Developer Manual

Z-set

Version 8.2

Transvalor / ENSMP
Centre des Matériaux
B.P. 87 – F-91003 EVRY Cedex
http://www.mat.ensmp.fr

Northwest Numerics, Inc.
1219 Westlake Ave. N. #210
Seattle, WA 98115
http://www.nwnumerics.com

Fall 2000

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 redefined, 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 5.25). 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 therfore 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 it 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 interdependances between header files and can be very useful to speed up the `make`.

***.c files**:

It is recommanded 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 Zebulon 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 is 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, member s 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` definitons 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 discriptive 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 occured.

- `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 stongly 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 be 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.

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

zLanguage

The base language

.1 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.

.2 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 Zebulondirectory ($Z7PATH/lib/Scripts) then in the current working directory.

.3 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 consist 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 dommand 'Zrun -zpt hello.z7p'. The '-zpt' 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 -zpt 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 modify, 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.

.4 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)` Exit 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` Execute 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` Execute 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` Execute 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 execute `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

.5 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 `ZPD` (for Zset Program Debugger). The user who wants to debug has just to launch ZSet with options [`-s ZPD 1`] whatever the name of the command is (examples : `Zrun -s ZPD 1 -pp ...` or `Zmaster -s ZPD 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.

.6 Object

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

- A *methods* 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 *members* 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;` assign 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.

.7 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.
  - `++` : 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.
  - `> < >= <= == !=` : 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.
  - `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 or a TENSOR2 object.
  - `/` : Divide a TENSOR2 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.
  - `[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.
  - `resize(int)` : resize TENSOR2.
  - `print()` : print current value.

- `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 vector 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.

- `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.

.8 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.

.9 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 chi,vy,vx;
  double t0,t1;
  double ti,tf;
  int i;

  chi.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.;
}
```

.10 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 union between different meshes.

  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.

How to use a Zlanguage script in post_processing ?

.1 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 input()
{
  ARRAY<string> i;

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

ARRAY output()
{
  ARRAY<string> o;

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

void main()
{
  TENSOR2 sig;

  sig.resize(4);
  sig.reassign(4,in,0);
  out.resize(1);
  out=sig.mises();
}
```

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 input()
{
  ARRAY<string> i;

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

ARRAY 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;
}

void destroy()
{
  global double max_mises,min_mises;

  endl.print();
```

```
  ("Max mises ="+max_mises+endl).print();
  ("Min mises ="+min_mises+endl).print();
  flush();
}

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

  for(i=0;i<!ape;i=i+1) {
    (" "+i).print(); flush();
    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

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 acess 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 (ie 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 ith 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 informations).
  - `int fake4` (rw) : set this member to 1 for three sides domains.
  - `string element_type` (rw) : element type to be created.

- `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 informations).
  - `double propagation` (rw) : method parameter.
  - `double critical_angle` (rw) : method parameter.
  - `string element_type` (rw) : element type to be created.

- `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 informations) !
  - `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 Zebulon, 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 Zebulondirectories ($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 conventionnal. 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.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();
  flush();
//
  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.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;
  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();
//
  if(direction==1) { vx[0]=ri; vy[0]=0.; }
  else { vx[!vx-1]=ri; vy[!vy-1]=0.; }
  for(i=1;i<!vx;i=i+1) {
    if(direction==1) {
      theta=theta_max/disc*i;
      vx[i]=ri*(cos(theta)+theta*sin(theta));
      vy[i]=ri*(sin(theta)-theta*cos(theta));
    } else if(direction==-1) {
      theta=-theta_max/disc*i;
      vx[!vx-i-1]=ri*(cos(theta)+theta*sin(theta));
      vy[!vy-i-1]=ri*(sin(theta)-theta*cos(theta));
    } 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)
{
//
// Build all points needed to mesh a gear tooth.
//
  VECTOR p;
  MATRIX rot;
  double alpha;
  int i;

//
  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 lines)
{
//
// Build the geometry of a gear tooth using lines
//
  VECTOR r_vx,r_vy;
  VECTOR l_vx,l_vy;
  VECTOR p;
  MATRIX rot;
  int i,iline,jpoint;
  ARRAY 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(PAVING domain, int n_teeth, double awidth, double re,
             double ri, int disc, int dea, int dia)
{
//
// Build the geometry of a complete gear
//
  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(); flush();
    angle=2*pi/n_teeth*itooth;
    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(); flush();
    angle=2*pi/n_teeth*itooth;
    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.element_type="c2d3";
//
  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]);
    domain.add(arcs[itooth]);
    for(i=!(all_lines[itooth])/2;i<!all_lines[itooth];i=i+1)
      domain.add(all_lines[itooth][i]);
    domain.add(other_arcs[itooth]);
  }
  domain.make_connectivity();
  domain.link();
}

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;
  double awidth;
  PAVING domain;

// external and internal radius
  r_ext=1.; r_int=.8;
// number of teeth
  nteeth=15;
// number of edges on involute curves, external and internal arcs
  di=5; d_i_arc=3; d_e_arc=2;
// angular width of a tooth
  awidth=pi/12.;
// center of the gear
  center.x=0.; center.y=0.;
//
  do_gear(domain,nteeth,awidth,r_ext,r_int,di,d_e_arc,d_i_arc);
  master("save_as mesh.mast");
}
```

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
```

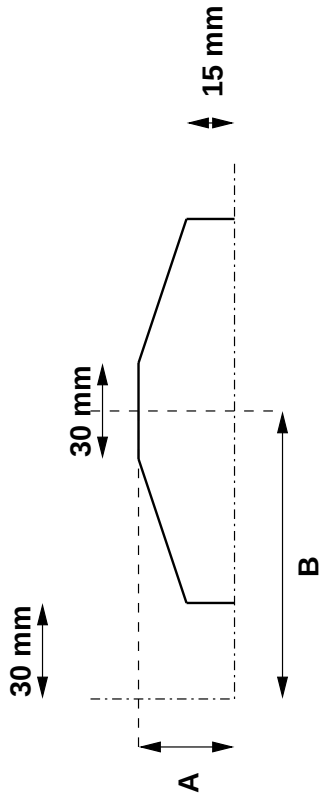
And use the following input file (with 'Zrun -o' command) to find A and B :

```
****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 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).

Chapter 4

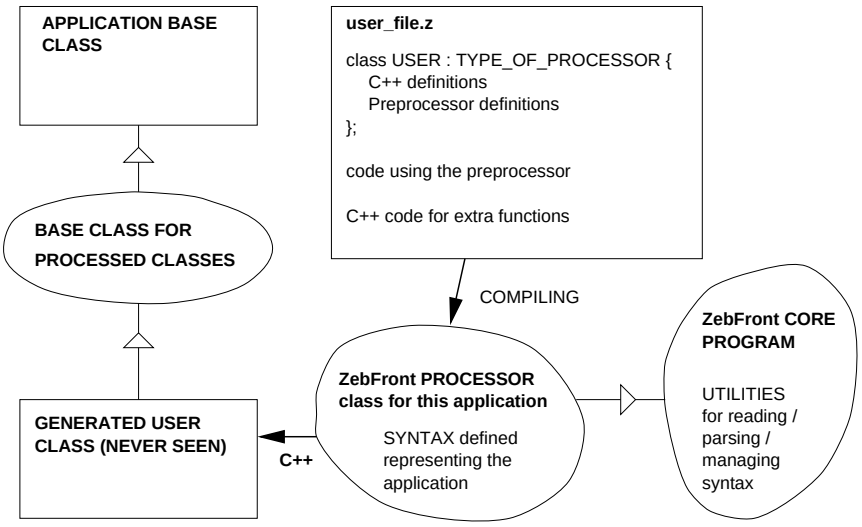
ZebFront pre-processor

ZébFront

Description:
The objective of the `Zebfront` pre-processor 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 pre-processor will be able to provide some symbolic math to reduce the efforts necessary to program implicit integration methods.

Several applications are envisioned for the pre-processor, 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 un-finished; a more complete version will be available shortly.

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



The pre-processor is seen to be composed of a standard Zebulon base class supporting the pre-processor mode, and a corresponding module in the pre-processor 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 4.6).
BASIC_NL_BEHAVIOR	Postulated non-linear FEM material behavior (page 4.8).
BASIC_SIMULATOR	Simulation model derived from a BASIC_NL_BEHAVIOR (page 4.11).

Syntax:
`% ZebFront` *[-o -d -h -od -g] problem* ↩

Bugs

- Debug numbering is not complete as far as the class def is concerned.
- There may be peculiarities in the syntax parsing... such as comma placement, differences in variable declaration order, etc. These are **extremely** important to report.
- The program sets *all* the internal variables in `derivative`. If not all the variables are used, some C++ compilers will generate warning messages for this.

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
};
```

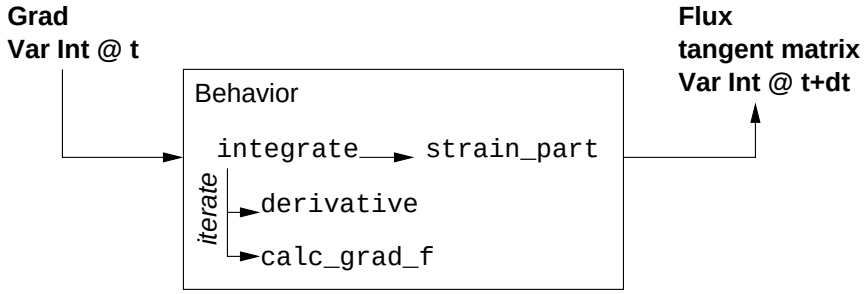
In addition to the limitations of commands, the syntax of those available are reduced as well. The above only permit comma separated lists of names to represent the data members. The coefficients in the model file may also be only single (real) values.

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 Zebulon 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.

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
`@CalcGradF`	implicit integration function for the variable residual and Jacobian matrix
`@UserRead`	extra read function to search user defined syntax

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 4.12
   @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 4.6-4.7). Using the notation **f** for the flux and **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;
};
```

@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 4.2).

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.

@StrainPart

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 @StrainPart` 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.

```
@StrainPart {
  @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;
}
```

@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 `@StrainPart` method.

Example:
This is the derivative function for the first example header on page 4.14.

```
@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 5

Basic Tools

Basic Tools

Description:
The basic tools library provides a number of classes, templates, and functions which are absolutely essential to the programming in Z-set, and which we feel are a big conveniance to general programming as well.

Library:
These functions all all contained in the libraries **libZmat_base.so** and **libzTools.so** on Unix platforms, and in **zmat_base.dll** on win32 platforms.

Classes:

`ARRAY<T>` An array class for general storage of fixed length data lists. The array can be resized, but the user must copy the data back in manually.

`CARRAY<T>` An array with some comparison operators defined. Generally if the `T` class has an `==` operator defined this class can be used.

`LIST<T>` This template is used for data items which need to be added and subtracted. Note that the implementation **is not a linked list**. This is because in computational work the item access operators must be efficient.

`PLIST<T>` A list automatically implementing a `PTR<T>` template on the stored item. Destruction of all items is automatic.

`BUFF_LIST<T>` A list with buffering of the data objects. Speed for adding many objects is thus improved.

`PTR<T>` A template to use instead of just using pointers. Desruction is automatic, and there is a lot of verification if `ZCHECK` is defined.

`STRING` a character string class.

`ASCII_FILE` Class used for parsing all input files.

`GLOBAL_PARAMETER` A class for serving up "parallel safe" and user adjustable global paramters, which are typically default values.

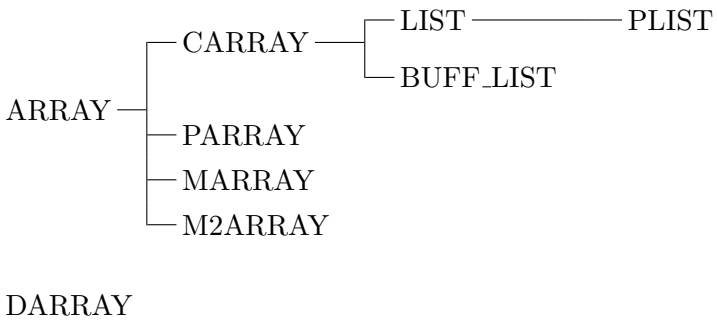
`Object Factory` A set of defines which implements automatic linking of new objects into the object-keyword lookup tables.

`Read/Write Binary` Utility functions for reading and writing Z-set platform independant binary files.

`Debug Prints` the set of `prn` overload functions for outputting debug data when the user launches with `-debug`.

Arrays/Lists

Description:
Array and List objects in Zebulon are made with templates. Lists are not chained lists but rather an expandable version of the linear array. Addition/subtraction of objects therefore costs more than chained lists, but random indexed access is much better. This corresponds to the general needs of FEA programming where setup operations are normally done once, and constitute a very small portion of the overall CPU use.



The base array classes are:

`ARRAY` the base template. It is basically a holder for a `T*` with a fixed size.

`CARRAY` this class can be used for objects which have the `operator==` defined.

`PARRAY` uses the `PTR` class so all objects held in the array are destroyed when the array is deleted or goes out of scope.

`MARRAY` is for math arrays. It is inteded for classes with a `resize` method defined, so the dimension of the array and all the objects in it can be set in one step.

The list templates are:

- `LIST` expands or contracts as objects are `add`ed or `suppress`ed.
- `PLIST` uses the `PTR` template for all objects in the list.
- `BUFF_LIST` Uses a buffer in the list, so additions are done more efficiently. Note this uses more memory than is required.

Files:
The following include files relate to the array/list template library:

```
#include <Array.h>
#include <List.h>
#include <Buffered_list.h>
```

```
#include <Pointer.h>
#include <Marray.h>
#include <Darray.h>
```

ARRAY¡T¿

Description:
As mentioned above, the `ARRAY` is the base class for these templates, and is essentially a managed vector of objects.

```
template <class T> class ARRAY {
  public :
       ARRAY() : x(NULL), sz(0) {}
       ARRAY(int n);
       ARRAY(const ARRAY<T>& a);
       ARRAY(int,T**);
       virtual ~ARRAY();

       const T& operator[](int rank);
             T& operator[](int rank);
       const T& last()const;
             T& last();
       const T& first()const;
             T& first();

       ARRAY<T>& operator=(const ARRAY<T>& a);
       ARRAY<T>& operator=(const T& xxx);
       int       operator!() const;
       void      copy(ARRAY<T>& a)const;
       const T*  ptr()const;

       virtual void resize(int n);
};
```

Class Use:
The use of an array will consist of an object delaration either as a class data member or as a local variable, possibly one or more resize statements, and a variety of statements manipulating the data within the array. An heuristic example could be:
`class A { public : ARRAY<double> data; };` ...

One should note that this class is strict about bounds checking (in debug mode - DZCHECK). This is to say, that performing equivilence of unlike sized arrays is an error, as opposed to doing an automatic resize operation. The choice was made to be consistent with overall program standards, which in tern have been stipulated for performence reasons. Proper use will involve a resize operation such as:

```
void set_eq(ARRAY<USER_TYPE>& a, const ARRAY<USER_TYPE>& b)
{
  /* if (!a != !b) */
  a.resize(!b);
  a = b;
}
```

The commented check for sizing is noted to be **redundant** because a similar check is provided in the `resize` method.

Note : the ARRAY template deletes the storage for the array itself. In the case that the array is of dynamically allocated objects (pointers), the objects themselves will **not** be deleted. Use the PLIST template in this case.

If the array size is zero, setting the array equal to another array will put the array to the same size. Otherwise behavior is undefined (an assertion is hit with `ZCHECK` defined).

Creators and initialization:
The creators for an array object consist of an empty default constructor which initialized the object to a zero size, a constructor taking an interger for the size (storage is allocated to that size), and creation based on another array of like objects. The later is noted to leave the input array unchanged (`const`).

Methods:

`operator!` returns the size of the array.

`operator[]` takes an integer as an index and returns the corresponding element of the array. Please rebember that the arrays count from zero, and there is bounds checking in debug mode.

`first` returns a reference or const reference to the first object in the array.

`last` returns a reference or const reference to the last object in the array; equivilent to `obj[!obj-1];` .

User Methods:

`resize` public method changing the size of the array. Storage is created for the given number of objects, and the size attribute reassigned. If the new size is the same as the old size, nothing is done.

`copy` takes an array of the same type and performs a resize operation to the size of the accessing object, and calls equivilence for all elements in the passed array to the corresponding elements in the current array. i.e. calling `a.copy(b);` is equivilent to `{ b.resize(!a); b = a; }` .

CARRAY¡T¿

Description:
The CARRAY template may be used if the objects to be stored have the `operator==` defined. This operator is used in the new member methods presented here for searching particular objects.

```
template <class T> class CARRAY : public ARRAY<T> {
   public :
     CARRAY();
     CARRAY(int n);
     CARRAY(const ARRAY<T>& a);
     CARRAY(const CARRAY<T>& a);
     CARRAY(int,T**);
     CARRAY<T>& operator=(const CARRAY<T>& n);
     bool is_in(const T&)const;
     bool is_in(const T&,int&)const;
     int  count_occurrence(const T&)const;
     int  first_occurrence(const T&) const;
     bool operator==(const CARRAY<T>& t)const;
};
```

Creators and initialization:
This class is created and destroyed in an analogous fashon to the ARRAY class.

If the array size is zero, setting the array equal to another array will put the array to the same size. Otherwise behavior is undefined (an assertion is hit with `ZCHECK` defined).

User Methods:

`is_in` method used to check if a particular object is contained in an array (`TRUE` if it is, `FALSE` otherwise). The method taking an integer reference will return the index position of the *first* instance of this object. no further searching is performed.

`count_occurrence` counts the number of occurences of a given object within the array. For a `CARRAY<int>` this would return the number of elements which have the given integer value (e.g. 5).

`operator==` operator checking that the values of one array are the same as in another array. This will fail if the sizes are not the same, or if the objects are in differerent orders.

LIST¡T¿

Description:

The list class is used for arrays of objects which may expand or contract dynamically. Local optimizations are performed internally to avoid excesses in the creation or distruction of memory, copying, etc. The class is very useful for both concrete data and object pointers.

```
template<class T> class LIST : public CARRAY<T> {
   public :
       LIST();
       LIST(const LIST<T>&);
       LIST(int n);

       void add(const T& xx);
       void add_to(const ARRAY<T>& xx);
       bool suppress();
       bool suppress_item(const T& xx);
       bool suppress(int rk);
       LIST<T>& operator=(const LIST<T>&);
};
```

Class Use:
As stated above, this class is used when an array of objects is to be intermittently added to or subtracted from. As part of the local optimizations, a buffer size is defined, the default size of which is given by `DEFAULT_LIST_BUFFER_SIZE`. For a given list object this buffer size may be changed in order that a good compromise between memory efficiency and CPU efficiency may be reached for a particular problem.

User Methods:

`add` adds an object to the list at the end.

`add_to` adds the contents of a list to the end and re-sizes accordingly.

`suppress` Removes either the last item in the list, or the item indexed by the integer paramter.

`suppress_item` Removes the *first occurence* of the object given as a parameter. If this item is a pointer, only the pointer is removed, while the object storage remains (lost) in memory.

PLIST¡T¿

Description:
This class provides a list of object pointers which will be destroyed after the list is deleted which would normally be when it goes out of scope.

```
template<class T> class PLIST : public LIST< PTR<T> > {
     public :
       PLIST();
       PLIST(const PLIST<T>& a);

       virtual            ~PLIST();
       virtual void       resize(int n);
               void       add(T* p);
               bool       suppress();
               bool       suppress(int rk);
               CARRAY<T*> make_ptr_array();
};
```

Class Use:
This class is used as was the list object described above, except pointer objects are expeceted and deleted automatically. On could do for instance:

```
  class XX {
         PLIST<STRING>  names;
     public :
         virtual ~XX() { }
         void add_name(const char* nm) { names.add(new STRING(nm)); }
         ...
  };
```

BUFF_LIST¡T¿

Description:
This template implements a list using memory buffering. The holder of data stored in the list is allocated in chunks, so the whole list will not be copied each time an object is added. The list uses `memcopy` for this as well.

This class is very useful for list manipulation within algorithms (temporary management lists), but should normally be avoided in classes themselves if many instances of those classes will be kept.

```
#define DEFAULT_BUFFERED_LIST_SIZE 8

template<class T> class BUFF_LIST : public CARRAY<T> {
     public :
       BUFF_LIST(int bsz=DEFAULT_BUFFERED_LIST_SIZE);
       BUFF_LIST(const ARRAY<T>& a, int bsz=DEFAULT_BUFFERED_LIST_SIZE);
       BUFF_LIST(const BUFF_LIST<T>& a, int bsz=DEFAULT_BUFFERED_LIST_SIZE);

       void add(const T& xx);
       void add_to(const ARRAY< T >& xx);
       void add_at(int rank, const T& xx);

       ZBOOL suppress();
       ZBOOL suppress_item(const T& xx);
       ZBOOL suppress(int rk);
       BUFF_LIST<T>& operator=(const BUFF_LIST<T>&);
       BUFF_LIST<T>& operator=(const LIST<T>&);
       virtual void resize(int);

       operator LIST<T>()const;
};
```

Class Use:
The class can be used as a `LIST` class. Note that it is *not* a `LIST` however but a `CARRAY`. There is a conversion operator a `BUFF_LIST` can be passed as a `LIST`, but there will be creation/copy employed.

Creators and initialization:
The `BUFF_LIST` can be created passing an optional buffer size, using an `ARRAY` or another `BUFF_LIST`, both with an optional buffer size `bsz`. It is tempting to make the buffer size large, but normally there is very little to gain by making it too big. Making it 10 will cut the time spent in the copying/allocations by a factor of ten, which is normally enough to make the CPU for that insignificant in the overall scheme of things. For another order of magnitude difference the buffer would have to be 100, therby greatly increasing the memory use.

`add, add_to, add_at` add members at the end, or at a specific location.

`suppress` remove data at index i, or remove an item. With no parameters it erases the last object. The storage size remains at the largest size.

PTR¡T¿

Description:
This template is used to encapsulate pointers, perform checks on their use with ZCHECK defined, and delete the objects contained within automatically.

```
template <class T> class PTR {
  public :
     PTR();
     PTR(T* p);
     PTR(const PTR<T>& p);

     ~PTR();
     operator T*();
     void          operator=(T* p);
     void          operator=(const PTR<T>& p);

           T*      operator->();
     const T*      operator->()const;

           T*      operator()();
     const T*      operator()() const;

           T&      operator*();
     const T&      operator*()  const;

     bool          if_null() const;
     bool          if_not_null()const;
     void          erase();
     void          dont_delete();
     void          swap_pointer(PTR<T>&);

     bool          operator==(const PTR<T>& p)const;
     bool          operator==(const T *const& p)const;

     friend bool operator==(const T*& t,const PTR<T>& p);
     friend bool operator!=(const T*& t,const PTR<T>& p);
};
```

Class Use:

The class can be used as a regular pointer (and there is no performance penalty for doing so). If the PTR already contains an object, it will issue an error with ZCHECK defined if a new pointer is assigned to it.

an example use:

```
  class XX {
         PTR<MYCLASS>  task;
     public :
         XX() { task=new MYCLASS; }
         virtual ~XX() { }
```

```
};

MYCLASS* tmp  = task();
MYCLASS& tmp2 = *task;

if (task.if_not_null()) task->do_task();
```

Methods:

`operator()` returns the pointer contained within.

`operator*` returns the data pointed to by the pointer.

`if_null` tells if the pointer held is NULL (which it is until something is set in the PTR).

`if_not_null` tells if the pointer held is not NULL.

`erase` delete the pointer and reset it to NULL.

`dont_delete` flags the class to skip the auto-deletion (which can cause a problem if the data gets deleted elsewhere).

`swap_pointer` swaps pointers between `PTR` objects.

ZBOOL

Description:
The boolean type is confusing in C++ because some compilers have defines for it, some have a built in type for it (the standard now) and some have nothing for it. This is pricipally a problem with the rate at which compiler vendors have implemented the cutting edge of C++, and too be portable in todays environments, another solution must be made. For Zebulon versions post summer 1999, there is a class implemented `ZBOOL`.

```
  #include <Zbool.h>

#ifndef TRUE
  #define TRUE 1
#endif
#ifndef FALSE
  #define FALSE 0
#endif

class ZBOOL {
  public :
    ZBOOL();
    ZBOOL(int b);
    ZBOOL(const ZBOOL &b);

    virtual ~ZBOOL();

    operator int()const;
    ZBOOL operator!()const;
    ZBOOL operator&&(ZBOOL b)const;
    ZBOOL operator||(ZBOOL b)const;
    ZBOOL operator&&(int b)const;
    ZBOOL operator||(int b)const;
    ZBOOL operator==(int in)const;
};
```

Class Use:

Use the `ZBOOL` type when it must be clear that a value can only be true or false. This is most important for function calls which return a success/fail status, or a flag paramter is given to select on/off type behavior. Using an integer for these applications is endlessly confusing.

Example:
The following is a function declaration which uses a boolean parameter whether to issue an error with default true. The function returns if it is ok or not.

```
  ZBOOL calculate_stuff(DATA& dat, ZBOOL issue_error=TRUE);
```

STRING

Description:
Utility class to encapsulate character string data as was described by Stroustrup, but with a particular character of the Z7 library. In particuar, the class provides may methods to assist the Z7 text file parsing, and will cooperate with the `ASCII_FILE` class. `STRING` objects are limited to 255 characters (sorry).

```
class STRING {
  public :
      static const STRING EMPTY;

      STRING();
      STRING(const char* buf_in);
      STRING(const STRING& str_in);
      STRING(char c_in):

      STRING&         operator=(const STRING& str_in);
      STRING&         operator=(const char* buf_in);
      STRING&         operator=(char c_in);

      STRING&         operator+=(const STRING& str_in);
      STRING&         operator+=(const char* buf_in);
      STRING&         operator+=(const char c_in);

      int             operator!()const;
      const char*     operator()()const;
      char&           operator[](int i);
      const char&     operator[](int i)const;

      int             operator==(const STRING& str_in)const;
      int             operator!=(const STRING& str_in)const;
      int             operator==(const char* buf_in)const;
      int             operator!=(const char* buf_in)const;

            char&    first();
      const char&    first()const;
            char&    last();
      const char&    last()const;

      STRING&  cut_out(char);
      STRING&  cut_out(const char*);

      char*    locate(char);
      char*    locate(const STRING&);
      char*    locate(const char*);
      STRING&  locate_and_cut(char c_in);
      STRING&  locate_and_cut_after(char c_in);
      STRING&  locate_and_cut_before(char c_in);

      STRING&  clear_space(void);
      STRING&  clear_extra_space(void);
```

```
      void     remove_all(char);

      ZBOOL    if_int()const;
      ZBOOL    if_double()const;

      int      to_int()const ;                  //
      double   to_double()const ;            //

      ZBOOL    start_with(const STRING&)const;
      ZBOOL    end_with  (const STRING&)const;

      friend STRING operator+(const STRING& str1,const  STRING& str2);
      friend STRING operator+(const STRING& str1,const  char* buf_in);
      friend STRING operator+(const char* buf_in,const  STRING& str2);
      friend STRING operator+(const STRING& str1,const  char c_in);
      friend STRING operator+(char c_in,const  STRING& str2);
      friend int strn_cmp(const STRING& str1,const  STRING& str2, int n);
      friend int strn_cmp(const STRING& str1,const  char* buf_in, int n);
      friend int strn_cmp(const char* buf_in,const  STRING& str2, int n);
      friend ostream& operator<<(ostream& stream,const STRING& str);
      friend istream& operator>>(istream& stream,STRING& str);
      friend void write(ostream&,const STRING&);
      friend void read(istream&,STRING&);
      friend void toupper_(STRING& str);
      friend void tolower_(STRING& str);

      STRING getSTRING();
      double getdouble();
      int    getint();
};
```

Class Use:

The STRING class facolitates the handling of string data. Generally the value of a string will be set directly, or as input from an ACII_FILE object. Some simple creations are shown below:

```
STRING tmp = "Hello world";
cout << tmp << " has "<<!tmp<<" characters in it"<<endl;
const char* ptr = tmp();
```

More examples are given below.

Creators and initialization:
Creation of a STRING object may use the default constructor which initilizes the string to be equivilent to `""`, and has a length of zero. Otherwise, the string object may be initialized using another STRING, or a `const char*` pointer. These laster two creations perform a string copy of the character data into an initialized buffer.

Inquiries:

`operator []`

`operator !`

`operators ==, !=` classical implementations to be used for comparison of the entire string with another string object. The entire string must be equivilent for a return value greater than zero.

`operator ()` provides a pointer to the character storage for the string. kept for compatibility... the cast operators eliminate use of this method.

