concern.

****material_file** indicates that a separate material file which is to be used for reading material coefficients if no ***model_coef** is entered in a ***process** command.

****nset** define the node set of concern.

****output_number** define the period of time which will be used. The default is the full time period of existing results.

****at** is an alternative to **output_number**: specific maps can be selected through their time.

**output_to_node indicates that the post output should be directed to a node file and not a ctnod file. The distinction is that if copying displacement variables the new files can have deformed geometry displayed in Zmaster.

packet_size can be used for large output results files to decrease physical memory requirements. In this case post computations are performed by packets of *size* elements/nodes. **Note: This option is not compatible with all data-output formats.

**process add a new post computation and begin reading the input for it.

Note:
In the first instance of **local_post_processing or **global_post_processing the full description of the file/localization (e.g. elset)/output range, etc. must be specified. Subsequent processing entries will however use the last entered data, so that easy switching of local and global post computations can take place.

Example:
The following is another example input from the test Rainflow_test/INP/fatigue.inp:

```
****post_processing
    ***precision 5
    ***local_post_processing
       **file integ
       **elset ALL_ELEMENT
       **output_number 1-40
       **material_file fatigue.mat
       **process range
         *alpha 0.001
         *var sig
       **process fatigue_S
         *mode simple
         *var sig
       **process multirange
         *var sig
         *reverse 5
       **process fatigue_rainflow
         *var sig
         *mode simple
         *reverse 5
    ***global_post_processing
       **output_number 1
       **process average
         *list_var Dsig NF_S ncyc D1sig NF1
****return
```

**output_number

Description:
This command is used to define the period of time which will be active for the post computation.

Syntax:

**output_number [*n1-n2*| *n1*, *n2*, ..., *nN*]

The user indicates the number of "maps" in the form of a list of integers, or intervals of integers. The elements of the list are separated by commas or blank spaces. The intervals are specified with two integers separated by a dash (-).

In the absence of an **output_number entry, the post-processor will use all the solution maps in the file *problem*.ut.

Example:
Output numbers have to be specifically asked for. The following input file snippets demonstrate the syntax allowable for this.

```
% to get outputs 1,2,3,4,5,20,35
**output_number   1-5,20,35

% to get outputs 1 and 20
**output_number   1 20

% to get outputs 1, 20 and 30
**output_number   1,20 30
```

**at

Description:

This command selects which maps which will be active for the post computation by their time, rather than by their number. Thus, it is an alternative to `**output_number`.

Syntax:

`**at` *t1 t2 ... tN*

Note that specified time steps that do not exist in the results files are ignored. Explicitly specifying *t1 t2 ... tN* in the `***resolution` bloc ensures that such maps will exist.

**nset

Description:
This command defines the nodes to be treated by specifying a node group (nset). One can use the mesher (`****mesher`) mode of operation to generate these sets.

Syntax:

`**nset` *nset*

The groups of nodes *nset* must of course be defined in the geometry file *mesh*`.geof` designated after the instruction `**meshfile file` in the file *problem*`.ut`. All the nodes of this group will be taken into account for the post calculation. To simply designate all the nodes in the structure, one may always use the pre-defined node set `ALL_NODE`. The nset selection can be modified at any time with a new `**nset` instruction.

Example:

```
**nset   surface

% with the following definition in the .geof file:
**nset  surface
10 11 12 13 19 20 22 30 31
...
```

**elset

Description:
This option defines the elements to be treated with post computations by specifying an element group (elset). Elsets can be generated in a mesh using the batch mesher (chapter 4).

Syntax:

`**elset` *elset*

The group of elements *elset* must of course be defined in the geometry file *mesh*`.geof` designated after the instruction `**meshfile` in the file *problem*`.ut`. All the elements of this group will be taken into account for the post calculation; that is to say that the post computations will be evaluated at all the elements Gauss points. To simply designate all the elements in the structure, one may always use the pre-defined element set `ALL_ELEMENT`. The elset selection can be modified at any time with a new `**elset` instruction.

Example:

```
**elset   wheel

% with the following definition in the .geof file:
** elset  wheel
1 2 3 4 5 6 7 8
...
```

**ipset

Description:
This command is used to select the integration point set (ipset) to treat with post computations.

Syntax:

`**ipset` *ipset*

The group of Gauss points *ipset* must be defined in the geometry file *mesh*`.geof` designated after the instruction `**meshfile` in the file *problem*`.ut`. Gauss points are specified in the *mesh*`.geof` file under the group section `**ipset` by a series of element/integration point couples (*nel/nip* with *nel* the element id and *nip* the integration point number indexed from 1). The ipset selection can be modified at any time with a new `**ipset` instruction.

Example:

```
    **ipset   failure

% with the following definition in the .geof file:
    **ipset  failure
    % nb_gp  elem_id  ip_list
      2      1        1 2
      2      2        1 2
      2      3        1 2
      2      4        1 2
```

**file

Description:
This command is used to specify on what types of file the post calculation will be calculated. **There is no selection by default**.

Syntax:

```
**file  filetype
```

where *filetype* is the selection of files to read, among the possible files which were output by the finite element calculation. In the general case, the type of output file produced is the same as the input data file (i.e. if the input is an `.integ` file the output `.integp` file has the exact same format with different variables).

The following table is a summary of the different possibilities:

filetype	file read	file created
`integ`	*problem*`.integ`	*problem*`.integp`
`ctele`	*problem*`.ctele`	*problem*`.ctelep`
`ctmat`	*problem*`.ctmat`	*problem*`.ctmatp`
`node`	*problem*`.node`	*problem*`.ctnodp`
	+ *problem*`.ctnod`	+ *problem*`.ctnodp`

Certain post-processors do not respect this generality. For example `node_interpolation` reads its input data in the file *problem*`.node` but produces results in the file *problem*`.integp`.

The "active" file selection can be modified at any instant with a new `**file` instruction.

Example:

```
% to select the file extension .integ
% the results will be written in a file with a .integp
% extension
**file  integ
```

**material_file

Description:
This command is used to specify the name of a material file containing coefficients when they are necessary. This file can be updated during the progression through the post calculations if it is necessary. The current material file is the one selected by the last instance of this command.

Syntax:

```
**material_file  file
```

Example:

```
**material_file  aluminum
```

Material file syntax:

As this material file can be the same for the finite element, simulation and post processing calculations, it is necessary to localize the post-processor's specific data. This is done with an entry of the command `***post_processing_data`[1]. In this data-entry section, different entries for the different post processors requiring coefficients should be entered. Each section will contain coefficients specific to the post processor given. Coefficients use the `COEFFICIENT` objects given in the Z-mat manual in Chapter 5.

Note that the `COEFFICIENT` can depend on the internal variables, auxiliary variables, flux and grad variables, or the external parameters (if the `save_parameter` option was used for the later).

Syntax:

```
***post_processing_data
 **process PROCESS [ option1 ... optionP ]
 name1 COEFFICIENT
 ....
 nameN COEFFICIENT
```

The optional input parameters *option1 ... optionP* can be used to specify the particular use of the given post computation by giving an application option keyword, the tensor or variable on which it operates, etc.

The names *name1 ... nameN* are the local names of the coefficients necessary to define for the corresponding calculation. These coefficient names are given for each process computation in the manual sections which follow.

Example:

[1] Other behavior commands such as `***behavior` terminate at the next command of level `***`. This effectively distinguishes the components of the material file, just as the `****` level commands do in the input file

```
***post_processing_data
 **process  fatigue_S  sig
    M   2400.
    beta  5.
    sigma_l 120.
    sigma_u 450.

***post_processing_data
 **process  creep xtv
    A   temperature
    1400. 1050.
    2350.  900.
    5000.  700.
    r    5.
```

**process

Description:
This keyword introduces a post-computation to be added to the set of available results. most post computations take one or more *subject* variables which will provide the basis of the calculations (e.g. finding principal values of a tensor takes a tensor name as the subject, which could be `sig` to find the principal stresses).

Post-computations which take subject variables introduced by the key word `*list_var` are applied uniquely to scalar variables. All the character strings following the key word are interpreted as such.

Certain post-computations which use the keyword `*var` to introduce the subject variable are applied to either scalar or tensorial variables. In this case, there will also be a command option named `*type` which is provided to specify the type of variable. Nevertheless, in order to simplify the input, the following variable name convention is given:

- All character strings which end in `11`, `22`, `33`, `12`, `23` and `13` will be interpreted as the name of a scalar variable.
- Otherwise the character string is interpreted as a tensorial variable name.

The following local post-processor types are available in the standard distribution.

HCF	LCF	adiabatic_temperature
base_fatigue	copy	creep
delay	derive	deviator
ductile_failure	eigen2	evcum_sum
external	fatigue_E	fatigue_EE
fatigue_S	fatigue_rainflow	fmax
fmin	format	function
harmonic	hyper_visco	inc_creep_damage
inelastic	initiation	integrate
load	local_frame_axes	mat_sim
max	min	mises
multirange	neu_sehitoglu	norm
oxidation	rainflow	range
trace	transform_frame	tresca
triax	z7p	

**process copy

Description:
The `copy` post processor is used to re-name a variable as a copy into the results database files. There are several occasions where this operation would be useful:

- For Z-mat runs where all the state variables are named `SDV#` (ABAQUS) or `SVR#` (ANSYS), etc. The copy operation can be used importing the original input file using a `**data_source` (see page 4.14) and re-naming them in a more meaningful way. Note also that if the output naming respects the `11 22 33 12 23 31` ordering those sub-variables will be considered as tensors in further Z-post operations.

  Remember that the `SDV#` name mapping can be determined using the `Zpreload` command or by looking at the calculation message files.

- To copy a database into a different format, but making certain to respect important fundamental names (such as `U1 sig11` etc. Using the copy functionality can therefore be used to reduce the size of a results set, or to translate between different supported formats.

- To fake the name of a tensor if needed. For example one could open an Abaqus ODB file, and copy the plastic strain tensor to be the total strain tensor and submit that to another code requiring total strain only.

There are no doubt other uses for such a capability, but these are the primary intent of the command.

Syntax:

```
**process copy
  **list_var var1 var2 ... varN
[ **out_var  o-var1 o-var2 ... o-varN ]
```

The variables will be copied in the order given. Memory can be reduced for large models by breaking up man copies into several `**process copy` operations.

Note:
If the `**out_var` option is not given, the output names will be the same as the input names, so that should only be used when the post results are desired to "stand alone."

. . . *continued*

```
****post_processing
 ***local_post_processing
  **process copy
```

Example:
The following copies an Abaqus odb file to a new one while re-naming the state variables (in this case a `***data_storage` command was used to reduce the SDV allocations to just 2 variables).

```
****post_processing
  ***global_parameter
    ODB.MaxSteps 100

  ***data_source odb
   **open my_abaqus_run.odb
   **write_geof
   **nset interesting_nodes *elset MAIN_ELSET *function 1.0;

  ***data_output odb
   *problem_name zpost_copy_odb
   *elset MAIN_ELSET

  ***local_post_processing
     **output_number 1-999
     **file node
     **nset interesting_nodes
     % DO NOT USE PACKET SIZE HERE

     **process copy
      *list_var SDV1     SDV2    NT11
      *out_var  evcum    Dsum    temperature

     **process copy
      *list_var sig11 sig22 sig33 sig12 sig23 sig31

     **process copy
      *list_var LE11  LE22  LE33  LE12  LE23  LE31
      *out_var  eto11 eto22 eto33 eto12 eto23 eto31

****return
```

**process creep

Description:
This post calculation calculates creep damage at an instant t in relation to a variable T, such as the integration of time between t_1 and t:

$$I(t) = (k+1)\int_{t_1}^{t}\left(\frac{S-S_0}{A}\right)^r$$

with

$$S = (1-\alpha-\beta)\mathbf{Mises}(T) + \alpha\mathbf{Eig1}(T) + \beta\mathbf{Trace}(T)$$

where **Mises**(T) indicates the second invariant of the deviatoric part of a tensor **T**, **Eig1**(T) the maximum principal value, and **Trace**(T) the trace (first invariant.

In accordance with the users choice, the result is given in terms of time to rupture or number of cycles to rupture. Two cases are envisioned:

- $I(t)$ attains 1 for a value of time between t_1 and t_2: the corresponding value is the time to rupture, and the number of cycles to rupture is set to one.
- $I(t_2)$ is less than 1: the time to rupture is obtained by superimposing a cyclic time and linear accumulation law. The number of cycles to failure is simply the inverse of $I(t_2)$ and the time to rupture is obtained by multiplying this number by the length of loading considered.

Syntax:

```
**process creep
  *var name
 [*type scalar | tensor ]
 [*express_life_as cycle | time ]
 [*delay ]
 [*scale lin | log ]
 [*precision prec ]
```

name is the name of the variable to treat. The option `*express_life_as` is used to specify the output mode to be number of cycles or time to failure. With the option `*delay` a scaled stress measure is used: $dS' = (S - S')/\tau$.

The user must furnish the coefficients `S0`, `A`, and `r`. Coefficients `k`, `alpha` and `beta` are optional with zero value by default. If it is used, the coefficient `k` must be a constant. One more supplemental coefficient `tau` is expected with the option `*delay`.

In the file *problem*`.utp`, we call the variable for time to rupture `TC_S`, and the number of cycles to rupture `NC_S`. If the output was put on a logarithmic scale, the names will be `LTC_S` and `LNC_S`. We generally prefer to work with log values when generating contour plots.

Creep damage is integrated using a second-order runge-kutta method. The precision required for integration can be specified by the optional `*precision` keyword. Default value is 1.e-4.

```
****post_processing
 ***local_post_processing
  **process creep
```

Example:

```
% Use stress tensor to compute time to failure and give
% a logarithmic output
**process creep
  *var sig
  *scale log
  *express_life_as time

% The following syntax is used in the material file :
**process creep
   A  1130.
   r    7.
   S0   0.
   alpha   0.3
   beta    0.

% ... but the coefficients may be non constant
**process creep
   A  temperature
   1130.   1050.
   2000.    700.
   ....
```

**process cycle

Description:
The `cycle` post processor may be used to apply the same post-processing operations on all cycles of a cyclic calculation. Typical application is to monitor the evolution of critical quantities (stress amplitude, mean stress, number of cycles to failure, etc...) from one cycle to the other.

This post-processor takes an arbitrary number of sub-processors as arguments. The output maps available in the results file that correspond to each cycle are automatically selected, and given as input to the embedded sub-processors. The definition of the time period of the cyclic calculation is used as a basis to sort out which maps do belong to a particular cycle.

A single map of post-processing results will be generated for each cycle detected in the sequence of output maps available in the input results files. In this context two types of embedded sub-processors may be distinguished:

- (i) post-processors that generate one map of results for each input map (eg. `*process mises`, `*process function`, etc ...)
- (ii) post-processors that generate one single map for the whole range of input cards (eg. `*process max`, `*process range`, etc ... and all the damage post-processor in general)

Type (ii) needs no further specification, since the results given for each cycle correspond to a standard calculation on the associated cycle input maps (that may be selected by a repeated use of the `**output_number` command). On the opposite, for type (i) post-processors 2 options are available to specify if either the *max* of *average* value should be retained for each cycle.

Syntax:

```
  **process cycle
    *period period
  [ *start  tstart  ]
  [ *end    tend  ]
    *process proc1 [ typ1 ] [sec1 ]
    *process proc2 [ typ2 ] [sec2 ]
...
    *process procn [ typn ] [secn ]
```

where *period* if the time period value of the cycle.

The `*start` command may be added if a preload sequence is stored in the results file, that needs to be skipped before scanning for cyclic results.

Similarly, the optional `*end` command will stop the scanning of cyclic results for maps whose time values are greater than the *tend* specified.

Embedded sub-processes are specified by an arbitrary number of `*process` commands. Argument *proci* is the name of the sub-process, and *seci* is the number of the input file section where the actual `**process` *proci* definition will be given. Default value for this section number is 2. For type (ii) post-processors (see description above) argument *typ1* can be either `max` or `average` (default value is `max`).

```
****post_processing
 ***local_post_processing
  **process cycle
```

Variables written in the post-processing files are the cycle number (variable `cyc`) and output variables associated to the various sub-processes. For the latter, the "`_cyc`" character string is appended to the conventional variable name.

Note:
Specification of a material file is mandatory when using `*process cycle`, even if embedded sub-processors don't need any material coefficients. In the latter case an empty material is required:

```
***post_processing_data
***return
```

Example:

```
The following commands can be used to compute
the max value of the von mises stress invariant
the mean value of the average stress, and the
multiaxial stress amplitude for each
cycle stored in the FE results files.
The process format allows to store the result
in an ascii file: note the "_cyc" character
string added to the name of the sub-process
output variables.

% first section
****post_processing
  ...
  % note the need of an empty material file
  **material_file fake
  **process cycle
   *period 10.
   *process mises max 2
   *process trace average 2
   *process range 2
  **process format
   *list_var cyc sigmises_cyc sigii_cyc Dsig_cyc
   *file cycle.post
****return

% second section: sub processes definition
****post_processing
**process mises
*var sig
**process trace
*var sig
**process range
*var sig
****return
```

```
****post_processing
 ***local_post_processing
  **process cycle
```

The *process fatigue_S should in theory be applied
on the stabilized cycle. The next example illustrates
how to use *process cycle to evaluate the influence
of an incomplete material stabilization on
the number of cycles to failure predicted by the
fatigue_S model.

```
% first section
****post_processing
  ...
  **material_file fatigue_S_coefs
  **process cycle
   *period 50.
   *process fatigue_S
  **process format
   *list_var cyc NF_S_cyc
   *file nf_evolution.post
****return

% second section: sub processes definition
****post_processing
**process fatigue_S
*var sig
****return
```

**process cycle_projection

Description:
This post is used when we have a multiaxial loading which is close to proportional and it is desired to see the primary stress-strain hysteresis loops. The user will pick a specific point in a cyclic loading (or series of points at similar conditions) and the requested tensors will be projected onto the eigen directions determined at that point.

The user can specify the point from which we take the eigen values/vectors from:

- A map number can be given. This will be indexed base 1 from the start of the selected cards in the active post section (that is if we select maps 20-200, a selection of 20 will be number 40 in the global maps).
- A time can be given. If the time is between 2 maps the values will be interpolated.
- The post calculation uses a POST_CYCLE_TOOL object to determine the cycle positions, from which we can use a cycle number combined with start/mid/end tokens. The default is this method, with cycle 1/mid selected.

Using the eigen vectors of the ordered principal calculation of the given `*primary_variable` selection we project the selected tensors into that frame and output them with either the given list of output names, or as *tname*`_cpj`. If the primary tensor is desired as output then it should also be given in the `*var` listing.

Syntax:

```
**process cycle_projection tool-type
  *primary_variable name
  *var     name1  name2  ...  nameN
[ *output name1  name2  ...  nameN ]
[ *time time-val ]
[ *map  map-index ]
[ *cycle num  (start|mid|end) ]
[ *remap_each_cycle ]
```

Note:
The POST_CYCLE_TOOL expects that `temperature` be defined as a field variable. If it does not, one can use the `make_field` local post computation in order to do so.

. . . *continued*

```
****post_processing
 ***local_post_processing
  **process cycle_project
```

Example:
A common usage is to use the `hot_spot` post computation to identify critical regions, and then study the mechanical cycles at that location.

In this example we have a nset named `min_life.nset` stored via a `hot_spot` (see page 4.124). That nset is split in the beginning mesher command `**split_nset` run when loading the data source so we can specify node locations symbolically. The worst case node is then named `min_life_1`.

Note that the `tmf` keyword after the `**process cycle_projection` command identifies the type of cycle identifying tool to use. Currently the reader should refer to the `sehitoglu` life prediction methods (page 4.85) for a discussion. Future versions of the documentation will have a more expansive discussion on that.

```
****post_processing
  ***data_source Z7
    **open zpost_matsim.utp
    **nset min_life
     *nodes_in_file min_life.nset
    **split_nset min_life

  ***local_post_processing
     **output_number 1-9999
     **file node
     **nset min_life_1

     **process cycle_projection tmf
      *primary_variable sig
      *var   sig eps_me evi
      *out   sig_x eps_x evi_x
      *cycle 1 mid
      *remap_each_cycle mid
      *total_strain_for_range

  ***global_post_processing
   **process curve curves_1.test
    *precision   3
    *node min_life_1  eps_x11 eps_x22 eps_x33 eps_x12 eps_x23 eps_x31
                      sig_x11 sig_x22 sig_x33 sig_x12 sig_x23 sig_x31
                      evcum  temperature
****return
```

**process derive

Description:
This post calculation computes the derivative of some fields with respect to a variable (time or another variable). It is computed using a first order formula:

$$\dot{f}(t_i) = \frac{f(t_i) - f(t_{i-1})}{t_i - t_{i-1}} \quad \text{if the field is derived wrt. time}$$

$$\frac{\partial f}{\partial x}(t_i) = \frac{f(t_i) - f(t_{i-1})}{x(t_i) - x(t_{i-1})} \quad \text{if the field is derived wrt. another variable}$$

Syntax:

```
**process derive
  *list_var names base
```

names is the list of variables to derive with respect to *base*.

Example:
The following example computes the velocity in each direction. It will generate the fields `D_time_U1`, `D_time_U2` and `D_time_U3`.

```
****post_processing
 ***local_post_processing
  **file node
  **nset ALL_NODE
  **process derive
   *list_var U1 U2 U3 time

 ***local_post_processing
  **file integ
  **elset ALL_ELEMENT
  **process derive
   *list_var eto11 time
****return
```

**process deviator

Description:
This post calculation calculates the deviatoric stress tensor. The prefix `dev_` is prepended to the tensor name.

Syntax:

```
     **process deviator
       *var name
```

Example:

```
****post_processing
 ***local_post_processing
  **file integ
  **elset ALL_ELEMENT
  **process deviator
   *var sig
****return
```

**process ductile_failure

Description:
This post treatment is applied to predict ductile rupture under monotonic loading. The defines a damage variable D at the time t which is the integration of time between t_0 and t:

$$D(t) = \exp\left[\int_{t_0}^{t} a \exp\left(b\frac{\text{Trace}(S)/3.}{\text{Mises}(S)}\right) dp\right]$$

where S is a tensor and p is a scalar variable (normally the equivalent plastic strain).

One output is generated for each solution "map." As the evaluation of this criterion is based on an integration, a sufficient number of result outputs are required for a precise answer. The user should therefore ask for a complete set of intermediate outputs (see `output_number` on page 4.21).

Syntax:

```
**process ductile_failure
  *stress name1
  *strain name2
 [*rice_tracey]
```

name1 is the name of the tensor, and *name2* that for the scalar. With the option `*rice_tracey` the criterion is evaluated with the stress tensor `sig` and the cumulated plastic deformation[2].

The default values for the coefficients `a` and `b` are those used by the Rice-Tracey criterion: `a`=0.283 and `b`=1.5.

The variable names generated (in the *problem*`.utp` file, and appearing in the Zmaster results) is `DAMAGE` in the case of a general ductile criterion, and `R_R0` if the particular case of Rice-Tracey was chosen.

Example:

```
% Applying the Rice-Tracey criterion needs only:
**process ductile_failure
  *rice_tracey

% The following instructions allow the user to also compute
% Rice-Tracey criterion. sig and epcum are respectively
% the names of the stress tensor and of the
```

[2]this assumes the name `epcum` for the cumulated plastic strain. Note that `gen_evp` behaviors name this variable according to the users input after a potential keyword

```
                                                    ****post_processing
                                                     ***local_post_processing
                                                      **process ductile_failure

% cumulated plastic strain in the prob.utp file.
**process ductile_failure
  *stress sig
  *strain epcum

% The following syntax must be used in the material file :
**process ductile_failure
  a    0.283
  b    1.5

% This is another criterion:
**process ductile_failure
  *stress x1v
  *strain evcum

% ... with other values in the material file:
**process ductile_failure
  a    0.33333
  b    2.
```

**process eigen2

Description:
This process is used to compute the eigenvalues of a symmetric second order tensor. No coefficients are needed. The process provides three values for each point. The eigenvalues are given in decreasing order, starting by the larger one. Their name in the *problem*.utp file is built from the name of the tensor by adding respectively p1, p2 and p3.

