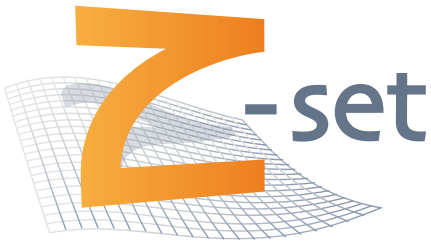
Material & Structure Analysis Suite



Developer Manual
Version 8.6

Z-set 8.6 is distributed by

Transvalor S.A.
Centre des Matériaux
B.P. 87 – 91003 EVRY Cedex
France

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

Contents

Chapter 1

Introduction

Introduction

This manual covers the interfaces for developing material laws and FEA methods in Zebulon/Z-mat. Unlike other FEA codes which provide specific user routines to be re-defined, Z-mat and Z-set come with full featured application programming interfaces API, and a custom build environment. On unix machines makefiles are generated automatically, and on windows platforms a Microsoft Visual Studio project is generated, with all appropriate options set.

The API allows the user to make developments which span the whole range of analysis, adding for example new material behaviors, elements, post calculations, and even meshing and visualization options in the Zmaster program (without knowing GUI programming). The code is supplemented with many example additions in the form of our plug-in source repository, and ZebFront example files.

If there are specific issues of interest which are not documented, please feel free to inquire about them through the technical support hotline or e-mail.

Code Changes:
Zebulon/Z-mat are products under intensive development, and the code reflects this. All programs and utilities within Zebulon are done in C++ and use the same base classes. We emphasis modularity in design, but are not currently making any attempt to provide a static (fixed) interface design. This means that we will feel free to modify class implementations and interfaces as seen fit to allow rapid advancement of the code. The best way to avoid this constant change is to use ZebFront where ever it is implemented.

Environment:
The most classical error with Zebulon is the incorrect setting of environment variables, or mixing versions (a script from one version trying to run with a different `$Z7PATH`. Please make sure that you change versions cleanly:

```
setenv Z7PATH ‘pwd‘
source $Z7PATH/lib/Z7_cshrc
```

Make sure that the desired version is listed before others in your path and `LD_LIBRARY_PATH` variable (or `SHLIB_PATH` on HP-UX or `LIBPATH` on AIX).

Compiler options:
Compiler options can be set in a variety of ways. First, when compiling (using `Zmake`) the script looks for an entry in `$Z7PATH/lib/MACHINE_TYPES` headed by the hostname. If so found, it will take the compiler and associated options from there. This file lets any computer use any options as required. If the file is not there, or the current machine is not listed, the script will look through the compilers listed in `$Z7PATH/lib/COMPILER_DEFS`, and use the first one found.

Additional options and link libraries can be found in files named `$Z7PATH`lib/Makefile.XXX/ The exact file of interest is indicated in your project `library_files` file. These Makefile headers may in turn include additional files, such as `$Z7PATH/lib/Motif_lib_$(Z7MACHINE)`.

Why libraries:
The use of shared libraries allows different configurations of Zebulon/Z-mat to be built, and user additions to be made. For use with other codes such as in Z-mat for ABAQUS, the libraries alleviate the need for a compiler when running ABAQUS. The following summarizes the different component libraries in Zebulon.

The libraries are found in `$Z7PATH/PUBLIC/lib-$Z7MACHINE`. Static libraries end with `.a` (except on AIX, where these are shared) and shared libraries ending in `.so` (or `.sl` on HP-UX or `.a` on AIX). For windows platforms the libraries are called Dynamic linked libraries or DLLs. They are found in the `$Z7PATH\win32` directory.

Foundations:

`$Z7PATH/PUBLIC/lib-$Z7MACHINE/libzTools.so` library of just the tools items. Not used for Z-set, but useful for making utility programs.

`$Z7PATH/PUBLIC/lib-$Z7MACHINE/libZmat_base.so` `%Z7PATH%/win32/zmat_base.dll`

Utility Library of items used in the Z-mat product, including optimization and simulation.

`$Z7PATH/PUBLIC/lib-$Z7MACHINE/libZfem_base.so` `%Z7PATH%/win32/zfem_base.dll`

Library of items used in Z-set, and the Zmaster interface. Not used when executing Z-mat.

User plug-ins:
Plug in functionality is added by making extra libraries which are loaded dynamically. The libraries will be searched in for first in the `$Z7PATH/PUBLIC/lib-$Z7MACHINE` directory on unix systems, or in `$Z7PATH/win32` for windows systems, followed by any libraries in the directory pointed to by `$ZEBU_PATH` and finally in the current working directory of the problem.

- `libZmat`*xxx*`.so` `zmat`*xxx*`.dll` User library plug-ins which will be loaded for Z-mat and Z-set applications.
- `lib`*xxx*`.so` *xxx*`.dll` Any other library in the search path not included

Running Zmat will cause only shared libraries named `Zmat*.so` (or `Zmat*.dll`, etc) to be loaded. Otherwise the software will load all extra libraries found (not part of the standard distribution), if possible. If the library causes an undefined symbol or other error, it will not be loaded. There are messages issued at the beginning of the program launch indicating what is getting loaded.

Example project:
There is an example project in the test directory `$Z7PATH/User_project/` Please note that the format of the projects in this directory may change from version to version. Each time you upgrade, you should at least look for differences in the configuration and source, and possibly re-build your personal project using the new `User_project` directory of interest.

In these projects, there are different ways of making user additions. The most common (recommended) way is to build shared libraries, which will be loaded dynamically as plug-ins (see above).

An example unix configuration file for 2 libraries, one for Z-mat use and one for general Z-set use follows (from the example directory). The file name by default is `library_files` but one can use any file by executing `Zsetup -f` *file*.

```
%
% User-project/library_files
%
!MESSAGE User Z7 project

% !TOP Makefile.top
!TOP Makefile.Motif.c++

!DYNAMIC
!CFLAGS -I${Z7PATH}/include
!BFLAGS -L${Z7PATH}/PUBLIC/lib-${Z7MACHINE}

!MAKE target: lib

!INC   material
!SRC   material material
!INC   finite_element
!SRC   finite_element finite_element
!DEBUG material finite_element

!LIB Zmat_utest material
!LIB Zfem_utest finite_element

!!RETURN
```

Any time a new file is added, the makefile should be re-generated using the `Zsetup` command.

The file for Win32 development is different. Unfortunately the options and linking libraries are somewhat complicated here, but the standard default file should work as-is for most cases.

```
%
% User-project/proj.zpr
%
% !ALLOW_DEBUG
!VCPP  6.00
!OPT   CPP /D "ZEXCEPTIONS" /Tp /D "DLL2" /D "_WIN32" /I "include"
!O_OPT CPP /O2 /D "NDEBUG"
!D_OPT CPP /D "ZCHECK" /Od /D "NDEBUG" /Zi

!OPT LINK32 /NODEFAULTLIB kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib rpcrt4.lib ole32.lib oleaut32.lib
uuid.lib odbc32.lib odbccp32.lib   libcimt.lib LIBCMT.lib OLDNAMES.lib
libcpmt.lib msvcrt.lib /INCREMENTAL:YES /debug

!DLL zmat_test_models
```

```
!GROUPS
   material
!USE
   zmat_base

!OPT LINK32 /NODEFAULTLIB kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib rpcrt4.lib ole32.lib oleaut32.lib
uuid.lib odbc32.lib odbccp32.lib   libcimt.lib LIBCMT.lib OLDNAMES.lib
libcpmt.lib msvcrt.lib MFC42.LIB /INCREMENTAL:YES /debug

!DLL zfem_test_models
!GROUPS
   finite_element
!USE
   zmat_base
   zfem_base
!!RETURN
```

Patching Auto-loaded Classes:
Plug-in additions to the code are interfaced with the standard libraries through the *Object-factory* interface (see page **??**). This interface loads keyword to class creation mappings, and will look for existing entries before adding a new one. If a keyword exists beforehand, it will be replaced by the new one. **It is important however that the class name (the C++ name) be different**. Since the shared libraries are loaded after the program starts executing, this allows patches to be very easily made.

Basic rules

Language:
English will be used as programming language. Things like :

```
// calcul de stress
//
    contrainte=strain*elas;
```

will therefore be avoided.

***.h files**:

*.h files will be organized according to the following template:

```
#ifndef __File_name__
#define __File_name__
```

Do not forget to insert a `#endif` at the end of the file. For instance if the file is called `String.h` it should contain

```
#ifndef __String__
#define __String__
...
#endif
```

This is mandatory in order to avoid to include the same file several times. Classes will then be defined. Other header files can also be included when it is necessary. Note that if a pointer or a reference is used, it is not necessary to include the header file where the class is defined. Therefore

```
#include <String.h>
...
   void myfunction(const STRING&);
```

can be replaced by

```
class STRING;
...
   void myfunction(const STRING&);
```

This decreases the number of interdependencies between header files and can be very useful to speed up the `make`.

***.c files**:

It is recommended to put together in a single file all member functions of a class, according to the following template:

```
#include <...>  // C++ : alphabetical order
...
#include <...>  // Zebulon : alphabetical order

// implementation of class X

    CREATE AND INITIALIZE STATIC DATA MEMBER[S]

    CREATOR[S]

    DESTRUCTOR

    MEMBER FUNCTION[S]

    FRIEND FUNCTION[S]

// implementation of class Y
...
```

`include` will be inserted at the beginning of the file by alphabetical order. C++ files and Z-set will be separated.

Comments:

Comments should be written according to the specific C++ syntax (ie `//`) instead of the C syntax (ie. `/*  */`).

```
/*   this is a comment
     on the way to write comments
*/
```

should be replaced by

```
//    this is a comment
//    on the way to write comments
```

Please write a self–commented program. Instead of

```
TENSOR2 s(d);  // s : stress, d : dimension
```

write:

```
TENSOR2 stress(dimension);
```

names:

Give explicit names to variables, classes and functions.
Use `enum` rather than integers.

```
switch(type) {
       case 1 : do_ps(); break;
       case 2 : do_pe(); break;
};
```

is usefully replaced by

```
switch(type) {
    case PLAIN_STRESS : plane_stress(); break;
    case PLAIN_STRAIN : plane_strain(); break;
};
```

Global variables:

Avoid global variables. Rather use static class data members. [1]

Static variables:

Static data variables (non-const) should be avoided. They cause problems in parallel applications where one instance of the variable may be manipulated by multiple processes at the same time.

Enumerations:

Do not create global enumerations. Attach them to classes.

```
class DOF {
        ....
  public :
       enum DOF_TYPE {U1,U2,U3,EZ,R1,R2,R3,PR,TP};
       ...
};
```

They are used in the following manner:

```
ARRAY<DOF::DOF_TYPE> dof_names(2);
dof_names[0]=DOF::U1; dof_names[1]=DOF::U2;
```

[1] See Reference manual for the list of global variables.

Style, definition

Class Organisation:

class organisation
The following order will be used when defining a class.

```
class T {
   private   :
   protected :
   public    :
};
```

The class name should be written using uppercase letters. `private` members are defined before `protected` members which are defined before `public` members. In each group, members are defined in the following order.

```
enum
data
static data
[creator]
[destructor]
function
virtual function
virtual function=0
static function
friend function
friend class
```

Class Syntax:

- The class name should be written using uppercase letters
- static members (data and functions) should be defined using names in which the first letter is uppercase.
- non static members should be defined in lowercase.
- comments will be inserted before the class definition.
- constructors and destructors will be defined before the other member functions.
- short comments can be inserted in the class definition.

Example:

```
// *************  Comment
//
// DRAW_OBJECT :
//   base class for all object that can be drawn
//   draw() is defined as virtual pure, so don't forget
//   to define it when you add your own object
//
//   Add a new item to OBJECT_TYPE enumeration when you
//   create a new object inheriting from DRAW_OBJECT

// SQUARE      :
//   this is an exemple for a fully define DRAW_OBJECT
//   note that draw() has been defined and reset()
//   overloaded.


class DRAW_OBJECT : public OBJECT {
   private :
     int left,right,top,bottom; // position of the corners
     static int Nb_draw_object;
   protected :
     enum OBJECT_TYPE { SQUARE, TRIANGLE; };
     virtual void reset();
  public :
     DRAW_OBJECT();
     DRAW_OBJECT(const DRAW_OBJECT&);
     virtual void draw()=0;
     static int Nb_object() { return Nb_draw_object; }
     friend class PLOTTER;
};

class SQUARE : public DRAW_OBJECT {
   private :
     int size;  // length of one side
     void reset();
   public :
     SQUARE();
     SQUARE(const SQUARE&);
     virtual void draw();
};
```

Errors and messages

Errors:
All errors and important warnings should use the `ERROR` definitions in `Error_messager.h`. The basic messages are summarized below:

- `ERROR` Basic error. For GUI programming, you should not expect that the program will terminate after this call. If the platform compiler supports exceptions, a `Z7_MAIN_ERROR` will be thrown. Please be descriptive with your messages. An example use:

  ```
  if (bad) ERROR("the calculated result was not ok: "+dtoa(calc));
  ```

  Use of `ERROR` will print the file location where the error occurred.

- `INPUT_ERROR` Use this call if there is an error reading an `ASCII_FILE`. It will print the location of the last file read as part of the message. Example:

  ```
  if (!file.ok) INPUT_ERROR("Trouble reading the precision: "+GLSTR(file));
  ```

- `DBL_REQ` Use this as a shortcut when a real (double) value is required for input. Use of this function will unify the error messages, so it is strongly recommended. Example:

  ```
  eps = file.getdouble(); if (!file.ok) DBL_REQ("eps", file);
  ```

- `INT_REQ` similar to `DBL_REQ` for int values.

- `VEC_REQ` similar to `DBL_REQ` for vector values. Vectors are in the format `( v1 v2 v3)`.

- `GLSTR` utility to get a string value for the last token attempted to be read.

Assertions:
It can be useful to perform tests during the execution of the program to test if there are some bugs. This however slows down the program so that it should be done in some particular occasions. The `assert` function can perform this task. It is implemented as follows (`Assert.h`):

```
#ifdef ZCHECK
#define assert(ex) { if (!(ex)){(void)fprintf(stderr,\
     #ex " : assertion failed: file \"%s\", line %d\n",\
      __FILE__, __LINE__);abort();} }
#else
#define assert(ex)
#endif
```

If ZCHECK is defined (option -DZCHECK while compiling) assert is actually defined. If the test fails, the program stops and indicates the file and the file where the stop was triggered. If ZCHECK is not defined, assert is not compiled and nothing appends.

A good example of this mechanism is given by the generic class ARRAY. When asking for item i using the overloaded operator [] (in Arrayh.).

```
template <class T> class ARRAY
{
  protected : T* x;
              int size;
....
   T& operator[](int rank)
   { assert(rank>=0 && rank<sz);
     return x[rank];
   }
....
```

There is another assertion Assert which works similarly, except that it is always active. This function is placed in locations which should technically be impossible for a user to reach. It outputs the stinging message:

```
Congratulations !! You have generated a Zebulon critical error
XXXX : assertion failed: file SomeFile.c at line XXX
```

Which sends a user directly to the telephone with no real data for debugging the problem, so don't use this thing unless the code really will not be reached. Use an ERROR with some diagnostic data instead.

Debugging Utilities

Debug prints:

Running in gdb (unix):

Running in VC debugger (win32):

Chapter 2

Compiling Utilities

Zmake

Description:
This command makes an executable for the current archetecture based on the **Zsetup** generated pre-makefile `Makefile.dat`. If the `Makefile.dat` was not yet generated, **Zmake** will run **Zsetup** automatically. See below for pre-defined targets.

Syntax:
`% Zmake [`*options*`] [`*target* `] ↩`

CODE	DESCRIPTION
`-h`	gives a list of available switches for the command
`-alt`	use an alternate compiler definition
`-g`	make debug version
`-md`	specify filename to replace `Makefile.dat`
`-p`	make a profile version
`-pg`	make a pg type profile version

Some pre-defined targets are summarized below. Please be carefull with the cleaning targets, as no questions are asked before running.

`objs` build objects only

`lib` build libraries defined with `!LIB` statements in the `library_files` configuration file.

`clean` clean out makefiles, etc.

`archclean` clean out binaries and object files for the current archetecture only. Makefiles are left alone.

`distclean` wipe out all binaries, objects, and generated makefiles, etc. This should leave a minimum set of files to make a source distribution of your code.

Zsetup (unix only)

Description:
This program generates a `Makefile.dat` from a configuration file named `library_files`. The program is efficient for setting up dependencies, adding new files, managing source directories, and configuring libraries to create. The program is provided as a utility, and means for generating appropriate Makefiles for user additions.

Syntax:
In order to configure a source project, define the project structure in `library_files` and run **Zsetup**: `% Zsetup [`*options* `]`↩

CODE	DESCRIPTION
`-f` *filename*	specify a filename to be used in place of the default `library_files`
`-od`	original debug info for ZebFront programs

The `-od` switch can be used to debug ZebFront program compilations. By default ZebFront re-assigns line numbers to the corresponding number in the `.z` file, and the `.c` C++ source is removed after successful compilation. Because many lines are added to the resulting C++ file, there is a possibility that syntax errors are first referenced in the generated code. This method of compiling will leave the `.c` file, and error messages from the compiler will refer to that file.

The syntax of the `library_files` is partially summarized below. All commands start with an exclamation point (`!`) and are followed by free format lists of parameters. In most cases the parameters are read until the next command appears (no so for `!MAKE`). The parameters may be split with comments. The comment character is the pound sign (`#`). The file reading will end with the `!!RETURN` command, which must exist in the file.

`!MAKE` Makefile lines which will be added to the top of the `Makefile.dat` which is generated. Only the remaining part of the line will be taken. Target command lines must have a tab in them, so these lines should have a tab immediately following the `!MAKE` command.

`!DYNAMIC` Indicates that all libraries to be generated, and those to be used, will be shared libraries (`.so` or `.sl` format) and not object archives `.a`.

`!CFLAGS` These are command options which are added on each source file compilation line. One could also use the `MACHINE_TYPES` compiler description file.

`!BFLAGS` These are command options passed on at the link stage of the compilation.

`!INC` This command takes only one argument, the name of a directory containing header files (`.h`).

`!SRC` Description of the dependencies for a directory with C++ source files. The directory name must be given, followed by the include directories with `.h` files used in those sources.

`!FSRC` Fortran files directory.

`!DEBUG` Takes a list of the directories or file names to be compiled in debug when **Zmake** is run using the `-g` switch.

`!LIB` Defines a library to be created. The first parameter is the name of the library, followed by the source directories which are contained in it.

`!TARGET` Defines a target executable. The first parameter is the prefix name of the executable, followed by the source directories which are contained in it.

Source files are only taken if they begin with a capitol letter, and they must all have unique names.

Example:
The `library_files` file used for the small user project is as follows:

```
!MESSAGE User Z7 project
!DYNAMIC
!CFLAGS -I${Z7PATH}/include
!BFLAGS -L${Z7PATH}/PUBLIC/lib-${Z7MACHINE}
!INC   source
!SRC   source source
!DEBUG source
!LIB ZeBaBa_User source
!TARGET NONE source
!!RETURN
```

win_proj.exe (win32 only)

Description:
This program generates a Microsoft Visual Studio project for user development with Z-mat and Z-set. It configures automatically the `EXE` or `DLL` targets, sets up appropriate compile options, and target directories. It also (very importantly) sets up the custom build options for ZebFront development.

Syntax:
In order to configure a source project, define the project structure in *file*`.zpr` and run **win_proj**: `% win\_proj [`*options* `]` *file*`.zpr` ↩ The program accepts a drag and drop interface, which is what we recomend. Open the `%Z7PATH%\win32` directory and drag/drop your personal configuration file on the `win_proj.exe` icon.

The syntax of the `.zpr` file is significantly different from the `library_files` one for unix systems. Normally the pre-configured options given in the `User-project` example files are suitable for general use.

`!Z7DIR` sets the directory to be used in the dev project (helpful if making the project on unix, for use on another win32 machine, or for configuring another project without re-setting the `Z7PATH`).

`!O_DEST_DIR` destination for optimized target

`!D_DEST_DIR` destination for debug target.

`!PROJDIR` the working directory. this sets where temp files will go (i.e. in `o_Win32` or `og_Win32`.

`!ALLOW_DEBUG` Includes a debug target. The debug one is usually the default when first loading a newly generated project.

`!VCPP` Define which Visual C++ you are using.

`!OPT` Basic C++ compiler options shared between optimize and debug

`!O_OPT` Optimize C++ options

`!D_OPT` Debug C++ options

`!OPT LINK32` linker options. Add additional libraries here (for the very brave).

`!DLL` build a dll (plug-in).

`!EXE` Build a win32 exe (GUI interface).

`!CONSOLE` Build a console based (command line) app.

`!RES` Include the listed resource files.

!GROUPS Specify directory groups where source files are kept. All *.c files will be added to the project.

!USE Use the listed DLLs in the final executable.

