GiD
The universal, adaptative and user friendly pre and post processing system for computer analysis in science and engineering

Customization Manual

Table of Contents

Chapters	Pag.
1 FEATURES	1
2 INTRODUCTION	3
3 CONFIGURATION FILES	7
3.1 XML file	7
3.1.1 ValidatePassword node	7
3.2 Conditions file (.cnd)	9
3.2.1 Example: Creating the conditions file	13
3.3 Problem and intervals data file (.prb)	15
3.3.1 Example: Creating the PRB data file	16
3.4 Materials file (.mat)	17
3.4.1 Example: Creating the materials file	18
3.5 Special fields	20
3.6 Unit System file (.uni)	24
3.7 Conditions symbols file (.sim)	25
3.7.1 Example: Creating the Symbols file	26
4 TEMPLATE FILES	29
4.1 Commands used in the .bas file	30
4.1.1 Single value return commands	30
4.1.2 Multiple values return commands	34
4.1.3 Specific commands	38
4.2 General description	48
4.3 Detailed example - Template file creation	49
4.3.1 Formatted nodes and coordinates listing	56
4.3.2 Elements, materials and connectivities listing	58
4.3.3 Nodes listing declaration	59
4.3.4 Elements listing declaration	61
4.3.5 Materials listing declaration	62
4.3.6 Nodes and its conditions listing declaration	63
5 EXECUTING AN EXTERNAL PROGRAM	67
5.1 Showing feedback when running the solver	68
5.2 Commands accepted by the GiD command.exe	68
5.3 Managing errors	75
5.4 Examples	75
6 POSTPROCESS DATA FILES	77
6.1 Postprocess results format: ProjectName.post.res	78
6.1.1 Gauss Points	79
6.1.2 Result Range Table	84
6.1.3 Result	85
6.1.4 Results example	89
6.1.5 Result group	94
6.2 Postprocess mesh format: ProjectName.post.msh	98

6.2.1 Mesh example 101
6.2.2 Group of meshes 103
6.3 Postprocess list file: ProjectName.post.lst 108
6.4 Postprocess graphs file: ProjectName.post.grf 108
7 TCL AND TK EXTENSION 111
7.1 Event procedures 111
7.2 GiD_Process function 119
7.3 GiD_Info function 120
7.3.1 materials 120
7.3.2 conditions 121
7.3.3 layers 122
7.3.4 gendata 122
7.3.5 intvdata 123
7.3.6 project 123
7.3.7 geometry 124
7.3.8 mesh 124
7.3.9 coordinates 125
7.3.10 variables 125
7.3.11 localaxes 125
7.3.12 ortholimits 126
7.3.13 perspectivefactor 126
7.3.14 graphcenter 126
7.3.15 meshquality 126
7.3.16 postprocess 126
7.3.17 automatictolerance 128
7.3.18 rgbdefaultbackground 129
7.3.19 list_entities 129
7.3.20 parametric 131
7.3.21 check 132
7.3.22 listmassproperties 133
7.3.23 problemtypepath 134
7.3.24 gidversion 134
7.3.25 view 134
7.3.26 ispointinside 134
7.4 Special Tcl commands 134
7.4.1 Geometry 134
7.4.2 Mesh 138
7.4.3 Data 139
7.4.4 Results 142
7.4.5 Graphs 144
7.4.6 OpenGL 145
7.4.7 Other 148
7.5 HTML help support 151
7.5.1 GiDCustomHelp 151
7.5.1.1 HelpDirs 152

7.5.1.2 Structure of the help content 152
7.5.1.3 TocPage 152
7.5.1.4 IndexPage 153
7.5.2 HelpWindow 153
7.6 Managing menus 154
7.7 Custom Data Windows 158
7.7.1 TkWidget 158
7.7.2 Data Windows Behavior 161
7.8 GiD version 162
7.9 Detailed example 163
8 PLUG-IN EXTENSIONS 167
8.1 Tcl plug-in 167
8.2 GiD dynamic library plug-in 167
8.2.1 Introduction 167
8.2.2 In GiD 168
8.2.3 Developing the plug-in 169
8.2.4 Functions provided by GiD 171
8.2.5 List of examples 174
9 APPENDIX (PRACTICAL EXAMPLES) 177

1 FEATURES

GiD offers the following customization features:

- Complete menu´s can be customised and created to suit the specific needs of the user´s simulation software.
- Simple interfaces can be developed between the data definition and the simulation software.
- Simple interfaces based on scalar, vector and matrix quantities can be developed for the results visualisation.
- Menus for the results visualisation can be customised and created according to the needs of the application or analysis.

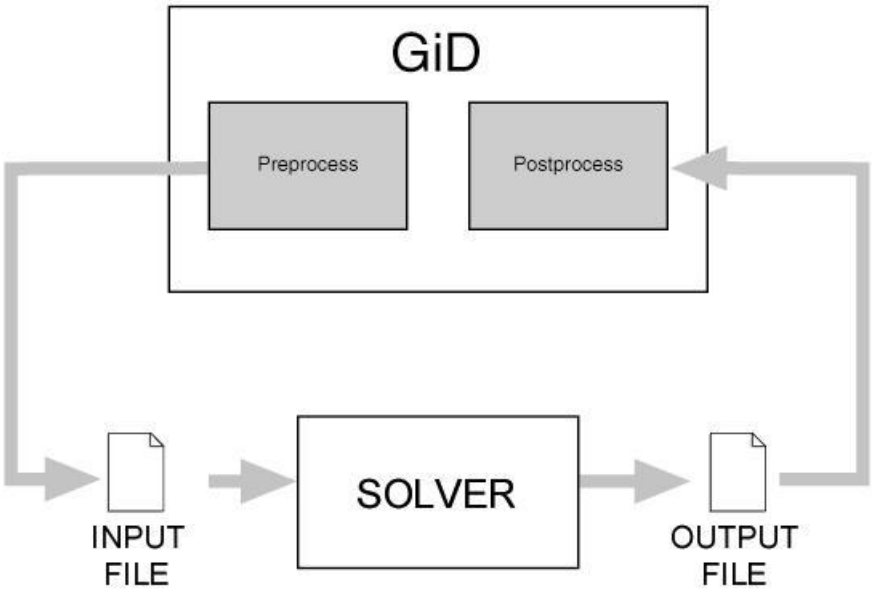
The customization in GiD is done by creating a **Problem Type**.

2 INTRODUCTION

When GiD is to be used for a particular type of analysis, it is necessary to predefine all the information required from the user and to define the way the final information is given to the solver module. To do so, some files are used to describe conditions, materials, general data, units systems, symbols and the format of the input file for the solver. We give the name **Problem Type** to this collection of files used to configure GiD for a particular type of analysis.

Note: You can also learn how to configure GiD for a particular type of analysis by following the **Problem Type Tutorial**; this tutorial is included with the GiD package you have bought. You can also download it from the GiD support web page (http://www.gidhome.com/support).

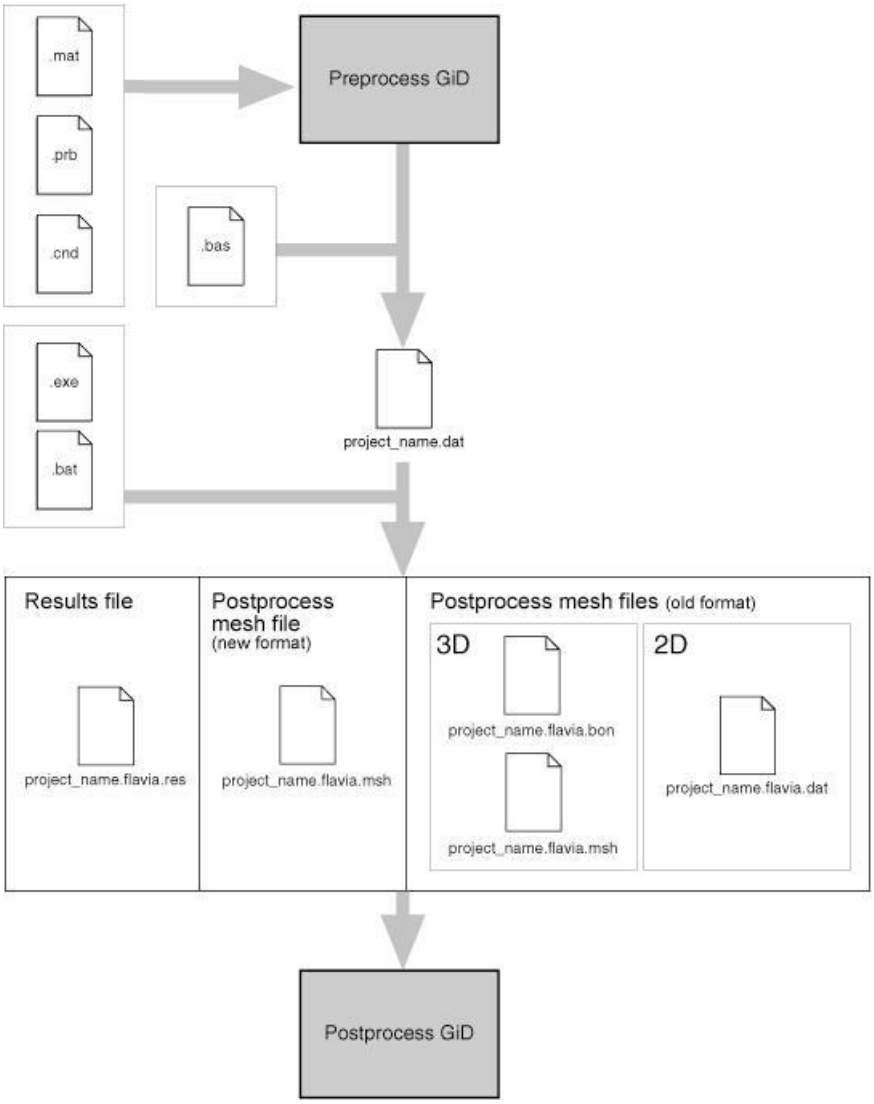
GiD has been designed to be a general-purpose Pre- and Postprocessor; consequently, the configurations for different analyses must be performed according to the particular specifications of each solver. It is therefore necessary to create specific data input files for every solver. However, GiD lets you perform this configuration process inside the program itself, without any change in the solver, and without having to program any independent utility.



To configure these files means defining the data that must be input by the user, as well as the materials to be implemented and other geometrical and time-dependent conditions. It is also possible to add symbols or drawings to represent the defined conditions. GiD offers the opportunity to work with units when defining the properties of the data mentioned above, but there must be a configuration file where the definition of the units systems can be found. It is also necessary to define the way in which this data is to be written inside the file that will be the input file read by the corresponding solver.

The creation of a **Problem Type** involves the creation of a directory with the name of the problem type and the extension .gid. This directory can be located in the current working

directory or the main GiD executable directory. The former can be useful during the development of the project. Once it is finished, it may be advisable to move the directory to the one where GiD is stored; in this way, your problem type will be added to those included in the system and it will appear in the GiD menu (see Problem type from Reference Manual). In both cases, the series of files must be inside the problem type directory. The name for most of them will follow the format problem_type_name.xxx where the extension refers to their particular function. Considering problem_type_name to be the name of the problem type and project_name the name of the project, file configuration is described by the following diagram:



- **Directory name**: problem_type_name.gid
- **Directory location**: c:\a\b\c\GiD_directory\problemtypes

Configuration files

- problem_type_name.xml XML-based configuration
- problem_type_name.cnd Conditions definitions
- problem_type_name.mat Materials properties
- problem_type_name.prb Problem and intervals data
- problem_type_name.uni Units Systems
- problem_type_name.sim Conditions symbols

- ***.geo Symbols geometrical definitions
- ***.geo Symbols geometrical definitions ...

- **Template files**
- problem_type_name.bas Information for the data input file
- ***.bas Information for additional files
- ***.bas Information for additional files ...

- **Tcl extension files**
- problem_type_name.tcl Extensions to GiD written in the Tcl/Tk programming language

- **Command execution files**
- problem_type_name.bat Operating system shell that executes the analysis process

The files problem_type_name.sim, ***.geo and ***.bas are not mandatory and can be added to facilitate visualization (both kinds of file) or to prepare the data input for restart in additional files (just ***.bas files). In the same way problem_type_name.xml is not necessary; it can be used to customize features such as: version info, icon identification, password validation, etc.

3 CONFIGURATION FILES

These files generate the conditions and material properties, as well as the general problem and intervals data to be transferred to the mesh, at the same time giving you the chance to define geometrical drawings or symbols to represent some conditions on the screen.

3.1 XML file

The file problem_type.xml contains information related to the configuration of the problem type, such as file browser, icon, password validation or message catalog location. Besides this, the file can be used to store assorted structured infomation such as version number, news added from the last version, and whatever the developer decides to include. This file can be read using the Tcl extension tcom which is provided with **GiD**.

The data included inside the xml file should observe the following structure:

```
<Infoproblemtype version="1.0">
  <Program>
  </Program>
</Infoproblemtype>
```

We suggest that the following nodes are included (the values of these nodes are just examples):

- `<Name>Nastran 4.1</Name>` to provide a long name for the problem type.
- `<Version>4.1</Version>` dotted version number of the problem type.
- `<MinimumGiDVersion>11.0</MinimumGiDVersion>` to state the minimun **GiD** version required.
- `<ImageFileBrowser> images/ImageFileBrowser.gif </ImageFileBrowser>` icon image to be used in the file browser to show a project corresponding to this problem type. The recommended dimensions for this image are 17x12 pixels.
- `<MsgcatRoot> scripts/msgs </MsgcatRoot>` a path, relative or absolute, indicating where the folder with the name msgs is located. The folder msgs contains the messages catalog for translation.
- `<PasswordPath>..</PasswordPath>` a path, relative or absolute, indicating where to write the password information see ValidatePassword node -pag. 7-).
- `<ValidatePassword></ValidatePassword>` provides a custom validation script in order to overide the default **GiD** validation (see ValidatePassword node -pag. 7-).

3.1.1 ValidatePassword node

The default action taken by **GiD** when validating a problem type password is verifying that it is not empty. When a password is considered as valid, this information is written in the file 'password.txt' which is located in the problem type directory. In order to override this behaviour, two nodes are provided in the .xml file

- PasswordPath: The value of this node specifies a relative or absolute path describing where to locate/create the file password.txt. If the value is a relative path it is taken with respect to the problem type path.

Example:

```
<PasswordPath>..</PasswordPath>
```

- ValidatePassword: The value of this node is a Tcl script which will be executed when a password for this problem type needs to be validated. The script receives the parameters for validation in the following variables:

key with the contents of the password typed,

dir with the path of the problem type, and

computer_name with the name of host machine.

Note: It's like this Tcl procedure prototype: proc PasswordPath { key dir computer_name } { ... body... }

The script should return one of three possible codes:

0 in case of failure.

1 in case of success.

2 in case of success; the difference here is that the problem type has just saved the password information so **GiD** should not do it.

Furthermore, we can provide a description of the status returned for **GiD** to show to the user. If another status is returned, it is assumed to be 1 by default.

Below is an example of a <ValidatePassword> node.

```
<ValidatePassword>
  #validation.exe simulates an external program to validade the key for this computername
  #instead an external program can be used a tcl procedure
  if { [catch {set res [exec [file join $dir validation.exe] $key $computername]} msgerr] } {
     return [list 0 "Error $msgerr"]
  }
  switch -regexp -- $res {
    failRB {
      return [list 0 "you ask me to fail!"]
    }
    okandsaveRB {
      proc save_pass {dir id pass} {
        set date [clock format [clock second] -format "%Y %m %d"]
```

```
        set fd [open [file join $dir .. "password.txt"] "a"]
        puts $fd "$id $pass # $date Password for Problem type '$dir'"
        close $fd
      }
      save_pass $dir $computername $key
      rename save_pass ""
      return [list 2 "password $key saved by me"]
    }
    okRB {
      return [list 1 "password $key will be saved by gid"]
    }
    default {
      return [list 0 "Error: unexpected return value $res"]
    }
  }
</ValidatePassword>
```

3.2 Conditions file (.cnd)

Files with extension .cnd contain all the information about the conditions that can be applied to different entities. The condition can adopt different field values for every entity. This type of information includes, for instance, all the displacement constraints and applied loads in a structural problem or all the prescribed and initial temperatures in a thermial analysis.

An important characteristic of the conditions is that they must define what kind of entity they are going to be applied over, i.e. over points, lines, surfaces, volumes or layers, and what kind of entity they will be transferred over, i.e. over nodes, over face elements or over body elements.

- **Over nodes** This means that the condition will be transferred to the nodes contained in the geometrical entity where the condition is assigned.
- **Over face elements ?multiple?** If this condition is applied to a line that is the boundary of a surface or to a surface that is the boundary of a volume, this condition is transferred to the higher elements, marking the affected face. If it is declared as **multiple**, it can be transferred to more than one element face (if more than one exists). By default it is considered as **single**, and only one element face will be marked.
- **Over body elements** If this condition is applied to lines, it will be transferred to line elements. If assigned to surfaces, it will be transferred to surface elements. Likewise, if applied to volumes, it will be transferred to volume elements.

Note: For backwards compatibility, the command 'over elements' is also accepted; this will transfer the condition either to elements or to faces of higher level elements.

Another important feature is that all the conditions can be applied to different entities with different values for all the defined intervals of the problem.

Therefore, a condition can be considered as a group of fields containing the name of the particular condition, the geometric entity over which it is applied, the mesh entity over which it will be transferred, its corresponding properties and their values.

The format of the file is as follows:

```
CONDITION: condition_name

CONDTYPE: 'over' ('points', 'lines', 'surfaces', 'volumes', 'layer')

CONDMESHTYPE: 'over' ('nodes', 'face elements','face elements multiple',
'body elements')

QUESTION: field_name['#CB#'(...,optional_value_i,...)]

VALUE: default_field_value['#WIDTH#'(optional_entry_length)]

  ...

QUESTION: field_name['#CB#'(...,optional_value_i,...)]

VALUE: default_field_value['#WIDTH#'(optional_entry_length)]

END CONDITION

CONDITION: condition_name

  ...

END CONDITION
```

Note: #CB# means Combo Box.

Note: #WIDTH# means the size of the entry used by the user to enter the value of the condition. Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget's font.

Local Axes

```
QUESTION: field_name['#LA#'('global', 'automatic', 'automatic alternative')]

VALUE: default_field_value['#WIDTH#'(optional_entry_length)]
```

This type of field refers to the **local axes** system to be used. The position of the values indicates the kind of **local axes**.

If it only has a single default value, this will be the name of the global axes. If two values are given, the second one will reference a system that will be computed automatically for every node and will depend on geometric constraints, like whether or not it is tangent, orthogonal, etc. If a third value is given, it will be the name of the automatic alternative axes, which are the automatic axes rotated 90 degrees.

All the different user-defined systems will automatically be added to these default possibilities.

To enter only a specific kind of local axes it is possible to use the modifiers #G#,#A#,#L#.

- #G#: global axes;
- #A#: automatic axes;
- #L#: automatic alternative axes.

When using these modifiers the position of the values does not indicate the kind of local axes.

Example

```
QUESTION: Local_Axes#LA#(Option automatic#A#,Option automatic_alt#L#)
VALUE: -Automatic-
```

Note: All the fields must be filled with words, where a word is considered as a string of characters without any blank spaces. The strings signaled between quotes are literal and the ones inside brackets are optional. The interface is case-sensitive, so any uppercase letters must be maintained. The default_field_value entry and various optional_value_i entries can be alphanumeric, integers or reals. GiD treats them as alphanumeric until the moment they are written to the solver input files.

Global axes:

X=1 0 0

Y=0 1 0

Z=0 0 1

Automatic axes:

For surfaces, this axes are calculated from the unitary normal N:

z'=N

if N is coincident with the global Y direction (Nx or Nz > some tolerance) then

  x'=Y x N / |Y x N|

else

  x'=Z x N / |Z x N|

y'=N x x'

z'=N

For lines, this axes are calculated from the unitary tangent T:

x'=T

if T is coincident with the global Z direction (Nx or Ny > some tolerance) then

  y'=Y x x' / |Y x x'|

else

y'=Z x x' / |Z x x'|

z'=x' x y'

Automatic alternative axes:

They are calculated like the automatic case and then swap x and y axes:

x''= y'

y''= - x'

z''= z'

For curves

x'=unitary tangent to the curve on the place where the condition is applied

If this tangent is different of the Z global axe=(0,0,1) then

y'=Y x x'

else

y'=Z x x'

z'=x' x y'

Note: the tangent x' is considered different of (0,0,1) is the first or second component is greater than 1/64

One flag that can optionally be added to a condition is:

```
CANREPEAT: yes
```

It is written after CONDMESHTYPE and means that one condition can be assigned to the same entity several times.

Self Calculated #FUNC# fields:

Another type of field that can be included inside a condition is a #FUNC# to do some calculation,

where the key #FUNC#, means that the value of this field will be calculated just when the mesh is generated. It can be considered as a function that evaluates when meshing.

Valid variables for a #FUNC# field are:

- `NumEntity`: to track the numerical id of the geometric source entity
- `x y z` : to use the coordinates of the node or entity center where the condition is applied
- `Cond(num_field,REAL)`: to use the value of other fields of this condition (REAL or INT declare that must be considered as a real or a integer number)
- Valid mathematical operations are the same as the used for the *Operation template command.

e.g.

```
QUESTION: Surface_number#FUNC#(NumEntity)
```

```
VALUE: 0
```

In the above example, `NumEntity` is one of the possible variables of the function. It will be substituted by the label of the geometrical entity from where the node or element is generated.

```
QUESTION:                    X_press#FUNC#(Cond(3,REAL)*(x-Cond(1,REAL))/
(Cond(2,REAL)-Cond(1,REAL)))
VALUE: 0
```

In this second example, the `x` variable is used, which means the x-coordinate of the node or of the center of the element. Others fields of the condition can also be used in the function. Variables `y` and `z` give the y- and z-coordinates of this point.

Note: There are other options available to expand the capabilities of the Conditions window (see Special fields -pag. 20-).

3.2.1 Example: Creating the conditions file

Here is an example of how to create a conditions file, explained step by step:

3.2.1.1 First, you have to create the folder or directory where all the problem type files are located, problem_type_name.gid in this case.

3.2.1.2 Then create and edit the file (problem_type_name.cnd in this example) inside the recently created directory (where all your problem type files are located). As you can see, except for the extension, the names of the file and the directory are the same.

3.2.1.3 Create the first condition, which starts with the line:

```
CONDITION: Point-Constraints
```

The parameter is the name of the condition. A unique condition name is required for this conditions file.

3.2.1.4 This first line is followed by the next pair:

```
CONDTYPE: over points
CONDMESHTYPE: over nodes
```

which declare what entity the condition is going to be applied over. The first line, CONDTYPE:... refers to the geometry, and may take as parameters the sentences "over points", "over lines", "over surfaces" or "over volumes".

The second line refers to the type of condition applied to the mesh, once generated. GiD does not force you to provide this second parameter, but if it is present, the treatment and evaluation of the problem will be more acurate. The available parameters for this statement are "over nodes" and "over elements".

3.2.1.5 Next, you have to declare a set of questions and values applied to this condition.

```
QUESTION: Local-Axes#LA#(-GLOBAL-)
```

```
VALUE: -GLOBAL-

QUESTION: X-Force

VALUE: 0.0

QUESTION: X-Constraint:#CB#(1,0)

VALUE: 1

QUESTION: X_axis:#CB#(DEFORMATION_XX,DEFORMATION_XY,DEFORMATION_XZ)

VALUE: DEFORMATION_XX

END CONDITION
```

After the QUESTION: prompt, you have the choice of putting the following kinds of word:

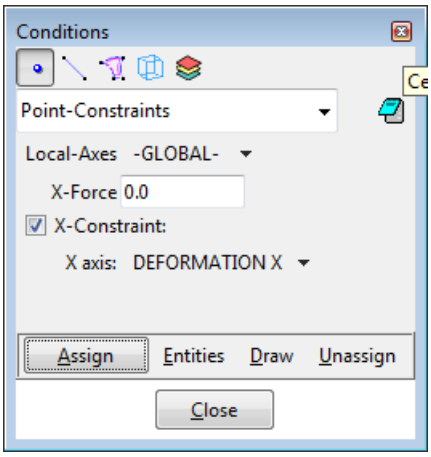
- An alphanumeric field name.
- An alphanumeric field name followed by the #LA# statement, and then the single or double parameter.
- An alphanumeric field name followed by the #CB# statement, and then the optional values between parentheses.

The VALUE: prompt must be followed by one of the optional values, if you have declared them in the previous QUESTION: line. If you do not observe this format, the program may not work correctly.

In the previous example, the X-Force QUESTION takes the value 0.0. Also in the example, the X-Constraint QUESTION includes a Combo Box statement (#CB#), followed by the declaration of the choices 1 and 0. In the next line, the value takes the parameter 1. The X_axis QUESTION declares three items for the combo box: DEFORMATION_XX,DEFORMATION_XY,DEFORMATION_XZ, with the value DEFORMATION_XX chosen.

Beware of leaving blank spaces between parameters. If in the first question you put the optional values (-GLOBAL, -AUTO-) (note the blank space after the comma) there will be an error when reading the file. Take special care in the Combo Box question parameters, so as to avoid unpredictable parameters.

3.2.1.6 The conditions defined in the .cnd file can be managed in the Conditions window (found in the **Data** menu) in the Preprocessing component of GiD.



Conditions window in GiD Preprocessing

3.3 Problem and intervals data file (.prb)

Files with the extension .prb contain all the information about general problem and intervals data. The general problem data is all the information required for performing the analysis and it does not concern any particular geometrical entity. This differs from the previous definitions of conditions and materials properties, which are assigned to different entities. An example of general problem data is the type of solution algorithm used by the solver, the value of the various time steps, convergence conditions and so on.

Within this data, one may consider the definition of specific problem data (for the whole process) and intervals data (variable values along the different solution intervals). An interval would be the subdivision of a general problem that contains its own particular data. Typically, one can define a different load case for every interval or, in dynamic problems, not only variable loads, but also variation in time steps, convergence conditions and so on.

The format of the file is as follows:

```
PROBLEM DATA
QUESTION: field_name['#CB#'(...,optional_value_i,...)]
VALUE: default_field_value
  ...
QUESTION: field_name['#CB#'(...,optional_value_i,...)]
VALUE: default_field_value
END PROBLEM DATA

INTERVAL DATA
QUESTION: field_name['#CB#'(...,optional_value_i,...)]
VALUE: default_field_value
  ...
```

```
QUESTION: field_name['#CB#'(...,optional_value_i,...)]
VALUE: default_field_value
END INTERVAL DATA
```

All the fields must be filled with words, where a word is considered as a string of characters without any blank spaces. The strings signaled between quotes are literal and the ones inside brackets are optional. The interface is case-sensitive, so any uppercase letters must be maintained. The default_field_value entry and various optional_value_i entries can be alphanumeric, integers or real numbers, depending on the type.

Note: There are other options available to expand the capabilities of the Problem Data window (see Special fields -pag. 20-).

3.3.1 Example: Creating the PRB data file

Here is an example of how to create a problem data file, explained step by step:

3.3.1.1 Create and edit the file (problem_type_name.prb in this example) inside the problem_type_name directory (where all your problem type files are located). Except for the extension, the names of the file and the directory must be the same.

3.3.1.2 Start the file with the line:

```
PROBLEM DATA
```

3.3.1.3 Then add the following lines:

```
QUESTION: Unit_System#CB#(SI,CGS,User)
VALUE: SI
QUESTION: Title
VALUE: Default_title
```

The first question defines a combo style menu called Unit_System, which has the SI option selected by default. The second question defines a text field called Title, and its default value is Default_title.

3.3.1.4 To end the file, add the following line:

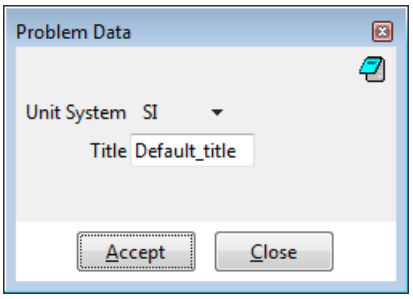
```
END PROBLEM DATA
```

3.3.1.5 The whole file is as follows:

```
PROBLEM DATA
QUESTION: Unit_System#CB#(SI,CGS,User)
VALUE: SI
QUESTION: Title
VALUE: Default_title
```

```
END PROBLEM DATA
```

3.3.1.6 The options defined in the .prb file can be managed in the Problem Data window (found in the **Data** menu) in the Preprocessing component of GiD.



Problem Data window in GiD Preprocessing

3.4 Materials file (.mat)

Files with the extension .mat include the definition of different materials through their properties. These are base materials as they can be used as templates during the Preprocessing step for the creation of newer ones.

You can define as many materials as you wish, with a variable number of fields. None of the unused materials will be taken into consideration when writing the data input files for the solver. Alternatively, they can be useful for generating a materials library.

Conversely to the case of conditions, the same material can be assigned to different levels of geometrical entity (lines, surfaces or volumes) and can even be assigned directly to the mesh elements.

In a similar way to how a condition is defined, a material can be considered as a group of fields containing its name, its corresponding properties and their values.

The format of the file is as follows:

```
MATERIAL: material_name
QUESTION: field_name['#CB#'(...,optional_value_i,...)]
VALUE: default_field_value
  ...
QUESTION: field_name['#CB#'(...,optional_value_i,...)]
VALUE: default_field_value
END MATERIAL

MATERIAL: material_name
  ...
END MATERIAL
```

If a material has a variable property (an example would be where a property was

dependent on temperature and was defined with several values for several temperatures) a table of changing values may be declared for this property. When the solver evaluates the problem, it reads the values and applies a suitable property value.

The declaration of the table requires two lines of text:

The first is a QUESTION line with a list of alphanumeric values between parentheses.

```
QUESTION: field_name:(...,optional_value_i,...)
```

These values are the names of each of the columns in the table so that the number of values declared is the number of columns.

This first line is followed by another with the actual data values. It starts with the words VALUE: #N#, and is followed by a number that indicates the quantity of elements in the matrix and, finally, the list of values.

```
VALUE: #N# number_of_values ... value_number_i ...
```

The number of values declared for the matrix obviously has to be the number of columns multiplied by the number of rows to be declared.

This kind of material specification is most likely to be used in thermo-mechanical simulations, where the problem is exposed to a temperature variation, and the properties of the materials change for each temperature value.

All the fields must be filled with words, where a word is considered as a string of characters without any blank spaces. The strings signaled between quotes are literal and the ones within brackets are optional. The interface is case-sensitive, so any uppercase letters must be maintained. The default_field_value entry and various optional_value_i entries can be alphanumeric, integers or real numbers, depending on their type.

Note: There are other options available to expand the capabilities of the Materials window (see Special fields -pag. 20-).

3.4.1 Example: Creating the materials file

Here is an example of how to create a materials file, explained step by step:

3.4.1.1 Create and edit the file (problem_type_name.mat in this example) inside the problem_type_name directory (where all your problem type files are located). As you can see, except for the extension, the names of the file and the directory are the same.

3.4.1.2 Create the first material, which starts with the line:

```
MATERIAL: Air
```

The parameter is the name of the material. A unique material name is required for this into this materials file (do not use blank spaces in the name of the material).

3.4.1.3 The next two lines define a property of the material and its default value:

```
QUESTION: Density
VALUE: 1.0
```

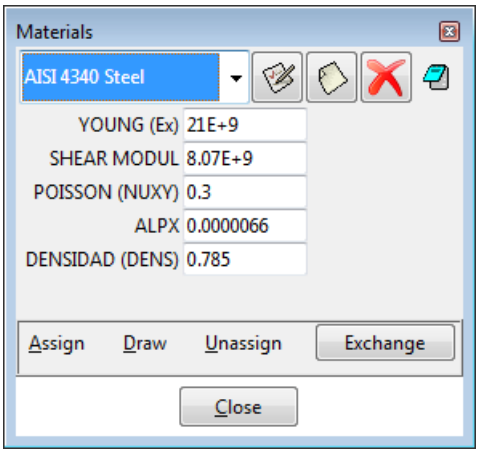
You can add as many properties as you wish. To end the material definition, add the following line:

```
END MATERIAL
```

3.4.1.4 In this example we have introduced some materials; the .mat file would be as follows:

```
MATERIAL: Air
QUESTION: Density
VALUE: 1.01
END MATERIAL

MATERIAL: AISI_4340_Steel
QUESTION: YOUNG_(Ex)
VALUE: 21E+9
QUESTION: SHEAR_MODUL
VALUE: 8.07E+9
QUESTION: POISSON_(NUXY)
VALUE: 0.3
QUESTION: ALPX
VALUE: 0.0000066
QUESTION: DENSIDAD_(DENS)
VALUE: 0.785
END MATERIAL

MATERIAL: Concrete
QUESTION: Density
VALUE: 2350
END MATERIAL
```

3.4.1.5 The materials defined in the .mat file can be managed in the Materials window (found in the **Data** menu) in the Preprocessing component of GiD.



Materials window in GiD Preprocessing

3.5 Special fields

These fields are useful for organizing the information within data files. They make the information shown in the data windows more readable. In this way you can better concentrate on the data properties relevant to the current context.

- **Book:** With the **Book** field it is possible to split the data windows into other windows. For example, we can have two windows for the materials, one for the steels and another for the concretes:

```
BOOK: Steels
```

...

All steels come here

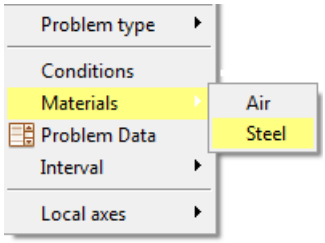
...

```
BOOK: Concretes
```

...

All concretes come here

...



Options corresponding to books

The same applies to conditions. For general and interval data the **book** field groups a set of properties.

- **Title:** The **Title** field groups a set of properties on different tabs of one book. All properties appearing after this field will be included on this tab.

```
TITLE: Basic
```

...

Basics properties

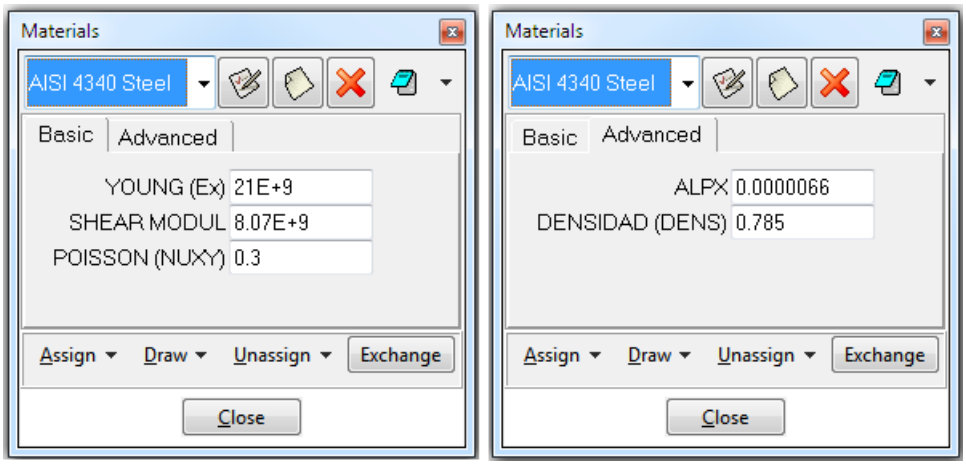
....

```
TITLE: Advanced
```

...

Advanced properties

....

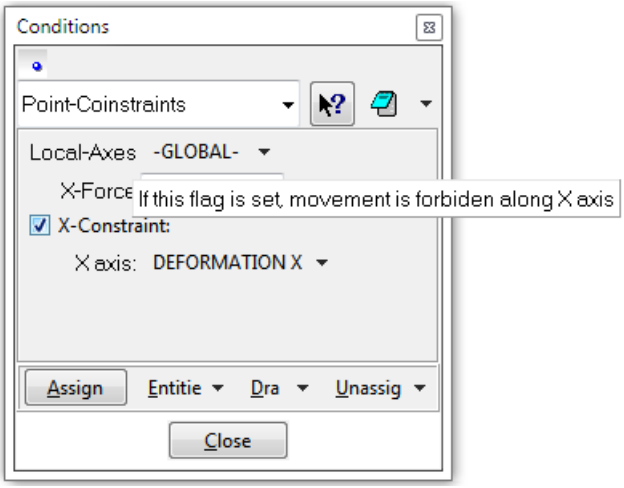


- **Help:** With the **Help** field it is possible to assign a description to the data property preceding it. In this way you can inspect the meaning of the property through the help context function by holding the cursor over the property or by right-clicking on it.

QUESTION: X-Constraint#CB#(1,0)

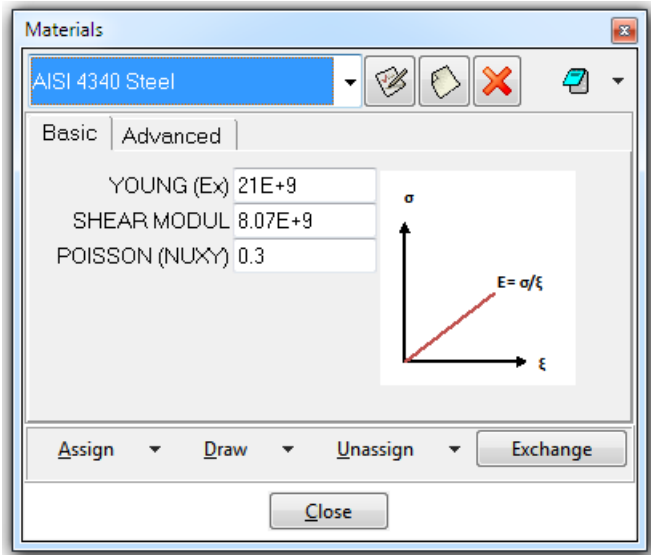
VALUE: 1

HELP: If this flag is set, movement is ...



- **Image:** The **Image** field is useful for inserting descriptive pictures in the data window. The value of this field is the file name of the picture relating to the problem type location.