Syntax:

```
**process eigen2
  *var name
```

Example:

```
% this will provide sigp1, sigp2, sigp3
% with sigp1 >= sigp2 >= sigp3
**process eigen2
  *var sig
```

**process fatigue_E

Description:
This post-processor is used to predict fatigue lifetime using a strain measure. This strain measure can be a plastic strain, or another one such as the total strain. The critical variable is the amplitude of the strain (plastic) for uni-axial loading with a generalization pertinent for multi-dimensional loading (see the post computation `range`). If the loading is well known, the user can give simply the output "maps" necessary to calculate this amplitude (mode *n1 n2* in `**output_number`). In the other case, a period of loading must be input (mode *n1-n2* in `**output_number`).

The strain amplitude is denoted **DEQ** . The post calculation will generate a single result record for the entire loading history. The number of cycles to failure is defined as follows:

$$N_f = \left(\frac{\mathbf{DEQ}}{A}\right)^{-\alpha}$$

Syntax:

```
**process fatigue_E
  *var name
 [*type scalar | tensor ]
 [*scale lin | log]
 [*range section]
```

name is the name of the variable to use as input to the computation.

For multi-dimensional loading the amplitude calculation will be made using a `range` type post-computation. In order to be able to input options to the range processor, the user can give a section number for that user input after the `*range` keyword.

The computation requires input of the coefficients `A` and `alpha` in the material file.

Output is given as number of cycles to failure in the variable `NF_E`. If the user specifies a logarithmic scale using the `*scale` option, the output variable will be `LNF_E`.

Example:

```
**process fatigue_E
  *var eto
  *scale log
% The following syntax must be used in the material file :
**process fatigue_E
   A         1.7
   alpha     2.
```

**process fatigue_EE

Description:
This post-computation is used to determine component life by a criterion dependent on a strain measure (elastic and plastic). The critical variable is the amplitude of the strain for uniaxial loading, with a generalization for multi-dimensional loading (see the post processor `range`). If the loading is well known, the user can give simply the output "maps" necessary to calculate this amplitude (mode *n1 n2* in `**output_number`). In the other case, a period of loading must be input (mode *n1-n2* in `**output_number`).

The strain amplitude is denoted **DEQ** . The post computation will generate a single output record for the total loading history input. This is the number of cycles to failure, given by:

$$\mathbf{DEQ} = \mathrm{A}\ \mathrm{N_f}^{-1/\alpha} + \mathrm{B}\ \mathrm{N_f}^{-1/\beta}$$

Syntax:

```
**process fatigue_EE
  *var name
 [*type scalar | tensor ]
 [scale lin | log]
 [range section]
```

name is the name of the subject variable of the computation.

For multi-dimensional loading the amplitude calculation will be made using a `range` type post-computation. In order to be able to input options to the range processor, the user can give a section number for that user input after the `*range` keyword.

The computation requires input of the coefficients `A` and `B`, `alpha` and `beta` in the material file.

Output is given as number of cycles to failure in the variable `NF_EE`. If the user specifies a logarithmic scale using the `*scale` option, the output variable will be `LNF_EE`.

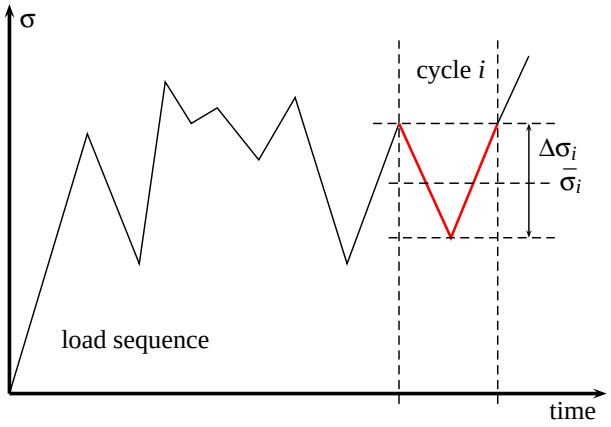
Example:

```
**process fatigue_EE
  *var sig
  *scale log

% The following syntax must be used in the material file :
**process fatigue_EE
   A       3.
```

```
****post_processing
 ***local_post_processing
  **process fatigue_EE

alpha   3.
B       1.7
beta    1.5
```

**process fatigue_rainflow

Description:
This command is used in replacement of `**process fatigue_S` for rainflow counting evaluations of fatigue life. Considering the complex load sequence of the next figure:



for each elementary cycle detected by the multiaxial rainflow algorithm (`**process multirange`, see page 4.75), a number of cycle to failure N_{fi} is calculated using the Chaboche fatigue stress-based model:

$$N_{fi} = \frac{\langle \sigma_u - \sigma_i^{max} \rangle}{a\ (\beta+1)\ \langle \sigma_i^a - \sigma_l(\bar{\sigma}_i) \rangle} \left[\frac{\sigma_i^a}{M(\bar{\sigma}_i)} \right]^{-\beta} \tag{1}$$

where $< . >$ denotes the positive part operator, $\sigma_i^a = \frac{\Delta\sigma_i}{2}$ the amplitude, $\bar{\sigma}_i$ the mean, and σ_i^{max} the maximum value of the stress path for the current elementary cycle. The previous scalar quantities are computed in the following way from the multi-axial stress path:

- the stress amplitude σ_i^a of an arbitrary multiaxial stress path is computed by `**process multirange`,
- the mean stress $\bar{\sigma}_i$ is defined as the mean trace of tensor $\underset{\sim}{\sigma}$

$$\bar{\sigma}_i = \frac{1}{2} \left[\max_{cycle\ i} \left\{ tra(\underset{\sim}{\sigma}) \right\} + \min_{cycle\ i} \left\{ tra(\underset{\sim}{\sigma}) \right\} \right]$$

- the maximum stress σ_i^{max} is the maximum value of either the greatest principal stress (default mode) or the mises invariant (`*use_mises` option) of stress tensor $\underset{\sim}{\sigma}$ for the current elementary cycle.

To account for the mean stress dependence of N_{fi}, coefficients $\sigma_l(\bar{\sigma}_i)$ (fatigue limit) and $M(\bar{\sigma}_i)$ may depend on $\bar{\sigma}_i$ according the following expressions:

- mode `*mean_stress default`:

$$\sigma_l(\bar{\sigma}_i) \;=\; \sigma_{l0}\;(1 - b_1\;\bar{\sigma}_i) \quad , \quad M(\bar{\sigma}_i) \;=\; M_0\;(1 - b_2\;\bar{\sigma}_i)$$

- mode `*mean_stress variant`:

$$\sigma_l(\bar{\sigma}_i) \;=\; \frac{\sigma_{l0}}{1 \;+\; b_1 \;<\bar{\sigma}_i>} \quad , \quad M(\bar{\sigma}_i) \;=\; \frac{M_0}{1 + b_2 \;<\bar{\sigma}_i>}$$

- mode `*mean_stress tos`:

$$\sigma_l(\bar{\sigma}_i) \;=\; \sigma_{l0}\left(\frac{1}{1 + b_1\;\frac{\bar{\sigma}_i}{\sigma_i^a}}\right) \quad , \quad M(\bar{\sigma}_i) \;=\; M_0\left(\frac{1}{1 \;+\; b_2\;\frac{\bar{\sigma}_i}{\sigma_i^a}}\right) \quad ,$$

`variant` is now preferred to the legacy `default` mode, because in the latter case, large mean stress $\bar{\sigma}_i$ may give rise to negative $M(\bar{\sigma}_i)$ values, leading to an indefinite value of N_f in equation (1) (in this case the program will return $N_f = 1$).

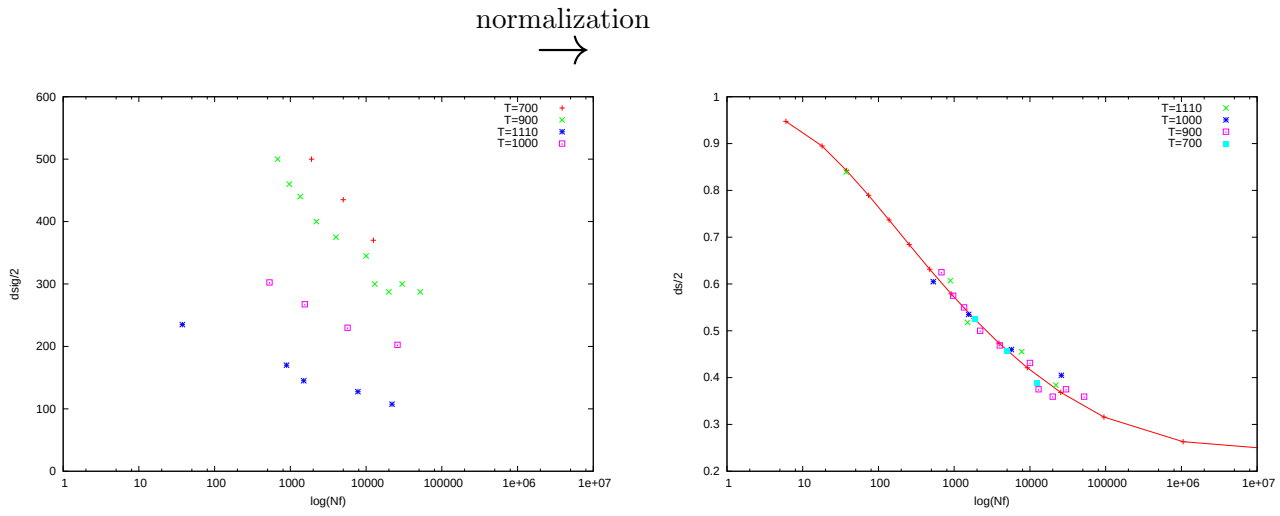
Anisothermal case and coefficients temperature dependence

When the temperature is changing during the cycle, it is customary to use a normalized $\underset{\sim}{\mathbf{s}}(t)$ stress values, defined in the following way:

$$\underset{\sim}{\mathbf{s}}(t) \;=\; \frac{1}{\sigma_n(T(t))}\;\underset{\sim}{\sigma}(t)$$

where $(\underset{\sim}{\sigma}(t), T(t))$ are the stress/temperature values stored in the results file for the current output time t, and σ_n is a temperature-dependent normalization coefficient defined in the material file. Note that defining σ_n is in fact optional, since when σ_n is not explicitly declared, the UTS σ_u coefficient is automatically used to normalize the stress input.

In most cases, using this normalization procedure, all fatigue points obtained at different temperatures may be gathered on a single master Woehler curve, as represented on the next figure.



All model coefficients (except σ_n or σ_u) may then be given as temperature-independent constant values. Note that in this case the `*normalized_coef` option is mandatory, and that values expected for coefficients M, σ_l, $b1$ and $b2$ should be modified accordingly to correspond to a normalized stress input, ie. the standard values should be:

- divided by σ_n (coefficients σ_l, M)
- multiplied by σ_n (coefficients b_1, b_2)

However, if fatigue data at different temperature exhibits too much scatter on such a normalized representation, it may be necessary to define temperature dependence for the remaining coefficients M, β, a, b_1 and b_2.

In this case, a direct computation of N_f by equation (1) is not possible, because coefficients no longer have a constant value for the whole cycle. A special Taira procedure is then applied to compute a temperature T^* (the so-called *equivalent temperature*), and coefficient values are evaluated at this particular temperature:

$$N_{fi} \;=\; \frac{\langle \sigma_u(T^*) - s_i^{max} \rangle}{a(T^*)\;(\beta(T^*)+1)\;\langle s_i^a - \sigma_l(T^*,\bar{s}_i)\rangle} \;\left[\frac{s_i^a}{M(T^*,\bar{s}_i)}\right]^{-\beta(T^*)} \tag{2}$$

where stress σ is replaced by the corresponding normalized value noted s in equation (2).

To calculate this equivalent temperature T^*, a fatigue incremental model, corresponding to a simplified version of equation (1), is first introduced hereafter :

$$\dot{D}^* \;=\; \frac{\beta^*}{4\;B^*}\;\left(\frac{J\left(\underset{\sim}{\mathbf{s}}-\underset{\sim}{\mathbf{X}}\right)}{M^*}\right)^{\beta^*-1} J\left(\dot{\underset{\sim}{\mathbf{s}}}\right) \tag{3}$$

where $J(.)$ denotes the von mises invariant, $\underset{\sim}{\mathbf{s}}(t) \;=\; \frac{1}{\sigma_n(T(t))}\;\underset{\sim}{\sigma}(t)$ the normalized stress tensor defined above, and $\underset{\sim}{\mathbf{X}}$ the center of the normalized loading path $\left\{\underset{\sim}{\mathbf{s}}(t_i)\right\}$ (also an output of `**process multirange`). Integration of equation (3) on an isothermal symmetric cycle ($R=-1$), gives the following expression for the total damage D^*, and the corresponding number of cycles to failure $N_f^* = \frac{1}{D^*}$:

$$N_f^* \;=\; \frac{1}{D^*} \;=\; \left(\frac{s^a}{M^*}\right)^{-\beta^*} \tag{4}$$

It can be checked that equation (4) is indeed a simplified version of (1), where both asymptotes ($\sigma^{max} > \sigma_u$ and $\sigma^a < \sigma_l$) are removed. Coefficients M^* and β^* in (3,4) are chosen such as to give the same damage as the one obtained for the full model (1) at a mid point on the Woehler curve, ie. for a stress amplitude $\sigma^a = \frac{\sigma_l+\sigma_u}{2}$. This condition defines the following relationship between coefficients (M^*,β^*) of the simplified model, and coefficients (M,β,a,σ_u,σ_l) of the full expression (1):

$$\beta^* \;=\; \beta \;+\; \frac{\sigma_u+\sigma_l}{\sigma_u-\sigma_l} \quad , \quad M^* \;=\; (a(\beta+1))^{-\frac{1}{\beta^*}}\;\left(\frac{\sigma_u+\sigma_l}{2}\right)^{1-\frac{\beta}{\beta^*}}\; M_0^{\frac{\beta}{\beta^*}} \tag{5}$$

T^* is then defined as the temperature giving the same damage on an isothermal test performed at temperature T^*, as the one obtained on the initial anisothermal loading (according to the incremental model of equation 3):

$$T^* \ \ such\ that \ \ \int_{cycle} \frac{\beta^*(T^*)}{4\ B^*(T^*)} \left(\frac{J\left(\underset{\sim}{\mathbf{s}}(t) - \underset{\sim}{\mathbf{X}}\right)}{M^*(T^*)} \right)^{\beta^*(T^*)-1} J\left(\dot{\underset{\sim}{\mathbf{s}}}(t)\right)\ dt \ = \int_{cycle} \frac{\beta^*(T(t))}{4\ B^*(T(t))} \left(\frac{J\left(\underset{\sim}{\mathbf{s}}(t) - \underset{\sim}{\mathbf{X}}\right)}{M^*(T(t))} \right)^{\beta^*(T(t))-1} J\left(\dot{\underset{\sim}{\mathbf{s}}}(t)\right)\ dt \tag{6}$$

For complex anisothermal loadings, where coefficients depend on temperature, the non-linear equation (6) is then solved to compute T^*, before calculation of N_f using equation (2), where M_0, β, a, $b1$, $b2$ values are taken at temperature $T = T^*$.

Out of phase correction

The stress amplitude σ^a (or s^a in the normalized case) in equation (1) is computed as the radius of the smallest hypershere containing all multiaxial stress points $\underset{\sim}{\sigma}(t)$ for the current elementary cycle. This radius is the same for proportional, out-of-phase, or even completely random variations of tensor components during the cycle. Consequently, according to the model, there is no decrease of N_f for out of phase loadings, a result in contradiction with experiments.

A dedicated option has been implemented to correct this problem (option `*out_of_phase`). The method uses again the simplified incremental model defined by equation (3). A corrected stress amplitude $\frac{\Delta s^*}{2}$, that accounts for out-of-phase effects, is thus defined as the one corresponding to the integration of equation (3):

$$\left(\frac{\frac{\Delta s^*}{2}}{M^*} \right)^{-\beta^*} = \int_{cycle} \frac{\beta^*}{4\ B^*} \left(\frac{J\left(\underset{\sim}{\mathbf{s}}(t) - \underset{\sim}{\mathbf{X}}\right)}{M^*} \right)^{\beta^*-1} J\left(\dot{\underset{\sim}{\mathbf{s}}}(t)\right)\ dt \tag{7}$$

Equation (7) defines the amplification factor f ($f = \frac{\Delta s^*}{\Delta s} > 1$) applied to the stress-amplitude. This correction will result in a decrease of the number of cycles to failure in case of strongly out of phase loadings.

Syntax:

```
**process fatigue_rainflow
  *var name
[ *type scalar | tensor ]
[ *reverse  cmax ]
[ *mode with_a | simple ]
[ *use_mises ]
[ *mean_stress  ( standard | variant | tos ) ]
[ *normalized_coeff ]
```

```
[ *out_of_phase ]
[ *infinity_is  infty ]
[ *scale lin | log ]
```

*var *name* is the name of the stress component used in the fatigue damage equation. This component may be a tensor (eg. sig) or a simple scalar quantity (eg. uniaxial stress values, read from an ASCII file, using the ***data_source ASCII command)

*type this optional command allows to specify the type (tensor or scalar) of the stress component (default is tensor)

*reverse as described above, the post-processor first decomposes the input stress path in elementary cycles using a multi-axial rainflow procedure. Then, the number of cycles to failure NF*i* associated to each elementary cycle is computed, even if only the *cmax* most damaging cycles are stored in the results files (default value is *cmax*=3)

*mode coefficient a in the fatigue equation (1) is mainly used for non-linear cumulation with the NLC_ONERA option (cf. *process onera at page 4.78), but may be skipped to compute the number of cycles to failure. This strategy is activated when *mode simple is declared, and in this case the following simplified fatigue equation is used:

$$N_{fi} = \frac{\langle \sigma_u - \sigma_i^{max} \rangle}{\langle \sigma_i^a - \sigma_l(\bar{\sigma}_i) \rangle} \left[\frac{\sigma_i^a}{M(\bar{\sigma}_i)} \right]^{-\beta} \tag{8}$$

Note that the above expression is equivalent to equation (1) if the M coefficient is modified according to the following equation:

$$M^{mode\ simple} = \left(\frac{1}{a\ (\beta+1)} \right)^{\frac{1}{beta}} M^{mode\ with\ a} \tag{9}$$

*use_mises when this option is used the maximum value σ_i^{max} is calculated as the maximum of the von mises invariant of tensor $\underset{\sim}{\sigma}$ (default is the maximum principal stress)

*mean_stress this keyword allows the choice of the mean stress correction method (ie. default, variant or tos) as detailed above. Default mode is default.

*normalized_coef this option is mandatory for anisothermal loadings. In this case a normalized stress tensor $\underset{\sim}{\mathbf{s}}(t) = \frac{1}{\sigma_n(T(t))}\ \underset{\sim}{\sigma}(t)$ is used to compute N_f by equation (1). When a σ_n normalization coefficient is not explicitly defined in the material file, the stress input is normalized by σ_u. In both cases, other coefficient should be given for the normalized stress $\underset{\sim}{\mathbf{s}}$.

*infinity_is this command is used to define a maximum for the N_f value stored in the results file (default is *infty*=1.e12).

*scale log with this option $log(N_f)$ is stored in the results file instead of N_f (default mode *scale lin), a choice that may be more practical to draw contour plots.

```
****post_processing
 ***local_post_processing
  **process fatigue_rainflow
```

Example:

Example with the basic normalization mode used for anisothermal loadings. In the material file the only coefficient that depends on temperature is `sigma_u`, and the values given will be used to normalize the stress input. Note also that for other coefficients (`sigma_l`, `M` ...), *normalized* values are declared in the material file, in agreement with the `*normalized_coeff` option selected in the input file.

```
**process fatigue_rainflow
 *var sig
 *mode with_a
 *mean_stress variant
 *normalized_coeff
```

```
***post_processing_data
 **process fatigue_rainflow
   sigma_u temperature
   800.0   20.0
   700.0   200.0
   400.0   1000.0
   sigma_l 0.2
   M       20.0
   beta    3.0
   a       0.1
   b1      0.05
   b2      0.05
***return
```

Example:
Same as before, but with definition of an explicit normalization coefficient `sigma_n`, instead of the default normalization by `sigma_u`. Note that in this case the value given for `sigma_u` is *normalized*.

```
**process fatigue_rainflow
 *var sig
 *mode with_a
 *mean_stress variant
 *normalized_coeff
```

```
***post_processing_data
 **process fatigue_rainflow
   sigma_n temperature
   800.0   20.0
   700.0   200.0
   400.0   1000.0
   sigma_u 1.0
   sigma_l 0.2
   M       20.0
   beta    3.0
   a       0.1
   b1      0.05
   b2      0.05
***return
```

Example:

Same as before but with the additional definition of `M` and `beta` as temperature-dependent coefficients. In this case calculation of an equivalent temperature by the Taira method described above will be automatically triggered.

```
**process fatigue_rainflow
 *var sig
 *mode with_a
 *mean_stress variant
 *normalized_coeff

***post_processing_data
 **process fatigue_rainflow
   sigma_n temperature
   800.0   20.0
   700.0   200.0
   400.0   1000.0
```

```
                    ****post_processing
                     ***local_post_processing
                      **process fatigue_rainflow
   sigma_u 1.0
   sigma_l 0.2
   M       temperature
   20.0    20.0
   15.0    200.0
   5.0     1000.0
   beta    temperature
   3.0     20.0
   3.0     200.0
   5.0     1000.0
   a       0.1
   b1      0.05
   b2      0.05
***return
```

**process fatigue_S

Description:
Using the stress as critical variable, the formula implemented by this option predicts a fatigue life (Wöhler curve). Even though the criterion can be applied to any variable contained in the problem, the description of the equations is made in terms of stress.

The model is capable of taking into account a mean stress and a multi-axial loading history. The critical variables are the stress amplitude and the maximum stress (with generalization pertinent for multi-axial values), calculated over the time period specified by the user.

The model is written in terms of reduced stress (The components of the stress divided by the last stress under monotonic loading σ_u). The stress amplitude is noted **SEQ** , and the equivalent maximal stress is **SMAX** , and the mean trace is **TRMEAN** . The post computation returns a rupture in one cycle if **SMAX** attains the ultimate stress σ_u, and an infinite number of cycles if **SEQ** is less than the fatigue limit σ_l. Otherwise the number of cycles to failure N_f is defined as follows:

$$N_f = \frac{1}{a(\beta+1)} \left\langle \frac{\sigma_u - \mathbf{SMAX}}{\mathbf{SEQ} - \sigma_l'} \right\rangle \left(\frac{\mathbf{SEQ}}{\overline{M}} \right)^{-\beta}$$

with $\sigma_l' = \sigma_l(1. - b_1 * \mathbf{TRMEAN})$; $\overline{\mathrm{M}} = \mathrm{M}(1. - \mathrm{b}_2 * \mathbf{TRMEAN})$

Syntax:

```
**process fatigue_S
  *var name
 [*type scalar | tensor ]
 [*mode with_a | simple ]
 [*normalized_coeff ]
 [*use_eigen2 ]
 [*scale lin | log ]
 [*range section ]
```

name is the name of the subject variable if the computation (one would expect `sig` here). Using the extra keyword `simple` after the option `*mode`, the post-computation uses a simpler formulation where the ratio $1/a(\beta+1)$ is included in the coefficient `M`.