`operator const char*` cast to a const char pointer.

`operator char*` cast to a char pointer.

`if_int, if_double` verifies if the string is compatible with either integer or double format.

`start_with, end_with` Used to check if the string either begins or ends respectively with a given character string.

`strn_cmp` equivilent to the standard c function, but re-defined to use the STRING objects. The return value is the difference between the first n characters of the two given string objects (a value of 0 means no difference).

`locate` returns a pointer to the start of the first instance of a particular character, or string of characters contained in the string object. A return of `NULL` is given when the inquary is not found in the string.

User Methods:

`cut_out, locate_and_cut, locate_and_cut_after, locate_and_cut_before`

`clear_space, clear_extra_space`

`to_int, to_double`

`remove_all`

`operator<<, >>`

`read, write`

`toupper_, tolower_`

Examples:

Here is an example of changing a filename extension in a string:

```
STRING new_name=fzebulon;
if(new_name.locate(".geof")) {
  *(new_name.locate('.'))=0;
  new_name=new_name+".geo";
}
```

Here is an example of parsing data from a string, which would take for example a command like "print lp -d tek contour.ps":

```
bool POSTSCRIPT_DRAWING_AREA::do_command(STRING cmd)
{  STRING orig = cmd;
   STRING str = cmd.getSTRING();
   if (str == "print") {
      STRING lpr = cmd.getSTRING();
      STRING nm  = cmd.getSTRING();
      for (;cmd.ok;) {
         STRING tmp = cmd.getSTRING();
         if (cmd.ok) {
             lpr += " "+nm;
             nm   = tmp;
         }
      }
      STRING sendc = lpr+" "+ofile;
      system(sendc());
      unlink(ofile());
   }
   ...
```

ASCII_FILE

Description:

This class is used to manange input ASCII text files in a generalize fashon, and using the other utilities in the standardized Zebulon library. The class derives from `Zifstream`. This later provides automatic management for parallel implementations, including buffering so seaks and rewinds are not expensive.

```
class ASCII_FILE : public Zifstream {
   public :
      const STRING& name;
      const bool&   ok;

           ASCII_FILE(bool set_default=BOOL_TRUE);
           ASCII_FILE( const char *const name,
                       ZFWHERE w=ZLOCAL,
                       bool set_default=BOOL_TRUE);

           virtual ~ASCII_FILE();

           void  open(const char* const, ZFWHERE w=ZLOCAL);
           void  try_to_open(const char* const, ZFWHERE w=ZLOCAL);
           void  close();

           int   get_next_char();
           void  add_character(int);
           void  remove_character(int);
           bool  is_eol_char(int);

           void  add_comment_char(int);
           void  remove_comment_char(int);
           bool  is_comment_char(int);

           void  add_separator_char(int);
           void  remove_separator_char(int);
           bool  is_separator_char(int);

           void empty_line();

           void locate(const STRING& criterion);
           void locate(const char* criterion);
           void locate_until_word_starts_with(const char*,const char*);
           void locate_next(const STRING& criterion);
           void locate_next(const char* criterion);
           void locate_next_until_word_starts_with(const char*,const char*);
           void locate_begin_line_next(const char*);
           void locate_begin_line(const char*);
           void locate_at_level(int,const STRING&);
           void locate_next_at_level(int,const STRING&);

           void skip_until_word_start_with(const STRING&);
```

```
          char      get_char();
          int       getint();
          STRING    getSTRING();
          double    getdouble();
          double    getnumeral();
          VECTOR    getVECTOR(int vector_size);
          VECTOR    getVECTOR();
          STRING    getline();
          STRING    getSTRING_sl();
          double    getdouble_sl();
          double    getnumeral_sl();
          int       getint_sl();

          void      back(void);  // goto to previous position
          void      rewind(void);
          streampos get_position();
          void      goto_position(streampos);
          void      goto_position_and_set_previous_pos(streampos);
          void      get(LIST<int>&);
          void      get(ARRAY<int>&,int);

          friend istream& operator>>(istream&,STRING&);

          void make_from(STRING data);
          void make_from(const LIST<STRING>& data);
};
```

Class Use:

The use of this class will generally involve a local variable declaration or class data member declaration, after which an open operation is performed, followed by various file manipulations, and finally a close operation.

The following loop is provided as a typical example of reading Zebulon input files:

```
void read_file(const STRING& data_name)
{  ASCII_FILE input;
   input.try_to_open(data_name());
   if (input.ok==FALSE) ERROR("cannot open file: "+data_name);

   for(;;) {
       str=file.getSTRING();
       if (str.start_with("**")) { file.back(); break; }
       else if (str=="*my_command") {
           ...
       }
       else if (str=="*add_val") {
           val = file.getdouble();
           if (!file.ok) DBL_REQ(val, file);
       }
       else INPUT_ERROR("Unknown command: "+str);
   }
}
```

Much more complicated parsing is however possible with the ASCII_FILE class, including changes in seperator, comment character, and so on.

Data members:

`name` visible data member giving a STRING name for the file.

`ok` boolean member indicating the status of the file. The value of `ok` is for the last command access to the file.

Creators and initialization:
The ASCII file class can be created in two different ways. The first is with only one parameter which defaults to `TRUE`. This parameter will inidicate that the standard Z7 definitions for comments, seperators, etc will be implemented.

`open`

`try_to_open`

`close`

Internal adjustment:
These methods manipulate the way the file reads characters, comments, and tokens. By default the comments are `%` and `#`. The seperation characters are (white space) `,` (comma) and `;` (semi-colon).

`add_character, remove_character` add and remove characters from the allowable string characters.

`add_comment_char, remove_comment_char` add and remove what is considered a comment. Comments cause all which remains on a line to be ignored.

`add_separator_char, remove_separator_char` manipulate the seperation characters. These define the boundaries for queries such as `getSTRING()`.

`set_vector_key` Used to set the `VECTOR` delimiters. For instance to change the default values:

```
set_vector_key('[',']');
```

Searching and parsing:
These methods allow you to skip around in a file, and get pieces of data from it. The design is obviouly catored to the needs of parsing Zebulon input data files.

`empty_line` clear and skip past whatever is left on a line.

`skip_until_word_start_with, locate, locate_until_word_starts_with, locate_next, locate_next_until_word_starts_with, locate_begin_line_next, locate_begin_line` various searching methods. Note that global "find first from the file start" operations are discouraged.

`get_char, getSTRING, getdouble, getnumeral` get a value of the specified type. If the type is not found (i.e. a double with no `.`) the file position does not change from before the call, and ok is set to `FALSE`.

`getVECTOR` Get a vector with a given dimension `dim`. A `VECTOR` is an array of `double` delimited by two characters that can be defined by the user using `set_vector_key`. The defaults values are `(` and `)`. For instance

```
( 04 0.5 0.7 ).
```

can be read be using `getVECTOR(3)`

`getSTRING_sl, getdouble_sl, getnumeral_sl, getint_sl` similar except the data must be on the same line as the last read token (use is generally discouraged).

`getline` get the whole line to the next return char, white space included.

`get` Is used to read a list of `int`. If `nb` is not equal to `0`, the function tries to read exactly `nb int`. Otherwise it reads as many `int` as possible.

File Positioning Methods:

`back` go back to position before last read.

`rewind` obvious.

`get_position, goto_position, goto_position_and_set_previous_pos` set position markers and skip around using them.

GLOBAL_PARAMETER

Description:
This class gives a way of safely having global variables in Zebulon (where they are necessary).

```
class GLOBAL_PARAMETER {
  public :
    GLOBAL_PARAMETER(const char*);
    virtual ~GLOBAL_PARAMETER();
    virtual void set_default_value()=0;
    virtual void write()const=0;
    virtual void set_value(const char*)=0;
    void* value();
};

class INT_PARAMETER : public GLOBAL_PARAMETER {
  public :
    INT_PARAMETER(const char*);
    virtual ~INT_PARAMETER();
    virtual void set_default_value();
    virtual void set_value(const char*);
    virtual void write()const;
};
```

Also:

```
DOUBLE_PARAMETER
STRING_PARAMETER
ADJ_INT_PARAMETER      // adjustable int parameter
ADJ_DOUBLE_PARAMETER   // adjustable double parameter
```

Class Use:

an example use:

```
static ADJ_DOUBLE_PARAMETER mesh_fusion("Mesh Fusion",1.e-3);

double crit = GPDOUBLE("Mesh Fusion");

GPDOUBLE("Mesh Fusion") = dialog->get_text_field("Mesh Fusion").to_double();
```

Defines

All compiler dependant `#ifdef`s should be grouped into "classifications" of compiler types, so the instance of machine specifics is localized to one file. For example, some compilers require that template functions be inline, others require that they are not. For this, a define for `INLINE` is given in `Defines.h` and one can select the appropriate value for each compiler there only. Afterward one can use `INLINE` for all templates, and know that the value is correct. Other examples are `INCLUDE_TEMPLATE_C_FILE` and `DETAILED_OPERATORS_REQUIRED`.

The file contains the following at the time of this writing (March 1999).

```
#if ( __GNUC_MINOR__>=6 )
   #define USE_EXPLICIT_TEMPLATES

   #define MK_P_LIST(a) \
      template class PTR< a >; \
      template class PLIST< a >; \
      template class PARRAY< a >; \
      template class LIST< PTR< a > >; \
      template class CARRAY< PTR< a > >; \
      template class ARRAY< PTR< a > >; \
      template class ARRAY< a* >; \
      template class CARRAY< a* >; \
      template class LIST< a* >;

   #define MK_LIST(a) \
      template class LIST< a >; \
      template class CARRAY< a >; \
      template class ARRAY< a >;

#endif

#if defined HPUX
#else
#define INCLUDE_TEMPLATE_C_FILE
#endif

#ifdef __GNUG__
  #define INLINE inline
#elif defined __GNUC__
  #define INLINE
#elif defined HPUX
  #define INLINE
#else
  #define INLINE inline
  #define DETAILED_OPERATORS_REQUIRED
#endif

#if defined ibm || defined _WIN32
#else
#define FCNTL_IN_SYS
```

```
#endif
```

Object Factory

Description:
The object factory is not used as a class (although there is a class there), but rather using several preprocessor `#define`s which "install" and object type and a string name associated with it.

Methods:

`DECLARE_OBJECT(base, object, name)` Add a new class to the object factory creatable classes. No quotes are given.

`Create_object(base, str)` create an object using a tring for the desired "type."

Class Use:

The class can be used as a regular pointer (and there is no performance penalty for doing so). If the `PTR` already contains an object, it will issue an error with `ZCHECK` defined if a new pointer is assigned to it.

an example declaration use for a new class:

```
class MY_BC : public BC {
   public :
       MY_BC();
       virtual ~MY_BC() { }
       void initialize(ASCII_FILE& file);
};

DECLARE_OBJECT(BC, MY_BC, the_name_for_my_bc);
```

The class could be read with (a rather typical format):

```
  if (str=="*bc") {
     str = file.getSTRING();
     BC* tmp = Create_object(BC, str);
     if (tmp==NULL) INPUT_ERROR("a BC type is required, got:"+str);
     tmp->initialize(file);
     its_bc.add(tmp);
  }
```

Read/Write Binary

Description:
Different operating systems have different internal storage for fundimental multi-byte data types (i.e. int, long, float, double). The difference is in the byte order from least to most significant bytes in the storage word, and is also known as big endian and little endian properties of a CPU/OS. Zebulon stores all its binary data files in big endian format, using the routines of this section to handle conversions for different OS types.

```
void read_int(int* i, istream& fd);
void read_long(long* i, istream& fd);
void read_float(float* i, istream& fd);
void read_double(double* i, istream& fd);

void read_nint(int* i, int n, istream& fd);
void read_nlong(long* i, int n, istream& fd);
void read_nfloat(float* i, int n, istream& fd);
void read_ndouble(double* i, int n, istream& fd);

void write_int(const int* i, ostream& fd);
void write_long(const long* i, ostream& fd);
void write_float(const float* i, ostream& fd);
void write_double(const double* i, ostream& fd);

void write_nint(const int* i, int n, ostream& fd);
void write_nlong(const long* i, int n, ostream& fd);
void write_nfloat(const float* i, int n, ostream& fd);
void write_ndouble(const double* i, int n, ostream& fd);

void read_real_long(long* i, istream& fd);
void read_real_nlong(long* i, int n, istream& fd);
void write_real_long(const long* i, ostream& fd);
void write_real_nlong(const long* i, int n, ostream& fd);

void read_real_double(double* i, istream& fd);
void write_real_double(const double* i, ostream& fd);
void read_real_ndouble(double* i, int n, istream& fd);
void write_real_ndouble(const double* i, int n, ostream& fd);
```

Methods:

`read/write_int` read/write int from/to file.

`read/write_long` read/write from file *an int* and put it or get it from a long.

`read/write_float` read/write float from/to file.

`read/write_double` read/write from file *a float* and put or get it to/from a double.

`read/write_long` read/write from file a double.

`read/write_double` read/write from file a double.

Debug Prints

Description:
The `Print.h` file defines many functions which can be called to help in debugging. For Z-mat, this may be the only way to safely debug a user routine within a running ABAQUS problem. The `prn` statments output to a file named `OUT` by default. The prints will only output if the `-debug` switch was given to Zebulon, or a `***debug` command is given in the Z-mat input file.

```
void prn(STRING s, const TENSOR2& tt);
void prn(STRING s, const VECTOR& tt);
void prn(STRING s, const MATRIX& ss);
void prn(STRING s, const DMATRIX& ss);
void prn(STRING s, const ARRAY<float>& ss);
void prn(STRING s, const ARRAY<STRING>& ss);
void prn(STRING s, const double& ss, int r=0);
void prn(STRING s);
void prn(STRING s, int s2);
void prn(STRING s, const SCALAR& ss, int r=0);

void prn2(int, STRING, const TENSOR2&);
void prn2(int, STRING, const VECTOR&);
void prn2(int, STRING, const MATRIX&);
void prn2(int, STRING, const DMATRIX&);
void prn2(int, STRING, const ARRAY<float>&);
void prn2(int, STRING, const ARRAY<STRING>&);
void prn2(int, STRING, const double&, int);
void prn2(int, STRING);
void prn2(int, STRING, int);
void prn2(int, STRING, const SCALAR&, int);
```

Class Use:

an example use:

```
prn("+++++ NL_M_TLE_B_SD::integrate +++");

prn("ret flux",curr_mat_data()->flux());
prn("ret vint",curr_mat_data()->var_int());
prn("ret vaux",curr_mat_data()->var_aux());

prn("select id:"+itoa(select_id)+" reset:"+itoa(reset));
```

Methods:

`prn` put a variable data out with a lavel.

`prn2*` do a prn output if the flag id given as the first parameter is in the value passed after `-debug`.

Globals:

`extern STRING PRN_FLE_NAME` the output file name (file is opened after first prn statement).

`extern int DeBuG` if non-zero `prn` output will be done. `prn2` output will be done if `flag&DeBuG` is non-zero, where `flag` is the 1st parameter passed to the `prn2` statement.

`prn2*` do a prn output if the flag id given as the first parameter is in the value passed after `-debug`.

Visible

Description:
This file has a pre-processor define which makes some data members when a protected but world visible data member is required. The owning class is the only one capable of actually changing the value of the variable, but the world can see and use a constant alias for it.

```
  #define visible(type,var) \
    private : type _ ## var; \
    public : const type& var

#define visible_protect(type,var) \
    protected : type _ ## var; \
    public : const type& var
```

Class Use:
The following example is from the ASCII_FILE implementation:

```
   class ASCII_FILE : public Zifstream {
            int mode_opening;
            visible(STRING,name);
            visible(bool,ok);
            visible(int,level);
            LIST<int> loop_level;
  ...
```

The typical creator would then be (remember to order the creations as they were declared in the class definition):

```
ASCII_FILE::ASCII_FILE(bool _set_default) :
   name(_name),
   _ok(FALSE), ok(_ok),
   _level(-1),level(_level),
   cr_read(FALSE)
{ is_temp_file = 0;
  if(_set_default) set_default();
  set_eol_char();
}
```

Utility functions

Description:
Utility function relating to strings-math obj's etc. this file helps reduce dependancies between file.h and Vector.h and shortens code particularly with tensor names.

```
void tens_name      (ARRAY<STRING>& str, int st, int len, const STRING& prefix);
void tens_name_no_us(ARRAY<STRING>& str, int st, int len, const STRING& prefix);
void vector_name(ARRAY<STRING>& str, int st, int len, const STRING& s);

void output_names(const ARRAY<STRING>& str);
void output_names(const ARRAY<STRING>& str, int& zebaba_index);

void Get_dir_content(const char*,LIST<STRING>&);
void Select_extension(const ARRAY<STRING>&, LIST<STRING>&, const char*);

ZBOOL split(const STRING&,char,STRING&,STRING&);
```

`tens_name, tens_name_no_us` get a list of tensor names given a prefix, tensor size `len` and starting position in the string array `str`. `str` must be pre-sized to greater than `st+len`. Shear components have and underscore in the name for symmetric tensors in calls to `tens_name`, but not for `tens_name_no_us`.

`vector_name` similar to `tens_name` but for vectors.

Environment

Description:

Z_getenv.h:

```
char *zebu_getenv(const char *name)
```

`zebu_getenv` Used in the place of `getenv` for non-unix platforms.

Chapter 6

Mathematical Tools

Math Tools

Description:
These classes and functions provide the basis for a mathematical library with the specific interest of continuum mechanics.

Library:
These functions all all contained in the libraries **libZmat_base.so** and **libzTools.so** on Unix platforms, and in **zmat_base.dll** on win32 platforms.

Classes:

`FUNCTION` The function class is a means for the user to enter in interpreted functions of any number of variables.

`MATRIX` The matrix class is a 2 dimensional matrix of `double` values. The implementation is intended for relatively small sizes.

`SMATRIX` **Square matrix** (not symmetric). This class includes extra functions, and can also act as a 4th order tensor. Z8.2 and later versions prefer use of the `TENSOR4` class.

`DMATRIX` Diagonal matrix. Any off diagonal terms are ignored.

`TENSOR2` 2nd order tensor class.

`VECTOR` Vector class for general array of `double` uses and also acts as a first order tensor. Z8.2 and later versions prefer use of the `TENSOR1` class.

FUNCTION

Description:
The FUNCTION class is used for interpreted functions. The function can either be created from an ASCII_FILE or with a STRING.

```
class VARIABLE {
  public :
    STRING name;
    double value;

    VARIABLE();
   ~VARIABLE();

    VARIABLE& operator=(const VARIABLE &v);
    int       operator==(const VARIABLE &v);
};

class FUNCTION {
  public :
   enum FUNC_ERROR { OK=0, NC_PAR=1, NO_PAR=2 ,
                     SYN=3, B_NAG=4, B_FSY=5, TFA=6,
                     B_ASS=7,E_EXI=8};

   int            narg;
   LIST<VARIABLE> l_var;
   STRING         algebraic;

                    FUNCTION();
                    FUNCTION(const FUNCTION&);

   static FUNCTION* read(ASCII_FILE&);
   static FUNCTION* make(const STRING&);

   virtual ~FUNCTION();

   ZBOOL   operator==(const FUNCTION& fin);

   int     analyze(const STRING& al,ZBOOL _issue_error=FALSE) ;
   double  compute(int &err,ZBOOL _issue_error=FALSE);
   double  compute();
   double  compute(ARRAY<double>&);
   double  compute(VECTOR&);
   double  compute(double);
   double  compute_derivative(int,int &err,ZBOOL _issue_error=FALSE);

   double& get_var(STRING&);
   void    check_and_get_vars(ARRAY<STRING>&,ARRAY<double*>&);

   void    fOut(void);

};
```

Class Use:

instances of `FUNCTION` can be created easily using the `read` or `make` methods.

Example:
The following example is taken from the nset transform class for the batch mesher. Here a list of functions can be entered, and the functions are expected to be defined using variables named `x` `y` or `z`. When the functions are evaluated therefore these values must be set in the function, and then the function evaluated (see second code snipit).

```
//
// Read a bunch of functions & put them in a list.
//
for (;;) {
     STRING tmp = file.getSTRING();
     file.back();
     if (tmp[0]=='*') break;
     STRING f=file.getline(); file.back();
     flist.add(FUNCTION::read(file));
     cmd_lines.add(f);
}
...

double NSET_TRANSFORM::eval_func(const VECTOR& v, FUNCTION& f)
{ int vsz = !v;
  for (int j=0;j<!(f.l_var);j++) {
      VARIABLE &va=f.l_var[j];
      double val=0.;

      if (va.name=="x") val=v[0];
      if (va.name=="y") if (vsz<2) val=0.; else val=v[1];
      if (va.name=="z") if (vsz<3) val=0.; else val=v[2];
      va.value=val;
  }
  double ret = f.compute();
  return ret;
}
```

Data members:
The following data members can be used from a function.

`narg` number of arguments which should be passed to the function.

`l_var` storage of the variables which are used in the function.

`algebraic` copy of the algebreic expression of the function.

Methods:

`read, make` create and initialize a function object using the given `ASCII_FILE` or `STRING` object. The `analyze` method is called for both with `issue_error` set to `TRUE`.

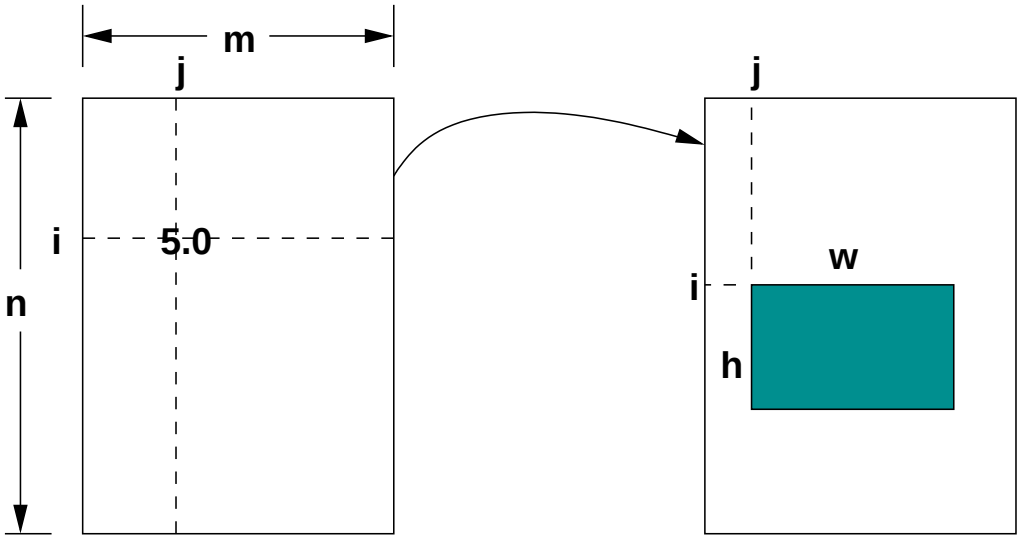
`analyze` parse and otherwise digest a given string as the basis of the function definition.

`compute` calculate the function.

MATRIX

Description:
This class is a standard mathematical utility for matrix data. It is meant for relatively small matrix sizes (i.e. not a finite element stiffness matrix).

An important aspect of this class is its ability to support sub-objects attached to a location within the matrix. The figure below demonstrates the indexing and creation of matrix objects.



MATRIX mat(12,6);

// mat.n==12 mat.m==6

mat(i,j) = 5.0;

MATRIX sub(h,w,mat,i,j);

Caution: the matrix class uses the `!` operator to get the total size. It also uses indexing with parenthesis such as `A(10,2)` instead of using the `[]` operators as is used elsewhere.

```
class MATRIX : public MATH_OBJECT {
   public:
      enum    LC       { _LINE_, _COLUMN_, _DIAGONAL_ };
      const   int&     n;  // number of lines
      const   int&     m;  // number of columns

      MATRIX();
      MATRIX(int width,int height);
      MATRIX(int width,int height,const double* data_storage);
      MATRIX(const MATRIX&);
      MATRIX(const DMATRIX& dm);
      MATRIX(int width,int height,const MATRIX& master,int start_n,int start_m);
      ~MATRIX();

      void resize(int,int);
```

```
      void reassign(int n, int m,const MATRIX&, int start_n, int start_m);
      void set(const MATRIX&); // resize and copy

              double&  operator()(int i, int j);
      const   double&  operator()(int i, int j) const;

      int      operator!() const { return m*n; }

      int      operator==(const MATRIX& m);

      MATRIX&  operator=(const MATRIX&); //    copy the data
      MATRIX&  operator=(double c);
      MATRIX&  operator=(const DMATRIX&);
      MATRIX&  operator += (const MATRIX&);
      MATRIX&  operator -= (const MATRIX&);
      MATRIX&  operator *= (double);
      MATRIX&  operator /= (double);

      MATRIX& add_all_element(double);
      MATRIX& add_to_diagonal(double);

      MATRIX& operator=(const TENSOR2&);
      MATRIX& operator=(const VECTOR&);
      MATRIX& operator+=(const TENSOR2&);
      MATRIX& operator+=(const VECTOR&);
      MATRIX& operator-=(const TENSOR2&);
      MATRIX& operator-=(const VECTOR&);


      friend  ostream& operator<<(ostream&,const MATRIX&);
      friend  istream& operator>>(istream&,MATRIX&);

      friend  MATRIX  operator*(const MATRIX&,const MATRIX&);
      friend  MATRIX  operator*(const MATRIX&,double);         // m*d
      friend  MATRIX  operator*(double,const MATRIX&);         // d*m

      friend  MATRIX  operator/(const MATRIX&,double);         // m/d
      friend  MATRIX  operator^(const VECTOR&,const VECTOR&);  // Mij=v1i*v2j

      friend  MATRIX  operator+(const MATRIX&,const MATRIX&);
      friend  MATRIX  operator-(const MATRIX&,const MATRIX&);

      friend  MATRIX  transpose(const MATRIX& mat);
      friend  SMATRIX  expand_out(const SMATRIX& in);
      friend  SMATRIX  expand_in (const SMATRIX& in);

      friend void _read_(MATRIX&,Zfstream&);
      friend void _write_(const MATRIX&,Zfstream&);
};
```

Class use:
The matrix object is a concrete type for storing and performing operations on matrix data. See below for some examples.

Sizing, access, and setup:

`resize` Resize the matrix. All exisiting data is lost. If the matrix was a sub- of something else, this association is lost.

`reassign` re-sets the matrix to be a sub-entity. It is possible to have a "stand-alone" matrix in use, and re-assign it to be a sub of something else.

`set` resizes and sets the values of another matrix to the current one. example: `a.set(b);`

`operator()(int i, int j)` it is the index operator. Access an element with this (e.g. `a(3,4) = 5.0;`)

`operator!()` returns the total storage size mn.

Manipulation:

`add_all_element` Adds a value to all elements in a matrix.

`add_to_diagonal` Adds a value to the diagonal of a matrix. There is an assertion that the matrix be square.

`transpose` transpose a matrix, with resize and transfer of data.

Other methods:

`expand_out` make a 5×5 (resp. 9×9) matrix out of a 4×4 (resp. 6×6) matrix so that it is consistent with the definition of `TENSOR2`. For instance if you have a 6×6 matrix and want to multiply it by a 3D non–symmetric tensor, `operator(const SMATRIX\&,const TENSOR2\&)` will give a size mismatch; in order to solve the problem create first a new matrix using `expand\_out` and then multiply titemexpand_in make a 4×4 (resp. 6×6) out of a 5×5 (resp. 9×9) matrix; similafr use as `expand\_out`

`_read_, _write_`

Hacker notes:

The storage for the matrix class uses a double dimensioned analogue to that used for the vector and tensor classes.

