

Introducing the open-source `mfront` code generator: Application to mechanical behaviours and material knowledge management within the PLEIADES fuel element modelling platform

Thomas Helfer^{a,*}, Bruno Michel^a, Jean-Michel Proix^{b,*}, Maxime Salvo^a,
Jérôme Sercombe^a, Michel Casella^a

^a CEA, DEN/DEC, Département d'Études des Combustibles, Saint Paul lez Durance F-13108 Cedex, France

^b EDF R&D, Département Analyses Mécaniques et Acoustique, 1 Avenue du Général de Gaulle, F-92141 CLAMART, France

ARTICLE INFO

Article history:

Received 2 September 2014

Received in revised form 1 March 2015

Accepted 24 June 2015

Available online 22 July 2015

Keywords:

Material knowledge management

Mechanical behaviour integration

Implicit integration schemes

Single crystal plasticity

Domain specific languages

ABSTRACT

The PLEIADES software environment is devoted to the thermomechanical simulation of nuclear fuel elements behaviour under irradiation. This platform is co-developed in the framework of a research cooperative program between Électricité de France (EDF), AREVA and the French Atomic Energy Commission (CEA). As many thermomechanical solvers are used within the platform, one of the PLEIADES's main challenge is to propose a unified software environment for capitalisation of material knowledge coming from research and development programs on various nuclear systems.

This paper introduces a tool called `mfront` which is basically a code generator based on C++ (Stroustrup and Eberhardt, 2004). Domain specific languages are provided which were designed to simplify the implementations of new material properties, mechanical behaviours and simple material models. `mfront` was recently released under the GPL open-source licence and is available on its web site: <http://tfel.sourceforge.net/>.

The authors hope that it will prove useful for researchers and engineers, in particular in the field of solid mechanics. `mfront` interfaces generate code specific to each solver and language considered.

In this paper, after a general overview of `mfront` functionalities, a particular focus is made on mechanical behaviours which are by essence more complex and may have significant impact on the numerical performances of mechanical simulations. `mfront` users can describe all kinds of mechanical phenomena, such as viscoplasticity, plasticity and damage, for various types of mechanical behaviour (small strain or finite strain behaviour, cohesive zone models). Performance benchmarks, performed using the `Code_Aster` finite element solver, show that the code generated using `mfront` is in most cases on par or better than the behaviour implementations written in `fortran` natively available in this solver. The material knowledge management strategy that was set up within the PLEIADES platform is briefly discussed. A material database named `sirius` proposes a rigorous material verification workflow.

* Corresponding authors.

E-mail addresses: thomas.helfer@cea.fr (T. Helfer), bruno.michel@cea.fr (B. Michel), jean-michel.proix@edf.fr (J.-M. Proix), maxime.salvo@cea.fr (M. Salvo), jerome.sercombe@cea.fr (J. Sercombe), michel.casella@cea.fr (M. Casella).

<http://dx.doi.org/10.1016/j.camwa.2015.06.027>

0898-1221/© 2015 Elsevier Ltd. All rights reserved.

We illustrate the use of `mfront` through two case of studies: a simple FCC single crystal viscoplastic behaviour and the implementation of a recent behaviour for the fuel material which describes various phenomena: fuel cracking, plasticity and viscoplasticity.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

The PLEIADES software environment, co-developed in the framework of a research cooperative program between Électricité de France (EDF), AREVA and the French Atomic Energy Commission (CEA), is devoted to the simulation of nuclear fuel elements behaviour under irradiation. The PLEIADES platform is the basis of several fuel performance codes [1–5]. Capitalisation of the material knowledge coming from research and development programs on various nuclear systems (Pressurised Water Reactor, Sodium Fast Reactor, Gas Fast Reactor, Material Testing Reactor, etc.) is a key part of the work made within the PLEIADES platform.

Thus, a rigorous material knowledge management strategy was put in place, that partly relies on a code generator called `mfront`. `mfront` provides a set of domain specific languages handling material properties, mechanical behaviours and simple material models. Those languages are meant to be easy to use and learn by researchers and engineers. Authors of `mfront` paid particular attention to the robustness, reliability and numerical efficiency of the generated code. `mfront` interface allows to generate code to be plugged in various software environment (languages or solver). Two general-purpose finite element solvers, `Code_Aster` and `Cast3M`, which are developed respectively by EDF [6] and CEA [7], will be used in this paper to illustrate the use of `mfront`.

Section 2 is the core of this paper. It gives an overview of the `mfront` code generator, which is based on the C++ language [8], and shows how various material properties and mechanical behaviours can be implemented. Mechanical behaviours which are by essence more complex and may have significant impact on the numerical efficiency of mechanical solvers are discussed in depth. Performance assessments are then presented. We conclude this section by introducing the material knowledge management strategy of the PLEIADES platform. Section 2.7 introduces the `sirius` database. We focus on the capitalisation and verification processes that were put in place within the PLEIADES platform.

Section 3 describes how a behaviour describing FCC single crystal plasticity can be implemented with `mfront` with an implicit scheme. This section compares various algorithms in terms of computational performances and robustness.

Section 4 introduces a specific mechanical behaviour used to describe the fuel material in PWR fuel elements within the PLEIADES platform. This section provides a detailed description of the implementation of a complex mechanical behaviour using an implicit scheme and may only interest potential end users. Mechanical simulations are handled through the `Cast3M` finite element solver.

2. Overview of the `mfront` code generator

`mfront` is a code generator which translates a set of closely related domain specific languages into plain C++ sources. Those languages cover three kinds of material knowledge: material properties (for instance the YOUNG modulus, the thermal conductivity, etc.), mechanical behaviours which are discussed in depth in this paper and simple point-wise models, such as material swelling used in fuel performance codes. Models will not be considered in this paper.

2.1. A first example: uranium dioxide YOUNG modulus

Material properties generally depend on some state variables which describe the current thermodynamical state of the material. The implementation of the YOUNG modulus of uranium dioxide is given in Fig. 1. The code computing the YOUNG modulus, introduced by the `@Function` keyword on line 8, is reasonably close to the mathematical expression of the material property [9]:

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

where T and f are respectively the temperature and the porosity.

This file can be processed from the command line:

```
mfront --obuild --interface=python UO2_YoungModulus_Martin1989.tex
```

which first creates a set of C++ sources and then compiles them into a module for the Python language [10]. Interfaces, a key feature of `mfront`, enable the reusability of this material property in a wide variety of software contexts (see examples provided in Section 2.2).

```

1  @Parser MaterialLaw;           // treating a material property
2  @Material UO2;                 // material name
3  @Law YoungModulus_Martin1989; // name of the material property
4  @Output E;                     // output of the material property
5  @Input T, f;                   // inputs of the material property
6  @Function                      // implementation body
7  {
8      E = 2.2693e11 * (1. - 2.5 * f) * (1 - 6.786e-05 * T - 4.23e-08 * T * T);
9  }

```

Fig. 1. Implementation of the uranium dioxide YOUNG modulus using the correlation recommended by MARTIN [9].

