

GiDPost

Library for writing GiD postprocess results

GiDPost Library	3
Compiling the library and examples	4
As python module	7
List of examples	8
Changes	9
Introduction	14
Using the library	15
Functions references	17
Library use	18
GiD_PostInit	19
GiD_PostDone	20
GiD_PostGetVersion	21
GiD_PostIsThreadSafe	22
GiD_PostSetFormatReal	23
GiD_PostGetFormatReal	24
GiD_PostGetFormatStep	25
Mesh file functions	26
GiD_fOpenPostMeshFile	27
GiD_fClosePostMeshFile	28
GiD_fBeginMeshGroup	29
GiD_fEndMeshGroup	30
GiD_fBeginMesh, GiD_fBeginMeshColor	31
GiD_fMeshUnit	32
GiD_fEndMesh	33
GiD_fBeginCoordinates	34
GiD_fEndCoordinates	35
GiD_fWriteCoordinates	36
GiD_fBeginElements	37
GiD_fEndElements	38
GiD_fWriteElement	39
GiD_fWriteElementMat	40
GiD_fWriteCoordinatesBlock (array of coordinates)	41
GiD_fWriteElementsBlock (array of connectivities)	42
Results file functions	43
GiD_fOpenPostResultFile	44
GiD_fFlushPostFile	45
GiD_fClosePostResultFile	46
GiD_fBeginGaussPoint	47
GiD_fEndGaussPoint	48
GiD_fWriteGaussPoint	49
GiD_fBeginRangeTable	50
GiD_fEndRangeTable	51
GiD_fWriteRange	52
GiD_fBeginResult (DEPRECATED)	53
GiD_fBeginResultHeader	54

GiD_fResultRange	55
GiD_fResultComponents	56
GiD_fResultUnit	57
GiD_fBeginResultGroup	58
GiD_fResultDescription	59
GiD_fResultValues	60
GiD_fEndResult	61
GiD_fBeginOnMeshGroup	62
GiD_fEndOnMeshGroup	63
GiD_fResultUserDefined	64
GiD_fWriteScalar / Vector / Matrix / LocalAxes / Complex* / NurbsSurface*	65
GiD_fWriteResultsBlock (array of result values)	66
Download the library	68

GiDPost Library

version 2.12 [🔗](#)



Library to write post-process results for GiD. [🔗](#)

Compiling the library and examples

GiDPost can be build as library to be linked into the calculation program or as python module (look into [As python module](#)).

This page explains how to build gidpost as library to be linked to you program.

Third part libraries: [🔗](#)

- **zlib**: it is necessary previously obtain or compile the zlib compression library (www.zlib.net)
- **hdf5**: this library is optional www.hdfgroup.org. Required only to write postprocess files with this format (files opened with GiD_PostHDF5=3 flag)

To install the required packages in Linux just do the following **as root**:

Ubuntu (Debian and the like):

```
1 apt-get install zlib1g-dev libhdf5-dev libhdf5-hl-100      ;# version number depends on linux
   distribution and version
2 apt-get install hdf5-tools hdf5view                      ;# recommended tools to manage and look into
   hdf5 files
```

Scientific Linux (Redhat and the like):

```
1 yum install zlib-devel.x86_64 hdf5-devel.x86_64
```

Note:

Bear in mind that there are several hdf5 development packages and that you should choose the right for you.

For instance, in Ubuntu these are the HDF5 development packages available:

```
1 libhdf5-mpi-dev
2 libhdf5-mpich-dev
3 libhdf5-openmpi-dev
4 libhdf5-serial-dev      ;# by default the package 'libhdf5-dev' installs the -serial version
```

And in Scientific Linux the HDF5 development packages available:

```
1 hdf5-devel.x86_64
2 hdf5-mpich-devel.x86_64
3 hdf5-openmpi.x86_64
```

Required third part tool: [🔗](#)

- **CMake**: The CMake tool (www.cmake.org) is used to have a cross-platform build system.

Must download and install this program to compile gidpost.lib (gidpost.a on Linux).

To use the graphical interface of cmake use **cmake-gui** or **ccmake** .

Note: For MS Visual Studio a project obtained from CMake is also included in the distribution, to avoid the requirement of install CMake.

Note: Users that don't want to use CMake could also manually create off course its own makefile for Unix/Linux or its own compiler project.

All the source code to generate the library is included, it can be compiled in other platforms.

The third party code, "cfortran.h", is also provided as a link between C and FORTRAN. It has its own distribution policy. Please, read the file "cfortran.doc" about the licence terms of this code.

Precompiled version:

The 'binaries' folder stores precompiled release versions of the library for Windows and Linux (x32 and x64 platforms), to avoid the requirement of compile them. [↗](#)

Compiling gidpost:



to compile the gidpost library using the command line: [↗](#)

How to build (Linux - gcc): [↗](#)

```
1 $ cd ../gidpost
2 $ mkdir build-linux
3 $ cd build-linux
4 $ cmake -DENABLE_HDF5=ON -DENABLE_FORTRAN_EXAMPLES=ON ..      ;# gfortran is needed to
   BUILD_FORTRAN_EXAMPLES (by default it's off)
5 # more options are: -DENABLE_SHARED_LIBS=ON -DENABLE_EXAMPLES=ON -DENABLE_PARALLEL_EXAMPLE=ON
6 $ make
7 $ cd examples
8 $ ./testc -help        ;# to view format options
9 $ ./testc -f hdf5      ;# to write a gid post file in hdf5 format
10 $ ./testf90           ;# fortran 90 example writing hdf5 gid post file
```

How to build (Linux - nvidia hpc sdk): [↗](#)

Using nVidia HPC sdk <https://developer.nvidia.com/hpc-sdk>:

First, remember to add the environment variables to your `$HOME/.bashrc` :

<https://docs.nvidia.com/hpc-sdk/hpc-sdk-install-guide/index.html#install-linux-end-user-env-settings>

```
1 $ NVARCH=`uname -s`_`uname -m`; export NVARCH
2 $ NVCOMPILERS=/opt/nvidia/hpc_sdk; export NVCOMPILERS
3 $ MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/23.5/compiler/man; export MANPATH
4 $ PATH=$NVCOMPILERS/$NVARCH/23.5/compiler/bin:$PATH; export PATH
```

Once done, log in again and:

```
1 $ cd ../gidpost
```

```

2 $ mkdir build-linux
3 $ cd build-linux
4 $ cmake -DHDF5=ON -DBUILD_FORTRAN_EXAMPLES=ON \
5 -DMAKE_CXX_COMPILER=/opt/nvidia/hpc_sdk/Linux_x86_64/23.5/compilers/bin/nvc++ \
6 -DMAKE_C_COMPILER=/opt/nvidia/hpc_sdk/Linux_x86_64/23.5/compilers/bin/nvc \
7 -DMAKE_Fortran_COMPILER=/opt/nvidia/hpc_sdk/Linux_x86_64/23.5/compilers/bin/nvfortran \
8 ..      ;# you can also use 'ccmake/cmake-gui ..' with 'Advanced options' enabled.
9 $ make
10 $ cd examples
11 $ ./testc -help      ;# to view format options
12 $ ./testc -f hdf5    ;# to write a gid post file in hdf5 format
13 $ ./testf90          ;# fortran 90 example writing hdf5 gid post file

```

For the graphical interface of cmake use **cmake-gui** or **ccmake** .

e.g. for Windows, to compile with Microsoft Visual Studio 2005, fill in

Where is the source code: C:/gid project/gidpost

Where to build the binaries: C:/gid project/gidpost/win/vs2005

and press "Configure"

select generate a project of "Visual Studio 8 2005", "use default native compilers"

then some errors appear (lines in red)

select GIDPOST_USE_SYSTEM_ZLIB and press "Configure"

then must specify the path to the zlib include and the zlib library

(zlib must be compiled previously using its own mechanism, or get it precompiled)

To add HDF5 extra features the HDF5 library must be compiled using its own mechanism, or get it precompiled and then set in Cmake the values pointing to its include directory and library

press Configure, and then in 'Ungrouped Entries' appear HDF5, select it to be used.