IMAGE: young.gif



Data window with an image

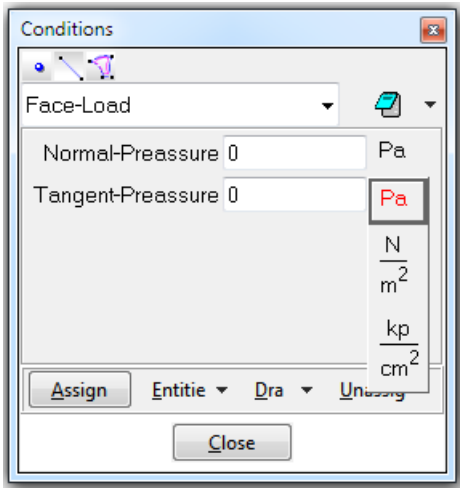
- **Unit field:** With this feature it is possible to define and work with properties that have units. **GiD** is responsible for the conversion between units of the same magnitude

....

QUESTION: Normal_Pressure#UNITS#

VALUE: 0.0Pa

...



Data property with units

- **Dependencies:** Depending on the value, we can define some behavior associated with the property. For each value we can have a list of actions. The syntax is as follows:

DEPENDENCIES:(<V1>,[TITLESTATE,<Title>,<State>],<A1>,<P1>,<NV1>,...,<An>,<Pn>,<NVn>) ... (<Vm>,<Am>,<Pm>,<NVm>,...)

where:

3.5..1 <Vi> is the value that triggers the actions. A special value is **#DEFAULT#**, which refers to all the values not listed.

3.5..2 **[TITLESTATE,<Title>,<State>]** this argument is optional. **Titlestate** should be used to show or hide book labels. Many **Titlestate** entries can be given. **<Title>** is the title defined for a book (TITLE: Title). **State** is the visualization mode: normal or hidden.

3.5..3 **<Ai>** is the action and can have one of these values: **SET, HIDE, RESTORE**. All these actions change the value of the property with the following differences: SET disables the property, HIDE hides the property and RESTORE brings the property to the enabled state.

3.5..4 **<Pi>** is the name of the property to be modified.

3.5..5 **<NVi>** is the new value of **<Pi>**. A special value is **#CURRENT#**, which refers to the current value of **<Pi>**.

Here is an example:

```
...
TITLE: General
QUESTION: Type_of_Analysis:#CB#(FILLING,SOLIDIFICATION)
VALUE: SOLIDIFICATION
DEPENDENCIES: (FILLING,TITLESTATE,Filling-Strategy,normal,RESTORE,
Filling_Analysis,GRAVITY,HIDE,Solidification_Analysis,#CURRENT#)
DEPENDENCIES: (SOLIDIFICATION,TITLESTATE,Filling-Strategy,hidden,HIDE,
Filling_Analysis,#CURRENT#,RESTORE,Solidification_Analysis,#CURRENT#)
TITLE: Filling-Strategy
QUESTION: Filling_Analysis:#CB#(GRAVITY,LOW-PRESSURE,FLOW-RATE)
VALUE: GRAVITY
QUESTION: Solidification_Analysis:#CB#(THERMAL,THERMO-MECHANICAL)
VALUE: THERMAL
...
```

- **State:** Defines the state of a field; this state can be: disabled, enabled or hidden. Here is an example:

```
...
QUESTION: Elastic modulus XX axis
VALUE: 2.1E+11
STATE: HIDDEN
...
```

- **#MAT#('BookName'):** Defines the field as a material, to be selected from the list of materials in the book 'BookName'. Here is an example:

```
QUESTION:Composition_Material#MAT#(BaseMat)
VALUE:AISI_4340_STEEL
```

3.6 Unit System file (.uni)

When GiD is installed, the file units.gid is copied within the GiD directory. In this file a table of magnitudes is defined. For each magnitude there is a set of units and a conversion factor between the unit and the reference unit. The units systems are also defined. A unit system is a set of mangnitudes and the corresponding unit.

```
BEGIN TABLE
 LENGTH : m, 100 cm, 1e+3 mm
 ...
 STRENGTH : kg*m/s^2, N, 1.0e-1 kp
END

BEGIN SYSTEM(INTERNATIONAL)
 LENGTH : m
 MASS : kg
 STRENGTH : N
 ...
 TEMPERATURE : Cel
END
```

The syntax of the unit file (problem_type_name.uni) within the problem type is similar. It can include the line:

```
USER DEFINED: ENABLED
```

(or `DISABLED`)

meaning that the user is able (or not able) to define his own system unit within the project. If the line does not appear in the file the value is assumed to be ENABLED.

It is possible to ignore all units systems defined by default inside the file units.gid:

```
USE BASE SYSTEMS: DISABLED
```

(or `ENABLED`)

With the command HIDDEN: 'magnitude', 'magnitude' certain magnitudes will not be displyed in the Problem units window.

```
HIDDEN: strength, pressure
```

If the problem type uses a property which has a unit, then GiD creates the file project_name.uni in the project directory. This file includes the information related to the unit used in the geometric model and the unit system used. The structure of this file is:

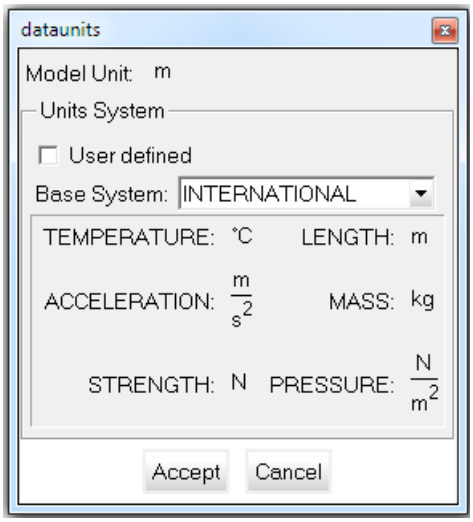
```
MODEL: km
PROBLEM: USER DEFINED
```

```
BEGIN SYSTEM
LENGTH: m
PRESSURE: Pa
MASS: kg
STRENGTH: N
END
```

In this file, **MODEL** refers to the unit of the geometric model and **PROBLEM** is the name of the units system used by GiD to convert all the data properties in the output to the solver. If this name is **USER DEFINED**, then the system is the one defined within the file. The block

```
BEGIN SYSTEM
...
END
```

corresponds to the user-defined system.



Data unit window

3.7 Conditions symbols file (.sim)

Files with the extension .sim comprise different symbols to represent some conditions during the preprocessing stage. You can define these symbols by creating ad hoc geometrical drawings and the appropriate symbol will appear over the entity with the applied condition every time you ask for it.

One or more symbols can be defined for every condition and the selection will depend on the specified values in the file, which may be obtained through mathematical conditions.

The spatial orientation can also be defined in this file, depending on the values taken by the required data. For global definitions, you have to input the three components of a vector to express its spatial direction. GiD takes these values from the corresponding

conditions window. The orientation of the vector can be understood as the rotation from the vector (1,0,0) towards the new vector defined in the file.

For line and surface conditions, the symbols may be considered as local. In this case, GiD does not consider the defined spatial orientation vector and it takes its values from the line or surface orientation. The orientation assumes the vector (1,0,0) to be the corresponding entity's normal.

These components, making reference to the values obtained from the adequate conditions, may include C-language expressions. They express the different field values of the mentioned condition as cond(type,i), where type (real or int) refers to the type of variable (not case-sensitive) and i is the number of the field for that particular condition.

3.7.1 Example: Creating the Symbols file

Here is an example of how to create a symbols file. Create and edit the file (problem_type_name.sim in this example) inside the problem_type_name directory (where all your problem type files are located). Except for the extension, the names of the file and the directory must be the same.

The contents of the problem_type_name.sim example should be the following:

```
cond Point-Constraints
3
global
cond(int,5)
1
0
0
Support3D.geo
global
cond(int,1) && cond(int,3)
1
0
0
Support.geo
global
cond(int,1) || cond(int,3)
cond(int,3)
cond(int,1)*(-1)
0
```

```
Support-2D.geo
cond Face-Load
1
local
fabs(cond(real,2)) + fabs(cond(real,4)) + fabs(cond(real,6))>0.
cond(real,2)
cond(real,4)
cond(real,6)
Normal.geo
```

This is a particular example of the .sim file where four different symbols have been defined. Each one is read from a ***.geo file. There is no indication of how many symbols are implemented overall. GiD simply reads the whole file from beginning to end.

The ***.geo files are obtained through GiD. You can design a particular drawing to symbolize a condition and this drawing will be stored as problem_name.geo when saving this project as problem_name.gid. You do not need to be concerned about the size of the symbol, but should bear in mind that the origin corresponds to the point (0,0,0) and the reference vector is (1,0,0). Subsequently, when these ***.geo files are invoked from problem_type_name.sim, the symbol drawing appears scaled on the display at the entity's location.

Nevertheless, the number of symbols and, consequently, the number of ***.geo files can vary from one condition to another. In the previous example, for instance, the condition called Point-Constraints, which is defined by using cond, comprises three different symbols. GiD knows this from the number 3 written below the condition's name. Next, GiD looks to see if the orientation is relative to the spatial axes (global) or moves together with its entity (local). In the example, the three symbols concerning point constraints are globally oriented.

Imagine that this condition has six fields. The first, third and fifth field values express if any constraint exist along the X-axis, the Y-axis and the Z-axis, respectively. These values are integers and in the case that they are null, the degree of freedom in question is assumed to be unconstrained.

For the first symbol, obtained from the file Support3D.geo, GiD reads cond(int,5), or the Z-constraint. If it is false, which means that the value of the field is zero, the C-condition will not be satisfied and GiD will not draw it. Otherwise, the C-condition will be satisfied and the symbol will be invoked. When this occurs, GiD skips the rest of the symbols related to this condition. Its orientation will be the same as the original drawing because the spatial vector is (1,0,0).

All these considerations are valid for the second symbol, obtained from the file Support.geo, but now GiD has to check that both constraints (&&) - the X-constraint and the Y-constraint - are fixed (their values are not zero).

For the third symbol, obtained from the file Support-2D.geo, only one of them has to be fixed (||) and the orientation of the symbol will depend on which one is free and which one is fixed, showing on the screen the corresponding direction for both degrees of freedom.

Finally, for the fourth symbol, onbtained from the file Normal.geo, it can be observed that the drawing of the symbol, related to the local orientation will appear scaled according to the real-type values of the second, fourth and sixth field values. Different types of C-language expressions are available in GiD. Thus, the last expression would be equivalent to entering '(fabs(cond(real,2))>0. || fabs(cond(real,4))!=0. || fabs(cond(real,6))>1e-10)'.

Note: As previously mentioned, GiD internally creates a project_name.geo file when saving a project, where it keeps all the information about the geometry in binary format. In fact, this is the reason why the extension of these files is .geo. However, the file project_name.geo is stored in the project_name.gid directory, whereas these user-created ***.geo files are stored in the problem_type_name.gid directory.

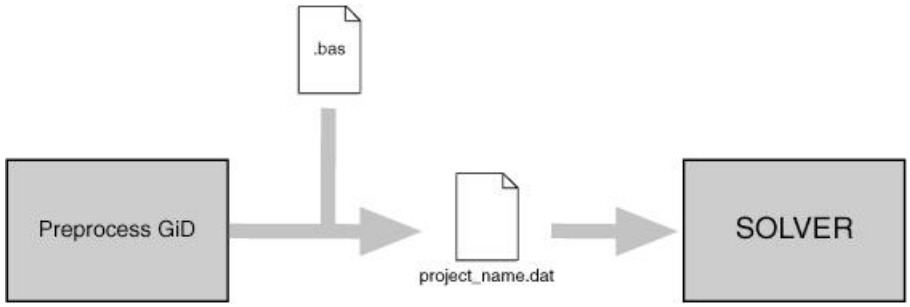
4 TEMPLATE FILES

Once you have generated the mesh, and assigned the conditions and the materials properties, as well as the general problem and intervals data for the solver, it is necessary to produce the data input files to be processed by that program.

To manage this reading, GiD is able to interpret a file called problem_type_name.bas (where problem_type_name is the name of the working directory of the problem type without the .bas extension).

This file (template file) describes the format and structure of the required data input file for the solver that is used for a particular case. This file must remain in the problem_type_name.gid directory, as well as the other files already described - problem_type_name.cnd, problem_type_name.mat, problem_type_name.prb and also problem_type_name.sim and ***.geo, if desired.

In the case that more than one data input file is needed, GiD allows the creation of more files by means of additional ***.bas files (note that while problem_type_name.bas creates a data input file named project_name.dat, successive ***.bas files - where *** can be any name - create files with the names project_name-1.dat, project_name-2.dat, and so on). The new files follow the same rules as the ones explained next for problem_type_name.bas files.



These files work as an interface from GiD's standard results to the specific data input for any individual solver module. This means that the process of running the analysis simply forms another step that can be completed within the system.

In the event of an error in the preparation of the data input files, the programmer has only to fix the corresponding problem_type_name.bas or ***.bas file and rerun the example, without needing to leave GiD, recompile or reassign any data or re-mesh.

This facility is due to the structure of the template files. They are a group of macros (like an ordinary programming language) that can be read, without the need of a compiler, every time the corresponding analysis file is to be written. This ensures a fast way to debug mistakes.

4.1 Commands used in the .bas file

4.1.1 Single value return commands

When writing a command, it is generally not case-sensitive (unless explicitly mentioned), and even a mixture of uppercase and lowercase will not affect the results.

- ***npoin**, ***ndime**, ***nnode**, ***nelem**, ***nmats, *nintervals.** These return, respectively, the number of points, the dimensions of the project being considered, the number of nodes of the element with the highest number, the number of elements, the number of materials and the number of data intervals. All of them are considered as integers and do not carry arguments (see `*format,*intformat)`, except `*nelem`, which can bring different types of elements. These elements are: `Point`, `Linear`, `Triangle`, `Quadrilateral`, `Tetrahedra`, `Hexahedra`, `Prism`, `Pyramid`, `Sphere`, depending on the number of edges the element has, and All, which comprises all the possible types. The command `*nmats` returns the number of materials effectively assigned to an entity, not all the defined ones.

- ***GenData.** This must carry an argument of integer type that specifies the number of the field to be printed. This number is the order of the field inside the general data list. This must be one of the values that are fixed for the whole problem, independently of the interval (see Problem and intervals data file (.prb) -pag. 15-). The name of the field, or an abreviation of it, can also be the argument instead. The arguments `REAL` or `INT`, to express the type of number for the field, are also available (see `*format,*intformat,*realformat,*if)`. If they are not specified, the program will print a character string. It is mandatory to write one of them within an expression, except for strcmp and strcasecmp. The numeration must start with the number 1.

Note: Using this command without any argument will print all fields

- ***IntvData.** The only difference between this and the previous command is that the field must be one of those fields varying with the interval (see Problem and intervals data file (.prb) -pag. 15-). This command must be within a loop over intervals (see `*loop`) and the program will automatically update the suitable value for each iteration.

Note: Using this command without any argument will print all fields

- ***MatProp.** This is the same as the previous command except that it must be within a loop over the materials (see `*loop`). It returns the property whose field number or name is defined by its argument. It is recommended to use names instead of field numbers.

  If the argument is 0, it returns the material's name.

Note: Using this command without any argument will print all fields

Caution: If there are materials with different numbers of fields, you must ensure not to print non-existent fields using conditionals.

- ***ElemsMatProp.** This is the same as Matprop but uses the material of the current element. It must be within a loop over the elements (see *loop). It returns the property whose field number or name is defined by its argument. It is recommended to use names instead of field numbers.

Example:

```
*loop elements
  *elemsnum *elemsmat *elemsmatprop(young)
*end elements
```

- ***Cond.** The same remarks apply here, although now you have to notify with the command *set (see *set) which is the condition being processed. It can be within a loop (see *loop) over the different intervals should the conditions vary for each interval.

Note: Using this command without any argument will print all fields

- ***CondName.** This returns the conditions's name. It must be used in a loop over conditions or after a *set cond command.
- ***CondNumFields.** This returns the number of fields of the current condition. It must be used in a loop over conditions or after *set cond
- ***CondHasLocalAxes**. returns 1 if the condition has a local axis field, 0 else
- ***CondNumEntities.** You must have previously selected a condition (see *set cond). This returns the number of entities that have a condition assigned over them.

- ***ElemsNum**: This returns the element's number.

***NodesNum**: This returns the node's number.

***MatNum**: This returns the material's number.

***ElemsMat**: This returns the number of the material assigned to the element.

All of these commands must be within a proper loop (see `*loop`) and change automatically for each iteration. They are considered as integers and cannot carry any argument. The number of materials will be reordered numerically, beginning with number 1 and increasing up to the number of materials assigned to any entity.

- ***LayerNum**: This returns the layer's number.

***LayerName**: This returns the layer's name.

***LayerColorRGB**: This returns the layer's color in RGB (three integer numbers between 0 and 256). If parameter (1), (2) or (3) is specified, the command returns only the value of one color. RED is 1, GREEN is 2 and BLUE is 3.

The commands ***LayerName**, ***LayerNum** and ***LayerColorRGB** must be inside a loop over layers; you cannot use these commands in a loop over nodes or elements.

Example:

```
*loop layers
*LayerName *LayerColorRGB
*Operation(LayerColorRGB(1)/255.0)        *Operation(LayerColorRGB(2)/255.0)
*Operation(LayerColorRGB(3)/255.0)
*end layers
```

- ***NodesLayerNum**: This returns the layer's number. It must be used in a loop over nodes.

***NodesLayerName**: This returns the layer's name. It must be used in a loop over nodes.

***ElemsLayerNum**: This returns the layer's number. It must be used in a loop over elems.

***ElemsLayerName**: This returns the layer's name. It must be used in a loop over elems.

- ***LayerNumEntities.** You must have previously selected a layer (see *set layer). This returns the number of entities that are inside this layer.

- ***LoopVar.** This command must be inside a loop and it returns, as an integer, what is considered to be the internal variable of the loop. This variable takes the value 1 in the first iteration and increases by one unit for each new iteration. The parameter elems,nodes,materials,intervals, used as an argument for the corresponding loop, allows the program to know which one is being processed. Otherwise, if there are nested loops, the program takes the value of the inner loop.

- ***Operation.** This returns the result of an arithmetical expression what should be written inside parentheses immediately after the command. This operation must be defined in C-format and can contain any of the commands that return one single value. You can force an integer or a real number to be returned by means of the parameters INT or REAL. Otherwise, GiD returns the type according to the result.

The valid C-functions that can be used are:

- `+,-,*,/,%,(,),=,<,>,!,&,|,` numbers and variables
- `sin`
- `cos`
- `tan`
- `asin`
- `acos`
- `atan`
- `atan2`
- `exp`
- `fabs`
- `abs`
- `pow`
- `sqrt`

- log
- log10
- max
- min
- strcmp
- strcasecmp

The following are valid examples of operations:

```
*operation(4*elemsnum+1)
```

```
*operation(8*(loopvar-1)+1)
```

Note: There cannot be blank spaces between the commands and the parentheses that include the parameters.

Note: Commands inside `*operation` do not need `*` at the beginning.

- ***LocalAxesNum.** This returns the identification name of the local axes system, either when the loop is over the nodes or when it is over the elements, under a referenced condition.
- ***nlocalaxes.** This returns the number of the defined local axes system.
- ***IsQuadratic.** This returns the value 1 when the elements are quadratic or 0 when they are not.
- ***Time.** This returns the number of seconds elapsed since midnight.
- ***Clock.** This returns the number of clock ticks (aprox. milliseconds) of elapsed processor time.

Example:

```
*set var t0=clock
*loop nodes
    *nodescoord
*end nodes
*set var t1=clock
ellapsed time=*operation((t1-t0)/1000.0) seconds
```

- ***Units('magnitude').** This returns the current unit name for the selected magnitude (the current unit is the unit shown inside the unit window).

Example:

*Units(LENGTH)

- ***BasicUnit('magnitude').** This returns the basic unit name for the selected magnitude (the basic unit is the unit defined as { Basic } in the *.uni file).

Example:

```
*BasicUnit(LENGTH)
```

- ***FactorUnit('unit').** This returns the numeric factor to convert a magnitude from the selected unit to the basic unit.

Example:

```
*FactorUnit(PRESSURE)
```

4.1.2 Multiple values return commands

These commands return more than one value in a prescribed order, writing them one after the other. All of them except LocalAxesDef are able to return one single value when a numerical argument giving the order of the value is added to the command. In this way, these commands can appear within an expression. Neither LocalAxesDef nor the rest of the commands without the numerical argument can be used inside expressions. Below, a list of the commands with the appropriate description is displayed.

- ***NodesCoord.** This command writes the node's coordinates. It must be inside a loop (see *loop) over the nodes or elements. The coordinates are considered as real numbers (see *realformat and *format). It will write two or three coordinates according to the number of dimensions the problem has (see *Ndime).

If *NodesCoord receives an integer argument (from 1 to 3) inside a loop of nodes, this argument indicates which coordinate must be written: x, y or z. Inside a loop of nodes:

`*NodesCoord` writes three or two coordinates depending on how many dimensions there are.

`*NodesCoord(1)` writes the **x** coordinate of the actual node of the loop.

`*NodesCoord(2)` writes the **y** coordinate of the actual node of the loop.

`*NodesCoord(3)` writes the **z** coordinate of the actual node of the loop.

If the argument real is given, the coordinates will be treated as real numbers.

Example: using *NodesCoord inside a loop of nodes

```
Coordinates:
Node X Y
*loop nodes
*format "%5i%14.5e%14.5e"
*NodesNum *NodesCoord(1,real) *NodesCoord(2,real)
*end nodes
```

This command effects a rundown of all the nodes in the mesh, listing their identifiers and coordinates (**x** and **y**).

The contents of the project_name.dat file could be something like this:

```
Coordinates:
Node X Y
    1 -1.28571e+001 -1.92931e+000
    2 -1.15611e+001 -2.13549e+000
    3 -1.26436e+001 -5.44919e-001
    4 -1.06161e+001 -1.08545e+000
    5 -1.12029e+001 9.22373e-002
    ...
```

*NodesCoord can also be used inside a loop of elements. In this case, it needs an additional argument that gives the local number of the node inside the element. After this argument it is also possible to give which coordinate has to be written: x, y or z.

Inside a loop of elements:

`*NodesCoord(4)` writes the coordinates of the 4th node of the actual element of the loop.

`*NodesCoord(5,1)` writes the **x** coordinate of the 5th node of the actual element of the loop.

`*NodesCoord(5,2)` writes the **y** coordinate of the 5th node of the actual element of the loop.

`*NodesCoord(5,3)` writes the **z** coordinate of the 5th node of the actual element of the loop.

- ***ElemsConec.** This command writes the element's connectivities, i.e. the list of the nodes that belong to the element, displaying the direction for each case (anti-clockwise direction in 2D, and depending on the standards in 3D). For shells, the direction must be defined. However, this command accepts the argument swap and this implies that the ordering of the nodes in quadratic elements will be consecutive instead of hierarchical. The connectivities are considered as integers (see *intformat and *format).

If *ElemsConec receives an integer argument (begining from 1), this argument indicates which element connectity must be written:

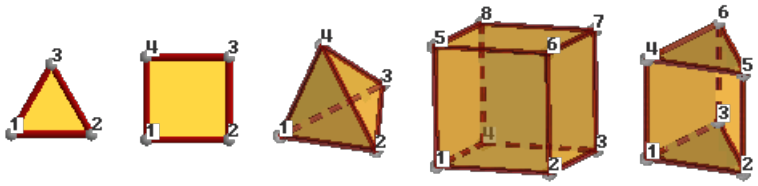
```
*loop elems
  all conectivities: *elemsconec
  first conectivity *elemsconec(1)
*end elems
```

Note: In the first versions of GiD, the optional parameter of the last command explained

was invert instead of swap, as it is now. It was changed due to technical reasons. If you have an old .bas file prior to this specification, which contains this command in its previous form, when you try to export the calculation file, you will be warned about this change of use. Be aware that the output file will not be created as you expect.

- ***GlobalNodes.** This command returns the nodes that belong to an element's face where a condition has been defined (on the loop over the elements). The direction for this is the same as for that of the element's connectivities. The returned values are considered as integers (see `*intformat` and `*format`).If `*GlobalNodes` receives an integer argument (beginning from 1), this argument indicates which face connectity must be written.

So, the local numeration of the faces is:



Triangle: 1-2 2-3 3-1

Quadrilateral: 1-2 2-3 3-4 4-1

Tetrahedra: 1-2-3 2-4-3 3-4-1 4-2-1

Hexahedra: 1-2-3-4 1-4-8-5 1-5-6-2 2-6-7-3 3-7-8-4 5-8-7-6

Prism: 1-2-3 1-4-5-2 2-5-6-3 3-6-4-1 4-5-6

Pyramid: 1-2-3-4 1-5-2 2-5-3 3-5-4 4-5-1

- ***LocalNodes.** The only difference between this and the previous one is that the returned value is the local node's numbering for the corresponding element (between 1 and nnode).

- ***CondElemFace.** This command return the number of face of the element where a condition has been defined (beginning from 1). The information is equivalent to the obtained with the localnodes command

- ***ElemsNnode.** This command returns the number of nodes of the current element (valid only inside a loop over elements).

Example:

```
*loop elems
  *ElemsNnode
*end elems
```

- ***ElemsNnodeCurt.** This command returns the number of vertex nodes of the current element (valid only inside a loop over elements). For example, for a quadrilateral of 4, 8 or 9 nodes, it returns the value 4.

- ***ElemsNNodeFace.** This command returns the number of face nodes of the current element face (valid only inside a loop over elements onlyincond, with a previous *set cond of a condition defined over face elements).

Example:

```
*loop elems
*ElemsNnodeFace
*end elems
```

- ***ElemsNNodeFaceCurt.** This command returns the short (corner nodes only) number of face nodes of the current element face (valid only inside a loop over elements onlyincond, with a previous *set cond of a condition defined over face elements).

Example:

```
*loop elems
*ElemsNnodeFaceCurt
*end elems
```

- ***ElemsType**: This returns the current element type as a integer value: 1=Linear, 2=Triangle, 3=Quadrilateral, 4=Tetrahedra, 5=Hexahedra, 6=Prism, 7=Point,8=Pyramid,9=Sphere,10=Circle. (Valid only inside a loop over elements.)
- ***ElemsTypeName**: This returns the current element type as a string value: `Linear`, `Triangle`, `Quadrilateral`, `Tetrahedra`, `Hexahedra`, `Prism`, `Point`, `Pyramid`, `Sphere`, `Circle`. (Valid only inside a loop over elements.)
- ***ElemsCenter**: This returns the element center. (Valid only inside a loop over elements.)

Note: This command is only available in GiD version 9 or later.

- ***ElemsRadius**: This returns the element radius. (Valid only inside a loop over sphere or Circle elements.)

Note: This command is only available in GiD version 8.1.1b or later.

- ***ElemsNormal.** This command writes the normal's coordinates. It must be inside a loop (see *loop) over elements, and it is only defined for triangles, quadrilaterals, and circles (and also for lines in 2D cases).

If *ElemsNormal receives an integer argument (from 1 to 3) this argument indicates which coordinate of the normal must be written: x, y or z.

- ***LocalAxesDef.** This command returns the nine numbers that define the transformation matrix of a vector from the local axes system to the global one.

Example:

```
*loop localaxes
```

```
*format "%10.4lg %10.4lg %10.4lg"
x'=*LocalAxesDef(1) *LocalAxesDef(4) *LocalAxesDef(7)
*format "%10.4lg %10.4lg %10.4lg"
y'=*LocalAxesDef(2) *LocalAxesDef(5) *LocalAxesDef(8)
*format "%10.4lg %10.4lg %10.4lg"
z'=*LocalAxesDef(3) *LocalAxesDef(6) *LocalAxesDef(9)
*end localaxes
```

- ***LocalAxesDef(EulerAngles).** This is as the last command, only with the EulerAngles option. It returns three numbers that are the 3 Euler angles (radians) that define a local axes system$^{(\phi,\theta,\psi)}$

$$\begin{pmatrix} \cos\psi\cos\phi-\sin\psi\cos\theta\sin\phi & \cos\psi\sin\phi+\sin\psi\cos\theta\cos\phi & \sin\psi\sin\theta \\ -\sin\psi\cos\phi-\cos\psi\cos\theta\sin\phi & -\sin\psi\sin\phi+\cos\psi\cos\theta\cos\phi & \cos\psi\sin\theta \\ \sin\theta\sin\phi & -\sin\theta\cos\phi & \cos\theta \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Rotation of a vector expressed in terms of euler angles.

- ***LocalAxesDefCenter.** This command returns the origin of coordinates of the local axes as defined by the user. The "Automatic" local axes do not have a center, so the point (0,0,0) is returned. The index of the coordinate (from 1 to 3) can optionally be given to LocalAxesDefCenter to get the x, y or z value.

Example:

```
*LocalAxesDefCenter
*LocalAxesDefCenter(1) *LocalAxesDefCenter(2) *LocalAxesDefCenter(3)
```

4.1.3 Specific commands

- ***** To avoid line-feeding you need to write *****, so that the line currently being used continues on the following line of the file filename.bas.
- ***#** If this is placed at the beginning of the line, it is considered as a comment and therefore is not written.
- ****** In order for an asterisk symbol to appear in the text, two asterisks ****** must be written.

- ***Include.** The include command allows you to include the contents of a slave file inside a master .bas file, setting a relative path from the Problem Type directory to this secondary file.

Example:

```
*include includes\execntrlmi.h
```

Note: The *.bas extension cannot be used for the slave file to avoid multiple output files.

- ***MessageBox.** This command stops the execution of the .bas file and prints a message in a window; this command should only be used when a fatal error occurs.

Example:

*MessageBox error: Quadrilateral elements are not permitted.

- ***WarningBox.** This is the same as MessageBox, but the execution is not stopped.

Example:

```
WarningBox Warning: Exist Bad elements. A STL file is a collection of
triangles bounding a volume.
```

The following commands must be written at the beginning of a line and the rest of the line will serve as their modifiers. No additional text should be written.

- ***loop**, ***end**, ***break**. These are declared for the use of loops. A loop begins with a line that starts with *loop (none of these commands is case-sensitive) and contains another word to express the variable of the loop. There are some lines in the middle that will be repeated depending on the values of the variable, and whose parameters will keep on changing throughout the iterations if necessary. Finally, a loop will end with a line that finishes with *end. After *end, you may write any kind of comments in the same line. The command ***break** inside a ***loop** or ***for** block, will finish the execution of the loop and will continue after the ***end** line.

The variables that are available for *loop are the following:

- **elems**, **nodes**, **materials,** **conditions**, **layers**, **intervals**, **localaxes**. These commands mean, respectively, that the loop will iterate over the elements, nodes, materials, conditions, layers, intervals or local axes systems. The loops can be nested among them. The loop over the materials will iterate only over the effectively assigned materials to an entity, in spite of the fact that more materials have been defined. The number of the materials will begin with the number 1. If a command that depends on the loop is located outside it, the number will also take by default the value 1.

After the command *loop:

- If the variable is **elems** or **nodes,** you can include one of the modifiers: ***all**, ***OnlyInCond** or ***OnlyInLayer** . The first one signifies that the iteration is going to be performed over all the entities; the *OnlyInCond modifier implies that the iteration will only take place over the entities that satisfy the relevant condition. This condition must have been previously defined with *set cond. *OnlyInLayer implies that the iteration will only take place over the entities that are in the specified layer; layers must be specified with the command *set Layer. By default, it is assumed that the iteration will affect all the entities.

- If the variable is **material** you can include the modifier ***NotUsed** to make a loop over those materials that are defined but not used.

- If the variable is **conditions** you must include one of the modifiers: ***Nodes**, ***BodyElements**, ***FaceElements** or ***Layers** , to do the loop on the conditions

defined over this kind of mesh entity.

- If the variable is **layers** you can include modifiers: **OnlyInCond** if before was set a condition defined 'over layers'

Example 1:

```
*loop nodes
*format "%5i%14.5e%14.5e"
*NodesNum *NodesCoord(1,real) *NodesCoord(2,real)
*end nodes
```

This command carries out a rundown of all the nodes of the mesh, listing their identifiers and coordinates (**x** and **y** coordinates).

Example 2:

```
*Set Cond Point-Weight *nodes
*loop nodes OnlyInCond
*NodesNum *cond(1)
*end
```

This carries out a rundown of all the nodes assigned the condition "Point-Weight" and provides a list of their identifiers and the first "weight" field of the condition in each case.

Example 3:

```
*Loop Elems
*ElemsNum *ElemsLayerNum
*End Elems
```

This carries out a rundown of all the elements and provides a list of their identifier and the identifier of the layer to which they belong.

Example 4:

```
*Loop Layers
*LayerNum *LayerName *LayerColorRGB
*End Layers
```

This carries out a rundown of all the layers and for each layer it lists its identifier and name.

Example 5:

```
*Loop Conditions OverFaceElements
  *CondName
```

```
  *Loop Elems OnlyInCond
*elemsnum *condelemface *cond
*End Elems
*End Conditions
```

This carries out a rundown of all conditions defined to be applied on the mesh 'over face elements', and for each condition it lists its name and for each element where this condition is applied are printed the element number, the marked face and the condition field values.

- ***if**, ***else**, ***elseif**, ***endif**. These commands create the conditionals. The format is a line which begins with *if followed by an expression between parenthesis. This expression will be written in C-language syntax, value return commands, will not begin with *, and its variables must be defined as integers or real numbers (see *format, *intformat, *realformat), with the exception of strcmp and strcasecmp. It can include relational as well as arithmetic operators inside the expressions.

The following are valid examples of the use of the conditionals:

```
*if((fabs(loopvar)/4)<1.e+2)
*if((p3<p2)||p4)
*if((strcasecmp(cond(1),"XLoad")==0)&&(cond(2)!=0))
```

The first example is a numerical example where the condition is satisfied for the values of the loop under 400, while the other two are logical operators; in the first of these two, the condition is satisfied when p3<p2 or p4 is different from 0, and in the second, when the first field of the condition is called XLoad (with this particular writing) and the second is not null.

If the checked condition is true, GiD will write all the lines until it finds the corresponding *else, *elseif or *endif (*end is equivalent to *endif after *if). *else or *elseif are optional and require the writing of all the lines until the corresponding *endif, but only when the condition given by *if is false. If either *else or *elseif is present, it must be written between *if and *endif. The conditionals can be nested among them.

The behaviour of *elseif is identical to the behaviour of *else with the addition of a new condition:

```
*if(GenData(31,int)==1)
...(1)
*elseif(GenData(31,int)==2)
...(2)
*else
...(3)
```

```
*endif
```

In the previous example, the body of the first condition (written as 1) will be written to the data file if GenData(31,int) is 1, the body of the second condition (written as 2) will be written to the data file if GenData(31,int) is 2, and if neither of these is true, the body of the third condition (written as 3) will be written to the data file. **Note:** A conditional can also be written in the middle of a line. To do this, begin another line and write the conditional by means of the command *****.

- ***for**, ***end**, ***break**. The syntax of this command is equivalent to *for in C-language.

```
*for(varname=expr.1;varname<=expr.2;varname=varname+1)
*end for
```

The meaning of this statement is the execution of a controlled loop, since varname is equal to expr.1 until it is equal to expr.2, with the value increasing by 1 for each step. varname is any name and expr.1 and expr.2 are arithmetical expressions or numbers whose only restrictions are to express the range of the loop.

The command ***break** inside a ***loop** or ***for** block, will finish the execution of the loop and will continue after the ***end** line.

Example:

```
*for(i=1;i<=5;i=i+1)
variable i=*i
*end for
```

- ***set.** This command has the following purposes:
  - *set cond: To set a condition.
  - *set Layer "layer name" *elems|nodes: To set a layer.
  - *set elems: To indicate the elements.
  - *set var: To indicate the variables to use.

It is not necessary to write these commands in lowercase, so *Set will also be valid in all the examples.

***set cond.**: In the case of the conditions, GiD allows the combination of a group of them via the use of *add cond. When a specific condition is about to be used, it must first be defined, and then this definition will be used until another is defined. If this feature is performed inside a loop over intervals, the corresponding entities will be chosen. Otherwise, the entities will be those referred to in the first interval.

It is done in this way because when you indicate to the program that a condition is going to be used, GiD creates a table that lets you know the number of entities over which this condition has been applied. It is necessary to specify whether the condition takes place over the ***nodes**, over the ***elems** or over ***layers** to create the table.

So, a first example to check the nodes where displacement constraints exist could be:

```
*Set Cond Volu-Cstrt *nodes
*Add Cond Surf-Cstrt *nodes
*Add Cond Line-Cstrt *nodes
*Add Cond Poin-Cstrt *nodes
```

These let you apply the conditions directly over any geometric entity.

***Set Layer "layer name" *elems|nodes**

***Add Layer "layer name"**

***Remove Layer "layer name"**

This command sets a group of nodes. In the following loops over nodes/elements with the modifier *OnlyInLayer, the iterations will only take place over the nodes/elements of that group.

Example 1:

```
*set Layer example_layer_1 *elems
*loop elems *OnlyInLayer
     N°:*ElemsNum Name of Layer:*ElemsLayerName N° of Layer :*ElemsLayerNum
*end elems
```

Example 2:

```
*loop layers
*set Layer *LayerName *elems
*loop elems *OnlyInLayer
     N°:*ElemsNum Name of Layer:*ElemsLayerName N° of Layer :*ElemsLayerNum
*end elems
*end layers
```

In this example the command `*LayerName` is used to get the layer name.

There are some modifiers available to point out particular specifications of the conditions.

If the command `*CanRepeat` is added after `*nodes` or `*elems` in `*Set cond`, one entity can appear several times in the entities list. If the command `*NoCanRepeat` is used, entities will appear only once in the list. By default, `*CanRepeat` is off except where one condition has the `*CanRepeat` flag already set.

A typical case where you would not use `*CanRepeat` might be:

```
*Set Cond Line-Constraints *nodes
```

In this case, when two lines share one endpoint, instead of two nodes in the list, only one is written.

A typical situation where you would use `*CanRepeat` might be:

```
*Set Cond Line-Pressure *elems *CanRepeat
```

In this case, if one triangle of a quadrilateral has more than one face in the marked boundary then we want this element to appear several times in the elements list, once for each face.

