The `MFront` book

Thomas Helfer, Maxence Wangermez, Jérémy Hure

06/2022

Contents

Chapter 1

Introduction

1.1 What is MFront ?

MFront is an open-source code generator dedicated to material knowledge [1, 2] developed by the French Alternative Energies and Atomic Energy Commission (CEA) and Électricité de France (EDF).

MFront can be downloaded from the github repository of the TFEL project described below:

https://github.com/thelfer/tfel

The main source of information about MFront is the website of the TFEL project:

https://thelfer.github.io/tfel/web/index.html

MFront is part of the PLEIADES numerical platform, which is devoted to multi-physics nuclear fuel simulations and is developed by CEA and its industrial partners EDF and Framatome.

Since it was released as an open-source project in 2014, MFront has been used in numerous applications covering a wide range of materials (ceramics, metals, concrete, woods, etc.) and physical phenomena (viscoplasticity, plasticity, damage, etc.).[1]

MFront categorises knowledge about material in three groups:

- **Material properties** are defined here as functions of the current state of the material. A typical example is the Young modulus of a material.
- **Behaviours** describe how a material evolves and reacts locally due to gradients inside the material. Here, the material reaction is associated with fluxes (or forces) thermodynamically conjugated to gradients. For instance, Fourier's law relates the heat flux to the temperature gradient. Mechanical behaviour in infinitesimal strain theory relates the stress and the strain and may describe (visco)elasticity, (visco)plasticity, or damage.
- **Point-wise models** describe the evolution of some internal state variables with the evolution of other state variables. Point-wise models may be seen as behaviours without gradients. Phase transition, swelling under irradiation or shrinkage due to dessication are examples of point-wise models.

Broadly speaking, the role of MFront is to:

1. simplify the implementation all those kinds of material knowledge,
2. make those implementations usable in a various solvers or languages,
3. encourage the writing of industrial-quality implementations. This include many aspects such as documentation, conventions, numerical performances, unit-testing, etc.

MFront achieves the first and third of these goals by different means that will unveiled in this book. The most important one is however to provide a set of so-called Domain Specific Languages (DSLs). The main part of this book is dedicated to describe the various DSLs available.

The achieve the second goal, MFront introduces the concept of interfaces through so-called interfaces (see Section 1.1.2 below for a list of available interfaces).

[1] See this page for a list of publications based on MFront: https://thelfer.github.io/tfel/web/publications.html

1.1.1 MFront usage in practice

In pratice, the following steps are typically involved when using MFront:

1. Write a text file containing the implementation of the considered material knowledge. Those files generally mixes MFront keywords (which starts with the at @ character) and blocks of code in C++.
2. Call MFront to interpret the file and to generate C++ sources for the interface of the considered language or solver.
3. As C++ is a compiled language, a build system is used to compile those sources into optimized shared libraries that are usable in the language or solver considered.

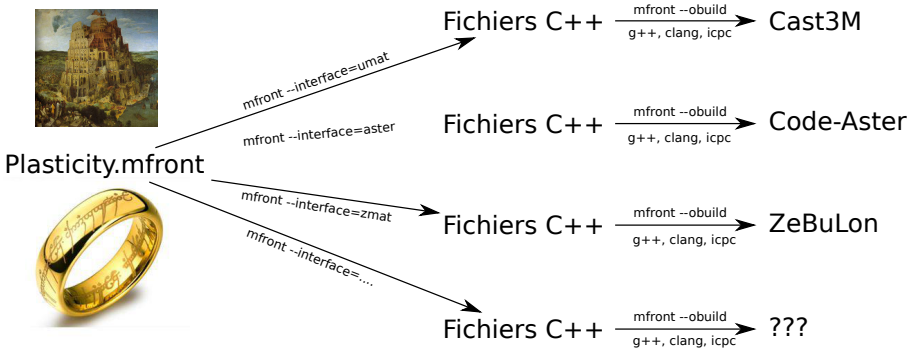

Figure 1.1: "MFront in action"

Those steps are illustrated in Figure 1.1.

1.1.2 Interfaces

MFront provides so-called interfaces to ensure that a material knowledge is portable, i.e. can be used in large number of contexts. For example, mechanical behaviours can be compiled for the following commercial or academic solvers: Cast3M, code_aster, Europlexus, Abaqus/Standard, Abaqus/Explicit, Ansys, AMITEX_FFTP, CalculiX, ZSet, DIANA FEA. Furthermore, thanks to the generic interface, those mechanical behaviours are also available in all solvers using the MFrontGenericInterfaceSupport projet (MGIS) [3], including: OpenGeoSys [4], MFEM-MGIS, MANTA [6], mgis.fenics, MoFEM, XPER, etc.

1.2 About the TFEL project

MFront is part of a project named TFEL [2]. Aside from MFront, TFEL project provides:

- The TFEL/Math library which aims at providing the mathematical objects and algorithms required for MFront.
- The TFEL/Material library which aims to provide various functions and classes associated with material modeling.
- The MTest simulation tool at the material point level. Simulation of axisymmetrical pipes is also supported.
- Various utilities around MFront, including mfront-query, mfront-doc, mfm, etc.

Although originally developed by the TFEL developers for their own usage focused on the unit testing of mechanical behaviours, MTest has proven useful to many users to simulate simple mechanical tests. Section 11 describes the main features of this tool.

Appendix A provides an overview of the tensorial operations provided by the TFEL/Math and TFEL/Material libraries.

1.3 About the MFrontGallery project

This project has two main almost orthogonal goals:

1. The first one is to show how solver developers may provide to their users a set of ready-to-use (mechanical) behaviours which can be parametrized by their users to match their needs.
2. The second one is to show how to set up a high-quality material knowledge management project based on MFront, able to meet the requirements of safety-critical studies.

While the first goal is common to all (mechanical) solvers, one originality of the MFrontGallery project is to address the second goal.

The MFrontGallery project also contains various high-quality MFront implementations. Those implementations may originate from the MFront tutorials, i.e. the MFront gallery page (hence the name of the project). This project is also meant to store various contributions of academic or industrial users of MFront willing to share their material knowledge and also benefit from the continuous integration process to guarantee that no regression would happen as MFront evolves.

In particular, the project provides:

- a cmake infrastructure that can be duplicated in (academic or industrial) derived projects. This infrastructure allows:
  - to compile MFront sources using all interfaces supported by MFront. - to execute unit tests based on MTest. Those unit tests generate XML result files conforming to the JUnit standard that can readily be used by continuous integration platforms such as jenkins.
  - generate the documentation associated with the stored implementations.

  This page describes how to create a derived project based on the same infrastructure as the MFrontGallery.
- a documentation of best practices to handle material knowledge implemented using MFront implementations
- a set of high-quality MFront implementations.

1.4 About the MGIS project

This project aims at proving tools (functions, classes, bindings, etc…) to handle behaviours generated using MFront generic interface. For information about MFront. Those tools are meant to be used by (FEM, FFT, etc.) solver developers.

The following projects used MGIS or can be coupled with MGIS:

- code-aster
- OpenGeoSys. See this talk for details.
- FEniCS. Coupling with FEniCS can be made using the python bindings (See this talk and this tutorial for details). However, we recommend using the higher level mgis.fenics module.
- MFEM through the MFEM-MGIS. See this talk for details.
- MoFEM. See this talk for details.
- Europlexus
- XPer. See this talk for details.
- MBDyn. See this demo.
- The (yet to be released) solver MANTA developed at CEA [6].

MGIS is written in C++, but the following bindings are available:

- c binding,
- python binding,
- fortran binding.

Chapter 2

C++ survival guide for MFront users

MFront is built on top of C++ and a minimal knowledge of this language is required to efficiently use it [7].

2.1 A compiled language

First of all, C++ is a compiled programming language, contrary to scripting languages such as python. This means that a C++ compiler, such as g++, clang++, icpx or cl.exe (Visual Studio), must be available to transform a C++ source into an object file in (machine-level) assembly language. Objects files are gathered into executables or shared libraries by the linker.

MFront usually generates C++ files and handles the compilation of those files into shared libraries that can be plugged in the solver of interest.

Let us consider the simpliest C++ program:

```
#include <cstdlib>
#include <iostream>
int main(){
  std::cout << "Hello, world !\n";
  return EXIT_SUCCESS;
}
```

This program can be compiled and linked into an executable using g++ as follows:

```
$ g++ hello.cxx -o hello
$ ./hello
Hello, world !
```

2.2 About the syntax

C++ is case sensitive.

2.3 Comments

C++ supports two syntaxes for comments:

- a comment may start with a double slash (//) and ends a the end of the line:

```
// declaration of variable a as an integer
auto a = int{};
```

- a (possibly multi-line) comment starts with a slah followed by a star (/* and ends by a star followed by a slash (*/)

```
/*
  This introduces the variable a and initializes it to twelve
 */
auto a = int{12};
```

2.4 End of an instruction

The end of an instruction is marked by a semi-colon `;`. There can be several instructions on the same line and an instruction can span over several lines.

2.5 Code-blocks

A block of code begins with an open curly bracket `{` and ends with a closing curly bracket `}`.

2.6 Declaration of variables (historical syntax)

C++ has several syntaxes to define a new variable. Historically, variables were declare as follows:

```
<type> <variable_name>;
```

For instance, a integer can be declared as follows:

```
int a;
```

Note that the value of `a` is undefined in this case and can contain any value in memory. A general recommandation is to always initialize a variable to a value. The latter example can be modified as follows:

```
int a = 0;
```

Contrary to other languages, C++ is statically typed, which means that the type of the variable `a` can't change.

The lifetime of a variable is defined by it's scope. Generally this scope is associated with the end of the current code block.

```
{
  int a = 0;
} // a is destroyed here
// a is not accessible here
```

A variable is accessible in sub block of codes :

```
{
  int a = 0;
  {
    a = 12;
  }
  // now a egals 12
}
```

Hidden variables
In theory, a variable with the same name can be defined a a sub block of code, as follows:

```
  int a = 12;
{
 // this variable hides the previous one
  int a = 0;
}
// here a still egals 12
```

Hidding variables is a bad practice and most compilers would emit a warning when this happens.

2.7 Constant variables

The `const` keyword allows to specify that the value of a variable can't change after its declaration.

```
const int a = 12;
```

A general recommandation is to use the `const` keyword as often as possible.

2.8 Variables known at compile-time

The `constexpr` keyword allows to specify that the value of a variable is known at compile time.

```
constexpr double pi = std::numbers::pi_v<double>;
```

Note that `constexpr` implies `const`.

2.9 Declaration using the `auto` keyword

The `auto` keyword lets the compiler deduces the type of a variable for the right-hand size of an assignement:

```
/* the type of a is deduced from the right hand size
    which is an integer */
const auto a = 12+13;
```

One advantage of using this syntax is that a variable must be initialized to a value.

2.10 Plain old data types

`C++` introduces some basic types, also called plain old data types. From the `MFront` perspective, the most useful ones are:

- `float`: single precision floating point numbers,
- `double`: double precision floating point numbers,
- `long double`: floating point numbers in extended precision,
- `short`: small integers
- `unsigned short`: small positive integers
- `int`: integers
- `unsigend int`: positive integers
- etc..

Concerning integers, the standard does not specify their sizes (i.e. the range that they can represent).

> **Floating-point numbers in MFront**
> An `MFront` source file may be used in a variety of context. While most solvers uses double precision arithmetics, some do not, such as `Abaqus/Explicit` which, by default, works in single precision.
> `MFront` thus introduces the `real` alias which is replaces by the floating point number type depending on the final context.

2.11 Function calls

Arguments of a function are specified using parenthesis:

```
const auto c = cos(theta);
```

2.12 Mathematical functions

Any `MFront` implementation automatically includes the `cmath` header, which declares most of the mathematical functions defined by the `C++` standard. The most useful ones are:

- the trigonometric functions: `cos`, `sin`, `tan`,
- the exponential `exp`,
- the natural logarithm `ln`,
- the square root `sqrt`,
- the cubic root `cbrt`,
- the exponentiation `pow`,
- etc.

Those functions ared declared in the `std` namespace (see next paragraph).

The power function from the TFEL/Math library
The pow function deserves some care, in particular with integer exponent.
In many cases, it is more efficient to use the power function provided by the TFEL/Math library. For instance, the square of a variable x can be computed as:

```
// strictly equivalent to x * x
const auto y = power<2>(x);
```

Note that the exponent is passed using a special syntax (as a template parameter). The description of the syntax is out of the scope of this section.
The power function also treats fractional exponents:

```
// strictly equivalent to sqrt(x * x *x)
const auto y = power<3, 2>(x);
```

2.13 Namespaces

To avoid conflicts between libraries, functions (and data structures) can be declared in so-called namespace. The std namespace is reserved by the standard.

The TFEL project declares several namespaces, such as tfel::math (from the TFEL/Math library) and tfel::material from the TFEL/Material library).

In the code blocks of an MFront files, all the contents of the std, tfel::math and tfel::material libraries are automatically available. Hence, its is not required to write:

```
const auto c = std::cos(theta);
```

as the std:: qualifier is optional.

Most MFront users won't need to take care about namespaces. This is not the case if one needs to extend the TFEL/Math or TFEL/Material libraries.

2.14 Printing information on the terminal

C++ provides several streams to print information on the terminal, namely std::cout (standard output), std::cerr (error stream) and std::log (the standard output by default).

For debugging purposes, we strongly advise to use std::cerr. Contrary to std::cout, std::cerr does not use an intermediate buffer: it is thus less efficient but more predicatable than std::cout.

2.15 Data structures and classes

In C++, the user (and the standard library) has the ability to define new types. For instance, the TFEL/Math library defines a set of tensorial types.

Chapter 3

A basic introduction to `MFront` for material properties

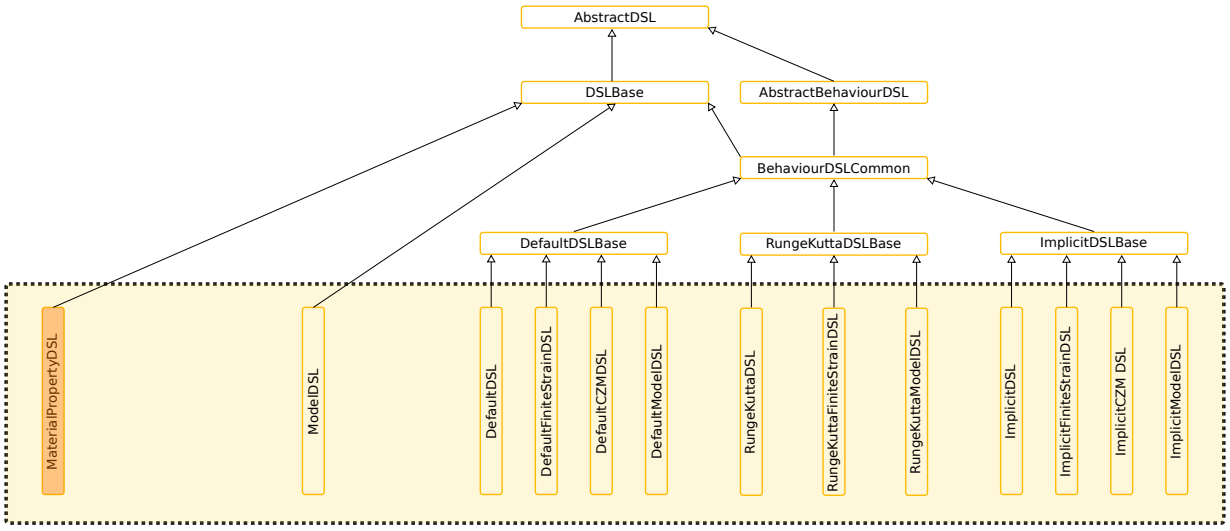

Figure 3.1: Inheritance between Domain Specific Languages (DSL). The `MaterialProperty` DSL described in this chapter is highlighted in red.

3.1 Introduction

Material properties are function of some state variables which describe the current thermodynamical state of the material.

Compared to behaviours and models, which often require to solve a system of ordinary differential equations to describe the evolution of the state variables describing the material, material properties are conceptually much simpler. However, their implementations, while straightforward, allow to introduce some key concepts of `MFront`, making a perfect introduction for beginners.

More precisely, this chapter is dedicated to the `MaterialProperty` Domain Specific Language (DSL). Figure 3.1 highlight the fact that material properties are treated quite differently than models and behaviours.

This tutorial considers the example of the Young's modulus of uranium dioxide, which can be represented, according to Martin [8], by the following expression:

$$E(T, f) = 2.2693\,10^{11}\,(1 - 2.5\,f)\,(1 - 6.786\,10^{-5}\,T - 4.23\,10^{-8}\,T^2) \tag{3.1}$$

where T is the temperature (Kelvin) and f is the porosity. This expression is assumed valid for a temperature in the range $[273.15 : 2610.15]$ and a porosity in the range $[0 : 0.3]$.

The examples shown in this chapter can be downloaded on this repository: https://github.com/thelfer/MFrontBookExamples/tree/master/material-properties. In particular, the final version of the implementation is

available in the file UO2_YoungModulus_Martin1989.mfront. We however strongly recommend not downloading this file before having implemented this material property at least once, step by step, as proposed by this chapter.

About the MFrontBookExamples project
The MFrontBookExamples project contains all the examples of this book. It is derived from the MFront-Gallery project.

3.2 First implementation and first tests in Python

3.2.1 Creating a first MFront file

Expression (3.1) can be implemented by the following code (file YoungModulus.mfront):

```
@DSL MaterialLaw;
@Law YoungModulus;
@Input T, f;
@Function {
  res = 2.2693e11 * (1 - 2.5 * f) * (1 - 6.786e-05 * T - 4.23e-08 * T * T);
}
```

This code is assumed to be in a standard text file called YoungModulus.mfront which can be edited using any text editor.

About editing MFront files
Most text editors have special modes for C++ highlighting and we strongly recommand to associate files with the extension .mfront with C++ mode.
Note that an experimental text editor called tfel-editor with specific support for MFront is also available here: https://github.com/thelfer/tfel-editor

3.2.2 Compilation of the MFront file as a python module

Assuming that MFront was compiled with support for the python interface, the YoungModulus.mfront can now be used to generate a python module in two steps:

1. Generate sources files from the MFront file. This is done by the following call to the mfront executable:

   ```
   $ mfront --interface=python YoungModulus.mfront
   ```

   MFront will generally generate two sub-directory named respectively include and src.

2. Compile the generated sources in shared libraries (here a single python module):

   ```
   $ mfront --obuild
   Treating target : all
   The following library has been built :
   - materiallaw.so :  YoungModulus
   ```

   Note that the materiallaw shared library is compiled in the src directory. Its name may differ depending on the operating system considered (typically MacOS, Linux or Windows).

   Note also that obuild stands for "optimised build," i.e. MFront automatically specifies compiler flags selected for performances.

Those two steps can be done with this single instruction:

```
$ mfront --obuild --interface=python YoungModulus.mfront
```

At any of those two steps, errors may happen. Section 3.7.1 provides a brief overview on how to handle compiling errors.

3.2.3 Testing

At this stage, a shared library called `materiallaw.so` shall have been generated by `MFront` in the `src` directory. This shared library is a module directly usable in `python` as follows:

```
import materiallaw
import numpy as np
from matplotlib import pyplot as plt

T = np.linspace(400, 1600)
E = np.array([materiallaw.YoungModulus(Ti, 0.1) for Ti in T])
plt.xlabel("Temperature (K)")
plt.ylabel("Young's modulus (Pa)")
plt.plot(T, E)
plt.show()
```

Location of the module
Note that `python` to find the `materiallaw.so` module. This can be ensured by:
1. Executing this script in the `src` directory
2. Updating the `PYTHONPATH` environment variable to add the `src` directory.

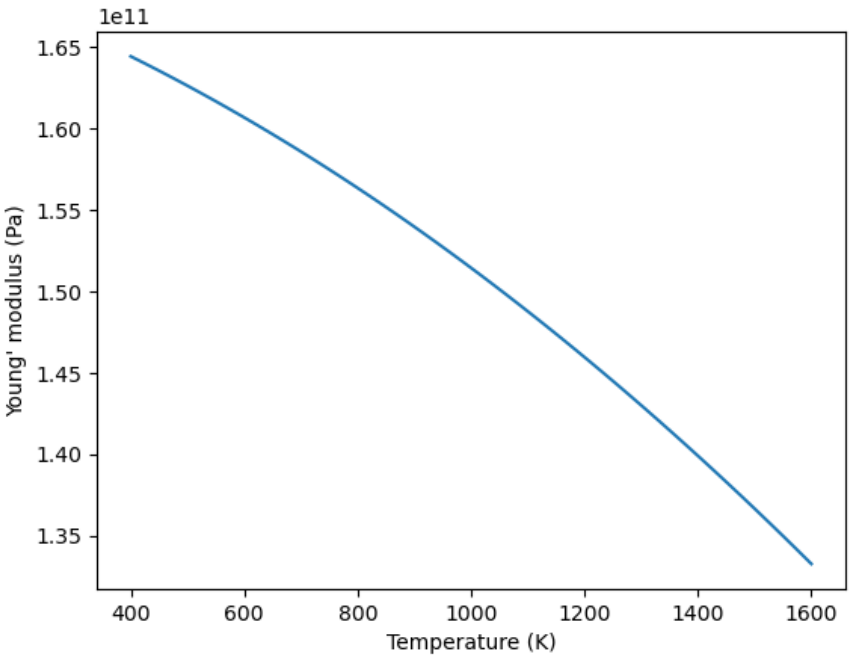

Figure 3.2: Plot of the material property using `MatPlotlib`

The result of the previous script is shown on Figure 3.2.

3.2.4 Description of the `MFront` implementation

Listing in Section 3.2.1 shows that an `MFront` file contains special keywords starting with the at sign `@`.

3.2.4.1 The `@DSL` keyword

The first line, starting with the `@DSL` keyword, defines the domain specific language used by `MFront` to interpret the file. Domain specific languages define the kind of material knowledge treated by the file (material property, behaviour, model) and also, for behaviours, a class of numerical algorithms used to integrate the constitutive equations over a time step. Each DSL has its own conventions and defines its own keywords (however, many keywords are common to several DSLs).

About domain specific languages and getting help
The list of DSLs, as well as a small description of each DSL, can be retrieved as follows:

```
$ mfront --list-dsl
available dsl:
- DefaultCZMDSL                 : this parser is the most generic one as it
does not make any restriction on the behaviour or the integration method that
may be used.
....
```

The list of keywords associated with a DSL can be requested as follows:

```
$ mfront --help-keywords=<DSLName>
```

where `DSLName` is the name of the DSL. For example, the list of keywords defined by the `MaterialLaw` DSL used in this tutorial is given by:

```
$ mfront --help-keywords-list=MaterialLaw
- @Author          (documented)
- @Bounds          (documented)
- @Constant        (documented)
- @DSL             (documented)
- ...
```

Finally, the documentation of a keyword can be displayed by:

```
$ mfront --help-keyword=MaterialLaw:@Input
The `@Input` keyword specifies one or several inputs of a material
law. This keyword is followed by the names of the inputs, separated by
commas.
....
```

3.2.4.2 The `@Law` keyword

The second line defines the name of the material property. For this first implementation, the very generic and unspecific name `YoungModulus` is used.

This name can be replaced by a more explicit name as `YoungModulus` is a poor name for the material law.

Some restrictions apply to the name of the law, see the documentation of the `@Law` keyword for details.

A good practice for the choice of a more explicit name for the material property is to add a descriptive and explicit identifier. For example, one may add a reference to the original paper as follows:

```
@Law YoungModulus_Martin1989; // name of the material property
```

Note that the generated name of the function also changes, so the previous test case must be adapted. This is stated by the ouput of `MFront`:

```
$ mfront --obuild --interface=python YoungModulus_Martin1989.mfront
Treating target : all
The following libraries have been built :
- materiallaw.so :  YoungModulus YoungModulus_Martin1989
```

Note that the previous implementation is still present. The user may want to remove the `src` and `include` directories to avoid conflicts and avoid using the previous implementation by mistake.

Note that we also recommend to change the name of the `MFront` file to reflect the name of the material law, although this is not required by `MFront`.

3.2.4.3 The `@Input` keyword

The third line defines the two input variables of the material property, named respectively `T` and `f`.

3.2.4.4 The `@Function` keyword

The implementation of Expression (3.1) is done in a code block introduced by the `@Function` keyword. By default, this code block must store the result of the material property in a variable called `res`.

It is worth noting that this implementation here is reasonably close to Equation (3.1).

The code introduced by `@Function` is standard `C++` with a few implicit additions, such as:

- the declaration of some `MFront` types, such as `real`, `stress`, `temperature`, etc. (see Section 3.3.8),
- the declaration of external material properties (not discussed in this tutorial, see the documentation of the `@MaterialLaw` keyword for details),
- the fact that every function of the standard library is automatically used (i.e. the line `using namespace std;` is inserted before the code block).

Most of the time, only a very limited subset of the `C++` language is required to use `MFront`.

Note that the `cmath` header, which contains the declaration of all the standard mathematical functions, is automatically included. Usage of external libraries, although not discussed in this tutorial, is possible using the `@Includes` and `@Library` keywords.

> **On the usage of `C++` exceptions**
> In most interfaces (except the `C++` interface notably), the code block is also placed inside a `try/catch` block to avoid the propagation of `C++` exceptions in languages and solvers that can't handle them. Interfaces are responsible for translating `C++` exceptions into errors in an appropriate way if possible. For example, most interfaces will handle exceptions derived from `std::exception` specifically and use the `what` method to retrieve the error message.

3.3 Improvements

At this stage, the `MFront`'s implementation is barely working. This section is devoted to the improvements that must be considered to have a production-ready implementation.

3.3.1 Associating the material property to a material

At this stage, nothing indicates that the material property describes uranium dioxide. This can be done using the `@Material` keyword as follows:

```
@Material UO2; // material name
```

Associating a material name to the material property has two effects:

- it changes the name of the generated function,
- it changes the name of the library generated.

This can be readily seen on the output of `MFront`:

```
$ mfront --obuild --interface=python UO2_YoungModulus_Martin1989.mfront
Treating target : all
The following libraries have been built :
- uo2.so :  UO2_YoungModulus_Martin1989
```

At this stage, a shared library called `uo2.so` shall have been generated by `MFront` in the `src` directory and shall be directly usable in `python` as follows:

```
from uo2 import UO2_YoungModulus_Martin1989 as UO2YoungModulus
import numpy as np
from matplotlib import pyplot as plt

T = np.linspace(400, 1600)
E = np.array([UO2YoungModulus(Ti, 0.1) for Ti in T])
plt.xlabel("Temperature (K)")
plt.ylabel("Young's modulus (Pa)")
plt.plot(T, E)
plt.show()
```

The `@Library` keyword
The `@Library` provides more control on the name of the generated shared library.
However, we do not recommend to use this keyword in the `MFront` source. This keyword can be specified on the command line as follows:

```
$ mfront --@Library=UraniumDioxide --obuild --interface=python \
  UO2_YoungModulus_Martin1989.mfront
Treating target : all
The following libraries have been built :
- uraniumdioxide.so :  UO2_YoungModulus_Martin1989
```

3.3.2 Changing the name of the output

`res` is a poor name for the ouput variable. It can be changed using the `@Ouput` keyword as follows:

```
@Output E;
```

As a consequence, the name body of the function must be changed to reflect the change in the name of the output:

```
@Function {
  E = 2.2693e11 * (1 - 2.5 * f) * (1 - 6.786e-05 * T - 4.23e-08 * T * T);
}
```

3.3.3 Physical bounds and standard bounds

3.3.3.1 Physical bounds

At this stage, the function barely works, but can be used very inappropriately. For example, one may request the value of the Young's modulus for a negative temperature, something which in our opinion shall lead to an error to indicate to the user that something very wrong is happening their code.

Indeed, many quantities are intrinsically bounded. In our example, the temperature (in Kelvin) can't be negative and the porosity shall be bounded in the range $[0 : 1]$. Those bounds are depicted as *physical bounds* in `MFront`.

Such bounds can be introduced using the `@PhysicalBounds` keyword as follows:

```
@PhysicalBounds T in [0:*[;
@PhysicalBounds f in [0:1];
```

where the `*` character means infinity (and is followed by an open-bracket).

A violation of the physical bounds is **always** a cause for the calculation stop and the return of an error.

3.3.3.2 Standard bounds

Another issue is the fact that the considered material property is only reliable on a limited range of temperature and porosity. Outside of this domain of validity, the prediction may become inaccurate.

This domain of validity is described in `MFront` by (standard) bounds, introduced by the `@Bounds` keyword.

In our example, the prediction are considered reliable for temperatures in the range $[273.15 : 2610.15]$:

```
@Bounds T in [273.15:2610.15]; // Validity range
```

`MFront` defines three policies to let the solver choose how to treat an out-of-bounds evaluation:

- `None`: nothing is done,
- `Warning`: the user shall be informed but the computation is performed as usual. How the user is informed depends on the interface considered,
- `Strict`: the computation are stopped and an error is reported. The mechanism used to stop the computation depends on the interface considered.

`MFront` automatically selects the `None` policy by default.

3.3.3.2.1 Changing the default policy with DSL options - Since Version 4.1, this behaviour can now be changed by using the `default_out_of_bounds_policy` string option of the `MaterialLaw` DSL as follows:

```
@DSL MaterialLaw {
  default_out_of_bounds_policy: "Strict"
};
```

List of available DSL options
DSL options are a powerful tool to modify the default behaviour of `MFront`. The list of options associated with a DSL, as well as a short description of each option, can be retrieved using the `--list-dsl-options` command line argument, as follows:

```
$ mfront --list-dsl-options=MaterialLaw
- default_out_of_bounds_policy  : string specifying the default out
  of bounds policy. Allowed values are `None`, `Warning`, `Strict`.
- out_of_bounds_policy_runtime_modification: boolean stating if the
  runtime modification of the out of bounds policy is allowed.
- parameters_as_static_variables: boolean stating if parameters
  shall be treated as static variables.
- parameters_initialization_from_file: boolean stating if the values
  of parameters can be changed from an external file.
- build_identifier              : string specifying a build identifier.
  This option shall only be specified on the command line.
```

We do not recommend to hard coding DSL options in the source file as `MFront` files must be sharable across different users which may have different needs.

3.3.3.2.2 Changing the default policy with command line - `DSL` options can be specified on the command line as follows:

```
$ mfront --obuild --interface=python                    \
  --dsl-option=default_out_of_bounds_policy:'"Strict"' \
  UO2_YoungModulus_Martin1989.mfront
```

Note that a double-quote is required here to pass a string option.

Statement of need
DSL options have been introduced to customize `MFront`. While most users may want to control the out-of-bounds policy at runtime, this may not be compatible with quality assurance in safety critical studies.
Consider a nuclear fuel performance code which delivers a set of well-qualified material properties to its users. The developpers of the fuel performance code must guarantee the proper the usage of their code. In particular, the material properties must be used inside their validity domain. A `Strict` policy may be a convenient choice.
Users shall thus not be able to use the code outside its validation domain without explicit stating it in their input files using a specific keyword. In this case, the developpers of the fuel performance code are no more responsible of the results obtained and may explicitly state it in the result files by a warning (for example).

3.3.3.3 Testing in `python`

After specifying the physical bounds of our variables, the implementation is as follows:

```
@DSL MaterialLaw;
@Material UO2;
@Law YoungModulus_Martin1989;
@Output E;
@Input T, f;
@PhysicalBounds T in [0:*[;    // Temperature is positive
@PhysicalBounds f in [0:1.];   // Porosity is positive and lower than one
@Bounds T in [273.15:2610.15]; // Validity range
```

```
@Function {
  E = 2.2693e11 * (1 - 2.5 * f) * (1 - 6.786e-05 * T - 4.23e-08 * T * T);
}
```

After recompiling the python module, we may try to evaluate the Young's modulus for a negative temperature. The output is the following:

```
>>> import uo2
>>> print(uo2.UO2_YoungModulus_Martin1989(-1, 0.1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: YoungModulus : T is below its physical lower bound (-1<0).
```

In python, violation of physical bounds is reported using the standard exception mechanism of Python. More precisely, an exception of the RuntimeError type is thrown.

With the default policy, an out-of-bounds evaluation does not trigger any specific behaviour:

```
>>> print(uo2.YoungModulus_Martin1989(3000, 0.1))
70754504700.0
```

3.3.3.3.1 Changing the out-of-bounds policy - By default, the out-of-bounds policy can be changed using the environment variable PYTHON_OUT_OF_BOUNDS_POLICY as follows:

```
>>> import os
>>> os.environ['PYTHON_OUT_OF_BOUNDS_POLICY'] = 'WARNING'
>>> print(uo2.YoungModulus_Martin1989(3000, 0.1))
YoungModulus : T is over its upper bound (3000>2610.15).
70754504700.0
```

Here, one sees that a warning is displayed in the terminal on the standard error stream.

If the out-of-bounds policy is set to STRICT, violation of standard bounds is treated as the violation of physical bounds:

```
>>> os.environ['PYTHON_OUT_OF_BOUNDS_POLICY'] = 'STRICT'
>>> print(uo2.YoungModulus_Martin1989(3000, 0.1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: YoungModulus : T is over its upper bound (3000>2610.15).
```

3.3.3.3.2 Disabling the runtime modification of the out-of-bounds policy - Since Version 4.1, it is possible to use the out_of_bounds_policy_runtime_modification DSL option boolean to enable or disable the runtime modification of the out-of-bounds policy.

If this option is set to false, the environment variable PYTHON_OUT_OF_BOUNDS_POLICY will have any effect.

3.3.4 Declaring the unit system

The @UnitSystem keyword declares that the state variables, external state variables an parameters are expressed in a given unit system. In the current version of MFront, the only supported unit system is the international system of units, denoted SI.

This keyword is used as follows:

```
@UnitSystem SI;
```

One advantage of declaring an unit system is that physical bounds of variables associated with a glossary entry can automatically be declared by MFront. For instance the declaration of the physical bounds for the temperature and the porosity is no more required.

3.3.5 Documenting the variables for the calling solvers

The names of the inputs variables are very generic and not self-explanatory. For example, nothing indicates to the reader that f stands for the porosity of the material. More precisely, MFront exports information about the

material property that can be used by the calling solver. Among others, these information include the name of the inputs and the name of the output, as well as description of those variables.

To emphasize this point, let us consider a handy tool called `mfront-query` which can be used to request all kind of information about an `MFront` file from the command line. Using `mfront-query` to list the inputs of this implementation leads to this disappointing and unuseful result:

```
$ mfront-query --inputs YoungModulus.mfront
- T
- f
```

While `MFront`'s users hardly ever use `mfront-query` directly, its mention here is meant to highlight the fact that an `MFront` file (or the shared library generated thanks to this `MFront` file, see below) generally must be self-descriptive and provide meaningful information to the solver that will use it.

3.3.5.1 About using explicit variable names

A first idea to solve this issue would be to used explicit variable names such as `Temperature` or `Porosity`, instead of `T` and `f` respectively. However, this solution rapidely becomes cumbersome in practice. We do not recommend this solution.

3.3.5.2 About standard C++ comments

Another idea would be to highlight the meaning of those variables using `C++` comments as follows:

```
// T stands for the temperature
@Input T;
// f stands for the porosity
@Input f;
```

However, those comments are not associated to the variables by `MFront`, so the output of `mfront-query` is unchanged.

3.3.5.3 `doxygen`-like comments

`MFront` supports `doxygen`-like comments to document variables, as follows:

```
//! T stands for the temperature
@Input T;
//! f stands for the porosity
@Input f;
```

In this case, the output of `mfront-query` is a bit more informative:

```
$ mfront-query --inputs YoungModulus.mfront
- T: T stands for the temperature
- f: f stands for the porosity
```

3.3.5.4 The `TFEL` glossary and the `setGlossaryName` method

Still, all the previous solutions lack standardisation: all implementations using the temperature shall state explicitly and unambiguously that they are using this quantity.

The `TFEL` glossary defines a set of uniquely defined names that can be used to qualify a variable. A variable can be associated with a glossary name using the `setGlossaryName` as follows:

```
@Input T;
T.setGlossaryName("Temperature");
@Input f;
f.setGlossaryName("Porosity");
```

The glossary name automatically defines the **external name** of the variable and automatically adds an appropriate documentation. The output of `mfront-query` is now:

```
$ mfront-query --inputs YoungModulus.mfront
- Temperature (T): The temperature
- Porosity (f): Porosity of the material
```

Variables not covered by the TFEL glossary and the setEntryName method
The TFEL glossary is finite. If a variable is not described by a glossary entry, we highly recommend to give it an explicit external name using the setEntryName and to document it using doxygen-like comments.

3.3.6 Parameters

Sensitivity analysis, uncertainties propagations, re-identification, require to be able to modify the coefficients of the material property.

MFront introduces the concept of *parameters*: a parameter is a variable which has a default value that can be changed at runtime. In practice, parameters can be seen as global variables. The way to change the value of a parameter depends on the interface considered.

Parameters are introduced using the @Parameter keyword. The implementation can be turned into a more explicit version as follows:

```
@Parameter E0 = 2.2693e11;
@Parameter dE_dT =  -1.53994698e7;
@Parameter d2E_dT2 = -1.9198278e4;
@Parameter f0 = 0.4;

@Function {
  E = (1 - f / f0) * (E0 + dE_dT * T + (d2E_dT2 / 2) * T * T);
}
```

3.3.6.1 Modification of parameters

3.3.6.1.1 Using external txt files - Parameters can usually be changed using an external txt file, although this may depend on the interface considered.

Since Version 4.1, the name of expected parameters file can be retrieved using mfront-query as follows:

```
$ mfront-query --parameters-file UO2_YoungModulus_Martin1989.mfront
UO2_YoungModulus_Martin1989-parameters.txt
```

This file has the following structure:

```
<parameter name> <new parameter value>
```

For example, the file UO2_YoungModulus_Martin1989-parameters.txt may looks like:

```
## new material parameters
E0 2.2693e11
dE_dT  -1.53994698e7
d2E_dT2 -1.9198278e4
f0 0.4
```

This file may contain commented lines starting with the # character.

Note that this file is read, if present in the current working directory, only at the very first evaluation of the material property. As a consequence, changing this file after this first evaluation will not have any effect.

3.3.6.1.2 Using the ExternalLibraryManager - The ExternalLibraryManager class, provided by the TFEL/System library provides the setParameter method which can be used to modify the parameters dynamically.

This method expects that the interface exports a dedicated function. While this is the case for most interfaces, it has only been introduced in the python interface in Version 4.1.

This method can be used as follows:

```
>>> from uo2 import UO2_YoungModulus_Martin1989
>>> print(UO2_YoungModulus_Martin1989(800, 0.1))
156350208000.0
>>> from tfel.system import ExternalLibraryManager
>>> elm = ExternalLibraryManager.getExternalLibraryManager()
```

```
>>> elm.setParameter('uo2.so', 'UO2_YoungModulus_Martin1989', 'E0', 2e11)
>>> print(UO2_YoungModulus_Martin1989(800, 0.1))
136152708000.0
```

3.3.6.1.3 Disabling the modification of parameter from external txt file - Since Version 4.1, initializing parameters from an external file can be disabled using the `parameters_initialization_from_file` boolean DSL option:

```
$ mfront --obuild --interface=python                          \
         --dsl-option=parameters_initialization_from_file:false \
         UO2_YoungModulus_Martini1989.mfront
```

This option has mostly been introduced to let solver developpers have control on the users ability to modify parameters of well-qualified material properties: if such parameter is modified, strict assurance quality of the solver is compromised.

3.3.6.1.4 Disabling the runtime modification of parameters - Since Version 4.1, it is possible to use the boolean DSL option `parameters_as_static_variables` in order to modify the way parameters are processed:

```
$ mfront --obuild --interface=python                    \
         --dsl-option=parameters_as_static_variables:true \
         UO2_YoungModulus_Martini1989.mfront
```

When treated as static variables, parameters can not be modified at runtime (neither using an external txt file nor using the `setParameter` function of the `ExternalLibraryManager` class).

3.3.6.2 Effect on the runtime performances

Modification of parameters may have runtime performances since a global variable is used to store the values of the parameters.

3.3.7 Other metadata for quality assurance

Metadata describing the author of the implementation, the date of the implementation and a short description can be added by the `@Author`, `@Date` and `@Description` keywords respectively.

For example, those information can be added as follows:

```
@Author   T. Helfer;  // author name
@Date     04/04/2014; // implentation date
@Description          // detailled description
{
  The elastic constants of polycrystalline UO2 and
  (U, Pu) mixed oxides: a review and recommendations
  Martin, DG
  High Temperatures. High Pressures, 1989
};
```

Those information will be exported by `MFront` and can be retrieved by the calling solver.

3.3.8 Explicit coding: add types and quantity support

3.3.8.1 The `real` type alias

By default, the input variables and the output variable will be defined as standard floating-point values with single precision (i.e. the `float` type in `C++`), double precision (`double` type in `C++`) or extended precision (`long double` type in `C++`) depending on the interface selected.

`MFront` always defines a type alias named `real` which corresponds to the floating point type effectively used.

We strongly encourage `MFront` users to never use directly any of the `C++` standard types and use the `real` type alias instead.

3.3.8.2 Default variable type in the MaterialLaw DSL

By default, variables in the MaterialLaw DSL have the real type. Hence the declaration:

```
@Input f;
```

is equivalent to:

```
@Input real f;
```

3.3.8.3 Explicit type aliases

MFront defines a set of type aliases, listed on this page, such as temperature, strain, stress, etc.

Hence, the temperature could be defined as follows:

```
@Input temperature T;
```

3.3.8.4 Using quantities

By default, temperature, strain, stress are defined as type aliases to floating point number type used by the interface.

The keyword @UseQT allows the usage of quantities, i.e. floatting point values associated with units. Quantities allow the compiler to perform *dimensional analysis* at compile-time.

It must clearly emphasized that quantities are only used *inside* the body of the function and does not change the interface of the generated functions.

Note that quantities are implemented by the TFEL/Math library and supported in MFront only since Version 4.0.

3.3.8.4.1 Error generated by invalid operations on quantities - The following code will not compile if quantities are set:

```
const auto a = strain{1e-8};
const auto b = stress{10e6};
const auto c = a + b;
```

For instance, the gcc compiler generates the following error:

```
TFEL/Math/Quantity/qtOperations.hxx:302:19: error: static assertion failed:
invalid operation
    static_assert(std::is_same_v<UnitType, UnitType2>, "invalid operation");
```

3.3.8.4.2 The derivative_type metafunction - Defining the type of the parameters E0 and f0 is straightforward:

```
@Parameter stress E0 = 2.2693e11;
@Parameter real f0 = 0.4;
```

Defining the type of the parameters dE_dT and d2E_dT2 is a bit more involved. The derivative_type metafunction allows to build their types as follows:

```
@Parameter derivative_type<stress, temperature> dE_dT =  -1.53994698e7;
@Parameter derivative_type<derivative_type<stress, temperature>, temperature>
           d2E_dT2 = -1.9198278e4;
```

In Version 4.1, the definition of d2E_dT2 can be simplified as follows:

```
@Parameter derivative_type<stress, temperature, temperature> d2E_dT2 = -1.9198278e4;
```

3.3.8.4.3 Impact on runtime performances - Quantities have a compile-time overhead as:

- the compiler must perform the dimensional analysis to check if every operation is valid,
- quantities have been implemented to have a negligible runtime overhead, if any (see comment below), at least in optimised builds. This requires that the compiler does extra work, such as function inlining and intermediate variable elimination.

The following listings show the exact same assemblies resulting from the compilation of the code generated by the `c` interface with (right) and without (left) the usage of quantities.

```
UO2_YoungModulus_Martin1989:                    | UO2_YoungModulus_Martin1989:
.LFB4378:                                       |.LFB4714:
    .cfi_startproc                              | .cfi_startproc
    vmulsd   .LC0(%rip), %xmm0, %xmm2           | vmulsd   .LC0(%rip), %xmm0, %xmm2
    vmovsd   .LC1(%rip), %xmm3                  | vmovsd   .LC1(%rip), %xmm3
    vdivsd   .LC3(%rip), %xmm1, %xmm1           | vdivsd   .LC3(%rip), %xmm1, %xmm1
    vfmadd213sd .LC2(%rip), %xmm0, %xmm3        |     vfmadd213sd .LC2(%rip), %xmm0, %xmm3
    vfmadd132sd %xmm2, %xmm3, %xmm0             | vfmadd132sd %xmm2, %xmm3, %xmm0
    vmovsd   .LC4(%rip), %xmm2                  | vmovsd   .LC4(%rip), %xmm2
    vsubsd   %xmm1, %xmm2, %xmm2                | vsubsd   %xmm1, %xmm2, %xmm2
    vmulsd   %xmm2, %xmm0, %xmm0                | vmulsd   %xmm2, %xmm0, %xmm0
    vmovq    %xmm0, %rax                        | vmovq    %xmm0, %rax
    shrq     $52, %rax                          | shrq     $52, %rax
    andl     $2047, %eax                        | andl     $2047, %eax
    cmpl     $2047, %eax                        | cmpl     $2047, %eax
    je   .L4                                    | je   .L4
    ret                                         | ret
    .p2align 4,,10                              | .p2align 4,,10
    .p2align 3                                  | .p2align 3
```

In this case, the compiler was able to generate exactly the same assembly in both cases. Hence, in this simple case, usage of quantities does not have any runtime overhead.

3.4 Advanced topics

3.4.1 `mfront-query`

`mfront-query` is a convenient tool to analyse an `MFront` source file.

The list of queries for material properties can be displayed using the `--help-material-property-queries-list` command line argument, as follows:

```
$ mfront-query --help-material-property-queries-list
Usage: mfront-query [options] [files]

Available options are :
--class-name                    : show the class name
--date                          : show the file implementation date
--description                   : show the file description
--generated-headers             : show all the generated headers
--generated-sources             : show all the generated sources
--inputs                        : show the list of inputs
--interface                     : define an interface
--law-name                      : show the law name
....
```

For example, the following query shows the list of `C++` files generated when using the `python` interface:

```
$ mfront-query --interface=python \
               --generated-sources UO2_YoungModulus_Martin1989.mfront
uo2 : UO2_YoungModulus_Martin1989-python.cxx UO2lawwrapper.cxx
$ ## do not sort by library
$ mfront-query --interface=python \
               --generated-sources=unsorted UO2_YoungModulus_Martin1989.mfront
UO2_YoungModulus_Martin1989-python.cxx UO2lawwrapper.cxx
```

`mfront-query` is the basis for advanced compilation projects such as `MFrontGallery`.

3.4.2 The tfel::system::ExternalMaterialPropertyDescription class

The TFEL/System provides the ExternalMaterialPropertyDescription class which can be used to retrieve numerous information about a material property contained in a shared library. This class has been wrapped in python.

The ExternalMaterialPropertyDescription is a convenient layer around the ExternalLibraryManager which is more flexible but also more difficult to use directly. It allows to retrieve various other information, such as:

- most metadata exported by MFront (list of arguments and parameters, author, date, description, etc..),
- the standard and physical bounds of the inputs (arguments) of a material property,
- the default values of the parameters of a material property.

3.4.2.1 Example of usage

```
>>> from tfel.system import ExternalMaterialPropertyDescription
>>> d = ExternalMaterialPropertyDescription('uo2.so',
                                            'UO2_YoungModulus_Martin1989')
>>> print(d.author)
T . Helfer
>>> print(d.description)
The elastic constants of polycrystalline UO2 and
( U , Pu ) mixed oxides : a review and recommendations
Martin , DG
High Temperatures . High Pressures , 1989
>>> print(d.arguments)
['Temperature', 'Porosity']
>>> print(d.parameters)
['E0', 'dE_dT', 'd2E_dT2']
>>> if (d.hasBounds('Temperature')):
...     if (d.hasLowerBound('Temperature')):
...         print(d.getLowerBound('Temperature'))
...     if (d.hasUpperBound('Temperature')):
...         print(d.getUpperBound('Temperature'))
...
273.15
2610.15
```

3.4.3 The mtest::MaterialProperty class

The mtest::MaterialProperty class allows to load a material property generated using one of the following interfaces: castem, cyrano and generic. This class has been wrapped in python, as described on this page.

This class has the following core interface:

- getVariablesNames, which returns the names of the arguments of the material property,
- setVariableValue which allows to set the value of an argument of the material property,
- getValue which evaluates the material property for the values of the arguments set by the setVariableValue method,
- getParametersNames, which returns the names of the parameters of material property,
- setParameter which allows to change the value of a parameter.

As described in the next paragraph, several convenient methods and operators are also provided to simplify the usage of this class.

3.4.3.1 Usage

Here is an example of the usage of the MaterialProperty class in python.

```
import mtest
young_modulus = mtest.MaterialProperty(
    'src/libCastemUO2.so', 'UO2_YoungModulus_Martin1989')
```

Note that the constructor of MaterialProperty automatically detects the interface used to generate the material property and instantiates the correct implementation internally.

In this example, an instantiation of the CastemMaterialProperty class is created. The arguments of the material property can then be set and the material property can be evaluated:

```
young_modulus.setVariableValue('Temperature', 562)
young_modulus.setVariableValue('Porosity', 0.1)
E = young_modulus.getValue()
```

Setting the variables' values and evaluating the material property can be tedious. To avoid this, overloaded versions of the getValue are available:

```
E = young_modulus.getValue({'Temperature': 562, 'Porosity': 0.1})
```

To make the code even more explicit, the call operator can also be used, as follows:

```
E = young_modulus({'Temperature': 562, 'Porosity': 0.1})
```

3.4.4 The mfm utility

Once a shared library has been generated by MFront, it is convenient to be able to analyse its content rapidly. This can be done in python and C++ using the tfel::system::ExternalLibraryManager class.

The mfm utility is a small command-line program built on top of the ExternalLibraryManager class. It allows to request all the entry points generated by MFront in shared library, i.e. the list of material properties, behaviours and models. Those entry points can be filtered by name and material using regular expressions.

3.4.4.1 Example of usage

```
$ mfm --filter-by-type=material_property --filter-by-name='.+Young.+' src/uo2.so
- UO2_YoungModulus_Martin1989
```

3.5 Usage in other MFront files

3.5.1 The @MaterialLaw keyword

The @MaterialLaw keyword, which is available in all DSLs, allows to import one or several material properties. This keyword is followed by a string or an array of strings. Those strings are paths to other MFront files.

For example, the following statement:

```
@MaterialLaw "UO2_YoungModulus_Martin1989.mfront";
```

defines a set of functions named UO2_YoungModulus_Martin1989 using a dedicated interface named mfront.

Determining the name of the functions defined by importing external MFront files
The mfront interface generates functions whose names are determined as follows:

- If a material name has been defined thanks to the @Material keyword, the names of the generated functions are <material_name>_<law_name>.
- If no material name is defined, the names of the generated functions are simply <law_name>.

where <law_name> is the name of the material property defined thanks to the @Law keyword.

In practice, the Young's modulus can be computed by this simple function call (see the example below):

```
const auto E = UO2_YoungModulus_Martin1989(T, f);
```

About the functions generated by the mfront interface and overloading resolution
The mfront interface generates functions for each standard numeric types (float, double and long double) and also functions for quantities if the imported MFront file support quantities. The function to be called is chosen by standard C++ rules governing function overloading. Hence a material property supporting quantities can be imported from an implementation supporting quantities or not. If both supports quantities, dimensional analysis is performed automatically and transparently.
This complex design has been made to guarantee backward compatibility with versions of MFront which did not support quantities.

Material properties with no inputs
Overload resolution is obviously impossible if the material property has no inputs. In this case, the `mfront` interface chooses a default returned type as follows:

- If the imported material property uses quantity, a quantity based on the `double` floating-point type is returned.
- If the imported material property don't use quantity, the `double` floating-point type is returned.

Again, this choice has been driven by requirements on backward compatibility. We however highly recommend to use a template argument to specifiy the type of returned type expected.
Consider the following example:

```
@DSL MaterialLaw;
@Law YoungModulusDummyExample;
@UseQt true;

@Output stress E;
E.setGlossaryName("YoungModulus");

@Function{
  E = stress{150e9};
}
```

This material property shall be evaluated as follows:

```
const auto E = YoungModulusDummyExample<stress>();
```

This evaluation does work in any context. On the contrary, the evaluation without returned type specification

```
const auto E = YoungModulusDummyExample();
```

may work in most cases, but `E` will be systematically defined as a quantity based of the `double` floating point type. Thus, troubles can appear if:

- The material property is used in a context where quantities are not supported.
- The material property is used in a context with a different underlying floating point type (i.e. `float` or `long double`).

3.5.1.1 A simplified implementation of the first Lamé's coefficient λ of uranium dioxide

Let us consider the first Lamé's coefficient λ of uranium dioxide which can computed from the Young's modulus E and Poisson's ratio ν by the following formula:

$$\lambda = \frac{\nu\, E}{(1+\nu)\,(1-2\,\nu)}$$

For the sake of simplicity, we assume that the Poisson's ratio of uranium dioxide is constant and egal to 0.3.

With this gross assumption, a possible implementation of the first Lamé's coefficient is as follows (file `UO2_FirstLameCoefficient_MFrontBook2022.mfront`):

```
@DSL MaterialLaw;
@Material UO2;
@Law FirstLameCoefficient_MFrontBook2022;
@Author Thomas Helfer;
@Date 25 / 08 / 2022;
@Description {
  An example showing how to use the `@MaterialLaw` keyword
}

@UnitSystem SI;
@UseQt true;

@Output stress λ;
λ.setGlossaryName("FirstLameCoefficient");

@StateVariable temperature T;
```

```
T.setGlossaryName("Temperature");
@StateVariable real f;
f.setGlossaryName("Porosity");

@Parameter real ν = 0.3;
ν.setGlossrayName("PoissonRatio");

@MaterialLaw "UO2_YoungModulus_Martin1989.mfront";

@Function {
  const auto E = UO2_YoungModulus_Martin1989(T, f);
  λ = ν * E / ((1 + ν) * (1 - 2 * ν));
}
```

3.6 Examples of usage of material properties generated by MFront in different solvers

This section is devoted to show how to use material properties generated by MFront in the following solvers:

- Cast3M
- code_aster

3.6.1 Usage in Cast3M

In this section, we consider the thermal conductivity of mixed Uranium-Plutonimum carbide which is assumed to be a function of three state variables: the temperature, the porosity and the burn-up of the material.

This material property can be declared in Cast3M by passing a table to the MATERIAU command, as follows:

```
* Creation of isotropic heat transfer model
ModT1 = 'MODELISER' s1 'THERMIQUE' 'ISOTROPE';
* Creation of a table able to call a material property generated
* by the `castem` interface of MFront. This table must contain
* the following entries:
* - 'MODELE' which contains the name of the material property
* - 'LIBRAIRIE' which contains the path the the external library
* - 'VARIABLES' which contains the list of parameter of the material property
Tmat = 'TABLE';
Tmat. 'MODELE' = 'CHAINE' 'UPuC_ThermalConductivity';
Tmat. 'LIBRAIRIE' = 'CHAINE' 'libUPuCMaterialProperties.so';
Tmat. 'VARIABLES' = 'MOTS' 'T' 'PORO' 'FIMA';
* The material characteristic can be created based on the previous model and table
MatT1 = 'MATERIAU' ModT1 'K' Tmat;
```

When using the PASAPAS procedure, the temperature is known and denoted by T. The porosity and the burn-up of the material must be defined by user using loadings named respectively PORO and FIMA. Those names can be choosen freely by the user as long as they do not conflict with variables defined by the resolution procedures of Cast3M. Internally, the thermal conductivity is evaluated thanks to the VARI NUAGE command.

> **A word of caution**
> It is worth emphasing that the user is responsible for ordering correctly the arguments of the material property. The only check performed by the castem interface concerns the number of arguments.

3.6.2 Usage in code_aster

The easiest way to use a material property implemented by MFront is to use the python interface to generate a python module, import that module in the code_aster input file and use the FORMULE command to evaluate this material property.

3.7 Appendices

3.7.1 About the analysis of errors generated by MFront or the C++ compiler

As described in Section 3.2.2, the MFront file is first parsed, interperted by MFront and used to generate a set of C++ source files. In a second step, those C++ source files are compiled by a C++ compiler into a shared library.

Each step may lead to errors, as described in the following paragraphs.

3.7.1.1 Example of an error generated by MFront

3.7.1.1.1 First example Consider the following incorrect code which declares the T variable twice:

```
@Input T, T;
```

Compiling a file with this error leads to the following message:

```
$ mfront --interface=python YoungModulus.mfront
Error while treating file 'UO2YoungModulus.mfront'
MaterialPropertyDSL::analyse: error while treating keyword '@Input'.
MaterialPropertyDescription::reserveName: name 'T' already reserved
Error at line 4
```

3.7.1.1.2 Second example If the @Function is not used, the following error is generated:

```
$ mfront --interface=python YoungModulus.mfront
Error while treating file 'UO2YoungModulus-test.mfront'
MaterialPropertyDSL::generateOutputFiles: no function defined.
```

3.7.1.1.3 Third example If the @Function body is empty, the following error is generated:

```
$ mfront --interface=python YoungModulus.mfront
Error while treating file 'UO2YoungModulus-test.mfront'
MaterialPropertyDSL::treatFunction: function is empty.
Error at line 5
```

3.7.1.1.4 Fourth example If the name of the material property is not defined, the following error is generated:

```
$ mfront --interface=python YoungModulus.mfront
MaterialPropertyDSL::generateOutputFiles: no material property name defined.
```

3.7.1.1.5 Fifth example If the name of the material property is ill formed (for example @YoungModulus), the following error is generated:

```
$ mfront --interface=python YoungModulus.mfront
MaterialPropertyDSL::treatLaw: '@YoungModulus' is not a valid law name
Error at line 3
```

The rules defining a valid law name are given in the help of the @Law keyword:

```
$ mfront --help-keyword=MaterialLaw:@Law
The `@Law` keyword allows the user to associate a name to the material
law being treated. This keyword is followed by a name.

This name must be a valid C++ identifier. The following characters are
legal as the first character of an identifier, or any subsequent
character:

`_` `a` `b` `c` `d` `e` `f` `g` `h` `i` `j` `k` `l` `m`
`n` `o` `p` `q` `r` `s` `t` `u` `v` `w` `x` `y` `z`
`A` `B` `C` `D` `E` `F` `G` `H` `I` `J` `K` `L` `M`
`N` `O` `P` `Q` `R` `S` `T` `U` `V` `W` `X` `Y` `Z`

The following characters are legal as any character in an identifier
```

```
except the first:

`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

#### Name of the generated functions or classes

The names of the generated functions or classes depend on the law name
as specifified with the `@Law` keyword but may also include the
material name, as specified by the `@MaterialLaw` keyword.
```

3.7.1.2 Example of an error generated by the C++ compiler

Assume that one forgets to declare the porosity f. This error will not detected by MFront but by the C++ compiler as illustrated below:

```
$ mfront --interface=python YoungModulus.mfront
$ mfront --obuild
UO2YoungModulus.mfront: In function 'PyObject* YoungModulus_wrapper(PyObject*, PyObject*)':
UO2YoungModulus.mfront:5:31: error: 'f' was not declared in this scope
   res = 2.2693e11 * (1 - 2.5 * f) * (1 - 6.786e-05 * T - 4.23e-08 * T * T);
```

Although the message mentions an implementation detail (the name of the function doing the interface with python, i.e. YoungModulus_wrapper), the error clearly mentions the MFront source file and not the generated C++ files. The syntax UO2YoungModulus.mfront:5:31 states that the error has been detected in the file UO2YoungModulus.mfront at Line 5 (column 31).