Example:
The proj.zpr file used for the small user project is as follows:

```
# Optional but perhaps helpful. Using network paths is generally better than
# using drive mapped paths (i.e. M:\Z8.0)

!Z7DIR       \\Vache\bison\Z8.0
!O_DEST_DIR  \\Vache\bison\Z8.0\win32
!D_DEST_DIR  \\Vache\bison\Z8.0\debug32
!PROJDIR     \\Vache\bison\Z8.0\\User-project

% !ALLOW_DEBUG
!VCPP    6.00
!OPT   CPP /D "ZEXCEPTIONS" /Tp /D "DLL2" /D "_WIN32" /I "include"
!O_OPT CPP /O2 /D "NDEBUG"
!D_OPT CPP /D "ZCHECK" /Od /D "NDEBUG" /Zi

!OPT LINK32 /NODEFAULTLIB kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib rpcrt4.lib ole32.lib oleaut32.lib
uuid.lib odbc32.lib odbccp32.lib   libcimt.lib LIBCMT.lib OLDNAMES.lib
libcpmt.lib msvcrt.lib /INCREMENTAL:YES /debug

!DLL zmat_test_models
!GROUPS
   material
!USE
   zmat_base

!OPT LINK32 /NODEFAULTLIB kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib rpcrt4.lib ole32.lib oleaut32.lib
uuid.lib odbc32.lib odbccp32.lib   libcimt.lib LIBCMT.lib OLDNAMES.lib
libcpmt.lib msvcrt.lib MFC42.LIB /INCREMENTAL:YES /debug

!DLL zfem_test_models
!GROUPS
   finite_element
!USE
   zmat_base zfem_base

!!RETURN
```

ZebFront

Description:
This command runs a file through the pre-processor ZebFront to compile `.z` files. Normally the program will not be used directly by the end-user, but rather as part of a generated makefile or development project.

Syntax:
`% ZebFront [`*options*`]` *file.z* ↩

CODE	DESCRIPTION
`-h`	gives a list of available switches for the command
`-d`	run ZebFront in the `gdb` debugger
`-od`	"original debug" where the line numbers are in the generated `.c` file (which is kept)
`-o`	specify output filename (otherwise output is to stdout)

Zcc

Description:
Compile a single zebulon file. Handles the difference between `.c` and `.z` files.

Syntax:
`% Zcc [`*options*`]` *file.[z,c]* /↩

CODE	DESCRIPTION
`-C`	give the compiler to use
`-I`	add an include directory
`-D`	add a define
`-g`	compile in debug
`-h`	gives list of options
`-od`	use “original debug” for ZebFront files
`-nc`	don’t clear the screen first
`-S`	stop at assembly code
`-E`	stop after C preprocessor
`-alt`	use alt compiler definition in `MACHINE_TYPES`

Chapter 3

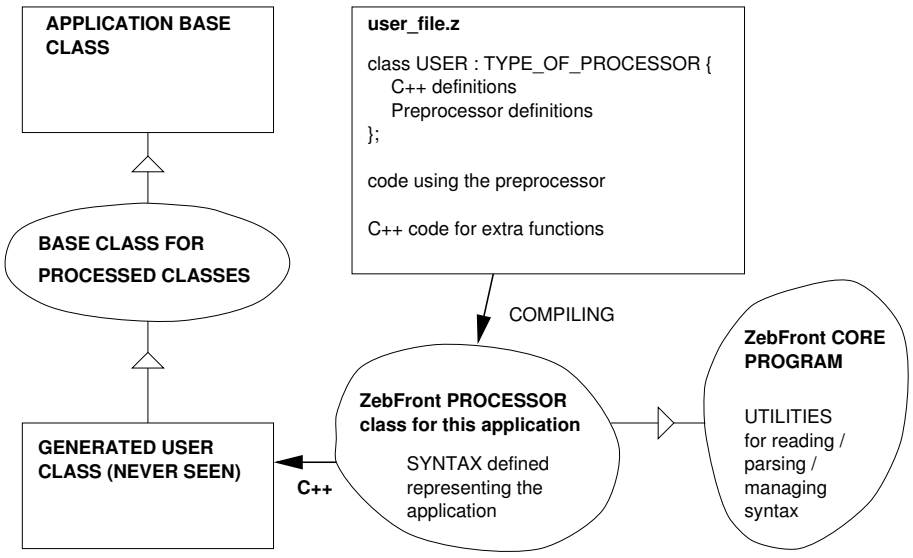
ZebFront pre-processor

ZebFront

Description:
The objective of the `ZebFront` preprocessor is to provide a means of reducing the technical programming tasks involved in material models. This will hopefully encourage material theorists to implement their models early in the development process. Resulting code should be of high enough quality for final implementations as well and therefore provides a general tool. Eventually, the preprocessor will be able to provide some symbolic math to reduce the efforts necessary to program implicit integration methods.

Several applications are envisioned for the preprocessor, each giving a similar interface (modeling language). As mentioned above, the main purpose is the creation of material behaviors. This currently applies to simulation models and FEM material behaviors. Soon, the preprocessor will be able to construct class definitions based on a generalized template, thereby allowing the user to create whole class hierarchies for different applications. Implementation of the template method of class formatting involves some significant modifications and influences largely the use of the `@SubClass` command. This command should thus be considered as unfinished; a more complete version will be available shortly.

The basic functionality of the `ZebFront` program may be summarized by the following figure:



The preprocessor is seen to be composed of a standard Z-set base class supporting the preprocessor mode, and a corresponding module in the preprocessor program defining the syntax and directives which are allowed. `ZebFront` is therefore expandable through the addition of these supporting pairs.

The user (model programmer) then writes a personalized model in a file suffixed by `.z` which is the program source. This source file is a complete model definition, and does not require any modifications to other program files. There is also no limit to the number of `.z` files in a given user project.

A model seen to be defined by a class definition, and code defining the implementation. The class definition resembles a C++ class and requires that the class derive from a defined class type, which specifies the class type which will be created. Note that different class types may have very different syntax and rules. The allowable modes are summarized below:

CODE	DESCRIPTION
SIMUL_MODEL	Simulation model (page 3.7).
BASIC_NL_BEHAVIOR	Presumed non-linear FEM material behavior (page 3.9).
BASIC_SIMULATOR	Simulation model derived from a BASIC_NL_BEHAVIOR (page 3.13).

Math Object Summary

Description:
This section briefly summarizes the use of math objects which are available to the ZebFront program modules. These classes aid the mathematical programming greatly, and are at least partially mandatory in that all input and output variables are stored with these objects. The use of high-level utility classes is very beneficial to the program longevity because it hides implementation assumptions. The main developers may thus optimize personal code without ever touching the source. In fact, this principle applies to the ZebFront methodology at every level.

CODE	DESCRIPTION
VECTOR	Array of double storage to be used as general vector storage; vectors may be of any size, but do support some methods of a first order tensor
MATRIX	basic matrix class
SMATRIX	square matrix class with some methods for a fourth order tensor
TENSOR2	second order tensor class
ARRAY<T>	template for a list of objects; size must be given
DARRAY<T>	double dimensioned array template
LIST<T>	single dimensioned array which may be added to dynamically

- Size checking All objects verify the size equality between objects at each function or operator call when the program is compiled in debug mode. For example, to use an equality operator between two objects A and B, the size attributes of both A and B *must* be equal. Sizes may always be adjusted using the resize methods. The syntax of a resize method always parallels the creator syntax.
- Creators Objects usually may be created as local variables without parameters, with a size attribute, or with using another object of the same type.
- Indexing The method of indexing differs between single dimension and two dimension entities. Single dimension entities VECTOR, TENSOR2, ARRAY<T>, and LIST<T> are accessed using the [ ] operator as a normal C vector. Double dimensioned objects use the form (i,j) where i and j are integers. This operators are defined inline, and are thus as efficient as direct pointer access. The indexing implements bounds checking however in debug mode.

- `Sub-objects` Math objects `VECTOR`, `TENSOR2`, and `MATRIX` may be sub-entities of another math object. Sub-objects may be attached in their creation, or using the `reassign` method. The syntax for all objects follows the same convention for both the creator and the `reassign` methods. This dictates that the first parameters of the method begins with the size attribute as in the standard creator. The following parameter is the object which is being attached onto, and the final parameters are the location in the "host" object.

Some basic operators are summarized below:

CODE	DESCRIPTION
!	Size-of operator; returns tensor / vector / array sizes
^	Outside or cross product; two tensors multiplied returns a `SMATRIX`
|	Contracted product; returns a scalar

SIMUL_MODEL

Description:
The SIMUL_MODEL ZebFront class type is used for the simplest models, where a pure differential model is given. One good application of this type of model is for verification of user FEM models, or to speed simulations in an optimization procedure.

A model is defined by its "observable" variables which can be imposed as loading within the simulator (although it is up to the model programmer to sort out different loading conditions). In order to simulate different equations, "integrated" variables will be used having been given the time (or pseudo-time) derivative functions. The observable variables will be naturally part of the integrated variables, so their evolution is given in the simulation routine according to the active loading (see below for example). Additional variables may be output by assigning them "auxiliary" space in the class definition.

Syntax:
A summary follows of the pre-processor directives available in simulation models:

CODE	DESCRIPTION
@Class	declares a user-class
@Derivative	explicit integration function calculating variable time derivatives
@UserRead	extra read function to search user defined syntax
@UserOutput	function for extra output which may be desired.

The class:
A model of type SIMUL_MODEL is more limited than other models in ZebFront. Only the following commands sub-set of commands are available:

```
@Class NAME_OF_CLASS : SIMUL_MODEL {
   @Name      class_name;
   @Coeff coeff, ..., coeffN;
   @VarInt list;
   @VarAux list;
   @Observable list;
   additional C++ code
};
```

In addition to the limitations of commands, the syntax of those available are reduced as well. The above only permit comma separated lists of names to represent the data members. The coefficients in the model file may also be only single (real) values.

Example:
The following is a complete example of a one dimensional model for a viscoplastic behavior. Note that the syntax is simple and there are really no pre-defined utilities. The load array specifies the variables which are given in the input file, and can be chosen from the variables defined as @Observable in the class heading.

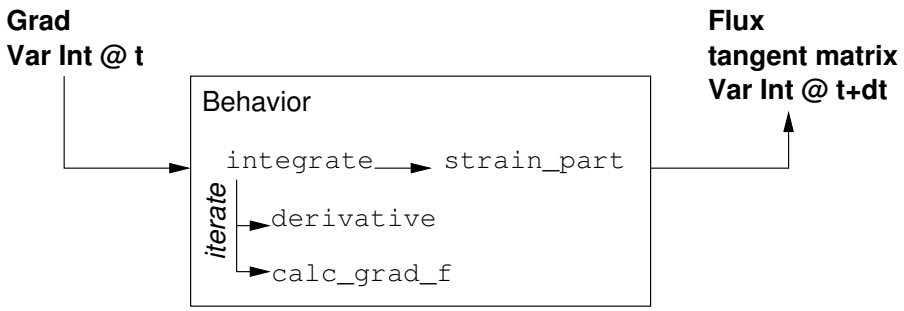
```
//
// Viscoplasticity with one kinematic variable, 1d loading
//
@Class CNLV1 : SIMUL_MODEL {
   @Coefs   young, K, n R0, Q, b, C, D;
   @VarInt  eel, evcum, alpha, evi;
   @VarAux  X, R, sigeff, f;
   @Observable sig, eto;
};

@Derivative {
  eto = eel + evi;
  sig = young*eel;
  X = C*alpha;
  R = R0 + Q*(1.0-exp(-b*evcum));
  sigeff = sig-X;
  f =  fabs(sigeff) - R;
  if (f>0.0) {
     devcum   = exp(n*log(f/K));
     devi     = sign(sigeff)*devcum;
     if (C>0.0)  dalpha  = devi - devcum*D*X/C;
  }

  if      (load[0]=="sig") deel = dglobal[0]/young;
  else if (load[0]=="eto") deel = dglobal[0] - devi;
}
```

BASIC_NL_BEHAVIOR

The principle functionality of behaviors in the Z-set FEM mode may be summarized by the following schematic:



Here, the last FEM solution is at time t, which is to be advanced in the global solution iterations to time $t+dt$. The *ZebFront* mode for FEM material behaviors therefore is centralized for this functionality. The internal calling sequence will be such that a method `integrate` will be called, which in turn may call several different methods. The *ZebFront* may be used to support solely the `integrate` method, allowing the user to re-define the entire model. Complicated multi-function programs may therefore be created, while taking advantage of the programs coefficient and model variable management capabilities. More often than not however, a model may be implemented more simply using the standard integration formats (Runge-Kutta or θ-method integration).

A summary follows of the pre-processor directives available in material behaviors:

CODE	DESCRIPTION
`@Class`	declares a user-class
`@SetUp`	method which is called before the calculation begins; one may set up variable storage here for example
`@Integrate`	user-integrate function; used for behaviors which are not using the standard Runge-Kutta or θ-method integration
`@Derivative`	explicit integration function calculating variable time derivatives, used by the standard Runge-Kutta implementation
`@CalcGradF`	implicit integration function for the variable residual and Jacobian matrix, used by the standard θ-method implementation
`@UserRead`	extra read function to search user defined syntax
`@PreStep`	this method performs calculations before the local integration begins
`@PostStep`	this method is used to perform calculations after the local integration is finished (was called @StrainPart)

If the user has selected the standard integration methods, the `integrate` method should not be re-defined. The default will perform some internal set-up operations, and then dispatch

control to the integration method. Integration methods will be selected in the user input file for each calculation using the `*integration` option under `***material` in `****calcul`. Each method then calls back on the *ZebFront* source in the method `derivative` or `calc_grad_f` if they have been defined. A model developer is not required to define all integration methods, only the ones he wishes. *ZebFront* will generate automatically default error messages for the methods not implemented.

Standard methods:
The pre-processor defines a number of class methods which are defined only with pre-processor directives, and without parameters. The absence of parameters allows the pre-processor to change its implementation of the methods without affecting the user program source. Different methods define a different set of variables depending on their application.

Pre-defined variables:
The class creation includes definition of several variables which will be kept up to date in the program. Use of these variables remove dependence on the actual program internals, and therefore assure compatibility with future versions of the code.

CODE	DESCRIPTION
`psz`	symmetric tensor (problem) size
`usz`	unsymmetric tensor size
`unit`	deviator multiplier $s' =$ `unit` $\boldsymbol{\sigma}$
`m_tg_matrix`	the tangent matrix to be returned
`m_flags`	behavior flags
`Time_ini`	initial time of the increment
`Time`	time at the ending of the increment
`Dtime`	increment of time over the increment

Gradiant-flux variables:
Previous versions of ZebFront used the reserved names `grad`, `dgrad`, and `flux` for the imposed variable names. Now that several gradient-flux variables may be given for a particular problem, these will be accessed using their names proper. Thus, for a program with the grad/flux combination of `eto`/`sig`, one has the variables `eto`, `deto`, and `sig` directly available.

By default (in the absence of a `@Flux` or `@Grad` command in the class declaration) the flux will be a symmetric Cauchy stress tensor named `sig`, while the gradient will be the symmetric small strain tensor `eto`. The names of these variables **must** be compatible with the element specification in all applications.

Generated class:
The generated `BEHAVIOR` class will follow the generalized syntax:

```
***behavior behavior_name modifier
 **sub_class_type SUB_CLASS
   ...
 **model_coef
   cname1 COEFFICIENT
   ...
   cnameN COEFFICIENT
 **user_option
   ...
***return
```

Example:
A class definition of the following type:

```
@Class PLASTIC_BEHAVIOR : BASIC_NL_BEHAVIOR {
   @Name plastic;
   @SubClass ELASTICITY elasticity;
   @Coefs    R0, Q, b;
   @tVarInt  eel;
   @sVarInt  epcum;
   @tVarAux  epi;
};
```

May accept the following input:

```
***behavior plastic_behavior
**elasticity isotropic
     young 260000.
     poisson 0.3
**model_coef
     R0  130.
     Q   20.0
     b   500.0
***return
```

BASIC_SIMULATOR

Description:
The BASIC_SIMULATOR ZebFront class type is used for FEM behaviors which are meant to be run in the simulation mode as well. Classes of this type must be valid BASIC_NL_BEHAVIOR classes, and require special code to calculate the mixed-mode loading case, but also allow definition of yield surface functions.

Syntax:
The class declaration is the same as that for BASIC_NL_BEHAVIOR with some extensions. No new class methods (main functions) are provided for the simulation mode.

The class definition for BASIC_SIMULATOR is the following:

```
@class NAME_OF_CLASS : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
   standard behavior options from page 3.19
   @Criterion list;
   additional C++ code
};
```

Resolving mixed flux-grad loading:
One of the main benefits of the simulator is the ability to solve mixed loadings exactly, while displacement based FE solutions are approximate and require iterations in the non-linear case (and therefore desiring a good tangent matrix calculation). The disadvantage is a formulation must be made to solve the mixed rate loading. To address the later, special methods are given in the BASIC_SIMULATOR base class for resolving such a difficulty (this is in contrast to the solution in the class SIMUL_MODEL discussed on pages 3.7-3.8). Using the notation $\mathbf{f}$ for the flux and $\mathbf{g}$ for the gradient, the following forms are allowed. See the developers manual for a description of how these methods are implemented.

CODE	DESCRIPTION
`resolve_flux_grad(E, de, dg, de2)`	$\dot{\mathbf{f}} = \mathbf{E}(\dot{\mathbf{g}} - \dot{\mathbf{e}} - \dot{\mathbf{e}}_2)$
`resolve_flux_grad(E, de, dg)`	$\dot{\mathbf{f}} = \mathbf{E}(\dot{\mathbf{g}} - \dot{\mathbf{e}})$

Example:
Here's an example of a combined FEM-simulator model with a criterion object.

```
@Class FEM_SIM_BEHAVIOR : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
   @Name example;
   @Coefs    E, poisson;
   @Coefs    R0, H, Q, b;
   @Coefs    alpha, beta, A, k, r;
   @Coefs    dmax;
   @tVarInt  eel;
   @sVarInt  evcum, D;
   @sVarAux  R,j0,j1,j2,chi;
```

```
   @tVarAux  evi;
   @Criterion yield, damage;
};
```

Resolving mixed loading for time independent plasticity and damage:
Mostly, the so-called function resolve_flux_grad(E, deel, deto, devi) is used in `Zmat` routines to compute the increment of elastic strain *eel* knowing the Hooke's tensor E, the increment of the total strain *eto* and the increment of the plastic strain *evi*. However, this function is not suitable in the case of time independent plasticity and it is even less if damage is introduced into the model.
The problem to be solved is the determination of unknown components of the total strain rate based on the Hooke's law

$$\dot{\sigma} = (1-D)E\left(\dot{\varepsilon} - \dot{\underset{\sim}{\varepsilon}}^p\right) - E\left(\underset{\sim}{\varepsilon} - \underset{\sim}{\varepsilon}^p\right)\dot{D}$$

and the consistency condition

$$(1-D)nE\dot{\varepsilon} = (H + nEn)\,\dot{p}$$

A linear system is built and solved on the light of the two previous equations. The unkowns are the strain components that are not involved in the loading.