Per se, this implementation, although fully functional misses some important features:

1. assurance quality requires some additional meta-data:

```

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

```

2. the validity bounds of the material property were not set:

```

@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

```

If one of the inputs, whether the temperature or the porosity, is out of the physical bounds, computations are stopped. What happens if an input is out of the (standard) bounds of the material property, depends on which interface is used. Most interfaces allow the user to choose one of the following policy: do nothing, print a warning, stop the computations;

3. the inputs and the output of this material property are not clearly specified. The PLEIADES platform introduced normalised identifiers called glossary names:

```

E.setGlossaryName("YoungModulus");
T.setGlossaryName("Temperature");
f.setGlossaryName("Porosity");

```

Once glossary names are set, it is clear that this correlation depends on the temperature (variable T) and the porosity (variable f).

Since the material libraries are loaded dynamically (at runtime-time) by the PLEIADES fuel performance codes and the `mtest` tool presented in Section 2.5, glossary names are mandatory to be able to evaluate a material property appropriately.

2.2. Main objectives of the *mfront* code generator

mfront is intended for researchers and engineers that have to introduce new material knowledge in their simulations and thus addresses the following goals:

- Ease of use. The library should provide a lower entry barrier for new users by providing, for each type of material knowledge, a domain specific language which is as close as possible to its natural mathematical formulation: it should try to minimise the up-front investment in time to just learn a few basic rules and guidelines. In particular, a set of tensorial objects are provided for expressing the constitutive equations of the mechanical behaviour. Those objects are part of a library called TFEL described in [Appendix A](#).
- Interoperability. Material knowledge written using *mfront* shall be useable in a wide variety of contexts (language or solver). For example, material properties written in *mfront* can be used in `fortran`, `C`, `C++`, `python`, `GNU Octave`, `Microsoft Excel`, etc.
- Efficiency. *mfront* generated code uses the advanced and optimised techniques provided by the TFEL library, described in [Appendix A](#), for most mathematical operations.
- Portability. This point is discussed in [Appendix A.6](#).

The most valuable feature of *mfront* is however that it can be the basis of a rigorous material knowledge management. The point will be discussed in Section 2.7.

2.3. Design choices

Stringent design choices had to be made during the development of *mfront* are now discussed to match the objectives listed in the previous paragraph. The most important ones are now discussed.

Relying on C++. We first decided not to create a new language and to rely on C++ for various reasons:

- we felt that creating our own languages would cause maintainability and portability issues.
- we felt that creating our own languages could harm performances. C++ has been designed to be as efficient as possible and compilers are now very mature. By relying on C++, we also could use the advanced techniques described in [Appendix A](#).
- interoperability is one of main goals of *mfront*. The most interoperable language today is C: a C function can be called from virtually all the languages or software of interest for numerical simulations, including *fortran* (77 or later), C++, *java* (through the JNI), *Python*, GNU Octave, *Scilab*, *MATLAB*, etc. *fortran* support is of primary importance as most general purpose finite element solvers (*Abaqus*, *Ansys*, *Cast3M*, *Code_Aster*, *Europlexus*, etc.) are written in that language. C++ inherits from this interoperability advantage.
- C++ allows operator overloading. We used this feature to build the tensorial library described in [Appendix A](#) to make the code as natural as possible: tensorial operations implementations are closed to their mathematical expressions.
- we felt that one shall let the user be able to use the numerous scientific libraries available, if necessary. Most of them can be used directly from C++.
- although their use shall be reduced as much as possible, one shall have access to standard flow control statements: conditionals, loops, and so on. Those statements are part of C++.

Code generation. As acknowledged by its creator, C++-98 has become an expert-friendly language and is difficult to learn and use appropriately. The introduction of a code generator allowed us:

- to alleviate the intrinsic difficulty of C++. Most *mfront* implementations use a very limited subset of C++.
- let the user focus on the physics.
- generate specific implementations of standard numerical algorithms.
- hide most of the boiler-plate code.

Interfaces. As described earlier, *mfront* implementations are meant to be used in a wide variety of languages or solvers. All of them have their own particularities that have to be taken into account.

We introduced the concept of *interfaces*: an interface creates appropriate and optimised wrapper code to a specific language or solver.

Support for a new solver can be added by creating a new interface. The required work greatly depends on the considered solver.

2.4. Mechanical behaviours

Versatility of general purpose mechanical solver mainly relies on its ability to let the user define the material behaviour. *mfront* provides a high level language to write mechanical behaviours and can be compared to the *ZebFront* tool developed by the Centre des Matériaux de Mines ParisTech as part of the *Zset* software [11–13]. One major difference between *ZebFront* and *mfront* is the programming techniques used: *ZebFront* mostly relies on object oriented techniques where *mfront* relies on generic programming leading to almost orthogonal design choices (see [Appendix A](#) for details).

Three kinds of mechanical behaviour are currently considered with *mfront*: small and finite strain behaviours and cohesive zone models.

Mechanical behaviour role. We now precise the role of the mechanical behaviour in standard displacement-based finite element solvers [14–16]. For the sake of simplicity, we only treat the case of the small strain behaviours for the rest of this paragraph.

At each time step, the following resolution procedure is used:

- (1) a prediction of the displacement is made. Such a prediction may use the derivative of the stress tensor with respect of the strain tensor $\frac{\partial \sigma}{\partial \varepsilon^{t_0}}$ or an approximation of it. This prediction step will not be discussed in this article but can also be handled by behaviour implementations made with *mfront*;
- (2) an iterative process similar to the NEWTON–RAPHSON algorithm used to find a displacement that will satisfy the mechanical equilibrium. Given an estimation of the displacement at the end of the time step, one computes at each integration point an estimation of the increment of the deformation tensor $\Delta \varepsilon^{t_0}$. The mechanical behaviour is then called and provides an associated estimation of the stress tensor $\underline{\sigma}$ and the values of some internal state variables Y_i at the end of the time step. If the solver requires it, the mechanical behaviour may also provide an estimation of the tangent consistent operator $\frac{\partial \Delta \underline{\sigma}}{\partial \Delta \varepsilon^{t_0}}$ [17] which is used to estimate a more accurate displacement field.

```

1  @Parser IsotropicPlasticMisesFlow; //< domain specific language
2  @Behaviour Plasticity; //< name of the behaviour
3  @Parameter H = 22e9; //< hardening slope
4  @Parameter s0 = 200e6; //< elasticity limit
5  @FlowRule{ //< flow rule
6      f = seq-H*p-s0;
7      df_dseq = 1;
8      df_dp = -H;
9  }

```

Fig. 2. Implementation of a simple J_2 plastic behaviour with isotropic hardening. By default, elastic material properties (YOUNG modulus, Poisson ratio) must be given by the calling program.