Chapter 4

Implementing point-wise models

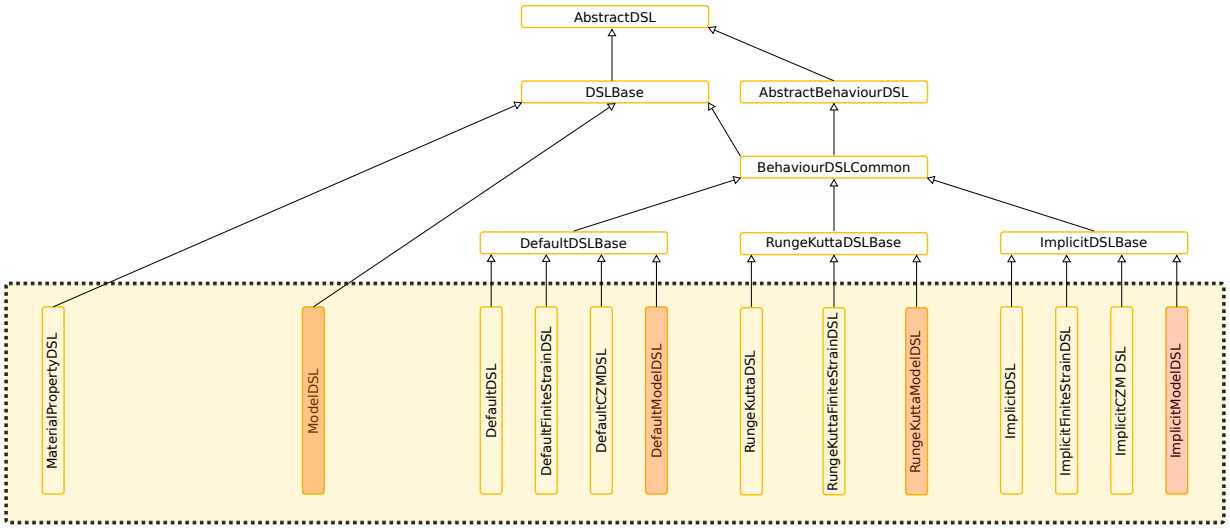

Figure 4.1: Inheritance between Domain Specific Languages (DSL). The DSLs described in this chapter are highlighted in red.

4.1 Introduction

Point-wise models describes the local evolution of a material du to a specific physical phenomenon (phase transition, swelling due to solid fission products). To a limited extent, those models can also be used to describe phenomea at the structural scale, such as chemical reactions in an homogenous media and the oxidation of pipes.

This chapter describes how to implement point-wise models with `MFront` and introduces four new domain specific languages (see Figure 4.1). The examples shown in this chapter can be downloaded on this repository: https://github.com/thelfer/MFrontBookExamples/tree/master/models

The `Model` DSL is historically the first DSL devoted to models introduced in `MFront` and mostly targets simple models used in the fuel performance codes of the `PLEIADES` plateform, such as the `licos` fuel performance code [11]. At that time, `MFront` only supported small strain and finite strain behaviours and there was no clear link between models and behaviours. As a consequence, the `Model` DSL has specific conventions which are somehow inconsistent with the rest of `MFront`.

The introduction of generic behaviours has allowed to treat point-wise models as a special case of generic behaviours without gradient (and associated thermodynamic force). The three DSLs `DefaultModel`, `RungeKutta-Model` and `ImplicitModel` directly inherits most features from their counterparts dedicated to generic behaviours but enforces the aforementioned restrictions (no gradient). See also figure 4.1 for an overview of the relationships between the DSLs available in `MFront`.

4.2 Basic concepts

Models describes how a material evolves du to a given physical phenomena du to a change in its thermodynamic environment during a time step $\Delta\,t$.

The state of the material is describes by a set of n_y so-called *internal state variables* $\left(y_j\right)_{j\in[1:n_y]}$ packed in a vector $\vec{Y}$:

$$\vec{Y} = \begin{pmatrix} y_1 \\ \vdots \\ y_{n_y} \end{pmatrix}$$

Note that this notation does not make any assumption on tensorial nature of the state variable y_j that may a scalar, a symmetric tensor, an unsymmetric tensor, etc...

The change of the thermodynamic environment is described by a set of n_z *external state variables* $\left(z_j\right)_{j\in[1:n_z]}$ packed in a vector $\vec{Z}$:

$$\vec{Z} = \begin{pmatrix} z_1 \\ \vdots \\ z_{n_z} \end{pmatrix}$$

The evolution of the internal state variables $\vec{Y}$ is assumed to obey a standard first-order ordinary differential equation (ODE) given by:

$$\dot{\vec{Y}} = \frac{\mathrm{d}\vec{Y}}{\mathrm{d}\tau} = \vec{G}\left(\vec{Y}(\tau), \vec{Z}(\tau)\right) \tag{4.1}$$

where τ denotes the time variable.

4.2.1 Time discretization

Time is discretized in intervals named time steps. In the following, we generally consider a single time step, named "the" time step for the sake of simplicity.

t denotes the time at the beginning of the time step. $\Delta\,t$ is the time increment during the time step, so that $t+\Delta\,t$ corresponds to the time at the end of the step.

Models in `MFront` assumes that the values of the external state variables at the beginning of the time step, denoted $\vec{Z}\Big|_t$, and their increments during the time step, denoted $\Delta\,\vec{Z}$, are known.

Note that, since only $\vec{Z}\Big|_t$ and $\Delta\,\vec{Z}$ are known, the evolution of $\vec{Z}$ during the time step can only be described using a linear interpolation. More precisely, let θ be a number in the range $[0,1]$, the values of $\vec{Z}$ at $t+\theta\,\Delta\,t$ is denoted $\vec{Z}\Big|_{t+\theta\,\Delta\,t}$ and is given by:

$$\vec{Z}\Big|_{t+\theta\,\Delta\,t} = \vec{Z}\Big|_t + \theta\,\Delta\,\vec{Z}$$

This assumption generally implies that high order algorithms to solve ODE (4.1) are generally useless.

4.2.2 Numerical resolution

Different algorithms must be used to solve ODE (4.1), including. `MFront` proposes two main classes of algorithms:

- Explicit Runge-Kutta algorithms. Such algorithms are the basis of the `RungeKuttaModel` DSL (Section 4.5).
- Semi-implicit scheme. Such algorithms are the basis of the `ImplicitModel` DSL (Section 4.6).

In some cases, the user may want to provide its own algorithm. In this case, the `Model` and the `DefaultModel` may be used (see Sections 4.3 and 4.4).

4.2.3 Simulation of simple chemical reaction as a toy example

To illustrate the implementation of models with `MFront` , let us consider a system of two chemical species A and B whose evolution is given by the following reaction:

$$A \underset{k_2}{\overset{k_1}{\rightleftharpoons}} B$$

In other words, the molar concentrations $[A]$ and $[B]$ of the chemical species A and B satisfies the following ordinary differential equation:

$$\begin{cases} \dfrac{\mathrm{d}\,[A]}{\mathrm{d}\tau} = k_2\ [B]\,(\tau) - k_1\ [A]\,(\tau) \\ \dfrac{\mathrm{d}\,[B]}{\mathrm{d}\tau} = k_1\ [A]\,(\tau) - k_2\ [B]\,(\tau) \end{cases} \tag{4.2}$$

The summation of those equations shows that the sum of $[A] + [B]$ is constant, which simply yields the conservation of mass. Hence, the evolution of the system is driven by the following equation:

$$\frac{\mathrm{d}\,[A]}{\mathrm{d}\tau} = k_2\big(\,[A]|_t + [B]|_t\big) - (k_1 + k_2)\ [A]\,(\tau) \tag{4.3}$$

Algorithms preserving invariants and physical constraints
The Ordinary Differential Equation (4.2) exhibits one invariant (the conservation of mass) and two physical constraints as the molar concentrations $[A]$ and $[B]$ must remain positive.
Invariants and physical constraints must be taken with great care when designing integration schemes.

Two cases will be considered in the following:

1. the reaction rate coefficients k_1 and k_2 are constant.
2. the reaction rate coefficients k_1 and k_2 depends on the temperature following the Arrhenius law, as follows:

$$\begin{cases} k_1 = k_{10}\ \exp\left(-\dfrac{T}{T_{a1}}\right) \\ k_2 = k_{20}\ \exp\left(-\dfrac{T}{T_{a2}}\right) \end{cases} \tag{4.4}$$

where k_{10}, T_{a1}, k_{20} and T_{a2} are material coefficients.

Table 4.1: Values of the reaction rate coefficients and associated material coefficients

Material coefficient	Value
k_1	$\frac{1}{60}\ s^{-1}$
k_2	$\frac{1}{120}\ s^{-1}$
k_{10}	$0.018377505387559667\ s^{-1}$
k_{20}	$0.01013198112809354\ s^{-1}$
T_{a1}	$3000\ K^{-1}$
T_{a2}	$1500\ K^{-1}$

The values of those material coefficients for the numerical applications presented below are reported in Table 4.1. The values of k_{10} and k_{20} have been choosen so that the values of k_1 and k_2 in both cases are equal for a temperature T of 293.15.

4.2.4 Equilibrium

At a constant temperature T, the system will approach an equilibrium and the molar concentration $[A]$ will tend to a limit value $[A]|_\infty$ that can be determined using Equation (4.3).

Setting the value of the concentration rate to zero leads to the following expression of $[A]|_\infty$:

$$[A]|_\infty\,(T) = \frac{k_2(T)\,\big(\,[A]|_t + [B]|_t\big)}{k_1(T) + k_2(T)} \tag{4.5}$$

Figure 4.2 plots $[A]|_\infty\,(T)$ for an initial concentration of species A equal to 0 and an initial concentration of species B equal to 0.1 mol.

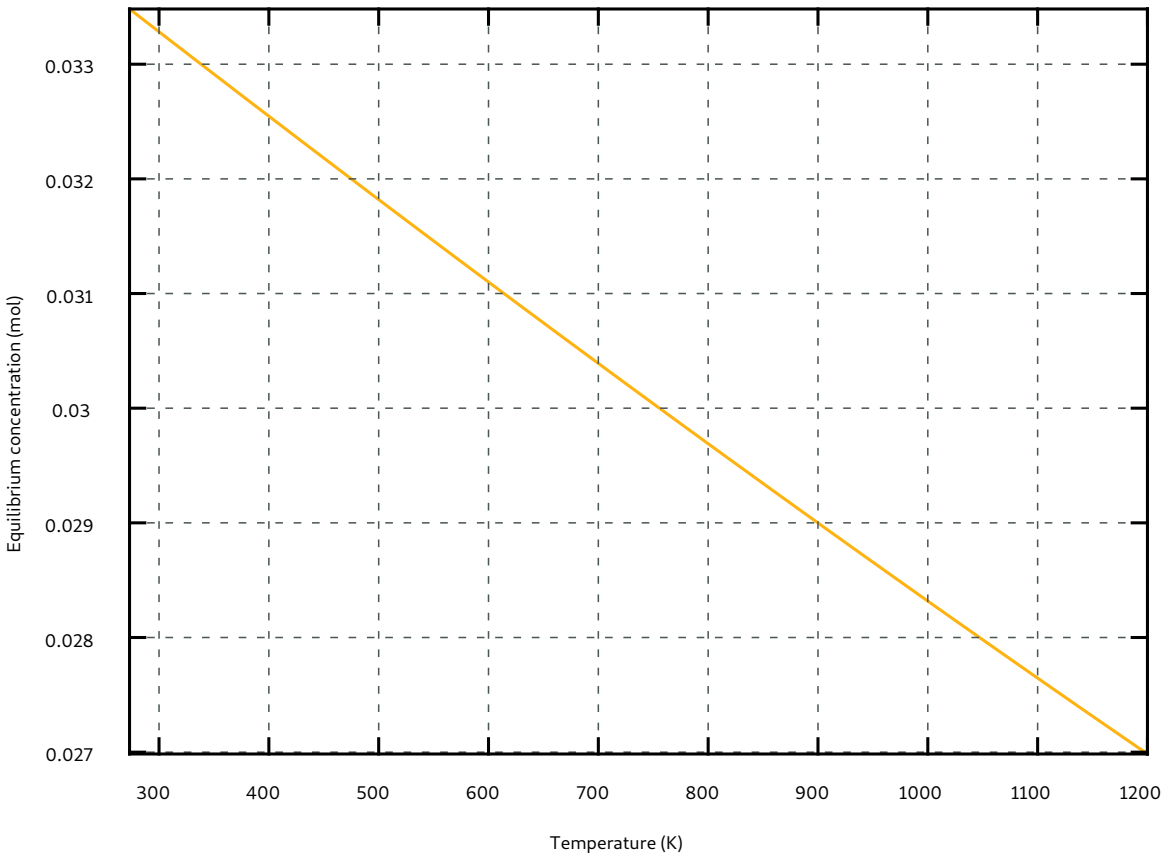

Figure 4.2: Equilibrium concentration as a function of the temperature for an initial concentration of 0.1 mol of species B

On the importance of Equation (4.5)
Equations such as Equation (4.5) are highly valuable in practice to check the numerical results of our integration schemes.
When implementing a model, it is always worth, according to our experience, to study limit cases and special cases. Interesting special cases can be obtained in this example by setting k_1 or k_2 to zero.

4.3 The Model DSL

The Model DSL let the user define its own integration algorithm. It is well suited in three extreme (and exclusive) cases:

1. A closed-form solution of the Ordinary Differential Equation (4.1) is known.
2. A highly specific algorithm is required, i.e. none of the algorithms provided by the RungeKuttaModel and ImplicitModel DSLs is suitable.
3. MFront is used as a wrapper to an already existing implementation. Several examples for behaviours are available in the documentation of the TFEL or MFrontGallery project:

- Introducing small strain legacy Abaqus/UMAT implementations in MFrontGallery
- Using MFront as a wrapper for a thermo-hydro-mechanical behaviour for bentonite available in the TRIAX package

4.3.1 Constant reaction rate coefficients

4.3.1.1 Numerical scheme in the case of constant reaction rate coefficients

If the reaction rate coefficients k_1 and k_2 are assumed constant, Equation (4.3) can be integrated analytically and the value of the concentration at the end of the time step is given by:

$$[A]|_{t+\Delta t} = \frac{B}{K} + \left(\frac{K\,[A]|_t - B}{K}\right)\exp\left(-K\,\Delta\,t\right) \tag{4.6}$$

with $B = k_2\left([A]|_t + [B]|_t\right)$ and $K = k_1 + k_2$.

4.3.1.2 Implementation

A possible implementation of this model is as follows (file ChemicalReaction1.mfront):

```
@DSL Model;
@Model ChemicalReaction1;
@Author Thomas Helfer;
@Date 09/07/2022;
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
ca.setDepth(1);
//! molar concentration of species B
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");
cb.setDepth(1);

//! rate coefficient of the reaction transforming species A to species B
@Parameter frequency k1 = 0.016666666666666666;
k1.setEntryName("ReactionRateCoefficientAB");
//! rate coefficient of the reaction transforming species B to species A
@Parameter frequency k2 = 0.008333333333333333;
k2.setEntryName("ReactionRateCoefficientBA");

@Function ChemicalReaction {
```

```
  const auto B = k2 * (ca_1 + cb_1);
  const auto K = k1 + k2;
  const auto e = exp(-K * dt);
  ca = ca_1 * e + (B / K) * (1 - e);
  cb = ca_1 + cb_1 - ca;
}
```

Most keywords have already been discussed in depth in the tutorial on material properties. However, a few remarks can be made.

4.3.1.2.1 Declaration of state variables

In the first place, there is no alias for defining the type of an amount of substance. Hence, we directly use the quantity class to define the type of the molar concentration $[A]$ and $[B]$. The integers following the template parameters have the following meaning in the international system of units:

$$kg^{i_1}\ m^{i_2}\ s^{i_3}\ A^{i_4}\ K^{i_5}\ cd^{i_6}\ mol^{i_7}$$

The `Model` DSL has a concept of depth associated with state variables (and external state variables):

- A depth of zero means that one only have access to the value of a variable at the end of the time step.
- A depth of one means that one have access to the value of a variable at the beginning and at the end of the time step.

Values greater than one are not accepted by most interfaces. In pratice, assigning a depth of 1 to $[A]$ means that two variables will be defined `MFront`:

- `ca`: the value of $[A]$ at the end of time step.
- `ca_1`: the value of $[A]$ at the beginning of the time step.

Those conventions are specific to the `Model` DSL and are not consistent with any of the other DSLs.

4.3.1.2.2 The `@Function` keyword

The `@Function` keyword introduces a code block where the values of the state variables at the end of the time step must be computed.

This code block must be named, so that the `@Function` keywords could be used several times to separate independant computation. In practice, this feature is rarely used, and could be considered as another oddity of the `Model` DSL.

The time increment is automatically defined and accessible in a variable named `dt`.

The implementation of the model is then a direct translation of Equation (4.6) in `C++`.

4.3.1.3 Testing with `MTest`

This section provides a small introduction to `MTest`, an handful tool delivered with `MFront` which can be used to test this model rapidely.

To do this, we need to compile the our implementation, saved in a file `ChemicalReaction1.mfront` using the `generic` interface, as follows:

```
$ mfront --obuild --interface=generic ChemicalReaction1.mfront
Treating target : all
The following library has been built :
- libModel.so :  ChemicalReaction1_AxisymmetricalGeneralisedPlaneStrain
ChemicalReaction1_AxisymmetricalGeneralisedPlaneStress ChemicalReaction1_Axisymmetrical
ChemicalReaction1_PlaneStress ChemicalReaction1_PlaneStrain
ChemicalReaction1_GeneralisedPlaneStrain ChemicalReaction1_Tridimensional
```

The `generic` interface for the `Model` DSL is meant to be compatible with the `generic` interface for behaviours. It thus creates a function for every modelling hypothesis supported by `MFront`, although this notion of modelling hypothesis is meaningless for models.

On the interfaces for the Model DSL
The fact that the Model DSL has its own set of interfaces is also a particularity of this interface.
In pratice, the Model DSL was mostly used inside the PLEIADES numerical plateform which provided its own (private) interface. In this plateform, models had a role totally different from behaviours, a fact which explains the special treatment of the Model DSL.

Then we can create a file named ChemicalReaction1.mtest containing the following script:

```
// loading the model
@Model 'src/libModel.so' 'ChemicalReaction1';
// initial value of the molar concentration of species B
@Real 'B0' 0.1;
@StateVariable 'MolarConcentrationOfSpeciesB' 'B0';
// time discretization
@Times {
  0, 360 in 100
};
```

As MFront files, this script may contain standard C++ comments.

The @Model keyword loads the model generated by MFront. By the default, the Tridimensional modelling hypothesis is assumed and MTest automatically selects the corresponding function (i.e. ChemicalReaction1_Tridimensional in this case). MTest not only loads the function associated with the model but also all the relevant metadata, such as:

- the number of the state variables and their "kind" (scalar, tensor, symmetric tensor, etc..),
- the number of parameters, their default values (see below), etc...

Keywords documentation
Most keywords in MTest are documented and their documentation can be retrieved using the --help-keyword command line argument as follows:

```
$ mtest --help-keyword=@Model
The `@Model` keyword is mostly a synomymous of the `@Behaviour`. The
only difference is that a check is done to see if the behaviour does not
declare any gradient, thermodynamic force nor any tangent operator
block.

....
```

We then initialize the initial values of the state variables using the @StateVariable keyword. Note that MTest initializes all state variable to zero by default, so the initialization of molar concentration of species A can be ommited.

One may notice that we declared a constant named B0 to store the initial value thanks to @Real keyword. This value will be useful to create a non-regression test in Section 4.3.1.4. Such constants can be used in formulae which are introduced by string. The simpliest formula is 'B0' which is used in the script to provide the initial value of the molar concentration of species B.

Finally, we declare the time discretisation using the @Time keyword. Here, the simulation starts at time 0 and ends after 360 seconds. This interval is divided in 100 steps.

This script can be executed as follows:

```
$ mtest ChemicalReaction1.mtest
....
resolution from 345.6 to 349.2
resolution from 349.2 to 352.8
resolution from 352.8 to 356.4
resolution from 356.4 to 360
Execution succeeded
-number of period:     100
-number of iterations: 0
```

```
-number of sub-steps:  0
Result of test 'unit behaviour test' of group 'MTest'            : SUCCESS
End of Test Suite                                                : SUCCESS
```

This executation generates a result file named by default `ChemicalReaction1.res` (this can be changed using the `@OutputFile` keyword). The header of this file describes its content:

```
# first column: time
# 2 column: MolarConcentrationOfSpeciesA
# 3 column: MolarConcentrationOfSpeciesB
# 4 column: stored energy
# 5 column: disspated energy
```

The two last column (stored energy and dissipated energy) are automatically defined by `MTest` and are not meaningful here.

This file can be directly computed using `gnuplot` which is available on most systems or imported in any spreadsheet software. Here, we use a small tool named `tplot`[1] as follows:

```
$ tplot "ChemicalReaction1.res" -u 1:2 --title="Species A"                  \
        "ChemicalReaction1.res" -u 1:3 --title="Species B"                  \
        --with-grid --xlabel="Time (s)" --ylabel="Molar concentration (mol)"
```

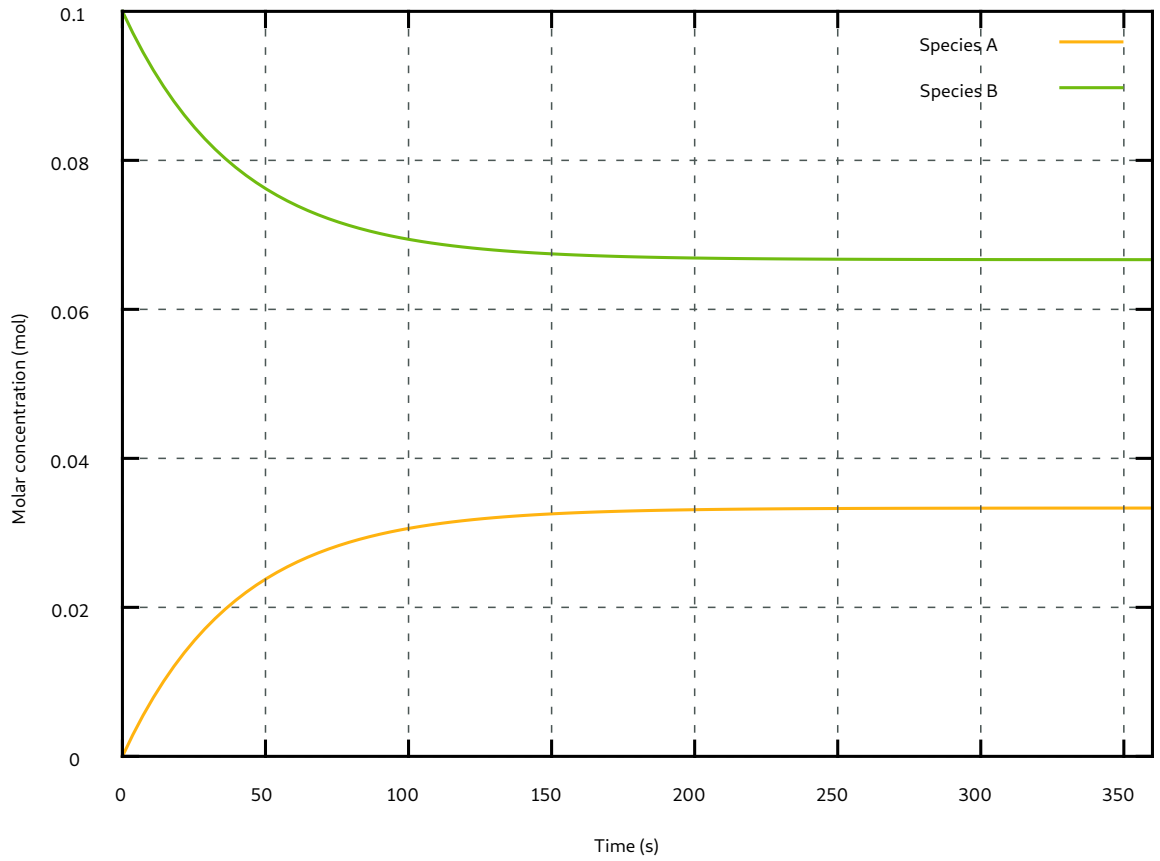

Figure 4.3: Evolution of the molar concentrations of species A and B

The evolution of the molar concentrations of the two species is depicted on Figure 4.3.

[1] https://github.com/thelfer/tfel-plot‘

Using MTest in python
Most features of MTest are directly accessible from python in the mtest module. The description of this module can be found on this page:
https://thelfer.github.io/tfel/web/mtest-python.html
The previous script is equivalent to this python script (file ChemicalReaction1.py):

```
import std
import tfel.tests
import mtest

mtest.setVerboseMode(mtest.VerboseLevel.VERBOSE_QUIET)

m = mtest.MTest()
m.setAuthor("Thomas Helfer")
m.setDate("09/08/2022")
m.setModel('generic', 'src/libModel.so','ChemicalReaction1')
m.setStateVariableInitialValue('MolarConcentrationOfSpeciesB', 0.1)
m.setTimes([3.6 * i for i in range(0, 100)])
output_file = "ChemicalReaction1-python.res".format(k1)
m.setOutputFileName(output_file)
m.execute()
```

It shall be noticed that most MTest keywords are associated to a method of the mtest.MTest class and that the translation of an MTest script into an equivalent python script can be straightforward.

4.3.1.4 Non regression tests with MTest

MTest has initially been created for the developpers of MFront to build non regression tests.

The @Test keyword allows to compare the results of the simulation to an analytical solution or to reference results stored in an external file.

Since an analytical solution, given by Equation (4.6), is known, the following non regression tests can be set up:

```
@Real 'k1' 'ChemicalReaction1::ReactionRateCoefficientAB';
@Real 'k2' 'ChemicalReaction1::ReactionRateCoefficientBA';
@Real 'K'  'k1 + k2';
@Real 'B'  'k2 * B0';
@Test<function> 'MolarConcentrationOfSpeciesA' '(B/K) * (1 - exp(-K * t))'      1e-14;
@Test<function> 'MolarConcentrationOfSpeciesB' 'B0 - (B/K) * (1 - exp(-K * t))' 1e-14;
```

The two first lines rename the parameters of the model as k1 and k2 for convenience from variables that are automatically defined when loading the model. This can be made explicit by lauching MTest in verbose mode as follows:

```
$ mtest  mtest ChemicalReaction1.mtest --verbose=full
SchemeParserBase::execute : treating keyword '@Author' at line '1'
SchemeParserBase::execute : treating keyword '@Date' at line '2'
SingleStructureSchemeParser::execute : treating keyword '@Model' at line '3'
No hypothesis defined, using default
Adding evolution 'ChemicalReaction1::ReactionRateCoefficientAB' from parameter
'ReactionRateCoefficientAB' with value '0.0166667'
Adding evolution 'ChemicalReaction1::ReactionRateCoefficientBA' from parameter
'ReactionRateCoefficientBA' with value '0.00833333'
SchemeParserBase::execute : treating keyword '@Real' at line '5'
....
```

Two other variables, named K and B are also declared for convenience. Finally, the @Test keyword is used to introduce a first non regression test. The function option indicates that an MTest result is to be compared to an analytical solution. The @Test keyword is followed by:

- the name of the result is then given, the molar concentration of species A in this example.
- a formula in which t stands for the current time.

- a criterion value which is used to test if the absolute value of the difference bewteen the simulation and the analytical value is small enough. A value of 10^{-14} is close enough to the machine precision.

With those additional tests, the output of `MTest` is sightly modified:

```
$ mtest ChemicalReaction1.mtest
....
Result of test 'unit behaviour test' of group 'MTest'            : SUCCESS
* AnalyticalTest::check : comparison for variable 'MolarConcentra: SUCCESS
* AnalyticalTest::check : comparison for variable 'MolarConcentra: SUCCESS
End of Test Suite                                                : SUCCESS
```

Continuous integration and material knowledge management projects
`MTest` can also generate an `XML` file in the `JUnit` format by using the `--xml-output` command line argument. Such file can be parsed by common automation servers, such as jenkins and can be used to build continuous integration of material knowledge management projects built on top of `MFront`. The `MFrontGallery` provides a `cmake` infrastructure which eases build such projects.

4.3.1.5 Parametric studies

The values of the parameters k_1 and k_2 can be changed at runtime using the `@Parameter` keyword as follows:

```
@Parameter 'ReactionRateCoefficientAB' 0.016666666666666666;
```

Note that:

- The `@Parameter` can only be called after the declaration of the model using '`@Model`.
- The value of the variable `ChemicalReaction1::ReactionRateCoefficientAB` is not affected. This variable is defined when loading the model and contains the default value of the `ReactionRateCoefficientAB` parameter.

We can use two additional features of `MTest` to automate parametric studies:

1. The ability to specify some commands in the line command line. Those commands are added at the beginning of the file. For example, the name of the output file can be specified by the `--@OutputFile="'output.res'"` command line argument. This is equivalent to the following line:
   ```
   @OutputFile 'output.res';
   ```
   Note that the double quote are required in most shells.
2. The ability to define substitution variables on the command line. For example, every occurrence of `@k1@` will be replaced by the option given to the command line argument `--@k1@`.

These two features can be combined to make a parametric study. To do this, one may modify the call to the `@Parameter` keyword as follows:

```
@Parameter 'ReactionRateCoefficientAB' @k1@;
```

After this modification (file ChemicalReaction1-parametric.mtest), `MTest` can be called as follows:

```
$ export k1=0.016666666666666666
$ mtest ChemicalReaction1-parametric.mtest --@k1@=${k1}          \
        --@OutputFile="'ChemicalReaction1-parametric-${k1}.res'"
```

This can be automated by the following script (file ChemicalReaction1-parametric.sh):

```
#! /usr/bin/env bash

for k1 in 0.016666666666666666                      \
          0.008333333333333333 0.004166666666666667 \
          0.033333333333333333 0.066666666666666667
do
  mtest ChemicalReaction1-parametric.mtest --@k1@=${k1}          \
        --@OutputFile="'ChemicalReaction1-parametric-${k1}.res'"
done
```

The comparison of the evolution of the molar concentration of species A resulting from this script is reported on Figure 4.4.

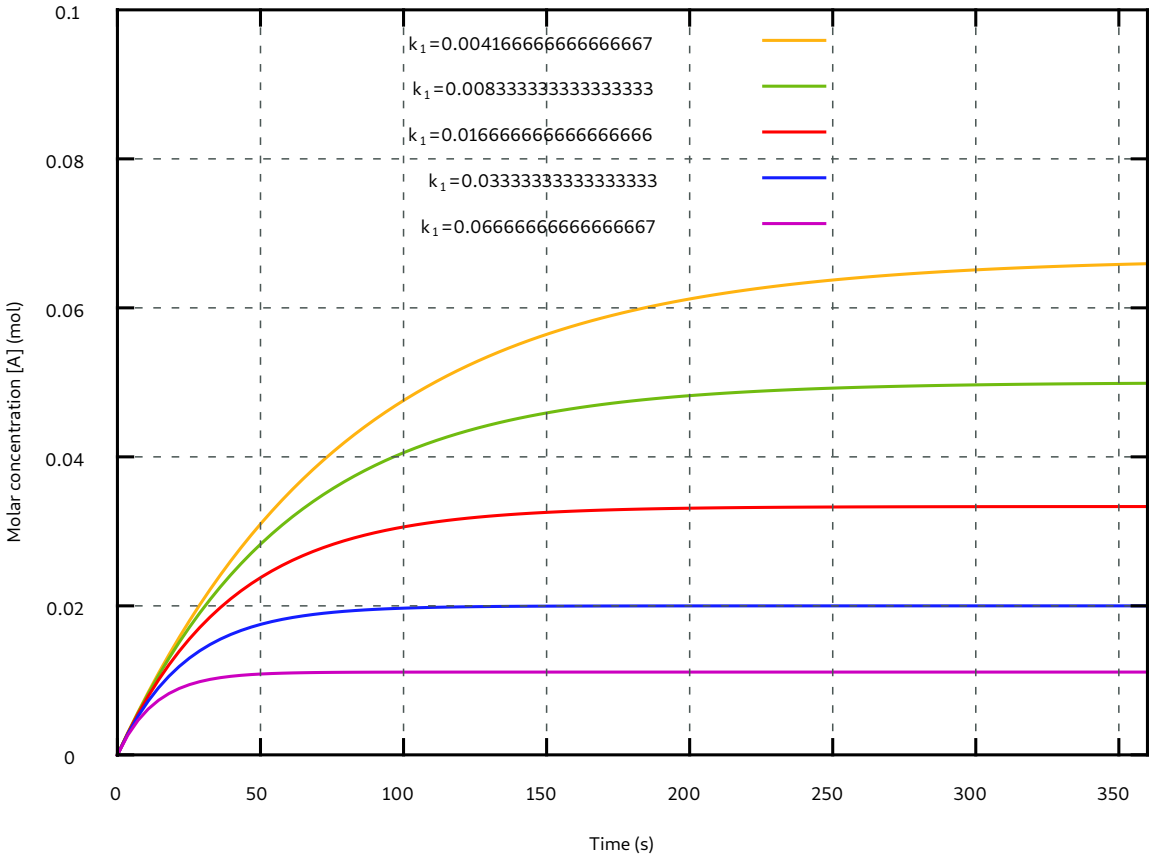

Figure 4.4: Result of the parametric study

Specifying keywords and substitution variables on the command line is a powerful feature that can be used to create template scripts. In our experience, this strategy works well with the cmake build system and is used extensively in the TFEL and MFrontGallery projects to define unit tests, the same script being used for several behaviours and/or to tests several resolution options.

Parametric studies in `python`
Parametric studies and/or parameters fitting are much easier if the `python` bindings of `MTest` are used. The following script (file ChemicalReaction1-parametric.py) performs the same parametric study:

```
import std
import tfel.tests
import mtest

mtest.setVerboseMode(mtest.VerboseLevel.VERBOSE_QUIET)

for k1 in [0.016666666666666666, 0.008333333333333333,
           0.004166666666666667, 0.033333333333333333,
           0.06666666666666667]:
    m = mtest.MTest()
    m.setAuthor("Thomas Helfer")
    m.setDate("09/08/2022")
    m.setModel('generic', 'src/libModel.so','ChemicalReaction1')
    m.setStateVariableInitialValue('MolarConcentrationOfSpeciesB', 0.1)
    m.setParameter('ReactionRateCoefficientAB', float(k1))
    m.setTimes([3.6 * i for i in range(0, 100)])
    output_file = "ChemicalReaction1-python-parametric-{}.res".format(k1)
    m.setOutputFileName(output_file)
    m.execute()
```

4.3.2 Temperature dependent reaction rate coefficients

As mentionned in the Section 4.2.1, the variation of external state variables, such as the temperature, can be at most linear on the time step. However, even with this hypothesis, the evolution of the molar concentration can't be estblished in closed form.

This section will discuss three implementations of the model:

- The first one uses the trapezoidal rule to determine the evolution of the molar concentration.
- The second one uses the midpoint rule to determine the evolution of the molar concentration.
- The third one assumes that the temperature is slowly varying over the time step and that it can be considered almost constant.

A implementation based on a Runge-Kutta algorithms will be described in Section 4.5.

4.3.2.1 Implementation based on the trapezoidal rule

4.3.2.1.1 Numerical scheme The general Equation (4.1) can be integrated over the time step as follows:

$$\Delta\,\vec{Y} = \int_{t}^{t+\theta\,\Delta\,t} \vec{G}\left(\vec{Y}(\tau),\vec{Z}(\tau)\right)\mathrm{d}\,\tau$$

The integral on the right hand size can be approximated by the trapezoidal rule as follows:

$$\Delta\,\vec{Y} \approx \frac{\Delta\,t}{2}\left(\vec{G}\left(\vec{Y}(t),\vec{Z}(t)\right)+\vec{G}\left(\vec{Y}(t+\Delta\,t),\vec{Z}(t+\Delta\,t)\right)\right) \tag{4.7}$$

Applied to Equation (4.3), this scheme leads to the approximation of the increment $\Delta\,[A]$:

$$\begin{aligned}\Delta\,[A] &= \frac{\Delta\,t}{2}\left(k_2\left(T|_t\right)\left([A]|_t+[B]|_t\right)-\left(k_1\left(T|_t\right)+k_2\left(T|_t\right)\right)[A]|_t\right)\\ &+\frac{\Delta\,t}{2}\left(k_2\left(T|_{t+\Delta\,t}\right)\left([A]|_t+[B]|_t\right)-\left(k_1\left(T|_{t+\Delta\,t}\right)+k_2\left(T|_{t+\Delta\,t}\right)\right)\left([A]|_t+\Delta\,[A]\right)\right)\\ &= B-K\,\Delta\,[A]\end{aligned}$$

where:

- $B=\Delta\,t\left(\langle k_2\rangle\;[B]|_t-\langle k_1\rangle\;[A]|_t\right)$ with $\langle k_1\rangle=\dfrac{k_1\left(T|_{t+\Delta\,t}\right)+k_1\left(T|_t\right)}{2}$ and with $\langle k_2\rangle=\dfrac{k_2\left(T|_{t+\Delta\,t}\right)+k_2\left(T|_t\right)}{2}$

- $K = \dfrac{\Delta\, t}{2}\left(k_1\left(T|_{t+\Delta\, t}\right) + k_2\left(T|_{t+\Delta\, t}\right)\right)$.

The increment $\Delta\ [A]$ can thus be computed as follows:

$$\Delta\ [A] = \frac{B}{1+K}$$

This closed-form expression of $\Delta\ [A]$ is due to the linear nature of the function $\vec{G}$. If the function $\vec{G}$ is not linear, the increment of the unknowns must be determined numerically. The `ImplicitModel` DSL, described in Section 4.6, provides several algorithms for this purpose.

A small exercice for the reader
A generalized trapezoidal rule can be devised by introducing a numerical parameter between 0 and 1 as follows:

$$\Delta\,\vec{Y} \approx \Delta\, t\left(\theta\,\vec{G}\left(\vec{Y}(t), \vec{Z}(t)\right) + (1-\theta)\,\vec{G}\left(\vec{Y}(t+\Delta\, t), \vec{Z}(t+\Delta\, t)\right)\right)$$

The standard trapezoidal rule is recovered by choosing θ egal to one half.
This generalized trapezoidal rule can be readily implemented, altough the code becomes a bit more tedious to write.
This implementation is left as an exercice to the reader. The reader may also study the influence of θ on the stability and accuracy of the method.

4.3.2.1.2 Implementation

The implementation of this numerical scheme is also straightforward:

```
@DSL Model;
@Model ChemicalReaction2;
@Author Thomas Helfer;
@Date 09/07/2022;
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
ca.setDepth(1);
//! molar concentration of species B
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");
cb.setDepth(1);

@ExternalStateVariable temperature T;
T.setGlossaryName("Temperature");
T.setDepth(1);

//! reference rate coefficient of the reaction transforming species A to species B
@Parameter frequency k01 = 0.018377505387559667;
k01.setEntryName("ReferenceReactionRateCoefficientAB");
//! reference rate coefficient of the reaction transforming species B to species A
@Parameter frequency k02 = 0.01013198112809354;
k02.setEntryName("ReferenceReactionRateCoefficientBA");
//! activation temperature reaction transforming species B to species A
@Parameter temperature Ta1 = 3000;
Ta1.setEntryName("ActivationTemperatureAB");
//! activation temperature reaction transforming species B to species A
@Parameter temperature Ta2 = 1500;
Ta2.setEntryName("ActivationTemperatureBA");

@Function ChemicalReaction {
  constexpr auto zero = quantity<real, 0, 0, 0, 0, 0, 0, 1>{};
  const auto k1_bts = k01 * exp(-T_1 / Ta1);
```

```
  const auto k1_ets = k01 * exp(-T / Ta1);
  const auto k2_bts = k02 * exp(-T_1 / Ta2);
  const auto k2_ets = k02 * exp(-T / Ta2);
  const auto mean_k1 = (k1_bts + k1_ets) / 2;
  const auto mean_k2 = (k2_bts + k2_ets) / 2;
  const auto B = dt * (mean_k2 * cb_1 - mean_k1 * ca_1);
  const auto K = dt * (k1_ets + k2_ets) / 2;
  ca = ca_1 + B / (1 + K);
  cb = ca_1 + cb_1 - ca;
  // imposing positivity of the molar concentrations
  if(cb < zero){
    cb = zero;
    ca = ca_1 + cb_1;
  }
  if(ca < zero){
    ca = zero;
    cb = ca_1 + cb_1;
  }
}
```

4.3.2.1.3 Code factorization with the `@Import` keyword

To factorize the code with the next implementations, its is worth splitting this implementation and separate the part that are shareable in two files named respectively `ChemicalReaction-common.mfront` and `ChemicalReaction-parameters.mfront`.

The `ChemicalReaction2.mfront` file then contains the following code:

```
@DSL Model;
@Model ChemicalReaction2;
@Author Thomas Helfer;
@Date 09/07/2022;

@Import "ChemicalReaction-common.mfront"

@Function ChemicalReaction {
  constexpr auto zero = quantity<real, 0, 0, 0, 0, 0, 0, 1>{};
  const auto k1_bts = k01 * exp(-T_1 / Ta1);
  const auto k1_ets = k01 * exp(-T / Ta1);
  const auto k2_bts = k02 * exp(-T_1 / Ta2);
  const auto k2_ets = k02 * exp(-T / Ta2);
  const auto mean_k1 = (k1_bts + k1_ets) / 2;
  const auto mean_k2 = (k2_bts + k2_ets) / 2;
  const auto B = dt * (mean_k2 * cb_1 - mean_k1 * ca_1);
  const auto K = dt * (k1_ets + k2_ets) / 2;
  ca = ca_1 + B / (1 + K);
  cb = ca_1 + cb_1 - ca;
  // imposing positivity of the molar concentrations
  if(cb < zero){
    cb = zero;
    ca = ca_1 + cb_1;
  }
  if(ca < zero){
    ca = zero;
    cb = ca_1 + cb_1;
  }
}
```

The `ChemicalReaction-common.mfront` file contains the following code:

```
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
```

```
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
ca.setDepth(1);
//! molar concentration of species B
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");
cb.setDepth(1);

@ExternalStateVariable temperature T;
T.setGlossaryName("Temperature");
T.setDepth(1);

@Import "ChemicalReaction-parameters.mfront";
```

The ChemicalReaction-parameters.mfront file contains the following code:

```
//! reference rate coefficient of the reaction transforming species A to species B
@Parameter frequency k01 = 0.018377505387559667;
k01.setEntryName("ReferenceReactionRateCoefficientAB");
//! reference rate coefficient of the reaction transforming species B to species A
@Parameter frequency k02 = 0.01013198112809354;
k02.setEntryName("ReferenceReactionRateCoefficientBA");
//! activation temperature reaction transforming species B to species A
@Parameter temperature Ta1 = 3000;
Ta1.setEntryName("ActivationTemperatureAB");
//! activation temperature reaction transforming species B to species A
@Parameter temperature Ta2 = 1500;
Ta2.setEntryName("ActivationTemperatureBA");
```

4.3.2.2 Implementation based on the midpoint rule

4.3.2.2.1 Numerical scheme The general Equation (4.1) can be integrated over the time step using the generalized midpoint rule as follows:

$$\Delta\,\vec{Y} = \Delta\,t\,\vec{G}\left(\vec{Y}\Big|_{t+\theta\,\Delta\,t},\vec{Z}\Big|_{t+\theta\,\Delta\,t}\right)$$

where θ is a numerical parameter between 0 and 1. Applied to Equation (4.3), this scheme leads to the approximation of the increment $\Delta\,[A]$:

$$\begin{aligned}\Delta\,[A] &= \Delta,\,t\left(k_2\left(T|_{t+\theta\,\Delta\,t}\right)\left([A]|_t+[B]|_t\right)-\left(k_1\left(T|_{t+\theta\,\Delta\,t}\right)+k_2\left(T|_{t+\theta\,\Delta\,t}\right)\right)[A]|_{t+\theta\,\Delta\,t}\right)\\ &= \Delta,\,t\left(k_2\left(T|_{t+\theta\,\Delta\,t}\right)[B]|_t-k_1\left(T|_{t+\theta\,\Delta\,t}\right)[A]|_{t+\theta\,\Delta\,t}\right)-\Delta\,t\;\;k_1\left(T|_{t+\theta\,\Delta\,t}\right)\theta\,\Delta\,[A]\\ &= B-K\,\theta\,\Delta\,[A]\end{aligned}$$

with $B=\Delta\,t\left(k_2\left(T|_{t+\theta\,\Delta\,t}\right)[B]|_t-k_1\left(T|_{t+\theta\,\Delta\,t}\right)[A]|_{t+\theta\,\Delta\,t}\right)$ and $K=\Delta,\,t\;\;k_1\left(T|_{t+\theta\,\Delta\,t}\right)$.

4.3.2.2.2 Implementation The implementation of this scheme is very close to the second implementation, except for the declaration of the θ parameter and the body of the implementation, as follows (file ChemicalReaction3.mfront):

```
@DSL Model;
@Model ChemicalReaction3;
@Author Thomas Helfer;
@Date 09/07/2022;

@Import "ChemicalReaction-common.mfront";

//! numerical parameter of the generalized midpoint rule
@Parameter real theta = 0.5;
theta.setEntryName("Theta");

@Function ChemicalReaction {
```

```
  constexpr auto zero = quantity<real, 0, 0, 0, 0, 0, 0, 1>{};
  const auto T_mts = T_1 * (1 - theta) + theta * T;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto B = dt * (k2_mts * cb_1 - k1_mts * ca_1);
  const auto K = dt * (k1_mts + k2_mts);
  ca = ca_1 + B / (1 + K * theta);
  cb = ca_1 + cb_1 - ca;
  // imposing positivity of the molar concentrations
  if(cb < zero){
    cb = zero;
    ca = ca_1 + cb_1;
  }
  if(ca < zero){
    ca = zero;
    cb = ca_1 + cb_1;
  }
}
```

4.3.2.3 Third implementation

If the temperature increment is assumed small over a time step, the rate coefficients k_1 and k_2, given by Equation (4.4), can be approximated by their values at the middle of the time step $t + \frac{\Delta t}{2}$.

This third implementation is thus very similar to the one of Section 4.3.1 and is given by the following listing (file `ChemicalReaction4.mfront`):

```
@DSL Model;
@Model ChemicalReaction4;
@Author Thomas Helfer;
@Date 09/07/2022;

@Import "ChemicalReaction-common.mfront";

@Function ChemicalReaction {
  const auto T_mts = (T_1 + T) / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto B = k2_mts * (ca_1 + cb_1);
  const auto K = k1_mts + k2_mts;
  const auto e = exp(-K * dt);
  ca = ca_1 * e + (B / K) * (1 - e);
  cb = ca_1 + cb_1 - ca;
}
```

The advantage of this implementation is that the exact solution is retrieved if the temperature is constant.

4.3.2.4 Comparison of the three schemes

4.3.2.4.1 Test at constant temperature Let us consider the test of Section 4.3.1.3. This test can be used for all three implementation, although the definition of the temperature is required. This definition can be done using the `@ExternalStateVariable` as follows:

```
@ExternalStateVariable 'Temperature' 293.15;
```

This temperature is choosen to retrieve the results given by the model with constant reaction rate coefficients (see Section 4.3.1).

Note that, at a constant temperature, the schemes given by the trapezoidal rule (Section 4.3.2.1) and the midpoint rule are exactly equivalent (Section 4.3.2.2). The third implementation (Section 4.3.2.3) is exact in this case and equivalent, by construction, to the implementation provided in Section 4.3.1.

Hence, the only interesting thing to investiguate is to evaluate the accuracy of the implementations based on the trapezoidal rule and the midpoint rule.

This can be done in two ways. First, one can resuse the unit tests introducted in Section 4.3.1.4 and try to see the minimal criterion value that passes the test.

For 100 time steps, this criterion value must be of the order of 10^{-5}, which is relatively high. The graphical comparison with the exact solution is however quite satisfying and the numerical curves are undistinguishable from the analytical curve.

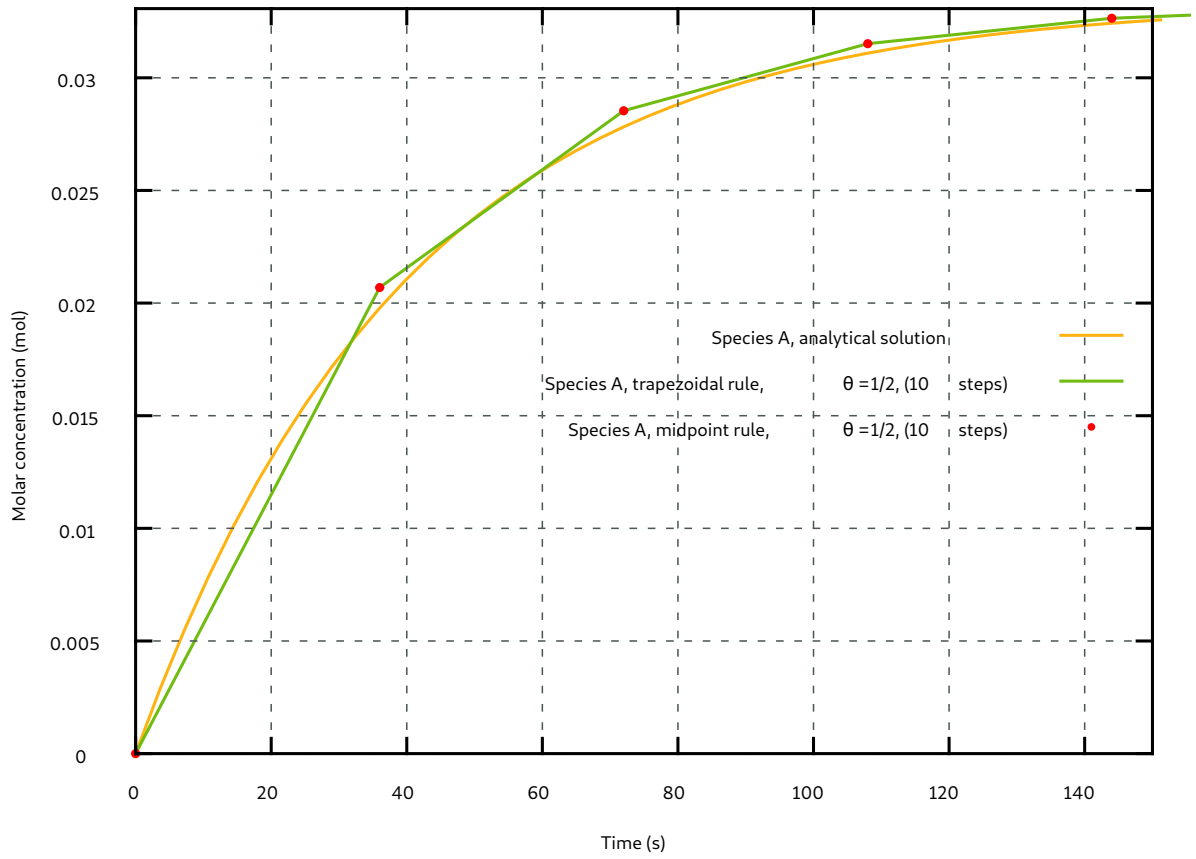

Figure 4.5: Evolution of the molar concentrations of species A and B for the test at constant temperature. Comparison of the numerical results given by the trapezoidal rule and the midpoint rule to the analytical solution at the beginning of the simulation.

Another test would be to see the effect of the number of time steps on the accuracy of the solution. The numerical results with ten time step are visually quite different from the analytical solution, in particular at the beginning of the simulation, as depicted on Figure 4.5.

4.3.2.4.2 Test with varying temperature Equation (4.6) shows that the typical relaxation time scale of the reaction is $\frac{1}{k_1+k_2}$ which is of the order of $40\,s$. One thus expects interesting things to happen is the temperature varies significantly with a similar time scale.

We then choose to impose this temperature evolution:

$$T(\tau) = T_0 + T_1 \sin\left(\frac{\tau}{\tau_0}\right)$$

which oscillates between $T_0 - T_1$ and $T_0 + T_1$. For our tests, we choose $T_0 = 700K$, $T_1 = 400K$ and $\tau_0 = 30$.

This is imposed in `MTest` as follows:

```
@Real 'T0' 700;
@Real 'T1' 400;
@Real 'tau0' 30;
@ExternalStateVariable<function> 'Temperature' 'T0 + T1 * sin(t/tau0)';
```

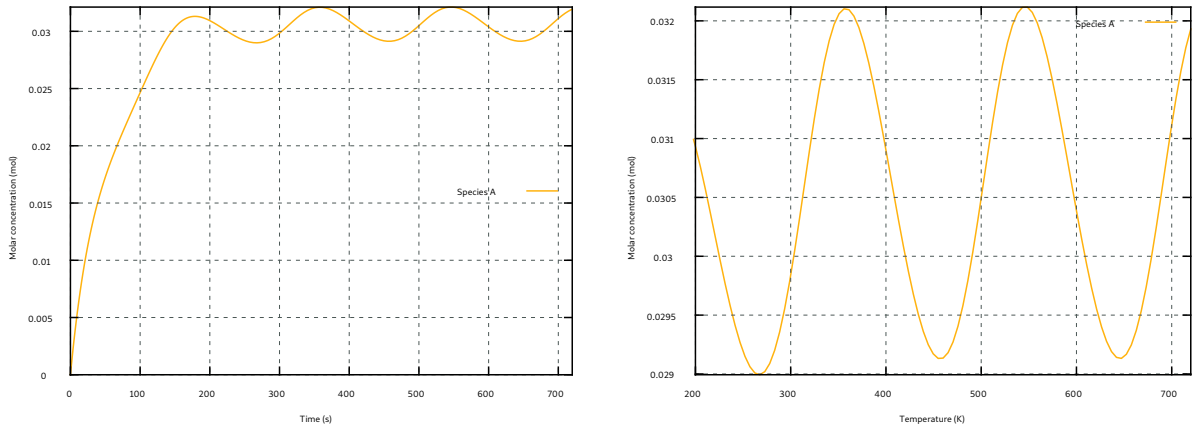

Figure 4.6: Evolution of the molar concentration of species A in the test with varying temperature (left). Evolution of the molar concentration of species A after the transition phase for the three integration schemes presented for a discretisation of the interval $[200 : 720]$ in 20 time steps (right).

This test leads to the reference results for a sufficiently small time step depicted on Figure 4.6 (left). This figure shows an first phase with a rapid variation followed by stabilized oscillary phase after approximatively 200 s, as depicted in Figure 4.6 (right).

We can now plot the results of the three integration schemes with the following time discretization:

```
@Times {
  0, 200 in 100, 720 in 20
};
```

The discretisation of the interval $[0 : 200]$ in 100 time steps is sufficient to have an accurate starting point for all schemes. The discretisation of the interval $[200 : 720]$ in 20 time steps is considered loose with approximatively 7 time steps per cycle.

Figure 4.7 shows that the results obtained with the three schemes are closed to the the reference result but cleary distinguishable.

4.4 The `DefaultModel` DSL

The `DefaultModel` domain specific language is close to the `Model` DSL except that its conventions are much more consistent with the ones used in the DSL related to behaviours.

More specifically, the main differences are the followings:

- The `@Integrator` code block replaces the `@Function` code block.
- `ca` contains the value of $[A]$ at the beginning of the time step and must be updated.
- `T` contains the value of the temperature at the beginning of the time step and `dT` contains the value of the increment of the temperature over the time step. Hence, the value of the temperature in the middle of the time step can be computed by `T + dT / 2`.

The `DefaultModel` DSL, like the `RungeKuttaModel` and `ImplicitModel` DSLs, also introduces additional optional features, such as:

- The computation of a time scaling factor allowing the model to propose to the calling solver to increase or reduce the time step (Section 4.4.4).
- The computation of so-called tangent operator blocks, which in case of models are the derivatives of the internal state variables with respect to external state variables. This feature is however quite advanced and rarely used. As a consequence, its description is deported at the end of this chapter (Section 4.10).

4.4.1 First implementation

With the change of conventions in mind, the third implementation based on the `Model` DSL (see Section 4.3.2.3) can be readily adapted as follows (file `ChemicalReaction5.mfront`):

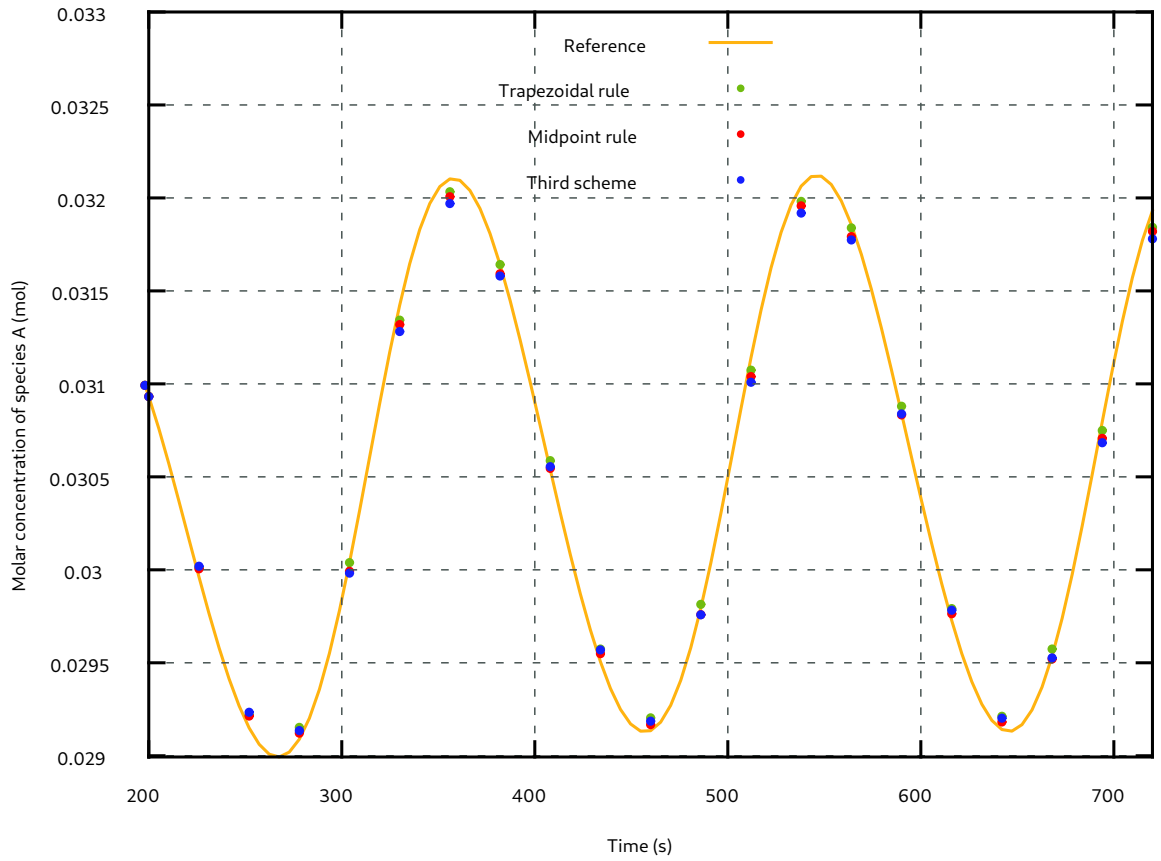

Figure 4.7: Evolution of the molar concentration of species A after the transition phase for the three integration schemes presented for a discretisation of the interval $[200 : 720]$ in 20 time steps.

```
@DSL DefaultModel;
@Model ChemicalReaction5;
@Author Thomas Helfer;
@Date 09/07/2022;
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@Import "ChemicalReaction-parameters.mfront";

@Integrator{
  // value of the temperature at the middle of the time step (mts)
  const auto T_mts = T + dT / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto B = k2_mts * (ca + cb);
  const auto K = k1_mts + k2_mts;
  const auto e = exp(-K * dt);
  const auto sum = ca + cb;
  ca = ca * e + (B / K) * (1 - e);
  cb = sum - ca;
}
```

The main difference is that the sum of the concentration must be saved in an intermediate variable named `sum` before updating the value of the molar concentration of species A.

This implementation gives rigorously the same solutions than the implementation based on the `Model` DSL. In particular, the `MTest` files used to test the third implementation (Section 4.3.2.3) can be used without change, except the name of the behaviour.

4.4.2 An alternative implementation

Some users do not like mixing variables at the beginning of the time step and variables at the end of the time step and may find the following implementation clearer:

```
@Integrator{
  // values at the beginning of the time step (bts)
  const auto ca_bts = ca;
  const auto cb_bts = cb;
  // value of the temperature at the middle of the time step (mts)
  const auto T_mts = T + dT / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto B = k2_mts * (ca_bts + cb_bts);
  const auto K = k1_mts + k2_mts;
  const auto e = exp(-K * dt);
  ca = ca * e + (B / K) * (1 - e);
  cb = ca_bts + cb_bts - ca;
}
```

Another solution is to use state variables rather than auxiliary state variables as described in the next section.

4.4.3 State variables and auxiliary state variables in the `Default` DSL

A careful reader shall have noticed that we declared the molar concentrations with the `@AuxiliaryStateVariable` and not using the `@StateVariable`.

The reason for this is a bit subtle, as this implementation does work without change if the `@StateVariable` keyword was used to declare `ca` and `cb`.

The difference is that `@StateVariable` declares the increments of molar concentrations as `dca` and `dcb`. After the `@Integrator` code block, the molar concentrations `ca` and `cb` are updated using `dca` and `dcb` automatically.