CODE	DESCRIPTION
`resolve_flux_grad_consistency(E,norm,H,deto,devcum,eel,D,dD_dp)`	

Numerical backgrounds
In the following, subscripts "k" and "u" refer to as "known" and "unknown" total strain respectively. Thus, vectors containing total strain components and stress components together with the Hooke's tensor write:

$$\dot{\varepsilon} = \begin{pmatrix} \dot{\varepsilon}^k : \text{known} \\ \dot{\varepsilon}^u : \text{unknown} \end{pmatrix} \qquad \dot{\sigma} = \begin{pmatrix} \dot{\sigma^k} : \text{unknown} \\ \dot{\sigma^u} : \text{known} \end{pmatrix} \qquad E = \begin{pmatrix} E_{kk} & E_{ku} \\ E_{uk} & E_{uu} \end{pmatrix}$$

The linear system allowing to compute the strain rates $\dot{\varepsilon}^u$ and the plastic multiplier $\dot{p}$ is presented in the case of time independent model without damage and then including damage:

- time independent model without damage
  The Hooke's law writes:

$$\dot{\sigma} = E\left(\dot{\varepsilon} - \dot{p}n\right)$$

$$\begin{pmatrix} \dot{\sigma}^k \\ \dot{\sigma}^u \end{pmatrix} = \begin{pmatrix} E_{kk} & E_{ku} \\ E_{uk} & E_{uu} \end{pmatrix} \begin{pmatrix} \dot{\varepsilon}^k - \dot{p}n^k \\ \dot{\varepsilon}^u - \dot{p}n^u \end{pmatrix}$$

$$\dot{\sigma}^u = E_{uk}\dot{\varepsilon}^k + E_{uu}\dot{\varepsilon}^u - \dot{p}\left(E_{uk}n^k + E_{uu}n^u\right)$$

  "u" équations are then be obtained:

$$\boxed{E_{uu}\dot{\varepsilon}^u - (En)^u\,\dot{p} = \dot{\sigma}^u - E_{uk}\dot{\varepsilon}^k}$$

The consistency condition $\dot{f}=0$ gives:

$$n\dot{\sigma}=H\dot{p} \qquad n\left(E\dot{\varepsilon}\right)=\left(H+nEn\right)\dot{p}$$

$$\left(\begin{array}{c} n^k \\ n^u \end{array}\right)\left(\begin{array}{cc} E_{kk} & E_{ku} \\ E_{uk} & E_{uu} \end{array}\right)\left(\begin{array}{c} \dot{\varepsilon}^k \\ \dot{\varepsilon}^u \end{array}\right)=\left(H+nEn\right)\dot{p}$$

$$\left(\begin{array}{c} n^k \\ n^u \end{array}\right)\left(\begin{array}{c} E_{kk}\dot{\varepsilon}^k+E_{ku}\dot{\varepsilon}^u \\ E_{uk}\dot{\varepsilon}^k+E_{uu}\dot{\varepsilon}^u \end{array}\right)=\left(H+nEn\right)\dot{p}$$

and leads to the scalar equation:

$$\boxed{\left(n^kE_{ku}+n^uE_{uu}\right)\dot{\varepsilon}^u-\left(H+nEn\right)\dot{p}=-\left(n^kE_{kk}+n^uE_{uk}\right)\dot{\varepsilon}^k}$$

The system to be solved in the case of time independent plasticity is as follows:

$$\left(\begin{array}{cc} H_{cc} & H_{cp} \\ H_{pc} & H_{pp} \end{array}\right)\left(\begin{array}{c} \dot{\varepsilon}^u \\ \dot{p} \end{array}\right)=\left(\begin{array}{c} b_c \\ b_p \end{array}\right)$$

$$\begin{aligned} H_{cc}&=E_{uu}\\ H_{cp}&=-\left(En\right)^u\\ H_{pc}&=\left(n^kE_{ku}+n^uE_{uu}\right)\\ H_{pp}&=-\left(H+nEn\right)\\ b_c&=\dot{\sigma}^u-E_{uk}\dot{\varepsilon}^k\\ b_p&=-\left(n^kE_{kk}+n^uE_{uk}\right)\dot{\varepsilon}^k \end{aligned}$$

- time independent model with damage
  The Hooke's law is now such that:

$$\sigma=(1-D)E\left(\varepsilon-\varepsilon^p\right)$$

$$\dot{\sigma}=(1-D)E\left(\dot{\varepsilon}-\dot{p}n/(1-D)\right)-E\left(\varepsilon-\varepsilon^p\right)\dot{D}$$

$$\left(\begin{array}{c} \dot{\sigma}^k \\ \dot{\sigma}^u \end{array}\right)=(1-D)\left(\begin{array}{cc} E_{kk} & E_{ku} \\ E_{uk} & E_{uu} \end{array}\right)\left(\begin{array}{c} \dot{\varepsilon}^k-\dot{p}n^k/(1-D) \\ \dot{\varepsilon}^u-\dot{p}n^u/(1-D) \end{array}\right)-\left(\begin{array}{cc} E_{kk} & E_{ku} \\ E_{uk} & E_{uu} \end{array}\right)\left(\begin{array}{c} \varepsilon^k-\varepsilon_p^k \\ \varepsilon^u-\varepsilon_p^u \end{array}\right)\dot{D}$$

$$\dot{\sigma}^u=(1-D)\left(E_{uk}\dot{\varepsilon}^k+E_{uu}\dot{\varepsilon}^u\right)-\dot{p}\left(E_{uk}n^k+E_{uu}n^u\right)-\dot{D}\left(E_{uk}\left(\varepsilon^k-\varepsilon_p^k\right)+E_{uu}\left(\varepsilon^u-\varepsilon_p^u\right)\right)$$

  The "u" equations depend on the damage law. If, for example, the damage evolution rule $\dot{D}=(Y/S)^s\dot{p}$ is used:

$$\boxed{(1-D)E_{uu}\dot{\varepsilon}^u-\left[\left(En\right)^u+\left(Y/S\right)^s\left(E\varepsilon^e\right)^u\right]\dot{p}=\dot{\sigma}^u-(1-D)E_{uk}\dot{\varepsilon}^k}$$

  The damage variable leads to the modification of the consistency condition as:

$$(1-D)n\left(E\dot{\varepsilon}\right)=\left(H+nEn\right)\dot{p}$$

$$(1-D)\left(\begin{array}{c} n^k \\ n^u \end{array}\right)\left(\begin{array}{cc} E_{kk} & E_{ku} \\ E_{uk} & E_{uu} \end{array}\right)\left(\begin{array}{c} \dot{\varepsilon}^k \\ \dot{\varepsilon}^u \end{array}\right)=\left(H+nEn\right)\dot{p}$$

$$(1-D)\begin{pmatrix} n^k \\ n^u \end{pmatrix}\begin{pmatrix} E_{kk}\dot{\varepsilon}^k + E_{ku}\dot{\varepsilon}^u \\ E_{uk}\dot{\varepsilon}^k + E_{uu}\dot{\varepsilon}^u \end{pmatrix} = (H+nEn)\,\dot{p}$$

$$\boxed{(1-D)\left(n^k E_{ku} + n^u E_{uu}\right)\dot{\varepsilon}^u - (H+nEn)\,\dot{p} = -(1-D)\left(n^k E_{kk} + n^u E_{uk}\right)\dot{\varepsilon}^k}$$

Finally, the system to be solved is of the form:

$$\begin{pmatrix} H_{cc} & H_{cp} \\ H_{pc} & H_{pp} \end{pmatrix}\begin{pmatrix} \dot{\varepsilon}^u \\ \dot{p} \end{pmatrix} = \begin{pmatrix} b_c \\ b_p \end{pmatrix}$$

where:

$$\begin{aligned} H_{cc} &= (1-D)E_{uu} \\ H_{pc} &= (1-D)\left(n^k E_{ku} + n^u E_{uu}\right) \\ H_{pp} &= -(H+nEn) \\ b_p &= -(1-D)\left(n^k E_{kk} + n^u E_{uk}\right)\dot{\varepsilon}^k \end{aligned}$$

If a damage evolution law of the form $\dot{D} = (Y/S)^s \dot{p}$ is chosen:

$$\begin{aligned} H_{cp} &= -\left[\,(En)^u + (Y/S)^s\,(E\varepsilon^e)^u\,\right] \\ b_c &= \dot{\sigma}^u - (1-D)E_{uk}\dot{\varepsilon}^k \end{aligned}$$

Example:
Here's an example of a time independent model involving damage

$$f = J(\underset{\sim}{\overline{\sigma}} - \underset{\sim}{\overline{\mathbf{X}}}) - \overline{R} - R_0 = 0$$

$$\underset{\sim}{\overline{\sigma}} = \underset{\sim}{\mathbf{\Lambda}} : \underset{\sim}{\varepsilon}^e \qquad \underset{\sim}{\overline{\mathbf{X}}} = \frac{2}{3}C\underset{\sim}{\alpha} \qquad \overline{R} = bQr \qquad Y = \frac{1}{2}\underset{\sim}{\varepsilon}^e : \underset{\sim}{\mathbf{\Lambda}} : \underset{\sim}{\varepsilon}^e$$

$$\frac{\partial f}{\partial \underset{\sim}{\overline{\sigma}}} : \dot{\underset{\sim}{\overline{\sigma}}} + \frac{\partial f}{\partial \underset{\sim}{\overline{\mathbf{X}}}} : \dot{\underset{\sim}{\overline{\mathbf{X}}}} + \frac{\partial f}{\partial \overline{R}} : \dot{\overline{R}} = 0$$

Notation $\qquad \underset{\sim}{\mathbf{n}} = \dfrac{\partial f}{\partial \underset{\sim}{\overline{\sigma}}} = \dfrac{3}{2}\,\dfrac{\left(\underset{\sim}{\overline{\mathbf{s}}} - \underset{\sim}{\overline{\mathbf{X}}}\right)}{J(\underset{\sim}{\overline{\sigma}} - \underset{\sim}{\overline{\mathbf{X}}})}$

$$\frac{\partial f}{\partial \underset{\sim}{\overline{\sigma}}} = \underset{\sim}{\mathbf{n}} \qquad \frac{\partial f}{\partial \underset{\sim}{\overline{\mathbf{X}}}} = -\underset{\sim}{\mathbf{n}} \qquad \frac{\partial f}{\partial \overline{R}} = -1$$

$$\dot{\underset{\sim}{\overline{\sigma}}} = \underset{\sim}{\mathbf{\Lambda}} : \underset{\sim}{\varepsilon}^e \qquad \dot{\underset{\sim}{\overline{\mathbf{X}}}} = (2/3)C\dot{\underset{\sim}{\alpha}} \qquad \dot{\overline{R}} = bQ\dot{r} \qquad \dot{D} = \left(\frac{Y}{S}\right)^s \dot{\lambda}_p$$

$$\dot{\underset{\sim}{\varepsilon}}^p = \frac{1}{1-D}\underset{\sim}{\mathbf{n}}\dot{\lambda}_p \qquad \dot{\underset{\sim}{\alpha}} = \frac{1}{1-D}\left(\underset{\sim}{\mathbf{n}} - \gamma\underset{\sim}{\alpha}\right)\dot{\lambda}_p \qquad \dot{r} = \frac{1-br}{1-D}\dot{\lambda}_p$$

$$\dot{\lambda}_p = (1-D)\frac{\underset{\sim}{\mathbf{n}} : \dot{\underset{\sim}{\overline{\sigma}}}}{H} \qquad \dot{\lambda}_p = (1-D)\frac{\underset{\sim}{\mathbf{n}} : \underset{\sim}{\mathbf{\Lambda}} : \dot{\underset{\sim}{\varepsilon}}}{H + \underset{\sim}{\mathbf{n}} : \underset{\sim}{\mathbf{\Lambda}} : \underset{\sim}{\mathbf{n}}}$$

$$H = (2/3)C\left(\underset{\sim}{\mathbf{n}} - \gamma\underset{\sim}{\alpha}\right)\underset{\sim}{\mathbf{n}} + bQ(1-br)$$

```
@Class  TEST_CDM : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
   @Name     test_cdm;
   @SubClass ELASTICITY elasticity;
   @tVarInt  evi,eel,alpha;
   @sVarInt  evcum,D,r;
   @Coefs    C,B,b,Q,R0;
   @Coefs    S,s;
};
@StrainPart {
  double umD = 1.0;
  if(D>0.0) {
     umD = 1.0 - Zmin(0.99,D);
  }
 sig = umD*(*elasticity*eel);
 m_tg_matrix=*elasticity;
}
@Derivative {

   double    J,Reff,Y,crit,umD,H,miso;
   TENSOR2   m,norm,sigeff,sprime,Xeff;
   ELASTICITY& E=*elasticity;
   umD = 1.0;
   if(D>0.0) {
     umD = 1.0 - Zmin(0.99,D);
   }
   sig          = umD*(E*eel);
   sigeff       = E*eel;
   Xeff         = (2./3.)*C*alpha;
   Reff         = b*Q*r;
   sprime       = deviator(sigeff-Xeff);
   J            = sqrt(1.5*(sprime|sprime));
   crit         = J - Reff - R0 ;
   if(crit<=0.){
      devcum=0.; dalpha=0.; dr=0.0; dD=0.0;
      resolve_flux_grad(*elasticity*umD, deel, deto);
}
   else {
      norm   = 1.5*sprime/J;
      m      = norm - B*alpha;
      miso   = 1. - b*r;
      H      = norm|((C/1.5)*m);
      H     += b*Q*miso;
      Y      = (1./2.)*(eel|sigeff);
      double dD_dp = pow(Y/S,s);
      resolve_flux_grad_consistency(E,norm,H,deto,devcum,eel,D,dD_dp);
      devi      = devcum*norm/umD;
      deel      = deto - devi;
      dalpha    = devcum*(norm-B*alpha)/umD;
      dr        = devcum*(1.-b*r)/umD;
      dD        = devcum*dD_dp;
  }
}
```

@Class

Description:
User behavior models are declared in a class definition which resembles a C++ class, including special pre-processor directives. These directives always begin with a `@` sign. It is always possible to include C++ language code in the class definition, including variable and method declarations.

The class creator is generated by the pre-processor, so that method is not allowed in the class declaration. Manipulations are possible in the creator however by using the `@SetUp` or `@UserRead` commands.

Syntax:
The basic form of a `ZebFront` class declaration is summarized below:

```
@Class NAME_OF_CLASS : PROCESSOR_TYPE {
   @Name      class_name;
   @SubClass  type  obj_name;
   @SubClass  type  obj_name @Params param_list;
   @Coeff  coeff, ..., coeffN;
   @Coeff  coeff size, ...  ;
   @Coeff  list @Factor fact;
   @Grad   var_size  declaration;
   @Flux   var_size  declaration;
   @sVarInt list;
   @sVarAux list;
   @sVarUtil list;
   @Tags   tag declarations
   @Integrate
   @Implicit
   additional C++ code
};
```

The processor type defines the type of model which will be created, and therefore the allowable syntax and the base class from which it is derived (as shown on page 3.3).

The *parameters* in the above syntax have the following meanings:

class_name the name to be used in an input file to identify an object of this type. For behaviors, this corresponds to the command `***behavior`. The default name is the class name in all lower case.

type a valid sub-class type. Sub-classes may in essence be any class which satisfies a certain set of requirements[1]. Sub classes may not yet have integrated or auxiliary variables; these classes are just used as utility equations with coefficients.

[1] As these requirements will soon change, no more description will be given... e-mail `foerch@nwnumerics.com` if you have questions

obj_name the name to be used for the object instance. This name will result in an input command with the same name.

param_list The parameters to be sent to a sub-class read method. The default list is the file, problem (tensor) size, and behavior. The default would be input as: `SubClass X x  @Params file,psz,this;`

coeff a character name for a coefficient. Names must not be duplicated with other variable names.

size a size specifier. Coefficients may be loaded as single values (no *size*), as fixed dimension arrays using the syntax `[x]` where *x* is the desired size, as variable diemsnioned entities using `[0-N]` for the size, or in terms of another object using `[!X]` with *X* being another object. Constant or other function values are also allowed, permitting the following two cases: `@sVarInt gamma [12]; @sVarInt gamma1 [get_gamma_size(1)];` where the method `get_gamma_size` is defined in standard C++ within the class.

factor a factor to calculate which will be multiplied to the coefficient. This may be any expression calculatable after the coefficients are loaded, such as: `@Coeff C @Factor 2./D;` with `D` being another coefficient. The factor only applies the initial value of `D` however, and thus cannot be used for variable coefficient interrelations.

s a size specifier. Sizes may be `s` for scalar variables, `t` for symmetric tensors, `u` for unsymmetric tensors, or `v` for vectors[2].

num is an optional size declaration. This size is the number of *repeated* instances of the variable to be stored as an array. The size may either be an explicit number or use an indicator based on the number of a specific coefficient entered. For example, if a coefficient `C` was given an internal variable associated to it may be declared: `@tVarInt alpha [!C];` from which variable are accessed as `alpha[1]`, to `alpha[!C-1]` or equally `alpha[!alpha-1]` (note that indexes start at 0.

list a comma separated list of object names. The syntax is the same as standard C++ variable list. See the end of this page for various examples. Each variable may be given an optional size specification for variables to be stored in list format.

Example:
The examples here attempt to demonstrate some more complex combinations of the class declaration. First is a FEM behavior for finite deformation. The gradiant variable is the deformation gradiant **F** while the flux variable is the Cauchy stress *zsig*. The porgram can use the default flux name of `sig` while it defines a non-symmetric tensor named `F` for **F**.

```
@Class BATHE_ROT : BASIC_NL_BEHAVIOR {
   @SubClass ELASTICITY elasticity;
   @Coefs    K, n, R0, Q, b, C1, D1, C2, D2;
   @Grad     usz F;
   @uVarInt  Fp;
   @sVarInt  evcum;
   @tVarInt  alpha1,alpha2;
};
```

The following example class definition is more modular in nature. [3] This class uses several of the behavior "bricks" which are standard in the Z7 behavior library. Also, the coeffcient `C` is variable in number, while the coefficient `D` and variables `X` and `alpha` are sized to match `C`. This allows the user to enter as many `C` terms as desired.

[2] vectors are not yet implemented
[3] `$Z7PATH/calcul/zZfrontBehavior/ModularPlastic.z`

```
@Class MODULAR_PLASTIC_BEHAVIOR : BASIC_NL_BEHAVIOR {
   @Name     modular_plastic;
   @SubClass ELASTICITY elasticity;
   @SubClass CRITERION  criterion;
   @SubClass FLOW       flow;
   @SubClass ISOTROPIC_HARDENING  isotropic;

   @Coefs    C [0-N] @Factor 1./1.5;
   @Coefs    D [!C];
   @tVarInt  eel, alpha [!C];
   @sVarInt  evcum;
   @tVarAux  X [!C], evi;
   @tVarAux  Xtot;
   @tVarUtil m [!C];
   @tVarUtil Xdot;
};
```

In one final example, a case of multiple gradiant flux variables is given. Here we have a behavior designed to be compatible with a small deformation, icompressible element.

```
@Class ADIABATIC_INCOMP_PLASTICITY : BASIC_NL_BEHAVIOR {
   @Name     adiabatic_incompressible;
   @SubClass ELASTICITY elasticity;
   @SubClass ISOTROPIC_HARDENING  isotropic;

   @Grad     tsz eto;
   @Grad     1 press;
   @Flux     tsz sig;
   @Flux     1 dvolu;

   @Coefs    pCp, chi, alpha_t;
   @tVarInt  eel;
   @sVarInt  epcum, T;
   @tVarAux  epi;
   @Implicit
};
```

@UserRead

Description:
This directive indicates that there are user defined tokens which must be added to the model syntax. The *ZebFront* program provides a standard base of automated reading and variable management, but it is sometimes desirable to add custom reading for more complicated applications. It is preferential to use the default reading because that sets a standard and reduces dependence on the code internals.

Syntax:

```
@UserRead {
  C++ reading code
}
```

The routine enters with an `ASCII_FILE` named `file`, and the token string read named `str`. The `str` parameter is a non-`const STRING&` object. Care must thus be taken to not overwrite the string if the token is not a user defined one.

Example:
The following small example gives a hypothetical input where the model allows an external file to be used for additional vector data, or the data to be input in the material file itself:

```
#define VERIF if (!file.ok) INP_ERROR(&str,&file.name);
@UserRead {
   if (str=="**data_file") {
      if (!data_fname) ERROR ("Multiple filenames");
      data_fname = file.getSTRING_sl();
      return TRUE;
   } else if (str=="**localization") {
      VECTOR tmp(3);
      STRING ctl = file.getSTRING();
      while (file.ok && ctl[0]!='*') {
           tmp[0] = ctl.to_double();
           tmp[1] = file.getdouble_sl(); VERIF;
           tmp[2] = file.getdouble_sl(); VERIF;
           loc.add(tmp);
      } return TRUE;
   } return FALSE;
}
```

`**data_file` is used to load a file name, and `**localization` will be used to load vectors in the material file. The data file could be read in `@SetUp` routine if the size of `data_fname` is greater than zero. The total vector set will be the union of all `**localization` inputs and what is read from the filename.

@UserRead

@PostStep

Description:
This method is used to perform calculations after the local integration is finished. Normally, there will be a number of auxiliary variables which may be desired and can be calculated as a direct function of the new internal variable values. The material tangent matrix is also required for the global solution, and this function may be used for that as well.

If the behavior has re-defined the method `integrate` `@PostStep` has no relevance.

Syntax:
This command is a standard pre-processor method. The method does not define any of the user variables (coefficients are of course available). Use the `@SetVar` command to set all required variables.

Example:
The following example (from `Plastic.z`) is for a behavior using both explicit and implicit integration.

```
@PostStep {
  @SetVar eel,epi;
  epi  = grad - eel;
  flux = *elasticity*eel;
  if (integration&LOCAL_INTEGRATION::THETA_ID) {
     SMATRIX tmp(psz,f_grad,0,0);
     if (Dtime>0.0) m_tg_matrix=*elasticity*tmp;
     else           m_tg_matrix=*elasticity;
  } else if (m_flags&CALC_TG_MATRIX) m_tg_matrix=*elasticity;
}
```

```
@PostStep
```

@Derivative

Description:
This method is used for explicit integration. The model will be required to furnish time derivatives for all the `VarInt` variables as a function of the gradient rate and the current `VarInt` values.

$$\dot{\mathcal{V}}_{int} = fn\left(\dot{\mathbf{g}}_{tot}, \mathcal{A}\left(\mathcal{V}^t_{int}\right)\right)$$

where **g** being the gradient variables. The program automatically initializes the gradient under the name `grad`, and its time derivative named `dgrad`.

For each variable *vname* defined, there will be an associated allocated math object named `d`*vname*. This object is attached appropriately to the integration variable vector such that all calculated variable rates are to be assigned to these variables. The method therefore has the gradient variable, all the current `VarInt` variables, and the `d`*vname* rate variables active. Other variables may be activated using the `@SetVar` command.

CODE	DESCRIPTION
`tau`	double for the fractional time in the increment
`var`	total vector of the integrated variables (const)
`dvar`	vector to be filled with the variable derivatives
`d`*grad*	time derivative of gradient variables; this is changed from the standard definition which is the increment of gradient

For finite element behaviors, after the behavior is integrated, assignment of the auxiliary variables, calculation of the flux variable, `flux`, be made, and assignment of the tangent matrix must be made. These calculations should be made in the `@PostStep` method.