A mechanical behaviour can thus be viewed as a functional:

$$\left(\underline{\sigma}|_{t+\Delta t}, Y_i|_{t+\Delta t}, \frac{\partial \Delta \underline{\sigma}}{\partial \underline{\varepsilon}^{to}} \right) = \mathcal{F} \left(\underline{\sigma}|_t, Y_i|_t, \Delta \underline{\varepsilon}^{to}, \Delta t, \dots \right).$$

The dots ... mean that the behaviour may also depend on external state variables evolutions with time, such as the temperature, the irradiation damage, and so on.

2.4.1. Isotropic J_2 plastic/viscoplastic behaviours. Example of finite strain pre- and post-processing

Four domain specific languages address the case of small strain isotropic J_2 plastic and/or viscoplastic behaviours which are of common use and for which efficient implicit scalar radial return mapping algorithms exist [18]. Fig. 2 shows how a simple plastic behaviour can be implemented. The plastic behaviour is governed by the following yield surface [15,19]:

$$f(\sigma_{eq}, p) = \sigma_{eq} - H p - \sigma_0 \leq 0$$

where σ_{eq} is the VON MISES stress (see in Appendix C for details), p the equivalent plastic strain, H the isotropic hardening slope and σ_0 the initial elasticity limit.

The generated code represents a total amount of 1512 lines of C++ code and provides:

- optimised implementations of the behaviour for various modelling hypotheses (axisymmetrical generalised plane strain, plane strain, plane stress, generalised plane strain, axisymmetry, tridimensional) thanks to template metaprogramming and template specialisations. A small overview of the programming techniques used can be found in Appendix A.
- the computation of the prediction operator;
- the computation of a tangent matrix operator (various choices are possible: elastic, secant, consistent);
- meta-data about the required material properties, the number of states variables, etc. that can be retrieved dynamically. (For the sake of simplicity, no glossary name was specified in this example.) This mechanism is used by PLEIADES applications to appropriately call the behaviour;
- dynamically loadable functions allowing the user to change various parameters of the behaviour (the convergence criterion of the implicit algorithm, the θ parameter of the implicit algorithm which will be defined later by Eq. (2), the hardening slope H and the initial elasticity limit σ_0 , etc.). Those functions by-pass the standard behaviour call and are an extremely light-weight manner to dynamically modify a behaviour (almost no runtime cost).

Local divergence. Local divergence of the implicit algorithm can be handled through an appropriate substepping procedure. This feature is not enabled by default, but appropriate keywords give the end user explicit control on this procedure.

Finite strain strategies. If not handled directly by the calling code, appropriate pre- and post-processing can be generated, allowing the use of behaviours written in the framework of the infinitesimal strain theory in finite strain computations. Two lagrangian finite strain strategies are currently available:

- finite rotations, small strains [21]. This method allows the re-use of a behaviour whose material parameters H and σ_0 were identified through small strain computations in the context of finite rotations without any re-identification. The physical meaning of the pre- and post-processing stages are discussed by Doghri [22];
- lagrangian logarithmic strains as proposed by MIEHE, APEL and LAMBRECHT [20,23]. Fig. 3 shows how our example can be used to model a notched specimen under a tensile test. In this case, the material parameters H and σ_0 of the behaviour must be identified by tests implying finite strains. As an additional remark, the results found using logarithmic strains were remarkably close to those obtained by the classical formulation based on an $F_e F_p$ decomposition proposed by SIMO and MIEHE [24] (that was also implemented using mfront). While numerically less efficient, the use of the logarithmic strains has several advantages:
 - (a) it preserves the small-strain classical meaning of the variables, which is truly appreciated by engineers;
 - (b) it may can also be used for arbitrary complex models (kinematic hardening, initial or induced anisotropy, etc.).

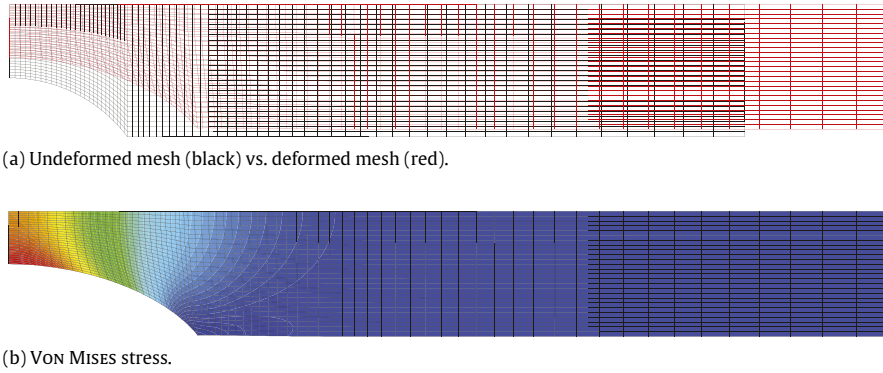


Fig. 3. Axisymmetric modelling of a notched specimen at finite strain using a simple J_2 behaviour in the logarithmic strain space [20]. Computations were performed using **Cast3M**, a finite element solver developed by CEA [7]. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

```

1  @Parser DefaultCZMPParser;      // domain specific language
2  @Behaviour Tvergaard;           // name of the behaviour
3  @MaterialProperty stress kn;    // normal stiffness
4  @MaterialProperty stress ks;    // tangential elastic stiffness
5  @MaterialProperty stress smax; // maximal stress
6  @MaterialProperty real delta;  // maximal normal opening displacement
7  @StateVariable real d;         // damage variable
8  @Integrator{
9      const real C = real(27)/real(4);
10     t_t = ks*(u_t+du_t);        // tangential behaviour
11     if (u_n+du_n<0){           // normal behaviour in compression
12         t_n = kn*(u_n+du_n);
13     } else {                   // normal behaviour in traction
14         const real rod = (u_n+du_n)/delta; // reduced opening displacement
15         const real d_l = d;      // previous damage
16         d = min(max(d,rod),0.99); // damage indicator
17         const real K1 = C*smax/delta; // initial stiffness
18         const real K = K1*(1-d)*(1-d); // secant stiffness
19         t_n = K*(u_n+du_n);
20     }
21 } // end of @Integrator

```

Fig. 4. Implementation of the TVERGAARD cohesive zone model using the Default domain specific language. Details about the computation of the consistent tangent operator were eluded. The opening displacement \bar{u} is automatically decomposed into the normal opening displacement u_n and its tangential opening displacement \bar{u}_t .