The coefficients `sigma_l`, `M`, `b1` and `b2` can be optionally given in normalized form. If they are normalized (using the option `*normalized_coeff`) the expected names in the material files are changed to `N_sigma_l`, `N_M`, `N_b1` and `N_b2`.

The other coefficients necessary to input are `beta` and `a` if the option `with_a` is given.

By default the equivalent max stress SMAX is the von Mises stress. Using the option `*use_eigen2` makes SMAX equal the maximal eigen stress S_I. Caution: in the `**process fatigue_rainflow` the default equivalent max stress SMAX is the maximal eigen stress S_I.

Number of cycles to failure is given in the output, with the variable named `NF_S`. The user can ask that a logarithmic scale be used with the option `*scale`. If so the output will be named `LNF_S`.

Example:

```
**process fatigue_S
  *var sig
  *scale log

% The following syntax must be used in the material file :
**process fatigue_S
   M       2400.
   beta    5.
   sigma_l 120.
   sigma_u 450.
   b1  0.002
   b2  0.03
```

**process format

Description:
This post processor is used to produce an "export file" containing the specified variables for all the maps being run. The output file is a formatted ASCII file. The variables should exist in the file specified by the **file command (see page 4.26). For instance, the command *list_var U1 should not be given with **file integ. Note: this post processor also exists for ***global_post_processing, with similar options (see page 4.122).

Syntax:

```
**process format
 *file file
 *list_var name1 ... nameN
[ *precision digits ]
[ *optimizer ]
[ *blank_line ]
```

where *file* denotes the name of the output file, and *name1*, ..., *nameN* is a list of scalar variable names to output. The values are stored point by point for each map. The option *optimizer inhibits the writing of the character string # time name1 ... nameN. The option *precision determines the number of *digits* with which the results will be written (default is 10), and the option *blank_line writes an empty line between the output of each location. Some plotting programs, for instance gnuplot, may need this.

Example:
The following example

```
**output_number 1-3
**process format
 *file myfile
 *list_var U1 U2
```

will produce a file called myfile like this:

```
#    time   U1     U2
      1     0      0
      2     0      0
      3     0      0
      1     0      0.01
      2     0      -0.01
      3     0      0
      1     0.01   0.01
      2     -0.01  -0.01
```

```
****post_processing
 ***local_post_processing
  **process format
```

```
3     0      0
...
```

Adding the *blank_line and *optimizer options will give

```
1     0      0
2     0      0
3     0      0

1     0      0.01
2     0      -0.01
3     0      0

1     0.01   0.01
2     -0.01  -0.01
3     0      0

...
```

**process function

Description:
This post processor allows the user to write an interpreted function of scalar variables.

Syntax:

```
**process function
  *output name
  *expression expression
```

name is the name of the resulting variable to be generated by the function evaluation. *expression* is the function expression to be input (see the function reference on page 6.2). A semicolon must always mark the end of the function.

Example:

```
% this example computes FMAX_sig ...
**process fmax
  *list_var sig
% and uses it in the calculation of the variable Pr.
**process function
  *output Pr
  *expression 1.-exp(-(FMAX_sig)/412.);
```

**process HCF

Description:
This post-computation gives an evaluation of the equivalent stress to compare with the fatigue limit, in order to define the high cycle fatigue (HCF) resistance.

Four different criteria are implemented for HCF. The variables which provide the basis of the criterion are the hydrostatic pressure (for each of them), and a stress amplitude in terms of the von Mises invariant (for three of them), or in terms of a shear (in one case). In the following, the stress amplitude is designated $Dsig$ (and defined in relation to stresses tensor), p_{max} the maximum hydrostatic pressure, and p_{mean} the mean value of hydrostatic pressure ($p_{mean} = 0.5(p_{max} + p_{min})$.

The following criteria are implemented for the HCF model:

- The criterion from Sines (mode `SI`) uses a coefficient b to calculate the equivalent stress:

$$\sigma_e q = Dsig + bp_{mean}$$

- The criterion due to Crossland (mode `CR`) uses a coefficient b to calculate its equivalent stress:

$$\sigma_e q = (1-b)Dsig + bp_{max}$$

- The criterion from Dang Van uses a coefficient b. Its characteristic, if we compare the two first two models, resides in the combination of two variables at the same time. Two versions are implemented:
  - The classical model of Dang Van (mode `DV`) searches for the maximum value in all the physical space directions ($\vec{n}$), at all instants t_i of the equivalent stress. This is constructed from the current shear stress amplitude τ and from the hydrostatic stress:

$$\sigma_e q = \max(\vec{n}) \min(t_i)(\ 2(1-b)\ \tau(t_i) + 3b\ p(t_i)\ )$$

  - An alternative formulation of the Dang Van criterion (mode `DV2`) is also implemented. Following the same philosophy as previously, this modification provides a simpler evaluation because it uses a von Mises stress measure. Knowing the stress amplitude one can calculate the value σ_0 corresponding to the loading path "center." The critical variable $DJ_2(t_i)$ then corresponds to the the von Mises invariant of the difference in current stress and σ_0:

$$\sigma_e q = \max(t_i)(\ 2(1-b)\ DJ_2(t_i) + 3b\ p(t_i)\ )$$

Syntax:

```
****post_processing
 ***local_post_processing
  **process HCF
```

```
**HCF
  *mode SI | CR | DV | DV2
```

The user must choose the *mode* of operation (`SI`, `CR`, `DV`, or `DV2`). The coefficient b must be defined in the material file. The post-computation produces an output for each input point at each time, which is the equivalent stress. The output variable name is `HCF_`*mode*.

Example:

```
**process HCF
  *mode DV2

% With the following definition in the material file
**process HCF  DV2
   b  0.3
```

**process initiation

Description:
This post processor is used to estimate the time to crack initiation. For a cycle the initiation damage is calculated by the following expression:

$$I_a = \frac{1}{c}\int_{cycle} \left\langle \frac{< SII - Sla(1-Dox)}{(1-Dox) - S^{eq}_{max}} \right\rangle^b dN$$

with

$$Sla = Sla0(1 - h\overline{\mathbf{Trace}(S)})$$

and $Dox = \sqrt{N/Nox}$ if we're taking into account the effect of oxidation, $Dox = 0$ otherwise.

A single output map is generated for the total history of loading. The number of cycles to initiation N_a which corresponds to the moment where the damage attains a value of 1. The variable name generated is `Na` in the *problem*`.utp` file. The number of oxidation cycles ahead of the start of initiation (stress lower than the fatigue limit) is also saved with the name `Na-ox`.

Syntax:

```
**process initiation
  *var name
  *type scalar | tensor ]
 [*normalized_coeff]
 [*oxidation [section1]]
 [*range section2]
```

name is the variable name. It is systematically normalized by the coefficient `sigma_u` given in the material file.

The option `*oxidation` is optional. If it is present, it can be followed by the number of section containing the user input for oxidation. Otherwise the oxidation will be initialized with its default values.

For multi-dimensional loading the amplitude calculation will be made using a `range` type post-computation. In order to be able to input options to the range processor, the user can give a section number for that user input after the `*range` keyword.

The coefficients `Sla0` and `h` are given normed or not. If they are normalized (the option `*normalized_coeff` was given), the expected names in the material file will be `N_Sla0` `N_h`. The other coefficients which must be input are `b` and `c`.

Example:

```
% a complete example
**process initiation
  *var sig
  *oxidation
  *norm
  *normalized_coeff

%with the following syntax in the material file :
**process initiation
  sigma_u  130.
  N_h      5.2
  N_Sla0   0.023
  b  2.
  c  3500.
```

**process integrate

Description:
This post calculation computes the integral of some fields with respect to a variable (time or another variable).

Syntax:

```
**process integrate
  *list_var names base
```

names is the list of variables to integrate with respect to *base*.

Example:
The following example computes the integral of `U2` with respect to time. It will generate the fields `I_time_U1` `I_time_U2`.

```
****post_processing
 ***local_post_processing
  **file node
  **nset ALL_NODE
  **process integrate
   *list_var U1 U2 time
****return
```

**process LCF

Description:
This process is used to apply a cumulation model for the life prediction under creep-fatigue interaction. The reference number of cycles to failure in pure fatigue and in pure creep must have been previously computed. The number of cycles to failure under creep-fatigue loading is defined as N_r, from the number of cycles to failure under creep N_c and the number of cycles to failure N_f . Several cumulation rules can applied, according to users choice:

- linear cumulation, LC

$$\frac{1}{N_r} = \frac{1}{N_c} + \frac{1}{N_f}$$

- bilinear cumulation, BLC

  A "knee-point" is defined in the cumulative diagram, from the numerical value of the material coefficients Kc and Kf (both values must be between O and 1).

$$\text{if } \frac{N_c}{N_f} \geq \frac{K_c}{K_f} : \quad \frac{1}{N_r} = \frac{1}{N_f} + \frac{1-K_f}{K_c}\frac{1}{N_f} \tag{10}$$

$$\text{if } \frac{N_c}{N_f} \leq \frac{K_c}{K_f} : \quad \frac{1}{N_r} = \frac{1}{N_c} + \frac{1-K_c}{K_f}\frac{1}{N_c} \tag{11}$$

  The resulting cumulation rule must :

  - be equivalent to the linear cumulation rule if $K_c + K_f = 1$

  - predict lower life if $K_c + K_f < 1$

  - predict longer life if $K_c + K_f > 1$

- nonlinear cumulation, NLC
  The cumulation rule uses the quantities $C = \frac{1}{N_c}$ and $F = \frac{1}{N_f}$ to compute damage evolution from D_i to D_f in each cycle, according to:

$$C = (1 - D_i)^{k+1} - (1 - D)^{k+1} \tag{12}$$

$$F = \left[1 - (1 - D_f)^{\beta+1}\right]^{1-\alpha} - \left[1 - (1 - D)^{\beta+1}\right]^{1-\alpha} \tag{13}$$

  The resulting cumulation rule predicts lower lives than linear cumulation does.

- another nonlinear cumulation, NLC_ONERA
  This is the same cumulation rule as previously given. The difference rests in the the coefficient α which is constant in the mode `NLC` and calculated as follows in this mode:

$$\alpha = 1 - a * \left(\frac{Dsig/2 - \sigma_l}{\sigma_u - \sigma_{max}} \right) \tag{14}$$

  with

$$\sigma_l = \sigma_{l0}(1 - b1 * \overline{tr(\sigma)}) \tag{15}$$

Syntax:
The syntax for this criterion is as follows.

```
**process LCF
  *mode LC | BLC | NLC | NLC_ONERA
  *fatigue name1 section1
  *creep name2 section2
 [*initiation name3 section3 ]
 [*scale lin | log ]
```

The user chooses his method of cumulation among the key words `LC`, `BLC`, `NLC` and `NLC_ONERA` after `*mode`. *name1* is the name of the fatigue processor to apply, and *name2* the name of the creep processor. Those which are currently available for fatigue are `fatigue_S`, `fatigue_E` and `fatigue_EE`. The current creep processor is uniquely `creep`. Plug-ins could however be created for new models of either of these. For each the user indicates after the processor type a number of post processing section containing the input for that processor.

The user can add an initiation phase, of which the name and the post processing section are specified after the option `*initiation`. Only the post-computation `initiation` is currently available. The number of cycles N_a is then taken into account in the cumulation rule.

The model `LC` does not require any material coefficients. Two coefficients `Kc` and `Kf` are necessary for the model `BLC`, three coefficients `k`, `alpha` and `beta` for the model `NLC`, and five coefficients `k`, `beta`, `a`, `sigma_l0` and `b1`, are required for the model `NLC_ONERA`. Some of these coefficients appear in the criterions for creep or fatigue. In this case the reading will attempt to find the coefficients in the respective creep or fatigue sections. For example, the coefficient k of the mode `NLC` will be read in the section `**process creep`, if it is present, otherwise in the section `**process LCF` of the material file.

The number of cycles to failure N_r is called `NR_`*mode* in the output variables. The user can also ask for a logarithmic scale which will re-name the output to `LNR_`*mode*. The number of cycles to failure N_f, N_c and N_a are also stored in the output (see these processors to find the specifics of variable naming).

Example:
A complete example with non linear cumulation rule follows:

```
**process LCF
  *mode NLC
  *fatigue fatigue_S 2
  *creep creep 2
  *scale log

% with in an other ****post_processing section
**process creep
  *var sig
**process fatigue_S
  *var sig
  *range 2
**range
  *var sig
  *method 2
  *alpha 0.2

% With the following definition in the material file
% beta and k are readed in their specific section
**process LCF
  alpha  0.9
**process fatigue_S
  M       980.
  beta    2.5
  sigma_l 70.
  sigma_u 180.
  b1      0.003
  b2      0.003
**process creep
  S0 0.
  A  420.
  r  10.
  k  30.
```

**process make_field

Description:
This post is used to create field variables from some other source than directly from a results database. For example, a uniform temperature can be imposed (satisfying post computations requiring that variable be present), or for doing analytical solutions or graphing over a mesh body.

Syntax:
The following syntax summary is available:

```
**process make_field
  *var     name1
[ *single_value val ]
[ **single_field scale-val  fname ]
[ *constant_values ]
   time#  val#
   ...
[ *constant_field ]
   time#  scale-val#  fname#
   ...
```

Multiple instances of `*constant_values` and `*constant_field` can be used to mix uniform fields and fields loaded in from an external file.

Example:
There are examples of this process in the tests `plast3.inp` and `hot_spot.inp` under `test/Post_test/INP`. An excerpt of the `hot_spot.inp` test case follows, and is a good example of how one can use FEA post processors as a general visualization tool for spatial functions. The data for that test is generated in a small Perl script `hot_spot.pl`.

```
****post_processing
 ***data_source mesh_only
  **format Z7
  **open hot_spot.geof
  **maps  0. 1. 2. 3.
 ***local_post_processing
  **process make_field
   *var X
   *constant_values
     0.    0.2
   *constant_field
     1.    2.    hot_spot.dat
     3.    1.    hot_spot.dat
```

**process mat_sim

Description:
This post-processor re-simulates material behavior given a strain and parameter (e.g. temperature) history. This post computation is most useful with very large structures and with imported results files. For example, ABAQUS ODB files are much larger than the Z-post files for the same stored results. In that case only the strain need be stored from the FEA run, and additional material state variables can be generated as need-be using this process. This can also be used to re-establish "named" material variables consistent with the material model instead of having SDV indexed names.

Input:
The full strain history is required in the variable name `eto##` listed as full tensor. The dimension will be consistent with the finite element geometry dimension. If the name `eto` is not directly available (for example with abaqus calculations using `*NLGEOM` we have `LE##`) a `copy` post process can be used beforehand to re-map the name.

If a `*parameter` option is given the parameters must be in input results as well. The user has the option to re-map those names using the `*change_param_name`. For example with abaqus results one would specify `*parameter NT11` and then re-map that with `*change_param_name NT11 temperature` .

Output:
The output variables are selected by the user. Before running, the `Zpreload` command can be run on the material file to get a listing of available material variables.

Syntax:
Basic options for this post are:

```
**process mat_sim
[ *change_param_name from-name to-name ]
[ *every_increment ]
[ *file       fname ]
[ *integration INTEGRATION  ]
[ *parameter p1  p2  ...  pN ]
[ *p_init       pval1  pval2  ...  pvalN ]
[ *rotation     ROTATION  ]
[ *save         v1  v2  ...  vN ]
```

The command descriptions are given below:

`*every_increment` selects if the output will be for every solution map, or just the result at the end of the calculation (or selected maps). If this is not careful use of `**output_number 1-999` can allow the user to investigate full material variables at specific points in time.

*change_param_name used to re name a parameter to be consistent with Z-mat names.

*file specifies the material file (as in standard Z-mat or Z-set input procedures).

*integration specifies the material integration rule to use rather than the default.

*parameter specifies parameter names which are to be added and having constant values.

*p_init specifies parameter values in sequence that they have been defined which represent the initial values. This is important because often the first FEA result map has already had some loading, and thermal strain for example will be calculated based on the temperature difference from the rest state before that results map.

*rotation specifies the material rotation.

*save gives a listing of the variables which are desired to be stored in the new results files.

Example:
The following example is from Post_test/INP/disk6_matsim.inp

```
***local_post_processing
 **output_number 1-999
 **file integ
 **elset ALL_ELEMENT

 **process mat_sim
    *every_increment
    *file disk6_matsim.inp

    *parameter temperature
    *p_init    800.

    *integration theta_method_a 1.0 1.e-8 100

    *save_variables
      evi11 evi22 evi33 evi12 evi23 evi31
      eto11 eto22 eto33 eto12 eto23 eto31
      sig11 sig22 sig33 sig12 sig23 sig31
      evcum
```

Note:
The material simulation should always be run from the beginning of the loading history, and following the complete loading path. This is because the material state variables will be tracked, and has substantial impact on the results.

**process max

Description:
This post computation gives at each node or Gauss point, the maximum value of a variable in the course of time. One output is produced for the totality of the time period for each variable specified.

Syntax:

```
**process max
  *list_var name1 ... nameN
```

The names *name1*, ..., *nameN* designate the list of scalar variables to treat. Outputs appearing in the file *problem*`.utp` will be named `MAX_`*name1*, ..., `MAX_`*nameN*.

Example:

```
% compute the maximum value of the U1 and U2 displacements
**process max
  *list_var U1 U2

% compute von Mises invariant of the stress tensor, ...
**process mises
  *var sig
% ... then its maximum value
**process max
  *list_var mises
```

**process fmax

Description:
This post computation produces for each variable specified an output per "map" of time, which corresponds to the maximum value achieved in the loading instant.

Syntax:

```
**process fmax
  *list_var name1... nameN
```

The names *name1*, ..., *nameN* designate the scalar variables to be treated. The output names which are generated will be of the form `FMAX_`*name1*, ..., `FMAX_`*nameN* (see the file *problem*`.utp`).

Example:

```
% this will provide FMAX_U1 and FMAX_U2
**process fmax
  *list_var U1 U2
```

**process min

Description:
This post computation gives at each node or Gauss point, the minimum value of a variable in the course of time. One output is produced for the totality of the time period for each variable specified.

Syntax:

```
**process min
  *list_var name1 ... nameN
```

The names *name1*, ..., *nameN* designate the list of scalar variables to treat. Outputs appearing in the file *problem*.utp will be named MIN_*name1*, ..., MIN_*nameN*.

Example:

```
% compute the minimum value of the temperature
**process min
  *list_var TP

% compute eigenvalues of the stress tensor, ...
**process eigen2
  *var sig
% 3 new variables are now available for the processing,
% sigp3 <= sigp2 <= sigp1
% ... then the minimum value of the two smaller ones
**process min
  *list_var sigp2 sigp3
```

**process fmin

Description:
This post computation produces for each variable specified an output per "map" of time, which corresponds to the minimum value achieved up to each loading instant.

Syntax:

```
**process fmin
  *list_var name1... nameN
```

The names *name1*, ..., *nameN* designate the scalar variables to be treated. The output names which are generated will be of the form `FMIN_`*name1*, ..., `FMIN_`*nameN* (see the file *problem*`.utp`).

Example:

```
% this will provides FMIN_TP
**process fmin
  *list_var TP
```

**process mises

Description:
This process is used to compute the von Mises invariant of a symmetric second order tensor. One value per node or Gauss point is obtained for each requested record. If `output_name` is not specified, the name of the output variable is obtained by suffixing `mises` to the input name.

Syntax:

```
**process mises
  *var name
[ *output_name output_name ]
```

Example:

```
% this will provide sigmises
**process mises
  *var sig
```

**process multirange

Description:
This post-processor can be used to analyze complex tridimensional loading paths in order to extract and count cycles (rainflow counting). It can only be applied to tensorial input variables.

The method is based upon multisurface theories of plasticity to detect closed loops (cycles) in the input stress or strain histories and their corresponding amplitude. It is an extension to the general case of multiaxial non-proportional loading paths of the well-known "rainflow technique" conventionally applied to uniaxial loadings. This post-processor can be used in conjunction with post-processors `post_fatigue_rainflow` (calculation of fatigue life for each sub-cycle) and `rainflow` (cumulation of damage generated by each sub-cycle).

Syntax:

```
**process multirange
 *var name
[*center ]
[*reverse number]
```

name is the input (tensorial) variable which is the subject of the range calculation.

number is the maximum number of sub-cycles given as output (default value is 3). The code gives a warning in the case where the number of cycles detected is greater than the one requested for output.

With the option `*center`, the code sends back additionally the center of the sphere containing the loading path associated with each subcycle.

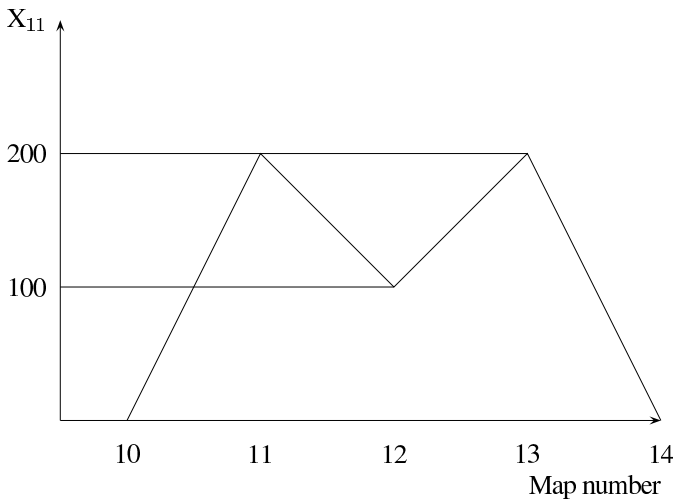
The output is the following:

- `ncyc` the number of cycles detected. If the algorithm detects more cycles than the number requested by the `*reverse` *number* keyword, the code return `ncyc` = *number*.
- for each cycle `i`:
  - D*iname* the amplitude of cycle i for input variable *name*,
  - ni*iname* is the number of the output map (within the ones selected by the `**output_number` command) corresponding to the beginning of cycle i,
  - nf*iname* is the number of the output map (within the ones selected by the `**output_number` command) corresponding to the end of cycle i,
  - optionally if the `*center` keyword is included, the components of tensor C*iname* at the center of the sphere including the loading path for cycle i.

Output cycles are stored by decreasing values of amplitude, such that the most meaningful cycles are included even if the number of requested cycles (specified by the `*reverse` keyword)

is smaller than the one detected by the rainflow algorithm. In the the event where the number of requested cycles is larger than the one detected, all corresponding quantities are zero.

Example:



Applying the following commands to the loading path of the above figure:

```
**output_number 10-14
**process multirange
  *var X
  *reverse 3
  *center
```

will yield as output:

- ncyc = 2
- D1X = 200, ni1X = 1, nf1X = 4, C1X11 = 100, C1X22 =0 ...
- D2X = 100, ni2X = 1, nf2X = 3, C2X11 = 150, C2X22 =0 ...
- D3X = 0, ni3X = 0, nf3X = 0, C3X11 = 0, C3X22 =0 ...

**process norm

Description:
The `norm` post processor is used to norm a tensor or scalar variable. It returns a variable of the same type (tensor or scalar) with name constructed by pre-fixing the original variable name(s) with an `N`.

Syntax:

```
**process norm
  *var name
 [*type tensor|scalar]
```

The user must give a material coefficient `sigma_u`. The coefficient can be constant or depend on other parameters.