Other modifiers are used to inform the program that there are nodes or elements that can satisfy a condition more than once (for instance, a node that belongs to a certain number of lines with different prescribed movements) and that have to appear unrepeated in the data input file, or, in the opposite case, that have to appear only if they satisfy more than one condition. These requirements are achieved with the commands `*or(i,type)` and `*and(i,type)`, respectively, after the input of the condition, where i is the number of the condition to be considered and type is the type of the variable (integer or real).

For the previous example there can be nodes or elements in the intersection of two lines or maybe belonging to different entities where the same condition had been applied. To avoid the repetition of these nodes or elements, GiD has the modifier *or, and in the case where two or more different values were applied over a node or element, GiD only would consider one, this value being different from zero. The reason for this can be easily understood by looking at the following example. Considering the previous commands transformed as:

```
*Set Cond Volu-Cstrt *nodes *or(1,int) *or(2,int)
*Add Cond Surf-Cstrt *nodes *or(1,int) *or(2,int)
*Add Cond Line-Cstrt *nodes *or(1,int) *or(2,int)
*Add Cond Poin-Cstrt *nodes *or(1,int) *or(2,int)
```

where *or(1,int) means the assignment of that node to the considered ones satisfying the condition if the integer value of the first condition's field is different from zero, and (*or(2,int) means the same assignment if the integer value of the second condition's field is different from zero). Let us imagine that a zero in the first field implies a restricted movement in the direction of the X-axis and a zero in the second field implies a restricted movement in the direction of the Y-axis. If a point belongs to an entity whose movement in the direction of the X-axis is constrained, but whose movement in the direction of the Y-axis is released and at the same time to an entity whose movement in the direction of the Y-axis is constrained, but whose movement in the direction of the X-axis is released, GiD will join both conditions at that point, appearing as a fixed point in both directions and as a node satisfying the four expressed conditions that would be counted only once.

The same considerations explained for adding conditions through the use of *add cond apply to elements, the only difference being that the command is *add elems. Moreover, it can sometimes be useful to remove sets of elements from the ones assigned to the specific conditions. This can be done with the command *remove elems. So, for instance, GiD allows combinations of the type:

```
*Set Cond Dummy *elems
*Set elems(All)
*Remove elems(Linear)
```

To indicate that all dummy elements apart from the linear ones will be considered, as well as:

```
*Set Cond Dummy *elems
*Set elems(Hexahedra)
*Add elems(Tetrahedra)
*Add elems(Quadrilateral)
*Add elems(Triangle)
```

The format for *set var differs from the syntax for the other two *set commands. Its syntax is as follows:

*Set var varname = expression

where varname is any name and expression is any arithmetical expression, number or command, where the latter must be written without ***** and must be defined as Int or Real.

A Tcl procedure can also be called, but it must return a numerical result.The following are valid examples for these assignments:

```
*Set var ko1=cond(1,real)
*Set var ko2=2
*Set var S1=CondNumEntities
*Set var p1=elemsnum()
*Set var b=operation(p1*2)
*tcl(proc MultiplyByTwo { x } { return [expr {$x*2}] })*\
 *Set var a=tcl(MultiplyByTwo *p1)
```

- ***intformat**, ***realformat**, ***format.** These commands explain how the output of different mathematical expressions will be written to the analysis file. The use of this command consists of a line which begins with the corresponding version, *intformat, *realformat or *format (again, these are not case-sensitive), and continues with the desired writing format, expressed in C-language syntax argument, between double quotes (**"**).

The integer definition of *intformat and the real number definition of *realformat remain unchanged until another definition is provided via *intformat and *realformat, respectively. The argument of these two commands is composed of a unique field. This is the reason why the *intformat and *realformat commands are usually invoked in the initial stages of the .bas file, to set the format configuration of the integer or real numbers to be output during the rest of the process.

The *format command can include several field definitions in its argument, mixing integer and real definitions, but it will only affect the line that follows the command's instance one. Hence, the *format command is typically used when outputting a listing, to set a temporary configuration.

In the paragraphs that follow, there is an explanation of the C format specification, which refers to the field specifications to be included in the arguments of these commands. Keep in mind that the type of argument that the *format command expects may be composed of several fields, and the *intformat and *realformat commands' arguments are composed of an unique field, declared as integer and real, respectively, all inside double quotes:

A format specification, which consists of optional and required fields, has the following form: **%[flags][width][.precision]type**The start of a field is signaled by the percentage symbol (**%**). Each field specification is composed of: some flags, the minimum width, a separator point, the level of precision of the field, and a letter which specifies the type of the data to be represented. The field type is the only one required.

The most common flags are:

- To left align the result

+ To prefix the numerical output with a sign (+ or -)

To force the real output value to contain a decimal point.

The most usual representations are integers and floats. For integers the letters d and i are available, which force the data to be read as signed decimal integers, and u for unsigned decimal integers.

For floating point representation, there are the letters e, f and g, these being followed by a decimal point to separate the minimum width of the number from the figure giving the level of precision.The number of digits after the decimal point depends on the requested level of precision.

Note: The standard width specification never causes a value to be truncated. A special command exists in GiD: *SetFormatForceWidth, which enables this truncation to a prescribed number of digits.

For string representations, the letter s must be used. Characters are printed until the precision value is reached.

The following are valid examples of the use of **format**:

```
*Intformat "%5i"
```

With this sentence, usually located at the start of the file, the output of an integer quantity is forced to be right aligned on the fifth column of the text format on the right side. If the number of digits exceeds five, the representation of the number is not truncated.

```
*Realformat "%10.3e"
```

This sentence, which is also frequently located in the first lines of the template file, sets

the output format for the real numbers as exponential with a minimum of ten digits, and three digits after the decimal point.

```
*format "%10i%10.3e%10i%15.6e"
```

This complex command will specify a multiple assignment of formats to some output columns. These columns are generated with the line command that will follow the format line. The subsequent lines will not use this format, and will follow the general settings of the template file or the general formats: *IntFormat, *RealFormat.

- ***SetFormatForceWidth**, ***SetFormatStandard** The default width specification of a "C/C+" format, never causes a value to be truncated.

*SetFormatForceWidth is a special command that allows a figure to be truncated if the number of characters to print exceeds the specified width.

*SetFormatStandard changes to the default state, with truncation disabled.

For example:

```
*SetFormatForceWidth
*set var num=-31415.16789
*format "%8.3f"
*num
*SetFormatStandard
*format "%8.3f"
*num
```

Output:

```
-31415.1
-31415.168
```

The first number is truncated to 8 digits, but the second number, printed with "C" standard, has 3 numbers after the decimal point, but more than 8 digits.

- ***Tcl**This command allows information to be printed using the Tcl extension language. The argument of this command must be a valid Tcl command or expression which must return the string that will be printed. Typically, the Tcl command is defined in the Tcl file (.tcl , see TCL AND TK EXTENSION -pag. 111- for details).

Example: In this example the ***Tcl** command is used to call a **Tcl** function defined in the problem type **.tcl** file. That function can receive a variable value as its argument with *variable. It is also possible to assign the returned value to a variable, but the Tcl procedure must return a numerical value.

In the .bas file:

```
*set var num=1
*tcl(WriteSurfaceInfo *num)
*set var num2=tcl(MultiplyByTwo *num)
```

In the .tcl file:

```
proc WriteSurfaceInfo { num } {
    return [GiD_Info list_entities surfaces $num]
}
proc MultiplyByTwo { x } {
   return [expr {$x*2}]
}
```

4.2 General description

All the rules that apply to filename.bas files are also valid for other files with the .bas extension. Thus, everything in this section will refer explicitly to the file filename.bas. Any information written to this file, apart from the commands given, is reproduced exactly in the output file (the data input file for the numerical solver). The commands are words that begin with the character *****. (If you want to write an asterisk in the file you should write ****** .) The commands are inserted among the text to be literally translated. Every one of these commands returns one (see Single value return commands -pag. 30-) or multiple (see Multiple values return commands -pag. 34-) values obtained from the preprocessing component. Other commands mimic the traditional structures to do loops or conditionals (see Specific commands -pag. 38-). It is also possible to create variables to manage some data. Comparing it to a classic programming language, the main differences will be the following:

- The text is reproduced literally, without printing instructions, as it is write-oriented.
- There are no indices in the loops. When the program begins a loop, it already knows the number of iterations to perform. Furthermore, the inner variables of the loop change their values automatically. All the commands can be divided into three types:
  - Commands that return one single value. This value can be an integer, a real number or a string. The value depends on certain values that are available to the command and on the position of the command within the loop or after setting some other parameters. These commands can be inserted within the text and write their value where it corresponds. They can also appear inside an expression, which would be the example of the conditionals. For this example, you can specify the type of the variable, integer or real, except when using strcmp or strcasecmp. If these commands are within an expression, no ***** should precede the command.
  - Commands that return more than one value. Their use is similar to that of the previously indicated commands, except for the fact that they cannot be used in other expressions. They can return different values, one after the other, depending on some values of the project.

- Commands that perform loops or conditionals, create new variables, or define some specifications. The latter includes conditions or types of element chosen and also serves to prevent line-feeding. These commands must start at the beginning of the line and nothing will be written into the calculations file. After the command, in the same line, there can be other commands or words to complement the definitions, so, at the end of a loop or conditional, after the command you can write what loop or conditional was finished.

The arguments that appear in a command are written immediately after it and inside parenthesis. If there is more than one, they will be separated by commas. The parentheses might be inserted without any argument inside, which is useful for writing something just after the command without inserting any additonal spaces. The arguments can be real numbers or integers, meaning the word `REAL` or the word `INT` (both in upper- or lowercase) that the value to which it points has to be considered as real or integer, respectively. Other types of arguments are sometimes allowed, like the type of element, described by its name, in the command `*set elem`, or a chain of characters inserted between double quotes **"** for the C-instructions `strcmp` and `strcasecmp`. It is also sometimes possible to write the name of the field instead of its ordering number.

EXAMPLE:

Below is an example of what a .bas file can be. There are two commands (***nelem** and ***npoin**) which return the total number of elements and nodes of a project.

```
%%%% Problem Size %%%%

Number of Elements & Nodes:

*nelem *npoin
```

This .bas file will be converted into a project_name.dat file by GiD. The contents of the project_name.dat file could be something like this:

```
%%%% Problem Size %%%%

Number of Elements & Nodes:

5379 4678
```

(5379 being the number of elements of the project, and 4678 the number of nodes).

4.3 Detailed example - Template file creation

Below is an example of how to create a Template file, step by step.

Note that this is a real file and as such has been written to be compatible with a particular solver program. This means that some or all of the commands used will be non-standard or incompatible with the solver that another user may be using.

The solver for which this example is written treats a line inside the calculation input file as a comment if it is prefixed by a $ sign. In the case of other solvers, another convention

may apply.

Of course, the real aim of this example is familiarize you with the commands GiD uses. What follows is the universal method of accessing GiD's internal database, and then outputting the desired data to the solver.

It is assumed that files with the .bas extension will be created inside the working directory where the problem type file is located. The filename must be problem_type_name.bas for the first file and any other name for the additional .bas files. Each .bas file will be read by GiD and translated to a .dat file.

It is very important to remark that any word in the .bas file having no meaning as a GiD compilation command or not belonging to any command instructions (parameters), will be written verbatim to the output file.

First, we create the header that the solver needs in this particular case.

It consists of the name of the solver application and a brief description of its behaviour.

```
$-----------------------------------------------------
CALSEF: PROGRAM FOR STRUCTURAL ANALYSIS
```

What follows is a commented line with the ECHO ON command. This, when uncommented, is useful if you want to monitor the progress of the calculation. While this particular command may not be compatible with your solver, a similar one may exist.

```
$-----------------------------------------------------
$ ECHO ON
```

The next line specifies the type of calculation and the materials involved in the calculation; this is not a GiD related command either.

```
$-----------------------------------------------------
ESTATICO-LINEAL, EN SOLIDOS
```

As you can see, a commented line with dashes is used to separate the different parts of the file, thus improving the readability of the text.

The next stage involves the initialization of some variables. The solver needs this to start the calculation process.

The following assignments take the first (parameter (1)) and second (parameter (2)) fields in the general problem, as the number of problems and the title of the problem.

The actual position of a field is determined by checking its order in the problem file, so this process requires you to be precise.

Assignment of the first (1) field of the Problem data file, with the command *GenData(1):

```
$-----------------------------------------------------
```

```
$ NUMBER OF PROBLEMS: NPROB = *GenData(1)
$-----------------------------------------------------
```

Assignment of the second (2) field assignment, *GenData(2):

```
$ TITLE OF THE PROBLEM: TITULO= *GenData(2)
$-----------------------------------------------------
```

The next instruction states the field where the starting time is saved. In this case, it is at the 10th position of the general problem data file, but we will use another feature of the *GenData command, the parameter of the command will be the name of the field.

This method is preferable because if the list is shifted due to a field deing added or subtracted, you will not lose the actual position. This command accepts an abbreviation, as long as there is no conflict with any other field name.

```
$-----------------------------------------------------
$ TIME OF START: TIME= *GenData(Starting_time)
$-----------------------------------------------------
```

Here comes the initialization of some general variables relevant to the project in question - the number of points, the number of elements or the number of materials.

The first line is a description of the section.

```
$ DIMENSIONS OF THE PROBLEM:
```

The next line introduces the assignments.

```
DIMENSIONS :
```

This is followed by another line which features the three variables to be assigned. NPNOD gets, from the *npoin function, the number of nodes for the model; NELEM gets, from *nelem, either the total number of elements in the model or the number of elements for every kind of element; and NMATS is initialized with the number of materials:

```
NPNOD= *npoin, NELEM= *nelem, NMATS= *nmats, \
```

In the next line, NNODE gets the maximum number of nodes per element and NDIME gets the variable *ndime. This variable must be a number that specifies whether all the nodes are on the plane whose Z values are equal to 0 (NDIME=2), or if they are not (NDIME=3):

```
NNODE= *nnode, NDIME= *ndime, \
```

The following lines take data from the general data fields in the problem file. NCARG gets the number of charge cases, NGDLN the number of degrees of freedom, NPROP the properties number, and NGAUSS the gauss number; NTIPO is assigned dynamically:

NLOAD= *GenData(Load_Cases), *\

You could use NGDLN= *GenData(Degrees_Freedom), *\, but because the length of the

argument will exceed one line, we have abbreviated its parameter (there is no conflict with other question names in this problem file) to simplify the command.

```
NGDLN= *GenData(Degrees_Fre), *\
NPROP= *GenData(Properties_Nbr), \
NGAUS= *GenData(Gauss_Nbr) , NTIPO= *\
```

Note that the last assignment is ended with the specific command *\ to avoid line feeding. This lets you include a conditional assignment of this variable, depending on the data in the General data problem.

Within the conditional a C format-like strcmp instruction is used. This instruction compares the two strings passed as a parameter, and returns an integer number which expresses the relationship between the two strings. If the result of this operation is equal to 0, the two strings are identical; if it is a positive integer, the first argument is greater than the second, and if it is a negative integer, the first argument is smaller than the second.

The script checks what problem type is declared in the general data file, and then it assigns the coded number for this type to the NTIPO variable:

```
*if(strcmp(GenData(Problem_Type),"Plane-stress")==0)
1 *\
*elseif(strcmp(GenData(Problem_Type),"Plane-strain")==0)
2 *\
*elseif(strcmp(GenData(Problem_Type),"Revol-Solid")==0)
3 *\
*elseif(strcmp(GenData(Problem_Type),"Solid")==0)
4 *\
*elseif(strcmp(GenData(Problem_Type),"Plates")==0)
5 *\
*elseif(strcmp(GenData(Problem_Type),"Revol-Shell")==0)
6 *\
*endif
```

You have to cover all the cases within the if sentences or end the commands with an elseif you do not want unpredictable results, like the next line raised to the place where the parameter will have to be:

```
$ Default Value:
*else
0*\
*endif
```

In our case this last rule has not been followed, though this can sometimes be useful, for example when the problem file has been modified or created by another user and the new specification may differ from the one we expect.

The next assignment is formed by a string compare conditional, to inform the solver about a configuration setting.

First is the output of the variable to be assigned.

```
, IWRIT= *\
```

Then there is a conditional where the string contained in the value of the Result_File field is compred with the string "Yes". If the result is 0, then the two strings are the same, while the output result 1 is used to declare a boolean TRUE.

```
*if(strcmp(GenData(Result_File),"Yes")==0)
1 ,*\
```

Then we compare the same value string with the string "No", to check the complementary option. If we find that the strings match, then we output a 0.

```
*elseif(strcmp(GenData(Result_File),"No")==0)
0 ,*\
*endif
```

The second to last assignment is a simple output of the solver field contents to the INDSO variable:

```
INDSO= *GenData(Solver) , *\
```

The last assignment is a little more complicated. It requires the creation of some internal values, with the aid of the *set cond command.

The first step is to set the conditions so we can access its parameters. This setting may serve for several loops or instructions, as long as the parameters needed for the other blocks of instructions are the same.

This line sets the condition Point-Constraints as an active condition. The *nodes modifier means that the condition will be listed over nodes. The *or(... modifiers are necessary when an entity shares some conditions because it belongs to two or more elements.

As an example, take a node which is part of two lines, and each of these lines has a different condition assigned to it. This node, a common point of the two lines, will have these two conditions in its list of properties. So declaring the *or modifiers, GiD will decide which condition to use, from the list of conditions of the entity.

A first instruction will be as follows, where the parameters of the *or commands are an integer - (1, and (3, in this example - and the specification int, which forces GiD to read the condition whose number position is the integer.

In our case, we find that the first (1) field of the condition file is the X-constraint, and the third (3) is the Y-constraint:

GiD still has no support for substituting the condition's position in the file by its corresponding label, in contrast to case for the fields in the problem data file, for which it is possible.

```
*Set Cond Surface-Constraints *nodes *or(1,int) *or(3,int)
```

Now we want to complete the setting of the loop, with the addition of new conditions.

```
*Add Cond Line-Constraints *nodes *or(1,int) *or(3,int)
*Add Cond Point-Constraints *nodes *or(1,int) *or(3,int)
```

Observe the order in which the conditions have been included: firstly, the surface constraints with the *Set Cond command, since it is the initial sentence; then the pair of *Add Cond sentences, the line constraints; and finally, the point constraints sentence. This logical hierarchy forces the points to be the most important items.

Last of all, we set a variable with the number of entities assigned to this particular condition.

Note that the execution of this instruction is only possible if a condition has been set previously.

```
NPRES= *CondNumEntities
```

To end this section, we put a separator in the output file:

```
$-----------------------------------------------------
```

Thus, after the initialization of these variables, this part of the file ends up as:

```
$ DIMENSIONS OF THE PROBLEM:
DIMENSIONES :
 NPNOD= *npoin, NELEM= *nelem, NMATS= *nmats, \
 NNODE= *nnode, NDIME= *ndime, \
 NCARG= *GenData(Charge_Cases), *\
 NGDLN= *GenData(Degrees_Fre), *\
 NPROP= *GenData(Properties_Nbr), \
 NGAUS= *GenData(Gauss_Nbr) , NTIPO= *\
*if(strcmp(GenData(Problem_Type),"Tens-Plana")==0)
1 *\
*elseif(strcmp(GenData(Problem_Type),"Def-Plana")==0)
2 *\
```

```
*elseif(strcmp(GenData(Problem_Type),"Sol-Revol")==0)
3 *\
*elseif(strcmp(GenData(Problem_Type),"Sol-Tridim")==0)
4 *\
*elseif(strcmp(GenData(Problem_Type),"Placas")==0)
5 *\
*elseif(strcmp(GenData(Problem_Type),"Laminas-Rev")==0)
6 *\
*endif
, IWRIT= *\
*if(strcmp(GenData(Result_File),"Yes")==0)
1 ,\
*elseif(strcmp(GenData(Result_File),"No")==0)
0 ,\
*endif
   INDSO= *GenData(Solver) , *\
*Set Cond Surface-Constraints *nodes *or(1,int) *or(3,int)
*Add Cond Line-Constraints *nodes *or(1,int) *or(3,int)
*Add Cond Point-Constraints *nodes *or(1,int) *or(3,int)
NPRES=*CondNumEntities
$-----------------------------------------------------
```

After creating or reading our model, and once the mesh has been generated and the conditions applied, we can export the file (project_name.dat) and send it to the solver.

The command to create the .dat file can be found on the File -> Export -> Calculation File GiD menu. It is also possible to use the keyboard shortcut Ctrl-x Ctrl-c.

These would be the contents of the project_name.dat file:

```
$-----------------------------------------------------
CALSEF: PROGRAM FOR STRUCTURAL ANALYSIS
$-----------------------------------------------------
$ECHO ON
$-----------------------------------------------------
LINEAR-STATIC, SOLIDS
$-----------------------------------------------------
$NUMBER OF PROBLEMS:
```

```
NPROB = 1
$-----------------------------------------------------
$ PROBLEM TITLE
TITLE= Title_name
$-----------------------------------------------------
$DIMENSIONS OF THE PROBLEM
DIMENSIONS :
  NPNOD= 116 , NELEM= 176 , NMATS= 0 , \
  NNODE= 3 , NDIME= 2 , \
  NCARG= 1 , NGDLN= 1 , NPROP= 5 , \
  NGAUS= 1 , NTIPO= 1 , IWRIT= 1 , \
  INDSO= 10 , NPRES= 0
$-----------------------------------------------------
```

This is where the calculation input begins.

4.3.1 Formatted nodes and coordinates listing

As with the previous section, this block of code begins with a title for the subsection:

```
$ NODAL COORDINATES
```

followed by the header of the output list:

```
$ NODE COORD.-X COORD.-Y COORD.-Z
```

Now GiD will trace all the nodes of the model:

```
*loop nodes
```

For each node in the model, GiD will generate and output its number, using *NodesNum, and its coordinates, using *NodesCoord.

The command executed before the output *format will force the resulting output to follow the guidelines of the specified formatting.

In this example below, the *format command gets a string parameter with a set of codes: %6i specifies that the first word in the list is coded as an integer and is printed six points from the left; the other three codes, all %15.5f, order the printing of a real number, represented in a floating point format, with a distance of 15 spaces between columns (the number will be shifted to have the last digit in the 15th position of the column) and the fractional part of the number will be represented with five digits.

Note that this is a C language format command.

```
*format "%6i%15.5f%15.5f%15.5f"
```

```
*NodesNum *NodesCoord
*end nodes
```

At the end of the section the end marker is added, which in this solver example is as follows:

```
END_GEOMETRY
```

The full set of commands to make this part of the output is shown in the following lines.

```
GEOMETRY
$ ELEMENT CONNECTIVITIES
$ ELEM. MATER. CONNECTIVITIES
*loop elems
  *elemsnum *elemsmat *elemsConec
*end elems
$ NODAL COORDINATES
$ NODE COORD.-X COORD.-Y COORD.-Z
*loop nodes
*format "%6i%15.5f%15.5f%15.5f"
  *NodesNum *NodesCoord
*end
END_GEOMETRY
```

The result of the compilation is output to a file (project_name.dat) to be processed by the solver program.

The first part of the section:

```
$-----------------------------------------------------
GEOMETRY
$ ELEMENT CONNECTIVITIES
$ ELEM. MATER. CONNECTIVITIES
     1 1 73 89 83
     2 1 39 57 52
     3 1 17 27 26
     4 5 1 3 5
     5 5 3 10 8
     6 2 57 73 67
     . . . . .
     . . . . .
```

```
     . . . . .
   176 5 41 38 24
```

And the second part of the section:

```
$ NODAL COORDINATES
$ NODE COORD.-X COORD.-Y COORD.-Z
       1 5.55102 5.51020
       2 5.55102 5.51020
       3 4.60204 5.82993
       4 4.60204 5.82993
       5 4.88435 4.73016
       6 4.88435 4.73016
       . . .
       . . .
       . . .
     116 -5.11565 3.79592
END_GEOMETRY
```

If the solver module you are using needs a list of the nodes that have been assigned a condition, for example, a neighborhood condition, you have to provide it as is explained in the next example.

4.3.2 Elements, materials and connectivities listing

Now we want to output the desired results to the output file. The first line should be a title or a label as this lets the solver know where a loop section begins and ends. The end of this block of instructions will be signalled by the line END_GEOMETRY.

```
GEOMETRY
```

The next two of lines give the user information about what types of commands follow.

Firstly, a title for the first subsection, ELEMENTAL CONNECTIVITIES:

```
$ ELEMENTAL CONNECTIVITIES
```

followed by a header that preceeds the output list:

```
$ ELEM. MATER. CONNECTIVITIES
```

The next part of the code concerns the elements of the model with the inclusion of the *loop instruction, followed in this case by the elems argument.

```
*loop elems
```

For each element in the model, GiD will output: its element number, by the action of the

*elemsnum command, the material assigned to this element, using the *elemsmat command, and the connectivities associated to the element, with the *elemsConec command:

```
*elemsnum *elemsmat *elemsConec
*end elems
```

You can use the swap parameter if you are working with quadratic elements and if the listing mode of the nodes is non-hierarchical (by default, corner nodes are listed first and mid nodes afterwards):

```
*elemsnum *elemsmat *elemsConec(swap)
*end elems
```

4.3.3 Nodes listing declaration

First, we set the necessary conditions, as was done in the previous section.

```
*Set Cond Surface-Constraints *nodes *or(1,int) *or(3,int)
*Add Cond Line-Constraints *nodes *or(1,int) *or(3,int)
*Add Cond Point-Constraints *nodes *or(1,int) *or(3,int)
NPRES=*CondNumEntities
```

After the data initialization and declarations, the solver requires a list of nodes with boundary conditions and the fields that have been assigned.

In this example, all the selected nodes will be output and the 3 conditions will also be printed. The columns will be output with no apparent format.

Once again, the code begins with a subsection header for the solver program and a commentary line for the user:

```
BOUNDARY CONDITIONS
$ RESTRICTED NODES
```

Then comes the first line of the output list, the header:

```
$ NODE CODE PRESCRIPTED VALUES
```

The next part the loop instruction, in this case over nodes, and with the specification argument *OnlyInCond, to iterate only over the entities that have the condition assigned. This is the condition that has been set on the previous lines.

```
*loop nodes *OnlyInCond
```

The next line is the format command, followed by the lines with the commands to fill the fields of the list.

```
*format "%5i%1i%1i%f%f"
*NodesNum *cond(1,int) *cond(3,int) *\
```

The *format command influence also includes the following two if sentences. If the degrees of freedom field contains an integer equal or greater than 3, the number of properties will be output.

```
*if(GenData(Degrees_Freedom_Nodes,int)>=3)
*cond(5,int) *\
*endif
```

And if the value of the same field is equal to 5 the output will be a pair of zeros.

```
*if(GenData(Degrees_Free,int)==5)
0 0 *\
*endif
```

The next line ouputs the values contained in the second and fourth fields, both real numbers.

```
*cond(2,real) *cond(4,real) *\
```

In a similar manner to the previous if sentences, here are some lines of code which will output the sixth condition field value if the number of degrees of freedom is equal or greater than three, and will output a pair of zeros if it is equal to five.

```
*if(GenData(Degrees_Free,int)>=3)
*cond(6,real) *\
*endif
*if(GenData(Degrees_Free,int)==5)
0.0 0.0 *\
*endif
```

Finally, to end the section, the *end command closes the previous *loop. The last line is the label of the end of the section.

```
*end
END_BOUNDARY CONDITIONS
$-----------------------------------------------------
```

The full set of commands included in this section are as follows:

```
BOUNDARY CONDITIONS
$ RESTRICTED NODES
$ NODE CODE PRESCRIPTED VALUES
*loop nodes *OnlyInCond
*format "%5i%1i%1i%f%f"
   *NodesNum *cond(1,int) *cond(3,int) *\
```

```
*if(GenData(Degrees_Free,int)>=3)
*cond(5,int) *\
*endif
*if(GenData(Degrees_Free,int)==5)
0 0 *\
*endif
*cond(2,real) *cond(4,real) *\
*if(GenData(Degrees_Free,int)>=3)
*cond(6,real) *\
*endif
*if(GenData(Degrees_Free,int)==5)
0.0 0.0 *\
*endif
*end
END_BOUNDARY CONDITIONS
$-------------------------------------------------
```

4.3.4 Elements listing declaration

First, we set the loop to the interval of the data.

*loop intervals

The next couple of lines indicate the starting of one section and the title of the example, taken from the first field in the interval data with an abbreviation on the label. They are followed by a comment explaining the type of data we are using.

LOADS

TITLE: *IntvData(Charge_case)

$ LOAD TYPE

We begin by setting the condition as before. If one condition is assigned twice or more to the same element without including the *CanRepeat parameter in the *Set Cond, the condition will appear once; if the *CanRepeat parameter is present then the number of conditions that will appear is the number of times it was assigned to the condition.

```
*Set Cond Face-Load *elems *CanRepeat
```

Then, a condition checks if any element exists in the condition.

```
*if(CondNumEntities(int)>0)
```

Next is a title for the next section, followed by a comment for the user.

```
DISTRIBUTED ON FACES
$ LOADS DISTRIBUTED ON ELEMENT FACES
```

We assign the number of nodes to a variable.

```
$ NUMBER OF NODES BY FACE NODGE = 2
$ LOADED FACES AND FORCE VALUES
*loop elems *OnlyInCond
    ELEMENT=*elemsnum(), CONNECTIV *globalnodes
    *cond(1) *cond(1) *cond(2) *cond(2)
*end elems
END_DISTRIBUTED ON FACES
*endif
```

The final section deals with outputting a list of the nodes and their conditions.

4.3.5 Materials listing declaration

This section deals with outputting a materials listing.

As before, the first lines must be the title of the section and a commentary:

```
MATERIAL PROPERTIES
$ MATERIAL PROPERTIES FOR MULTILAMINATE
```

Next there is the loop sentence, this time concerning materials:

```
*loop materials
```

Then comes the line where the number of the material and its different properties are output:

```
*matnum() *MatProp(1) *MatProp(2) *MatProp(3) *MatProp(4)
```

Finally, the end of the section is signalled:

```
*end materials
END_MATERIAL PROPERTIES
$-----------------------------------------------------
```

The full set of commands is as follows:

```
MATERIAL PROPERTIES
$ MATERIAL PROPERTIES FOR MULTILAMINATE
*loop materials
   *matnum() *MatProp(1) *MatProp(2) *MatProp(3) *MatProp(4)
*end materials
```

```
END_MATERIAL PROPERTIES
$-------------------------------------------------
```

The next section deals wth generating an elements listing.

4.3.6 Nodes and its conditions listing declaration

As for previous sections, the first thing to do set the conditions.

```
*Set Cond Point-Load *nodes
```

As in the previous section, the next loop will only be executed if there is a condition in the selection.

```
*if(CondNumEntities(int)>0)
```

Here begins the loop over the nodes.

```
PUNCTUAL ON NODES
*loop nodes *OnlyInCond
    *NodesNum *cond(1) *cond(2) *\
```

The next *if sentences determine the output writing of the end of the line.

```
*if(GenData(Degrees_Free,int)>=3)
*cond(3) *\
*endif
*if(GenData(Degrees_Free,int)==5)
0 0 *\
*endif
*end nodes
```

To end the section, once again you have to include the end label and the closing *endif.

```
END_PUNCTUAL ON NODES
*endif
```

Finally, a message is written if the value of the second field in the interval data section inside the problem file is equal to "si" (yes).

```
*if(strcasecmp(IntvData(2),"Si")==0)
SELF_WEIGHT
*endif
```

To signal the end of this part of the forces section, the following line is entered.

```
END_LOADS
```

Before the end of the section it remains to tell the solver what the postprocess file will be. This information is gathered from the *IntvData command. The argument that this command receives (3) specifies that the name of the file is in the third field of the loop iteration of the interval.

```
$------------------------------------------------
$POSTPROCESS FILE FEMV = *IntvData(3)
```

To end the forces interval loop the *end command is entered.

```
$------------------------------------------------
*end nodes
```

Finally, the complete file is ended with the sentence required by the solver.

```
END_CALSEF $------------------------------------------------
```

The preceding section is compiled completely into the following lines:

```
*Set Cond Point-Load *nodes
*if(CondNumEntities(int)>0)
PUNCTUAL ON NODES
*loop nodes *OnlyInCond
      *NodesNum *cond(1) *cond(2) *\
*if(GenData(Degrees_Free,int)>=3)
*cond(3) *\
*endif
*if(GenData(Degrees_Free,int)==5)
0 0 *\
*endif
*end
END_PUNCTUAL ON NODES
*endif
*if(strcasecmp(IntvData(2),"Si")==0)
SELF_WEIGHT
*endif
END_LOADS
$------------------------------------------------
$POSTPROCESS FILE
FEMV = *IntvData(3)
$------------------------------------------------
```

```
*end nodes

END_CALSEF

$--------------------------------------------------
```

This is the end of the template file example.

5 EXECUTING AN EXTERNAL PROGRAM

Once all the problem type files are finished (.cnd, .mat, .prb, .sim, .bas files), you can run the solver. You may wish to run it directly from inside GiD.

To do so, it is necessary to create the file `problem_type_name.bat` in the Problem Type directory. This must be a shell script that can contain any type of information and that will be different for every operating system. When you select the Calculate option in GiD Preprocess this shell script is executed (see CALCULATE from Reference Manual).

Because the .bat file will be different depending on the operating system, it is possible to create two files: one for Windows and another for Unix/Linux. The Windows file has to be called: `problem_type_name.win.bat`; the Unix/Linux file has to be called: `problem_type_name.unix.bat`.

If **GiD** finds a `.win.bat` or `.unix.bat` file, the file `problem_type_name.bat` will be ignored.

If a `.bat` file exists in the problem type directory when choosing Start in the calculations window, GiD will automatically write the analysis file inside the example directory assigning the name `project_name.dat` to this file (if there are more files, the names `project_name-1.dat` ... are used). Next, this shell script will be executed. GiD will assign three arguments to this script:

- **1st argument:** `project_name` (name of the current project);
- **2nd argument:** `c:\a\b\c\project_name.gid` (path of the current project);
- **3rd argument:** `c:\a\b\c\problem_type_name.gid` (path of the problem type selected);

Among other utilities, this script can move or rename files and execute the process until it finishes.

Note 1: This file must have the executable flag set (see the UNIX command chmod +x) in UNIX systems.

Note 2: GiD sets as the current directory the model directory (example: `c:\examples\test1.gid)` just before executing the .bat file. Therefore, the lines (`cd $directory`) are not necessary in the scripts.

Note 3: In UNIX platforms check you have installed the shell you are using in the .unix.bat script, there are more than one possibilities: bash, csh, tcsh, ...

The first line of the script specify the shell to be used, for example

```
#!/bin/bash
```

or

```
#!/bin/csh
```

In Windows platforms, the command.exe provided by GiD is used instead the standard cmd.exe or command.com

5.1 Showing feedback when running the solver

The information about what is displayed when Output view: is pressed is also given here. To determine what will be shown, the script must include a comment line in the following form:

For Windows :

```
rem OutputFile: %1.log
```

For Linux/Unix:

```
# OutputFile: "$1.log"
```

where "$1.log" means to display in that window a file whose name is: project_name.log. The name can also be an absolute name like output.dat. If this line is omitted, when you press Output view:, nothing will be displayed.

5.2 Commands accepted by the GiD command.exe

The keywords are as follows:

- %
- Shift
- Rem
- Chdir (Cd)
- Del (Delete, Erase)
- Copy
- Rename (Ren, Move)
- Mkdir (Md)
- Set
- Echo (@echo)
- If
- Call
- Goto
- :
- Type

Unknown instructions will be executed as from an external file.

Not all the possible parameters and modifiers available in the operating system are implemented in the GiD executable command.exe.

Note: At the moment, **command.exe** is only used in Windows operating systems as an alternative to command.com or cmd.exe. With the **GiD command.exe** some of the disadvantages of Windows can be avoided (the limited length of parameters, temporary use of letters of virtual units that sometimes cannot be eliminated, fleeting appearance of the console window, etc).

If GiD finds the file **command.exe** located next to **gid.exe** , it will be used to interpret the *.bat file of the problem type; if the file command.exe cannot be found, the *.bat file

will be interpreted by the windows command.com.

If conflicts appear by the use of some instruction still not implemented in the **GiD command.exe** , it is possible to rename the command.exe file, so that GiD does not find it, and the operating system command.com is used.

%

Returns the value of a variable.

%number

%name%

Parameters

number

The number is the position (from 0 to 9) of one of the parameters which the *.bat file receives.

name

The name of an environment variable. That variable has to be declared with the instruction "set".

Note: GiD sends three parameters to the *.bat file: %1, %2, %3

%1 is the name of the current project (project_name)

%2 is the path of the current project (c:\a\b\c\project_name.gid)

%3 is path of the problem type (c:\a\b\c\problem_type_name.gid)

For example, if GiD is installed in `c:\gidwin`, the "problemtype" name is `cmas2d.gid` and the project is test.gid, located in `c:\temp` (the project is a directory called `c:\temp\test.gid` with some files inside), parameters will have the following values:

```
%1 test
%2 c:\temp\test.gid
%3 c:\gidwin\problemtypes\cmas2d.gid
```

Note: It is possible that the file and directory names of these parameters are in the short mode Windows format. So, parameter %3 would be: `c:\GIDWIN\PROBLE~\CMAS2D.GID`.

Examples

```
echo %1 > %2\%1.txt
echo %TEMP% >> %1.txt
```

Shift

The shift command changes the values of parameters %0 to %9 copying each parameter in the previous one. That is to say, value %1 is copied to %0, value %2 is copied to %1, etc.

Shift

Parameter

None.

Note: The shift command can be used to create a batch program that accepts more than 10 parameters. If it specifies more than 10 parameters in the command line, those that appear after tenth (%9) will move to parameter %9 one by one.

Rem

Rem is used to include comments in a *.bat file or in a configuration file.

rem*[comment]*

Parameter

comment

Any character string.

Note: Some comments are GiD commands.

Chdir (Cd)

Changes to a different directory.

chdir*[drive:path] [..]*

-or-

cd*[drive:path] [..]*

Parameters

[drive:path]

Disk and path of the new directory.

[..]

Goes back one directory. For example if you are within the C:\WINDOWS\COMMAND> directory this would take you to C:\WINDOWS>.

