

GiD plug-in (Post import DLL's) development manual

6. June 2011

version 1.4

Miguel Pasenau, GiD Team

Changes:

v 1.0	12. November 2010	→	Initial version
v 1.0.1	13. November 2010	→	Small corrections
v 1.0.2	15. November 2010	→	Added Mac OS X support
v 1.4	6. June 2010	→	Same version as gid_plugin_import.h, added support for complex scalars and complex vectors.

Index

1. Introduction.....	4
2. In GiD.....	5
3. Developing the plug-in.....	8
3.1. Header inclusion.....	8
3.2. Functions to be defined by the plug-in.....	8
3.3. Functions provided by GiD.....	9
3.4. List of examples.....	12
3.4.1 OBJ: Wavefront OBJ format.....	12
3.4.2 OFF: Object file format.....	12
3.4.3 PLY: Polygon file format.....	13
3.4.4 PLY + Tcl : Polygon file format.....	13
4. On-going work.....	14

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.

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':

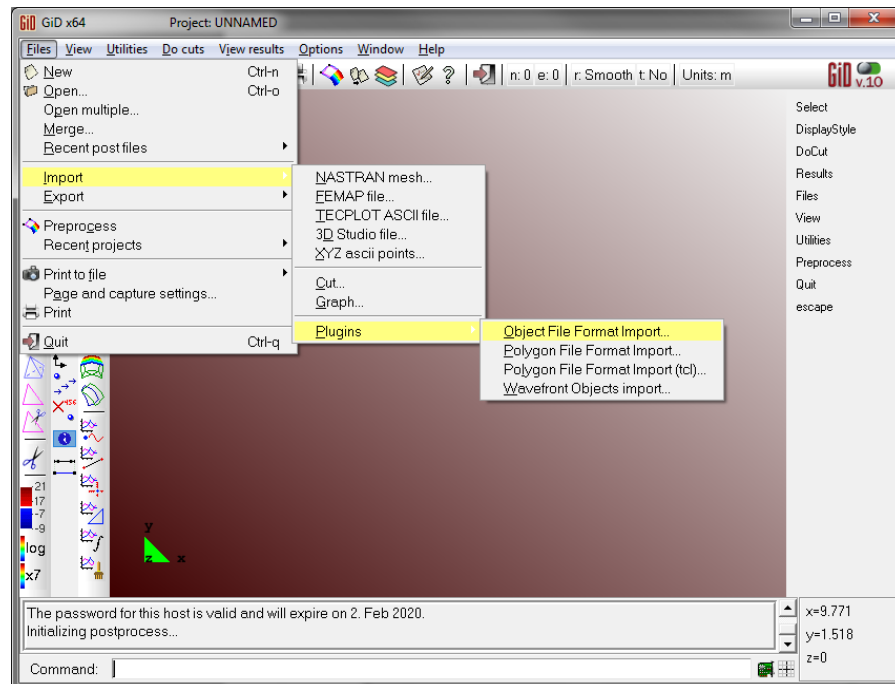


Figure 1: Import plug-ins menu showing the import plug-in examples included in GiD

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:

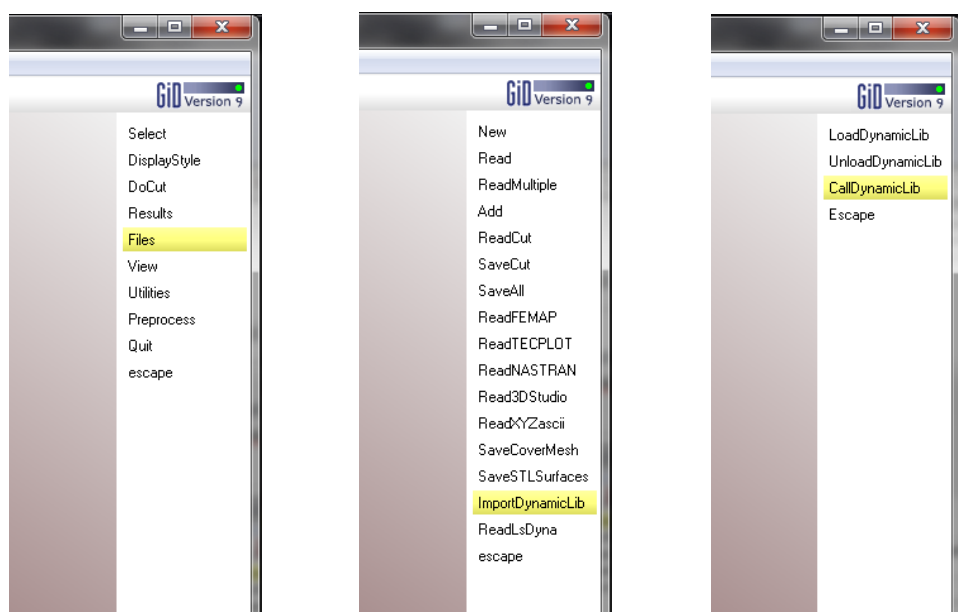


Figure 2: Right post-process menu showing the options to load, unload and call the import dynamic libraries.