SMATRIX

Description:
This class is an implementation of a **square** matrix. The class also acts as a 4th order tensor in versions before Z8.2. Many of the creators are duplicated from MATRIX to avoid ambiguity (the compiler requires this), and to add assertions regarding its square property.

```
class SMATRIX : public MATRIX {
   public :
      SMATRIX();
      SMATRIX(int ni);
      SMATRIX(int ni,const double* x);
      SMATRIX(const SMATRIX& mi);
      SMATRIX(const MATRIX&  mi);
      SMATRIX(int n,const MATRIX&, int start_n, int start_m);

      void resize(int n);
      void reassign(int n,MATRIX&, int start_n, int start_m);

      int      operator==(const SMATRIX& m);
      int      operator==(const MATRIX& m);

      SMATRIX& operator=(const MATRIX& mi);
      SMATRIX& operator=(const SMATRIX& mi);
      SMATRIX& operator=(double c);
      SMATRIX& transpose();
      SMATRIX& add_diagonal(double d);
      double determin() const;

      int gauss_solver(const VECTOR& b, VECTOR& u,bool error=FALSE)const;
      int gauss_solver2(const VECTOR& b,const VECTOR& u,VECTOR&,VECTOR&,bool error=FALSE)const;

      SMATRIX& inverse(double&);
      SMATRIX& inverse();
      SMATRIX& inverse(double&,bool&);
      SMATRIX& inverse(bool&);

      SMATRIX&  operator += (const SMATRIX& mi);
      SMATRIX&  operator -= (const SMATRIX& mi);
      SMATRIX&  operator *= (double d);
      SMATRIX&  operator /= (double d);

      static const SMATRIX& unity(int size);

      friend SMATRIX& tXAX( const MATRIX& X, const SMATRIX& A, SMATRIX& B);
      friend SMATRIX& tXAXs(const MATRIX& X, const SMATRIX& A, SMATRIX& B);

      friend SMATRIX  operator^(const TENSOR2& m1,const  TENSOR2& m2);
      friend void     symmetrize(SMATRIX& mat);

      static void      eigen(const SMATRIX& in, VECTOR& vals, SMATRIX& vecs);
};
```

Manipulation:

`determin` Returns the determinant of the matrix. The ,matrix is unchanged.

`gauss_solver,gauss_solver2` Solve $Ku = b$ returning 1 if successful, 0 otherwise. The `bool` indicates if an error message (e.g. null pivot) should be issued within the function. Normally it is better to handle that in the calling routine.

`inverse`

`SMATRIX& SMATRIX::inverse(double&,bool&)`	Member, returns `this`
`SMATRIX& SMATRIX::inverse(bool&)`	Member, returns `this`
`SMATRIX inverse(const SMATRIX&,bool&)`	friend
`SMATRIX inverse(const SMATRIX&,double&,bool&)`	friend
`SMATRIX& SMATRIX::inverse(double&)`	Member, returns `this`
`SMATRIX& SMATRIX::inverse()`	Member, returns `this`
`SMATRIX inverse(const SMATRIX&)`	friend
`SMATRIX inverse(const SMATRIX&,double&`	friend

The `double&` is the computed determinant. The `bool&` argument has two purposes: (1) as entry it indicates if the program should be stopped in cases of failure (`TRUE`), (2) as output it indicates is the inversion has succeded (`TRUE`)

`unity` static function returning a reference to a SMATRIX of a given size with one on the diagonal, and zero elsewhere. This is for very lazy programmers; don't use it.

`tXAX,tXAXs` Optimized multiplications for a $\mathbf{X}^T\mathbf{A}\mathbf{X}$ multiplication. The value is returned in the 3rd parameter `B`. The `tXAXs` case is further optimized for symmetric matrices.

`operator^` outside product of two tensors. This case is acting like a 4th order tensor. *Depreciated after version 8.2.*

DMATRIX

Description:
This class is for situations where a `MATRIX` object is called for, but the specific use only has values on the diagonal.

```
class DMATRIX : public SMATRIX {
   public :
      DMATRIX();
      DMATRIX(int);
      DMATRIX(const DMATRIX&);
      DMATRIX(const TENSOR2&);
      DMATRIX(int n,const MATRIX&, int start_n, int start_m);

      void resize(int);
      DMATRIX& inverse();
      DMATRIX& inverse(double&);
      DMATRIX& inverse(bool&);
      DMATRIX& inverse(double&,bool&);
      double& operator[](int i);
      const double& operator[](int i)const;
      DMATRIX& operator=(const TENSOR2&);
      DMATRIX& operator=(double);
      DMATRIX& operator=(const DMATRIX&);
      friend MATRIX  operator*(const MATRIX& m1,const DMATRIX& m2);
      friend MATRIX  operator*(const DMATRIX& m1,const MATRIX& m2);
      friend DMATRIX operator*(const DMATRIX&,const DMATRIX&);
      friend DMATRIX operator*(const DMATRIX&,double);
      friend DMATRIX operator*(double,const DMATRIX&);
      friend VECTOR  operator*(const DMATRIX&,const VECTOR& );
      friend VECTOR  operator*(const VECTOR&, const DMATRIX&);
      friend SMATRIX operator*(const DMATRIX&,const SMATRIX&);
      friend SMATRIX operator*(const SMATRIX&,const DMATRIX&);
      friend DMATRIX inverse(const DMATRIX&);
      friend DMATRIX inverse(const DMATRIX&,double&);
      friend DMATRIX inverse(const DMATRIX&,bool&);
      friend DMATRIX inverse(const DMATRIX&,double&,bool&);
};
```

TENSOR2

Description:
The class TENSOR2 is used for manipulating second order tensor data. The class includes many methods and verifications specific to tensorial operations. This class should be used without exception for data which is known to be tensorial. If the general statement of a particular data entity is not known to be tensorial, but a special case uses it as such, one can place a tensor object as a sub-tensor on either a MATRIX or VECTOR

Conventions on second order tensors are as follows:

— 1D tensors (size 3)

$$\begin{pmatrix} t_{11} \\ t_{22} \\ t_{33} \end{pmatrix}$$

— 2D tensors (size 4 or 5)

$$\text{symmetric} \begin{pmatrix} t_{11} \\ t_{22} \\ t_{33} \\ \sqrt{2}t_{12} \end{pmatrix} \quad \text{non–symmetric} \begin{pmatrix} t_{11} \\ t_{22} \\ t_{33} \\ t_{12} \\ t21 \end{pmatrix}$$

— 3D tensors (size 6 or 9)

$$\text{symmetric} \begin{pmatrix} t_{11} \\ t_{22} \\ t_{33} \\ \sqrt{2}t_{12} \\ \sqrt{2}t_{23} \\ \sqrt{2}t_{31} \end{pmatrix} \quad \text{non–symmetric} \begin{pmatrix} t_{11} \\ t_{22} \\ t_{33} \\ t_{12} \\ t_{21} \\ t_{23} \\ t_{32} \\ t_{31} \\ t_{13} \end{pmatrix}$$

```
class TENSOR2 : public MATH_OBJECT {
  public :
    TENSOR2();
    TENSOR2(int n);
    TENSOR2(const TENSOR2&);
    TENSOR2(const VECTOR&);
    TENSOR2(const SMATRIX&,int);
    TENSOR2(const MATRIX&,int);
    TENSOR2(int sz,const VECTOR&,int start);
    virtual ~TENSOR2();

    virtual void resize(int n);
    virtual void reassign(int sz,const VECTOR&,int start);
    virtual void reassign(int sz,double*,int start);

    int operator!(void) const { return size; }

    double& operator[](int i);
    const double& operator[](int i)const;
```

```
TENSOR2& operator= (const TENSOR2&);
TENSOR2& operator= (double);
TENSOR2& operator= (const SMATRIX&);
TENSOR2& operator= (const MATRIX&);
TENSOR2& operator= (const VECTOR&);
int      operator==(const TENSOR2& t);

TENSOR2& operator+=(const TENSOR2&);
TENSOR2& operator-=(const TENSOR2&);
TENSOR2& operator*=(const TENSOR2&);
TENSOR2& operator*=(double);
TENSOR2& operator/=(double);
double   operator| (const TENSOR2& t)const;
TENSOR2& transpose(void);
double   determin(void) const;
TENSOR2& inverse();

double   trace(void)const;
TENSOR2& deviator(void);
double   mises(void) const;

TENSOR2 FtF(void) const;
TENSOR2 FFt(void) const;
int      size_sym(void) const;
int      size_nonsym(void) const;

friend WIN_THINGIE TENSOR2 operator+(const TENSOR2&, const  TENSOR2&);
friend WIN_THINGIE TENSOR2 operator-(const TENSOR2&, const  TENSOR2&);
friend WIN_THINGIE TENSOR2 operator*(const TENSOR2&, const  TENSOR2&);

friend WIN_THINGIE TENSOR2 operator*(const TENSOR2&,double);
friend WIN_THINGIE TENSOR2 operator*(double,const TENSOR2&);
friend WIN_THINGIE TENSOR2 operator/(const TENSOR2&,double);

friend WIN_THINGIE TENSOR2 operator*(const SMATRIX&, const TENSOR2&);
friend WIN_THINGIE TENSOR2 operator*(const TENSOR2&, const SMATRIX&);

friend WIN_THINGIE TENSOR2 operator*(const DMATRIX&, const TENSOR2&);
friend WIN_THINGIE TENSOR2 operator*(const TENSOR2&, const DMATRIX&);

friend WIN_THINGIE TENSOR2 syme(const TENSOR2&);
friend WIN_THINGIE TENSOR2 antisyme(const TENSOR2&);
friend WIN_THINGIE TENSOR2 to_5_9(const TENSOR2&);
friend WIN_THINGIE TENSOR2 rotate_tensor(const TENSOR2&,const TENSOR2&);
friend WIN_THINGIE double norm(const TENSOR2& t);
friend WIN_THINGIE double trace(const TENSOR2& t);
friend WIN_THINGIE double triaxiality(const TENSOR2&);
friend WIN_THINGIE int t2code(const TENSOR2& t1,const  TENSOR2& t2);

friend WIN_THINGIE ostream& operator<<(ostream&,const TENSOR2&);

static TENSOR2  transpose(const TENSOR2&);
```

```
    static TENSOR2  inverse(const TENSOR2&);
    static TENSOR2  deviator(const TENSOR2&);

    static const TENSOR2& unity(int n);
    static const TENSOR2& one(int n);
    static const SMATRIX& Jmat(int);
    static const SMATRIX& Kmat(int);
    static int give_symmetric_size(int dim);
    static int give_nonsymmetric_size(int dim);
    static int convert_to_symmetric_size(int tensor_size);
    static int convert_to_nonsymmetric_size(int tensor_size);
    static bool allowed_size(int n);

    double    I1()const;
    double    I2()const;
    double    I3()const;

    double    J2()const;
    double    J2(TENSOR2& dJ2_dt)const;
    double    J3()const;
    double    J3(TENSOR2& dJ3_dt)const;

    double    ijjkki()const;
    double    ijjkki(TENSOR2& dsIII_dt)const;

    void      strain_partion(TENSOR2& r, TENSOR2& u)const;
    void      strain_partion_left(TENSOR2& v, TENSOR2& r)const;
    TENSOR2   eigen_vecs(TENSOR2& evals)const;
    TENSOR2   log_tensor()const;
    TENSOR2   exp_tensor()const;
    TENSOR2   pow_tensor(double d)const;

    void  eigen(VECTOR& vals, TENSOR2& vecs)const;
    SMATRIX mkmat()const;
    friend WIN_THINGIE void cross(TENSOR2& tout, const VECTOR& v1, const VECTOR& v2);
};

inline TENSOR2  transpose(const TENSOR2& t) { return TENSOR2::transpose(t); }
inline TENSOR2  inverse(const TENSOR2& t)   { return TENSOR2::inverse(t); }
inline TENSOR2  deviator(const TENSOR2& t)  { return TENSOR2::deviator(t); }

TENSOR2   d_determ(const TENSOR2& t);
VECTOR    operator*(const TENSOR2&,const VECTOR&);
TENSOR2   dp1_dsig(const TENSOR2& princip, const TENSOR2& sig);
```

Class Use:

Creators and initialization:

Operators:

`operator|` contracted product $d = a_{ij}b_{ij}$ with summation on indicies.

`operator` outside product $M_{ijkl} = a_{ij}b_{kl}$

`operator*` product $t_{ik} = a_{ij}b_{jk}$ returns a *non-symmetric* result.

`operator!` number of independant components of the tensor (i.e. the size of stored variables).

`operator[]` index operator. Accesses the individual components in order as described above, with index starting with zero.

User Methods:

`inverse` returns $\mathbf{t}^{-1}$ with the tensor remaining constant.

`trace` returns a double value for $t_{11} + t_{22} + t_{33}$

`deviator` return

Hacker Notes:

The internal optimized storage for this class uses a fixed size of 9 for the allocation proceedure.

VECTOR

Description:
This class is used for "vectors" of doubles. This is not meant to be a vector as defined in mechanics, which is instead defined in the class `TENSOR1` (vector is a 1st order tensor).

```
class VECTOR : public MATH_OBJECT {
   public:
      VECTOR() : size(0), v(NULL) { }
      VECTOR(int n,int ist=0);
      VECTOR(const VECTOR& vv,int ist=0);
      VECTOR(int n ,double x,int ist=0);
      VECTOR(int sz, const VECTOR&,int start);
      VECTOR(const ARRAY<double>& a,int ist=0);
      VECTOR(const TENSOR2&,int ist=0);
      virtual ~VECTOR();

      virtual void resize(int n);
      virtual void resize(int n,int ist);
      virtual void reassign(int length, const VECTOR&, int start_pos);

      int operator!()const { return size; }

      double&        operator[](int i);
      const double&  operator[](int i)const;

      double&        first();
      const double&  first()const;
      double&        last();
      const double&  last()const;

      double*        ptr();
      const double*  ptr();

      void sort(int, int);
      void sort();

      VECTOR&  operator=(const VECTOR&);
      void     set(const VECTOR&);
      VECTOR&  operator=(double c);
      VECTOR&  operator=(const TENSOR2& t);
      VECTOR&  operator=(const ARRAY<double>& a);

      VECTOR&  operator+=(const VECTOR&);
      VECTOR&  operator-=(const VECTOR&);
      VECTOR&  operator*=(double);
      VECTOR&  operator/=(double);

      int operator==(const VECTOR& vec_in)const;
      int operator!=(const VECTOR& vec_in)const;

      friend  double operator|(const VECTOR&, const VECTOR&);
      friend  VECTOR operator+(const VECTOR&, const VECTOR&);
```

```
      friend  VECTOR operator-(const VECTOR&, const VECTOR&);
      friend  VECTOR operator*(double,const VECTOR&);
      friend  VECTOR operator*(const VECTOR&,double);
      friend  VECTOR operator/(const VECTOR&,double);
      friend  VECTOR operator*(const MATRIX&,const VECTOR&);
      friend  VECTOR operator*(const VECTOR&,const MATRIX&);
      friend  VECTOR operator*(const SMATRIX&,const VECTOR&);
      friend  VECTOR operator*(const VECTOR&,const SMATRIX&);
      friend  VECTOR operator*(const DMATRIX&,const VECTOR&);
      friend  VECTOR operator*(const VECTOR&,const DMATRIX&);
      friend  ostream& operator<<(ostream&, const VECTOR&);

      friend  void _write_(const VECTOR&,Zfstream&);
      friend  void _read_ (      VECTOR&,Zfstream&);

      double max_component(int &rank)const;

      friend void normalize(VECTOR&);
      friend void max_normalize(VECTOR&);
      friend double norm(const VECTOR& v);
      friend double max_absolute_component(const VECTOR&,int& i);

      friend double max_absolute_component(const VECTOR&);
      friend  ZBOOL if_colinear(const VECTOR&, const VECTOR&);

      friend  VECTOR vectorial_product(const VECTOR&,const VECTOR&);
};
```

Class Use:

Methods:

`resize` resize the vector.

`reassign` attach the vector a a sub- on aother vector.

Chapter 7

Base classes for Material Behaviors

MATERIAL_PIECE

Description:

This class is used as the base for *all* material behaviors, and also the sub-classes used in materials programming. The use of this class provides many useful pre-defined services for the management of coefficients, internal and auxiliary variables, and other MATERIAL_PIECE objects.

```
class MATERIAL_PIECE {
  protected :
    int                          vint_index, vaux_index, flux_index, grad_index;
    int                          _dim,_tsz, _utsz;
    MATERIAL_PIECE*              its_boss;
    EXTERNAL_PARAMETER_VECTOR    curr_ext_param;
    MAT_DATA*                    curr_mat_data;

  public :
    LIST<STORED_VARIABLE_SPEC*> local_vars;

    MATERIAL_PIECE();
    MATERIAL_PIECE(MATERIAL_PIECE*);
    MATERIAL_PIECE(MATERIAL_PIECE*,int);
    void initialize(MATERIAL_PIECE* boss);         // connect
    void initialize(MATERIAL_PIECE* boss,int dim); // connect
    MATERIAL_PIECE(const MATERIAL_PIECE& svs_in, MATERIAL_PIECE* boss); // copy

    virtual MATERIAL_PIECE* copy_self(MATERIAL_PIECE*);
    virtual ~MATERIAL_PIECE();

    MATERIAL_PIECE& operator=(const MATERIAL_PIECE& svs_in);
    void            make_curr_ext_param_recursive(int size);

    bool depends_on(const STRING&)const;

    virtual void add_name_prefix_recursive(const STRING& sin);
    virtual void set_name(const STRING& sin);
    virtual const STRING& name();
    void    rename_vars(const STRING&);

    virtual void attach_all( MAT_DATA&);
    virtual void attach( VECTOR& chi);
    virtual void attach( VECTOR& chi, VECTOR& d_chi );
    virtual void update_var_aux();
    virtual void set_var_int_to_var_int_ini();
    virtual void set_var_aux_to_var_aux_ini();

    virtual bool if_constant_coefs()const;
    virtual bool calc_coef();
    virtual int  calc_material(double);

    virtual void setup(int& flux_pos, int& grad_pos, int& vi_pos, int& va_pos);
    virtual void output_var_names(int flags)const;
    virtual int  default_output_variables( LIST<STRING>& names,
```

```
                                        LIST<STRING>& options);
virtual int  rank_of_variable(const STRING&) const;

//
// Find variable, with optimization now (12/22/98) the key classes
// are all destroyed in this class
//
LIST<MATERIAL_PIECE_VARIABLE_KEY*> variable_keys;
LIST<MATERIAL_PIECE_VARIABLE_KEY*> flux_var_keys;
LIST<MATERIAL_PIECE_VARIABLE_KEY*> grad_var_keys;
LIST<MATERIAL_PIECE_VARIABLE_KEY*> vint_var_keys;
LIST<MATERIAL_PIECE_VARIABLE_KEY*> vaux_var_keys;
MATERIAL_PIECE_VARIABLE_KEY* find_key(const char* what, MP_TYPE type,
                     LIST<MATERIAL_PIECE_VARIABLE_KEY*>& where_to_search);

//
// These are actually mutable const.. with the keys being mutable
//
STORED_VARIABLE_SPEC* get_var(const char*  what)const;
STORED_FLUX* get_flux_var(const char* what, bool give_err=TRUE)const;
STORED_GRAD* get_grad_var(const char* what, bool give_err=TRUE)const;
STORED_VINT* get_vint_var(const char* what, bool give_err=TRUE)const;
STORED_VAUX* get_vaux_var(const char* what, bool give_err=TRUE)const;

virtual STORED_VARIABLE_SPEC* output_get_var(const STRING& what)const;
const double& get_param(const STRING& what, MDAT_WHEN when=MDAT_CURR)const;

int  int_sz()const     { return int_var_sz; }
int  aux_sz()const     { return aux_var_sz; }
int  int_pos()const    { return int_var_pos; }
int  aux_pos()const    { return aux_var_pos; }
int  bb_sz() const { return(bb_var_sz); }

enum BEHAVIOR_LEVEL { FIRST, NEXT, TOP };
virtual BEHAVIOR*   find_behavior(BEHAVIOR_LEVEL top_level=FIRST)const;
MATERIAL_PIECE*     give_boss()const { return its_boss; }
void                impose_boss(MATERIAL_PIECE* bin);

int  tsz()const  { return _tsz; }
int  utsz()const { return _utsz; }
int  dim()const   { return _dim; }

virtual void    setup_name(STRING class_type, STRING stub);
virtual STRING  get_next_name_for(STRING class_type);
virtual int     get_next_index_for(STRING class_type);

virtual int  index_size();                  // INDEXES
virtual void set_flux_index(int idx);
virtual void set_grad_index(int idx);
virtual void set_vint_index(int idx);
virtual void set_vaux_index(int idx);

int  get_flux_index()const { return flux_index; }
```

```
    int  get_grad_index()const { return grad_index; }
    int  get_vint_index()const { return vint_index; }
    int  get_vaux_index()const { return vaux_index; }

    DECLARE_ASK;
    HIERARCHY_BASE;

    friend void add_var(LIST<STORED_VARIABLE_SPEC*>& var, const MATERIAL_PIECE* mp);

    virtual bool do_command(STRING msg);
];
```

Model Use:

The use of MATERIAL_PIECE is forced for all classes deriving from BEHAVIOR. Generally this will only have consequence if the user wants to know specific information about the varaible vector sizes, or needs to overload the default methods to do specialized setup or optimization. The class is designed such that behaviors may be created very quickly while allowing substantial later modifications to optimize. The ease of programming often comes from the automated handling of other sub-objects of type MATERIAL_PIECE.

Generally sub-classes used in behavior programming should be derived from this class if any of the following are true:

- The class has internal, auxiliary, flux, or gradient variables specific to the class. Examples could be a hardening model with an internal variable or a thermal heating with temperature coupling with the spacial problem.
- The class has coefficients of type COEFF. The MATERIAL_PIECE class will automatically update the coefficients every time the BEHAVIOR method calc_local_coefs is called.
- The class will have other sub-classes with the above classifications.

Setup of member data may be initialized either in the class creator, or the `setup` method called by the BEHAVIOR class during initialization. The later is perhaps preferable, as the complete problem object set will have been created at this point. If the `setup` method is overrided, it is essential to call the base implementation.

In your integration methods, the methods `set_var_int_to_var_int_ini` and `set_var_aux_to_var_aux_ini` should be used in place of manipulating those vectors themselves. The reason is encapuslating behaviors may have updated variables affected by the copy. These local methods only copy the portion of vectors which belong specifically to the behavior.

Error messages should only be issued using the standardized ERROR_MESSAGER routines. Please remember to assign the `err_name` member to identify the current class to the user.

Data members:

`tsz` is the symmetric tensor size to be used. This variable is available as a utility for the different material pieces. Note that although the variable is globally writable, there should be no reason to do so.

`its_boss` is a pointer on the MATERIAL_PIECE class which created the current object. The top-level BEHAVIOR object has NULL for `its_boss` thereby indicating that it is the "controlling" material piece. Note that a behavior object may be created within a behavior class, passing the top-level behavior pointer in creation (see `Update.[h,c]`).

`curr_ext_param` is an EXTERNAL_PARAMETER_VECTOR used in the calculation of coefficients. This vector **must be updated by the class user**. The vector will however be appropriatly sized, so one could have:

```
if (!curr_ext_param) curr_ext_param = *mdat.param_set();
```

in a material which is integrated exactly (no intermediate points during integration).

`curr_mat_data` is a pointer on the current MAT_DATA object as passed to the BEHAVIOR::integration method. Access of material variables will generally be given through the STORED_VARIABLE_SPEC objects. Sometimes, however, it is necessary to use the MAT_DATA directly to obtain initial values, or other unmaintained information.

Creators and initialization:

The creators for this class generally pass a super class of type MATERIAL_PIECE which is used to establish the connectivity beteen the different objects. The sucess of automated opterations depends on the integrity of this interconnectivity. Generally, a sub-class object will be passed a pointer of the current object when it is created.

`setup` This method is called recursively through all the MATERIAL_PIECE objects by the behavior in initialization. During this chain of calls the problem variables are assigned their relative positions in the MAT_DATA vectors. The MATERIAL_PIECE's sizing variables are also assigned during this sequence. Overloading this method therefore gives a good oportunity to initialize local optimization variables, etc. **It is however essential that the default method be called**. For example, suppose a class B is to be added, deriving from a user class A which in turn is a MATERIAL_PIECE. An overloaded method would be:

```
void B::setup(int& flux_pos, int& grad_pos, int& vi_pos, int& va_pos)
{   A::setup(flux_pos,grad_pos,vi_pos,va_pos);
    local_tensor.resize(tsz);
    total_vector.resize(int_sz());
}
```

The default is called through the class A in case there was additional setup in that class.

Inquiries:

`set_name, name()` These two methods are to be used for assigning the material-piece's name identifier. The use of this name is entirely user-defined, but nevertheless maintained as an encapsulated MATERIAL_PIECE member.

`int_pos(), aux_pos()` The start-positions in the MAT_DATA internal and auxialiary variable vectors for this object's local variables. It is generally better to make inquiries to each of the STORED_VARIABLE_SPEC objects however.

`int_sz(), aux_sz()` give the sizes of the *local* internal and auxiliary variables which are managed by the current object. The actual sizes cannot be changed by the user, so that **use of STORED_VARIABLE_SPEC is required**.

`if_constant_coefs` returns TRUE if *all* the coefficients in the object, *and* also in *all* this class's MATERIAL_PIECE objects do not depend on any external parameters, auxiliary variables, or internal variables.

`depends_on` Returns TRUE if the current object has coefficients which depend on the given named variable (ext. param, int var, aux var). This method recursivly calls the class's MATERIAL_PIECE objects as well.

`find_behavior` This method will search the chain of bosses until the hishest level is found (having a NULL boss. If the behavior is used frequently in a MATERIAL_PIECE implementation, it is best to make a local pointer on the behavior initialized in the creator or `setup` method.

`give_boss` public method which returns the current object's boss pointer. Returns NULL for the top-level behavior.

`get_var` This method returns a pointer on the appropriate named stored variable. A NULL pointer is returned in the event that no such variable exists in the MATERIAL_PIECE.

`get_param` used to get the value for a named external parmater at a given time in the problem.

`default_output_variables` Assigns STRINGs for the output variable names to the first passed parameter. The second parameter is a list of strings used to set user output, or specify the default (*rf:not sure anymore... take a look at it + comment!!*).

User Methods:

`attach_all( MAT_DATA& mdat )` Used to assign all the object's STORED_VARIABLE_SPEC objects onto the current integration point data. This method is usually called in the behavior `integrate` method as:

```
INTEGRATION_RESULT* USER::integrate(MAT_DATA& mdat,
                  const VECTOR& dg, MATRIX*& tg_matrix,
                  int flags)
{ attach_all(mdat);
  ...
}
```

The method is virtual so a derived class may use the opportunity to do initialization only required once for every `integrate` call (rare).

`attach( VECTOR& chi )` This version of `attach` is used to reassign the local variables to the current integration variables. Only variables attached to the MAT_DATA `var_int` vector are affected. This method will be called by a behavior in the integration method's functions (`derivative` or `calc_grad_f` for example). Overloading this method may be a nice way to perform some local calculations required in each step or iteration of the integration.

`attach( VECTOR& chi, VECTOR& d_chi )` This alternate of the preceeding method may be used for cases where the trial increment of the integrated variables is interesting. The default method merely calls the above single parameter method.

`calc_coef` This method is used to calculate an object's coefficients, and dispatches `calc_coef` calls to all the MATERIAL_PIECE data objects. This method should only be overloaded in the event that intermediate values are desirable (i.e. to calculate the ratio of two coefficients which will be used freqeuently). A `bool` must be returned indicating that something was calculated.

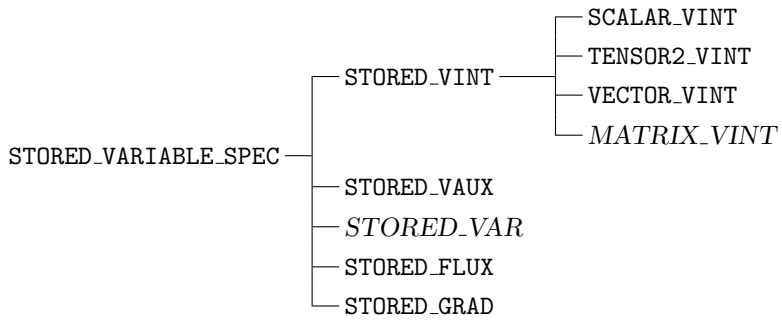
Hacker Notes:

This class keeps lists of pointers on the objects which compose the material piece analogy. These include COEFF, STORED_VARIABLE_SPEC, MATERIAL_PIECE, and STORED_VINT. Additions are made to these lists in the constructors which take a MATERIAL_PIECE*. The passed object will be assigned to `its_boss`, and the new obejct added to this boss's list data.

The class does *not* destroy the pointers in these lists because the real objects are usually stored as real data members, and not with dynamically created objects.