Example:
This is the derivative function for the first example header on page 3.21.

```
@Derivative {
  double CC1=C1/1.5; double CC2=C2/1.5; double fact=1.0;
  if (Fp.determin()<=0.0) { Fp=0.0; Fp[0]=Fp[1]=Fp[2]=1.0; }
  if (Dtime>0.0) fact=(tau-Time_ini)/Dtime;

  TENSOR2 Ft    = F - ((1.-fact)*Dtime)*dF;
  TENSOR2 Fe    = Ft*inverse(Fp);
  TENSOR2 R(usz), U(psz); Fe.strain_partion(R, U);
  TENSOR2 E     = U.log_tensor();

               sig = *elasticity*E;
  TENSOR2      Xv1 = CC1*alpha1;
```

```
  TENSOR2      Xv2 = CC2*alpha2;
  double       Rad = isotropic->radius(evcum);

  TENSOR2 sprime = deviator(sig);
  TENSOR2 sigeff = sprime-Xv1-Xv2;
  double  J      = sqrt(1.5*(sigeff|sigeff));
  double  f      = J - Rad;

  if (f>0.0) {
     TENSOR2          norm     = sigeff*(1.5/J);
                      devcum = flow->flow_rate(evcum,f);
                      dFp      = norm*devcum*Fp;
     if (CC1>0.0)     dalpha1  = devcum*(norm - (D1/CC1)*Xv1);
     if (CC2>0.0)     dalpha2  = devcum*(norm - (D2/CC2)*Xv2);
  }
}
```

@CalcGradF

Description:
This method is used for generalized midpoint implicit integration. The routine will be required to furnish the residual of incremental variable evolution equations, and the Jacobian matrix of all the residual equations with respect to the internal variables.

$$\Delta\chi = fn\left(\Delta\mathbf{g}_{tot}, \mathcal{A}\left(\chi_\theta\right)\right)$$

$$\chi_\theta = \chi_{ini} + \theta\Delta\chi$$

$$\mathcal{R}^k = \mathcal{F}^k(\chi_\theta) - \mathcal{F}_0$$

$$\mathcal{F}(\chi_\theta, \Delta\chi) = \mathcal{F}_0$$

$$\Delta\chi^{k+1} = \Delta\chi^k + [\nabla\mathcal{F}_\theta^k]^{-1}\mathcal{R}^k$$

CODE	DESCRIPTION
theta	the θ value used for the midpoint
f_0	vector of imposed variable increments
f_vec	vector for the variable residual
f_grad	Jacobian matrix storage
chi_vec	variables calculated at θ
d_chi	increment of integrated variables

CODE	DESCRIPTION
f_vec_*vname*$_i$	residual space for variable *vname*$_i$
d*vname*$_i$_d*vname*$_j$	Jacobian space for the residual equation of *vname*$_i$ with respect to the variable *vname*$_j$

Chapter 4

zLanguage

The base language

Introduction:

zLanguage is an end-user and powerfull scripting language. Its main goal is to provide to the end-user an acess to all ZeBuLoN internal code without recompiling anything. Using *zLanguage* allows end-users to enrich specific parts of the code, mainly post-processing calculations and boundary conditions. It also allows to generate parametric meshes. It is difficult to explain how rich and powerfull this extension is : the reader is invited to read the following pages which contain a lot of different examples from very different areas. There is especially some very interessting sections named "How to use a Zlanguage script to ... ".

zLanguage consists in several packages devoted to specific parts of the code. Usually these packages are overlaped : the master package inherits from the base package.

The purpose of this chapter is to explain the base grammar and syntax of *zLanguage*. *This chapter is not a programming course*; the user is supposed to know the basis of structural programming.

General remark:
It is necessary to declare all the object at the begin of each block (void, function if etc ..

Script file format

A script file is a regular plain text file as any other ZeBuLoN input file. All instructions must be ended by a semicolumn (except after a closing block brace); a line can contains multiple instructions. The two following zLanguage fragments are both legal and code the same algorithm piece :

```
void main()
{
  double a,b;

  a=2.;
  b=-1.;
}
```

```
void main()
{
  double a,b;

  a=2.; b=-1.;
}
```

One can split its script into different files, using `#include` directive to include one or more files into another. Suppose that file named 'f1.h' contains : `void do_something() ....` , one can use function `do_something` in another script file using :

```
#include <f1.h>

void main()
{
.....
  do\_something();
.....
}
```

In order to avoid infinitely recursive include files, a file is included by another only if this file has never been included before : suppose that a file named 'f1.z7p' includes 'f2.h' and that 'f2.h' includes 'f3.h', if 'f3.h' includes 'f1.z7p' or 'f2.h', a warning message is printed and the file is not included (to avoid infinite include recursion).

Include files are looked for first in Z-setdirectory ($Z7PATH/lib/Scripts) then in the current working directory.

Functions

zLanguage can be seen as a subset of C++. It provides the main characteristics and mechanisms of all object oriented languages, with the exception of defining new types : *zLanguage* programs can not define new types and can only use predefined types.

A script consists in a given number of functions. The definition of a function respects C/C++ standard :

```
double do_something(double a, int b, string s)
{
}
```

defines a function named "do_something" taking three agruments : a floating point value (a), an integer value (b) and a string value (s). This function is also supposed to return a floating point value.

One special function must at least be defined : the main function. An execution of any script always start in the main function which must be defined (unless otherwise informed) as :

```
void main()
{
}
```

void is a special type used to informe *zLanguage* that this function do not return a value. The parameter list is empty because main is the script starting point.

It is now time to write the well known "Hello world !" program using *zLanguage* : put the following script in a file named "hello.z7p" and run the command 'Zrun -zp hello.z7p'. The '-zp' switch starts *zLanguage* interpreter with the only the base package activated.

```
void main()
{
  ("Hello world !"+endl).print();
}
```

it should write the words "Hello world !" on a single line. Some explanations about this first script :

- "Hello world !" is a constant character string
- endl is a predefined *global* object (i.e. you can acess this object everywhere in your scripts). It is just a string object containing a new line character.
- operator + applied to strings is the concatenation operator

after it built the string to write, the script makes a calls to a *method*[1] named 'print' which write the contens of the string on the terminal. All objects in zLanguage have a 'print()' method to write their contens (sometimes this method is a default method doing nothing).

[1] A method is a function attached to a precise type. The contens of this function is type dependant.

A somewhat more complex example (write the following script in a file named e.g. 'test_0.z7p' and run the command 'Zrun -zp test_0.z7p') :

```
void set_parameters(double p1, double p2)
{
  p1=1.; p2=2.;
}

void main()
{
  double a,b;

  a=0.; b=0.;
  ("Before the call, a="+a+" and b="+b+endl).print();
  set_parameters(a,b);
  ("After the call, a="+a+" and b="+b+endl).print();
}
```

this script should normally print the following lines (if not, please contact your ZeBuLoN hotline) :

```
Before the call, a=0.000000 and b=0.000000
After the call, a=1.000000 and b=2.000000
```

This script demonstrates :

1. it is possible to add a string and a floating point value. The result is the concatenation between the first string and the string *representation* of the floating point value
2. all function parameters are passed by reference : it means that modify a function parameter modifies, in fact, the calling object.

Each function can contain a declaration block and a statements block. The declaration block consists in a list of typed list.

zLanguage statements

This section list all valid *zLanguage* statements, i.e. all basic instructions you can use in a script, with a trivial example for each statement (the left column contains the script; the right column contains execution result).

- `return(exp)` Exits from the current function and return the given expression to the caller. Care must be taken to not return a parameter in a function returning void !

```
double func()
{
  return(3.);
}

void main()
{
  ("Function returns : "
  +func()+endl).print();
}
```

Execution :
Function returns : 3.000000

- `if(exp) lstatement` Executes lstatement if exp if true (i.e. if exp is a positive integer).

```
void func(double value)
{
  if(value>=0.)
    ("pos"+endl).print();
  if(value<0.)
    ("neg"+endl).print();
}

void main()
{
  double a;

  a=2.;
  "First call : ".print();
  func(a);
  a=-1.;
  "Second call : ".print();
  func(a);
}
```

Execution :
First call : pos
Second call : neg

- `if(exp) lstatement1 else lstatement2` Executes lstatement1 if exp is true, lstatements2 otherwise.

```
void func(double value)
{
  if(value>=0.)
    ("pos"+endl).print();
  else
    ("neg"+endl).print();
}

void main()
{
  double a;

  a=2.;
  "First call : ".print();
  func(a);
  a=-1.;
  "Second call : ".print();
  func(a);
}
```

Execution :
First call : pos
Second call : neg

- `while (exp) lstatement` Executes lstatement while exp is true.

```
void main()
{
  double a;

  a=0.;
  while(a<10.) {
    ("a="+a+endl).print();
    a=a+1.;
  }
}
```

Execution :
a=0.000000
a=1.000000
a=2.000000
a=3.000000
a=4.000000
a=5.000000
a=6.000000
a=7.000000
a=8.000000
a=9.000000

- `do lstatement while(exp)` Same as the `while` statement except that lstatement is executed *before* the evaluation of exp (in other words, using `do` statement, lstatement is at least one time executed).

```
void main()
{
  double a;

  a=0.;
  do {
    ("a="+a+endl).print();
    a=a+1.;
  } while(a<10.);
}
```

Execution :
a=0.000000
a=1.000000
a=2.000000
a=3.000000
a=4.000000
a=5.000000
a=6.000000
a=7.000000
a=8.000000
a=9.000000

- `for(init;test;next) lstatement` This statement executes `init` and `test`. While `test` is true, it executes lstatement and `next`.

```
void main()
{
  double a;

  for(a=0.;a<10.;a=a+1) {
    ("a="+a+endl).print();
    a=a+1.;
  }
}
```

Execution :
a=0.000000
a=2.000000
a=4.000000
a=6.000000
a=8.000000

Debugger

zLanguage implementation provides a simple debugger to help debugging scripts. The debugger is launched every time ZSet wants to execute a script, as soon as a specific global parameter is given on the command line. This parameter is named `Zlanguage.Debug`. The user who wants to debug has just to launch ZSet with the option [`-s Zlanguage.Debug 1`] whatever the name of the command is (examples : `Zrun -s Zlanguage.Debug 1 -pp ...` or `Zmaster -s Zlanguage.Debug 1 -B ...`).

Each time the execution of a script is requested control is given to the internal debugger which prints a prompt (`zld ->`) and waits for commands. The main commands are summarized below (bracketed letters design short cuts : the user may type `break` or just `b`) :

- `[b]reak <li>` : inserts a breakpoint at line li
- `[c]ont` : continue a suspended execution
- `[d]delete <id>` : delete breakpoint ⟨id⟩
- `[i]nfo` : print breakpoints informations (usefull to get ⟨id⟩ to be given to [d]elete)
- `[n]ext` : continue execution and break at next line code
- `[p]rint <obj>` : prints the contens of object named ⟨obj⟩
- `[f]rame <id>` : set active frame to ⟨id⟩
- `[w]here` : print call stack and frame ids
- `[q]uit` : quit and abort execution of the current script
- `[h]elp` : lists debugger commands

A classical debugging session usually consists in defining some breakpoints ([b]reak command) and in printing some objects.

Object

Every accessible type in *zLanguage* is an object : it contains *methods* and *members*.

- A *method* is a specific function attached to a particular type : two methods attached to two different types may have the same name and do completely different things.
- A *member* is an object attached to a particular type.

Methods and members are accessed using '.' operator : `alpha.print()` executes the `print` method of alpha (depending of alpha's type); `beta.size=2;` assigns 2 to member `size` of variable beta (assuming that this assignement has a sense).

Templated types

Some object type are "templatized" : they are associated to an underlying type and the behavior of such type depends on the sub-type. An array object is a very classical templated type : it is an object by itself, but on the other hand the user has to specify what the type of the contained sub-objects is.

Such templates classes are intoduced using ⟨ and ⟩ brackets. The following piece of script declares an array of string and initializes it :

```
void main()
{
  int i;
  ARRAY<string> aos;

  aos.resize(5);
  for(i=0;i<!aos;i=i+1) aos[i]=("str "+i);
  ("Sub type of ARRAY<string> is : "
              +aos.type+endl).print();
  ("Nb elements : "+!aos+endl).print();
  for(i=0;i<!aos;i=i+1) (aos[i]+endl).print();
}
```

Execution :
Sub type of ARRAY⟨string⟩ is : string
Nb elements : 5
str 0
str 1
str 2
str 3
str 4

In the following sections templated types are explicitely signaled.

Base types

This section describes base zLanguage types (these types are always accessible, whatever package you are using). `i` denotes an integer value, `d` a floating point value.

- `int` : represents an integer.

  Operators :

  - `+ - * /` : classical arithmentic operators. With two integer operands, the result is always an integer (`/` represents the euclidian divide operator). When the second operand is a floting point value, the result is also a floting point value (`/` represents the floating point divide operator).
  - `%` : the remainder of an euclidian division (7%4=3). Usefull to test if an integer value is odd or even.
  - `=` : assignment operator (accept int or double).
  - `++` : increase value by one (3++=4).
  - `--` : decrease value by one (3--=2).
  - `> < >= <= == !=` : classical comparison operators.
  - `! | &` : boolean operation not, or and.

  Methods :

  - `print()` : print current value.

- `double` : represents a floating point value.

  Operators :

  - `+ - * /` : classical arithmentic operators.
  - `=` : assignment operator (accept int or double).
  - `> < >= <= == !=` : classical comparison operators.

  Methods :

  - `print()` : print current value.

- `string` : represents a character string.

  Operators :

  - `+` : concatenation operator. The second operand can be : a string, an integer or a floating point value.
  - `=` : assignment operator.
  - `== !=` : classical comparison operators.

  Methods :

  - `print()` : print current value.

- `VECTOR` : array of floating point values.

  Operators :

  - `+ -` : + and - operators. Both operands must be vectors with the same size.
  - `* /` : Multiply or divide a vector by a floating point or an integer value.
  - `=` : assignment operator. It is possible to assign a vector with another vector (component to component copy), with an integer or a floating point value (all component are set to this value), or with a TENSOR2 object.
  - `!` : current size.
  - `|` : dot product (v1|v2=sum(v1[i]*v2[i],i)).
  - `[i]` : component access (v[i] denotes the ith component of vector v).

  Methods :

  - `set(...)` : initialize vector. This method takes a list of floating or integer values, resize and initialize vector.
  - `norm()` : return the euclidian norm of the vector.
  - `resize(int)` : resize vector.
  - `print()` : print current value.

- `TENSOR2` : tensor of order 2.

  Operators :

  - `+ -` : + and - operators. Both operands must be tensors of order 2 with the same size.
  - `*` : Multiply a TENSOR2 by a floating point, an integer value, a TENSOR2 object or a TENSOR4 object.
  - `/` : Divide a TENSOR2 by a floating point or an integer value.
  - `=` : assignment operator. It is possible to assign a TENSOR2 with another TENSOR2 (component to component copy), with an integer or a floating point value (all component are set to this value), or with a VECTOR object.
  - `!` : current size.
  - `|` : return the contracted prodcut of two tensors.
  - `[i]` : component access (v[i] denotes the ith component of vector v).

  Methods :

  - `set(...)` : initialize TENSOR2. This method takes a list of floating or integer values. It resizes and initializes TENSOR2.
  - `mises()` : return mises norm.
  - `trace()` : return trace.
  - `deviator()` : return deviatoric part.
  - `add_sqrt2()` : add sqrt(2) terms 3,4 and 5 of the tensor.
  - `remove_sqrt2()` : remove sqrt(2) terms 3,4 and 5 of the tensor.

  - `eigen(VECTOR, TENSOR2)` : return the ....
  - `eigen_vecs(VECTOR)` : return the principal values of the tensor.
  - `resize(int)` : resize TENSOR2.
  - `print()` : print current value.

- `TENSOR4` : tensor of order 4.

  Operators :

  - `+ -` : + and - operators. Both operands must be tensors of order 4 with the same size.
  - `*` : Multiply a TENSOR4 by a floating point, an integer value or a TENSOR2 object.
  - `/` : Divide a TENSOR4 by a floating point or an integer value.
  - `=` : assignment operator. It is possible to assign a TENSOR4 with another TENSOR4 (component to component copy), with an integer or a floating point value (all component are set to this value).
  - `!` : current size.

  Methods :

  - `resize(int)` : resize TENSOR4.

- `MATRIX` : general matrix.

  Operators :

  - `+ -` : + and - operators. Both operands must be matrices with the same sizes.
  - `*` : Multiply a MATRIX. Right operand can be of type double, int, VECTOR or MATRIX.
  - `=` : assignment operator. It is possible to assign a MATRIX with another MATRIX (component to component copy), with an integer or a floating point value (all component are set to this value).
  - `(i,j)` : component access (m(i,j) denotes the component i,j of matrix m).

  Methods :

  - `set_rotation(double alpha)` : initialize matrix to be a 2D rotation operator with center (0.,0.) and angle alpha.
  - `resize(int n,int m)` : resize a MATRIX with n rows and m columns.
  - `print()` : print current value.

  Members :

  - `int n` (ro) : current number of rows.
  - `int m` (ro) : current number of columns.

- `SMATRIX` : general square matrix.

  Operators :

- `+ -` : + and - operators. Both operands must be SMATRIX with the same sizes.
- `*` : Multiply a SMATRIX. Right operand can be of type double, int, VECTOR, TENSOR2 or SMATRIX.
- `=` : assignment operator. It is possible to assign a SMATRIX with another SMATRIX (component to component copy), with an integer or a floating point value (all component are set to this value), with a TENSOR4 or a MATRIX.
- `(i,j)` : component access (m(i,j) denotes the component i,j of matrix m).

Methods :

- `resize(int n)` : resize a SMATRIX with n rows and n columns.
- `inverse()` : inverse a SMATRIX.
- `transpose()` : transpose a SMATRIX.
- `print()` : print current value.

Members :

- `int n` (ro) : current number of rows.
- `int m` (ro) : current number of columns.

- `ARRAY<T>` : general array of object. **Templated type**

Operators :

- `!` : current size.
- `[i]` : component access (a[i] denotes the ith object of ARRAY a).

Methods :

- `resize(int)` : resize an ARRAY. This operation is safe regarding to stored components : previously stored components are preserved.
- `print()` : print current value.

Members :

- `string type` (ro) : a string containing the sub-type of this templated object.

- `POINTER` : generic pointer.

Operators :

- `=` : pointer assignement. Right operand may be any object.

Methods :

- `set_type(string)` : define the underlying type.

Members :

- `string type` (ro) : current object type.
- `OBJECT object` (rw) : current object.

Note that this type **is not a templated type** (the reason is that this generic pointer can stores "on the fly" any object changing during the execution).

- `LIST` : list of objects.

  Operators :

  - `[i]` : acess the ith object of the list.
  - `!` : return current list size.

  Methods :

  - `add(OBJECT)` : add an object.
  - `reset()` : reinitialize the list (i.e. set the list length to zero).
  - `suppress(int i)` : suppress the ith element of the list.
  - `print()` : print all list objects.

- `Zfstream` : general file object.

  Operators :

  - `<<` : to write in a file (see $Z7PATH/tests/Program_test/Base_test/INP for examples

  Methods :

  - `open(STRING name, int t)` : to open a file named "name", t is open mode (ios::in to read, ios::out to write and ios::app to add something at the end of the file).
  - `close()` : to close a file

Predefined objects

This section lists all predefined objects to be used anywhere in a script.

1. Mathematical functions :
   - `sin(double)` : sin
   - `cos(double)` : cos
   - `tan(double)` : tan
   - `asin(double)` : asin
   - `acos(double)` : acos
   - `atan(double)` : atan
   - `sinh(double)` : sinh
   - `cosh(double)` : cosh
   - `tanh(double)` : tanh
   - `asinh(double)` : asinh
   - `acosh(double)` : acosh
   - `atanh(double)` : atanh
   - `floor(double d)` : rounds d downwards to the nearest integer
   - `ceil(double d)` : rounds d upwards to the nearest integer
   - `sqrt(double d)` : square root of d
   - `abs(double d)` : absolute value of d
   - `ln(double d)` : logarithm value of d, base e
   - `int(double d)` : converts double d to integer
   - `log(double d)` : logarithm value of d, base 10
   - `exp()` : exponential
   - `random()` : return a random floating point number in the interval [0..1]
2. Mathematical constants :
   - `pi` : π
   - `pi2` : $\pi/2$
   - `pi3` : $\pi/3$
   - `pi4` : $\pi/4$
   - `e` : $e = exp(1)$
   - `inve` : $1/e = exp(-1)$
3. String constants :
   - `endl` : "⟨CRLF⟩" (a string containing the single newline character)

4. General purpose objects :

   - `cout` : an object to handle streamed console outputs as C++ does. Use `cout<<object` to print something on terminal. One can also nest such instructions : `console<<"Hello world !"<<endl` prints the string "Hello world !" followed by an carriage return.
   - `cin` : an object to handle streamed console inputs as C++ does. Use `cin>>object` to get something from terminal.
   - `cflush` : used to flush output stream. The only method is `()` : the statement '`cflush();`' flushes output stream.
   - `ERROR` : used to trigger a fatal error. The only method is `(string)` : the statement '`ERROR("this is an error");` ' prints the words "this is an error" on the output stream and exits the script.
   - `plot` : an object used to draw a curve from vectors. See next section.
   - `runge` : an object used to solve differential systems. See next section.
   - `data_file` : an object used to load and save many things from a file. See next section.