Note: When GiD calls the *.bat file, the path of the project is the current path, so it is not necessary to use cd %2 at the beginning of the *.bat file.

Examples

```
chdir e:\tmp cd ..
```

Delete (Del, Erase) Command used to delete files permanently from the computer.

delete*[drive:][path] fileName [fileName]*

Parameters

[drive:][path] fileName [fileName] Parameters that specify the location and the name of the file that has to be erased from disk. Several file names can be given.

Note: Files will be eliminated although they have the hidden or read only flag. Use of

wildcards is not allowed. For example del *.* is not valid. File names must be separated by spaces and if the path contains blank spaces, the path should be inside inverted commas (the short path without spaces can also be used).

Examples

```
delete %2\%1\file.cal

del C:\tmp\fa.dat C:\tmp\fb.dat

del "C:\Program files\test 4.txt"
```

Copy

Copies one or more files to another location.

copy *source [+ source [+ ...]] destination*

Parameters

source Specifies the file or files to be copied.

destination Specifies the filename for the new file(s).

To append files, specify a single file for destination, but multiple files for source (using the file1 + file2 + file3 format).

Note: If the destination file already exists, it will be overwritten without prompting whether or not you wish to overwrite it.

File names must be separated by spaces. If the destination only contains the path but not the filename, the new name will be the same as the source filename.

Examples

```
copy f1.txt f2.txt

copy f1.txt c:\tmp

rem if directory c:\tmp exists, c:\tmp\f1.txt will be created, if it does
not exist, file c:\tmp will be created.

copy a.txt + b.txt + c.txt abc.txt
```

Rename (Ren, Move)

Used to rename files and directories from the original name to a new name.

rename*[drive:][path] fileName1 fileName2*

Parameter [drive:][path] fileName1 Specifies the path and the name of the file which is to be renamed.

fileName2 Specifies the new name file.

Note: If the destination file already exists, it will be overwritten without prompting whether or not you wish to overwrite it. Wildcards are not accepted (*,?), so only one file can be renamed every time. Note that you cannot specify a new drive for your destination. A directory can be renamed in the same way as if it was a file.

Examples

```
Rename fa.txt fa.dat

Rename "c:\Program Files\fa.txt" c:\tmp\fa.txt

Rename c:\test.gid c:\test2.gid
```

Mkdir (md)

Allows you to create your own directories.

mkdir[drive:]pathmd [drive:]path

Parameter

drive: Specifies the drive where the new directory has to be created.

path Specifies the name and location of the new directory. The maximum length of the path is limited by the file system.

Note: mkdir can be used to create a new path with many new directories.

Examples

```
mkdir e:\tmp2

mkdir d1\d2\d3
```

Set

Displays, sets, or removes Windows environment variables.

set *variable=[string]*

Parameters

variable Specifies the environment-variable name.

string Specifies a series of characters to assign to the variable.

Note: The set command creates variables which can be used in the same way as the variables %0 to %9. Variables %0 to %9 can be assigned to new variables using the set command.

To get the value of a variable, the variable has to be written inside two % symbols. For example, if the environment-variable name is V1, its value is %V1%. Variable names are not case-sensitive.

Examples

```
set basename = %1

set v1 = my text
```

Echo (@echo)

Displays messages.

echo *[message]*

Parameters

message Specifies the text that will be displayed in the screen.

Note: The message will not be visible because the console is not visible, since GiD hides it.Therefore, this command is only useful if the output is redirected to a file (using > or >>). The symbol > sends the text to a new file, and the symbol >> sends the text to a file if it already exists. The if and echo commands can be used in the same command line.

Examples

```
Echo %1 > out.txt

Echo %path% >> out.txt

If Not Exist %2\%1.flavia.res Echo "Program failed" >> %2\%1.err
```

If

Executes a conditional sentence. If the specified condition is true, the command which follows the condition will be executed; if the condition is false, the next line is ignored.

- **if***[not] exist fileName command*
- **if** *[not] string1==string2 command*
- **if***[not] errorlevel number command*

Parameters

not Specifies that the command has to be executed only if the condition is false.

exist file Returns true if file exists.

command Is the command that has to be executed if the condition returns true.

string1==string2 Returns true if string1 and string2 are equal. It is possible to compare two strings, or variables (for example, %1).

errorlevel number Returns true if the last program executed returned a code equal to or bigger than the number specified.

Note: Exist can also be used to check if a directory exists.

Examples

```
If exist sphere.igs echo File exists >> out.txt

if not exist test.gid echo Dir does not exist >> out.txt

if %1 == test echo Equal %1 >> out.txt
```

Call

Executes a new program.

call*[drive:][path] file [parameters]*

Parameter

[drive:][path] file Specifies the location and the name of the program that has to be

executed.

parameters Parameters required by the program executed.

Note: The program can be *.bat file, a *.exe file or a *.com file. If the program does a recursive call, some condition has to be imposed to avoid an endless curl.

Examples

```
call test.bat %1
```

```
call gid.exe -n -PostResultsToBinary %1.flavia.res %1.flavia.bin
```

Goto

The execution jumps to a line identified by a label.

goto *label*

Parameter

label It specifies a line of the *.bat file where the execution will continue. That label must exist when the Goto command is executed. A label is a line of the *.bat file and starts with (:). Goto is often used with the command if, in order to execute conditional operations. The Goto command can be used with the label :EOF to make the execution jump to the end of the *.bat file and finish.

Note: The label can have more than eight characters and there cannot be spaces between them. The label name is not case-sensitive.

Example

```
goto :EOF
```

```
if exist %1.err goto end
```

```
...
```

```
:end
```

:

Declares a label.

:*labelName*

Parameter

labelName A string which identifies a line of the file, so that the Goto command can jump there. The label line will not be executed.

Note: The label can have more than eight characters and there cannot be spaces between them. The label name is not case-sensitive.

Examples

```
:my_label
```

```
:end
```

Type

Displays the contents of text files.

type*[drive:][path] fileName*

Parameters

[drive:][path] fileName Specifies the location and the name of the file to be displayed. If the file name contains blank spaces it should be inside inverted commas ("file name").

Note: The text will not be visible because the console is not visible, since GiD hides it Therefore, this command is only useful if the output is redirected to a file (using > or >>). The symbol > sends the text to a new file, and the symbol >> sends the text to a file if it already exists. It is recommended to use the copy command instead of type.

In general, the **type** command should not be used with binary files.

Examples

```
type %2\%1.dat > %2\%1.txt
```

5.3 Managing errors

A line of code like

For Windows

```
rem ErrorFile: %1.err
```

For Linux/UNIX

```
# ErrorFile: "$1.err"
```

included in the .bat file means that the given filename is the error file. At the end of the execution of the .bat file, if the errorfile does not exist or is zero, execution is considered to be successful. If not, an error window appears and the contents of the error file are considered to be the error message. If this line exists, GiD will delete this file just before calculating to avoid errors with previous calculations.

A comment line like

```
rem WarningFile: %1.wrn
```

or

```
# WarningFile: "$1.wrn"
```

included in the .bat file means that the given filename is the warning file. This file stores the warnings that may appear during the execution of the .bat file.

5.4 Examples

Here are two examples of easy scripts to do. One of them is for Unix/Linux machines and the other one is for MS-Dos or Windows.

- **UNIX/Linux example:**

```
#!/bin/bash
basename = $1
directory = $2
ProblemDirectory = $3
# OutputFile: "$1.log" IT IS USED BY GiD
# ErrorFile: "$1.err" IT IS USED BY GiD
rm -f "$basename.flavia.res"
"$ProblemDirectory/myprogram" "$basename"
mv "$basename.results" "$basename.post.res"
```

- **MS-DOS/Windows example:**

```
rem basename=%1 JUST INFORMATIVE
rem directory=%2 JUST INFORMATIVE
rem ProblemDirectory=%3 JUST INFORMATIVE
rem OutputFile: %1.log IT IS USED BY GiD
rem ErrorFile: %1.err IT IS USED BY GiD
del %1.flavia.res%3\myprogram %1
move %1.post %1.flavia.res
```

6 POSTPROCESS DATA FILES

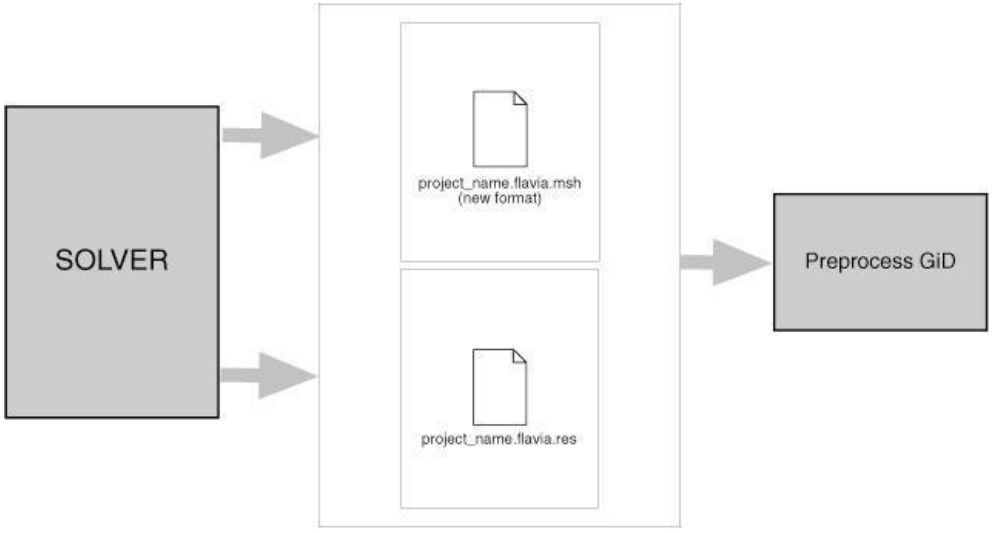
In GiD Postprocess you can study the results obtained from a solver program. The solver and GiD Postprocess communicate through the transfer of files. The solver program has to write the results to a file that must have the extension .post.res, or the old .flavia.res, and its name must be the project name.

The solver program can also send the postprocess mesh to GiD (though this is not mandatory), where it should have the extension .post.msh, or the old version .flavia.msh. If this mesh is not provided by the solver program, GiD uses the preprocess mesh in Postprocess.

The extensions .msh and .res are also allowed, but only files with the extensions .post.res or .flavia.res - and potentially .post.msh or .flavia.msh - will automatically be read by GiD when postprocessing the GiD project.

Postprocessing data files are ASCII files, and can be separated into two categories:

- Mesh Data File: `project_name.post.msh` (or `project_name.flavia.msh`) for volume and surface (3D or 2D) mesh information, and
- Results Data File: `project_name.post.res` (or `project_name.flavia.res`) for results information.



Note: ProjectName.post.msh, or the old ProjectName.flavia.msh, handles meshes of different element types: points, lines, triangles, quadrilaterals, tetrahedra and hexahedra. The old format, which only handles one type of element per file, is still supported inside GiD (see Old postprocess mesh format).

If a project is loaded into GiD, when changing to GiD Postprocess it will look for ProjectName.post.res, or the old ProjectName.flavia.res. If a mesh information file with the name ProjectName.post.msh, or the old ProjectName.flavia.msh is present, it will also be read, regardless of the information available from GiD Preprocess.

- **ProjectName.post.msh** (or the old **ProjectName.flavia.msh)**: This file should

contain nodal coordinates of the mesh and its nodal connectivities as well as the material of each element. At the moment, only one set of nodal coordinates can be entered. Different kinds of elements can be used but separated into different sets. If no material is supplied, GiD takes the material number to be equal to zero.

- **ProjectName.post.res** (or the old **ProjectName.flavia.res**): This second file must contain the nodal or gaussian variables. GiD lets you define as many nodal variables as desired, as well as several steps and analysis cases (limited only by the memory of the machine). The definitions of the Gauss points and the results defined on these points should also be written in this file.

The files are created and read in the order that corresponds to the natural way of solving a finite element problem: mesh, surface definition and conditions and finally, evaluation of the results. The format of the read statements is normally free, i.e. it is necessary only to separate them by spaces.

Thus, files can be modified with any format, leaving spaces between each field, and the results can also be written with as many decimal places as desired. Should there be an error, the program warns the user about the type of mistake found.

GiD reads all the information directly from the preprocessing files whenever possible in order to gain efficiency.

6.1 Postprocess results format: ProjectName.post.res

Note: The new postprocess results format requires GiD version 6.1.4b or higher.

Note: Code developers can download the GiDpost tool from the GiD web page (http://www.gidhome.com/gid-plus/tools/gidpost/); this is a C/C++/Fortran library for creating postprocess files for GiD in both ASCII and compressed binary format.

This is the ASCII format description:

The first line of the files with results written in this new postprocess format should be:

```
GiD Post Results File 1.0
```

Comment lines are allowed and should begin with a '#'. Blank lines are also allowed.

Results files can also be included with the keyword **include**, for instance:

```
include "My Other Results File"
```

This is useful, for instance, for sharing several GaussPoints definitions and ResultRangeTable among different analyses.

This 'include' should be outside the **Blocks** of information.

There are several types of **Blocks** of information, all of them identified by a keyword:

- GaussPoints: Information about gauss points: name, number of gauss points, natural coordinates, etc.;

- ResultRangesTable: Information for the result visualization type **Contour Ranges**: name, range limits and range names;
- Result: Information about a result: name, analysis, analysis/time step, type of result, location, values;
- ResultGroup: several results grouped in one block. These results share the same analysis, time step, and location (nodes or gauss points).

6.1.1 Gauss Points

If Gauss points are to be included, they must be defined before the Result which uses them. Each Gauss points block is defined between the lines GaussPoints and End GaussPoints.

The structure is as follows, and should:

- Begin with a header that follows this model:

GaussPoints "gauss_points_name" **Elemtype** my_type "mesh_name"

where

  - GaussPoints, elemtype: are not case-sensitive;
  - `"gauss_points_name"`: is a name for the gauss points set, which will be used as reference by the results that are located on these gauss points;
  - `my_type`: decribes which element type these gauss points are for, i.e. `Point, Linear, Triangle, Quadrilateral, Tetrahedra or Hexahedra` ;
  - `"mesh_name"`: is an optional field. If this field is missing, the gauss points are defined for all the elements of type my_type. If a mesh name is given, the gauss points are only defined for this mesh.

- Be followed by the gauss points properties:

Number of Gauss Points: number_gauss_points_per_element

Nodes included

Nodes not included

Natural Coordinates: Internal

Natural Coordinates: Given

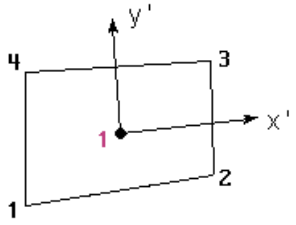
natural_coordinates_for_gauss_point_1 . . . natural_coordinates_for_gauss_point_n

where

- Number of Gauss Points: number_gauss_points_per_element: is not case-sensitive and is followed by the number of gauss points per element that defines this set. If Natural Coordinates: is set to Internal, number_gauss_points_per_element should be one of:
  - 1, 3, 6 for Triangles;
  - 1, 4, 9 for Quadrilaterals;
  - 1, 4, 10 for Tetrahedra;

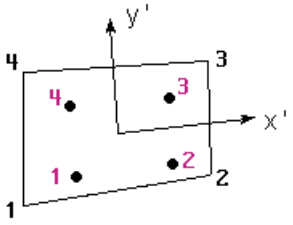
- 1, 8, 27 for Hexahedra;
- 1, 6 for Prisms;
- 1, 5 for Pyramids; and
- 1, ... n points equally spaced over lines.

For triangles and quadrilaterals the order of the gauss points with **Internal** natural coordinates will be this:



Internal coordinates:

(0, 0)

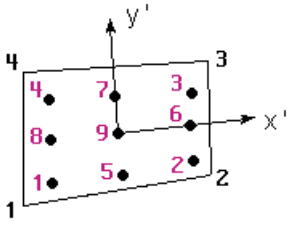


Internal coordinates:

a=0.57735027

(-a,-a) (a,-a)

(a, a) (-a, a)



Internal coordinates:

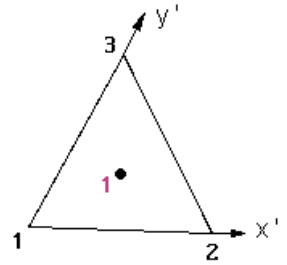
a=0.77459667

(-a,-a) (a,-a) (a, a)

(-a, a) (0,-a) (a, 0)

(0, a) (-a, 0) (0, 0)

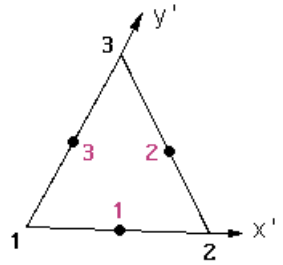
Gauss Points positions of the quadrature of Gauss-Legendre Quadrilaterals



Internal coordinates:

a=1/3

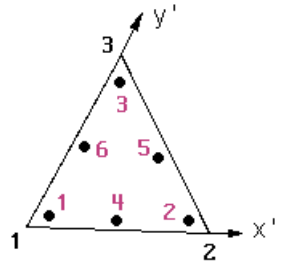
(a, a)



Internal coordinates:

a=1/2

(a, 0) (a, a) (0, a)



Internal coordinates:

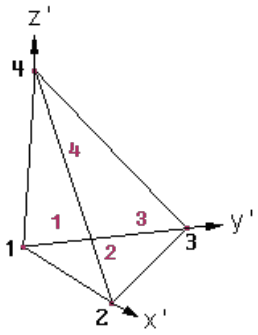
a=0.09157621
b=0.81684757
c=0.44594849
d=0.10810301

(a, a) (b, a) (a, b)

(c, d) (c, c) (d, c)

Gauss Points positions of the quadrature of Gauss for Triangles

For tetrahedra the order of the **Internal** Gauss Points is this:

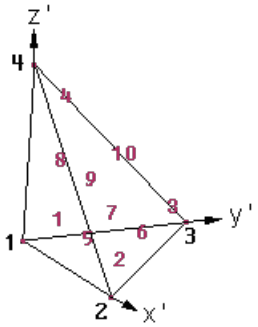


Internal coordinates:

a=(5+3*sqrt(5))/20=0.585410196624968

b=(5-sqrt(5))/20 =0.138196601125010

(b, b, b) (a, b, b) (b, a, b) (b, b, a)



Internal coordinates:

a=0.108103018168070

b=0.445948490915965

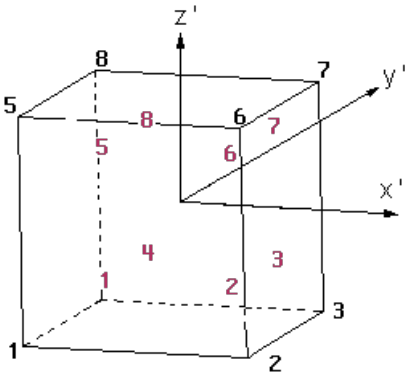
c=0.816847572980459

(a, a, a) (c, a, a) (a, c, a) (a, a, c)

(b, a, a) (b, b, a) (a, b, a)

(a, a, b) (b, a, b) (a, b, b)

For hexahedra the order of the **Internal** Gauss Points is this:

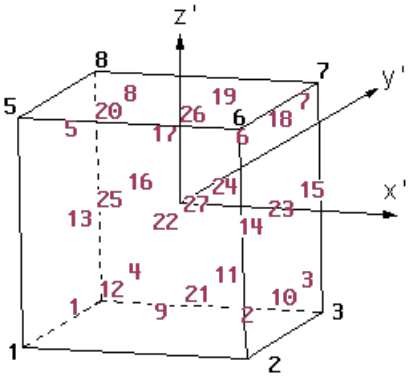


Internal coordinates:

a=0.577350269189626

(-a,-a,-a) (a,-a,-a) (a, a,-a) (-a, a,-a)

(-a,-a, a) (a,-a, a) (a, a, a) (-a, a, a)



Internal coordinates:

a = 0.774596669241483

(-a,-a,-a) (a,-a,-a) (a, a,-a) (-a, a,-a)

(-a,-a, a) (a,-a, a) (a, a, a) (-a, a, a)

(0,-a,-a) (a, 0,-a) (0, a,-a) (-a, 0,-a)

(-a,-a, 0) (a,-a, 0) (a, a, 0) (-a, a, 0)

(0,-a, a) (a, 0, a) (0, a, a) (-a, 0, a)

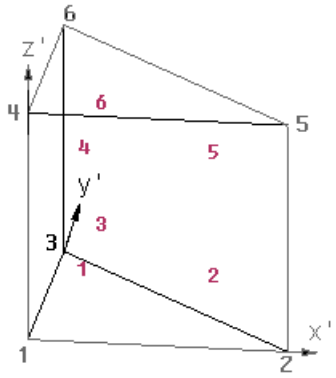
(0, 0,-a)

(0,-a, 0) (a, 0, 0) (0, a, 0) (-a, 0, 0)

(0, 0, a)

(0, 0, 0)

For prisms the order of the **Internal** Gauss Points is this:



Internal coordinates:

a=1/6=0.166666666666666

b=4/6=0.666666666666666

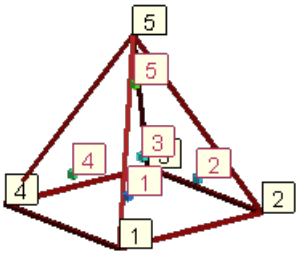
c=1/2-1/(2sqrt(3)) =0.211324865405187

d=1/2+1/(2sqrt(3))=0.788675134594812

(a, a, c) (b, a, c) (a, b, c)

(a, a, d) (b, a, d) (a, b, d)

For pyramids the order of the **Internal** Gauss Points will be this:



Internal coordinates:

a=8.0*sqrt(2.0/15.0)/ 5.0 =0.584237394672177

b=-2/3 =-0.666666666666666

c=2/5 = 0.4

(-a, -a, b)

(a, -a, b)

(a, a, b)

(-a, a, b)

(0.0, 0.0, c)

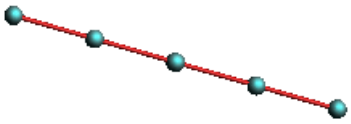
The **given** natural coordinates for Gauss Points should range:

- between **0.0** and **1.0** for Triangles, Tetrahedra and Prisms, and

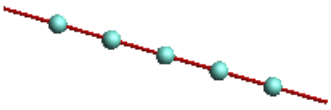
- between **-1.0** and **1.0** for Quadrilaterals, Hexahedra and Pyramids.

Note: If the natural coordinates used are the internal ones, almost all the Results visualization possibilities will have some limitations for tetrahedra and hexahedra with more than one gauss points. If the natural coordinates are given, these limitations are extended to those elements with number_gauss_points_per_element not included in the list written above.

- Nodes Included / Nodes not Included: are not case-sensitive, and are only necessary for gauss points on Linear elements which indicate whether or not the end nodes of the Linear element are included in the number_gauss_points_per_element count.



Nodes included



Nodes not included

The default value is nodes not included

Note: By now, Natural Coordinates for linear elements cannot be "Given"

- Natural Coordinates: Internal / Natural Coordinates: Given: are not case-sensitive, and indicate whether the natural coordinates are calculated internally by GiD, or are given in the following lines. The natural coordinates should be written for each line and gauss point.

- End with this tail:

End GaussPoints

where End GaussPoints: is not case-sensitive.

Here is an example of results on Gauss Points:

```
GaussPoints "Board gauss internal" ElemType Triangle "board"
  Number Of Gauss Points: 3
  Natural Coordinates: internal
end gausspoints
```

Internal Gauss points

The following Internal gauss points are automatically defined.

Results can use this names without explicitly define them with a GaussPoints / End GaussPoints statement.

GP_POINT_1

GP_LINEAR_1

GP_TRIANGLE_1 GP_TRIANGLE_3 GP_TRIANGLE_6

GP_QUADRILATERAL_1 GP_QUADRILATERAL_4 GP_QUADRILATERAL_9

GP_TETRAHEDRA_1 GP_TETRAHEDRA_4 GP_TETRAHEDRA_10

GP_HEXAHEDRA_1 GP_HEXAHEDRA_8 GP_HEXAHEDRA_27

GP_PRISM_1 GP_PRISM_6

GP_PIRAMID_1 GP_PIRAMID_5

GP_SPHERE_1

GP_CIRCLE_1

6.1.2 Result Range Table

If a Result Range Table is to be included, it must be defined before the Result which uses it.

Each Result Range Table is defined between the lines ResultRangesTable and End ResultRangesTable.

The structure is as follows and should:

- Begin with a header that follows this model:

ResultRangesTable "ResultsRangeTableName"

where ResultRangesTable: is not case-sensitive; "ResultsRangeTableName": is a name for the Result Ranges Table, which will be used as a reference by the results that use this Result Ranges Table.

- Be followed by a list of Ranges, each of them defined as follows:

Min_Value **-** Max_Value**:** "Range Name"

where

  - Min_value : is the minimum value of the range, and may be void if the Max_value is given. If void, the minimum value of the result will be used;
  - Max_value : is the maximum value of the range, and may be void if the Min_value is given. If void, the maximum value of the result will be used;
  - "Range Name" : is the name of the range which will appear on legends and labels.

- End with this tail:

End ResultRangesTable

where

End ResultRangesTable: is not case-sensitive.

Here are several examples of results range tables:

- Ranges defined for the whole result:

```
ResultRangesTable "My table"
# all the ranges are min <= res < max except
```

```
# the last range is min <= res <= max
      - 0.3: "Less"
  0.3 - 0.7: "Normal"
  0.7 - : "Too much"
End ResultRangesTable
```

- Just a couple of ranges:

```
ResultRangesTable "My table"
  0.3 - 0.7: "Normal"
  0.7 - 0.9: "Too much"
End ResultRangesTable
```

- Or using the maximum of the result:

```
ResultRangesTable "My table"
  0.3 - 0.7: "Normal"
  0.7 - : "Too much"
End ResultRangesTable
```

6.1.3 Result

Each Result block is identified by a Result header, followed by several optional properties: component names, ranges table, and the result values, defined by the lines Values and End Values.

The structure is as follows and should:

- Begin with a header that follows this model:

Result "result name" "analysis name" step_value my_result_type my_location "location name"

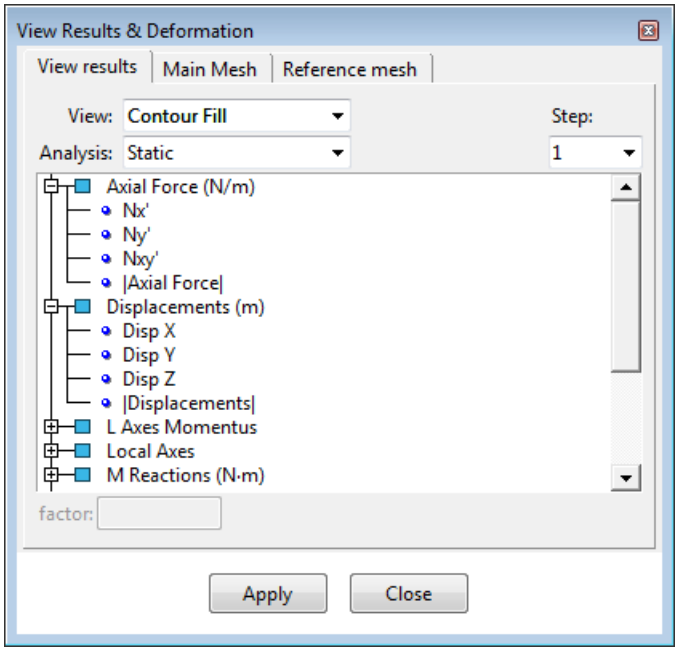
where

- Result: is not case-sensitive;
- `"result name"`: is a name for the Result, which will be used for menus; if the result name contains spaces it should be written between "" or between {}.
- `"analysis name"`: is the name of the analysis of this Result, which will be used for menus; if the analysis name contains spaces it should be written between "" or between {}.
- `step_value`: is the value of the step inside the analysis "analysis name";
- `my_type`: describes the type of the Result. It should be one of the following:
  - `Scalar`: one component per result
  - `Vector`: two, three or four components for result: x, y, z and (signed) modulus
  - `Matrix`: three components for 2D matrices, six components for 3D matrices
  - `PlainDeformationMatrix`: four components: Sxx, Syy, Sxy, Szz

- `MainMatrix`: the three main unitary eigen vectors vectors (three components each) and three eigen values of the matrix
- `LocalAxes`: three euler angles to specify the local axis
- `ComplexScalar`: two components to specify a + b · i
- `ComplexVector`: four components for 2D complex vectors, six or nine components for 3D vectors: rX iX rY iY rZ iZ |r| |i| |vector| --> to specify the vector (rX + iX, rY + iY, rZ + iZ)

- Look at the customization and reference manuals to get mor information on how to specify complex numbers for GiD.
  - `my_location`: is where the Result is located. It should be one of the following: `OnNodes`, `OnGaussPoints`. If the Result is `OnGaussPoints`, a `"location name"` should be entered;
  - `"location name"`: is the name of the Gauss Points on which the Result is defined.

Note: Results can be grouped into 'folders' like in the following picture



Results grouped into folders

by just grouping of results using slahes in the result names:

Result "`Mechanical//Pressures//Water pressure`" "Time analysis" 60 ...

Result "`Physical//Saturation`" "Time analysis" 60 Scalar OnNodes

and so on...

- Be followed (optionally) by result properties:

ResultRangesTable "Name of a result ranges table"

ComponentNames "Name of Component 1", "Name of Component 2"

Unit "result unit"

where

- ResultRangesTable "Name of a result ranges table": (optional) is not case-sensitive,

followed by the name of the previously defined Result Ranges Table, which will be used if the Contour Ranges result visualization is chosen (see Result Range Table -pag. 84-);

- ComponentNames "Name of Component 1", "Name of Component 2": (optional) is not case-sensitive, followed by the names of the components of the results which will be used in GiD. Missing components names will be automatically generated. The number of Component Names are:
  - One for a Scalar Result
  - Three for a Vector Result
  - Six for a Matrix Result
  - Four for a PlainDeformationMatrix Result
  - Six for a MainMatrix Result
  - Three for a LocalAxes Result
  - Two for a ComplexScalar Result
  - Six or nine for ComplexScalar
- Unit: the unit of the result.

- End with the result values:

Values

```
node_or_elem_number component_1_value component_2_value
. . . node_or_elem_number component_1_value component_2_value
```

End Values

where

- Values: is not case-sensitive, and indicates the beginning of the results values section;
- The lines
  - node_or_elem_number component_1_value component_2_value
  - . . .
  - node_or_elem_number component_1_value component_2_value

are the values of the result at the related 'node_or_elem_number'.

The number of results values are limited thus:

- If the Result is located OnNodes, they are limited to the number of nodes defined in ProjectName.flavia.msh.
- If the Result is located OnGaussPoints "My GP", and if the Gauss Points "My GP" are defined for the mesh "My mesh", the limit is the number of gauss points in "My GP" multiplied by the number of elements of the mesh "My mesh".

For results in gauss points, each element must have 'ngauss' lines of results.

For example, if the number of gauss points is 3, then for an element, 3 lines of gauss point result must appear.

```
Values
1 1.155
```

```
  2.9
  3.955
End Values
```

Holes are allowed in any result. The element nodes with no result defined will not be drawn, i.e. they will appear transparent.

The number of components for each Result Value are:

- for Scalar results: one component result_number_i scalar_value
- for Vector results: three components, with an optional fourth component for signed modules result_number_i x_value y_value z_value result_number_i x_value y_value z_value signed_module_value
- for Matrix results: three components (2D models) or six components (3D models)2D: result_number_i Sxx_value Syy_value Sxy_value3D: result_number_i Sxx_value Syy_value Szz_value Sxy_value Syz_value Sxz_value
- for PlainDeformationMatrix results: four components result_number_i Sxx_value Syy_value Sxy_value Szz_value
- for MainMatrix results: twelve components result_number_i Si_value Sii_value Siii_value Vix_value Viy_value Viz_value Viix_value Viiy_value Viiz_value Viiix_value Viiiy_value Viiiz_value
- for LocalAxes results: three components describing the Euler angles result_number_i euler_ang_1_value euler_ang_2_value euler_ang_3_value. The local axes willl be calculated like this:

```
VectorX(x) = cosC*cosA - cosB*sinA*sinC
VectorX(y) = -sinC*cosA - cosB*sinA*cosC
VectorX(z) = sinB*sinA
VectorY(x) = cosC*sinA - cosB*cosA*sinC
VectorY(y) = -sinC*sinA - cosB*cosA*cosC
VectorY(z) = -sinB*cosA
VectorZ(x) = sinC*sinB
VectorZ(y) = cosC*sinB
VectorZ(z) = cosB
```

where

```
cosA=cos(e1)
sinA=sin(e1)
cosB=cos(e2)
sinB=sin(e2)
cosC=cos(e3)
sinC=sin(e3)
```

and

```
e1=euler angle 1
e2=euler angle 2
e3=euler angle 3
```

Look for LocalAxesDef(EulerAngles) at Specific commands -pag. 38- for a more detailed explanation.

- for ComplexScalar results: two components to specify a + b · i
- for ComplexVector results: four components for 2D complex vectors, six or nine components for 3D vectors: rX iX rY iY rZ iZ |r| |i| |vector| --> to specify the vector (rX + iX, rY + iY, rZ + iZ)

- End Values: is not case-sensitive, and indicates the end of the results values section.

Note: For Matrix and PlainDeformationMatrix results, the Si, Sii and Siii components are calculated by GiD, which represents the eigen values & vectors of the matrix results, and which are ordered according to the eigen value.

,

6.1.4 Results example

Here is an example of results for the table in the previous example (see Mesh example -pag. 101-):

```
GiD Post Results File 1.0

GaussPoints "Board gauss internal" ElemType Triangle "board"
  Number Of Gauss Points: 3
  Natural Coordinates: internal
end gausspoints

GaussPoints "Board gauss given" ElemType Triangle "board"
  Number Of Gauss Points: 3
  Natural Coordinates: Given
      0.2 0.2
      0.6 0.2
      0.2 0.6
End gausspoints

GaussPoints "Board elements" ElemType Triangle "board"
  Number Of Gauss Points: 1
  Natural Coordinates: internal
```

```
end gausspoints

GaussPoints "Legs gauss points" ElemType Linear
  Number Of Gauss Points: 5
  Nodes included
  Natural Coordinates: Internal
End Gausspoints

ResultRangesTable "My table"
# el ultimo rango es min <= res <= max
      - 0.3: "Less"
  0.3 - 0.9: "Normal"
  0.9 - 1.2: "Too much"
End ResultRangesTable

Result "Gauss element" "Load Analysis" 1 Scalar OnGaussPoints "Board elements"
Values
    5 0.00000E+00
    6 0.20855E-04
    7 0.35517E-04
    8 0.46098E-04
    9 0.54377E-04
   10 0.60728E-04
   11 0.65328E-04
   12 0.68332E-04
   13 0.69931E-04
   14 0.70425E-04
   15 0.70452E-04
   16 0.51224E-04
   17 0.32917E-04
   18 0.15190E-04
   19 -0.32415E-05
   20 -0.22903E-04
   21 -0.22919E-04
   22 -0.22283E-04
End Values
```

```
Result "Displacements" "Load Analysis" 1 Vector OnNodes
ResultRangesTable "My table"
ComponentNames "X-Displ", "Y-Displ", "Z-Displ"
Values
    1 0.0 0.0 0.0
    2 -0.1 0.1 0.5
    3 0.0 0.0 0.8
    4 -0.04 0.04 1.0
    5 -0.05 0.05 0.7
    6 0.0 0.0 0.0
    7 -0.04 -0.04 1.0
    8 0.0 0.0 1.2
    9 -0.1 -0.1 0.5
   10 0.05 0.05 0.7
   11 -0.05 -0.05 0.7
   12 0.04 0.04 1.0
   13 0.04 -0.04 1.0
   14 0.05 -0.05 0.7
   15 0.0 0.0 0.0
   16 0.1 0.1 0.5
   17 0.0 0.0 0.8
   18 0.0 0.0 0.0
   19 0.1 -0.1 0.5
End Values
Result "Gauss displacements" "Load Analysis" 1 Vector OnGaussPoints "Board
gauss given"
Values
    5 0.1 -0.1 0.5
         0.0 0.0 0.8
         0.04 -0.04 1.0
    6 0.0 0.0 0.8
        -0.1 -0.1 0.5
        -0.04 -0.04 1.0
    7 -0.1 0.1 0.5
         0.0 0.0 0.8
```

```
      -0.04 0.04 1.0
 8 0.0 0.0 0.8
       0.1 0.1 0.5
       0.04 0.04 1.0
 9 0.04 0.04 1.0
       0.1 0.1 0.5
       0.05 0.05 0.7
10 0.04 0.04 1.0
       0.05 0.05 0.7
      -0.04 0.04 1.0
11 -0.04 -0.04 1.0
      -0.1 -0.1 0.5
      -0.05 -0.05 0.7
12 -0.04 -0.04 1.0
      -0.05 -0.05 0.7
       0.04 -0.04 1.0
13 -0.1 0.1 0.5
      -0.04 0.04 1.0
      -0.05 0.05 0.7
14 -0.05 0.05 0.7
      -0.04 0.04 1.0
       0.05 0.05 0.7
15 0.1 -0.1 0.5
       0.04 -0.04 1.0
       0.05 -0.05 0.7
16 0.05 -0.05 0.7
       0.04 -0.04 1.0
      -0.05 -0.05 0.7
17 0.0 0.0 0.8
      -0.04 -0.04 1.0
      -0.04 0.04 1.0
18 0.0 0.0 0.8
       0.04 0.04 1.0
       0.04 -0.04 1.0
19 0.04 -0.04 1.0
```

```
         0.04 0.04 1.0
         0.0 0.0 1.2
   20 0.04 -0.04 1.0
         0.0 0.0 1.2
        -0.04 -0.04 1.0
   21 -0.04 -0.04 1.0
         0.0 0.0 1.2
        -0.04 0.04 1.0
   22 -0.04 0.04 1.0
         0.0 0.0 1.2
         0.04 0.04 1.0
End Values

Result "Legs gauss  displacements" "Load  Analysis" 1 Vector  OnGaussPoints
"Legs gauss points"

Values
    1 -0.1 -0.1 0.5
        -0.2 -0.2 0.375
        -0.05 -0.05 0.25
         0.2 0.2 0.125
         0.0 0.0 0.0
    2 0.1 -0.1 0.5
         0.2 -0.2 0.375
         0.05 -0.05 0.25
        -0.2 0.2 0.125
         0.0 0.0 0.0
    3 0.1 0.1 0.5
         0.2 0.2 0.375
         0.05 0.05 0.25
        -0.2 -0.2 0.125
         0.0 0.0 0.0
    4 -0.1 0.1 0.5
        -0.2 0.2 0.375
        -0.05 0.05 0.25
         0.2 -0.2 0.125
         0.0 0.0 0.0
```

```
End Values
```

6.1.5 Result group

Results can be grouped into one block. These results belong to the same time step of the same analysis and are located in the same place. So all the results in the group are nodal results or are defined over the same gauss points set.