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:

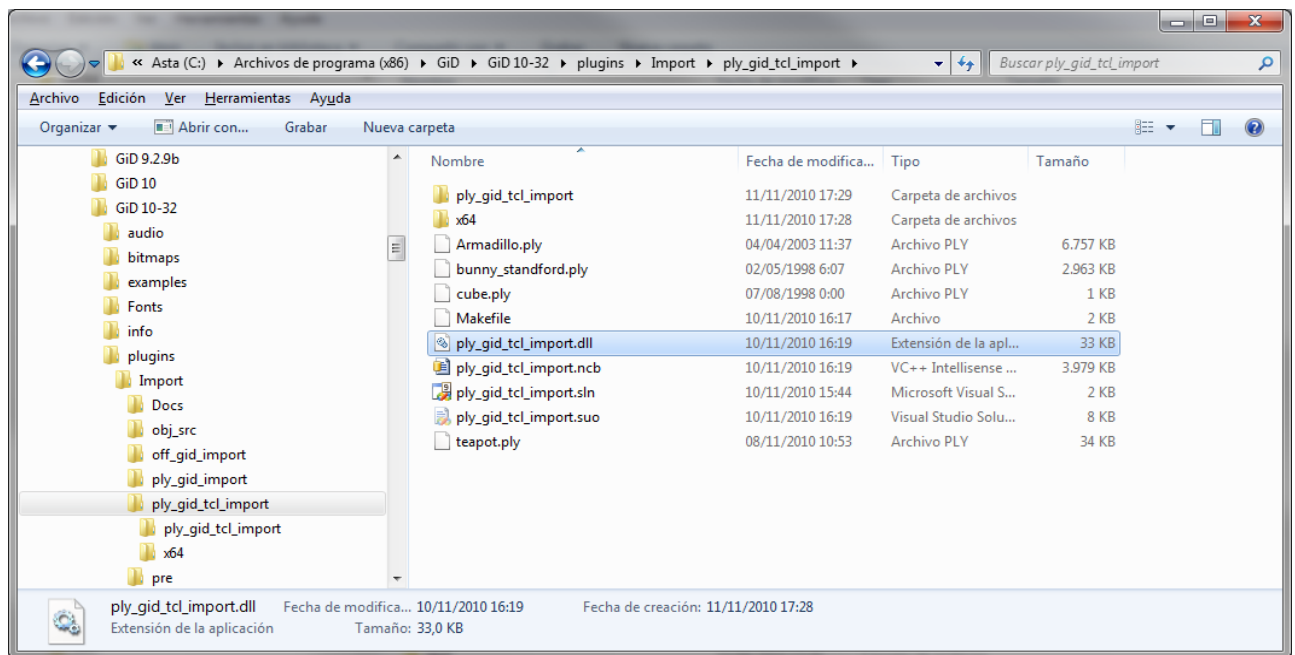


Figure 3: Location of the **ply_gid_tcl_import.dll** under **%GiD_DIRECTORY%/plugins/Import/ply_gid_tcl_import** .

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, as [figure 4](#) shows:

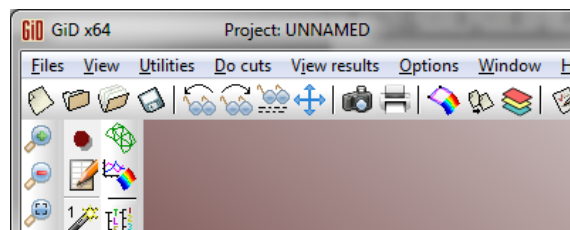
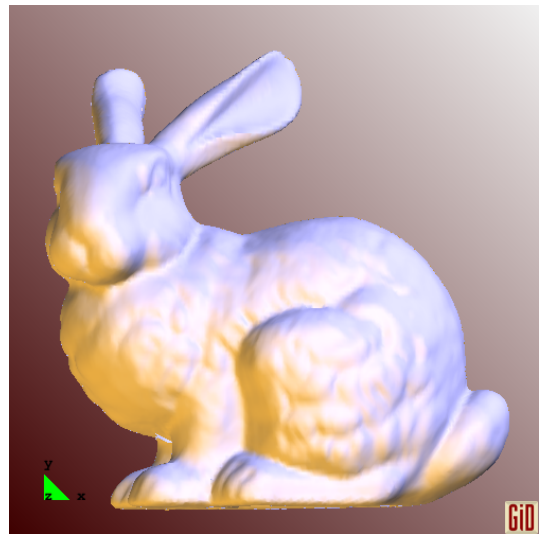
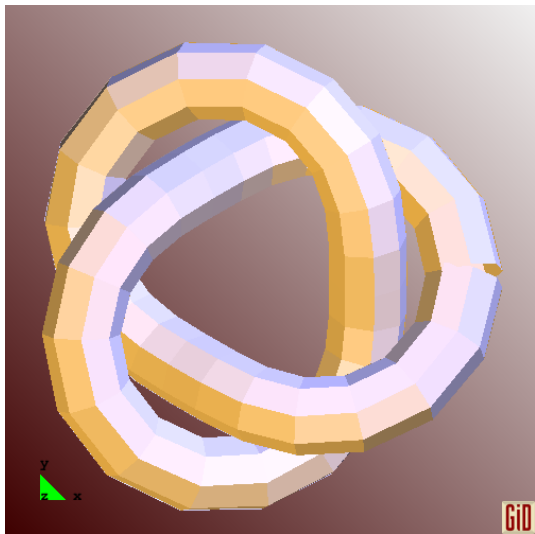