STORED_VARIABLE_SPEC

This class is used to handle the cutting up of the behavior variables into usable matematical entities. The class provides the following hierarchy:



class listing

```
class STORED_VARIABLE_SPEC : public ERROR_MESSAGER {
    friend class MATERIAL_PIECE;
  public :
    visible_protect(int, start_pos);
    visible_protect(int, var_size);

    STORED_VARIABLE_SPEC();
    STORED_VARIABLE_SPEC(MATERIAL_PIECE* mp);
    STORED_VARIABLE_SPEC(MATERIAL_PIECE* mp, int size);
    STORED_VARIABLE_SPEC(const STORED_VARIABLE_SPEC& svs_in);
    virtual ~STORED_VARIABLE_SPEC();

    STORED_VARIABLE_SPEC& operator=(const STORED_VARIABLE_SPEC& svs_in);
    bool operator==(const STORED_VARIABLE_SPEC& svs_in);
    int operator==(const STRING& str_in)const ;

            void initialize(STRING name, int ten_sz, int def_output=1 );
    virtual void resize(int sz);
    virtual void setup( STRING name, int def_output=1 );

    void set_output()    { output_var=1; }
    void set_no_output() { output_var=0; }
    int operator!()const { return var_size; }

    virtual void attach( MAT_DATA& mdat )=0;
            void place_var( SCALAR&  wht, VECTOR& the_vec);
            void place_var( TENSOR2& wht, VECTOR& the_vec);
            void place_var( VECTOR&  wht, VECTOR& the_vec);

    virtual bool  is_in(STRING what)const=0;
    virtual const double& value(const MAT_DATA& mdat)const=0;
            const double& value(const STRING& what, const MAT_DATA& mdat)const;
```

```
    virtual const double& d_value(const MAT_DATA& mdat)const;
            const double& d_value(const STRING& what, const MAT_DATA& mdat)const;
    void                  set_value(const STRING& what, MAT_DATA& mdat, double val)const;

    virtual void  set_value(MAT_DATA& mdat, double val)const;

    const double& filter(const double& d)const;
};
```

Class Use:

All the end types are seen to have a certain attachment, and also a mathematical type. The later is established using a multiple inheritence relationship (not shown) with the corresponding math library class. Note also the use of SCALAR for scalar variables. Because the built-in `double` type cannot have an equivilence operator defined for the SCALAR type, the operator `()` is available:

```
double tempo = user_var();
```

Normally these variables should be specified in a MATERIAL_PIECE class definition. The structure of these variables is suitable for concrete declaration which would be preferable to using pointers on dynamically created objects.

Note that the variables have assignment operators, and equivilence `==`. The later may be used to compare a variable with another, or with a string. The string comparison uses the name given in `setup` or `initialize`.

When using these variables, it may arrise that a single material law will prefer to place the same variable in different locations (e.g. in var_int or var_aux), or that a given variable is only needed sometimes. Instead of requiring that different implementations be given for each case, the variables have some methods to control their behavior. A variable will *not* be considered an active part of the problem (taking up space in the MAT_DATA structure or being output if any of the the following are true (even if they are added to the MATERIAL_PIECE's list of local variables):

- The variable has not been sized greater than zero.
- The variable has been re-sized to zero.
- The variable has been reassigned to a local utility storage. This is used in `Update.c` to allow the change of gradient variables for finite strain.

Data members:

`start_pos, var_size` These two variables give the variable position and length in the assiciated MAT_DATA vector (be it flux, grad, var_int, or var_aux). Their access may be useful if another variable must be assigned to the trial increment or initial value vectors during integration. Note that one may use the `place_var` method instead.

Creators and initialization:

The creators require a MATERIAL_PIECE object to be given for each declared variable. The default creator is given to avoid compiler errors which arise when expanding ARRAY¡¿ or LIST¡¿ templates of the stored variables, and gives a default error message at run time. These variables must therefore be initialized in for each class which holds them. An example is the following:

```
class USER : public MATERIAL_PIECE {
     TENSOR2_VINT  alpha;
     SCALAR_VINT   beta;
   public:
```

```
    USER(MATERIAL_PIECE* mp) : MATERIAL_PIECE(mp), alpha(this), beta(this) {
       alpha.initialize("alpha",tsz,0);
       beta.setup("alpha");
    }
    double dbeta() { return beta*(alpha|alpha); }
    ...
};
```

After creation, **all active variables must be initialized**. The following methods are thus provided to assign a name, allocate storage, and specify that the variable sould be output by default or not:

`initialize` sets simultaniously the variable size, the variable name and if it is default output (default parameter = 1 = yes).

`resize` sizes the variable approporately. The method is overloaded for the differernt types of stored variable in order that the correct math class `resize` member will be called.

`setup` sets the variable name. This is called from the `initialize` method.

`set_output, set_no_output` These methods are used to set the internal flagging for default output variables.

User Methods:

`attach` This method is called from within the MATERIAL_PIECE `attach` methods. Each different type of variable re-defines this method to attach the proper data type to the proper variable vector.

`place_var` These methods are provide a shorthand way to assign other variables to a vector as if it were the variable itself (??). For example if the trial vector of integrated variables is `d_chi`, one can attach a utility variable for the delta value using one of the STORED_VINT variables. `TENSOR2 deel; eel.place_var(deel,d_chi);`

`is_in` returns true if the given variable name is valid for the stored variable object. This marks internally the position so the next call to `value` etc will return the last searched variable value. This method responds to `sig22` in a tensorial stored variable named `sig`. Other responses may be possible such as `sig::mises` could return the equivilent of a tensor variable.

`value, d_value, set_value` These three methods are used in the output and spatial integration routines. What happens is a function will ask the behavior for a certain variable. This is actually calling the MATERIAL_PIECE::get_var method which returns a STORED_VARIABLE_SPEC object. In searching for the variable (e.g. `sig22`), the the `is_in` methods will called for all the problem variables until a yes response is found. The sucessful `is_in` method will mark the position of the requested value in the variable. Thus the value or delta value may be had in calling `value` or `d_value` with the MAT_DATA as the only parameter. The method `set_value` is used to initilize the internal variables.

`filter` *method to be used for filtering noise in variables (not yet implemented). The idea is to have internal flags for different filters... or maybe a filter object which will elminate noise on the variables. This could be positive only variables, variables of a certain magnitude or over, etc.*

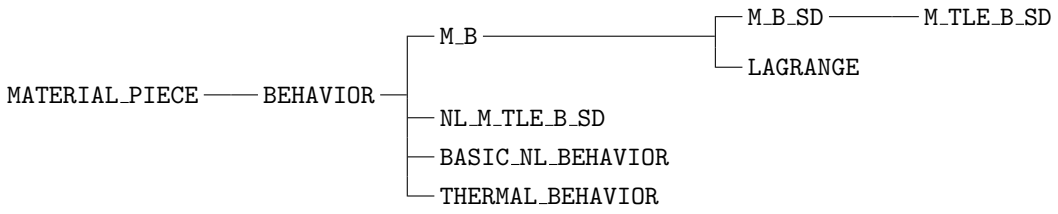
Hacker Notes:

The class is managed with the following protected data:

```
protected :
  STRING          stub;
  int             output_var;
  MATERIAL_PIECE* its_mp;

  enum {  NOWHERE_VAR=0,
          VINT_VAR=1,
          VAUX_VAR=2,
          FLUX_VAR=4,
          GRAD_VAR=8 } location;

  CARRAY<STRING> local_names;
  int            last_var_index;
  double         factor;
  static double  zero;

  void output_var_names()const;  // MUST BE CALLED AFTER init !!!
  virtual void init(int& flux_pos, int& grad_pos, int& vi_pos, int& va_pos)=0;
```

BEHAVIOR

The basic class structure based on BEHAVIOR is at present the folowing:



These are the base classes from which we derive actual behavior implementations. These final behaviors are discussed within the context of their particular model and coding. The intermediate classes desribed here rather provide some standard services in order to facilitate repeditive tasks within a particular domain of problem.

The base class `BEHAVIOR` provides the mechanism for gauss point data storage and user access, defines the interface for material law integration functions, and initiates the reading of the material file. Global material coefficients are read and stored in the base class. Each `BEHAVIOR` class is identified by a `STRING` name.

class listing

```
class BEHAVIOR : public MATERIAL_PIECE {
  protected :
      PTR<ROTATION>    its_rotation;
      int              flux_sz, grad_sz;

  public :
      enum FLAG_TYPE  {    CALC_TG_MATRIX=1,
                           GIVE_INCREMENTAL_FLUX=2,
      };
      PLIST<BEHAVIOR>      sub_behaviors;
      const int            space_dimension;
      STRING               zone_name;
      int                  integration;

      ARRAY<STRING>        name_initialized_var_int;
      ARRAY<double>        value_initialized_var_int;
      ARRAY<STRING>        file_name_initialized_var_int;
      PARRAY<fstream>      file_initialized_var_int;

      enum {MASVOL, CAPACITY,L_COEF,_NB_COEF_};
      PARRAY<COEFFICIENT> coef;

      static BEHAVIOR* read(int dim,
                            const STRING& zone_name,
                            const STRING& mat_file,
                            LOCAL_INTEGRATION* integr);
      BEHAVIOR(int);
```

```
      virtual ~BEHAVIOR();

      virtual int  base_read(const STRING& tok, ASCII_FILE& file);
      void         read_base_info(ASCII_FILE& file);
      virtual bool if_broken(const MAT_DATA& mdat)const;

      void  set_rotation(ROTATION* rot);
      const ROTATION* rotation()const;

      static BEHAVIOR* make_modifier(const STRING&,BEHAVIOR*,ASCII_FILE&,int);

      int flux_size()const;
      int grad_size()const;

      COEFFICIENT*   get_coef_named(const STRING&);

      virtual void   init();
      virtual void   init_material_data(MAT_DATA&);

     // -------------------------------------------------------------
     // Mechanical behavior methods... default error
     // -------------------------------------------------------------
     virtual SMATRIX get_elasticity_matrix(MAT_DATA& mdat,double theta);
     virtual TENSOR2 get_non_mechanical_strain(MAT_DATA& mdat,double theta);

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

     // -------------------------------------------------------------
     // Thermal behavior methods... default error
     // -------------------------------------------------------------
     virtual INTEGRATION_RESULT* integrate( MAT_DATA&      mdat,
                                            const VECTOR&  grad,
                                            double         T,  double   T0,
                                            bool           ifCS,
                                            DMATRIX&       k,  DMATRIX& dk_dT,
                                            double&        dH, double&  dH_dT);

      LIST<COEFF*> Local_coefs;
      void         calc_local_coefs( );
};
```

Class Use:

The use of the behavior class is better seen through the derived class implementations. Typically one would find a behavior "close" to the desired type, and follows the previous implementation to start. External use (in the FEM classes) is largely established, and should not require much modification. In reality, the behavior class serves a minimal functionality in terms of the FEM program. The basic points may be summarized as follows:

- Used to update the internal variables, auxiliary variables, and flux vector using the `integrate` method. Different syntaxes are available for this method according to the type of problem

at hand. Default methods give a default "not-implemented" error. The behavior interface is thus exensible without requiring modification of the existing classes.

- access global coefficients which are required for things such as garvitational force, etc.
- to decipher the material variable storage, and give appropriate values given a named request. This functionality is now handled in the MATERIAL_PIECE class.
- Some special methods are given for specialized problems. Examples are the `if_broken` method or the `get_elasticity_matrix`. Default error messages or empty default implementations are given depending on the generality of the method. If doing nothing is not harmfull to a particular application, this will be the default; otherwise a "not-implemented" error message is issued.

Basic implementations have a normal calling sequence which may be exploited to gain an efficient imlementation for the material behaviors. One principle rule may be derived from the fact that normal use creates a small number of behavior objects in the life of a calculation. These objects are also created in the problem startup and kept until the calculation is complete. Great gains in performence can thus be obtained by setting up all the required data structures only during the initialization of the behavior. There are two opportunities for this setup. The first is in the class constructor. The behavior now has knowlege of the integration method to be used in the constructor, and can thus plan accordingly for the type of data required. The variable sizes and positions are however not yet initialized (see discussions in MATERIAL_PIECE and STORED_VARIABLE_SPEC). The second opportunity to perform setup is thus by overloading the `MATERIAL_PIECE::setup` method (remember to call the default!).

During execution of the problem, the `integrate` method will be called repeatidly, passing the current MAT_DATA object and the imposed gradient. Behaviors which integrate directly will want to call `attach_all(mdat);` followed by `calc_local_coefs()`. After this point the programming of the integration should be straightforward. Behaviors with a numerical integration should call `attach_all(mdat);` first, determine what integration is active if there are multiple choices, and dispatch approporately to the integration method. In the integration function (`derivative` or `calc_grad_f`), the MATERIAL_PIECE method `attach` should be called with the current trial vector of integrated varaibles, and `calc_local_coefs()` re-called to update possible coefficient dependencies. External parameters must also be updated for the intermediate point before the coefs are calculated, so it is often a good idea to make a sub-proceedure to perform the coefficient calculation:

```
void USER::calc_material(double fract)
{  if (Variable_coefs==0 && Coef_evaluated) return;
   Coef_evaluated=TRUE;
   assert(fract>=0. && fract<=1.);
   if (!curr_ext_param) {
         curr_ext_param  = (*curr_mat_data->param_set_ini())*(1.0-fract);
         curr_ext_param += (*curr_mat_data->param_set())*(fract);
   } calc_local_coefs();
}
```

Data members:

`its_rotation` is a pointer on the active material coordinate / global coordinate frame rotation. To go from global to material one `apply`'s the rotation, and conversly to go back to global the rotation object `remove`'s the rotation. Remeber to document in which referencial the internal variables are stored when there is a rotation[1].

[1] we could attach the behavior's rotation to the TENSOR2_VINT objects etc, and have them remove the rotaion during output...

`flux_sz, grad_sz` keep track of the total flux and grad size as composed by all *active* STORED_FLUX and STORED_GRAD objects.

`space_dimension` The spatial dimension. I would prefer if this were an `enum` of SPACE_DIMENSION which could take on more info describing the dimension. A more precise method would have to be given to translate the space dimension to a tensor size...

`sub_behaviors` **Should be changed to use the** MATERIAL_PIECE **methods**.

`integration` an `enum` value used to desribe the type of integration method. This must now be set by the user in the BEHAVIOR creator. In the event that NULL is passed to the creator, the object will be responsible for creating a default integration method if necessary.

`zone_name` name of the element set in which the current behavior applies.

`FLAG_TYPE` this enum is used to set and verify bits in the integrate method's `flag` parameter.

`coef` this array of COEFFICIENT **should be changed**. Currently they are accessed using the preceeding enumerations. We should rather have individual COEFF objects introduced at the point within the class hierarchy where they become physically valid.

`local_coefs` list of COEFF* pointers for all the COEFF objects which exist in the current behavior. These pointers are maintained by the COEFF class during creation and destruction. The user has no reason to manipulate or acess this list. Calling the `calc_local_coefs()` method will update the coefficients with the data used in the last `attach_all` or `attach` call (defined in MATERIAL_PIECE/).

There are some data memebers which still need documentation:

`name_initialized_var_int`

`value_initialized_var_int`

`file_name_initialized_var_int`

`file_initialized_var_int`

Creators and initialization:

The base behavior class only includes the contructor taking an integer for the space dimension (**should be changed to space dim. enum**). The different behaviors may be created by using the `read` method which calls an "auto-read" object installation. The constructor for derived behaviors must thus be consistent with the prototype given for the auto-read functions. This prototype is the following:

```
NEW_CLASS::NEW_CLASS( ASCII_FILE&          file,
                      int                  dim,
                      LOCAL_INTEGRATION*   integ );
```

The auto-read declaration is simplified for the behavior class using another define:
`BEHAVIOR_READER(NEW_CLASS, behavior-string-name);`

`read_base_info` Performs a `ASCII_FILE::locate` call to search for `***coefficient` in the material file. Global coefficients are then read and the file re-wound to its original position. The reqinding is considered to be a ***bad*** thing.

`base_read` This is another option for loading the coefficients, or other stuff which exists in a base class. A sample material file read function could be:

```
for (;;) {
    STRING str = file.getSTRING();
    if (str.start_with("***") { file.back(); break; }
    else if (BASE::base_read(str,file)) { }
    else if (str == "**user_tok) {
        /* user input code */
    }
    else INPUT_ERROR("Unkown command: "+str);
}
```

The method is overrided for the different base classes given above (M_B, etc), to give a default reading for the basic data contained in these classes, and re-used for all derived classes. The call of `base_read` should always be to the immediate super-class of the current, even if there is no overrided `base_read` method. This allows one to be added at a later date without updating of the derived classes.

`init` Used to request that the behavior initialize itself after the problem is loaded. The external parameters may now be observed, and the `MATERIAL_PIECE::setup` recursive calling is initiated. Method is called only from the `BEHAVIOR::read` method.

`init_material_data` called in `P_element.c` to initialize the MAT_DATA structures for the problem.

`make_modifier` is used to manage the behavior modifiers.

Inquiries and flags:

`if_broken` used for breakable elements or interfaces so the particular element may be removed from the problem. The dafault implementation never breaks.

`set_rotation, rotation` used to set and access the behavior's rotation structure.

`get_coef_named` method used in the global code to get access to the different global coefficients. **This method should be changed**.

`get_elasticity_matrix` Used for the MMC model.

`get_non_mechanical_strain` Used for the MMC model.

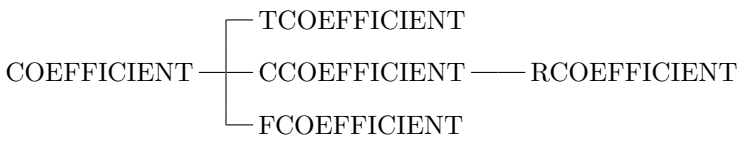
User Methods:

`integrate` This method is where the material must be updated over a loading increment.

`calc_local_coefs` This method is used to loop through the list of COEFF object to update their values. The call uses the most current MAT_DATA structure. Coefficients will call their MATERIAL_PIECE object in asking for a certain variable if there are variable dependencies. Thus a coefficient may have limited access to the variable set if its `boss` is a material piece other than the behavior itself. For example, suppose there is a material piece HARDENING which has some coefficients. Initializing the coefficients in HARDENING with `this` would limit the possible dependencies ofr that coefficient to the variable set in the HARDENING class *only*. Using instead `find_behavior()` would give access to the entire problem's variable set.

COEFFICIENT

COEFFICIENT are used be BEHAVIOR. There are three different types derived from the base class.



TCOEFFICIENT are coefficients defined by a TABLE, CCOEFFICIENT coefficients that are constant and FCOEFFICIENT coefficients that are defined by a FUNCTION, RCOEFFICIENT} coefficients that are constant but vary in the structure.

See Also : COEFF, MATERIAL_PIECE

class listing

```
class COEFFICIENT : public ERROR_MESSAGER {
  protected :
        const STRING            name;
        ARRAY<STRING>           param_name;
        const MATERIAL_PIECE*   const its_boss;

        VECTOR   param_value;
        void     set_param_value();
  public :
        COEFFICIENT(const STRING& name, const MATERIAL_PIECE* const boss);
        COEFFICIENT(const COEFFICIENT& cin);
        virtual ~COEFFICIENT();
        virtual COEFFICIENT* copy_self()const;

        static COEFFICIENT* read( ASCII_FILE&,
                                  const STRING& name,
                                  MATERIAL_PIECE* boss=NULL );

        static void read_coefficient(PARRAY<COEFFICIENT>& coefs,
                        const CARRAY<STRING>& coef_names,
                        const ARRAY<bool>& enforce,
                        const STRING& err_msg,
                        ASCII_FILE&,
                        MATERIAL_PIECE* boss=NULL);

        const ARRAY<STRING>& parameters()const { return param_name; }

        const STRING get_name()const {return name;};

        void set_parameters();
        void set_parameters(double x);

        bool depends_on(const char* const)const;
        virtual double  compute_value()const=0;
        virtual double  initial_value();
```

```
        virtual double  delta_value();
        double  compute_value(double x);
        virtual double  d_param(const char* const param_name)const=0;
        virtual double  d_param(double x)const;
        virtual double  d_param()const;
        virtual void init();
        static LIST<COEFFICIENT*>* List_of_coefficient;
        static void Init_all_coefficient();
        HIERARCHY_BASE;
};
```

Class Use:

The COEFFICIENT classes should **never** be used directly in a material behavior, or in a part of a material behavior. Instead, the COEFF envelope class should be used to maintain the coefficients. This file is given for those wishing to add new types of coefficients only.

Data members:

`name` The coefficient name. This is used for error messages, etc.

`param_name` vector of the STRING names of all dependency varaibles[2].

`its_boss` the MATERIAL_PIECE which is managing the current coefficient. The coefficient may depend on any variable at or below `its_boss` if that class allows[3].

`param_value` vector of the current parameter values. These are arranged in the same order as the `param_name`.

`List_of_coefficient` List of all the problem's coefficients which is required for some reason. Maybe for spatially varying coefficients?

Hacker Notes:

The current version of COEFFICIENT has been modified to use the MATERIAL_PIECE methodology for material components. Basically now it all happens in the `COEFFICIENT::set_param_value()` method. The coefficient keeps the current values of params in a vector `param_value` which must be updated at each `set_param_value()` call. Updateing will call the coefficient's MATERIAL_PIECE boss with a named request for a value. This is in fact a two part process. Thr first part calls for an external parameter value, and if not found the second part is a search for an internal or auxiliary variable with the appropriate name. This process is *still under developpment*. Modifications should concentrate on elegence and not efficiency. Optimization will be added later to the MATERIAL_PIECE search process.

[2] we should have a utility class such as PARAMETER_SPEC which contains the name, pointer on the variable class, current value, etc... lump it all together

[3] the mechanism to limit variable access to the coefficients is cureently missing. There should be some type of "publish" methods in the stored variable classes.

COEFF

This class serves much the same function for coefficients as the STORED_VARIABLE_SPEC class does for stored material variables. The point here is to hide the different types of COEFFICIENT inside an envelope class which can be used as a concrete data type (no pointers).

class listing

```
class COEFF : public SCALAR {
  public :
        COEFF();
        void read(const STRING& name, ASCII_FILE&, MATERIAL_PIECE*);
        void set_coefficient(COEFFICIENT* cin, MATERIAL_PIECE* mp);

        COEFF(const COEFF& cin);
        COEFF&  operator=(const COEFF& c);
        COEFF&  operator=(const double& d);
        COEFF&  operator*=(const double& d);
        COEFF&  operator/=(const double& d);

        bool    ok()const;
        double& operator()();
        const double& operator()()const;

        void  set_factor(double d);

        virtual ~COEFF();

        const ARRAY<STRING>& parameters()const;
        const STRING get_name()const;

        void  set_parameters();
        void  set_parameters(double x);

        bool depends_on(STRING wht)const;
        virtual void init();

        double  initial_value();
        double  delta_value();
        double  compute_value();
        double  compute_value(double x);

        virtual double  d_param(const char* const param_name)const;
                double  d_param(double x)const;
                double  d_param()const;

        bool areYouA(const char* wht) const;
        const char* isA()const;
        bool verification();
```

Class Use:

This class intends that the material coefficients be as easy to use as a `double` value in the models.

The class thus derives from a base class SCALAR having a multitude of operators pre-defined. Alas, the one thing which cannot be done is re-define an equivilence operator for the `double` class itself, so to do assignment to a `double` one must use the `()` operator: `double tmp = user_coef();`

This class hids the fact that coefficients may be any of the coefficient class's derived types. The use of pointers is thus hidden to the user while still supporing the polymorphic constructions. The class also permits the use of a scaling factor to be added to the coefficient in order to elminate almost totally the need for user `calc_coef` methods. User classes should normally include the COEFF objects as concrete data in their declarations:

```
class MODEL : public MATERIAL_PIECE {
   protected :
      COEFF A, B;
   public :
      MODEL(ASCII_FILE& file, MATERIAL_PIECE* mp);
};
```

The coefficients are usually read in the creator, which could be for example:

```
MODEL::MODEL(ASCII_FILE& file, MATERIAL_PIECE* mp) :
   MATERIAL_PIECE(mp)
{  err_name = "MODEL";
   for (;;) {
       STRING str = file.getSTRING();
       if (str[0]=='*') { file.back(); break; }
       else if (str=="A") A.read(str,file,this);
       else if (str=="B") B.read(str,file,mp);
       else INPUT_ERROR("Unknown coefficient: "+str);
   }
   if (!B.ok()) INPUT_ERROR("Coefficient B required");
   B *= 1.5;
}
```

Note that the coefficient `B` may depend on the variables maintained by the boss material piece `mp`, while the coefficient `A` may only depend on the variables in the MODEL class.

Data members:

Creators and initialization:

Inquiries:

User Methods:

Hacker Notes:

```
protected :
      COEFFICIENT* coef;
      double       factor;
```

COEFFICIENT_MATRIX

This class is used for matricies which are composed of coeffient values (e.g. elasticity matrix, etc). The class supports many different forms of matrix, and may be used as a SMATRIX object (it is one).

class listing

```
class COEFFICIENT_MATRIX : public SMATRIX,
                           public MATERIAL_PIECE {

  protected :
    virtual void compute_matrix()=0;

  public :

             COEFFICIENT_MATRIX( MATERIAL_PIECE* boss ,
                                 MATRIX_TYPE a_type);
             COEFFICIENT_MATRIX(const COEFFICIENT_MATRIX& cin);

    virtual ~COEFFICIENT_MATRIX();

    static  COEFFICIENT_MATRIX* read(  ASCII_FILE&        file,
                                       MATERIAL_PIECE*    boss,
                                       const char*        nm="y",
                                       const char*        def="default" );

    virtual bool calc_coef();

    bool    if_constant()const;
    virtual SMATRIX d_param(const STRING& p_name);
};
```

Class Use:

The coefficient matrix provides a flexible interface for reading the material input file, which is necessitated by the different applications it can be used for. Prinipally, the user may select what matrix type will be the default (in the absence of a specific declaration), and also what will be the naming scheme for the individual elements of the matrix. The most basic application of this class is to load an elasticity object in a material behavior: `E = COEFFICIENT_MATRIX::read(file,this);` which allows the following syntax to be read: `*elasticity  young 200000.0  poisson 0.3` slap

Other applications may be to use the matrix as an optional anisotropic coefficient M. Here one would read the matrix using the command: `M = COEFFICIENT_MATRIX::read(file,this,"M","diagonal");` which will be able to read the following material file excerpt: `*interaction M 2000.0`

CRITERION

class listing

```
class CRITERION : public MATERIAL_PIECE {
protected:
   SMATRIX         _unit;
   TENSOR2         norm;
public:
                      CRITERION(MATERIAL_PIECE* boss);
                      CRITERION(const CRITERION& cin);
   virtual CRITERION* copy_self();

   virtual            ~CRITERION();

   static  CRITERION* read(ASCII_FILE&, MATERIAL_PIECE*);

   virtual double  yield(const TENSOR2& sig_eff, double radius);

   virtual double  eq_stress(const TENSOR2& sig_eff);

   virtual const   TENSOR2&  dcrit_dsig();
   virtual const   TENSOR2&  normal();
   virtual         MATRIX    dnorm_dsig();
   virtual         TENSOR2   dcrit_dbs();
   virtual         double    dcrit_dradius();
   virtual         MATRIX    dnorm_dbs();
           const   TENSOR2&  last_normal() { return norm; }

           const   SMATRIX&  M_matrix()    { return _unit; }


   virtual double  yield(const TENSOR2& sig, const TENSOR2& x1,
                         const TENSOR2& x2,  double radius);
   virtual const   TENSOR2&  dcrit_dsig(int num);
   virtual         MATRIX    dnorm_dsig(int num);
   virtual         TENSOR2   dcrit_dbs(int num);
   virtual         MATRIX    dnorm_dbs(int num);
   virtual    const TENSOR2& normal(int num);

   HIERARCHY_BASE;
};
```

Class Use:

Data members:

Creators and initialization:

Inquiries:

User Methods:

Hacker Notes:

FLOW

class listing

```
class FLOW : public MATERIAL_PIECE {
protected:
   double              _k_coef;
public:
                       FLOW(MATERIAL_PIECE* boss);
                       FLOW(const FLOW& in);

   static  FLOW*       read(ASCII_FILE& file, MATERIAL_PIECE* b);
   MATERIAL_PIECE*     copy_self()const;

   const   double&     k_coef()     { return _k_coef;  }
   virtual int         plasticity() { return 0; }
   void                set_variables(double v, double stress);

   virtual double      flow_rate(double v, double stress);
   virtual double      dflow_dv();
   virtual double      dflow_dcrit();

   virtual double      kappa(double dp, double dt);
   virtual double      dkappa_dv();
   virtual double      dkappa_dcrit();

   virtual bool        deriv_wrt(double& ret, const char* const wht);

   DERIVED;
};
```

Class Use:

Data members:

Creators and initialization:

Inquiries:

User Methods:

Hacker Notes:

Chapter 8

Specific behaviors

THERMO_LINEAR_ELASTIC_SD

This class is the basic implemntation of elastic behavior with a possible thermal strain component. The implementation is very short and therefore a good example to start with. The whole model is contained in a single source file (.c), and because of the auto-read connectivity the no other file has dependency on this model.

class listing

```
class THERMO_LINEAR_ELASTIC_SD : public M_TLE_B_SD {
  public :
     THERMO_LINEAR_ELASTIC_SD(ASCII_FILE& file, int dim, LOCAL_INTEGRATION*);
     INTEGRATION_RESULT* integrate(MAT_DATA& mdat,const VECTOR& delta_grad,
                                   MATRIX*& tg_matrix, int flags);
};
```

Creaton and setup:

```
BEHAVIOR_READER(THERMO_LINEAR_ELASTIC_SD,linear_elastic);

THERMO_LINEAR_ELASTIC_SD::THERMO_LINEAR_ELASTIC_SD(
                  ASCII_FILE&          file,
                  int                  dim,
                  LOCAL_INTEGRATION*   /* integ */ ) :
  M_TLE_B_SD(file, dim)
{
  for (;;) {
      STRING str=file.getSTRING();
      if (str.start_with("***") ) break;
      else if (!base_read(str,file)) INPUT_ERROR(str());
  } file.back();
}
```

Implementation:

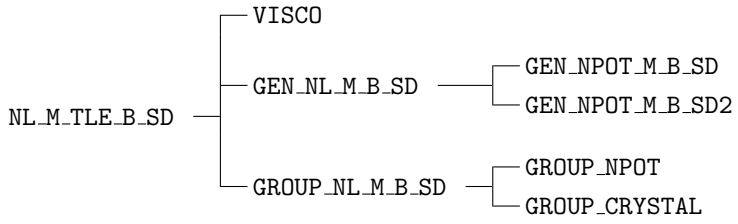
```
INTEGRATION_RESULT* THERMO_LINEAR_ELASTIC_SD::integrate( MAT_DATA&        mdat,
                                                         const VECTOR&    delta_grad,
                                                         MATRIX*&         tg_matrix,
                                                         int              flags        )
{ attach_all(mdat);
  if (!curr_ext_param) curr_ext_param = *mdat.param_set();
  calc_local_coefs();

  if (thermal_strain.if_not_null() && mdat.param_set())
        sig = *elasticity*(eto - thermal_strain->compute_strain());
  else  sig = *elasticity*eto;

  if (flags&CALC_TG_MATRIX) tg_matrix = elasticity();

  return NULL;
}
```

The `attach_all` call will setup the flux tensor `sig` and the grad tensor `eto`. The data `curr_ext_param` defined in MATERIAL_PIECE must be assigned to the current value, after which all the coefficients may be calculated in `calc_local_coefs`. This later will calculate the /COEFFICIENT_MATRIX/ COEFF objects because /COEFFICIENT_MATRIX/ is a MATERIAL_PIECE.