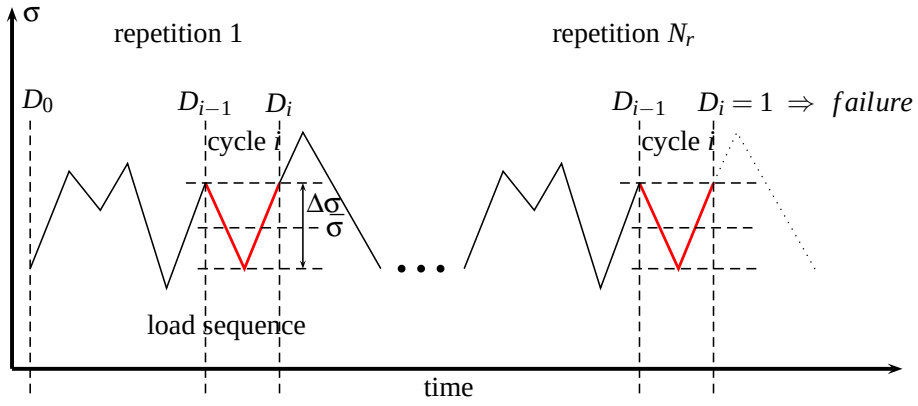
Example:

```
% this will provide 4 components :
% Nsig11, Nsig22, Nsig33 and Nsig12
**process norm
  *var sig

% the following syntax must be used in the material file:
**process norm
  sigma_u  170.
```

**process onera

Description:
This command can be used in replacement of `**process LCF` to perform damage cumulation within a rainflow counting procedure. Cumulation rules and material parameters are the same as in the case of `**process LCF`. Given a complex multiaxial load sequence the post-processor performs the following operations:

- for each point (integ point, node) decompose the input into elementary sub-cycles using the multiaxial rainflow algorithm (`**process multirange`, see page 4.75)
- for each elementary cycle, compute the number of cycles to failure according to the onera fatigue model (`*process fatigue_rainflow`, see page 4.47)
- optionally, if creep is included, compute the creep damage associated to each elementary cycle, using the `*process creep` model (page 4.32),
- performs damage cumulation, starting from an initial D_0 (that may be non-zero if a preloading is defined). As shown in the next figure, the input loading sequence is repeated until a damage value of $D = 1$ is obtained. The number of cycles to failure computed by the post processor, then corresponds to the maximum number of repetitions N_r applied for the current point. During each repetition of the loading sequence, the damage D_i associated to each elementary sub-cycle, is cumulated to the total damage D using the selected cumulation rule (options `LC`, `NLC` or `NLC_ONERA`).



Syntax:

```
**process onera
[ *reverse reverse ]
  *mode LC | NLC | NLC_ONERA
  *fatigue fatigue_rainflow section1
[ *creep name2 section2]
```

[*scale lin | log]

[*preload *b1-e1*[/*r1*] *c2-e2*[/*r2*] ... *bn-en*[/*rn*]]

[*cycle *b1-e1*[/*r1*] *b2-e2*[/*r2*] ... *bn-en*[/*rn*]]

The main difference with **process LCF is the *reverse keyword, that allows to specify the maximum number of sub-cycles stored in the calculation, within the ones detected by the rainflow counting algorithm (see the **process multirange command). Default value is *reverse*=3. Note that **process multirange gives out cycles with maximum amplitude first, such that output is always given for the *reverse* most damaging elementary cycles. However, regardless of the *reverse* number, damage cumulation is always done for all elementary cycles detected by the rainflow algorithm.

Contrary to **process LCF, the only fatigue model allowed is the one implemented in the **process fatigue_rainflow post-processor (see this command).

Note also, that the creep damage part is optional in this post-processor, and that cumulation can be restrained only to the fatigue damage due to sub-cycles detected by rainflow counting.

The optional command *cycle can be used to define a a complex cycle from the output maps available in the results file (that may or not be selected by means of the **output_number command see 4.21). Cycle definitions have the form *bi*-*ei*/*ri*, where *bi* and *ei* are map numbers defining respectively the beginning and the end of sequence number *i*. *r1* is the number of times this sequence should be repeated (default is *ri*=1, when this optional argument is not defined). An arbitrary number of sequences can be given, thus allowing to re-arrange output maps stored in results files in a loading sequence of arbitrary complexity. In this case, the number of cycles to failure, will correspond to the number of times this complex loading path can be repeated until failure.

Command *preload, with a syntax equivalent to *cycles, can also be used to define a preloading phase generating an initial damage before cycling occurs. In this case the number of cycles to failure calculated will correspond to the number of times the cyclic loading path can be repeated, taking into account a non-zero initial damage corresponding to the *preload definition.

Note that when *preload is defined, it is mandatory to add also a *cycle command to specify which subset of total input maps do correspond to the cycle definition. Also, map numbers given as argument to the *preload and *cycle commands are in fact *relative*, and depend on the **output_number definition specified before the current **process onera block, ie. those are ranks in the current selection.

Output is as follows:

- ncyc the number of elementary cycles detected by the rainflow algorithm.
- the number of cycles of failure NF*i* corresponding to fatigue damage associated with the *reverse* most damaging sub-cycles obtained by decomposition of the input sequence.
- if the option *creep is included the number of cycles to failure NC associated with creep damage for the whole loading sequence.

- the number of cycles to failure `NR_`*mode* obtained by cumulation of the previous damage mechanisms,
- the initial damage `Dpreload` corresponding to preloading sequences when the command `*preload` is used.

**process oxidation

Description:
This post-processor can be used to quantify the effects of oxidation which contributes to the mechanisms of damage.

We calculate, for a cycle, the oxidation damage I_{ox} with the following expression:

$$I_{ox} = \int_{cycle} \left(\frac{K0}{e0} exp\left(\frac{-Q}{RT} \right) \left[1 + \left\langle \frac{S - S_{lox}}{B} \right\rangle \right]^m \right)^2 dN$$

with

$$S = (1 - \alpha - \beta)\mathbf{Mises}(\sigma) + \alpha\mathbf{Eig1}(\sigma) + \beta\mathbf{Trace}(\sigma)$$

and

$$Slox = Slox0(1 - h\overline{\mathbf{Trace}(\sigma)}) + \overline{\mathbf{Trace}(\sigma)}$$

The output is generated as a single map for the whole loading history. The cycle number is N_{ox}, defined as the inverse of oxidation damage per cycle and denoted `Nox` in the output.

Syntax:

```
**process oxidation
  *var name
 [*type scalar | tensor ]
 [*delay ]
 [*norm
 [*normalized_coeff ]]
 [*isotherm ]
```

name is the variable name to treat. If the option `*norm` is specified the subject variable will be normed by the coefficient `sigma_u` furnished in the material file.

With the option `*delay` the lag stress S' will be used. The lag stress is computed from

$$dS' = \frac{S - S'}{\tau}$$

The coefficients `B`, `Slox0`, and `h` are to be given in the material file regardless of the use of normalization. If they are normalized (option `*normalized_coeff`) the expected names in the material file will be `N_B`, `N_Slox0` and `N_h`.

With the option `*isotherm` the temperature is read in the material file using a "coefficient" `temperature`. Otherwise the temperature must be included in the output files (use the `**save_parameter` output option in the FEA calculation.

alpha and beta are optional coefficients, having a zero value by default. The other coefficients which must be input are m, K0, e0 and Q. The additional coefficient tau must be supplied if the *delay option was input.

Example:

```
% a simple example
**process oxidation
  *var sig
  *norm

% The following syntax must be used in the material file :
**process oxidation
  sigma_u 850.
  B       46.5
  m       2.5
  Slox0   0.12
  h       3.5
  e0      50.e-06
  K0      1.42
  Q       214.
```

**process range

Description:
This post-processor calculates the amplitude of a scalar or tensorial variable from its history.

In the case of a tensorial variable, an invariant of the type von Mises is used to calculate the space distance in the six different dimensions, and the result is the diameter of the smallest sphere encompassing the point of interest's path during the loading.

Syntax:

```
**process range
  *var name
 [*type scalar | tensor ]
 [*method num ]
 [*alpha value ]
 [*center ]
 [*delta ]
```

name is the input variable which is the subject of the range calculation.

Two methods are available. By default, the first method is applied. If the first method fails, it will automatically roll over to the second method. This later depends on a coefficient `alpha` which must be between 0 and 1. Its default value is 0.2.

With the option `*center`, the code sends back additionally a tensor corresponding to the center of the sphere containing the loading path.

With the option `*delta`, implemented only with the `scalar` keyword, the code just deliver the difference of the prescribed variable between the last two increments.

The generated output will consist of the amplitude named after the input subject prefixed with a `D`. If the center tensor is output, the prefix `C` is added.

Example:

```
% this will provide Dtemper
**process range
  *type scalar
  *var temper

% this will provide DX
% using directly the second method
% and the center (CX11, CX22, CX33, CX12)
**process range
```

```
                                                    ****post_processing
                                                     ***local_post_processing
                                                      **process range

*var X
*method 2
*alpha 0.15
*center
```

**process neu_sehitoglu

Description:
This post-processor implements the thermo-mechanical fatigue model proposed by Neu and Sehitoglu [Neu89]. This model is a bit of a departure from the rest of the fatigue analysis post-processing in that it does not re-use components for LCF, oxidation and creep damages, but rather implements the specific forms defined in the referenced paper.

The damage is a linear summation of three components as per standard practice in thermo-mechanical fatigue studies.

$$D = D_{fatigue} + D_{oxidation} + D_{creep}$$

And the number of cycles to failure can be found from the cumulated damage over a number of studied cycles, N_c:

$$N_f = N_c/D$$

The interesting part of the Neu and Sehitoglu is their assignment of a damage phase factor, which accounts for the tendency for creep damage in in-phase loading (such as found in rotating machinery or pressure vessels at high temperature), and oxidation cracking in out-of-phase loading (such as around notches in constrained geometries).

$$\begin{aligned}
\phi_{\text{ox}} &= \frac{1}{t}\int_0^{t_c} \exp\left[-\frac{1}{2}\left(\frac{\dot{\epsilon}_{th}/\dot{\epsilon}_{mech}+1}{\xi_{ox}}\right)^2\right] dt \\
\phi_{\text{cr}} &= \frac{1}{t}\int_0^{t_c} \exp\left[-\frac{1}{2}\left(\frac{\dot{\epsilon}_{th}/\dot{\epsilon}_{mech}-1}{\xi_{cr}}\right)^2\right] dt
\end{aligned}$$

So the coefficients `xi_ox` and `xi_cr` control the width of oxidation and creep effects about pure out-of-phase and pure in-phase loading respectively.

Fatigue part:
The fatigue part can be determined in several ways. In general we look at the strain range as being the maximum shear strain range in from the cycle calculated as follows:

$$\Delta\gamma = \tfrac{1}{2}(\Delta e_1^{loading} - \Delta e_3^{loading}) + \tfrac{1}{2}(\Delta e_1^{unloading} - \Delta e_3^{unloading})$$

with the Δe_1 and Δe_3 terms being the 1st and 3rd ordered eigenvalue of the strain increment from across the cycle. Note that the default method does not take the critical plane of all loading points in the cycle, but rather just the points of the major temperature cycle. If the `*use_loading` or `*use_unloading` option is given, we use only one term above instead of the average. Normally for a stabilized cycle the loading and unloading parts should be symmetric.

In the above strain range calculation, we also allow for either the total mechanical strain to be used (model `neu_sehitoglu`) or the plastic strain range (model `neu_sehitoglu_evi`), and currently there is no HCF (stress based) part taken into account.

Lifetime is predicted via:

$$N_f^{fatigue} = \frac{1}{D_{fatigue}} = \frac{1}{2}\left[\frac{\Delta\gamma}{2\epsilon'_c}\right]^{1/c}$$

where we have the coefficients `eps_f` and `c` . If temperature dependent coefficients are given for the fatigue part, the values calculated at the cycle average temperature are used.

Oxidation part:
The oxidation damage is calculated via the following equation:

$$D_{ox} = \frac{1}{N_f^{oxide}} = \left[\frac{\overline{h}_f \delta_o}{B\phi_{ox}K_p}\right]^{-1/\beta} \frac{2(\Delta\epsilon_{mech})^{2/\beta+1}}{\dot{\epsilon}^{1-a/\beta}}$$

Which uses a term for the critical oxide depth:

$$\overline{h}_f = \frac{\delta_0}{(\Delta\epsilon_m)^2\,\phi_{ox}\,\dot{\epsilon}^a}$$

And an assumed parabolic oxidation rate:

$$K_p = \frac{1}{t_c}\int_0^{t_c} D_o \exp\left[\frac{-Q}{RT(t)}\right] dt$$

For the oxidation part the coefficients to be entered are `D_0`, `Q`, `B`, `beta`, `delta_0`, `a`, `h_cr` . Note that increasing `D_0` or `B` increases the damage effect, bigger `Q` will emphasize an abrupt change in high temperature degrading (but require a change in `B` to compensate), while decreasing either `delta_0` or `h_cr` will increase the oxide damage. These later parameters should be calibrated from primary metallurgical parameters from specialized oxidation tests. If the model is being used empirically then there is quite a great deal of redundancy in these parameters and the user should use caution.

Creep part:
The creep damage equation is:

$$D_{cr} = \frac{1}{N_f^{creep}} = \phi_{cr}\int_0^{t_c} Ae^{-\Delta H/RT}\left[\frac{\alpha_1 J(\boldsymbol{\sigma}) + \alpha_2 \frac{1}{3}Tr(\boldsymbol{\sigma})}{K}\right]^m dt$$

with $J(\boldsymbol{\sigma})$ being the von Mises equivalent stress. The creep damage will occur only in tensile loading under uniaxial conditions if $\alpha_1 = 1/3$ and $\alpha_2 = 1$ (these are the defaults).

For the creep part the coefficients to be entered are `alpha1`, `alpha2`, `A`, `K`, `m`, `dH` which should be obvious names from the above equation.

Output:
A large output set is generated by this processor, but with only one solution map for the whole history analyzed. The following summarizes the output variables:

`Nc` the number of cycles detected for validation. There may be zones where the number is incorrect in totally unloaded portions of a model.

`Nf, log_Nf` the life prediction based on the history so far, and log base 10 of that lifetime for flatter contour plots.

`D, Dox, Dcr` the "damage" so far in the loading history, with 0 being undamaged, and 1 full failure. D is total, Dox oxidation part, Dcr the creep part.

`extr_Nf-1 extr_log_Nf-1` extrapolations of the lifetime to account for the trend in evolving strain ranges.

`extr_N-1` the number of cycles forward in the extrapolation.

`range_ave` average value of the plastic strain range (or effective range used for LCF calculations).

`phi_ox_ave, phi_cr_ave` "phase factors" for the oxidation and creep effects as defined by sehitoglu. zero means that effect is not considered because of the type of loading.

Syntax:
Basic options for this post are:

```
**process neu_sehitoglu
[ *extrapolation_cycles n1 n2 n3 ]
[ *extrapolation_sequence n1 n2 n3 ]
[ *max_cycles maxc ]
[ *model_coef ]
    ...
[ *R R-value ]
[ *skip_first skip ]
[ *small_strain_rate sm-rate ]
[ *use_loading ]
[ *use_unloading ]
```

`*extrapolation_cycles` gives absolute cycle numbers which will be used to extrapolate the trend in fatigue life. This is used when cycles are not yet stabilized.

`*extrapolation_sequence` gives a series of cycles ending with the last detected cycle which will be used to extrapolate the trend in fatigue life. This is used when cycles are not yet stabilized.

`*max_cycles` limits output cycle number as essentially infinite life. This is convenient to limit the ranges for contour plotting.

`*R` sets the gas constant in case of different units. The default value is 8.3144e-3 KJ/(mol K).

*skip_first causes the first *skip* cycles to be left from consideration in order to limit damage prediction from large initial transient cycles.

*small_strain_rate gives a small strain rate which will be used in several locations to limit divide by zero conditions, including the strain rate effect and prediction of the constraint average.

*use_loading indicates that only the loading portion of the strain range will be used for fatigue. This would normally be detected as the heat-up side of a TMF cycle.

*use_unloading indicates that only the unloading portion of the strain range will be used for fatigue.

Additional controls are allowed for the CYCLE_TOOL part. Currently these are[3].

```
[ *width wid ]
[ *major_tensor tens-name ]
[ *range_tensor tens-name ]
```

*width gives a size control for detecting cycles. If too few cycles are detected the number is probably too large; too many cycles detected and the size is probably too small (default 10^{-4}). Units will be that of the major tensor.

*major_tensor select the name of the variable to be used for cycle detection (default sig).

*range_tensor select the name of the variable to be used for strain range calculations (default eto).

For the TMF cycle tool (default) the following additional commands are available (however *major_tensor and range_tensor are disallowed).

```
[ *total_strain_for_range  ]
[ *mechanical_strain_for_range  ]
[ *temperature_unit celcius|fahrenheit|kevin ]
```

*total_strain_for_range detect the cycles based on the total strain. Note for fully constrained test cases this will always be zero so the mechanical range should be used. Conversely for basically unconstrained conditions using the total strain will be a better choice.

*mechanical_strain_for_range use the mechanical strain for the range calculation.

Note:
If the coefficients D_0, Q, B, beta are not entered the oxidation part will be skipped without mention. Likewise if alpha1, alpha2, A, K are not entered the creep damage will be skipped. Partially entering these coefficients will cause an error message.

[3] section to be separated in next version

```
****post_processing
 ***local_post_processing
  **process neu_sehitoglu
```

Example:
The following example is from `Post_test/INP/sehitoglu1.inp`

```
***local_post_processing
  **file integ
  **elset ALL_ELEMENT
  **output_number 1-99999999

  **process neu_sehitoglu_evi
   *total_strain_for_range
   *model_coef
     eps_f  0.20
      c    -0.64
      b        1.0
      sig0_E  0.0
      xi_ox        1.0
      xi_cr        0.20
      a  0.0
      D_0       2000.
      Q        200.
      B    7.00e-03
      beta        1.5
      delta_0    2.00e-07
      h_cr        500.
      alpha1        0.33
      alpha2        1.0
      A    1.00e+08
      K  temperature
       1.00e+06  0.
       1.00e+06  450.
       20.0      600.
       20.0      1200.
      m     2.0
      dH    200.
```

**process swt

Description:
This process allows to calculate number of cycles to failure according to a unified strain-stress fatigue model based on the Smith-Watson-Topper criterion.

The fatigue life is calculated by solving the following equation:

$$\sigma_{eff} = F(N_f)$$

where the effective stress σ_{eff} is calculated as a product of the stress and strain amplitudes multiplied by a correction factor depending on the mean stress:

$$\sigma_{eff} = \sqrt{E\frac{\Delta\sigma}{2}\frac{\Delta\epsilon}{2}}f(\frac{\Delta\sigma}{2},\bar{\sigma})$$

In the above formula E is the material Young's modulus, while:

- $\frac{\Delta\sigma}{2}$ is the multiaxial stress amplitude (measured in terms of the von mises J2 stress invariant) as calculated by the `**process range` post-processor,
- a special treatment is needed for the calculation of the multiaxial strain amplitude $\frac{\Delta\epsilon}{2}$ in order to be compatible with the uniaxial case (because of nonzero strains that arise in directions orthogonal to the uniaxial stress direction). The expression used to calculate $\frac{\Delta\epsilon}{2}$ is the following one:

  $$\frac{\Delta\epsilon}{2} = \frac{\Delta\sigma}{2E} + \frac{\Delta\epsilon^p}{2}$$

  where the plastic strain amplitude $\frac{\Delta\epsilon^p}{2}$ is computed either using the `**process range` post-processing, or by means of a cyclic hardening relation $\Delta\epsilon^p = f(\Delta\sigma)$, depending on the option selected (see syntax). Note that when the `range` post-processor is used, the distance measuring the amplitude in the deviatoric strain space is calculated by:

  $$J2^*(\underset{\sim}{\epsilon}^p) = \sqrt{\frac{2}{3}\,\mathrm{dev}(\underset{\sim}{\epsilon}^p) : \mathrm{dev}(\underset{\sim}{\epsilon}^p)}$$

  the $\frac{2}{3}$ factor being used instead of the conventional $\frac{3}{2}$ of the mises stress invariant, for compatibility with the uniaxial case:

  $$\underset{\sim}{\epsilon}^p = (\epsilon^p_{11},\epsilon^p_{22},\epsilon^p_{33},\epsilon^p_{12},\epsilon^p_{23},\epsilon^p_{31}) = (\epsilon^p_{11},-0.5\epsilon^p_{11},-0.5\epsilon^p_{11},0,0,0)$$

  such that:$J2^*(\underset{\sim}{\epsilon}^p) = \epsilon^p_{11}$ in this case.
- in the multiaxial case, the mean stress value $\bar{\sigma}$ is calculated as the mean trace of tensor $\underset{\sim}{\sigma}$:

  $$\bar{\sigma} = \frac{1}{2}\left[\max(\mathrm{Trace}(\underset{\sim}{\sigma})) + \min(\mathrm{Trace}(\underset{\sim}{\sigma}))\right]$$

- the mean stress correction factor $f(\frac{\Delta\sigma}{2},\bar{\sigma})$ is defined as the following function of $\bar{\sigma}$:
  - traction : $\bar{\sigma} > 0$

    $$f(\frac{\Delta\sigma}{2},\bar{\sigma}) = \left(1 + \frac{\bar{\sigma}}{\left(\frac{\Delta\sigma}{2}\right)}\right)^{n}$$

    where n is a model coefficient.
  - compression : $\bar{\sigma} < 0$

    $$f(\frac{\Delta\sigma}{2},\bar{\sigma}) = \left(1 - \frac{\bar{\sigma}}{\left(\frac{\Delta\sigma}{2}\right)}\right)^{-1}$$

    which yields an effect similar to the one predicted by the Sines criterion, or alternatively $f(\frac{\Delta\sigma}{2},\bar{\sigma}) = 1$ when the `*skip_compression` option is specified. The latter option ignores any beneficial effect of a compressive stress, and may be too pessimistic.
  - traction $\bar{\sigma} > 0$ and option `*chaboche_2012` [Chab12]:

    $$f(\frac{\Delta\sigma}{2},\bar{\sigma}) = \begin{cases} 1 + b_1 \frac{\bar{\sigma}}{\left(\frac{\Delta\sigma}{2}\right)} & \text{if } t^* \geq \frac{\bar{\sigma}}{\left(\frac{\Delta\sigma}{2}\right)} \\ 1 + (b_1 - b_2)\, t^* + b_2 \frac{\bar{\sigma}}{\left(\frac{\Delta\sigma}{2}\right)} & \text{otherwise} \end{cases}$$

    where b_1, b_2 and t^* are model coefficients.

Several definitions of $F(N_f)$ are allowed depending on the options:

- the first possibility (keyword `*expression manson`) is to derive $F(N_f)$ from a prior Manson-Coffin model calibration on symmetric cyclic tests (loading factor R=-1, $\bar{\sigma} = 0$).

  $$\frac{\Delta\epsilon}{2} = \frac{\Delta\epsilon^p}{2} + \frac{\Delta\epsilon^e}{2} = AN_f^{-\alpha} + BN_f^{-\beta}$$

  which yields by replacing $\frac{\Delta\epsilon^p}{2} = AN_f^{-\alpha}$ and $\frac{\Delta\epsilon^e}{2} = BN_f^{-\beta}$ in the effective stress formula:

  $$F(N_f) = E\sqrt{ABN_f^{-\alpha-\beta} + B^2N_f^{-2\beta}}$$

  In that case model coefficients A, B, α, β are directly known from the Manson-Coffin calibration, and the only additional coefficient still left to define is the n exponent involved in the mean stress correction factor,
- a second possibility lets the user define an arbitrary expression for $F(N_f)$, using the keyword `*expression <FUNCTION>`, where `<FUNCTION>` is a valid function definition. For example, an user-defined expression of $F(N_f)$ equivalent to the `manson` option could be specified by:

  ```
  *expression E*(A*B*Nf^(-alpha-beta) + B*B*Nf^(-2.0*beta)^(0.5);
  ```

  Note that it is mandatory that a variable named `Nf` or `nf` should be used in the function definition. Other function coefficients are assumed to be model coefficients and should be given in the post-processing material file.