Figure 4: Title bar of GiD's window showing 'GiD x64', so the current GiD is the 64 bits version.

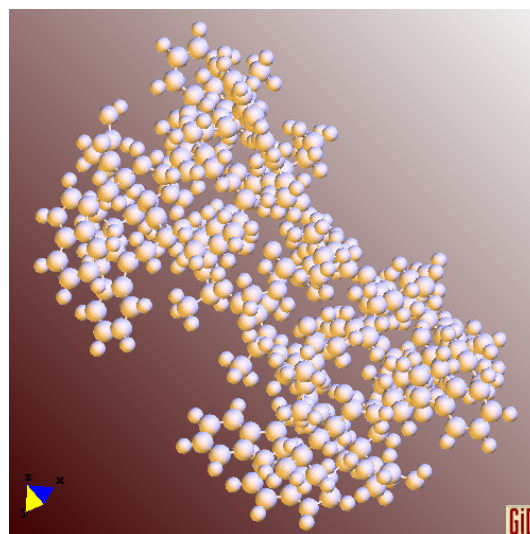
Together with the GiD installation, following import plug-ins are provided:

- 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.



[Figure 5](#): The 'tref.off' Object File Format example. [Figure 6](#): The 'bunny_standford.ply' Polygon File Format example.



[Figure 7](#): The 'Y9135_diagram.obj' Wavefront Object file format.

3. Developing the plug-in

GiD is compiled with the Tcl/Tk libraries version 8.5.8.

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. GiD 64 bits can be easily recognized by the **GiD x64** label which appears in the main window's title bar as the [figure 4](#) shows.

3.1. 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.

3.2. 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, as in [figure 8](#).

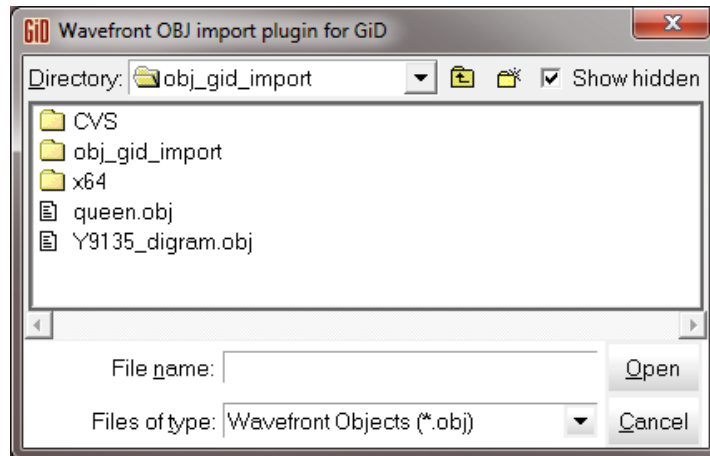


Figure 8: The file selection window showing the plug-in description as title of the window and filtering the file list with the registered extension.

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.

3.3. 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 $\text{num_vertices} - 1$;
 - 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 be 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 S_{xx} , S_{yy} and S_{xy}), `GIDPOST_MATRIX_4` (S_{xx} , S_{yy} , S_{xy} and S_{zz} , `GIDPOST_MATRIX_6` (S_{xx} , S_{yy} , S_{xy} , S_{zz} and S_{yz} and S_{xz}), `GIDPOST_EULER` (with 3 euler angles), `GIDPOST_COMPLEX_SCALAR` (real and imaginary part), `GIDPOST_COMPLEX_VECTOR_4` (2d complex vector: V_{xr} , V_{xi} , V_{yr} and V_{yi}), `GIDPOST_COMPLEX_VECTOR_6` (3d complex vector: V_{xr} , V_{xi} , V_{yr} , V_{yi} , V_{zr} and V_{zi}) and `GIDPOST_COMPLEX_VECTOR_9` (3d complex vector: V_{xr} , V_{xi} , V_{yr} , V_{yi} , V_{zr} , V_{zi} , |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 results 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_EndRMesh` : 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.

3.4. 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:

3.4.1 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.

3.4.2 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.

3.4.3 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 or 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.

3.4.4 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.

4. On-going work

This plug-in mechanism does not pretend to be a finished work as it does not cover all the possibilities to enter data into GiD.

Be aware also that the results cache mechanism, present in GiD to allow the management of very big files, at the moment, does not work with this plug-in mechanism.