2.4.2. Generic domain specific languages

Apart from the domain specific languages dealing with isotropic J_2 plastic and/or viscoplastic behaviours presented in the previous paragraph, **mf front** also provides several general-purpose domain specific languages:

1. the **Default** domain specific language allows the user to write its own integration algorithm. This is very useful for explicit behaviours such as the classical TVERGAARD cohesive zone model [25], the implementation of which is given in Fig. 4.
2. the **Runge-Kutta** domain specific language allows the user to write the constitutive equations given as a system of ordinary time differential equations. Using those algorithms is generally less efficient than using the implicit integration described in the next section. Various algorithms are available:
 - (a) explicit EULER algorithm, second and fourth order RUNGE-KUTTA algorithms. As those algorithms do not provide any error control mechanism, they are only used for demonstration purpose.
 - (b) adaptive $4/2$ and $5/4$ RUNGE-KUTTA algorithms. The latter, also known as the FEHLBERG algorithm [26], is generally the most efficient algorithm and is used by default. Fig. 5 shows how a simple generalisation of the NORTON behaviour for an orthotropic material can be implemented.

```

1  @Parser      RungeKutta;           // domain specific language
2  @Behaviour   OrthotropicCreep;     // name of the behaviour
3  @OrthotropicBehaviour;             // treating an orthotropic behaviour
4  @RequireStiffnessTensor;          // requires the stiffness tensor to be computed
5  @StateVariable Stensor evp;        // viscoplastic strain
6  @StateVariable strain p;           // Equivalent viscoplastic strain
7  @Includes{                          // header files
8  #include<TFEL/ Material/ Hill.hxx>
9  }
10 @ComputeStress{                    /* stress computation */
11     sig = D*eel;
12 }
13 @Derivative{                        /* constitutive equations */
14     st2tost2<N,real> H;             // Hill Tensor
15     H = hillTensor<N,real>(0.371,0.629,4.052,1.5,1.5,1.5);
16     stress sigeq = sqrt(sig|H*sig); // equivalent Hill stress
17     if(sigeq>1e9){                  // automatic sub-stepping
18         return false;
19     }
20     Stensor n(0.);                  // flow direction
21     if(sigeq > 10.e-7){
22         n = H*sig/sigeq;
23     }
24     dp = 8.e-67*pow(sigeq,8.2);    // evolution of p
25     devp = dp*n;                   // evolution of the viscoplastic strains
26     deel = deto - devp;             // evolution of the elastic strains
27 }

```

Fig. 5. Implementation of a generalisation of the NORRON creep law for anisotropic materials. Integration is performed in the material reference system. The elastic strain state variable $\underline{\varepsilon}^{\text{el}}$ is automatically declared. For each state variable Y , its time derivative dY is automatically declared.

- the `Implicit` domain specific language allows the user to perform the local integration using an implicit algorithm. An introduction to those algorithms is given in the next paragraph.

2.4.3. Implicit integration

The evolution of the state variables, grouped into a single vector Y whose components Y_i may be scalars or symmetric tensors, is given by the following system of differential equations:

$$\dot{Y} = G(Y, t) \quad \Leftrightarrow \quad \begin{cases} \dot{Y}_0 = g_{Y_0}(Y, t) \\ \dots \\ \dot{Y}_i = g_{Y_i}(Y, t) \\ \dots \\ \dot{Y}_N = g_{Y_N}(Y, t) \end{cases}$$

where the dependency with respect to time stands for the evolution of some external state variables and the evolution of strains (for small strain behaviours) which are supposed to evolve *linearly* during the time step.

The integration of this ordinary differential equations over a time step Δt using an implicit algorithm leads to the (generally non-linear) system of equations [27]:

$$F(\Delta Y) = 0 \quad \Leftrightarrow \quad \begin{cases} f_{Y_0} = \Delta Y_0 - \Delta t g_{Y_0}(Y|_{t+\theta \Delta t}, t) = 0 \\ \dots \\ f_{Y_i} = \Delta Y_i - \Delta t g_{Y_i}(Y|_{t+\theta \Delta t}, t) = 0 \\ \dots \\ f_{Y_N} = \Delta Y_N - \Delta t g_{Y_N}(Y|_{t+\theta \Delta t}, t) = 0 \end{cases} \quad (1)$$

where the unknowns are the state variables increments ΔY_i , and we introduced the following notation:

$$Y|_{t+\theta \Delta t} = Y + \theta \Delta Y. \quad (2)$$

Algorithms used to solve this system of equations may require the jacobian matrix J of F which can be computed by blocks [15]:

$$J = \frac{\partial F}{\partial \Delta Y} = \begin{pmatrix} \frac{\partial f_{y_1}}{\partial \Delta y_1} & \dots & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \frac{\partial f_{y_i}}{\partial \Delta y_j} & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & \dots & \dots & \dots & \frac{\partial f_{y_N}}{\partial \Delta y_N} \end{pmatrix}. \quad (3)$$

Time independent mechanisms. For state variables associated with time-independent mechanisms, Eq. (1) is replaced by imposing that the system lies on the yield surface when plastic loading occurs.

Available algorithms. Several algorithms are available to solve System (1):

- **NewtonRaphson** is the standard NEWTON–RAPHSON algorithm:

$$\Delta Y^{(n+1)} = \Delta Y^{(n)} - J^{-1} \cdot F(\Delta Y^{(n)}).$$

The user must explicitly compute the jacobian matrix, which constitutes the main difficulty of this method. For debugging purposes, **mfront** may generate the comparison of each block of the jacobian matrix with a numerical approximation.

- **NewtonRaphson_NumericalJacobian** is a variation of the standard NEWTON–RAPHSON algorithm using a jacobian matrix computed by a second order finite difference. Writing behaviour implementations using this algorithm is as easy as using the RUNGE–KUTTA domain specific languages. It can be considered as a first step toward an implicit implementation with an analytical jacobian matrix.
- **Broyden** algorithms which do not require the computation of the jacobian matrix: those algorithms update an approximation of the jacobian matrix (first Broyden algorithm) or its inverse (second Broyden algorithm) at each iteration. The first Broyden algorithm can sometimes be interesting as one may compute analytically some part of the jacobian matrix and let the algorithm compute the other parts. If the computation of those other parts takes a significant amount of CPU time, this algorithm can in some cases outperform the **NewtonRaphson** algorithm.
- **PowellDogLeg_XX** algorithm, where X is one of the previous algorithm. Those trust-region algorithms implement the classical **Powell dogleg** method [28] to improve the robustness of the resolution.

The user has several ways to modify the previous algorithms. For example, if the user has given physical bounds for a state variable, the increment of this state variable is automatically bound to satisfy those physical bounds at the end of the time step. Some other ways are discussed in Section 3.4.

Consistent tangent operator. For most small strain behaviours, algorithms providing the jacobian matrix J of the implicit System (1) have a significant advantage: the consistent tangent operator $\frac{\partial \Delta \sigma}{\partial \Delta \varepsilon^{\text{to}}}$ can be computed almost automatically with only a small numerical cost (see Appendix B.1). Examples of behaviour implementations using the **Implicit** domain specific language are considered in Sections 3 and 4.

Plane stress modelling hypothesis. Appendix B.2 shows how the implementation of a behaviour valid for the generalised plane strain modelling hypothesis can be reused to treat the plane stress modelling hypothesis with small modifications to the implicit System (1).