Introduction of a fatigue limit effect

An additional model coefficient ϵ_D may be used to add a fatigue limit effect when the `manson` option is selected. In this case the Manson-Coffin model used during calibration is rewritten:

$$\frac{\Delta\epsilon}{2} = \frac{\Delta\epsilon^p}{2} + \frac{\Delta\epsilon^e}{2} + \epsilon_D$$

such that $N_f = \infty$ when $\frac{\Delta\epsilon}{2} < \epsilon_D$.

In that case the $F(N_f)$ expression used in the `swt` model becomes:

$$F(N_f) = E\sqrt{ABN_f^{-\alpha-\beta} + B^2N_f^{-2\beta} + \epsilon_D\left(2BN_f^{-\beta} + AN_f^{-\alpha} + \epsilon_D\right)}$$

Syntax:

```
**process swt
  *var name_sig
 [*type scalar | tensor ]
 [*expression manson | func_exp ]
 [*derivative func_deriv ]
 *plastic_strain name_ep
| *cyclic hardening function func_hard
| *cyclic hardening file fname sig_col eto_col
 [*precision prec ]
 [*iter iter ]
 [*skip_compression ]
 [*chaboche_2012 ]
```

name_sig is the name of the variable used to store the stress in the results file (one would expect `sig` here).

The `*expression` command allows to define the $F(N_f)$ function. Default is `*expression manson` that defines a $F(N_f)$ expression derived from a Manson-Coffin calibration as detailed above. A user-defined expression may alternatively be given by means of argument *func_exp*.

The `*derivative` command allows to define the derivative $\frac{dF(N_f)}{dN_f}$ of $F(N_f)$ in the *func_deriv* argument, and may be needed when a user defined function is given after `*expression`. In this case a Newton-Raphson method is used to solve the $\sigma_{eff} = F(N_f)$ equation to calculate N_f. Otherwise, a slower dichotomy method is used. Note that the `*derivative` command is not needed in the default `*expression manson` mode, and that the Newton-Raphson method is always used in this case.

The `*plastic_strain` command is used to define the name of the variable where the plastic strain $\underset{\sim}{\epsilon}^{\mathbf{p}}$ is stored in the results files. In this case the $\frac{\Delta\epsilon^p}{2}$ amplitude needed in the effective stress expression will be calculated by applying the `range` post-processor on the variable specified by the argument *name_ep*.

Alternative specification of $\frac{\Delta\epsilon^p}{2}$ calculation can be given by means of the `*cyclic_hardening function` command. In that case a function $\frac{\Delta\epsilon^p}{2} = f(\frac{\Delta\sigma}{2})$ is expected for the *func_hard* argument.

$\frac{\Delta\epsilon^p}{2}$ may also be interpolated from cyclic hardening curve results by means of the `*cyclic_hardening file` command. In this case argument *fname* is the name of the file containing the cyclic hardening curve, and *sig_col*, *eto_col* are the column numbers in this file used to store $\frac{\Delta\sigma}{2}$ and $\frac{\Delta\epsilon^p}{2}$ respectively. Note that the half-amplitude is expected for both stress and strain values.

The optional `*precision` command defines the precision required when solving the $\sigma_{eff} = F(N_f)$ by either dichotomy or Newton-Raphson. Default value is 10^{-4}. Similarly, the maximum number of iterations allowed may be given after `*iter` command (default is 500). In general, the Newton-Raphson method converges in at most a few dozens of iterations, while the dichotomy method may need larger values depending on the precision required.

When the optional `*skip_compression` command is given, the mean stress correction factor $f(\frac{\Delta\sigma}{2}, \bar{\sigma})$ is set to 1.0, negative value of the mean stress $\bar{\sigma}$.

Default output gives the number of cycles to failure in a variable named `Nf`, while the effective stress is stored in a variable named "*name_sig*`_swt`", obtained by concatenation of the stress tensor name "*name_sig*" specified with the "`_swt`" character string.

The optional `*full_output` command will add the following results to the output:

- the stress amplitude under the name "*name_sig*`_alt`"
- the average stress under the name "*name_sig*`_ave`"
  where "*name_sig*" is the name of the stress tensor specified
- the elastic and plastic amplitudes in variables `eel_alt` and `ein_alt`

Example:

```
% standard definition using the default *expression manson option
**process swt
   *var sig
   *plastic_strain ev
   *full_output

% The following syntax must be used in the material file in this case:
***post_processing_data
```

```
 **process swt
   young 98000.00
   alpha 0.7438  % mandatory Manson-Coffin coefficients
   A 165.e-2
   beta 0.0745
   B 0.8015e-2
   epsd 0.13e-2  % optional fatigue limit coefficient
   n 0.65        % mandatory mean stress correction coefficient

   % if *chaboche_2012
   b1 0.7 % mandatory mean stress correction coefficient
   b2 0.3 % mandatory mean stress correction coefficient
   t_star 0.4 % mandatory mean stress correction coefficient

% user definition of the F(Nf) function
**process swt
   *var sig
   *expression Au*Nf^(-au)+Bu*Nf^(-bu);
   % optional derivative specification to allow faster newton-raphson resolution
   *derivative -au*Au*Nf^(-au-1.0)-bu*Bu*Nf^(bu-1.0);
   *plastic_strain ev
   *full_output

% The following syntax must be used in the material file in this case:
% note the Au, au, Bu, bu coefficients needed depend of the user expression
***post_processing_data
 **process swt
   young 98000.00
   au 0.7438
   Au 165.e-2
   bu 0.0745
   Bu 0.8015e-2
   n 0.65

   % if *chaboche_2012
   b1 0.7 % mandatory mean stress correction coefficient
   b2 0.3 % mandatory mean stress correction coefficient
   t_star 0.4 % mandatory mean stress correction coefficient

% definition of a cyclic hardening function
**process swt
   *var sig
   % note that the function variable should be called "dsig"
   *cyclic_hardening function (dsig/K)^m;
   *full_output

% The following syntax must be used in the material file in this case:
```

```
                                                    ****post_processing
                                                     ***local_post_processing
                                                      **process swt

% note the K, m coefficients needed depend on the *cyclic_hardening user definition
***post_processing_data
 **process swt
   young 98000.00
   alpha 0.7438
   A 165.e-2
   beta 0.0745
   B 0.8015e-2
   epsd 0.13e-2
   n 0.65
   K 1000.
   m 0.07
```

**process trace

Description:
This post-computation calculates the trace of a second order tensor. One value per node or per Gauss point is added to the output records, for each solution map. The new variable name is constructed using the tensor name suffixed with `ii`.

Syntax:

```
**process trace
  *var name
```

Example:

```
% this will provide eplii
**process trace
  *var epl
```

**process transform_frame

Description:
This post calculation calculates existing variables in a new local frame. Variables can be either tensors or vectors.

Syntax:

```
**process transform_frame
  *local_frame type
[ *tensor_variables tensor_var1 tensor_var2 ...]
[ *vector_variables vector_var1 vector_var2 ...]
[ *suffix suffix ]
[ *output_variables name1 name2 ... ]
```

`*local_frame` specifies the local frame type. *type* can be euler, cartesian, cylindrical, spherical or be specified by a z7p script. Refer to `**local_frame` (page 3.170) for more details and syntax.

`*tensor_variables` is the list of tensor variables to transform.

`*vector_variables` is the list of vector variables to transform.

`*suffix` this suffix is added to the variable names (default is `-rot`);

`*output_variables` can alternatively be specified to specify a different output name.

Example:

```
****post_processing
 ***local_post_processing
  **file node
  **nset ALL_NODE
  **process transform_frame
     *local_frame cylindrical
       (1. 2.)
     *tensor_variables sig eto
     *vector_variables U
****return
```

**process tresca

Description:
This processor computes the Tresca criterion of a symmetric second order tensor. One output variable is generated for each point at each map. The output variable name is constructed from the input variable by adding `tresca`.

Syntax:

```
**process tresca
  *var name
```

Example:

```
% this will provide sigtresca
**process tresca
  *var sig
```

**process triax

Description:
This computes the triaxiality of a symmetric second order tensor. The expression is as follows

SMEAN/SEQ

One value is produced per input point. The new variable name will be constructed from the input variable by adding `triax`.

Syntax:

```
**process triax
  *var name
```

Example:

```
% this will provide sigtriax
**process triax
  *var sig
```

***global_post_processing

Description:
This command is used to define "global" post processing treatments to apply over sets of nodes and integration points. Global post processors produce global values saved in the file *problem*.post and/or in local (field) variables in the node or integration point files. This distinction is indicated by the sign ^{G} for global variable producing processors, and ^{L} for local variable producing processors. The reason for the distinction is that file data will be loaded in a manner which may be less efficient than the local post processing methods.

Syntax:

```
***global_post_processing
[ **deformed ] GLOBAL OPTION
[ **elset eset ]
[ **file file-key ]
[ **ipset ipset ]
[ **material_file fname ]
[ **nset nset ]
[ **output_number out-num-list ]
[ **process type ]
   ...
[ **undeformed ] GLOBAL OPTION
```

The different sub-options define the geometrical groups of concern, the time period, materials files, and the post treatments themselves. The majority of these options are exactly the same as for ***local_post_processing. The following *additional* options are relevant to the global post processing.

**deformed Use the mesh in its deformed configuration.

**undeformed Use the mesh in its undeformed configuration.

The following global post computations are available in the current release:

```
average              average_around      average_in_element
beremin              beremin_max         continue_curve
coordinates          copy                couple
crack_front          curve               cylindrical
extract2d3d          format              gil_sevillano
gp_vol               gp_xyz              gradient
```

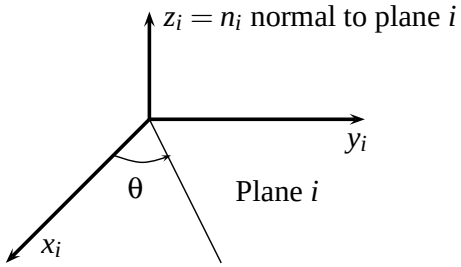
input_damage	max	min
mm_localization	momentum	node_interpolation
selective_int	static_torsor	surface_normals
test_gp	torque	transform_frame
volume	volume_above	volume_integrate
weibull	z7p	max_in_element

**process anisotropic_failure

Description:
This process is an extension of the weibull or batdorf models to the case of anisotropic brittle materials. A probability of failure P_f^i is evaluated on an arbitrary number of characteristic planes for the material (eg. particular crystallographic planes), and the total probability of failure P_f is then computed in the following way:

$$P_f = 1 - \prod_i \left(1 - P_f^i\right)$$

A particular plane i is characterized by its normal vector $\underline{\mathbf{n}}_i$ and the weibull model parameters (strength function $\sigma_u^i(\theta)$ and modulus $m(\theta)$) depend on an angle θ that defines a particular direction in this plane:



The probability of failure P_f^i associated to failure on plane i is then calculated by means of one of the following equations:

- Weibull mode:

$$P_f^i = 1 \; - \; exp\left(- \int_V \frac{dV}{V_0} \; \frac{1}{\theta_{max}} \; \int_0^{\theta_{max}} \left(\frac{\sigma_e(\theta)}{\sigma_u^i(\theta)} \right)^{m^i(\theta)} \; d\theta \right)$$

  In the above equation, the equivalent stress $\sigma_e(\theta)$ is chosen as the normal stress in direction θ:

$$\sigma_e(\theta) \; = \; \underline{\mathbf{d}}_\theta \; \cdot \; \underset{\sim}{\sigma} \; \cdot \; \underline{\mathbf{d}}_\theta$$

  where $\underset{\sim}{\sigma}$ is the stress tensor at the current integration point.

  The modulus $m^i(\theta)$ and the scaling function $\sigma_u^i(\theta)$ are arbitrary functions of θ that should be defined by the user. Coefficients of this function are automatically added as model coefficients by the post-processor, and need to be given in the material file. For example, one can express $\sigma_0^i(\theta)$ in term of the strength $s0$ ($\theta = 0$) and $s90$ ($\theta = \frac{\pi}{2}$) in the following way:

$$\sigma_u^i(\theta) \; = \; s0 \; cos^2(\theta) \; + \; s90 \; sin^2(\theta)$$

  with material coefficients $s0$ and $s90$ declared in the material file.

- Batdorf mode:
  This mode is a slight modification of the previous one, where Batdorf-type statistics are preferred (integration over all possible crack sizes, characterized by their critical stress σ_c):

$$P_f^i = 1 \; - \; exp\left(- \int_V \frac{dV}{V_0} \; \frac{2}{\pi} \; \int_0^{\frac{\pi}{2}} \left\{ \int_0^{\sigma_I} P(\sigma_c, \theta) \; g\left(\sigma_c, \sigma_e(\theta)\right) \; d\sigma_c \right\} d\theta \right)$$

  where:

  - ⋆ The equivalent stress $\sigma_e(\theta)$ is taken as the normal stress in direction θ (same definition as in the previous mode)
  - ⋆ $P(\sigma_c, \theta)$ is the crack density with critical stress σ_c. As in the conventional batdorf model this quantity is expressed as a power law, but depends on θ by means of the anisotropic scale parameter $\sigma_u^i(\theta)$:

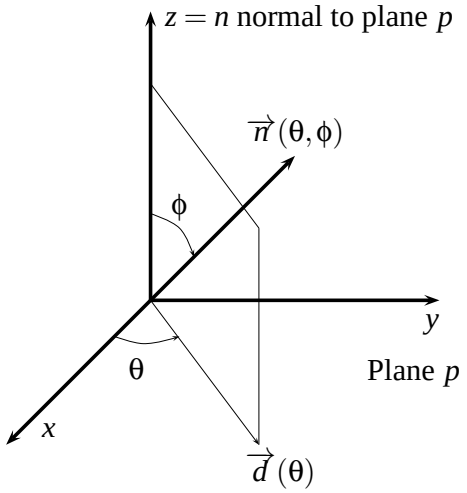
$$P(\sigma_c, \theta) \; = \; \left(\frac{\sigma_c}{\sigma_u^i(\theta)} \right)^{m_i}$$

  - ⋆ $g\left(\sigma_c, \sigma_e(\theta)\right)$ is equal to 1 when the equivalent stress is greater than the critical stress σ_c:
    $g\left(\sigma_c, \sigma_e(\theta)\right) \; = \; 1$ if $\sigma_e(\theta) >= \sigma_c$, else $g\left(\sigma_c, \sigma_e(\theta)\right) = 0$

- Full Batdorf mode:

  This mode fully generalizes the Batdorf approach to the anisotropic case, and allows some flexibility in the choice of the equivalent stress σ_e. Integration is taken over the whole space, using 2 angles θ and ϕ as represented in the following figure, where:

  - $\overrightarrow{d}(\theta)$ is the direction at angle θ in the crystallographic plane as before,
  - ϕ is the angle between the z axis (normal to the crystallographic plane) and a particular $\overrightarrow{n}(\theta, \phi)$ direction in plane $(\overrightarrow{d}(\theta), z)$



The probability of failure is then computed by the following equation:

$$P_f = 1 - exp\left[- \int_V \frac{1}{V_0}\, dV \int_0^{\sigma_I} d\sigma_c \left\{ \frac{1}{2\pi} \int_0^{\pi} d\theta \int_0^{\pi} P(\sigma_c,\theta,\phi)\; g(\sigma_e(\theta,\phi),\sigma_c)\; sin(\phi)\; d\phi \right\} \right]$$

where:

- for each gauss point, integration is performed on the half space only (which correspond to a solid angle of 2π) because of symmetry of the equivalent stress $\sigma_e(\theta,\phi)$
- $P(\sigma_c,\theta,\phi)$ is the density of cracks having a critical stress σ_c:

$$P(\sigma_c,\theta,\phi) \;=\; \frac{1}{\sigma_u(\theta,\phi)} \left(\frac{\sigma_c}{\sigma_u(\theta,\phi)} \right)^{m(\theta,\phi)}$$

  where coefficients σ_u and m depend on the angles (θ,ϕ) defining the current direction $\overrightarrow{n}(\theta,\phi)$.
- $g(\sigma_e(\theta,\phi),\sigma_c)$ is equal to 1 when the equivalent stress is higher than the crack critical stress:

$$g(\sigma_e(\theta,\phi),\sigma_c) = 1 \;\text{ if }\; \sigma_e(\theta,\phi) > \sigma_c \;\;,\;\; \text{else }\; g(\sigma_e(\theta,\phi),\sigma_c) = 0$$

- The equivalent stress $\sigma_e(\theta,\phi)$ is allowed to depend both on the normal stress in direction $\overrightarrow{n}(\theta,\phi)$ and the norm of the shear stress vector acting on the facet normal to $\overrightarrow{n}$:

$$\begin{aligned} \sigma_n(\theta,\phi) &= \overrightarrow{n}(\theta,\phi) \cdot \underset{\sim}{\sigma} \cdot \overrightarrow{n}(\theta,\phi) \\ \overrightarrow{\tau}(\theta,\phi) &= \underset{\sim}{\sigma} \cdot \overrightarrow{n}(\theta,\phi) - \sigma_n(\theta,\phi)\, \overrightarrow{n}(\theta,\phi) \\ \tau(\theta,\phi) &= \| \overrightarrow{\tau}(\theta,\phi) \| \end{aligned}$$

  The implementation then allows to choose between several expression for the equivalent stress σ_e:
  - ⋆ normal stress ("ns") : $\sigma_e(\theta,\phi) \;=\; \sigma_n(\theta,\phi)$
  - ⋆ critical energy ("ce") : $\sigma_e(\theta,\phi) \;=\; \sqrt{\sigma_n(\theta,\phi)^2 + \tau(\theta,\phi)^2}$
  - ⋆ maximal strength ("ms") : $\sigma_e(\theta,\phi) \;=\; \frac{1}{2}\left(\sigma_n(\theta,\phi) \;+\; \sqrt{\sigma_n(\theta,\phi)^2 + \tau(\theta,\phi)^2} \right)$

Syntax:

```
**process anisotropic_failure
  *var name
[ *mode weibull | batdorf | [ full_batdorf ( ns | ce | ms ) ] ]
  *plane
   [ tmin  tmin  ]
   [ tmax  tmax  ]
     normal          <VECTOR>
     function        <FUNCTION>
```

```
   [ m_function     <FUNCTION> ]
   [ phi_function   <FUNCTION> ]
   [ phi_m_function <FUNCTION> ]
[ *plane
   ... ]
[ *orientation | *local_frame <LOCAL_FRAME> ]
[ *nb_constant_step steps  ]
[ *nb_sigc_step sc_steps  ]
[ *precision eps  ]
```

name is the name of the variable used to store the stress in the results file (one would expect `sig` here)

The `*mode` command allows to select the model used to calculate the probability of failure. Possible options, as detailed before, are: `weibull` (default), `batdorf` or `full_batdorf`. In the `full_batdorf` mode an additional keyword is needed to define the choice of the equivalent stress expression. As described before, candidates are: `ns` (normal stress), `ce` (critical energy) or `ms` (maximum strength).

The `*plane` command is used to define the material (or crystallographic) planes characterizing anisotropy of the failure properties. Note that as many `*planes` definitions as needed may be included. The total probability of failure is then computed as:

$$P_f = 1 - \prod_i \left(1 - P_f^i\right)$$

where P_f^i is the probability of failure computed on a particular plane (numbered i in the previous equation). Subcommands of the `*plane` keyword are the following ones:

- the optional commands `tmin` and `tmax` allow to define the bounds (*tmin* and *tmax* values in degrees) for the θ angle integration:

$$P_f \;=\; 1 - exp\left\{ - \int_V \left(\frac{1}{\theta_{max} - \theta_{min}} \int_{\theta_{min}}^{\theta_{max}} \ldots \; d\theta \right) dV \right\}$$

  Default values are *tmin*=0 and *tmax*=90.

- a VECTOR object is expected after the `normal` keyword to define the normal to the current plane.

```
*plane
 normal (0.0 0.0 1.0)
```

- a FUNCTION object is expected after the `function` keyword to define the θ dependence of the strength weibull parameter $\sigma_u(\theta)$.

```
*plane
 function s0*(cos(theta)^2.0)+s90*(sin(theta)^2.0);
```

As is conventional in Zebulon input files, the function definition must be ended by a ";" character, and all libc mathematical functions are allowed (sin, cos, sinh, log, exp, ..). Note that in this particular context a "`theta`" character string is expected in the definition for the name of the function variable. All other parameters in the function definition are assumed to be coefficients (eg. `s0` and `s90` in the previous example) and need to be given in the material file.

- an optional `m_function` keyword can be used to declare a dependence of the m weibull modulus in term of the direction in the current plane. Conventions for the function definition are the same as in the previous case. Without this option, m is assumed to be a constant in all directions. The name of the coefficients expected in the material file are then the following one: `m` in the case of a single plane, or `m1`, `m2` ... when several planes are declared.
- In the `full_batdorf` mode only, additional function definitions are needed to specify the values of the σ_u and m parameters in a particular (θ, ϕ) direction. Corresponding keywords are `phi_function` (for σ_0) and `phi_m_function` (for m).

```
*plane
 function s0*(cos(theta)^2.0)+s90*(sin(theta)^2.0);
 phi_function stheta*(sin(phi)^2) + sc*(cos(phi)^2);
```

Functions in the previous example will define $\sigma_u(\theta, \phi)$ in the following way:

$$\sigma_u(\theta) = s0\ cos^2(\theta)\ +\ s90\ sin^2(\theta)$$

$$\sigma_u(\theta, \phi) = \sigma_u(\theta)\ sin^2(\phi)\ +\ sc\ cos^2(\phi)$$

where `stheta` is a predefined name denoting the result of the σ_0 `function` in a $(\theta, \phi = 0)$ direction, and `s0`, `s90`, `sc` are model coefficients that need to be declared in the material file. As before, those coefficients are arbitrarily named by the user, and automatically added to the model definition after parsing the function expressions.

The optional command `*orientation` or `*local_frame` command allows to transform the stress tensor $\underset{\sim}{\sigma}$ from the global frame to a local material frame prior to volume integration. Such a material frame is in general more convenient to define the anisotropic failure coefficients.

The default method used to perform integration over the θ angle is a simple constant step method with a midpoint rule. The default number of steps is 20, and this value can be modified by means of the `*nb_constant_step` command.

Similarly, integration on the critical stress σ_c in the `batdorf` and `full_batdorf` modes is done by a constant step method and the number of steps involved can be specified by means of the `*nb_sigc_step` command (default value is 20).

Note that when of value of 0 is used as argument of the previous two commands, a second-order runge-kutta method will be used to perform the corresponding integrations. This method is more precise and will automatically adapt the size of the steps to verify a target integration error. However, CPU times may be too long for practical applications, in particular in the `batdorf` or `full_batdorf` modes, and this option should only be used in debug mode to check the implementation against analytical solutions. Experience has shown that a constant step method with a default value of 20 steps is in general sufficient to provide reasonable accuracy. However, if runge-kutta integration is defined (by setting `*nb_constant_step 0` and/or `*nb_sigc_step 0`) the command `*precision` *eps* may to set the precision of the runge-kutta method (default value is *eps*=1.e-4).

For each plane i the probability of failure P_{ip}^{i} associated to each integration point is stored in the results file under the name P*name*_plane*i*, where *name* is the variable name (specified by the `*var` command) and i the plane index. The result of the volume integration of the probability of failure on each plane, and the total P_f for the complete set of planes is written in the `.post` file.