NL_M_TLE_B_SD

NonLinear Mechanical Thermal Linear Elastic Behavior Small Displacement.
— Class is derived from : **M_B_SD, RUNGE_INTEGRATOR, THETA_INTEGRATOR** [1]

The basic class structure based on NL_M_TLE_B_SD is at present the folowing:



This class and its derivatives are intended to be used for generalized non-linear mechanical behavior taking into account for thermal strains. The derivation from both `RUNGE_INTEGRATOR` and `THETA_INTEGRATOR` allows us to write the general functions for initial treatment in `integrate` while the actual material law is computed in `derivative` and `calc_grad_f`. We are of course free to override these functions. Virtual functions are usually designed to be called at the head of the redefined function such as in `calc_material`. This allows the part which is always required to be done n the base class, while at the same time we are adding on to it.

The derived classes implement specific models of plastic / viscoplastic behavior either explicitly as in the case of `VISCO` or generally for multi-potential systems in the `GEN_NL_M_B_SD` and `GROUP_NL_M_B_SD` classes.

class listing

```
class NL_M_TLE_B_SD :   public M_B_SD,
                        public RUNGE_INTEGRATOR,
                        public THETA_INTEGRATOR {
protected:
    int                         psz;
    int                         tot_sz;
    int                         rank_of_temperature;
    double                      m_last_temp;
    EXTERNAL_PARAMETER_VECTOR  *m_param_set_int;
    RUNGE                      *m_intrk;
    THETA                      *m_inttn;
    GLOBAL_HANDLER             *m_glob;

    TENSOR2                     m_detot;
    VECTOR                      m_f_vec;
    VECTOR                      m_d_chi;
    VECTOR                      m_f_0;
    SMATRIX                     m_f_grad;
    virtual void          set_up_theta();

    virtual void          do_strain_part();
    virtual bool          integrate_theta_a();
```

[1] These classes are derived from M_B_SD because The input for `ELASTCITY` is redefined.

```
    virtual bool          integrate_theta_b();
    virtual bool          integrate_runge();
    virtual int           calc_material(double fract);
        void    error(int id,...)const;
        MAT_DATA    *m_mdat;
  const VECTOR      *m_delta_grad;
        SMATRIX     *m_tg_matrix;
public:
                NL_M_TLE_B_SD(ASCII_FILE& file, int prob_size);
      FEM_IDENTIFY3(NL_M_TLE_B_SD,M_B_SD,RUNGE_INTEGRATOR,THETA_INTEGRATOR);

      virtual   ~NL_M_TLE_B_SD();
      static     NL_M_TLE_B_SD* read(ASCII_FILE& file, STRING& typ, int dim);
     void       default_integration();
     bool       allowed_integration(PTR0<LOCAL_INTEGRATION>&);
  virtual bool  integrate(MAT_DATA& mdat,const VECTOR& delta_grad, SMATRIX& tg_matrix);
};
```

protected member data

general data

psz	problem size (symmetric tensor size)
tot_sz	total internal variable storage size
rank_of_temperature	position of the temperature in the parameter vector. Temperature variables should not be of use if the base calc_material has been called
m_last_temp	storage used to prevent recalculation of coefficients
m_glob	class to hadle treatment of the elastic stress-strain relationship. Possibility of energy-based damage models to be included here. Replaces elasticity here.
m_mdat	history data for integration. this pointer is attached to the structure passed in `integrate` so that we have access everywhere in the class.
m_delta_grad	similar function for the `delta_grad` vector.
m_tg_matrix	similar function for `tg_matrix`.

integration method related data

m_intrk	Runge Kutta integration
m_inttn	Theta Newton integration
m_detot	tensor used to store the imposed strain increment or rate. acts as an interface between the strain handling functions of NL_M_TLE_B_SD and the integrating functions in derived classes
m_f_vec	holds the residual for a theta method or the internal variable rates for runge-kutta. it is handled such that sub elements attached to it are kept for the duration of the problem.
m_d_chi	internal variable rate increment in theta-method. Same storage as the `var_int` vector.
m_f_0	vector to hold the theta-system result or Runge-Kutta internal variable rates. For Runge-kutta storage is the same as in `var_int`.
m_f_grad	Jacobian matrix for the theta method.

protected member functions

set_up_theta	function called if a theta method is chosen. Default function gives an error so the actual implementation of the theta method is not forced on a model.
XX	XX
XX	XX
error	function used to issue error messages

Model Use:

The derived class VISCO[2] presents an implementation of classical viscoplastic behavior based on the the NL_M_TLE_B_SD class. That implementation can be taken as a starting point for rapidly implementing new non-linear material laws.

The integrate function tags the MAT_DATA structure as well as the tangent matrix and imposed gradient so they can be accessed from other member functions. There should probably not be any reason to redefine this function for new nonlinear behaviors.

The functions integrate_runge, integrate_theta_a, and integrate_theta_b provide pre-treatment for their namesake integration methods. These functions call do_strain_part in turn for the calculation of inelastic strain tensors. Provided the elastic strain tensor composes the first elements in the var_int vector, these functions should be valid for derived behaviors.

[2]not yet implemented

GEN_NL_M_B_SD

GENeralized NonLinear Mechanical Behavior for Small Displacement.
— Class is derived from : **NL_M_TLE_B_SD**

This behavior implements classical multi-potential plastic / viscoplastic behavior with arbitrary potential definitions without limit in number of potentials. The integration is thus carried out first for the elastic strain evolution, followed by the evolution of internal variables in the dissipation potentials (see class `POTENTIAL` definition). The treatment of stress-elastic strain relations are handled by the classes `GLOBAL_HANDLER` as inclrperated in `NL_M_TLE_B_SD` .

The integration vector is assembled automatically with sub elements defined during the calling of a setup function such as `setup_theta`. We can expect in the future that conditions may be set so that during a calculation the internal variable vector can be rearranged in order to add or subtract newly active and inactive dissipation potentials. The present assembly of the integation vector follows the form:

$$[\boldsymbol{\epsilon}_{el}\dots][v_1\ \mathbf{h}_1][v_2\ \mathbf{h}_2]\dots[v_n\ \mathbf{h}_n] \tag{1}$$

where the first element contains the elastic strain and any additional "global" variables. Such global variables are treated by the class `HANDLER` discussed further on. All the elements defined in the list of `POTENTIAL`s are stored with their cumulated multiplyer followd by the sub vector of internal variables. The `POTENTIAL` classes are unaware of this structure.

The internal; variable vector is complemented by the auxilary variable vector used to store parameters which can be calculated directly at the end of the time step. This vector is stored in the following fashon:

$$[\boldsymbol{\epsilon}_i n][\boldsymbol{\epsilon}_1\dots][\boldsymbol{\epsilon}_2\dots]\dots[\boldsymbol{\epsilon}_n\dots] \tag{2}$$

At the time we have only one inelastic potential, the term $\boldsymbol{\epsilon}_1$ is dropped, leaving only those additional parameters required for that potential.

For an integration by Runge-Kutta, the integrated variables are restricted to the following model form:

$$\dot{\boldsymbol{\epsilon}}_{el} + \dot{\epsilon}_i = \dot{\boldsymbol{\epsilon}}_{tot} - \dot{\boldsymbol{\epsilon}}_{th} \tag{3}$$

$$\dot{v}_i = \dot{\lambda}_i(\mathbf{p}, f) \tag{4}$$

$$\dot{\mathbf{h}}_i = \dot{v}\ \mathbf{m}_i(\mathbf{p}\boldsymbol{\sigma},\ \mathbf{H}_i) - \dot{\omega}_i(\mathbf{H}_i) \tag{5}$$

where the indicies are summed through the list of potentials selected by the user. Time independant potentials are placed at the end of the in order to calculate the plastic multiplyer calculated in function `derivative` (see the derived classes for the specific implementation).

$$\mathbf{H}_i = \mathbf{H}_{o_i}(\mathbf{h}_i) + \mathbf{M}_{ij}(\mathbf{p}, \mathbf{h}_i) \cdot \mathbf{h}_j \tag{6}$$

The integration by the theta method is carried out using an incremental version of the rate equations 3-4. These equations define the residual vector:

$$\Delta\boldsymbol{\epsilon}_{el} + \Delta\lambda_i \mathbf{n}_i = \Im_{el} \tag{7}$$

$$\Delta v_i - \Delta t \cdot \dot{\lambda}_i(\mathbf{p}, f) = \Im_{v_i} \tag{8}$$

$$\Delta\mathbf{h}_i - \Delta v_i \cdot \mathbf{m}_i(\mathbf{p}, \boldsymbol{\sigma}, \mathbf{H}) - \Delta t \cdot \dot{\boldsymbol{\omega}}(\mathbf{H}) = \Im_{h_i} \tag{9}$$

protected member data

xx	xx
xx	xx
xx	xx
xx	xx

LINEAR_VISCOELASTIC

This class derives from M_B_SD.

protected member data

K0 PTR<COEFFICIENT> K0
K_0 coefficient

K_inf PTR<COEFFICIENT> K_inf
K_∞ coefficient

G0 PTR<COEFFICIENT> G0
G_0 coefficient

G_inf PTR<COEFFICIENT> G_inf
G_∞ coefficient

shear LIST<SHEAR*> shear
list of all shear components

volumic LIST<VOLUMIC*> volumic
list of all volumetric components

protected member class

SHEAR class SHEAR ...
member class used to define shear components

VOLUMIC class VOLUMIC ...
member class used to define volumetric components

protected member functions

error void error(int id,...)const
function used to issue error messages

public member functions

LINEAR_VISCOELASTIC LINEAR_VISCOELASTIC(ASCII_FILE& file,int dim)
creator, calls M_B_SD::M_B_SD

~LINEAR_VISCOELASTIC ~LINEAR_VISCOELASTIC()
destructor

var_int_size virtual int var_int_size(int dimen) const
cf. BEHAVIOR, (1+!shear)*TENSOR2::give_symmetric_size(dimen) + !volumic

var_aux_size virtual int var_aux_size(int dimen) const
cf. BEHAVIOR, return 1

init_var_int_name virtual void init_var_int_name(int dimen)
cf. BEHAVIOR

init_var_aux_name virtual void init_var_aux_name(int dimen)
cf. BEHAVIOR

nb_var_in_default_output virtual int nb_var_in_default_output(int dimen)const
cf. BEHAVIOR, 2*TENSOR2::give_symmetric_size(dimen)

default_output virtual ARRAY<STRING> default_output(int dimen) const
cf. BEHAVIOR, sig and eto

integrate virtual bool integrate(MAT_DATA&,const VECTOR&, SMATRIX&)
cf. BEHAVIOR

LINEAR_VISCOELASTIC::SHEAR

LINEAR_VISCOELASTIC member class used to store data relative to shear deformation mechanisms.

private member functions

tau PTR<COEFFICIENT> tau
τ coefficient

omega PTR<COEFFICIENT> omega
ω coefficient

private member functions

SHEAR SHEAR(ASCII_FILE&,BEHAVIOR*)
constructor, BEHAVIOR* is the pointer to the behavior which used that SHEAR

error void error(int id,...) const
error function

friend class

LINEAR_VISCOELASTIC

LINEAR_VISCOELASTIC::VOLUMIC

LINEAR_VISCOELASTIC member class used to store data relative to shear deformation mechanisms.

private member functions

tau PTR<COEFFICIENT> tau
τ coefficient

omega PTR<COEFFICIENT> omega
ω coefficient

private member functions

VOLUMIC VOLUMIC(ASCII_FILE&,BEHAVIOR*)
constructor, BEHAVIOR* is the pointer to the behavior which used that VOLUMIC

error void error(int id,...) const
error function

friend class

LINEAR_VISCOELASTIC

POROUS_PLASTIC

Implementing constitutive equations for porous materials

•Equivalent stress

The equivalent stress $\sigma_\star$ is supposed to be explicitly or implicitly defined by the following equation:

$$\phi(\underset{\sim}{\sigma}, f, \sigma_\star) = 0$$

where $\underset{\sim}{\sigma}$ is the stress tensor and f the porosity.

•Internal variables

The internal variables are:
— the elastic deformation tensor $\underset{\sim}{\epsilon}_e$
— the isotropic hardening variable p
— the porosity f (f_n for crack like nucleation)
The internal variables are related to the forces by:

$$\underset{\sim}{\epsilon}_e = \overline{\overline{C}}\underset{\sim}{\sigma}$$

where $\underset{\sim}{\sigma}$ is the stress tensor.

•External variables

The external variables are:
— the equivalent plastic deformation $p^\star = \int \dot{\lambda}\,\mathrm{d}t$

•Interaction matrix
It is possible to define an interaction matrix $\mathcal{I}$ between internal/external variables and forces:

$$\begin{pmatrix} \underset{\sim}{\sigma} \\ p \\ f \\ \Delta\lambda \end{pmatrix} = \mathcal{I} \begin{pmatrix} \underset{\sim}{\epsilon}_e \\ p \\ f \\ \Delta\lambda \end{pmatrix}$$

with

$$\mathcal{I} = \begin{pmatrix} \overline{\overline{C}} & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix}$$

•θ–method: residuals

■ The total deformation increment $\Delta\underset{\sim}{\epsilon}$ is imposed. Strain partitioning gives:

$$\dot{\underset{\sim}{\epsilon}}_e + \dot{\underset{\sim}{\epsilon}}_p = \dot{\underset{\sim}{\epsilon}} - \dot{\underset{\sim}{\epsilon}}_{th}$$

where $\underset{\sim}{\epsilon}_{th}$ is the thermal deformation. The normality rule gives:

$$\dot{\underset{\sim}{\epsilon}}_p = (1-f)\dot{\lambda}\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}}$$

Integrating the preceeding equation gives:

$$R_e = \Delta\underset{\sim}{\epsilon}_e + \Delta\lambda(1-f)\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}} = \Delta\underset{\sim}{\epsilon} - \Delta\underset{\sim}{\epsilon}_{th}$$

■ The evolution of the porosity is simply given by mass conversation:

$$\dot{f} = (1-f)\text{Trace}\left(\dot{\underset{\sim}{\varepsilon}}_p\right)$$

or after integration

$$R_f = \Delta f - (1-f)^2 \Delta\lambda \text{Trace}\left(\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}}\right)$$

■ The evolution of the plastic strain p is given by (no recovery) the condition that the plastic work of the matrix material is equal to the plastic work of the homogeneous material:

$$(1-f)\sigma_\star \dot{p} = \dot{\underset{\sim}{\varepsilon}}_p : \underset{\sim}{\sigma} = (1-f)\dot{\lambda}\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}} : \underset{\sim}{\sigma}$$

The plastic multiplier is therefore given by:

$$\dot{\lambda} = \frac{\dot{p}}{\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}} : \frac{\underset{\sim}{\sigma}}{\sigma_\star}}$$

$\dot{p}$ is given by the flow law of the matrix material:

$$\dot{p} = F(\sigma_\star - R)$$

The corresponding residuals are therefore:

$$\Delta\lambda = \frac{\Delta p}{\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}} \frac{\underset{\sim}{\sigma}}{\sigma_\star}}$$

and

$$\Delta p = F(\sigma_\star - R)\Delta t$$

•θ–method: tangent matrix

In order to use the θ–method it is necessary to have the partial derivatives of the preceeding residuals (R_e, R_f, R_p and R_λ) with respect to the "associated" forces. Following definitions are used:

$$\begin{array}{ll} \tau & = \text{Trace}\left(\frac{\partial \phi}{\partial \underset{\sim}{\sigma}}\right) \\ h & = \left(\frac{\partial \phi}{\partial \sigma_\star}\right)^{-1} \\ \omega & = \sigma_\star - R \ \text{(overstress)} \end{array}$$

Note that

$$\frac{\partial h}{\partial X} = -h^2 \frac{\partial^2 \phi}{\partial \sigma_\star \partial X}$$

$\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}}$ is calculated as follows. For f =constant and $\phi = 0$, one has:

$$\delta\phi = \frac{\partial \phi}{\partial \underset{\sim}{\sigma}}\delta\underset{\sim}{\sigma} + \frac{\partial \phi}{\partial \sigma_\star}\delta\sigma_\star = 0$$

or

$$\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}}\,|_{f,\ \phi=0} = -h\frac{\partial \phi}{\partial \underset{\sim}{\sigma}}$$

More generally one has (using the same arguments)

$$\frac{\partial \sigma_\star}{\partial x} = -h\frac{\partial \phi}{\partial x}$$

Residuals can be rewritten as follows:

$$R = R^1(\Delta v) + R^2(F) = R^0$$

with

$$R^1 = \begin{pmatrix} \Delta \underset{\sim}{\epsilon}_e \\ \Delta p \\ \Delta f \\ K\Delta\lambda \end{pmatrix} \quad R^2 = \begin{pmatrix} -(1-f)\Delta\lambda h\dfrac{\partial \phi}{\partial \underset{\sim}{\sigma}} \\ -F(\sigma_\star - R)\Delta t \\ \Delta\lambda h(1-f)^2\tau \\ -\dfrac{\Delta p}{\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}} \underset{\sim}{\sigma}_\star} \end{pmatrix} \quad R^0 = \begin{pmatrix} \Delta \underset{\sim}{\epsilon} - \Delta \underset{\sim}{\epsilon}_{th} \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$\dfrac{\partial R}{\partial \Delta v}$ is calculated as:

$$\frac{\partial R^1}{\partial \Delta v} = \begin{pmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & \\ & & & K \end{pmatrix}$$

and

$$\frac{\partial R^2}{\partial \Delta v} = \frac{\partial R^2}{\partial F}\frac{\partial F}{\partial v}\frac{\partial v}{\partial \Delta v}$$

on has:

$$\frac{\partial F}{\partial v} = \mathcal{I} \qquad \frac{\partial v}{\partial \Delta v} = \begin{pmatrix} \theta & & & \\ & \ddots & & \\ & & \theta & \\ & & & 1 \end{pmatrix}$$

$\dfrac{\partial R^2}{\partial F}$ has then to be calculated.

$$\frac{\partial R^2_e}{\partial \underset{\sim}{\sigma}} = (1-f)h\Delta\lambda\left[h\frac{\partial^2\phi}{\partial \underset{\sim}{\sigma}\partial\sigma_\star} \otimes \frac{\partial \phi}{\partial \underset{\sim}{\sigma}} - \frac{\partial^2\phi}{\partial \underset{\sim}{\sigma}^2}\right]$$

$$\frac{\partial R^2_e}{\partial f} = \Delta\lambda h\left[\left[1 + (1-f)h\frac{\partial^2\phi}{\partial\sigma_\star \partial f}\right]\frac{\partial \phi}{\partial \underset{\sim}{\sigma}} - (1-f)\frac{\partial^2\phi}{\partial \underset{\sim}{\sigma}\partial f}\right]$$

$$\frac{\partial R^2_e}{\partial p} = 0$$

$$\frac{\partial R^2_e}{\partial \Delta\lambda} = -(1-f)h\frac{\partial \phi}{\partial \underset{\sim}{\sigma}}$$

$$\frac{\partial R^2_p}{\partial \underset{\sim}{\sigma}} = F'(\omega)h\frac{\partial \phi}{\partial \underset{\sim}{\sigma}}\Delta t$$

$$\frac{\partial R^2_p}{\partial p} = F'(\omega)R'_\theta\Delta t - \frac{\partial F}{\partial p}\Delta t$$

$$\frac{\partial R^2_p}{\partial f} = F'(\omega)h\frac{\partial \phi}{\partial f}\Delta t$$

$$\frac{\partial R_p^2}{\partial \Delta\lambda} = 0 \text{ with } F'(\omega) = \frac{\partial F}{\partial \omega}.$$

$$\frac{\partial R_f^2}{\partial \underset{\sim}{\sigma}} = (1-f)^2 \Delta\lambda h \left[\underset{\sim}{\mathbf{1}} \frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}^2} - h\tau \frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma} \partial \sigma_\star} \right]$$
$$\frac{\partial R_f^2}{\partial p} = 0$$
$$\frac{\partial R_f^2}{\partial f} = (1-f)\Delta\lambda h \left[-2\tau - (1-f)\tau h \frac{\partial^2 \phi}{\partial \sigma_\star \partial f} + (1-f)\text{Trace}\left(\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma} \partial f} \right) \right]$$
$$\frac{\partial R_f^2}{\partial \Delta\lambda} = (1-f)^2 h\tau$$

One assumes that $h \frac{\partial \phi}{\partial \underset{\sim}{\sigma}} : \frac{\underset{\sim}{\sigma}}{\sigma_\star}$ varies very slowly compared to other variables. In the case of the elliptic and Von Mises (i.e. all potentials for f close to 0) potentials this quantity is always equal to 1.

$$\frac{\partial R_\lambda^2}{\partial \underset{\sim}{\sigma}} = 0$$
$$\frac{\partial R_\lambda^2}{\partial p} = -\frac{1}{h \frac{\partial \phi}{\partial \underset{\sim}{\sigma}} : \frac{\underset{\sim}{\sigma}}{\sigma_\star}}$$
$$\frac{\partial R_\lambda^2}{\partial f} = 0$$
$$\frac{\partial R_\lambda^2}{\partial \Delta\lambda} = 0$$

•Adding strain controlled nucleation

$\dot{f}$ is now given by:

$$\dot{f} = -(1-f)^2 h \dot{\lambda} \tau + \dot{f}_n$$
$$\dot{f}_n = A(p, \dots)\dot{p}$$

the corresponding residual R_f is now:

$$R_f = \Delta f + (1-f)^2 h\tau \Delta\lambda - A(p, \dots)\Delta p$$

Partial derivatives are modified as follows:
Add to $\partial R_f^2 / \partial p$

$$-\left[A + \frac{\partial A}{\partial p} \theta \Delta p \right] \tag{10}$$

since p at θ is equal to $p_0 + \theta\Delta\lambda$, where p_0 is the value of p at the beginning of the increment.

•Adding stress controlled nucleation

Ask for it and propose tests if you want it!

•Adding sintering strain

The plastic strain rate is modified as follows:

$$\dot{\underset{\sim}{\epsilon}}_p = \dot{\lambda}(1-f)\frac{\partial \sigma_\star}{\partial \underset{\sim}{\sigma}} + \dot{\underset{\sim}{\epsilon}}_s(f, \dots)$$

where $\dot{\underset{\sim}{\epsilon}}_s$ represents the sintering strain. Other constitutive equations are not modified.

R_e is now expressed as:

$$R_e = \Delta\epsilon_e - \Delta\lambda h(1-f)\frac{\partial\phi}{\partial\underset{\sim}{\sigma}} + \dot{\underset{\sim}{\epsilon}}_s\Delta t$$

and R_f as

$$R_f = \Delta f + (1-f)^2\Delta\lambda h\tau - (1-f)\Delta t\text{Trace}\left(\dot{\underset{\sim}{\epsilon}}_s\right)$$

The following terms are added to the partial derivatives:
to $\partial R_e^2/\partial f$

$$\Delta t\frac{\partial\dot{\underset{\sim}{\epsilon}}_s}{\partial f}$$

to $\partial R_f^2/\partial f$

$$\Delta t\left[\text{Trace}\left(\dot{\underset{\sim}{\epsilon}}_s\right) - (1-f)\text{Trace}\left(\frac{\partial\dot{\underset{\sim}{\epsilon}}_s}{\partial f}\right)\right]$$

•Adding sintering stess
Ask for it and propose tests if you want it!
•Adding several nucleation/sintering mechanisms
Several nucleation/sintering mechanisms can be added without any problem since the preceeding equations are linear.

• Crack like nucleation

In some cases nucleation does not result from inclusion/matrix debonding (which results in the formation of voids so that the volum effectively occupied by the material decreases) but in cracks which weaken the material without generating actual voids. Indeed these cracks will grow as deformation increases and finally create voids. In order to represent this phenomenon a new internal variable f_n has to be introduced in the model. It represents the damage due to cracks. The evolution of f_n is depicted by strain/stress nucleation, so that:

$$\Delta f_n = A(\dots)\Delta p$$

The equivalent stress is defined by:

$$\phi(\underset{\sim}{\sigma}, f + f_n, \sigma_\star)$$

Since of actual fraction of material in a unit volume is still $(1-f)$, the plastic strain rate is still:

$$\dot{\underset{\sim}{\epsilon}}_p = \dot{\lambda}(1-f)\frac{\partial\sigma_\star}{\partial\underset{\sim}{\sigma}}$$

An other residual equation has to be added to the system:

$$R_{f_n} = \Delta f_n - \Delta p\sum A(\dots)$$

Partial derivative of $R_{f_n}^2$ are given by eq. [D2fdp]

It is also necessary to calculate the partial derivative of the other residuals with respect to f_n.

$$\frac{\partial R_e^2}{\partial f_n} = \Delta\lambda(1-f)h\left[h\frac{\partial^2\phi}{\partial\sigma_\star\partial f}\otimes\frac{\partial\phi}{\partial\underset{\sim}{\sigma}} - \frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial f}\right]$$

$$\frac{\partial R_p^2}{\partial f_n} = \frac{\partial R_p^2}{\partial f}$$

$$\frac{\partial R_f^2}{\partial f_n} = (1-f)^2\Delta\lambda h\left[-\tau h\frac{\partial^2\phi}{\partial\sigma_\star^2}f + \text{Trace}\left(\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial f}\right)\right]$$

$$\frac{\partial R_\lambda^2}{\partial f_n} = 0$$

Once again several nucleation can be added both depicting decohesion and cracking.