Each Result group is identified by a ResultGroup header, followed by the results descriptions and its optional properties - such as components names and ranges tables, and the results values - all between the lines Values and End values.

The structure is as follows and should:

- Begin with a header that follows this model

ResultGroup "analysis name" step_value my_location "location name"

where

- ResultGroup: is not case-sensitive;
- "analysis name": is the name of the analysis of this ResultGroup, which will be used for menus; if the analysis name contains spaces it should be written between "" or between {}.
- step_value: is the value of the step inside the analysis "analysis name";
- my_location: is where the ResultGroup is located. It should be one of the following: OnNodes, OnGaussPoints. If the ResultGroup is OnGaussPoints, a "location name" should be entered.
- "location name": is the name of the Gauss Points on which the ResultGroup is defined.

- Be followed by at least one of the results descriptions of the group

ResultDescription "result name" my_result_type[:components_number]

ResultRangesTable "Name of a result ranges table"

ComponentNames "Name of Component 1", "Name of Component 2"

where

- ResultDescription: is not case-sensitive;
- "result name": is a name for the Result, which will be used for menus; if the result name contains spaces it should be written between "" or between {}.
- my_type: describes the type of the Result. It should be one of the following: Scalar, Vector, Matrix, PlainDeformationMatrix, MainMatrix, or LocalAxes. The number of components for each type is as follows:
  - One for a Scalar: the_scalar_value
  - Three for a Vector: X, Y and Z
  - Six for a Matrix: Sxx, Syy, Szz, Sxy, Syz and Sxz
  - Four for a PlainDeformationMatrix: Sxx_value, Syy, Sxy and Szz
  - Twelve for a MainMatrix: Si, Sii, Siii, ViX, ViY, ViZ, ViiX, ViiY, ViiZ, ViiiX, ViiiY and

ViiiZ
- Three for a LocalAxes: euler_ang_1, euler_ang_2 and euler_ang_3

Following the description of the type of the result, an optional modifier can be appended to specify the number of components separated by a colon. It only makes sense to indicate the number of components on vectors and matrices:

- Vector:2, Vector:3 or Vector:4: which specify:
  - Vector:2: X and Y
  - Vector:3: X, Y and Z
  - Vector:4: X, Y, Z and |Vector| (module of the vector, with sign for some tricks)

The default (Vector) is 3 components per vector.

- Matrix:3 or Matrix:6: which specify:
  - Matrix:3: Sxx, Syy and Sxy
  - Matrix:6: Sxx, Syy, Szz, Sxy, Syz and Sxz

The default (Matrix) is 6 components for matrices.

Here are some examples:

```
ResultDescription "Displacements" Vector:2
```

```
ResultDescription "2D matrix" Matrix:3
```

```
ResultDescription "LineDiagramVector" Vector:4
```

and where (optional properties)

- ResultRangesTable "Name of a result ranges table": (optional) is not case-sensitive, and is followed by the name of the previously defined Result Ranges Table which will be used if the Contour Ranges result visualization is chosen (see Result Range Table -pag. 84-);
- ComponentNames "Name of Component 1", "Name of Component 2": (optional) is not case-sensitive, and is followed by the names of the components of the results which will be used in GiD. The number of Component Names are:
  - One for a Scalar Result
  - Three for a Vector Result
  - Six for a Matrix Result
  - Four for a PlainDeformationMatrix Result
  - Six for a MainMatrix Result
  - Three for a LocalAxes Result

- End with the results values:

Values

```
location_1       result_1_component_1_value       result_1_component_2_value
result_1_component_3_value                        result_2_component_2_value
result_2_component_2_value result_2_component_3_value

 . . .

location_n       result_1_component_1_value       result_1_component_2_value
```

```
result_1_component_3_value                    result_2_component_2_value
result_2_component_2_value result_2_component_3_value
```

End Values

where

- Values: is not case-sensitive, and indicates the beginning of the results values section;
- The lines
  - location_1 result_1_component_1_value result_1_component_2_value result_1_component_3_value result_2_component_2_value result_2_component_2_value result_2_component_3_value
  - . . .
  - location_n result_1_component_1_value result_1_component_2_value result_1_component_3_value result_2_component_2_value result_2_component_2_value result_2_component_3_value

are the values of the various results described with ResultDescription for each location. All the results values for the location 'i' should be written in the same line 'i'.

The number of results values are limited thus:

- If the Result is located OnNodes, they are limited to the number of nodes defined in ProjectName.post.msh, or the old ProjectName.flavia.msh.
- If the Result is located OnGaussPoints "My GP", and if the Gauss Points "My GP" are defined for the mesh "My mesh", the limit is the number of gauss points in "My GP" multiplied by the number of elements of the mesh "My mesh".

Holes are allowed. The element nodes with no result defined will not be drawn, i.e. they will appear transparent.

The number of components for each ResultDescription are:

- for Scalar results: one component result_number_i scalar_value
- for Vector results: three components result_number_i x_value y_value z_value
- for Matrix results: six components (3D models)3D: result_number_i Sxx_value Syy_value Szz_value Sxy_value Syz_value Sxz_value
- for PlainDeformationMatrix results: four components result_number_i Sxx_value Syy_value Sxy_value Szz_value
- for MainMatrix results: twelve components result_number_i Si_value Sii_value Siii_value Vix_value Viy_value Viz_value Viix_value Viiy_value Viiz_value Viiix_value Viiiy_value Viiiz_value
- for LocalAxes results: three components describing the Euler angles result_number_i euler_ang_1_value euler_ang_2_value euler_ang_3_value

- End Values: is not case-sensitive, and indicates the end of the results group values section.

Note: Vectors in a ResultGroup always have three components.

Note: Matrices in a ResultGroup always have six components.

Note: All the results of one node or gauss point should be written on the same line.

Note: For Matrix and PlainDeformationMatrix results, the Si, Sii and Siii components are calculated by GiD, which represents the eigen values & vectors of the matrix results, and which are ordered according to the eigen value.

Nodal ResultGroup example:

ResultGroup "Load Analysis" 1 OnNodes

ResultDescription "Ranges test" Scalar

ResultRangesTable "My table"

ResultDescription "Scalar test" Scalar

ResultRangesTable "Pressure"

ResultDescription "Displacements" Vector

ComponentNames "X-Displ", "Y-Displ" "Z-Displ"

ResultDescription "Nodal Stresses" Matrix

ComponentNames "Sx", "Sy", "Sz", "Sxy", "Syz", "Sxz"

Values

1 0.0 0.000E+00 0.000E+00 0.000E+00 0.0 0.550E+00 0.972E-01 -0.154E+00 0.0 0.0 0.0

2 6.4e-01 0.208E-04 0.208E-04 -0.191E-04 0.0 0.506E+00 0.338E-01 -0.105E+00 0.0 0.0 0.0

3 0.0 0.355E-04 0.355E-04 -0.376E-04 0.0 0.377E+00 0.441E-02 -0.547E-01 0.0 0.0 0.0

...

115 7.8e-01 0.427E-04 0.427E-04 -0.175E-03 0.0 0.156E-01 -0.158E-01 -0.300E-01 0.0 0.0 0.0

116 7.4e-01 0.243E-04 0.243E-04 -0.189E-03 0.0 0.216E-02 -0.968E-02 -0.231E-01 0.0 0.0 0.0

End Values

Gauss Points ResultGroup example:

GaussPoints "My Gauss" ElemType Triangle "2D Beam"

Number Of Gauss Points: 3

Natural Coordinates: Internal

End gausspoints

ResultGroup "Load Analysis" 1 OnGaussPoints "My Gauss"

ResultDescription "Gauss test" Scalar

```
ResultDescription "Vector Gauss" Vector
ResultDescription "Gauss Points Stresses" PlainDeformationMatrix
Values
  1 1.05 1 0 0.0 -19.4607 -1.15932 -1.43171 -6.18601
    2.1 0 1 0.0 -19.4607 -1.15932 -1.43171 -6.18601
    3.15 1 1 0.0 -19.4607 -1.15932 -1.43171 -6.18601
  2 1.2 0 0 0.0 -20.6207 0.596461 5.04752 -6.00727
    2.25 0 0 0.0 -20.6207 0.596461 5.04752 -6.00727
    3.3 2.0855e-05 -1.9174e-05 0.0 -20.6207 0.596461 5.04752 -6.00727
  3 1.35 2.0855e-05 -1.9174e-05 0.0 -16.0982 -1.25991 2.15101 -5.20742
    2.4 2.0855e-05 -1.9174e-05 0.0 -16.0982 -1.25991 2.15101 -5.20742
    3.45 2.0855e-05 -1.9174e-05 0.0 -16.0982 -1.25991 2.15101 -5.20742
...
191 29.55 4.2781e-05 -0.00017594 0.0 -0.468376 12.1979 0.610867 3.51885
    30.6 4.2781e-05 -0.00017594 0.0 -0.468376 12.1979 0.610867 3.51885
    31.65 4.2781e-05 -0.00017594 0.0 -0.468376 12.1979 0.610867 3.51885
192 29.7 4.2781e-05 -0.00017594 0.0 0.747727 11.0624 1.13201 3.54303
    30.75 4.2781e-05 -0.00017594 0.0 0.747727 11.0624 1.13201 3.54303
    31.8 2.4357e-05 -0.00018974 0.0 0.747727 11.0624 1.13201 3.54303
```

End Values

6.2 Postprocess mesh format: ProjectName.post.msh

Note: This postprocess mesh format requires GiD version 6.0 or higher.

Comments are allowed and should begin with a '#'. Blank lines are also allowed.

To enter the mesh names and result names in another encoding, just write # encoding your_encoding

for example:

```
 # encoding utf-8
```

Inside this file one or more MESHes can be defined, each of them should:

- Begin with a header that follows this model:

MESH "mesh_name" **dimension** my_dimension **Elemtype** my_type **Nnode** my_number

where

- MESH, dimension, elemtype, nnode: are keywords that are not case-sensitive;
- `"mesh_name":` is an optional name for the mesh;
- `my_dimension:` is 2 or 3 according to the geometric dimension of the mesh;
- `my_type:` describes the element type of this MESH. It should be one of the following; `Point, Linear, Triangle, Quadrilateral, Tetrahedra, Hexahedra, Prism, Pyramid, Sphere, Circle` ;
- `my_number:` the number of nodes of my_type element:
  - Point: 1 node,

Point connectivity:

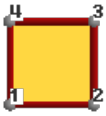
  - Linear: 2 or 3 nodes,

Line connectivities:

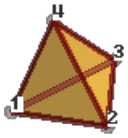
  - Triangle: 3 or 6 nodes,

Triangle connectivities:

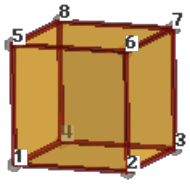
  - Quadrilateral: 4, 8 or 9 nodes,

Quadrilateral connectivities: 

  - `Tetrahedra`, 4 or 10 nodes,

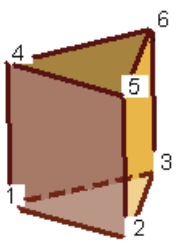
`Tetrahedra`, connectivities: 

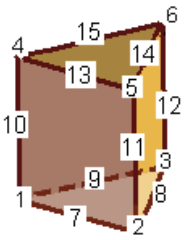
  - `Hexahedra`, 8, 20 or 27 nodes.

`Hexahedra`, connectivities: 

  - Prism: 6 or 15 nodes,

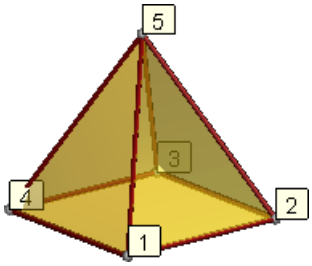
Prism connectivities:

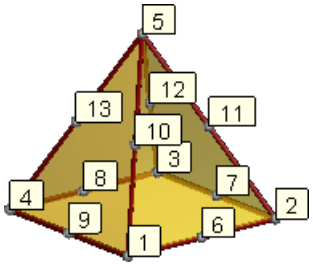




- Pyramid: 5 or 13 nodes,

Pyramid connectivities:





- Sphere: 1 node and a radius
- Circle: 1 node, a radius and a normal (x, y, z)

Note: For elements of order greater than linear, the connectivities must written in hierarchical order, i.e. the vertex nodes first, then the middle ones.

- An optional line describing its colour with # color R G B A, where R, G, B and A are the Red, Green, Blue and Alpha components of the color written in integer format between 0 and 255, or in floating (real) format between 0.0 and 1.0. (Note that if 1 is found in the line it will be understood as integer, and so 1 above 255, rather than floating, and so 1 above 1.0). Alpha values represent the transparency of the mesh when this visualization options is active, being 0.0 totally opaque and 1.0 totally transparent.

 # color 127 127 0

In this way different colours can be specified for several meshes, taking into account that the # color line must be between the MESH line and the Coordinates line.

- Be followed by the coordinates:

coordinates

```
1 0.0 1.0
3.0 . . .1000
-2.5 9.3 21.8
```

end coordinates

where

- the pair of keywords coordinates and end coordinates are not case-sensitive;
- between these keywords there should be the nodal coordinates of all the MESHes or the current one.

Note: If each MESH specifies its own coordinates, the node number should be unique, for

instance, if MESH "mesh one" uses nodes 1..100, and MESH "other mesh" uses 50 nodes, they should be numbered from 101 up.

- Be followed by the elements connectivity

elements

```
#el_num node_1 node_2 node_3 optional_material_number
1 1 2 3 215
. . .
1000 32 48 23 215
```

end elements

where

- the pair of keywords elements and end elements are not case-sensitive;
- between these keywords there should be the nodal connectivities for the my_type elements.

Note: On elements of order greater than linear, the connectivities must written in hierarchical order, i.e. the vertex nodes first, then the middle ones;

- There is optionally a material number.
- For **Sphere elements** : `Element_number Node_number Radius [optional_material_number]`
- For **Circle elements** : `Element_number Node_number Radius [optional_normal_x optional_normal_y optional_normal_z] [optional_material_number]`

If the normal is not written for circles, normal (0.0, 0.0, 1.0) will be used.

6.2.1 Mesh example

This example clarifies the description:

```
#mesh of a table
MESH "board" dimension 3 ElemType Triangle Nnode 3
# color 127 127 0
Coordinates
# node number coordinate_x coordinate_y coordinate_z
    1 -5 3 -3
    2 -5 3 0
    3 -5 0 0
    4 -2 2 0
    5 -1.66667 3 0
    6 -5 -3 -3
    7 -2 -2 0
```

```
    8 0 0 0
    9 -5 -3 0
   10 1.66667 3 0
   11 -1.66667 -3 0
   12 2 2 0
   13 2 -2 0
   14 1.66667 -3 0
   15 5 3 -3
   16 5 3 0
   17 5 0 0
   18 5 -3 -3
   19 5 -3 0
end coordinates

#we put both material in the same MESH,
#but they could be separated into two MESH

Elements
# element node_1 node_2 node_3 material_number
    5 19 17 13 3
    6 3 9 7 3
    7 2 3 4 3
    8 17 16 12 3
    9 12 16 10 3
   10 12 10 4 3
   11 7 9 11 3
   12 7 11 13 3
   13 2 4 5 3
   14 5 4 10 3
   15 19 13 14 3
   16 14 13 11 3
   17 3 7 4 3
   18 17 12 13 3
   19 13 12 8 4
   20 13 8 7 4
   21 7 8 4 4
```

```
   22 4 8 12 4
end elements
MESH dimension 3 ElemType Linear Nnode 2
Coordinates
#no coordinates then they are already in the first MESH
end coordinates
Elements
# element node_1 node_2 material_number
    1 9 6 5
    2 19 18 5
    3 16 15 5
    4 2 1 5
end elements
```

6.2.2 Group of meshes

If the same meshes are used for all the analyses, the following section can be skipped.

A new concept has been introduced in Postprocess: Group, which allows the postprocessing of problems which require re-meshing or adaptive meshes, where the mesh change depending on the time step.

Normal operations, such as animation, displaying results and cuts, can be done over these meshes, and they will be actualized when the selected analysis/step is changed, for example by means of View results -> Default analysis/step

There are two ways to enter in GiD the diferent meshes defined por diferent steps or analysis:

1. define separate files for each mesh, for instance:

- binary format: mesh+result_1.post.bin, mesh+result_2.post.bin, mesh+result_3.post.bin, ...

- ascii format: mesh_1.post.msh + mesh_1.post.res, mesh_2.post.msh + mesh_2.post.res, ...

Note: the steps values (or analysis) should be diferent for each pair mesh+result.

To read them you can use File-->Open Multiple (see POSTPROCESS > Files menu from Reference Manual)

2. define on binary file or two ascii files (msh+res):

Meshes that belong to a group should be defined between the following highlighted commands

Group "group name"

MESH "mesh_name" dimension ...

...

end elements

MESH "another_mesh" ...

...

end elements

end group

Results which refer to one of the groups should be written between these highlighted commands

OnGroup "group name"

Result "result name"

...

end values

...

end ongroup

Note: GiD versions 7.7.3b and later only allow one group at a time, i.e. only one group can be defined across several steps of the analysis. Care should be taken so that groups do not overlap inside the same step/analysis.

For instance, an analysis which is 10 steps long:

- For steps 1, 2, 3 and 4: an 'environment' mesh of 10000 elements and a 'body' mesh of 10000 elements are used

```
MESH "environment"
... Coordinates
...
10000 ...
end elements
MESH "body" ...
...
20000 ...
end elements
```

and its results

```
GiD Post Results File 1.0
...
```

```
Results "result 1" "time" 1.0 ...
...
Results "result 1" "time" 2.0 ...
...
Results "result 1" "time" 3.0 ...
...
Results "result 1" "time" 4.0 ...
...
end values
```

- For steps 5, 6, 7 and 8: with some refinement, the 'environment' mesh now being used has 15000 elements and the 'body' mesh needs 20000 elements

```
MESH "environment"
...
Coordinates
...
15000 ...
end elements
MESH "body" ...
...
 35000 ...
end elements
```

and its results are

```
GiD Post Results File 1.0
...
Results "result 1" "time" 5.0 ...
...
Results "result 1" "time" 6.0 ...
...
Results "result 1" "time" 7.0 ...
...
Results "result 1" "time" 8.0 ...
...
end values
```

- For steps 9 and 10: the last meshes to be used are of 20000 and 40000 elements,

respectively

```
MESH "environment" ...
Coordinates
...
20000 ...
end elements
MESH "body" ...
...
60000 ...
end elements
```

and its results are

```
GiD Post Results File 1.0
...
Results "result 1" "time" 9.0 ...
...
Results "result 1" "time" 10.0 ...
...
end values
```

There are two ways to postprocess this:

- store the information in three pairs (or three binary files), thus:
  - steps_1_2_3_4.post.msh and steps_1_2_3_4.post.msh (or steps_1_2_3_4.post.bin)
  - steps_5_6_7_8.post.msh and steps_5_6_7_8.post.msh (or steps_5_6_7_8.post.bin)
  - steps_9_10.post.msh and steps_9_10.post.msh (or steps_9_10.post.bin)

and use the 'Open multiple' option (see POSTPROCESS > Files menu from Reference Manual) to selected the six (or three) files; or

- write them in only two files (one in binary) by using the **Group** concept:
  - all_analysis.post.msh (note the group - end group pairs)

Group "steps 1, 2, 3 and 4"

MESH "environment" ...

...

MESH "body" ...

...

end group

Group "steps 5, 6, 7 and 8"

MESH "environment" ...

...

MESH "body" ...

...

end group

Group "steps 9 and 10"

MESH "environment" ...

...

MESH "body" ...

...

end group

and

- all_analysis.post.res (note the ongroup - end ongroup pairs)

GiD Post Results File 1.0

OnGroup "steps 1, 2, 3 and 4"

...

Results "result 1" "time" 1.0 ...

...

Results "result 1" "time" 2.0 ...

...

Results "result 1" "time" 3.0 ...

...

Results "result 1" "time" 4.0 ...

...

end ongroup

OnGroup "steps 5, 6, 7 and 8"

...

Results "result 1" "time" 5.0 ...

...

Results "result 1" "time" 6.0 ...

...

Results "result 1" "time" 7.0 ...

...

Results "result 1" "time" 8.0 ...

...

end ongroup

OnGroup "steps 9 and 10"

...

Results "result 1" "time" 9.0 ...

...

Results "result 1" "time" 10.0 ...

...

end ongroup

and use the normal Open option.

6.3 Postprocess list file: ProjectName.post.lst

New file *.post.lst can be read into GiD, postprocess. This file is automatically read when the user works in a GiD project and changes from pre to postprocess.

This file can also be read with File-->Open

The file contains a list of the files to be read by the postprocess:

- The first line should contain one of these words: Single / Merge / Multiple to read a single file, merge the list of files or handle them as several meshes (for diferent time steps);
- rest of lines: the files to be read, with one filename per line;
- both comments starting with '#' and blank lines are admited;
- if the filenames do not have its absolute path, then the path of the file containg the list will be used;
- graphs files can also be read (*.grf).

6.4 Postprocess graphs file: ProjectName.post.grf

The graph file that GiD uses is a standard ASCII file.

Every line of the file is a point on the graph with X and Y coordinates separated by a space.

Comment lines are also allowed and should begin with a '#'.

The title of the graph and the labels for the X and Y axes can also be configured with some 'special' comments:

- Title: If a comment line contains the Keyword 'Graph:' the string between quotes that follows this keyword will be used as the title of the graph.
- Axes labels: The string between quotes that follows the keywords 'X:' and 'Y:' will be used as labels for the X- and Y-axes respectively. The same is true for the Y axis, but with the Keyword 'Y:'.

- Axes units:

Example:

```
# Graph: "Stresses"
#
# X: "Szz-Nodal_Streess" Y: "Sxz-Nodal_Stress"
# Units: Pa Pa
-3055.444 1672.365
-2837.013 5892.115
-2371.195 666.9543
-2030.643 3390.457
-1588.883 -4042.649
-1011.5 1236.958
# End
```

The file *.grf (which contains graphs) is read when changing from pre to post process and projectName.gid/projectName.post.grf exists, or the postprocess files are read through File-->Open, then example.msh/res/bin and example.grf are read.

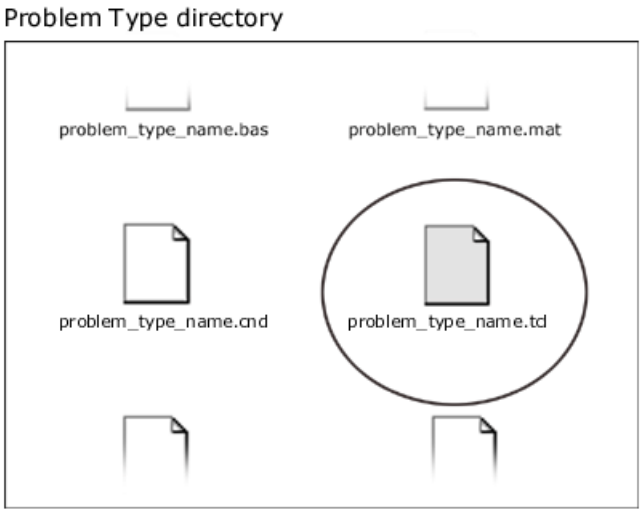
The post list file (*.post.lst) can also contain a list of graphs (*.grf).

7 TCL AND TK EXTENSION

This chapter looks at the advanced features of GiD in terms of expandability and total control. Using the Tcl/Tk extension you can create script files to automatize any process created with GiD. With this language new windows and functionalities can be added to the program.

For more information about the Tcl/Tk programming language itself, look at www.scriptics.com http://www.scriptics.com.

If you are going to use a Tcl file, it must be located in the Problem Type directory and be called problem_type_name.tcl.



7.1 Event procedures

The structure of problem_type_name.tcl can optionally implement some of these Tcl prototype procedures (and other user-defined procedures). The procedures listed below are automatically called by GiD. Their syntax corresponds to standard Tcl/Tk language:

```
proc InitGIDProject { dir } {
}

proc BeforeInitGIDPostProcess {} {
}

proc InitGIDPostProcess {} {
  ...body...
  set value ...
  return $value
}

proc EndGIDProject {} {
```

```
}

proc EndGIDPostProcess {} {

}

proc AfterOpenFile { filename format error } {

}

proc AfterSaveImage { filename format } {

}

proc LoadGIDProject { filespd } {

}

proc SaveGIDProject { filespd } {

}

proc LoadResultsGIDPostProcess { file } {

}

proc BeforeMeshGeneration { elementsize } {
  ...body...
  set value ...
  return $value
}

proc AfterMeshGeneration { fail } {
  ...body...
  set value ...
  return $value
}

proc AfterRenumber { useof leveltype renumeration } {

}

proc SelectGIDBatFile { dir basename } {
  ...body...
  set value ...
  return $value
}

proc  BeforeRunCalculation { batfilename basename dir problemtypedir gidexe
args } {
```

```
  ...body...
  set value ...
  return $value
}
proc    AfterRunCalculation {  basename dir   problemtypedir  where  error
errorfilename } {
  ...body...
  set value ...
  return $value
}
proc ChangedLanguage { language } {
}
proc BeforeWriteCalcFileGIDProject { file } {
  ...body...
  set value ...
  return $value
}
proc AfterWriteCalcFileGIDProject { file error } {
  ...body...
  set value ...
  return $value
}
proc BeforeTransformProblemType { file oldproblemtype newproblemtype } {
  ...body...
  set value ...
  return $value
}
proc AfterTransformProblemType { file oldproblemtype newproblemtype } {
}
proc TclCalcModelBoundaries{ useof } {
  ...body...
  return "$xmin $ymin $zmin $xmax $ymax $zmax"
}
```

```
proc AfterChangeBackground { } {
}
proc BeforeCopy { useof transformation error } {
}
proc AfterCopy { useof transformation error } {
}
proc BeforeMove { useof transformation error } {
}
proc AfterMove { useof transformation error } {
}
proc AfterCreatePoint { num } {
}
proc AfterCreateLine { num } {
}
proc AfterCreateSurface { num } {
}
proc AfterCreateVolume { num } {
}
proc BeforeDeletePoint { num } {
}
proc BeforeDeleteLine { num } {
}
proc BeforeDeleteSurface { num } {
}
proc BeforeDeleteVolume { num } {
}
proc AfterCreateLayer { name } {
}
proc AfterRenameLayer { oldname newname } {
}
proc BeforeDeleteLayer { name } {
```

```
  ...body...
  set value ...
  return $value
}
proc AfterChangeLayer { name property } {
}
proc AfterSetLayerToUse { name } {
}
proc AfterChangeLicenceStatus { status } {
}
proc AfterCreateMaterial { name } {
}
proc AfterRenameMaterial { oldname newname } {
}
proc BeforeDeleteMaterial { name } {
  ...body...
  set value ...
  return $value
}
proc AfterChangeMaterial { name changedfields } {
}
proc AfterAssignMaterial { name leveltype } {
}
proc BeforeMeshErrors { filename } {
   ...body...
  set value ...
  return $value
}
proc BeforeResultReadErrors { filename msg format } {
   ...body...
  set value ...
  return $value
```

```
}
```

- **InitGIDProject**: will be called when the problem type is selected. It receives the dir argument, which is the absolute path to the problem_type_name.gid directory, which can be useful inside the routine to locate some alternative files.
- **BeforeInitGIDPostProcess**: will be called just before changing from pre to postprocess, and before read any postprocess file (this event can be used for example to check the results file existence and/or rename files). It has no arguments.

If it returns *-cancel-* as a value then the swapping to postprocess mode will be cancelled.

- **InitGIDPostProcess**: will be called when postprocessing starts. It has no arguments.
- **EndGIDProject**: will be called when this project is about to be closed. It has no arguments.
- **EndGIDPostProcess**: will be called when you leave Postprocess and open Preprocess. It has no arguments.
- **AfterOpenFile**: will be called after a geometry or mesh file is read inside GiD. It receives as arguments:
  - filename: the full name of the file that has been read;
  - format: ACIS_FORMAT, CGNS_FORMAT, DXF_FORMAT, GID_BATCH_FORMAT, GID_GEOMETRY_FORMAT, GID_MESH_FORMAT, IGES_FORMAT, NASTRAN_FORMAT, PARASOLID_FORMAT, RHINO_FORMAT, SHAPEFILE_FORMAT, STL_FORMAT, VDA_FORMAT, VRML_FORMAT or 3DSTUDIO_FORMAT;
  - error: boolean 0 or 1 to indicate an error when reading.
- **AfterSaveImage**: will be called after a picture or model is saved to disk. It receives as arguments:
  - filename: the full name of the file that has been saved;
  - format: eps, ps, tif, bmp, ppm, gif, png, jpg, tga, wrl
- **LoadGIDProject**: will be called when a GiD project or problem type is loaded. It receives the argument filespd, which is the path of the file which is being opened, but with the extension .spd (**s**pecific **p**roblemtype **d**ata). This path can be useful if you want to write specific information about the problem type in a new file.
- **SaveGIDProject**: will be called when the currently open file is saved to disk. It receives the argument filespd, which is the path of the file being saved, but with the extension .spd (**s**pecific **p**roblemtype **d**ata). This path can be useful if you want to write specific information about the problem type in a new file.
- **LoadResultsGIDPostProcess**: will be called when a results file is opened in GiD Postprocess. It receives one argument, the name of the file being opened without its extension.
- **BeforeMeshGeneration**: will be called before the mesh generation. It receives the mesh size desired by the user as the elementsize argument. This event can typically be used to assign some condition automatically.

If it returns *-cancel-* the mesh generation is cancelled.

- **AfterMeshGeneration**: will be called after the mesh generation. It receives as its fail argument a true value if the mesh is not created.

If it returns *-cancel-* ?<message>? then the window showing the end of the mesh

generation will not be raised, but the mesh is not deleted. <message> is an optional message to be shown in the message bar.

- **AfterRenumber**: will be called after renumber the geometry or the mesh (to update for example fields storing entity identifiers)
  - useof : could be GEOMETRYUSE or MESHUSE
  - leveltype: the kind of entity that was renumbered.

  Geometry: must be ALL_LT.

  Mesh: could be NODE_LT or ELEM_LT.

  - renumeration:

  Geometry: four sublists with the old and new idenfiers for points, lines, surfaces and volumes.

  Mesh: a sublist with the old and new identifiers for nodes or elements.

- **SelectGIDBatFile**: must be used to switch the default batch file for special cases.

This procedure must return as a value the alternative pathname of the batch file. For example it is used as a trick to select a different analysis from a list of batch calculation files.

- **BeforeRunCalculation**: will be called before running the analysis. It receives several arguments:
  - batfilename: the name of the batch file to be run (see EXECUTING AN EXTERNAL PROGRAM -pag. 67-);
  - basename: the short name model;
  - dir: the full path to the model directory;
  - problemtypedir: the full path to the Problem Types directory;
  - gidexe: the full path to gid;
  - args: an optional list with other arguments.

If it returns *-cancel-* ?<message>? then the calculation is not started. <message> is an optional message to be shown.

- **AfterRunCalculation**: will be called just after the analysis finishes.

If it returns *nowindow* as a value then the window that inform about the finished process will not be opened.

It receives as arguments:

  - basename: the short name model;
  - dir: the full path to the model directory;
  - problemtypedir: the full path to the Problem Types directory;
  - where: must be local or remote (remote if it was run in a server machine, using ProcServer);
  - error: returns 1 if an calculation error was detected;
  - errorfilename: an error filename with some error explanation, or nothing if everything was ok.

- **ChangedLanguage**: will be called when you change the current language. The argument is the new language (en, es, ...). It is used, for example, to update problem

type customized menus, etc.

- **BeforeWriteCalcFileGIDProject**: will be called just before writing the calculation file. It is useful for validating some parameters.

If it returns *-cancel-* as a value then nothing will be written.

  - file: the name of the output calculation file.
- **AfterWriteCalcFileGIDProject**: will be called just after writing the calculation file and before the calculation process. It is useful for renaming files, or cancelling the analysis.

If it returns -cancel- as a value then the calculation is not invoked.

  - file: the name of the output calculation file error: an error code if there is some problem writing the output calculation file.
- **BeforeTransformProblemType**: will be called just before transforming a model from a problem type to a new problem type version. If it returns *-cancel-* as a value then the transformation will not be invoked.
  - file: the name of the model to be transformed;
  - oldproblemtype: the name of the previous problem type;
  - newproblemtype: the name of the problem type to be transformed.
- **AfterTransformProblemType**: will be called just after transforming a model from a problem type to a new problem type version. If must return 1 if there were changes, 0 else.
  - file: the name of the model to be transformed;
  - oldproblemtype: the name of the previous problem type;
  - newproblemtype: the name of the problem type to be transformed.
- **TclCalcModelBoundaries**: will be called when recalculating the bounding box, for example when user select "zoom frame"
  - useof: can be "GEOMETRYUSE", "MESHUSE", "POSTUSE" or "GRAFUSE".

This procedure must return xmin ymin zmin xmax ymax zmaz of the bounding box of the entities directly managed by the problemtype (this entities must be directly drawn with the drawopengl command).

If "" is returned instead "xmin ymin zmin xmax ymax zmaz" then any additional bounding box is considered.

- **AfterChangeBackground**: will be called just after change some background property, like color, direction or image.
- **BeforeCopy/Move AfterCopy/Move**: will be called just before or after use copy or move tools.
  - useof : could be GEOMETRYUSE or MESHUSE
  - transformation : could be ROTATION, TRANSLATION, MIRROR, SCALE, OFFSET, SWEEP or ALIGN
- **AfterCreatePoint/Line/Surface/Volume**: will be called just after create the entity, providing its number
- **BeforeDeletePoint/Line/Surface/Volume**: will be called just before delete the entity, providing its number
- **AfterCreateLayer:** will be called just after create the layer 'name'
- **AfterRenameLayer**: will be called just after the layer 'oldname' has been renamed to

'newname'

- **BeforeDeleteLayer**: will be called just before delete the layer 'name'

If it returns *-cancel-* the layer deletion is cancelled.

- **AfterChangeLayer**: will be called just after change some property of the layer 'name'
  - 'property' could be ON, OFF, FROZEN, UNFROZEN, ALPHA <AAA>, COLOR <RRRGGGBBB?AAA?>

whit RRR, GGG, BBB, AAA from 0 to 255

- **AfterSetLayerToUse**: will be called when setting 'name' as current layer to use
- **AfterChangeLicenceStatus**: will be called when the licence status of GiD changes. Possible status could be "academic", "professional" or "temporallyprofessional"
- **AfterCreateMaterial:** will be called just after create the material 'name'
- **AfterRenameMaterial**: will be called just after the mateial 'oldname' has been renamed to 'newname'
- **BeforeDeleteMaterial**: will be called just before delete the material 'name'

If it returns *-cancel-* the material deletion is cancelled.

- **AfterChangeMaterial**: will be called just after change some field value of the material 'name'.

`changedfields` is a list with the index of the changed fields (index starting from 1)

- **AfterAssignMaterial**: will be called just after assign or unassign the material of some entities.

name is the name of the new material. If it is "" then the material has been unassigned.

leveltype: is the kind of entity, if could be:

For geometry: POINT_LT, LINE_LT, SURFACE_LT,VOLUME_LT

For mesh: ELEM_LT.

- **BeforeMeshErrors:** filename is the full path to the file that has information about the meshing errors.

Retuning -cancel- the standard 'Mesh error window' won't be opened

- **BeforeResultReadErrors:** filename is the results file that was read, msg is the error message, format provide information about the kind of file: cand be "GID_RESULTS_FORMAT", "3DSTUDIO_FORMAT", "TECPLOT_FORMAT", "FEMAP_FORMAT", "XYZ_FORMAT"

Retuning -cancel- the standard 'Read results error window' won't be opened

Note: To use Tcl to improve the capabilities of writing the calculations file, it is possible to use the command *tcl in the template file (.bas file); see Specific commands -pag. 38- for details.

7.2 GiD_Process function

GiD_Process *command_1 command_2 ...*

Tcl command used to execute GiD commands.

This is a simple function, but a very powerful one. It is used to enter commands directly inside the central event manager. The commands have the same form as those typed in the command line within GiD.

You have to enter exactly the same sequence as you would do interactively, including the escape sequences (using the word escape) and selecting the menus and operations used.

You can obtain the exact commands that GiD needs by checking the **Right buttons** menu (Utilities -> Tools -> Toolbars). It is also possible to save a batch file (Utilities -> Preferences) where the commands used during the GiD session can be checked.

Here is a simple example to create one line:

```
GiD_Process Mescape Geometry Create Line 0,0,0 10,0,0 escape
```

Note: `Mescape` is a multiple `'escape'` command, to go to the top of the commands tree.

7.3 GiD_Info function

GiD_Info <option>

Tcl command used to obtain information about the current GiD project.

This function provides any information about GiD, the current data or the state of any task inside the application.

Depending on the arguments introduced after the GiD_Info sentence, GiD will output different information:

7.3.1 materials

GiD_Info materials

This command returns a list of the materials in the project.

These options are also available:

- **<material_name>**: If a material name is given, its properties are returned. It is also possible to add the option **otherfields** to get the fields of that material, or the option [BOOK] to get the book of that material.
- **books***:* If this option is given, a list of the material books in the project is returned.