Example:

```
    **file integ % mandatory
    **process anisotropic_failure
     *mode weibull
     *var sig
     *plane
      tmin 0.0 tmax 90.0
      normal (0.0 0.0 1.0)
      function   sm*(cos(3.0*theta)^2.0)+sa*(sin(3.0*theta)^2.0);
      m_function mm*(cos(3.0*theta)^2.0)+ma*(sin(3.0*theta)^2.0);
     *plane
      tmin 0.0 tmax 90.0
      normal (0.0 1.0 0.0)
      function   sc*(cos(theta)^2)+sm*(sin(theta)^2);
      m_function mc*(cos(theta)^2.0)+mm*(sin(theta)^2.0);
     *plane
      tmin 0.0 tmax 90.0
      normal (1.0 0.0 0.0)
      function   sa*(cos(theta)^2)+sc*(sin(theta)^2);
      m_function ma*(cos(theta)^2.0)+mc*(sin(theta)^2.0);

    % material file
***post_processing_data
  **process anisotropic_failure
    V0  2.0
    mm  4.0
    ma  3.5
    mc  3.2
    sm  880.0
```

```
                                                    ****post_processing
                                                     ***global_post_processing
                                                      **process anisotropic_failure

    sa  1200.0
    sc  2000.0
***return
```

Example:

```
    **file integ % mandatory
    **process anisotropic_failure
     *mode full_batdorf ms
     *var sig
     *plane  % m,a
      tmin 0.0 tmax 90.0
      normal (0.0 0.0 1.0)
      function   sm*(cos(3.0*theta)^2.0)+sa*(sin(3.0*theta)^2.0);
      m_function mm*(cos(3.0*theta)^2.0)+ma*(sin(3.0*theta)^2.0);
      phi_function stheta*(sin(phi)^2) + sc*(cos(phi)^2);
      phi_m_function mtheta*(sin(phi)^2) + mc*(cos(phi)^2);

    % material file
***post_processing_data
**process anisotropic_failure
    V0  2.0
    mm  4.0
    ma  3.5
    mc  3.2
    sm  880.0
    sa  1200.0
    sc  2000.0
***return
```

**process average

Description:
This post computation calculates the spatial average of variables specified over a group of elements (`elset`) for each solution step. The average of x is defined as:

$$\bar{x} = \frac{1}{V}\int_V x dV$$

Syntax:

```
**process average
  *list_var name1 ... nameN
```

name1, ... *nameN* are scalar values to be averaged.

Example:

```
**process average
  *list_var sig11 epcum sigmises
```

**process momentum

Description:
This post calculation is used to find then n-th order moment of specified variables over a group of elements (`elset`) at each solution time. The moment of order $n > 1$ of x is defined as:

$$\sigma_n(x) = \left(\frac{1}{V} \int_V (x - \bar{x})^n \, dV \right)^{1/n}$$

Syntax:

```
**process momentum
  *order n
  *list_var name1 ... nameN
```

n is the order (integer value), and *name1*, ... *nameN* are scalar variable names to treat.

Example:
Example:

```
**process  momentum
  *order 2
  *list_var sig11 epcum sigmises
```

**process average_around

Description:
This post processor calculates an average around each integration point in an element set and assigns this average value to each integration point.

Syntax:

```
**process average_around
  *length value
  *list_var name1 ... nameN
```

value is a real value to indicating the distance to take into account around each integration point to make the average. *name1*, ... *nameN* are the scalar variable names to average. The output variable names are generated by adding the prefix `aa_` to each input variable name.

Symmetries are not taken into account. The method is not valid for axisymmetric geometries.

Example:

```
% this will provide aa_epcum
**process average_around
  *list_var epcum
  *length 1.5
```

**process average_in_element

Description:
This post processor calculates the average of the specified variables in each element of an elset, and assigns the mean value to each integration point.

Syntax:

```
**process average_in_element
  *list_var name1 ... nameN
```

where *name1*, ... *nameN* are the scalar variables to average. The output names are created by adding the prefix `ae_` to each variable.

Example:

```
% this will provide ae_epcum
**process average_in_element
  *list_var epcum
```

**process max_in_element

Description:
This post processor sets all element Gauss points values to the element max value. This can be useful when using a visualization software that doesn't take into account Gauss points.
Syntax:

```
**process max_in_element
  *list_var name1 ... nameN
```

where *name1, ... nameN* are the scalar variables to average. The output names are created by adding the prefix `max_in_elt_` to each variable.

Example:

```
% this will provide ae_epcum
**process max_in_element
  *list_var epcum
```

**process batdorf

Description:
This process computes a probability of failure for brittle materials according to the Batdorf model. This model is an extension of the Weibull model that may be more relevant in the case of multiaxial stress states, and also allows some flexibility as regards of the definition of the equivalent stress causing failure.

The probability of failure P_f is expressed in the following form;

$$P_f \;=\; 1 - exp\left(- \int_V dV \int_0^{\sigma_1} \theta(\sigma_c)\; \frac{\Omega}{4\pi}(\sigma_c)\; d\sigma_c \right)$$

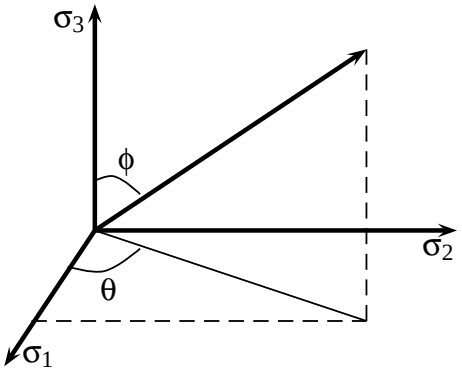
where:

- the integration $\int_0^{\sigma_1}$ ... $d\sigma_c$ is computed over all possible crack sizes, characterized by the critical stress σ_c (σ_1 is the maximum principal stress at the current integration point),
- $\theta(\sigma_c)$ is the density of cracks having σ_c as critical stress
  A classical power law is chosen for $\theta(\sigma_c)$:

$$\theta(\sigma_c) \;=\; \frac{1}{V_0\; \sigma_u} \left\langle \frac{\sigma_c - \sigma_0}{\sigma_u} \right\rangle^m$$

  where V_0, σ_u, σ_0 and m are material parameters.
- $\Omega(\sigma_c)$ is the solid angle of the region of space where the equivalent stress σ_e is higher than the critical stress σ_c
  ($\Omega = 4\pi$ for the whole space around the current integration point in the volume)

  To evaluate the solid angle $\Omega(\sigma_c)$ the following coordinate system that depends on the principal stress values σ_1, σ_2, σ_3 ($\sigma_3 <= \sigma_2 <= \sigma_1$) at the current integration point is defined:



  and the following expression is used to calculate $\Omega(\sigma_c)$:
  $\frac{\Omega(\sigma_c)}{4\pi} = \frac{1}{2\pi} \int_0^{\pi} \left(\int_0^{\pi} g(\sigma_e(\theta,\phi),\sigma_c)\; sin\phi\; d\phi \right) d\theta$

In the previous equation $g(\sigma_e(\theta,\phi),\sigma_c)$ indicates if the equivalent stress σ_e is greater than the critical stress σ_c:
$g(\sigma_e(\theta,\phi),\sigma_c) = 1$ if $\sigma_e(\theta,\phi) >= \sigma_c$, else $g(\sigma_e(\theta,\phi),\sigma_c) = 0$

- Various expression of the equivalent stress σ_e may be chosen:
  - normal stress mode ("ns") : $\sigma_e = \sigma_n$
  - critical energy release rate mode ("ce") : $\sigma_e = \sqrt{\sigma_n^2 + \tau^2}$
  - maximal strength mode ("ms") : $\sigma_e = \frac{1}{2}\left(\sigma_n + \sqrt{\sigma_n^2 + \tau^2}\right)$

  where σ_n an τ are the normal and shear stress components on a facet with normal $\vec{n}(\theta,\phi)$. In the coordinate system of the above figure, the following expressions are used to evaluate σ_n and τ from the principal stress values:

$$\sigma_n = \sigma_1\, cos^2\theta\, sin^2\phi + \sigma_2\, sin^2\theta\, sin^2\phi + \sigma_3\, cos^2\phi$$

$$\sigma_n^2 + \tau^2 = \sigma_1^2\, cos^2\theta\, sin^2\phi + \sigma_2^2\, sin^2\theta\, sin^2\phi + \sigma_3^2\, cos^2\phi$$

Syntax:

```
**process batdorf
  *var name
[ *mode ns | ce | ms ]
[ *steps st  ]
[ *precision eps  ]
[ *nb_constant_step const  ]
[ *force_numeric ]
```

name is the name of the variable used to store the stress in the results file (one would expect `sig` here)

The `*mode` command allows to select the type of equivalent stress used in the model as defined above. Mode `ns` (normal stress) is the default.

The optional command `*steps` is used to defined the number of sub-steps involved in runge-kutta integrations of the various integrals defined above. The default value is *st*=4 (eg. 4 substeps of $\frac{\pi}{4}$ radians for an integration from 0 to π).

eps is the runge-kutta precision required for the previous integrations. Default value is *eps*=1.e-4, that may be decreased to $1.e-3$ for faster integration at the expense of a lower precision of the failure probabilities calculated by the post-processor.

In the `ns` and `ce` modes, analytical expressions can be derived and are used to accelerate the calculation of the $\Omega(\sigma_c)$ critical solid angle. In that case a double integration is needed (integration over σ_c and θ only) to evaluate Ω. In the `ms` mode, no analytical acceleration is possible, and an additional integration over angle ϕ is needed. Use of runge-kutta for

this inner integration is not manageable for efficiency reasons, and a simple constant step integration method is preferred in this case. The `*nb_constant_step` optional command may then be used to specify the number of sub-steps needed for ϕ integration. The default value is *const*=20.

The optional command `*force_numeric` may be used in the `ns` and `ce` mode to replace the previous analytical ϕ integration by the numeric constant step method.

The probability of failure P_{ip} associated to each integration point:

$$P_{ip} = \int_0^{\sigma_1} \theta(\sigma_c)\ \frac{\Omega}{4\pi}(\sigma_c)\ d\sigma_c$$

is stored in the results file under a name constructed by adding "**_ba**" to the variable name, while the total probability P_f obtained after integration on the volume of the component is written in the `.post` file.

Example:

```
**file integ % mandatory
**process batdorf
 *mode ns
 *var sig

% material file
**process batdorf
  V0 1.0
  sigma_u 200.0
  m 3.0
```

**process beremin

Description:
This command performs the Weibull stress and the rupture probability according to the Beremin model (see Beremin F.M., "A local criterion for cleavage fracture of a nuclear pressure vessel steel", Met. Trans. A, 14A, 2277–2287 (1983)). The Weibull stress is defined as:

$$\sigma_W = \left(\frac{1}{V_0} \int_{V,\, p>p_c} \sigma_I^m \, dV \right)^{1/m}$$

where V_0, p_c and m are material parameters. σ_I is the stress tensor maximum principle stress. The integral is taken over the volume where the plastic strain is higher that a critical value p_c. The rupture probability P_r is given by:

$$P_r = 1 - \exp\left(-\left(\frac{\sigma_W}{\sigma_u} \right)^m \right) = 1 - \exp\left(-\frac{1}{V_0} \int_{V,\, p>p_c} \left(\frac{\sigma_I}{\sigma_u} \right)^m dV \right)$$

σ_u is a material parameter.

Syntax:

```
**process beremin
  *stress name1
  *strain name2
```

name1 is the name of the stress tensor and *name2* the name of the inelastic deformation measure (scalar). The material file must contain the values V_0, p_c m and σ_u.

Example:

```
% input file
**process beremin
  *stress sig
  *strain epcum

% material file
**process beremin
  V0      10.
  m       20.
  sigma_u 1200.
  p_c     0.01
```

**process clip_image

Description:
This command is used to generate maps of Gauss point variables inside a predefined cut plane. Produced images are 256 colors GIF files where the value for each pixel is the value of the closest Gauss point. It is also possible to obtain the mean of the element Gauss point values. This process can be quite long as the image is generated finding the element location of each pixel in the structure.

Syntax:

```
**process clip_image
  *list_var var1 var2 ...
  *output output-file-name
  *P0 vector
  *P1 vector
  *P2 vector
[ *ortho ortho-dir-value ]
[ *step spatial-step-value ]
[ *dimx dimx-value ]
[ *dimy dimy-value ]
[ *color_map color-map-name ]
[ *min_max mode ]
[ *mini min-value ]
[ *maxi max-value ]
[ *values type ]
[ *transparency background-color ]
[ *draw_limits_between_elsets ]
[ *elsets_start_with prefix ]
[ *latex orientation ]
```

`*list_var` specifies Gauss point variables to be plotted.

`*output` is used to specify the gif output file name.

`*P0, P1, P2` specify points describing the cut plane: image will be plotted for all $P = P0 + \alpha\, P0P1 + \beta\, P0P2$ with $(\alpha, \beta) \in [0,1]^2$.

`*ortho` is used to impose an orthogonal (P0,P1,P2) base. If value 1 is given a new P2 point will be searched as the projection of the original P2 point on a direction normal

to P0P1. If value 2 is given, the same process if done for P1 point respecting P0P2 direction.

`*step` is used to impose the spatial pitch of generated image pixels.

`*dimx, dimy` are used to imposed the number of pixel used in P0P1 and P0P2 direction (length and height of the GIF image). If only one parameter is set, the other will be computed as to respect a spatial aspect ratio. Note that at least one of the commands `step dimx dimy` must be specified.

`*color_map` specifies the color map that will be employed, choice are one of: default, bone, broken, BW, BW2, cool, copper, hot, HSV, inv_default, inverse, jet, Low_white, pink, prism, PS-default, White_to_dark (same as those in the Zmaster interface).

`*min_max` is used to specify how the scale bar limits are determined and can be `manual` or `auto`. Default behavior is `auto`. If manual mode is selected, user has to provide min and max values thanks to the `*mini` and `*maxi` keywords.

`*values` is used to specify if the values are taken as closest Gauss point (`integration_point`, which is the default behavior), as the mean of the element Gauss point values (`average_in_element`) or as the values of the continuous field which has been extrapolated to the nodes from the Gauss points values (at the global scale, not element-wise) and then interpolated (`interp`).

`*transparency` is used to specify the background color in order to produce transparent pixels when no element are found. It can be `white` or `black`.

`*draw_limits_between_elsets` print limits between elsets with background color (useful to visualize grain boundaries inside polycrystals).

`*elsets_start_with` is used specify what is the prefix of the involved elsets to print limits between. By defaults, all elsets are involved.

`*latex` requires portrait or landscape modifier. This command will generate latex and PDF files containing generated images with color bars, if a valid imagemagick and latex environment is present.

Example:

```
**process clip_image
 *list_var sig23
 *output cuts
 *P0 (-1. -1. 0.)
 *P1 (41. 0. 4.)
 *P2 (0. 41. 4.)
 *dimx 1000
 *ortho 1
 *latex portrait
```

**process coordinates

Description:
This post-processing is used to produce coordinates (X, Y, Z) as field variables; they can then be used by other post-processings. A "node id" field is also produced (resp. "element id" if the process is applied on an integ file).

Syntax:

```
**process coordinates
[ *prefix prefix ]
[ *out X Y Z id ]
```

By default, output fields are named `X`, `Y`, `Z` and `id`. These names can be prefixed by *prefix*, or given completely with the `*out` option.

**process extensometer

Description:
This post computes a virtual extensometer between two points, i.e.:

$$strain = \frac{L - L_0}{L_0}$$

Syntax:

```
**process extensometer
 *p0 vector
 *p1 vector
[ *tolerance value ]
[ *locator_type locator ]
```

`*p0` `*p1` are the two ends of the extensometer.

`*tolerance` is passed to the element locator, to allow slightly out-of-elements point localization. Default is 10^{-12} (the value of tiny displacement, see 3.118).

`*locator_type` is the localization method. It defaults to `bb_tree` (currently the optimal method available in Z-set).

Example:
The following example is taken from `Post_test/INP/extensometer.inp`

```
****post_processing
 ***global_post_processing
  **file node
  **process extensometer
   *p0 (0. 0. 0.)
   *p1 (1. 0. 0.)
  **process extensometer
   *p0 (0. 0. 0.)
   *p1 (1. 1. 0.)
****return
```

**process format

Description:
This post processor is used to produce an "export file" containing the specified variables for all the maps being run. The output file is a formatted ASCII file. The variables should exist in the file specified by the `**file` command (see page 4.26). For instance, the command `*list_var U1` should not be given with `**file integ`. Note: this post processor also exists for `***local_post_processing`, with similar options (see page 4.56).

Syntax:

```
**process format
 *file file
 *list_var name1 ... nameN
[ *precision digits ]
[ *optimizer ]
[ *blank_line ]
[ *write_nodal_coordinates coord | coord0 | none ]
[ *write_gp_coordinates]
```

where *file* denotes the name of the output file, and *name1*, ..., *nameN* is a list of scalar variable names to output. For each map, the values at all locations (nodes, integration points) are stored. The option `*optimizer` inhibits the writing of the output lines containing the character string `# === time` followed by the time associated with the current map, and the character string listing all variables. The option `*precision` determines the number of digits with which the results will be written (default is `10`). The option `*blank_line` writes an empty line between each output map. Some plotting programs, for instance `gnuplot`, may need this.

For variables associated to nodes, it is possible to write out the nodal coordinates as well by using the `*write_nodal_coordinates` option. This options takes as argument either `coord` to indicate that the *current* coordinates should be written, or `coord0` to indicate that the *initial* coordinates should be written. The default value `none` indicates that the nodal coordinates should not be written. Note: currently this option does not exist for the `***local_post_processing **format` version of this command.

For integrated variables, it is also possible to write out the Gauss point coordinates, by specifying the `*write_gp_coordinates` option.

**process gradient

Description:
Calculate the gradient of $(U1, U2)$ with respect to the given direction.

Syntax:

```
**process gradient  dir
```

---

where *dir* should be either `X`, `Y` or `Z`.

The resulting (integ) field are named `gradXU1`, `gradXU2` (resp. with `Y` or `Z`).

Example:

```
****post_processing
 ***global_post_processing
  **file node
  **nset ALL_NODE
  **process gradient X
  **process gradient Y

 ***local_post_processing
  **file integ
  **elset ALL_ELEMENT
  **process function
   *output epsilon12
   *expression 0.5*(gradXU2 + gradYU1) ;
****return
```

**process hot_spot

Description:
This post processor is used to find locations which are local maxima or mimima of a given variable, and within a specified range, but isolated from other peaks by a given radius. This post computation should be useful for automating fatigue or critical stress analysis when meshes and loading conditions are often changing (see for example the usage with `cycle_projection` on page 4.38).

Syntax:

```
**process hot_spot
  *format fmt-string
  *function FUNCTION
  *max_val max
  *min_val min
  *minimum
  *num fname
  *radius rad
  *screen_output
  *variable name
  *write_nodes fname
```

The following summarizes the different commands and their parameters:

`*format` is used to format the screen output for aesthetic value to the user. The command reads the characters remaining on the same line and uses that as a `printf` style formatting.

`*function` applies a FUNCTION to the selected field values before evaluating the hot spot condition (remember the semicolon after the function declaration).

`*max_val` specifies a real value for the maximum values to be considered. This can be used to truncate the analysis to regions of reasonable scale.

`*min_val` specifies a real value for the minimum values to be considered. An example use is when regions of a structure do not have a fatigue life prediction (and therefore values of zero), but we are interested in the minimum locations with values over 10 for example.

`*minimum` is a switch to indicate that we want minimum values as the hot spot. The default is to search for maximum absolute values.

`*num` specifies an integer for the number of values to report.

*radius specifies a real value for the spatial radius separating the hot spot locations.

*variable specifies the scalar variable of interest. This can of course be a derived variable coming from previous post processing operations.

*write_nodes is used to give a file name where a Zmesh compatible node set will be listed in order of the *num* hot spot points.

Note:
This command currently applies to nodal data only.

Example:
The following example is from the test case hot_spot.inp in test/Post_test/INP.

```
***global_post_processing
 **process hot_spot
  *radius   5.
  *variable X
  *num      2
  *function -1.0*x;
  *screen_output
```

Because we asked for *screen_output the following summary is printed to the standard out (and .msgp file). At the first timestep (t=0) the field is uniform so the first 2 nodes respecting the zone rule are printed. In the following the field is just scaled so the location results are the same. That would of course not be the case in a general FEA analysis.

```
+++++ HOTSPOT ANALYSIS X +++++
Time:0.000000
  1             -2.00e-01
  7             -2.00e-01
running :  25 %

+++++ HOTSPOT ANALYSIS X +++++
Time:1.000000
  152           -2.80e+00
  309           -2.15e+00
running :  50 %

+++++ HOTSPOT ANALYSIS X +++++
Time:2.000000
  152           -2.10e+00
  309           -1.61e+00
running :  75 %

+++++ HOTSPOT ANALYSIS X +++++
Time:3.000000
  152           -1.40e+00
  309           -1.08e+00
```

**process max

Description:
This post-processor gives the maximal value of the specified variables over the current group of elements (selected by `**elset`).

Syntax:

```
**process max
  *list_var name1 ... nameN
[ *localization ]
[ *position ]
[ *associated var1 ... varM ]
```

`*list_var` *name1*, ... *nameN* are scalar variable names;

`*localization` outputs the element/IP where the maximum is reached;

`*position` outputs the coordinates where the maximum is reached, note that this requires that the variables `gpX`, `gpY` and `gpZ` have been generated with a `process coordinates` as in the example below;

`*associated` also outputs the given variables values where the maximum is reached.

Example:
A simple example is:

```
**process max
 *list_var sig11 sig22
```

A more elaborate example, featuring all possible options follows:

```
****post_processing
 ***local_post_processing
  **file integ
  **elset ALL_ELEMENT
  **process mises  *var sig
 ***global_post_processing
  **elset ALL_ELEMENT
  **process coordinates *prefix gp
  **process max
   *list_var sigmises sig33
   *associated sig22
   *localization
   *position
****return
```

**process min

Description:
This post-processor gives the minimal value of the specified variables over the current group of elements (selected by `**elset`).

Syntax:

```
**process min
  *list_var name1 ... nameN
[ *localization ]
[ *position ]
[ *associated var1 ... varM ]
```

See the documentation of `process max` on previous page (4.126) for the meaning of these options.

**process node_extrapolation

Description:
This post-computation extrapolates integration point data to either the nodes (CTNOD format), or to the element nodes (CTELE).

Syntax:

```
**process node_extrapolation
  *list_var name1 ... nameN
[ *output_var o-name1 ... o-nameN ]
```

name1, ... , *nameN* are the scalar variable names to treat.

The input data is read in the results files *problem*.integ. The calculated results are stored in the file *problem*.ctnodp.

These output variables are named gp*name1*, ... gp*nameN*, or *o-name1* ... *o-nameN* if they were specified.

Example:

```
% Generates the variable gpsig11 at nodes
% from the sig11 variable in the .integ file
**file integ
**process node_extrapolation
  *list_var sig11
```

**process node_interpolation

Description:
This post-computation interpolates nodal variables at the Gauss points.

Syntax:

```
**process node_interpolation
  *list_var name1 ... nameN
[ *output_var o-name1 ... o-nameN ]
```

name1, ... , *nameN* are the scalar variable names to treat.

The input data is read in the results files *problem*`.node` or *problem*`.ctnod` (`**file node`). The calculated results are stored in the file *problem*`.integp`.

These variables are named `gp`*name1*, ... `gp`*nameN*.