• Kinematic Hardening (Not yet implemented)

Kinematic hardening has been introduced in porous materials using differents methods (see Needleman or Leblond). Another method is used here; it is based on a simple extension of the kinematic (linear and non—linear) hardening model of Lemaitre & Chaboche

— Base model ($f = 0$)

The dissipation potential Ψ is introduced as follows (linear hardening corresponds to $D/C = 0$)

$$\Psi = \sqrt{\frac{3}{2}(\sigma - X)' : (\sigma - X)'} - k + \frac{3}{4}\frac{D}{C}X : X + \dots$$

$$X = \frac{2}{3}C\alpha$$

$$\dot{\alpha} = -\dot{\lambda}\frac{\partial \Psi}{\partial X} = \dot{\epsilon}_p - \frac{3}{2}\dot{\lambda}\frac{D}{C}X = \dot{\lambda}\left(n - \frac{3}{2}\frac{D}{C}X\right) \tag{11}$$

— Modified model ($f \neq 0$)

The dissipation potential is now written as

$$\Psi = \sigma_* - k + \mu\frac{D}{C}X_*^2 + \dots$$

where X_* is defined by

$$\phi(X, X_*, f) = 0$$

The linear relation between X and α is given by

$$X = H\left(\frac{2}{3}C\alpha\right)$$

where H is a fourth order tensor depending on f with $H(0) = 1$. H is defined for each potential. The evolution of α is determined by

$$\begin{aligned}\dot{\alpha} &= -\dot{\lambda}(1-f)\frac{\partial \Psi}{\partial X} = -\dot{\lambda}(1-f)\left[\left.\frac{\partial \sigma_*}{\partial \sigma}\right|_{\sigma - X} + 2\mu\frac{D}{C}X_*\left.\frac{\partial X_*}{\partial X}\right|_X\right] \\ &= \dot{\epsilon}_p - 2\dot{\lambda}(1-f)\mu\frac{D}{C}X_*\left.\frac{\partial X_*}{\partial X}\right|_X\end{aligned}$$

this formula must be equivalent to eq. 11 for $f = 0$. In that case $X_* = \sqrt{\frac{3}{2}X : X}$ $(Tr\,(X) = 0)$ and $\frac{\partial X_*}{\partial X} = \frac{3}{2}\frac{X}{X_*}$, so that

$$\dot{\alpha} = \dot{\epsilon}_p - 3\dot{\lambda}\mu\frac{D}{C}X \Rightarrow \mu = \frac{1}{2}$$

One finally gets

$$\begin{aligned}\Psi &= \sigma_* - k + \frac{D}{2C}X_*^2 + \dots \\ \dot{\alpha} &= \dot{\epsilon}_p - \dot{\lambda}(1-f)\frac{D}{C}X_*\frac{\partial X_*}{\partial X}\end{aligned}$$

In order to have an expression similar to eq. 11 one can define an intermediate stress tensor χ as

$$\chi = \frac{2}{3} X_* \frac{\partial X_*}{\partial X} \quad \dot{\alpha} = \dot{\epsilon}_p - \dot{\lambda}\frac{3}{2}(1-f)\frac{D}{C}\chi = (1-f)\dot{\lambda}\left(n - \frac{3}{2}\frac{D}{C}\chi\right)$$

- **Adiabatic Heating**

Addition of adiabatic heating is simply done by adding a new internal variable: the temperature T. The evolution of T is related to the plastic dissipation by the relation:

$$C_p \dot{T} = \beta \dot{\underset{\sim}{\epsilon}}_p : \underset{\sim}{\sigma}$$

where C_p^P is the heat capacity of the porous material. Indeed C_p^P is related to the heat capacity of the dense material C_p^D by

$$C_p^P = (1-f) C_p^D$$

Some porous plastic criterion and their derivatives

The following matrices will be used:

$$\overline{\overline{V}} = \begin{pmatrix} 2/3 & -1/3 & -1/3 & & \\ -1/3 & 2/3 & -1/3 & 0 & \\ -1/3 & -1/3 & 2/3 & & \\ & & & 1 & \\ & 0 & & & 1 \\ & & & & & 1 \end{pmatrix} \qquad \overline{\overline{U}} = \begin{pmatrix} 1 & 1 & 1 & & \\ 1 & 1 & 1 & 0 & \\ 1 & 1 & 1 & & \\ & & & & \\ & 0 & & 0 & \\ & & & & \end{pmatrix}$$

In the expression of the porous potential

$$\phi(\underset{\sim}{\sigma}, f, \sigma_\star) = 0$$

, $\underset{\sim}{\sigma}$ is function of J_2, the second invariant of the stress deviator tensor, and I_1, the first invariant of the stress tensor :

$$\begin{array}{ll} I_1 & = \text{Trace}\left(\underset{\sim}{\sigma}\right) \\ \underset{\sim}{\mathbf{s}} & = \underset{\sim}{\sigma} - \frac{I_1}{3}\underset{\sim}{\mathbf{1}} \\ J_2 & = \frac{1}{2}\underset{\sim}{\mathbf{s}} : \underset{\sim}{\mathbf{s}} \end{array}$$

In order to extend the application of porous criteria to anisotropic behaviour, it is more adequate to express $\underset{\sim}{\sigma}$ as a function of σ_E and σ_M defined as below :

$$\begin{array}{ll} \sigma_M & = \text{Trace}\left(\underset{\sim}{\sigma}\right) \\ \sigma_E & = \sqrt{\frac{3}{2}\underset{\sim}{\mathbf{s}} : H : \underset{\sim}{\mathbf{s}}} \end{array}$$

so that anisotropy of the material behavior is accounted for within the Hill tensor H.
Note that:

$$\sigma_E = \sqrt{3J_2}$$

in isotropic case.
Derivatives of the potential can be expressed as follows :
$\frac{\partial \phi}{\partial \underset{\sim}{\sigma}} = \frac{\partial \phi}{\partial \sigma_E}\underset{\sim}{\mathbf{n}} + \frac{\partial \phi}{\partial \sigma_M}\ \frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}^2} =$

• Gurson criterion

The potential is given by

$$\phi = \frac{\sigma_E^2}{\sigma_\star^2} + 2f^\star q_1 \cosh\left(\frac{q_2 sigma_M}{2\sigma_\star}\right) - (1 + q_1^2 f^{\star 2})$$

where

$$A = \frac{q_2 I_1}{2\sigma_\star}$$

$$\frac{\partial \phi}{\partial \underset{\sim}{\sigma}} = \frac{3\underset{\sim}{\mathbf{s}}}{\sigma_\star^2} + \frac{f^\star q_1 q_2}{\sigma_\star}\sinh(A)\underset{\sim}{\mathbf{1}}$$

$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}^2} = \frac{3}{\sigma_\star^2}\overline{\overline{V}} + \frac{f^\star q_1 q_2^2}{2\sigma_\star^2}\cosh(A)\overline{\overline{U}}$$

$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma} \partial f} = f^{\star\prime}\frac{q_1 q_2}{\sigma_\star}\sinh(A)\underset{\sim}{\mathbf{1}}$$

$$\frac{\partial \phi}{\partial \sigma_\star} = -\frac{6J_2}{\sigma_\star^3} - \frac{f^\star q_1 q_2 I_1}{\sigma_\star^2}\sinh(A)$$
$$\frac{\partial \phi}{\partial f} = 2f^{\star\prime} q_1(\cosh(A) - q_1 f^\star)$$
$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}\partial \sigma_\star} = -\frac{6\underset{\sim}{\mathbf{s}}}{\sigma_\star^3} - \frac{f^\star q_1 q_2}{\sigma_\star^2}\left(\sinh(A) + \frac{q_2 I_1}{2\sigma_\star}\cosh(A)\right)\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial^2 \phi}{\partial \sigma_\star \partial f} = -\frac{q_1 q_2 I_1}{\sigma_\star^2} f^{\star\prime}\sinh(A)$$

The effective equivalent stress has to be calculated numerically by solving $\phi = 0$ by using an iterative method (either Newton–Raphson or dichotomy). The following value will be used as initial value for the iterative process:

$$\sigma_\star \approx \frac{\left[3J_2 + q_1 f^\star (q_2 I_1/2)^2\right]^{1/2}}{1 - q_1 f^\star}$$

- **Elliptic criterion**

The potential is given by:

$$\phi = 3CJ_2 + FI_1^2 - \sigma_\star^2$$

where F and C are two functions of the porosity f.

$$\frac{\partial \phi}{\partial \underset{\sim}{\sigma}} = 3C\underset{\sim}{\mathbf{s}} + 2FI_1\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}^2} = 3C\overline{\overline{V}} + 2F\overline{\overline{U}}$$
$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}\partial f} = 3C'\underset{\sim}{\mathbf{s}} + 2F'I_1\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial \phi}{\partial \sigma_\star} = -2\sigma_\star$$
$$\frac{\partial \phi}{\partial f} = 3C'J_2 + F'I_1^2$$
$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}\partial \sigma_\star} = \underset{\sim}{\mathbf{0}}$$
$$\frac{\partial^2 \phi}{\partial \sigma_\star \partial f} = 0$$

- **Rousselier criterion**

The potential is given by:

$$\phi = \frac{\sqrt{3J_2}}{1-f} + \sigma_1 f D\exp\left(\frac{I_1}{3(1-f)\sigma_1}\right) - \sigma_\star$$

where σ_1 and D are parameters. With

$$A = \frac{I_1}{3(1-f)\sigma_1}$$

$$\frac{\partial \phi}{\partial \underset{\sim}{\sigma}} = \frac{1}{1-f}\frac{\sqrt{3}}{2}\frac{\underset{\sim}{\mathbf{s}}}{\sqrt{J_2}} + \frac{fD\exp(A)}{3(1-f)}\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma}^2} = \frac{1}{1-f}\frac{\sqrt{3}}{2}\left[\frac{\overline{\overline{V}}}{\sqrt{J_2}} - \frac{1}{2}J_2^{-3/2}\underset{\sim}{\mathbf{s}}\otimes\underset{\sim}{\mathbf{s}}\right] + \frac{fD\exp(A)}{9(1-f)^2\sigma_1}\overline{\overline{U}}$$

$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial f} = \frac{1}{(1-f)^2}\frac{\sqrt{3}}{2}\frac{\underset{\sim}{\mathbf{s}}}{\sqrt{J_2}} + \frac{D\exp(A)}{3(1-f)}\left[1+\frac{f}{1-f}+\frac{fI_1}{3(1-f)^2\sigma_1}\right]\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial\phi}{\partial\sigma_\star} = -1$$
$$\frac{\partial\phi}{\partial f} = \frac{\sqrt{3J_2}}{(1-f)^2} + \sigma_1 D\exp(A)\left(1+\frac{fI_1}{3(1-f)^2\sigma_1}\right)$$
$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial\sigma_\star} = \underset{\sim}{\mathbf{0}}$$
$$\frac{\partial^2\phi}{\partial\sigma_\star\partial f} = 0$$

• Modified Rousselier criterion

The potential is given by:

$$\phi = \frac{\sqrt{3J_2}}{1-f} + \sigma_1 f D\exp\left(\frac{q_2 I_1}{3(1-f)\sigma_\star}\right) - \sigma_\star$$

where q_2 and D are parameters. With

$$A = \frac{q_2 I_1}{3(1-f)\sigma_\star}$$

$$\frac{\partial\phi}{\partial\underset{\sim}{\sigma}} = \frac{1}{1-f}\frac{\sqrt{3}}{2}\frac{\underset{\sim}{\mathbf{s}}}{\sqrt{J_2}} + \frac{fD\exp(A)\sigma_1 q_2}{3(1-f)\sigma_\star}\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}^2} = \frac{1}{1-f}\frac{\sqrt{3}}{2}\left[\frac{\overline{\overline{V}}}{\sqrt{J_2}} - \frac{1}{2}J_2^{-3/2}\underset{\sim}{\mathbf{s}}\otimes\underset{\sim}{\mathbf{s}}\right] + \frac{fDq_2^2\sigma_1\exp(A)}{9(1-f)^2\sigma_\star^2}\overline{\overline{U}}$$
$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial f} = \frac{1}{(1-f)^2}\frac{\sqrt{3}}{2}\frac{\underset{\sim}{\mathbf{s}}}{\sqrt{J_2}} + \frac{Dq_2\sigma_1\exp(A)}{3(1-f)\sigma_\star}\left[1+\frac{f}{1-f}+\frac{fq_2I_1}{3(1-f)^2\sigma_\star}\right]\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial\phi}{\partial\sigma_\star} = -\frac{\sigma_1 fDq_2I_1\exp(A)}{3(1-f)\sigma_\star^2} - 1$$
$$\frac{\partial\phi}{\partial f} = \frac{\sqrt{3J_2}}{(1-f)^2} + \sigma_1 D\exp(A)\left(1+\frac{q_2 fI_1}{3(1-f)^2\sigma_\star}\right)$$
$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial\sigma_\star} = -\frac{fD\exp(A)q_2\sigma_1}{3(1-f)\sigma_\star^2}\left[1+\frac{q_2I_1}{3(1-f)\sigma_\star}\right]\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial^2\phi}{\partial\sigma_\star\partial f} = -\frac{\sigma_1 Dq_2I_1\exp(A)}{3\sigma_\star^2(1-f)}\left[1+\frac{f}{1-f}+\frac{fq_2I_1}{3(1-f)^2\sigma_\star}\right]$$

• Cam–clay criterion

The potential is given by:

$$\phi = \frac{3J_2}{m^2} + \left(\frac{I_1}{3}+p_c\right)^2 - p_c^2 - \sigma_\star^2$$

$$\frac{\partial\phi}{\partial\underset{\sim}{\sigma}} = \frac{3\underset{\sim}{\mathbf{s}}}{m^2} + \frac{2}{3}\left(\frac{I_1}{3}+p_c\right)\underset{\sim}{\mathbf{1}}$$
$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}^2} = \frac{3\overline{\overline{V}}}{m^2} + \frac{2}{9}\overline{\overline{U}}$$
$$\frac{\partial^2\phi}{\partial\underset{\sim}{\sigma}\partial f} = -\frac{6\underset{\sim}{\mathbf{s}}}{m^3}m' + \frac{2}{3}p_c'\underset{\sim}{\mathbf{1}}$$

$$\frac{\partial \phi}{\partial \sigma_\star} = -2\sigma_\star$$
$$\frac{\partial \phi}{\partial f} = \frac{2}{3} I_1 p'_c - \frac{6 J_2}{m^3} m'$$
$$\frac{\partial^2 \phi}{\partial \underset{\sim}{\sigma} \partial \sigma_\star} = \underset{\approx}{\mathbf{0}}$$
$$\frac{\partial^2 \phi}{\partial \sigma_\star \partial f} = 0$$

MMC

- **Mean field model**

Let σ_c and ϵ_c be the stress and the total deformation of the composite. Each phase ($i = 0, \ldots, n$) is characterized by its volume fraction c_i, its stress tensor σ_i and its deformation tensor ϵ_i. The phase 0 corresponds to the matrix. On has

$$\sigma_c = \sum_i c_i \sigma_i \qquad \epsilon_c = \sum_i c_i \epsilon_i \qquad \sum_i c_i = 1$$

The deformation of each phase can be separated into an elastic deformation and a "free" deformation (ϵ_i^l) which correspond for an elastic material to thermal dilation and phase transformation. On therefore has

$$\epsilon_i = \mathbf{C}_i^{-1} \sigma_i + \epsilon_i^l \tag{12}$$

where $\mathbf{C}_i$ represents the elasticity operator of phase (i). For each phase one defines the incompatibility deformation with respect to the matrix

$$\epsilon_i^{in} = \epsilon_i^l - \epsilon_0^l + \left(\mathbf{C}_i^{-1} - \mathbf{C}_0^{-1}\right) \sigma_i \tag{13}$$

The difference between the deformation of the matrix and of the inclusion (i) is related to the incompatibility deformation by

$$\begin{aligned} \epsilon_i - \epsilon_0 &= \mathbf{S}_i \epsilon_i^{in} \\ &= \mathbf{S}_i \left(\epsilon_i^l - \epsilon_0^l + \left(\mathbf{C}_i^{-1} - \mathbf{C}_0^{-1}\right) \sigma_i\right) \end{aligned} \tag{14}$$

where $\mathbf{S}_i$ is the Eshelby operator of phase (i). One can define $\mathbf{S}_0$ as $\mathbf{I}$. Let $\mathbf{K}_i$ be

$$\mathbf{K}_i = \left[\mathbf{C}_i^{-1} - \mathbf{S}_i \left(\mathbf{C}_i^{-1} - \mathbf{C}_0^{-1}\right)\right]^{-1} \tag{15}$$

with $\mathbf{K}_0 = \mathbf{C}_0$. Combining 12 and 14, one gets

$$\begin{aligned} \epsilon_i - \epsilon_0 &= \mathbf{C}_i^{-1} \sigma_i - \mathbf{C}_0^{-1} \sigma_0 + \epsilon_i^l - \epsilon_0^l \\ &= \mathbf{S}_i \left(\epsilon_i^l - \epsilon_0^l\right) + \mathbf{S}_i \left(\mathbf{C}_i^{-1} - \mathbf{C}_0^{-1}\right) \sigma_i \end{aligned}$$

or

$$\sigma_i = \mathbf{K}_i \mathbf{C}_0^{-1} \sigma_0 + \mathbf{K}_i \left(\mathbf{S}_i - \mathbf{I}\right) \left(\epsilon_i^l - \epsilon_0^l\right) \qquad \forall i \tag{16}$$

Letγ_i be

$$\gamma_i = \mathbf{K}_i \left(\mathbf{S}_i - \mathbf{I}\right) \left(\epsilon_i^l - \epsilon_0^l\right) \qquad \gamma_0 = 0$$

and

$$\sigma_c = \sum_i c_i \sigma_i = \left[\sum_i c_i \mathbf{K}_i\right] \mathbf{C}_0^{-1} \sigma_0 + \sum_i c_i \gamma_i$$

Let γ_c and $\mathbf{K}_c$ be

$$\gamma_c = \sum_i c_i \gamma_i \qquad \text{and} \qquad \mathbf{K}_c = \left[\sum_i c_i \mathbf{K}_i\right]$$

Then

$$\sigma_0 = \mathbf{C}_0 \mathbf{K}_c^{-1} \left(\sigma_c - \gamma_c\right)$$

Using 16, one gets for phase

$$\begin{aligned}\sigma_i &= \mathbf{K}_i\mathbf{K}_c^{-1}\left(\sigma_c-\gamma_c\right)+\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\epsilon_i^l-\epsilon_0^l\right) \\ &= \mathbf{K}_i\mathbf{K}_c^{-1}\left(\sigma_c-\gamma_c\right)+\gamma_i\end{aligned} \tag{17}$$

the deformation

$$\epsilon_i=\mathbf{C}_i^{-1}\mathbf{K}_i\mathbf{K}_c^{-1}\left(\sigma_c-\gamma_c\right)+\mathbf{C}_i^{-1}\gamma_i+\epsilon_i^l$$

The deformation of the composite is then equal to

$$\epsilon_c=\sum_i c_i\epsilon_i=\left[\sum_i c_i\mathbf{C}_i^{-1}\mathbf{K}_i\right]\mathbf{K}_c^{-1}\left(\sigma_c-\gamma_c\right)+\sum_i c_i\mathbf{C}_i^{-1}\gamma_i+\sum_i c_i\epsilon_i^l \tag{18}$$

Without "free" deformations the previous formula gives

$$\epsilon_c=\left[\sum_i c_i\mathbf{C}_i^{-1}\mathbf{K}_i\right]\mathbf{K}_c^{-1}\sigma_c$$

The elasticity operator of the composite is then equal to

$$\mathbf{C}_c=\left[\sum_i c_i\mathbf{K}_i\right]\left[\sum_i c_i\mathbf{C}_i^{-1}\mathbf{K}_i\right]^{-1}=\mathbf{K}_c\left[\sum_i c_i\mathbf{C}_i^{-1}\mathbf{K}_i\right]^{-1}$$

Considering thermal deformation as "free" deformation ($\epsilon_i^l=\Delta T\alpha_i$) in absence of external stress ($\sigma_c=0$) one gets

$$\begin{aligned}\alpha_c &= -\mathbf{C}_c^{-1}\left[\sum_i c_i\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\alpha_i-\alpha_0\right)\right] \\ &\quad+\sum_i c_i\mathbf{C}_i^{-1}\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\alpha_i-\alpha_0\right)+\sum_i c_i\alpha_i \\ &= \sum_i c_i\left[\alpha_i+\left(\mathbf{C}_i^{-1}-\mathbf{C}_c^{-1}\right)\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\alpha_i-\alpha_0\right)\right]\end{aligned}$$

- **Plasticity**

It is assumed that the thermo—elastic of the different phase are constant ($\mathbf{C}_i$ et α_i). Inclusions are elastic and the matrix has a plastic behavior. The strain rate of the composite $\dot{\epsilon}_c$ is suppose to be known (FEM formulation). The "free" deformations are then given by

$$\text{matrix}\quad \dot{\epsilon}_0^l=\dot{\epsilon}_0^p+\dot{\epsilon}_0^{th}\quad \text{inclusions}\quad \dot{\epsilon}_i^l=\dot{\epsilon}_i^{th}$$

where $\dot{\epsilon}_0^p$ plastic formation rate of the matrix and $\dot{\epsilon}_i^{th}=\dot{T}\alpha_i$. Eq. 18 are be used using the following differential formulation

$$\begin{aligned}\dot{\epsilon}_c &= \mathbf{C}_c^{-1}\left(\dot{\sigma}_c-\sum_i\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\epsilon}_i^l-\dot{\epsilon}_0^l\right)\right) \\ &\quad+\sum_i c_i\mathbf{C}_i^{-1}\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\epsilon}_i^l-\dot{\epsilon}_0^l\right)+\sum_i c_i\dot{\epsilon}_i^l\end{aligned}$$

$\dot{\sigma}_c$ can then be computed

$$\begin{aligned}\dot{\sigma}_c &= \mathbf{C}_c\left[\dot{\epsilon}_c - \sum_i c_i \mathbf{C}_i^{-1}\mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\epsilon}_i^l-\dot{\epsilon}_0^l\right) - \sum_i c_i\dot{\epsilon}_i^l\right] \\ &\quad \underbrace{+\sum_i \mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\epsilon}_i^l-\dot{\epsilon}_0^l\right)}_{\dot{\gamma}_c}\end{aligned}$$

then $\dot{\sigma}_i$

$$\dot{\sigma}_i = \mathbf{K}_i\mathbf{K}_c^{-1}\left(\dot{\sigma}_c-\dot{\gamma}_c\right) + \mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\epsilon}_i^l-\dot{\epsilon}_0^l\right)$$

and the deformation in each phase

$$\dot{\epsilon}_i = \mathbf{C}_i^{-1}\dot{\sigma}_i + \dot{\epsilon}_i^l$$

- **Modification of the plasticity formulation**

It is however well known the previous formulation leads to a strong overestimation of the stresses in the composite material. This problem can be solved using the secant (or tangent) modulus of the matrix instead of its elasticity operator $\mathbf{C}_0$. This solution cannot be used for non–proportional loading paths with are usually found in FE calculations.

A solution to this problem consists in the introduction of an intermediate internal variable β_0 equivalent to a deformation and whose evolution is given by

$$\dot{\beta}_0 = \Theta(\dot{\epsilon}_0^p, \beta_0) + \dot{\epsilon}_0^{th} \qquad \text{so that} \quad \|\beta_0\| \leq \|\epsilon_0^p\|$$

$\dot{\beta}_0$ replaces $\dot{\epsilon}_0^p$ in the previous equations. The fact that $\|\beta_0\| \leq \left\|\epsilon_0^l\right\|$ allows to reduce the incompatibilities between the matrix and the inclusions, and that consequently be used to model the stresses of the composite. Eq. (12) is still valid

$$\dot{\epsilon}_i = \mathbf{C}_i^{-1}\dot{\sigma}_i + \dot{\epsilon}_i^l$$

For each phase the incompatibility deformation rate is given by

$$\dot{\epsilon}_i^{in} = \dot{\beta}_i^l - \dot{\beta}_0^l + \left(\mathbf{C}_i^{-1}-\mathbf{C}_0^{-1}\right)\dot{\sigma}_i \tag{19}$$

With $\dot{\beta}_i^l = \dot{T}\alpha_i$ for $i \neq 0$. By analogy with eq. 14, one relates the difference the corrected deformation in the matrix and the inclusions by

$$\left(\mathbf{C}_i^{-1}\dot{\sigma}_i + \dot{\beta}_i^l\right) - \left(\mathbf{C}_0^{-1}\dot{\sigma}_0 + \dot{\beta}_0^l\right) = \mathbf{S}_i\dot{\epsilon}_i^{in} = \mathbf{S}_i\left(\dot{\beta}_i^l - \dot{\beta}_0^l + \left(\mathbf{C}_i^{-1}-\mathbf{C}_0^{-1}\right)\dot{\sigma}_i\right)$$

as before one gets

$$\dot{\sigma}_i = \mathbf{K}_i\mathbf{C}_0^{-1}\dot{\sigma}_0 + \mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\beta}_i^l-\dot{\beta}_0^l\right) \qquad \forall i$$

$\dot{\gamma}_i$ is now defined as

$$\dot{\gamma}_i = \mathbf{K}_i\left(\mathbf{S}_i-\mathbf{I}\right)\left(\dot{\beta}_i^l-\dot{\beta}_0^l\right) \qquad \dot{\gamma}_0 = 0 \qquad \dot{\gamma}_c = \sum_i c_i\dot{\gamma}_i$$

and

$$\dot{\sigma}_c = \sum_i c_i\dot{\sigma}_i = \left[\sum_i c_i\mathbf{K}_i\right]\mathbf{C}_0^{-1}\dot{\sigma}_0 + \sum_i c_i\gamma_i$$

then

$$\dot{\sigma}_i = \mathbf{K}_i \mathbf{K}_c^{-1} \left(\dot{\sigma}_c - \dot{\gamma}_c \right) + \dot{\gamma}_i \qquad i = 0, \ldots, n$$

and for deformations ($\dot{\epsilon}_0^l$ is used here)

$$\dot{\epsilon}_i = \mathbf{C}_i^{-1} \mathbf{K}_i \mathbf{K}_c^{-1} \left(\dot{\sigma}_c - \dot{\gamma}_c \right) + \mathbf{C}_i^{-1} \dot{\gamma}_i + \dot{\epsilon}_i^l$$

The deformation rate of the composite is then equal to

$$\dot{\epsilon}_c = \sum_i c_i \dot{\epsilon}_i = \left[\sum_i c_i \mathbf{C}_i^{-1} \mathbf{K}_i \right] \mathbf{K}_c^{-1} \left(\dot{\sigma}_c - \dot{\gamma}_c \right) + \sum_i c_i \mathbf{C}_i^{-1} \dot{\gamma}_i + \sum_i c_i \dot{\epsilon}_i^l$$

so that the stress rate is given by

$$\dot{\sigma}_c = \mathbf{K}_c \left[\sum_i c_i \mathbf{C}_i^{-1} \mathbf{K}_i \right]^{-1} \left(\dot{\epsilon}_c - \sum_i c_i \mathbf{C}_i^{-1} \dot{\gamma}_i - \sum_i c_i \dot{\epsilon}_i^l \right) \tag{20}$$

- **Implementation**

  - Internal variables

Internal variables representative of the materials state are

$$V = \left[\left[\beta_0 \right], \left[\epsilon_0^e, h_0 \right], \left[\epsilon_1^e \right], \ldots, \left[\epsilon_n^e \right] \right]$$

where ϵ_i^e represents the elastic deformation of each phase and h_0 "hardening" variables of the matrix. These can include isotropic and kinematic hardening but also damage variables.

- Integration

Integration of the constitutive equations will be done using a Runge—Kutta method according to the following steps.