Global objects plot, runge and data_file

`plot` :

Methods :

- `plot(VECTOR vx, VECTOR vy)` : plot curve $vy = f(vx)$ using gnuplot external program.

Members :

- `string labelx` (rw) : label of x axis.
- `string labely` (rw) : label of y axis.
- `string title` (rw) : title of the curve.
- `string style` (rw) : style of the curve (see gnuplot online help for more info).

A test program about `plot` object, which plots the sinus function :

```
void main()
{
  VECTOR vx,vy;
  double x;
  int i;

  vx.resize(20); vy.resize(!vx);
  for(i=0;i<!vx;i++) {
    x=2*pi/!vx*i;
    vx[i]=x; vy[i]=sin(vx[i]);
  }
  plot.labelx="x";
  plot.labely="sin(x)";
  plot.title="The sinus function";
  plot.style="linespoints";
  plot.plot(vx,vy);
}
```

`runge` :

Methods :

- `integrate(FUNCTION f, double xi, double xf, VECTOR chi, VECTOR dchi, int sample)` : solve the differential system $\partial\chi/\partial x = f(\chi, x)$ using a Runge-Kutta method. The f FUNCTION must be a function with signature : `void f(double t, VECTOR chi, VECTOR dchi)`. xi and xf are the lower and uppper bounds of the interval solution. sample is the number of samples across the interval solution. chi and dchi are two VECTORs.

- `xintegrate(FUNCTION f, double xi, double xf, VECTOR chi, VECTOR dchi)` : solve the same differential system, but gives only the solution at the upper bound (ie no interval sampling).
- `print()` : print current Runge-Kutta parameters.

Members :

- `double eps_r` (rw) : ε_r parameter.
- `double ymax_r` (rw) : $ymax_r$ parameter.

A test program about `runge` object, which solves the differential system

$$\frac{\partial f}{\partial x} = sin(x)$$

using `integrate` and `xintegrate`.

```
void main()
{
  solve_cos();
  x_solve_cos();
}

void derivative(double time, VECTOR chi, VECTOR dchi)
{
  dchi.resize(1);
  dchi[0]=sin(time);
}

void solve_cos()
{
  VECTOR vy,vx;
  double ti,tf;

  runge.eps_r=.001;
  runge.ymax_r=.001;
  runge.print();
  vy.resize(50);
  vx.resize(vy.size());
  ti=0.; tf=4*pi;
  vy[0]=-1.;
  runge.integrate(derivative,ti,tf,vx,vy,vx.size());
  plot.labelx="x";
  plot.labely="y";
  plot.style="linespoints lw 2 ps 2";
  plot.title="y=integrate(sin,0.,x)";
  plot.plot(vx,vy);
}
```

```
void x_solve_cos()
{
  VECTOR dchi,chi,vy,vx;
  double t0,t1;
  double ti,tf;
  int i;

  chi.resize(1);
  dchi.resize(1);
  runge.eps_r=.001;
  runge.ymax_r=.001;
  runge.print();
  vy.resize(50);
  vx.resize(vy.size());
  ti=0.; tf=2*pi;
  for(i=0;i<!vx;i=i+1) {
    if(i==0) t0=ti; else t0=vx[i-1];
    vx[i]=tf/vx.size()*i;
    t1=vx[i];
    if(i==0) chi[0]=0.; else chi[0]=vy[i-1];
    runge.xintegrate(derivative,t0,t1,chi);
    vy[i]=chi[0];
  }
  plot.labelx="x";
  plot.labely="y";
  plot.style="linespoints";
  plot.title="y=integrate(sin,0.,x)";
  plot.plot(vx,vy);
}
```

`data_file` :

Methods :

- `load_globals(string fname)` : try to open file named `fname`. If suceeded, look on this file to define global double variables.
- `load_vectors(string fname, VECTOR v1,...)` : try to open file named `fname`. If suceeded, try to load column vectors onto VECTORs v1,...
- `save_vectors(string fname, VECTOR v1,...)` : save VECTORs v1,... into a file named `fname`.

To explain how `load_globals` works, suppose that a file named 'globals.dat' contains :

```
A 1.
B 2.
```

then the two following functions f1 and f2 are strictly equivalent :

```
void f1()
{
  data_file.load_globals("globals.dat");
}

void f2()
{
  global double A,B;

  A=1.; B=2.;
}
```

Mesher object

It is also possible to acess some mesher objects inside *zLanguage*. This can be very usefull to make 3D extension, renumbering,...

- `UTILITY_MESH` : an object representing a mesh (a .geof file).

  Methods :

  - `load(string fmt)` : load the current mesh file name using format fmt.
  - `save()` : save mesh using current file name and .geof format.
  - `print()` : print some informations about the stored mesh.
  - `add(a)` : add an UTILITY_NODE, UTILITY_ELEMENT, UTILITY_NSET, UTILITY_ELSET or UTILITY_IPSET.
  - `int nb_node()` : return number of nodes
  - `int nb_elem()` : return number of elements
  - `int nb_nset()` : return number of node sets
  - `int nb_elset()` : return number of element sets
  - `int nb_ipset()` : return number of integration point sets
  - `UTILITY_NODE get_node(int n)` : return the nth node
  - `UTILITY_ELEMENT get_elem(int n)` : return the nth element
  - `UTILITY_NSET get_nset(int n)` : return the nth node set
  - `UTILITY_ELSET get_elset(int n)` : return the nth element set
  - `UTILITY_IPSET get_ipset(int n)` : return the nth integration point set

  Members :

  - `string name` (rw) : the mesh file name.

- `UTILITY_NODE` : a mesh node object.

  Methods :

  - `print()` : print some informations about the current node.
  - `set_rank(int)` : set rank of node.
  - `set_id(int)` : set id of node.
  - `int nb_elem()` : number of elements attached to node.
  - `UTILITY_ELEMENT get_elem(int n)` : return nth element attached to node.

  Members :

  - `int id` (ro) : node id.
  - `int rank` (ro) : node rank.
  - `VECTOR position` (rw) : geometrical position of node.

- UTILITY_ELEMENT : a mesh element object.

  Methods :

  - print() : print some informations about the current element.
  - set_rank(int) : set rank of element.
  - set_id(int) : set id of element.
  - int nb_node() : number of nodes attached to element.
  - UTILITY_NODE get_node(int n) : return nth node attached to element.

  Members :

  - int id (ro) : node id.
  - int rank (ro) : node rank.
  - string type (rw) : element type.

- UTILITY_NSET : mesh node set object.

  Methods :

  - print() : print some informations about the current node set.
  - suppress(UTILITY_NODE n) : suppress node n from current node set.
  - add(UTILITY_NODE n) : add node n to current node set.
  - reset() : reinitialize curent set (zero size).
  - int nb_node() : return number of node in set.
  - UTILITY_NODE get_node(int n) : return nth node in set.

  Members :

  - string name (rw) : node set name.

- UTILITY_ELSET : mesh element set object.

  Methods :

  - print() : print some informations about the current element set.
  - suppress(UTILITY_ELEMENT n) : suppress element n from current node set.
  - add(UTILITY_ELEMENT n) : add element n to current node set.
  - reset() : reinitialize curent set (zero size).
  - int nb_elem() : return number of element in set.
  - UTILITY_ELEMENT get_elem(int n) : return nth element in set.

  Members :

  - string name (rw) : node element name.

- UTILITY_IPSET : mesh integration point set object.

  Methods :

- `print()` : print some informations about the current set.
- `suppress(n)` : suppress integration point from current set. n can be either an UTILITY_ELEMENT or an UTILITY_NODE.
- `add(UTILITY_ELEMENT n, int i)` : add (element n, ip i) to current set.
- `reset()` : reinitialize curent set (zero size).
- `int nb_elem()` : return number of ip in set.
- `UTILITY_ELEMENT get_elem(int n)` : return nth element in set.
- `int get_ip(int n)` : return nth ip in set.

Members :

- `string name` (rw) : set name.

- `MESHER_UNION` : an object to make union between different meshes.

  Methods :

  - `union(UTILITY_MESH m)` : make the union between all different meshes previously designated by method `add()`. The resulting mesh is stored into m.
  - `add(UTILITY_MESH m)` : add a mesh m for later union.
  - `reset()` : reinitialize this object (forget all meshes previously designated by method `add()`.

  Members :

  - `double tolerance` (rw) : tolerance parameter (two nodes are considered to be the same node if the distance between them is lower than tolerance).
  - `string elset` (rw) : elset name to be created with all elements added by this object.

- `MESHER_EXTENSION` : an object to make a 3D extension on an utility_mesh

  Methods :

  - `apply(UTILITY_MESH m)` : make the extension using current parameters.

  Members :

  - `string elset` (rw) : name of the elset to be extended.
  - `string elset2` (rw) : name of the second optionnal elset to be extended (see zMesher manual for more informations).
  - `double progression` (rw) : geometrical progression of the extension.
  - `double distance` (rw) : distance of the extension along 3rd axis.
  - `int cuts` (rw) : number of elements along 3rd axis.
  - `VECTOR direction` (rw) : direction of the 3rd axis.

- `RENUMBERING` : an object to renumber a mesh (to lower front or band width).

  Methods :

– `renumbering(UTILITY_MESH m)` : renumber given mesh m.

Members :

– `int type` (rw) : front or band width optilisation (default=0, front optimization).
– `int subdomain` (rw) : equal 1 if domain renumbering (0 by default).
– `double w1` (rw) : parameter of the modified Sloan method.
– `double w2` (rw) : parameter of the modified Sloan method.

Post Processing

Local post processing

It is very easy to write a local post processing criterion using zLanguage. Just write your script assuming that two global variables of type VECTOR already exists (named `in` and `out`). For instance to compute mises norm, one can use :

```
ARRAY<STRING> input()
{
  ARRAY<STRING> i;

  i.resize(4); i[0]="sig11"; i[1]="sig22"; i[2]="sig33"; i[3]="sig12";
  return(i);
}

ARRAY<STRING> output()
{
  ARRAY<STRING> o;

  o.resize(1); o[0]="z7pmises";
  return(o);
}

void compute()
{
  TENSOR2 sigma;
  global double max_mises,min_mises;
  double mises;

  sigma.reassign(4,in,0);
  sigma.add_sqrt2();
  mises=sigma.mises();
  out.resize(1);
  out[0]=mises;
  if(out[0]<min_mises) min_mises=out[0];
  if(out[0]>max_mises) max_mises=out[0];
}
```

Then write a "classical" post-processing input file :

```
****post_processing
  ***local_post_processing
    **elset ALL_ELEMENT
    **output_number 1-2
    **file integ
    **process z7p
      *program post.z7p
****return
```

Input components are automatically extracted using the script function called `input`; output components are taken from function `output`.

It is also possible to supply two additional function in your script file : `void initialize()` and `void destroy()` which are called just before and just after executing the post computation. It allows for instance global variables initialization (before computation) and printing (after).

.2 Global post processing

It is also possible to write global post processing scripts. You only have to assume that two global variables are defined : `ape` and `apn` which are arrays of POST_ELEMENT and of POST_NODE. The input file is exactly the same (except for the global options) :

```
****post_processing
  ***global_post_processing
    **elset ALL_ELEMENT
    **output_number 1-2
    **file integ
    **process z7p
      *program post.z7p
****return
```

The script file could be, for instance (it computes and prints the maximum and minimum values of mises stress over the mesh) :

```
ARRAY<STRING> input()
{
  ARRAY<STRING> i;

  i.resize(4); i[0]="sig11"; i[1]="sig22"; i[2]="sig33"; i[3]="sig12";
  return(i);
}

ARRAY<STRING> output()
{
  ARRAY<STRING> o;

  o.resize(1); o[0]="z7pmises";
  return(o);
}

void initialize()
{
  global double max_mises,min_mises;

  max_mises=-MAX_DOUBLE; min_mises=MAX_DOUBLE;
}

void destroy()
{
  global double max_mises,min_mises;
  Zfstream f;

  endl.print();
```

```
  ("Max mises ="+max_mises+endl).print();
  ("Min mises ="+min_mises+endl).print();
  cflush();
  f.open("arm2_glob.post",18);
  f<<max_mises<<" "<<min_mises<<endl;
  f.close();
}

void compute()
{
  int i,j;
  TENSOR2 sigma;
  global double max_mises,min_mises;
  double mises;

  for(i=0;i<!ape;i=i+1) {
    for(j=0;j<ape[i].nb_idata();j=j+1) {
      sigma.reassign(4,ape[i].idata(j).data,0);
      mises=sigma.mises();
      ape[i].idata(j).out[0]=mises;
      if(mises<min_mises) min_mises=mises;
      if(mises>max_mises) max_mises=mises;
    }
  }
}
```

.3 Post processing end user objects

- `INTEG_DATA` : represents datas coming from .integ or .ctnod file (ie finite element results at integration points)

  Operators : none

  Methods : none

  Members :

  - `VECTOR out` : the result vector. Its size should be equal to the number of components declared in function `output`
  - `VECTOR data` : the input vector. Its size should be equal to the number of components declared in function `input`

- `NODE_DATA` : represents datas coming from .node file (ie finite element results at nodes)

  Operators : none

  Methods : none

  Members :

  - `VECTOR out` : the result vector. Its size should be equal to the number of components declared in function `output`
  - `VECTOR data` : the input vector. Its size should be equal to the number of components declared in function `input`

- `POST_NODE` : an entity to store nodal datas

  Operators :

  - `int operator!()` : return the number of attached `NODE_DATA`

  Methods :

  - `POST_NODE data(int n)` : return the nth `NODE_DATA` object

  Members : none

- `POST_ELEMENT` : an entity to store element datas

  Operators : none

  Methods :

  - `int nb_idata()` : return the number of integ datas (should be equal to the number of integration points)
  - `int nb_ndata()` : return the number of ctnod datas (should be equal to the number of nodes)
  - `ELEM_DATA idata(int n)` : return the nth integ data
  - `ELEM_DATA ndata(int n)` : return the nth ctnod data

- `void start(MATRIX elem_coord)` : starts an element loop
- `void next(MATRIX elem_coord)` : next integration point in the element loop
- `int ok()` : return 0 if the number of integration points has been excedeed, 1 otherwise
- `VECTOR shape()` : return shape vector for the current integration point
- `VECTOR shape_inv()` : return "inverse" shape vector for the current integration point
- `void get_elem_coord(MATRIX &m)` : return int matrix m the elements coordinates
- `void get_position_of_integration_point(VECTOR &v)` : return in vector v the coordinate of the current integration point
- `void integrate(T &v, T &tot)` : increment `tot` with the integral contribution associated with the current integration point. `T` may be `double`, `VECTOR` or `MATRIX`.

Members : none

```
Post Processing
 Local post processing
```

How to use a Zlanguage script to produce parametric meshes?

The association of zMaster module and zLanguage provides a very fast and efficient way to produce parametric meshes. One have simply to write a script describing the geometry of the mesh and run this script through zMaster module. To activate master objects, the code must be started with either -B option (batch meshing) or -G option (graphical interface).

A script started this way has access to a number of new object types.

.1 Master zLanguage types

- `POINT` : geometrical point.

  Operators :

  - `+ -` : + and - operators. Right operand can be a POINT or a VECTOR object.
  - `* /` : multiply or divide coordinates by a double.
  - `=` : assignment operator. Right operand can be of type : double, POINT or VECTOR.

  Methods :

  - `print()` : print current value.

  Members :

  - `double x` (rw) : x coordinate.
  - `double y` (rw) : y coordinate.
  - `double z` (rw) : z coordinate.

- `LINE` : geometrical line.

  Operators :

  - `=` : assignment operator. Right operand can be of type LINE.

  Methods :

  - `bind(POINT a, POINT b)` : anchors line between POINTs a and b. Care must be taken of order : a is always the *final* point and b the *starting* point.
  - `bind1(POINT a)` : set line start point.
  - `bind2(POINT a)` : set line end point.
  - `length()` : return the line length.
  - `set_nodes(int n, double p)` : set the number n of edge nodes on the line, with a geometrical progression of p (use p=1. for linear progression).
  - `print()` : print current value.

  Members :

  - `POINT p1` (rw) : start point. This member must not be requested before a call to bind() or bind1().
  - `POINT p2` (rw) : end point. This member must not be requested before a call to bind() or bind2().

- `ARC` : geometrical arc.

  Operators :

  - `=` : assignment operator. Right operand can be of type ARC.

Methods :

- `set_center(POINT c)` : set center POINT c.
- `set_parameters(POINT c, double r, double a1, double a2, POINT p1, POINT p2)` : set all arc parameters : center c, radius r, start angle a1 (rad), end angle a2 (rad), start POINT p1, end POINT p2.
- `set_arc(POINT c, double r, double a1, double a2)` : initialize ARC to be an arc with center c, radius r, start angle a1 and end angle a2. POINTs p1 and p2 are internally automatically created.
- `circle(POINT c, double r)` : initialize ARC to be a circle of center c and radius r.
- `set_nodes(int n, double p)` : set the number n of edge nodes on the line, with a geometrical progression of p (use p=1. for linear progression).
- `print()` : print current value.

Members :

- `double radius` (rw) : ARC radius.
- `double aplha1` (rw) : start angle.
- `double aplha2` (rw) : end angle.

- `RULED` : a ruled meshed domain (i.e. a domain with three or four sides, meshed with a ruled method).

  Operators :

  - `=` : assignment operator. Right operand can be of type ARC.

  Methods :

  - `add(int s, EDGE e)` (an EDGE is either a LINE or an ARC) : add edge e to the s^{th} side of domain.
  - `remesh()` : mesh only this domain.
  - `reset()` : reinitialize this domain.
  - `print()` : print current value.

  Members :

  - `int method` (rw) : linear or not method (see zMaster manual for more information).
  - `int fake4` (rw) : set this member to 1 for three sides domains.
  - `string element_type` (rw) : element type to be created.
  - `string elset_type` (rw) : give an elset name to the domain

- `DELAUNAY` : a Delaunay meshed domain.

  Methods :

  - `add(EDGE e)` (an EDGE is either a LINE or an ARC) : add edge e. Please respect order (see zMaster manual for more informations) !

  - `remesh()` : mesh only this domain.
  - `reset()` : reinitialize this domain.
  - `print()` : print current value.

  Members :

  - `int method` (rw) : method (see zMaster manual for more information).
  - `double propagation` (rw) : method parameter.
  - `double critical_angle` (rw) : method parameter.
  - `string element_type` (rw) : element type to be created.
  - `string elset_type` (rw) : give an elset name to the domain

- `PAVING` : a paving meshed domain.

  Methods :

  - `add(EDGE e)` (an EDGE is either a LINE or an ARC) : add edge e. Please respect order (see zMaster manual for more information) !
  - `remesh()` : mesh only this domain.
  - `reset()` : reinitialize this domain.
  - `print()` : print current value.

  Members :

  - `double seam_factor` (rw) : method parameter.
  - `double size_factor` (rw) : method parameter.
  - `string element_type` (rw) : element type to be created.

.2 Memory management

Memory is automatically managed inside *zLanguage*: objects are created and destroyed on demand, when it is necessary. As *zLanguage*and *zMaster* are two separate modules of Z-set, the user has to tell *zLanguage*which objects have not to be deleted (because these objects, for instance `PAVING` will be used by *zMaster*). This is done using the method `link()` : all master objects have a method named `link()` which ensures that the current object *and all its attachments* will not be deleted. Usually, one has only to call `link()` for domains since this call ensures also that all lines, arc, points, etc... belonging to this domain will not be deleted.

.3 Interfacing with graphical user interface

It is possible to run script in zMaster graphical user interface : choose 'Run script' option in 'Action' menus. A choice window appears with two script lists : the top list contains scripts located in Z-setdirectories ($Z7PATH/lib/Scripts), the second list contains scripts in the current working directory. Choose a script then click on 'Setup' button : anew window will appear asking for some parameters with default values. How is it possible?

To activate this possibility, one has only to put at the beginning of its script a special commented header describing the script and its parameters. The syntax of this header is as follows:

```
//
// MASTERSCRIPT
//   begin
//     STRING name  Name A0
//     STRING etype Element c2d
//     double cx Cx 0.
//     double cy Cy 0.
//     double alpha  Angle 0.
//     double radius  Radius 1.
//     double rel_dist rel_dist 0.66
//     int    nedges Nedges 4
//     bool   use_inner_line InnerLine 1
//   end
//
```

The two first lines and the last line are mandatory and conventional. The remainder lines describe some global variables with their type (first word), the variable name inside the script (second word), the name which will appears in dialog window (third word) and at last a default value. Currently only types int, bool, double and string are supported (it is not possible to declare an ARRAY this way, for instance).