Example:

```
   % provides gptemperature at the Gauss points
   % from the temperature in .node file
***global_post_processing
 **file node
 **nset ALL
 **process node_interpolation
  *list_var TP
```

**process relocalised_post

Description:
This post computation interpolates the specified DOFs at specified coordinates.

Syntax:

```
**process relocalised_post
  *variables list-of-variables
[ *station vector ]
[ *station_file infile ]
[ *p1 (x y z) *p2 (x y z) *num n ]
[ *output_file outfile ]
[ *tolerance value ]
[ *locator_type locator ]
```

`*variables` is the list of DOFs to output.

`*station` is used if a single point is probed; use `*station_file` if more than one point is required.

`*station_file` is the name of an ASCII file containing coordinates of desired output points.

`*p1 *p2 *num` are used to probe the results between the specified points, with n points. Note that the 3 options `*station`, `*station_file` and this last triplet are mutually exclusive.

`*output_file` specifies in which file output should be given. If it is omitted, results are printed on screen. If the filename ends with `.vtk`, the output file will be in vtk format; otherwise, standard ASCII file format suited to gnuplot is used.

`*tolerance` is passed to the element locator, to allow slightly out-of-elements point localization. Default is 0.

`*locator_type` is the localization method. It defaults to `bb_tree` (currently the optimal method available in Z-set).

Example:
The following example shows how to probe a computation result along a line:

```
****post_processing
 ***global_post_processing
  **file integ
  **process coordinates   # to create variables X Y Z
```

```
  **process relocalised_post
   *variables X Y Z sig11
   *p1 (0.   -9.666 2.096)
   *p2 (125. -9.666 2.096)
   *num 500
   *output_file sig_along_line.dat
****return
```

Example:
This example shows how to extract the displacement at specific coordinates:

```
****post_processing
***global_post_processing
 **file node
%  **at 1.  30. 365. 3650. 36500.
 **output_number 0-999
 **process relocalised_post
  *station_file GPS_stations.dat
  *variables U1 U2 U3
  *output_file GPS-out.vtk
  *tolerance 1. % in distance unit
****return
```

where the station file looks like (note that for convenience all text after the 3rd column is considered as a label for the point, and is ignored):

```
#  X              Y              Z             label
-1.129764e+06   6.229049e+06   7.156934e+05    ARAU
-1.155688e+06   6.239179e+06   5.715508e+05    BABH
-1.834799e+06   6.058418e+06  -7.202233e+05    BAKO
-9.878078e+05   6.183675e+06   1.173047e+06    BANH
# [...]
```

**process static_torsor

Description:
This post computes the static torsor equivalent to the sum of every nodal forces over a nset for each solution step. The equivalent static torsor is defined as:

$$\{ \mathcal{T} \}_O = \left\{ \begin{array}{c} \sum_{i \in \text{nset}} \vec{F}_i \\ \sum_{i \in \text{nset}} O\vec{O}_i \wedge \vec{F}_i \end{array} \right\}_O$$

Syntax:

```
**process static_torsor
 *point vector
 *file file
```

where *vector* is the position where the momentum is computed (O). The output gives the coordinates of point O, the 3 components of the resultant and the 3 components of the momentum for each time step. The output file is named *file*.

Example:

```
***global_post_processing
 **nset top
 **file node
 **process static_torsor
  *point (1. 0.5 0.5)
  *file torsor.dat
```

**process cohesion_torsor

Description:
This post computes the equivalent cohesion torsor of the internal nodal forces (summed over a specified nset) on the specified elset (the elset corresponds to the isolated body).

Syntax:

```
**process cohesion_torsor
 *point vector
 *file file
 *nset cut_nset
```

where *vector* is the position where the momentum is computed (point O). The output gives the coordinates of point O, the 3 components of the resultant and the 3 components of the momentum for each time step. The output file is named *file*. *cut_nset* is a nset that cuts the body. It must corresponds to a boundary of the elset specified with `**elset`.

Example:

The following example computes the torsor of the action of the right part of the structure on the left part. *cut* is a nset that fully separates the elsets *left* and *right*.

```
***global_post_processing
 **elset left
 **file integ
 **process cohesion_torsor
  *point (0. 6. 6.)
  *file cohesion_torsor.test
  *nset cut
```

**process surface_normals

Description:
This post processor computes the outer-pointing normals at nodes in the specified nset, and generates associated nodal fields. It currently works in 3D only.

Defining a normal direction at a node on a ridge is naturally ambiguous. Each node receives a contribution of its attached elements, *if all nodes of the elements are in the specified nset*. Thus, on ridges the normal is averaged over adjacent elements. Restricting the computation to a smooth subset of the surface will usually generate the desired field.

Syntax:

```
**surface_normals
[ *prefix prefix ]
```

prefix is used to name the output variable. Its default value is `N` so that normal vector components are `N1`, `N2` and `N3`.

Example:

```
% This post processing calculates outer-pointing
% heat flux from a thermal calculation
****post_processing
 ***global_post_processing
  **file node
  **nset interior
  **process surface_normals
   *prefix N
    % Can be visualized in paraview with this calculator:
    % N1*iHat+N2*jHat+N3*kHat

 ***local_post_processing
  **file node
  **nset interior
  **process function
   *expression q1*N1 + q2*N2 + q3*N3 ;
   *output normalflux
****return
```

**process volume

Description:
Calculate the volume of a group of elements (`elset`).

Syntax:

```
     **process volume
```

Example:

```
****post_processing
    ***precision 3
    ***data_source mesh_only
     **format Z7
     **open mymesh.geof
     **maps 1.

    ***global_post_processing
     **file integ
     %
     % volume of the whole structure
     %
     **elset ALL_ELEMENT
     **process volume
     %
     % volume of a part
     %
     **elset PART1
     **process volume

****return
```

**process volume_of_element

Description:
Compute the volume of each elements and set this value as an integ field (called "volume").

Syntax:

```
     **process volume_of_element
```

Example:

The following example is extracted from `Post_test/INP/volume_of_element.inp`:

```
****post_processing

 ***data_source mesh_only
  **format Z7
  **open volume_of_element.geof
  **maps 1.

 ***global_post_processing
  **file integ
  **elset ALL
  **process volume_of_element

****return
```

**process volume_above

Description:
The `volume_above` processor is used to calculate the volume of elements of the active element set for which the value of the specified variable is greater than a criterion fixed by the used.

The result can be normalized to find the relative percentage of volume meeting the given criterion.

Syntax:

```
**process volume_above
  *var name
  *threshold value
  *normalize
```

where *name* is the name of a scalar variable. *value* is a real (floating point) value indicating the criterion to consider. With the option `normalize` the given result corresponds to the total volume fraction of elements meeting the criterion.

Example:

```
**process volume_above
  *var epcum
  *threshold  1.
  *normalize
```

**process volume_integrate

Description:
Integrates the variable over a group of elements (`elset`).

Syntax:

```
**process volume_integrate
  *list_var var
```

Example:

The following example (cf. `Thermal_test/INP/bc_volumetric_heat.inp`) compares the FE and analytical solutions, by computing the L^2 norm of their difference.

```
****post_processing
 ***global_post_processing
  **file node
  **nset ALL
  **process coordinates

 ***local_post_processing
  **file node
  **nset ALL
  **process function
   *expression ( TP - exp(X)*sin(Y)*cos(Y+Z)*(X^2 + Z^3/125) )^2;
   *output DIFF2

 ***global_post_processing
  **file node
  **nset ALL
  **process node_interpolation
   *list_var DIFF2

 ***global_post_processing
  **file integ
  **elset ALL
  **process volume_integrate
   *list_var gpDIFF2
****return
```

**process weibull

Description:
This post computation provides the means for doing a Weibull analysis on a structure. Two modes of operation are available:

`eigenstress`

$$\sigma' = \left(\frac{\sigma_1^M - \sigma_0}{\sigma_u} \right)^m$$

`independent`

$$\sigma' = \left(\frac{\sigma_1^M - \sigma_0}{\sigma_u} \right)^m + \left(\frac{\sigma_2^M - \sigma_0}{\sigma_u} \right)^m + \left(\frac{\sigma_3^M - \sigma_0}{\sigma_u} \right)^m$$

The Weibull stress is then defined as:

$$\sigma_W = \left(\frac{\sigma_u}{V_0} \int_V \sigma'^m \, dV \right)^{1/m}$$

where σ_1^M, σ_2^M and σ_3^M are the post-computation results from a local `fmax` applied successively to the three principal stresses σ_1, σ_2 and σ_3, in descending order. These sub-posts are run automatically by the `weibull` processor.

V_0, σ_u, σ_0 and m are material parameters.

The probability of failure P_r is is given by:

$$P_r = 1 - \exp\left(- \left(\frac{\sigma_W}{\sigma_u} \right)^m \right)$$

In addition to the output of σ_W and P_r. the values of σ' are stored at each Gauss point under a name constructed by adding `_wb` to the variable name.

Syntax:

```
**process weibull
  *var name
  *mode eigenstress | independant
  *coefmin value1
  *coefmax value2
  *file namef
```

If the user has only specified a single map (with `output_number`), the history is reconstructed from the values read after the options `*coefmin` and `*coefmax`. σ_1^M varies linearly over 100 maps of *value1*σ_1 to *value2*σ_1 (where σ_1 is calculated at the specified map).

With the option `*file`, the history is read in the file `namef`, of the form:

```
****post_processing
 ***global_post_processing
  **process weibull
```

```
0.01     10
0.02     15
0.03     20
0.04     40
0.05     60
...
```

where the first column represents the time and the second the value of σ_1.

These two methods presented for the mode `eigenstress` extend to the mode `independant` as well.

Example:

```
**output_number 10
**process weibull
  *var sig
  *mode independent
  *coefmin 0.5
  *coefmax 2.5

% material file
**process weibull
  V0      10.
  m       20.
  sigma_u 1200.
  sigma_0 0.
```

Chapter 5

Results reading and management

****results_management

Description:

Results management is a utility to rewrite Z-set output files, after a computation or after a post-processing.

It can extract some maps from a computation, thus reducing output files size to only those maps that are necessary for archiving. It may also pick some variables, discarding unnecessary ones.

For example, a user may want to output all strain components during a computation, in order to assert its validity; then after this verification, only keep the cumulated plastic strain at a few maps to continue with some post-processing while saving disk space.

Results management may also concatenate two or more computations, to produce a single results database.

The execution command is:

`Zrun -resm` *file*`.inp`

It will store results as *file*`.ut`, *file*`.node`, *file*`.integ`.

Syntax:
This utility works as follows:

```
****results_management
 ***add fname1
 [ *list_var      var-names ]
 [ *out_var       outvar-names ]
 [ *output_number map-numbers ]
 [ *output_times  time range ]
 [ *post ]
[ ***add fname2  ... ]
[ ***only_check_number_nodes_elements ]
```

`***add` specifies which file contains the original data. Use multiple `add` to combine their databases together.

`*list_var` specifies which variables are copied. It can be either nodal or integration points variables. Each component of vectorial or tensorial variables must be specified.

`*out_var` if specified, is a translation for each input variable name.

`*output_number` specifies the selected maps. One may use single values or ranges. For example valid *map-numbers* would be: `2`, or `4-10`, or `1-100-2` (every odd map between 1 and 100, 100 being excluded).

`*output_times` specifies the selected times ranges with time increment. For example valid *times* would be: `4.-10.-2.` selects the maps just before 4., 6., 8. and 10.

`*post` process post-processing results (i.e. work on problem.utp instead of problem.ut)

`***only_check_number_nodes_elements` allows an "unsafe" use of `results_management`. By default, it verifies that all added databases share the same mesh. Use this option to replace this verification by a simple assertion that the mesh sizes (number of nodes and elements) are compatible; the user should be sure that the concatenation operation is still meaningful.

Example:

The following example extracts maps 10,12,14,16,18 from a computation:

```
****results_management
 ***add square1      % from square1.inp ...
  *list_var sig11
  *output_number 10-20-2
****return
```

The following example is from `$Z7TEST/Transfer_test/INP/results_management.inp`. It will create a single map, combining 2 output variables (sig11 and eto11), read from two different computations (square1 and square2 respectively):

```
****results_management
 ***add square1      % from square1.ut ...
  *list_var sig11    % ... extract sig11
  *output_number 2   % ... at the 2nd map

 ***add square2      % ... and combine this with square2.ut results
  *list_var eto11
  *output_number 2
****return
```

The following example will create 2 maps, with 2 output variables

```
****results_management
 ***add square1
  *list_var sig11
  *output_number 1

 ***add square2
```

```
  *list_var eto11
  *output_number 2
****return
```

The following example extracts maps corresponding to times just before (or at) 1. 2. 3. 4. 4.1 4.2 4.3 4.4 4.5

```
****results_management
 ***add square1      % from square1.inp ...
  *list_var U1 U2 U3 sig11
  *output_times 1.-4.-1.
  *output_times 4.-4.5-0.1
****return
```

****forge

.

Description:

This utility is used to dump FORGE or REM3D databases and translate the results into Z-set format for further post-processing or for initial state initialization at the beginning of the Finite Element simulation. The main purpose of this utility is to read and concatenate two or more FORGE result files stored on the same mesh, to produce a single Z-set results database. To read single FORGE or REM3D result files, an alternative method is to use ****post_processing with ***data_source option, as described on page 4.14.

Syntax:
This utility works as follows:

```
****forge
 ***input   input_database_name[.fg3|.may|.in3]
 ***output  output_name
[ ***prefix input_database_prefix ]
[ ***mat    zmat_file_name ]
[ ***from   incr_id ]
```

The execution command is:

```
Zrun -forge file.inp
```

It will store results as *output_name*.ut, *output_name*.node, *output_name*.integ.

***input specifies the name of the FORGE result database

***output specifies the name of the resulting Z-set database

***prefix is the filename prefix of the FORGE results files, this keyword is used when multiple FORGE results are to be dumped and concatenated.

***mat Z-mat material file name for SDV names translation. This option can be used when translating Z-mat/Forge simulation results.

***from selection of specific maps starting from given FORGE increment.

Example:

The following example translates mesh as well as nodal and elemental results from FORGE computation :

```
****forge
 ***input upper_die_0d50.fg3
```

```
 ***output zset_results
****return
```

The following example will concatenate multiple FORGE results files with prefix *upper_die_0d*, to produce a single Z-set results database :

```
****forge
***prefix upper_die_0d
***output z_upper_die
****return
```

Chapter 6

Reference

Functions

Description:
Functions have been generalized in code versions greater than 7.1 to have a natural syntax, and applied more widely throughout the program. In addition to coefficient definitions, functions may now be used to define load waveforms, used in the optimizer, etc. More applications will be added as well.

Functions follow a C-like syntax, and may be defined in terms of predefined functions (sin, log, etc) and named parameters of the problem. What parameters are allowed and available depends on the application.

Syntax:
Objects which are functions may now be entered using a C-like format. This *must* include a semi-colon at the end of the function definition to terminate the reading of the function. Many operators exist, which have the same meaning and order of selection as in C. Coefficients which are functions still require the keyword `function` to follow the coefficient name, but are otherwise generally given in terms of the relevant parameters.

The following operators are defined in functions:

```
:= == >= <= + - * / ^ > <
```

as are the following functions:

```
asinh acosh atanh sinh cosh tanh asin acos atan sin cos tan
sign floor ceil sqrt sqr neg abs exp ln Hx H min max
triangle top_triangle urandom
```

Most of these functions take on the classical meaning from standard C programming.

sqr square of the parameter (is equivalent to `^2`)

sqrt square-root of the parameter

triangle function used to make a sawtooth waveform with period 1 of the parameter.

floor largest integral value not greater than x.

H Heaviside function. Equal to 1 for $x \geq 0$ and 0 otherwise.

Hx Heaviside of its parameter times its parameter. Equal to x for $x > 0$ and 0 otherwise.

urandom(a,b) Uniform random distribution that returns values between a and b

Example:

Here are 3 typical examples using functions (a mesher, a calcul and a behavior):

```
% create a bset based on coordinates
**bset edge *function (y>=5.0)*(z>=10) ;

% define a loading table as a function of time:
***function load_tab sin(6.28319*time);

% define a material coefficient depending on a parameter:
***elasticity
   young function 2.e6-100.*temperature -2.*temperature^2.;
   poisson 0.3
```

Adding new functions:

Global function definitions is often useful when complex functions are required, or as a means to adjust local parameters (e.g. in a material) via the input file. There are three places to make function declarations. One is by using the `***`-level command `***function_declarations` in the main problem input file, one is in the material file itself using the command `**functions` command, and the last is in an external file loaded via the `-learn` command-line argument.

For example:

```
****calcul
 ***function_declarations
    TR := 250.0;
    pi := 3.14159265359;
...


 ***behavior gen_evp
  **functions
    zmax := (a>b)*a + (a<=b)*b;
    zmin := (a<b)*a + (a>=b)*b;
    lfunc := zmax( a , 1.e-6 ) ;

Zrun -learn my_functions.txt calcul.inp
```

where `my_functions.txt` is for instance:

```
    T   := x^2 * exp(y) * cos(z);
    LT  := 2* exp(y)* cos(z);
```

Degrees of Freedom (DOF)

Description:
The degree of freedom (DOFs) types in Z-set each have a particular name used for their identification in various parts of the code. These applications may be in specifying the boundary conditions (`**impose_nodal_dof` takes a DOF name for example), specifying output curves, or use within the post-processor.

The names currently active in the code are summarized in the following table:

CODE	DESCRIPTION
U1	displacement along the axis 1
U2	displacement along the axis 2
U3	displacement along the axis 3
EZ	Deformation along the axis 3 for plane stress
R1	Rotation about the axis 1
R2	Rotation about the axis 2
R3	Rotation about the axis 3
W1	Micro-polar rotation about the axis 1
W2	Micro-polar rotation about the axis 2
W3	Micro-polar rotation about the axis 3
PR	Pressure
TP	Temperature

Element Geometries

Description:
This section defines the ordering of node and Gauss point numbering for the different element geometries. The following figures display the element information with numbering corresponding to the order of definition in the `.geof` file.

The following element geometries are available:

`spr1, spr2` springs with fixed behavior (in `.geof` file. These accept spring parameters after the node list.

`l2d2, l2d2r, l3d2, l3d2r` line elements. These are used for truss or spring elements with two nodes. There is only a force acting between the nodes.

`l2d1, l3d1` one node spring or dashpot elements. These oppose motion from their original location.

`rve1d, rve2d, rve3d` representative volume element (RVE) to be used for material simulation. There are no nodes. and only one "integration" point.

`deb2` 2D debonding element geometry.

`c2d3r, c2d3, c2d4r, c2d4, c2d6r, c2d6, c2d8r, c2d8` 2D continuum elements. "r" denotes reduced integration.

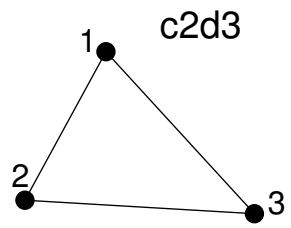
`cax3r, cax3, cax4r, cax4, cax6r, cax6, cax8r, cax8` axisymmetric 2D continuum elements.

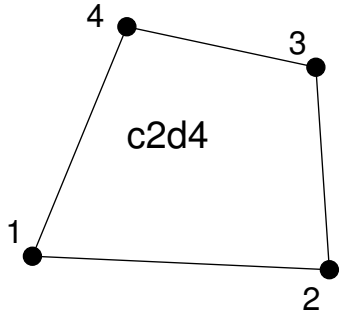
`c3d4, c3d4r, c3d10r, c3d10, c3d6r, c3d6, c3d8r, c3d8, c3d20r c3d20, c3d15r, c3d15` 3D continuum elements.

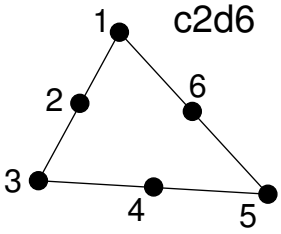
`sax3, sax3r, sax3rr, sax2, sax2r, sax2rr` axisymmetric shells.

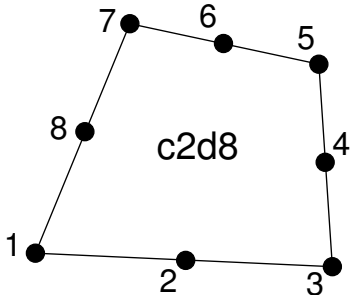
`s3d3r, s3d3, s3d4r, s3d4, s3d6r, s3d6, s3d8r, s3d8` 3D shells.

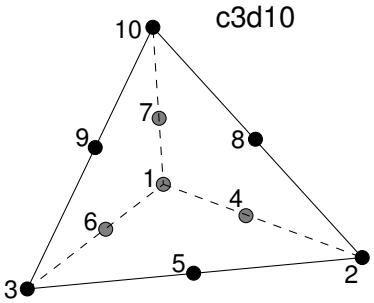
Node definitions for different element geometries are given on the following page. Note that all other information about the element chosen (formulation, material, real constants) are specified in terms of element sets in the *problem*`.inp` file.

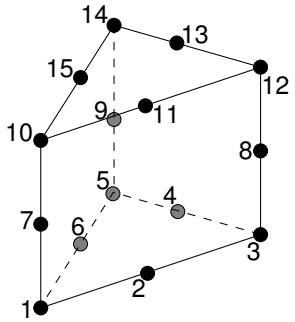

c2d3
1
2
3


c2d4
4
3
1
2

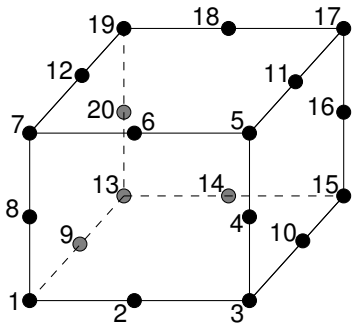

c2d6
1
2
6
3
4
5


c2d8
7
6
5
8
4
1
2
3


c3d10
10
7
9
8
1
4
6
5
3
2


14
13
15
12
9
11
10
8
5
4
7
6
3
2
1

c3d15


19
18
17
12
11
20
6
5
16
7
13
14
15
8
9
4
10
1
2
3

c3d20

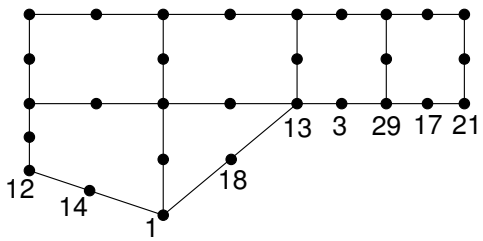
Boundary sets

Description:
The boundary sets (faces in 3D or lines in 2D) must be ordered properly such that the normals may be calculated, positions interpolated, and surface integrals evaluated over them. Incorrect definitions of these sets are often responsible for boundary condition problems.