Conventions. Following Eq. (1), the f_{y_j} terms of the implicit system are initialised by the increment ΔY_j and the jacobian matrix is initialised to the identity matrix before each iteration.

2.5. Unit mechanical behaviour testing: introducing the *mtest* tool

mfront comes with a handy easy-to-use tool called **mtest** that can test the local behaviour of a material, by imposing independent constraints on each component of the strain or the stress. This tool has been much faster (from ten to several hundred times depending on the test case) than using a full-fledged finite element solver. It is equivalent to the **SIMU_POINT_MAT** operator available within the **Code_Aster** finite element solver [29] or to the **SiDoLo** software [30].

mtest can be used to model various experiments, as far as a stage implying strain localisation is not reached: tensile, compressive or shear tests driven by stresses or deformations (see example in Fig. 6), pipe loaded by internal or external pressure, **SATOH** test, etc.

mtest generates a text file containing the evolution of the strains (for small strains behaviours), the stresses and the state variables during the loading history. Other **mtest** functionalities include:

- the ability to test all the behaviours generated by **mfront** (small strain and finite strain behaviours, cohesive zone models);


```

1  @Behaviour<aster> 'src/libAsterBehaviour.so' 'asterplasticity';
2  @MaterialProperty<constant> 'YoungModulus' 150.e9;
3  @MaterialProperty<constant> 'PoissonRatio' 0.3;
4  @MaterialProperty<constant> 'H' 100.e9;
5  @MaterialProperty<constant> 's0' 100.e6;
6  @ExternalStateVariable 'Temperature' {0:293.15,3600.:800};
7  @ImposedStrain<function> 'EXX' '1.e-3*t';
8  @Times {0.,1 in 20};

```

Fig. 6. Modelling a tensile test along the x-axis using the *mtest* tool. Temperature definition is mandatory.

- the ability to test isotropic and orthotropic behaviours;
- the support of various modelling hypotheses, notably the plane stress and axisymmetric generalised plane stress hypotheses;
- many features to evaluate the numerical performances of mechanical behaviours. For example, user can compare the computed tangent consistent operator to a numerical approximation;
- a **C++** library and a **Python** interface [10]. *mtest* can be embedded in general purpose scientific environment to fit behaviour parameters against experimental data. In particular, *mtest* can be used in ADAO [31], a module for Data Assimilation and Optimisation of the *Salomé* platform [32];
- comparison of the results (strains, stresses, internal state variables) to analytical results or references results stored in an external file. *mtest* automatically generates XML file conforming to the JUnit format.¹ Those files can be used for reporting using the Jenkins continuous integration application.² This functionality is central in the assurance quality procedure of *mfront*.

Through an appropriate option, a behaviour implementation generated through *mfront* may create a *mtest* file in case of integration failure: this *mtest* file only describes the failed time step with the appropriate initial conditions. This feature is particularly useful to analyse the failure of large simulations which may happen after several hours of computations.

2.6. Performance assessments

Numerous performances assessments were made within the PLEIADES platform: replacing *fortran* implementations by their *mfront* counterparts led to significant improvements, from 30% to 50% of the total computational time of some fuel performance codes developed in the platform. This improvement is mainly due to the fact that the behaviour integration schemes changed from explicit RUNGE–KUTTA schemes to implicit ones. The main benefit of *mfront* was to grant users an easier access to the implicit schemes.

Developers of the *Code_Aster* general purpose finite element solver made independent extensive tests, comparing their own native implementations to the ones generated with *mfront*, generally using an implicit scheme in both cases. Without discussing the very details of each test performed (some of them were reported in Table 1), several general conclusions can be drawn:

- native implementations offer superior performances in the case of simple explicit behaviours (MAZARS damaging behaviour [41]) or in the case of isotropic behaviours that can be reduced to one scalar equation [42]. For explicit behaviours, the difference will be reduced by the development of an optimised treatment of *mfront* behaviours. In the second case, the difference can be explained by the fact that the *Code_Aster* implementations uses the BRENT algorithm [43] which clearly outperforms the standard NEWTON method. The availability of this algorithm in *mfront* is planned.
- for more complex behaviours, *mfront* implementations are on par or outperform the native implementations.

For a given behaviour, the development time was found significantly lower with *mfront*.

2.7. Material knowledge management within the PLEIADES platform: the *sirius* database

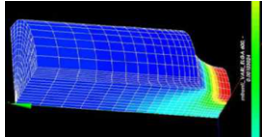
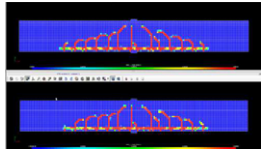
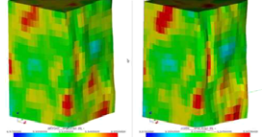
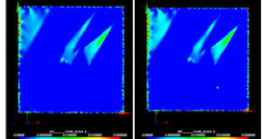
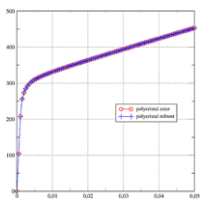
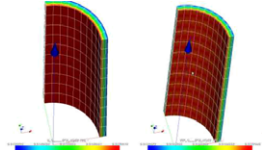
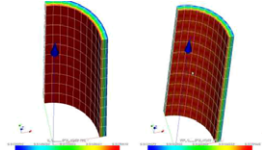
Material knowledge management is a central part of the development of the PLEIADES platform to guarantee that the simulations made meet a high level of quality requirements. The strategy built up is based on a database called *sirius* which stores material properties and behaviour laws involved in the French nuclear fuels R&D program. Currently, the database encompasses 105 distinct materials. 853 entries related to material properties are registered.

¹ <http://junit.org>.

² <http://jenkins-ci.org/>.

Table 1

Some benchmarks comparing the implementation generated through `mfront` to the native implementation provided by the `Code_Aster` finite element solver. Graphical illustrations show that the results obtained with both implementations are indistinguishable.

Behaviour and test description	Algorithm	Total computational times		Graphical illustration
		Native	<code>mfront</code>	
Visco-plastic and damaging for steel [33,34]—3D Notched specimen implying large deformation	Implicit, 10 equations	17 min 43 s	7 min 58 s	
Damaging for concrete [35], 3D beam bending	Default parser	45 min	63 min	
Generic Single crystal viscoplasticity [36,37], 3D aggregate, 300 grains	Implicit, 18 equations	28 min	24 min	
FCC single crystal viscoplasticity [38,37], 2D specimen with displacement boundary conditions from EBSD experiment	Implicit, 18 equations	33 min 54 s	29 min 30 s	
FCC homogenised polycrystals [39,37], unit testing	Runge–Kutta 4/5, 30 grains, 1272 equations	9 s 67	8 s 22	 
Anisotropic creep with phase transformation, 3D pipe [40]	Implicit	180 s	171 s	

The `sirius` database is intended to:

- collect, manage, share nuclear fuel material properties and behaviour laws between the fuel performance codes developed within the PLEIADES platform.
- put in place a rigorous checking procedure by an internal expert (for the considered material) who assesses:
 - the consistency of the implementations with respect to reference documents which were used for its development.
 - that the computed results are consistent with the experimental results.
 - the most reliable set of properties and behaviour for a specific material.