One has only to use these global variables inside the script : these variables will be automatically initialized according to the user specifications. A simple example is :

```
//
// MASTERSCRIPT
//   begin
//     double cx Cx 0.
//     double cy Cy 0.
//   end
//
void main()
{
  global double cx,cy;

  ("Got cx="+cx+" and cy="+cy+endl).print();
}
```

.4 Examples

This section contains some example (very easy to more complex) concerning the use of *zLanguage*to make parametric meshes.

• A very simple example : how to mesh a unit square with paving method. Put the following lines in a file named 'square.z7p'.

```
void main()
{
  POINT a,b,c,d;
  LINE c1,c2,c3,c4;
  PAVING square;

  a.x=0.; a.y=0.; b.x=1.; b.y=0.;
  c.x=1.; c.y=1.; d.x=0.; d.y=1.;
//
  c1.bind(b,a); c2.bind(c,b);
  c3.bind(d,c); c4.bind(a,d);
//
  c1.set_nodes(5,1.); c2.set_nodes(6,1.);
  c3.set_nodes(7,1.); c4.set_nodes(8,1.);
//
  square.add(c1); square.add(c2);
  square.add(c3); square.add(c4);
//
  square.name="square";
  square.element_type="c2d4";
  square.elset_name="square";
  square.mesh.name="square.geof";
//
  square.make_connectivity();
  square.link();
//
  master("save_as square.mast");
}
```

please note the last instruction (`master("save_as square.mast");`)... `master` is a global object which allows to pass string messages to zMaster. The message "save_as toto.mast" asks zMaster to save the current geometry in a file named 'toto.mast'.

You can run this script either using 'Zrun -B square.z7p' (the script will be executed and a master file created), 'Zmaster square.z7p' (the geometry will be created just after launch of zMaster), or using 'Run script' option in 'Action' menu.

• how to mesh a parametric AE specimen...

```
//
// MASTERSCRIPT
//   begin
```

```
//     STRING name Specimen_name ae_specimen
//     STRING eltype Element_type c2d8
//     double height Specimen_height 10.
//     double width  Specimen_width 3.
//     double depth  Notch_depth 1.5
//     double radius Notch_radius 1.5
//     int n_left N_edges_left 10
//     int n_right N_edges_right 8
//     int n_top  N_edges_top 5
//     int n_bottom  N_edges_bottom 5
//     int n_arc  N_edges_arc 5
//     bool reduce Reduced_integration 0
//   end
//
void main()
{
  global double depth,radius,height,width;
  double theta1,theta2;
  double s;
  global int n_left,n_right,n_top,n_bottom,n_arc;
  int tot_edge;
  global int reduce;
  global string name,eltype;
  PAVING specimen;
  POINT A,C,D,F,G,H,G1,F1;
  LINE l1,l2,l3,l4;
  ARC a1;
  LISET left,right,top,bottom,notch;

  ("Got the following values :"+endl).print();
  ("width="+width+endl).print();
  ("height="+height+endl).print();
  ("depth="+depth+endl).print();
  ("radius="+radius+endl).print();
  endl.print();
//
  A.x=0.; A.y=0.;
  C.x=0.; C.y=height;
  D.x=width; D.y=C.y;
  H.x=radius+width-depth; H.y=0.;
  G.x=width-depth; G.y=H.y;
  if(G.x<=0.) ERROR("Invalid width/depth");
//
  s=(width-H.x)/radius;
  if((s>0.)|(s<-1.)) ERROR("Invalid radius");
  theta1=acos((width-H.x)/radius);
  theta2=pi;
```

```
  F.x=width;
  F.y=H.y+radius*sin(theta1);
//
  tot_edge=n_top+n_bottom+n_left+n_right+n_arc;
  if(tot_edge%2) {
    ("Odd number of edges detected. Adding one edge to top side."+endl).print();
    n_top=n_top+1;
  }
  l1.bind(D,F); l2.bind(C,D); l3.bind(A,C); l4.bind(G,A);
  l1.set_nodes(n_right,1.);
  l2.set_nodes(n_top,1.);
  l3.set_nodes(n_left,1.);
  l4.set_nodes(n_bottom,1.);
  a1.set_parameters(H,radius,theta1,theta2,F,G);
  a1.set_nodes(n_arc,1.);
//
  specimen.add(l1);
  specimen.add(l2);
  specimen.add(l3);
  specimen.add(l4);
  specimen.add(a1);
  specimen.name=name;
  specimen.elset_name=name;
  specimen.element_type=eltype;
  right.add(l1); right.name="right";
  top.add(l2); top.name="top";
  left.add(l3); left.name="left";
  bottom.add(l4); bottom.name="bottom";
  notch.add(a1); notch.name="notch";
//
  specimen.remesh();
//
  left.link(); right.link(); top.link(); bottom.link(); notch.link();
  specimen.link();
//
  master("save_as toto.mast");
  specimen.mesh.name="toto.geof";
  specimen.mesh.save();
}
```

• A very complex example : how to mesh a 'real' gear, using involute curves. This example makes an intensive use of many objects, and is thus very interesting to show the possibilities of the coupling between zMaster and zLanguage.

```
void do_involute(int direction, VECTOR vx, VECTOR vy, double re, double ri, int disc)
{
//
```

```
// Makes a involute curve
//
  double theta,theta_max,t;
  double l;
  double a,b,c,delta;
  double d1,d2,dt;
  int i,nnode;

  vx.resize(disc+1); vy.resize(!vx);
  theta_max=1./ri*sqrt(re^2.-ri^2.);
  ("theta_max="+theta_max+endl).print();
  l=.5*ri*theta_max^2./disc;
//
  if(direction==1) { vx[0]=ri; vy[0]=0.; }
  else { vx[!vx-1]=ri; vy[!vy-1]=0.; }
  theta=0.;
  for(i=0;i<!vx;i=i+1) {
    if(direction==1) {
      vx[i]=ri*(cos(theta)+theta*sin(theta));
      vy[i]=ri*(sin(theta)-theta*cos(theta));
      t=sqrt(2.*l/ri+theta*theta);
      dt=t-theta;
      theta=theta+dt;
    } else if(direction==-1) {

      vx[!vx-i-1]=ri*(cos(theta)+theta*sin(theta));
      vy[!vy-i-1]=ri*(sin(theta)-theta*cos(theta));
      t=-sqrt(2.*l/ri+theta*theta);
      dt=t-theta;
      theta=theta+dt;
    } else ERROR("Unkown direction : "+direction);
  }
}

void do_tooth_points(VECTOR r_vx, VECTOR r_vy, VECTOR l_vx, VECTOR l_vy,
                     double awidth, double re, double ri, int disc, double tm)
{
  VECTOR p;
  MATRIX rot;
  double alpha;
  int i,d;

//
  do_involute(1,r_vx,r_vy,re,ri,disc);
  tm=acos(r_vx[!r_vx-1]);
  do_involute(-1,l_vx,l_vy,re,ri,disc);
//
  alpha=(awidth-2*tm)/2.;
//
  rot.set_rotation(-(tm+alpha));
  p.resize(2);
  for(i=0;i<!r_vx;i=i+1) {
    p[0]=r_vx[i]; p[1]=r_vy[i];
```

```
    p=rot*p;
    r_vx[i]=p[0]; r_vy[i]=p[1];
  }
//
  rot.set_rotation(tm+alpha);
  p.resize(2);
  for(i=0;i<!l_vx;i=i+1) {
    p[0]=l_vx[i]; p[1]=l_vy[i];
    p=rot*p;
    l_vx[i]=p[0]; l_vy[i]=p[1];
  }
}

void do_tooth(double tm, double awidth, double re, double ri, int disc, double angle, ARRAY<LINE> lines)
{
  VECTOR r_vx,r_vy;
  VECTOR l_vx,l_vy;
  VECTOR p;
  MATRIX rot;
  int i,iline,jpoint;
  ARRAY<POINT> points;

  do_tooth_points(r_vx,r_vy,l_vx,l_vy,awidth,re,ri,disc,tm);
  rot.set_rotation(angle);
  p.resize(2);
  for(i=0;i<!r_vx;i=i+1) {
    p[0]=r_vx[i]; p[1]=r_vy[i];
    p=rot*p;
    r_vx[i]=p[0]; r_vy[i]=p[1];
  }
  for(i=0;i<!l_vx;i=i+1) {
    p[0]=l_vx[i]; p[1]=l_vy[i];
    p=rot*p;
    l_vx[i]=p[0]; l_vy[i]=p[1];
  }
//
  lines.resize(2*(!l_vx-1));
  points.resize(2*(!l_vx));
  jpoint=0;
  for(i=0;i<!r_vx;i=i+1) {
    points[jpoint].x=r_vx[i];
    points[jpoint].y=r_vy[i];
    jpoint=jpoint+1;
  }
  for(i=0;i<!l_vx;i=i+1) {
    points[jpoint].x=l_vx[i];
    points[jpoint].y=l_vy[i];
    jpoint=jpoint+1;
  }
//
  jpoint=0;
  iline=0;
  for(i=0;i<!r_vx-1;i=i+1) {
```

```
    lines[iline].bind(points[jpoint+1],points[jpoint]);
    iline=iline+1;
    jpoint=jpoint+1;
  }
  jpoint=jpoint+1;
  for(i=0;i<!l_vx-1;i=i+1) {
    lines[iline].bind(points[jpoint+1],points[jpoint]);
    iline=iline+1;
    jpoint=jpoint+1;
  }
}

void do_gear(DELAUNAY domain, LISET ext, int n_teeth, double awidth, double re, double ri, int dis
{
  ARRAY< ARRAY<LINE> > all_lines;
  ARRAY<ARC> arcs,other_arcs;
  ARRAY<POINT> pts1,pts2,ppts1,ppts2;
  double tm,wa,space,angle,alpha1,alpha2;
  int i,itooth;
  VECTOR p1,p2;

  space=(2*pi-n_teeth*awidth)/n_teeth;
//
  all_lines.resize(n_teeth);
  other_arcs.resize(n_teeth);
  arcs.resize(n_teeth);
  pts1.resize(n_teeth);
  pts2.resize(n_teeth);
  ppts1.resize(n_teeth);
  ppts2.resize(n_teeth);
//
  for(itooth=0;itooth<n_teeth;itooth=itooth+1) {
    ("Tooth n. "+itooth+endl).print(); cflush();
    angle=2*pi/n_teeth*(itooth+1);
    do_tooth(tm,awidth,re,ri,disc,angle,all_lines[itooth]);
    wa=(awidth-2*tm)/2.;
    arcs[itooth].set_parameters(center,re,angle-wa,angle+wa,ppts1[itooth],ppts2[itooth]);
    arcs[itooth].set_nodes(dea,1.);
  }
//
  for(itooth=0;itooth<n_teeth;itooth=itooth+1) {
    ("Arc n. "+itooth+endl).print(); cflush();
    angle=2*pi/n_teeth*(itooth+1);
    alpha1=awidth/2.+angle;
    alpha2=alpha1+space;
    other_arcs[itooth].set_parameters(center,ri,alpha1,alpha2,pts1[itooth],pts2[itooth]);
    if(itooth==0) other_arcs[itooth].set_nodes(dia+1,1.);
    else other_arcs[itooth].set_nodes(dia,1.);
  }
//
  domain.name="gear";
  domain.elset_name="gear";
  domain.element_type="c2d3";
```

```
  domain.propagation=1;
//
  for(itooth=0;itooth<n_teeth;itooth=itooth+1) {
    for(i=0;i<!(all_lines[itooth])/2;i=i+1) domain.add(all_lines[itooth][i]);
    for(i=0;i<!(all_lines[itooth])/2;i=i+1) ext.add(all_lines[itooth][i]);
    domain.add(arcs[itooth]);
    for(i=!(all_lines[itooth])/2;i<!all_lines[itooth];i=i+1) {
      domain.add(all_lines[itooth][i]);
      ext.add(all_lines[itooth][i]);
    }
    domain.add(other_arcs[itooth]);
  }
  domain.link();
//  domain.make_connectivity();
}

void main()
{
  global POINT center;
  VECTOR px,py;
  double r_int,r_ext;
  global double ang_t;
  int d_i_arc,d_e_arc,di,nteeth,i;
  double awidth,rtrou;
  int n_trou,neh;
  DELAUNAY domain;
  LISET internal,internal2,external;
  ARRAY<POINT> pts_trou;
  ARRAY<LINE> trou;
  double angle;

  r_ext=1.; r_int=.725;
  nteeth=8; di=9; d_i_arc=2; d_e_arc=4;
  awidth=pi/6.;
//
  n_trou=5; rtrou=.15; neh=4;
//
  center.x=0.; center.y=0.;
//
  do_gear(domain,external,nteeth,awidth,r_ext,r_int,di,d_e_arc,d_i_arc);
//
  pts_trou.resize(n_trou);
  trou.resize(n_trou);
  for(i=0;i<n_trou;i=i+1) {
    pts_trou[i].x=center.x+rtrou*cos(2.*pi/n_trou*i+angle);
    pts_trou[i].y=center.y+rtrou*sin(2.*pi/n_trou*i+angle);
  }
  for(i=0;i<n_trou-1;i=i+1)
    trou[i].bind(pts_trou[n_trou-i-2],pts_trou[n_trou-i-1]);
  trou[n_trou-1].bind(pts_trou[n_trou-1],pts_trou[0]);
  for(i=0;i<!trou;i=i+1) {
    trou[i].set_nodes(neh,1.);
  }
```

```
  for(i=0;i<!trou;i=i+1) domain.add(trou[i]);
  for(i=0;i<!trou;i=i+1) internal.add(trou[i]);
  for(i=0;i<!trou/2;i=i+1) internal2.add(trou[i]);
  internal.name="internal"; internal2.name="internal2";
  internal.elset_name="internal"; internal2.elset_name="internal2";
  external.name="external";
  external.elset_name="external";

  domain.link(); internal.link(); internal2.link(); external.link();
}
```

How to use a Zlanguage script in optimizer ?

One can use *zLanguage*base package inside an optimization loop. Let us consider the following differential system :

$$\frac{\partial \chi}{\partial x} = B + A\sin(x)\chi(x)$$

The optimization problem to be solved is to find parametres A and B so that curve $\chi = f(x)$ fits a predefined "experimental" result.

First write a script solving this differential system for any parameter A and B (these parameters are loaded from a file names 'solve.dat') ,:

```
void main()
{
  VECTOR vx,vy;
  double ti,tf;
  global double A,B;

  data_file.load_globals("solve.dat");
  runge.eps_r=.001;
  runge.ymax_r=.001;
  vy.resize(50);
  vx.resize(vy.size());
  ti=0.; tf=2.;
  vy[0]=0.;
  runge.integrate(derivative,ti,tf,vx,vy,vx.size());
  data_file.output_vectors("solve.test",vx,vy);
}

void derivative(double time, VECTOR chi, VECTOR dchi)
{
  dchi.resize(1);
  dchi[0]=B+A*sin(time)*chi[0];
}
```

In file 'solve.dat.tmpl', just put (as usual with zOptimizer) the unknowns :

```
A ?A
B ?B
```

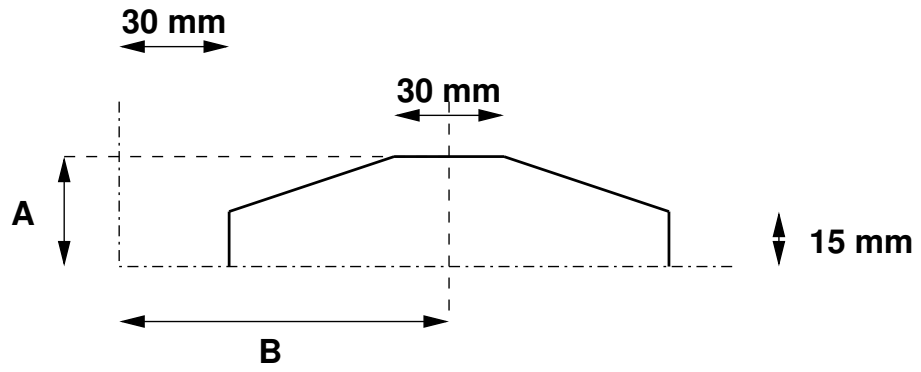
And use the following input file (with 'Zrun -o' command) to find A and B :

```
****optimize sqp
  ***files solve.dat
  ***shell Zrun -zpt opti.z7p 2>&1 > /dev/null
```

```
  ***values
    A 5. min .5 max 50.
    B 10. min 1. max 100.
  ***compare
    g_file_file solve.test 1 2 solve.ref 1 2 weight 50.
****return
```

.1 An example using zLanguage/zMaster and zOptimiser

The goal of this problem is to optimize geometrical parameters of a turbine disk to lower both mises equivalent stress and mises equivalent strain. The axisymmetric geometry, with two unknown parameters A and B is the following :



The mesh script file (named 'mesh.z7p') is the following :

```
void main()
{
  global double A,B;
  double dl;
  POINT a,b,c,d,ee,f;
  LINE l1,l2,l3,l4,l5,l6;
  LISET li;
  DELAUNAY domain;
  RENUMBERING renum;

  data_file.load_globals("AB.dat");
//
  dl=3.;
//
  a.x=30.; a.y=0.;
  b.x=a.x+120.; b.y=0.;
  c.x=b.x; c.y=30.;
  d.x=B+15.; d.y=A;
  ee.x=B-15.; ee.y=A;
  f.x=a.x; f.y=30.;
//
  l1.bind(b,a); l2.bind(c,b);
  l3.bind(d,c); l4.bind(ee,d);
  l5.bind(f,ee); l6.bind(a,f);
//
  l1.set_nodes(int(l1.length()/dl)+1,1.);
  l2.set_nodes(int(l2.length()/dl)+1,1.);
  l3.set_nodes(int(l3.length()/dl)+1,1.);
```

```
  l4.set_nodes(int(l4.length()/dl)+1,1.);
  l5.set_nodes(int(l5.length()/dl)+1,1.);
  l6.set_nodes(int(l6.length()/dl)+1,1.);
//
  domain.add(l1); domain.add(l2);
  domain.add(l3); domain.add(l4);
  domain.add(l5); domain.add(l6);
  li.add(l1); li.name="bottom";
//
  domain.link(); li.link();
  domain.mesh.name="disk.geof";
  domain.element_type="cax6";
  domain.name="disk";
  domain.remesh();
  renum.renumbering(domain.mesh);
  domain.mesh.save();
}
```

The file 'AB.dat.tmpl' only contains :

```
A ?A
B ?B
```

The finite element input file ('mesh.inp') is :

```
****calcul
  ***mesh
    **file disk.geof
  ***resolution newton
    **sequence
      *time 100.
      *algorithm eeeeee
      *increment 1
      *iteration 10
  ***bc
    **impose_nodal_dof
      bottom U2 0.0
    **centrifugal ALL_ELEMENT (0. 0.) d2 1.e5 time
  ***material
      *file mesh.inp
****return

***behavior linear_elastic
 **elasticity
     young 200000.0
     poisson 0.3
 ***coefficient
   masvol 1.e-6
```

```
***return

****post_processing
  ***local_post_processing
    **elset ALL_ELEMENT
    **output_number 1
    **file integ
    **process mises sig
    **process mises eto
  ***global_post_processing
    **output_number 1
    **elset ALL_ELEMENT
    **file integ
    **process max sigmises
    **process max etomises
****return
```

And the optimizer input file ('opti.inp') is :

```
****optimize sqp
  ***files AB.dat
  ***shell
    Zrun -B mesh.z7p
    Zrun mesh
    Zrun -pp mesh
    ./traite.csh
  ***values
    A 30. min 16. max 49.
    B 90. min 46. max 134.
  ***compare i_file_file mesh.post 1 mesh.ref 1 weight 50.
****return
```

The file 'mesh.ref' is supposed to contain the following lines (it represents the 'objective', ie the best mises value which is zero in this case) :

```
0.
0.
```

The shell script named 'traite.csh' slightly transforms post processing result file to make comparison easier :

```
#!/bin/csh
awk '(NR==3)||(NR==6) { if(NR==6) fact=1.e5;
  else fact=1.; printf("%f\n",$2*fact); }' mesh.post > mesh.res
```

This optimization problem leads to the solution ($A = 49., B = 46.$).

How to use a Zlanguage script in optimizer ?

How to use a Zlanguage script in post_processing ?

The main drawback in using *zLanguage* scripts is that *zLanguage* is an interpreted language. It provides high functionality to dynamically detect runtime errors such as : type checking, generic pointers, array bounds errors ... But these checks require time to execute : an interpreted script will never be as fast as the corresponding binary executable.

zLanguage syntax is very close to C++ syntax which is the language used to write ZSet. All *zLanguage* objects are in fact only tiny wrappers to handle the underlying existing C++ classes. It is then pretty easy to write a C++ class, having a script. ZSet incorporates a module which does this work automatically.

Let us suppose the the user has a working local post-processing script which computes the Mises equivalent stress. It is possible to build a C++ file and then a dynamic loadable library which will be automatically loaded by ZSet.