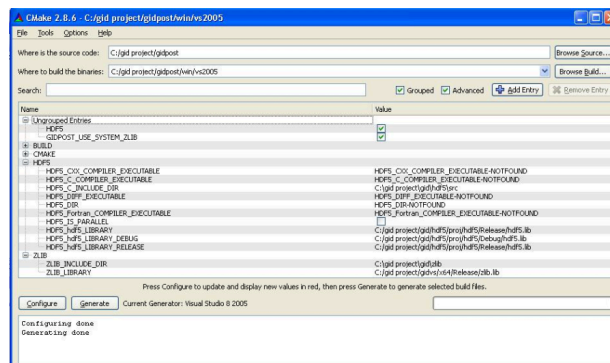
and press Configure once more

Finally in the CMake tree, for Visual Studio is preferred instead /MD the flag /MT (to avoid dependencies of MSCVRTxxx.dll)

and instead /MDd /MTd for the debug version

Press Configure, and then "Generate", to create the VS project, then the project is created

C:\gid project\gidpost\win\vs2005\gidpost.sn1 and can be opened with Visual Studio



As python module

To build GiDPost as Python module you'll need:

- python3
- SWIG: (Simplified Wrapper and Interface Generator) a tool used to connect C or C++ code with scripting languages like python. [SWIG Simplified Wrapper and Interface Generator](#)

To install the required packages in Linux just do the following **as root**:

Ubuntu (Debian and the like):

```
1 apt install python3 swig
```

Scientific Linux (Redhat and the like):

```
1 yum install python3.x86_64 swig.x86_64
```

How to build python module (Linux):

Once you have configured and built the GiDPost library as described [Compiling the library and examples](#), then:

```
1 $ cd ../gidpost/gidpost-swig/  
2 $ make  
3 $ make test
```

Examples of use:

- `gidpost_test.py` - example of writing ascii / hdf5 gid post file with the basic API.
- `gidpost_test_array.py` - example of writing ascii / hdf5 gid post file using the Write*Block API.

Not all gidpost library functions are exposed

List of examples

C examples (in `examples` folder):

- `testpost_fd.c` : uses `GiD_f*` functions and `...Write*Block` functions to write a small 2d triangle mesh with nodal and elemental results.
Usage: `testc_fd [-h] [-f ascii | bin | hdf5] filename_prefix`
- `testpost.c` : uses the deprecated API to write the same example. It has the options to write *MeshGroups* and to create an early-terminate to show the importance of using [GiD_fFlushPostFile](#) .
Usage: `testc [-h] [-abort] [-f ascii | bin | hdf5] [-g] filename_prefix`
- `ex1_sphere_fd.c` : writes a mesh with spheres and circles, with no results, using the `GiD_f*` functions.
- `ex1_sphere.c` : writes a mesh with spheres and circles, with results, using the deprecated API.
- `IGA_test_1.c` : uses the `GiD_WriteNurbSurface*` functions to write a result file for an IGA mesh.

C++ examples (in `examples` folder):

- `TestMultiplePartition.cxx` : in serial, generates and writes meshes and results of a partitioned domain using the `GiD_f*` functions.
- `MultiplePartition/SimulateMultiplePartition.cxx` : in parallel, using threads, generates and writes meshes and results of a partitioned domain using the `GiD_f*` functions.
Usage: `SimulateMultiplePartition [-h] [-v] [-f ascii | bin | hdf5] filename_prefix`

Fortran examples (in `examples` folder):

- `testpost_fd.f90` : Fortran 90 example of writing a quadrilateral mesh with results located and the nodes and at the Gauss Points, with `GiD_f*` and `...Write*Block` functions.
- `testpost.f90` : Fortran 90 example of writing a quadrilateral mesh with nodal results using the deprecated API.
- `testpostfor.f` : Fortran 77 example writing the same example as `testpost.f90` .

Python examples (in `python-swig` folder):

- `gidpost_test_array.py` : writing example of a mesh with triangles and quadrilaterals, and with nodal and elemental results using the `GiD_f*` functions.
- `gidpost_test.py` : writing example of the same mesh and results but with the `...Write*Block` functions.

Changes

From version 2.11 to 2.12

- CMake changes: HDF5 to be optional, Intel fortran compiler detection,
- some refactoring about GiD_fOpenPostResultFile() in Fortran API, utf-8 encoding.
- GiD_fWriteResultBlock() added unit_name option, and optional NULL parameters.
- corrected some warnings and errors, for instance, when using hdf5 1.8.x.

From version 2.10 to 2.11

- HDF5 several corrections when using SWMR mode;
- New GiD_fOpenPostMeshFile_utf8 and GiD_fOpenPostResultFile_utf8 expecting filename utf-8 encoded (and old open functions expected ASCII with current system encoding)
- removed deprecated examples;
- some re-factoring;
- debug build mode provides more error messages.

From version 2.9 to 2.10

- HDF5 writes using **Single Write Multiple Read**, when using hdf5 version 1.10 or higher, i.e. GiD and other programs (if they have also SWMR support) can read the hdf5 file when gidpost is writing it.
<https://docs.hdfgroup.org/archive/support/HDF5/Tutor/swmr.html>.

From version 2.8 to 2.9

- GiD_fWriteCoordinates*Block() Added coordinates block function to write array of coordinates with a single call.
- GiD_fWriteElements*Block() Added elements block function to write array of element's connectivities with a single call.
- GiD_fWriteResultBlock() Added result block function to write array of result values with a single call.
- GiD_PostGetVersion() added to return library version, and if hdf5 is enabled, hdf5 library version too.
- GiD_PostIsThreadSafe(format) : function which returns if format is thread-safe, i.e. several files with this format can be opened and written at the same time. For instance, some HDF5 compilations are not thread-safe, and can not be used to write several hdf5 files at the same time from the same program.
- gidpost-swig/ : to use gidpost library as python module.
- several corrected bugs in library.
- updated some C and Fortran examples to use the new GiD_fWrite*Block functions.
- updated `examples/CmakeLists.txt` to include and build more examples.
- updated documentation.

From version 2.7 to 2.8

- CMake updated and corrected to compile fortran90 example.
- Added functions to allow the user to add mesh and result properties / attributes:

User defined properties defined inside Mesh or Result blocks

HDF5: stored as properties/attributes (Name, value) of the current Mesh/N or Result/N folder

ASCII / raw binary: stored as comments

Name: value

Define the macro COMPASSIS_USER_ATTRIBUTES_FORMAT

to have it like compassis wants:

```

    # ResultUserDefined \"%s\" \"%s\"    or

    # ResultUserDefined \"%s\" %s

int GiD_WriteMeshUserAttribute(GP_CONST char * Name,GP_CONST char * Value);

int GiD_fWriteMeshUserAttribute( GiD_FILE fd, GP_CONST char *Name, GP_CONST char *Value );

int GiD_WriteResultUserAttribute( GP_CONST char *Name, GP_CONST char *Value );

int GiD_fWriteResultUserAttribute( GiD_FILE fd, GP_CONST char *Name, GP_CONST char *Value );

```

From version 2.6 to 2.7

- Added support for MeshGroups (BeginMeshGroup(), End..., BeginOnMeshGroup(), End...) in HDF5 as used in GiD, useful for dynamic meshes, refinement, multi-stage simulation, etc.
- All non implemented functions for HDF5 now return -1, i.e. all GiD_fXXX() functions.

From version 2.5 to 2.6

- Added support for ComplexMatrix Result types.
- More exhaustive example and some corrections.

From version 2.4 to 2.5

- Default format to print real numbers changed from "%g" to "%.9g" increasing the amount of significant digits to be printed in ASCII.
- New function GiD_PostSetFormatReal to allow change the default format of real numbers.

From version 2.3 to 2.4

- Allow write OnNurbsSurface results for iso-geometrical analysis

From version 2.1 to 2.3

- Allow use HD5 from FORTRAN interface
- Fixed bug with mesh group

From version 2.0 to 2.1

- Allow write complex scalar and complex vector results
- Write units of mesh or results
- Enhanced FORTRAN 77 and 90 interface to be more compatible.

From version 1.7.2 to 2.0

- New set of functions "GiD_fxxx" that allow specify the file to write, in case of have multiple files of mesh or results.
- Library rewritten in C avoiding C++ to be more compatible linking with other languages.
- Optional define of HDF5 to enable write postprocess files using the HDF5 library