As those increments are initialized to zero, it is possible to modify directly `ca` and `cb`.

Using auxiliary state variables thus only avoids declaring those increments and this automatic update.

An alternative implementation, one may implement the same model by declaring `ca` and `cb` as state variables and computing their increments `dca` and `dcb` as follows (file `ChemicalReaction5b.mfront`):

```
@DSL DefaultModel;
@Model ChemicalReaction5b;
@Author Thomas Helfer;
@Date 09/07/2022;
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@Import "ChemicalReaction-parameters.mfront";

@Integrator{
  const auto T_mts = T + dT / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto B = k2_mts * (ca + cb);
  const auto K = k1_mts + k2_mts;
  const auto e = exp(-K * dt);
  dca = ((B / K) - ca) * (1 - e);
  dcb = -dca;
}
```

4.4.4 Computation of a time step scaling factor. Local variables

The model can propose to the calling solver to increase or decrease the current time step using the `@APrioriTimeStepScalingFactor` or the `@APosterioriTimeStepScalingFactor` code blocks:

- The `@APrioriTimeStepScalingFactor` is called before the model integration (after the `@InitLocalVariables` code block). This code block can thus be used, for example, to check:
  - the time increment is small enough.
  - the increments of the external state variables over the time step are reasonable.
- The `@APrioriTimeStepScalingFactor` is called after the model integration. This code block can thus be used to check if the increments of the state variables over the time steps are reasonable.

Those code blocks can return a pair containing a boolean stating if the time step shall be rejected (in this case, the model indicates to the solver that the integration has failed) and a time step reduction factor.

Consider the algorithm based on the midpoint rule presented in Section 4.3.2.2. The difference between the increments computed with $\theta = 1/2$ and $\theta = 1$ can be used to control the integration error and reduce the time step if needed. The following implementation is based on this idea (file `ChemicalReaction3b.mfront`):

```
@DSL DefaultModel;
@Model ChemicalReaction3b;
@Author Thomas Helfer;
@Date 09/07/2022;
@UseQt true;
@UnitSystem SI;
```

```
//! molar concentration of species A
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@Import "ChemicalReaction-parameters.mfront";

//! numerical parameter of the generalized midpoint rule
@Parameter real theta = 0.5;
theta.setEntryName("Theta");

@LocalVariable frequency k1_mts;
@LocalVariable frequency k2_mts;
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> B;

@InitLocalVariables {
  const auto T_mts = T + dT / 2;
  k1_mts = k01 * exp(-T_mts / Ta1);
  k2_mts = k02 * exp(-T_mts / Ta2);
  B = dt * (k2_mts * cb - k1_mts * ca);
}

@Integrator {
  constexpr auto zero = quantity<real, 0, 0, 0, 0, 0, 0, 1>{};
  const auto K = dt * (k1_mts + k2_mts);
  const auto sum = ca + cb;
  ca += B / (1 + K * theta);
  cb = sum - ca;
  // imposing positivity of the molar concentrations
  if(cb < zero){
    cb = zero;
    ca = sum;
  }
  if(ca < zero){
    ca = zero;
    cb = sum;
  }
}

@APosterioriTimeStepScalingFactor{
  constexpr auto eps = quantity<real, 0, 0, 0, 0, 0, 0, 1>{1e-10};
  const auto K = dt * (k1_mts + k2_mts);
  // increment for theta = 1/2
  const auto dca_1 = 2 * B / (2 + K);
  // increment for theta = 1
  const auto dca_2 = B / (1 + K);
  if (abs(dca_1 - dca_2) > eps) {
    return {false, 0.1};
  }
}
```

This implementation introduces the notion of *local variables* and the code block `@InitLocalVariables`. Local variables can be passed from one code block to the other. The `@InitLocalVariables` is the first code block called every time the behaviour integration is called.

The local variables `k1_mts`, `k2_mts` and `B` are used to minimize the computational call, in particular the Arrhenius terms which requires the evaluation of an exponential.

The `@APosterioriTimeStepScalingFactor` code block returns false if the absolue value of the difference between

the increments computed with $\theta = 1/2$ and $\theta = 1$ is greater than a threshold. If this is the case, the behaviour integration is said to have failed and we recommend that the time step is divided by a factor 10 (i.e. the recommended time step scaling factor is 0.1).

Note that the time step scaling factor are automatically bounded by two parameters `minimal_time_step_scaling_factor` and `maximal_time_step_scaling_factor` as highlighted by the output of `mfront-query`:

```
$ mfront-query --parameters ChemicalReaction3b.mfront
- ReferenceReactionRateCoefficientAB (k01)
- ReferenceReactionRateCoefficientBA (k02): reference rate coefficient`...
- ActivationTemperatureAB (Ta1): activation temperature reaction...
- ActivationTemperatureBA (Ta2): activation temperature reaction...
- Theta (theta): numerical parameter of the generalized midpoint rule
- minimal_time_step_scaling_factor: minimal value for the time step scaling factor
- maximal_time_step_scaling_factor: maximal value for the time step scaling factor
```

The default value of the parameters can be retrieved with `mfront-query` as follows:

```
$ mfront-query --parameter-default-value=minimal_time_step_scaling_factor \
               --parameter-default-value=maximal_time_step_scaling_factor \
               ChemicalReaction3b.mfront
0.1
1.79769e+308
```

Those default values can be changed using the keywords `@MinimalTimeStepScalingFactor` and `@MaximalTimeStepScalingFactor`.

By default, the time step scaling factor returned by the behaviour is ignored by `MTest`. It can be enabled by the `@DynamicTimeStepScaling` keyword as follows (file `ChemicalReaction3b.mtest`):

```
@Author Thomas Helfer;
@Date   09/08/2022;
@Model 'src/libBehaviour.so' 'ChemicalReaction3b';

@MaximumNumberOfSubSteps 100;
@DynamicTimeStepScaling true;
@MinimalTimeStepScalingFactor 0.1;
@MaximalTimeStepScalingFactor 1.4;

@Real 'B0' 0.1;
@Real 'T' 293.15;

@StateVariable 'MolarConcentrationOfSpeciesB' 'B0';
@ExternalStateVariable 'Temperature' 'T';

@Times {
  0, 360 in 10
};
```

With those setting, the comparison to the analytical solution is correct for a criterion value a low as 10^{-10}. However, the number of substeps is quite large as shown by the output of `MTest`:

```
Execution succeeded
-number of period:     154000
-number of iterations: 0
-number of sub-steps:  37
```

As a rule of thumb, relying on the solver for substepping is a bad idea and shall only be exceptional. The `RungeKuttaModel` DSL proposes several algorithms with automatic substepping which are much more efficient but have other drawbacks.

4.5 The `RungeKuttaModel` DSL

The `RungeKuttaModel` DSL is meant to integration the Ordinary Differential Equation (4.3) using a Runge-Kutta algorithm [12].

4.5.1 Numerical scheme

Those Runge-Kutta algorithms estimates the solution at the end of the time step by the weighted sum of approximations $\Delta\vec{\tilde{Y}}$ of the increment $\Delta\vec{Y}$ as follows:

$$\vec{Y}\Big|_{t+\Delta t} = \vec{Y}\Big|_t + \sum_{i=1}^{n} b_i \Delta\vec{\tilde{Y}}^{(i)}$$

The approximations $\Delta\vec{\tilde{Y}}^{(i)}$ are given by successive evaluations of the rate function $\vec{G}$:

$$\begin{cases} \Delta\vec{\tilde{Y}}^{(1)} = \Delta t\,\vec{G}\Big(\vec{Y}^{(1)}, \vec{Z}\Big|_t\Big) & \text{with} \quad \vec{Y}^{(1)} = \vec{Y}\Big|_t \\ \Delta\vec{\tilde{Y}}^{(2)} = \Delta t\,\vec{G}\Big(\vec{Y}^{(2)}, \vec{Z}\Big|_{t+c_2\,\Delta t}\Big) & \text{with} \quad \vec{Y}^{(2)} = \vec{Y}\Big|_t + a_{21}\,\Delta\vec{\tilde{Y}}^{(1)} \\ \Delta\vec{\tilde{Y}}^{(3)} = \Delta t\,\vec{G}\Big(\vec{Y}^{(3)}, \vec{Z}\Big|_{t+c_3\,\Delta t}\Big) & \text{with} \quad \vec{Y}^{(3)} = \vec{Y}\Big|_t + a_{31}\,\Delta\vec{\tilde{Y}}^{(1)} + a_{32}\,\Delta\vec{\tilde{Y}}^{(2)} \\ \quad\vdots \\ \Delta\vec{\tilde{Y}}^{(n)} = \Delta t\,\vec{G}\Big(\vec{Y}^{(n)}, \vec{Z}\Big|_{t+c_n\,\Delta t}\Big) & \text{with} \quad \vec{Y}^{(n)} = \vec{Y}\Big|_t + a_{n1}\,\Delta\vec{\tilde{Y}}^{(1)} + \ldots + a_{n,n-1}\,\Delta\vec{\tilde{Y}}_{n-1} \end{cases}$$

with $\vec{Z}\Big|_{t+c_j\,\Delta t} = \vec{Z}\Big|_t + c_j\,\Delta\vec{Z}$ being the estimates of $\vec{Z}$ at time $t + c_j\,\Delta t$.

In the following, $\vec{Y}^{(j)}$ is denoted as the estimate of $\vec{Y}$ at the j-th step of the algorithm.

A particular algorithm is thus caractherized by the number of evaluations n and the coefficients a_{ij}, b_i and c_i.

A classical fourth order Runge-Kutta algorithm
The classical fourth order Runge-Kutta is based on the following scheme:

$$\vec{Y}\Big|_{t+\Delta t} = \vec{Y}\Big|_t + \frac{1}{6}\left(\Delta\vec{\tilde{Y}}^{(1)} + 2\,\Delta\vec{\tilde{Y}}^{(2)} + 2\,\Delta\vec{\tilde{Y}}^{(3)} + \Delta\vec{\tilde{Y}}^{(4)}\right)$$

with:

$$\begin{cases} \Delta\vec{\tilde{Y}}^{(1)} = \Delta t\,\vec{G}\Big(\vec{Y}^{(1)}, \vec{Z}\Big|_t\Big) & \text{with} \quad \vec{Y}^{(1)} = \vec{Y}\Big|_t \\ \Delta\vec{\tilde{Y}}^{(2)} = \Delta t\,\vec{G}\left(\vec{Y}^{(2)}, \vec{Z}\Big|_t + \dfrac{\Delta\vec{Z}}{2}\right) & \text{with} \quad \vec{Y}^{(2)} = \vec{Y}\Big|_t + \dfrac{\Delta\vec{Y}^{(1)}}{2} \\ \Delta\vec{\tilde{Y}}^{(3)} = \Delta t\,\vec{G}\left(\vec{Y}^{(3)}, \vec{Z}\Big|_t + \dfrac{\Delta\vec{Z}}{2}\right) & \text{with} \quad \vec{Y}^{(3)} = \vec{Y}\Big|_t + \dfrac{\Delta\vec{Y}^{(2)}}{2} \\ \Delta\vec{\tilde{Y}}_4 = \Delta t\,\vec{G}\Big(\vec{Y}^{(4)}, \vec{Z}\Big|_{t+\Delta t}\Big) & \text{with} \quad \vec{Y}^{(4)} = \vec{Y}\Big|_t + \Delta\vec{Y}^{(3)} \end{cases}$$

Sometimes, an algorithm of high order allow to recover a lower order estimate by recombining the increment estimates $\Delta\vec{\tilde{Y}}$. In this case, the higher order estimate is called the corrector and the lower order estimate is called the predictor. The difference between the predictor and the corrector can be used to reduce the time step locally to satisfy a given criterion. Such a procedure is called substepping in the following.

4.5.2 A brief overview of the `RungeKuttaModel` DSL

The `RungeKuttaModel` DSL provides the following algorithms:

- `euler`: the explicit Euleur algorithm.
- `rk2`: a second order Runge-Kutta algorithm.
- `rk4`: a fourth order Runge-Kutta algorithm.
- `rk42`: a Runge-Kutta algorithm with automatic time substepping with an predictor of order 2 and a corrector of order 4.
- `rk54`: the Fehlberg algorithm with automatic time substepping an predictor of order 4 and a corrector of order 5.

4.5.2.1 The `@Derivative` code block

The ordinary differential equation is defined in a code block introduced by the `@Derivative` keyword. This code block has the following conventions:

- For each state variable $\vec{y}$, `y` denotes the current estimate of $\vec{y}$ for the current step of the algorithm, i.e. $\vec{y}^{(j)}$.
- For each state variable `y`, the variable `dy` stands, inside this code block, for the rate of `y`. Using unicode symbols, the rate of `y` can also be refered to as ∂_t`y`. This variable must be set by the `@Derivative` code block.
- For each external state variable $\vec{z}$, the variable `dz` (or ∂_t`z`) stands, inside this code block, for the rate of $\vec{z}$.

4.5.2.2 The `@UpdateAuxiliaryStateVariables` code block

The difference between state variables and auxiliary state variables is much clearer as auxiliary state variables have no associated rate. Auxiliary state variables are meant to be updated after the integration of the ordinary differential equation in a code block named `@UpdateAuxiliaryStateVariables`.

More precisely, the `@UpdateAuxiliaryStateVariables` code block can be called several times during a time step if the alorithm selected handles time substepping: in this case, the `@UpdateAuxiliaryStateVariables` code block is called after each successful substep.

4.5.3 Implementation

With those conventions in mind, a suitable implementation is as follows (file `ChemicalReaction6.mfront`):

```
@DSL RungeKuttaModel;
@Model ChemicalReaction6;
@Author Thomas Helfer;
@Date 09/07/2022;
@UseQt true;
@UnitSystem SI;

@Algorithm rk54;
@Epsilon 1e-14;

//! molar concentration of species A
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@Import "ChemicalReaction-parameters.mfront";

//! sum of the molar concentrations of species A and B
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> sum;

@InitLocalVariables {
  sum = ca + cb;
}

@Derivative{
  const auto k1 = k01 * exp(-T / Ta1);
  const auto k2 = k02 * exp(-T / Ta2);
  dca = k2 * sum - (k1 + k2) * ca;
}

@UpdateAuxiliaryStateVariables {
  cb = sum - ca;
}
```

The implementation of the `@Derivative` code block is very close to the mathematical expression given by Equation (4.3).

In this example, the Fehlberg algorithm is used. A stringent criterion value of 10^{-14} is used to validate the time step: if the norm of the difference between the state variables computed by the corrector and by the predictor is greater than this value, automatic time substepping is triggered.

The implementation can be tested with the file `ChemicalReaction6.mtest`) file which is readily adapted from the script discussed in Sections 4.3.1.3 and 4.3.1.4.

This implementation satisfies the non regression test (comparison to the analytical value) with a comparison criterion as low as 10^{-14}.

The `debug` mode of `MFront` gives some information about the local convergence of the Runge-Kutta algorithm. This mode is activatived by the `--debug` command line option as follows:

```
$ mfront --obuild --interface=generic --debug ChemicalReaction6.mfront
```

The output of the execution of `ChemicalReaction6.mtest`) file is then much more verbose:

```
$ mtest ChemicalReaction6.mtest
....
resolution from 356.4 to 360

ChemicalReaction6::integrate() : beginning of resolution
ChemicalReaction6::integrate() : from 0 to 3.6 with time step 3.6
ChemicalReaction6::integrate() : error 3.52248e-14
ChemicalReaction6::integrate() : reducing time step by a factor 0.6219
ChemicalReaction6::integrate() : from 0 to 2.23884 with time step 2.23884
ChemicalReaction6::integrate() : error 3.23635e-15
ChemicalReaction6::integrate() : increasing time step by a factor 1.00249
ChemicalReaction6::integrate() : from 2.23884 to 3.6 with time step 1.36116
ChemicalReaction6::integrate() : error 2.52165e-16
```

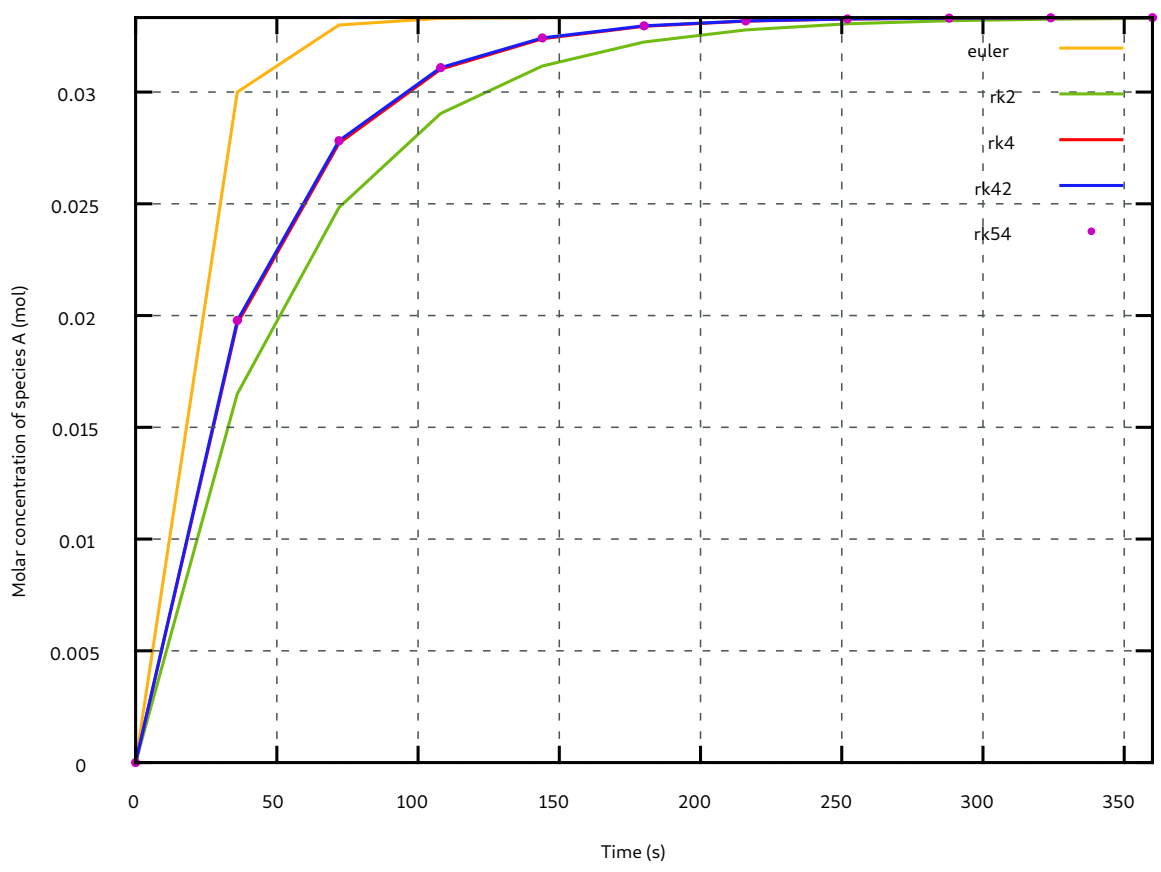

Figure 4.8: Comparison of the evolutions of the molar concentration of species A for the various Runge-Kutta algorithms.

A time discretisation in 100 steps does not allow to distinguish clearly the results of the various algorithms, except for the `euler` algorithm which is already showing its limits. With 10 steps, the difference are much clearer, as shown on Figure 4.8:

- The results obtained with the `rk42` and `rk54` algorithms are undistinguishable and both satisfies the non-regression test.
- The `rk4` algorithm is reasonably accurate, despite the lack of automatic sub-stepping.
- The `euler` and `rk2` algorithms gives significant errors.

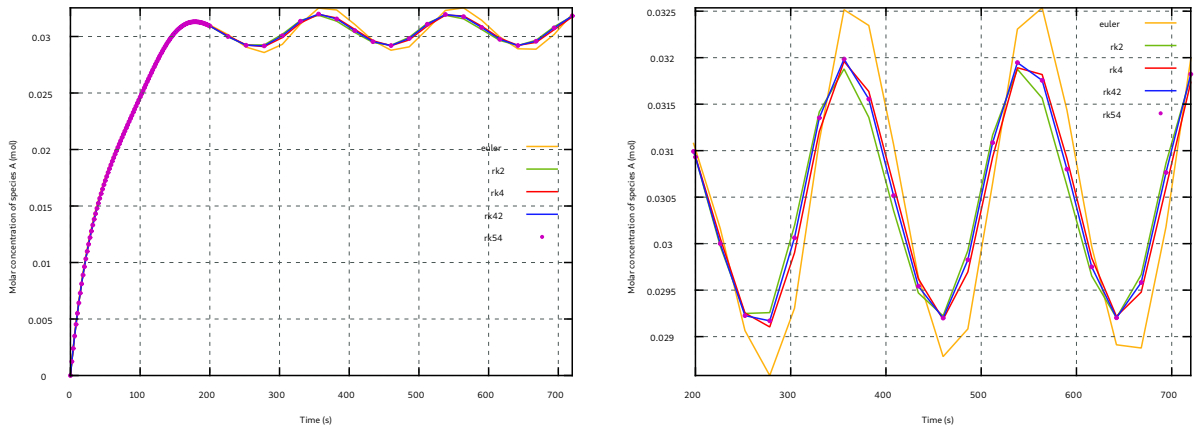

Figure 4.9: Comparison of the evolutions of the molar concentration of species A for the various Runge-Kutta algorithms in case of a varying temperature.

Conclusions are similar when considering the test with a varying temperature, as reported on Figure 4.9. It must however be highlighted that the relatively good performances of the `euler`, `rk2` and `rk4` algorithms are mostly du to the fact that the rate function `G` is linear in this example. As soon as `G` becomes non linear, automatic time substespping is mandatory to obtain reliable results.

4.5.4 [Advanced topic] Some words about the analysis of the `@Derivative` code block and code generation

In our experience, some users can be intrigated by how `MFront` works. Some are even repelled by the magic done by `MFront` during the analysis of the `MFront` file and the code generation step. This section is meant to provide details about how `MFront` analyses the `@Derivative` code block, modifies it and generate the various Runge-Kutta algorithms. Note that the material of this section is rather advanced: most `MFront` users can safely skip this section.

For the sake of simplicity, only the two simpliest algorithms (`euler` and `rk2`) are analysed.

4.5.4.1 Files generated by `MFront`

When processing a file, `MFront` creates two subdirectories named respectively `include` and `src` which basically contains the following files if no interface is specified:

```
$ mfront ChemicalReaction6.mfront
$ find include/ src/ -type f
include/TFEL/Material/ChemicalReaction6.hxx
include/TFEL/Material/ChemicalReaction6IntegrationData.hxx
include/TFEL/Material/ChemicalReaction6BehaviourData.hxx
src/targets.lst
src/ChemicalReaction6.cxx
```

The `targets.lst` file contains all the instructions to build librairies. This file is updated each time `MFront` is called.

The `ChemicalReaction6.cxx` source file mostly contains methods to handle the runtime modification of parameters of the model.

The `ChemicalReaction6BehaviourData.hxx` header file implements a class named `ChemicalReaction6BehaviourData` which contains the values at the beginning of the time step. The `ChemicalReaction6IntegrationData.hxx` header

file implements a class named `ChemicalReaction6IntegrationData` which contains the increments of the external state variable, the time increment, etc.

The most interesting header file is `ChemicalReaction6.hxx` which implements a class name `ChemicalReaction6`. This class inherits from the two previous classes `ChemicalReaction6BehaviourData` and `ChemicalReaction6IntegrationData`.

The class `ChemicalReaction6` contains the local variables, most of the user defined code, and the numerical algorithm used to integrate the model.

All thoses classes are declared in the `tfel::material` classes.

The main methods of interest of the `ChemicalReaction6` class are:

- `initialize`. This method mostly contains the code declared in the `@InitiaLocalVariables` code block.
- `integrate`. This methods implements the model integrate over the time step. In the case of the `RungeKuttaModel` DSL, this methods relies on a method called `computeDerivative` which contains a transformed version of the the code declared in the `@Derivative` code block.

The `integrate` method depends on the algorithm selected. This method is described in the case of the `euler` and `rk2` algorithms in paragraphs 4.5.4.3 and 4.5.4.4 respectively.

The next paragraph is devoted to the implementation of the `computeDerivative` method which is independant of the algorithm selected.

> **A word of caution**
> The implementations described below have been generated by Version 4.1 of `TFEL`. The reader must be aware that these implementations can evolve significantly from one version to the other.
> However, those implementations illustrates the work done by `MFront` behind the scene.

4.5.4.2 Implementation of the `computeDerivative` method

The `computeDerivative` method is implemented as follows:

```
[[nodiscard]] bool computeDerivative() {
  using namespace std;
  using namespace tfel::math;
#line 27 "ChemicalReaction6.mfront"
  const auto k1 = this->k01 * exp(-this->T_ / this->Ta1);
#line 28 "ChemicalReaction6.mfront"
  const auto k2 = this->k02 * exp(-this->T_ / this->Ta2);
#line 29 "ChemicalReaction6.mfront"
  this->dca = k2 * this->sum - (k1 + k2) * this->ca_;
  return true;
} // end of ChemicalReaction6::computeDerivative
```

This implementation mostly contains the code of the `@Derivative` code block after a slight transformation. One may notice that:

- The two namespaces `std` and `tfel::math` are automatically declared. As the `ChemicalReaction6` is declared in the `tfel::material` namespace, those declarations allows the user to have access to the features of the standard `C++` library and features of the `TFEL/Math` and `TFEL/Material` libraries.
- `MFront` inserted lines starting with the preprocessor directive `#line`, which refers to the initial lines in the `MFront` source file. Thanks to those lines, the compiler will indicate errors in the `MFront` source file and not in the generated sources. Note that those lines are not inserted in `debug` mode.
- The `this->` specifier has been added to add data members to comply with `C++` rules regarding inheritance of template classes.
- the state variable `ca` and the external state variable `T` have been replaced by two variables named respectively `ca_` and `T_`. Those variables contains the estimates `ca` and `T` for the current step of the algorithm (See Section 4.5.1).

The variables `ca_` and `T_` are initialized in the `initialize` code block. They can be update by the `integrate` between two steps of the selected algorithm.

The reader may have noticed that the `computeDerivative` method returns a boolean. This boolean is used to validate the current step of the algorithm. This is very useful with algorithm with automatic time substepping as the first estimates provided by the `Runge-Kutta` algorithm can be unphysical for very non linear rate functions

G. Guided by physical considerations, the user can thus force a substepping by returing `false` in the `@Derivative` code block. One may also notice that `MFront` automatically adds the `return true;` statement at the end of the method.

> **A small exercice for the reader**
> In the case of the chemical reaction considered in this section, the user may force the substepping if the estimate of the molar concentration of species A is lower than zero and if it exceed the total amount of matter of species A and B at the beginning of the time step.

4.5.4.3 Implementation of the `integrate` method for the `euler` algorithm

The Euler algorithm is based on the following scheme:

$$\vec{Y}\Big|_{t+\Delta t} = \vec{Y}\Big|_t + \Delta\,\vec{\tilde{Y}}^{(1)}$$

with $\Delta\,\vec{\tilde{Y}}^{(1)} = \Delta t\,\vec{G}\left(\vec{Y}\Big|_t, \vec{Z}\Big|_t\right)$

The `integrate` method is thus implemented as follows:

```
[[nodiscard]] IntegrationResult integrate(const SMFlag, const SMType smt) override {
  using namespace std;
  using namespace tfel::math;
  if (!this->computeDerivative()) {
    return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                               hypothesis, NumericType, use_qt>::FAILURE;
  }
  this->ca += this->dt * (this->dca);
  this->updateAuxiliaryStateVariables(this->dt);
  if (smt != NOSTIFFNESSREQUESTED) {
    return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                               hypothesis, NumericType, use_qt>::FAILURE;
  }
  return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                             hypothesis, NumericType, use_qt>::SUCCESS;
} // end of ChemicalReaction6::integrate
```

This method takes two arguments which are not meaningful for models. The `computeDerivative` method is called with the `ca_` and `T_` variables initialized respectively to the values of molar concentration of species A and temperature at the beginning of the time step (this initialization is done in the `initialize` method).

If the call to `computeDerivative` is successful, the molar concentration of species A is updated and the `updateAuxiliaryStateVariables` method is called. The latter mostly contains the code provided by the `@AuxiliaryStateVariable` code block.

After a check that no tangent operator block have been requested (see Section 4.10), the `integrate` method ends successfully. Note that the name `NOSTIFFNESSREQUESTED` clearly highlights that the `RungeKuttaModel` is inherited from classes that were dedicated to (mechanical) behaviours.

4.5.4.4 Implementation of the `integrate` method for the `rk2` algorithm

The rk2 algorithm is based on the following scheme:

$$\vec{Y}\Big|_{t+\Delta t} = \vec{Y}\Big|_t + \Delta\,\vec{\tilde{Y}}^{(2)}$$

with:

$$\begin{cases} \Delta\,\vec{\tilde{Y}}^{(1)} = \Delta t\,\vec{G}\left(\vec{Y}\Big|_t, \vec{Z}\Big|_t\right) \\ \Delta\,\vec{\tilde{Y}}^{(2)} = \Delta t\,\vec{G}\left(\vec{\tilde{Y}}^{(2)}, \vec{Z}\Big|_t + \dfrac{\Delta t}{2}\,\vec{Z}\right) \end{cases} \qquad \text{with} \quad \vec{\tilde{Y}}^{(2)} = \vec{Y}\Big|_t + \frac{1}{2}\,\Delta\,\vec{\tilde{Y}}^{(1)}$$

The `integrate` method is thus implemented as follows:

```
[[nodiscard]] IntegrationResult integrate(const SMFlag, const SMType smt) override {
  using namespace std;
  using namespace tfel::math;
  constexpr auto cste1_2 = NumericType{1} / NumericType{2};
  // Compute K1's values
  if (!this->computeDerivative()) {
    return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                               hypothesis, NumericType, use_qt>::FAILURE;
  }
  this->dca_K1 = (this->dt) * (this->dca);
  this->ca_ += cste1_2 * (this->dca_K1);
  this->T_ = this->T + cste1_2 * (this->dT);
  if (!this->computeDerivative()) {
    return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                               hypothesis, NumericType, use_qt>::FAILURE;
  }
  // Final Step
  this->ca += this->dt * (this->dca);
  this->updateAuxiliaryStateVariables(this->dt);
  if (smt != NOSTIFFNESSREQUESTED) {
    return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                               hypothesis, NumericType, use_qt>::FAILURE;
  }
  return MechanicalBehaviour<MechanicalBehaviourBase::GENERALBEHAVIOUR,
                             hypothesis, NumericType, use_qt>::SUCCESS;
} // end of ChemicalReaction6::integrate
```

The only noticeable difference with the implementation of the `euler` algorithm is the update of variables `ca_` and `T_` between two calls to the `computeDerivative` method.

4.6 The `ImplicitModel` DSL

The `ImplicitModel` DSL is meant to simplify the implementation of the models using the generalized midpoint rule for non linear rate function $\vec{G}$.

Although the domain specific languages of the `Implicit` family are of utmost importance for non linear behaviours, they are seldom used for models. For instance, the `ImplicitModel` DSL is not appropriate to integrate Equation (4.3), whose rate function is linear. The corresponding implementation is however described for the sake of consistency with the other sections.

4.6.1 Numerical scheme

Following the generalized midpoint rule, the increment $\Delta\vec{Y}$ of the state variables satisfies the following equation (Section 4.3.2.2):

$$\Delta\vec{Y} = \Delta t\,\vec{G}\left(\vec{Y}\Big|_{t+\theta\Delta t}, \vec{Z}\Big|_{t+\theta\Delta t}\right)$$

This equation can be rewritten by introducing the residual function $\vec{F}$ as follows:

$$\vec{F}\left(\Delta\vec{Y}\right) = \vec{0} \quad \text{with} \quad \vec{F}\left(\Delta\vec{Y}\right) = \Delta\vec{Y} - \Delta t\,\vec{G}\left(\vec{Y}\Big|_{t+\theta\Delta t}, \vec{Z}\Big|_{t+\theta\Delta t}\right). \tag{4.8}$$

The `ImplicitModel` DSL proposes several iterative algorithms to solve Equation (4.8).

The default algorithm is the standard Newton-Raphson's algorithm. Let $\Delta\vec{Y}^{(n)}$ the current estimate of the solution, the next estimate $\Delta\vec{Y}^{(n+1)}$ is the root of the linear approximation of F at $\Delta\vec{Y}^{(n)}$, i.e.:

$$\Delta\vec{Y}^{(n+1)} = \Delta\vec{Y}^{(n)} - \left(\frac{\mathrm{d}F}{\mathrm{d}\Delta\vec{Y}}\left(\Delta\vec{Y}^{(n)}\right)\right)^{-1} F\left(\Delta\vec{Y}^{(n)}\right).$$

$\dfrac{\mathrm{d}F}{\mathrm{d}\Delta\vec{Y}}\left(\Delta\vec{Y}^{(n)}\right)$ is called the jacobian matrix and it generally denoted J.

By default, this algorithm converges when the norm of the residual becomes lower than a given threshold.

Other algorithms available
The other algorithms encompass the Powell's dog algorithm (named `PowellDogLeg`), the first and second Broyden algorithms (named `Broyden` and `Broyden2` respectively) and the Levenberg-Marquardt's algorithms (named `LevenbergMarquardt`).
Those algorithms are described in the documentation of the `TFEL/Math` library.
When one of these algorithms require the computation of the jacobian matrix, `MFront` proposes an alternative version where the jacobian matrix is computed by a centered finite difference scheme.

4.6.2 A brief overview of the `ImplicitModel` DSL

The main code blocks of the `ImplicitModel` DSL are:

- the `@Predictor` code block, which is called before the first iteration of the algorithm, but after the `@InitLocalVariables` code block. This code block is meant to provide an initial estimate $\Delta\,\vec{Y}^{(0)}$ of the increments. This code block is optional as the $\Delta\,\vec{Y}^{(0)}$ is initialized by default to zero.
- the `@Integrator` code block, which is meant to compute the residual $\vec{F}$ and the jacobian matrix J, if required by the selected algorithm. However, the residual and the jacobian matrix are computed by blocks, as described in the next paragraph.
- the `@AdditionalConvergenceChecks` code block which is meant to implement additional convergence checks.

4.6.2.1 Block decompositions

The increment $\Delta\,\vec{Y}$ and the residual $\vec{F}$ can be decomposed as follows:

$$\Delta\,\vec{Y}=\begin{pmatrix}\Delta\,y_1\\ \vdots\\ \Delta\,y_{n_y}\end{pmatrix}\quad\text{and}\quad\vec{F}=\begin{pmatrix}f_{y_1}\\ \vdots\\ f_{y_{n_y}}\end{pmatrix}. \tag{4.9}$$

The jacobian matrix can also be decomposed by blocks as follows:

$$J=\begin{pmatrix}\dfrac{\partial f_{y_1}}{\partial y_1} & \cdots & \dfrac{\partial f_{y_1}}{\partial y_{n_y}}\\ \vdots & \ddots & \vdots\\ \dfrac{\partial f_{y_{n_1}}}{\partial y_1} & \cdots & \dfrac{\partial f_{y_{n_y}}}{\partial y_{n_y}}\end{pmatrix} \tag{4.10}$$

4.6.2.2 Automatically defined variables

The following variables are automatically defined and available in all code blocks:

- `zeros`, which contains the current estimate of the increments of the state variables $\Delta\,\vec{Y}^{(n)}$. This vector is initialized to zero.
- `fzeros`, which contains the residual $\vec{F}$.
- `jacobian` (if required by the underlying algorithm), which contains the jacobian matrix J.
- for each state variable y, the variable `y` contains either the value of $\vec{y}$ at the beginning of the time step (before the convergence of the algorithm) or the value $\vec{y}$ at the end of the time step (after the convergence of the algorithm).
- for each state variable y, the variable `dy` corresponds to the current estimate of $\Delta\,y$. Note that `dy` is a view to a block of the `zero` variable. Hence, modifying `dy` directly modifies the `zero` variable.

The variables `zeros`, `fzeros` and `jacobian` are rarely used directly in pratice, except for debugging.

4.6.2.3 The `@Integrator` code block

The `@Integrator` code block defines appropriate views to the `fzeros` and `jacobian` variables to compute the residual and the jacobian by blocks:

- For each state variable y, a variable `fy` corresponds to the block of the residual associated with y (see Equation (4.9)).

- For each pairs of state variables y_1 and y_2, a variable `dfy1_ddy2` is defined and corresponds to the appropriate jacobian block (see Equation (4.10)).

Inspired by Equation (4.8), `MFront` automatically initializes the residual and the jacobian matrix before each iteration of the algorithm (i.e. before each call to the `@Integrator` code block) as follows:

- The residuals `fy` are initialized to the corresponding increments `dy`.
- The jacobian matrix is initalized to the identity.

4.6.3 Implementation

Applying Equation (4.8) in the case of our chemical reaction example (Equation (4.3)) leads to the following residual and jacobian:

$$\begin{cases} f_{[A]} = \Delta\,[A] - \Delta\,t\,k_2\left(\,[A]|_t + [B]|_t\right) - \Delta\,t\,(k_1+k_2)\left(\,[A]|_t + \theta\,\Delta\,[A]\right) \\ \dfrac{\partial f_{[A]}}{\partial \Delta\,[A]} = 1 - \theta\,\Delta\,t\,(k_1+k_2) \end{cases} \tag{4.11}$$

With the convention of the `@Integrator` code block in mind (Section 4.6.2.3), a suitable implementation of is a follows (file ChemicalReaction7.mfront):

```
@DSL ImplicitModel;
@Model ChemicalReaction7;
@Author Thomas Helfer;
@Date 09 / 07 / 2022;
@UseQt true;
@UnitSystem SI;
`
@Epsilon 1e-14;
@Theta 0.5;

//! molar concentration of species A
@StateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@Import "ChemicalReaction-parameters.mfront";

//! sum of the molar concentrations of species A and B
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> sum;
/*!
 * reaction rate coefficient of the reaction transforming species A to species
 * B at the middle of the time step
 */
@LocalVariable frequency k1_mts;
/*!
 * reaction rate coefficient of the reaction transforming species B to species
 * A at the middle of the time step
 */
@LocalVariable frequency k2_mts;

@InitLocalVariables {
  const auto T_mts = T + dT / 2;
  k1_mts = k01 * exp(-T_mts / Ta1);
  k2_mts = k02 * exp(-T_mts / Ta2);
  sum = ca + cb;
}

@Integrator {
  const auto vca = k2_mts * sum - (k1_mts + k2_mts) * (ca + theta * dca);
```

```
  fca -= dt * vca;
  dfca_ddca += theta * dt * (k1_mts + k2_mts);
}

@UpdateAuxiliaryStateVariables {
  cb = sum - ca;
}
```

The most interesting part concerns the `@Integrator` code block which directly translates Equation (4.11).

The update of the residual `fca` takes into account the fact that `fca` has been initialized to `dca`. Hence the line:

```
fca -= dt * vca;
```

is equivalent to the line:

```
fca = dca - dt * vca;
```

although more efficient.

This implementation gives exactly the same results as the implementation based on the midpoint rule presented in Section 4.3.2.2, as the reader may see by running the file `ChemicalReaction7.mtest`.

If the previous file is compiled in `debug` mode (See Section 4.5.3), the output of the execution of the `ChemicalReaction7.mtest` script would show that the Newton algorithms always converges in two steps. At the second iteration, the residual is zero, up to machine precision, as expected when solving a linear problem:

```
resolution from 356.4 to 360

ChemicalReaction7::integrate() : beginning of resolution
ChemicalReaction7::integrate() : iteration 0 : 4.02664e-07
ChemicalReaction7::integrate() : iteration 1 : 8.91261e-20
ChemicalReaction7::integrate() : convergence after 1 iterations
```

4.7 Applications

This section is devoted to several real-world applications:

- The oxidations of pipes of Zircaloy alloys (Section 4.7.1).
- The swelling of mixed Uranium Plutonium carbide (Section 4.7.4).
- The austenite - martensite transformation following the Koistinen-Marburger model (Section 4.7.5).
- The $\alpha \rightarrow \beta$ phase transition in Zircaloy alloys (Section 4.7.1).

4.7.1 Oxidation of pipes of Zircaloy alloys

Many models of the litterature describing the oxidation of pipes made of Zircaloy alloys at constant temperature takes the following form [13, 14]:

$$l_0^2(\tau) = K(T)\,\tau \tag{4.12}$$

where l_0 is the oxidation length and τ the time from the beginning of the experiment.

The function $K(T)$ is generally choosen of the form:

$$K(T) = K_0 \exp\left(-\frac{E_a}{RT}\right)$$

where K_0 is characteristic of the pipe considered (material and size), E_a is an activation energy ($J.mol^{-1}$) and R is the perfect gas constant.

A model appliable in case of non constant temperature can be obtained by differentiating the previous equation with respect to time:

$$2\,l_0\,\frac{\mathrm{d}l_0}{\mathrm{d}\tau} = K(T) \tag{4.13}$$

4.7.2 Numerical scheme

The Ordinary Differential Equation (4.13) can be integrated over a time step $[t : t + \Delta t]$ as follows:

$$l_0|_{t+\Delta t}{}^2 - l_0|_t{}^2 = \int_t^{t+\Delta t} K(T(\tau))\,\Delta\tau \approx K(T + \theta\,\Delta T)\,\Delta t,$$

which leads to the following update scheme:

$$l_0|_{t+\Delta t} = \sqrt{l_0|_t{}^2 + K(T + \theta\,\Delta T)\,\Delta t} \tag{4.14}$$

4.7.3 Implementation using the `Model` DSL

Following the update rule provided by Equation (4.14), the model can be readily implemented as follows:

```
@DSL Model;
@Model OxidationModel;
@Material Zy4;
@Author T. Guilbert, A. Charbal
@UnitSystem SI;
@UseQt true;

@Output length e;
e.setEntryName("OxidationLength");
e.setDepth(1);

@Input temperature T;
T.setGlossaryName("Temperature");
T.setDepth(1);
@Bounds T in [1273.15 : 1673.15];

@Parameter quantity<real, 0, 2, -1> K0 = 1500.7e-6;
@Parameter quantity<real, 1, 2, -2, 0, 0, 0, -1> Ea = 75085.0;

@Function OxidationLength {
  constexpr auto R = PhysicalConstants::R;
  const auto T_mts = (T + T_1) / 2;
  const auto K = K0 * exp(-Ea / (R * T_mts));
  e = power<1, 2>(e_1 * e_1 + K * dt);
}
```

Two things can be noted:

1. The use of the `TFEL/PhysicalConstants` library which the values of various physical constants in the SI unit system. This library is detailled on this page: https://thelfer.github.io/tfel/web/physical-constants.html. This library is compatible with quantities.
2. The use of the `power` function provided by the `TFEL/Math` library which is suitable for to raise a scalar to an integer or fractional value. In this case, the call to `power<1, 2>` is equivalent to the standard function `std::sqrt` for computing the square root, except that the `power` function is compatible with quantities.

4.7.4 Solid swelling of mixed Uranium–Plutonium carbide fuel

A simple model describing the swelling of mixed Uranium–Plutonium carbide fuels du to solid fission products (generally referred to as solid swelling) can be devised from the work of Billone et al. [15].

This model states that the solid swelling rate $\dot{s}$ is function of the porosity f and the burn-up rate $\dot{B}_u$ (in $at.\%$) as follows:

$$\dot{s} = C_1\,\exp\left(p_0 - p\right)\dot{B}_u$$

where C_1 and p_0 are two parameters of the model.

The implementation of the model is straightforward using the `DefaultModel` DSL (file UPuCSolidSwelling-Model.mfront) :

```
@DSL DefaultModel;
@Model SolidSwellingModel;
@Material UPuC;
@Author Thomas Helfer;
@Date 06 / 12 / 2007;
@Description {
  "Billone, M.C. and Jankus, V.Z. and Kramer, J.M. and Yang, C.I."
  "Progress in modeling carbide and nitride fuel performance in advanced "
  "LMFBRs. Advanced LMFBR fuels conference. INIS Reference Number: 9379733"
}

@StateVariable strain s;
s.setGlossaryName("SolidSwelling");

@ExternalStateVariable real Bu;
Bu.setGlossaryName("BurnUp (at.%)");

@ExternalStateVariable real f;
f.setGlossaryName("Porosity");

//! multiplicative coefficient of the swelling rate
@Parameter strain C1 = 8.e-3;
C1.setEntryName("SwellingRateCoefficient");
//! reference porosity
@Parameter real p0 = 4.e-2;
p0.setEntryName("ReferencePorosity");

@Integrator {
  const auto f_mts = f + df / 2;
  s += C1 * exp(p0 - f_mts) * dBu;
}  // end of function compute
```

4.7.5 Austenite - martensite transformation following the Koistinen-Marburger model

Koistinen and Marburger proposed a simple equation to predict the austenite phase fraction y in pure Fe-C alloys and plain carbon steels [16] as a function of the temperature T:

$$y(T) = \begin{cases} 1 & \text{if} \quad T > M_s \\ \exp\left(-K_m\left(M_s - T\right)\right) & \text{if} \quad T \leq M_s \end{cases}$$

where M_s is a limit temperature at which the austenite - martensite transformation begins and K_m a parameter of model.

The implementation of the model is straightforward using the `Model` DSL (file KoistinenMarburgerPhaseTransformation.mfront) :

```
@DSL Model;
@Model KoistinenMarburgerPhaseTransformation;
@Author Thomas Helfer;
@Date 27 / 01 / 2021;
@UnitSystem SI;
@UseQt true;
@Description{
  "Koistinen Dp, Marburger Re.. A general equation prescribing extent of"
  "austenite - martensite transformation in pure Fe-C alloys and plain carbon "
  "steels. Acta Metallurgica, vol.7, pp. 59-60, 1959."
}

@StateVariable real y;
y.setEntryName("AustenitePhaseFraction");
```

```
@PhysicalBounds y in [0:1];

@ExternalStateVariable temperature T;
T.setGlossaryName("Temperature");

@Parameter invert_type<temperature> Km = 2.47e-2;
@Parameter temperature Ms = 653.15;

@Function PhaseTransition {
  if (T < Ms) {
    y = exp(-Km * (Ms - T));
  } else {
    y = 1;
  }
}
```

4.7.6 Transformation of Zircaly alloys between α and β phases

At high temperature, Zircaloy alloys undergoes a phase transition between approximatively $700°C$ and $1\,000°C$ from a phase of compact hexagonal structure (phase α) to a phase of cubic structure (β structure).

This section implements to a simplified version of the model proposed by Massih and Jernkvist [17].

4.7.6.1 Description of the model

The model describes the evolution of the β phase fraction, denoted y_β.

4.7.6.1.1 Equilibrium

Table 4.2: Values of the material parameters T_0^α, A_1^α, A_2^α, T_0^β, A_1^β and A_2^β

T_0^α	1075
A_1^α	$2.713\,10^5$
A_2^α	$1.52\,10^5$
m^α	1.732
T_0^β	1250
A_1^β	$3.138\,10^4$
A_2^β	$2.2\,10^4$
m^β	0.929

For a given temperature T, the equilibrium β phase fraction $y_\beta^{\text{eq}}(T)$ is given by the following equation:

$$y_\beta^{\text{eq}}(T) = \frac{1}{2}\left[1 + \tanh\left(\frac{T - T_{\text{mid}}(x_O, x_H)}{T_{\text{span}}(x_O, x_H)}\right)\right]$$

where the temperatures T_{mid} and T_{span} are function of the weight fraction of excess oxygen x_0 and hydrogen x_H as follows:

$$\begin{cases} T_{\text{mid}}(x_O, x_H) = \dfrac{T_\alpha(x_O, x_H) + T_\alpha(x_O, x_H)}{2} \\ T_{\text{span}}(x_O, x_H) = \dfrac{1}{2.3}\,\dfrac{T_\beta(x_O, x_H) - T_\beta(x_O, x_H)}{2} \end{cases}$$

where the temperatures T_α and T_β are given by the following relationships:

$$\begin{cases} T_\alpha(x_O, x_H) = T_0^\alpha + A_1^\alpha\, x_O^{m^\alpha} - A_2^\alpha\, x_H \\ T_\beta(x_O, x_H) = T_0^\beta + A_1^\beta\, x_O^{m^\beta} - A_2^\beta\, x_H \end{cases}$$

The 6 material parameters T_0^α, A_1^α, A_2^α, T_0^β, A_1^β and A_2^β are reported in Table 4.2.

About the simplification of the model
The paper of [17] indicates that the model shall consider the dependency of the temperatures T_α and T_β to the temperature rate, as proposed by [18].
This refinement is not taken into account in this section and may explain some of the differences between our results (Figure 4.10 below) and the results of Massih and Jernkvist [17].

4.7.6.1.2 Kinetics

Table 4.3: Values of the material parameters A_{τ_r}, B_{τ_r}, T_{τ_r} and C_{τ_r}

A_{τ_r}	60 457
B_{τ_r}	18 129
T_{τ_r}	16 650
C_{τ_r}	25

The evolution of the β phase fraction, denoted y_β, is assumed to obey the following law:

$$\dot{y}_\beta = \frac{1}{\tau_r(T,\dot{T})}\left(y_\beta^{\text{eq}}(T) - y_\beta\right) \tag{4.15}$$

where the relaxation time $\tau_r(T,\dot{T})$ is given by the following formula:

$$\begin{cases} \tau_r(T,\dot{T}) = \dfrac{1}{A_{\tau_r} + B_{\tau_r}\left|\dot{T}\right|}\exp\left(\dfrac{T_{\tau_r}}{T}\right) & \text{if} \quad y_\beta \leq y_\beta^{\text{eq}}(T) \qquad \text{(heating)} \\ \tau_r(T,\dot{T}) = \dfrac{1}{A_{\tau_r} + B_{\tau_r}\left|\dot{T}\right|}\exp\left(\dfrac{T_{\tau_r}}{T + C_{\tau_r}\dot{T}}\right) & \text{if} \quad y_\beta \geq y_\beta^{\text{eq}}(T) \qquad \text{(cooling)} \end{cases}$$

The material parameters A_{τ_r}, B_{τ_r}, T_{τ_r} and C_{τ_r} are reported in Table 4.3.

4.7.6.2 Implementation based on the `RungeKuttaModel` DSL

A possible implementation of this model based on the `RungeKuttaModel` DSL is given by the following script (file ZircaloyAlloy_PhaseTransformation_MassihJernkvist2021.mfront):

```
@DSL RungeKuttaModel;
@Material ZircaloyAlloy;
@Model PhaseTransformation_MassihJernkvist2021;
@Author Thomas Helfer;
@Date 31 / 08 / 2022;
@UseQt true;
@UnitSystem SI;

@Description {
  "Solid state phase transformation kinetics in Zr-base alloys"
  "Massih, A. R. and Jernkvist, Lars O"
  "Scientific Reports. 2021"
}

@Algorithm rk54;
@Epsilon 1e-4;

//! fraction of the beta phase
@StateVariable real yβ;
yβ.setEntryName("BetaPhaseFraction");

@ExternalStateVariable real xₒ;
xₒ.setEntryName("ExcessOxygenWeightFraction");
@Bounds xₒ in [0:1e-2];
```

```
@ExternalStateVariable real xₕ;
xₕ.setEntryName("HydrogenWeightFraction");
@Bounds xₕ in [0:1e-3];

@Parameter temperature T0α = 1075;
@Parameter temperature A1α = 2.713e5;
@Parameter temperature A2α = 1.52e5;
@Parameter real mα  = 1.732;
@Parameter temperature T0β  = 1250;
@Parameter temperature A1β  = 3.138e4;
@Parameter temperature A2β  = 2.2e4;
@Parameter real mβ = 0.929;
@Parameter real cT = 2.3;

@Parameter temperature Tτ = 16650;
@Parameter frequency Aτ = 60457;
@Parameter invert_type<temperature> Bτ = 18129;
@Parameter time tτ = 25;

@Derivative {
  const auto Tα = T0α + A1α * pow(xₒ, mα) - A2α * xₕ;
  const auto Tβ = T0β + A1β * pow(xₒ, mβ) - A2β * xₕ;
  const auto T_mid = (Tβ + Tα) / 2;
  const auto T_span = (Tβ - Tα) / (2 * cT);
  const auto yβ_eq = (1 + tanh((T - T_mid) / T_span)) / 2;
  const auto τ = [this, yβ_eq] {
    if (yβ > yβ_eq) {
      const auto Tbτ = tτ * abs(dT);
      return exp(Tτ / (T + Tbτ)) / (Aτ + Bτ * abs(∂ₜT));
    }
    return exp(Tτ / T) / (Aτ + Bτ * abs(∂ₜT));
  }();
  ∂ₜyβ = (yβ_eq - yβ) / τ;
}
```

A few remarks can be made:

- We used unicode symbols, as detailled on this page to make the code closer to mathematical notations. The variable `∂ₜyβ` is interpreted as `dyβ` and `∂ₜT` is interpreted as `dT`.
- We used a `C++` trick to evaluate the characteristic time τ_r by relying on an λ-expression that is evaluated immediately after being defined. This requires to pass the `this` pointer to the capture list of the λ-expression. Note that only DSL derived from behaviours are guaranteed to implemented as `C++` classes. This code is equivalent to the following simplier by more verbose code:

```
auto τ = time{};
if (yβ > yβ_eq) {
  const auto Tbτ = tτ * abs(dT);
  τ = exp(Tτ / (T + Tbτ)) / (Aτ + Bτ * abs(∂ₜT));
} else {
  τ = exp(Tτ / T) / (Aτ + Bτ * abs(∂ₜT));
}
```

After being compiled with the `castem` interface, the previous implementation can be tested with `MTest` by the following script (file ZircaloyAlloy_PhaseTransformation_MassihJernkvist2021.mtest):

```
@Author Thomas Helfer;
@Date   31/08/2022;

@Behaviour<castem> 'src/libUmatZircaloyAlloy.so'
                   'umatzircaloyalloyphasetransformation_massihjernkvist2021';

@Real 'q' 100;
```

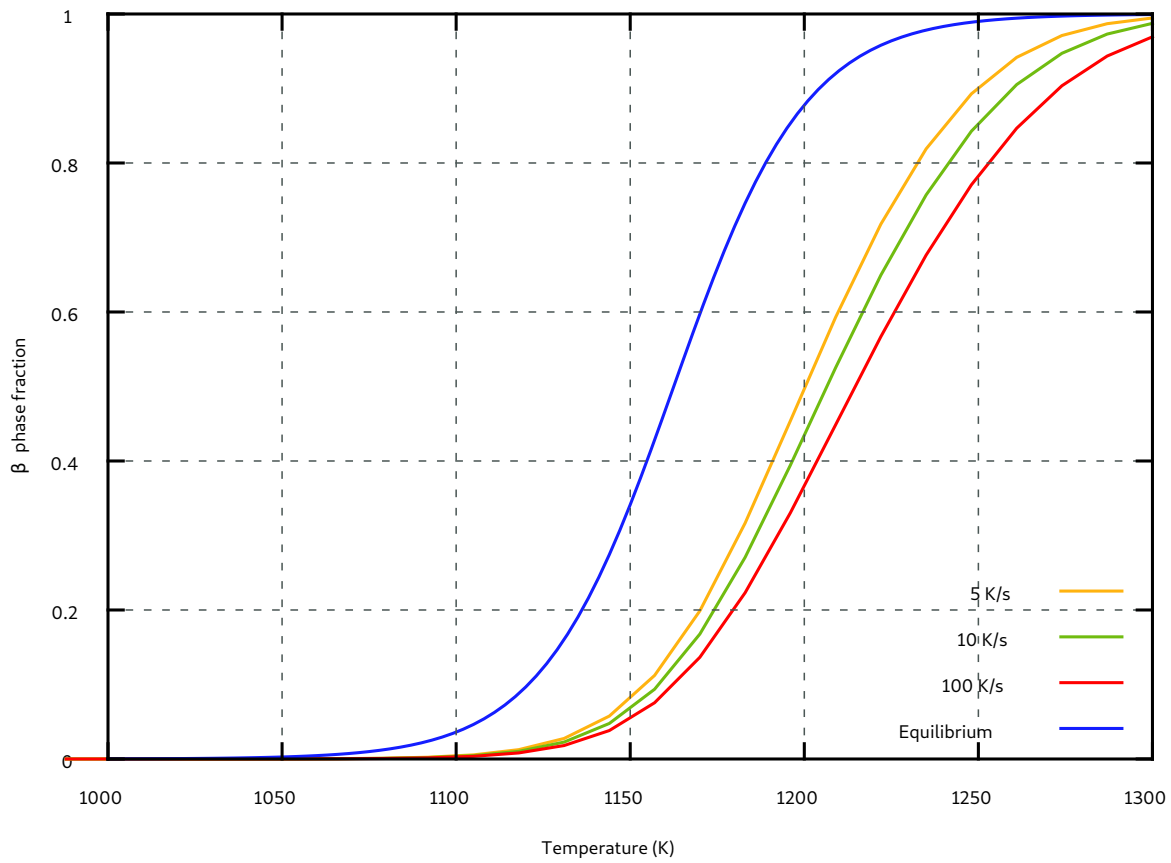

Figure 4.10: Evolution of the β phase fraction predicted by the simplified model Massih and Jernkvist at various heating rate

```
@ExternalStateVariable<function> 'Temperature' 'q*t' ;
@ExternalStateVariable 'ExcessOxygenWeightFraction' 0;
@ExternalStateVariable 'HydrogenWeightFraction' 0;

@Times{0, '1300/q' in 100};
```

The results of this script for various heating rate are reported on Figure 4.10.

An exercice for the reader
The previous implementation evaluates the equilibrium phase and the relaxation time at each evaluation of the rate function which involves evaluation of the special functions hyberolic tangent (`tanh`) and exponential (`exp`).
If one assumes that the temperature is slowly varying over the time step, Equation (4.15) can be approximated by the following equation:

$$\dot{y}_\beta = \frac{1}{\tau_r\left(T|_{t+\Delta t}, \dfrac{\Delta T}{\Delta t}\right)}\left(y_\beta^{\mathrm{eq}}\left(T|_{t+\Delta t}\right) - y_\beta\right)$$

This equation can be integrated exactly as follows:

$$y_\beta|_{t+\Delta t} = y_\beta^{\mathrm{eq}}\left(T|_{t+\Delta t}\right) + \left(y_\beta|_t - y_\beta^{\mathrm{eq}}\left(T|_{t+\Delta t}\right)\right)\exp\left(-\frac{\Delta t}{\tau_r\left(T|_{t+\Delta t}, \dfrac{\Delta T}{\Delta t}\right)}\right)$$

Implementing this relation with the `Model` DSL or the `DefaultModel` DSL is left as an exercice to the reader.

4.8 Usage in `MFront` behaviours (and models derived from behaviours)

The `@Model` keyword, which is available in all DSLs describing behaviours, including `DefaultModel`, `RungeKuttaModel` and `ImplicitDSLModel` allows to use external `MFront` models (see Figure 4.1).

For DSL describing mechanical behaviours, the keywords `@Swelling` and `@AxialGrowth` allow to define stress free expansions using external models.

Using an external model has the following consequences:

- The state variables of the external model will be automatically added to the list of the auxiliary state variables of the (calling) behaviour. Their increments will be declared as local variables.
- The external state variables of the external model will be added to the external state variables of the behaviour if required. The calling behaviour uses the external names of the external state variables to evaluate the model properly. Thanks to this external name, the behaviour searches if one of its external state variables, material properties and parameters matches. If not, a new external state variable is called.
- The external model will be called in the initialization phase, before the `@InitializeLocalVariables` code block. Hence, the increments of the state variables of the external model can be used by the user in all code blocks.