Examples:

```
in: GiD_Info materials
```

```
out: "Air Steel Aluminium Concrete Water Sand"
```

```
in: GiD_Info materials Steel
```

```
out: "1 Density 7850"
```

```
GiD_Info materials Steel otherfields
```

```
GiD_Info materials books
```

7.3.2 conditions

GiD_Info conditions ovpnt | ovline | ovsurf | ovvol

This command returns a list of the conditions in the project. One of the arguments ovpnt, ovline, ovsurf, ovvol must be given to indicate the type of condition required, respectively, conditions over points, lines, surfaces or volumes.

Instead of ovpnt, ovline, ovsurf, ovvol, the following options are also available:

- **[-interval <intv>] [-localaxes | -localaxesmat | -localaxescenter | -localaxesmatcenter] <condition_name> [geometry | mesh]**: if a condition name is given, the command returns the properties of that condition. It is also possible to add the options geometry or mesh, and all geometry or mesh entities that have this condition assigned will be returned.

If -interval "intv" is set, then the conditions on this interval ("intv"=1,...n) are returned instead of those on the current interval.

If -localaxes is set, then the three numbers that are the three Euler angles that define a local axes system are also returned (only for conditions with local axes, see DATA>Local Axes from Reference Manual).

Selecting -localaxesmat, the nine numbers that define the transformation matrix of a vector from the local axes system to the global one are returned.

If -localaxescenter is set, then the three Euler angles and the local axis center are also returned.

Selecting -localaxesmatcenter returns the nine matrix numbers and the center.

Adding the number id of an entity (["entity_id"]) after the options mesh or geometry, the command returns the value of the condition assigned to that entity.

Other options available if the condition name is given are **[otherfields]**, to get the fields of that condition, and **[book]**, to get the book of the condition.

- **[books]**: If this option is given, a list of the condition books of the project is returned.

Examples:

```
in: GiD_Info conditions ovpnt
out: "Point-Weight Point-Load"

in: GiD_Info conditions Point-Weight
out: "ovpnt 1 Weight 0"

in: GiD_Info conditions Point-Weight geometry
out: "E 1 - 2334 , E 2 - 2334 , E 3 - 343"

in: GiD_Info Conditions -localaxes Concrete_rec_section mesh 2
out: {E 2 - {4.7123889803846897 1.5707963267948966 0.0} N-m 0.3 0.3 HA-25}
```

7.3.3 layers

GiD_Info layers

This command returns a list of the layers in the project.These options are also available:

- **[<layer_name>]***:* If a layer name is given, the command returns the properties of that layer.
- **[-on]***:* Returns a list of the visible layers.
- **[-off]***:* Returns a list of the hidden layers.
- **[-hasbacklayers]***:* Returns 1 if the project has entities inside back layers.
- **GiD_Info back_layers** returns a list with the back layers

Example:

```
in: GiD_Info back_layers
out: Layer2_*back*
```

- **[-bbox[-use geometry|mesh]]: layer_name_1 layer_name_2 ...]**: Returns two coordinates (x1,y1,z1,x2,y2,z2) which define the bounding box of the entities that belong to the list of layers.

If the option [-use geometry|mesh] is used, the command returns the bounding box of the **geometry** or the bounding box of the **mesh**.

If the list of layers is empty, the maximum bounding box is returned.

- **[-entities <type> ?-elementtype <elementtype>? ?-higherentity <num>?]***:* One of the following arguments must be given for *<type>*: **nodes, elements, points, lines, surfaces or volumes***.* A layer name must also be given. The command will return the nodes, elements, points, lines surfaces or volumes of that layer.

For elements it is possible to specify **-elementtype <elementtype>**to get only this kind of elements.

-higherentity <num>Allow to get only entities with this amount of higherentities.

Examples:

```
in: GiD_Info layers
out: "layer1 layer2 layer_aux"

in: GiD_Info layers -on
out: "layer1 layer2"

in: GiD_Info layers -entities lines layer2
out: "6 7 8 9"
```

7.3.4 gendata

GiD_Info gendata

This command returns the information entered in the Problem Data window (see Problem

and intervals data file (.prb) -pag. 15-).

The following options are available:

- **[otherfields]***:* It is possible to add this option to get the additional fields from the Problem Data window.
- *[books]:* If this option is given, a list of the Problem Data books in the project is returned.

Example:

```
in: GiD_Info gendata
```

```
out: "2 Unit_System#CB#(SI,CGS,User) SI Title M_title"
```

7.3.5 intvdata

GiD_Info intvdata

This command returns a list of the interval data in the project (see Problem and intervals data file (.prb) -pag. 15-).

The following options are available:

- **-interval <number>**: To get data from an interval different from the number 0 (default).
- **[otherfields]***:* It is possible to add this option to get the additional fields from the Interval Data window.
- **[books]***:* If this option is given, a list of the Interval Data books in the project is returned.
- **[num]***:* If this option is given, a list of two natural numbers is returned. The first element of the list is the current interval and the second element is the total number of intervals.

7.3.6 project

GiD_Info Project <item>?

This command returns information about the project. More precisely, it returns a list with:

- Problem type name.
- Current model name.
- 'There are changes' flag.
- Current layer to use.
- Active part (GEOMETRYUSE, MESHUSE, POSTUSE or GRAFUSE).
- Quadratic problem flag.
- Drawing type (normal, polygons, render, postprocess).
- NOPOST or YESPOST.
- Debug or nodebug.
- GiD temporary directory.
- Must regenerate the mesh flag (0 or 1).
- Last element size used for meshing (NONE if there is no mesh).
- BackgroundFilename is the name of a background mesh file to assign mesh sizes.

- RequireMeshSize. (1 if all sizes are specified by the number of divisions, then user is not required to specify the mesh size)
- RecommendedMeshSize. (The value of the mesh size that the program will recommend, based on the model size)

It is possible to ask for a single item only rather than the whole list, with <item> equal to:

ProblemType | ModelName | AreChanges | LayerToUse | ViewMode | Quadratic | RenderMode | ExistPost | Debug | TmpDirectory | MustReMesh | LastElementSize | BackgroundFilename | RequireMeshSize | RecommendedMeshSize

Example:

```
in: GiD_Info Project
out: "cmas2d e:\models\car_model 1 layer3 MESHUSE 0 normal YESPOST nodebug C:\TEMP\gid2 0 1.4"

in: GiD_Info Project ModelName
out: "e:\models\car_model"
```

7.3.7 geometry

GiD_Info Geometry

This command gives the user information about the geometry in the project. For each entity, there are two possibilities:

- **[NumPoints]***:* Returns the total number of points in the model.
- **[NumLines]***:* Returns the total number of lines.
- **[NumSurfaces]***:* Returns the total number of surfaces.
- **[NumVolumes]***:* Returns the total number of volumes.
- **[NumDimensions]***:* Returns the total number of dimensions.
- **[MaxNumPoints]***:* Returns the maximum point number used (can be higher than NumPoints).
- **[MaxNumLines]***:* Returns the maximum line number used.
- **[MaxNumSurfaces]***:* Returns the maximum surface number used.
- **[MaxNumVolumes]***:* Returns the maximum volume number used.
- **[MaxNumDimensions]***:* Returns the maximum dimension number used.

7.3.8 mesh

GiD_Info Mesh

This command gives the user information about the selected mesh in the project.

It returns a 1 followed by a list with all types of element used in the mesh.

- **[NumElements [Elemtype] [nnode]]**: returns the number of elements of the mesh.

  Elemtype can be: **Linear / Triangle / Quadrilateral / Tetrahedra / Hexahedra / Prism / Pyramid / Point / Sphere / Circle / Any**.

nnode is the number of nodes of an element.

- **[NumNodes]***:* Returns the total number of nodes of the mesh.
- **[MaxNumElements]***:* Returns the maximum element number.
- **[MaxNumNodes]***:* Returns the maximum node number.
- **[-pre | -post -step <step>]***:* To specify to use the preproces or postprocess mesh, and the time step if it changes along the time.
- **[Elements Elemtype[First_idLast_id]]***: R*eturns a list with the element number, the connectivities , radius if it is a sphere, normal if it is a circle, and the material number, from 'first_id' to 'last_id, if they are specified.
- **[Nodes[First_idLast_id]]***:* Returns a list with the node number and x y z coordinates, from 'first_id' to 'last_id', if they are specified.
- **[-sublist]** *:* Returns each result item as a Tcl list (enclosed in braces)
- **[-array]** *:* Returns the resuls as a list of vectors (more efficient)

Examples:

in: GiD_Info Mesh

out: "1 Tetrahedra Triangle"

in: GiD_Info Mesh MaxNumNodes

out: "1623"

7.3.9 coordinates

- **GiD_Info Coordinates point_id|node_id [geometry|mesh]**

This command returns the coordinates (x,y,z) of a given point or node.

7.3.10 variables

GiD_Info variables "variable_name"

This command returns the value of the variable indicated.

GiD variables can be found in the Right buttons *menu (see UTILITIES>Tools from Reference Manual), under the option Utilities -> Variables.*

7.3.11 localaxes

GiD_Info localaxes ?<name>? ?-localaxesmat?info localaxes

Returns a list with all the user defined local axes.

info localaxes <name> returns the parameters (three euler angles and the center) that define the local axes called <name>.

info localaxes <name> -localaxesmat instead of returning the three euler angles, it returns the nine numbers that define the transformation matrix of a vector from the local axes system to the global one.

7.3.12 ortholimits

GiD_Info ortholimits

This command returns a list (Left, Right, Bottom, Top, Near, Far, Ext) of the limits of the geometry in the project.

In perspective mode near and far have the perspective distance substracted.

7.3.13 perspectivefactor

GiD_Info perspectivefactor

This command returns which perspective factor is currently being used in the project.

7.3.14 graphcenter

GiD_Info graphcenter

This command returns the coordinates (x,y,z) of the center of rotation.

7.3.15 meshquality

GiD_Info MeshQuality

This command returns a list of numbers. These numbers are the **Y** *relative values of the graph shown in the option* Meshing -> Mesh quality *(see MESH>Mesh Quality from Reference Manual) and two additional real numbers with the minimum and maximum limits.*

This command has the following arguments:

- **MinAngle / MaxAngle / ElemSize / ElemMinEdge / ElemMaxEdge / ElemShapeQuality / ElemMinJacobian / Radius***: quality criterion.*
- **Linear / Triangle / Tetrahedra / Quadrilateral / Hexahedra / Prism / Pyramid / Point / Sphere / Circle***: type of element.*
- **<min_value>***: e.g. minimum number of degrees accepted.*
- **<max_value>***: e.g. maximum number of degrees accepted.*

if min_value and max_value are set to 0 then limits will be automatically set to the minimum and maximum of the mesh

- **<num_divisions>***: number of divisions.*

Example:

in: GiD_Info MeshQuality MinAngle Triangle 20 60 4

out: "13 34 23 0 20.0 60.0"

7.3.16 postprocess

GiD_Info postproces

- **GiD_Info postproces arewein**

This command returns YES if the user is in the GiD postprocess, and NO, if not.

- **GiD_Info postprocess get**

This command returns information about the GiD postprocess. The following options are available:

- **all_volumes***: Returns a list of all volumes.*
- **all_surfaces***: Returns a list of all surfaces.*
- **all_cuts***: Returns a list of all cuts.*
- **all_graphs***: Returns a list of all graphs.*
- **all_volumes_colors***: Returns a list of the volume colors used in the project. Colors are represented in RGB hexadecimal format. Example: #000000 would be black, and #FFFFFF would be white.*
- **all_surfaces_colors***: Returns a list of the surface colors used in the project. Colors are represented in RGB hexadecimal format. Example: #000000 would be black, and #FFFFFF would be white.*
- **all_cuts_colors** *: Returns a list of the cut colors used in the project. Colors are represented in RGB hexadecimal format. Example: #000000 would be black, and #FFFFFF would be white.*
- **cur_volumes***: Returns a list of the visible volumes.*
- **cur_surfaces***: Returns a list of the visible surfaces.*
- **cur_cuts***: Returns a list of the visible cuts.*
- **all_display_styles***: Returns a list of all types of display styles available.*
- **cur_display_style***: Returns the current display style.*
- **all_display_renders***: Returns a list of all types of rendering available.*
- **cur_display_render***: Returns the current rendering method.*
- **all_display_culling***: Returns a list of all types of culling available.*
- **cur_display_culling***: Returns the current culling visualization.*
- **cur_display_transparence***: Returns Opaque or Transparent depending on the current transparency. Transparency is chosen by the user in the* **Select & Display Style** *window.*
- **cur_display_body_type***: Returns Massive if the option* **Massive** *is selected in the* **Select & Display Style** *window. It returns Hollow if that option is not activated.*
- **all_analysis***: Returns a list of all analyses in the project.*
- **all_steps "analysis_name"***: Returns the number of steps of "analysis_name".*
- **cur_analysis***: Returns the name of the current analysis.*
- **cur_step***: Returns the current step.*
- **all_results_views***: Returns all available result views.*
- **cur_results_view***: Returns the current result view.*
- **cur_results_list***: This option has one more argument: the kind of result visualization must be given. The available kinds of result visualization are given by the option* **all_results_views***. The command returns a list of all the results that can be represented with that visualization in the current step of the current analysis.*
- **results_list***: This option has three arguments: the first argument is the kind of result visualization. The available kinds of result visualization are given by the option* **all_results_views***. The second argument is the analysis name. The third argument is the step number. The command returns a list of all the results that can be*

represented with that visualization in the given step.

- **cur_result***: Returns the current selected result. The kind of result is selected by the user in the* **View results** *window.*
- **cur_components_list "result_name"***: Returns a list of all the components of the result "result_name".*
- **cur_component***: Returns the current component of the current result.*
- **main_geom_state***: Returns whether the main geometry is Deformed or Original*
- **main_geom_all_deform***: Returns a list of all the deformation variables (vectors) of the main geometry.*
- **main_geom_cur_deform***: Returns the current deformation variable (vectors) of the main geometry.*
- **main_geom_cur_step***: Returns the main geometry current step.*
- **main_geom_cur_anal***: Returns the main geometry current analysis.*
- **main_geom_cur_factor***: Returns the main geometry current deformation factor.*
- **show_geom_state***: Returns whether the reference geometry is Deformed or Original.*
- **show_geom_cur_deform***: Returns the current deformation variable (vectors) of the reference geometry.*
- **show_geom_cur_analysis***: Returns the reference geometry current analysis.*
- **show_geom_cur_step***: Returns the reference geometry current step.*
- **iso_all_display_styles***: Returns a list of all available display styles for isosurfaces.*
- **iso_cur_display_style***: Returns the current display style for isosurfaces.*
- **iso_all_display_renders***: Returns a list of all types of rendering available for isosurfaces.*
- **iso_cur_display_render***: Returns the current rendering method for isosurfaces.*
- **iso_cur_display_transparence***: Returns* **Opaque** *or* **Transparent** *depending on the current transparency of isosurfaces.*
- **contour_limits***: Returns the minimum and maximum value of the contour limits. Before each value, the word STD appears if the contour limit value is the default value, and USER if it is defined by the user.*
- **animationformat***: Returns the default animation format.*
- **cur_show_conditions***: Returns the option selected in the* **Conditions** *combo box of the* **Select & Display Style** *window. (Possible values: Geometry Mesh None)*
- **all_show_conditions***: Returns all the options available in the* **Conditions** *combo box of the* **Select & Display Style** *window. (Geometry Mesh None)*
- **cur_contour_limits***: Returns the minimum and maximum value of the current value.*
- **current_color_scale***: Returns a list of the colors used for the color scale; the first element of the list is the number of colors. Each color is represented in RGB hexadecimal format. Example: #000000 would be black, and #FFFFFF would be white.*

7.3.17 automatictolerance

GiD_Info AutomaticTolerance

This command returns the value of the Import Tolerance *used in the project. This value is*

defined in the Preferences *window under* Import*.*

7.3.18 rgbdefaultbackground

GiD_Info RGBDefaultBackground

This command returns the default background color in RGB. The format is three 8 bit numbers separated by #. Example: 255#255#255 would be white.

7.3.19 list_entities

GiD_Info list_entities

- **GiD_Info list_entities Status|PreStatus|PostStatus**

This command returns a list with general information about the current GiD project.

PreStatus ask for the information of preprocess

PostStatus ask for the information of postprocess

Status return the infomation of pre or postprocess depending of where are now, in pre or post mode.

Example:

in: `GiD_Info list_entities PreStatus`

out:

Project name: UNNAMED

Problem type: UNKNOWN

Changes to save(0/1): 1

Necessary to mesh again (0/1): 1

Using LAYER: NONE

Interval 1 of 1 intervals

Degree of elements is: Normal

Using now mode(geometry/mesh): geometry

number of points: 6

number of points with 2 higher entities: 6

number of points with 0 conditions: 6

number of lines: 6

number of lines with 1 higher entities: 6

number of lines with 0 conditions: 6

number of surfaces: 1

number of surfaces with 0 higher entities: 1

number of surfaces with 0 conditions: 1

number of volumes: 0

number of nodes: 8

number of nodes with 0 conditions: 8

number of Triangle elements: 6

number of elements with 0 conditions: 6

Total number of elements: 6

Last size used for meshing: 10

Internal information:

Total MeshingData:0 Active: 0 0%

- **GiD_Info list_entities**

This command returns information about entities.

It has the following arguments:

- **Points / Lines / Surfaces / Volumes / Dimensions / Nodes / Elements / Results***: Type of entity or Results.* **Note:** *If the user is postprocessing in GiD, the information returned by* **Nodes/Elements** *concerns the nodes and elements in postprocess, including its results information. To access preprocess information about the preprocess mesh, the following entity keywords should be used:* **PreNodes/PreElements***.*
- **entity_id***: The number of an entity. It is also possible to enter a list of entities (e.g.: 2 3 6 45), a range of entities (e.g.: entities from 3 to 45, would be 3:45) or a layer (e.g.: layer:layer_name).*

Using "list_entities Results" you must also specify <analysis_name> <step> <result_name> <indexes>With the option **-more,** *more information is returned about the entity. The* **-more** *option used with lines returns the length of the line, its radius (arcs), and the list of surfaces which are higher entities of that line; used with elements it returns the type of element, its number of nodes and its volume.*

Example 1:

in: `GiD_Info list_entities Points 2 1`

out:

POINT

Num: 2 HigherEntity: 1 conditions: 0 material: 0

LAYER: car_lines

Coord: -11.767595 -2.403779 0.000000

END POINT

POINT

Num: 1 HigherEntity: 1 conditions: 0 material: 0

LAYER: car_lines

Coord: -13.514935 2.563781 0.000000

END POINT

Example 2:

in: `GiD_Info list_entities lines layer:car_lines`

out:

STLINE

Num: 1 HigherEntity: 0 conditions: 0 material: 0

LAYER: car_lines

Points: 1 2

END STLINE

STLINE

Num: 13 HigherEntity: 0 conditions: 0 material: 0

LAYER: car_lines

Points: 13 14

END STLINE

Example 3 (using **-more***):*

in: `GiD_Info list_entities -more Lines 2`

out:

STLINE

Num: 2 HigherEntity: 2 conditions: 0 material: 0

LAYER: Layer0

Points: 2 3

END STLINE

LINE (more)

Length=3.1848 Radius=100000

Higher entities surfaces: 1 3

END LINE

7.3.20 parametric

GiD_Info parametric

This command returns geometric information (coordinates, derivates, etc.) about parametric lines or surfaces.

For lines it has the following syntax:

GiD_Info parametric line entity_id coord | deriv_t | deriv_tt | t_fromcoord | t_fromrelativelength | length_to_t | t | x y z

And for surfaces:

GiD_Info parametric surface entity_id coord | deriv_u | deriv_v | deriv_uu | deriv_vv | deriv_uv | normal | uv_fromcoord | maincurvatures u v | x y z

The result for each argument is:

- **line|surface***: Type of entity.*
- **entity_id***: The number of an entity.*
- **coord***: 3D coordinate of the point with parameter t (line) or u,v (surface).*
- **deriv_t***: First curve derivative at parameter t.*
- **deriv_tt***: Second curve derivative at parameter t.*
- **t_fromcoord***: t parameter for a 3D point.*
- **t_fromrelativelength***: parameter corresponding to a relative (from 0 to 1) arc length t*
- **length_to_t***: lenght of the curve until the parameter t (if t=1.0 then it is the total lengh)*
- **deriv_u,deriv_v***: First partial u or v surface derivatives.*
- **deriv_uu,deriv_vv,deriv_uv***: Second surface partial derivatives.*
- **normal***: Unitary surface normal at u,v parameters.*
- **uv_fromcoord***: u,v space parameters for a 3D point.*
- **maincurvatures***: return a list with 8 numbers: v1x v1y v1z v2x v2y v2z c1 c2*

  v1x v1y v1z : first main curvature vector direction (normalized)

  v2x v2y v2z : second main curvature vector direction (normalized)

  c1 c2: main curvature values

Note: *The vector derivatives are not normalized.*

Example:

in: `GiD_Info parametric line 26 deriv_t 0.25`

out: `8.060864 -1.463980 0.000000`

7.3.21 check

GiD_Info check

This command returns some specialized entities check.

For lines it has the following syntax:

GiD_Info check line <entity_id> isclosed

For surfaces:

GiD_Info check surface <entity_id> isclosed | isdegeneratedboundary | selfintersection

And for volumes:

GiD_Info check volume <entity_id> orientation | boundaryclose

The result for each argument is:

- **line | surface | volume***: Type of entity.*
- **<entity_id>***: The number of an entity.*
- **isclosed***: For lines: 1 if start point is equal to end point, 0 otherwise. For surfaces: A surface is closed if its coordinate curves (of the full underlying surface) with parameter 0 and 1 are equal. It returns a bit encoded combination of closed directions: 0 if it is not closed, 1 if it is closed in u, 2 if it is closed in v, 3 if it is closed in both u and v directions.*
- **isdegeneratedboundary***: A surface is degenerated if some boundary in parameter space (south, east, north or west) becomes a point in 3d space. It returns a bit encoded combination of degenerated boundaries, for example: 0 if it is not degenerated, 9=2^0+2^3 if south and west boundaries are degenerated.*
- **selfintersection***: Intersections check between surface boundary lines. It returns a list of detected intersections. Each item contains the two line numbers and their parameter values.*
- **orientation***: For volumes, it returns a two-item list. The first item is the number of bad oriented volume surfaces, and the second item is a list of these surfaces' numbers.*
- **boundaryclose***: For volumes, a boundary is topologically closed if each line is shared by two volume surfaces. It returns 0 if it is not closed and must be corrected, or 1 if it is closed.*

Example:

in: GiD_Info check volume 5 orientation

out: 2 {4 38}

7.3.22 listmassproperties

GiD_Info ListMassProperties

This command returns properties of the selected entities.

It returns the **length** *if entities are lines,* **area** *if surfaces,* **volume** *if volumes, or the* **center of gravity** *if entities are nodes or elements. It has the following arguments:*

- **Points/Lines/Surfaces/Volumes/Nodes/Elements***: Type of entity.*
- **entity_id***: The number of an entity. It is also possible to enter a list of entities (e.g.: 2 3 6 45) or a range of entities (e.g.: entities from 3 to 45, would be 3:45).*

Example:

in: `GiD_Info ListMassProperties Lines 13 15`

out:

LINES

n. Length

13 9.876855

15 9.913899

Selected 2 figures

_____

Total Length=19.790754

7.3.23 problemtypepath

GiD_Info problemtypepath

This command returns the absolute path to the current problem type.

7.3.24 gidversion

- **GiD_Info GiDVersion**

This command returns the GiD version number. For example 10.0.8

7.3.25 view

GiD_Info view

This command returns the current view parameters. Something like:

{x -13.41030216217041 13.41030216217041} {y 10.724431991577148 -10.724431991577148} {z -30.0 30.0} {e 10.0} {v 0.0 0.0 0.0} {r 1.0}

{m 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0} {c 0.0 0.0 0.0} {pd 0.0} {pno 0.0} {pfo 0.0} {pf 4.0} {pv 0.0} {NowUse 0}

{DrawingType 0} {LightVector 90.0 90.0 150.0 0.0}

See VIEW>View entry>Save/Read View of Reference Manual for a brief explanation of this parameters

7.3.26 ispointinside

GiD_Info IsPointInside

GiD_Info IsPointInside ?-tolerance <tol>? Line|Surface|Volume <num> {<x> <y> <z>}

This commands returns 1 if the point {x y z} is inside the specified volume/surface/curve, or 0 if lies outside.

7.4 Special Tcl commands

GiD add to the standard Tcl/Tk keywords some extra commands, to do specific thinks.

7.4.1 Geometry

GiD_Geometry create|delete|get|list point|line|surface|volume <num>|append <data>

To create, delete, get data or list the identifiers of geometric entities:

- **<num>|append**: <num> is the entity identifier (integer > 0). You can use the word 'append' to set a new number automatically.
- **<data>**: is all the geometric definition data (**create**) or a selection specification (**delete**, **get** or **list**):

create: to make new geometric entities

- GiD_Geometry create volume <num> | append layer numsurfaces {surface1 verso1} ... ?contactvolume <transformation_matrix>?

for contactvolume is necessary to specify the <transformation_matrix> : a vector of 16 reals representing a 4x4 transformation matrix that maps surface1 into surface2

- GiD_Geometry create surface <num> | append plsurface | nurbssurface | coonsurface | meshsurface | contactsurface layer numlines ?<nurbs_data>? {line1 verso1} ... <geometrical_data>

<nurbs_data> must be provided only for NURBS surfaces and are this variables:

u_degree v_degree numpoints_u numpoints_v istrimmed isrational

<geometrical data> depends of each entity type (see get command)

- GiD_Geometry create line <num>|append nurbsline layer inipoint endpoint degree numpoints isrational {point1_x point1_y point1_z ?point1_w?} ... knot_1 ...

Instead the NURBS parameters is possible to create a curve that interpolates a list of points (also tangents at start and end can be specified)

GiD_Geometry create line <num> | append nurbsline layer inipoint endpoint -interpolate numpoints {p1_x p1_y p1_z} ... {pn_x pn_y pn_z} ?-tangents {t0_x t0_y t0_z} {t1_x t1_y t1_z}?

- GiD_Geometry create line <num>|append stline layer inipoint endpoint
- GiD_Geometry create point <num>|append layer point_x point_y point_z

delete: to erase model entities

- GiD_Geometry delete point|line|surface|volume <args> with <args>: num numa:numb numa: layer:layer_name

get: to obtain all the geometrical data to define a single entity

- GiD_Geometry get point|line|surface|volume <args> with <args>: num
  - GiD_Geometry get point will return:

<layer> <geometrical data>

<layer> is the layer name

<geometrical data> the coordinates x y z

  - GiD_Geometry get line will return:

<type> <layer> <p1> <p2> <geometrical data>

<type> can be: stline, nurbsline, arcline, polyline

<layer> is the layer name

<p1> identifier of start point

<p2> identifier of end point

<geometrical data> depends of each entity type

stline: nothing

nurbsline: <d> <n> <rat> {x y z ?w?}1 ... {x y z ?w?}n <k0> ... <kn+d>

<d>degree

<n>number of control points

<rat> 1 if rational, else 0

{xi yi zi ?wi?} control points coordinates. If rational wi is the weight

<ki> knots

arcline: {xc yc} <r> <sa> <ea> {m11 ... m44}

{xc yc} 2D center

<r> radius

<sa> start angle (rad)

<ea> end angle (rad)

{m11 ... m44} transformation 4x4 matrix (the identity for a 2D case)

m11 ...m33 is a rotation 3x3 matrix

m14 ...m34 is a translation 3x1 vector

m44 is an scale factor

m41 ... m43 must be 0

polyline: not implemented

- GiD_Geometry get surface will return:

<type> <layer> <nl> ?<nurbs_data>? {l1 o1} ... {lnl onl} <geometrical data>

<type> can be: nurbssurface plsurface coonsurface meshsurface

<layer> is the layer name

<nl> number of boundary lines (including holes)

<nurbs_data> only for NURBS surfaces (<du> <dv> <nu> <nv> <istrimmed> <isrational>)

{li oi} identifier of line and its orientation for the surface (1 if opposite to the line advance, 0 else)

Note: turning left of a line with orientation 0 we go inside the surface.

<geometrical data> depends of each entity type

plsurface: nothing

coonsurface: nothing

nurbssurface {x y z ?w?}1 ... {x y z ?w?}n_uxn_v <ku_0> ... <ku_{nu+du}> <kv_0> ...

<kv_{nv+dv}>

<du> <dv>degree in u, v direction

<nu> <nv>number of control points in each direction

<ratu> <ratv> 1 if rational, 0 else

{x_i y_i z_i ?w_i?} control points coordinates. If rational w_i is the weight

<ku_i> <kv_i> knots in each direction

meshsurface: nn ne nnode {x_1 y_1 z_1 ... x_{nn} y_{nn} z_{nn}} {a_1 b_1 c_1 ?d_1? ... a_{ne} b_{ne} c_{ne} ?d_{ne}?}

nn: number of nodes

ne: number or elements (triangles or quadrilaterals)

nnode: number of nodes by element: 3 or 4

x_i y_i z_i: coordinates

a_i b_i c_i d_i: connectivities (d_i only for quadrilaterals)

- GiD_Geometry get volume will return:

<layer> <ns> {s1 o1} ... {snl onl}

<layer> is the layer name

<ns> number of boundary surfaces (including holes)

{si oi} identifier of surface and its orientation for the volume (1 if opposite to the surface normal, 0 else)

Note: the normal of a surface with orientation 0 points inside the volume

list: to get a list of entity identifiers of a range or inside some layer

- GiD_Geometry list point | line | surface | volume <args>

with <args>: <num> | <num_min>:<num_max> | <num_min>:end

layer:<layer_name>

unrendered (only valid for surface)

Examples:

Creation of a new NURBS surface:

```
GiD_Geometry create surface 1 nurbssurface Layer0 4 1 1 2 2 0 0 {1 1} {4 1} {3 1} {2 1} \
  {0.17799 6.860841 0.0} {-8.43042200 6.86084199 0.0} {0.17799400 0.938510 0.0} \
    {-8.43042 0.938510 0.0} 0.0 0.0 1.0 1.0 0.0 0.0 1.0 1.0
```

Get the list of points of the layer named 'layer_name':

```
GiD_Geometry list point layer:layer_name
```

Get the list of problematic surfaces that couldn't be rendered:

```
GiD_Geometry list surface unrendered
```

7.4.2 Mesh

GiD_Mesh create|delete|edit|get node|element <num>|append <elemtype> <nnode> <N1 ... Nnnode> <radius> <nx> <ny> <nz> ?<matname>? | <x y z>

To create, delete, modify or know information about mesh nodes or elements of the preprocess:

- **<num>|append**: <num> is the identifier (integer > 0) for the node or element. You can use the word 'append' to set a new number automatically. The number of the created, deleted or modified entity is returned as the result. When deleting, it is possible to use a list of entities;
- **<elemtype>**: must be one of "point | linear | triangle | quadrilateral | tetrahedra | hexahedra | prism | pyramid | sphere | circle"
- **<nnode>** is the number of nodes an element has
- **<N1 ... Nnnode>** is a Tcl list with the element connectivities
- **<radius>** is the element radius, only for sphere and circle elements
- **<nx> <ny> <nz>** is the normal of the plane that contain the circle, must be specified for circle elements only
- **<matname>** is the optional element material name
- **<x y z>** are the node coordinates. If the z coordinate is missing, it is set to z=0.0.

Examples:

```
GiD_Mesh create node append {1.5 3.4e2 6.0}
```

```
GiD_Mesh create element 58 triangle 3 {7 15 2} steel
```

```
GiD_Mesh delete element {58 60}
```

GiD_Cartesian get|set ngridpoints|boxsize|corner|dimension|coordinates|iscartesian <values>

To get and set cartesian grid properties

- **ngridpoints**: the number of values of the grid axis on each direction x, y,z (3 integers)
- **boxsize**: the size of the box of the grid on each direction (3 reals)
- **corner**: the location of the lower-left corner of the grid box (3 reals)
- **dimension**: the dimension of the grid: 2 for 2D or 3 for 3D
- **coordinates**: the list of grid coordinates on each direction (nx+ny+nz reals)
- **iscartesian**: return 1 ir current mesh is cartesian, 0 else.

GiD_MeshPost create <meshname> <elemtype> <elementnnodes> ?-zero_based_array? <node_ids> <nodes> <element_ids> <elements> ?<radius+?normals?>? ?<r g b a>?

To create a postprocess mesh.

This command create all mesh nodes and elements in a single step (unlike GiD_Mesh that create each node or element one by one)

- **<meshname>**: the name of the mesh
- **<elemtype>**: must be one of "point | linear | triangle | quadrilateral | tetrahedra | hexahedra | prism | pyramid | sphere | circle"
- **<elementnnodes>**: is the number of nodes an element has. All elements of the mesh must have the same number of nodes.
- **-zero_based_array**: optional flag. By default node and element indexes start from 1, but setting this flag indexes must start from 0.
- **<node_ids>**: list of node indentifiers. If it is an empty list them numeration is implicitly increasing.
- **<nodes>:** a list of real numbers with the thee coordinates of each node {x_0 y_0 z_0 ... x_{nn-1} y_{nn-1} z_{nn-1}}
- **<element_ids>**: list of element indentifiers. If it is an empty list them numeration is implicitly increasing.
- **<elements>:** a list of integers with the <elementnnodes> nodes of each element: the id of each node is the location on the vector of nodes, starting from 0
- **<radius+normals>:**
  - **<radius>**:only for spheres. Is a list of reals with the radius or each sphere {r_0 ... r_{ne-1}}
  - **<normals>**:only for circles. Is a list of reals with the radius and normal to the plane or each circle {r_0 nx_0 ny_0 nz_0 ... r_{ne-1} nx_{ne-1} ny_{ne-1} nz_{ne-1}}
- **<r g b a>**: optional color components, to set the mesh color. r g b a are the red, green, blue and alpha transparency components of the color, must be real numbers from 0.0 to 1.0. If the color is not specified, an automatic color will be set.

7.4.3 Data

GiD-Tcl special commands to manage materials, conditions, intervals, general data or local axes:

GiD_CreateData create|delete material ?<basename>? <name> ?<values>?

To create or delete materials:

- **<basename>** this only applies to the **create** operation, and is the base material from which the new material is derived;
- **<name>** is the name of material itself;
- **<values>** is a list of all field values for the new material.

Example:

```
GiD_CreateData create material Steel Aluminium {3.5 4 0.2}

GiD_CreateData delete material Aluminium
```

GiD_AssignData material|condition <name> <over> ?<values>? <entities>

To assign materials or conditions over entities:

- **<name>** is the name of the material or condition;

- **<over>** must be: points, lines, surfaces, volumes, layers, nodes, elements, body_elements, or face_elements (elements is equivalent to body_elements);
- **<newvalues>** is only required for conditions. If it is set to "" then the default values are used;
- **<entities>** a list of entities (it is valid to use ranges as a:b ,can use "all" to select everything, "end" to specify the last entity, layer:<layername> to select the entities in this layer) ; if <over> is face_elements then you must specify a list of "entitynumface" instead just "entity".

Example:

```
GiD_AssignData materials Steel Surface {2:end}
GiD_AssignData condition Point-Load Nodes {3.5 2.1 8.0} all
GiD_AssignData condition Face-Load face_elements {3.5 2.1 8.0} {15 1 18 1 20 2}
```

GiD_UnAssignData material|condition <name> <over> <entities> ?wherefield <fieldname> <fieldvalue>?

To unassign materials or conditions of some entities:

- **<name>** is the name of the material or condition; Can use "*" to match all materials
- **<over>** must be: points, lines, surfaces, volumes, layers, nodes, elements, body_elements, or face_elements (elements is equivalent to body_elements);
- **<entities>** a list of entities (it is valid to use ranges as a:b ,can use "all" to select everything, "end" to specify the last entity, layer:<layername> to select the entities in this layer) ; if <over> is face_elements then you must specify a list of "entitynumface" instead just "entity".
- **wherefield <fieldname> <fieldvalue>** To unassign this condition olny for the entities where the field named 'fieldname' has the value 'fieldvalue'

Example:

```
GiD_UnAssignData materials * Surface {end-5:end}
GiD_UnAssignData condition Point-Load Nodes layer:Layer0
GiD_UnAssignData condition Face-Load face_elements {15 1 18 1 20 2}
```

GiD_ModifyData materials|intvdata|gendata ?<name>? <values>

To change all field values of materials, interval data or general data:

- **<name>** is the material name or interval number;
- **<values>** is a list of all the new field values for the material, interval data or general data.

Example:

```
GiD_ModifyData materials Steel {2.1e6 0.3 7800}
GiD_ModifyData intvdata 1 ...
GiD_ModifyData gendata ...
```

GiD_AccessValue set|get materials|conditions|intvdata|gendata ?<name>? <question> ?<attribute>? <value>

To change only some field values of materials, interval data or general data:

- **<name>** is the material, condition name or interval number (not necessary for gendata);
- **<question>** is a field name;
- **<attribute>** is the attribute name to be changed (STATE, HELP, etc.) instead of the field value;
- **<value>** is the new field or attribute value.

Example:

```
GiD_AccessValue set gendat Solver Direct
```

GiD_IntervalData <mode> <number>|?copyconditions?

To create, delete or set interval data;

- **<mode>** must be 'create', 'delete' or 'set';
- **<number>** is the interval number (integer >=1). **Create** returns the number of the newly created interval and can optionally use 'copyconditions' to copy to the new interval the conditions of the current one.

For **set** mode, if <number> is not supplied, the current interval number is returned.

Example:

```
set current [GiD_IntervalData set]

GiD_IntervalData set 2

set newnum [GiD_IntervalData create]

set newnum [GiD_IntervalData create copyconditions]
```

GiD_LocalAxes <mode> <name> ?<type>? <Cx Cy Cz> <PAxex PAxey PAxez> <PPlanex PPlaney PPlanez>?