Old changes

2008/09/30

- Updated cfortran to release 4.4

2008/02/25

- changing to release 1.8
- M build/Jamroot, source/gidpost.cpp source/gidpostInt.h: removed C++ dependencies.
2008/01/29
- source/gidpost.cpp:
 - GiD_OpenPostMeshFile should not write a header. Thanks to Janosch Stascheit <janosch.stascheit@rub.de>
 - invalid access in GiD_BeginMeshGroup: should ensure that a valid mesh file object is available so etMeshFile() should be called before writing. Thanks to Janosch Stascheit<janosch.stascheit@rub.de> for providing the fix.
2008/01/18
 - source/gidpost.cpp: GiD_BeginOnMeshGroup should be "OnGroup\"%s\" instead of "Group \"%s\"". Thanks to Janosch Stascheit <janosch.stascheit@rub.de>
2007/06/13
 - source/gidpost.cpp: GiD_WritePlainDefMatrix can be written in binary files.
2007/06/11
 - GiD_Sphere and GiD_Circle
2007/06/11
 - source/gidpost.cc: Write2DMatrix is now available on binary format, Z component is set to 0.
2007/01/23:
 - source/gidpostfor.c: bug reported by a user: GiD_Begin3DMatResult must be ncomp = SetupComponents(6, not 4Factoring fortran declaration in gidpostfor.h
Defining symbols for both gcc and icc.
2006/06/25
- doc/*:
- sources/gidpost.cpp,gidpost.h: included pyramid element.
2006/06/25
- tag 1.7: reL1_7
- examples/testpost.c: fixed a bug, AsciiZipped must has the mesh file in ascii format.
2006/05/25
- gidpost.h: dllexport.
2006/05/18
- all: GiD_BeginMeshColor, GiD_BeginMeshGroup, GiD_EndMeshGroup, GiD_BeginOnMeshGroup, GiD_EndOnMeshGroup
2005/10/24
- doc:
- gidpost.h : documented GiD_ResultDescriptionDim
2005/10/21
- gidpost* : added support to new type specification for result groups. The type could be Type:dim, where Type is one valid result type and dim is a number giving a valid dimension. Updated fortran interface for recently added functions.
2005/10/07
- all: new version 1.60. "Result Groups" can now be written. The macro USE_CONST can be used to enable (if defined) or disable (if not defined) the use of const char* arguments.
2005/09/22
- all: ha sido un error cambiar de 'char *' a 'const char *'
2005/09/22
- gidpost.cpp: flag_begin_values = 0; must be set in GiD_BeginResultGroup.

2005/09/21

- gidpost.h:
- gidpost.cpp: values_location_{min,max} not needed, CBufferValue
- gidpostInt.cpp: now controls the types of results written within
- gidpostInt.h: a ResultGroup.

2005/09/16

- gidpost.cpp: in GiD_WriteVectorModule, when writing if we are inside a ResultGroup block the module value is ignored.
- gidpostInt.cpp: in CBufferValues::WriteValues, buffer overflow is checked after checking for change in location.

2005/09/15

- gidpost.cpp:
- gidpostInt.cpp:
- gidpostInt.h: fixed a bug when writing results in a "Result Group", vectors can be 3 or 4 component long.
- examples/testpost.c : bug when passing location id in result group sample code.

2005/06/27

- source/gidpost.{h,cpp} including Prism element
- doc/gidpost.{html,subst} including Prism element documentation.
- all: change to version 1.52

2005/05/09

- all: constification, version change from 1.5 to 1.51

2005/01/04

- source/gidpostInt.cpp: fixed a bug when writing 3D vectors.

2005/01/07

- source/gidpost.cpp added const char and a filter to remove double quotes from names which can cause problems within gid.

2003/07/29

- doc/gidpost.html : commented the GiD_ResultUnit
- source/gidpost.h : interface because it is not
- source/gidpostfor.c: supported yet inside GiD.

2003/07/28

- doc/gidpost.html: updated documentation for release 1.5
- binary/gidpost.lib: binary release for windows

2003/07/15

- gidpost.h:
- gidpost.cpp:
- gidpostfor.c: new function 'GiD_WriteCoordinates2D' (to write coordinates in 2D but only in ASCII format, the function can be used in binary but the library provide the 'z' coordinate a zero.

2003/07/15

- gidpost.h:
- gidpost.cpp: removed 'char * UnitName' argument , new functions: GiD_BeginResultHeader, GiD_ResultRange, GiD_ResultComponents, GiD_ResultUnit, GiD_BeginResultGroup, GiD_ResultDescription, GiD_ResultValues, GiD_FlushPostFile. Validation in debug mode.
- gidpostInt.h:

- gidpostInt.cpp: new member
CPostFile::WriteValues(int id, int n, double * buffer), new class
CBufferValues to write the values in a result group, validation in
debug mode.
- gidpostfor.c: updated fortran interface for the new
functionality.
- testpost.c:
- testpostfor.f: added test code for the new functionality.

Introduction

gidpost is a set of functions (library) for writing postprocess results for GiD in ASCII or binary compressed format.

This software is copyrighted by CIMNE www.gidsimulation.com. The software can be used freely under the terms described in `license.terms`, all terms described there apply to all files associated with the software unless explicitly disclaimed in individual files. Particular terms apply to the third party code, "cfortran.h", which has its own distribution policy (please read the "cfortran.doc" for this code).

This description assumes that the reader is familiar with the postprocess terminology.
For further details please check the online help available in GiD (Postprocess data files chapter).

The library was implemented taking into account two of the must widely used development environments: C/C++ and FORTRAN.

Here we are going to describe how to compile and use the library. At the end the reference of the library functions can be found.

Using the library

C / C++ language:

Include the file header file gidpost.h to use the library gidpost

```
#include "gidpost.h"
```

Small examples:

- testpost.c, is provided to show the use of the library from C/C++.
- testpost_fd.c that use the *f functions where the file handler is explicitly specified (to allow multiple files)
- IGA_test_1.c, to show the use of OnNurbsSurface isogeometric results

FORTRAN language:

A small example, called testpost.f, is provided to show the use of the library with FORTRAN 77
the same case is implemented with FORTRAN 90 in the example testpost.f90, using the gidpost.F90 interface module

Note: The gidpost fortran 90 module use the new ISO_C_BINDING intrinsic module defined in the Fortran 2003 standard.
(Tested in intel fortran >= 10.1, Sun/Oracle studio >= 12.1, and GNU gfortran >= XX)

Link libraries: gidpost.lib zlib.lib hdf5.lib (Linker - Input - Additional Dependencies)
and remember set the path to these libraries (e.g. Linker - General - Additional Library Directories - ..\..\binaries\win\x32)

Note: the hdf5.lib library is only required to write files with HDF5 format.

Note: For Linux platforms the extension of libraries is .a instead of .lib

Python module:

Inside the folder gidpost/gidpost-swig there are a couple of examples on how to use gidpost python module.

Once the module has been built and installed, then you can use it like this:

gidpost_test.py

```
$ python3
Type "help", "copyright", "credits" or "license" for more information.
>>> import gidpost
gidpost.GiD_PostInit()
file_id = gidpost.GiD_fOpenPostResultFile( "gidpost-test-array.post.h5", gidpost.GiD_PostHDF5)

# Create mesh with num_nodes, num_triangles and num_quads
# coords_list_ids = array with the id's of the nodes ( gidpost.new_intArray(...))
# coords_list_xyz = array with the x y z coordinades of the nodes x1 y1 z1 x2 y2 z2 .... ( gidpost.new_doubleArray(...))
# triangs_list_ids = array with the id's of the triangles
# triangs_connectivities = array with the connectivities of the triangles
# quads_list_ids = array with the id's of the quads
# quads_connectivities = array with the connectivities of the quads

gidpost.GiD_fBeginMeshColor( file_id, "test gp_py triangles", gidpost.GiD_3D, gidpost.GiD_Triangle, 3, 0.
fail = gidpost.GiD_fWriteCoordinatesIdBlock( file_id, num_nodes, coords_list_ids, coords_list_xyz)
fail = gidpost.GiD_fWriteElementsIdBlock( file_id, num_triangles, triangs_list_ids, triangs_connectivities)

gidpost.GiD_fBeginMeshColor( file_id, "test gp_py quadrilaterals", gidpost.GiD_3D, gidpost.GiD_Quadrilate
fail = gidpost.GiD_fWriteElementsIdBlock( file_id, num_quads, quads_list_ids, quads_connectivities)

# Create results
# scalar_nodes = array of a scalar result for all nodes
# elements_list_ids = array with the id's of the elements
# scalar_elemental = array of a elemental result for all elements

list_comp_names = gidpost.new_stringArray( 0);
fail = gidpost.GiD_fWriteResultBlock( file_id, "test nodal", "gidpost", 1.0,
gidpost.GiD_Scalar, gidpost.GiD_OnNodes, "", "", 0, list_comp_names,
num_nodes, coords_list_ids, 1, scalar_nodes)

fail = gidpost.GiD_fWriteResultBlock( file_id, "test elemental", "gidpost", 1.0,
```



```

gidpost.GiD_Scalar, gidpost.GiD_OnGaussPoints, "GP_ELEMENT_1", "", 0
num_elements, elements_list_ids, 1, scalar_elemental)

fail = gidpost.GiD_fClosePostResultFile( file_id)
fail = gidpost.GiD_PostDone()

gidpost.delete_intArray( coords_list_ids)
gidpost.delete_intArray( triangs_list_ids)
gidpost.delete_intArray( triangs_connectivities)
gidpost.delete_intArray( quads_list_ids)
gidpost.delete_intArray( quads_connectivities)
gidpost.delete_intArray( elements_list_ids)

gidpost.delete_doubleArray( coords_list_xyz)
gidpost.delete_doubleArray( scalar_nodes)
gidpost.delete_doubleArray( scalar_elemental)

```

Functions references

Library use

Before call other library functions the [GiD_PostInit](#) function must be called (and [GiD_PostDone](#) when finishing its use)

There are two collection of functions: multi-file and single-file

- [GiD_f*](#) functions with the 'f' letter, and an extra attribute for the file handler (new interface from library version 2.0)
- [GiD_*](#) functions without 'f' letter. There is an implicit single file (old interface)

The HDF5 format ([GiD_fOpenPostResultFile](#)) is only available with old single-file style, the multiple-file interface will be implemented in future versions.