Inside `sirius`, information is stored using the domain specific languages provided by `mfront`. `sirius` also stores references (technical reports, publications, books chapters, etc.) that were used to define properties and behaviours.

2.8. Available documentation and web site

We paid particular attention to the documentation. Reference guides, tutorials and courses can be found on the `mfront` web site:

<http://tfel.sourceforge.net/documentations.html>.

Those documents cover:

- `mfront` general usage, implementation of material properties and models.
- implementation of standard mechanical behaviours.
- implementation of finite strain mechanical behaviours.
- plane stress and axisymmetrical generalised plane stress modelling hypothesis handling.
- usage of `mtest`.

Most of the available documentation has been written in French. From the moment `mfront` has been open-sourced, most new documents have been written in English. The website also provides a “Frequently asked questions” section.

All these documents, including the website, are part of the `mfront` sources. New users may also consider looking at the various examples used by the unit tests.

The Sourceforge project allowed us to host the website, put in place a forum and a mailing-list, and provide a download section where the sources of various version of TFEEL are available.

3. Application to the plasticity of FCC single crystals

In this section, we illustrate the `mfront` abilities in the case of a simple crystal constitutive law integrated using the implicit scheme described in Section 2.4.3. For the sake of simplicity, we describe crystal plasticity in the framework of the infinitesimal perturbation hypothesis. A finite strain formulation, based on MANDEL isoclinic configuration, of the same behaviour is also available in the `mfront` sources.

Although the presented law is basic, it shares a common structure with more elaborate descriptions of single crystal plasticity to which the proposed implementation can be extended [38,44,45,37].

We rely on results given in Appendix B for the automatic computation of the consistent tangent operator.

3.1. Constitutive equations

Partition of strain. The total strain $\underline{\varepsilon}^{\text{to}}$ is partitioned in an elastic strain $\underline{\varepsilon}^{\text{el}}$ and a plastic strain $\underline{\varepsilon}^{\text{p}}$:

$$\underline{\varepsilon}^{\text{el}} = \underline{\varepsilon}^{\text{to}} - \underline{\varepsilon}^{\text{p}}. \quad (4)$$

Elasticity. FCC single crystals are orthotropic. Elasticity is described by the Hooke law (Eq. (B.2) in Appendix B).

Plasticity. FCC single crystals have 12 distinct sliding systems. A sliding system s ($s \in [1 : 12]$) is characterised by its normal \underline{n}_s and its slip direction \underline{m}_s . The tensor of directional sense \underline{M}_s is equal to $\frac{1}{2} (\underline{m}_s \otimes \underline{n}_s + \underline{n}_s \otimes \underline{m}_s)$. The plastic deformation rate $\underline{\dot{\varepsilon}}^{\text{p}}$ is the sum of the contribution of each sliding planes:

$$\underline{\dot{\varepsilon}}^{\text{p}} = \sum_{i=1}^{12} \dot{\gamma}_s \underline{M}_s$$

where $\dot{\gamma}_s$ is the sliding rate along the s sliding system.

Resolved shear stress. Sliding along a particular system is driven by the resolved shear stress τ_s :

$$\tau_s = \underline{\sigma} : \underline{M}_s. \quad (5)$$

Sliding rate. The sliding rate is governed by the following equation:

$$\dot{\gamma}_s = \dot{p}_s \text{sgn}(\tau_s - C\alpha_s) \quad (6)$$

where p_s is the equivalent plastic strain, C is a material constant and α_s an internal state variable describing kinematic hardening effects. p_s evolution is given by:

$$\dot{p}_s = \left\langle \frac{|\tau_s - C\alpha_s| - R(p_s)}{K} \right\rangle^m \quad (7)$$

where $\langle x \rangle$ is the positive part of x and where K and m are the material constants. Eq. (7) only describes plasticity approximately: large values of m are thus often used in practice.

Hardening. In this flow rule, isotropic hardening $R(p_s)$ is defined by:

$$R(p_s) = R_0 + Q \sum_r h_{sr} (1 - \exp(-bp_r)). \quad (8)$$

Interaction matrix. Depending on the knowledge of the crystalline material, different choices can be made for the interaction matrix h_{ij} :

Table 2

Comparison of various implementations of the FCC single crystal plastic behaviour. The number of iterations mandatory to converge globally reflects the quality of the tangent consistent operator (see [Appendix B.1](#)): quadratic convergence is achieved in each case.

Implementation	Number of time steps	Number of iterations to reach mechanical equilibrium	Time spend in behaviour integration	Total cpu time
<code>mfront</code> , exact jacobian	60	193	9 min 7 s	38 min 54 s
<code>mfront</code> , numerical jacobian	60	193	21 min 47 s	56 min 17
Code_Aster , exact jacobian	60	197	30 min 35 s	1 h 6 min 57 s

- identity matrix, leading to self isotropic hardening (a system s is only hardened by himself). This case has no physical meaning, but is useful for testing purpose.
- uniform matrix (with an optionally different values on the diagonal). This simple form takes into account the interaction between two systems and seems a bit more physical than the previous one.
- the most appealing approach is to deduce the shape and the values of the interaction matrix from dislocation dynamics simulations [38,44].

Kinematic hardening. The evolution of kinematic hardening α_s is defined by:

$$\dot{\alpha}_s = \dot{\gamma}_s - D\alpha_s \dot{p}_s. \quad (9)$$

3.2. Implicit implementation in `mfront`

In this paragraph, we describe how the crystal plasticity constitutive equations can be integrated in `mfront`.

A cautious analysis of those equations shows that the increments Δp_s and $\Delta \alpha_s$ can be deduced from $\Delta \gamma_s$:

$$\begin{cases} \Delta p_s = |\Delta \gamma_s| \\ \Delta \alpha_s = \frac{1}{1 + D\theta |\Delta \gamma_s|} [\Delta \gamma_s - D\alpha_s|_t |\Delta \gamma_s|]. \end{cases}$$

In `mfront`, such a variable is called *auxiliary* state variables.