To create, delete or modify local axes:

- **<mode>**: must be one of "create|delete|edit|exists", which correspond to the operations: create, delete, edit or exists;
- **<name>**: is the name of local axes to be created or deleted;
- **<type>**: must be one of "rectangular|cylindrical|spherical C_XZ_Z|C_XY_X". Currently, GiD only supports rectangular axes. **C_XZ_Z** is an optional word to specify one point over the XZ plane and another over the Z axis (default). **C_XY_X** is an optional word to specify one point over the XY plane and another over the X axis;
- **<Cx Cy Cz>** is a Tcl list with the real coordinates of the local axes origin;
- **<PAxex PAxey PAxez>** is a Tcl list with the coordinates of a point located over the Z' local axis (where Z' is positive). The coordinates must be separated by a space. If the z coordinate is missing, it is set to z=0.0;
- **<PPlanex PPlaney PPlanez>** is a Tcl list with the coordinates of a point located over the Z'X'half-plane (where X' is positive).

For the 'exists' operation, if only the <name> field is specified, 1 is returned if this name exists, and 0 if it does not. If the other values are also apecified, <name> is ignored.

The value returned is:

-1 if the global axes match;

-2 if the automatic local axes match;

-3 if the automatic alternative local axes match;

0 if it does not match with any axes;

<n> if the user-defined number <n> (n>0) local axes match.

Example:

```
GiD_LocalAxes create "axes_1" rectangular C_XY_X {0 0 0} {0 1 0} {1 0 0}
GiD_LocalAxes delete axes_1
GiD_LocalAxes exists axes_1
GiD_LocalAxes exists "" rectangular C_XY_X {0 0 0} {0 1 0} {1 0 0}
```

this last sample returns -1 (equivalent to global axis)

7.4.4 Results

GiD_Result create|delete|get|get_nodes|gauss_point|result_ranges_table ?-array? <data>

To create, delete or get postprocess results:

- **GiD_Result create ?-array? {Result header} ?{Unit <unit_name>}? ?{componentNames name1 ...}? {entity_id scalar|vector|matrix_values} {...} {...}** : these creation parameters are the same as for the postprocess results format (see Result -pag. 85- of Postprocess results format: ProjectName.post.res -pag. 78-) where each line is passed as Tcl list argument of this command;

  Optionally the names of the result's components could be specified, with the componentNames item, and the unit label of the result with the Unit item

  if the -array flag is used (recommended for efficiency), then the syntax of the data changes. Instead to multiple items {id1 vx1 vy1 ...} ... {idn vxn vyn} a single item with sublists is required, {{id1 ... idn} {{vx1...vxn} {vy1...vyn}}}, where idi are the integers of the node or element where the result are defined, and vi are the real values. The amount of values depends on the type of result: 1 for Scalar, 2 for ComplexScalar, 3 for Vector (4 if signed modulus is provided), 6 for Matrix.

- **GiD_Result delete {Result_name result_analysis step_value}** : deletes one result;
- **GiD_Result get [-max | -min | -compmax | -compmin | -info -array] {Result_name result_analysis step_value}** : retrieves the results value list of the specified result; or if one of the **-max**, **-min**, **-compmax**, **-compmin**, or **-info** flags was specified: the minimum/maximum value of the result, every minimum/maximum of the components of the result, or the header information of the

result (with type and location) is retrieved, respectively;

- **GiD_Result get_nodes**: returns a list of nodes and their coordinates.
- **GiD_Result gauss_point create|get|names|delete <name> <elemtype> <npoint> ?-nodes_included? <coordinates> ?<mesh_name>?**
  - create <name> <elemtype> <npoint> ?-nodes_included? <coordinates> ?<mesh_name>?

  Define a new kind of gauss point where element results could be related.

  <name> is the gauss point name. Internal Gauss points are implicitly defined, and its key names (GP_LINEAR_1,GP_TRIANGLE_1,...) are reserved words and can't be used to create new gauss points or be deleted. (seeGauss Points -pag. 79-)

  <elemtype> must be one of "point | linear | triangle | quadrilateral | tetrahedra | hexahedra | prism | pyramid | sphere | circle". (see Postprocess mesh format: ProjectName.post.msh -pag. 98-)

  <npoint> number of gauss points of the element

  -nodes_included :optional word, only for line elements, to specify that start and end points are considered (by default are not included)

  <coordinates> : vector with the local coordinates to place the gauss points: 2 coordinates by node for surface elements, 3 coordinates for volume elements. For line elements now is not possible to specify its coordinates, the n points will be equispaced.

  <mesh_name>: optional mesh name where this definition is applied, by default it is applied to all meshes

  - get <name>

  Return the information of this gauss point

  - names

  Return a list with the names of all gauss points defined

  - delete <name>

- **GiD_Result result_ranges_table create|get|names|delete <name> {<min1> <max1> <label1> ... <minn> <maxn> <labeln> }**
  - create <name> {<label1> <min1> <max1> ... <labeln> <minn> <maxn>}

  Define a new kind of result ranges table to map ranges of result values to labels.

  <name> is the result ranges table name.

  <min_i> <max_i> <$label_i$>: is the label to show for result values from min to max

  - get <name>

Return the information of this result ranges table

- names

Return a list with the names of all result ranges tables defined

- delete <name>

-array flag can be specified, for create and get subcommands, to use list of vectors to handle the information in a more efficient way

Examples:

```
GiD_Result create -array {Result "MyVecNodal" "Load analysis" 10 Vector OnNodes} {ComponentNames "Vx" "Vy" "Vz" "|velocity|"} {{1 3} {{2.0e-1 -3.5e-1} {2.0e-1 4.5e-1} {0.4 -2.1}}}

GiD_Result create {Result "Res Nodal 1" "Load analysis" 1.0 Scalar OnNodes} {1 2} {2 2} {113 2} {3 5} {112 4}

GiD_Result gauss_point create GPT1 Quadrilateral 1 {0.5 0.5}

GiD_Result create {Result "Res Gauss 1" "Load analysis" 1.0 Scalar OnGaussPoints GPT1} {165 2} {2} {3} {164 5} {4} {3}

GiD_Result get {"Res Nodal 1" "Load analysis" 4}

GiD_Result delete {"Res Nodal 1" "Load analysis" 4}

GiD_Result create {Result "Res Nodal 2" "Load analysis" 4 Vector OnNodes} {ComponentNames "x comp" "y comp" "z comp" "modulus"} {1 0.3 0.5 0.1 0.591} {2 2.5 0.8 -0.3 2.641}

GiD_Result create -array {Result "Res Nodal 2" "Load analysis" 4 Vector OnNodes} {ComponentNames "x comp" "y comp" "z comp" "modulus"} {{1 2} {{0.3 2.5} {0.5 0.8} {0.1 -0.3} {0.591 2.641}}}
```

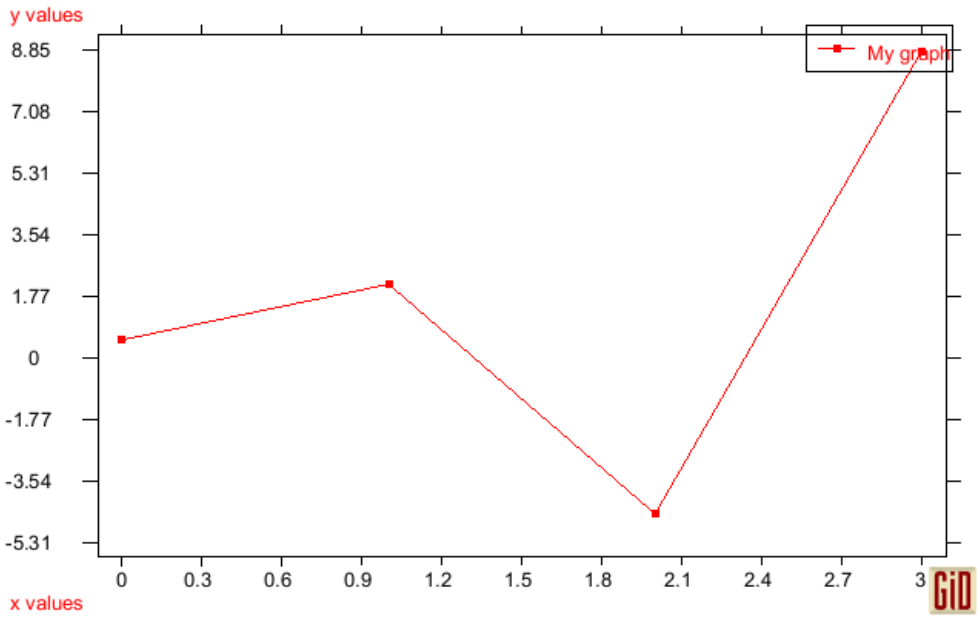
7.4.5 Graphs

GiD_Graph list|show|hide|clear|get|delete|create

To create, delete or get postprocess graphs:

- **list** : gets a list of the existent graphs, an empty list if there is no graph;
- **show** : switches the graphic view and shows the graphs;
- **hide** : hides the graphs and switches back to mesh view;
- **clear** : delete all graphs in GiD;
- **get <graph_name>** : gets a list with the values of the graph with name "graph_name", the values are the same used to create a graph: <label_x> <label_y> <x_values> <y_values>
- **delete <graph_name>** : deletes the graph "graph_name" causing an error if does not exists;
- **create <graph_name> <label_x> <label_y> <x_values> <y_values> <x_unit> <y_unit>** : creates the graph "graph_name" with the provided information, causing an error if the graph already exists: for instance the graph of the picture was created with

  ```
  GiD_Graph create "My graph" "x values" "y values" {0 1 2 3} {0.5 2.1 -4.5 8.8} "" ""
  ```



7.4.6 OpenGL

GiD_OpenGL

It is possible to use OpenGL commands directly from GiD-Tcl by using the command "GiD_OpenGL draw". For example, for C/C++ use:

```
glBegin(GL_LINES);
glVertex(x1,y1,z1);
glVertex(x2,y2,z2);
glEnd();
```

for GiD-Tcl use:

```
GiD_OpenGL draw -begin lines
GiD_OpenGL draw -vertex [list $x1 $y1 $z1]
GiD_OpenGL draw -vertex [list $x2 $y2 $z2]
GiD_OpenGL draw -end
```

The standard syntax must be changed according to these rules: - OpenGL constants: "GL" prefix and underscore character '_' must be removed; the command must be written in lowercase.

Example:

```
GL_COLOR_MATERIAL -> colormaterial
```

- OpenGL functions: "GL" prefix must be removed and the command written in lowercase. Pass parameters as list, without using parentheses ()

Example:

```
glBegin(GL_LINES) -> glbegin lines
```

The subcommand "GiD_OpenGL draw" provides access to standard OpenGL commands, but other "GiD_OpenGL" special GiD subcommands also exist:

- **register <tclfunc>** Register a Tcl procedure to be invoked automatically when redrawing the scene. It returns a handle to unregister.

Example:

```
proc MyRedrawProcedure { } { ...body... }
set id [GiD_OpenGL register MyRedrawProcedure]
```

- **unregister <handle>** Unregister a procedure previously registered with **register**.

Example:

```
GiD_OpenGL unregister $id
```

- **registercondition <tclfunc> <cond>** Register a Tcl procedure to be invoked automatically when redrawing the specified condition. It returns a handle to unregister.
- **unregistercondition <cond>** Unregister a procedure previously registered with **registercondition**.
- **draw <-cmd args -cmd args>** This is the most important subcommand, it calls standard OpenGL commands. See the list of supported OpenGL functions.
- **drawtext <text>** Draw a text more easily than using standard OpenGL commands (draw in the current 2D location, see rasterpos OpenGL command).

Example:

```
GiD_OpenGL draw -rasterpos [list $x $y $z]
GiD_OpenGL drawtext "hello world"
```

- **font push <font_name font_size> | pop | measure <text> | current | metrics ?-ascent|-descent|-linespace|-fixed?**

push sets the current OpenGL font, pop restores the previous one

measure <text> returns the amount of space in pixels to display this <text>

current returns a list with the current font name and size

metrics returns a list with current font metrics information: -ascent -descent and -linespace in pixels, -fixed is 1 if all characters have equal size

Example:

```
GiD_OpenGL font push {"Times New Roman" 18}
set with [GiD_OpenGL measure "hello world"]
GiD_OpenGL drawtext "hello world"
GiD_OpenGL pop
```

- **drawentity ?-mode normal | filled? point | line | surface | volume | node |**

element | dimension <id list> To draw an internal GiD preprocess entity.

Example:

```
GiD_OpenGL drawentity -mode filled surface "1 5 6"
```

- **project <x y z>** Given three world coordinates, this returns the corresponding three window coordinates.
- **unproject <x y z>** Given three window coordinates, this returns the corresponding three world coordinates.
- **doscrzoffset <boolean>** Special trick to avoid the lines on surfaces hidden by the surfaces.

List of supported OpenGL functions:

accum alphafunc begin blendfunc call calllist clear clearaccum clearcolor cleardepth clearstencil clipplane color colormask colormaterial copypixels cullface deletelists depthfunc depthmask dfactorBlendTable disable drawbuffer drawpixels edgeflag enable end endlist evalcoord1 evalcoord2 evalmesh1 evalmesh2 finish flush fog frontface frustum genlists getstring hint hintModeTable initnames light lightmodel linestipple linewidth loadidentity loadmatrix loadname lookat map1 map2 mapgrid1 mapgrid2 material matrixmode modeColorMatTable multmatrix newlist newListTable normal opStencilTable opStencilTable ortho perspective pickmatrix pixeltransfer pixelzoom pointsize polygonmode popattrib popmatrix popname pushattrib pushmatrix pushname rasterpos readbuffer readpixels rect rendermode rotate scale scissor selectbuffer shademodel stencilfunc stencilmask stencilop texcoord texenv texgen teximage1d teximage2d texparameter translate vertex viewport

List of special non OpenGL standard functions:

getselection

List of supported OpenGL constants:

accum accumbuffer accumbufferbit add alphatest always allattrib allattribbits ambient ambientanddiffuse autonormal aux0 aux1 aux2 aux3 back backleft backright blend bluebias bluescale ccw clamp clipplane0 clipplane1 clipplane2 clipplane3 clipplane4 clipplane5 colorbuffer colorbufferbit colorindex colormaterial compile compileandexecute constantattenuation cullface current currentbit cw decal decr depthbuffer depthbufferbit depthtest diffuse dither dstalpha dstcolor enable enablebit emission equal eval evalbit exp exp2 extensions eyelinear eyeplane feedback fill flat fog fogbit fogcolor fogdensity fogend fogmode fogstart front frontandback frontleft frontright gequal greater greenbias greenscale hint hintbit incr invert keep left lequal less light0 light1 light2 light3 light4 light5 light6 light7 lighting lightingbit lightmodelambient lightmodellocalviewer lightmodeltwoside line linebit linear linearattenuation lineloop lines

```
linesmooth linestipple linestrip list listbit load map1color4 map1normal
map1texturecoord1 map1texturecoord2 map1texturecoord3 map1texturecoord4
map1vertex3 map1vertex4 map2color4 map2normal map2texturecoord1
map2texturecoord2 map2texturecoord3 map2texturecoord4 map2vertex3
map2vertex4 modelview modulate mult nearest never none normalize notequal
objectlinear objectplane one oneminusdstalpha oneminusdstcolor
oneminussrcalpha oneminussrccolor packalignment packlsbfirst packrowlength
packskippixels packskiprows packswapbytes pixelmode pixelmodebit point
pointbit points polygon polygonbit polygonoffsetfill polygonstipple
polygonstipplebit position projection q quadraticattenuation quads quadstrip
r redbias redscale render renderer repeat replace return right s scissor
scissorbit select shininess smooth specular spheremap spotcutoff
spotdirecion spotexponent srcalpha srcalphasaturate srccolor stenciltest
stencilbuffer stencilbufferbit t texture texture1d texture2d texturebit
texturebordercolor textureenv textureenvcolor textureenvmode texturegenmode
texturegens texturegent texturemagfilter textureminfilter texturewraps
texturewrapt transform transformbit triangles trianglefan trianglestrip
unpackalignment unpacklsbfirst unpackrowlength unpackskippixels
unpackskiprows unpackswapbytes vendor version viewport viewportbit zero
```

You can find more information about standard OpenGL functions in a guide to OpenGL.

7.4.7 Other

GiD_Set <varname> ?<value>?

This command is used to set or get GiD variables. GiD variables can be found through the **Right buttons** menu under the option Utilities -> Variables:

- **<varname>** is the name of the variable;
- **<value>** if this is omitted, the current variable value is returned (analogous with 'GiD_Info variables <varname>').

Example:

```
GiD_Set CreateAlwaysNewPoint 1
```

GiD_SetModelName <name>

To change the current model name.

GiD_ModifiedFileFlag set|get ?<value>?

There is a GiD internal flag to indicate that the model has changed, and must be saved before exit.

With this command it is possible to set or get this flag value:

- **<value>** is only required for **set**: must be 0 (false), or 1 (true).

Example:

```
GiD_ModifiedFileFlag set 1
```

GiD_ModifiedFileFlag get

GiD_MustRemeshFlag set|get ?<value>?

There is a GiD internal flag to indicate that the geometry, conditions, etc. have changed, and that the mesh must be re-generated before calculations are performed.

With this command it is possible to set or get this flag value:

- **<value>** is only required for **set**: must be 0 (false), or 1 (true).

Example:

```
GiD_MustRemeshFlag set 1
GiD_MustRemeshFlag get
```

GiD_BackgroundImage get|set show|filename|location <values>

This command allow to get and set the background image properties

Valid set values are:

- **show**: 1 or 0
- **filename**:

the full filename of some valid GiD image format to be used as background image

or "", to release the current image

- **location**:

'fill' to fill the whole screen,

or six floating values for a real size image, to set the origin and x,y local axes: ox oy ix iy jx jy

GiD_RegisterExtensionProc <.extension> PRE|POST|PREPOST <procedure>

To register a Tcl procedure to be automatically called when dropping a file with this extension

e.g.

GiD_RegisterExtensionProc ".h5" PRE Amelet::ReadPre

GiD_RegisterPluginAddedMenuProc <procedure>

To register a Tcl procedure to be automatically called when re-creating all menus (e.g. when doing files new)

this procedure is responsible to add its own options to default menu.

e.g.

GiD_RegisterPluginAddedMenuProc Amelet::AddToMenu

Some special commands exist to control the redraw and wait state of GiD:

.central.s disable graphics 'value' The value 0/1 Enable/Disable Graphics (GiD does not redraw)

EXAMPLE to disable the redraw:

```
.central.s disable graphics 1
```

.central.s disable graphinput 'value' The value 0/1 Enable/Disable GraphInput (enable or disable peripherals: mouse, keyboard, ...)

EXAMPLE to disable the peripherals input:

```
.central.s disable graphinput 1
```

.central.s disable windows 'value' The value 0/1 Enable/Disable Windows (GiD displays, or not, windows which require interaction with the user)

EXAMPLE to disable the interaction windows:

```
.central.s disable windows 1
```

.central.s disable writebatch 'value' The value 0/1 Enable/Disable writting the batch file that records the commands send to be processed.

.central.s waitstate 'value' The value 0/1 Enable/Disable the Wait state (GiD displays a hourglass cursor in wait state)

EXAMPLE to set the state to wait:

```
.central.s waitstate 1
```

Usually these command are used jointly:

EXAMPLE

```
#deactivate redraws, etc wit a widget named $w
$w conf -cursor watch .central.s waitstate 1
update
.central.s disable graphics 1
.central.s disable windows 1
.central.s disable graphinput 1
...
#reactivate all and redraw
.central.s disable graphics 0
.central.s disable windows 0
.central.s disable graphinput 0
GiD_Redraw
$w conf -cursor ""
.central.s waitstate 0
```

Note: It is recommended for a Tcl developer to use the more 'user-friendly' procedures defined inside the file 'dev_kit.tcl' (located in the \scripts directory). For example, to

disable and enable redraws, you can use:

::GidUtils::DisableGraphics

::GidUtils::EnableGraphics

GiD_Thumbnail get [width height]

returns the byte stream of an downscaled view of the graphical window. The image is a downscaled from the current size to **width** x **height.** The parameters **widht** and **height** are optinal and by default the view is scaled to 192x144. The result of this command can be directly used by the Tk image command, like this:

label .l -image [image create photo -data [GiD_Thumbnail get]]

GiD_GetWorldCoord screen_x screen_y

given the screen coordinates (screen_x, screen_y) returns a list with sis coordinates:

```
{ x y z nx ny nz}
```

being

(x, y, z) the coordinates mapped into the world (model) of the screen coordinates,

(nx, ny, nz) the normal vector components of the world (model) pointing to the user.

the mapping screen --> world (model) is done by intersecting the line perpendicular to the screen, passing through the coordinates (screen_x, screen_y), with the plane parallel to the screen (in real, model, world) at the centre of the view / model. The returned normal is the normal of this plane.

7.5 HTML help support

Problem type developers can take advantage of the internal HTML browser if they wish to provide online help.

7.5.1 GiDCustomHelp

With GiD version 7.4 and later, problem type developers can take advantage of the new help format. It is essentially the same html content, but now with an enhanced look and structure. The GiDCustomHelp procedure below is how you can show help using the new format:

```
GiDCustomHelp ?args?
```

where args is a list of pairs option value. The valid options are:

- **-title** : specifies the title of the help window. By default it is "Help on <problem_type_name>".
- **-dir** : gives the path for the help content. If **-dir** is missing it defaults to "<ProblemType dir>/html". Multilingual content could be present; in such a case it is assumed that there is a directory for each language provided. If the current language is not found, language

'en' (for English) is tried. Finally, if 'en' is not found the value provided for **-dir** is assumed as the base directory for the help content.

- **-start** : is a path to an html link (and is relative to the value of **-dir**). For instance:

```
-start html-version

-start html-tutorials/tutorial_1
```

- **-report** : is a boolean value indicating if the window format is report. If **-report** is 1, no tree is shown and only the content pane is displayed.

7.5.1.1 HelpDirs

With HelpDirs we can specify which of the subdirectories will be internal nodes of the help tree. Moreover, we can specify labels for the nodes and a link to load when a particular node is clicked. The link is relative the node. For instance:

```
HelpDirs {html-version "GiD Help" "intro/intro.html"} \
               {html-customization "GiD Customization"} \
               {html-faq "Frequently Asked Questions"} \
               {html-tutorials "GiD Tutorials" "tutorials_toc.html"} \
               {html_whatsnew "What's New"}
```

7.5.1.2 Structure of the help content

Assuming that html has been chosen as the base directory for the multilingual help content, the following structure is possible:

```
html
    \__ en - English content
    \__ es - Spanish content
```

Each content will probably have a directory structure to organize the information. By default the help system builds a tree resembling the directory structure of the help content. In this way there will be an internal node for each subdirectory, and the html documents will be the terminal nodes of the tree.

You can also provide a help.conf configuration file in order to provide more information about the structure of the help. In a help file you can specify a table of contents (TocPage), help subdirectories (HelpDirs) and an index of topics (IndexPage).

7.5.1.3 TocPage

TocPage defines an html page as a table of contents for the current node (current directory). We have considered two ways of specifying a table of contents:

`<UL> <LI> ... </UL>` (default)

`<DT> <DL> ... </DT>`

The first is the one generated by texinfo.

For instance:

```
TocPage gid_toc.html
```

```
TocPage contents.ht DT
```

7.5.1.4 IndexPage

If we specify a topic index by IndexPage, we can take advantage of the search index. In IndexPage we can provide a set of html index pages along with the structure type of the index. The type of the index could be:

`<DIR> <LI> ... </DIR>` (default)

`<UL> <LI> ... </UL>` (only one level of <UL>)

The first is the one generated by texinfo.

For instance:

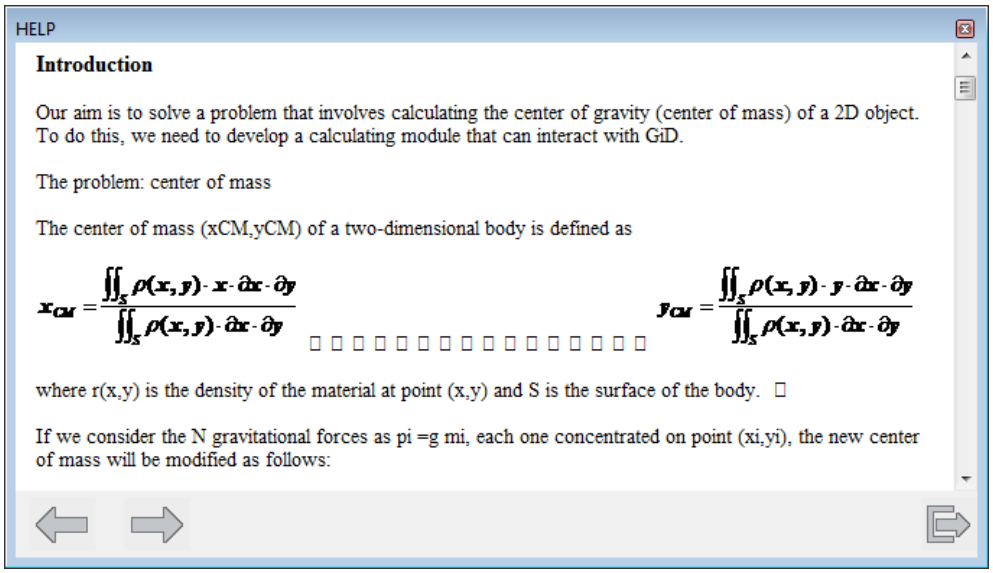
```
IndexPage html-version/gid_18.html html-faq/faq_11.html
```

7.5.2 HelpWindow

7.5.2.1 Create a directory named html inside your Problem Type directory

7.5.2.2 Call **HelpWindow "CUSTOM_HELP" "problem_type_name"**, where problem_type_name is the name of your problem type with the .gid extension (e.g. Examples/cmas2d.gid).

7.5.2.3 The function **HelpWindow** opens the file "index.html" which must be inside the html folder.



It is a good idea to call the function **HelpWindow "CUSTOM_HELP" "problem_type_name"** using the menu functions (see Managing menus -pag. 154-).

EXAMPLE: Adding a customized HTML help in the Help menu for the CMAS2D

problem type:

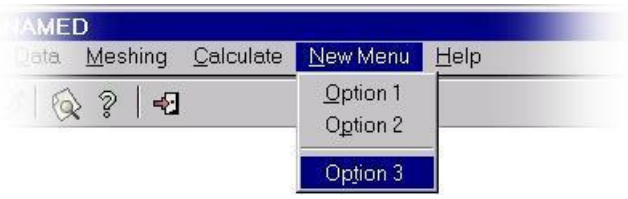
```
GiDMenu::InsertOption "Help" [list "Help CMAS2D"] 0 PREPOST {HelpWindow
"CUSTOM_HELP" "Examples/cmas2d.gid"} "" "" insert _

GiDMenu::UpdateMenus
```

Note: In order to test this example, must provide the html file:

'problemtypes/Examples/cmas2d.gid/html/index.html'

7.6 Managing menus

GiD offers you the opportunity to customize the **pull-down** menus. You can add new menus or to change the existing ones. If you are creating a problem type, these functions should be called from the InitGIDProject or InitGIDPostProcess functions (see TCL AND TK EXTENSION -pag. 111-).



Note: Menus and option menus are identified by their names.

Note: It is not necessary to restore the menus when leaving the problem type, GiD does this automatically.

The Tcl functions are:

- **GiDMenu::Create { menu_name_untranslated prepost {pos -1} {translationfunc _} }**

Creates a new menu. New menus are inserted between the **Calculate** and **Help** menus.

- menu_name_untranslated: text of the new menu (English).
- prepost can have these values:

  "PRE" to create the menu only in GiD Preprocess.

  "POST" to create the menu only in GiD Postprocess.

  "PREPOST" to create the menu in both Pre- and Postprocess.

- pos: optional, index where the new menu will be inserted (by default it is inserted before the 'Help' menu)
- translationfunc: optional, must be _ for GiD strings (default), or = for problemtype strings

- **GiDMenu::Delete { menu_name_untranslated prepost {translationfunc _} }**

Deletes a menu.

- menu_name_untranslated: text of the menu to be deleted (English).
- prepost can have these values:

"PRE" to delete the menu only in GiD Preprocess.

"POST" to delete the menu only in GiD Postprocess.

"PREPOST" to delete the menu in both Pre- and Postprocess.

- translationfunc: optional, must be _ for GiD strings (default), or = for problemtype strings

- **GiDMenu::InsertOption { menu_name_untranslated option_name_untranslated position prepost command {acceler ""} {icon ""} {ins_repl "replace"} {translationfunc _} }**

Creates a new option for a given menu in a given position (positions start at 0, the word 'end' can be used for the last one).

- menu_name_untranslated: text of the menu into which you wish to insert the new option (English), e.g "Utilities"
- option_name_untranslated: name of the new option (English) you want to insert.

The option name, is a menu sublevels sublevels list, like [list "List" "Points"]

If you wish to insert a separator line in the menu, put "---" as the option_name.

- position: position in the menu where the option is to be inserted. Note that positions start at 0, and separator lines also count.
- prepost: this argument can have the following values:

"PRE" to insert the option into GiD Preprocess menus.

"POST" to insert the option into GiD Postprocess menus.

"PREPOST" to insert the option into both Pre- and Postprocess menus.

- command: is the command called when the menu option is selected.
- acceler: optional, key accelerator, like "Control-s"
- icon: optional, name of a 16x16 pixels icon to show in the menu
- ins_repl: optional, if the argument is:
  - replace: (default) the new option replaces the option in the given position
  - insert: the new option is inserted before the given position.
  - insertafter: the new option is inserted after the given position.
- translationfunc: optional, must be _ for GiD strings (default), or = for problemtype strings

- **GiDMenu::RemoveOption {menu_name_untranslated option_name_untranslated prepost {translationfunc _}}**

Removes an option from a given menu.

- menu_name_untranslated: name of the menu (English) which contains the option you want to remove. e.g "Utilities"
- option_name_untranslated: name of the option (English) you want to remove. The option name, is a menu sublevels list, like [list "List" "Points"]
- prepost: this argument can have the following values:

"PRE" to insert the option into GiD Preprocess menus.

"POST" to insert the option into GiD Postprocess menus.

"PREPOST" to insert the option into both Pre- and Postprocess menus.

- translationfunc: optional, must be _ for GiD strings (default), or = for problemtype strings

To remove separators, the option_name is --- , but you can append an index (starting from 0) to specify wich separator must be removed, if there are more than one.

e.g.

```
GiDMenu::RemoveOption "Geometry" [list "Create" "---2"] PRE
```

- **GiDMenu::ModifyOption { menu_name_untranslated option_name_untranslated prepost new_option_name {new_command -default-} {new_acceler -default-} {new_icon -default-} {translationfunc _} }**

Edit an existent option from a given menu

some parameters can be '-default-' to keep the current value for the command, accelerator, etc

- **GiDMenu::UpdateMenus {}**

Updates changes made on menus. This function must be called when all calls to create, delete or modify menus are made.

- **GiD_RegisterPluginAddedMenuProc** and **GiD_UnRegisterPluginAddedMenuProc**

This commands can be used to specify a callback procedure name to be called to do some change to the original menus

GiD_RegisterPluginAddedMenuProc <procname>

GiD_UnRegisterPluginAddedMenuProc<procname>

The procedure prototype to be registered must not expect any parameter, something like this.

proc <procname> { } {

... do something ...

}

e.g. a plugin can modify a menu to add some entry, but this entry will be lost when GiD create again all menus, for example when starting a new model. Registering the procedure will be applied again when recreating menus.

- **GiD_RegisterExtensionProc** and **GiD_UnRegisterExtensionProc**

This tcl command must be used to register a procedure that is able to handle when using 'drag and drop' of a file on the GiD window.

It is possible to specify the extension (or a list of extensions) of the files to be handled, the mode PRE or POST where it will be handled, and the name of the callback procedure to be called.

GiD_RegisterExtensionProc <list of extensions> <prepost> <procname>

GiD_UnRegisterExtensionProc <list of extensions> <prepost>

<extension> is the file extension, preceded by a dot

<prepost> could be PRE or POST

The procedure prototype to be registered must expect a single parameter, the dropped file name, something like this.

proc <procname> { filename } {

... do something ...

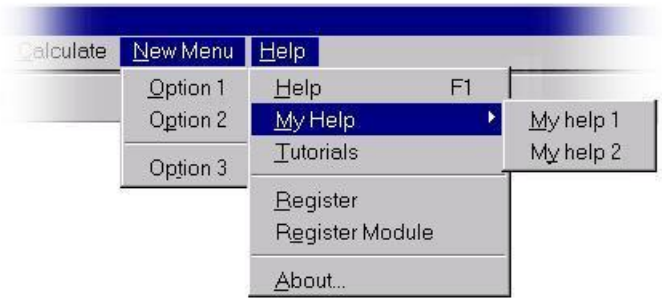
}

Example:

GiD_RegisterExtensionProc ".gif .png" PRE MyImageProcedure

EXAMPLE: creating and modifying menus

In this example we create a new menu called "New Menu" and we modify the GiD **Help** menu:



The code to make these changes would be:

```
GiDMenu::Create "New Menu" "PRE" -1 =
GiDMenu::InsertOption "New Menu" [list "Option 1"] 0 PRE "Command_1" "" ""
replace =
GiDMenu::InsertOption "New Menu" [list "Option 2"] 1 PRE "Command_2" "" ""
replace =
GiDMenu::InsertOption "New Menu" [list "---"] 2 PRE "" "" "" replace =
GiDMenu::InsertOption "New Menu" [list "Option 3"] 3 PRE "Command_3" "" ""
replace =

GiDMenu::InsertOption "Help" [list "My Help"] 1 PRE "" "" "" insert _
GiDMenu::InsertOption "Help" [list "My Help" "My help 1"] 0 PRE
"Command_help1" "" "" replace _
GiDMenu::InsertOption "Help" [list "My Help" "My help 2"] 1 PRE
"Command_help2" "" "" replace _

GiDMenu::RemoveOption "Help" [list "Customization Help"] PRE _
```

```
GiDMenu::RemoveOption "Help" [list "What is new"] PRE _
GiDMenu::RemoveOption "Help" [list "FAQ"] PRE _
GiDMenu::UpdateMenus
```

7.7 Custom Data Windows

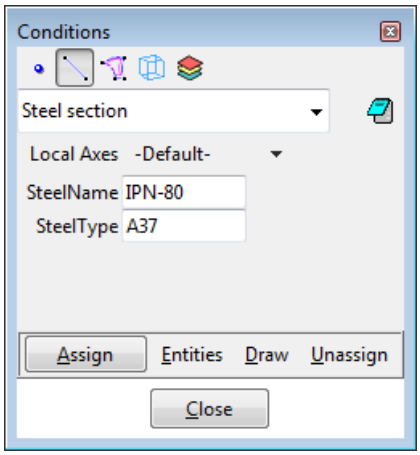
In this section the **Tcl/Tk** (scripted) customization of the look and feel of the data windows is shown. The layout of the properties drawn in the interior of any of the data windows - either Conditions, Materials, Interval Data or Problem Data - can be customized by a feature called TkWidget; moreover, the common behaviour of two specific data windows, Conditions and Materials, can be modified by a Tcl procedure provided for that purpose. This common behaviour includes, in the case of Materials for example, assigning/unassigning, drawing, geometry types, where to assign materials, creating/deleting materials, etc.

7.7.1 TkWidget

The problem type developer can change the way a **QUESTION** is displayed and if he wishes he can also change the whole contents of a window, while maintaining the basic behavior of the data set, i.e. in the Condition window: assign, unassign, draw; in the Material window: create material, delete material; and so on.

With the default layout for the data windows, the questions are placed one after another in one column inside a container frame, the **QUESTION**'s label in column zero and the **VALUE** in column one. For an example see picture below.

```
CONDITION: Steel_section
CONDTYPE: over lines
CONDMESHTYPE: over body elements
QUESTION: Local_Axes#LA#(-Default-,-Automatic-)
VALUE: -Default-
QUESTION: SteelName
VALUE: IPN-80
QUESTION: SteelType
VALUE: A37
END CONDITION
```



The developer can override this behavior using **TKWIDGET**. **TKWIDGET** is defined as an attribute of a **QUESTION** and the value associated with it must be the name of a Tcl procedure, normally implemented in a Tcl file for the problem type. This procedure will take care of drawing the **QUESTION**. A **TKWIDGET** may also draw the entire contents of the window and deal with some events related to the window and its data.

The prototype of a **TKWIDGET** procedure is as follow:

```
proc TKWidgetProc {event args} {
  switch $event {
    INIT {
       ...
    }
    SYNC {
       ...
    }
    DEPEND {
       ...
    }
    CLOSE {
       ...
    }
  }
}
```

The procedure should return:

- an empty string "" meaning that every thing was OK;
- a two-list element {ERROR-TYPE Description} where ERROR-TYPE could be ERROR or WARNING. ERROR means that something is wrong and the action should be aborted. If ERROR-TYPE is the WARNING then the action is not aborted but Description is shown as

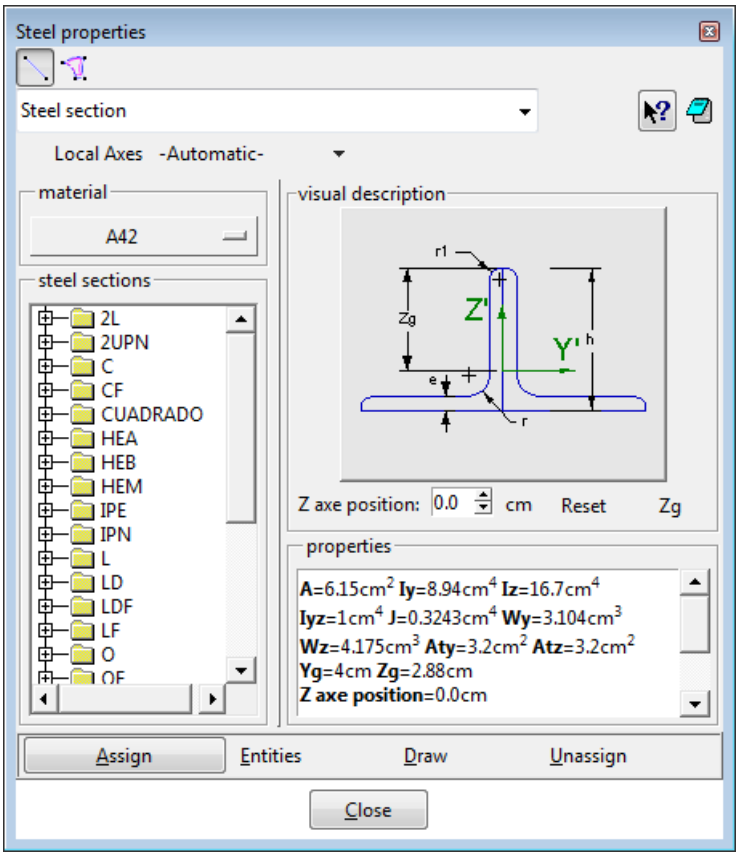
a message. In any case, if Description is not empty a message is displayed.