GiD_PostInit

This procedure must be invoked once before use other commands

It initialize some internal structures. Must be called from the main thread of the program.

GiD_PostDone

This procedure must be invoked once at the end.

It release the internal structures initialized by [GiD_PostInit](#). Must be called from the main thread of the program.

GiD_PostGetVersion

This functions returns a string with the version number of the library. If the library was compiled with HDF5 support it also returns the version number of the HDF5 library and whether it is thread-safe or not.

```
1  printf( "version = %s\n", GiD_PostGetVersion() );  
2  // --> version = GiDpost 2.8 - HDF5 1.12.2 ( Thread-safety disabled )
```

GiD_PostIsThreadSafe

```
int GiD_PostIsThreadSafe( GiD_PostMode format ); // returns -1 on error
```

Description: returns 1 if `format` can be used thread-safe or not. Some HDF5 compilations are not thread-safe. Thread-safe is required to open and write on several separated gidpost files at the same time. For instance, some HDF5 compilations are not thread-safe, and can not be used to write several hdf5 files at the same time from the same program.

Parameters:

`GiD_PostMode format`

`format` = `GiD_PostAscii` | `GiD_PostAsciiZipped` | `GiD_PostBinary` | `GiD_PostHDF5`

Example:

C/C++

```
1  if ( GiD_PostIsThreadSafe( GiD_PostHDF5) == 0) {  
2      printf( "gidpost HDF5 is not thread safe, please open and write on a single HDF5 file.\n");  
3  }
```

GiD_PostSetFormatReal

int GiD_PostSetFormatReal(GP_CONST char * format)

Description: overwrite the default format to print real numbers in ASCII strings (by default, this is "%.9g")

Parameters:

char* format

a valid C/C++ format to print a real number

Example:

C/C++

```
GiD_PostSetFormatReal("%15.5f");
```


GiD_PostGetFormatReal

```
GP_CONST char *GiD_PostGetFormatReal( void );
```

Description: This function returns the format used to print real numbers in ASCII strings (by default, returns “%.9g”)
This format can be set with [GiD_PostSetFormatReal](#)

GiD_PostGetFormatStep

```
GP_CONST char *GiD_PostGetFormatStep( void );
```

Description: This function returns the format used to print the *step value* in ASCII strings. It always returns “%.16g”. This is done so, to not truncate time step values when writing strings in ASCII files.

Mesh file functions

The mesh file is not necessary if the postprocess mesh is the same used in GiD preprocess.

GiD_fOpenPostMeshFile

```
GiD_FILE GiD_fOpenPostMeshFile(const char * FileName,GiD_PostMode Mode );
```

```
GiD_FILE GiD_fOpenPostMeshFile_utf8(const char * FileName,GiD_PostMode Mode );
```

Description: Open a new post mesh file, and return its file handler.

Parameters:

char* FileName: name of the mesh file (*.post.msh),

Note: the function _utf8 expects the name utf-8 encoded (more robust in different languages)

GiD_PostMode Mode

GiD_PostAscii=0 for ascii output

GiD_PostAsciiZipped=1 for compressed ascii output

GiD_PostBinary=2 for compressed binary output

GiD_PostHDF5=3 for HDF5 output (the library must be compiled with HDF5 support)

GiD_FILE and unsigned int that identify the channel.

Example:

C/C++

```
fd=GiD_fOpenPostMeshFile( "testpost.post.msh", GiD_PostAscii);
```

FORTRAN

```
fd=GiD_FOPENPOSTMESHFILE('testpost.post.msh',0)
```

GiD_fClosePostMeshFile

```
int GiD_fClosePostMeshFile(GiD_FILE fd);
```

Description: Close the post mesh file

Parameters:

GiD_FILE fd, the handler returned by GiD_fOpenPostMeshFile

Example:

C/C++

```
GiD_fClosePostMeshFile(fd);
```

FORTRAN

```
CALL GID_FCLOSEPOSTMESHFILE(fd)
```

GiD_fBeginMeshGroup

int GiD_fBeginMeshGroup(GiD_FILE fd, const char* Name);

Description: This function open a group of mesh. This enable specifying multiples meshes withing the group.

Parameters:

GiD_FILE fd the file descriptor

char* Name

Name of the group. This name can be used later when givin the set of results that apply to this group, see GiD_fBeginOnMeshGroup.

Example:

C/C++

```
GiD_fBeginMeshGroup(fd, "steps 1, 2, 3 and 4" );
```

FORTRAN

```
CALL GID_FBEGINMESHGROUP(fd, "steps 1, 2, 3 and 4" )
```

GiD_fEndMeshGroup

```
int GiD_fEndMeshGroup(GiD_FILE fd);
```

Description: This function close the previously opened group of mesh. See GiD_fBeginMeshGroup.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndMeshGroup(fd);
```

FORTRAN

```
CALL GiD_FENDMESHGROUP(fd)
```

GiD_fBeginMesh, GiD_fBeginMeshColor

```
int GiD_fBeginMesh(GiD_FILE fd,const char* MeshName,GiD_Dimension Dim,GiD_ElementType EType,int NNode);
int GiD_fBeginMeshColor(GiD_FILE fd, const char* MeshName,GiD_Dimension Dim, GiD_ElementType EType,int NNode,double Red,
double Green, double Blue);
```

Description: Begin a new mesh. After that you should write the nodes and the elements. When writing the elements it is assumed a connectivity of size given in the parameter NNode. The second function allows to specify a color for the mesh where each component take values on the interval [0,1].

Parameters:

GiD_FILE fd : the file descriptor

char* MeshName : Name of the mesh

GiD_Dimension Dim :

- GiD_2D=2 for a 2D mesh (is assumed coordinates z=0)
- GiD_3D=3 for a 3D mesh

GiD_ElementType EType :

- GiD_Point=1 for a 1 noded-element
- GiD_Linear=2 for a line element
(the number of nodes can be 2 for a linear case or 3 for the quadratic case)
- GiD_Triangle=3 for a triangle elements
(the number of nodes can be 3 for a linear case or 6 for the quadratic case)
- GiD_Quadrilateral=4 for a quadrilateral elements
(the number of nodes can be 4 for a linear case and 8 or 9 for the quadratic case)
- GiD_Tetrahedra=5 for a tetrahedral element
(the number of nodes can be 4 for a linear case or 10 for the quadratic case)
- GiD_Hexahedra=6 for a hexahedral element
(the number of nodes can be 8 for a linear case and 20 or 27 for thequadratic case)
- GiD_Prism=7 for a Prism element
(the number of nodes can be 6 for the linear case or 15 for the quadratic case)
- GiD_Pyramid=8 for a Pyramid element
(the number of nodes can be 5 for the linear case or 13 for the quadratic case)
- GiD_Sphere=9 for a sphere element with 1 node and a radius
- GiD_Circle=10 for circle element, with 1 node, radius and a plane normal

int NNode : Number of nodes of this type of element. The element type an nnods must be constant for all mesh elements, but it is valid to define more that one mesh.

Example:

C/C++

```
GiD_fBeginMesh(fd,"TestMsh",GiD_2D,GiD_Triangle,3);
```

FORTRAN

```
CALL GiD_fBEGINMESH(fd,'quadmesh',2,4,4)
```


GiD_fMeshUnit

```
int GiD_fMeshUnit(GiD_FILE fd, const char* UnitName);
```

Description: Define the unit string associated to the mesh coordinates

Parameters:

GiD_FILE fd the file descriptor

char* UnitName ,the string of the unit (length magnitude)

Example:

C/C++

```
GiD_fBeginMesh(fd, "TestMsh", GiD_2D, GiD_Triangle, 3);  
GiD_fMeshUnit(fd, "m");
```

FORTRAN

```
CHARACTER*10 UNITNAME  
UNITNAME = 'm'  
GID_FMESHUNIT(fd, UNITNAME)
```

GiD_fEndMesh

```
int GiD_fEndMesh(GiD_FILE fd);
```

Description: End the current mesh.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndMesh(fd);
```

FORTRAN

```
CALL GID_ENDMESH(fd)
```

GiD_fBeginCoordinates

```
int GiD_fBeginCoordinates(GiD_FILE fd);
```

Description: Start a coordinate block in the current mesh. All nodes coordinates must be written between a to this function and GiD_fEndCoordinates().