4.9 Usage in different solvers

4.9.1 Usage in `MGIS` and `MFEM/MGIS`

```
import os
import math
import numpy
import mgis.behaviour as mgis_bv
import mgis.model
```

```
# path to the test library
lib = 'src/libZircaloyAlloy-generic.so'
# modelling hypothesis
h = mgis_bv.Hypothesis.Tridimensional
# loading the behaviour
model = mgis.model.load(lib, 'ZircaloyAlloy_PhaseTransformation', h)
# number of integration points
nig = 1
# material data manager
m = mgis_bv.MaterialDataManager(model, nig)
# index of the fraction phase in the array of state variable
o = mgis_bv.getVariableOffset(model.isvs, 'BetaPhaseFraction', h)
# setting the temperature
T = 293.15 * numpy.ones(nig)
# type of storage
Ts = mgis_bv.MaterialStateManagerStorageMode.LocalStorage
mgis_bv.setExternalStateVariable(m.s1, 'Temperature', T, Ts)
# copy d.s1 in d.s0
mgis_bv.update(m)
# index of the first integration point
ni = 0
# index of the last integration point
ne = nig - 1
# values of the beta fraction phase for the first integration point
ybeta_i = [m.s0.internal_state_variables[ni][o]]
# values of the beta fraction phase for the last integration point
ybeta_e = [m.s0.internal_state_variables[ne][o]]
# time discretization
t0 = 0.
te = 3600.
dt = 1.
T0 = 293.15
Te = 1200
# integration
t = 0
while(t < te):
    t = t + dt
    T = ((Te-T0) / te * t + T0)  * numpy.ones(nig)
    mgis_bv.setExternalStateVariable(m.s1, 'Temperature', T, Ts)
    it = mgis_bv.IntegrationType.IntegrationWithoutTangentOperator
    mgis_bv.integrate(m, it, dt, 0, m.n)
    mgis_bv.update(m)
    ybeta_i.append(m.s1.internal_state_variables[ni][o])
    ybeta_e.append(m.s1.internal_state_variables[ne][o])
    print("{} {} {}".format(t, T[-1], ybeta_i[-1]))
```

4.9.2 Usage in `Cast3M`

4.9.3 Usage in `Manta`

4.9.4 Usage in `code_aster`

4.10 [Advanced topic] Computation of tangent operator blocks

One advantage of the DSLs derived from behaviours, such as the `DefaultModel` DSL, over the `Model` DSL is the ability to define tangent operator blocks, i.e. derivatives of state variables or auxiliary state variables with respect to the external state variables.

In this section, we will add the computation of the derivatives $\frac{\partial [A]|_{t+\Delta t}}{\partial \Delta T}$ and $\frac{\partial [B]|_{t+\Delta t}}{\partial \Delta T}$.

Such derivatives may be required when using models as a building block to solve multi-physics problems, for

exemple by coupling this model of chemical reaction with a partial differential equation treating heat transfer. One must however acknowledge that this kind of coupling is quite rare, so this section is mainly meant to introduce the concept of tangent operator which is fundamental for behaviours.

This section is thus quite advanced and can be skipped on a first read.

> **Blocks**
> The term block refers to the fact that those derivatives are packed in a contiguous array of memory which must be cut into blocks to retrieve the derivatives.

The compution of those derivatives introduces three changes described in depth in the following paragraphs:

- One must declare the tangent operator blocks using the `@TangentOperatorBlocks` keyword.
- One must write the code computing the tangent operator blocks in a code block introduced by the `@TangentOperator` keyword.
- For more efficiency, one can factorize some computations between the `@Integrator` and `@TangentOperator` code blocks.

4.10.1 Declaration of the tangent operator blocks

By default, the `DefaultModel` DSL does not declare any tangent operator block. This declaration can be done using any of the following keywords:

- `@TangentOperatorBlock`, which allows to declare one block.
- `@TangentOperatorBlocks`, which allows to declare several blocks at once.
- `@AdditionalTangentOperatorBlock`, which allows to add one block to the ones already declared.
- `@AdditionalTangentOperatorBlocks`, which allows to add several blocks to the ones already declared.

The `@AdditionalTangentOperatorBlock` and `@AdditionalTangentOperatorBlocks` do not really make sense in the `DefaultModel` DSL. There are more useful when a DSL already declares some tangent operator blocks by default, as it is the case for behaviours.

Note that those keywords must be called after the declaration of the state variables and external state variables involved.

4.10.2 Computation of the tangent operator blocks in the `DefaultModel` DSL

The tangent operator blocks can be computed in a code block named `@TangentOperator`. This code block is called, if required by the calling solver, **after** the `@Integrator` code block and **after** the update of the internal state variables.

This implementation requires to save the values of the molar concentrations at the beginning of the time step using local variables.

The implementation is as follows (file `ChemicalReaction5c.mfront`):

```
@DSL DefaultModel;
@Model ChemicalReaction5c;
@Author Thomas Helfer;
@Date 09 / 07 / 2022;
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@TangentOperatorBlocks{dca_ddT, dcb_ddT};

@Import "ChemicalReaction-parameters.mfront";
```

```
//! molar concentration of species A at the beginning of the time step
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca_bts;
//! molar concentration of species B at the beginning of the time step
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb_bts;

@InitLocalVariables {
  ca_bts = ca;
  cb_bts = cb;
}

@Integrator {
  const auto T_mts = T + dT / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto sum = ca_bts + cb_bts;
  const auto B = k2_mts * sum;
  const auto K = k1_mts + k2_mts;
  const auto e = exp(-K * dt);
  ca = ca * e + (B / K) * (1 - e);
  cb = sum - ca;
}

@TangentOperator {
  // avoid warning
  static_cast<void>(smt);
  const auto T_mts = T + dT / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto dk1_mts_ddT = -k1_mts / (2 * Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto dk2_mts_ddT = -k2_mts / (2 * Ta2);
  const auto sum = ca_bts + cb_bts;
  const auto B = k2_mts * sum;
  const auto dB_ddT = (ca_bts + cb_bts) * dk2_mts_ddT;
  const auto K = k1_mts + k2_mts;
  const auto dK_ddT = dk1_mts_ddT + dk2_mts_ddT;
  const auto e = exp(-K * dt);
  const auto de_ddT = -dt * e * dK_ddT;
  dca_ddT = ca_bts * de_ddT - (B / K) * de_ddT +  //
            ((1 - e) / K) * (dB_ddT - dK_ddT / K);
  dcb_ddT = -dca_ddT;
}
```

The previous implementation is however quite inefficient as the variables `k1_mts`, `k2_mts` and `e` are evaluated twice, leading to the evaluation of three exponentials per behaviour integration if the consistent tangent operator is computed.

There are mostly two solutions to this:

1. Use more local variables.
2. Compute the tangent operator in the `@Integrator` code block.

Each solutions has its advantages and drawbacks.

4.10.2.1 First alternative implementation of the computation of the tangent operator blocks

The first solution is to define `k1_mts`, `k2_mts` and `e` as local variables and computed them in the `@InitLocalVariables` code block. This can be done as follows (file `ChemicalReaction5d.mfront`):

```
@DSL DefaultModel;
@Model ChemicalReaction5d;
@Author Thomas Helfer;
@Date 09 / 07 / 2022;
@UseQt true;
```

```
@UnitSystem SI;

//! molar concentration of species A
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@TangentOperatorBlocks{dca_ddT, dcb_ddT};

@Import "ChemicalReaction-parameters.mfront";

//! molar concentration of species A at the beginning of the time step
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca_bts;
//! molar concentration of species B at the beginning of the time step
@LocalVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb_bts;
/*!
 * rate coefficient of the reaction transforming species A to species B at the
 * middle of the time step
 */
@LocalVariable frequency k1_mts;
/*!
 * rate coefficient of the reaction transforming species B to species B at the
 * middle of the time step
 */
@LocalVariable frequency k2_mts;
//! exponential decay
@LocalVariable real e;

@InitLocalVariables {
  const auto T_mts = T + dT / 2;
  k1_mts = k01 * exp(-T_mts / Ta1);
  k2_mts = k02 * exp(-T_mts / Ta2);
  ca_bts = ca;
  cb_bts = cb;
}

@Integrator {
  const auto sum = ca_bts + cb_bts;
  const auto B = k2_mts * sum;
  const auto K = k1_mts + k2_mts;
  e = exp(-K * dt);
  ca = ca * e + (B / K) * (1 - e);
  cb = sum - ca;
}

@TangentOperator {
  // avoid warning
  static_cast<void>(smt);
  const auto sum = ca_bts + cb_bts;
  const auto B = k2_mts * sum;
  const auto K = k1_mts + k2_mts;
  const auto dk1_mts_ddT = -k1_mts / (2 * Ta1);
  const auto dk2_mts_ddT = -k2_mts / (2 * Ta2);
  const auto dB_ddT = (ca_bts + cb_bts) * dk2_mts_ddT;
  const auto dK_ddT = dk1_mts_ddT + dk2_mts_ddT;
  const auto de_ddT = -K * e * dK_ddT;
  dca_ddT = ca * de_ddT - (B / K) * de_ddT +  //
            ((1 - e) / K) * (dB_ddT - dK_ddT / K);
```

```
  dcb_ddT = -dca_ddT;
}
```

4.10.2.2 Second alternative implementation of the computation of the tangent operator blocks

The `@ProvidesTangentOperator` keyword must be used to indicate that the tangent operator blocks are computed in the `@Integrator`.

A boolean value named `computeTangentOperator_` is set to `true` if the calling solver requested the computation of the tangent operator blocks.

With those details in mind, the following implementation can be written (file `ChemicalReaction5e.mfront`):

```
@DSL DefaultModel;
@Model ChemicalReaction5e;
@Author Thomas Helfer;
@Date 09 / 07 / 2022;
@UseQt true;
@UnitSystem SI;

//! molar concentration of species A
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> ca;
ca.setEntryName("MolarConcentrationOfSpeciesA");
//! molar concentration of species B
@AuxiliaryStateVariable quantity<real, 0, 0, 0, 0, 0, 0, 1> cb;
cb.setEntryName("MolarConcentrationOfSpeciesB");

@TangentOperatorBlocks{dca_ddT, dcb_ddT};

@Import "ChemicalReaction-parameters.mfront";

@ProvidesTangentOperator;
@Integrator {
  const auto ca_bts = ca;
  const auto cb_bts = cb;
  const auto sum = ca_bts + cb_bts;
  const auto T_mts = T + dT / 2;
  const auto k1_mts = k01 * exp(-T_mts / Ta1);
  const auto k2_mts = k02 * exp(-T_mts / Ta2);
  const auto B = k2_mts * sum;
  const auto K = k1_mts + k2_mts;
  const auto e = exp(-K * dt);
  ca = ca_bts * e + (B / K) * (1 - e);
  cb = sum - ca;
  if (computeTangentOperator_) {
    const auto dk1_mts_ddT = -k1_mts / (2 * Ta1);
    const auto dk2_mts_ddT = -k2_mts / (2 * Ta2);
    const auto dB_ddT = sum * dk2_mts_ddT;
    const auto dK_ddT = dk1_mts_ddT + dk2_mts_ddT;
    const auto de_ddT = -dt * e * dK_ddT;
    dca_ddT = ca_bts * de_ddT - (B / K) * de_ddT +  //
              ((1 - e) / K) * (dB_ddT - B * dK_ddT / K);
    dcb_ddT = -dca_ddT;
  }
}
```

4.10.3 Computation of the tangent operator blocks in the `ImplicitModel` DSL

Chapter 5

The role of behaviours

Point wise models describes the evolution over a time step of a set internal state variables of a material given imposed changes of as set of external state variables describing its thermodynamic environment.

In a sense, behaviours generalises point wise models. Not only behaviours describes the evolution of a set internal state variables but also describes the local reaction of a material, described by a set of thermodynamical forces, given the evolution of a set of gradients. Those thermodynamic forces are used in conservation laws to describe the equilibrium of the material.

For the sake of clarity, let us give a set of examples:

- The classical Fourier's computes the heat flux as a function of the temperature gradient. The thermal equilibrium is associated with the conservation of energy.
- In mechanics, under the infinitesimal perturbation hypothesis, the role of mechanical behaviour is to compute the stress at the end of the time step given the strain increment. The mechanical equilibrium is associated to the conservation of the linear and angular momenta.

Chapter 6

Implementing behaviours using domain specific languages of the `Default` family

This chapter is devoted to the implementation of behaviours using one of the following domain specific languages:

- `Default`, which handles to strain based mechanical behaviours.
- `DefaultFiniteStrain`, which handles finite strain mechanical behaviours.
- `DefaultCZM`, which handles to cohesive zone models.
- `DefaultGenericBehaviour`, which handles to generalized behaviours.
- `DefaultModel`, which handles to simple point-wise models (see Section **??**).

All those DSLs shares the same syntax and conventions and mostly only differs from the fact that most of them (except the `DefaultGenericBehaviour` DSL) automatically set the gradients and the thermodymanic forces. For instance, the `Default` DSL imposes that the only gradient is the strain $\underline{\varepsilon}^{\mathrm{to}}$ and the only thermodynamic force is the stress $\underline{\sigma}$.

Section **??**, devoted to point-wise models, has already described most of the keywords used by thoses DSLs. This chapter is mostly focused on aspects specific to behaviours, i.e. the computation of the thermodynamic forces, the prediction operator and the tangent operator.

6.1 Typical usage of the domain specific languages of the `Default` family

The domain specific languages of the `Default` family does not impose any resolution algorithm to integrate the constitutive equations over the time step.

As such, they are generally used in two diametrically opposed situations:

- for simple behaviours for which the integration can be performed analytically. This case incompasses simple isotropic plastic materials with linear isotropic hardening or linear kinematic hardening, some damage behaviours, visco-elasticity behaviours of the Maxwell kind, hyperelastic behaviours, etc..
- for interfacing external material libraries. This strategy has been used to use behaviours built on neural networks and to reuse existing `UMAT` implementations in `fortran`. In those cases, `MFront` is mostly used a wrapper.

6.2 Implementing the isotropic Hooke's law

The Hooke's law assumes a linear relationship between the stress $\underline{\sigma}$ and the strain $\underline{\varepsilon}^{\mathrm{to}}$. In the isotropic case, this relationship can be expressed as:

$$\underline{\sigma} = \lambda\,\mathrm{tr}(\underline{\varepsilon}^{\mathrm{to}})\,\underline{I} + 2\,\mu\,\underline{\varepsilon}^{\mathrm{to}}$$

where λ and μ are respectively the first Lamé's coefficient and the shear modulus.

After time discretization, this relationship can be written:

$$\begin{aligned}\underline{\sigma}|_{t+\Delta t} &= \lambda|_{t+\Delta t}\,\mathrm{tr}\left(\underline{\varepsilon}^{\mathrm{to}}|_{t+\Delta t}\right)\underline{I} + 2\,\mu\,\underline{\varepsilon}^{\mathrm{to}}|_{t+\Delta t}\\ &= \lambda|_{t+\Delta t}\,\mathrm{tr}\left(\underline{\varepsilon}^{\mathrm{to}}|_{t} + \Delta\,\underline{\varepsilon}^{\mathrm{to}}\right)\underline{I} + 2\,\mu|_{t+\Delta t}\left(\underline{\varepsilon}^{\mathrm{to}}|_{t} + \Delta\,\underline{\varepsilon}^{\mathrm{to}}\right)\end{aligned}$$

The consistent tangent operator $\frac{\partial \underline{\sigma}|_{t+\Delta t}}{\partial \Delta \underline{\varepsilon}^{\mathrm{to}}}$ and the prediction operator are both given by the elastic stiffness tensor at the end of the time step:

$$\frac{\partial \underline{\sigma}|_{t+\Delta t}}{\partial \Delta \underline{\varepsilon}^{\mathrm{to}}} = \lambda|_{t+\Delta t}\, \underline{\underline{I}} \otimes \underline{\underline{I}} + 2\,\mu|_{t+\Delta t}\, \underline{\underline{\mathbf{I}}}$$

About incremental update of the stress tensor
For reasons mostly related to rate-form approaches to finite strain, Hooke's law is often implemented using an incremental update of the stress tensor:

$$\underline{\sigma}|_{t+\Delta t} = \underline{\sigma}|_{t} + \lambda\,\mathrm{tr}(\Delta \underline{\varepsilon}^{\mathrm{to}})\,\underline{\underline{I}} + 2\,\mu\,\Delta \underline{\varepsilon}^{\mathrm{to}}$$

It should be emphasised that this incremental update is equivalent to the Hooke law only if the elastic properties are constant (throughout simulation).
As the solver calling the behaviour is free to modify the material properties passed to the behaviour, such incremental update discouraged in `MFront` and is never used internally.

The `Default` DSL automatically defines the following variables:

- `eto`, which denotes the strain at the beginning of the time step $\underline{\varepsilon}^{\mathrm{to}}|_{t}$.
- `deto`, which denotes the strain increment over the time step $\Delta\,\underline{\varepsilon}^{\mathrm{to}}$.
- `dt`, which denotes the time increment $\Delta\,t$.
- `sig`, which denotes the stress. This variable is initialized to the value of the stress at the beginning of the time step $\underline{\sigma}|_{t}$ and must be updated to the value of the stress at the end of the time step $\underline{\sigma}|_{t+\Delta t}$.
- in the `@PredictionOperator` code block, `Dt` denotes the prediction operator to be computed.
- in the `@Integrator` and `@TangentOperator` code blocks, `Dt` denotes the tangent operator to be computed.

With those conventions and the tensorial operations provided by the `TFEL/Math` library described in the next paragraph, the implementation of the Hooke's law is straightforward:

```
@DSL DefaultDSL;
@Behaviour Elasticity;
@Author HELFER Thomas;
@Date 06 / 09 / 2023;
@UseQt true;

@MaterialProperty stress yg;
yg.setGlossaryName("YoungModulus");
@MaterialProperty real nu;
nu.setGlossaryName("PoissonRatio");

//! first Lamé's coefficient
@LocalVariable stress lambda;
//! shear modulus
@LocalVariable stress  mu;

@InitLocalVariables {
  lambda = computeLambda(yg, nu);
  mu = computeMu(yg, nu);
}

@PredictionOperator {
  Dt = lambda * Stensor4::IxI() + 2 * mu * Stensor4::Id();
  static_cast<void>(smt); // avoid unused variable warning
}

@Integrator {
  constexpr auto id = Stensor::Id();
  const auto e = eto + deto;
  sig = lambda * trace(e) * id + 2 * mu * e;
}
```

```
@TangentOperator {
  Dt = lambda * Stensor4::IxI() + 2 * mu * Stensor4::Id();
  static_cast<void>(smt);  // avoid unused variable warning
}
```

Most of the keywords used have already been discussed in Section **??**. The main new features are:

1. The two code blocks `@PredictionOperator` and `@TangentOperator` which deal with the computation of the prediction and tangent operators respectively.
2. The use of tensorial objects and tensorial operations provided by the `TFEL/Math` library. Appendix A describes some of the most useful tensorial operations provided by this library.
3. The use of the functions `computeLambda` and `computeMu` provided by the `TFEL/Material` library. These functions calculate the first Lamé's coefficient and the shear modulus from Young's modulus and Poisson's ratio.

6.2.1 About tensorial operations

In `TFEL/Math`, symmetric tensor are stored as vectors. For example, the stress tensor is stored as follows in $3D$:

$$\underline{\sigma} = \left(\sigma_{11} \quad \sigma_{22} \quad \sigma_{33} \quad \sqrt{2}\,\sigma_{12} \quad \sqrt{2}\,\sigma_{13} \quad \sqrt{2}\,\sigma_{23}\right)^T$$

Contrary to Voigt's notations, there is no distinction bewteen stress and strain. The $\sqrt{2}$ factor allows to calculte the double contraction of two symmetric tensors by the scalar product of their vectorial representation.

The `TFEL/Math` library has been designed to optimize tensorial operations for a given space dimension, which is associated in `MFront` to a modelling hypothesis. Symmetric tensors are implemented by the `stensor` template class which has two template parameters: the space dimension and the type hold. Those details, while extremely important for the performances of the generated code, are usually transparent in `MFront` files for the following reasons:

- Under the hood, the behaviour is compiled once per modelling hypothesis supported, as described by the note "Some insights on the code generated by `MFront`."
- `MFront` provides some convenient aliases for the most useful types:
  - `Stensor` and `StrainStensor` denotes a symmetric tensor with no unit.
  - `StressStensor` denotes a symmetric tensor with contains stress values if support of quantities as been enabled through the `@UseQt` keyword (see Section 3.3.8.4).

> **Some insights on the code generated by `MFront`**
> Under the hood, `MFront` generates a `C++` template class for each behaviour. This template class is parametrised by the modelling hypothesis, the floating-point type and a boolean stating if quantities must be used. One instance of this class is instanciated by the compiler for each modelling hypothesis supported by the behaviour.
> The template parameter describing the modelling hypothesis is named `hypothesis`. The space dimension, deduced from the modelling hypothesis, is accessible in a static variable named `N`. The floating point type used is associated the `NumericType` template parameter.

An important consequence is that direct access to the components of a tensor components is generally discouraged in favor of purely tensorial operations.

The `stensor` class provides the `Id` static method which returns the symmetric part of the identity tensor. Using the vectorial notations of the `TFEL/Math` library, this tensor is represented by the following vector in $3D$:

$$\underline{I} = \left(1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0\right)^T$$

Finally, the `TFEL/Math` library provides the `trace` function to compute the trace of a symmetric tensor.

Fourth-order tensors, which transforms linearly a symmetric tensor into a symmetric tensor, are stored in a matrix format consistent with this convention. In particular, the result of applying a fourth order tensor on a symmetric tensor is the product of respective matrix and vector representations. Such fourth order tensor class is implemented in the `TFEL/Math` library by the `st2tost2` class.

Again, `MFront` automatically declares some usefull aliases:

- `Stensor4`, which declares a fourth-order tensor holding values with no unit.
- `StiffnessTensor`, which declares a fourth-order tensor holding stress values.

In the case of a strain based behaviour, the prediction and tangent operators are declared using the `StiffnessTensor` alias.

Non symmetric tensors and other fourth order tensors
Non symmetric tensor are implemented by the `tensor` class.
In addition to the `st2tost2` class, the `TFEL/Math` library also provides three other types of fourth-order tensor:

- `t2tost2`, which transform a non symmetric second order tensor to a symmetric one,
- `st2tot2`, which transform a symmetric second order tensor to a non symmetric one,
- `t2tot2`, which transform a non symmetric second order tensor to a non symmetric tensor.

The `st2tost2` class provides the following static methods:

- `Id`, which returns the identity fourth-order tensor.
- `IxI`, which returns the tensor `\tenseur{I}\,\otimes\,\tenseur{I}`.

6.2.2 Firt test using `MTest`

For small strain behaviours, `MTest` attempts to find the evolution of the strain tensor under a set of constraints. These constraints can impose:

- the value of a component of is strain, using the `@ImposedStrain` keyword.
- the value of a component of the stress, using the `@ImposedStress` keyword.
- a general non-linear constraint mixing arbitrary strain and stress components using the `@NonLinearConstraint` keyword.

For example, the value of the first component of the stress imposed as follows:

```
@ImposedStress<function> 'SXX' '150e6 * t';
```

where `t` denotes the current time. `t` is a built-in variable in `MTest`.

The reader is referred to the online documentation of `MTest` for a complete description of those keywords:

```
$ mtest --help-keyword=@ImposedStrain
```

Table 6.1: Components of strain tensor

Modelling Hypothesis	Strain components
AxisymmetricalGeneralisedPlaneStrain	ERR EZZ ETT
Axisymmetrical	ERR EZZ ETT ERZ
PlaneStress	EXX EYY EZZ EXY
PlaneStrain	EXX EYY EXY
GeneralisedPlaneStrain	EXX EYY EZZ EXY
Tridimensional	EXX EYY EZZ EXY EXZ EYZ

Table 6.2: Components of stress tensor

Modelling Hypothesis	Stress components
AxisymmetricalGeneralisedPlaneStrain	SRR SZZ STT
Axisymmetrical	SRR SZZ STT SRZ
PlaneStress	SXX SYY SXY
PlaneStrain	SXX SYY SZZ SXY
GeneralisedPlaneStrain	SXX SYY SZZ SXY
Tridimensional	SXX SYY SZZ SXY SXZ SYZ

The names of the components of the strain tensor and those of the stress tensor depend on the selected modelling hypothesis (see the `@ModellingHypothesis` keyword for details). They are given in Tables 6.1 and 6.2.

6.2.2.1 Uniaxial tensile test

The following script describes a simple uniaxial tensile test and checks that the results are the expected ones:

```
@Behaviour 'src/libBehaviour.so' 'Elasticity';
@MaterialProperty<constant> 'YoungModulus' 150e9;
@MaterialProperty<constant> 'PoissonRatio' 0.3;
@ExternalStateVariable 'Temperature' 293.15;
@ImposedStrain 'EXX' {0 : 0, 1 : 1e-2};
@Times{0, 1};
@Test<function> {
  'SXX' : 'YoungModulus * EXX', 'SYY' : '0','SZZ' : '0',
  'SXY' : '0', 'SXZ' : '0','SYZ' : '0'
} 1e-2;
@Test<function> {
  'EYY' : '-PoissonRatio * EXX','EZZ' : '-PoissonRatio * EXX',
  'EXY' : '0', 'EXZ' : '0','EYZ' : '0'
} 1e-14;
```

6.2.2.2 Shear test

The following script describes a simple shear test and checks that the results are the expected ones:

```
@Behaviour 'src/libBehaviour.so' 'Elasticity';
@MaterialProperty<constant> 'YoungModulus' 150e9;
@MaterialProperty<constant> 'PoissonRatio' 0.3;
@ExternalStateVariable 'Temperature' 293.15;
@ImposedStrain 'EYZ' {0 : 0, 1 : 'sqrt(2)*1e-2'};
@Times{0, 1};
@Real 'ShearModulus' 'YoungModulus / (2 * (1 + PoissonRatio))';
@Test<function> {
  'SYZ' : '2 * ShearModulus * EYZ',
  'SXX' : '0', 'SYY' : '0', 'SZZ' : '0', 'SXY' : '0', 'SXZ' : '0'
} 1e-2;
@Test<function> {
  'EXX' : '0', 'EYY' : '0', 'EZZ' : '0', 'EXY' : '0', 'EXZ' : '0'
} 1e-14;
```

Again, caution must be taken regarding the conventions used by `MFront` and `MTest` regarding the storage of the shear components of tensors: the component `EYZ` contains the value of $\sqrt{2}\,\varepsilon^{\mathrm{to}}_{yz}$.

6.2.3 Finite strain generalisation

6.2.3.1 Green-Lagrange strain

6.2.3.2 Hencky strain

Chapter 7

Domain specific languages dedicated to (visco-) plasticity of isotropic materials

In this section the focus is made on a particular family of behavior laws, widely used in mechanics, and which represent the majority of the behaviors of the `pleiades` platform. These laws apply to *isotropic* materials and describe a *plastic* or *viscoplastic* behavior, with or without *collapsing*, whose residual deformations are *isochore*. These particularities allow the optimization of numerical integration techniques.

After a presentation of the integration techniques adapted to this family of laws, the syntax of the four `MFront` analyzers dedicated to them is detailed.

7.1 Generalities and implicit resolution

After a description of the particularities of the mechanical behavior laws treated in this section, the integration techniques applied to them in `MFront` are exposed. These are based on an implicit resolution: a *theta*-method similar to those described in section 10. They also allow the calculation of the coherent tangent matrix.

7.1.1 Expression of behavior laws

The behavior laws treated in this section are made up of an elastic part, and several plastic or viscoplastic flow mechanisms.

7.1.1.1 The additive split of deformation

The additive split of deformation is such as the total deformation $\underline{\epsilon}^{to}$ is the sum of an elastic strain $\underline{\epsilon}^{el}$ and an inelastic strain $\underline{\epsilon}^{an}$:

$$\underline{\epsilon}^{to} = \underline{\epsilon}^{el} + \underline{\epsilon}^{an}. \tag{7.1}$$

The latter is decomposed into several deformations related to the different flow mechanisms, indicated by i, and assumed to be independent:

$$\underline{\epsilon}^{an} = \sum_i \underline{\epsilon}_i^{an}. \tag{7.2}$$

7.1.1.2 Isotropy

The materials described here are assumed to be *isotropic*. This assumption will be considered to formulate all the mechanisms: elasticity, plastic or viscoplastic flows.

7.1.1.3 Tensors

To simplify the forthcoming expressions, it is convenient to introduce tensor notations, detailed in appendix A:

- the tensor product of two tensors a, b: $a \otimes b$;
- the inner product of two tensors a, b: $a : b$;
- identity tensor of order two: $\underline{I}$;
- identity tensor of order four: $\underline{\underline{\mathbf{I}}}$.

7.1.1.4 Elastic behaviour

The stresses $\underline{\sigma}$ are deduced from elastic strain $\underline{\epsilon}^{el}$ by means of the HOOKE law. For an isotropic material, this relationship can be be written:

$$\underline{\sigma} = \lambda \operatorname{tr}(\underline{\epsilon}^{el})\underline{\underline{\mathbf{I}}} + 2\,\mu\,\underline{\epsilon}^{el} \tag{7.3}$$

where $\operatorname{tr}(\underline{\epsilon}^{el})$ is the trace of the tensor $\underline{\epsilon}^{el}$, λ and μ are the LAMÉ coefficients of the material deduced from the YOUNG modulus and the POISSON ratio. In tensor form this law is written in a more compact way:

$$\underline{\sigma} = \underline{\underline{\mathbf{D}}} : \underline{\epsilon}^{el}, \quad \text{with} \quad \underline{\underline{\mathbf{D}}} = \lambda\,\underline{I} \otimes \underline{I} + 2\,\mu\,\underline{\underline{\mathbf{I}}}.$$

The law is then represented by the elastic tensor $\underline{\underline{\mathbf{D}}}$.

7.1.1.5 Flows

The i flows, plastic or viscoplastic, are the mechanisms that create the inelastic deformations $\underline{\epsilon}_i^{an}$ from ((7.2)). For the laws described here, the flow direction is assumed to be proportional to the stress deviator tensor:

$$\dot{\underline{\epsilon}}_i^{an} \propto \underline{s}, \tag{7.4}$$

defined by:

$$\underline{s} = \underline{\sigma} - \frac{1}{3}\operatorname{tr}(\underline{\sigma})\,\underline{I} = \underline{\underline{\mathbf{K}}} : \underline{\sigma}, \quad \text{with:} \quad \underline{\underline{\mathbf{K}}} = \underline{\underline{\mathbf{I}}} - \frac{1}{3}\underline{I} \otimes \underline{I}. \tag{7.5}$$

With this assumption, the modeled flows satisfy:

$$\operatorname{tr}(\underline{\epsilon}_i^{an}) = 0,$$

i.e. do not induce a change in volume: they are said to be *isochoric*.

7.1.1.6 von-Mises stresses

The flows described here are assumed to depend on the stress tensor $\underline{\sigma}$ through the norm of its deviator ((7.5)), called von-Mises stresses:

$$\sigma_{eq} = \sqrt{\frac{3}{2}\,\underline{s} : \underline{s}}. \tag{7.6}$$

The von-Mises stress is one of the isotropic stress invariants. The assumption thus ensures the isotropic character of the flow law.

7.1.1.7 Hardening

For each flow i, the flow law can also depend on on its strain *hardening*, or *cumulative inelastic strain* p_i, defined by the differential equation:

$$\dot{p}_i = \sqrt{\frac{2}{3}\,\dot{\underline{\epsilon}}^{an}{}_i : \dot{\underline{\epsilon}}^{an}{}_i}. \tag{7.7}$$

The factor $\frac{3}{2}$, introduced by convention in the von-Mises stress ((7.6)), is compensated here in the definition of $dotp_i$. In general, at the beginning of the calculation, at time $t = t_0$, the material is assumed to be unhardened:

$$p_i(t_0) = 0.$$

With the assumption on the flow direction ((7.4)), the relation ((7.7)) reverses to:

$$\dot{\underline{\epsilon}}_i^{an} = \dot{p}_i\,\underline{n}, \quad \text{with} \quad \underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{\text{eq}}}, \tag{7.8}$$

where the tensor $\underline{n}$ is called *normal* to the flow. It is a deviatoric tensor of constant norm, which satisfies:

$$\underline{\underline{\mathbf{K}}} : \underline{n} = \underline{n}, \quad \text{et} \quad \underline{n} : \underline{n} = \frac{3}{2}. \tag{7.9}$$

7.1.1.8 Supported flows

It is now possible to formulate the three types of mechanisms that require a specific integration in `MFront`:

- viscoplastic flows of the following form, from which Norton's law [19] is derived:

$$\underline{\dot{\epsilon}}_i^{an} = f_i^{\text{an}}(\sigma_{eq})\,\underline{n}, \quad \text{i.e.} \quad \dot{p}_i = f_i^{\text{an}}(\sigma_{eq}); \tag{7.10}$$

- viscoplastic flows with work hardening of the following form, from which the Lemaitre's law [19] arises:

$$\underline{\dot{\epsilon}}^{an}{}_i = f_i^{\text{an}}(\sigma_{eq}, p_i)\,\underline{n}, \quad \text{i.e.} \quad \dot{p}_i = f_i^{\text{an}}(\sigma_{eq}, p_i). \tag{7.11}$$

- and plastic flows that satisfy a function such as:

$$f_i^{\text{an}}(\sigma_{eq}, p_i) \leq 0 \qquad \dot{p}_i \geq 0 \qquad f_i^{\text{an}}(\sigma_{eq}, p_i)\,\dot{p}_i = 0. \tag{7.12}$$

  It is classical that the function f_i^{an} of the plastic flow has the same dimension as a stress. `MFront` assumes that the user has respected this convention, and thus divides the function f_i^{an} by the Young's modulus of the material, so that the set of inequations ((7.12)) have the dimension of a deformation.

The function f_i^{an}, which defines the flow, can possibly involve external variables evolving independently of the mechanics (fission density, fast neutron flux, grain size flux, grain size, *etc.*).

7.1.1.9 Remarks

1. Viscoplastic flows without strain hardening ((7.10)) are a special case of flows with strain hardening ((7.11)). To optimize the computation time, it is advantageous to keep their distinction during the numerical implementation. For theoretical developments, it is preferable to group these flows under their common form ((7.11)).
2. With the flow direction assumption ((7.8)), both viscoplastic flows ((7.11)) and plastic flows ((7.12)) depend on the strain hardening p_i instead of depending on the deformation $\underline{\epsilon}_i^{an}$. This particularity explains the possible optimization when integrating the ((7.10),(7.11),(7.12)) laws: it is enough to integrate equations with a scalar unknown (p_i) rather than a tensor unknown ($tepsilonan_i$).
3. The evolution of the viscoplastic deformation ((7.11)) is given by a differential equation, while the formulation of the plastic laws is based on an equation ((7.12)): respect of the load surface.

7.1.1.10 A system of equations

An integration of the behavioral law proposed by `MFront` consists in loading a material point with the total deformation $\underline{\epsilon}^{to}$ and to compute the stress evolution $\underline{\sigma}$.For this purpose, the law relies on a set of internal variables, kept in memory, which evolve simultaneously. The internal variables chosen are the following:

- the elastic deformation $\underline{\epsilon}^{el}$;
- the hardening p_i associated with each mechanism.

The equations which allow to calculate this evolution are gathered here. They are:

1. the partition of the deformations ((7.1)):

$$\underline{\epsilon}^{to} = \underline{\epsilon}^{el} + \underline{\epsilon}^{an}, \quad \text{with} \quad \underline{\epsilon}^{an} = \sum_i \underline{\epsilon}^{an}{}_i;$$

2. the elasticity law ((7.3))

$$\underline{\sigma} = \underline{\underline{\mathbf{D}}} : \underline{\epsilon}^{el};$$

3. the stress tensor decomposition $\underline{\sigma}$ into deviatoric part and von-Mises stress ((7.5)) and von-Mises stress ((7.6)):

$$\underline{s} = \underline{\underline{\mathbf{K}}} : \underline{\sigma}, \quad \text{and} \quad \sigma_{eq} = \sqrt{\frac{3}{2}\,\underline{s} : \underline{s}};$$

4. the flow direction, oriented along the flow normal ((7.8)):

$$\underline{\dot{\epsilon}}^{an}{}_{i} = \dot{p}_i\,\underline{n}, \quad \text{with} \quad \underline{n} = \frac{3}{2}\frac{\underline{s}}{\sigma_{eq}}$$

5. viscoplastic flows ((7.11)) or plastic flows ((7.12)):

$$\dot{p}_i = f_i^{\text{an}}(\sigma_{eq}, p_i),$$
$$f_i^{\text{an}}(\sigma_{eq}, p_i) \leq 0, \quad \dot{p}_i \geq 0, \quad f_i^{\text{an}}(\sigma_{eq}, p_i)\,\dot{p}_i = 0.$$

7.1.2 Numerical integration method

In `MFront`, the ((7.1),) behavior law is integrated by an implicit method, a θ-method, similar to the one presented in the **??** paragraph, except that the system of equations here can be reduced to a single scalar equation [20].

7.1.2.1 Time increment

According to the foregoing, the internal variables are the elastic strain $\underline{\epsilon}^{el}$ and the strain hardening p_i associated to each mechanism. They are known at time t, at the beginning of the time step, and for a total deformation $\underline{\epsilon}^{to}|_t$. The integration consists in computing their new values induced by an increment of total deformation $\Delta\underline{\epsilon}^{to}$ at the end of a time step Δt. Their respective increments will be noted $\Delta\underline{\epsilon}^{el}$ and Δp_i.

The notion of θ-method, consists in considering also the values of some variables at the intermediate time $t + \theta\,\Delta t$, where the parameter θ, determined in advance, is between 0 and 1. A first approximation consists in supposing the internal variables to be linear on the time step Δt, and thus to assume:

$$\begin{aligned} \underline{\epsilon}^{el}\big|_{t+\theta\,\Delta t} &\approx \underline{\epsilon}^{el}\big|_t + \theta\,\Delta\underline{\epsilon}^{el}, \\ p_i\big|_{t+\theta\,\Delta t} &\approx p_i\big|_t + \theta\,\Delta p_i. \end{aligned} \tag{7.13}$$

7.1.2.2 Stresses

According to the elastic law ((7.3)) and its own definition ((7.5)), the stress deviator is:

$$\underline{s} = 2\,\mu\,\underline{\underline{\mathbf{K}}} : \underline{\epsilon}^{el},$$

and according to the approximations ((7.13)), its value at intermediate time is:

$$\underline{s}\big|_{t+\theta\,\Delta t} = 2\,\mu\,\underline{\underline{\mathbf{K}}} : \left(\underline{\epsilon}^{el}\big|_t + \theta\,\Delta\underline{\epsilon}^{el}\right), \tag{7.14}$$

and the Von-Mises stress ((7.6)) and flow normal ((7.8)) for that same date are deduced:

$$\sigma_{eq}\big|_{t+\theta\,\Delta t} = \sqrt{\frac{3}{2}\,\underline{s}\big|_{t+\theta\,\Delta t} : \underline{s}\big|_{t+\theta\,\Delta t}}, \quad \text{et} \quad \underline{n}\big|_{t+\theta\,\Delta t} = \frac{3}{2}\frac{\underline{s}\big|_{t+\theta\,\Delta t}}{\sigma_{eq}\big|_{t+\theta\,\Delta t}}. \tag{7.15}$$

7.1.2.3 Partition of the deformations

The deformation partition ((7.1)), and the flow direction ((7.8)) integrate into:

$$\Delta\underline{\epsilon}^{to} = \Delta\underline{\epsilon}^{el} + \sum_i \int_t^{t+\Delta t} \dot{p}_i\,\underline{n}\,\mathrm{d}t \approx \Delta\underline{\epsilon}^{el} + \underline{n}\big|_{t+\theta\,\Delta t} \sum_i \Delta p_i. \tag{7.16}$$

This approximation is based on the value of the $\underline{n}$ normal at intermediate time. With the previous relations ((7.14)) and ((7.15)), and the deviator type ((7.9)) of $\underline{n}$, one can then write:

$$\underline{n}\big|_{t+\theta\,\Delta t} = \frac{3\,\mu}{\sigma_{eq}\big|_{t+\theta\,\Delta t}}\left[\underline{\underline{\mathbf{K}}} : \left(\underline{\epsilon}^{el}\big|_t + \theta\,\Delta\underline{\epsilon}^{to}\right) - \underline{n}\big|_{t+\theta\,\Delta t}\,\theta\sum_i \Delta p_i\right],$$

that is:

$$\left(\sigma_{eq}\big|_{t+\theta\,\Delta\,t} + 3\mu\theta\sum_i \Delta p_i\right)\underline{n}\big|_{t+\theta\,\Delta\,t} = 3\mu\underline{B}, \quad \text{with} \quad \underline{B} = \underline{\underline{\mathbf{K}}}\colon \left(\underline{\epsilon}^{el}\big|_t + \theta\,\Delta\underline{\epsilon}^{to}\right). \tag{7.17}$$

The norm of this expression ((7.17)), given the norm ((7.9)) of $\underline{n}|_{t+\theta\,\Delta\,t}$, gives finally for the equivalent stress:

$$\sigma_{eq}\big|_{t+\theta\,\Delta\,t} = \mu\sqrt{6\,\underline{B}\colon\underline{B}} - 3\,\mu\,\theta\sum_i \Delta p_i. \tag{7.18}$$

This value, injected into the relationship ((7.17)), gives the normal:

$$\underline{n}\big|_{t+\theta\,\Delta\,t} = \frac{3}{\sqrt{6\,\underline{B}\colon\underline{B}}}\underline{B}. \tag{7.19}$$

The equation ((7.18)) will allow to form, from the flow equations, a non linear system of equations of equations whose unknowns are the increments Δp_i. We must now construct this system by integrating viscoplastic flows, then plastic flows.

7.1.2.4 Viscoplastic flow

The integral over the time step Δt of the viscoplastic flow ((7.11)) is written:

$$\Delta p_i = \int_t^{t+\Delta t} f_i^{\text{an}}(\sigma_{eq}, p_i)\,\mathrm{d}t \approx \Delta t\, f_i^{\text{an}}\Big(\sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big),$$

where the integral is approximated by the value of f_i^{an} at intermediate time. With this approximation, checking the flow consists in cancelling a three-parameter function:

$$f_{p_i}\Big(\Delta p_i,\, \sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big) = \Delta p_i - \Delta t\, f_i^{\text{an}}\Big(\sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big). \tag{7.20}$$

7.1.2.5 Plastic flow

The plastic flow consists in nullify a function of the same form, depending on the same three parameters:

$$f_{p_i}\Big(\Delta p_i,\, \sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big) = \begin{cases} f_i^{\text{an}}\Big(\sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big) & \text{in case of plastic load,} \\ \Delta p_i & \text{otherwise.} \end{cases} \tag{7.21}$$

The plastic loading condition, which increases the strain hardening p_i, must meet the plastic flow conditions ((7.12)). In practice, the increments Δp_i, unknowns of a nonlinear system, are computed iteratively. Each new iteration proposes new values of these increments. For these new values, the plastic load condition is plastic load condition is activated if one of the following conditions is verified:

1. $f_i^{\text{an}}\Big(\sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big) > \varepsilon$ et $\Delta p_i \geq 0$,
2. $\Delta p_i > \varepsilon$,

where ε is a numerical parameter, which stabilizes the iterations.

7.1.2.6 Non-linear system

The flow equations ((7.20),(7.21)):

$$f_{p_i}\Big(\Delta p_i,\, \sigma_{eq}\big|_{t+\theta\,\Delta\,t},\, p_i\big|_{t+\theta\,\Delta\,t}\Big) = 0, \quad \forall\ p_i,$$

modified with the equations ((7.13)) and ((7.18)), form a nonlinear system of equations, whose unknowns are the increments Δp_i:

$$f_{p_i}\left(\Delta p_i, \mu\sqrt{6\,\underline{B}\colon\underline{B}} - 3\,\mu\,\theta\sum_j \Delta p_j,\, p_i\big|_t + \theta\,\Delta p_i\right) = 0, \quad \forall\ p_i. \tag{7.22}$$

To solve it, a Newton-Raphson method is used. This method, discussed in detail in paragraph **??**,requires the partial derivatives of the functions f_{p_i} with respect to the increments Δp_i. These are obtained by a compound derivation f_{p_i}:

$$\frac{\partial f_{p_i}}{\partial \Delta p_j} = \frac{\partial f_{p_i}}{\partial \Delta p_i}\frac{\partial \Delta p_i}{\partial \Delta p_j} + \frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\frac{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}{\partial \Delta p_j} + \frac{\partial f_{p_i}}{\partial p_i|_{t+\theta\,\Delta t}}\frac{\partial p_i|_{t+\theta\,\Delta t}}{\partial \Delta p_j},$$

with, according to the equations ((7.13)) and ((7.18)):

$$\frac{\partial \Delta p_i}{\partial \Delta p_j} = \delta_{ij}, \quad \frac{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}{\partial \Delta p_j} = -3\,\mu\,\theta, \quad \text{and} \quad \frac{\partial p_i|_{t+\theta\,\Delta t}}{\partial \Delta p_j} = \theta\,\delta_{ij}, \quad \text{with} \quad \delta_{ij} = \begin{cases} 1 & \text{si } i = j, \\ 0 & \text{sinon.} \end{cases}$$

Applied to flows, these differential relations give:

- for viscoplastic flows ((7.20)):

$$\frac{\partial f_{p_i}}{\partial \Delta p_j} = \begin{cases} 1 - 3\,\mu\,\theta\,\Delta t\left[\dfrac{\partial f_i^{\text{an}}}{\partial \sigma_{eq}}\Big(\sigma_{eq}|_{t+\theta\,\Delta t}, p_i|_{t+\theta\,\Delta t}\Big)\right] - \theta\,\Delta t\left[\dfrac{\partial f_i^{\text{an}}}{\partial p_i}\Big(\sigma_{eq}|_{t+\theta\,\Delta t}, p_i|_{t+\theta\,\Delta t}\Big)\right] & \text{si } i = j, \\ -3\,\mu\,\theta\,\Delta t\left[\dfrac{\partial f_i^{\text{an}}}{\partial \sigma_{eq}}\Big(\sigma_{eq}|_{t+\theta\,\Delta t}, p_i|_{t+\theta\,\Delta t}\Big)\right] & \text{si } i \neq j; \end{cases}$$

- for plastic flows ((7.21)), in case of plastic release:

$$\frac{\partial f_{p_i}}{\partial \Delta p_j} = \begin{cases} \theta\left(3\,\mu\left[\dfrac{\partial f_i^{\text{an}}}{\partial \sigma_{eq}}\Big(\sigma_{eq}|_{t+\theta\,\Delta t}, p_i|_{t+\theta\,\Delta t}\Big)\right] + \left[\dfrac{\partial f_i^{\text{an}}}{\partial p_i}\Big(\sigma_{eq}|_{t+\theta\,\Delta t}, p_i|_{t+\theta\,\Delta t}\Big)\right]\right) & \text{si } i = j, \\ -3\,\mu\,\theta\,\dfrac{\partial f_i^{\text{an}}}{\partial \sigma_{eq}}\Big(\sigma_{eq}|_{t+\theta\,\Delta t}, p_i|_{t+\theta\,\Delta t}\Big) & \text{si } i \neq j. \end{cases}$$

  and in case of no plastic load:

$$\frac{\partial f_{p_i}}{\partial \Delta p_j} = \begin{cases} 1 \text{ si } i = j, \\ 0 \text{ si } i \neq j. \end{cases}$$

7.1.2.7 Stop criterion

The Newton algorithm stops when the difference between two estimates of the increments of the cumulative viscoplastic deformations viscoplastic deformations is lower than a certain criterion ε:

$$\frac{1}{N}\sum_{i=1}^{N} |\Delta_n \Delta p_i| < \varepsilon \tag{7.23}$$

where $\Delta_n \Delta p_i$ denotes the difference between the estimate of the cumulative viscoplastic strain increment $\Delta\, p_i$ at step $n+1$ and the estimate at step n.

The algorithm fails if the number of iterations exceeds a maximum bound.

7.1.2.8 Steps

Finally, the different steps of the integration of the mechanical behavior law mechanical behavior by the θ-méthode is as follows:

1. tensor calculation $\underline{B}$ ((7.17)):

$$\underline{B} = \underline{\underline{\mathbf{K}}} : \left(\underline{\epsilon}^{el}\big|_t + \theta\,\Delta\underline{\epsilon}^{to}\right) \;;$$

2. compute the increments Δp_i, by solving the nonlinear system ((7.22)):

$$f_{p_i}\left(\Delta p_i, \mu\sqrt{6\,\underline{B}\colon\underline{B}} - 3\,\mu\,\theta\sum_j \Delta p_j,\, p_i|_t + \theta\,\Delta p_i\right) = 0, \quad \forall\ p_i,$$

using the Newton-Raphson method, and calculation of the new new hardening values:

$$p_i|_{t+\Delta t} = p_i|_t + \Delta p_i\ ;$$

3. calculation of the elastic deformation increment, using the strain partition ((7.16)), of the normal ((7.19)):

$$\Delta\underline{\epsilon}^{el} = \Delta\underline{\epsilon}^{to} - \underline{n}|_{t+\theta\,\Delta t}\sum_i \Delta p_i, \quad \text{with} \quad \underline{n}|_{t+\theta\,\Delta t} = \frac{3}{\sqrt{6\,\underline{B}\colon\underline{B}}}\underline{B},$$

and the new value of elastic deformation:

$$\underline{\epsilon}^{el}\big|_{t+\Delta t} = \underline{\epsilon}^{el}\big|_t + \Delta\underline{\epsilon}^{el}\ ;$$

4. calculation of the stress at the end of the time step, by the elasticity law ((7.3)):

$$\underline{\sigma}|_{t+\Delta t} = \underline{\underline{\mathbf{D}}}\colon\underline{\epsilon}^{el}\big|_{t+\Delta t}.$$

7.1.3 Consistent tangent matrix

The implicit solution method allows the calculation of the coherent tangent matrix. This coherent tangent matrix allows a quadratic convergence of the structure calculation, which can significantly speed up the calculations.

7.1.3.1 Definition

The numerical integration described in the previous paragraph, allows to compute the stress evolution $\Delta\underline{\sigma}$ induced by a deformation increment $\Delta\underline{\epsilon}^{to}$ on a time step Δt. The coherent tangent matrix is defined as the tensor:

$$\underline{\underline{\mathbf{L}}}^{tc} = \frac{\partial\Delta\underline{\sigma}}{\partial\Delta\underline{\epsilon}^{to}}. \tag{7.24}$$

The elasticity law ((7.3)) and the strain partition ((7.16)) allow to express the increment of stress increment:

$$\Delta\underline{\sigma} = \underline{\underline{\mathbf{D}}}\colon\Delta\underline{\epsilon}^{el} = \underline{\underline{\mathbf{D}}}\colon\left(\Delta\underline{\epsilon}^{to} - \underline{n}|_{t+\theta\,\Delta t}\sum_i \Delta p_i\right)$$

and deduct by derivation ((7.24)) the coherent tangent matrix:

$$\underline{\underline{\mathbf{L}}}^{tc} = \underline{\underline{\mathbf{D}}} - \underline{\underline{\mathbf{D}}}\colon\left(\underline{n}|_{t+\theta\,\Delta t}\otimes\sum_i\frac{\partial\Delta p_i}{\partial\Delta\underline{\epsilon}^{to}} + \sum_i \Delta p_i\frac{\partial\,\underline{n}|_{t+\theta\,\Delta t}}{\partial\Delta\underline{\epsilon}^{to}}\right). \tag{7.25}$$

It is now required to compute the derivatives of the normal $\underline{n}|_{t+\theta\,\Delta t}$ and the increments Δp_i for each flow.

7.1.3.2 Normal

The derivative of the tensor $\underline{B}$ ((7.17)) and of the normal ((7.19)) are respectively:

$$\frac{\partial\underline{B}}{\partial\Delta\underline{\epsilon}^{to}} = \theta\,\underline{\underline{\mathbf{K}}}, \tag{7.26}$$

and:

$$\begin{aligned}\frac{\partial\,\underline{n}|_{t+\theta\,\Delta t}}{\partial\Delta\underline{\epsilon}^{to}} &= \frac{3}{\sqrt{6\,\underline{B}\colon\underline{B}}}\left(\frac{\partial\underline{B}}{\partial\Delta\underline{\epsilon}^{to}} - \frac{2}{3}\frac{\partial\underline{B}}{\partial\Delta\underline{\epsilon}^{to}}\colon\underline{n}|_{t+\theta\,\Delta t}\otimes\underline{n}|_{t+\theta\,\Delta t}\right)\\ &= \frac{3\,\theta}{\sqrt{6\,\underline{B}\colon\underline{B}}}\left(\underline{\underline{\mathbf{K}}} - \frac{2}{3}\underline{n}|_{t+\theta\,\Delta t}\otimes\underline{n}|_{t+\theta\,\Delta t}\right).\end{aligned} \tag{7.27}$$

This calculation uses the property ((7.9)) of the tensor $\underline{n}|_{t+\theta\,\Delta t}$.

7.1.3.3 Increments

The strain hardening increments Δp_i are derived from the resolution of the of the nonlinear system, formed by the flow equations flow equations ((7.20),(7.21)):

$$f_{p_i}\left(\Delta p_i,\, \sigma_{eq}|_{t+\theta\,\Delta t},\, p_i|_{t+\theta\,\Delta t}\right) = 0, \quad \forall\ p_i.$$

The derivation of these equations leads to:

$$\frac{\partial f_{p_i}}{\partial \Delta p_i}\frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} + \frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\frac{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}{\partial \underline{\underline{\epsilon}}^{to}} + \frac{\partial f_{p_i}}{\partial p_i|_{t+\theta\,\Delta t}}\frac{\partial p_i|_{t+\theta\,\Delta t}}{\partial \underline{\underline{\epsilon}}^{to}} = 0, \quad \forall\ p_i. \tag{7.28}$$

The derivation of the equations ((7.13)) and ((7.18)), using the relations ((7.9)) and ((7.26)):

$$\frac{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}{\partial \underline{\underline{\epsilon}}^{to}} = 2\,\mu\,\theta\ \underline{\underline{n}}|_{t+\theta\,\Delta t} - 3\,\mu\,\theta\sum_j \frac{\partial \Delta p_j}{\partial \underline{\underline{\epsilon}}^{to}},$$
$$\frac{\partial p_i|_{t+\theta\,\Delta t}}{\partial \underline{\underline{\epsilon}}^{to}} = \theta\,\frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}},$$

allows to modify the equations ((7.28)):

$$\left(\frac{\partial f_{p_i}}{\partial \Delta p_i} + \theta\,\frac{\partial f_{p_i}}{\partial p_i|_{t+\theta\,\Delta t}}\right)\frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} - 3\,\mu\,\theta\,\frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\sum_j \frac{\partial \Delta p_j}{\partial \underline{\underline{\epsilon}}^{to}} = -\frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\,2\,\mu\,\theta\ \underline{\underline{n}}|_{t+\theta\,\Delta t}, \quad \forall\ p_i. \tag{7.29}$$

It is a set of linear equations, whose unknowns are the derivative tensors $\dfrac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}}$.

7.1.3.4 Scalar solutions

The contracted product of the equations ((7.29)) by $\underline{\underline{n}}|_{t+\theta\,\Delta t}$ gives a system of equations:

$$\left(\frac{\partial f_{p_i}}{\partial \Delta p_i} + \theta\,\frac{\partial f_{p_i}}{\partial p_i|_{t+\theta\,\Delta t}}\right) d_i - 3\,\mu\,\theta\,\frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\sum_j d_j = -\frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\,3\,\mu\,\theta, \quad \forall\ p_i. \tag{7.30}$$

with for unknowns, the contracted products:

$$d_i = \frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} : \underline{\underline{n}}|_{t+\theta\,\Delta t}.$$

The contracted product of the equations ((7.29)) by any tensor $\underline{\underline{x}}$ orthogonal to $\underline{\underline{n}}|_{t+\theta\,\Delta t}$ gives a system of equations:

$$\left(\frac{\partial f_{p_i}}{\partial \Delta p_i} + \theta\,\frac{\partial f_{p_i}}{\partial p_i|_{t+\theta\,\Delta t}}\right)\frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} : \underline{\underline{x}} - 3\,\mu\,\theta\,\frac{\partial f_{p_i}}{\partial \sigma_{eq}|_{t+\theta\,\Delta t}}\sum_j \frac{\partial \Delta p_j}{\partial \underline{\underline{\epsilon}}^{to}} : \underline{\underline{x}} = 0, \quad \forall\ p_i,$$

with a null second member. Its unknowns are therefore zero:

$$\frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} : \underline{\underline{x}} = 0, \quad \forall\ p_i,$$

which shows that the tensors $\dfrac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}}$ are all collinear to the tensor $\underline{\underline{n}}|_{t+\theta\,\Delta t}$. They are therefore written:

$$\frac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} = \frac{\dfrac{\partial \Delta p_i}{\partial \underline{\underline{\epsilon}}^{to}} : \underline{\underline{n}}|_{t+\theta\,\Delta t}}{\underline{\underline{n}}|_{t+\theta\,\Delta t} : \underline{\underline{n}}|_{t+\theta\,\Delta t}}\ \underline{\underline{n}}|_{t+\theta\,\Delta t} = \frac{2\,d_i}{3}\ \underline{\underline{n}}|_{t+\theta\,\Delta t}, \tag{7.31}$$

where the components d_i are the solutions of the system ((7.30)).

7.1.3.5 Coherent tangent matrix

Finally, the coherent tangent matrix can be reconstructed from from the expressions ((7.25)), ((7.27)) and ((7.31)), and noting that:

$$\underline{\underline{\mathbf{D}}} : \underline{\underline{\mathbf{K}}} = 2\,\mu\,\underline{\underline{\mathbf{K}}}, \qquad \underline{\underline{\mathbf{D}}} : \underline{n}|_{t+\theta\,\Delta\,t} = 2\,\mu\,\underline{n}|_{t+\theta\,\Delta\,t}.$$

It is equal to:

$$\underline{\underline{\mathbf{L}}}^{tc} = \underline{\underline{\mathbf{D}}} - 2\,\mu\left(\frac{3\,\theta\,\sum_i \Delta p_i}{\sqrt{6\,\underline{B} : \underline{B}}}\,\underline{\underline{\mathbf{K}}} + \left(\frac{2\,\sum_i d_i}{3} - \frac{2\,\theta\,\sum_i \Delta p_i}{\sqrt{6\,\underline{B} : \underline{B}}}\right)\,\underline{n}|_{t+\theta\,\Delta\,t} \otimes \underline{n}|_{t+\theta\,\Delta\,t}\right), \tag{7.32}$$

with the components d_i solutions of the system ((7.30)).

7.1.4 Expression of the coherent tangent matrix for a single mechanism

In the case where the system is limited to a single mechanism, the expression of the coherent tangent matrix is simplified.

7.1.4.1 Viscoplastic flow

In the case of a single viscoplastic flow ((7.20)), of increment Δp, the system ((7.30)) reduces to an equation of unknown d:

$$\left(1 - \Delta t\,\theta\,\frac{\partial f^{\text{an}}}{\partial p} + 3\,\mu\,\Delta t\,\theta\,\frac{\partial f^{\text{an}}}{\partial \sigma_{eq}}\right)\,d = \frac{\partial f^{\text{an}}}{\partial \sigma_{eq}}\,3\,\mu\,\Delta t\,\theta.$$

The expression ((7.32)) of the coherent tangent matrix simplifies then into:

$$\underline{\underline{\mathbf{L}}}^{tc} = \underline{\underline{\mathbf{D}}} - 2\,\mu\,\theta\left(\frac{3\,\Delta p}{\sqrt{6\,\underline{B} : \underline{B}}}\,\underline{\underline{\mathbf{K}}} + \left(\frac{2\,\mu\,\Delta t\,\dfrac{\partial f^{\text{an}}}{\partial \sigma_{eq}}}{1 + \theta\,\Delta t\left(3\,\mu\,\dfrac{\partial f^{\text{an}}}{\partial \sigma_{eq}} - \dfrac{\partial f^{\text{an}}}{\partial p}\right)} - \frac{2\,\Delta p}{\sqrt{6\,\underline{B} : \underline{B}}}\right)\,\underline{n}|_{t+\theta\,\Delta\,t} \otimes \underline{n}|_{t+\theta\,\Delta\,t}\right) \tag{7.33}$$

7.1.4.2 Plastique flow

In the case of a single viscoplastic flow, of increment Δp, the system ((7.30)) reduces to an equation of unknown d. If the plastic flow is activated ((7.21)), this equation becomes:

$$\left(\theta\,\frac{\partial f^{\text{an}}}{\partial p} - 3\,\mu\,\theta\,\frac{\partial f^{\text{an}}}{\partial \sigma_{eq}}\right)\,d = -\frac{\partial f^{\text{an}}}{\partial \sigma_{eq}}\,3\,\mu\,\theta.$$

The expression ((7.32)) of the coherent tangent matrix simplifies then into:

$$\underline{\underline{\mathbf{L}}}^{tc} = \underline{\underline{\mathbf{D}}} - 2\,\mu\,\theta\left(\frac{3\,\Delta p}{\sqrt{6\,\underline{B} : \underline{B}}}\,\underline{\underline{\mathbf{K}}} + \left(\frac{2\,\mu\,\dfrac{\partial f^{\text{an}}}{\partial \sigma_{eq}}}{3\,\mu\,\theta\,\dfrac{\partial f^{\text{an}}}{\partial \sigma_{eq}} - \theta\,\dfrac{\partial f^{\text{an}}}{\partial p}} - \frac{2\,\Delta p}{\sqrt{6\,\underline{B} : \underline{B}}}\right)\,\underline{n}|_{t+\theta\,\Delta\,t} \otimes \underline{n}|_{t+\theta\,\Delta\,t}\right) \tag{7.34}$$

When plastic flow is not activated, the flow equation ((7.21)) and the system ((7.30)) reduce to:

$$\Delta p = 0, \quad \text{et} \quad d = 0,$$

and the tangent matrix consistent with the elasticity tensor:

$$\underline{\underline{\mathbf{L}}}^{tc} = \underline{\underline{\mathbf{D}}}.$$

7.1.5 Tangent matrix

The tangent matrix defined by the relation in velocity:

$$\dot{\underline{\sigma}} = \underline{\underline{\mathbf{L}}} : \dot{\underline{\epsilon}}^{to}$$

Without seeking to justify this statement, the tangent matrix can be calculated as the limit of the coherent tangent matrix when the time step $\Delta\,t$ and the cumulative plastic strain increment $\Delta\,p$ tend to 0 and by posing $\theta = 1$:

$$\underline{\underline{\mathbf{L}}} = \lim_{\substack{\Delta\,t \to 0 \\ \Delta\,p \to 0 \\ \theta = 1}} \underline{\underline{\mathbf{L}}}^{tc}$$

The previous expressions can be reused by taking care of the moment where we want to compute the tangent matrix (at the beginning of the the moment where we want to calculate the tangent matrix (at the beginning or at the end of or end of the time step).