The argument event is the type of event and args is the list of arguments depending on the event type. The possible events are: **INIT**, **SYNC**, **CLOSE** and **DEPEND**. Below is a description of each event.

- INIT: this event is triggered when **GiD** needs to display the corresponding **QUESTION** and the list of arguments is {frame row-var GDN STRUCT QUESTION}: frame is the container frame where the widget should be placed; row-var is the name of the variable, used by **GiD**, with the current row in the frame; **GDN** and **STRUCT** are the names of internal variables needed to access the values of the data; **QUESTION** is the QUESTION's name for which the **TKWIDGET** procedure was invoked. Normally the code for this event should initialize some variables and draw the widget.
- SYNC: this is triggered when **GiD** requires a synchronization of the data. Normally it involves updating some of the QUESTIONs of the data set. The argument list is {GDN STRUCT QUESTION}.
- CLOSE: this is triggered before closing the window, as mentioned this can be canceled if an **ERROR** is returned from the procedure.
- DEPEND: this event is triggered when a dependence is executed over the **QUESTION** for which the **TKWIDGET** is defined, ie. that **QUESTION** is an lvalue of the dependence. The list of arguments is {GDN STRUCT QUESTION ACTION value} where **GDN**, **STRUCT** and **QUESTION** are as before, **ACTION** could be **SET**, **HIDE** or **RESTORE** and value is the value assigned in the dependence.

The picture below shows a fragment of the data definition file and the **GUI** obtained. This sample is taken from the problem type RamSeries/rambshell and in this case the **TKWIDGET** is used to create the whole contents of the condition windows. For a full implementation, please download the problem type and check it.

```
CONDITION: Steel_section

CONDTYPE: over lines

CONDMESHTYPE: over body elements

QUESTION: Local_Axes#LA#(-Default-,-Automatic-)

VALUE: -Default-

QUESTION: SteelName

VALUE: -

QUESTION: SteelType

VALUE: -

TKWIDGET: SteelSections

END CONDITION
```



7.7.2 Data Windows Behavior

In this subsection we explain a Tcl procedure used to configure the common behaviour of Materials. We are working on providing a similar functionality for Conditions using the same interface.

GiD_DataBehaviour controls properties of data windows for Materials and Conditions (not currently implemented). For Materials we can modify the behaviour of assign, draw, unassign, impexp (import/export), new, modify and delete. We can also specify the entity type list with the assign option throught the subcommands geomlist and meshlist.

The syntax of the procedure is as follows:

```
GiD_DataBehaviour data_class name ?cmd? proplist
```

where

- data_class could be "material" if we want to modify the behaviour of a particular material, or "materials" if a whole book is to be modified;
- name takes the value of a material's name or a book's name, depending on the value of data_class;
- cmd can take one of the values: show, hide, disable, geomlist and meshlist;
- proplist is a list of options or entity types. When the value of cmd is show, hide or disable, then proplist can be a subset of {assign draw unassign impexp new modify delete}. If the value of cmd is show it makes the option visible, if the value is hide then the option is not visible, and when the value is disable then the option is visible but unavailable. When the value of cmd is geomlist then proplist can take a subset of

{points lines surfaces volumes} defining the entities that can have the material assigned when in geometry mode; if the value of cmd is meshlist then proplist can take the value elements. Bear in mind that only elements can have a material assigned in mesh mode. If cmd is not provided, the corresponding state for each of the items provided in proplist is obtained as a result.

Example:

```
GiD_DataBehaviour materials Table geomlist {surfaces volumes}
GiD_DataBehaviour materials Solid hide {delete impexp}
```

GiD_ShowBook is a procedure to hide/show a book from the menus

```
GiD_ShowBook class book show
```

where

- class must be: gendata materials conditions or intvdata
- book is the name of the book to be show or hidden
- show must be 0 or 1

After change the book properties is necessary to call to GiDMenu::UpdateMenus

Example:

```
GiD_ShowBook materials tables 0
GiDMenu::UpdateMenus
```

7.8 GiD version

Normally, a problem type requires a minimum version of GiD to run. Because the problem type can be distributed or sold separately from GiD, it is important to check the GiD version before continuing with the execution of the problem type. GiD offers a function, **GidUtils::VersionCmp**, which compares the version of the GiD currently being run with a given version.

GidUtils::VersionCmp { Version }

This returns a negative integer if Version is greater than the currently executed GiD version; zero if the two versions are identical; and a positive integer if Version is less than the GiD version.

Note: This function will always return the value -1 if the GiD version is previous to 6.1.5.

Example:

```
proc InitGIDProject { dir } {
    set VersionRequired "10.0"
    if {[GidUtils::VersionCmp $VersionRequired] < 0 } {
        WarnWin [= "This interface requires GiD %s or later"
$VersionRequired]
```

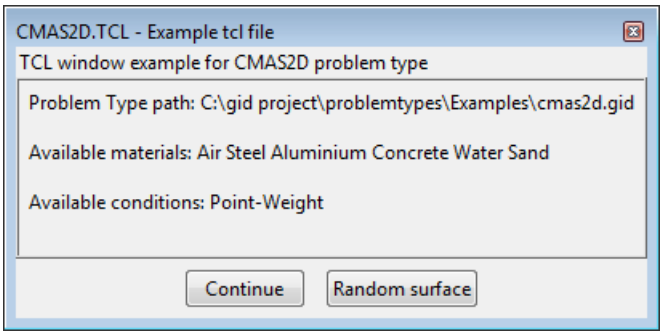
```
    }
}
```

7.9 Detailed example

Here is a step by step example of how to create a Tcl/Tk extension. In this example we will create the file cmas2d.tcl, so we will be extending the capabilities of the **cmas2d** problem type. The file cmas2d.tcl has to be placed inside the **cdmas2d** Problem Type directory.

Note: The **cmas2d** problem type calculates the center of mass of a 2D surface. This problem type is located inside the Problem Types directory, in the GiD directory.

In this example, the cmas2d.tcl creates a window which appears when the problem type is selected.



Window created in the cmas2d.tcl example file

This window gives information about the location, materials and conditions of the problem type. The window has two buttons: **CONTINUE** lets you continue working with the **cmas2d** problem type; **RANDOM SURFACE** creates a random 2D surface in the plane XY.

What follows is the Tcl code for the example. There are three main procedures in the cmas2d.tcl file:

- **proc InitGIDProject {dir}**

```
proc InitGIDProject {dir } {
  set materials [GiD_Info materials]
  set conditions [GiD_Info conditions ovpnt]
  CreateWindow $dir $materials $conditions
}
```

This is the main procedure. It is executed when the problem type is selected. It calls the **CreateWindow** procedure.

- **proc CreateWindow {dir mat cond}**

```
proc CreateWindow {dir mat cond} {
    if { [GidUtils::AreWindowsDisabled] } {
```

```
        return
    }
    set w .gid.win_example
    InitWindow $w [= "CMAS2D.TCL - Example tcl file"] ExampleCMAS "" "" 1
    if { ![winfo exists $w] } return ;# windows disabled || usemorewindows
== 0
    ttk::frame $w.top
    ttk::label  $w.top.title_text -text [= "TCL window example for CMAS2D
problem type"]
    ttk::frame $w.information -relief ridge
    ttk::label $w.information.path -text [= "Problem Type path: %s" $dir]
    ttk::label  $w.information.materials -text [= "Available materials: %s"
$mat]
    ttk::label $w.information.conditions -text [= "Available conditions: %s"
$cond]
    ttk::frame $w.bottom
    ttk::button $w.bottom.start -text [= "Continue"] -command "destroy $w"
    ttk::button   $w.bottom.random  -text  [=  "Random  surface"]  -command
"CreateRandomSurface $w"
    grid $w.top.title_text -sticky ew
    grid $w.top -sticky new
    grid $w.information.path -sticky w -padx 6 -pady 6
    grid $w.information.materials -sticky w -padx 6 -pady 6
    grid $w.information.conditions -sticky w -padx 6 -pady 6
    grid $w.information -sticky nsew
    grid $w.bottom.start $w.bottom.random -padx 6
    grid $w.bottom -sticky sew -padx 6 -pady 6
    if { $::tcl_version >= 8.5 } { grid anchor $w.bottom center }
    grid rowconfigure $w 1 -weight 1
    grid columnconfigure $w 0 -weight 1
}
```

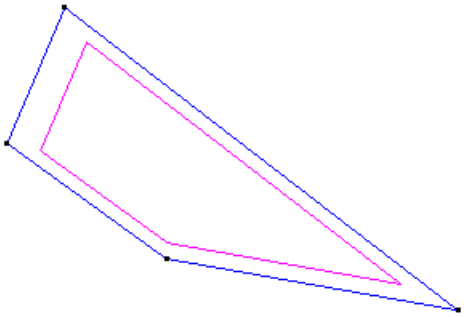
This procedure creates the window with information about the path, the materials and the conditions of the project. The window has two buttons: if **CONTINUE** is pressed the window is dismissed; if **RANDOM SURFACE** is pressed, it calls the **CreateRandomSurface** procedure.

- **proc CreateRandomSurface {w}**

```
proc CreateRandomSurface {w} {
 set ret [tk_dialogRAM $w.dialog [= "Warning"] \
    [= "Warning: this will create a nurbs surface in your current project"]
"" 1 [= "Ok"] [= "Cancel"]]
 if {$ret ==0} {
       Create_surface
       destroy $w
  }
}
```

This procedure is called when the **RANDOM SURFACE** button is pressed. Before creating the surface, a dialog box asks you to continue with or cancel the creation of the surface. If the surface is to be created, the **Create_surface** procedure is called. Then, the window is destroyed.

```
proc Create_surface {} {
    set a_x [expr rand()*10]
    set a_y [expr rand()*10]
    set b_x [expr $a_x + rand()*10]
    set b_y [expr $a_y + rand()*10]
    set c_x [expr $b_x + rand()*10]
    set c_y [expr $b_y - rand()*10]
    if {$a_y < $c_y} {
        set d_y [expr $a_y - rand()*10]
        set d_x [expr $a_x + rand()*10]
    } else {
        set d_y [expr $c_y - rand()*10]
        set d_x [expr $c_x - rand()*10]
    }
    GiD_Process escape escape escape geometry create line \
        $a_x,$a_y,0.000000       $b_x,$b_y,0.000000       $c_x,$c_y,0.000000
$d_x,$d_y,0.000000 close
    GiD_Process escape escape escape escape geometry create NurbsSurface
Automatic \
        4 escape
    GiD_Process 'Zoom Frame escape escape escape escape
}
```



A 2D surface (a four-sided 2D polygon) is created. The points of this surface are chosen at random.

8 PLUG-IN EXTENSIONS

This section explains a new way to expand GiD capabilities: the plug-in mechanism.

Plug-ins which should be used by GiD shoud be present inside the **$GID/plugins** directory.

There are two possible plugin mechanisms:

- Tcl plug-in

8.1 Tcl plug-in

If a file with extension .tcl is located inside the GiD 'plugins' folder, with the same name as the folder containing it, then it is automatically sourced when starting GiD.

This allow to do what the developer want, with Tcl language, e.g. change menus, source other Tcl files or load a 'Tcl loadable library' that extend the Tcl language with new commands implemented at C level.

To know how to create a 'Tcl loadable library' some Tcl book must be read.

See chapter about 'C Programming for Tcl' of

"Practical Programming in Tcl and Tk" by Brent Welch, Ken Jones, and Jeff Hobbs at http://www.beedub.com/book

8.2 GiD dynamic library plug-in

Note that 'GiD dynamic libraries' are different of 'Tcl loadable libraries'

'GiD dynamic libraries' must do specifically the task that GiD expects: now it is only available an interface for libraries that import mesh and create results for GiD postprocess. In the future new interfaces to do other things could appear, and to be usable must follow the rules explained in this chapter.

8.2.1 Introduction

As the variety of existent formats worldwide is too big to be implemented in GiD and, currently, the number of supported formats for mesh and results information in GiD is limited, the GiD team has implemented a new mechanism which enables third party libraries to transfer mesh and results to GiD, so that GiD can be used to visualize simulation data written in whatever format this simulation program is using.

This new mechanism is the well know mechanism of plug-ins. Particularly GiD supports the loading of dynamic libraries to read any simulation data and transfer the mesh and results information to GiD.

Viewing GiD as a platform of products, this feature allows a further level of integration of the simulation code in GiD by means of transferring the results of the simulation to GiD in any format specified by this simulation code thus avoiding the use of a foreign format.

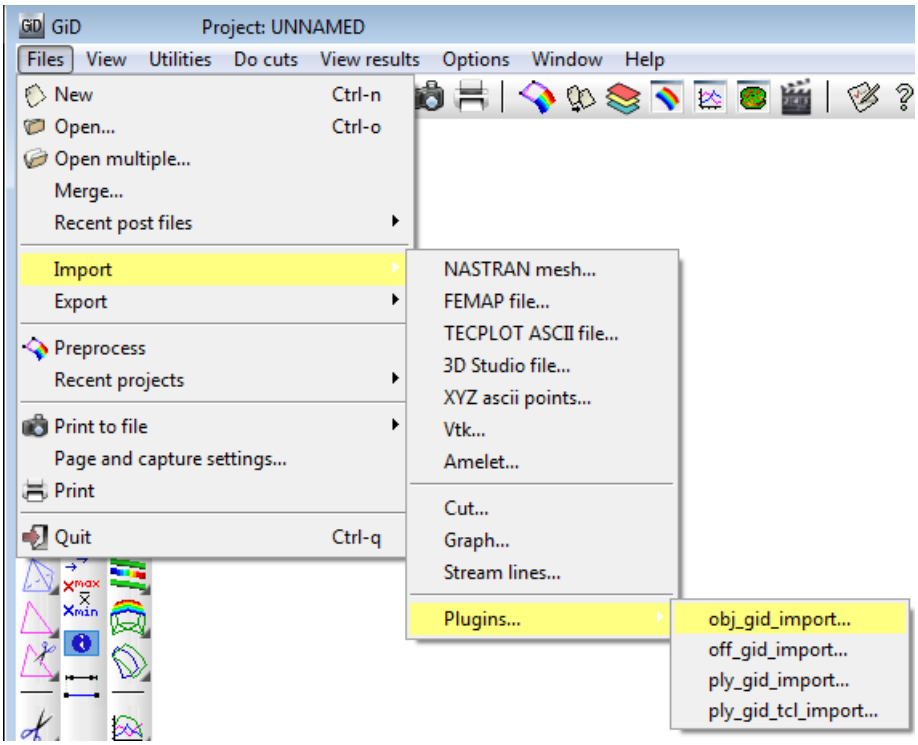
A manual loading mechanism was already available since GiD version 9.3.0-beta but since

GiD version 10.1.1e the recognized plug-ins are automatically loaded in GiD and appear in the top menu bar in the Files ☐ Import ☐ Plugins submenu.

Since GiD version 10.1.2d this mechanism not only works in Microsoft Windows and Linux, but also in Apple's Mac OS X.

8.2.2 In GiD

Since GiD 10.1.1e, the recognized import plug-ins appear in the top menu bar under the menu 'Files ☐ Import ☐ Plugins':



Import plug-ins menu showing the import plug-in examples included in GiD

But already since GiD version 9.3.0-beta these dynamic libraries can be manually loaded and called via TCL scripts, in GiD post-process's command line, or using the post-process's right menu 'Files ImportDynamicLib' and the options LoadDynamicLib, UnloadDynamicLib, CallDynamicLib:

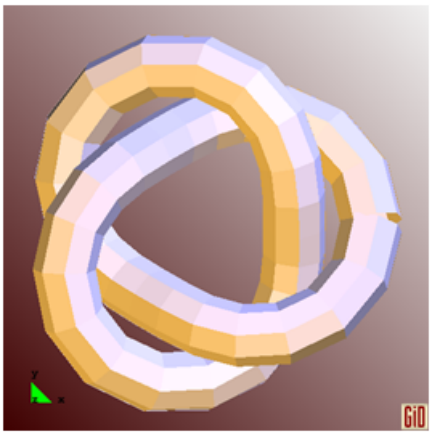
For one plug-in library, named **MyImportPlugin.dll** (or **MyImportPlugin.so** in Linux or **MyImportPlugin.dylib** in mac OS X) to be automatically recognized by GiD and to be loaded and listed in the top's menu Files ☐ Import ☐ Plugins, the library should lie inside a directory of the same name, i.e. MyImportPlugin/MyImportPlugin.dll, under any sub-folder of the %GID%/plugins/Import directory:

Note that only the GiD 32 bits version can handle 32 bits import plug-in dynamic libraries, and only GiD 64 bits can handle 64 bits import plug-in dynamic libraries. Which version of GiD is currently running can be easily recognized in the title bar of the main window (Title bar of GiD's window showing 'GiD x64', so the current GiD is the 64 bits version)

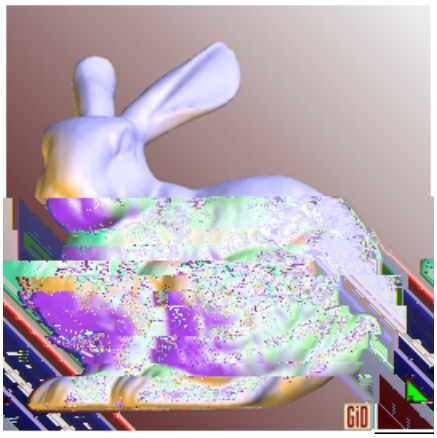
Together with the GiD installation, following import plug-ins are provided:

- OBJ: Wavefront Object format from Wavefront Technologies

- OFF: Object file format vector graphics file from Geomview
- PLY: Polygon file format, aka Stanford Triangle Format, from the Stanford graphics lab.
- PLY-tcl: this plug-in is the same as the above PLY one but with a tcl's progress bar showing the tasks done in the library while a ply file is imported. For all of these plug-in examples both the source code, the Microsoft Visual Studio projects, Makefiles for Linux and Mac OS X, and some little models are provided



The 'tref.off' Object File Format example



The 'bunny_standford.ply' Polygon File Format example

8.2.3 Developing the plug-in

GiD is compiled with the Tcl/Tk libraries version 8.5.11.

Remember that if the developed plugin is targeted for 32 bits, only GiD 32 bits can handle it. If the developed plugin is developed for 64 bits systems, then GiD 64 bits is the proper one to load the plugin.

Header inclusion

In the plug-in code, in one of the .cc/.cpp/.cxx source files of the plug-in, following definition must be made and following file should be included:

```
#define BUILD_GID_PLUGIN

#include "gid_plugin_import.h"
```

In the other .cc/.cpp/.cxx files which also use the provided functions and types, only the gid_plugin_import.h file should be included, without the macro definition.

The macro is needed to declare the provided functions as pointers so that GiD can find them and link with its internal functions.

Functions to be defined by the plug-in

Following functions should be defined and implemented by the plug-in:

```
extern "C" GID_DLL_EXPORT int GiD_PostImportFile( const char *filename) ) {
        ... ;
        return 0; // 1 - on error
}
```

```
extern "C" GID_DLL_EXPORT const char *GiD_PostImportGetLibName( void) {
        return "Wavefront Objects import";
}
extern "C" GID_DLL_EXPORT const char *GiD_PostImportGetFileExtensions( void)
{
        return "{{Wavefront Objects} {.obj}} {{All files} {*}}";
}
extern "C" GID_DLL_EXPORT const char *GiD_PostImportGetDescription( void) {
        return "Wavefront OBJ import plugin for GiD";
}
extern "C" GID_DLL_EXPORT const char *GiD_PostImportGetErrorStr( void) {
        return _G_err_str; // if error, returns the error string
}
```

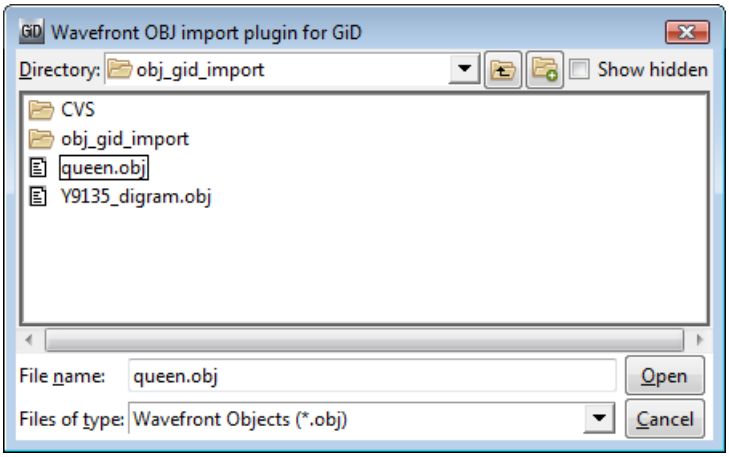
When GiD is told to load the dynamic library, it will look for, and will call these functions:

GiD_PostImportGetLibName : returns the name of the library and should be unique. This name will appear in the 'File ☐ Import ☐ Plugin' menu and in the right menu.

GiD_PostImportGetFileExtensions : which should return a list of extensions handled by the library and will be used as filter in the Open File dialogue window.

GiD_PostImportGetDescription: : returns the description of the library and will be displayed in the title bar of the Open File dialogue window.

Once the library is registered, when the user selects the menu entry 'File ☐ Import ☐ Plugin ☐ NewPlugin' the Open File dialogue window will appear showing the registered filters and description of the plug-in.



The file selection window showing the plug-in description as title of the window and filtering the file list with the registered extension

When the user selects a file then following functions are called:

GiD_PostImportFile : this function should read the file, transfer the mesh and results information to GiD and return 0 if no problems appeared while the file was read or 1 in case of error.

GiD_PostImportGetErrorStr : this function will be called if the previous one returns 1, to retrieve the error string and show the message to the user.

8.2.4 Functions provided by GiD

Inside the GiD_PostImportFile function, following functions can be called to pass information to GiD:

```
extern "C" int GiD_NewPostProcess( void);

extern "C" int GiD_NewMesh( _t_gidMode gid_mode, _t_gidMeshType mesh_type,
const char *name);

extern "C" int  GiD_SetColor( int id, float red, float green, float blue,
float alpha);

extern "C" int GiD_SetVertexPointer( int id,
   _t_gidBasicType basic_type,
   _t_gidVertexListType list_type,
   int num_components,
   int num_vertices,
   unsigned int offset_next_element,const void *pointer);

extern "C" int GiD_SetElementPointer( int id,
   _t_gidBasicType basic_type,
   _t_gidElementListType list_type,
   _t_gidElementType element_type,
   int num_elements,
   unsigned int offset_next_element,
   const void *pointer,
   unsigned int offset_float_data,
   const void *float_ptr);

extern "C" int GiD_NewResult( const char *analysis_name, double step_value,
   const char *result_name, int mesh_id);

extern "C" int GiD_SetResultPointer( int id,
   _t_gidBasicType basic_type,
   _t_gidResultListType list_type,
   _t_gidResultType result_type,
   _t_gidResultLocation result_location,
   int num_results,
   unsigned int offset_next_element,
```

```
   const void *pointer);

extern "C" int GiD_EndResult( int id);

extern "C" int GiD_EndMesh( int id);

extern "C" Tcl_Interp *GiD_GetTclInterpreter();
```

Here is the description for each provided function:

GiD_NewPostProcess : starts a new post-process session, deleting all mesh and results information inside GiD.

GiD_NewMesh : a new mesh will be transferred to GiD and an identifier will be returned so that more information can be defined for this mesh. Following parameters must be specified:

_t_gidMode gid_mode : may be one of GID_PRE or GID_POST. At the moment only GID_POST is supported

_t_gidMeshType mesh_type : may be one of GIDPOST_NEW_MESH, GIDPOST_MERGE_MESH and GIDPOST_MULTIPLE_MESH. At the moment only GIDPOST_NEW_MESH has been tested

const char *name : name of the mesh which will appear in the Display Style window.

GiD_SetColor : to specify a colour for the mesh identified by the given id. The red, green, blue and alpha components should be between 0.0 and 1.0.

GiD_SetVertexPointer : sets the vertices of the mesh identified by the given id. This vertices are the ones to be referred from the element's connectivity. Following parameters may be set:

_t_gidBasicType basic_type : data type of the coordinates of the vertices, should be one of GIDPOST_FLOAT or GIDPOST_DOUBLE;

_t_gidVertexListType list_type : herewith the format of the vertices is specified. Should be one of

GIDPOST_VERTICES: where all num_components coordinates are specified with no label and so they will be numerated between 0 and num_vertices-1

GIDPOST_IDX_VERTICES: where each set of num_components coordinates are preceded by a label indicating its node number (should be a 4-byte integer)

int num_components : number of coordinates per vertex

int num_vertices : number of vertices in the list

unsigned int offset_next_element : distance in bytes between the beginning of vertex i and the beginning of vertex i + 1. If 0 is entered then the vertices are all consecutive

const void *pointer : pointer to the list of vertices.

GiD_SetElementPointer : sets the elements of the mesh identified by the given id. The elements connectivity refers to the previous specified list of vertices. Note that for spheres

and circles not only their connectivity should be specified but also their radius and eventually their normal. In this case two separate vectors should passed: one for the integer data and another one for the floating point data. Following parameters may be set:

_t_gidBasicType basic_type : data type of the extra data entered for sphere and circle elements, should be one of GIDPOST_FLOAT or GIDPOST_DOUBLE.

_t_gidElementListType list_type : herewith the format of the elements is specified. Should be one of GIDPOST_CONNECTIVITIES: where all the elements are specified without element number, thus being automatically numbered between 0 and num_elements-1

GIDPOST_IDX_CONNECTIVITIES: where each element is preceded by a label indicating its element number (should be a 4-byte integer)

_t_gidElementType element_type : type of element to be defined. May be one of GIDPOST_TRIANGLE, GIDPOST_QUADRILATERAL, GIDPOST_LINE, GIDPOST_TETRAHEDRON, GIDPOST_HEXAHEDRON, GIDPOST_POINT, GIDPOST_PRISM, GIDPOST_PYRAMID, GIDPOST_SPHERE, GIDPOST_CIRCLE.

int num_elements : number of elements in the list

unsigned int offset_next_element : distance in bytes between the beginning of element i and the beginning of element i+1. If 0 is entered then the elements are all consecutive

const void *pointer : pointer to the list of the elements connectivity (integer data)

unsigned int offset_float_data : distance in bytes between the beginning of float data of element i and the beginning of float data of element i+1. If 0 is entered then the element's float data are all consecutive.

const void *float_ptr : pointer to the list of the floating point data for the elements. For spheres only the radius should be specified, so just a single value, and for circles four values should be specified: its radius and the three components of the normal.

GiD_NewResult : a new result will be defined for GiD and an identifier will be returned so that more information can be defined for this result. Following parameters must be specified:

const char *analysis_name : analysis name of the result

double step_value : step value inside the analysis where the result should be defined

const char *result_name : result name

int mesh_id : mesh identifier where the result is defined. If 0 is entered the result will be defined for all meshes.

GiD_SetResultPointer : specifies the list with the result values for a given result's id. Following parameters may be set:

_t_gidBasicType basic_type : data type of the results, should be one of GIDPOST_FLOAT or GIDPOST_DOUBLE

_t_gidResultListType list_type : herewith the format of the results is specified. Should be one of GIDPOST_RESULTS: where all results are defined consecutively and will refer to the nodes / elements between 0 and num_results-1

GIDPOST_IDX_RESULTS: where each result is preceded by a label indicating its node /element number (should be a 4-byte integer)

_t_gidResultType result_type : type of result which will be defined. May be one of GIDPOST_SCALAR, GIDPOST_VECTOR_2 (vector result with 2 components), GIDPOST_VECTOR_3 (vector with 3 components), GIDPOST_VECTOR_4 (vector with 4 components, including signed modulus), GIDPOST_MATRIX_3 (matrix with 3 components Sxx, Syy and Sxy), GIDPOST_MATRIX_4 (Sxx, Syy, Sxy and Szz, GIDPOST_MATRIX_6 (Sxx, Syy, Sxy, Szz and Syz and Sxz), GIDPOST_EULER (with 3 euler angles), GIDPOST_COMPLEX_SCALAR (real and imaginary part), GIDPOST_COMPLEX_VECTOR_4 (2d complex vector: Vxr, Vxi, Vyr and Vyi), GIDPOST_COMPLEX_VECTOR_6 (3d complex vector: Vxr, Vxi, Vyr, Vyi, Vzr and Vzi) and GIDPOST_COMPLEX_VECTOR_9 (3d complex vector: Vxr, Vxi, Vyr, Vyi, Vzr, Vzi, |real part|, |imaginary part| and signed |vector|)

_t_gidResultLocation result_location : location of the result. May be one of GIDPOST_NODAL, GIDPOST_ELEMENTAL or GIDPOST_GAUSSIAN. At the moment GIDPOST_GAUSSIAN is not supported

int num_results : number of results in the list

unsigned int offset_next_element : distance in bytes between the beginning of result i and the beginning of result i+1. If 0 is entered then the reuslts are all consecutive

const void *pointer : pointer to the list of results.

GiD_EndResult : indicates GiD that the definition of the result with the give id is finished. GiD will process the result.

GiD_EndMesh : indicates GiD that the definition of the mesh with the give id is finished. GiD will process the mesh.

GiD_GetTclInterpreter : returns a pointer to GiD's global interpreter so that the plug-in can open their windows or execute their tcl scripts using the predefined tcl procedures of GiD.

The developer should keep in mind that all the plug-in code is executed inside GiD's memory space and so, all the memory allocated inside the plug-in should also be freed inside the plug-in to avoid memory accumulation when the dynamic library is called repeatedly. This also includes the arrays passed to GiD, which can be deleted just after passing them to GiD.

8.2.5 List of examples

The plug-in examples provided by GiD also include some little models of the provided import format.

These are the import plug-ins provided by GiD so far:

OBJ: Wavefront OBJ format

This is a starter example which includes the create_demo_triangs function which creates a very simple mesh.

The obj format is a very simple ascii format and this plug-in:

reads the file,

creates a GiD mesh with the read triangles and quadrilaterals,

and, if the information about the vertex normals is present, then this information is passed to GiD as nodal vector results.

OFF: Object file format

This example is very similar to the previous one.

The off format is a very simple ascii format but including n-agons and colour on vertices and faces. So, this plug-in:

reads the file,

creates a GiD mesh with the read triangles and quadrilaterals and triangulates the read pentagons and hexagons (and discards bigger n-agons),

if colour information is present in the off file, which can be present on the nodes or on the elements, then this information is passed to GiD as nodal or elemental results.

PLY: Polygon file format

This example is a little bit more complex.

Ply files can be ascii or binary, and the code of this plug-in is based in Greg Turk's code, developed in 1998, to read ply files. This format allows the presence of several properties on nodes and faces, too. This plug-in:

reads the file,

creates a GiD mesh with the read lines, triangles and quadrilaterals,

if information about the vertex normals is found, then this information is passed to GiD as nodal vector results,

all the properties defined in the ply file are passed to GiD. This properties can be defined on the nodes of on the faces of the model, and so are they transferred to GiD.

Here the complexity also resides in the liberation of the reserved memory, which is wildly allocated in the ply code.

PLY + Tcl : Polygon file format

This plug-in is the same as the previous PLY plug-in but a tcl script is added inside the code to show a progress bar in tcl to keep the user entertained while big files are read.

9 APPENDIX (PRACTICAL EXAMPLES)

To learn how to configure GiD for a particular type of analysis, you can find some practical examples:

- By following the **Problem Type Tutorial**; this tutorial is included with the GiD package you've bought. You can also download the tutorial from support area of the GiD web page http://www.gidhome.com
- By studing and modifing some Problem Type

Problem types included in GiD by default as example in \problemtypes\Examples:

- **cmas2d:** This is the problem type created in the tutorial, which finds the distance of each element relative to the center of masses of a two-dimensional surface. It uses the following files: `.cnd`, `.mat`, `.prb`, `.bas`, `.tcl` and `.bat`. There is a help file inside directory **cmas2d.gid** called **cmas2d.html**
- **cmas2d_CompassLIB:** The same problem type, but implemented using the 'CompassLIB library' developed by CompassIS http://www.compassis.com.
- **problem_type_solid1:** This is the interface of what it could be a structure-calculating module. It contains the `.cnd`, `.mat`, `.prb`, `.sim`, `.bas` and `.bat` files.
- **problem_type_solid2:** Same as **problem_type_solid1** with a different interface.
- **problem_type_thermo_mec:** This is the interface of what it could be a thermo-mechanical module. It has programation in TCL/TK language. It uses the following files: `.cnd`, `.mat`, `.prb`, `.sim`, `.bas.bat` and `.tcl`.

Other problemtypes can be downloaded from the *Data->Problem type->Internet retrieve* menu:

- **rambshell:** This is a problem type which performs the structural analysis of either beams or shells or a combination of both using the Finite Element Method. This problem type uses the latest features offered by GiD . The `.exe` file for Windows systems is also included in a limited version.
- **NASTRAN**: Static and dynamic interface for the NASTRAN commercial analysis program (not included)
- **Tdyn**: Multiphysics solver (including CFD, heat transfer, species advection, pde solver and free surface problems)

For the full version without limitations check http://www.compassis.com.

10 INDEX

A

Add 38

B

bas file example 49

BasicUnit 30

bat file commands 68

bat file examples 75

Book 20

Break 38

C

Call 68

Clock 30

color for meshes 98

command.exe 68

commands *.bat 68

Commands used in the .bas file 29

Complex numbers 85

Cond 30

CondElemFace 34

CondHasLocalAxes 30

Conditions definitions 9

Conditions file example 13

Conditions symbols 25

CondName 30

CondNumEntities 30

CondNumFields 30

Copy command 68

Custom password validation 7

D

data files 76

data input files 48

Del 68

Dependencies 20

E

Echo 68

ElemsCenter 34

ElemsConec 34

ElemsLayerName 30

ElemsLayerNum 30

ElemsMat 30

ElemsMatProp 30

ElemsNnode 34

ElemsNnodeCurt 34

ElemsNNodeFace 34

ElemsNNodeFaceCurt 34

ElemsNormal 34

ElemsNum 30

ElemsRadius 34

ElemsType 34

ElemsTypeName 34

Else 38

ElseIf 38

encoding for mesh names 98

End 38

Endif 38

Erase 68

Errors when executing an external program 75

Event 111

Executing external program 65

External program execution 65

F

FactorUnit 30

field width 9

File NAME.bas 48

File NAME.cnd 9

File NAME.mat 17
File NAME.prb 15
File NAME.sim 25
File NAME.uni 23
File NAME.xml 7
File ProjectName.flavia.msh 98
File ProjectName.flavia.res 78
File ProjectName.post.msh 98
File ProjectName.post.res 78
Files ***.bas 48
Files ***.geo 25
For 38
Format 38
G
Gauss points format (new format) 79
GenData 30
GiD_DataBehaviour 161
GiD_GetWorldCoord 148
GiD_Thumbnail 148
GiDCustomHelp 151
GidUtils::VersionCmp 162
GlobalNodes 34
Goto 68
Graph Lines file format 108
Groups 103
H
Help 151
help structure 152
HelpDirs 152
HelpWindow 153
HTML support 151
I
If 38
If command 68
Image 20
Include 38
IndexPage 153
Info function 120
Intformat 38
IntvData 30
IsQuadratic 30
L
LayerColor 30
LayerName 30
LayerNum 30
LayerNumEntities 30
Listing of .bas commands 29
local axes 9
LocalAxesDef 34
LocalAxesDef(EulerAngles) 34
LocalAxesDefCenter 34
LocalAxesNum 30
LocalNodes 34
Loop 38
LoopVar 30
M
Managing menus 154
Materials file example 18
Materials properties 17
MatNum 30
MatProp 30
Menu Tcl functions 154
mesh example 101
Mesh groups 103
MessageBox 38
Mkdir 68
model unit 23
Multiple meshes 103
Multiple values return commands 34
N

Ndime 30
Nelem 30
Nintervals 30
Nlocalaxes 30
Nmats 30
Nnode 30
NodesCoord 34
NodesLayerName 30
NodesLayerNum 30
NodesNum 30
Npoin 30
O
OnlyInCond 38
OnlyInLayer 38
Operation 30
Output view 67
P
PasswordPath 7
Post-processing data files 76
Postprocess Files 76
Postprocess mesh format 98
Postprocess results format 78
PRB file example 16
Problem and intervals data 15
Problem and intervals file example 16
Problem Type Files 5
Process function 119
ProjectName.flavia.msh 98
ProjectName.flavia.res 78
ProjectName.post.msh 98
ProjectName.post.res 78
R
Re-mesh 103
Realformat 38
Rem 68
Remesh 103
Remove 38
Rename 68
Result entry, Result block 85
Result group 94
result groups 94
results example 89
Results file 78
S
Set 38
Set command 68
Set Layer 38
SetFormatForceWidth 38
SetFormatStandard 38
Shift 68
Single value return commands 30
Special functions 134
Specific commands 38
Symbols file example 26
Symbols geometrical definitions 25
T
table mesh example 101
Table results example 89
Tcl 38
Tcl Functions 111
Tcl Info Function 120
Tcl menu functions 154
Tcl Process Function 119
Tcl Special functions 134
Tcl/Tk extension 109
Tcl/Tk extension creation example 163
Template File (.bas) 28
Template file example 49
Template file. Description 48
Time 30

Title 20

TkWidget 158

TocPage 152

Type 68

U

Unit field 20

unit system 23

Units 30

V

ValidatePassword 7

Var 38

version 162

W

WarningBox 38

Writing data input files 48

X

XML file 7