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fBeginCoordinates(fd);
```

FORTRAN

```
CALL GiD_FBEGINCOORDINATES(fd)
```

GiD_fEndCoordinates

```
int GiD_fEndCoordinates(GiD_FILE fd);
```

Description: Close the current coordinate block.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndCoordinates(fd);
```

FORTRAN

```
CALL GID_FENDCOORDINATES(fd)
```

GiD_fWriteCoordinates

```
int GiD_fWriteCoordinates(GiD_FILE fd, int id, double x, double y, double z);  
int GiD_fWriteCoordinates2D(GiD_FILE fd, int id, double x, double y);
```

Description: Write a coordinate member at the current Coordinates Block.
If the mesh dimension is 2D then you can use GiD_fWriteCoordinates2D

Parameters:

GiD_FILE fd the file descriptor

int id

Node number identifier (starting from 1, is recommended to avoid jumps in the numeration)

double x,double y,double z

Cartesian coordinates

Example:

C/C++

```
int id=1;  
double x=3.5,y=1.5e-2,z=0;  
GiD_fWriteCoordinates(fd,id,x,y,z);
```

FORTRAN

```
REAL*8 rx, ry  
INTEGER*4 idx  
idx=1  
rx=3.5  
ry=-4.67  
CALL GID_FWRITECOORDINATES(fd,idx,rx,ry,0.0)
```

GiD_fBeginElements

```
int GiD_fBeginElements(GiD_FILE fd);
```

Description:Start a elements block in the current mesh.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fBeginElements(fd);
```

FORTRAN

```
CALL GID_FBEGINELEMENTS(fd)
```

GiD_fEndElements

```
int GiD_fEndElements(fd);
```

Description: Close the current elements block.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndElements(fd);
```

FORTRAN

```
CALL GiD_FENDELEMENTS(fd)
```

GiD_fWriteElement

```
int GiD_fWriteElement(GiD_FILE fd, int id, int nid[]);
int GiD_fWriteSphere(GiD_FILE fd, int id, int nid, double r);
int GiD_fWriteCircle(GiD_FILE fd, int id, int nid, double r, double nx, double ny, double nz);
```

Description: Write an element member at the current Elements Block.

Parameters:

GiD_FILE fd the file descriptor
int id Element number identifier
int nid[] connectivities of the element, the vector dimension must be equal to the NNode parameter given in the previous call to GiD_fBeginMesh
r the radius (sphere or circle element only)
nx,ny,nz unitary normal of the circle plane (circle element only)

Example:

C/C++

```
int id=2;
int nid[3];
nid[0]=4; nid[1]=7; nid[2]=3;
GiD_fWriteElementMat(fd,id,nid);
```

FORTRAN

```
INTEGER *4 nid(1:3)
INTEGER*4 idx
idx=2
nid(1)=4
nid(2)=7
nid(3)=3
CALL GiD_FWRITEELEMENT(fd,idx,nid)
```


GiD_fWriteElementMat

```
int GiD_fWriteElementMat(GiD_FILE fd, int id, int nid[]);  
int GiD_fWriteSphereMat(GiD_FILE fd, int id, int nid, double r, int mat);  
int GiD_fWriteCircleMat(GiD_FILE fd, int id, int nid, double r, double nx, double ny, double nz, int mat);
```

Description: Similar to GiD_WriteElement, but the last additional integer nid corresponds to a material number.

GiD_fWriteCoordinatesBlock (array of coordinates)

```
int GiD_fWriteCoordinatesBlock( GiD_FILE fd, int num_points, GP_CONST double *xyz_array );  
int GiD_fWriteCoordinatesIdBlock( GiD_FILE fd, int num_points, GP_CONST int *list_ids, GP_CONST double  
*xyz_array );
```

Description:

Write the coordinates array of the current mesh or of all meshes. These functions include `GiD_fBeginCoordinates()` and `GiD_fEndCoordinates()`.

If `GiD_fWriteCoordinatesBlock()` is used, i.e. no list of node id's are provided, then the node id's are assumed to be `1..num_points`.

Parameters:

`list_ids` is the array of node id's and expected to be `num_points` in size.

`xyz_array` is the array consecutive `x`, `y` and `z` coordinates of the `num_points` nodes, and it is expected to have `num_points * 3` doubles.

Example:

C/C++ as in `testpost_fd.c` :

```
1  int ids[ NUM_NODES ];  
2  double xyz[ NUM_NODES * 3 ];  
3  for ( int i = 0; i < NUM_NODES; i++ ) {  
4      ids[ i ] = G_nodes[ i ].id;  
5      xyz[ i * 3 + 0 ] = G_nodes[ i ].x;  
6      xyz[ i * 3 + 1 ] = G_nodes[ i ].y;  
7      xyz[ i * 3 + 2 ] = G_nodes[ i ].z;  
8  }  
9  GiD_fWriteCoordinatesIdBlock( fdm, NUM_NODES, ids, xyz );
```

GiD_fWriteElementsBlock (array of connectivities)

```
int GiD_fWriteElementsBlock( GiD_FILE fd, int num_elements, GP_CONST int *connectivities );
int GiD_fWriteElementsIdBlock( GiD_FILE fd, int num_elements, GP_CONST int *list_ids, GP_CONST int
*connectivities );

int GiD_fWriteElementsMatBlock( GiD_FILE fd, int num_elements, GP_CONST int *connectivities, GP_CONST int
*lst_material_id );

int GiD_fWriteElementsIdMatBlock( GiD_FILE fd, int num_elements, GP_CONST int *list_ids, GP_CONST int
*connectivities, GP_CONST int *lst_material_id );
```

Description:

Write the connectivities array of the elements of the current mesh. These functions include `GiD_fBeginElements()` and `GiD_fEndElements()`. All elements are of the same type `EType` as described in [GiD_fBeginMesh, GiD_fBeginMeshColor](#).

If `GiD_fWriteElementsBlock()` or `GiD_fWriteElementsMatBlock()` are used, i.e. no list of element's id's are provided, then the element's id's are assumed to be `1..num_elements`.

If `GiD_fWriteElementsBlock()` or `GiD_fWriteElementsIdBlock()` are used, i.e. no list of element's materials are provided, then the elements id's are assumed to be `0`.

Parameters:

`list_ids` is the array of element's id's and expected to be `num_elements` in size.

`list_material_id` is the array of element's material's and expected to be `num_elements` in size.

`connectivities` is the array consecutive element's connectivities of the `num_elements`, and it is expected to have `num_elements * Nnode` doubles. Being `Nnode` the number of nodes per element as specified in [GiD_fBeginMesh, GiD_fBeginMeshColor](#).

Example:

C/C++ as in `testpost_fd.c`:

```
1  int ids[ NUM_ELEMS ];
2  int connectivities[ NUM_ELEMS * 3 ];
3  int mat_ids[ NUM_ELEMS ];
4  for ( int i = 0; i < NUM_ELEMS; i++ ) {
5      ids[ i ] = G_elems[ i ].id;
6      connectivities[ i * 3 + 0 ] = G_elems[ i ].n1;
7      connectivities[ i * 3 + 1 ] = G_elems[ i ].n2;
8      connectivities[ i * 3 + 2 ] = G_elems[ i ].n3;
9      mat_ids[ i ] = 2;
10 }
11 GiD_fWriteElementsIdMatBlock( fdm, NUM_ELEMS, ids, connectivities, mat_ids );
```

Results file functions

GiD_fOpenPostResultFile

GiD_FILE GiD_fOpenPostResultFile(const char * FileName, GiD_PostMode Mode);

GiD_FILE GiD_fOpenPostResultFile_utf8(const char * FileName, GiD_PostMode Mode);

Description: Open a new post result file and return its file handler, to allow subsequent call to functions write the information into the desired file. If there is no mesh file opened then the output of the mesh is written into this file also.

Parameters:

char* FileName: name of the mesh file (*.post.res)

Note: the function _utf8 expects the name utf-8 encoded (more robust in different languages)

GiD_PostMode Mode

GiD_PostAscii=0 for ascii output

GiD_PostAsciiZipped=1 for compressed ascii output

GiD_PostBinary=2 for compressed binary output

GiD_PostHDF5=3 for HDF5 output (the library must be compiled with HDF5 support)

Note: In binary output, the mesh must be inside the same file as the results, you should not call GiD_fOpenPostMeshFile and GiD_fClosePostMeshFile in that case.

Example:

C/C++

```
fd=GiD_fOpenPostResultFile( "testpost.post.bin",GiD_PostBinary);
```

FORTRAN

```
fd=GID_OPENPOSTRESULTFILE('testfortran.post.res',2)
```

GiD_fFlushPostFile

```
int GiD_fFlushPostFile(GiD_FILE fd);
```

Description:

Flushes all pending output into the post-process file. This function should be called when necessary (particularly with *GiD_PostAsciiZipped* or *GiD_PostBinary* formats) and with caution as it impacts the performance of the whole program.