7.1.6 Expression of the tangent matrix for a single mechanism

In the case where the system is limited to a single mechanism, the expression of the tangent matrix simplifies.

7.1.6.1 Viscoplastique flow

In the case of a single viscoplastic flow, the expression ((7.33)) of the tangent matrix reduces to the elasticity matrix.

7.1.6.2 Plastique flow

In the case of a plastic flow, there are two cases whether the flow is under load or not.

In the case of plastic loading, the expression ((7.34)) of the tangent matrix leads to:

$$\underline{\underline{\mathbf{L}}} = \underline{\underline{\mathbf{D}}} - \frac{4\,\mu^2\,\dfrac{\partial f^{\text{an}}}{\partial \sigma_{eq}}}{3\,\mu\,\dfrac{\partial f^{\text{an}}}{\partial \sigma_{eq}} - \dfrac{\partial f^{\text{an}}}{\partial p}}\,\underline{n}\otimes\underline{n}$$

In the elastic domain or in case of discharge, the tangent matrix is reduced to the elastic matrix.

7.2 Use of specific parsers

In this paragraph the `MFront` parsers dedicated to the plastic and viscoplastic incompressible behavior laws of isotropic materials are presented. There are four of them:

- the `IsotropicMisesCreep` parser exclusively handles isotropic viscoplastic behavior laws of the form :
$$\dot{p} = f(\sigma_{eq})$$
- the `IsotropicStrainHardeningMisesCreep` parser manages exclusively the isotropic viscoplastic behavior laws of the form :
$$\dot{p} = f(\sigma_{eq}, p)$$
- the `IsotropicPlasticMisesFlow` parser supports exclusively the isotropic plastic behavior laws of the form :
$$f(\sigma_{eq}, p) \leq 0 \qquad \dot{p} \geq 0 \qquad f(\sigma_{eq}, p)\,\dot{p} = 0$$
- the `MultipleIsotropicMisesFlows` handles an arbitrary combination of flows of the three previous types. The different flows are assumed to be uncoupled.

7.2.1 The `@FlowRule` directive

The directive allows to define a flow.

7.2.1.1 Case of `IsotropicMisesCreep`, `IsotropicStrainHardeningMisesCreep` and `IsotropicPlasticMisesFlow` parsers

For the `IsotropicMisesCreep`, `IsotropicStrainHardeningMisesCreep` and `IsotropicPlasticMisesFlow` parsers, an internal variable named p representing the cumulative inelastic strain is automatically defined. No other internal variable be defined.

The evolution of this variable is introduced by the directive FlowRule@@FlowRule`@FlowRule`. This directive is followed by a block defining the flow. This block must inform the value of the function `f`, whose definition was given above according to the treated the treated flow, and its derivative with respect to the equivalent stress `dfdseq`. Fot the `IsotropicStrainHardeningMisesCreep` and `IsotropicPlasticMisesFlow` parsers it is also necessary to give the derivative of `f` with respect to the to the cumulative deformation `dfdp`.

In the block following the directive, the variables `f`, `dfdseq` and eventually `dfdp` are automatically defined. The updated equivalent stress in $t + \theta\,\Delta\,t$ is accessible through the variable `seq`. For the `IsotropicStrainHardeningMisesCreep` and `IsotropicPlasticMisesFlow` parsers. If required, the updated equivalent strain in $t + \theta\,\Delta\,t$ is available through the variable `p`.

7.2.1.2 Case of the `MultipleIsotropicMisesFlows` parser

Several blocks can be defined in the case of the `MultipleIsotropicMisesFlows` parser. The directive is followed by one of the three supported flow types, respectively `Creep`, `StrainHardeningCreep` and `Plasticity`. The next block describes the flow following the rules given in the previous paragraph. Note that the declaration of a new flow automatically declares the associated cumulative viscoplastic deformation under the name `pi` where `i`is the flow number defined so far. It is only possible to to associate glossary names or bounds to these variables only after the the definition of the block.

7.2.1.3 Code transformation in `@FlowRule` blocks

The code in the blocks following the directive is modified as follows:

- the external variables are replaced by their values in $t + \theta\,\Delta\,t$;
- the cumulative inelastic strain p is replaced by its value in $t + \theta\,\Delta\,t$.

7.2.2 Update of auxiliary variables

The directive allows to update the auxiliary variables.

7.2.2.1 Inelastic strains

The inelastic deformations $\underline{\epsilon}^{an}{}_{i}$ are not internal variables. To access them (for post-processing), it is possible to define auxiliary internal variables.

These variables are updated after the internal variables and the stresses, but before the external variables (including temperature) or the total deformation[^11].

7.2.2.2 Example of total inelastic deformation

Assume that the total inelastic deformation is represented by an auxiliary tensor variable tensor variable named `evp` (declared by the directive). This variable can be computed, in the block following the directive as follows:

```
evp += deto-deel;
```

Another way to calculate this variable is as follows:

```
evp = eto+deto-eel;
```

which shows that the elastic deformation has been updated and not the total deformation.

7.2.3 Numerical parameters defined automatically

The θ parameter of the θ-method is equal to 1 by default for `IsotropicPlasticMisesFlow` parser and $\frac{1}{2}$ for the other parsers. This default value can be modified by the directive . This value is also a parameter of the law, named `theta`, which can be modified at runtime.

The default value of the stop criterion is 10^{-8}. This default value value can be changed with the . This value is also a parameter of the law, named `epsilon`, which can be changed at runtime. at runtime.

The maximum number of iterations of the algorithm is 100 by default. This default value can be changed by the directive . This value is also a parameter of the law, named `iterMax`, which can be changed at runtime.

7.2.4 Reserved names

The names reserved by this parser are described in appendix (#sec:noms-de-variables).

7.2.5 Viscoplastic behavior of SiC

The aim is to describe the viscoplastic behavior of SiC. SiC is assumed to have an isotropic viscoplastic behavior given by:

$$\dot{p} = f(\sigma_{eq})$$

où

- p is the equivalent viscoplastic strain ;
- σ_{eq} is the von-Mises stress.

The flow function is given by :

$$f(\sigma_{eq}) = \left(A \exp\left(-\frac{B}{T}\right) + a\,\phi\right)\sigma_{eq} \tag{7.35}$$

where:

- A, B et a are coefficients ;
- T is the temperature ;
- ϕ is the fast neutron flux ;

This behavior law depends on an external variable, the flux of fast neutron ϕ.

To implement this behavior law, the `IsotropicMisesCreep` parser is used. The source code is as follows :

```
@Parser IsotropicMisesCreep;
@Behaviour SiCCreep;
@Author É.Brunon;
@Date 06 / 12 / 07;

@Description {
  Matériaux RCG - T et RCG -
      R Un point sur le carbure de Silicium NT SESC / LIAC 02 -
      024 ind 0 de décembre 2002 J.M.ESCLEINE
 §8.4.2 Page 34
}

@ExternalStateVar real flux;

@StaticVar real A = 4.4e3;
@StaticVar real B = 76.0e3;
@StaticVar real a = 1.0e-37;

@LocalVar real AF1;
@LocalVar real AF3;

@InitLocalVars {
  AF1 = A * exp(-B / (T + theta * dT));
  AF3 = a * (flux + theta * dflux);
}

@FlowRule {
  df_dseq = AF1 + AF3;
  f = seq * df_dseq;
}
```

- The `@DSL` keyword

  The first line, starting with the keyword `@DSL`, describes the used analyzer, here `IsotropicMisesCreep`.

- The `@Behaviour` keyword

  The second line, starting with the keyword , gives the name of the law of behavior.

- The `@Author` and `@Date` keywords

The third and fourth lines respectively inform the author of the file of the file and the creation date using the keywords and .

- The `@Description` keyword

  The keyword allows to give the bibliographic references from which the law is extracted.

- The `@ExternalStateVariable` keyword

  The keyword defines a scalar external variable (`real`) named `flux`. This variable, and its increment `dflux`, are then accessible.

- The `@StaticVariable` keyword

  The keyword is used to define the constants used by the law. These constants are scalars (`real`).

- The `@LocalVariable` keyword

  The keyword is used to define local working variables. The role of variables `AF1` and `AF3` will be explained in the following.

- The `@InitLocalVariables` keyword

  The keyword allows to write code called before any calculation. We initialize initialize the values of the coefficients of the law. Indeed, the method integration method used evaluates the function f at time $t + theta, \Delta t$ (implicit integration. The temperature-dependent coefficient is thus known and it is advantageous to evaluate it here rather than during the convergence iterations of the algorithm (we save calls to the exponential function). The temperature at time $t + \theta\,\Delta t$ is equal to $T + \theta\,\Delta T$. In the same way the value of the fast neutron flux ϕ is equal to $\phi + \theta\,\Delta\phi$.

- The `@FlowRule` keyword

  The keyword allows to fill in the flow function f and its derivative $\dfrac{\mathrm{d}f}{\mathrm{d}\sigma_{eq}}$.

Chapter 8

The `StandardElastoViscoPlasticity` brick

This page describes the `StandardElastoViscoPlasticity` brick. This brick is used to describe a specific class of strain based behaviours based on an additive split of the total strain $\underline{\varepsilon}^{\mathrm{to}}$ into an elastic part $\underline{\varepsilon}^{\mathrm{el}}$ and one or several inelastic strains describing plastic (time-independent) flows and/or viscoplastic (time-dependent) flows:

$$\underline{\varepsilon}^{\mathrm{to}} = \underline{\varepsilon}^{\mathrm{el}} + \sum_{i_{\mathrm{p}}=0}^{n_{\mathrm{p}}} \underline{\varepsilon}^{\mathrm{p}}_{i_{\mathrm{p}}} + \sum_{i_{\mathrm{vp}}=0}^{n_{\mathrm{vp}}} \underline{\varepsilon}^{\mathrm{vp}}_{i_{\mathrm{vp}}}$$

This equation defines the equation associated with the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$.

The brick decomposes the behaviour into two components:

- the stress potential which defines the relation between the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$ and possibly some damage variables and the stress measure $\underline{\sigma}$. As the definition of the elastic properties can be part of the definition of the stress potential, the thermal expansion coefficients can also be defined in the block corresponding to the stress potential.
- a list of inelastic flows.

Porous viscoplasticity
This page only introduces stress criteria which are not coupled with the evolution of the porosity. Stress criteria coupled with the evolution of porosity is described on a dedicated page.

8.0.1 A detailled Example

```
@Brick "StandardElastoViscoPlasticity" {
  // Here the stress potential is given by the Hooke law. We define:
  // - the elastic properties (Young modulus and Poisson ratio).
  //   Here the Young modulus is a function of the temperature.
  //   The Poisson ratio is constant.
  // - the thermal expansion coefficient
  // - the reference temperature for the thermal expansion
  stress_potential : "Hooke" {
    young_modulus : "2.e5 - (1.e5*((T - 100.)/960.)**2)",
    poisson_ratio : 0.3,
    thermal_expansion : "1.e-5 + (1.e-5  * ((T - 100.)/960.) ** 4)",
    thermal_expansion_reference_temperature : 0
  },
  // Here we define only one viscplastic flow defined by the Norton law,
  // which is based:
  // - the von Mises stress criterion
  // - one isotorpic hardening rule based on Voce formalism
  // - one kinematic hardening rule following the Armstrong-Frederick law
  inelastic_flow : "Norton" {
    criterion : "Mises",
```

```
    isotropic_hardening : "Voce" {R0 : 200, Rinf : 100, b : 20},
    kinematic_hardening : "Armstrong-Frederick" {
      C : "1.e6 - 98500 * (T - 100) / 96",
      D : "5000 - 5* (T - 100)"
    },
    K : "(4200. * (T + 20.) - 3. * (T + 20.0)**2)/4900.",
    n : "7. - (T - 100.) / 160.",
    Ksf : 3
  }
};
```

8.1 List of available stress potentials

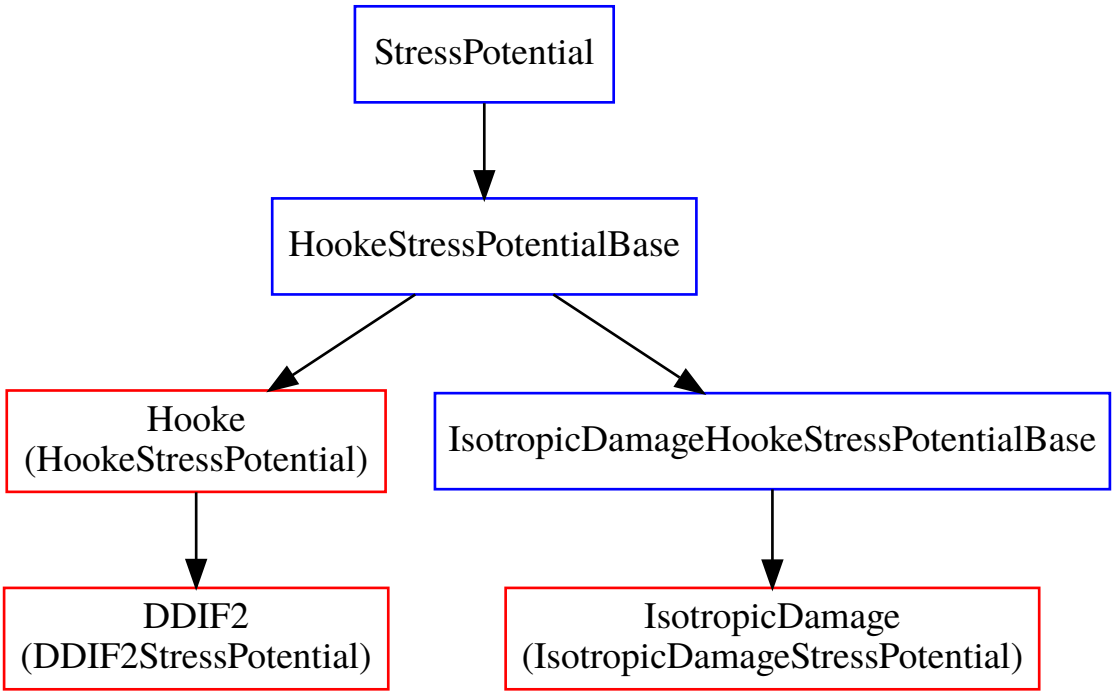

Figure 8.1: Relations between stress potentials. The stress potentials usable by the end users are marked in red.

8.1.1 The Hooke stress potential

This stress potential implements the Hooke law, i.e. a linear relation between the elastic strain and the stress, as follows:

$$\underline{\sigma} = \underline{\underline{\mathbf{D}}} : \underline{\varepsilon}^{\mathrm{el}}$$

where $\underline{\underline{\mathbf{D}}}$ is the elastic stiffness tensor.

This stress potential applies to isotropic and orthotropic materials. This stress potential provides:

- Automatic computation of the stress tensor at various stages of the behaviour integration.
- Automatic computation of the consistent tangent operator.
- Automatic support for plane stress and generalized plane stress modelling hypotheses (The axial strain is defined as an additional state variable and the associated equation in the implicit system is added to enforce the plane stess condition).
- Automatic addition of the standard terms associated with the elastic strain state variable.

The Hooke stress potential is fully described here.

8.1.2 The `IsotropicDamage` stress potential

This stress potential adds to the Hooke stress potential the description of an isotropioc damage. The relation

$$\underline{\sigma} = (1-d)\,\underline{\underline{\mathbf{D}}} : \underline{\varepsilon}^{\mathrm{el}}$$

where $\underline{\underline{\mathbf{D}}}$ is the elastic stiffness tensor and d is the isotropic damage variable.

This stress potential inherits all the features and options provided by the Hooke stress potential. The Hooke stress potential is fully described here.

8.1.3 The `DDIF2` stress potential

The `DDIF2` behaviour is used to describe the brittle nature of nuclear fuel ceramics and is usually coupled with a description of the viscoplasticity of those ceramics (See for example [23]).

This stress potential adds to the Hooke stress potential the description of cracking through an additional strain. As such, it inherits all the features provided by the Hooke stress potential.

The Hooke stress potential is fully described here.

The DDIF2 stress potential is fully described here.

8.2 Inelastic flows

8.2.1 List of available inelastic flows

8.2.1.1 The `Plastic` inelastic flow

The plastic flow is defined by:

- a yield surface f
- a plastic potential g

The plastic strain rate satisfies:

$$\dot{\underline{\varepsilon}}^{\mathrm{p}} = \dot{\lambda}\,\frac{\partial g}{\partial \underline{\sigma}}$$

The plastic multiplier satifies the Kuhn-Tucker relation:

$$\begin{cases} \dot{\lambda}\, f(\underline{\sigma}, p) = 0 \\ \dot{\lambda} \geq 0 \end{cases}$$

The flow is associated is f is equal to g. In practice f is defined by a stress criterion ϕ, a set of kinematic hardening rules, and an isotropic hardening rule, as follows:

$$f(\underline{\sigma}, p) = \phi\left(\underline{\sigma} - \sum_i \underline{X}_i\right) - \sum_i R_i(p)$$

where p is the equivalent plastic strain. Here we have decomposed the limit of the elastic domain as a sum, denoted $\sum_i R_i(p)$, to indicate that one may define it by combining various predefined forms of isotropic hardening rules (Voce, Linear, etc.) defined hereafter.

8.2.1.1.1 Maximum equivalent stress in the `Plastic` flow During the Newton iterations, the current estimate of the equivalent stress σ_{eq} may be significantly higher than the elastic limit R. This may lead to a divergence of the Newton algorithm.

One may reject the Newton steps leading to such high values of the elastic limit by specifying a relative threshold denoted α, i.e. if σ_{eq} is greater than $\alpha \cdot R$. A typical value for α is 1.5. This relative threshold is specified by the `maximum_equivalent_stress_factor` option.

In some cases, rejecting steps may also lead to a divergence of the Newton algorithm, so one may specify a relative threshold β on the iteration number which desactivate this check, i.e. the check is performed only if the current iteration number is below $\beta \cdot i_{\max}$ where $i_{\max}$ is the maximum number of iterations allowed for the Newton algorithm. A typical value for β is 0.4. This relative threshold is specified by the `equivalent_stress_check_maximum_iteration_factor` option.

```
@DSL Implicit;
@Behaviour PerfectPlasticity;
@Author Thomas Helfer;
@Date 17 / 08 / 2020;
@Description{};

@Epsilon 1.e-14;
@Theta 1;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 200e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "Linear" {R0 : 150e6},
    maximum_equivalent_stress_factor : 1.5,
    equivalent_stress_check_maximum_iteration_factor: 0.4
  }
};
```

8.2.1.1.1.1 Example

8.2.1.2 The `Norton` inelastic flow

The plastic flow is defined by:

- a function $f(\underline{\sigma})$ giving the flow intensity:

$$f(\underline{\sigma}) = A\left\langle\frac{\phi(\underline{\sigma}-\sum_i \underline{X}_i)-\sum_i R_i(p)}{K}\right\rangle^n$$

- a stress criterion ϕ
- a viscoplastic potential g
- one or more kinematic hardening rule (optional), denoted $\underline{X}_i$
- one isotropic hardening rule (optional), denoted R_i

8.2.1.3 The `HyperbolicSine` inelastic flow

The viscoplastic flow is defined by:

- a function $f(\underline{\sigma})$ giving the flow intensity given by:

$$f(\underline{\sigma}) = A\,\sinh\left(\frac{\left\langle\phi(\underline{\sigma}-\sum_i \underline{X}_i)-\sum_i R_i(p)\right\rangle}{K}\right)^n$$

- a stress criterion ϕ
- one or more kinematic hardening rule (optional), denoted $\underline{X}_i$
- one isotropic hardening rule (optional), denoted R_i

8.2.1.4 The `HarmonicSumOfNortonHoffViscoplasticFlows` inelastic flow

The equivalent viscoplastic strain rate $\dot{p}$ is defined as:

$$\frac{1}{\dot{p}} = \sum_{i=1}^{N}\frac{1}{\dot{p}_i}$$

where $\dot{p}_i$ has an expression similar to the the Norton-Hoff viscoplastic flow:

$$\dot{p}_i = A_i\left(\frac{\left\langle\phi\left(\underline{\sigma}-\sum_j \underline{X}_j\right)-\sum_j R_j(p)\right\rangle}{K_i}\right)^{n_i}$$

in which appear:

- a stress criterion ϕ
- one or more kinematic hardening rule (optional), denoted $\underline{X}_i$
- one isotropic hardening rule (optional), denoted R_i

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "HarmonicSumOfNortonHoffViscoplasticFlows" {
    criterion : "Mises",
    A : {8e-67, 8e-67},
    K : {1,1},
    n : {8.2,8.2}
  }
};
```

8.2.1.4.1 Example

8.2.1.4.2 Newton steps rejection The exponential nature of the hyperbolic sinus function may lead to divergence of the Newton method. To avoid this, one may specify a relative threshold denoted K_{sf}: if the stress estimate is greater than $K_{sf}\,K$, the step is rejected.

8.2.1.5 The `UserDefinedViscoplasticity` inelastic flow

The `UserDefinedViscoplasticity` inelastic flow allows the user to specify the viscoplastic strain rate `vp` as a function of `f` and `p` where:

- `f` is the positive part of the $\phi(\underline{\sigma} - \sum_i \underline{X}_i) - \sum_i R_i(p)$ where ϕ is the stress criterion.
- `p` is the equivalent viscoplastic strain.

This function shall be given by a string option named `vp`. This function must depend on `f`. Dependance to `p` is optional.

The function may also depend on other variables. Let `A` be such a variable. The `UserDefinedViscoplasticity` flow will look if an option named `A` has been given to the flow:

- If this option exists, it will be interpreted as a material coefficient as usal and this option can be a number, a formula or the name of an external `MFront` file.
- If this option does not exist, a suitable variable will be search in the variables defined in the behaviour (static variables, parameters, material properties, etc…).

If required, the derivatives of `vp` with respect to `f` and `p` can be provided through the options `dvp_df` and `dvp_dp`. The derivatives `dvp_df` and `dvp_dp` can depend on two additional variables, `vp` and `seps`, which denotes the viscoplastic strain rate and a stress threshold.

If those derivatives are not provided, automatic differentiation will be used. The user shall be warned that the automatic differentiation provided by the `tfel::math::Evaluator` class may result in inefficient code.

```
@Parameter temperature Ta = 600;
@Parameter strain p0 = 1e-8;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "UserDefinedViscoplasticity" {
    criterion : "Mises",
    E : 8.2,
    A : "8e-67 * exp(- T / Ta)",
    m : 0.32,
    vp : "A * (f ** E) / ((p + p0) ** m)",
    dvp_df : "E * vp / (max(f, seps))"
    // dvp_dp is evaluated by automatic differentiation (which is not recommended)
  }
};
```

Example of usage

8.2.2 Newton steps rejections based on the change of the flow direction between two successive estimates

Some stress criteria (Hosford 1972, Barlat 2004, Mohr-Coulomb) shows sharp edges that may cause the failure of the standard Newton algorithm, due to oscillations in the prediction of the flow direction.

Rejecting Newton steps leading to a too large variation of the flow direction between the new estimate of the flow direction and the previous estimate is a cheap and efficient method to overcome this issue. This method can be viewed as a bisectional linesearch based on the Newton prediction: the Newton steps magnitude is divided by two if its results to a too large change in the flow direction.

More precisely, the change of the flow direction is estimated by the computation of the cosine of the angle between the two previous estimates:

$$\cos\left(\alpha_{\underline{n}}\right) = \frac{\underline{n} : \underline{n}_p}{\|\underline{n}\| \, \|\underline{n}\|}$$

with $\|\underline{n}\| = \sqrt{\underline{n} : \underline{n}}$.

The Newton step is rejected if the value of $\cos\left(\alpha_{\underline{n}}\right)$ is lower than a user defined threshold. This threshold must be in the range $[-1 : 1]$, but due to the slow variation of the cosine near 0, a typical value of this threshold is 0.99 which is equivalent to impose that the angle between two successive estimates is below 8°.

8.2.3 List of available stress criteria

The following section describes stress criteria available by default. However, the `StandardElastoViscoPlasticity` brick can also be extended by the user:

- this page describes how to add a new stress criterion which is not coupled with the evolution of the porosity.
- this page describes how to add a new stress criterion coupled with the evolution of the porosity.

8.2.3.1 von Mises stress criterion

8.2.3.1.1 Definition The von Mises stress is defined by:

$$\sigma_{\text{eq}} = \sqrt{\frac{3}{2} \underline{s} : \underline{s}} = \sqrt{3 \, J_2}$$

where: - $\underline{s}$ is the deviatoric stress defined as follows:

$$\underline{s} = \underline{\sigma} - \frac{1}{3} \operatorname{tr}(\underline{\sigma}) \, \underline{I}$$

- J_2 is the second invariant of $\underline{s}$.

In terms of the eigenvalues of the stress, denoted by σ_1, σ_2 and σ_3, the von Mises stress can also be defined by:

$$\sigma_{\text{eq}} = \sqrt{\frac{1}{2}\left(\left|\sigma_1 - \sigma_2\right|^2 + \left|\sigma_1 - \sigma_3\right|^2 + \left|\sigma_2 - \sigma_3\right|^2\right)}$$

8.2.3.1.2 Options This stress criterion does not have any option.

```
criterion : "Mises"
```

8.2.3.2 Drucker 1949 stress criterion

The Drucker 1949 stress is defined by:

$$\sigma_{\text{eq}} = \sqrt{3} \sqrt[6]{J_2^3 - c \, J_3^2}$$

where:

- $J_2 = \frac{1}{2} \underline{s} : \underline{s}$ is the second invariant of $\underline{s}$.
- $J_3 = \det(\underline{s})$ is the third invariant of $\underline{s}$.

- $\underline{s}$ is the deviatoric stress defined as follows:

$$\underline{s} = \underline{\sigma} - \frac{1}{3}\operatorname{tr}(\underline{\sigma})\,\underline{I}$$

8.2.3.3 Example

```
criterion : "Drucker 1949" {c : 1.285}
```

8.2.3.3.1 Options

The user must provide the c coefficient.

8.2.3.4 Hosford 1972 stress criterion

The Hosford equivalent stress is defined by (see [24]):

$$\sigma_{\text{eq}}^{H} = \sqrt[a]{\frac{1}{2}\left(|\sigma_1 - \sigma_2|^a + |\sigma_1 - \sigma_3|^a + |\sigma_2 - \sigma_3|^a\right)}$$

where σ_1, σ_2 and σ_3 are the eigenvalues of the stress.

Therefore, when a goes to infinity, the Hosford stress reduces to the Tresca stress. When $n = 2$ the Hosford stress reduces to the von Mises stress.

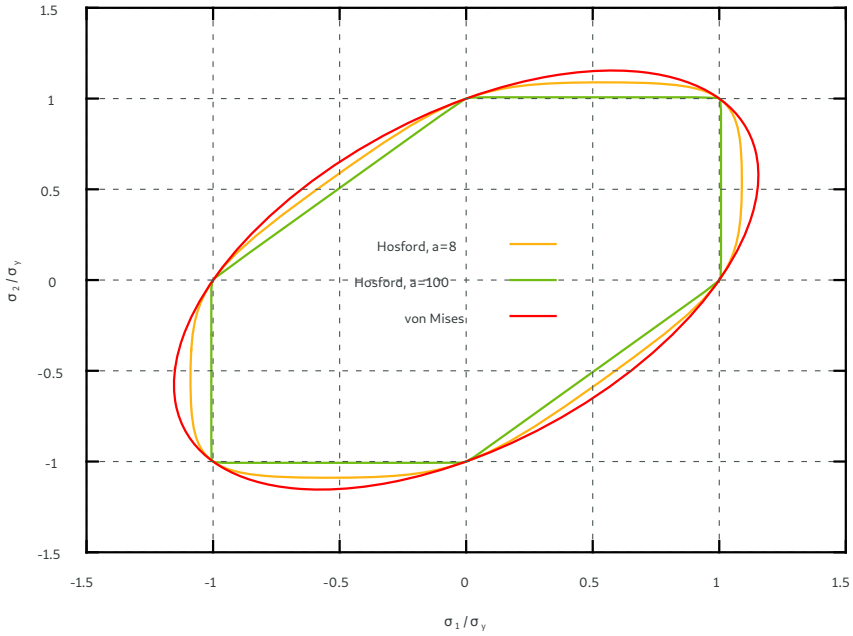

Figure 8.2: Comparison of the Hosford stress $a = 100, a = 8$ and the von Mises stress in plane stress

8.2.3.5 Options

The Hosford exponent `a` is mandatory.

Specifying the eigen solver using the `eigen_solver` option is optional. This option can have the value `default` or the value `Jacobi`.

8.2.3.6 Example

```
criterion : "Hosford" {a : 6}
```

8.2.3.7 Notes

The Hosford yield surface may have sharp edges which may lead to divergence of the Newton algorithm du to oscillations of the flow direction. Specifying a threshold for the angle between. See Section 8.2.2 for details.

8.2.3.7.1 Options

The user must provide the Hosford exponent a.

8.2.3.8 Isotropic Cazacu 2004 stress criterion

In order to describe yield differential effects, the isotropic Cazacu 2004 equivalent stress criterion is defined by (see [25]):

$$\sigma_{\text{eq}} = \sqrt[3]{J_2^{3/2} - c\, J_3}$$

where:

- $J_2 = \frac{1}{2}\, \underline{s} : \underline{s}$ is the second invariant of $\underline{s}$.
- $J_3 = \det(\underline{s})$ is the third invariant of $\underline{s}$.
- $\underline{s}$ is the deviatoric stress defined as follows:

$$\underline{s} = \underline{\sigma} - \frac{1}{3}\,\text{tr}(\underline{\sigma})\,\underline{I}$$

8.2.3.9 Example

```
criterion : "Isotropic Cazacu 2004" {c : -1.056}
```

8.2.3.10 Hill stress criterion

This `Hill` criterion, also called `Hill1948` criterion, is based on the equivalent stress σ_{eq}^H defined as follows:

$$\begin{aligned}\sigma_{\text{eq}}^H &= \sqrt{\underline{\sigma} : \underline{\underline{\mathbf{H}}} : \underline{\sigma}} \\ &= \sqrt{F\left(\sigma_{11}-\sigma_{22}\right)^2 + G\left(\sigma_{22}-\sigma_{33}\right)^2 + H\left(\sigma_{33}-\sigma_{11}\right)^2 + 2\,L\sigma_{12}^2 + 2\,M\sigma_{13}^2 + 2\,N\sigma_{23}^2}\end{aligned}$$

> **Warning** This convention is given in the book of Lemaître et Chaboche and seems to differ from the one described in most other books.

8.2.3.10.1 Options

This stress criterion has 6 mandatory options: `F`, `G`, `H`, `L`, `M`, `N`. Each of these options must be interpreted as material property.

> **Orthotropic axis convention** If an orthotropic axis convention is defined (See the `@OrthotropicBehaviour` keyword' documentation), the coefficients of the Hill tensor can be exchanged for some modelling hypotheses. The coefficients `F`, `G`, `H`, `L`, `M`, `N` must always correspond to the three dimensional case.

8.2.3.11 Example

```
criterion : "Hill" {F : 0.371, G : 0.629, H : 4.052, L : 1.5, M : 1.5, N : 1.5},
```

8.2.3.12 Cazacu 2001 stress criterion

Within the framework of the theory of representation, generalizations to orthotropic conditions of the invariants of the deviatoric stress have been proposed by Cazacu and Barlat (see [26]):

- The generalization of J_2 is denoted J_2^O. It is defined by:

$$J_2^O = a_6\, s_{yz}^2 + a_5\, s_{xz}^2 + a_4\, s_{xy}^2 + \frac{a_2}{6}\,(s_{yy}-s_{zz})^2 + \frac{a_3}{6}\,(s_{xx}-s_{zz})^2 + \frac{a_1}{6}\,(s_{xx}-s_{yy})^2$$

  where the $a_i\big|_{i\in[1:6]}$ are six coefficients describing the orthotropy of the material.

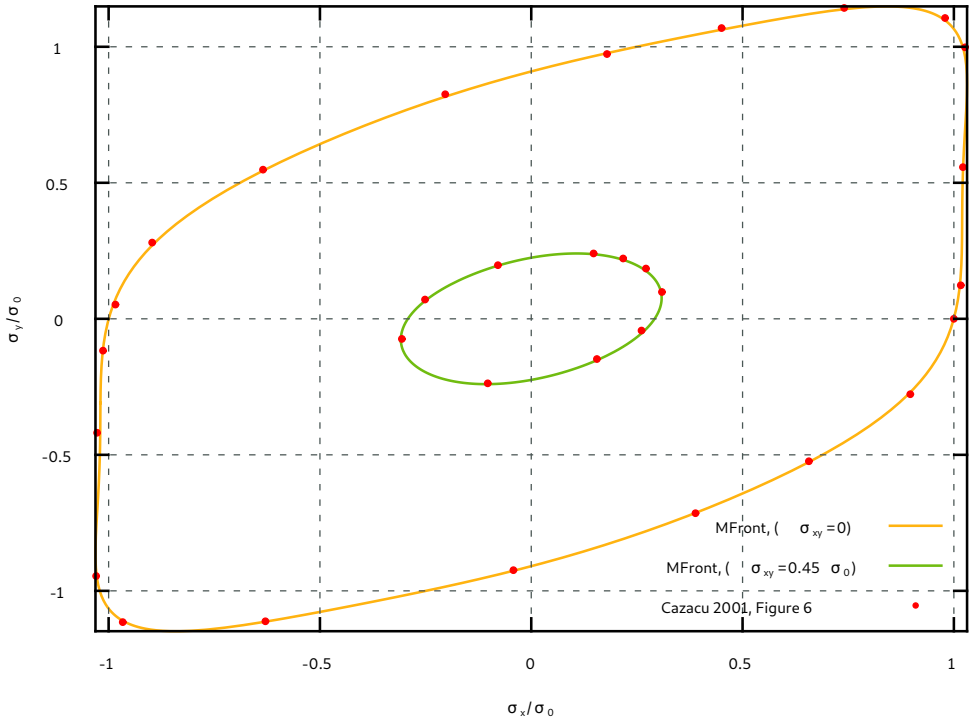

Figure 8.3: Plane stress yield surface ($\sigma_{xy} = 0$ and $\sigma_{xy} = 0.45\,\sigma_0$) of 2090-T3 alloy sheet as predicted by the generalization of the Drucker yield criterion using generalized invariants (See [26], Figure 6).

- The generalization of J_3 is denoted J_3^O. It is defined by:

$$\begin{aligned}
J_3^O =& \frac{1}{27}\left(b_1+b_2\right)s_{xx}^3 + \frac{1}{27}\left(b_3+b_4\right)s_{yy}^3 + \frac{1}{27}\left(2\left(b_1+b_4\right)-b_2-b_3\right)s_{zz}^3 \\
&- \frac{1}{9}\left(b_1\,s_{yy}+b_2 s_{zz}\right)s_{xx}^2 \\
&- \frac{1}{9}\left(b_3\,s_{zz}+b_4\,s_{xx}\right)s_{yy}^2 \\
&- \frac{1}{9}\left(\left(b_1-b_2+b_4\right)s_{xx}+\left(b_1-b3+b_4\right)s_{yy}\right)s_{zz}^3 \\
&+ \frac{2}{9}\left(b_1+b_4\right)s_{xx}\,s_{yy}\,s_{zz} \\
&- \frac{s_{xz}^2}{3}\left(2\,b_9\,s_{yy}-b_8\,s_{zz}-\left(2\,b_9-b_8\right)s_{xx}\right) \\
&- \frac{s_{xy}^2}{3}\left(2\,b_{10}\,s_{zz}-b_5\,s_{yy}-\left(2\,b_{10}-b_5\right)s_{xx}\right) \\
&- \frac{s_{yz}^2}{3}\left(\left(b_6+b_7\right)s_{xx}-b_6\,s_{yy}-b_7\,s_{zz}\right) \\
&+ 2\,b_{11}\,s_{xy}\,s_{xz}\,s_{yz}
\end{aligned}$$

  where the $b_i|_{i\in[1:11]}$ are eleven coefficients describing the orthotropy of the material.

Those invariants may be used to generalize isotropic yield criteria based on J_2 and J_3 invariants to orthotropy. The Cazacu 2001 equivalent stress criterion is defined as the orthotropic counterpart of the Drucker 1949 yield criterion, as follows (see [26]):

$$\sigma_{\text{eq}} = \sqrt{3}\sqrt[6]{\left(J_2^O\right)^3 - c\,\left(J_3^O\right)^2}$$

8.2.3.12.1 Options This criterion requires the following options:

- `a`, as an array of 6 material properties.
- `b`, as an array of 11 material properties.

- c, as a material property.

```
criterion : "Cazacu 2001" {
  a : {0.586, 1.05, 0.823, 0.96, 1, 1},
  b : {1.44, 0.061, -1.302, -0.281, -0.375, 1, 1, 1, 1, 0.445, 1},
  c : 1.285
},
```

8.2.3.12.2 Example

8.2.3.12.3 Restrictions Proper support of orthotropic axes conventions has not been implemented yet for the computation of the J_2^O and J_3^O. Thus, the following restrictions apply:

- if no orthotropic axis convention is defined, only the `Tridimensional` modelling hypothesis is supported.q
- if the `Plate` orthotropic axis convention is used, only the `Tridimensional` and `PlaneStress` modelling hypotheses are supported.
- if the `Pipe` orthotropic axis convention is used, only theI `Tridimensional`, `Axisymmetrical`, `AxisymmetricalGeneralisedPla` and `AxisymmetricalGeneralisedPlainStres` modelling hypotheses are supported.

8.2.3.13 Orthotropic Cazacu 2004 stress criterion

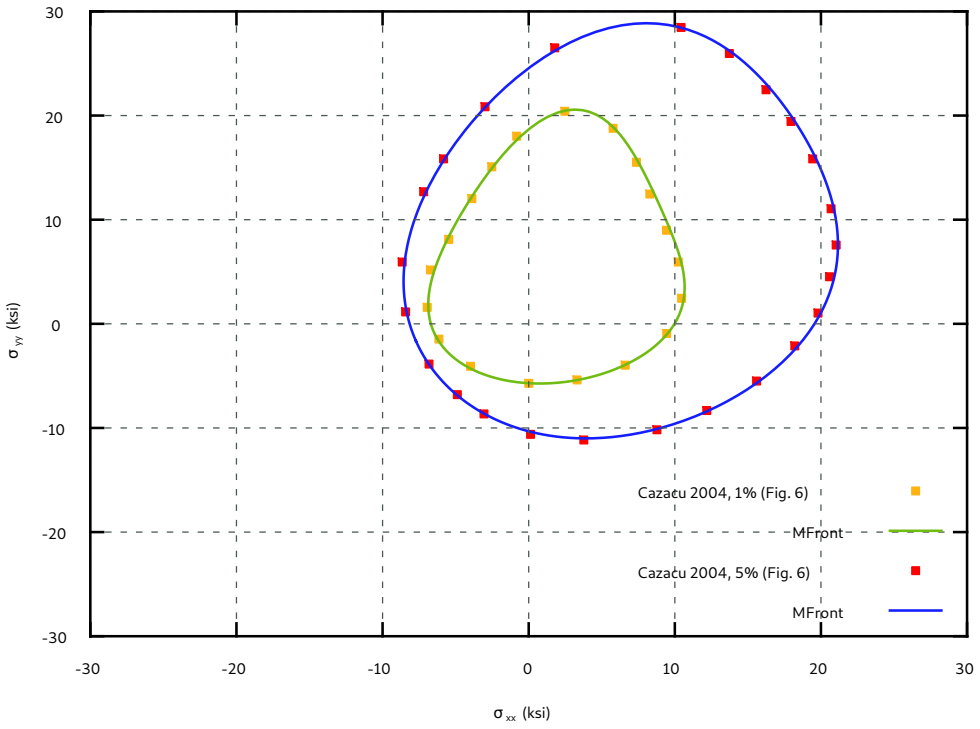

Figure 8.4: Plane stress yield loci for a magnesium sheet (See [25], Figure 6).

Using the invariants J_2^O and J_3^O previously defined, Cazacu and Barlat proposed the following criterion (See [25]):

$$\sigma_{\text{eq}} = \sqrt[3]{\left(J_2^O\right)^{3/2} - c\, J_3^O}$$

8.2.3.13.1 Options This criterion requires the following options:

- a, as an array of 6 material properties.
- b, as an array of 11 material properties.
- c, as a material property.

```
criterion : "Orthotropic Cazacu 2004" {
  a : {0.586, 1.05, 0.823, 0.96, 1, 1},
  b : {1.44, 0.061, -1.302, -0.281, -0.375, 1, 1, 1, 1, 0.445, 1},
  c : 1.285
},
```

8.2.3.13.2 Example

8.2.3.13.3 Restrictions Proper support of orthotropic axes conventions has not been implemented yet for the computation of the J_2^O and J_3^O. Thus, the following restrictions apply:

- if no orthotropic axis convention is defined, only the `Tridimensional` modelling hypothesis is supported.q
- if the `Plate` orthotropic axis convention is used, only the `Tridimensional` and `PlaneStress` modelling hypotheses are supported.
- if the `Pipe` orthotropic axis convention is used, only the `Tridimensional`, `Axisymmetrical`, `AxisymmetricalGeneralisedPlai` and `AxisymmetricalGeneralisedPlainStres` modelling hypotheses are supported.

8.2.3.14 Barlat 2004 stress criterion

The Barlat equivalent stress is defined as follows (See [27]):

$$\sigma_{\text{eq}}^{B} = \sqrt[a]{\frac{1}{4}\left(\sum_{i=0}^{3}\sum_{j=0}^{3}\left|s_i' - s_j''\right|^a\right)}$$

where s_i' and s_i'' are the eigenvalues of two transformed stresses $\underline{s}'$ and $\underline{s}''$ by two linear transformation $\underline{\underline{\mathbf{L}}}'$ and $\underline{\underline{\mathbf{L}}}''$:

$$\begin{cases} \underline{s}' = \underline{\underline{\mathbf{L}}}' : \underline{\sigma} \\ \underline{s}'' = \underline{\underline{\mathbf{L}}}'' : \underline{\sigma} \end{cases}$$

The linear transformations $\underline{\underline{\mathbf{L}}}'$ and $\underline{\underline{\mathbf{L}}}''$ are defined by 9 coefficients (each) which describe the material orthotropy. There are defined through auxiliary linear transformations $\underline{\underline{\mathbf{C}}}'$ and $\underline{\underline{\mathbf{C}}}''$ as follows:

$$\underline{\underline{\mathbf{L}}}' = \underline{\underline{\mathbf{C}}}' : \underline{\underline{\mathbf{M}}}$$
$$\underline{\underline{\mathbf{L}}}'' = \underline{\underline{\mathbf{C}}}'' : \underline{\underline{\mathbf{M}}}$$

where $\underline{\underline{\mathbf{M}}}$ is the transformation of the stress to its deviator:

$$\underline{\underline{\mathbf{M}}} = \underline{\underline{\mathbf{I}}} - \frac{1}{3}\underline{I} \otimes \underline{I}$$

The linear transformations $\underline{\underline{\mathbf{C}}}'$ and $\underline{\underline{\mathbf{C}}}''$ of the deviator stress are defined as follows:

$$\underline{\underline{\mathbf{C}}}' = \begin{pmatrix} 0 & -c_{12}' & -c_{13}' & 0 & 0 & 0 \\ -c_{21}' & 0 & -c_{23}' & 0 & 0 & 0 \\ -c_{31}' & -c_{32}' & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{44}' & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{55}' & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{66}' \end{pmatrix} \quad \text{and} \quad \underline{\underline{\mathbf{C}}}'' = \begin{pmatrix} 0 & -c_{12}'' & -c_{13}'' & 0 & 0 & 0 \\ -c_{21}'' & 0 & -c_{23}'' & 0 & 0 & 0 \\ -c_{31}'' & -c_{32}'' & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{44}'' & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{55}'' & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{66}'' \end{pmatrix}$$

When all the coefficients c_{ji}' and c_{ji}'' are equal to 1, the Barlat equivalent stress reduces to the Hosford equivalent stress.

8.2.3.14.1 Options This stress criterion has 3 mandatory options:

- the coefficients of the first linear transformation $\underline{\underline{\mathbf{L}}}'$, as `l1`;
- the coefficients of the second linear transformation $\underline{\underline{\mathbf{L}}}''$, as `l2`;
- the Barlat exponent a

Orthotropic axis convention If an orthotropic axis convention is defined (See the `@OrthotropicBehaviour` keyword' documentation), the coefficients of the linear transformationscan be exchanged for some modelling hypotheses. The coefficients given by the user must always correspond to the three dimensional case.

Specifying the eigen solver using the `eigen_solver` option is optional. This option can have the value `default` or the value `Jacobi`.

8.2.3.15 Example

```
criterion : "Barlat" {
  a : 8,
  l1 : {-0.069888, 0.079143, 0.936408, 0.524741, 1.00306, 1.36318, 0.954322,
        1.06906, 1.02377},
  l2 : {0.981171, 0.575316, 0.476741, 1.14501, 0.866827, -0.079294, 1.40462,
        1.1471, 1.05166}
}
```

8.2.3.16 Notes

The Barlat 2004 yield surface may have sharp edges which may lead to divergence of the Newton algorithm du to oscillations of the flow direction. Specifying a threshold for the angle between. See Section 8.2.2 for details.

8.2.4 List of available isotropic hardening rules

Note
The follwing hardening rules can be combined to define more complex hardening rules. For example, the following code adds to Voce hardening:

```
isotropic_hardening : "Voce" {R0 : 600e6, Rinf : 900e6, b : 1},
isotropic_hardening : "Voce" {R0 : 0, Rinf : 300e6, b : 10},
```

The previous code is equalivent to the following hardening rule:

$$R(p) = R_0^0 + \left(R_\infty^0 - R_0^0\right)\left(1 - \exp\left(-b^0\, p\right)\right) + R_\infty^1\left(1 - \exp\left(-b^1\, p\right)\right)$$

with:

- $R_0^0 = 600\,.\,10^6\ Pa$
- $R_\infty^0 = 900\,.\,10^6\ Pa$
- $R_\infty^1 = 300\,.\,10^6\ Pa$
- $b^0 = 1$
- $b^1 = 10$

8.2.4.1 The `Linear` isotropic hardening rule

The `Linear` isotropic hardening rule is defined by:

$$R(p) = R_0 + H\, p$$

8.2.4.1.1 Options The `Linear` isotropic hardening rule expects one of the two following material properties:

- `R0`: the yield strength
- `H`: the hardening slope

Note
If one of the previous material property is not defined, the generated code is optimised and there will be no parameter asscoiated with it. To avoid this, you must define the material property and assign it to a zero value.

8.2.4.1.2 Example

The following code can be added in a block defining an inelastic flow:

```
isotropic_hardening : "Linear" {R0 : 120e6, H : 438e6},
```

8.2.4.2 The `Swift` isotropic hardening rule

The `Swift` isotropic hardening rule is defined by:

$$R(p) = R_0 \left(\frac{p + p_0}{p_0} \right)^n$$

8.2.4.2.1 Options

The `Swift` isotropic hardening rule expects three material properties:

- `R0`: the yield strength
- `p0`
- `n`

8.2.4.2.2 Example

The following code can be added in a block defining an inelastic flow:

```
isotropic_hardening : "Swift" {R0 : 120e6, p0 : 1e-8, n : 5.e-2}
```

8.2.4.3 The `Power` isotropic hardening rule (since TFEL 3.4)

The `Power` isotropic hardening rule is defined by:

$$R(p) = R_0 \left(p + p_0 \right)^n$$

8.2.4.3.1 Options

The `Power` isotropic hardening rule expects at least the following material properties:

- `R0`: the coefficient of the power law
- `n`: the exponent of the power law

The `p0` material property is *optional* and generally is considered a numerical parameter to avoir an initial infinite derivative of the power law when the exponent is lower than 1.

8.2.4.3.2 Example

The following code can be added in a block defining an inelastic flow:

```
isotropic_hardening : "Linear" {R0 : 50e6},
isotropic_hardening : "Power" {R0 : 120e6, p0 : 1e-8, n : 5.e-2}
```

8.2.4.4 The `Voce` isotropic hardening rule

The `Voce` isotropic hardening rule is defined by:

$$R(p) = R_\infty + (R_0 - R_\infty)\, exp(-b\, p)$$

8.2.4.4.1 Options

The `Voce` isotropic hardening rule expects three material properties:

- `R0`: the yield strength
- `Rinf`: the utimate strength
- `b`

8.2.4.4.2 Example

The following code can be added in a block defining an inelastic flow:

```
isotropic_hardening : "Voce" {R0 : 200, Rinf : 100, b : 20}
```

8.2.4.5 User defined isotropic hardening rule

The `UserDefined` isotropic hardening rule allows the user to specify the radius of the yield surface as a function of the equivalent plastic strain `p`.

This function shall be given by a string option named `R` and must depend on `p`. The function may also depend on other variables. Let `A` be such a variable. The `UserDefined` isotropic hardening rule will look if an option named `A` has been given:

- If this option exists, it will be interpreted as a material coefficient as usal and this option can be a number, a formula or the name of an external `MFront` file.
- If this option does not exist, a suitable variable will be search in the variables defined in the behaviour (static variables, parameters, material properties, etc…).

If required, the derivative of `R` with respect to `f` and `p` can be provided through the option `dR_dp`. The derivative `dR_dp` can depend on the variable `R`.

If this derivative is not provided, automatic differentiation will be used. The user shall be warned that the automatic differentiation provided by the `tfel::math::Evaluator` class may result in inefficient code.

```
@Parameter stress R0 = 200e6;
@Parameter stress Hy = 40e6;
@Parameter real b = 100;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "UserDefined" {
      R : "R0 + Hy * (1 - exp(-b * p))",     // Yield radius
      dR_dp : "b * (R0 + Hy - R)"
    }
  }
};
```

Example of usage

8.2.4.6 Isotropic harderning rule based defined by points

The `Data` isotropic hardening rule allows the user to define an isotropic hardening rule using a curve defined by a set of pairs of equivalent strain and equivalent stress.

This isotropic hardening rule can be parametrised using three entries:

- `values`: which must a dictionnary giving the value of the yield surface radius as a function of the equivalent plastic strain.
- `interpolation`: which allows to select the interpolation type. Possible values are `linear` (default choice) and `cubic_spline`.
- `extrapolation`: which allows to select the extrapolation type. Possible values are `bound_to_last_value` (or `constant`) and `extrapolation` (default choice).

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 150e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "Data" {
      values : {0 : 150e6, 1e-3 : 200e6, 2e-3 : 400e6},
      interpolation : "linear"
    }
  }
};
```

8.2.4.6.1 Example of usage

8.2.5 List of available kinematic hardening rules

8.2.5.1 The `Prager` kinematic hardening rule

8.2.5.1.1 Example The following code can be added in a block defining an inelastic flow:

```
    kinematic_hardening : "Prager" {C : 33e6},
```

8.2.5.2 The `Armstrong-Frederick` kinematic hardening rule

The `Armstrong-Frederick` kinematic hardening rule can be described as follows (see [28]):

$$\begin{cases} \underline{X} = \dfrac{2}{3} C \, \underline{a} \\ \dot{\underline{a}} = \dot{p} \, \underline{n} - D \, \dot{p} \, \underline{a} \end{cases}$$

8.2.5.2.1 Example The following code can be added in a block defining an inelastic flow:

```
    kinematic_hardening : "Armstrong-Frederick" {C : 1.5e9, D : 5}
```

8.2.5.3 The `Burlet-Cailletaud` kinematic hardening rule

The `Burlet-Cailletaud` kinematic hardening rule is defined as follows (see [29]):

$$\begin{cases} \underline{X} = \dfrac{2}{3} C \, \underline{a} \\ \dot{\underline{a}} = \dot{p} \, \underline{n} - \eta \, D \, \dot{p} \, \underline{a} - (1 - \eta) \, D \, \dfrac{2}{3} \, \dot{p} \, (\underline{a} : \underline{n}) \, \underline{n} \end{cases}$$

8.2.5.3.1 Example The following code can be added in a block defining an inelastic flow:

```
    kinematic_hardening : "Burlet-Cailletaud" {C : 250e7, D : 100, eta : 0}
```

8.2.5.4 The `Chaboche 2012` kinematic hardening rule

The `Chaboche 2012` kinematic hardening rule is defined as follows (see [30]):

$$\dot{\underline{a}} = \dot{\underline{\varepsilon}}^p - \frac{3 \, D}{2 \, C} \, \Phi \left(p \right) \, \Psi^{(\underline{X})} \left(\underline{X} \right) \, \dot{p} \, \underline{X} = \dot{\underline{\varepsilon}}^p - D \, \Phi \left(p \right) \, \Psi \left(\underline{a} \right) \dot{p} \, \underline{a}$$

with:

- $\underline{X} = \frac{2}{3} \, C \, \underline{a}$
- $\Phi \left(p \right) = \phi_\infty + \left(1 - \phi_\infty \right) \exp \left(- b \, p \right)$
- $\Psi^{(\underline{X})} \left(\underline{X} \right) = \frac{\left\langle D \, J(\underline{X}) - \omega \, C \right\rangle^m}{1 - \omega} \, \frac{1}{\left(D \, J(\underline{X}) \right)^m}$
- $\Psi \left(\underline{a} \right) = \frac{\left\langle D \, J(\underline{a}) - \frac{3}{2} \omega \right\rangle^m}{1 - \omega} \, \frac{1}{\left(D \, J(\underline{a}) \right)^m}$

8.2.5.4.1 Example The following code can be added in a block defining an inelastic flow:

```
    kinematic_hardening : "Chaboche 2012" {
      C : 250e7,
      D : 100,
      m : 2,
      w : 0.6,
    }
```

Chapter 9

Extension of the StandardElastoViscoPlasticity brick to porous materials

9.1 Introduction

Constitutive equations for porous (visco-)plasticity are used to perform ductile fracture simulations, adding the porosity as an internal state variable. Porosity evolution is driven by nucleation laws and growth due to plastic flow.

While some implementations of these models based on implicit schemes are already available in `MFront` (Gurson [31], Gurson-Tvergaard-Needleman [32], ...), their numerical treatment have specific issues:

- Phenomenological laws describing porosity nucleation may have threshold to be activated or inhibited.
- Material failure usually happens when the porosity reaches a critical value corresponding to the collapse of the yield surface. Detection of material failure is awkward when the equations governing the porosity evolution are included in the implicit system to be solved. For example, the estimates of the solution may exceed the critical value during the iterations while the solution of the implicit scheme may be below this critical value.

Those issues were previously overcome by considering an explicit treatment of the porosity evolution, which requires small time steps.

The purpose of this work is twofold:

- Propose a robust fully implicit algorithm which can handle thresholds of nucleation laws and material failure.
- Extend the `StandardElastoViscoPlasticity` brick to this class of behaviours, allowing a declarative syntax similar as:

```
@Brick "StandardElastoViscoPlasticity" {
   stress_potential : "Hooke" {
      young_modulus : 200e9,
      poisson_ratio : 0.3
   },
   inelastic_flow : "Plastic" {
      criterion : "GursonTvergaardNeedleman" {
         q1 : 1.5,
         q2 : 1.0,
         q3 : 2.2,
         fc : 0.01,
         fr : 0.1
      },
      isotropic_hardening : "Linear" {
         R0 : 150e6
      }
```

```
   },
   porosity_evolution : {
      nucleation_model : "Chu_Needleman" {
         An : 0.01,
         pn : 0.1,
         sn : 0.1
      },
      growth_model : "StandardPlasticModel"
};
```

Note For backward compatibility, effect of the porosity on plastic flows will not be taken into account if no `porosity_evolution` section is declared.

This document is meant to be part of the MFront documentation. It will be available on the MFront website and on the ResearchGate page of the project. The authors have carefully checked that no unpublished material data has been used in the examples.

9.1.1 Outline

Section 9.2 provides a description of several stress criteria of interest for porous (visco-)plasticity and some common nucleation laws.

Section 9.3 describes two implicit schemes suitable for the integration of the previous constitutive laws:

- The first one is the standard implicit scheme where the implicit equations associated with all the state variables are solved all at once using a Newton-like algorithm.
- The second one is built on a staggered approach in which the porosity evolution and the evolution of the other state variables are solved independently. Both evolutions are repeated until the implicit equations associated with all the state variables are all satisfied. We show that, if required, an exact consistent tangent operator can be derived.

Section 9.4 discusses the design choices made for the extension of `MFront`' `StandardElastoViscoPlasticity` brick.

Section 9.5 is devoted to unit tests and verifications.

Section 9.6 describes early results that proves the robustness of the proposed schemes.

Appendix 9.8.1 is dedicated to the computation of the stress criterion which may require solving a non linear equation by Newton algorithm.

Appendix 9.8.2 gives the derivatives of some common nucleation laws.

Appendix 9.8.3 gives the derivatives of some common porous stress criteria.

Appendix 9.8.4 describes a regularization of the effective porosity in the Gurson-Tvergaard-Needleman stress criterion.

Appendix 9.8.5 details some numerical tricks to improve the robustness of finite element simulations performed with `Cast3M` using the constitutive laws from the extended `StandardElastoViscoPlasticity` brick.

Appendix 9.8.6 is dedicated to the description of the various options that can be declared in the `porosity_evolution` section inside the declaration of the `StandardElastoViscoPlasticity` brick.

9.2 General framework

In the framework adopted in this report, a porous plastic model is defined by a stress criterion $\sigma_\star$ which depends explicitly on the stress tensor $\underline{\sigma}$ and the porosity f:

$$\sigma_\star = \phi(\underline{\sigma}, f)$$

This stress criterion is used to define the yield surface or the intensity of the viscoplastic flow.

Concerning the flow rule, two main classes of models are classically found in the literature. In the first class of model, the porosity does not change the expression of the flow rule:

$$\underline{\dot{\varepsilon}}^{\mathrm{p}} = \dot{\lambda}\,\frac{\partial \sigma_\star}{\partial \underline{\sigma}} = \dot{p}\,\frac{\partial \sigma_\star}{\partial \underline{\sigma}}$$

i.e. the equivalent (visco-)plastic strain p is defined as the macroscopic equivalent (visco-)plastic strain and can still be identified as the plastic multiplier λ.

This class of models encompasses:

- the work of Ponte-Castaneda (see [33]), which is the theorical basis of several viscoplastic behaviours used in uranium dioxyde fuels [21, 23];
- the work of Rousselier [34].

In the second class, the flow rule is affected by a correction factor $1-f$ (see [35]):

$$\underline{\dot{\varepsilon}}^{\mathrm{p}} = \dot{\lambda}\,\frac{\partial \sigma_\star}{\partial \underline{\sigma}} = (1-f)\,\dot{p}\,\frac{\partial \sigma_\star}{\partial \underline{\sigma}}$$

where f is the porosity. Here, p can be identified to the equivalent (visco-)plastic strain in the matrix and it is no more equal to the plastic multiplier λ.