The implicit system can thus be reduced to a system of 18 equations whose unknowns $\Delta \underline{\varepsilon}^{\text{el}}$ and $\Delta \alpha_s$:

$$\begin{cases} f_{\underline{\varepsilon}^{\text{el}}} = \Delta \underline{\varepsilon}^{\text{el}} - \Delta \underline{\varepsilon}^{\text{to}} + \sum_{s=1}^{12} \Delta \gamma_s \underline{M}_s \\ f_{\gamma_s} = \begin{cases} \Delta \gamma_s & \text{(no sliding)} \\ \Delta \gamma_s \pm \Delta t \left(\frac{|\tau_s - C\alpha_s|_{t+\theta\Delta t} - R_s(p_s|_{t+\theta\Delta t})}{K} \right)^m & \text{(sliding).} \end{cases} \end{cases} \quad (10)$$

Sliding occurs when the condition $|\tau_s - C\alpha_s|_{t+\theta\Delta t} > R_s(p_s|_{t+\theta\Delta t})$ is met. The Implicit System (10) is implemented in [Fig. 7](#) using the `NewtonRaphson_NumericalJacobian` algorithm. A more efficient implementation of the same behaviour based on the standard `NewtonRaphson` algorithm is also available and distributed with the `mfront` sources.

3.3. Application to aggregate FCC computation

Full-field computations. We used this implementation for the simulation on a tensile test along the z axis for an aggregate of 172 grains, each of them having a distinct orientation of the lattice. Computations were performed with the [Code_Aster](#) finite element solver. Mesh and results are illustrated in [Fig. 8](#).

In terms of computational time, `mfront` implementations are competitive with the `fortran` implementation provided by [Code_Aster](#), as reported in [Table 2](#). Note that numerical jacobian offers a very interesting balance between performance and development time.

Macroscopic validation. It is interesting to extract the macroscopic stress–strain curve (mean σ_{zz} vs. mean $\varepsilon_{zz}^{\text{to}}$) from the previous aggregate computation and to compare it with:

- the response of a single crystal (with orientation [0 0 1]) to emphasise the effect the grains' interaction.
- the response of a homogenised behaviour using the BERVEILLER–ZAOUÏ scheme [39] with the same material properties. This homogenised behaviour was also implemented with `mfront` and integrated with the RUNGE_KUTTA domain specific language due to the large number of internal state variables involved (several thousand). This implementation is also distributed with `mfront` sources.

Those comparisons are depicted in [Fig. 9](#). The homogenised and full-field simulation gives very close stress–strain curves.

```

1  @ComputeStress{
2      sig = D*eel;  // Hooke law
3  }
4
5  @Integrator{
6      // Nss is the number of sliding systems
7      // the internal state variables a,p,g were defined as an array of size Nss
8      // tensor of directional sense are computed once and stored in this object
9      const SlidingSystems& ss = SlidingSystems::getSlidingSystems();
10     real pei[Nss]; // 1 - exp(-b * p[i]+theta*dp[i])
11     for(unsigned short i=0;i!=Nss;++i){
12         pei[i] = Q*(1.-exp(-b*(p[i]+theta*abs(dg[i]))));
13     }
14     feel -= deto; // feel = deel - eto
15     for(unsigned short i=0;i!=Nss;++i){
16         real Rp = R0; // isotropic hardening
17         for(unsigned short j=0;j!=Nss;++j){
18             Rp += mh(i,j)*pei[j] ; // mh is the interaction matrix
19         }
20         real tau = ss.mus[i] | sig ; // resolved shear stress
21         real da = (dg[i]-d1*a[i]*abs(dg[i]))/(1.+d1*theta*abs(dg[i]));
22         real tma = tau-C*(a[i]+theta*da);
23         real tmR = abs(tma)-Rp;
24         if (tmR > 0.){
25             const real sgn = tma > 0 ? 1. : -1.;
26             const strain vpi = dt*pow(tmR/K,m)*sgn;
27             feel += vpi*ss.mus[i] ; // partition of strain
28             fg[i] -= vpi; // g[i] evolution
29         }
30     }
31 }
32
33 @UpdateAuxiliaryStateVars{
34     for(unsigned short i=0;i!=Nss;++i){
35         p[i]+=abs(dg[i]);
36         a[i]+=(dg[i]-d1*a[i]*abs(dg[i]))/(1.+d1*abs(dg[i]));
37     }
38 }

```

Fig. 7. Implementation of the Implicit System (10). The values of the auxiliary state variables α_s and p_s are updated once the convergence is reached.

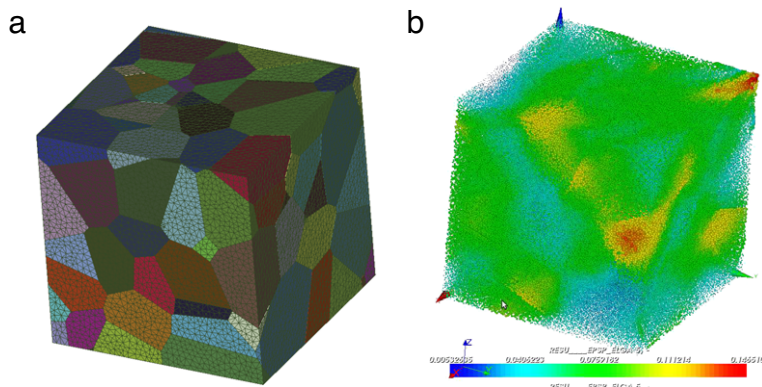


Fig. 8. (a) Mesh and grains (b) ϵ_{zz}^{10} at the end of the computation (values at GAUSS point).

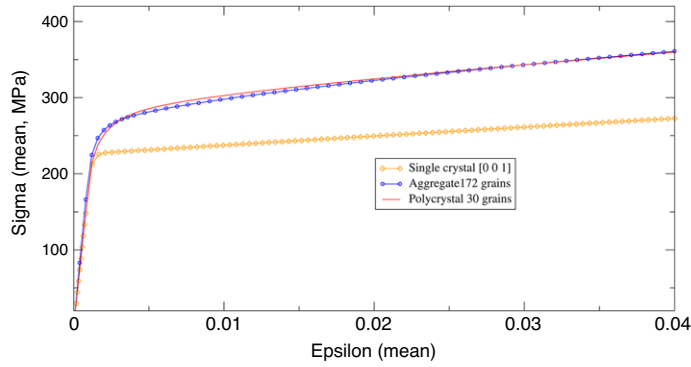


Fig. 9. Stress–strain curves for a 172 grains aggregate, 30 homogeneised grains and single crystal.

3.4. Robustness of the implicit scheme

In complex simulations, notably simulations involving crack propagation, the standard NEWTON–RAPHSON algorithm may fail due to severe local loading at crack tips. To circumvent such difficulty, several modifications, to the NEWTON–RAPHSON algorithm, which will be illustrated in this paragraph, are available in `mfront`.

3.4.1. Analysis of a failure

We analyse in this paragraph a case of failure: a `mtest` input was automatically generated that contains only one time step.

In this particular case, the failure is associated with the fact that at the second iteration the prediction of the stress is almost elastic, so, in Eq. (7), the term $|\tau_s - C\alpha_s| - R(p_s)$ becomes significantly higher than the material property K , which leads to an overestimation of the increment of the plastic strain increment $\Delta \gamma_s$. This problem increases with the value of the exponent m , which is taken equal to 20 in our example. Evolution of the residual $\|F(Y)\|$ with the iteration number is shown in Fig. 10.