It is suggested to be called between Analysis Steps, to ensure all results of the calculated step are written to disk.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fFlushPostFile(fd);
```

FORTRAN

```
CALL GID_FFLUSHPOSTFILE(fd)
```

GiD_fClosePostResultFile

```
int GiD_fClosePostResultFile(GiD_FILE fd);
```

Description: Close the post results file

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fClosePostResultFile(fd);
```

FORTRAN

```
CALL GID_CLOSEPOSTRESULTFILE(fd)
```

GiD_fBeginGaussPoint

int GiD_fBeginGaussPoint(GiD_FILE fd, const char* name, GiD_ElementType EType, const char* MeshName, int GP_number, int NodesIncluded, int InternalCoord);

Description: Begin Gauss Points definition. The gauss point definition should have a name which may be referenced in further results blocks. The gauss points could be internal (InternalCoord=1) or given (InternalCoord=0). If the gauss points are given then the list of its natural coordinates should be written using the function GiD_fWriteGaussPoint2D or GiD_fWriteGaussPoint3D depending on the dimension of the element type.

Parameters:

GiD_FILE fd the file descriptor

char* name

Name to reference this gauss points definition

GiD_ElementType EType

GiD_Point=1 for a 1 noded-element

GiD_Linear=2 for a line element

GiD_Triangle=3 for a triangle element

GiD_Quadrilateral=4 for a quadrilateral element

GiD_Tetrahedra=5 for a tetrahedral element

GiD_Hexahedra=6 for a hexahedral element

GiD_Prism=7 for a prism element

GiD_Pyramid=8 for a pyramid element

GiD_Sphere=9 for sphere element

GiD_Circle=10 for circle element

char* MeshName

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.

int GP_number

number of gauss points per element. The GiD internal accepted number should be:

1, 3, 6 for Triangles;

1, 4, 9 for quadrilaterals;

1, 4 for Tetrahedras;

1, 8, 27 for hexahedras;

1 or 6 for prism;

1 or 5 for pyramid and

1, ... n points equally spaced over lines.

int NodesIncluded

Can be 0 for nodes not included or 1 for included. Only used for gauss points on Linear elements which indicate whether the end nodes of the Linear element are included in the number of gauss points per element count or not.

int InternalCoord

Can be 0 for given coordinates or 1 for internal GiD gauss points location.

Example:

C/C++

```
GiD_fBeginGaussPoint(fd, "GPtria", GiD_Triangle, NULL, 1, 0, 1);
```

FORTRAN

```
CHARACTER*4 NULL
NULL = CHAR(0)//CHAR(0)//CHAR(0)//CHAR(0)
CALL GID_FBEGINGAUSSPOINT(fd, 'GPtria', 3, NULL, 1, 0, 1);
```


GiD_fEndGaussPoint

```
int GiD_fEndGaussPoint(GiD_FILE fd);
```

Description: End current Gauss Points definition

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndGaussPoint(fd);
```

FORTRAN

```
CALL GID_FENDGAUSSPOINT(fd)
```

GiD_fWriteGaussPoint

```
int GiD_fWriteGaussPoint3D(GiD_FILE fd, double x, double y, double z);  
int GiD_fWriteGaussPoint2D(GiD_FILE fd, double x, double y);
```

Description: Write internal gauss point local coordinates (only required if InternalCoord=0)

Parameters:

GiD_FILE fd the file descriptor
double x,double y,double z
Cartesian gauss points local coordinates

Example:

C/C++

```
double x=3.5;  
double y=-7;  
GiD_fWriteGaussPoint2D(fd,double x,double y);
```

FORTRAN

```
REAL*8 x, y  
rx=3.5  
ry=-7  
CALL GiD_FWRITEGAUSSPOINT2D(fd,x,y)
```

GiD_fBeginRangeTable

```
int GiD_fBeginRangeTable(GiD_FILE fd, const char * name);
```

Description: Begin a Range Table definition. With a range table you can group the result values into intervals and label each interval with a name. Inside GiD this can be visualized with a contour range.

Parameters:

GiD_FILE fd the file descriptor
char* name
name identifier

Example:

C/C++

```
GiD_fBeginRangeTable(fd, "table1");
```

FORTRAN

```
CALL GiD_FBEGINRANGETABLE(fd, 'table1')
```

GiD_fEndRangeTable

```
int GiD_fEndRangeTable(GiD_FILE fd);
```

Description: End a Range Table definition.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndRangeTable(fd);
```

FORTRAN

```
CALL GiD_FENDRANGETABLE(fd)
```

GiD_fWriteRange

```
int GiD_fWriteRange(GiD_FILE fd,double min,double max,const char* name);  
int GiD_fWriteMinRange(GiD_FILE fd,double max,const char* name);  
int GiD_fWriteMaxRange(GiD_FILE fd,double min,const char* name);
```

Description:Write Range functions. Must be between GiD_fBeginRangeTable and GiD_fEndRangeTable.

fWriteMinRange : write a range with an implicit minimum value, the minimum absolute in the result set.

fWriteRange : write an explicit range.

fWritemaxRange: write a range with an implicit maximum value, the maximum absolute in the result set.

Parameters:

GiD_FILE fd the file descriptor

double min, double max

Values to define a interval

char* name

string associated to be showed for this interval

Example:

C/C++

```
GiD_fWriteRange(fd,0.0,100.0,"Normal");
```

FORTRAN

```
CALL GID_FWRITERANGE(fd,0.0,100.0,'Normal')
```

GiD_fBeginResult (DEPRECATED)

Note: these functions are deprecated because can't provide the result unit, to specify unit fBeginResultHeader must be used.

```
int GiD_fBeginResult(GiD_FILE fd, const char * Result, const char * Analysis, double step, GiD_ResultType Type, GiD_ResultLocation Where, const char *
GaussPointsName, const char * RangeTable, int compc, const char * compv[]);
int GiD_fBeginScalarResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char* RangeTable, char*
Comp);
int GiD_fBeginVectorResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char* RangeTable, char*
Comp1, char* Comp2, char* Comp3, char* Comp4);
int GiD_fBegin2DMatResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char* RangeTable, char*
Comp1, char* Comp2, char* Comp3);
int GiD_fBegin3DMatResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char* RangeTable, char*
Comp1, char* Comp2, char* Comp3, char* Comp4, char* Comp5, char* Comp6);
int GiD_fBeginPDMMatResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char*
RangeTable, char* Comp1, char* Comp2, char* Comp3, char* Comp4);
int GiD_fBeginMainMatResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char*
RangeTable, char* Comp1, char* Comp2, char* Comp3, char* Comp4, char* Comp5, char* Comp6, char* Comp7, char* Comp8, char* Comp9, char* Comp10, char*
Comp11, char* Comp12);
int GiD_fBeginLAResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char* RangeTable, char*
Comp1, char* Comp2, char* Comp3);
int GiD_fBeginComplexScalarResult(GiD_FILE fd, char* Result, char* Analysis, double step, GiD_ResultLocation Where, char* GaussPointsName, char*
RangeTable, GP_CONST char * Re, GP_CONST char * Im);
int GiD_fBeginComplexVectorResult(GiD_FILE fd, GP_CONST char * Result, GP_CONST char * Analysis, double step, GiD_ResultLocation Where, GP_CONST
char * GaussPointsName, GP_CONST char * RangeTable, GP_CONST char * Rex, GP_CONST char * Imx, GP_CONST char * Rey, GP_CONST char * Imy,
GP_CONST char * Rez, GP_CONST char * Imz);
```

Description: Begin Result Block. This function open a result block.

These functions are deprecated because can't provide the result unit, to specify unit fBeginResultHeader must be used.

Parameters:

See [GiD_fBeginResultHeader](#)

char* RangeTable

A valid Range table name or NULL

int compc

The number of component names or 0

char* compv[]

array of 'compc' strings to be used as component names

Example:

C/C++

```
GiD_fBeginResult(fd, "Result", "Static", 1.0d0, GiD_Scalar, GiD_OnNodes, NULL, NULL, 0, NULL);
```

FORTRAN

```
CHARACTER*4 NULL
NULL = CHAR(0)//CHAR(0)//CHAR(0)//CHAR(0)
CALL GiD_fBEGINSCALARRESULT(fd, 'Result', 'Analy.', 1.0d0, 0, NULL, NULL, NULL)
```

GiD_fBeginResultHeader

int GiD_fBeginResultHeader(GiD_FILE fd,const char* Result,const char* Analysis,double step,GiD_ResultType Type,GiD_ResultLocation Where,const char* GaussPointsName);

Description:

Begin Result Block. This function open a result block. Only the result, analysis, location and location name are provided in the function call.