Note
The term $(1-f)$ comes from the definition of p as the equivalent plastic strain in the matrix through the modelling of hardening:

$$\underline{\sigma} : \underline{\dot{\varepsilon}}^{\mathrm{p}} = \dot{\lambda}\underline{\sigma} : \frac{\partial \sigma_\star}{\partial \underline{\sigma}} = \dot{\lambda}\sigma_\star = (1-f)R(p)\dot{p} \Rightarrow \dot{\lambda} = (1-f)\dot{p}$$

where $R(p)$ is the radius of the yield surface.

The evolution of porosity is composed of two terms: one due to plastic flow and another one due to nucleation $A_n^i dp$:

$$\dot{f} = (1-f)\mathrm{tr}(\underline{\dot{\varepsilon}}^{\mathrm{p}}) + \sum_j A_n^j \dot{p}$$

The first term describes the incompressibility of the matrix under the assumption that the change of volume associated with the elastic part of the strain is negligible. This assumption is common, although adding the elastic contribution is fairly easy, see Section 9.3.1.1.

9.2.1 Stress criteria

Different stress criteria implemented in the brick are described below.

- **Gurson-Tvergaard-Needleman** [32]

$$S = \left(\frac{\sigma_{vM}}{\sigma_\star}\right)^2 + 2q_1 f_\star \cosh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right) - 1 - q_3 f_\star^2 = 0 \tag{9.1}$$

which defines implicitly $\sigma_\star$, where σ_{vM} is the von Mises equivalent stress, and σ_m the mean stress. $f_\star$ is defined such that :

$$f_\star = \begin{cases} \delta f & \text{if } f < f_c \\ f_c + \delta(f - f_c) & \text{otherwise} \end{cases}$$

and :

$$\delta = \begin{cases} 1 & \text{if } f < f_c \\ \dfrac{f_u - f_c}{f_r - f_c} & \text{otherwise} \end{cases}$$

where f_u is the root of $2\,q_1\,f - 1 - q_3\,f^2$.

The parameters of the model are:

$$\{q_1, q_2, q_3, f_c, f_r\}$$

For $\{q_1, q_2, q_3, f_c, f_r\} = \{1, 1, 1, +\infty, -\}$, Gurson-Tvergaard-Needleman model reduces to the Gurson model [31] which has no free parameter.

- **Rousselier-Tanguy-Besson** [36]

$$S = \frac{\sigma_{vM}}{(1-f)\sigma_\star} + \frac{2}{3} f D_R \exp\left(\frac{3}{2} q_R \frac{\sigma_m}{(1-f)\sigma_\star}\right) - 1 = 0 \tag{9.2}$$

The effective stress $\sigma_\star$ which is solution of this equation is:

$$\sigma_\star = \frac{q_R \sigma_{vM}}{(1-f)\left(q_R - \frac{2}{3T} L\left(q_R D_R f T \exp\left(\frac{3}{2} q_R T\right)\right)\right)}$$

where L is the Lambert function, and $T = \sigma_m/\sigma_{vM}$ the stress triaxiality. The parameters of the model are:

$$\{q_R, D_R\}$$

9.2.2 Nucleation terms

Different nucleation terms are available and are described below. Multiple nucleation terms can be used simultaneously.

- **Chu-Needleman strain based** [37]

$$A_n = \frac{f_N}{s_N\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{p-\epsilon_N}{s_N}\right)^2\right) \tag{9.3}$$

where $\{f_N, s_N, \epsilon_N\}$ are material parameters, and p is the matrix equivalent plastic strain.

- **Chu-Needleman stress based** [37]

$$A_n = \frac{f_N}{s_N\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{\sigma_I-\sigma_N}{s_N}\right)^2\right)$$

where $\{f_N, s_N, \sigma_N\}$ are material parameters, and σ_I is the maximal positive principal stress.

- **Power-Law strain based**

$$A_n = f_N \left\langle \frac{p}{\epsilon_N} - 1 \right\rangle^m \quad \text{if} \quad \int A_n dp \leq f_{\text{nuc}}^{max} \tag{9.4}$$

where $\{f_N, m, \epsilon_N, f_{\text{nuc}}^{max}\}$ are material parameters, p is the matrix equivalent plastic strain. In the previous expression, $\langle . \rangle$ denotes the Macaulay bracket, i.e. positive part of a number:

$$\langle x \rangle = \max(x, 0)$$

- **Power-Law stress based**

$$A_n = f_N \left\langle \frac{\sigma_I}{\sigma_N} - 1 \right\rangle^m \quad \text{if} \quad \int A_n dp \leq f_{\text{nuc}}^{max}$$

where $\{f_N, m, \sigma_N, f_{\text{nuc}}^{max}\}$ are material parameters, and σ_I is the maximal positive principal stress.

9.3 Implicit schemes

This section is devoted to the presentation of two implicit schemes that may be used to integrate the constitutive equations presented in Section 9.2.

9.3.1 Standard implicit scheme

Assuming a single porous stress criterion whose flow is modified by the porosity evolution, the constitutive equations lead to the following system to be solved for an elastoplastic evolution:

$$\begin{aligned} \underline{\dot{\varepsilon}}^{\mathrm{el}} + \underline{\dot{\varepsilon}}^{\mathrm{p}} - \underline{\dot{\varepsilon}}^{\mathrm{to}} &= 0 \\ \sigma_\star - R(p) &= 0 \\ \dot{f} - (1-f)\mathrm{tr}(\underline{\dot{\varepsilon}}^{\mathrm{p}}) - \sum_j A_n^j \dot{p} &= 0 \end{aligned} \tag{9.5}$$

This system must be closed by adding the relation between the stress $\underline{\sigma}$ and the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$. In the report, we assume that the standard Hooke law holds:

$$\underline{\sigma} = \underline{\underline{\mathbf{D}}}\,\underline{\varepsilon}^{\mathrm{el}}$$

where $\underline{\underline{\mathbf{D}}}$ is the elastic stiffness tensor.

A semi-implicit method is used to solve these equations according to the variables $\{\Delta\,\underline{\varepsilon}^{\mathrm{el}}, \Delta\,p, \Delta\,f\}$

$$\mathcal{R}(\Delta\,\underline{\varepsilon}^{\mathrm{el}}, \Delta\,p, \Delta\,f) = 0 \tag{9.6}$$

The residual $\mathcal{R}$ can be decomposed by blocks as follows:

$$\begin{aligned} \mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}} &= \Delta\,\underline{\varepsilon}^{\mathrm{el}} + (1 - f|_{t+\theta\,\Delta\,t})\Delta\,p\,\left.\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right|_{t+\theta\,\Delta\,t} - \Delta\,\underline{\epsilon}_{to} = 0 \\ \mathcal{R}_{\Delta\,p} &= \sigma_\star - R(p|_{t+\theta\,\Delta\,t}) = 0 \\ \mathcal{R}_{\Delta\,f} &= \Delta\,f - (1 - f|_{t+\theta\,\Delta\,t})^2 \Delta\,p\,\mathrm{tr}\left(\left.\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right|_{t+\theta\,\Delta\,t}\right) - \sum_j \left. A_n^j\right|_{t+\theta\,\Delta\,t} \Delta\,p = 0 \end{aligned} \tag{9.7}$$

In the previous equation, θ is a numerical parameter ($\theta \in [0:1]$).

Solving this system using a Newton-Raphson algorithm requires computing the Jacobian matrix:

$$J = \begin{pmatrix} \dfrac{\partial \mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial \mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\,p} & \dfrac{\partial \mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta\,f} \\ \dfrac{\partial \mathcal{R}_{\Delta\,p}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial \mathcal{R}_{\Delta\,p}}{\partial \Delta\,p} & \dfrac{\partial \mathcal{R}_{\Delta\,p}}{\partial \Delta\,f} \\ \dfrac{\partial \mathcal{R}_{\Delta\,f}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial \mathcal{R}_{\Delta\,f}}{\partial \Delta\,p} & \dfrac{\partial \mathcal{R}_{\Delta\,f}}{\partial \Delta\,f} \end{pmatrix} \tag{9.8}$$

where the different terms can be written as [35]:

$$\frac{\partial \mathcal{R}_{\Delta \underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta \underline{\varepsilon}^{\mathrm{el}}} = \underline{\underline{\mathbf{I}}} + \theta \Delta p \left(1 - f|_{t+\theta\Delta t}\right) \left.\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma}^2}\right|_{t+\theta\Delta t} : \underline{\underline{\mathbf{D}}}|_{t+\theta\Delta t}$$

$$\frac{\partial \mathcal{R}_{\Delta \underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta p} = \left(1 - f|_{t+\theta\Delta t}\right) \left.\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right|_{t+\theta\Delta t}$$

$$\frac{\partial \mathcal{R}_{\Delta \underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta f} = \theta \Delta p \left(\left(1 - f|_{t+\theta\Delta t}\right) \left.\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma} \partial f}\right|_{t+\theta\Delta t} - \left.\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right|_{t+\theta\Delta t}\right)$$

$$\frac{\partial \mathcal{R}_{\Delta p}}{\partial \Delta \underline{\varepsilon}^{\mathrm{el}}} = \theta \left.\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right|_{t+\theta\Delta t} : \underline{\underline{\mathbf{D}}}|_{t+\theta\Delta t}$$

$$\frac{\partial \mathcal{R}_{\Delta p}}{\partial \Delta p} = -\theta \frac{dR(p)}{dp}$$

$$\frac{\partial \mathcal{R}_{\Delta p}}{\partial \Delta f} = \theta \left.\frac{\partial \sigma_\star}{\partial f}\right|_{t+\theta\Delta t}$$

$$\frac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta \underline{\varepsilon}^{\mathrm{el}}} = -\theta \Delta p \left(1 - f|_{t+\theta\Delta t}\right)^2 \left(\left.\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma}^2}\right|_{t+\theta\Delta t} : \underline{\underline{\mathbf{D}}}|_{t+\theta\Delta t}\right) : \underline{I} - \theta \Delta p \sum_j \left.\frac{\partial A_n^j}{\partial \underline{\sigma}}\right|_{t+\theta\Delta t} : \underline{\underline{\mathbf{D}}}|_{t+\theta\Delta t}$$

$$\frac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta p} = -\left(1 - f|_{t+\theta\Delta t}\right)^2 \left.\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right|_{t+\theta\Delta t} : \underline{I} - \theta \Delta p \sum_j \left.\frac{\partial A_n^j}{\partial p}\right|_{t+\theta\Delta t} - \sum_j A_n^j|_{t+\theta\Delta t}$$

$$\frac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta f} = 1 - \theta \Delta p \left(1 - f|_{t+\theta\Delta t}\right)\left(-2 \frac{\partial \sigma_\star}{\partial \underline{\sigma}} + \left(1 - f|_{t+\theta\Delta t}\right) \frac{\partial^2 \sigma_\star}{\partial \underline{\sigma} \partial f}\right)\underline{I} - \theta \Delta p \sum_j \left.\frac{\partial A_n^j}{\partial f}\right|_{t+\theta\Delta t}$$

Therefore, the definition of a porous model requires to give the following expressions for the stress criterion $\sigma_\star$:

$$\left\{\sigma_\star, \frac{\partial \sigma_\star}{\partial \underline{\sigma}}, \frac{\partial \sigma_\star}{\partial f}, \frac{\partial^2 \sigma_\star}{\partial \underline{\sigma}^2}, \frac{\partial^2 \sigma_\star}{\partial \underline{\sigma} \partial f}\right\}$$

as well as the following expressions for each nucleation mechanism:

$$\left\{A_n^i, \frac{\partial A_n^i}{\partial \underline{\sigma}}, \frac{\partial A_n^i}{\partial p}, \frac{\partial A_n^i}{\partial f}\right\}$$

These expressions are detailed for the stress criteria and nucleation formulas currently implemented into the brick in Appendixes 9.8.2 and 9.8.3.

9.3.1.1 Taking the elastic contribution to the porosity growth into account

Adding the elastic contribution to the porosity growth is trivially performed by substracting the following term to the implicit equation associated with the porosity evolution:

$$\mathcal{R}_{\Delta f} \mathrel{-}= \left(1 - f|_{t+\theta\Delta t}\right) \mathrm{tr}(\Delta \underline{\varepsilon}^{\mathrm{el}})$$

The following term must be added to the block $\dfrac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta f}$:

$$\frac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta f} \mathrel{+}= \theta \, \mathrm{tr}(\Delta \underline{\varepsilon}^{\mathrm{el}})$$

And the following term must be added to the block $\dfrac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta \underline{\varepsilon}^{\mathrm{el}}}$:

$$\frac{\partial \mathcal{R}_{\Delta f}}{\partial \Delta \underline{\varepsilon}^{\mathrm{el}}} \mathrel{+}= -\left(1 - f|_{t+\theta\Delta t}\right) \underline{I}$$

9.3.1.2 Special cases for the treatment of the porosity

In the absence of nucleation models and without elastic contribution, the evolution of the porosity is given by the following equation:

$$\dot{f} = (1-f)^{n_f}\,\dot{p}\,\mathrm{tr}(\underline{n}) \tag{9.9}$$

where n_f is equal to:

- 2 if the porosity evolution affects the (visco-)plastic flow
- 1 if this is not the case.

9.3.1.2.1 First case ($n_f = 1$)

Assuming $n_f = 1$, Equation (9.9) can be integrated by parts:

$$\int_{f|_t}^{f|_{t+\Delta t}} \frac{\mathrm{d}\,u}{1-u} = \log\left(\frac{1-f|_t}{1-f|_{t+\Delta t}}\right) = \int_t^{t+\Delta t} \dot{p}\,\mathrm{tr}(\underline{n})\,\mathrm{d}\,t \approx \Delta\,p\,\mathrm{tr}\left(\underline{n}|_{t+\theta\,\Delta\,t}\right)$$

which leads to the following expression of the increment of the porosity:

$$\Delta\,f = (1-f|_t)\left(1-\exp\left(-\Delta\,p\,\mathrm{tr}\left(\underline{n}|_{t+\theta\,\Delta\,t}\right)\right)\right)$$

Taking into account the elastic prediction
When taking into account the elastic prediction, the increment of the porosity can be known prior any integration:

$$\Delta\,f = (1-f|_t)\,(1-\exp\left(\mathrm{tr}(-\Delta\,\underline{\varepsilon}^{\mathrm{to}})\right)) \tag{9.10}$$

When applicable, Equation (9.10), could be very interesting because one can predict the material failure before the material integration. Moreover, removing the evolution of the porosity equation greatly simplifies the resolution of the implicit system.
However, one drawback of this relation is that it introduces the porosity as an implicit function of the increment of the total strain which:

- is incompatible with the assumptions of the `StandardElastoViscoPlasticity` brick. It thus won't be treated further in this report;
- complexifies the computation of the consistent tangent operator. In particular, Equation (9.18), detailed below, is no more valid. However, recent developments of `MFront` could simplify the computation of the tangent operator in this case, see [38] for details.

Another drawback is that the user would not be able to add its own contribution to the porosity evolution as described in Section 9.4.2.0.1.

9.3.1.2.2 Second case ($n_f = 2$)

Assuming $n_f = 2$, Equation (9.9) can be integrated by parts:

$$\int_{f|_t}^{f|_{t+\Delta t}} \frac{\mathrm{d}\,u}{(1-u)^2} = \frac{1}{1-f|_t} - \frac{1}{1-f|_{t+\Delta t}} = \int_t^{t+\Delta t} \dot{p}\,\mathrm{tr}(\underline{n})\,\mathrm{d}\,t \approx \Delta\,p\,\mathrm{tr}\left(\underline{n}|_{t+\theta\,\Delta\,t}\right)$$

which leads to the following expression of the increment of the porosity:

$$\Delta\,f = (1-f|_t)^2\,\frac{\Delta\,p\,\mathrm{tr}\left(\underline{n}|_{t+\theta\,\Delta\,t}\right)}{1+(1-f|_t)\,\Delta\,p\,\mathrm{tr}\left(\underline{n}|_{t+\theta\,\Delta\,t}\right)}$$

9.3.1.3 Treatment of bounds on nucleated porosity

Most nucleation models require to set an upper bound $f_{\mathrm{nuc}}^{j,\max}$ on the maximum nucleated porosity predicted by those models. In this case, the effective nucleated porosity increment is defined as:

$$\Delta\,f_n^j = \min\left(A_n^j\Big|_{t+\theta\,\Delta\,t}\Delta\,p,\; f_{\mathrm{nuc}}^{j,\max} - f_n^j\Big|_t\right)$$

This treatment requires that the value of the nucleated porosity at the beginning of the time step is known. Hence it is automatically stored in an auxiliary state variable whatever the value of the `save_porosity_increase` option passed to the nucleation model.

9.3.1.4 Specific treatment of strain-based nucleation models

A strain-based nucleation model only depends on the equivalent plastic strain as follows:

$$\frac{\mathrm{d}f_n^j}{\mathrm{d}t} = A_n^j(p)\,\frac{\mathrm{d}p}{\mathrm{d}t} \tag{9.11}$$

See the strain version of the Chu-Needleman nucleation model (Equation (9.3)) and the strain version of the power-law nucleation model (Equation (9.4)).

In Equation (9.7), the contribution of this nucleation model to the implicit equation was set equal to:

$$\Delta\, f_n^j \approx A_n^j\Big(p|_{t+\theta\,\Delta\,t}\Big)\,\Delta\, p$$

However, this is only an approximation.

A more precise estimation can be obtained by integrating Equation (9.11) by parts, which leads to the following expression of the porosity increment:

$$\Delta\, f_n^j = F_n^j\Big(p|_{t+\Delta\,t}\Big) - F_n^j\big(p|_t\big) \tag{9.12}$$

where F_n^j is one primitive of A, i.e. $F_n^j(p) = \int A_n^j(u)\,\mathrm{d}\,u$. Such a treatment of the nucleated porosity was only considered in [39].

When the function A also depends on some external state variables, (temperature, fluence), we use the value of those external state variables at the middle of the time step to evaluate the increment of the porosity using Equation (9.12).

9.3.1.5 Shortcomings of this implicit scheme

The Newton-Raphson algorithm may fail at finding the solution of the non-linear system of equations, especially when the time step leads to strong increase of the porosity.

Moreover, the porosity evolution leads to specific issues:

- Phenomenological laws describing porosity nucleation may have thresholds to be activated or inhibited.
- Material failure usually happens when the porosity reaches a critical value corresponding to the collapse of the yield surface. Detection of material failure is awkward when the equations governing the porosity evolution are included in the implicit system to be solved. For example, the estimates of the solution may exceed the critical value during the iterations while the solution of the implicit scheme may be below this critical value.

9.3.2 A staggered approach

As the main convergence issues of the standard implicit scheme are associated with the porosity evolution, we propose in this section a staggered approach where the porosity evolution and the evolution of the other state variables (i.e. the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$ and the equivalent plastic strain p) are decoupled. A fixed point algorithm is used to ensure that the increments $\{\Delta\,\underline{\varepsilon}^{\mathrm{el}}, \Delta\, p, \Delta\, f\}$ satisfies the Implicit System (9.7).

9.3.2.1 Reduced implicit system

Let i be the current step of the fixed point algorithm. The estimates of the increments of the elastic strain and the equivalent plastic strain at the next iterations, denoted respectively $\Delta\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}$ and $\Delta\, p_{(i+1)}$ satisfies a reduced implicit system:

$$\begin{aligned}
\mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}} &= \Delta\,\underline{\varepsilon}^{\mathrm{el}}_{(i+1)} + \big(1-f_{(i)}\big)\,\Delta\, p_{(i+1)}\,\frac{\partial\sigma_\star}{\partial\underline{\sigma}}\Big(\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}, f_{(i)}\Big) - \Delta\,\underline{\epsilon}^{to} = 0\\
\mathcal{R}_{\Delta\, p} &= \sigma_\star\Big(\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}, f_{(i)}\Big) - R(p_{(i+1)}) = 0
\end{aligned} \tag{9.13}$$

where:

- $\underline{\varepsilon}^{\mathrm{el}}_{(i+1)} = \underline{\varepsilon}^{\mathrm{el}}\big|_t + \theta\,\Delta\,\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}$
- $p_{(i+1)} = p|_t + \theta\,\Delta\,p_{(i+1)}$
- $f_{(i)} = f|_t + \theta\,\Delta\,f_{(i)}$

The Implicit System (9.13) is solved by a standard Newton-Raphson method with the reduced jacobian matrix:

$$\begin{bmatrix} \dfrac{\partial \mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}}}{\partial\,\Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial \mathcal{R}_{\Delta\,\underline{\varepsilon}^{\mathrm{el}}}}{\partial\,\Delta\,p} \\ \dfrac{\partial \mathcal{R}_{\Delta\,p}}{\partial\,\Delta\,\underline{\varepsilon}^{\mathrm{el}}} & \dfrac{\partial \mathcal{R}_{\Delta\,p}}{\partial\,\Delta\,p} \end{bmatrix} \tag{9.14}$$

Note
In practice, for reasons detailed in Section 9.3.2.10, we still solve the full Implicit System (9.7), but the implicit equation is modified as follows:

$$\mathcal{R}_{\Delta\,f} = \Delta\,f - \Delta\,f_{(i)}$$

9.3.2.2 Iterative determination of the porosity increment

Letting the constraints on the porosity evolution aside, the porosity is determined iteratively as follows:

$$\Delta\,f^{(uc)}_{(i+1)} = \big(1 - f_{(i)}\big)^2\,\Delta\,p_{(i+1)}\,\mathrm{tr}\Big(\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\big(\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}, f_{(i)}\big)\Big) + \sum_j A^j_n\big(\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}, p_{(i+1)}, f_{(i)}\big)\Delta\,p_{(i+1)} \tag{9.15}$$

Here the $^{(uc)}$ superscript means "uncorrected." Corrections to $\Delta\,f^{(uc)}_{(i+1)}$ will be introduced in the next paragraphs to deal with thresholds in nucleation laws and detection of material failure.

9.3.2.3 Treatments of thresholds in nucleation laws

Let:

- $\Delta\,f^{j,(uc)}_n\big|_{(i+1)} = A^j_n\big(\underline{\varepsilon}^{\mathrm{el}}_{(i+1)}, p_{(i+1)}, f_{(i)}\big)\Delta\,p_{(i+1)}$ be the uncorrected contribution of the j^{th} nucleation law to the porosition evolution.
- $f^j_n\big|_t$ be the value of the nucleated porosity due to this nucleation law at the beginning of the time step.
- $f^{j,\max}_n$ be the threshold of this nucleation law, i.e. the maximal value of the nucleated porosity due to this law.

$f^{j,\max}_n - f^j_n\big|_t$ is the maximum allowed increment of the porosity for this nucleation law.

We then define the corrected contribution $\Delta\,f^j_n\big|_{(i+1)}$ as follows:

$$\Delta\,f^j_n\big|_{(i+1)} = \min\Big(\Delta\,f^{j,(uc)}_n\big|_{(i+1)}, f^{j,\max}_n - f^j_n\big|_t\Big)$$

9.3.2.4 Treatment of material failure

Note
To avoid the introduction of another notation, we still used $\Delta\,f^{(uc)}_{(i+1)}$ even though corrections related to thresholds in nucleations laws may have been taken into account at this stage.

If $f|_t + \Delta\,f^{(uc)}_{(i+1)}$ is greater than the critical porosity denoted $\alpha_1\,f_r$, a dichotomic approach is used:

$$\Delta\,f_{(i+1)} = \frac{1}{2}\big(f + \Delta\,f_{(i)} + \alpha_1\,f_r\big) - f|_t;$$

This approach allows the porosity to approach the critical porosity smoothly.

The α_1 factor is a user defined parameter, chosen equal by default to 98.5 %. The reason for not allowing the porosity be closer to f_r is than the Implicit System (9.13) becomes very difficult to solve as the yield surfaces collapses.

9.3.2.5 Stopping criteria

The iterations are stopped when the porosity becomes stationnary, i.e. when:

$$\left|\Delta f_{(i+1)}^{(uc)} - \Delta f_{(i)}\right| < \varepsilon_f \tag{9.16}$$

where the value of ε_f is a user defined (by default, a stringent value of 10^{-10} is used).

9.3.2.6 Acceleration algorithm

9.3.2.6.1 Aitken acceleration (default) The Iterative Process (9.15) can be accelerated using the well known Aitken transformation.

The usage of the Aitken acceleration is enabled by default.

9.3.2.6.2 Relaxation To avoid spurious oscillations, a relaxation procedure is used:

$$\Delta f_{(i+1)} = \omega \Delta f_{(i)} + (1-\omega) \Delta f_{(i+1)}^{(uc)}$$

where the relaxation coefficient is chosen equal to 0.5.

9.3.2.7 Detection of the material failure (post-processing)

Once the Stopping Condition (9.16) is satisfied, material failure is detected when the final porosity is above a given threshold $\alpha_2 f_r$, where α_2 is a user defined constant chosen equal to 98 % by default.

When the material failure is detected, the value of the auxiliary state variable `broken` is set to one.

9.3.2.8 Discussion on the performance of the staggered approach

From a performance point of view, this staggered approach seems *a priori* less efficient than the standard implicit scheme because the reduced integration scheme will be solved once per iteration of the fixed point algorithm.

The staggered approach is however thought to be more robust, although we do not have performed an in-depth comparison of both monolithic and staggered approaches yet.

It shall also be noted that the number of iterations of the reduced integration implicit system is not proportional to the number of iterations of the fixed point algorithm. Indeed, the solution of the reduced implicit system obtained at the previous iteration of the fixed point algorithm can be used as the starting point of the current resolution of the reduced implicit system. Numerical experiments shows that this strategy is very effective and that close to convergence of the fixed point algorithm the number of iterations required to solve the reduced implicit system is low, i.e. only 2 or 3 iterations are required.

9.3.2.9 Access to the number of the iterations of the staggered scheme

The variable `staggered_scheme_iteration_counter` holds the number of iterations of the staggered scheme. One can save this variable in a auxiliary state variable as follows:

```
//! number of iterations of the staggered scheme at convergence
@AuxiliaryStateVariable real niter;
niter.setEntryName("StaggeredSchemeIterationCounter");

....

@UpdateAuxiliaryStateVariables{
  niter = staggered_scheme_iteration_counter;
}
```

The total number of evaluations of the implicit scheme is a bit trickier to save. It can be done as follows:

```
//! number of iterations of the implicit scheme
@AuxiliaryStateVariable real neval;
neval.setEntryName("NumberOfEvaluationOfTheImplicitScheme");

...

@InitLocalVariables{
  ...
  neval = 0;
}

...

@Integrator{
 ++neval;
}
```

Note
If local sub-stepping is allowed, those two numbers will only reflect the number of iterations and number of evaluations of the last step.

9.3.2.10 Implementation details and computation of the exact consistent tangent operator

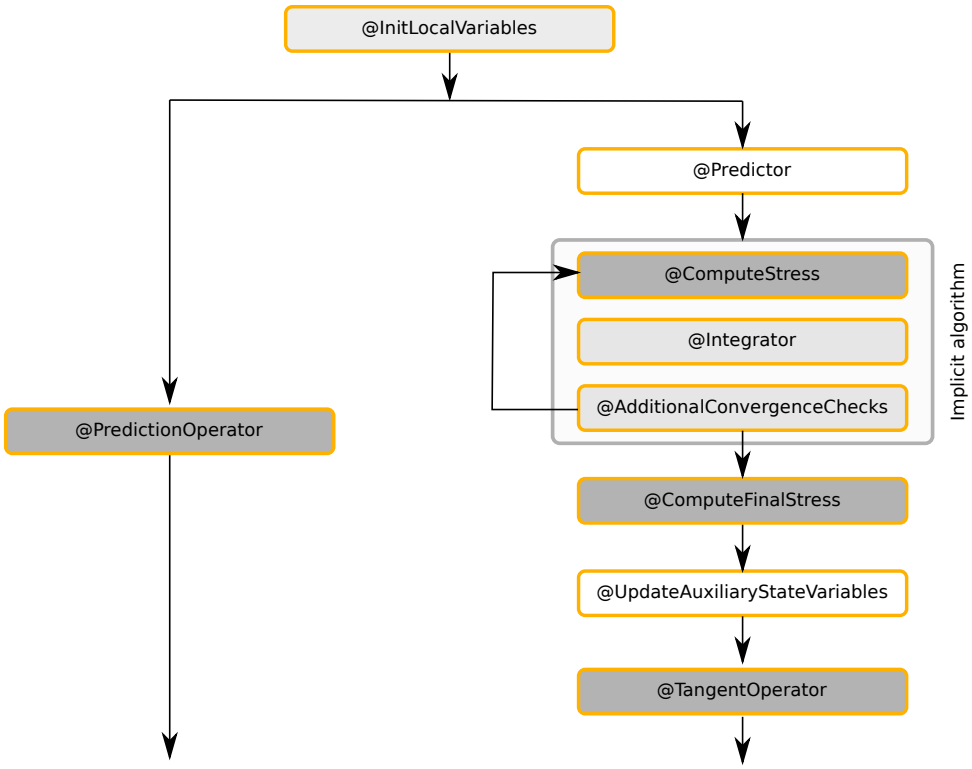

Figure 9.1: Description of the steps of the implicit scheme. Block in dark grey are automatically defined by the bricks and are not accessible by the user. Blocks in light grey can be completed by the user.

9.3.2.10.1 Implementation, the `@AdditionalConvergenceChecks` code block

As depicted on Figure 9.1, each iteration of the implicit algorithm are decomposed in three steps:

- Updating the stress in the `@ComputeStress` code block. This code block is automatically handled by the brick.
- Building the residual $\mathcal{R}$ and the jacobian matrix J in the `@Integrator` code block.
- Additional convergence checks in the `@AdditionalConvergenceChecks` code block. Additional here means that `MFront` has already checked the convergence of the resolution of implicit scheme using a built-in

criterion. This code block may be used to implement a status algorithm which allows activating and/or desactivating some (visco-)plastic flows (see [40]). Here, it is used to implement the staggered approach described in this section.

9.3.2.10.2 Computation of the exact consistent tangent operator As described in [38], the consistent tangent operator may be deduced for the jacobian of Implicit System (9.7) as follows:

$$\frac{\mathrm{d}\underline{\sigma}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = \underline{\underline{\mathbf{D}}}\Big|_{t+\Delta t} \cdot \frac{\partial \underline{\varepsilon}^{\mathrm{el}}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}} \tag{9.17}$$

where $\dfrac{\partial \underline{\varepsilon}^{\mathrm{el}}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ can be determined by solving the following systems of linear equations:

$$J \cdot \frac{\partial \underline{\varepsilon}^{\mathrm{el}}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}} = -\frac{\partial \mathcal{R}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$$

where $\dfrac{\partial \mathcal{R}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ have this simple matrix form (in $3D$):

$$\frac{\partial \mathcal{R}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}} = -\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{9.18}$$

It would cumbersome to compute the jacobian of the Reduced System (9.13) during the resolution and then to compute the jacobian of the Full System (9.7) after convergence for the computation of the consistent tangent operator:

- it would require a significant amount of boiler plate code to define the jacobian matrix for the full implicit system and appropriate views of its blocks (see Equation (9.8));
- it would require a partial duplication of the code computing the blocks of the jacobian of the reduced implicit system (See Equations (9.8) and (9.14)).

We thus rely on a simple trick. We define a flag which is set to true during the iterations of the fixed point algorithm. If this flag is true, the implicit equation of the Full Implicit System (9.7) is replaced by:

$$\mathcal{R}_{\Delta f} = \Delta f - \Delta f_{(i)}$$

where $\Delta f_{(i)}$ is the current estimation given by the fixed point algorithm.

This new equation is trivially solved so that the solutions of this modified system do verify the Reduced Implicit System (9.13). Once the fixed point algorithm has converged, one may simply set the flag to false and build the jacobian matrix (9.8) of the Full Implicit System (9.7) by calling the assembly of the implicit scheme once and finally apply Equation (9.17).

A few things may be noted:

- the overhead implied by this trick seems *a priori* limited;
- one can easily recover the standard implicit scheme;
- one can easily choose to treat the porosity explicitly.

9.4 Design choices

9.4.1 Enrichment of the `StressCriterion` and `InelasticFlow` interfaces

The `InelasticFlow` interface is used by the `StandardElastoViscoPlasticity` brick to handle inelastic flows. It uses the `StressCriterion` interface to describe it stress criterion and, for non-associated flows, its flow criterion.

For the brick to properly treat inelastic flows coupled with porosity, a new method called `isCoupledWithPorosityEvolution` has been added to those interface. This method returns a boolean value and does not take any argument.

For inelastic flows, the default implementation of the `isCoupledWithPorosityEvolution`, declared in the `InelasticFlowBase` class, returns `true` if its stress criterion or if its flow criterion (if any) declares to be coupled with the porosity evolution.

The `isCoupledWithPorosityEvolution` of a stress criterion must return `true` if and only if the expression of the stress criterion explicitly depends on the porosity.

> **Note**
> Implicitly, the brick expects that a stress criterion coupled with porosity is **not** deviatoric, i.e., that this stress criterion contributes to the porosity growth. The `isNormalDeviatoric` method of such a stress criterion must return `true`. This is checked in the `InelasticFlowBase` class.

As discussed in the introduction, a stress criterion can lead to a correction of the flow rule by a factor $1 - f$. The class `StressCriterion` now has a method called `getPorosityEffectOnEquivalentPlasticStrain` which return one of the two following values:

- `NO_POROSITY_EFFECT_ON_EQUIVALENT_PLASTIC_STRAIN`
- `STANDARD_POROSITY_CORRECTION_ON_EQUIVALENT_PLASTIC_STRAIN`

9.4.2 Conditions for the brick to contribute to the porosity evolution

The evolution of the porosity is taken into account:

- if a nucleation model is declared;
- if one stress inelastic flow declares being coupled with the porosity evolution, i.e. that its stress criterion or its flow criterion is coupled with the porosity evolution;
- if the `porosity_evolution` section is explicitly declared, even if empty.

For example, this declaration will automatically lead to a porosity growth:

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {
    young_modulus : 150.e3,
    poisson_ratio : 0.3
  },
  inelastic_flow : "Plastic" {
    criterion : "Gurson1975" {},
    isotropic_hardening : "Linear" {R0 : "0"}
  }
};
```

Note
It is important to note that, by default (if none of the three above conditions are met), no contribution to the porosity growth will be taken into account, even if the flow direction has an hydrostatic component. For example, the following declaration will not lead to any porosity growth:

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {
    young_modulus : 150.e3,
    poisson_ratio : 0.3
  },
  inelastic_flow : "Plastic" {
    criterion : "MohrCoulomb" {
      c : 3.e1,      // cohesion
      phi : 0.523598775598299,    // friction angle angle
      lodeT : 0.506145483078356,  // transition angle (Abbo and Sloan)
      a : 1e1       // tension cuff-off parameter
    },
    isotropic_hardening : "Linear" {R0 : "0"}
  }
};
```

However, this declaration will lead to a porosity growth:

```
@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {
    young_modulus : 150.e3,
    poisson_ratio : 0.3
  },
  inelastic_flow : "Plastic" {
    criterion : "MohrCoulomb" {
      c : 3.e1,      // cohesion
      phi : 0.523598775598299,    // friction angle angle
      lodeT : 0.506145483078356,  // transition angle (Abbo and Sloan)
      a : 1e1       // tension cuff-off parameter
    },
    isotropic_hardening : "Linear" {R0 : "0"}
  }
  porosity_evolution: {}
};
```

The `porosity_evolution` section has a significant importance in this document. The various options that can be declared in this section are described in Appendix 9.8.6.

9.4.2.0.1 Declaration of the porosity as a state variable (if required)

If the evolution of the porosity is taken into account, a state variable called `f` with the glossary name `Porosity` is automatically declared if no state variable with this glossary name has been declared. In the latter case, this variable is used (and updated) by the brick.

This choice allows the user to easily add new nucleation model or to add a new inelastic flow coupled with porosity by adding the appropriate state variables and adding the associated implicit equations in the `@Integrator` code block.

For example, assuming that the staggered approach described in Section 9.3.2 is used, one may add the contribution of the solid swelling as follows:

```
@ExternalStateVariable real ss;
ss.setGlossaryName("SolidSwelling");

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {
    young_modulus : 150.e3,
    poisson_ratio : 0.3
  },
```

```
  inelastic_flow : "Plastic" {
    criterion : "Gurson1975" {},
    isotropic_hardening : "Linear" {R0 : "0"}
  }
};

@Integrator{
  // adding the contribution of the solid swelling
  // the porosity increase in the implicit equation describing
  // the porosity.
  // This shall only been done after the convergence of the
  // staggered scheme has converged.
  if(compute_standard_system_of_implicit_equations){
    ff -= ;
  }
}

@AdditionalConvergenceChecks {
  // adding the contribution of the solid swelling
  // the porosity increase to the next extimate of the porosity by
  // the staggered scheme.
  // This shall only been done after the convergence of the
  // staggered scheme has converged.
  if (converged && (!compute_standard_system_of_implicit_equations)) {
    next_estimate_of_the_porosity_increment += ...
  }
}
```

The previous code uses variables that are automatically defined by the `StandardElastoViscoPlasticity` brick when the staggered approach is used:

- `compute_standard_system_of_implicit_equations` is a boolean value which is false until the staggered scheme converges;
- `next_estimate_of_the_porosity_increment` is a variable only accessible in the `@AdditionalConvergenceChecks` block which must hold the next estimate of the porosity increment. This variable is automatically initialized to zero.

Other variables automatically defined by the `StandardElastoViscoPlasticity` brick may be helpful to couple constitutive equations with user defined ones:

- in the `@InitLocalVariables` code block, the `upper_bound_of_the_porosity` variable may be updated by the user to change the value of the upper bound of the porosity as follows:

```
upper_bound_of_the_porosity = min(upper_bound_of_the_porosity,
                                  .... // user_defined_bound
                                  );
```

- in the `@AdditionalConvergenceChecks` code block, the `fixed_point_converged` boolean variable might be used add other convergence criteria to the staggered algorithm, as follows:

```
if (converged && (!compute_standard_system_of_implicit_equations)) {
  fixed_point_converged = ....
}
```

9.4.3 Effect of the porosity on the equivalent strain definition

By default, the value returned by the `getPorosityEffectOnEquivalentPlasticStrain` method of the stress criterion is used to decipher if the porosity affects definition of the equivalent plastic strain and the flow rule.

However, this may be changed by specifying the `porosity_effect_on_equivalent_plastic_strain` option in the definition of the inelastic flow. Valid strings are `StandardPorosityEffect` (or equivalently " `standard_porosity_effect`) or `None` (or equivalently `none` or `false`).

9.4.4 Saving individual contributions to the porosity evolutions in dedicated auxiliary state variables

The porosity evolves either due to nucleation or growth. For post-processing reasons, it may be worth to distinguish those contributions in dedicated auxiliary state variables. However, for performance reasons (mostly because additional auxiliary state variables implied a memory overhead), this shall not be the default choice.

The brick allows the user to specify if those additional variables shall be defined for all mechanisms in the `porosity- _evolution` section as follows:

```
porosity_evolution: {
  save_individual_porosity_increase: true
}
```

However, this behaviour can be overloaded in every inelastic flow section and every nucleation model by setting the `save_porosity_increase` boolean variable.

For example, one may use the following declaration:

```
inelastic_flow : "Plasticity" {
  criterion : "Gurson1975",
  isotropic_hardening : "Voce" {R0 : 200, Rinf : 100, b : 20},
  save_porosity_increase : true
}
```

> **Note**
> Inelastic flows which declare that the flow is deviatoric discard those options and never declare an auxiliary state variable associated with the porosity growth.

9.4.5 Taking into account the elastic contribution in the porosity growth

By default, the elastic contribution in the porosity growth is not taken into account.

This could be changed by setting the `elastic_contribution` options to true as follows:

```
 porosity_evolution: {
  elastic_contribution: true
}
```

9.5 Verifications

The stress criteria implemented are verified by comparing the results obtained through `MTest` simulations to reference solutions. Axisymmetric loading conditions are considered under small strains settings:

$$\underline{\sigma} = \sigma \begin{pmatrix} 1 & 0 & 0 \\ 0 & A & 0 \\ 0 & 0 & A \end{pmatrix}$$

A typical `MTest` file used for these simulations is shown below:

```
@ModellingHypothesis 'Tridimensional';
@Behaviour<Generic> 'src/libBehaviour.so' 'GTN';
@InternalStateVariable 'Porosity' 1e-3;
@ExternalStateVariable 'Temperature' 293.15 ;

@Real 'A' '0.4';
@NonLinearConstraint<Stress> 'SYY - A*SXX';
@NonLinearConstraint<Stress> 'SZZ - A*SXX';

@ImposedStrain 'EXX' {0 : 0, 1 : 0.5};
@Times{0, 1. in 1000};
```

Without hardening, the differential equations governing the evolutions of internal state variables are, for the GTN model:

$$\frac{d\varepsilon^p_{yy}}{d\varepsilon^p_{xx}} = \frac{q_1 q_2 f_\star \sinh\left(\frac{1}{2}q_2(1+2\alpha)\frac{\sigma}{\sigma_0}\right) + \frac{\sigma}{\sigma_0}(\alpha-1)}{q_1 q_2 f_\star \sinh\left(\frac{1}{2}q_2(1+2\alpha)\frac{\sigma}{\sigma_0}\right) + 2\frac{\sigma}{\sigma_0}(1-\alpha)} \qquad \text{with} \qquad S(\sigma, f) = 0$$

$$\frac{df}{d\varepsilon^p_{xx}} = (1-f)\left(1 + 2\frac{d\varepsilon^p_{yy}}{d\varepsilon^p_{xx}}\right)$$

The same equations also hold for the Gurson model with $(q_1, q_2, f_\star) = (1, 1, f)$. For the Rousselier model, the equations are:

$$\frac{d\varepsilon^p_{yy}}{d\varepsilon^p_{xx}} = \frac{D_R q_R f \exp\left(\frac{1}{2}\frac{q_R}{1-f}(1+2\alpha)\frac{\sigma}{\sigma_0}\right) - 1.5}{D_R q_R f \exp\left(\frac{1}{2}\frac{q_R}{1-f}(1+2\alpha)\frac{\sigma}{\sigma_0}\right) + 3} \qquad \text{with} \qquad S(\sigma, f) = 0$$

$$\frac{df}{d\varepsilon^p_{xx}} = (1-f)\left(1 + 2\frac{d\varepsilon^p_{yy}}{d\varepsilon^p_{xx}}\right)$$

These differential equations are solved with respect to ε^p_{xx} using the `ode45` function of the `Matlab` software, and the stress magnitude σ is computed by solving the stress criteria. Elastic strains are finally added to the plastic strains using Hooke's law to obtain total strains. These solutions, that can be obtained for arbitrary precision, are the reference solutions to which the results from `MTest` simulations are compared.

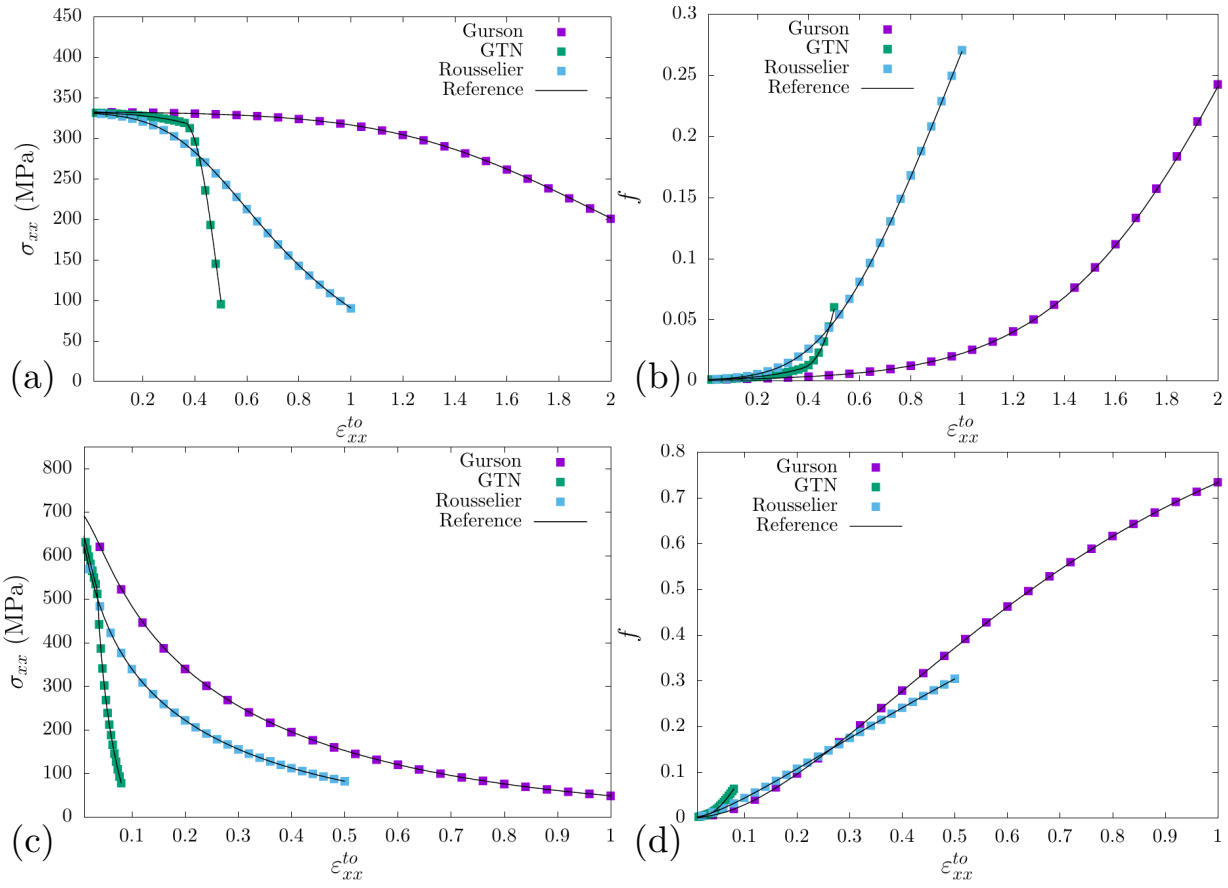

Figure 9.2: Comparisons between the reference solutions and the `MTest` simulations. The parameters used are: $q_1 = 2$, $q_2 = 1$, $q_3 = 4$, $f_c = 0.01$, $f_r = 0.10$ for the GTN model, $q_R = 1$, $D_R = 2$ for the Rousselier model. A is set to (a,b) 0.4 and (c,d) 0.7273, corresponding to stress triaxialities of 1 and 3, respectively. In all cases, $\sigma_0 = 200$MPa, $E = 200$GPa, $\nu = 0.3$ and $f_0 = 0.001$

A perfect agreement is obtained between the reference solutions and the `MTest` results, as depicted on Figure 9.2, validating the implementation of the equations of these models. Finally, the jacobians of all models have been verified with respect to the numerical jacobian.

9.6 Applications and Finite Element simulations

9.6.1 Axisymmetric tensile tests simulations

In this scope, the `StandardElastoViscoPlasticity` brick is tested using the finite elements software `Cast3M`. The Gurson-Tvergaard-Needleman stress criterion is used to model the ductile failure of Notched Tensile (NT) Samples and Simple Tensile (ST) ones. The $2D$ meshes are shown on Figure 9.3 .

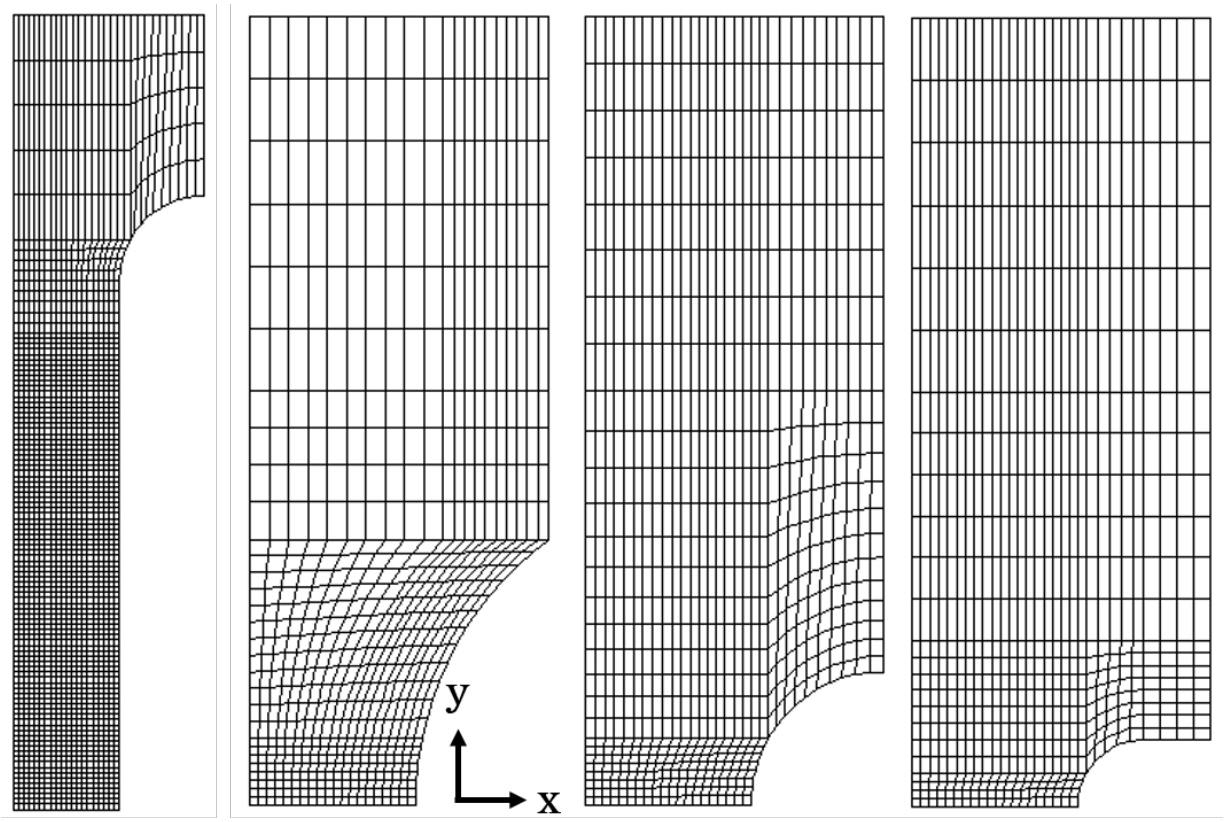

Figure 9.3: Meshes of the Notched Tensile (NT) Samples and Simple Tensile (ST) ones

The material's behavior is adapted to aluminum alloys from which the vessel of Fast Neutron Research Reactors are fabricated. The element size is chosen to be $100\,\mu m$ based on metallurgical studies that take into account the importance of the nucleation phase during damage. The nucleation is modeled by a Stress-based law that is implemented as follows [41] :

$$A_n = f_N \left\langle \frac{\sigma_I}{\sigma_N} - 1 \right\rangle^m \quad \text{if} \quad p \geq p_N, \quad \text{and} \quad \int A_n dp \leq f_{\text{nuc}}^{max}$$

where $\{f_N, p_N, m, \sigma_N, f_{\text{nuc}}^{max}\}$ are material parameters, and σ_I is the maximum positive principal stress.

Material's Behavior

```
@Brick StandardElastoViscoPlasticity{
 stress_potential : "Hooke" {young_modulus : 70e3, poisson_ratio : 0.3},
 inelastic_flow : "Plastic" {
   criterion : "GursonTvergaardNeedleman1982" {
     f_c : 0.04,
     f_r : 0.056,
     q_1 : 2.,
     q_2 : 1.,
     q_3 : 4.
   },
   isotropic_hardening : "Linear" {R0 : 274},
   isotropic_hardening : "Voce" {R0 : 0, Rinf : 85, b : 17},
   isotropic_hardening : "Voce" {R0 : 0, Rinf : 17, b : 262},
   nucleation_model : "Power-based" {
     An0 : 2.92,
     s1_n : 500,
     p_n : 0.0321,
     s1_max : 700,
     fg_max : 0.0215,
     ng : 2
   }
 }
```

The numerical response of NT specimens (notch radii are noted as : NT10, NT4, and NT2) is shown on Figure 9.4 and compared to the experimental results.

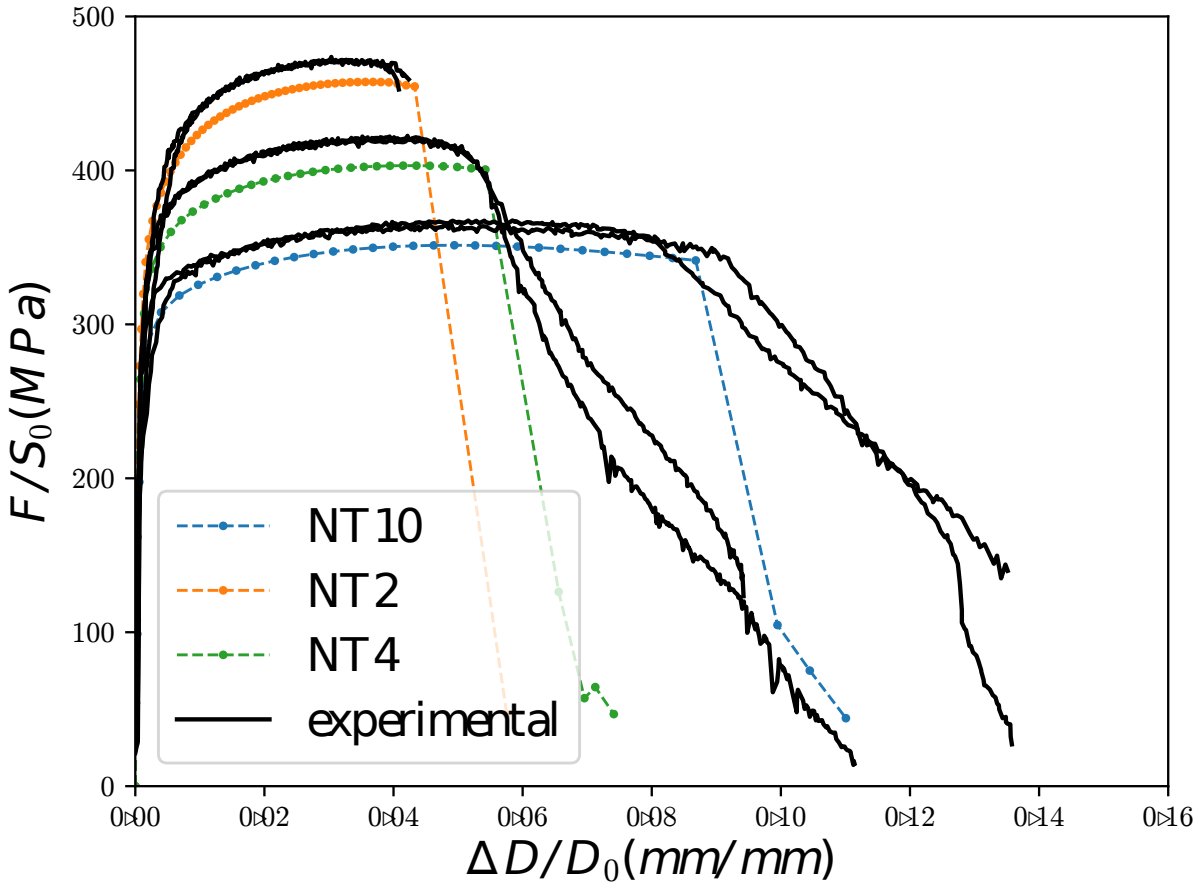

Figure 9.4: Numerical response of NT specimens

One can see that the numerical convergence is rather good at the crack initiation phase where the reaction force

decreases suddenly. Some aspects could be improved regarding the plastic flow and the damage parameters to smoothly follow the crack propagation modelled by the sudden force drop. Such improvements depends on the finite element solver used and must not be confused with the robustness of the `StandardElastoViscoPlasticity` brick.

Moreover, the evolution of the porosity was monitored during the simulation to observe the failure of elements that reached the critical porosity value. This could be seen on Figure 9.5 where the notch is pictured before the testing (left image), during the testing (middle image), and at failure (right image).

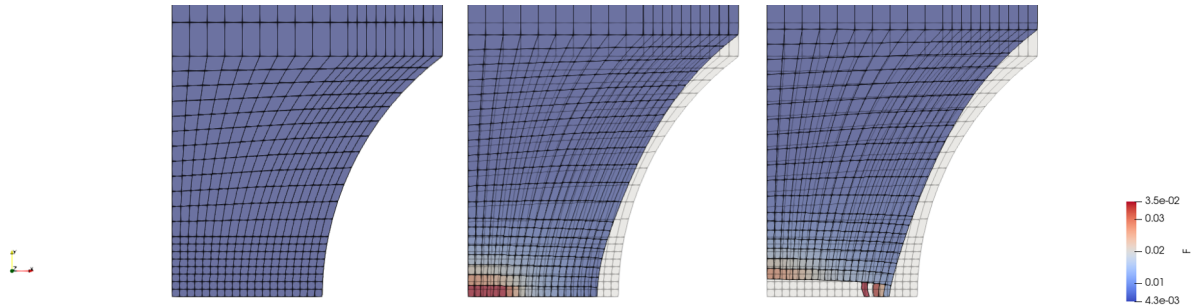

Figure 9.5: Evolution of the porosity for the NT10 specimen

9.6.2 Charpy test simulations

The GTN model is used to perform a simulation of a Charpy test. The parameters of the model are taken from [42] and correspond to the mechanical behavior of a Reactor Pressure Vessel steel at 300°C. The corresponding `MFront` implementation of the constitutive equations is:

Material's Behavior

```
@DSL Implicit;
@Behaviour GTN;
@UMATUseTimeSubStepping true;
@UMATMaximumSubStepping 100;
@ModellingHypotheses{".+"};
@StrainMeasure Hencky;
@Algorithm NewtonRaphson;
@Epsilon 1.e-12;
@Theta 1;
@Brick StandardElastoViscoPlasticity{
 stress_potential : "Hooke" {young_modulus : 210e3, poisson_ratio : 0.3},
 inelastic_flow : "Norton" {
   criterion : "GursonTvergaardNeedleman1982" {
     f_c : 0.001,
     f_r : 0.33,
     q_1 : 1.5,
     q_2 : 1,
     q_3 : 2.25
   },
   n : 1.15,
   K : 0.185,
   A : 1,
   isotropic_hardening : "Linear" {R0 : 421.5},
   isotropic_hardening : "Voce" {R0 : 0, Rinf : 125, b : 59.6},
   isotropic_hardening : "Voce" {R0 : 0, Rinf : 472, b : 1.72}
 },
 porosity_evolution : {
   nucleation_model : "PowerLaw (strain)" {
      fn : 0.038, en : 0.5, m : 0., fmax : 0.023}
 }
}
```

Simulations are performed with `Cast3M` finite element code, using second-order reduced-integration elements (user-modified `CU20` elements). Contact is modelled between the specimen and the striker and anvils accounting for friction ($\mu = 0.1$). Only one fourth of the system is modelled due to symmetries. The initial porosity is equal to 0.0175%. The displacement of the striker is imposed with a velocity of 5m.s^{-1}. The time-step is adjusted with a target on the maximal increase of local porosity of 0.1% (see Appendix F). Element deletion is activated when at least half of the Gauss points are broken. In addition, Gauss points where the cumulated plastic strain is higher than 200% are considered as broken.

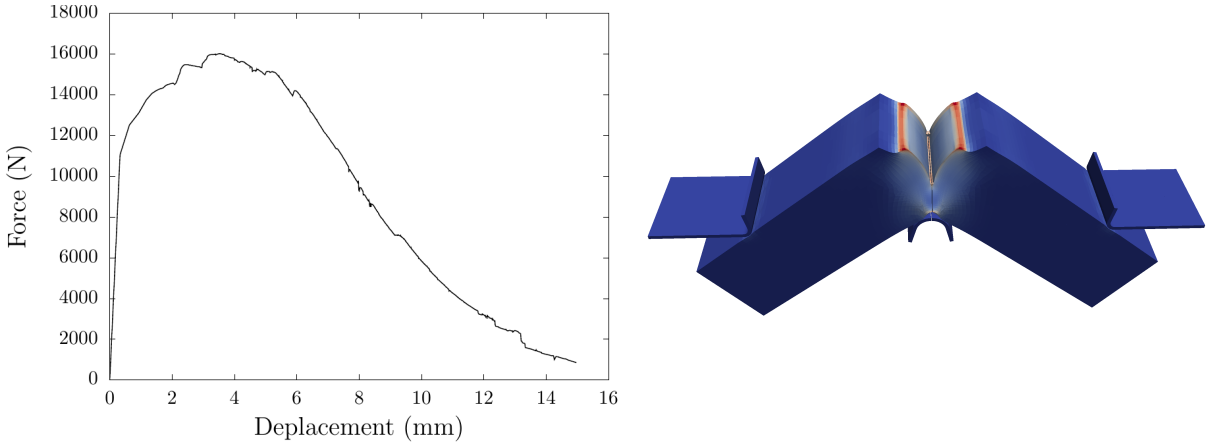

Figure 9.6: (a) Force - displacement curve (b) Cumulated plastic strain field

The evolution of the force as a function of the imposed displacement of the stricker is shown in Figure 9.6 where the decrease is related to the propagation of the crack. Figure 9.6 shows also the cumulated plastic strain field on the deformed configuration.