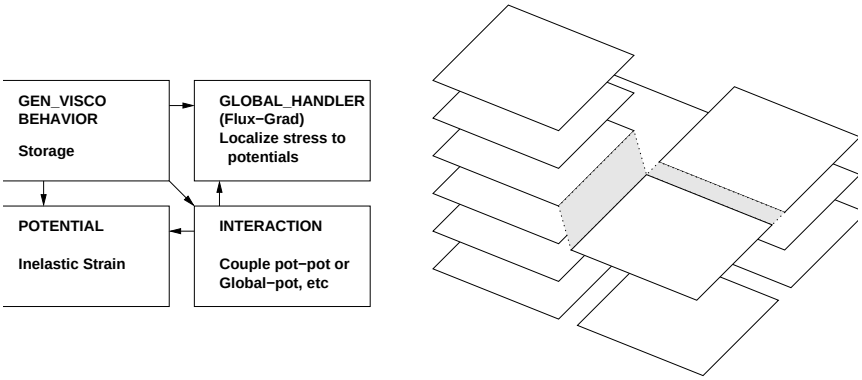
1. Calculation of $\dot{\epsilon}_c = \Delta\epsilon / \Delta t$
2. Call of the Runge–Kutta integration method
   (a) Calculation of $\dot{\epsilon}_0^p = \dot{\epsilon}_0^p(\epsilon_0^e, h_0)$ and $\dot{h}_0 = \dot{h}_0(\epsilon_0^e, h_0)$
   (b) Calculation of $\dot{\beta}_i$ and $\dot{\epsilon}_i^l$
   (c) Calculation de $\dot{\sigma}_c$ (eq. 20)
   (d) Calculation of $\dot{\sigma}_i$
   (e) Calculation of $\dot{\epsilon}_i^e = \mathbf{C}_i^{-1} \dot{\sigma}_i$
   (f) Building $\dot{V}$
3. Once the integration is performed one computes σ_c. The consistent stiffness matrix is taken equal as $\mathbf{C}_c$

Chapter 9

Gen-Evp

gen_evp

Description:
The "gen_evp" behavior is actually a system of behaviors, and material pieces which fit together. The main BEHAVIOR instance is really broken up into 3 levels, NL_M_TLE_B_SD which derives from BEHAVIOR, RUNGE_INTEGRATOR, and THETA_INTEGRATOR. The next class is GEN_NL_M_B_SD and then GEN_NPOT_M_B_SD. The levels of class can be considered legacy structure at this point; there is no real reason to have such a hierarchy. Another gen-evp type implementation is in REDUCED_PLASTIC



Data-storage:
The only real task handled in the gen_evp behavior itself is

The global handler:
The "global handler" class is responsible for the grad-flux (primal dual) variable relationship, and for localizing the global variables to the different potentials (e.g. calculating an effective stress in damage mechanics, or the grain stress in a polycrystal). Normally the handler carries a number of integrated variables, such as $\boldsymbol{\epsilon}_{el}$ or damage d, etc.

Potentials:

$$\mathbf{H}_i = \mathbf{M}_{ij}\mathbf{h}_j$$

Members:

`m_glob` The global handler.

`m_f_vec, m_d_chi, m_f_0, m_f_grad` The global handler.

`delta_param`

`curr_time_incr`

Methods:

`operator()` returns the pointer contained within.

`operator*` returns the data pointed to by the pointer.

`if_null` tells if the pointer held is NULL (which it is until something is set in the PTR).

`if_not_null` tells if the pointer held is not NULL.

`erase` delete the pointer and reset it to NULL.

`dont_delete` flags the class to skip the auto-deletion (which can cause a problem if the data gets deleted elsewhere).

`swap_pointer` swaps pointers between `PTR` objects.

Chapter 10

Utility mesh

UTILITY_MESH

Description:
The utility mesh is the base class holding the pure geometrical mesh entities, and is used directly for visualization in Zmaster and manipulated directly by the Mesher modules.

Please do not use any function assuming a single dimension for the mesh. The `dimension` variable is left for compatibilty, and should be set to the maximum dimension of the mesh. One should use `get_space_type` instead.

```
ZCLASS2 UTILITY_MESH {
   protected :

   public :
     STRING  given_name, fzebulon, pb_name;

     UTILITY_MESH_READER* mesh_reader;
     UTILITY_RESULTS_DATABASE* results_part;
     UTILITY_INP_FILE* input_file;

     LIST<STRING> elset_for_section, section_name;
     BUFF_LIST<UTILITY_NODE*>      nodes;
     BUFF_LIST<UTILITY_ELEMENT*>   elements;
     BUFF_LIST<UTILITY_NSET*>      nsets;
     BUFF_LIST<UTILITY_ELSET*>     esets;
     BUFF_LIST<B_UTILITY_SET*>     bsets;
     BUFF_LIST<UTILITY_IPSET*>     ipsets;

     UTILITY_MESH();
     UTILITY_MESH(const STRING&);
     virtual ~UTILITY_MESH();
     void set_name(const STRING&);

     void remove_orphan_nodes();
     void reestablish_connectivity();
     void reassign_numbers();
     void reassign_pointers();
     UTILITY_NODE*     find_node_with_id(int id, bool issue_error=TRUE);
     UTILITY_ELEMENT*  find_elem_with_id(int id, bool issue_error=TRUE);
     UTILITY_NSET*     get_nset(const STRING&,bool issue_error=TRUE);

     virtual bool results_load(STRING format);
     virtual void load(STRING format);
     virtual void write_zebulon(STRING format);

     virtual bool verify();
     virtual void reset(void);

     virtual void read_z7_geom(ASCII_FILE&);
     virtual void read_z7_section(ASCII_FILE&);
     virtual void read_z7_group(ASCII_FILE&);
     virtual void read_z7_nset(ASCII_FILE&);
     virtual void read_z7_bset(ASCII_FILE&);
     virtual void read_z7_elset(ASCII_FILE&);
```

```
     virtual void read_z7_ipset(ASCII_FILE&);
     virtual void sort_nset(ASCII_FILE&);

     virtual void element_renumbering(ARRAY<int> &order);
     virtual void node_renumbering(ARRAY<int>& order);

     static int              get_nb_integration_point(const STRING&);
     static GEOM_TYPE        get_geom_type(const STRING&);
     static int              get_nb_node(const STRING&);
     static int              get_dimension(const STRING&);
     static INTEGRATION_MODE get_integration_mode(const STRING&);
     static SPACE_TYPE       get_space_type(const STRING&);
};
```

Class Use:

Creators and initialization:
The startup of the mesh creation is a bit odd and difficult to follow because of the different uses of the class for different applications. Fortunately is is rather rare at this point to need to modify this creation. The problem stems mostly from the fact that the creation of a Zebulon mesh has not yet been abstracted to a mesh reader class.

Major components:

`mesh_reader` the class which handles reading and writing of the mesh if it is not a Zebulon native format. This gets assigned when doing an import or export in the mesher or in Zmaster.

`results_part` A "Results Database" to handle putting results on the mesh. This can be redefined along with the mesh reader, which was done for the ABAQUS fil reader / post processor.

`input_file` A class to handle input file (FEA solution) type items. The Gfm reader uses this, where at the of reading an UTILITY_INP_FILE is created, and filled out directly, then the reader asks the inp file to save itself.

UTILITY_ELEMENT

Description:
Utility elements have different types for the different element geometries. There are many derived forms in `Utility_elements.c` and `Utility_1d_elements.c`.

```
ZCLASS2 UTILITY_ELEMENT {
   public :
   char                 type[10];
   int                  space_dim;
   BUFF_LIST<UTILITY_ELEMENT*>  attached_elements;

   int  not_displayed;
   int  num_gp;
   ARRAY<UTILITY_BOUNDARY> faces;
   ARRAY<UTILITY_BOUNDARY> edges;
   ARRAY<UTILITY_NODE*>    nodes;
   ARRAY<double>           r_info;

   void write_in(fstream&);
   void write_id(fstream& file) { file << " " << id; }

   void set_id(int idin) { id = idin; }
   int  give_id() { return id; }
   void set_rank(int rk) { rank = rk; }
   int  give_rank() { return rank; }
   void set_id_rank(int idin);

   virtual void add_bsets(B_UTILITY_SET&, BUFF_LIST<UTILITY_NODE*>&);
   virtual ARRAY<UTILITY_BOUNDARY>& get_faces();
   virtual ARRAY<UTILITY_BOUNDARY>& get_edges();
   virtual void check_orientation();

   UTILITY_ELEMENT();
   UTILITY_ELEMENT(const UTILITY_ELEMENT&);
   virtual void initialize(const STRING& the_type);

   virtual ~UTILITY_ELEMENT();
   void* operator new(size_t) { return (void*)mem_buff.get_ptr(); }
   void  operator delete(void* oo) { mem_buff.put_ptr((char*)oo); }
   static UTILITY_ELEMENT* make(const STRING& the_type);
   virtual UTILITY_ELEMENT* element_copy_self();
   virtual UTILITY_ELEMENT* copy_to_linear(ARRAY<UTILITY_NODE*>& removed_nodes);

   UTILITY_ELEMENT& operator=(const UTILITY_ELEMENT&);
   bool operator==(const UTILITY_ELEMENT& in)const { return is_equal(in); }
   bool operator!=(const UTILITY_ELEMENT& in)const { return (*this==in) ? FALSE : TRUE; }
   virtual void local_draw(GRAPHICS_AREA* ga);
   virtual void run_face_setup();
   virtual void run_face_setup_disp(const ARRAY<VECTOR>& disp);
   virtual void set_ctnod(ARRAY<int>& markers, VECTOR& values);
   virtual void set_ctele(VECTOR& values, int& index);
   virtual void set_integ(VECTOR& values, int& index);
```

```
   virtual int  num_contour_vals()const { return !nodes; }
};
```

Class Use:

Creators and initialization:

Major components:

`type` Characters for the element key type. e.g. `c2d8r`, `c3d20`, `s3d6`.

`attached_elements` Pointers on attached elements. Established from the mesh `reestablish_connectivity` function.

`num_gp` number of gauss points. For inspection only.

`faces` The boundary sets making up the faces. 2D or 3D shells have 2 faces for the top and bottom. 3D faces should have their nodes ordered to have the normal pointing out of the element. Not active until the element `get_faces` is called.

`edges` Line edges of the element. Not active until the element `get_edges` is called.

`nodes` element nodes, ordered in normal Zebulon ordering.

`r_info` Real info. Thickness, etc.

`set_id, give_id` Manipulate the id tags. Default rank + 1.

`set_rank, give_rank` Utility mesh sets these up in `reestablish_connectivity`.

`set_id_rank` When using `set_id_rank`, you pass the ID!! so rank is idin-1.

C3D6:

face 1	0 1 2
face 2	1 4 5 2
face 3	0 3 4 1
face 4	0 2 5 3
face 5	3 5 4

(1)

C3D15:

face 1	0 5 4 3 2 1
face 2	0 1 2 71110 9 6
face 3	2 3 4 8131211 7
face 4	0 6 914138 4 5
face 5	91011121314

(2)

C3D10:

face 1	0 3 1 7 9 6

face 2	0 5 2 4 1 3
face 3	0 6 9 8 2 5
face 4	2 8 9 7 1 4

(3)

C3D4:

face 1	0 1 2
face 2	1 3 2
face 3	0 2 3
face 4	0 3 1

(4)

C3D8:

face 1	0 1 2 3
face 2	1 5 6 2
face 3	3 2 6 7
face 4	4 0 3 7
face 5	4 5 1 0
face 6	5 4 7 6

(5)

C3D20:

face 1	0 1 2 3 4 5 6 7
face 2	2 914151610 4 3
face 3	6 5 41016171811
face 4	0 7 611181912 8
face 5	0 8121314 9 2 1
face 6	1219181716151413

(6)

UTILITY_NODE

Description:
Utility nodes are a concrete type for nodal information storage. There is no need for derived types. Their connectivity to elements is done by the utlity mesh `reestablish_connectivity` function.

```
ZCLASS2 UTILITY_NODE {
 public :
   VECTOR    position;

   BUFF_LIST<UTILITY_ELEMENT*> attached_elements;

   bool read(ASCII_FILE&,int);
   bool read_old(ASCII_FILE&,int);
   void write_in(fstream&);
   void write_id(fstream& file);

   void set_id(int idin):
   int  give_id();
   void set_rank(int rk);
   int  give_rank();
   void set_id_rank(int idin);

   UTILITY_NODE();
   UTILITY_NODE(const UTILITY_NODE&);
   ~UTILITY_NODE() { }

   UTILITY_NODE& operator=(const UTILITY_NODE&);
   UTILITY_NODE& operator=(const Complex&);
   bool operator==(const UTILITY_NODE&)const;
   bool operator!=(const UTILITY_NODE& in)const;
};

double distance(const UTILITY_NODE&,const UTILITY_NODE&);
```

Class Use:

Manipulate the data freely here.

Major components:

`position` A vector with the node position.

`attached_elements` List of the elements holding this node. One can get nearby nodes then with the attached elements attached nodes.

`set_id, give_id` Manipulate the id tags. Default rank + 1.

`set_rank, give_rank` Utility mesh sets these up in `reestablish_connectivity`.

`set_id_rank` When using `set_id_rank`, you pass the ID!! so rank is idin-1.

UTILITY_BOUNDARY

Description:
The utility boundaries are used for BSET type items. They are also used for for graphical rendering of the mesh.

```
ZCLASS2 UTILITY_BOUNDARY {
  public :
    enum BTYPES { BTYPE_NONE,
                  BTYPE_LINE,
                  BTYPE_QUAD,
                  BTYPE_T3,
                  BTYPE_T6,
                  BTYPE_Q4,
                  BTYPE_Q8 } type;

    static const char* translate(BTYPES type);

    ARRAY<UTILITY_NODE*> nodes;

    // graphics related.. will only be sized for graphics
    // applications.. therefore they don't add a lot..
    // is_shared may be set sometimes... sometimes not
    //
    ARRAY<GRAPHICS_POINT>* points;
    PTR<GRAPHICS_FACE>     graphics_face;
    short             is_shared;
    short             sharable;

    UTILITY_BOUNDARY();
    UTILITY_BOUNDARY(const UTILITY_BOUNDARY& in);
    virtual ~UTILITY_BOUNDARY();

    UTILITY_BOUNDARY& operator=(const UTILITY_BOUNDARY&);
    bool operator==(const UTILITY_BOUNDARY&)const;
    bool operator!=(const UTILITY_BOUNDARY& in)const;
```

Class Use:

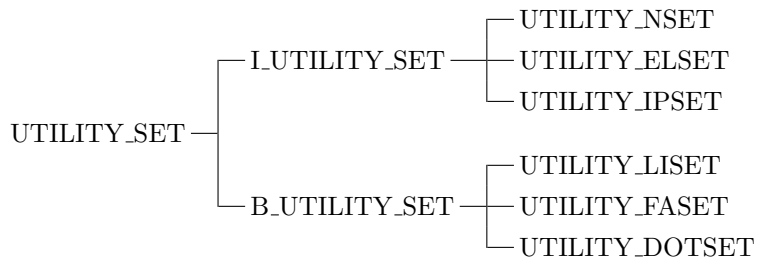
Major components:

`type` indicates the geometry type of the face. This can be translated to the character names used in the geof file (e.g. `t3` `q8`) using `translate`.

`nodes` nodes of the face. The mesh handles the storage (creating and deleting) of the nodes.

UTILITY_SETUtility_set.h

Description:
The set of classes deriving from UTILITY_SET are used to hold the information of Zebulon mesh sets. All set types derive from this class.




```
ZCLASS2 UTILITY_SET {
 protected :
 public :
   enum { NSET_CODE, ELSET_CODE, LISET_CODE, FASET_CODE, DOTSET_CODE , IPSET_CODE };

   STRING name;
   virtual bool verify()const;
   virtual int get_type()const=0;
   virtual void write_in(fstream&)=0;
   virtual ~UTILITY_SET() { }
};

ZCLASS2 I_UTILITY_SET : public UTILITY_SET {
  public :
   virtual int operator!()const=0;
};

ZCLASS2 UTILITY_NSET : public I_UTILITY_SET {
 protected :
 public :
   BUFF_LIST<UTILITY_NODE*> nodes;

   UTILITY_NSET() { }
   UTILITY_NSET(const UTILITY_NSET& in) : nodes(in.nodes) { name=in.name; }
   int operator!()const { return !nodes; }
   int get_type()const { return UTILITY_SET::NSET_CODE; }
   virtual void write_in(fstream&);
};

ZCLASS2 UTILITY_ELSET : public I_UTILITY_SET {
 public :
   BUFF_LIST<UTILITY_ELEMENT*> elem;

   UTILITY_ELSET() { }
   UTILITY_ELSET(const UTILITY_ELSET& in);
   int operator!()const;
   int get_type()const { return UTILITY_SET::ELSET_CODE; }
   virtual void write_in(fstream&);
};

ZCLASS2 UTILITY_IPSET : public I_UTILITY_SET {
```

```
 public :
   BUFF_LIST<UTILITY_ELEMENT*> elem;
   BUFF_LIST<int>              ip;

   UTILITY_IPSET(){}
   ~UTILITY_IPSET() {}
   int operator!()const;
   int get_type()const { return UTILITY_SET::IPSET_CODE; }
   virtual void write_in(fstream&);
   void add_ip(UTILITY_ELEMENT* e, int i);
};
```

Class Use:

B_UTILITY_SET

```
class BSET_TAG {
  public :
    char dat[5];
    BSET_TAG();
    BSET_TAG(const char* str);
    BSET_TAG(STRING str);

    BSET_TAG& operator=(const STRING& str_in);
    BSET_TAG& operator=(const char* buf_in);

    int operator==(const char* str_in)const;
    int operator==(const STRING& str_in)const;
    int operator==(const BSET_TAG& tag)const;

    const  char* operator() () const { return (char*)(&dat[0]); }
    operator char*()const   { return (char*)(&dat[0]); }
    operator STRING()const  { return STRING(dat); }
};

ZCLASS2 B_UTILITY_SET : public UTILITY_SET {
  public :
    BUFF_LIST< ARRAY<UTILITY_NODE*>* > lnode;
    BUFF_LIST<BSET_TAG>        type;

    B_UTILITY_SET() { }
    B_UTILITY_SET(const B_UTILITY_SET& in);
    virtual B_UTILITY_SET* copy_self();

    void add(const UTILITY_BOUNDARY& bnd);
    void add(BUFF_LIST<UTILITY_BOUNDARY*>& bnd);

    virtual void write_in(fstream&);
    virtual ~B_UTILITY_SET();
};

ZCLASS2 UTILITY_FASET : public B_UTILITY_SET {
  public :
   UTILITY_FASET() { }
   UTILITY_FASET(const UTILITY_FASET& in) : B_UTILITY_SET(in) { }
   virtual B_UTILITY_SET* copy_self() { return new UTILITY_FASET(*this); }

   int get_type()const { return UTILITY_SET::FASET_CODE; }

};

ZCLASS2 UTILITY_DOTSET : public B_UTILITY_SET {
  public :
```

```
   UTILITY_DOTSET() { }
   UTILITY_DOTSET(const UTILITY_DOTSET& in) : B_UTILITY_SET(in) { }
   virtual B_UTILITY_SET* copy_self() { return new UTILITY_DOTSET(*this); }

   int get_type()const { return UTILITY_SET::DOTSET_CODE; }
};
```

GEOM_TYPE

Description:
This file holds some global enumeration values used to specify characteristics of the space and geometries.

```
enum INTEGRATION_MODE {
     NO_INTEGRATION,
     GAUSS_POINT_INTEGRATION
};

enum GEOM_TYPE {
      NO_GEOMETRY,
      _1DOT,
      _2DOT,
      _3DOT,
      _1D,
      _SPH,
      _CYL,
      _2D,
      _AXI,
      _3D,
      _SPR1
};

enum SPACE_TYPE {
      ST_SPACE,
      ST_SURFACE,
      ST_LINE,
      ST_DOT,
      ST_SHELL,
      ST_BEAM
};
```

UTILITY_MESH_READER

Description:
An abstract class for making a new mesh import format.

```
ZCLASS2 UTILITY_MESH_READER {
   protected :
   public :
      UTILITY_MESH* its_mesh;

      UTILITY_MESH_READER();
      virtual ~UTILITY_MESH_READER();

      virtual void initialize(ASCII_FILE& file, UTILITY_MESH* mesh);
      virtual void write(STRING fname);
};
```

UTILITY_RESULTS_DATABASE

Description:
This class is the mechanism for grabbing results to display from a certain format of results file. It is currently far from general, and should be seriously re-designed. It should also be used as the interface to the data within the Post processor, which it is not.

```
ZCLASS2 UTILITY_RESULTS_DATABASE {
  public :
    int           if_linear_solution;
    STRING        pb_name;
    LIST<STRING>  mesh_names;

    UTILITY_MESH*        active_mesh;
    int                  mesh_changed;

    LIST<double>  map;        //  the times are stored here.

    enum LOCATIONS {
               RENDER=1,
               CTNOD=2,
               CTELE=4,
               CTMAT=8,
               INTEG=16,
               VECTR=32,
               DISPL=64,
               VELOC=128,
               VELOC3D=256
    };
    //
    // The "get" functions set these up... they may set up the
    // data differently for each type of data query.. NEED DOCUMENTAITON
    //
    MARRAY<VECTOR> disp;       // #nodes, dimension
    VECTOR         node_val;   // # nodes ...
    double         exact_map;

    ARRAY<int>     ct_marker;  // Contour marker.

    UTILITY_RESULTS_DATABASE();

    virtual ~UTILITY_RESULTS_DATABASE();
    virtual void initialize(const STRING& problem_name);
    static UTILITY_RESULTS_DATABASE* make(STRING format, const STRING& problem_name);

    virtual bool do_command(STRING cmd);

    //
    // Default error messages..
    //
    virtual bool get_node_results_through_time(UTILITY_NODE* node,
                                               STRING cname, int where);
    virtual bool get_integ_results_through_time(UTILITY_ELEMENT* ele, int gp, STRING cname);
```

```
    virtual bool get_results_at_time(double when, STRING cname, int where);
    virtual bool get_nodal_displacements(int map, STRING stub);
    virtual bool get_ctnod_results(int map, STRING comp);
    virtual bool get_ctele_results(int map, STRING comp);
    virtual bool get_ctmat_results(int map, STRING comp);
    virtual bool get_integ_results(int map, STRING comp);

    virtual void generate_location_list(int& all_location_flags);
    virtual void generate_node_variable_list(LIST<STRING>&);
    virtual void generate_integ_variable_list(LIST<STRING>&);

    static bool small_diff(double d1, double d2);
};
```

Class Use:

UTILITY_UT_FILE

Description:
This is a class for storing the information from the `.ut` file. The utility results database uses this internally.

```
ZCLASS2 UTILITY_UT_FILE {
  protected :
  public :
    UTILITY_UT_FILE();
    UTILITY_UT_FILE(ASCII_FILE& file);
    virtual ~UTILITY_UT_FILE();

    void read(ASCII_FILE& file);

    int           if_linear_solution;

    STRING        pb_name;
    STRING        meshfile;
    STRING        ctmat_suff, node_suff, integ_suff;
    STRING        ctele_suff;
    STRING        ctnod_suff;
    LIST<STRING>  node;
    LIST<STRING>  integ;
    LIST<STRING>  element;

    LIST<double>  map;        //  the times are stored here.
    LIST<STRING>  map_m_file; //  this is so the mesh can change
};
```

UTILITY_INP_FILE

Description:
A class to hold information from the FEA calculation input file. At the time of writing (02/2000) this is not close to being complete. A mesh reader can use this to build up data for how to run the calculation. Manipulate the records directly.

```
class UTILITY_BC {
  public :
    enum TYPE {
       IMPOSE_NODAL_DOF,
       IMPOSE_NODAL_REACTION,
       CENTRIFUGAL
    } type;

    STRING set;
    STRING DOF;
    double value;
    STRING tab;
};

class UTILITY_MAT_FILE {
  public :
};

class UTILITY_SEQUENCE {
  public :
   LIST<double> time;
   LIST<int>    incr;

};

class UTILITY_TABLE {
  public :
   int    id;
   STRING name;
   LIST<double> time;
   LIST<double> values;
};

class UTILITY_INP_FILE {
  protected :
  public :
     UTILITY_MESH* its_mesh;

     BUFF_LIST<UTILITY_BC*>       bc;
     BUFF_LIST<UTILITY_MAT_FILE*> mat;
     BUFF_LIST<UTILITY_SEQUENCE*> seq;
     BUFF_LIST<UTILITY_TABLE*>    tab;

     UTILITY_INP_FILE();
     virtual ~UTILITY_INP_FILE();
```

```
    virtual void load(ASCII_FILE& input);
    virtual void write(Zofstream& output);

    virtual void read_bc(ASCII_FILE& input);
    virtual void read_table(ASCII_FILE& input);
    virtual void read_resolution(ASCII_FILE& input);
};
```

MECHANICAL_BFGS_ALGORITHM

Implementing B.F.G.S. method for mechanical problems

•Generalities

In the B.F.G.S. method are implemented :
— an algorithm used to update the global stiffness matrix without new assembling
— a conjugated (??) gradient method to optimize the d.o.f. increment at each iteration.

The method may start at any iteration, the global matrix being no longer computed after starting. This first iteration is given by the user. The method needs at least two successive residual vectors and a solution vector. From the initial residual vector R_0 and the initial global stiffness matrix K_0, a computation of the first solution and of a new residual can be made:

$$\delta_1 = (K_0)^{-1} R_0$$

$$R_1 = \Psi(\delta_1)$$

The method is then initiated from R_0, R_1, δ_1. Generally speaking:

$$\delta_i = (K_{i-1})^{-1} R_{i-1}; R_i = \Psi(\delta_i)$$

The method uses R_{i-1}, R_i and δ_i to find δ_{i+1}.

•Matrix updating

The matrix $(K_i)^{-1}$ is updated from the following formula:

$$(K_i)^{-1} = (I + w_i \otimes v_i)(K_{i-1})^{-1}(I + v_i \otimes w_i)$$

with:

$$v_i = \left[1 + \sqrt{\frac{g_{i-1} - g_i}{g_{i-1}}}\right] R_{i-1} - R_i \ ; \ w_i = \frac{\delta_i}{\Delta g_i}$$

$$g_i = \delta_i R_i \ ; \ g_{i-1} = \delta_i R_{i-1} \ ; \ \Delta g_i = g_i - g_{i-1}$$

This algorithm can be rewritten :

$$\delta_{i+1} = (I + w_i \otimes v_i) \dots (I + w_1 \otimes v_1)(K_0)^{-1}(I + v_1 \otimes w_1) \dots (I + v_i \otimes w_i) R_i$$

The computation is then made in three steps:
— calculation of a pseudo-residual vector:

$$R_i^* = (I + v_1 \otimes w_1) \dots (I + v_i \otimes w_i) R_i$$

— computation of a pseudo-solution:

$$\delta_{i+1}^* = (K_0)^{-1} R_i^*$$

— computation of the d.o.f. increment:

$$\delta_{i+1} = (I + w_i \otimes v_i) \dots (I + w_1 \otimes v_1) \delta_{i+1}^*$$

- Notes:
- The updating should not be made if this operation destroy the numerical conditioning. This will be controlled by computing the eigenvalue

$$\sqrt{\frac{\delta_1 R_{i-1}}{\delta_i (R_i - R_{i-1})}}$$

  and avoiding the update procedure if it is greater than 10^{14}.
- The update procedure is simply made by two actions:
  — store two vectors v_i and w_i, the size of which is the total number of d.o.f.,
  — increment the index which memorizes the number of updating.
- The maximum number of updating in an increment is 15. If more iterations are needed to complete an increment, the method restart from the initial $(K_0)^{-1}$ matrix for iterations 16, 31, ...

•Conjugated gradient method

After each resolution, a scalar parameter α is optimized to make the successive increments "perpendicular", in the norm defined by the stiffness matrix:

$$\delta_{i+1} R_{i+1} = 0, \quad \text{with } R_{i+1} = \Psi(u_i + \alpha \delta_{i+1})$$

so that:

$$\delta_{i+1} K_{i+1} \delta_{i+2} = 0$$

The value of α is found by dichotomy.