The other result attributes as range table or components names are provided in a separated function calls. See [GiD_fResultComponents](#), [GiD_fResultRange](#) and [GiD_fResultUnit](#).

Before set the results [GiD_fResultValues](#) must be called

Parameters:

GiD_FILE fd the file descriptor

char* Result

a name for the Result, which will be used for menus.

char* Analysis

the name of the analysis of this Result, which will be used for menus.

double step

the value of the time step inside the analysis "analysis name".

(for multiple steps results)

GiD_ResultType Type

The type of defined result:

GiD_Scalar = 0,

GiD_Vector,

GiD_Matrix,

GiD_PlainDeformationMatrix,

GiD_MainMatrix,

GiD_LocalAxes,

GiD_ComplexScalar,

GiD_ComplexVector,

GiD_ComplexMatrix

GiD_ResultLocation

Where the location of the results

GiD_OnNodes=0

GiD_OnGaussPoints=1

GiD_OnNurbsLine=2

GiD_OnNurbsSurface=3

char* GaussPointsName

If Where is GiD_OnGaussPoints a "location name" (predefined in GiD_BeginGaussPoint) should be entered.

Example:

C++:

```
GiD_fBeginResultHeader(fd,"Result","Static",1.0d0,GiD_Scalar,GiD_OnNodes,NULL);
GiD_fResultUnit(fd,'m/s');
GiD_fResultValues(fd)
for(int i=1;i<10;i++){
    GiD_fWriteScalar(fd,i,value[i])
}
GiD_fEndResult(fd)
```

GiD_fResultRange

```
int GiD_fResultRange(GiD_FILE fd, const char * RangeTable);
```

Description:

Define the components names associated to the current result, either a single result block or the current result defined in a result group.

Parameters:

GiD_FILE fd the file descriptor

char * RangeTable → name of the range table previously defined

Example:

C/C++

```
GiD_fResultRange(fd, "MyTable");
```

FORTRAN

```
GID_FRESULTRANGE(fd, 'MyTable');
```


GiD_fResultComponents

int GiD_fResultComponents(GiD_FILE fd, int compc, const char * compv[]);

Description: Define the components names associated to the current result, either a single result block or the current result defined in a result group. In FORTRAN you should call a different function of each result type:

GiD_fScalarComp(GiD_FILE fd, STRING Comp1) → for scalar result

GiD_fVectorComp(GiD_FILE fd, STRING Comp1, STRING Comp2, STRING Comp3, STRING Comp4) → for vector result type

GiD_f2DMatrixComp(GiD_FILE fd, STRING Sxx, STRING Syy, STRING Sxy) → for matrix result type

GiD_f3DMatrixComp(GiD_FILE fd, STRING Sxx, STRING Syy, STRING Szz, STRING Sxy, STRING Syz, STRING Sxz) → for matrix result type

GiD_fPDMComp(GiD_FILE fd, STRING Sxx, STRING Syy, STRING Sxy, STRING Szz); → for plain deformation matrix result type

GiD_fMainMatrixComp(GiD_FILE fd, STRING Si, STRING Sii, STRING Siii, STRING Vix, STRING Viy, STRING Viz, STRING Viix, STRING Viy, STRING Viiz, STRING Viiix, STRING Viiy, STRING Viiiz) → for main matrix result type

GiD_fLAComponents(GiD_FILE fd, STRING axes_1, STRING axes_2, STRING axes_3) → for local axis result type

GiD_fComplexScalarComp(GiD_FILE fd, STRING Re, STRING Im)

GiD_fComplexVectorComp(GiD_FILE fd, STRING Rex, STRING Imx, STRING Rey, STRING Imy, STRING Rez, STRING Imz)

Parameters:

GiD_FILE fd the file descriptor

int compc → number of components names to write.

char * compv[] → array of names.

Example:

C/C++

```
char cnames[] = {"X", "Y", "Z", "Mod"};
GiD_fResultComponents(fd,3, cnames);
```

FORTRAN

```
CHARACTER*10 XN, YN, ZN, MN
XN = 'X'
YN = 'Y'
ZN = 'Z'
MN = 'Mod'
CALL GiD_fVectorComp(fd,XN, YN, ZN, MN)
```

GiD_fResultUnit

int GiD_fResultUnit(GiD_FILE fd, CONST char * UnitName);

Description: Define the unit string associated to the current result, either a single result block or the current result defined in a result group

Parameters:

See GiD_fBeginResult.

Example:

C/C++

```
GiD_fResultUnit(fd,"m");
```

FORTRAN

```
CHARACTER*10 UNITNAME  
UNITNAME = 'm'  
GID_FRESULTUNIT(fd,UNITNAME)
```

GiD_fBeginResultGroup

int GiD_fBeginResultGroup(GiD_FILE fd, const char * Analysis, double step, GiD_ResultLocation Where, const char * GaussPointsName);

Description: Begin a result group. All grouped in the same analysis and step. See GiD online help on this topic.

Parameters:

See GiD_fBeginResult.

Example:

C/C++

```
GiD_fBeginResultGroup(fd,"Analysis", 1.0d0, GiD_OnNodes, NULL);
GiD_fResultDescription(fd,"ScalarNodes", GiD_Scalar);
GiD_fResultDescription(fd,"VectorNodes", GiD_Vector);
GiD_fResultDescription(fd,"Matrix", GiD_Matrix);
GiD_fResultDescription(fd,"Local Axes", GiD_LocalAxes);
GiD_fResultDescription(fd,"A complex Scalar", GiD_ComplexScalar);
GiD_fResultDescription(fd,"A complex Vector", GiD_ComplexVector);
GiD_fResultDescription(fd,"A complex Matrix", GiD_ComplexMatrix);
for (int i = 0; i < 9; i++ ) {
    GiD_fWriteScalar(fd,nodes[i].id, Random());
    GiD_fWriteVector(fd,nodes[i].id, Random(), Random(), Random());
    GiD_fWrite3DMatrix(fd,nodes[i].id, Random(), Random(), Random(), Random(), Random(), Random());
    GiD_fWriteComplexScalar(fd,nodes[i].id, Random(), Random());
    GiD_fWriteComplexVector(fd,nodes[i].id, Random(), Random(), Random(), Random(), Random(), Random());
    GiD_fWrite3DMatrix(fd,nodes[i].id, Random(), Random(), Random(), Random(), Random(), Random(),
        Random(), Random(), Random(), Random(), Random(), Random());
}
GiD_EndResult(fd);
```

FORTRAN

```
CALL GiD_fBEGINRESULTGROUP(fd,"Analysis", 1.0d0, 0, NULL)
CALL GiD_fRESULTDESCRIPTION(fd,"ScalarNodes", 0)
CALL GiD_fRESULTDESCRIPTION(fd,"VectorNodes",1)
CALL GiD_fRESULTDESCRIPTION(fd,"Matrix",2)
CALL GiD_fRESULTDESCRIPTION(fd,"Local Axes",5)
do idx=1,9
    value = idx*1.5;
    CALL GiD_fWRITESCALAR(fd,nodes[i].id, value)
    CALL GiD_fWRITEVECTOR(fd,nodes[i].id, value, value*2, value*3)
    CALL GiD_fWRITE3DMATRIX(fd,i+1,value,value*2,value*3,value*4,value*7,value*1.1)
    CALL GiD_fWRITELOCALAXES(fd,i+1,value,value*3,value*5)
end do
GiD_endRESULT(fd);
```

GiD_fResultDescription

```
int GiD_fResultDescription(GiD_FILE fd, const char * Result, GiD_ResultType Type);  
int GiD_fResultDescriptionDim(GiD_FILE fd, const char * Result, GiD_ResultType Type, size_t dim);
```

Description: Define a result member of a result group given the name and result type. The second prototype enable us to specify the dimension of the result types. Most of the types do not allow more than one dimension. Bellow if the set of valid dimensions for the argument dim given the value of Type:

- Scalar : 1 (GiD_fWriteScalar)
- Vector : 2 (GiD_fWrite2DVector), 3 (GiD_fWriteVector) or 4 (GiD_fWriteVectorModule)
- Matrix : 3 (GiD_fWrite2DMatrix) or 6 (GiD_fWrite3DMatrix)
- PlainDeformationMatrix : 4 (GiD_fWritePlainDefMatrix)
- MainMatrix : 12 (GiD_fWriteMainMatrix)
- LocalAxes : 3 (GiD_fWriteLocalAxes)

Parameters:

See GiD_fBeginResult.

Example:

C/C++

```
See GiD_fBeginResultGroup
```

FORTRAN

```
See GiD_fBeginResultGroup
```

GiD_fResultValues

```
int GiD_fResultValues(GiD_FILE fd);
```

Description: This function is not needed anymore it is just maintained for backward compatibility.