9.7 Conclusions

This report has described the extension of the `StandardElastoViscoPlasticity` brick to porous plasticity. Various new stress criteria and nucleation models have been implemented and can be used "out of the box." A particular attention has been paid to ease the addition of new porous stress criteria and nucleation models.

An original staggered algorithm has been proposed and tested on ductile fracture simulations of notched specimens and Charpy tests. Note that the standard implicit scheme can still be chosen.

9.8 Appendix

9.8.1 Determination of the stress criterion through a local scalar Newton algorithm

In many cases, the stress criterion $\sigma_\star$ is an implicit function of the stress state of the form:

$$S(\sigma_\star, \underline{\sigma}, f) = 0 \tag{9.19}$$

See Equations (9.1) and (9.2) for the case of the Gurson-Tvergaard-Needleman stress criterion and the Rousselier-Tanguy-Besson stress criterion respectively.

For a given stress state and porosity, Equation (9.19) may be solved by a scalar Newton algorithm.

The Newton algorithm is coupled with bisection whenever root-bracketting is possible, which considerably increase its robustness.

More precisely, if not given by the user, the algorithm tries to determines a interval $\left[\sigma_\star^{\min}; \sigma_\star^{\max}\right]$ such that:

$$S(\sigma_\star^{\min})\, S(\sigma_\star^{\max}) < 0$$

i.e. $S(\sigma_\star^{\min})$ and $S(\sigma_\star^{\max})$ have opposite signs. If such a pair is found, a simple bissection algorithm, whose convergence is guaranteed, may be used.

In practice, one checks if the new estimate given by the Newton algorithm lies in $[\sigma_\star^{\min};\sigma_\star^{\max}]$. If this is the case, this new estimate is used to update the bounds of the interval.

This implementation handles properly `IEEE754` exceptional cases (infinite numbers, `NaN` values), even if advanced compilers options are used (such as `-ffast-math` under `gcc`).

9.8.2 Derivatives of common nucleation law

- **Chu-Needleman strain based** [37]

$$\begin{aligned}\frac{\partial A_n}{\partial \underline{\sigma}} &= \underline{0}\\ \frac{\partial A_n}{\partial p} &= -\frac{f_N}{s_N^2\sqrt{2\pi}}\left(\frac{p-\epsilon_N}{s_N}\right)\exp\left(-\frac{1}{2}\left(\frac{p-\epsilon_N}{s_N}\right)^2\right)\\ \frac{\partial A_n}{\partial f} &= 0\end{aligned}$$

- **Chu-Needleman stress based** [37]

$$\frac{\partial A_n}{\partial \underline{\sigma}} = -\frac{f_N}{s_N^2\sqrt{2\pi}}\left(\frac{\sigma_I-\sigma_N}{s_N}\right)\exp\left(-\frac{1}{2}\left(\frac{\sigma_I-\sigma_N}{s_N}\right)^2\right)e_I\otimes e_I$$

$$\frac{\partial A_n}{\partial p} = 0$$

$$\frac{\partial A_n}{\partial f} = 0$$

- **Power-Law strain based**

$$\begin{aligned}\frac{\partial A_n}{\partial \underline{\sigma}} &= \underline{0}\\ \frac{\partial A_n}{\partial p} &= \frac{mf_N}{\epsilon_N}\left\langle\frac{p}{\epsilon_N}-1\right\rangle^{m-1}\\ \frac{\partial A_n}{\partial f} &= 0\end{aligned}$$

- **Power-Law stress based**

$$\frac{\partial A_n}{\partial \underline{\sigma}} = \frac{mf_N}{\sigma_N}\left\langle\frac{\sigma_I}{\sigma_N}-1\right\rangle^{m-1}e_I\otimes e_I$$

$$\frac{\partial A_n}{\partial p} = 0$$

$$\frac{\partial A_n}{\partial f} = 0$$

9.8.3 Derivatives of standard stress criteria

9.8.3.1 Derivatives of the Gurson-Tvergaard-Needleman stress criterion

As $\sigma_\star$ is defined implicitly through $S(\underline{\sigma},\sigma_\star,f)=0$, the first and second order derivatives of the function S are required to compute the first and second order derivatives of $\sigma_\star$:

$$
\begin{aligned}
\frac{\partial S}{\partial \sigma_\star} &= -2\frac{\sigma_{vM}^2}{\sigma_\star^3} - \frac{3q_1q_2f_\star\sigma_m}{\sigma_\star^2}\sinh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\\
\frac{\partial S}{\partial \underline{\sigma}} &= 3\frac{\underline{s}}{\sigma_\star^2} + \frac{q_1q_2f_\star}{\sigma_\star}\sinh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\underline{I}\\
\frac{\partial S}{\partial f} &= 2q_1\delta\cosh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right) - 2q_3\delta f_\star\\
\frac{\partial^2 S}{\partial \underline{\sigma}^2} &= \frac{2}{\sigma_\star^2}\underline{\underline{\mathbf{M}}} + \frac{q_1\,q_2^2\,f_\star}{2\sigma_\star^2}\cosh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\underline{I}\otimes\underline{I}\\
\frac{\partial^2 S}{\partial \underline{\sigma}\partial \sigma_\star} &= -\frac{6\underline{s}}{\sigma_\star^3} - \frac{q_1\,q_2\,f_\star}{\sigma_\star^2}\sinh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\underline{I} - \frac{3\,q_1\,q_2^2\,f_\star\,\sigma_m}{2\,\sigma_\star^3}\cosh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\underline{I}\\
\frac{\partial^2 S}{\partial \underline{\sigma}\partial f} &= \frac{q_1\,q_2\,\delta}{\sigma_\star}\sinh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\underline{I}\\
\frac{\partial^2 S}{\partial \sigma_\star\partial f} &= -\frac{3\,q_1\,q_2\,\delta\,\sigma_m}{\sigma_\star^2}\sinh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)\\
\frac{\partial^2 S}{\partial \sigma_\star^2} &= 6\frac{\sigma_{vM}^2}{\sigma_\star^4} + \frac{6q_1q_2f_\star\sigma_m}{\sigma_\star^3}\sinh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right) + \frac{9q_1q_2^2f_\star\sigma_m^2}{2\sigma_\star^4}\cosh\left(\frac{3}{2}q_2\frac{\sigma_m}{\sigma_\star}\right)
\end{aligned}
$$

The first and second order derivatives of $\sigma_\star$ are:

$$\frac{\partial \sigma_\star}{\partial \underline{\sigma}} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1}\frac{\partial S}{\partial \underline{\sigma}}$$

$$\frac{\partial \sigma_\star}{\partial f} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1}\frac{\partial S}{\partial f}$$

$$\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma}^2} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1}\left(\frac{\partial^2 S}{\partial \underline{\sigma}^2} + \frac{\partial^2 S}{\partial \underline{\sigma}\partial \sigma_\star}\otimes\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right) + \left(\frac{\partial S}{\partial \sigma_\star}\right)^{-2}\left(\frac{\partial S}{\partial \underline{\sigma}}\right)\otimes\left(\frac{\partial^2 S}{\partial \underline{\sigma}\partial \sigma_\star} + \frac{\partial^2 S}{\partial \sigma_\star^2}\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right)$$

$$\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma}\partial f} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1}\left(\frac{\partial^2 S}{\partial \underline{\sigma}\partial f} + \frac{\partial^2 S}{\partial f\partial \sigma_\star}\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right) + \left(\frac{\partial S}{\partial \sigma_\star}\right)^{-2}\left(\frac{\partial^2 S}{\partial \underline{\sigma}\partial \sigma_\star} + \frac{\partial^2 S}{\partial \sigma_\star^2}\frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right)\left(\frac{\partial S}{\partial f}\right)$$

9.8.3.2 Derivatives of Rousselier-Tanguy-Besson stress criterion

The first and second order derivatives of $\sigma_\star$ are computed using $S(\underline{\sigma}, \sigma_\star, f) = 0$, which requires the first and second order derivatives of the function S:

$$
\begin{aligned}
\frac{\partial S}{\partial \sigma_\star} &= -\frac{\sigma_{vM}}{(1-f)\sigma_\star^2} - fD_Rq_R\frac{\sigma_m}{(1-f)\sigma_\star^2}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\\
\frac{\partial S}{\partial \underline{\sigma}} &= \frac{3}{2}\frac{\underline{s}}{(1-f)\sigma_{vM}\sigma_\star} + \frac{fD_Rq_R}{3}\frac{\underline{I}}{(1-f)\sigma_\star}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\\
\frac{\partial S}{\partial f} &= \frac{\sigma_{vM}}{\sigma_\star(1-f)^2} + \frac{2}{3}D_R\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\left(1 + \frac{3fq_R\sigma_m}{2\sigma_\star(1-f)^2}\right)\\
\frac{\partial^2 S}{\partial \underline{\sigma}^2} &= \frac{3}{2(1-f)\sigma_\star\sigma_{vM}}\left(\frac{2}{3}\underline{\underline{\mathbf{M}}} - \frac{3}{2}\frac{\underline{s}}{\sigma_{vM}}\otimes\frac{\underline{s}}{\sigma_{vM}}\right) + \frac{fD_Rq_R^2}{6(1-f)^2\sigma_\star^2}\underline{I}\otimes\underline{I}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\\
\frac{\partial^2 S}{\partial \underline{\sigma}\partial \sigma_\star} &= -\frac{3\underline{s}}{2(1-f)\sigma_{vM}\sigma_\star^2} - \left(1 + \frac{3q_R\sigma_m}{2(1-f)\sigma_\star}\right)\frac{fD_Rq_R}{3(1-f)\sigma_\star^2}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\underline{I}\\
\frac{\partial^2 S}{\partial \underline{\sigma}\partial f} &= \frac{3\underline{s}}{2(1-f)^2\sigma_{vM}\sigma_\star} + \left(1 + \frac{3q_R\sigma_m}{2(1-f)\sigma_\star} + \frac{1-f}{f}\right)\frac{fD_Rq_R}{3(1-f)^2\sigma_\star}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\underline{I}\\
\frac{\partial^2 S}{\partial \sigma_\star\partial f} &= -\frac{\sigma_{vM}}{(1-f)^2\sigma_\star^2} - \left(1 + \frac{3q_R\sigma_m}{2(1-f)\sigma_\star} + \frac{1-f}{f}\right)\frac{fD_Rq_R\sigma_m}{(1-f)^2\sigma_\star^2}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)\\
\frac{\partial^2 S}{\partial \sigma_\star^2} &= \frac{2\sigma_{vM}}{(1-f)\sigma_\star^3} + \left(1 + \frac{3q_R\sigma_m}{4(1-f)\sigma_\star}\right)\frac{2fD_Rq_R\sigma_m}{(1-f)\sigma_\star^3}\exp\left(\frac{3}{2}q_R\frac{\sigma_m}{(1-f)\sigma_\star}\right)
\end{aligned}
$$

The first and second order derivatives of $\sigma_\star$ are:

$$\frac{\partial \sigma_\star}{\partial \underline{\sigma}} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1} \frac{\partial S}{\partial \underline{\sigma}}$$

$$\frac{\partial \sigma_\star}{\partial f} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1} \frac{\partial S}{\partial f}$$

$$\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma}^2} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1} \left(\frac{\partial^2 S}{\partial \underline{\sigma}^2} + \frac{\partial^2 S}{\partial \underline{\sigma} \partial \sigma_\star} \otimes \frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right) + \left(\frac{\partial S}{\partial \sigma_\star}\right)^{-2} \left(\frac{\partial S}{\partial \underline{\sigma}}\right) \otimes \left(\frac{\partial^2 S}{\partial \underline{\sigma} \partial \sigma_\star} + \frac{\partial^2 S}{\partial \sigma_\star^2} \frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right)$$

$$\frac{\partial^2 \sigma_\star}{\partial \underline{\sigma} \partial f} = -\left(\frac{\partial S}{\partial \sigma_\star}\right)^{-1} \left(\frac{\partial^2 S}{\partial \underline{\sigma} \partial f} + \frac{\partial^2 S}{\partial f \partial \sigma_\star} \frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right) + \left(\frac{\partial S}{\partial \sigma_\star}\right)^{-2} \left(\frac{\partial^2 S}{\partial \underline{\sigma} \partial \sigma_\star} + \frac{\partial^2 S}{\partial \sigma_\star^2} \frac{\partial \sigma_\star}{\partial \underline{\sigma}}\right) \left(\frac{\partial S}{\partial f}\right)$$

9.8.4 Regularization of the effective porosity

The Gurson-Tvergaard-Needleman stress criterion models the void coalescence by the effective porosity calculated as shown `above`. The transition from the pre-coalescence to the coalescence phase might cause numerical problems if the δ parameter is high enough. This is due to the inflection point in the bilinear function describing $f_\star$ as a function of f. Therefore, the first derivative of the effective porosity with respect to the porosity ($\frac{\partial f_\star}{\partial f}$) undergoes a sudden jump that disturbs the Newton-Raphson algorithm.

A solution is proposed to avoid that derivative jump at the inflection point : to replace the bilinear function of $f_\star$ by a polynominal function of 5th degree. Therefore, $f_\star$ is defined such that :

$$f_\star = \begin{cases} \delta f & \text{if } f < (f_c - \epsilon) \\ \delta(a^5 f + b^4 f + c^3 f + d^2 f + e f + g) & \text{if } (f_c - \epsilon) \leq f \leq (f_c + \epsilon) \\ f_c + \delta(f - f_c) & \text{otherwise} \end{cases}$$

and :

$$\delta = \begin{cases} 1 & \text{if } f \leq (f_c + \epsilon) \\ \dfrac{1}{\dfrac{q_1}{f_r - f_c}} - f_c & \text{otherwise} \end{cases}$$

where ϵ is the linearization coefficient that allows to have a smooth transition on the $\partial f_\star$ as function of f. A big ϵ value will allow a higher contribution of the polynominal function and thus, smoother numerical convergence.

For numerical efficiency reasons, the polynominal function was implemented using the Horner method to be :

$$f(f(f(f(af + b) + c) + d) + e) + g$$

9.8.5 On the use of porous constitutive equations with `Cast3M`

The staggered approach improves robustness of the numerical integration of the porous constitutive equations, especially if sub-stepping is allowed. However, integration may still fail due to too severe input parameters. As integration failure is not handled in the finite element code `Cast3M`, additional stategies are required at the structural scale to avoid such problems. An efficient stategy is to adjust the time-step based on the maximal increase of porosity such as:

$$\Delta t_{i+1} = \min\left(\frac{\Delta f_{max}}{\Delta f_i} \Delta t_i, \Delta t_{ref}\right)$$

This equation is implemented in two steps. First, the following code is added to the `PERSO1` user procedure to compute the maximal increase of porosity at the last time-step on unbroken Gauss points.

```
'SI'( ('DIME' TAB1.DEPLACEMENTS) > 1);

TMP =  TAB1.'F_PREC';
TMP2 = 'EXCO' TAB1.ESTIMATION.VARIABLES_INTERNES 'F' 'SCAL';
TMP3 = 'EXCO' TAB1.ESTIMATION.VARIABLES_INTERNES 'BROK' 'SCAL';

TMPA = TMP*(1 '-' TMP3);
TMP2A = TMP2*(1 '-' TMP3);

DF = 'ABS' (('MAXIMUM' TMPA) '-' ('MAXIMUM' TMP2A));

TAB1.'F_PREC' = TMP2;

'SINON';

DF = 0;

'FINSI';

TAB1.WTABLE.'DF' = DF;
TAB1.WTABLE.'DFMAX' = TAB1.'DFMAX';
```

Then, the PASAPAS procedure is modified according to:

```
'SI' WTAB.'PAS_AJUSTE';
   DTV = DT_AVANT;
   DF = WTAB.'DF';
   DFMAX = WTAB.'DFMAX';
   DDF = DF '/' DFMAX;
   'SI' ('EXISTE' WTAB ISOUSPAS);
      ISP = WTAB.ISOUSPAS;
   'SINON';
      ISP = 0;
   'FINSI';
   'SI' WTAB.'CONV';
      'SI' ((DDF > 1));
         DTV = DTV '/' DDF;
      'FINSI';
      'SI' ((DDF < 1) 'ET' (ISP 'EGA' 0));
         'SI' IREREDU;
            'SI' (DDF > 0.5);
               DTV = DTV '/' DDF;
            'SINON';
               DTV = DTV '/' 0.5;
            'FINSI';
         'FINSI';
      'FINSI';
      DT_AVANT = DTV;
      IREREDU = VRAI;
      'SINON';
         DTV=0.0000000001D0* DT_AVANT;
         DT_AVANT= DT_AVANT /2.;
         IREREDU=FAUX;
      'FINSI';
      TTI = DTV* 1.0000000001 + TEMP0;
      'SI' ( TTI '<EG' TIV );
         TI=TTI;
         PASFINAL=0;
      'FINSI';
   'FINSI';
```

The time-step is adjusted in order to target a maximal local increase of porosity lower or equal to an user-defined value `DFMAX`. In addition, increasing the time-step is allowed only if the convergence of the last time-step was achieved without sub-stepping (through `CONVERGENCE_FORCEE`).

In addition, global convergence may be difficult to achieve in cases where heavily-distorted elements are present. Element-deletion is detailed below, but a slight modification of the `UNPAS` procedure can be made to lower the precision of the computation when sub-steps are done, allowing to avoid premature end of the calculation:

```
*Nombre maximum de sous-pas atteint? si oui, arret de pasapas
WTAB.'ISOUSPAS' = WTAB.'ISOUSPAS' + 1;
ZPREC = ZPREC * 2;
ZPRECM = ZPRECM * 2;
```

Finally, an example of element deletion that can be added to the `PERSO1` procedure can be added, based on the number of broken Gauss points and cumulated plastic strain:

```
MO1 = TAB1.WTABLE.'MO_TOT';
MA1 = TAB1.WTABLE.'MA_TOT';
MAIL1 = 'EXTRAIRE' MO1 'MAIL';
TMP1 = 'EXCO' (TAB1.ESTIMATION.VARIABLES_INTERNES) 'BROK' 'SCAL';
TMP2 = 'EXCO' (TAB1.ESTIMATION.VARIABLES_INTERNES) 'P' 'SCAL';
TMP2 = TMP2 'MASQ' SUPE (TAB1.'BROKEN_STRAIN');

TMP1 = (TMP1 + TMP2) MASQ SUPE 0.;
TMP1 = 'CHANGER' GRAVITE MO1 TMP1 'STRESSES';
TMP1 = TMP1 'MASQUE' 'EGSU' 0.5;

MEBR = TMP1 'ELEM' SUPE 0.;
MESO = 'DIFF' MAIL1 MEBR;
NBRO = 'NBEL' MEBR;

TAB1.'DBRO' = TAB1.'DBRO' '+' NBRO;
TAB1.'NBRO' = TAB1.'NBRO' '+' NBRO;

'SI' ('>' NBRO 0);

MO2 = 'REDU' MO1 MESO;
TAB1.WTABLE.'MO_TOT' = MO2;
TAB1.WTABLE.'MO_TOTAL' = MO2;
TAB1.WTABLE.'MOD_MEC' = MO2;

'FINSI';
```

9.8.6 Options of the `porosity_evolution` section

This appendix is dedicated to the description of the various options that can be declared in the `porosity_evolution` section inside the declaration of the `StandardElastoViscoPlasticity` brick:

- `save_individual_porosity_increase`: a boolean value stating if the each contribution to the porosity increase must be stored in a dedicated state variable or auxiliary state variable. By default, this option is set to `false`.
- `elastic_contribution`: a boolean value stating if the elastic contribution to the porosity growth must be taken into account. By default, this option is set to `false`.
- `nucleation_model`: this option introduces a new nucleation model.
- `algorithm`: a string value which allow the selection of the resolution algorithm. Valid values are:
  - `standard implicit scheme`
  - `standard_implicit_scheme`
  - `StandardImplicitScheme`
  - `staggered scheme`
  - `staggered_scheme`
  - `StaggeredScheme` The `algorithm` may have suboptions, as described in Section
- `safety_factor_for_the_upper_bound_of_the_porosity`: this option allows setting floatting-point value, denoted α, which is used to determine the maximum value of the porosity at the middle of the time step as

$\alpha\, f_r$, where f_r is the upper bound of the porosity. The upper bound of the porosity is equal the minimum of the coalescence porosity of all selected stress criteria or is equal to 1 if no stress criterion declares a coalescence porosity. By default, the value of this option is 0.985.

- `safety_factor_for_the_upper_bound_of_the_porosity_for_fracture_detection`: this option allows setting floatting-point value, denoted α, which is used by the staggered algorithm to detect the failure of the material. The failure is detected when the porosity converges to a value greater than $\alpha\, f_r$ where f_r is the upper bound of the porosity. By default, the value of this option is 0.984.

9.8.6.1 Suboptions of the `algorithm` option

When a staggered scheme is used, two numerical parameters can be specified:

- `convergence_criterion`: the value of the convergence criterion of the staggered algorithm. The staggered scheme stops when the difference between two estimates of the porosity is lower than this value. The default value is $1.e-10$.
- `maximum_number_of_iterations`: the maximum number of iterations allows for the staggered scheme. The default value is 100.

Here is a simple example on how to specify those numerical parameters:

```
porosity_evolution: {
  algorithm : "staggered_scheme" {
    convergence_criterion: 1.e-9,
    maximum_number_of_iterations: 200
  }
}
```

Chapter 10

The `Implicit` DSL

10.1 Description

Let $\vec{Y}$ be a vector holding all the integration variables[1]. The evolution of $\vec{Y}$ is assumed to satisfy the following differential equation:

$$\dot{\vec{Y}} = G\left(\vec{Y}, t\right)$$

where t is not meant to describe an explicit dependency to the time but rather a placeholder for variables whose evolutions are known (those variables are called external state variables in `MFront`).

The following notations are used:

- $\vec{Y}\Big|_t$ denotes the value of the integration variables at the beginning of the time step.
- $\Delta\,\vec{Y}$ denotes the increment of the integration variables.
- $\vec{Y}\Big|_{t+\Delta\,t}$ denotes the value of the integration variables at the end of the time step:

$$\vec{Y}\Big|_{t+\Delta\,t} = \vec{Y}\Big|_t + \Delta\,\vec{Y}$$

An implicit scheme turns this ordinary differential equation into a non linear system of equations whose unknowns are is increment $\Delta\,Y$ over a time step $\Delta\,t$:

$$\Delta\,\vec{Y} - G\left(\vec{Y}\Big|_t + \theta\,\Delta\,\vec{Y}, t + \theta\,\Delta\,t\right)\Delta\,t = 0$$

The increment $\Delta\,\vec{Y}$ of the state variables satisfies the following implicit system:

$$F\left(\Delta\,\vec{Y}, \Delta\,\underline{\varepsilon}^{\mathrm{to}}\right) = 0 \tag{10.1}$$

Equation (10.1) implicitly defines $\Delta\,\vec{Y}$ as an implicit function of the increment of the increment of the strain tensor $\Delta\underline{\varepsilon}^{\mathrm{to}}$ and may be rewritten as:

$$\vec{F}\left(\Delta\,\vec{Y}(\Delta\,\underline{\varepsilon}^{\mathrm{to}}), \Delta\,\underline{\varepsilon}^{\mathrm{to}}\right) = 0 \tag{10.2}$$

10.2 Available algorithms

The following algorithms are available:

- `NewtonRaphson`

[1] In `MFront`, the integration variables refers to variables which are part of the implicit systems, i.e. variables the increments of which are the solutions of the implicit system. A state variable, declared with `@StateVariable` keyword, is saved from one step to the other and is automatically added to the list of integration variables. An auxiliary state variable, declared with `@AuxiliaryStateVariable` keyword, is also saved from one step to the other, but is not added to the list of the integration variables. The `@IntegrationVariable` keyword allows to append a variable to the integration variables but its value wil not be saved from one step to the other.

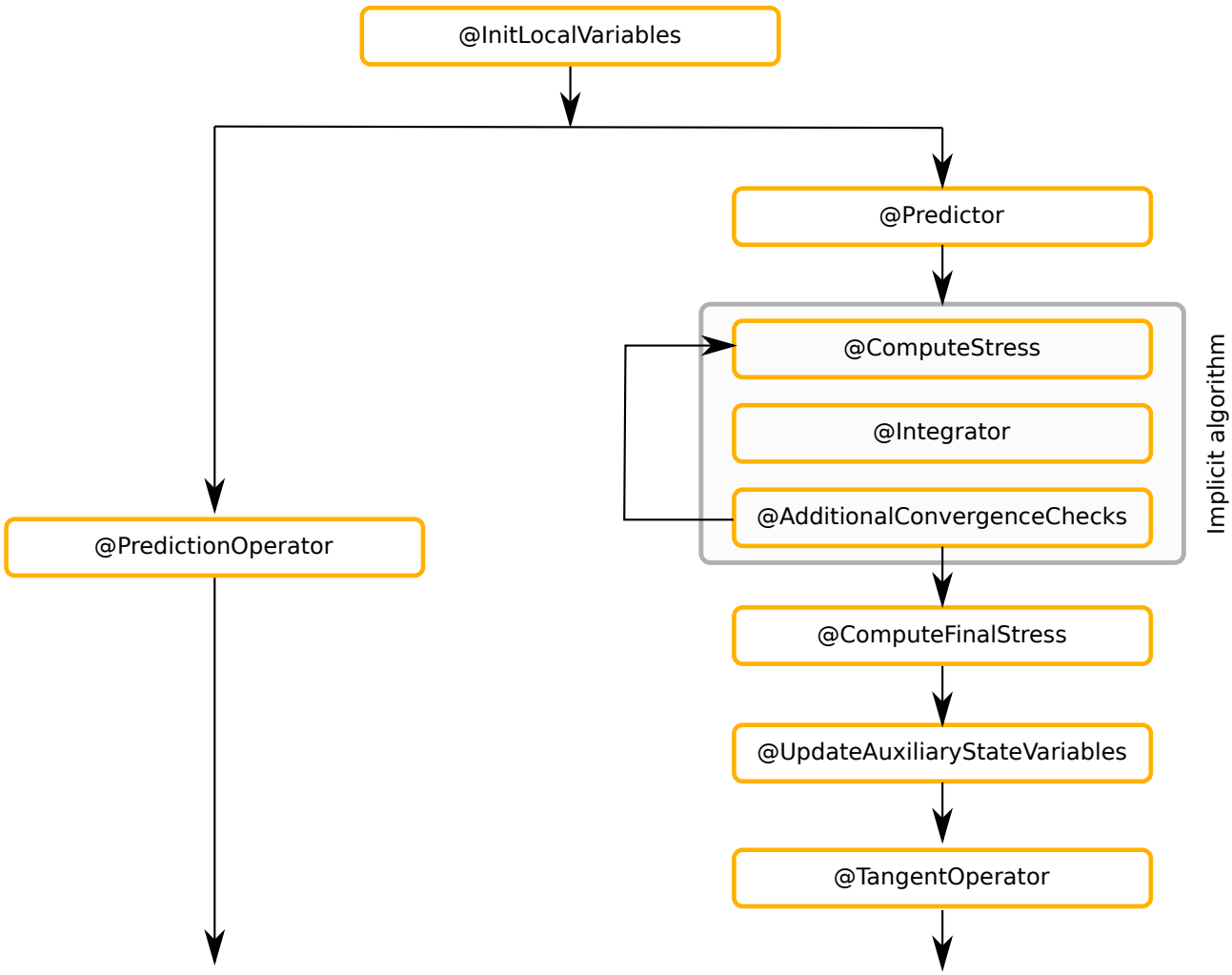

Figure 10.1: Description of the steps of the implicit scheme and the associated code blocks.

- `NewtonRaphson_NumericalJacobian`
- `PowellDogLeg_NewtonRaphson`
- `PowellDogLeg_NewtonRaphson_NumericalJacobian`
- `Broyden`
- `PowellDogLeg_Broyden`
- `Broyden2`
- `LevenbergMarquardt`
- `LevenbergMarquardt_NumericalJacobian`

Thoses algorithms are described in the documention of the `TFEL/Math` library. The main steps of those algorithms and the associated code blocks are depicted in Figure 10.1.

10.2.1 Notes about updating auxiliary state variable or local variables in the `Integrator` code blocks when the numerical evaluation of the jacobian is requested

In some cases, it is convenient to update auxiliary state variables in the `@Integrator` code block, rather than computing them in the `@UpdateAuxiliaryStateVariables` code block which is only called once the convergence is reached.

However, if the jacobian matrix is computed numerically (at least partially), such updates could be wrong, because they can be based on the perturbated values of the unknowns.

In `TFEL 3.1`, this can be circumvented by testing the value of the `perturbatedSystemEvaluation` boolean value as follows:

```
// let av be an auxiliary state variable
@AuxiliaryStateVariable StrainStensor av;

@Integrator{
  // put updated value of av in a temporary variable
  const auto av_ = eval(...);
  ...
  definition of the implicit system
  ...
  if(!perturbatedSystemEvaluation){
    // update auxiliary state variables
    av = av_;
  }
} // end of @Integrator
```

In many cases, rather than updating auxiliary variables during the Newton iterations, it can be more pratical to compute its increment, defined in by local variable and to update the auxiliary variable in the `@UpdateAuxiliaryStateVariables` code block. The previous trick can be used in this case in a straightforward manner.

10.3 Computation of the consistent tangent operator

10.3.1 Computation of the consistent tangent operator for small strain behaviours

In this section, a small strain behaviour is considered.

The increment $\Delta\,\vec{Y}$ of the state variables satisfies the following implicit system:

$$F\left(\Delta\,\vec{Y},\Delta\,\underline{\varepsilon}^{\mathrm{to}}\right)=0 \tag{10.3}$$

Equation (10.3) implicitly defines $\Delta\,\vec{Y}$ as an implicit function of the increment of the increment of the strain tensor $\Delta\underline{\varepsilon}^{\mathrm{to}}$ and may be rewritten as:

$$\vec{F}\left(\Delta\,\vec{Y}(\Delta\,\underline{\varepsilon}^{\mathrm{to}}),\Delta\,\underline{\varepsilon}^{\mathrm{to}}\right)=0 \tag{10.4}$$

The stress tensor $\underline{\sigma}$ are assumed to be an explicit function of the integration variables $\vec{Y}\big|_{t+\Delta t}$ at the end of the time step and the total strain $\underline{\varepsilon}^{\mathrm{to}}\big|_{t+\Delta t}$ at the end of time step:

$$\underline{\sigma}\left(\vec{Y}\big|_{t+\Delta t}, \underline{\varepsilon}^{\mathrm{to}}\big|_{t+\Delta t}\right)$$

10.3.1.1 Formal derivation of the consistent tangent operator

The consistent tangent operator is thus given by:

$$\frac{\mathrm{d}\underline{\sigma}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = \frac{\partial\underline{\sigma}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}} + \frac{\partial\underline{\sigma}}{\partial\Delta\,\vec{Y}}\frac{\partial\Delta\,\vec{Y}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}}$$

By differentiation:

$$\frac{\mathrm{d}\vec{F}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = \frac{\partial\vec{F}}{\partial\Delta\,\vec{Y}}\frac{\mathrm{d}\Delta\,\vec{Y}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} + \frac{\partial\vec{F}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = 0$$

$\dfrac{\partial\vec{F}}{\partial\Delta\,\vec{Y}}$ is the jacobian J of the implicit system.
Hence,

$$\frac{\mathrm{d}\Delta\,\vec{Y}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = -J^{-1}\frac{\partial\vec{F}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}}$$

Finally, the consistent tangent operator is formally given by:

$$\frac{\mathrm{d}\underline{\sigma}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = \frac{\partial\underline{\sigma}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}} - \frac{\partial\underline{\sigma}}{\partial\Delta\,\vec{Y}}J^{-1}\frac{\partial\vec{F}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}} \tag{10.5}$$

10.3.1.2 Discussion

Equation (10.5) may be extented almost directly to generalised behaviours. However, direct usage of Equation (10.5) has several practical disadvantages:

- The matrix $\dfrac{\partial\underline{\sigma}}{\partial\Delta\,\vec{Y}}$ is usually sparse. For example, most small strain behaviours are based on the Hooke law, which means that the stress $\underline{\sigma}$ only depends on the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$. In many behaviours, the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$ are only a small subset of the integration variables $\vec{Y}$.
- It requires to invert the jacobian matrix.
- The matrix $\dfrac{\partial\vec{F}}{\partial\Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ is also usually sparse. For example, the $\Delta\,\underline{\varepsilon}^{\mathrm{to}}$ commonly only appears in the implicit equation describing the split of the strain.

Hence, a computation of the consistent tangent based on Equation (10.5) may imply a significant performance hit which can be avoided in many most common cases, as discussed in the following section.

10.3.1.3 Simplication of the Equation (10.5) in common cases

Equation (10.5) can be simplified if the following assumptions are made:

- The integration variable $\vec{Y}$ is decomposed as:

$$\vec{Y} = \begin{pmatrix}\Delta\,\underline{\varepsilon}^{\mathrm{el}}\\ \Delta\,\vec{z}\end{pmatrix}$$

  where $\vec{z}$ is a sub-vector of $\vec{Y}$ containing the other state variables. The implicit system is also decomposed in a similar manner:

$$\vec{F} = \begin{pmatrix}f_{\underline{\varepsilon}^{\mathrm{el}}}\\ f_{\vec{z}}\end{pmatrix}$$

- The implicit equation $f_{\underline{\varepsilon}^{\mathrm{el}}}$ associated with the elastic strain holds the split of the strain, which means that is has the form:

$$f_{\underline{\varepsilon}^{\mathrm{el}}} = \Delta\underline{\varepsilon}^{\mathrm{el}} - \Delta\underline{\varepsilon}^{\mathrm{to}} + ...$$

- The strain increment $\Delta\underline{\varepsilon}^{\mathrm{to}}$ only appears in $f_{\underline{\varepsilon}^{\mathrm{el}}}$. Thus, $\dfrac{\partial \vec{F}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ has the form:

$$\frac{\partial \vec{F}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}} = -\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{10.6}$$

- The stress tensor only depends on the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$ through the Hooke law:

$$\underline{\sigma}|_{t+\Delta t} = \underline{\underline{\mathbf{D}}}|_{t+\Delta t} : \underline{\varepsilon}^{\mathrm{el}}|_{t+\Delta t} \tag{10.7}$$

  The expression of the consistent tangent operator is then:

$$\frac{\mathrm{d}\underline{\sigma}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = \underline{\underline{\mathbf{D}}}|_{t+\Delta t} : \frac{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{el}}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} \tag{10.8}$$

Those assumptions are more or the less the basis upon which various bricks in `MFront` are built (`StandardElasticity`, `StandardElastoViscoplasticity`, `DDIF2`).

Thanks to Equation (10.6), one sees that the product $-J^{-1}\dfrac{\partial \vec{F}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ only contains the the 6 first columns of the J^{-1}. This allows identifying $\dfrac{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{el}}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ with the 6×6 upper-left submatrx of J^{-1}. Let $J^{-1}_{\underline{\varepsilon}^{\mathrm{el}}}$ this submatrix:

$$J^{-1}_{\underline{\varepsilon}^{\mathrm{el}}} = \frac{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{el}}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}}$$

Thanks to Equation (10.8), the consistent tangent operator finally reads:

$$\frac{\mathrm{d}\underline{\sigma}}{\mathrm{d}\Delta\,\underline{\varepsilon}^{\mathrm{to}}} = \underline{\underline{\mathbf{D}}}|_{t+\Delta t} \cdot J^{-1}_{\underline{\varepsilon}^{\mathrm{el}}} \tag{10.9}$$

10.3.1.3.1 The `getPartialJacobianInvert` method The `getPartialJacobianInvert` method allows the computation of the $J^{-1}_{\underline{\varepsilon}^{\mathrm{el}}}$ tensors as follows:

```
Stensor4 iJe;
getPartialJacobianInvert(iJe);
```

The `getPartialJacobianInvert` method assumes that the jacobian matrix is decomposed (using the `LU` algorithm with partial pivoting in the current version of `TFEL`), which is guaranteed in the `@TangentOperatorBlock`. So $J^{-1}_{\underline{\varepsilon}^{\mathrm{el}}}$ is computed by solving 6 linear systems. In particular, the jacobian matrix is not inverted.

A typical computation of the consistent tangent operator is as follows:

```
    StiffnessTensor De;
    Stensor4 iJe;
    computeElasticStiffness<N,Type>::exe(De,lambda,mu);
    getPartialJacobianInvert(iJe);
    Dt = De * Je;
```

This code is loosely the one generated by the `StandardElastoViscoplasticy` brick.

10.3.1.3.2 Extensions to more general class of behaviours

The `getPartialJacobianInvert` may take several arguments which will give the derivatives of the other integration variables.

For example, the following code can be used to compute the consistent tangent operator of a behaviour in which the stress computed using a standard damaging law of the form $\underline{\sigma}|_{t+\Delta t} = \left(1 - d|_{t+\Delta t}\right) \underline{\underline{\mathbf{D}}}|_{t+\Delta t} : \underline{\varepsilon}^{\mathrm{el}}|_{t+\Delta t}$[2,3]:

```
StiffnessTensor De;
Stensor4 iJe;
Stensor iJd;
computeUnalteredElasticStiffness<N,Type>::exe(De, lambda, mu);
getPartialJacobianInvert(iJe, iJd);
Dt = (1 - d) * De * iJe - ((De * eel) ^ iJd);
```

10.3.2 Extension to generalised behaviours

The idea is to extend this method for generalised behaviours.

10.3.2.1 Discussion

In Section 10.3.1.2, we discussed various drawbacks of a direct use of Equation (10.5).

The equivalent of the consistent tangent operator for generalised behaviours are so called tangent operator blocks which can be:

- the derivatives of one of the thermodynamic force with respect to the increment of a gradient.
- the derivative of one of the thermodynamic force with respect to the increment of an external state variable gradient.
- the derivative of a state variable with respect to the increment of a gradient.
- the derivative of a state variable with respect to the increment of an external state variable.

In the spirit of the Section 10.3.1, the computation of those derivatives boils down to computing the derivatives of the increment of the state variables $\Delta\,\vec{Y}$ with respect to the increment of the considered variable X (X can thus be a gradient or an external state variable).

As the derivative $\dfrac{\partial F}{\partial \Delta\, X}$ can not be assumed to be sparse, two solutions can be considered:

- extend the `getPartialJacobianInvert` method.
- have access to the blocks of the invert of the jacobian matrix.

In both cases, $\dfrac{\partial F}{\partial \Delta\, X}$ matrix must be computed. The next paragraph is devoted to this computation.

10.3.2.2 Computation of the $\dfrac{\partial F}{\partial \Delta\, X}$ matrix.

The $\dfrac{\partial F}{\partial \Delta\, X}$ matrix can be decomposed by blocks:

$$\frac{\partial F}{\partial \Delta\, X} = \begin{pmatrix} \dfrac{\partial f_{y_1}}{\partial \Delta\, X} \\ \vdots \\ \dfrac{\partial f_{y_n}}{\partial \Delta\, X} \end{pmatrix} \tag{10.10}$$

where $(y_1, \cdots y_n)$ denotes the set of integration variables, i.e.:

$$\vec{Y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \quad \text{and} \quad \vec{F} = \begin{pmatrix} f_{y_1} \\ \vdots \\ f_{y_n} \end{pmatrix}$$

The advantage of the Decomposition (10.10) is that each blocks $\dfrac{\partial f_{y_i}}{\partial \Delta\, X}$ is a tensorial object that can be computed using the facilities offered by `TFEL/Math` library

[2] This assumes that the damage d is the second integration variables.

[3] This code uses the fact the state variables have been updated to their values at the end of the time step before the call to `@TangentOperator` code block.

If one of the block $\frac{\partial f_{y_i}}{\partial \Delta X}$ appears in a `@TangentOperator` code block, `MFront` automatically defines the matrix $\frac{\partial F}{\partial \Delta X}$ as an hidden variable and **initializes it to zero. This does mean that only non zero derivatives must be defined by the user**

For a standard integration variable, a view called $\frac{\partial f_{y_i}}{\partial \Delta X}$ is also automatically defined. For array of integration variables, a local function object $\frac{\partial f_{y_i}}{\partial \Delta X}$ is defined. This mechanism is similar to the one used to access the jacobian by block.

10.3.2.3 The `getIntegrationVariablesDerivatives_X` local function objects

When required in a `@TangentOperator` code block (i.e. when used), the `getIntegrationVariablesDerivatives_X` is defined as a local function object which may the blocks of $-J^{-1}\,\frac{\partial F}{\partial \Delta X}$

10.3.2.4 Block decomposition of the invert of the jacobian

The invert of the jacobian J can be decomposed as:

$$J^{-1} = \begin{pmatrix} J^{-1}_{(f_{y_1},y_1)} & \cdots & J^{-1}_{(f_{y_1},y_n)} \\ \vdots & \ddots & \vdots \\ J^{-1}_{(f_{y_n},y_1)} & \cdots & J^{-1}_{(f_{y_n},y_n)} \end{pmatrix}$$

When required in the `@TangentOperator` code block (i.e. when used), for each pair of state variables y, z, a variable called `iJ_x_z` is defined:

- If y and z are not arrays, `iJ_x_z` is directly a view to the corresponding block of the invert of the jacobian.
- If y and/or z are arrays, `iJ_x_z` is a function object taking indices as arguments and returning a view to the corresponding block of the invert of the jacobian.

For example:

```
@TangentOperator{
 // iJ_eel_p is the block corresponding to:
 // 1. the elastic strain (row)
 // 2. the equivalent viscoplastic strain
}
```

10.3.3 Applications

10.3.3.1 A first example of a viscoplastic behaviour coupled with heat transfer

10.3.3.1.1 Description of the behaviour Let us consider the following simple isotropic viscoplastic behaviour.

The total strain $\underline{\varepsilon}^{\mathrm{to}}$ is split into an elastic part $\underline{\varepsilon}^{\mathrm{el}}$, a thermal strain $\underline{\varepsilon}^{\mathrm{th}}$ and a viscoplastic part $\underline{\varepsilon}^{\mathrm{vp}}$:

$$\Delta\,\underline{\varepsilon}^{\mathrm{to}} = \Delta\,\underline{\varepsilon}^{\mathrm{el}} + \Delta\,\underline{\varepsilon}^{\mathrm{th}} + \Delta\,\underline{\varepsilon}^{\mathrm{vp}}$$

The thermal strain is given by:

$$\underline{\varepsilon}^{\mathrm{th}} = \alpha(T)\,(T - T_{\mathrm{ref}})$$

where:

- T is the temperature, assumed to be an external state variable.
- T_{ref} is a reference temperature.

The stress tensor $\underline{\sigma}$ is related to the elastic strain by the Hooke law:

$$\underline{\sigma} = \lambda(T)\,\mathrm{tr}(\underline{\varepsilon}^{\mathrm{el}})\,\underline{I} + 2\,\mu(T)\,\underline{\varepsilon}^{\mathrm{el}}$$

The viscoplastic strain rate is given by the famous Norton-Hoff law:

$$\underline{\dot{\varepsilon}}^{\mathrm{vp}} = \dot{p}\,\underline{n} = A\,\exp\left(-\frac{T_a}{T}\right)\sigma_{\mathrm{eq}}^{E}\,\underline{n}$$

where:

- σ_{eq} is the equivalent von Mises stress.
- $\underline{n}$ is the von Mises normal.
- A, T_a and E are material parameters.
- p is the equivalent plastic strain.

The integration variables are the elastic strain $\underline{\varepsilon}^{\mathrm{el}}$ and the equivalent plastic strain p.

The tangent operator blocks to be computed are $\dfrac{\partial \underline{\sigma}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ and $\dfrac{\partial \underline{\sigma}}{\partial \Delta\, T}$.

The computation is of $\dfrac{\partial \underline{\sigma}}{\partial \Delta\,\underline{\varepsilon}^{\mathrm{to}}}$ readily follows Section 10.3.1.3.1 and will not be described here.

10.3.3.1.2 Implicit scheme

$$\begin{cases} f_{\underline{\varepsilon}^{\mathrm{el}}} = \Delta\,\underline{\varepsilon}^{\mathrm{el}} + \Delta\,\underline{\varepsilon}^{\mathrm{th}} + \Delta\,p\,\underline{n}\big|_{t+\theta\,\Delta\,t} - \Delta\underline{\varepsilon}^{\mathrm{to}} \\ f_p = \Delta\,p - A\,\exp\left(-\dfrac{T_a}{T|_{t+\theta\,\Delta\,t}}\right)\left(\sigma_{\mathrm{eq}}\big|_{t+\theta\,\Delta\,t}\right)^{E} \end{cases}$$

10.3.3.2 Computation of $\dfrac{\partial F}{\partial \Delta T}$

$$\begin{cases} \dfrac{\partial f_{\underline{\varepsilon}^{\mathrm{el}}}}{\partial \Delta T} = \dfrac{\partial \Delta \underline{\varepsilon}^{\mathrm{th}}}{\partial \Delta T} = \dfrac{\partial\, \underline{\varepsilon}^{\mathrm{th}}\big|_{t+\Delta\,t}}{\partial \Delta T} = \left(\alpha\big(T|_{t+\Delta\,t}\big) + \left.\dfrac{\mathrm{d}\alpha}{\mathrm{d}T}\right|_{T|_{t+\Delta\,t}}\left(T|_{t+\Delta\,t} - T_{\mathrm{ref}}\right)\right)\underline{I} \\ \dfrac{\partial f_p}{\partial \Delta T} = -A\,\theta\,\dfrac{T_a}{T|_{t+\theta\,\Delta\,t}{}^{2}}\,\exp\left(-\dfrac{T_a}{T|_{t+\theta\,\Delta\,t}}\right)\left(\sigma_{\mathrm{eq}}\big|_{t+\theta\,\Delta\,t}\right)^{E} \end{cases}$$

```
∂fε^el/∂ΔT = ...;
∂fp/∂ΔT = ...;
```

10.3.3.3 Computing the derivate of the stress with respect to the temperature using `getIntegrationVariablesDerivatives_T`

```
@TangentOperatorBlock{
  ∂fε^el/∂ΔT = ...;
  ∂fp/∂ΔT = ...;
  auto ∂Δε^el/∂ΔT = Stensor{};
  getIntegrationVariablesDerivatives_T(∂Δε^el/∂ΔT);
  const auto λ =...;
  const auto μ =...;
  const auto ∂λ/∂T =...;
  const auto ∂μ/∂T =...;
  const auto De = λ  · (I₂ ⊗ I₂) + 2 · μ · I₄;
  const auto ∂De/∂T = ∂λ/∂T  · (I₂ ⊗ I₂) + 2 · ∂μ/∂T · I₄;
  ∂σ/∂ΔT = De · ∂Δε^el/∂ΔT + ∂De/∂T · ε^el;
}
```

10.3.3.4 Computing the derivate of the stress with respect to the temperature using the block decomposition of the invert of jacobian

```
@TangentOperatorBlock{
  const auto λ =...;
  const auto μ =...;
```

```
  const auto De = λ  · (I₂ ⊗ I₂) + 2 · μ · I₄;
  // computation of ∂σ/∂ΔT
  ∂fε^el/∂ΔT = ...;
  ∂fp/∂ΔT = ...;
  const auto ∂Δε^el/∂ΔT = -(iJ_fε^el_ε^el · ∂fε^el/∂ΔT + iJ_fε^el_p · ∂fp/∂ΔT);
  const auto ∂λ/∂T =...;
  const auto ∂μ/∂T =...;
  const auto ∂De/∂T = ∂λ/∂T  · (I₂ ⊗ I₂) + 2 · ∂μ/∂T · I₄;
  ∂σ/∂ΔT = De · ∂Δε^el/∂ΔT + ∂De/∂T · ε^el;
}
```

Chapter 11

Overview of `MTest`

11.1 Simulations at the integration point level

11.2 Simulations of pipes

11.3 Python bindings

Chapter 12

The MFrontGallery project

12.1 Introduction

The `MFrontGallery` project addresses the question of the management of `MFront` implementations including their compilation, unit testing and deployment.

The MFrontGallery project has two main, almost orthogonal, goals:

1. The first one is to show how solver developers may provide their users a set of ready-to-use (mechanical) behaviours which can be parametrized by their users to match their needs.
2. The second one is to describe how to set up a high-quality material knowledge management project based on `MFront`, able to meet the requirements of safety-critical studies.

While the first goal is common to all (mechanical) solvers, one originality of the `MFrontGallery` project is to address the second goal which is discussed in depth in Section 12.2.

In summary, the project provides:

- a `CMake` infrastructure that can be duplicated in (academic or industrial) derived projects. This infrastructure allows:
  - to compile `MFront` sources using all interfaces supported by `MFront`.
  - to execute unit tests based on `MTest`. Those unit tests generate `XML` result files conforming to the `JUnit` standard that can readily be used by continuous integration platforms such as jenkins.
  - generate the documentation associated with the stored implementations.
- a documentation of best practices to handle material knowledge implemented using `MFront` implementations, such as use of consistent unit systems, bound-aware physical quantities, consistent tangent operators, and others.
- a set of high-quality `MFront` implementations.

This paper aims to describe the project and is organized as follows:

Section 12.2 discusses why a new approach to material knowledge management is needed in the context of safety-criticial studies.

Section 12.3 provides an overview of the `CMake` infrastructure of the project, and describes how to create a derived project based on the same `CMake` infrastructure as the `MFrontGallery`.

12.2 Statement of need : material knowledge management for safety - criticial studies

12.2.1 Role of material knowledge in numerical simulations of solids

Numerical simulations of solids are based on the description of the evolution of the thermodynamical state of the materials of interest. In the context of the `MFrontGallery` project, this thermodynamical state is described at each point in space by a set of internal state variables which can evolve with time due to various physical phenomena.

The knowledge that one may have about a given material can be represented in different forms. In `MFront`, the following categorization is employed:

- **Material properties** are defined here as functions of the current state of the material. A typical example is the Young modulus of a material.
- **Behaviours** describe how a material evolves and reacts locally due to gradients inside the material. Here, the material reaction is associated with fluxes (or forces) thermodynamically conjugated to gradients. For instance, Fourier's law relates the heat flux to the temperature gradient. Mechanical behaviour in infinitesimal strain theory relates the stress and the strain and may describe (visco)elasticity, (visco)plasticity, or damage.
- **Point-wise models** describe the evolution of some internal state variables with the evolution of other state variables. Point-wise models may be seen as behaviours without gradients. Phase transition, swelling under irradiation or shrinkage due to dessication are examples of point-wise models.

12.2.2 Requirements related to safety-critical studies

The `MFrontGallery` project has been developed to address various issues related to material knowledge management for safety-critical studies:

- **Intellectual property**: Frequently, material knowledge reflects the know-how of industrials and shall be kept private for various reasons. For example, some mechanical behaviours result from years of experimental testing in dedicated facilities and are thus highly valuable. In some cases, material knowledge can be a competitive advantage. To solve this issue, the `MFrontGallery` allows to create private derived projects, as detailled in Section 12.3.2.
- **Portability**: safety-critical studies may involve several partners which use different solvers for independent assessment and review. From the same `MFront` source file, the `MFrontGallery` can generate shared libraries for all the solvers of interest. Moreover, the project employs best practices guidelines[1] to ensure that a given `MFront` implementation can be shared among several teams while assuring quality.
- **Maintainability over decades**: Some safety-critical studies involve the design of buildings, plants, or technological systems for operation periods of decades or more. Over such periods of time, both the solvers and the material knowledge will evolve. The safety-critical studies, however, on which design choices or decisions were based, need to remain accessible and reproducible. In the authors' experience, maintainability is more easily achieved by having a dedicated material knowledge project based on *self-contained* implementations, as discussed in Section 12.2.3.
- **Progression of the state of the art**: Safety-critical studies need to reflect the state of the art. As such, the material knowledge per se, numerical methods and software engineering need to evolve while at the same time ensuring the other principles listed here are not violated in order to maintain quality assurance of past, present and future analyses.
- **Continuous integration and unit testing**: Each implementation has associated unit tests which can check no-regression during the development of `MFront`.
- **Documentation**: the project can generate the documentation associated with the various implementations in an automated manner. Implementations of material knowledge can be associated to essential meta data.

12.2.3 Implementations and classification

`MFront` implementations can be classified in two main categories:

- **self-contained implementations** that contain all the physical information (e.g., model equations and parameters).
- **generic implementations** for which the solver is required to provide additional physical information to the material treated, e.g. the values of certain parameters. Those "generic" implementations are usually shipped with solvers as ready-to-use behaviours.

An alternative way of expressing the distinction between self-contained and generic implementations is to consider that self-contained implementations describes a set of constitutive equations **and** the material coefficients[2] identified on a well-defined set of experiments for a particular material while generic implementations only describe a set of constitutive equations.

[1] https://thelfer.github.io/MFrontGallery/web/best-practices.html

[2] In practice, the physical information described by self-contained implementations may be more complex than a set of material coefficients. For example, the Young modulus of a material may be defined by an analytical formula and cannot thus be reduced to a set of constants. This analytical formula shall be part of a self-contained mechanical behaviour implementation. Of course, this analytical formula could be included in the set of constitutive equations and parametrized to retrieve a certain degree of generality. In our experience, such a hybrid approach is fragile, less readable and cumbersome. Moreover, it does not address the main issue of generic behaviours which is the management of the physical information in a reliable and robust manner.

In the authors' experience, self-contained behaviours allows to **decouple the material knowledge management from the development (source code) of the solvers of interest** and thus allow a proper material knowledge management strategy suitable to meet the requirements depicted on Section 12.2.2.

12.3 The CMake infrastructure

This section provides an overview of the CMake infrastructure of the MFrontGallery and MFrontMaterials projects.

This infrastructure is fully contained in the cmake/modules directory, the file cmake/modules/mfm.cmake being the main entry point.

Section 12.3.1 describes the main CMake functions provided by the project.

Section 12.3.2 shows how to create a derived project.

12.3.1 Main functions

The CMake infrastructure provides:

- functions used to compile MFront files related to material properties, behaviours and models.
- functions related to unit testing. Those functions will not be described in this paper.
- functions related to documentation and website generation. Those functions will not be described in this paper.

12.3.2 Creation of a derived project

This section describes how to setup a new project based on the CMake infrastructure of the MFrontGallery and MFrontMaterials projects.

12.3.2.1 Fetching cmake/modules directory

To create a material management project derived from MFrontGallery, just copy the contents of the cmake/modules directory in your local directory.

If you intend to use git for version control, one easy way is to add the MFrontGallery as a remote ressource and check out the CMake repository from it as follows:

```
$ git remote add MFrontGallery https://github.com/thelfer/MFrontGallery
$ git fetch MFrontGallery master
$ git checkout MFrontGallery/master --cmake
```

Of course, the user may also want to follow another branch rather than the master branch.

12.3.2.2 Top-level CMakeLists.txt file

The next step consist of creating a top level CMakeLists.txt file. Here is a minimal example:

```
project(NewMaterialManagementProject)
set(PACKAGE new-material-management-project)
cmake_minimum_required(VERSION 3.0.2)

include(cmake/modules/mfm.cmake)

## testing
set(CTEST_CONFIGURATION_TYPE "${JOB_BUILD_CONFIGURATION}")
enable_testing()

## add subdirectories here

add_subdirectory(materials)
```

Now you are ready to create the subdirectory materials containing your MFront files.

12.4 Conclusions

The `MFrontGallery` project is dedicated to material knowledge management in safety-critical studies and is the result of long-standing experience gathered in the `PLEIADES` project. Key concepts built upon are portability, maintainability and reproducability over long time periods, continuous integration and unit testing, documentation and the safeguarding of intellectual property as well as attribution. Based on the technical infrastructure described in this article, it becomes possible to set up derived projects in similar contexts where these concepts are considered relevant.

Chapter 13

The MFrontGenericInterfaceSupport project

Appendix A

Tensors and tensorial operations in the TFEL/Math and TFEL/Material libraries

This page is meant to describe the various tensor objects and operations available in `TFEL/Math` and some functionalities provided by the `TFEL/Material` library.

A.1 Classes describing second and fourth order tensors

A.1.1 Symmetric second order tensors

When dealing with constitutive equations, most computations are performed on *symmetric* tensors.

Classes describing symmetric second order tensors satisfies the `StensorConcept`.

An end user will mostly use the `stensor` class, which have the following template arguments:

- The space dimension (`1`, `2` or `3`).
- The type used to perform the computation.

The following code declares a symmetric second order tensor in $2D$ using single precision floating-point number:

```
stensor<2,float> s;
```

Symmetric tensors are denoted as follows $\underline{s}$.

A.1.1.1 Aliases used in `MFront`

In `MFront`, various aliases are introduced to ease the implementation of mechanical behaviours:

- `Stensor` is an alias to `stensor<N,real>` where `N` is the space dimension and `real` is the numerical type used. The value of `N` depends of the current modelling hypothesis. The concrete type for `real` depends on the interface used.
- `StrainStensor` is an alias to `stensor<N,strain>`.
- `StressStensor` is an alias to `stensor<N,stess>`.

A.1.2 Vector notations for symmetric tensors

A symmetric tensor is stored as an array of values, as follows in $3D$:

$$\underline{s} = \left(s_{11} \quad s_{22} \quad s_{33} \quad \sqrt{2}\,s_{12} \quad \sqrt{2}\,s_{13} \quad \sqrt{2}\,s_{23}\right)^T$$

The contracted product of two symmetric tensors is the scalar product of their vector forms (hence the $\sqrt{2}$).

A.1.3 General (non symmetric) second order tensors

Classes describing symmetric second order tensors satisfies the `TensorConcept`.

An end user will mostly use the `tensor` class, which have the following template arguments:

- The space dimesion (`1`, `2` or `3`).
- The type used to perform the computation.

Non symmetric tensors are denoted as follows $\underset{\sim}{\mathbf{a}}$.

A.1.4 Vector notations for non symmetric tensors

A tensor is stored as an array of values, as follows in $3D$:

$$\underline{s} = \left(s_{11} \quad s_{22} \quad s_{33} \quad s_{12} \quad s_{21} \quad s_{13} \quad s_{31} \quad s_{23} \quad s_{32}\right)^T$$

A.1.4.1 The identity tensor

The symmetric second order identity tensor is returned by the `Id` static member of the `stensor` class as follows:

```
constexpr const auto id = stensor<3u,real>::Id();
```

A.1.5 Fourth order tensors

Fourth order tensors can be defined as linear mappings from the second order tensors to second order tensors. As there is two kinds of second order tensors (i.e. symmetric and non symmetric tensors), there are four kinds of fourth order tensors defined in the `TFEL/Math` library, which satisfy the following concepts:

- `ST2toST2Concept`: linear mapping from symmetric tensors to symmetric tensors.
- `ST2toT2Concept`: linear mapping from symmetric tensors to non symmetric tensors.
- `T2toST2Concept`: linear mapping from non symmetric tensors to symmetric tensors.
- `T2toT2Concept`: linear mapping from non symmetric tensors to non symmetric tensors.

An end user will mostly use the following implementations of those concepts: `st2tost2`, `st2tot2`, `t2tost2` and `t2tot2` respectively. Those classes have the following template arguments:

- The space dimesion (`1`, `2` or `3`).
- The type used to perform the computation.

A.1.5.1 Aliases used in `MFront`

In `MFront`, various aliases are introduced to ease the implementation of mechanical behaviours:

- `Stensor4` is an alias to `st2tost2<N,real>` where `N` is the space dimension and `real` is the numerical type used. The value of `N` depends of the current modelling hypothesis. The concrete type for `real` depends on the interface used.
- `StiffnessTensor` is an alias to `st2tost2<N,stress>`.