The starting script is :

```
ARRAY<string> input()
{
  ARRAY<string> i;

  i.resize(4); i[0]="sig11"; i[1]="sig22"; i[2]="sig33"; i[3]="sig12";
  return(i);
}

ARRAY<string> output()
{
  ARRAY<string> o;

  o.resize(1); o[0]="ffmises";
  return(o);
}

void initialize()
{
  global double max_mises,min_mises;

  max_mises=-MAX_DOUBLE; min_mises=MAX_DOUBLE;
  cout<<"Initialization : \n";
  cout<<"Max mises ="<<max_mises<<"\n";
  cout<<"Min mises ="<<min_mises<<"\n";
}

void destroy()
{
  global double max_mises,min_mises;
```

```
  cout<<"Max mises ="<<max_mises<<"\n";
  cout<<"Min mises ="<<min_mises<<"\n";
}

void compute()
{
  TENSOR2 sigma;
  global double max_mises,min_mises;
  double mises;

  sigma.reassign(4,in,0);
  sigma.add_sqrt2();
  mises=sigma.mises();
  if(mises<min_mises) min_mises=mises;
  if(mises>max_mises) max_mises=mises;
  out[0]=mises;
}
```

This script requires 7.8 seconds to execute on a medium sized 2D mesh (the real characteristics of the mesh and of the computer doe not matter). This script is stored in a file named `post.z7p`. The following lines go in a file named `convert.inp` :

```
****zpconv
  ***source_file post.z7p
  ***output_file A_post.c
  ***type local_post
  ***class_name APOST
  ***keyword z7plocalpost
****return
```

When the user launchs `Zrun -zpconv convert`, ZSet will analyze the script `post.z7p` and will generate a C++ class in `A_post.c`. This file can then be automatically compiled and inserted into a dynalic library (exactly the same way that a user will generate a dynalic library starting from some ZebFront source files).

The previous three stars commands are :

- `***source_file` : gives the name of the script to be converted
- `***output_file` : the C++ output file
- `***type` : the kind of module to be produced.
- `***class_name` : the C++ class name. User can choose about all names.
- `***keyword` : the keyword associated with the C++ class.

After the dynamic library is built, the user may use the local post processing named `z7plocalpost` as every other post-processing criterions.

NOTE : this converter is in a beta version ! Use it at your own risks... And do not hesitate to contact ZSet developers for more informations.

The same post-processing execution (ie same problem, same computer) using the dynamic library (insted of the interpreted script) requires only 0.79 seconds (speed-up of a factor 10).

How to use a Zlanguage script in post_processing ?

Chapter 5

Z-mat Programming Examples

Introduction

Programming examples:
The following sections show code pieces which are actually from the Z-mat and Z-set source code. We present these programming examples as they are in order to form a base for simple copy-paste starting for user-programming. All class names have been changed with an added `_DEMO` suffix in order that their generated symbols will not be overrriden by the symbols in the standard Z-set libraries. The keyworks are however the same, and these models will override what exists in the standard distribution.

These source files can be added to the `$Z7PATH/User-project` project directory (if they are not already there, use the `material` subdirectory for Z-mat extensions, and the `finite_element` subdirectory for Z-set add-ons) and the "user project" recompiled to make the plug-ins. To update the makefile adding new files, either run `Zsetup` on unix or double click the `proj.zpr` file on windows. In unix then simply type `Zmake`, or on windows open up the `zfem_user.dsw` Visual Studio project file to build.

Example compilation:
The following is a typical output when compiling the user project. The plug-in object files `libZmat_ut_Linux.so` and `libZfem_ut_Linux.so` are copied into the `$Z7PATH/PUBLIC/lib-$Z7MACHINE` directory where they are automatically serarched for when running `Zrun` or `Zmat`.

```
/home/tmpcalcul/M8.2/User-project
foerch.vache[305]% Zsetup
  ========================================
    ZeBuLoN 8.2.1  automatic configuration
    Zsetup      version 8/18/96

    * use the file: library_files to select
      debug files and personal libraries
  ========================================
  ....


Making new o_Linux directory
Making new og_Linux directory
foerch.vache[306]% Zmake
/home/tmpcalcul/M8.2/bin/Zmake: cd: z*: No such file or directory
Doing a 'make -f makefile.mak' in directory z*
make: makefile.mak: No such file or directory
make: *** No rule to make target `makefile.mak'.  Stop.

=======================================================
    ZeBuLoN C++  8.2.1         Mode optimize
    Making executable
-------------------------------------------------------
    Objects in directory  :     o_Linux
    Objects in directory  :     o_Linux
```

```
=====================================================

*** file compilation is listed here ***

g++ -c -Wall -O6 -fPIC -I/usr/Motif/include -I/usr/X11R6/include
-I/usr/local/include -DLinux -D_REENTRANT -D__ZverMaj=8 -D__ZverMin=2
-I/home/tmpcalcul/M8.2/include -Imaterial  -o o_Linux/User_flow_law.o
material/User_flow_law.c
echo o_Linux/Crystal_kinematic_recovery.o  o_Linux/DruckerKinematic.o
o_Linux/Empty.o  o_Linux/Partially_saturated_soil.o o_Linux/Plastic.o
o_Linux/Time_hardening_flow.o  o_Linux/User_flow_law.o    > files; Zlink
-inst  -o libZmat_ut_Linux.so  -ZL Zmat_base Linker compiler : g++
In link_gcc
Linking with dynamic blas
mv libZmat_ut_Linux.so /home/tmpcalcul/M8.2/PUBLIC/lib-Linux
g++ -c -Wall -O6 -fPIC -I/usr/Motif/include -I/usr/X11R6/include
-I/usr/local/include -DLinux -D_REENTRANT -D__ZverMaj=8 -D__ZverMin=2
-I/home/tmpcalcul/M8.2/include -Ifinite_element  -o o_Linux/Empty_calcul.o
finite_element/Empty_calcul.c
echo o_Linux/Empty_calcul.o    > files; Zlink -inst  -o libZfem_ut_Linux.so
-ZL Zmat_base -ZL Zfem_base
Linker compiler : g++
In link_gcc
Linking with dynamic blas
mv libZfem_ut_Linux.so /home/tmpcalcul/M8.2/PUBLIC/lib-Linux
foerch.vache[307]%
```

FLOW example

Description:
This is a model implementing a two term norton flow law.

Code Listing:

```
// --------------------------------------------------------------
//   CLASS DOUBLE_NORTON_FLOW... double power law vdot = <f/K>^n + <f/K2>^n2
// --------------------------------------------------------------
class DOUBLE_NORTON_FLOW : public FLOW {
   double  _os; // old overstress
protected:
   COEFF   K, K2, n, n2;
public:
           DOUBLE_NORTON_FLOW();
   virtual void initialize(ASCII_FILE& file, MATERIAL_PIECE*);
           DOUBLE_NORTON_FLOW(const DOUBLE_NORTON_FLOW& in, MATERIAL_PIECE*);
   virtual MATERIAL_PIECE* copy_self(MATERIAL_PIECE*);

   double  flow_rate(double v, double stress);
   double  dflow_dv();
   double  dflow_dcrit();
   DERIVED;
};

DERIVED_IMPLEMENT1(DOUBLE_NORTON_FLOW,FLOW)
DECLARE_OBJECT(FLOW,DOUBLE_NORTON_FLOW,double_norton)

DOUBLE_NORTON_FLOW::DOUBLE_NORTON_FLOW() {}

void DOUBLE_NORTON_FLOW::initialize(ASCII_FILE& file, MATERIAL_PIECE* mp)
{ FLOW::initialize(file,mp);
  STRING err = "Double Norton flow";
  for(;;) {
    STRING str=file.getSTRING();
    if(!strn_cmp(str,"*",1)) break;
    else if(str=="K" )      K.read(str,file,mp);
    else if(str=="n" )      n.read(str,file,mp);
    else if(str=="K2" )     K2.read(str,file,mp);
    else if(str=="n2" )     n2.read(str,file,mp);
    else INPUT_ERROR("Unknown coefficient:"+str);
  } file.back();
}

DOUBLE_NORTON_FLOW::DOUBLE_NORTON_FLOW(const DOUBLE_NORTON_FLOW& in,
                                       MATERIAL_PIECE* boss) :
   FLOW(in, boss),
   K(in.K, boss), K2(in.K2, boss),
```

```
   n(in.n, boss), n2(in.n2, boss)
{
}

MATERIAL_PIECE* DOUBLE_NORTON_FLOW::copy_self(MATERIAL_PIECE* mp)
{ MATERIAL_PIECE* ret = new DOUBLE_NORTON_FLOW(*this, mp);
  return ret;
}

double DOUBLE_NORTON_FLOW::flow_rate(double, double stress)
{  _os=stress;
   return pow(stress/K(),n) + pow(stress/K2(),n2);
}

double  DOUBLE_NORTON_FLOW::dflow_dcrit()
{  return (n()*pow(_os/K(),n) + n2()*pow(_os/K2(),n2))/_os;
}

double DOUBLE_NORTON_FLOW::dflow_dv() { return 0.0; }
```

ISOTROPIC example

Description:
The following code is for an isotropic hardening model which uses a FUNCTION oobject for describing the model.

Code Listing:

```
class ISO_FUNCTION_DEMO : public ISOTROPIC_HARDENING  {
protected:
   PTR<FUNCTION> funk;
   double pval;
public:
   virtual void initialize(ASCII_FILE& file, MATERIAL_PIECE*);
   virtual double radius(double evcum, VECTOR rvalue) {
             return ISOTROPIC_HARDENING::radius(evcum, rvalue);
   }
   virtual double radius_no_r0(double evcum, VECTOR rvalue) {
             return ISOTROPIC_HARDENING::radius_no_r0(evcum, rvalue);
   }
   double   radius(double evcum, double rvalue);
   double   radius_no_r0(double evcum, double rvalue);
   double   potential_term()const;
   double   dradius_dflow();
};

DECLARE_OBJECT(ISOTROPIC_HARDENING,ISO_FUNCTION_DEMO,function)

void ISO_FUNCTION_DEMO::initialize(ASCII_FILE& file, MATERIAL_PIECE* mp)
{ ISOTROPIC_HARDENING::initialize(file,mp);
  funk = FUNCTION::read(file);
  if (funk.if_null())  INPUT_ERROR("Unable to read function");
  if (!funk->l_var!=1)
     INPUT_ERROR("function can only depend on one parameter : plastic deformation");
}

double ISO_FUNCTION_DEMO::radius(double evcum, double)
{
  return funk->compute(evcum);
}

double ISO_FUNCTION_DEMO::radius_no_r0(double,double) { return 0.; }

double ISO_FUNCTION_DEMO::potential_term()const { return 0.; }

double ISO_FUNCTION_DEMO::dradius_dflow()
{ int err; return funk->compute_derivative(0,err); }
```

HYPERELASTIC_LAW example

Description:
This model is a hyper-elastic component for either the Hyperelastic model or the Hyper-viscoelastic model.

$$\boldsymbol{\sigma} = \mathbf{D}_{el} : \ln(\mathbf{U}) = \mathbf{D}_{el} : \ln(\mathbf{b}^{1/2}) \tag{1}$$

Code Listing:

```
#include <Behavior.h>
#include <Integration_result.h>
#include <Mechanical_behavior.h>
#include <Hyperelastic_behavior.h>
#include <Swap.h>
#include <ZMath.h>
#include <Print.h>
#include <Object_factory.h>
#include <Hyperelastic_behavior.h>
#include <Arruda_boyce_hyper.h>
#include <Tensor_operator.h>
#include <Print.h>

class LOG_HYPER_LAW : public HYPERELASTIC_LAW {
  protected :
      PTR<S_COEFFICIENT_TENSOR4> elasticity;

      TENSOR2       one;
      TENSOR2_VAUX  pk2_0;
      TENSOR4       dlnU_dB;
      TENSOR4       dB_dD;
      TENSOR4       C_star;
      bool          pull_back;
      bool          fd_log;
      bool          modify_for_jaumann;
      bool          modify_for_truesdale;
      bool          symmetrize;

  public :
      LOG_HYPER_LAW();
      virtual ~LOG_HYPER_LAW();

      virtual void initialize(ASCII_FILE&,MATERIAL_PIECE*);
      virtual int base_read(const STRING&,ASCII_FILE&);
      virtual void setup(int& flux_pos, int& grad_pos, int& vi_pos, int& va_pos);
      virtual bool calc_coef();
```

```
      virtual INTEGRATION_RESULT* integrate(const  VECTOR& delta_grad, int flags);

      TENSOR2& give_pk2_0();
      virtual BEHAVIOR::STRESS_MEASURE get_stress_measure() const;
};


DECLARE_OBJECT(HYPERELASTIC_LAW, LOG_HYPER_LAW, logarithmic);
BEHAVIOR_READER(HYPERELASTIC_BEHAVIOR,hyper_elastic_logarithmic);

LOG_HYPER_LAW::LOG_HYPER_LAW()
{ pull_back = FALSE;
  fd_log    = FALSE;
  modify_for_jaumann = FALSE;
  modify_for_truesdale = FALSE;
  symmetrize = FALSE;
}

LOG_HYPER_LAW::~LOG_HYPER_LAW()
{
}

void LOG_HYPER_LAW::initialize(ASCII_FILE& file,MATERIAL_PIECE* mp)
{ HYPERELASTIC_LAW::initialize(file,mp);
  sig.initialize(this,"sig",tsz(),1);
  tg.resize(tsz());
  one.resize(tsz());
  dlnU_dB.resize(tsz());
  dB_dD.resize(utsz());
  C_star.resize(tsz());
  one = TENSOR2::unity(tsz());
}

int LOG_HYPER_LAW::base_read(const STRING& strin,ASCII_FILE& file)
{ STRING str;
  prn("strin:"+strin);
  if (strin=="**pull_back") pull_back = TRUE;
  else if (strin=="**fd_log") fd_log = TRUE;
  else if (strin=="**modify_for_jaumann") modify_for_jaumann = TRUE;
  else if (strin=="**modify_for_truesdale") modify_for_truesdale = TRUE;
  else if (strin=="**dont_symmetrize") symmetrize = FALSE;
  else if (strin=="**symmetrize") symmetrize = TRUE;
  else if (strin=="**elasticity") {
       elasticity = S_COEFFICIENT_TENSOR4::read(file,this);
  }
  else return 0;
  return 1;
}

TENSOR2& LOG_HYPER_LAW::give_pk2_0()
{ return pk2_0;
}
```

```
BEHAVIOR::STRESS_MEASURE LOG_HYPER_LAW::get_stress_measure() const
{ if (pull_back) return BEHAVIOR::PK2_0;
  return BEHAVIOR::CAUCHY;
}


bool LOG_HYPER_LAW::calc_coef()
{ HYPERELASTIC_LAW::calc_coef();
  return TRUE;
}

void LOG_HYPER_LAW::setup(int& flux_pos, int& grad_pos, int& vi_pos, int& va_pos)
{
  if (pull_back) pk2_0.initialize(this,"pk2",tsz(),0);
  else           pk2_0.resize(tsz());

  HYPERELASTIC_LAW::setup(flux_pos,grad_pos,vi_pos,va_pos);
}

INTEGRATION_RESULT* LOG_HYPER_LAW::integrate(const VECTOR& df, int)
{ int i,c;
  set_var_aux_to_var_aux_ini();

  TENSOR2 Ft      = transpose(F);
  TENSOR2 B       = syme(F*Ft);

  //
  // All this is the calculation of ln(U) and its tangent
  //
  SMATRIX dlnU_dB(tsz());
  TENSOR2 lnU   = 0.5*log_tensor(B, dlnU_dB, TRUE);
  dlnU_dB      *= 0.5;
  dB_dD         = 2.0*Mr(to_5_9(B));

  //
  // Ok, now the model is simple
  //
  sig     = (*elasticity)*lnU;
  tg      = (*elasticity)*expand_in(expand_out(dlnU_dB)*dB_dD);

  //
  // Convert a trusdale rate to jaumann rate
  //
  if (modify_for_jaumann) {
      TENSOR2 ss      = to_5_9(sig);
      TENSOR4 sig_one = sig^one;
      C_star = expand_in(Mr(ss) + Ml(ss)) - sig_one;
      tg     = tg + C_star;
  }
  //
  // Convert a trusdale rate to jaumann rate
  //
  else if (modify_for_truesdale) {
      TENSOR2 ss      = to_5_9(sig);
```

```
        TENSOR4 sig_one = sig^one;
        C_star = expand_in(Mr(ss) + Ml(ss)) - sig_one;
        tg     = tg - C_star;
    }

    //
    // And we give the option if a total lagrange model would
    // make the user happy. this is a nice check on the meaning
    // of the tangents.
    //
    if (pull_back) {
        prn("pull back");
        double  J  = F.determin();
        TENSOR2 Fi = inverse(F);
        pk2_0 = J*rotate_tensor(sig,Fi);
        tg    = J*rotate_matrix(tg,Fi);
        sig   = pk2_0;
    }
    if (symmetrize) {
        tg = 0.5*(tg + transpose(tg));
    }

    return NULL;
}
```

Gen-evp example

Description:

BEHAVIOR example

Description:

```
#include <Error_messager.h>
#include <File.h>

#include <Clock.h>
#include <Coefficient.h>
#include <Coefficient_T4.h>
#include <Integration_method.h>
#include <Integration_result.h>
#include <Mat_data.h>
#include <Mechanical_behavior.h>
#include <Verbose.h>

//--------------------------------------------------------------------------
class DAMAGE_ELASTIC : public M_TLE_B_SD {
     SCALAR_VAUX   damage;
     SCALAR_VINT   y_max;
     SMATRIX       tgmat;
     COEFF         Y0, alpha;

  public:
     DAMAGE_ELASTIC();
     virtual void initialize(ASCII_FILE& file,int dim, LOCAL_INTEGRATION*);

     INTEGRATION_RESULT* integrate( MAT_DATA&     mdat,
                                    const VECTOR& delta_grad,
                                    MATRIX*&      tg_matrix,
                                    int           flags);
};
//--------------------------------------------------------------------------

BEHAVIOR_READER(DAMAGE_ELASTIC,damage_elasticity)

DAMAGE_ELASTIC::DAMAGE_ELASTIC() {}

void DAMAGE_ELASTIC::initialize(ASCII_FILE& file,int dim, LOCAL_INTEGRATION*)
{ M_TLE_B_SD::initialize(file, dim,NULL);
  damage.initialize(this,"damage",1);
  y_max.initialize(this,"y_max",1);

   for (;;) {
       STRING str=file.getSTRING();
       if (str.start_with("***") ) break;
       else if (str=="**Y0")     Y0.read(str,file,this);
       else if (str=="**alpha")  alpha.read(str,file,this);
```

```
        else if (!base_read(str,file)) INPUT_ERROR("Unknown coefficient: "+str);
     } file.back();

     tgmat.resize(tsz());
  }

  INTEGRATION_RESULT* DAMAGE_ELASTIC::integrate(MAT_DATA& mdat,
           const VECTOR&, MATRIX*& tg_matrix, int flags_in)
  { int step=0;
    tg_matrix=&tgmat;
    if (!curr_ext_param)
       curr_ext_param = *mdat.param_set();
    attach_all(mdat);
    calc_local_coefs();

    TENSOR2 eel = eto;
    if (thermal_strain.if_not_null() && mdat.param_set())
       eel -= thermal_strain->compute_strain();

    double  y_bar   = 0.5*(eel|(*elasticity*eel));
    if (y_bar>y_max) { y_max=y_bar; step=1; }
    double  ym_root = sqrt(y_max);
    double  yz_root = sqrt(Y0);

    if (ym_root <= yz_root) damage = 0.0;
    else                    damage = alpha()*(ym_root - yz_root);
    if (damage >= 0.99)     damage = 0.99;


    if (flags_in&CALC_TG_MATRIX) {
       tgmat=(1.0-damage)*(*elasticity);
       sig = tgmat*eel;
       if (step) tgmat-=(0.5*alpha/ym_root/(1.0-damage)/(1.0-damage))*(sig^sig);
    } else sig=*elasticity * ((1.0-damage)*eel);

    return NULL;
  }
```

ZebFront example

Description:
This is an example of a ZebFront model using modular components and Runge-kutta integration for the simulation mode of operation.

```
#include <Elasticity.h>
#include <External_parameter.h>
#include <Basic_nl_behavior.h>
#include <Flow.h>
#include <Criterion.h>
#include <Isotropic.h>
#include <Print.h>

@Class MODULAR_PLASTIC_BEHAVIOR : BASIC_NL_BEHAVIOR, BASIC_SIMULATOR {
   @Name     modular_plastic;
   @SubClass ELASTICITY elasticity;
   @SubClass CRITERION  criterion;
   @SubClass FLOW       flow;
   @SubClass ISOTROPIC_HARDENING  isotropic;

   @Coefs    C [0-N] @Factor 1./1.5;
   @Coefs    D [!C];
   @tVarInt  eel, alpha [!C];
   @sVarInt  evcum;
   @tVarAux  X [!C], evi;
   @tVarAux  Xtot;
   @tVarUtil m [!C];
   @tVarUtil Xdot;
};

@PostStep {
  evi  = eto - eel;
  sig  = *elasticity*eel;
  if (integration&LOCAL_INTEGRATION::THETA_ID) {
     SMATRIX tmp(psz,f_grad,0,0);
     if (Dtime>0.0) m_tg_matrix=*elasticity*tmp;
     else           m_tg_matrix=*elasticity;
  } else m_tg_matrix=*elasticity;
}

@Derivative {
  int c;
  ELASTICITY& E=*elasticity;
  sig = E*eel;

  Xtot = 0.0;
  for (c=0;c<!C;c++) Xtot += X[c] = C[c]*alpha[c];
  double    radius = isotropic->radius(evcum);
```

```
  TENSOR2   sigeff = deviator(sig) - Xtot;
  double    f      = criterion->yield(sigeff, radius);

  int yld = (flow->plasticity()) ? (f>=0.0) : (f>0.0);

  if (yld) {
     double miso=0.0;
     m.resize(!C);
     TENSOR2          norm     = criterion->normal();

     Xdot = 0.0;
     for (c=0;c<!C;c++) {
         m[c]  = norm - X[c]*(D[c]/C[c]);
         Xdot += C[c]*m[c];
     }

     if (flow->plasticity()) {
        devcum    = norm|(E*deto);
        @Simulation { ERROR("can't resolve multiplier in the simulation"); }
        if (!if_constant_coefs()) {
             LIST<EXTERNAL_PARAM*>& ep=EXTERNAL_PARAM::Get_EP_list();
             for (int i=0;i<!ep;i++) {
                 double dparam = ((*curr_mat_data->param_set())[i] -
                                  (*curr_mat_data->param_set_ini())[i])/Dtime;
                 devcum -= isotropic->dradius_dparam(ep[i]->name())*dparam;;
                 for (c=0;c<!C;c++) devcum -= (norm|alpha[c])*C[c].d_param(ep[i]->name())*dparam
             }
        }
        double          H1        = norm|(E*norm);
        double          H2        = norm|Xdot;
        double          H3        = isotropic->dradius_dflow();
                        devcum   /= H1 + H2 + H3;
     } else {
        devcum    = flow->flow_rate(evcum,f);
     }
     for (c=0;c<!C;c++) dalpha[c] = devcum*m[c];
     resolve_flux_grad(E, deel, deto, devcum*norm);

  } else resolve_flux_grad(E, deel, deto);

  @Simulation { evi = eto - eel; }
}
```

ZebFront Example 2

Description:

Chapter 6

Z-set Programming Examples

BC example 1

Description:
This boundary condition is used to impose on a NSET the displacement field given by the linear solution at a crack tip. The crack is loaded under mode I or II. This boundary condition is limited to 2D plane problems. It will work on axisymmetric problems , however its physical meaning will be uncertain.