GiD_fEndResult

int GiD_fEndResult(GiD_FILE fd);

Description: End Result Block.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndResult(fd);
```

FORTRAN

```
CALL GID_FENDRESULT(fd)
```

GiD_fBeginOnMeshGroup

```
int GiD_fBeginOnMeshGroup(GiD_FILE fd, char * Name);
```

Description:

Results which are referred to a mesh group (see GiD_fBeginMeshGroup) should be written between a call to this function and GiD_fEndOnMeshGroup.

Parameters:

GiD_FILE fd the file descriptor

char* Name

Name of the mesh group where the results will be specified. This group must be previously defined in a call to GiD_fBeginMeshGroup.

Example:

C/C++

```
GiD_fBeginOnMeshGroup(fd, "steps 1, 2, 3 and 4" );
```

FORTRAN

```
CALL GiD_FBEGINONMESHGROUP(fd, "steps 1, 2, 3 and 4" )
```

GiD_fEndOnMeshGroup

```
int GiD_fEndOnMeshGroup(GiD_FILE fd);
```

Description:

This function close a previously opened block of result over a mesh group. See GiD_fBeginOnMeshGroup.

Parameters:

GiD_FILE fd the file descriptor

Example:

C/C++

```
GiD_fEndOnMeshGroup(fd);
```

FORTRAN

```
CALL GiD_FENDONMESHGROUP(fd)
```


GiD_fResultUserDefined

int GiD_fResultUserDefined(GiD_FILE fd, GP_CONST char * Name,GP_CONST char * Value)

Description: Write a special 'ResultUserDefined' comment with a name and a value that GiD will ignore and could be interpreted by a problemtype for special uses.

Parameters:

GiD_FILE fd the file descriptor

Name: an arbitrary string

Value: an arbitrary string

Note: these strings must be ASCII, and avoid use the special character \n (end of line).

To avoid parsing problems double quotes characters will be replaced by single quote

Example:

C/C++

```
GiD_fResultUserDefined(fd,"ComponentsLocal","Nx Ny Nxy');
```

GiD_fWriteScalar / Vector / Matrix / LocalAxes / Complex* / NurbsSurface*

```
int GiD_fWriteScalar(GiD_FILE fd, int id, double v);
int GiD_fWrite2DVector(GiD_FILE fd, int id, double x, double y);
int GiD_fWriteVector(GiD_FILE fd, int id, double x, double y, double z);
int GiD_fWriteVectorModule(GiD_FILE fd, int id, double x, double y, double z, double mod);
int GiD_fWrite2DMatrix(GiD_FILE fd, int id, double Sxx, double Syy, double Sxy);
int GiD_fWrite3DMatrix(GiD_FILE fd, int id, double Sxx, double Syy, double Szz, double Sxy, double Syz, double Sxz);
int GiD_fWritePlainDefMatrix(GiD_FILE fd, int id, double Sxx, double Syy, double Sxy, double Szz);
int GiD_fWriteMainMatrix(GiD_FILE fd, int id, double Si, double Sii, double Siii, double Vix, double Viy, double Viz, double Viix, double Viiy, double Viiz, double Viii, double Viii, double Viii);
int GiD_fWriteLocalAxes(GiD_FILE fd, int id, double euler_1, double euler_2, double euler_3);
int GiD_fWriteComplexScalar( int id, double complex_real, double complex_imag);
int GiD_fWriteComplexScalar( GiD_FILE fd, int id, double complex_real, double complex_imag);
int GiD_fWrite2DComplexVector( GiD_FILE fd, int id, double x_real, double x_imag, double y_real, double y_imag);
int GiD_fWriteComplexVector( GiD_FILE fd, int id, double x_real, double x_imag, double y_real, double y_imag, double z_real, double z_imag);
int GiD_fWrite2DComplexMatrix(GiD_FILE fd, int id, double Sxx_real, double Syy_real, double Sxy_real, double Sxx_imag, double Syy_imag, double Sxy_imag);
int GiD_fWrite3DComplexMatrix(GiD_FILE fd, int id, double Sxx_real, double Syy_real, double Szz_real, double Sxy_real, double Syz_real, double Sxz_real, double Sxx_imag, double Syy_imag, double Szz_imag, double Sxy_imag, double Syz_imag, double Sxz_imag);
int GiD_fWriteNurbsSurface(GiD_FILE fd, int id, int n, double * v);
int GiD_fWriteNurbsSurfaceVector(GiD_FILE fd, int id, int n, int num_comp, double * v);
```

Description:

Write result functions. Must be between GiD_fBeginResult and GiD_fEndResult

Parameters:

GiD_FILE fd the file descriptor

For GiD_fWriteNurbsSurface and GiD_fWriteNurbsSurfaceVector:

n: the number of control points

num_comp: number of vector components

v: the results in order like:

R1 = u1, v1

R2 = u1, v2

R3 = u1, v3

...

Rn = u1, vn

Rn+1 = u2, v1

Rn+2 = u2, v2

Rn+3 = u2, v3

R2n = u2, vn

....

Rk = um, vn

where u and v are nurbs directions.

In case that one result is not defined use GP_UNKNOWN.

Note: GP_UNKNOWN is a special real value that is defined as (float)-3.40282346638528860e+38 and has a special meaning for GiD as NO_RESULT. Must not be operated like the rest of real numbers.

Example:

C/C++

```
GiD_fWriteScalar(fd,3,4.6);
```

FORTRAN

```
INTEGER*4 idx
REAL*8 value
idx=3
value=4.6
CALL GiD_FWRITESCALAR(fd,idx,value)
```

GiD_fWriteResultsBlock (array of result values)

```

1  int GiD_fWriteResultBlock( GiD_FILE fd, GP_CONST char *result_name, GP_CONST char *analysis_name,
    double step_value,
2
3      GiD_ResultType result_type, GiD_ResultLocation result_location,
    GP_CONST char *gauss_point_name, GP_CONST char *range_table_name, int
    num_component_names,
4
5      GP_CONST char *list_component_names[], GP_CONST char *unit_name,
    int num_result_values, GP_CONST int *list_result_ids,
6
7      // list_component_values ==
    // num_component_values == 1 --> scalar_array
8      // num_component_values == 2 --> VxVyVz_array
9      // num_component_values == 6 --> SxxSyySxySzzSxzSyz_array
10     // ...
11     int num_component_values, GP_CONST double *list_component_values );

```

Description:

Write the results values array. This functions include `GiD_fBeginResult()` and `GiD_fEndResult()`.

If `list_result_ids` is omitted, i.e. `== NULL`, then result value's id's are assumed to be `1..num_result_values`.

Parameters:

`result_name`, `analysis_name`, `step_value`, `result_type`, `result_location`, `gauss_point_name`, `range_table_name`, `num_component_names`, `list_component_names[]` are the same as in [GiD_fBeginResult \(DEPRECATED\)](#) and [GiD_fBeginResultHeader](#).

`gauss_point_name`, `range_table_name`, `num_component_names`, `list_component_names` can be `NULL`, `""` or `0`.

`unit_name` is the name of the unit of the result, as in [GiD_fResultUnit](#). it can be `NULL`, or `""`

`num_result_values` is the number of result values provided, i.e. number of scalars, or vectors, or tensors, etc.

`list_result_ids` is the array of result id's and expected to be `num_result_values` in size. It can be `NULL`.

`num_component_values` is the number of components per result value, for instance 1 for scalar, 2, 3, or 4 for vector, etc... as described in [GiD_fBeginResult \(DEPRECATED\)](#) and [GiD_fBeginResultHeader](#)

`list_component_values` is the array of consecutive result values, for instance `v1v2v3v1v2v3...` for 3-component vector result, and expected to have `num_result_values * num_component_values` doubles.

Example:

C/C++ as in `testpost_fd.c` :

```

1
2  const int num_components = 3; // xyz vector
3  const int offset_value = 3000;
4  double *scalar = create_dummy_result_block( num_components, NUM_NODES, offset_value ); // returns
    a malloc'ed vector
5  // GiD_fWriteResultBlock includes BeginResult()/BeginValues() and EndValues()/EndResult()
6  int list_ids[ NUM_NODES ];
7  for ( int i = 0; i < NUM_NODES; i++ ) {
8      list_ids[ i ] = G_nodes[ i ].id;
9  }
10  const char *unit_name = "m/s"; // just an example
11  GiD_fWriteResultBlock( fd, "VectorNodes", "Analysis", 1.0, GiD_Vector, GiD_OnNodes, NULL, NULL,
    0, NULL, unit_name,

```

```
12         NUM_NODES, list_ids, num_components, scalar );
13     free( scalar );
```

Download the library

The library can be downloaded at: <https://downloads.gidsimulation.com/#Tools/gidpost/>