E2_5

Generalized Degree Of Freedom Element (2.5D)

6 generalized degrees of freedom element
displacement field is given by

$$\vec{u}(x,y,z) = \vec{u}_0(x,y,z) + u_1(x,y)$$

where $\vec{u}_0$ is given by

$$\vec{u}_0(x,y,z) = z\left(\vec{t} + \vec{w} \wedge \overrightarrow{OM}\right)$$

z corresponds to the third direction. O is the center of gravity of the 2D structrure. Both vectors $\vec{t}$ and $\vec{w}$ are given by

$$\vec{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \qquad \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

$t_1 \dots t_3$ and $w_1 \dots w_3$ are the 6 generalized degrees of freedom. The displacement field is given by

$$\begin{aligned} u^x &= u_1^x + z t_1 - w_3 (y - Y_0) z \\ u^y &= u_1^y + z t_2 + w_3 (x - X_0) z \\ u^z &= u_1^z + z t_3 + w_1 (y - Y_0) z - w_2 (x - X_0) z \end{aligned}$$

All computations are done for $z = 0$. Let $q = (q_1^x, q_1^y, \dots, q_n^x, q_n^y)$ be the vector of nodal displacements. Deformations are given by

$$\begin{aligned} \epsilon_{xx} &= \sum_i \frac{\partial N_i}{\partial x} q^x \\ \epsilon_{yy} &= \sum_i \frac{\partial N_i}{\partial y} q^y \\ \epsilon_{zz} &= t_{3+} w_1 (y - Y_0) - w_2 (x - X_0) \\ \sqrt{2}\epsilon_{xy} &= \frac{1}{\sqrt{2}} \left(\sum_i \frac{\partial N_i}{\partial x} q^y + \sum_i \frac{\partial N_i}{\partial y} q^x \right) \\ \sqrt{2}\epsilon_{yz} &= \frac{1}{\sqrt{2}} (t_2 + w_3 (x - X_0)) \\ \sqrt{2}\epsilon_{zx} &= \frac{1}{\sqrt{2}} (t_1 - w_3 (y - Y_0)) \end{aligned}$$

so that one gets

$$\bar{\epsilon} = \begin{pmatrix} \mathbf{B}_{xx} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{B}_{yy} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{B}_{zz} & 0 & 0 & 1 & \Delta y & -\Delta x & 0 \\ \mathbf{B}_{xy} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{0} & 0 & 1/\sqrt{2} & 0 & 0 & 0 & \Delta x/\sqrt{2} \\ \mathbf{0} & 1/\sqrt{2} & 0 & 0 & 0 & 0 & -\Delta y/\sqrt{2} \end{pmatrix} \begin{pmatrix} q \\ t_1 \\ t_2 \\ t_3 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \mathcal{B} Q$$

where $\mathbf{B}$ is the usual matrix for small deformations plane strain problems and $\Delta x = x - X_0$, $\Delta y = y - Y_0$. The element stiffness matrix is given by

$$\mathbf{K}_e = \int_V \mathcal{B}^T D \mathcal{B} dv$$

and internal forces by

$$F_i = \int_V \mathcal{B}^T \bar{\sigma} dv$$

The last three terms of F_i correspond to reactions associated to the 6 generalized degrees of freedom.

$$\begin{aligned}
F_i(t_1) &= \int_v \sigma_{zx} dv \\
F_i(t_2) &= \int_v \sigma_{yz} dv \\
F_i(t_3) &= \int_v \sigma_{zz} dv \\
F_i(w_1) &= \int_v (y - Y_0)\, \sigma_{zz} dv \\
F_i(w_2) &= \int_v -(x - X_0)\, \sigma_{zz} dv \\
F_i(w_3) &= \int_v \left[(x - X_0)\, \sigma_{yz} - (y - Y_0)\, \sigma_{zx}\right] dv
\end{aligned}$$

MINDLIN_SHELL

- **3D mechanical shell elements**
- **Geometry**

The shell is geometrical described with ($\overline{x}$ position):

$$\overline{x}(\eta_1,\eta_2,z) = \overline{x}_p(\eta_1,\eta_2) + z\overline{n}(\eta_1,\eta_2)$$

where z represents the position in the thickness direction. $\overline{x}_p$ is the mapping of the mid–plane and $\overline{n}$ the normal to the shell. Differentiating the previous equation gives:

$$\begin{aligned} d\overline{x} &= \left(\frac{\partial \overline{x}_p}{\partial \eta_1} + z\frac{\partial \overline{n}}{\partial \eta_1}\right) d\eta_1 + \left(\frac{\partial \overline{x}_p}{\partial \eta_2} + z\frac{\partial \overline{n}}{\partial \eta_2}\right) d\eta_2 + \overline{n}dz \\ &= \overline{a}_1 d\eta_1 + \overline{a}_2 d\eta_2 + z\left(\frac{\partial \overline{n}}{\partial \eta_1} d\eta_1 + \frac{\partial \overline{n}}{\partial \eta_2} d\eta_2\right) + \overline{n}dz \\ &= \overline{a}'_1 d\eta_1 + \overline{a}'_2 d\eta_2 + \overline{n}dz \end{aligned}$$

with

$$\overline{a}_i = \frac{\partial \overline{x}_p}{\partial \eta_i} \quad \text{and} \quad \overline{a}'_i = \overline{a}_i + z\frac{\partial \overline{n}}{\partial \eta_i}$$

$\overline{a}_i$'s are tangent to the mid–plane. $\overline{a}'_i$'s account for variable thickness. $\underline{F}$ is defined as:

$$\underline{F} = \left[\overline{a}'_1 \vdots \overline{a}'_2 \vdots \overline{n}\right]$$

An orthogonal local base can be defined as:

$$\begin{aligned} \overline{t}'_1 &= \overline{a}'_1/|\overline{a}'_1| \\ \overline{t}'_2 &= \overline{n}' \wedge \overline{t}'_1 \\ \overline{n}' &= (\overline{a}'_1 \wedge \overline{a}'_2)/|\overline{a}'_1 \wedge \overline{a}'_2| \end{aligned}$$

an the rotation tensor:

$$\underline{Q}' = \left[\overline{t}'_1 \vdots \overline{t}'_2 \vdots \overline{n}'\right]$$

For the mid–plane, one defines:

$$\begin{aligned} \overline{t}_1 &= \overline{a}_1/|\overline{a}_1| \\ \overline{t}_2 &= \overline{n} \wedge \overline{t}_1 \\ \overline{n}' &= (\overline{a}_1 \wedge \overline{a}_2)/|\overline{a}_1 \wedge \overline{a}_2| \end{aligned}$$

an the rotation tensor:

$$\underline{Q} = \left[\overline{t}_1 \vdots \overline{t}_2 \vdots \overline{n}\right]$$

- **Kinematics**

The displacement is given by:

$$\overline{u}(\eta_1,\eta_2,z) = \overline{u}_p(\eta_1,\eta_2) + z\overline{\beta}(\eta_1,\eta_2)$$

with

$$\overline{\beta}.\overline{n} = 0$$

$\overline{\beta}$ is expressed as $\beta_1\bar{t}_1 + \beta_2\bar{t}_2$ in the local shell base and as $\beta_i\overline{e}_i$ $(i = 1\ldots3)$ in the cartesian base. In this base, the displacement vector is expressed as:

$$\overline{u}(\eta_1,\eta_2,z) = U_i(\eta_1,\eta_2)\overline{e}_i + z\left(\beta_i(\eta_1,\eta_2)\overline{e}_i\right)$$

Deformation is computed as:

$$\begin{aligned}\frac{\partial u_i}{\partial x_j} &= \left(\frac{\partial U_i}{\partial\eta_1}\frac{\partial\eta_1}{\partial x_j} + \frac{\partial U_i}{\partial\eta_2}\frac{\partial\eta_2}{\partial x_j}\right)\\ &\quad + z\left(\frac{\partial\beta_i}{\partial\eta_1}\frac{\partial\eta_1}{\partial x_j} + \frac{\partial\beta_i}{\partial\eta_2}\frac{\partial\eta_2}{\partial x_j}\right)\\ &\quad + \frac{\partial z}{\partial x_j}\beta_i\end{aligned}$$

Note that $\frac{\partial z}{\partial x_j} = 0$ for shell having a constant thickness.

• **FE discretization of the geometry**

Let $N^k(\eta_1,\eta_2)$ is the shape function associated to node k. The position vector is expressed as:

$$\overline{x} = N^k\overline{X}^k + \zeta N^k h^k\overline{n}^k \quad \zeta_{\min}\le\zeta\le\zeta_{\max}$$

where $\overline{X}^k$ is the position of node k, h^k the thickness at node k and $\overline{n}^k$ the shell normal at node k. $(\zeta_{\max}-\zeta_{\min})N^k h^k$ is the local shell thickness. One also has: $z = \zeta N^k h^k = \zeta h(\eta_1,\eta_2)$. The local base can be computed:

$$\begin{aligned}\overline{a}'_1 &= \frac{\partial\overline{x}}{\partial\eta_1} = N^k_{\eta_1}\left(\overline{X}^k + \zeta\overline{H}^k\right)\\ \overline{a}'_2 &= \frac{\partial\overline{x}}{\partial\eta_2} = N^k_{\eta_2}\left(\overline{X}^k + \zeta\overline{H}^k\right)\\ \overline{n}' &= \frac{\partial\overline{x}}{\partial\zeta} = N^k\overline{H}^k\end{aligned}$$

Note that:

$$d\overline{x} = \underline{F}\begin{pmatrix}d\eta_1\\ d\eta_2\\ d\zeta\end{pmatrix}$$

• **FE discretization of the kinematics**

The displacement is now written as:

$$\overline{u} = N^k\overline{U}^k + z(N^k\overline{\beta}^k)$$

or

$$u_i(\eta_1,\eta_2,\zeta) = N^kU^k_i + \zeta\left(N^kh^k\right)\left(N^k\beta^k_i\right) = N^kU^k_i + \zeta h\left(N^k\beta^k_i\right)$$

where U^k_i and β^k_i represent nodal values. The transformation gradient is:

$$\begin{aligned}\frac{\partial u_i}{\partial x_j} = u_{i,j} &= \frac{\partial N^k}{\partial x_j}U^k_i\\ &\quad + \left(\frac{\partial\zeta}{\partial x_j}\left(N^kh^k\right)N^k + \zeta\left(\frac{\partial N^k}{\partial x_j}h^k\right)N^k + \zeta\left(N^kh^k\right)\frac{\partial N^k}{\partial x_j}\right)\beta^k_i\\ &= \frac{\partial N^k}{\partial x_j}U^k_i + \left(\frac{\partial\zeta}{\partial x_j}hN^k + \zeta\frac{\partial h}{\partial x_j}N^k + \zeta h\frac{\partial N^k}{\partial x_j}\right)\beta^k_i\end{aligned}$$

$\frac{\partial \zeta}{\partial x_j}$ is computed as $F^{-1}_{3,j}$. $\frac{\partial N^k}{\partial x_j}$ is given by:

$$\frac{\partial N^k}{\partial x_j} = \frac{\partial N^k}{\partial \eta_1} F^{-1}_{1,j} + \frac{\partial N^k}{\partial \eta_2} F^{-1}_{2,j}$$

Let $\langle U \rangle$ be the vector of nodal unknowns:

$$\langle U \rangle = \begin{pmatrix} U^1_1 \\ \vdots \\ U^1_3 \\ \beta^1_1 \\ \vdots \\ \beta^1_3 \\ \vdots \\ U^N_1 \\ \vdots \\ U^N_3 \\ \beta^N_1 \\ \vdots \\ \beta^N_3 \end{pmatrix}$$

$[B]$ is defined so that the deformation tensor $\underline{\epsilon}$ ($\epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i})$) is given by:

$$\underline{\epsilon} = [B] \langle U \rangle$$

the local shell deformation $\underline{\epsilon}^L$ is obtained by rotation using $\underline{Q}$:

$$\underline{\epsilon}^L = \underline{Q}'^T \underline{\epsilon} \underline{Q}'$$

• **Plane stress condition**

The plane stress condition in the direction normal to the shell is usually enforced using a proper material constitutive equation. This necessitate to rewrite material constitutive equation specifically for the plane stress condition. In order to overcome this problem, one uses additional degrees of freedom representing the deformation in the $\overline{n}'$ direction: ϵ_{33}. There is therefore one additional dof per Gauss point. The local deformation tensor is now expressed as:

$$\underline{\epsilon}^L = \underline{Q}'^T \underline{\epsilon} \underline{Q}' + \epsilon_{33} \underline{J}$$

or

$$\underline{\epsilon} = [B] \langle U \rangle + \epsilon_{33} \underline{Q}' \underline{J} \underline{Q}'^T$$

with

$$\underline{J} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The elements unknown are: the nodal displacements and rotations $\langle U \rangle$ and the deformations attached to each Gauss point $\langle E \rangle$. Using the previous definition of the strain tensor it is possible to use the "standard" material constitutive equations to compute the stress tensor $\underline{\sigma}^L$ and the consistent stiffness matrix $\underline{\underline{D}}^L = \frac{\partial \underline{\sigma}^L}{\partial \underline{\epsilon}^L}$. $\underline{\sigma}$ and $\underline{\underline{D}}$ are obtained by rotation using $\underline{Q}'$.

Reactions are associated to dof's. $\langle F\rangle_u$ is the nodal reaction vector associated to $\langle U\rangle$ and F_e^p the reaction associated to the p^{th} Gauss point (ω^p: associated volume). One has:

$$\langle F\rangle_u = \int_{V_e} [B]^T \underline{\sigma} dV_e = \sum_p [B]^T \underline{\sigma}\omega^p \qquad F_e = \sigma_{33}^L \omega^p$$

The associated element stiffness matrix is given by:

$$[K]^e = \begin{bmatrix} [K]^{uu} & [K]^{ue} \\ [K]^{eu} & [K]^{ee} \end{bmatrix}$$

with

$$\begin{aligned} [K]^{uu} &= \sum_p [B]^T \underline{\underline{D}} [B] \omega^p \\ K_{pp}^{ee} &= D_{33}^L \omega^p \qquad K_{ij}^{ee} = 0 \text{ if } i \neq j \\ K_{pj}^{eu} &= \sum_i D_{3i} B_{ij} \omega^p \\ K_{ip}^{ue} &= \sum_j B_{ij}^T D_{j3} \omega^p \end{aligned}$$

$\underline{\underline{D}}$, $\underline{\underline{D}}^L$, $[B]$, $\underline{\sigma}$ correspond to the p^{th} Gauss point in the preceeding equations. Subscript p is omitted for clarity.

• **Enforcement of the $\overline{\beta}.\overline{n} = 0$ condition**

Let $\overline{n}^k$ be the vector normal to the shell mid–plane at node k. The $\overline{\beta}.\overline{n} = 0$ condition is written as:

$$N^k \beta_i^k n_i^k = 0$$

In order to enforced this condition one defines $\theta = \overline{\beta}.\overline{n}$ and an associated force $f^\theta = \alpha\theta$ where θ is a penalization factor. θ can be expressed as a function of the nodal unknowns as:

$$\theta = [\Omega] \langle U\rangle$$

with

$$[\Omega] = \left[0, 0, 0, N^1 n_1^1, N^1 n_2^1, N^1 n_3^1, \ldots, 0, 0, 0, N^N n_1^N, N^N n_2^N, N^N n_3^N\right]$$

Unsing the constitutive relation $f^\theta = \alpha\theta$ associated nodal forces are defined as:

$$\langle F\rangle^\theta = \int_{V_e} f^\theta [\Omega]^T dV$$

and the corresponding element stiffness matrix:

$$[K]^{\theta\theta} = \alpha \int_{V_e} [\Omega]^T \otimes [\Omega] dV$$

which as to be added to $[K]^{uu}$.

Implementation of the debonding model

References

[1] V. Tvergaard, "Effect of Fiber Debonding in a Whisker–reinforced Metal", Material Science and Engineering, A125, 203—213 (1990)

[2] A. Needleman, "A Continuum Model for Void Nucleation by Inclusion Debonding", Journal of Applied Mechanics, 54, 525—531 (1987)

One defines an boundary (B) and an nset (N). At the beginning of the calculation the nodes of each set correspond are at the same location. Two corresponding nodes are use to build a debounding element. One also defines an contact zone (using soft contact algorithm); warning distance is set to zero. Its boundary corresponds to B and its nset (C) is empty at the beginning of the calculation. When a debounding element breaks the node belonging to N is transfered to C.

Each debounding element has a behavior and degrees of freedom corresponding to the displacements of its nodes ($\beta \in B$ and $\nu \in N$). The "grad" variable corresponds to the relative displacement $v = u_\nu - u_\beta$. The "flux" variable correspond to the forces. One then computes the projection of n on B (n') and the normal to B at n' (a) pointing outside of the boundary. a and v are defined so that $av = u_a > 0$, when there is separation between the boundary and the nset. The "tangent" direction (b) is determined by the projection of v on B.

$$b = \frac{v - (vn)n}{\|v - (vn)n\|}$$

if $v//n$ any vector can be chosen. In 3D the third vector of the local frame (c) is defined as

$$c = a \wedge b$$

On can define the column rotation matrix

$$R(a,b) \quad \texttt{2D} \qquad R = (a,b,c) \quad \texttt{3D}$$

R is the orthogonal operator between $(a, b, [c])$ and $(e_x, e_y, [e_z])$. In the local frame one computes the "grad" :

$$\texttt{grad} = ((vn)n, \|v - (vn)n\|, [0]) = (u_a, u_b, [0])$$

One then calls the behavior to compute the flux variable

$$\texttt{flux} = (T_a, T_b, [0])$$

Behavior $u_a > 0$

The behavior's internal variable is $\lambda_{\max}$. One computes λ

$$\lambda = \left[\left(\frac{u_a}{\delta_n} \right)^2 + \left(\frac{u_b}{\delta_t} \right)^2 \right]^{1/2}$$

If $\lambda < \lambda_{\max}$ the flux is computed as follows (elastic unloading)

$$\begin{aligned} T_a &= \frac{u_a}{\delta_n} F(\lambda_{\max}) \\ T_b &= \alpha \frac{u_b}{\delta_t} F(\lambda_{\max}) \end{aligned}$$

otherwise

$$\begin{aligned} T_a &= \frac{u_a}{\delta_n} F(\lambda) \\ T_b &= \alpha \frac{u_b}{\delta_t} F(\lambda) \\ \lambda_{\max} &= \lambda \end{aligned}$$

The consistent stiffness matrix is given as

$$D = \begin{pmatrix} \frac{\partial T_a}{\partial u_a} & \frac{\partial T_a}{\partial u_b} \\ \frac{\partial T_b}{\partial u_a} & \frac{\partial T_a}{\partial u_b} \end{pmatrix} = \begin{pmatrix} D_{aa} & D_{ab} \\ D_{ba} & D_{bb} \end{pmatrix}$$

If $\lambda < \lambda_{\max}$ one has

$$\begin{aligned} D_{aa} &= \frac{F(\lambda_{\max})}{\delta_n} \\ D_{ab} &= 0 \\ D_{ba} &= 0 \\ D_{bb} &= \alpha\frac{F(\lambda_{\max})}{\delta_t} \end{aligned}$$

If $\lambda > \lambda_{\max}$

$$\begin{aligned} D_{aa} &= \frac{F(\lambda)}{\delta_n} + \frac{F'(\lambda)}{\delta_n}\frac{1}{\lambda}\frac{u_a^2}{\delta_n^2} \\ D_{ab} &= \frac{F'(\lambda)}{\delta_n}\frac{1}{\lambda}\frac{u_a u_b}{\delta_t^2} \\ D_{ba} &= \alpha\frac{F'(\lambda)}{\delta_t}\frac{1}{\lambda}\frac{u_a u_b}{\delta_n^2} \\ D_{bb} &= \alpha\frac{F(\lambda)}{\delta_t} + \alpha\frac{F'(\lambda)}{\delta_t}\frac{1}{\lambda}\frac{u_b^2}{\delta_t^2} \end{aligned}$$

Note: Needleman assumes that T derives from a potential $\phi(u_a, u_b)$ so that

$$D_{ab} = \frac{\partial^2\phi}{\partial u_a \partial u_b} = D_{ba} = \frac{\partial^2\phi}{\partial u_b \partial u_a}$$

One therefore has

$$\frac{F'(\lambda)}{\delta_n}\frac{1}{\lambda}\frac{u_a u_b}{\delta_t^2} = \alpha\frac{F'(\lambda)}{\delta_t}\frac{1}{\lambda}\frac{u_a u_b}{\delta_n^2} \Rightarrow \alpha = \frac{\delta_n}{\delta_t}$$

Otherwise D is not symmetric. For 3D problems there is also a third direction which must be accounted for in the stiffness computation. Indeed $u_c = 0$ and $T_c = 0$, however some of the partial derivatives are not null.

$$\begin{aligned} \frac{\partial T_a}{\partial u_c} &= \left.\frac{\partial T_a}{\partial u_b}\right|_{u_b=0} = 0 \\ \frac{\partial T_b}{\partial u_c} &= \left.\frac{\partial T_b}{\partial u_b}\right|_{u_b=0} = \alpha\frac{F(\lambda)}{\delta_t} \\ \frac{\partial T_c}{\partial u_a} &= \left.\frac{\partial T_b}{\partial u_a}\right|_{u_b=0} = 0 \\ \frac{\partial T_c}{\partial u_b} &= \left.\frac{\partial T_b}{\partial u_b}\right|_{u_b=0} = \alpha\frac{F(\lambda)}{\delta_t} \\ \frac{\partial T_c}{\partial u_c} &= \left.\frac{\partial T_b}{\partial u_b}\right|_{u_b=0} = \alpha\frac{F(\lambda)}{\delta_t} \end{aligned}$$

Finally one gets

$$D = \begin{pmatrix} \frac{F(\lambda)}{\delta_n} + \frac{F'(\lambda)}{\delta_n}\frac{1}{\lambda}\frac{u_a^2}{\delta_n^2} & \frac{F'(\lambda)}{\delta_n}\frac{1}{\lambda}\frac{u_a u_b}{\delta_t^2} & 0 \\ \alpha\frac{F'(\lambda)}{\delta_t}\frac{1}{\lambda}\frac{u_a u_b}{\delta_n^2} & \alpha\frac{F(\lambda)}{\delta_t} + \alpha\frac{F'(\lambda)}{\delta_t}\frac{1}{\lambda}\frac{u_b^2}{\delta_t^2} & \alpha\frac{F(\lambda)}{\delta_t} \\ 0 & \alpha\frac{F(\lambda)}{\delta_t} & \alpha\frac{F(\lambda)}{\delta_t} \end{pmatrix}$$

for $\lambda < \lambda_{\max}$

$$D = \begin{pmatrix} \frac{F(\lambda)}{\delta_n} & 0 & 0 \\ 0 & \alpha\frac{F(\lambda)}{\delta_t} & 0 \\ 0 & 0 & \alpha\frac{F(\lambda)}{\delta_t} \end{pmatrix}$$

Behavior $u_a < 0$.
This corresponds to normal compression. In that case

$$T_a = F(0)\frac{u_a}{\delta_n}$$

$$\lambda = \left|\frac{u_b}{\delta_t}\right|$$

$$\begin{aligned} T_b &= \alpha F(\lambda_{\max})\frac{u_b}{\delta_t} \qquad \lambda < \lambda_{\max} \\ T_b &= \alpha F(\lambda)\frac{u_b}{\delta_t} \qquad \lambda > \lambda_{\max} \end{aligned}$$

The stiffness matrix is then

$$D = \begin{pmatrix} \frac{F(0)}{\delta_n} & 0 & 0 \\ 0 & \alpha\frac{F(\lambda_{\max})}{\delta_t} & 0 \\ 0 & 0 & \alpha\frac{F(\lambda_{\max})}{\delta_t} \end{pmatrix} \qquad \lambda < \lambda_{\max}$$

$$D = \begin{pmatrix} \frac{F(0)}{\delta_n} & 0 & 0 \\ 0 & \alpha\frac{F(\lambda)}{\delta_t} + \alpha\frac{F'(\lambda)}{\delta_t}\frac{1}{\lambda}\frac{u_b^2}{\delta_t^2} & \alpha\frac{F(\lambda)}{\delta_t} \\ 0 & \alpha\frac{F(\lambda)}{\delta_t} & \alpha\frac{F(\lambda)}{\delta_t} \end{pmatrix} \qquad \lambda > \lambda_{\max}$$

Failure Behavior
Failure occurs when $\lambda = 1$. Node ν is transfered to C. This must be done as a postprocessing after a time increment. It is impossible to have nodes switching from N to C and from C to N. The transfert can be done for λ slightly inferior to 1.

Back to the element !
The result of the behavior integration is therefore T and D. They both have to be convected into the element frame using R

$$\begin{aligned} T' &= RT \\ D' &= RD^T R \end{aligned}$$

The stiffness of the element is given by. The force acting on node ν is equal to $-T'$ and T' on node β

$$\frac{\partial(-T', T')}{\partial(u_\nu, u_\beta)} = \Delta' = \begin{pmatrix} -D' & D' \\ D' & -D' \end{pmatrix}$$

Spacial integration
The forces (in fact forces per unit area, i.e. pressure) and stiffnesses have to be integrated to get nodal reactions and element rigidity matrices. Integration is performed using the boundary (B). Nodal reactions can be calculated has for pressure boundary conditions. Let H be the interpolation matrix ($d \times (Nd)$, with d dimension and N number of nodes of the element) of an element of the boundary. One has $u(x) = H(x)\hat{u}$, where x is the position and $\hat{u}$ the nodal displacements vector (size Nd). One has

$$\hat{R} = \int_S H^T(x)T(x)dS = \int_S H^T(x)H(x)\,\hat{T}\,dS = \left[\int_S H^T(x)H(x)dS\right]\hat{T}$$

The elementary stiffness matrix is obtained as follows. Writting the principle of virtual work for the boundary one gets

$$\cdots + \Delta t \int_S T(x)\dot{u}(x)dS + \ldots$$

or after differentiation

$$\cdots + \int_S \left(\frac{\partial T}{\partial u}\delta u\right) \underbrace{(\dot{u}(x)\Delta t)}dS + \ldots$$

which can be written using the interpolation matrix H and $\frac{\partial T}{\partial u} = D'$

$$\int_S \delta\hat{u}^T H^T D'(x) H \delta\hat{u} dS = \delta\hat{u}^T \left[\int_S H^T D'(x) H dS\right] \delta\hat{u} = \delta\hat{u}^T \left[\int_S H^T \left(\sum_N \Psi_N D'_N\right) H dS\right] \delta\hat{u}$$

Finally the stiffness matrix is equal to

$$K_e = \int_S H^T(x) \left(\sum_N \Psi_N(x) D'_N\right) H(x) dS$$

where D'_N is the sitffness matrix for node N, and $\Psi_N(x)$ the shape function for the same node. The previous equations are for the nodes of the boundary only. In order to construct the forces and stiffnesses for all nodes, we consider a special debonding element consisting in (1) the dofs of the `nset` N, (2) the nodes of B (there are $2Nd$ dofs). nodal reaction vector and the stiffness matrix are

$$\begin{bmatrix} -\hat{R} \\ \hat{R} \end{bmatrix} \quad \texttt{and} \quad \begin{bmatrix} -K_e & K_e \\ K_e & -K_e \end{bmatrix}$$

Implementation

1. Start of the calculation
   (a) create debonding super element including B and N
      i. create "spring" like elements
      ii. create double layered boundary elements (for integration)
   (b) create soft contact element
2. Integration : manager is the (iteration) debonding super element
   (a) for each "spring" like element compute T' and D', and store them
   (b) use double layered boundary elements to perform integration and affect forces and stiffnesses
3. After convergence, check for broken "springs". If any transfer $\nu \in N$ in the contact element.

MECHANICAL_RIKS_ALGORITHM

— Class is derived from : **ALGORITHM**

• **Implementation of the Riks–Wempner method** [1]

This method is a modification of the standard Newton–Raphson (NR) method. It is started after the determinant of the global stiffness matrix becomes non positive (`***resolution riks`) or at the beginning if the option `always` is specified after `***resolution riks`.

— For each increment:

Δt the time increment is known.

The algorithm is as follows for iteration $i > 0$. (Δ indicate increment variations, δ indicate iterations variations).

- apply boundary conditions
- compute internal reactions and eventually the stiffness matrix K (`if_compute_stiffness`)
- form F_E and F_I vectors (external/internal forces)
- compute δu_i^I and δu_i^{II} solutions of

  $$K\delta u_i^{II} = F_I \qquad K\delta u_i^I = -F_E$$

  for the second resolution the inverse of K has to to be computed whatever the value of `if_compute_inverse`. If K is not recomputed δu_i^I is constant.
- compute $\delta\lambda_i$ as

  $$\delta\lambda_i = -\frac{\Delta u_{i-1}\delta u_i^{II}}{\Delta u_{i-1}\delta u_i^I + \Delta\lambda_{i-1}} \simeq -\frac{\Delta u_{i-1}\delta u_i^{II}}{\Delta u_{i-1}\delta u_i^I}$$

  this enforces

  $$(\Delta u_{i-1}, \Delta\lambda_{i-1}) \perp (\delta u_i, \delta\lambda_i)$$
- form the solution for the increment as

  $$\begin{aligned}\Delta u_i &= \Delta u_{i-1} + \delta\lambda_i\delta u_i^I + \delta u_i^{II}\\ \delta u_i &= \delta\lambda_i\delta u_i^I + \delta u_i^{II}\\ \Delta\lambda_i &= \Delta\lambda_{i-1} + \delta\lambda_i\end{aligned}$$

— For the first iteration $i = 0$

- $\Delta\lambda_0 = 0$
- solve system as usual

The previous increment Δu_{i-1} has to be stored.

[1] E. Ramm, "The Riks/Wempner Approach — An extension of the displacement control method in nonlinear analyses",

Chapter 11

Z-mat Programming Examples

FLOW example

Description:

ISOTROPIC example

Description:

Gen-evp example

Description:

BEHAVIOR example

Description:

ZebFront Example 2

Description:

FLOW example

Description:

Chapter 12

Z-set Programming Examples

BC example 1

Description:

FLOW example

Description:

MPC example

Description:

Element example

Description:

FLOW example

Description:

FLOW example

Description:

FLOW example

Description:

Chapter 13

Index

Index