A.1.5.2 Special values

A.1.5.2.1 Special values of the `st2tost2` class

The `st2tost2` provides the following static methods:

- `Id`: returns the identity matrix.
- `IxI`: returns the tensor defined by $\underline{\underline{I}} \otimes \underline{\underline{I}}$. This tensor satisfies, for every symmetric tensor $\underline{s}$:

$$\underline{\underline{I}} \otimes \underline{\underline{I}} : \underline{s} = \mathrm{tr}(\underline{s})\,\underline{\underline{I}}$$

- `J`: returns the tensor defined by $\frac{1}{3}\,\underline{\underline{I}} \otimes \underline{\underline{I}}$. This tensor satisfies, for every symmetric tensor $\underline{s}$:

$$\underline{\underline{\mathbf{J}}} : \underline{s} = \frac{\mathrm{tr}(\underline{s})}{3}\,\underline{\underline{I}}$$

- `K`: returns the tensor defined by $\underline{\underline{I}} - \frac{1}{3}\,\underline{\underline{I}} \otimes \underline{\underline{I}}$. This is tensor is indeed the projector on the deviatoric space.
- `M`: returns the tensor defined by $\frac{3}{2}\left(\underline{\underline{I}} - \frac{1}{3}\,\underline{\underline{I}} \otimes \underline{\underline{I}}\right)$. This tensor appears in the definition of the von Mises stress:

$$\sigma_{\mathrm{eq}} = \sqrt{\underline{\sigma} : \underline{\underline{\mathbf{M}}} : \underline{\sigma}}$$

A.1.5.2.2 Special values of the `t2tot2` class

The `t2tot2` provides the following static method:

- `Id`: returns the identity matrix.
- `IxI`: returns the tensor defined by $\underset{\sim}{\mathbf{I}} \otimes \underset{\sim}{\mathbf{I}}$. This tensor satisfies, for every tensor $\underset{\sim}{\mathbf{a}}$:

$$\underset{\sim}{\mathbf{I}} \otimes \underset{\sim}{\mathbf{I}} : \underset{\sim}{\mathbf{a}} = \mathrm{tr}\left(\underset{\sim}{\mathbf{a}}\right) \underset{\sim}{\mathbf{I}}$$

- `J`: returns the tensor defined by $\frac{1}{3} \underset{\sim}{\mathbf{I}} \otimes \underset{\sim}{\mathbf{I}}$. This tensor satisfies, for every tensor $\underset{\sim}{\mathbf{a}}$:

$$\underset{\sim}{\mathbf{J}} : \underset{\sim}{\mathbf{a}} = \frac{\mathrm{tr}\left(\underset{\sim}{\mathbf{a}}\right)}{3} \underset{\sim}{\mathbf{I}}$$

- `K`: returns the tensor defined by $\underset{\sim}{\mathbf{I}} - \frac{1}{3} \underset{\sim}{\mathbf{I}} \otimes \underset{\sim}{\mathbf{I}}$. This is tensor is indeed the projector on the deviatoric space.

A.2 Standard operations

A.2.1 Basic operations

Thanks to operator overloading, the following operations are written using standard mathematical notations:

- Addition of two tensors.
- Subtraction of two tensors.
- Multiplication of two tensors. Be cautious of the fact that the multiplication of two symmetric tensors results in a non symmetric tensor.
- Multiplication and division of a tensor by a scalar.

For example, the following code shows how to perform the addition of two tensors `a` and `b`

```
const auto c=a+b;
```

A.2.1.1 Expression templates

For optimization purpose, the result of the previous operations are not evaluted. In the previous example, `c` is not the result but a special class built on the fly representing the addition of the tensors `a` and `b`.

The `eval` function can be used to explicitly evaluate the result of the operation.

```
const auto c=eval(a+b);
```

A.2.1.2 In-place operations

Operations like `b=b+a` can be also be written using operator `+=` as follows:

```
b+=a;
```

The operator `-=` is also available for operations like `b=b-a`.

The operators `*=` and `/=` are also available for inplace multiplication or division by a scalar :

```
// divide tensor a by 2
a/=2;
```

A.2.2 Symmetrization and unsymmetrization of second order tensors

A non symmetric second order tensor can be symmetrized using the `syme` function. The result match the `StensorConcept`.

A symmetric second order tensor can be unsymmetrized using the `unsyme` function. The result match the `TensorConcept`.

A.2.3 Frobenius inner product of second order tensors

The Frobenius inner product $\underset{\sim}{\mathbf{a}} : \underset{\sim}{\mathbf{b}}$ of two tensors $\underset{\sim}{\mathbf{a}}$ and $\underset{\sim}{\mathbf{b}}$ is defined by:

$$\underset{\sim}{\mathbf{a}} : \underset{\sim}{\mathbf{b}} = \mathrm{tr}\left(\underset{\sim}{\mathbf{a}}^T \cdot \underset{\sim}{\mathbf{b}}\right) = \sum_{i,j} a_{ij} b_{ij}$$

This operation is implemented in `TFEL/Math` using the `|` operator, as follows:

```
const auto r = a|b;
```

The user must be aware that this operator as a low priority in `C++`, so one must usually use parenthesis to properly evaluate operations involving those operations.

A.2.4 Diadic product

The diadic product $\underset{\sim}{\mathbf{a}} \otimes \underset{\sim}{\mathbf{b}}$ of two tensors $\underset{\sim}{\mathbf{a}}$ and $\underset{\sim}{\mathbf{b}}$ satisfies, for any tensor $\underset{\sim}{\mathbf{c}}$:

$$\begin{cases} \underset{\sim}{\mathbf{c}} : \left(\underset{\sim}{\mathbf{a}} \otimes \underset{\sim}{\mathbf{b}}\right) = \left(\underset{\sim}{\mathbf{c}} : \underset{\sim}{\mathbf{a}}\right) \underset{\sim}{\mathbf{b}} \\ \left(\underset{\sim}{\mathbf{a}} \otimes \underset{\sim}{\mathbf{b}}\right) : \underset{\sim}{\mathbf{c}} = \left(\underset{\sim}{\mathbf{c}} : \underset{\sim}{\mathbf{b}}\right) \underset{\sim}{\mathbf{a}} \end{cases}$$

The diadic product is implemented in `TFEL/Math` using operator `^`. The user must be aware that this operator as a low priority in `C++`, so one must usually use parenthesis to properly evaluate operations involving diadic products.

```
dfeel_ddeel += 2.*mu*theta*dp*iseq*(Stensor4::M()-(n^n));
```

The diadic product of two symmetric tensors results in an object matching the `ST2toST2Concept`.

The diadic product of two non symmetric tensors results in an object matching the `T2toT2Concept`.

A.2.5 Polar decomposition

The polar decomposition of a tensor `F` can be computed as follows in `MFront`:

```
tensor<N, real> R;
stensor<N, strain> U;
polar_decomposition(R, U, F);
```

where:

- `N` is the space dimension.
- `real` is an alias to the numeric type.
- `strain` is an alias to the numeric type.

A.2.6 Application of a fourth order tensor

A.2.7 Multiplication of second order tensors

A.2.7.1 Derivatives

A.2.8 Symmetric product of two symmetric second order tensors

The symmetric product of two symmetric second order tensors $\underline{a}$ and $\underline{b}$ can be defined as follows:

$$\underline{a} \cdot_s \underline{b} = \frac{1}{2}(\underline{a} \cdot \underline{b} + \underline{b} \cdot \underline{a})$$

This can be computed by the `symmetric_product` function.

A.2.8.1 Derivative

The derivative of the symmetric product $\underline{a} \cdot_s \underline{b}$ with respect to $\underline{a}$ can be computed using the `st2tost2::stpd` static method with takes $\underline{b}$ as argument.

A.2.9 Second symmetric product of two symmetric second order tensors

Another symmetric product of two symmetric second order tensors $\underline{a}$ and $\underline{b}$ can be defined as follows:

$$\underline{a} \cdot \underline{b} \cdot \underline{a}$$

This can be computed by the `symmetric_product_aba` function.

A.2.9.1 Derivative

The derivative of $\underline{a} \cdot \underline{b} \cdot \underline{a}$ with respect to $\underline{a}$ can be computed using the `symmetric_product_derivative_daba_da` function.

The derivative of $\underline{a} \cdot \underline{b} \cdot \underline{a}$ with respect to $\underline{b}$ can be computed using the `symmetric_product_derivative_daba_db` function.

A.2.9.2 Computation of the tensorial product of a tensor with itself

By definition, given a symmetric tensor $\underline{a}$, the tensor product $\underline{a}\,\overline{\otimes}\,\underline{a}$ is the fourth order tensor (of type `st2tost2`) which satisfies, for any tensor $\underline{b}$:

$$(\underline{a}\,\overline{\otimes}\,\underline{a}) : \underline{b} = \underline{a} \cdot \underline{b} \cdot \underline{a}$$

$\underline{a}\,\overline{\otimes}\,\underline{a}$ can thus readily be computed using the `symmetric_product_derivative_daba_db` function.

A.3 Special mathematical functions

A.3.1 Change the basis

The `change_basis` functions can:

- rotate a symmetric tensor
- rotate a (non-symmetric) tensor
- rotate a fourth order tensor of type `st2tost2`.
- rotate a fourth order tensor of type `t2tot2`.

Those functions takes two constant arguments: the object to be rotated and the rotation matrix. The rotated object is returned.

A.3.1.1 Example

```
const auto sr = change_basis(s,r);
```

A.3.1.2 Fourth order tensors standing for the rotation of tensors

The `st2tost2` class provide the `fromRotationMatrix` static method which computes a fourth order tensor which has the same effect on a symmetric tensor than applying a given rotation.

```
const auto rt = st2tost2<N,real>::fromRotationMatrix(r);
```

The `t2tot2` class provide the `fromRotationMatrix` static method which computes a fourth order tensor which has the same effect on a (non-symmetric) tensor than applying a given rotation.

```
const auto rt = t2tot2<N,real>::fromRotationMatrix(r);
```

> **Note**
> In pratice, the `fromRotationMatrix` static methods can be used to compute the rotation of any second and fourth order tensors (including the ones of the `t2tost2` and `st2tot2` types). They are used internally in the implementation of the `change_basis` functions provided for the fourth order tensors of types `st2tost2` and `t2tot2`.

A.3.2 Inverses

The `invert` functions can compute:

- the inverse of a symmetric tensor.
- the inverse of a tensor.
- the inverse of a fourth order tensor of type `st2tost2`.

Those functions takes the object to be inverted as constant argument and returned the inverse.

A.3.3 Square of a symmetric tensor

The product of two symmetric tensors is a non symmetric tensor. However, the square of a symmetric tensor is a symmetric tensor.

The square of a symmetric tensor can be computed using the `square` function, as follows:

```
const auto s2 = square(s);
```

A.3.3.1 Derivative of the square of a symmetric tensor

The derivative of the square of a symmetric tensor is a fourth order tensor mapping a symmetric tensor toward a symmetric tensor. It can be computed using the `dsquare` function, as follows:

```
const auto ds2_ds = st2tost2<N,real>::dsquare(s);
```

The result of this operation of mostly filled with zero. If $\underline{s}$ is a function of $\underline{c}$, this fact can be used to optimize the computation of the derivative of $\underline{s}^2$ with respect to $\underline{c}$:

```
const auto ds2_dc = st2tost2<N,real>::dsquare(s2,ds_dc);
```

which is more efficient than:

```
const auto ds2_dc = st2tost2<N,real>::dsquare(s2)*ds_dc;
```

A.3.4 Positive and negative parts of a symmetric tensor

The Positive and negative parts of a symmetric tensor can be computed respectively by the `positive_part` and `negative_part` function.

A.3.5 Transposition

A.3.5.1 Transposition of a second order tensor

A non symmetric second order tensor can be transpose using the `transpose` function:

```
const auto B = transpose(A);
```

A.3.5.1.1 Derivative of the transpose of a second order tensor The linear operation which turns a second order tensor into its transpose can be retrieved using the static method `transpose_derivative` of the `t2tot2` class as follows:

```
const auto dtA_dA = t2tot2<real,N>::transpose_derivative();
```

As its name suggests, this linear operation is also the derivative of the transpose of a second order tensor with respect to itself.

A.3.5.2 Transposition of a fourth order tensor

A fourth order tensor matching the `ST2toST2Concept` (i.e. a linear form mapping a symmetric second order tensor to a symmetric second order tensor) can be transposed using the `transpose` function:

```
const auto B = transpose(A);
```

A.3.6 Second order tensor invariants

A.3.6.1 Defintion

The three invariants of a second order tensor are defined by:

$$\begin{cases} I_1 = \mathrm{tr}\left(\underset{\sim}{\mathbf{a}}\right) \\ I_2 = \frac{1}{2}\left(\left(\mathrm{tr}\left(\underset{\sim}{\mathbf{a}}\right)\right)^2 - \mathrm{tr}\left(\underset{\sim}{\mathbf{a}}^2\right)\right) \\ I_3 = \det\left(\underset{\sim}{\mathbf{a}}\right) \end{cases}$$

A.3.7 Cauchy-Green tensor and derivative

The `computeRightCauchyGreenTensor` computes the right Cauchy-Green symmetric tensor $\underline{C}$ associated with a non symmetric tensor $\underset{\sim}{\mathbf{F}}$:

$$\underline{C} = \underset{\sim}{\mathbf{F}}^T \cdot \underset{\sim}{\mathbf{F}}$$

The derivative of $\underline{C}$ with respect to $\underset{\sim}{\mathbf{F}}$ can be computed using the `t2tost2::dCdF` static method.‘

A.3.8 Left Cauchy-Green tensor and derivative

The `computeLeftCauchyGreenTensor` computes the left Cauchy-Green symmetric tensor $\underline{B}$ associated with a non symmetric tensor $\underset{\sim}{\mathbf{F}}$:

$$\underline{B} = \underset{\sim}{\mathbf{F}} \cdot \underset{\sim}{\mathbf{F}}^T$$

The derivative of $\underline{B}$ with respect to $\underset{\sim}{\mathbf{F}}$ can be computed using the `t2tost2::dBdF` static method.‘

A.3.8.1 Computation

I_1 can be computed thanks to `trace` function as follows:

```
const auto I1 = trace(A);
```

Of course, I_1 can also be computed directly by accessing the components of the tensor, i.e. :

```
const auto I1 = A(0)+A(1)+A(2);
```

I_2 can be computed by translating its definition as follows:

```
const auto I2 = (I1*I1-trace(square(A)))/2;
```

I_3 can be computed thanks to `det` function as follows:

```
const auto I3 = det(A);
```

A.3.8.2 Derivatives of the invariants of a tensor

The derivative of the invariants are classically given by:

$$\begin{cases} \dfrac{\partial I_1}{\partial \underset{\sim}{\mathbf{a}}} = \underline{I} \\ \dfrac{\partial I_2}{\partial \underset{\sim}{\mathbf{a}}} = I_1\,\underline{I} - \underset{\sim}{\mathbf{a}}^T \\ \dfrac{\partial I_3}{\partial \underset{\sim}{\mathbf{a}}} = \det\left(\underset{\sim}{\mathbf{a}}\right)\underset{\sim}{\mathbf{a}}^{-T} = \left(\underset{\sim}{\mathbf{a}}^2 - I_1\,\underset{\sim}{\mathbf{a}} + I_2\,\underline{I}\right)^T \end{cases}$$

Those expressions are simplier for symmetric tensors:

$$\begin{cases} \dfrac{\partial I_1}{\partial \underline{s}} = \underline{I} \\ \dfrac{\partial I_2}{\partial \underline{s}} = I_1\,\underline{I} - \underline{s} \\ \dfrac{\partial I_3}{\partial \underline{s}} = \det\left(\underline{s}\right)\underline{s}^{-1} = \underline{s}^2 - I_1\,\underline{s} + I_2\,\underline{I} \end{cases}$$

$\dfrac{\partial I_1}{\partial \underset{\sim}{\mathbf{a}}}$ and $\dfrac{\partial I_2}{\partial \underset{\sim}{\mathbf{a}}}$ are trivial to compute.

$\dfrac{\partial I_3}{\partial \underset{\sim}{\mathbf{a}}}$ can be computed using the `computeDeterminantDerivative` function as follows:

```
const auto dI3_dA = computeDeterminantDerivative(A);
```

A.3.8.3 Second derivatives of the invariants of a tensor

$\dfrac{\partial^2 I_1}{\partial \underset{\sim}{\mathbf{a}}^2}$ is null.

$\dfrac{\partial^2 I_2}{\partial \underline{s}^2}$ can be computed as follows:

$$\frac{\partial^2 I_2}{\partial \underline{s}^2} = \underline{I}\otimes\underline{I} - \frac{\partial \underset{\sim}{\mathbf{a}}^T}{\partial \underset{\sim}{\mathbf{a}}}$$

The last term, $\dfrac{\partial \underset{\sim}{\mathbf{a}}^T}{\partial \underset{\sim}{\mathbf{a}}}$ can be computed using the `t2tot2::transpose_derivative` static method, see Paragraph A.3.5 for details.

```
const auto id = tensor<N,real>::Id();
const auto d2I2_dA2 = (id^id)-t2tot2<N,real>::transpose_derivative();
```

For symmetric tensors, this computation is much simplier:

```
const auto id = stensor<N,real>::Id();
const auto d2I2_dA2 = (id^id)-st2tot2<N,real>::Id();
```

The $\dfrac{\partial^2 I_3}{\partial \underset{\sim}{\mathbf{a}}^2}$ term can be computed using the `computeDeterminantSecondDerivative` function, as follows:

```
const auto d2I3_dA2 = computeDeterminantSecondDerivative(A);
```

A.3.9 Invariants of the stress deviator tensor [43]

Let $\underline{\sigma}$ be a stress tensor. Its deviatoric part $\underline{s}$ is:

$$\underline{s} = \underline{\sigma} - \frac{1}{3}\,\mathrm{tr}(\underline{\sigma})\,\underline{I} = \left(\underline{\underline{\mathbf{I}}} - \frac{1}{3}\,\underline{I}\otimes\underline{I}\right) : \underline{\sigma}$$

The deviator of a tensor can be computed using the `deviator` function.

As it is a second order tensor, the stress deviator tensor also has a set of invariants, which can be obtained using the same procedure used to calculate the invariants of the stress tensor. It can be shown that the principal directions of the stress deviator tensor s_{ij} are the same as the principal directions of the stress tensor σ_{ij}. Thus, the characteristic equation is

$$\left|s_{ij} - \lambda\delta_{ij}\right| = -\lambda^3 + J_1\lambda^2 - J_2\lambda + J_3 = 0,$$

where J_1, J_2 and J_3 are the first, second, and third *deviatoric stress invariants*, respectively. Their values are the same (invariant) regardless of the orientation of the coordinate system chosen. These deviatoric stress invariants can be expressed as a function of the components of s_{ij} or its principal values s_1, s_2, and s_3, or alternatively, as a function of σ_{ij} or its principal values σ_1, σ_2, and σ_3. Thus,

$$
\begin{aligned}
J_1 &= s_{kk} = 0, \\
J_2 &= \tfrac{1}{2} s_{ij} s_{ji} = \frac{1}{2}\mathrm{tr}(\underline{s}^2) \\
&= \frac{1}{2}(s_1^2 + s_2^2 + s_3^2) \\
&= \frac{1}{6}\left[(\sigma_{11}-\sigma_{22})^2 + (\sigma_{22}-\sigma_{33})^2 + (\sigma_{33}-\sigma_{11})^2\right] + \sigma_{12}^2 + \sigma_{23}^2 + \sigma_{31}^2 \\
&= \frac{1}{6}\left[(\sigma_1-\sigma_2)^2 + (\sigma_2-\sigma_3)^2 + (\sigma_3-\sigma_1)^2\right] \\
&= \frac{1}{3} I_1^2 - I_2 = \frac{1}{2}\left[\mathrm{tr}(\underline{\sigma}^2) - \frac{1}{3}\mathrm{tr}(\underline{\sigma})^2\right], \\
J_3 &= \det(\underline{s}) \\
&= \frac{1}{3} s_{ij} s_{jk} s_{ki} = \frac{1}{3}\mathrm{tr}(\underline{s}^3) \\
&= \frac{1}{3}(s_1^3 + s_2^3 + s_3^3) \\
&= s_1 s_2 s_3 \\
&= \frac{2}{27} I_1^3 - \frac{1}{3} I_1 I_2 + I_3 = \frac{1}{3}\left[\mathrm{tr}(\underline{\sigma}^3) - \mathrm{tr}(\underline{\sigma}^2)\mathrm{tr}(\underline{\sigma}) + \frac{2}{9}\mathrm{tr}(\underline{\sigma})^3\right].
\end{aligned}
$$

where I_1, I_2 and I_3 are the invariants of $\underline{\sigma}$ (see Section A.3.6).

The invariants J_2 and J_3 of deviatoric part of the stress are the basis of many isotropic yield criteria, some of them being described below.

A.3.9.1 First derivative

This paragraph details the first derivative of J_2 and J_3 with respect to $\underline{\sigma}$.

The computation of $\dfrac{\partial J_2}{\partial \underline{\sigma}}$ is straight-forward by chain rule, using the expression of the derivatives of the invariants of a tensor (see Section A.3.8.2):

$$
\frac{\partial J_2}{\partial \underline{\sigma}} = \frac{\partial J_2}{\partial \underline{s}} \cdot \frac{\partial \underline{s}}{\partial \underline{\sigma}} = \underline{s}
$$

In pratice, this can be implemented as follows:

```
const auto dJ2 = deviator(sig);
```

For the expression of $\dfrac{\partial J_3}{\partial \underline{\sigma}}$, one can derive its expression based of the three invariants of $\underline{\sigma}$, as follows:

$$
\begin{aligned}
\frac{\partial J_3}{\partial \underline{\sigma}} &= \frac{\partial}{\partial \underline{\sigma}}\left(\frac{2}{27} I_1^3 - \frac{1}{3} I_1 I_2 + I_3\right) \\
&= \frac{2}{9} I_1^2 \underline{I} - \frac{1}{3}\left[I_2 \underline{I} + I_1 \frac{\partial I_2}{\partial \underline{\sigma}}\right] + \frac{\partial I_3}{\partial \underline{\sigma}}
\end{aligned}
$$

For `TFEL` versions prior to 3.2, one can implement this formula as follows:

```
constexpr const auto id = stensor<N,real>::Id();
const auto I1  = trace(sig);
const auto I2  = (I1*I1-trace(square(sig)))/2;
const auto dI2 = I1*id-sig;
const auto dI3 = computeDeterminantDerivative(sig);
const auto dJ3 = eval((2*I1*I1/9)*id-(I2*id+I1*dI2)/3+dI3);
```

For `TFEL` versions greater than 3.2, one may want to use the optimised `computeDeviatorDeterminantDerivative` function, defined in the namespace `tfel::math`, as follows:

```
const auto dJ3 = computeDeviatorDeterminantDerivative(sig);
```

The `computeJ3Derivative`, defined in `tfel::material` namespace, is a synonym for the `computeDeviatorDeterminantDerivative` function.

A.3.9.2 Second derivative

This paragraph details the second derivatives of J_2 and J_3 with respect to $\underline{\sigma}$.

The second derivative $\dfrac{\partial^2 J_2}{\partial \underline{\sigma}^2}$ is straight-forward:

$$\frac{\partial^2 J_2}{\partial \underline{\sigma}^2} = \underline{\underline{\mathbf{I}}} - \frac{1}{3}\,\underline{I} \otimes \underline{I}$$

It can be readily implemented:

```
constexpr const auto id  = stensor<N,real>::Id();
constexpr const auto id4 = st2tost2<N,real>::Id();
const auto d2J2 = eval(id4-(id^id)/3);
```

The second derivative $\dfrac{\partial^2 J_3}{\partial \underline{\sigma}^2}$ is also straight-forward to compute (see also Section A.3.8.2):

$$\frac{\partial J_3}{\partial \underline{\sigma}} = \frac{4}{9}\,I_1\,\underline{I} \otimes \underline{I} - \frac{1}{3}\left[\underline{I} \otimes \frac{\partial I_2}{\partial \underline{\sigma}} + \frac{\partial I_2}{\partial \underline{\sigma}} \otimes \underline{I} + I_1\,\frac{\partial^2 I_2}{\partial \underline{\sigma}^2}\right] + \frac{\partial^2 I_3}{\partial \underline{\sigma}^2}$$

For `TFEL` versions prior to 3.2, one can implement this formula as follows:

```
constexpr const auto id = stensor<N,real>::Id();
constexpr const auto id4 = st2tost2<N,real>::Id();
const auto I1   = trace(sig);
const auto I2   = (I1*I1-trace(square(sig)))/2;
const auto dI2  = I1*id-sig;
const auto d2I2 = (id^id)-id4;
const auto d2I3 = computeDeterminantSecondDerivative(sig);
const auto d2J3 = eval((4*I1/9)*(id^id)-((id^dI2)+(dI2^id)+i1*d2I2)/3+d2I3);
```

For `TFEL` versions greater than 3.2, one may want to use the optimised `computeDeviatorDeterminantSecondDerivative` function, defined in the `tfel:math` namespace, as follows:

```
const auto d2J3 = computeDeviatorDeterminantSecondDerivative(J);
```

The `computeJ3SecondDerivative`, defined in `tfel::material` namespace, is a synonym for the `computeDeviatorDeterminantSecon` function.

A.3.10 Orthotropic generalization of the invariants of the stress deviator tensor

Within the framework of the theory of representation, generalizations to orthotropic conditions of the invariants of the deviatoric stress have been proposed by Cazacu and Barlat (see [26]):

- The generalization of J_2 is denoted J_2^O. It is defined by:

$$J_2^O = a_6\,s_{yz}^2 + a_5\,s_{xz}^2 + a_4\,s_{xy}^2 + \frac{a_2}{6}\,(s_{yy}-s_{zz})^2 + \frac{a_3}{6}\,(s_{xx}-s_{zz})^2 + \frac{a_1}{6}\,(s_{xx}-s_{yy})^2$$

  where the $a_i\big|_{i\in[1:6]}$ are six coefficients describing the orthotropy of the material.

- The generalization of J_3 is denoted J_3^O. It is defined by:

$$
\begin{aligned}
J_3^O = & \frac{1}{27}\left(b_1+b_2\right)s_{xx}^3+\frac{1}{27}\left(b_3+b_4\right)s_{yy}^3+\frac{1}{27}\left(2\left(b_1+b_4\right)-b_2-b_3\right)s_{zz}^3 \\
& -\frac{1}{9}\left(b_1\,s_{yy}+b_2 s_{zz}\right)s_{xx}^2 \\
& -\frac{1}{9}\left(b_3\,s_{zz}+b_4\,s_{xx}\right)s_{yy}^2 \\
& -\frac{1}{9}\left(\left(b_1-b_2+b_4\right)s_{xx}+\left(b_1-b3+b_4\right)s_{yy}\right)s_{zz}^3 \\
& +\frac{2}{9}\left(b_1+b_4\right)s_{xx}\,s_{yy}\,s_{zz} \\
& -\frac{s_{xz}^2}{3}\left(2\,b_9\,s_{yy}-b_8\,s_{zz}-\left(2\,b_9-b_8\right)s_{xx}\right) \\
& -\frac{s_{xy}^2}{3}\left(2\,b_{10}\,s_{zz}-b_5\,s_{yy}-\left(2\,b_{10}-b_5\right)s_{xx}\right) \\
& -\frac{s_{yz}^2}{3}\left(\left(b_6+b_7\right)s_{xx}-b_6\,s_{yy}-b_7\,s_{zz}\right) \\
& +2\,b_{11}\,s_{xy}\,s_{xz}\,s_{yz}
\end{aligned}
$$

  where the $b_i|_{i\in[1:11]}$ are eleven coefficients describing the orthotropy of the material.

Those invariants may be used to generalize isotropic yield criteria based on J_2 and J_3 invariants to orthotropy.

J_2^0, J_3^0 and their first and second derivatives with respect to the stress tensor $\underline{\sigma}$ can be computed by the following functions:

- `computesJ2O`, `computesJ2ODerivative` and `computesJ2OSecondDerivative`.
- `computesJ3O`, `computesJ3ODerivative` and `computesJ3OSecondDerivative`.

Those functions take the stress tensor as first argument and each orthotropic coefficients. Each of those functions has an overload taking the stress tensor as its firs arguments and a tiny vector (`tfel::math::tvector`) containing the orthotropic coefficients.

A.3.11 Eigenvalues, eigenvectors and eigentensors of symmetric tensors

A.3.11.1 Eigenvalue

The eigenvalues can be computed by the `computeEigenValues` method, as follows:

```
const auto vp = s.computeEigenValues();
```

> **Note**
> In `2D`, the last eigenvalue always corresponds to the out-of-plane direction.

Those eigen values can be ordered by using one of the following argument:

- `ASCENDING`: eigenvalues are sorted from the lowest to the greatest.
- `DESCENDING`: eigenvalues are sorted from the greatest to the lowest.

```
const auto vp = s.computeEigenValues(stensor::ASCENDING);
```

> **Note**
> In `1D`, the sorting parameter has no effect. In `2D`, the last eigenvalue always corresponds to the out-of-plane direction.

By default, the eigenvalues are computed using Cardano formula. However, one may use one of the following eigensolver decribed in the next paragraph as follows:

```
constexpr const auto es = stensor<3u,real>::FSESQLEIGENSOLVER;
const auto vp = s.computeEigenValues<es>();
```

A.3.11.2 Eigenvectors

The default eigen solver for symmetric tensors used in TFEL is based on analitical computations of the eigen values and eigen vectors. Such computations are more efficient but less accurate than the iterative Jacobi algorithm (see [44, 45]).

With the courtesy of Joachim Kopp, we have created a C++11 compliant version of his routines that we gathered in header-only library called FSES (Fast Symmetric Eigen Solver). This library is included with TFEL and provides the following algorithms:

- Jacobi
- QL with implicit shifts
- Cuppen
- Analytical
- Hybrid
- Householder reduction

We have also introduced the Jacobi implementation of the Geometric Tools library (see [46, 47]).

Those algorithms are available in 3D. For 2D symmetric tensors, we fall back to some default algorithm as described below.

Table A.1: List of available eigen solvers.

Name	Algorithm in 3D	Algorithm in 2D
TFELEIGENSOLVER	Analytical (TFEL)	Analytical (TFEL)
FSESJACOBIEIGENSOLVER	Jacobi	Analytical (FSES)
FSESQLEIGENSOLVER	QL with implicit shifts	Analytical (FSES)
FSESCUPPENEIGENSOLVER	Cuppen's Divide & Conquer	Analytical (FSES)
FSESANALYTICALEIGENSOLVER	Analytical	Analytical (FSES)
FSESHYBRIDEIGENSOLVER	Hybrid	Analytical (FSES)
GTESYMMETRICQREIGENSOLVER	Symmetric QR	Analytical (TFEL)

The various eigen solvers available are enumerated in Table A.1.

The eigen solver is passed as a template argument of the computeEigenValues or the computeEigenVectors methods as illustrated in the code below:

```
tmatrix<3u,3u,real> m2;
tvector<3u,real>    vp2;
std::tie(vp,m)=s.computeEigenVectors<stensor::GTESYMMETRICQREIGENSOLVER>();
```

> **Note**
> In 2D, the last eigenvector always corresponds to the out-of-plane direction.

The computeEigenVectors method can also order the eigenvalues in ascending or descending order, see the computeEigenValues method for details.

> **Note**
> In 1D, the ordering parameter has no effect. In 2D, the last eigenvector always corresponds to the out-of-plane direction.

A.3.12 Isotropic functions of a symmetric tensor

Given a scalar valuated function f, one can define an associated isotropic function for symmetric tensors as:

$$f(\underline{s}) = \sum_{i=1}^{3} f(\lambda_i)\underline{n}_i$$

where $\lambda_i\big|_{i\in[1,2,3]}$ are the eigen values of the symmetric tensor $\underline{s}$ and $\underline{n}_i\big|_{i\in[1,2,3]}$ the associated eigen tensors.

If f is $\mathcal{C}^1$, then f is a differentiable function of $\underline{s}$.

f can be computed with the `computeIsotropicFunction` method of the stensor class. $\dfrac{\partial f}{\partial \underline{s}}$ can be computed with `computeIsotropicFunctionDerivative`. One can also compute f and $\dfrac{\partial f}{\partial \underline{s}}$ all at once by the `computeIsotropicFunctionAndDerivative` method. All those methods are templated by the name of the eigen solver (if no template parameter is given, the `TFELEIGENSOLVER` is used).

Various new overloaded versions of those methods have been introduced. Those overloaded methods are meant to:

- allow the user to explicitly give the values of f or df, rather than the functions to compute them. This allows to reduce the computational cost of the evaluation of the isotropic function when the values of the derivatives can directly be computed from the values of f. See the example exp example below.
- return the results by value. This allow a much more readable code if the *structured bindings* feature of the `C++17` standard is available.

To illustrate this new features, assuming that the *structured bindings* feature of the `C++17` standard is available, one can now efficiently evaluate the exponential of a symmetric tensor and its derivative as follows:

```
const auto [vp,m] = s.computeEigenVectors();
const auto evp    = map([](const auto x){return exp(x)},vp);
const auto [f,df] = Stensor::computeIsotropicFunctionAndDerivative(evp,evp,vp,m,1.e-12);
```

A.4 Special operations for mechanical behaviours

A.4.1 Yield criteria

A.4.1.1 von Mises stress [48]

The von Mises stress is defined by:

$$\sigma_{\text{eq}} = \sqrt{\frac{3}{2}\,\underline{s} : \underline{s}} = \sqrt{3\,J_2}$$

where:

- $\underline{s}$ is the deviatoric stress defined as follows:

$$\underline{s} = \underline{\sigma} - \frac{1}{3}\,\text{tr}(\underline{\sigma})\,\underline{I}$$

- J_2 is the second invariant of $\underline{s}$ (see also Section A.3.9).

The previous expression can be rewritten by introducing a fourth order tensor called $\underline{\underline{\mathbf{M}}}$:

$$\sigma_{\text{eq}} = \sqrt{\sigma : \underline{\underline{\mathbf{M}}} : \underline{\sigma}}$$

The tensor $\underline{\underline{\mathbf{M}}}$ is given by:

$$\underline{\underline{\mathbf{M}}} = \frac{3}{2}\left(\underline{\underline{\mathbf{I}}} - \frac{1}{3}\,\underline{I} \otimes \underline{I}\right)$$

The tensor $\underline{\underline{\mathbf{M}}}$ is accessible through the `M constexpr static` method of the `st2tost2` class, as follows:

```
constexpr const auto M = st2tost2<N,real>::M();
```

In terms of the eigenvalues of the stress, denoted by σ_1, σ_2 and σ_3, the von Mises stress can also be defined by:

$$\sigma_{\text{eq}} = \sqrt{\frac{1}{2}\left(\left|\sigma_1-\sigma_2\right|^2 + \left|\sigma_1-\sigma_3\right|^2 + \left|\sigma_2-\sigma_3\right|^2\right)}$$

The von Mises stress can be computed using the `sigmaeq` function, as follows:

```
const auto seq = sigmaeq(s);
```

The derivative $\underline{n}$ of the von Mises stress with respect to the stress is called the normal and is given by:

$$\underline{n} = \frac{\partial \sigma_{\mathrm{eq}}}{\partial \underline{\sigma}} = \frac{3}{2\,\sigma_{\mathrm{eq}}}\,\underline{s} = \frac{1}{\sigma_{\mathrm{eq}}}\,\underline{\underline{M}} : \underline{\sigma}$$

The normal can be computed by:

```
const auto n = eval(3*deviator(sig)/(2*seq));
```

Note The `eval` function is used to evaluate the normal. Otherwise, the expression template mechanism used by `TFEL` would delay its evaluation.

Another way to compute it is to use the $\underline{\underline{\mathbf{M}}}$ tensor, as follows:

```
const auto n = eval(M*sig/seq);
```

The second derivative of the von Mises stress with respect to the von Mises stress is given by:

$$\frac{\partial^2 \sigma_{\mathrm{eq}}}{\partial \underline{\sigma}^2} = \frac{1}{\sigma_{\mathrm{eq}}}(\underline{\underline{M}} - \underline{n} \otimes \underline{n})$$

This second derivative can be computed as follows:

```
const auto dn_ds = eval((M-(n^n))/seq);
```

A.4.1.2 Hill stress

The Hill tensor $\underline{\underline{\mathbf{H}}}$ is defined by:

$$\underline{\underline{\mathbf{H}}} = \begin{pmatrix} F+H & -F & -H & 0 & 0 & 0 \\ -F & G+F & -G & 0 & 0 & 0 \\ -H & -G & H+G & 0 & 0 & 0 \\ 0 & 0 & 0 & L & 0 & 0 \\ 0 & 0 & 0 & 0 & M & 0 \\ 0 & 0 & 0 & 0 & 0 & N \end{pmatrix}$$

The Hill stress σ_{eq}^H is defined by:

$$\begin{aligned} \sigma_{\mathrm{eq}}^H &= \sqrt{\underline{\sigma} : \underline{\underline{\mathbf{H}}} : \underline{\sigma}} \\ &= \sqrt{F\left(\sigma_{11}-\sigma_{22}\right)^2 + G\left(\sigma_{22}-\sigma_{33}\right)^2 + H\left(\sigma_{33}-\sigma_{11}\right)^2 + 2\,L\sigma_{12}^2 + 2\,M\sigma_{13}^2 + 2\,N\sigma_{23}^2} \end{aligned}$$

Warning This convention is given in the book of Lemaître et Chaboche and seems to differ from the one described in most other books.

The first derivative of the Hill stress is given by:

$$\underline{n}^H = \frac{\partial \sigma_{\mathrm{eq}}^H}{\partial \underline{\sigma}} = \frac{1}{\sigma_{\mathrm{eq}}^H}\,\underline{\underline{H}} : \underline{\sigma}$$

The second derivative of the Hill stress is given by:

$$\frac{\partial^2 \sigma_{\mathrm{eq}}^H}{\partial \underline{\sigma}^2} = \frac{1}{\sigma_{\mathrm{eq}}^H}\left(\underline{\underline{H}} - \underline{n}^H \otimes \underline{n}^H\right)$$

The header `TFEL/Material/Hill.hxx` introduces various functions to build the Hill tensor:

- `hillTensor` or `makeHillTensor`, which has:
  - two template parameters: the space dimension and the underlying numeric type.
  - the six arguments giving the Hill coefficients F, G, H, L, M, N.

- `computeHillTensor` or `makeHillTensor`, which has:
  - three template parameters: the modelling hypothesis, the orthotropic axis convention, and the underlying numeric type.
  - the six arguments giving the Hill coefficients F, G, H, L, M, N.

Note In `MFront`, one shall use the `@HillTensor` to compute the `Hill` tensor, which takes into account the modelling hypothesis and the orthotropic axis convention.

A.4.1.3 Hosford stress

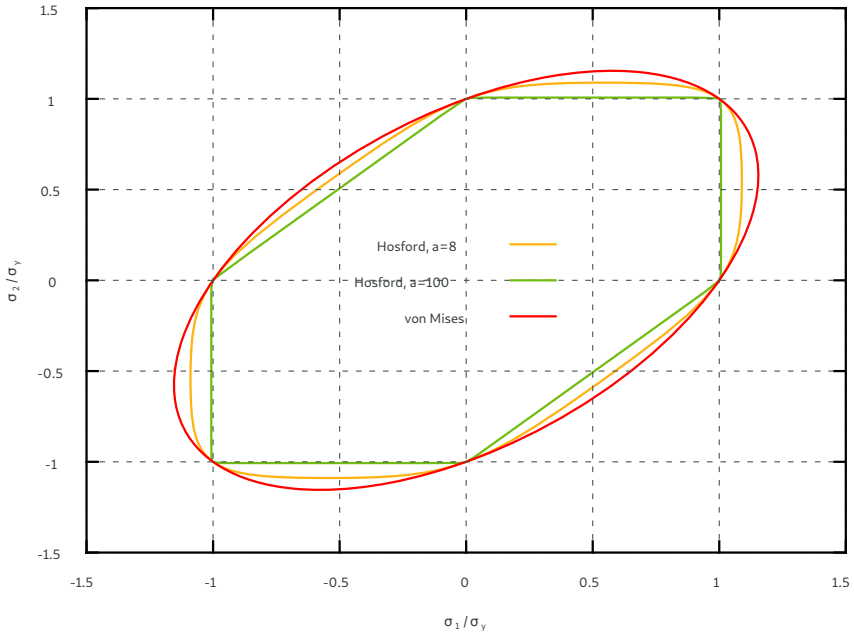

Figure A.1: Comparison of the Hosford stress $a = 100, a = 8$ and the von Mises stress

The header `TFEL/Material/Hosford1972YieldCriterion.hxx` introduces three functions which are meant to compute the Hosford equivalent stress and its first and second derivatives. *This header is automatically included by* `MFront`.

The Hosford equivalent stress is defined by (see [24]):

$$\sigma_{\text{eq}}^{H} = \sqrt[a]{\frac{1}{2}\left(\left|\sigma_1-\sigma_2\right|^a+\left|\sigma_1-\sigma_3\right|^a+\left|\sigma_2-\sigma_3\right|^a\right)}$$

where σ_1, σ_2 and σ_3 are the eigenvalues of the stress.

Therefore, when a goes to infinity, the Hosford stress reduces to the Tresca stress. When $n = 2$ the Hosford stress reduces to the von Mises stress.

The following function has been implemented:

- `computeHosfordStress`: return the Hosford equivalent stress
- `computeHosfordStressNormal`: return a tuple containg the Hosford equivalent stress and its first derivative (the normal)
- `computeHosfordStressSecondDerivative`: return a tuple containg the Hosford equivalent stress, its first derivative (the normal) and the second derivative.

The implementation of those functions are greatly inspired by the work of Scherzinger (see [49]). In particular, great care is given to avoid overflows in the computations of the Hosford stress.

Those functions have two template parameters:

- the type of symmetric tensors used for the stress tensor (automatically deduced, but required if the second parameter is specified).
- the eigen solver to be used (See Section A.3.11.2).

A.4.1.4 Barlat stress

The Barlat equivalent stress is defined as follows (See [27]):

$$\sigma_{\mathrm{eq}}^{B}=\sqrt[a]{\frac{1}{4}\left(\sum_{i=0}^{3}\sum_{j=0}^{3}\left|s_i'-s_j''\right|^{a}\right)}$$

where s_i' and s_i'' are the eigenvalues of two transformed stresses $\underline{s}'$ and $\underline{s}''$ by two linear transformation $\underline{\underline{\mathbf{L}}}'$ and $\underline{\underline{\mathbf{L}}}''$:

$$\begin{cases}\underline{s}'=\underline{\underline{\mathbf{L}}}':\underline{\sigma}\\ \underline{s}''=\underline{\underline{\mathbf{L}}}'':\underline{\sigma}\end{cases}$$

The linear transformations $\underline{\underline{\mathbf{L}}}'$ and $\underline{\underline{\mathbf{L}}}''$ are defined by 9 coefficients (each) which describe the material orthotropy. There are defined through auxiliary linear transformations $\underline{\underline{\mathbf{C}}}'$ and $\underline{\underline{\mathbf{C}}}''$ as follows:

$$\begin{aligned}\underline{\underline{\mathbf{L}}}'&=\underline{\underline{\mathbf{C}}}':\underline{\underline{\mathbf{M}}}\\ \underline{\underline{\mathbf{L}}}''&=\underline{\underline{\mathbf{C}}}'':\underline{\underline{\mathbf{M}}}\end{aligned}$$

where $\underline{\underline{\mathbf{M}}}$ is the transformation of the stress to its deviator:

$$\underline{\underline{\mathbf{M}}}=\underline{\underline{\mathbf{I}}}-\frac{1}{3}\underline{I}\otimes\underline{I}$$

The linear transformations $\underline{\underline{\mathbf{C}}}'$ and $\underline{\underline{\mathbf{C}}}''$ of the deviator stress are defined as follows:

$$\underline{\underline{\mathbf{C}}}'=\frac{1}{3}\begin{pmatrix}0&-c_{12}'&-c_{13}'&0&0&0\\-c_{21}'&0&-c_{23}'&0&0&0\\-c_{31}'&-c_{32}'&0&0&0&0\\0&0&0&c_{44}'&0&0\\0&0&0&0&c_{55}'&0\\0&0&0&0&0&c_{66}'\end{pmatrix}\quad\text{and}\quad\underline{\underline{\mathbf{C}}}''=\begin{pmatrix}0&-c_{12}''&-c_{13}''&0&0&0\\-c_{21}''&0&-c_{23}''&0&0&0\\-c_{31}''&-c_{32}''&0&0&0&0\\0&0&0&c_{44}''&0&0\\0&0&0&0&c_{55}''&0\\0&0&0&0&0&c_{66}''\end{pmatrix}$$

The following function has been implemented:

- `computeBarlatStress`: return the Barlat equivalent stress.
- `computeBarlatStressNormal`: return a tuple containg the Barlat equivalent stress and its first derivative (the normal).
- `computeBarlatStressSecondDerivative`: return a tuple containg the Barlat equivalent stress, its first derivative (the normal) and the second derivative.

The implementation of those functions are greatly inspired by the work of Scherzinger (see [49]). In particular, great care is given to avoid overflows in the computations of the Barlat stress.

Those functions have two template parameters:

- the type of symmetric tensors used for the stress tensor (automatically deduced, but required if the second parameter is specified).
- the eigen solver to be used (See Section A.3.11.2).

A.4.1.4.1 Linear transformations To define the linear transformations, the `makeBarlatLinearTransformation` function has been introduced. This function takes two template parameter:

- the space dimension (1, 2, and 3)
- the numeric type used (automatically deduced)

This functions takes the 9 coefficients as arguments, as follows:

```
const auto l1 = makeBarlatLinearTransformation<3>(c_12,c_21,c_13,c_31,
                                                  c_23,c_32,c_44,c_55,c_66);
```

Note In his paper, Barlat and coworkers seems to use the following convention for storing symmetric tensors:

$$(xx \quad yy \quad zz \quad yz \quad zx \quad xy)$$

which is not consistent with the `TFEL`/`Cast3M`/`Abaqus`/`Ansys` conventions:

$$(xx \quad yy \quad zz \quad xy \quad xz \quad yz)$$

Therefore, if one wants to uses coeficients c^B given by Barlat, one shall call this function as follows:

```
const auto l1 = makeBarlatLinearTransformation<3>(cB_12,cB_21,cB_13,cB_31,
                                                  cB_23,cB_32,cB_66,cBB_55,cBB_44);
```

The `TFEL/Material` library also provide an overload of the `makeBarlatLinearTransformation` which template parameters are the modelling hypothesis and the orthotropic axis conventions. The purpose of this overload is to swap appriopriate coefficients to get a consistent definition of the linear transforamtions for all the modelling hypotheses.

List of Figures

List of Figures

List of tables

List of Tables

References

1. Helfer, Thomas, Michel, Bruno, Proix, Jean-Michel, Salvo, Maxime, Sercombe, Jérôme and Casella, Michel. Introducing the open-source mfront code generator: Application to mechanical behaviours and material knowledge management within the PLEIADES fuel element modelling platform. *Computers & Mathematics with Applications*. 2015. DOI 10.1016/j.camwa.2015.06.027. Available from: http://www.sciencedirect.com/science/article/pii/S0898122115003132

2. CEA, EDF. TFEL/MFront website. 2022. Available from: https://thelfer.github.io/tfel/web/index.html

3. Helfer, Thomas, Bleyer, Jeremy, Frondelius, Tero, Yashchuk, Ivan, Nagel, Thomas and Naumov, Dmitri. The 'MFrontGenericInterfaceSupport' project. *Journal of Open Source Software*. 2020. Vol. 5, no. 48, p. 2003. DOI 10.21105/joss.02003. Available from: https://doi.org/10.21105/joss.02003
Publisher: The Open Journal

4. Bilke, Lars, Flemisch, Bernd, Kalbacher, Thomas, Kolditz, Olaf, Helmig, Rainer and Nagel, Thomas. Development of Open-Source Porous Media Simulators: Principles and Experiences. *Transport in Porous Media*. October 2019. Vol. 130, no. 1, p. 337–361. DOI 10.1007/s11242-019-01310-1. Available from: http://link.springer.com/10.1007/s11242-019-01310-1

6. Jamond, Olivier, Lelong, Nicolas, Fourmont, Axel, Bluthé, Joffrey, Breuze, Matthieu, Bouda, Pascal, Brooking, Guillaume, Drui, Florence, Epalle, Alexandre, Fandeur, Olivier, Folzan, Gauthier, Helfer, Thomas, Kloss, Francis, Latu, Guillaume, Motte, Antoine, Nahed, Christopher, Picard, Alexis, Prat, Raphael, Ramière, Isabelle, Steins, Morgane and Prabel, Benoit. MANTA : Un code HPC généraliste pour la simulation de problèmes complexes en mécanique. In : *CSMA 2022 15ème colloque national en calcul des structures*. Giens, France, May 2022. Available from: https://hal.archives-ouvertes.fr/hal-03688160

6. Jamond, Olivier, Lelong, Nicolas, Fourmont, Axel, Bluthé, Joffrey, Breuze, Matthieu, Bouda, Pascal, Brooking, Guillaume, Drui, Florence, Epalle, Alexandre, Fandeur, Olivier, Folzan, Gauthier, Helfer, Thomas, Kloss, Francis, Latu, Guillaume, Motte, Antoine, Nahed, Christopher, Picard, Alexis, Prat, Raphael, Ramière, Isabelle, Steins, Morgane and Prabel, Benoit. MANTA : Un code HPC généraliste pour la simulation de problèmes complexes en mécanique. In : *CSMA 2022 15ème colloque national en calcul des structures*. Giens, France, May 2022. Available from: https://hal.archives-ouvertes.fr/hal-03688160

7. Stroustrup, B. Le langage C++, édition spéciale revue et corrigée. Paris : Pearson Education France, 2003.

8. Martin, DG. The elastic constants of polycrystalline UO2 and (U,Pu) mixed oxides: A review and recommendations. *High Temperatures. High Pressures*. 1989. Vol. 21, no. 1, p. 13–24.

11. Helfer, Thomas, Bejaoui, Syriac and Michel, Bruno. Licos, a fuel performance code for innovative fuel elements or experimental devices design. *Nuclear Engineering and Design*. 2015.
Under review

11. Helfer, Thomas, Bejaoui, Syriac and Michel, Bruno. Licos, a fuel performance code for innovative fuel elements or experimental devices design. *Nuclear Engineering and Design*. 2015.
Under review

11. Helfer, Thomas, Bejaoui, Syriac and Michel, Bruno. Licos, a fuel performance code for innovative fuel elements or experimental devices design. *Nuclear Engineering and Design*. 2015.
Under review

12. Fortin, André. Analyse numérique pour ingénieurs. [Montréal] : Presses internationales Polytechnique, 2001. ISBN 2553009364 9782553009365.

13. Baker, Jr and Just, Louis C. ANL-6548: Studies of metal-water reactions at high temperatures. Iii. Experimental and theoretical studies of the zirconium-water reaction. Argonne National Lab. (ANL), Argonne, IL (United States), 1962. Available from: https://www.osti.gov/biblio/4781681-studies-metal-water-reactions-high-temperatures-iii-experimental-theoretical-studies-zirconium-water-reaction

14. Cathcart, J. V., Pawel, R. E., McKee, R. A., Druschel, R. E., Yurek, G. J., Campbell, J. J. and Jury, S. H. Zirconium metal-water oxidation kinetics IV reaction rate studies. United States, 1977.
ORNL/NUREG–17INIS Reference Number: 9357740

15. Billone, M. C., Jankus, V. Z., Kramer, J. M. and Yang, C. I. Progress in modeling carbide and nitride fuel performance in advanced LMFBRs. United States : American Nuclear Society, 1977.
INIS Reference Number: 9379733

16. Koistinen, DP and Marburger, Re. A general equation prescribing extent of austenite - martensite transformation in pure fe-c alloys and plain carbon steels. *Acta Metallurgica*. 1959. Vol. 7, p. 59–60.

17. Massih, A. R. and Jernkvist, Lars O. Solid state phase transformation kinetics in zr-base alloys. *Scientific Reports*. December 2021. Vol. 11, no. 1. DOI 10.1038/s41598-021-86308-w. Available from: http://www.nature.com/articles/s41598-021-86308-w

18. Massih, A. R. Transformation kinetics of zirconium alloys under non-isothermal conditions. *Journal of Nuclear Materials*. 28 February 2009. Vol. 384, no. 3, p. 330–335. DOI 10.1016/j.jnucmat.2008.11.033. Available from: https://www.sciencedirect.com/science/article/pii/S0022311508007654

19. Chaboche, Jean-Louis, Lemaître, Jean, Benallal, Ahmed and Desmorat, Rodrigue. Mécanique des matériaux solides. Paris : Dunod, 2009. ISBN 9782100516230 210051623X.

20. Proix, Jean-Michel. R5.03.92 révision : 8597: Intégration des relations de comportement élasto-plastique de von mises. Référence du Code Aster. EDF-R&D/AMA, 2012. Available from: http://www.code-aster.org

21. Monerie, Yann and Gatt, Jean-Marie. Overall viscoplastic behavior of non-irradiated porous nuclear ceramics. *Mechanics of Materials*. July 2006. Vol. 38, no. 7, p. 608–619. DOI 10.1016/j.mechmat.2005.11.004. Available from: http://www.sciencedirect.com/science/article/pii/S0167663605001882

22. Salvo, Maxime, Sercombe, Jérôme, Ménard, Jean-Claude, Julien, Jérôme, Helfer, Thomas and Désoyer, Thierry. Experimental characterization and modelling of UO2 behavior at high temperatures and high strain rates. *Journal of Nuclear Materials*. January 2015. Vol. 456, p. 54–67. DOI 10.1016/j.jnucmat.2014.09.024. Available from: http://www.sciencedirect.com/science/article/pii/S002231151400614X

23. Salvo, Maxime, Sercombe, Jérôme, Helfer, Thomas, Sornay, Philippe and Désoyer, Thierry. Experimental characterization and modeling of UO2 grain boundary cracking at high temperatures and high strain rates. *Journal of Nuclear Materials*. May 2015. Vol. 460, p. 184–199. DOI 10.1016/j.jnucmat.2015.02.018. Available from: http://www.sciencedirect.com/science/article/pii/S0022311515001130

24. Hosford, W. F. A generalized isotropic yield criterion. *Journal of Applied Mechanics*. 1972. Vol. 39, no. 2, p. 607–609.

25. Cazacu, Oana and Barlat, Frédéric. A criterion for description of anisotropy and yield differential effects in pressure-insensitive metals. *International Journal of Plasticity*. 1 November 2004. Vol. 20, no. 11, p. 2027–2045. DOI 10.1016/j.ijplas.2003.11.021. Available from: http://www.sciencedirect.com/science/article/pii/S0749641904000348

26. Cazacu, Oana and Barlat, Frédéric. Generalization of drucker's yield criterion to orthotropy. *Mathematics and Mechanics of Solids*. 1 December 2001. Vol. 6, no. 6, p. 613–630. DOI 10.1177/108128650100600603. Available from: https://doi.org/10.1177/108128650100600603

27. Barlat, F., Aretz, H., Yoon, J. W., Karabin, M. E., Brem, J. C. and Dick, R. E. Linear transfomation-based anisotropic yield functions. *International Journal of Plasticity*. 1 May 2005. Vol. 21, no. 5, p. 1009–1039. DOI 10.1016/j.ijplas.2004.06.004. Available from: http://www.sciencedirect.com/science/article/pii/S0749641904001160

28. Armstrong, P. J. and Frederick, C. O. RD/BfN 731: A mathematical representation of the multiaxial bauschinger effect. Central Electricity Generating Board, 1966.

29. Burlet, Hélène and Cailletaud, Georges. Modelling of cyclic plasticity in finite element codes. In : *Proceedings of 2nd international conference on constitutive laws for engineering materials; theory and application*. Tucson, AZ, 1987.

30. Chaboche, J. -L., Kanouté, P. and Azzouz, F. Cyclic inelastic constitutive equations and their impact on the fatigue life predictions. *International Journal of Plasticity*. 1 August 2012. Vol. 35, p. 44–66. DOI 10.1016/j.ijplas.2012.01.010. Available from: http://www.sciencedirect.com/science/article/pii/S0749641912000113

31. Gurson, A. L. Continuum theory of ductile rupture by void nucleation and growth: Part I - Yield criteria and flow rules for porous ductile media. *J. Eng. Mat. and Tech.* 1977. Vol. 99, p. 2–15.

32. Tvergaard, V. and Needleman, A. Analysis of the cup-cone fracture in a round tensile bar. *Acta. Metall.* 1984. Vol. 32, p. 157–169.

33. Castañeda, P. Ponte. The effective mechanical properties of nonlinear isotropic composites. *Journal of the Mechanics and Physics of Solids*. 1 January 1991. Vol. 39, no. 1, p. 45–71. DOI 10.1016/0022-5096(91)90030-R. Available from: http://www.sciencedirect.com/science/article/pii/002250969190030R

34. Rousselier, G. Finite deformation constitutive relations including ductile fracture damage. In : *Nemat-Nasser (ed.) Three Dimensional Constitutive Relations and Ductile Fracture*. North Holland, 1981. p. 331–355.

35. Berdin, C., Besson, J., Bugat, S., Desmorat, R., Feyel, F., Forest, E., S. Lorentz, Maire, E., Pardoen, T., Pineau, A. and Tanguy, B. Local approach to fracture. Ecole des Mines de Paris, 2004.

36. Tanguy, B. and Besson, J. An extension of the Rousselier model to viscoplastic temperature dependent materials. *Int. J. Fracture*. 2002. Vol. 116, p. 81–101.

37. Chu, C. C. and Needleman, A. Void nucleation effects in biaxially stretched sheets. *Journal of Engineering Materials and Technology*. 1980. Vol. 102, no. 3, p. 249–256.

38. Helfer, Thomas. Assisted computation of the consistent tangent operator of behaviours integrated using an implicit scheme. Theory and implementation in MFront. Documentation of mfront. CEA, 2020. Available from: https://www.researchgate.net/publication/342721072_Assisted_computation_of_the_consistent_tangent_operator_of_behaviours_integrated_using_an_implicit_scheme_Theory_and_implementation_in_MFront

39. Zhang, Y. Modélisation et simulation robuste de l'endommagement ductile. PhD thesis. Paris, France : Université de recherche Paris Sciences et Lettres, 2016.

40. Helfer, T. Implementation of a multi-surface, compressible and perfect plastic behaviour using the drucker-prager yield criterion and a cap. TFEL/MFront. 21 November 2017. Available from: https://thelfer.github.io/tfel/web//drucker-prager-cap.html

41. Petit, T., Besson, J., Ritter, C., Colas, K., Helfen, L. and Morgeneyer, T. F. Effect of hardening on toughness captured by stress-based damage nucleation in 6061 aluminum alloy. *Acta Metall.* 2019. Vol. 180, p. 349–365. DOI https://doi.org/10.1016/j.actamat.2019.08.055.

42. Tanguy, B. Modélisation de l'essai charpy par l'approche locale de la rupture: Application au cas de l'acier 16MND5 dans le domaine de la transition. PhD thesis. Ecole Nationale Supérieure des Mines de Paris, 2001.

43. Invariants of tensors. *Wikipedia*. 2017. Available from: https://en.wikipedia.org/w/index.php?title=Invariants_of_tensors&oldid=813063385
Page Version ID: 813063385

44. Kopp, Joachim. Efficient numerical diagonalization of hermitian 3x3 matrices. *International Journal of Modern Physics C*. March 2008. Vol. 19, no. 3, p. 523–548. DOI 10.1142/S0129183108012303. Available from: http://arxiv.org/abs/physics/0610206

45. Kopp, Joachim. Numerical diagonalization of 3x3 matrices. 2017. Available from: https://www.mpi-hd.mpg.de/personalhomes/globes/3x3/

46. Eberly, David. A robust eigensolver for 3×3 symmetric matrices. September 2016. Available from: https://www.geometrictools.com/Documentation/RobustEigenSymmetric3x3.pdf

47. Eberly, David. Geometric tools. 2017. Available from: http://www.geometrictools.com/

48. Von mises yield criterion. *Wikipedia*. 2017. Available from: https://en.wikipedia.org/w/index.php?title=Von_Mises_yield_criterion&oldid=812733048
Page Version ID: 812733048

49. Scherzinger, W. M. A return mapping algorithm for isotropic and anisotropic plasticity models using a line search method. *Computer Methods in Applied Mechanics and Engineering*. 15 April 2017. Vol. 317, p. 526–553. DOI 10.1016/j.cma.2016.11.026. Available from: http://www.sciencedirect.com/science/article/pii/S004578251630370X