The displacement field is given by:

Mode I:

$$u_1 = \frac{K_I}{2\mu}\sqrt{\frac{r}{2\pi}}\cos\frac{\theta}{2}\left(\kappa - 1 + 2\sin^2\frac{\theta}{2}\right)$$
$$u_2 = \frac{K_I}{2\mu}\sqrt{\frac{r}{2\pi}}\sin\frac{\theta}{2}\left(\kappa + 1 - 2\cos^2\frac{\theta}{2}\right)$$

Mode II:

$$u_1 = \frac{K_{II}}{2\mu}\sqrt{\frac{r}{2\pi}}\sin\frac{\theta}{2}\left(\kappa + 1 + 2\cos^2\frac{\theta}{2}\right)$$
$$u_2 = -\frac{K_{II}}{2\mu}\sqrt{\frac{r}{2\pi}}\cos\frac{\theta}{2}\left(\kappa - 1 - 2\sin^2\frac{\theta}{2}\right)$$

with

$$\kappa = \begin{cases} 3 - 4\nu & planestrain \\ (3-\nu)/(1+\nu) & planestress \end{cases}$$

For elastic problems:

$$J = \begin{cases} (1-\nu^2)\left(K_I^2 + K_{II}^2\right)/E & planestrain \\ \left(K_I^2 + K_{II}^2\right)/E & planestress \end{cases}$$

Code Listing:

```
class BC_K_FIELD : public BC {
     NNSET* set;
     VECTOR center;
     double poisson, young, kappa, two_mu;
     int mode;
  public :
     BC_K_FIELD();
```

```
      virtual void initialize(ASCII_FILE&,MESH&,int);
      virtual ~BC_K_FIELD() {}
      virtual void _update(MESH&);
      RTTI_INFO;
};

IMPL_RTTI_INFO(BC,BC_K_FIELD);

DECLARE_OBJECT(BC,BC_K_FIELD,K_field)

BC_K_FIELD::BC_K_FIELD() {}

void BC_K_FIELD::initialize(ASCII_FILE& file,MESH& mesh,int)
{ set = (NNSET*)mesh.find_nset(file.getSTRING());
  center  = file.getVECTOR(2); if(!file.ok) VEC_REQ("center",file);
  young   = file.getdouble();  if(!file.ok) DBL_REQ("young",file);
  poisson = file.getdouble();  if(!file.ok) DBL_REQ("poisson",file);
  STRING str=file.getSTRING();
  if     (str=="plane_stress") kappa = (3.-poisson)/(1.+poisson);
  else if(str=="plane_strain") kappa = 3.-4.*poisson;
  else INPUT_ERROR("expecting plane_stress or plane_strain");
  two_mu = young/(1.+poisson);
  str=file.getSTRING();
  if     (str=="I") mode = 1;
  else if(str=="II") mode=2;
  else INPUT_ERROR("expecting I or II to specify loading mode");
  base_read(file);
}

void BC_K_FIELD::_update(MESH&)
{ static DOF_TYPE Displ_name[2]={DOF::translate("U1",DOF_NODE_ID),
                                 DOF::translate("U2",DOF_NODE_ID)};
  double Komu=compute_increment()[0]/two_mu, r, thetas2, du1, du2, fact;
  VECTOR axx(2), vr; axx[0]=1.; axx[1]=0.;
  for(int i=0;i<!(*set);i++) {
     vr = (*set)[i]->coord-center;
     r = norm(vr);
     thetas2 = get_angle(axx,vr)/2.;
     fact = Komu*sqrt(r/2./M_PI);
     if(mode==1) {
         du1 = fact*cos(thetas2)*
               (kappa-1.+2*sin(thetas2)*sin(thetas2));
         du2 = fact*sin(thetas2)*
               (kappa+1.-2.*cos(thetas2)*cos(thetas2));
     } else {
         du1 = fact*sin(thetas2)*
               (kappa+1.+2.*cos(thetas2)*cos(thetas2));
         du2 = -fact*cos(thetas2)*
               (kappa-1.-2*sin(thetas2)*sin(thetas2));
     }
     (*set)[i]->add_fixed_dof(Displ_name[0],du1);
     (*set)[i]->add_fixed_dof(Displ_name[1],du2);
  }
```

```
}
```

Pressure boundary condition

Description:

MPC example

Description:
This code imposes that a node set remains straight (`mpc5`).

Code Listing:

```
class MPC5 : public RELATIONSHIP {
    NNSET* nset;
    ARRAY<double> eta;
    ARRAY<NODE*> nodes;
    NODE *n1, *n2;
  public :
    MPC5();
    virtual void initialize(ASCII_FILE&,MESH&);
    void set_relationship(MESH&);
};

// DECLARE_OBJECT(RELATIONSHIP,MPC5,mpc5);

MPC5::MPC5() {}

void MPC5::initialize(ASCII_FILE& file,MESH& mesh)
{ if(!mesh.areYouA("MECHANICAL_MESH"))
     INPUT_ERROR("MPC5 requires a mechanical mesh");
  nset=(NNSET*)mesh.find_nset(file.getSTRING());
  NNSET& ns = *nset;
  int dim = nset->get_dimension();

  if(!ns<2) INPUT_ERROR("The node set size is too small for: "+ns.get_name());
  VECTOR nn(dim), n1n2(dim);

  // compute coefficients
  n1=(NODE*)ns[0]; n2=(NODE*)ns[1];
  make_VECTOR(*n1,*n2,n1n2); double dist = norm(n1n2);
  eta.resize(!ns-2); nodes.resize(!ns-2);
  int count=0;
  for(int in1=2;in1<!ns;in1++) {
     //if(ns[in1]==n1 || ns[in1]==n2) continue;
     make_VECTOR(*n1,*ns[in1],nn); // d1=norm(nn);
     nodes[count]=(NODE*)ns[in1]; eta[count]=(n1n2|nn)/dist/dist;
     count++;
  }
}

void MPC5::set_relationship(MESH&)
{ static DOF_TYPE Displ_name[3]={DOF::translate("U1",DOF_NODE_ID),
                                 DOF::translate("U2",DOF_NODE_ID),
                                 DOF::translate("U3",DOF_NODE_ID)};
```

```
  ARRAY<DOF*> master_dof(2);
  ARRAY<double> coef(2); coef[0]=1.;
  DOF *slave;
  DOF_TYPE type;

  for(int idim=0;idim<nset->get_dimension();idim++) {
      type=Displ_name[idim];
      master_dof[0]=n1->get_dof(type);
      master_dof[1]=n2->get_dof(type);
      for(int in=0;in<!nodes;in++) {
          coef[0]=1.-eta[in]; coef[1]=eta[in];
          slave = nodes[in]->get_dof(type);
          slave->set_mpc(master_dof,coef,0.);
      }
  }
}
```

Element example

Description:
This element is for a rigid link utilizing a Lagrange multiplier for enforcing the condition.

Code Listing:

```
#include <Error_messager.h>

#include <Behavior.h>
#include <External_parameter.h>
#include <P_element.h>
#include <Global_matrix.h>
#include <Mesh.h>
#include <Utility_mesh.h>
#include <Node.h>
#include <Special_vector.h>
#include <GeomSpace.h>

#include <Print.h>

class RIGID_LINK : public D_ELEMENT {
  public :
    int nb_displ;
    virtual int dof_location(int)const;

    RIGID_LINK() { }
    virtual void setup_dofs(const char* key);

    virtual ~RIGID_LINK();
    virtual INTEGRATION_RESULT* internal_reaction(bool,VECTOR&,SMATRIX&,bool only_get_tg_matrix=FALSE);
};

INSTALL_ELEMENT(MECHANICAL_MESH,rigid_link,_2D,ST_LINE,RIGID_LINK)
INSTALL_ELEMENT(MECHANICAL_MESH,rigid_link,_3D,ST_LINE,RIGID_LINK)

int RIGID_LINK::dof_location(int idof)const
{ return (idof<nb_displ) ? DOF_NODE_ID : DOF_ELEMENT_ID;
}

void RIGID_LINK::setup_dofs(const char*)
{
  size_dofs(!node*geometry->space_dimension()+1);

  static DOF_TYPE u_name[3]={DOF::translate("U1",DOF_NODE_ID),
                             DOF::translate("U2",DOF_NODE_ID),
                             DOF::translate("U3",DOF_NODE_ID)};
  static DOF_TYPE RLF    = DOF::translate("RLF",DOF_ELEMENT_ID);
```

```
  int tot=0;
  for(int in=0;in<nb_node();in++) {
       for(int idime=0;idime<space_dimension();idime++) {
            dof_type[tot] = u_name[idime];
            map_dof_to_node[tot] = get_node(in);
            tot++;
       }
  }
  nb_displ = tot;

  dof_type[tot]=RLF;
  map_dof_to_node[tot] = NULL;
}

RIGID_LINK::~RIGID_LINK() {}

INTEGRATION_RESULT* RIGID_LINK::internal_reaction(bool ics,
       VECTOR& resi, SMATRIX& stiff,bool check)
{
  MATRIX elem_coord;
  VECTOR Dof(nb_dof()), dDof(nb_dof());
  int end = nb_dof()-1;
  get_elem_d_dof_tot(Dof);
  double force = Dof[end];

  VECTOR x1  = get_node(0)->get_coordw(END_OF_INCREMENT);
  VECTOR x2  = get_node(1)->get_coordw(END_OF_INCREMENT);

  VECTOR X1  = get_node(0)->get_coordw(BEGINNING_OF_PROBLEM);
  VECTOR X2  = get_node(1)->get_coordw(BEGINNING_OF_PROBLEM);

  VECTOR dX = X2-X1;
  VECTOR dx = x2-x1;
  double R0 = sqrt(dX|dX);
  double R1 = sqrt(dx|dx);
  VECTOR B  = dx*(1.0/R1);

  SMATRIX  dBdX  = B^B;
           dBdX.add_to_diagonal(1.0);
           dBdX *= 1.0/R1;
  SMATRIX mdBdX  = -1.0*dBdX;

  VECTOR subF1, subF2;
  int sd=space_dimension();

  subF1.reassign(sd,resi,0);  subF1= B*(-force);
  subF2.reassign(sd,resi,sd); subF2= B*force;
  resi[end] = R1 - R0;

  if (ics) {
     stiff = 0.0;
     MATRIX subS11(sd,sd,stiff,0,0);      subS11 = mdBdX*(-force);
     MATRIX subS12(sd,sd,stiff,0,sd);     subS12 =  dBdX*(-force);
```

```
    MATRIX subS21(sd,sd,stiff,sd,0);     subS21 = mdBdX*(force);
    MATRIX subS22(sd,sd,stiff,sd,sd);    subS22 =  dBdX*(force);

    for (int c=0;c<sd;c++) {
         stiff(c,   end) = -B[c];
         stiff(sd+c,end) =  B[c];

         stiff(end,c)    =  (x2[c]-x1[c])*(-1.0/R1);
         stiff(end,sd+c) =  (x2[c]-x1[c])*(1.0/R1);
    }

    stiff(end,end) = 0.0;
  }

  return NULL;
}
```

Element example

Problem component example

Description:

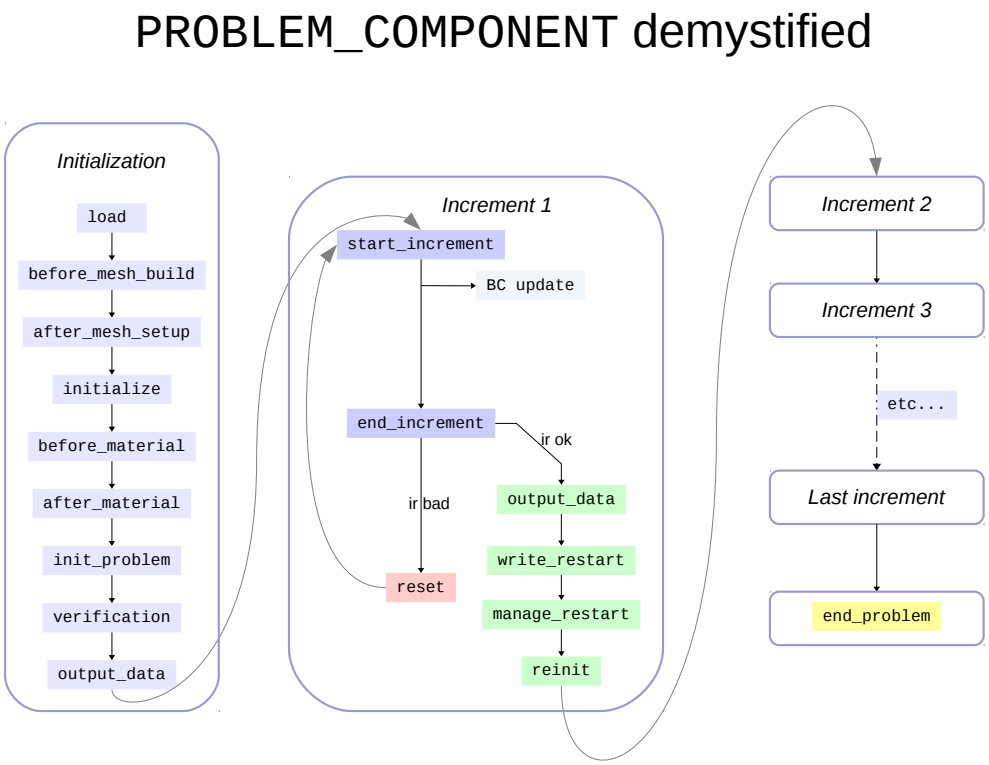


Figure 1: Schematic view of a Problem Component's methods calling sequence. For clarity, some methods were omitted (in particular `start_iteration` and `end_iteration`).

Problem component example

MESHER example

Description:

Code Listing:

```
#include <Calcul_timer.h>
#include <Defines.h>
#include <Error_messager.h>
#include <File.h>
#include <Function.h>
#include <Marray.h>
#include <Rotation.h>
#include <Print.h>
#include <Out_message.h>
#include <Utility_mesh.h>
#include <Modify_record.h>
#include <Transform_geometry.h>

class INVERSE_LISET : public TRANSFORMERS {
  protected :
    LIST<STRING> names;

  public :
     INVERSE_LISET() { }
     virtual ~INVERSE_LISET() { }
     virtual MODIFY_INFO_RECORD* get_modify_info_record();
     virtual void apply(UTILITY_MESH&);
};

DECLARE_OBJECT(TRANSFORMERS,INVERSE_LISET,inverse_liset)

MODIFY_INFO_RECORD* INVERSE_LISET::get_modify_info_record()
{
  MODIFY_INFO_RECORD* ret = new MODIFY_INFO_RECORD;

  ret->ptr = (void*)this;
  ret->info = "inverse_liset";
  ADD_SINGLE_CMD_TO_MODIF_REC(names,names);
  return(ret);
}

void INVERSE_LISET::apply(UTILITY_MESH& mesh)
{
  int iset;
  ARRAY< ARRAY<UTILITY_NODE*>* > n;
  ARRAY<UTILITY_NODE*> no;
```

```
  for(iset=0;iset<!names;iset++) {
    B_UTILITY_SET *set=mesh.get_bound(names[iset]);

    if(!set) ERROR("Can not find liset named "+names[iset]);
    if(set->get_type()!=UTILITY_SET::LISET_CODE) ERROR("boundary "+names[iset]+" is not a liset");
    else {
      UTILITY_LISET &li=*((UTILITY_LISET*)set);
      int iseg,inode;

      n.resize(!li.lnode);
      for(iseg=0;iseg<!n;iseg++) n[iseg]=li.lnode[!n-1-iseg];
      li.lnode.resize(0);
      for(iseg=0;iseg<!n;iseg++) {
        no.resize(!(*n[iseg]));
        for(inode=0;inode<!no;inode++) no[inode]=(*n[iseg])[!no-1-inode];
        for(inode=0;inode<!no;inode++) (*n[iseg])[inode]=no[inode];
        li.lnode.add(n[iseg]);
      }
    }
  }
  n.resize(0);
  no.resize(0);
}
```

POST example

Description:
This post computation generates a field of the tresca values for a given tensor variable.

Code Listing:

```
#include <Utility.h>

#include <Post_eigen.h>
#include <Post_simple.h>

ZCLASS2 POST_TRESCA :public POST_SIMPLE {
  PTR<POST_EIGEN2> lp_eigen;
  void set_eigen2();
 protected :
  int   tsz;

 public :
  POST_TRESCA();
  virtual ~POST_TRESCA();

  virtual MODIFY_INFO_RECORD* get_modify_info_record();
  virtual bool verify_info();

  virtual void input_i_need(int,ARRAY<STRING>&);
  virtual void output_i_give(bool&,ARRAY<STRING>&);
  virtual void compute(const ARRAY<VECTOR>&,ARRAY<VECTOR>&);
};

DECLARE_OBJECT(LOCAL_POST_COMPUTATION,POST_TRESCA,tresca);

POST_TRESCA::POST_TRESCA()
{ class_name="tresca";
}

POST_TRESCA::~POST_TRESCA() { }

MODIFY_INFO_RECORD* POST_TRESCA::get_modify_info_record()
{ MODIFY_INFO_RECORD* ret = POST_SIMPLE::get_modify_info_record();

  if (!lp_eigen()) lp_eigen = (POST_EIGEN2*)Create_object(LOCAL_POST_COMPUTATION, "eigen2");

  MODIFY_INFO_RECORD* ret_eigen = new MODIFY_INFO_RECORD;
  ret_eigen->ptr=(void*)lp_eigen; ret_eigen->info="eigen2";
  ret->commands.add(ret_eigen);

  return ret;
}
```

```
bool POST_TRESCA::verify_info()
{ bool ret = POST_SIMPLE::verify_info();

  lp_eigen->set_post_simple(var_name, scalar, diag);

  return ret;
}

void POST_TRESCA::input_i_need(int dim,ARRAY<STRING>& ret)
{ tsz=TENSOR2::give_symmetric_size(dim);
  lp_eigen->input_i_need(dim, ret);
}

void POST_TRESCA::output_i_give(bool& every_card,ARRAY<STRING>& ret)
{ every_card=TRUE;
  ret.resize(1); ret[0]=var_name+"tresca";
}

void POST_TRESCA::compute(const ARRAY<VECTOR>& in,ARRAY<VECTOR>& out)
{
  ARRAY<VECTOR> eigen_out(!in);
  for( int i=0;i<!in;i++) eigen_out[i].resize(3);
  lp_eigen->compute(in,eigen_out);
  for(int icard=0;icard<!in;icard++)
     out[icard][0]=eigen_out[icard][0]-eigen_out[icard][2];
}
```

Chapter 7

Index

Index