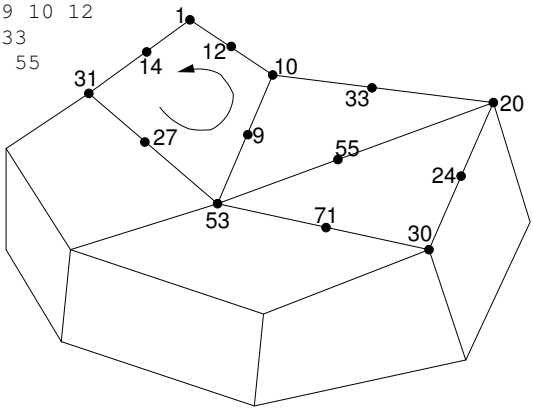
The following boundary types are available:

CODE	DESCRIPTION
line	linear (2 node) line
quad	quadratic (3 node) line
t3	linear 3 node triangle face
t6	quadratic 6 node triangle face
q4	linear 4 node quadrilateral face
q8	quadratic 8 node quadrilateral face

```
**liset bottom
 quad 12 14 1
 quad 1 18 13
 quad 13 3 29
 quad 29 17 21
```



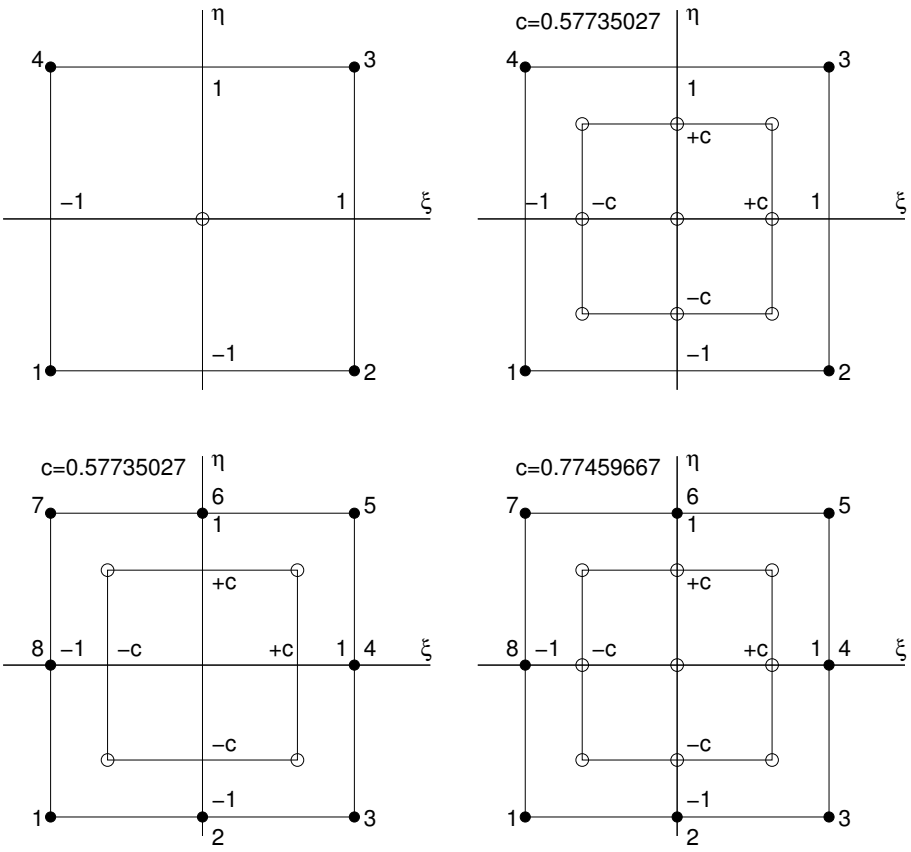

```
**faset top
q8 1 14 31 27 53 9 10 12
t6 10 9 53 55 20 33
t6 53 71 30 24 20 55
```

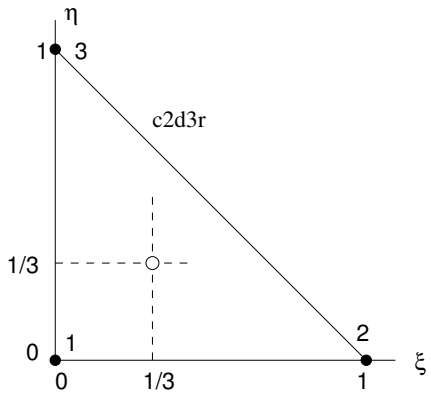


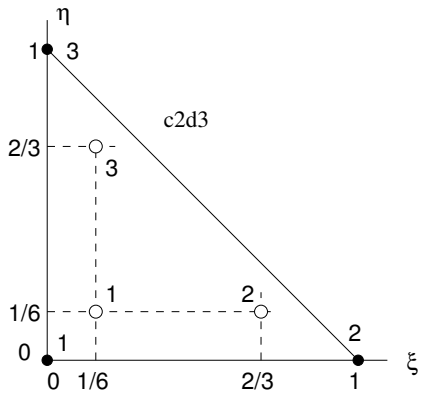
Element Integration

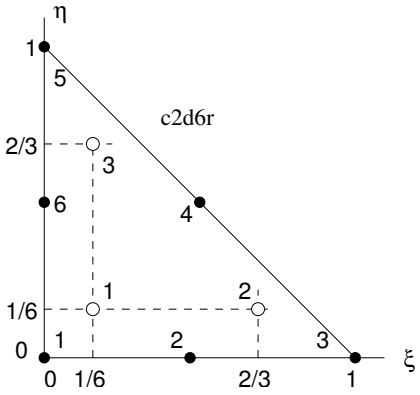
Description:
This section defines the positions and numbering of element Gauss points used for spatial integration. These points are the *only* places where material values (e.g. stress and strain) exist. If you need accurate magnitudes of these variables, it is highly advised to use a `**test` output (see page 3.175) to output them, or use the `**value_at_integration` output for the structure (page 3.172).

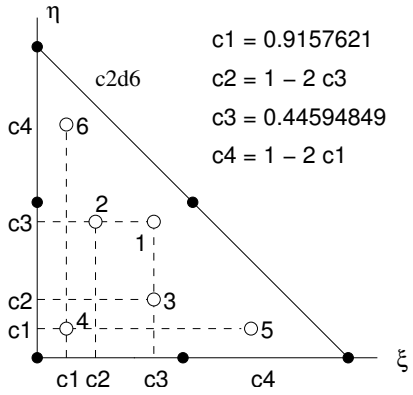
Gauss points are defined in a local element coordinate system which is mapped to the global coordinates (real element geometry) using the element shape functions. The element is defined in the local coordinates by -1 to 1 on the ξ and η element axis.




η
1
3
c2d3r
1/3
0
1
2
0
1/3
1
ξ


η
1
3
c2d3
2/3
3
1/6
1
2
0
1
2
0
1/6
2/3
1
ξ


η
1
5
c2d6r
2/3
3
6
4
1
2
1/6
0
1
2
3
0
1/6
2/3
1
ξ


η
c1 = 0.9157621
c2 = 1 – 2 c3
c3 = 0.44594849
c4 = 1 – 2 c1
c2d6
c4
6
2
c3
1
c2
3
4
c1
5
c1
c2
c3
c4
ξ

Structure of *problem*.geof

The structure of the ASCII file describing problem geometries is summarized below:

`***geometry`	
`**node`	
n dim	number of nodes, space dimension
x1 y1 [z1]	
...	coordinates
xn yn [zn]	
`**element`	
j	number of elements
1 type1 n11 .....n1k	element number, type, list of node numbers
...	
j typej nj1 .... njk	
`***group`	
`**nset` *name*	name of a node set
n1 n2 ... nk	list of the k nodes in the node set
`**elset` *name*	name of an element set
n1 n2 ... nk	list of the k elements in the element set
`**faset` *name*	surface (3D face) group name
type1 n11 ... n1k	type of face, ordered list of nodes in the face
...	
typej nj1 ... njk	
`**liset` *name*	surface (linear) group name
type1 n11 ... n1k	type of line, ordered list of nodes in the liset
...	
typej nj1 ... njk	
`***return`	

The position of the nodes must be given in Cartesian coordinates for 2D plane and 3D problems, or in cylindrical coordinates (r,z) for 2D axisymmetric problems.

The groups of nodes, faces, elements, and element lines segments must come after the node and element definitions. These groups are defined between the keywords `***group` and `***return`.

Z-set output formats

Z-set has two output formats. The default output format is `Z7`. A new output format has been implemented since `Z8.4` version, to allow remeshing.

Z7 output format

problem.ut The file *problem*.ut describes the contents of the binary output files *problem*.node, *problem*.integ, *problem*.ctnod, and *problem*.ctele. It contains:

- **meshfile gives the name of the file *problem*.geof used for computation.
- after **node the names of the nodal variables and parameters stored in *problem*.node are given.
- after **integ the names of integration (material) variables stored in *problem*.integ, *problem*.ctnod, and *problem*.ctele are given.
- **element is not presently used.

Following these four marked up sections, a list of the output timesteps is given. Each line represents the state of the calculation for one map. These lines contain the output number, the cycle number, the sequence number, the increment and corresponding time, in that order.

problem.node The file *problem*.node contains the nodal output variables. These variables consist of the nodal degrees of freedom (such as displacements, temperature, etc), the associated reactions to the nodal DOFs (forces, heat flux), and possibly the problem nodal parameters if the **save_parameter option was given after ***output. The stored variables will be the complete set of variables at each node, regardless if certain parameters do not exist at every node in the problem. For variables which do not exist at a given node, a zero value will be stored.

The *problem*.node file has the following structure:

$[d_1,\ldots,d_N]_1\ldots[d_1,\ldots,d_N]_{n_d}$ 1^{st} output

$[v_1,\ldots,d_N]_1\ldots[v_1,\ldots,d_N]_{n_d}$

$[d_1,\ldots,d_N]_1\ldots[d_1,\ldots,d_N]_{n_d}$ 2^{nd} output

[...]

where n_d is the number of DOFs given by **node in the *problem*.ut file, and N is the number of nodes.

The *problem*.node file may be written in C with the following code snippet:

```
for(card=0; card < stored_cards; card++) {
   for(variable=0; variable < nodal_variables_stored; variable++) {
      for(node=0; node < number_of_nodes_in_mesh ; node++) {
         fwrite(&component[node][variable][card],4,1,fp);
      }
   }
}
```

problem.integ The file *problem*.integ contains the integration point values issued from the material behavior laws (see **value_at_integration under ***output). In the event that certain variables do not exist at all integration points in the problem, a zero value will be stored in its place. Thus the number of variables is always the union of the different material variables in the problem.

Each record in the *problem*.integ file has the following format:

$v1_{pg1el1},\ v1_{pg2el1}\ \ldots\ v1_{pgnel1}$	1^{st} variable for element 1
$v2_{pg1el1},\ v2_{pg2el1}\ \ldots\ v2_{pgnel1}$	2^{nd} variable for element 1
...	
$vnv_{pg1el1},\ vnv_{pg2el1}\ \ldots\ vnv_{pgnel1}$	variable nv for element 1
$v1_{pg1el2},\ v1_{pg2el2}\ \ldots\ v1_{pgnel2}$	1^{st} variable for element 2
$v2_{pg1el2},\ v2_{pg2el2}\ \ldots\ v2_{pgnel2}$	2^{nd} variable for element 2
...	
$vnv_{pg1el2},\ vnv_{pg2el2}\ \ldots\ vnv_{pgnel2}$	variable nv for element 2
...	
...	
$v1_{pg1elNE},\ v1_{pg2elNE}\ \ldots\ v1_{pgnelNE}$	1^{st} variable for element NE
$v2_{pg1elNE},\ v2_{pg2elNE}\ \ldots\ v2_{pgnelNE}$	2^{nd} variable for element NE
...	
$vnv_{pg1elNE},\ vnv_{pg2elNE}\ \ldots\ vnv_{pgnelNE}$	variable nv for element NE

where NE is the number of elements in the structure, and nv the number of values stored per integration point (listed after **integ in the file *problem*.ut).

Elements generally do not have the same number of integration points. Due to this, the storage of different element types in a problem will generate element data structures of different sizes within the same record.

The file *problem*.integ may be generated using the following C code:

```
for(card=0; card < number_of_outputs; card++) {
   for(ele=0; ele < number_of_elements ; ele++) {
      for(var=0; var< number_of_integ_variables; var++) {
         for(point=0; point < number_of_IP_in_element[ele]; point++) {
            fwrite(&component[point][var][ele][card],4,1,fp);
         }
      }
   }
}
```

problem.ctnod The *problem*.ctnod file contains the integration point variables (material variables) extrapolated to the nodes and will be generated when the **contour option is selected under ***output. This file uses a storage format similar to *problem*.node files. The only difference is that the .ctnod file does not store values for orphan nodes (i.e. nodes not attached to an element).

The structure of each record in the *problem*.ctnod file is the following:

$v1_1$, $v1_2$ ... $v1_N$ variable 1

$v2_1$, $v2_2$ ... $v2_N$ variable 2

...

vnv_1, vnv_2 ... vnv_N variable nv

where N is the number of (non-orphan) nodes and nv the number of variables indicated by `**integ` in the *problem*`.ut` file. A *problem*`.ctnod` may be generated using the following C code:

```
for(card=0;card < stored_cards; card++){
   for(var=0; var< number_of_integ_variables; var++) {
      for(node=0; node < number_of_nodes_in_mesh; node++) {
         if (! is_orphan_node(node))
            fwrite(&component[node][var][card],4,1,fp);
      }
   }
}
```

problem`.ctele` The file *problem*`.ctele` contains the integration variables at nodes. The file duplicates the *problem*`.ctnod` data but uses on an element to element extrapolation basis. Element based storage retains the discontinuous fields between elements with different materials. The file will be generated by using the `**contour_ele` option under `***output`. Output records in the *problem*`.ctele` file each have the following format:

$v1_{no1el1}$, $v1_{no2el1}$ ... $v1_{nonel1}$ 1^{st} variable for element 1

$v1_{no1el2}$, $v1_{no2el2}$ ... $v1_{nonel2}$ 1^{st} variable for element 2

...

$v1_{no1elNE}$, $v1_{no2elNE}$ ... $v1_{nonelNE}$ 1^{st} variable for element NE

$v2_{no1el1}$, $v2_{no2el1}$ ... $v2_{nonel1}$ 2^{nd} variable for element 1

$v2_{no1el2}$, $v2_{no2el2}$ ... $v2_{nonel2}$ 2^{nd} variable for element 2

...

$v2_{no1elNE}$, $v2_{no2elNE}$ ... $v2_{nonelNE}$ 2^{nd} variable for element NE

...

...

vnv_{no1el1}, vnv_{no2el1} ... vnv_{nonel1} variable nv for element 1

vnv_{no1el2}, vnv_{no2el2} ... vnv_{nonel2} variable nv for element 2

...

$vnv_{no1elNE}$, $vnv_{no2elNE}$ ... $vnv_{nonelNE}$ variable nv for element NE

where NE is the number of the elements in the structure, and nv the number of values stored per integration point (as given by the `**integ` section in the file *problem*`.ut`).

Different elements do not have the same number of nodes. The storage for different elements within the problem may therefore have different data structure sizes within the same record.

The file *problem*`.ctele` may be generated using the following C code:

```
for(card=0; card < stored_cards; card++) {
   for(var=0; var < number_of_integ_variables; var++) {
      for(ele=0; ele < number_of_elements; ele++) {
         for(node=0; node < number_of_nodes_of_element[ele]; node++) {
            fwrite(&component[node][ele][var][card],4,1,fp);
         }
      }
   }
}
```

problem.eigen_info If the calculation is an eigen value problem, the resonant frequencies and associated energies for each mode are stored in the file *problem*.eigen_info. This file is an ASCII formatted text file.

problem.eigen When an eigen value calculation is run, the resonant vectors or modal displacements are stored in the file *problem*.eigen. The structure of the binary file is ordered mode number, frequency of the mode, and the energy associated to the mode. Thus we have:

$U1_1,\ U1_2\ \ldots\ U1_N$	displacement U1
$U2_1,\ U2_2\ \ldots\ U2_N$	displacement U2
$[U3_1,\ U3_2\ \ldots\ U3_N$	displacement U3]

end of the storage of the first mode.

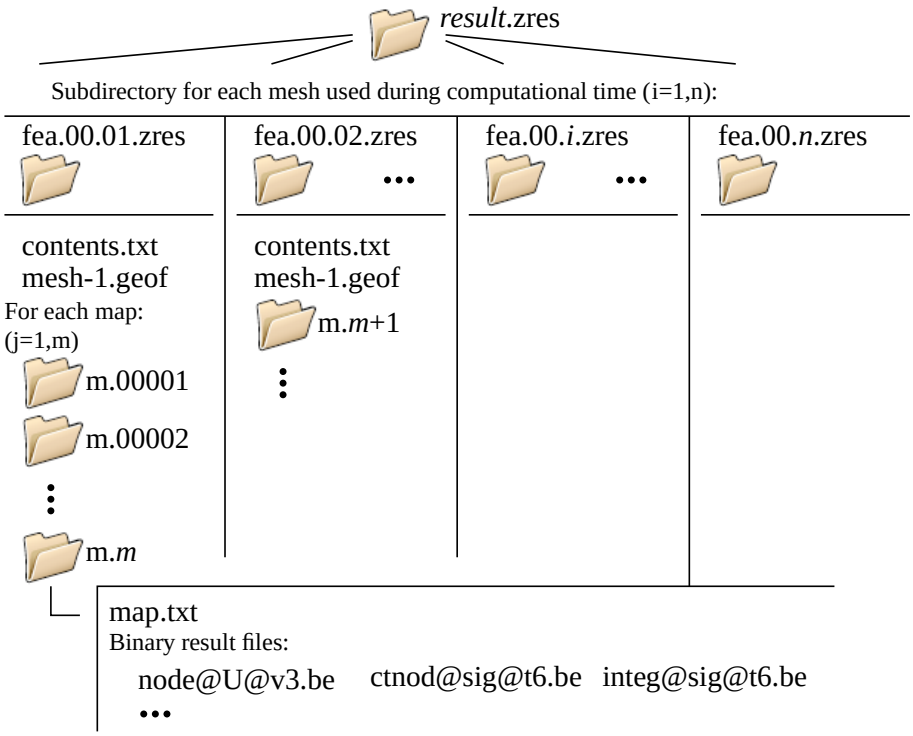
The file *problem*.eigen may be generated using the following C code:

```
for(mode=0; mode < number_of_modes; mode++) {
   fwrite(&mode,4,1,fp);
   fwrite(&mode_frequency[mode],4,1,fp);
   fwrite(&mode_energy[mode],4,1,fp);
   for(dim=0; dim < space_dimension ; dim++) {
      for(node=0; node < number_of_nodes_in_mesh; node++) {
         fwrite(&component[dim][node][mode],4,1,fp);
      }
   }
}
```

Z8 output format

Description:
The `Z8` output format has been implemented since `Z8.4` version, to allow remeshing. While the default `Z7` format uses *results*`.integ`, *results*`.node`, *results*`.ctnod` ... result files, the new `Z8` format uses a tree allowing to read results for which mesh changes from one map to another.

The figure below shows the structure of the file system containing result and mesh files.



Syntax:
User can activate the new `Z8` format by adding in the `.inp` file the following commands:

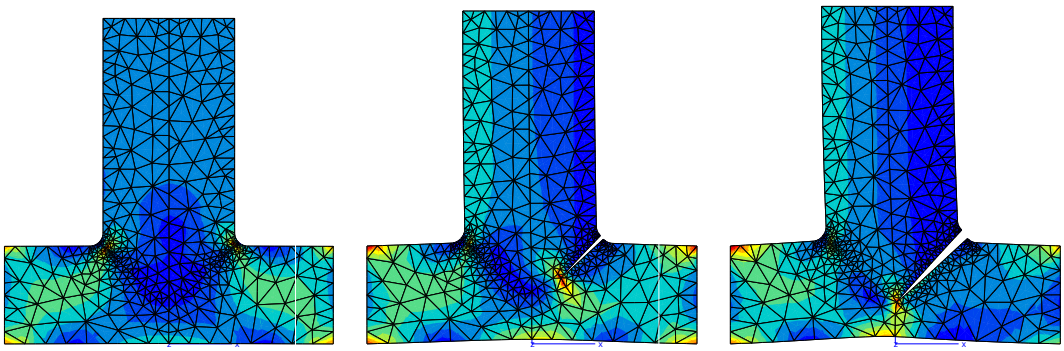
```
****calcul
   ...
 ***global_parameter
   Solver.OutputFormat Z8
   Zmaster.OutputFormat Z8
   ...
****return
```

The `Z8` format file structure is organized as follows:

- one main directory named -here- *result*`.zres` (implying that the `.inp` file is *result*`.inp`, and the execution command "`Zrun` *result*");
- some sub-directories respectively named `fea.00.01.zres`, `fea.00.02.zres` ... created at each remeshing. Each i^{th} remeshing generates a corresponding `fea.00.`*i*`.zres` sub-directory. Each sub-directory contains:
  - an ASCII file named `contents.txt` that describes the name, the type (scalar, tensor ...), and the location (at nodes, at integration points, extrapolated at averaged nodes ...) of the physical quantities stored in result files;
  - the current mesh file *i.e.* an ASCII file with a `.geof` extension;
  - a list of sub-directories named `m.00001`, `m.00002` ..., corresponding to the list of output maps (computational times for wich user wants results to be written), and containing:
    - an ASCII file named `map.txt` that describes the current output map (cycle number, sequence number, increment number, time value ...);
    - binary files containing the results computed for each physical quantity to be written.

This new file system is directly machine-readable using `Zmaster`. User may from now on graphically analyze computational results for which mesh changes from one map to another.

The figure below shows a crack propagation during a computation implying remeshing.



Environment Variables

The environment variables are used to personalize the Z-set program for individual users, and maintain a self-contained project structure. The function extends as well to manage multi-architecture sites, thereby allowing the transparent use of the program over a diverse network. The currently used environment variables are summarized as:

CODE	DESCRIPTION
Z7PATH	path to the head Z-set's directory structure
Z7TEST	directory containing tests and examples
Z7HANDBOOK	path to the documentation
Z7HANDBOOK_READER	the PDF reader called by Zman to open manuals
Z7LICENSE	path to a license file if not stored in `$Z7PATH/lib/Zebulon.License`
ZEBU_PATH	list of directories searched for user plugins

`Z7PATH` variable which indicates the location of the Z-set distribution. This variable allows easy switching to different distributions. It also is a cause of much difficulty with users if not configured correctly.

```
# csh syntax:
  setenv Z7PATH /usr/local/Zset/Z8.7.1
  source ${Z7PATH}/lib/Z7_cshrc
# bash syntax:
  export Z7PATH=/usr/local/Zset/Z8.7.1
  source $Z7PATH/lib/Z7_profile
```

`Z7LICENSE` fully qualified path to a license file. Extremely useful when there are multiple versions, or for running Zebulon directly off a CD-ROM where one can't put the license file in a read only lib dir.

Example:
In the C-shell, these variables are set using the `setenv` command. The variables may be directly in the user's `.cshrc` file, or in a separate file. For the latter case, one may add the line to the `.cshrc` file: `if (-f ~/lib/Z7_vars) source ~/lib/Z7_vars`

Chapter 7

Bibliography

Farh91 C. Farhat and F.-X. Roux, "A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm," *Int. J. Num. Meth. Eng.*, **32** 1205-1227 (1991).

Feye98 F. Feyel, *Application du Calcul Parallèle aux Modèles à Grand Nombre de Variables Internes*, Thesis Ecole Nationale Supérieure des Mines de Paris (1998).

Hors85 Horst G. DeLorenzi, "Energy release rate calculations by the finite element method", *Eng. Fract. Mech.*, **21** 129-143 (1985).

Matt79 H. Matthies and G. Strang, "The Solution of Nonlinear Finite Element Equations," *Int. J. Num. Meth. Eng.*, **14**, 1613-1626 (1979).

Neu89 Neu and Sehitoglu "Thermomechanical Fatigue Oxidation and Creep Part II Life Prediction" *Metall Trans A*, **20A** 1769-1783 (1989).

Park74 D.M. Parks, "A Stiffness Derivative Finite Element Technique for Determination of Crack Tip Stress Intensity Factors," *Int. J. Fracture*, **10**, 487-502 (1974).

Chab12 J.L. Chaboche "Cyclic inelastic constitutive equations and their impact on the fatigue life predictions" *Int. J. Plasticity*, **35**, 44-66 (2012).

Chapter 8

Index

Index