The following modifications of the NEWTON–RAPHSON were considered:

1. use the `PowellDog_NewtonRaphson` algorithm. The numerical cost of each iteration is higher than with the standard NEWTON–RAPHSON as the NEWTON step is combined with a GAUSS step. In our implementation, we considered a fixed parameter Δ which defines the size of the trust region: finding an appropriate value for Δ is a major difficulty of this algorithm. A more elaborate version of the POWELL algorithm shall be able to update appropriately Δ during the iterations [28]. Fig. 10 shows how this algorithm performs for various value of the trust region size: 10^{-4} , $2 \cdot 10^{-4}$, $3 \cdot 10^{-4}$ and $5 \cdot 10^{-4}$. Convergence is achieved for each value of the parameter, this algorithm is thus effectively more robust than the NEWTON–RAPHSON algorithm. If the value of the trust region size is too low (10^{-4}), the algorithm can be quite inefficient and favours the GAUSS step over the NEWTON one, thus requiring many iterations to reach convergence. If the value is too high, the regularisation becomes ineffective and the algorithm tends to have the same behaviour as the NEWTON algorithm. The optimal value, close to $3 \cdot 10^{-4}$, is difficult to generalise.
2. limit the size of the internal state variables increments at each iteration. The NEWTON–RAPHSON is thus modified like this:

$$\forall i \quad \Delta Y^{(n+1)}(i) = \Delta Y^{(n)}(i) - \begin{cases} \delta & \text{if } \Delta \Delta Y(i) > \delta \\ \Delta \Delta Y(i) & \text{if } |\Delta \Delta Y(i)| \leq \delta \\ -\delta & \text{if } \Delta \Delta Y(i) < -\delta \end{cases}$$

with $\Delta \Delta Y = J^{-1} \cdot F(\Delta Y^{(n)})$. Limiting the increment of a specified set of internal state variables is also possible. This method can be quite efficient in this example, but suffers the very same difficulty as the POWELL, which is to determine an optimal value for δ .

3. the last method is a manual check for invalid steps based on physical considerations. If the method `@Integrator` returns `false`, we keep the same direction but divide the increment step norm by two:

$$\Delta Y^{(n+1)} = \Delta Y^{(n)} - \frac{1}{2^j} \Delta \Delta Y$$

where j is the number of times the `@Integrator` method returned `false`. This procedure can be compared to a simplified line search. To do this, we simply add the following line in our implementation:

```
if (tmR>alpha*K){
return false;
}
```

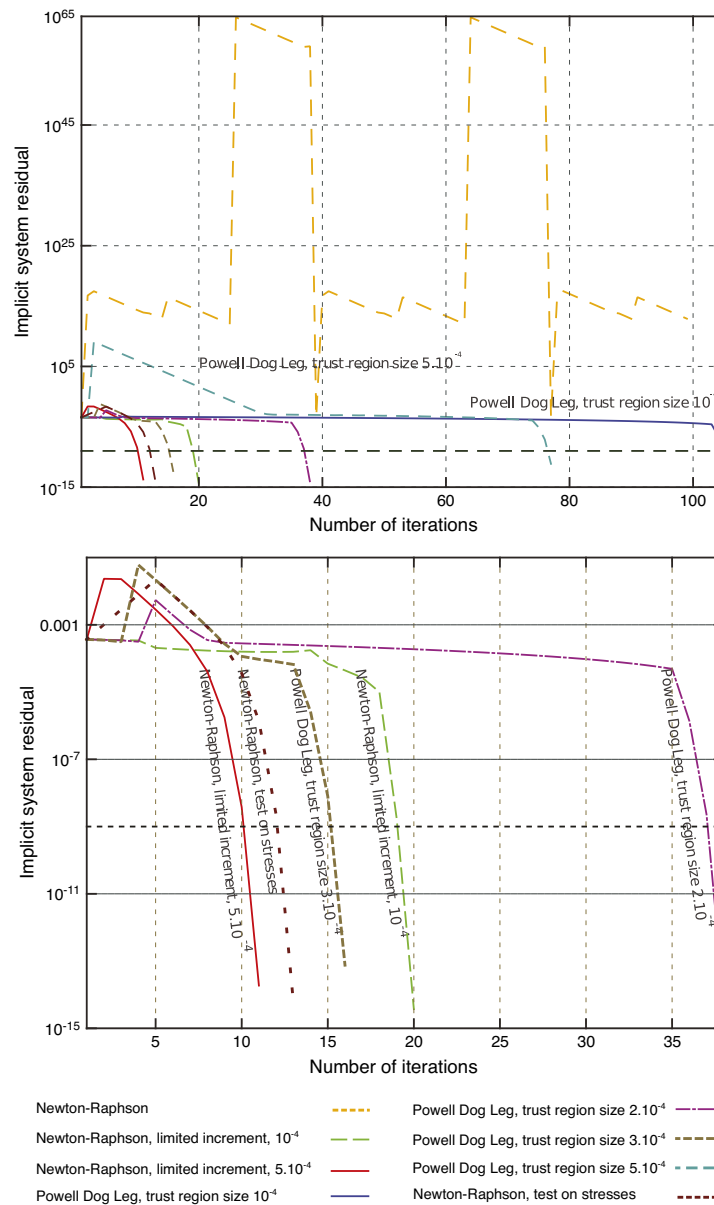


Fig. 10. Comparison of various options enhancing the robustness of the implicit scheme: analysis of a failure of the NEWTON_RAPHSON algorithm. The figure below focuses on the most efficient methods. The convergence criterion value is equal to 10^{-9} .

where \mathbf{tmR} is a variable equal to $|\tau_s - C\alpha_s| - R(p_s)$. Again, the α value is arbitrary, but we found that an appropriate value can be found on some intuitive considerations.³ In the example considered, the second iteration had to be divided four times. Afterwards, the standard NEWTON algorithm is left unchanged.

The number of iterations mandatory to reach convergence is reported in Table 3.

3.4.2. Application to the aggregate computations

For severe simulations, the description of which is out of the scope of this paper, the previous enhancement to the NEWTON_RAPHSON algorithm was found very helpful and efficient.

As those enhancements generally require additional operations to be performed, one may expect the behaviour integration to be less efficient when the standard NEWTON_RAPHSON algorithm gives satisfying results, as it is the case in our example of a simple aggregate computation. Table 4 shows the impact of those additional operations remains reasonable.

³ In our case, m is equal to 20, so 1.05^{20} is close to 2.6 s^{-1} . This may be a lower bound for α .

Table 3
Number of iterations required to reach convergence.

Algorithm	Number of iterations
Newton–Raphson	Divergence
Newton–Raphson, limited increment, $\delta = 10^{-4}$	20
Newton–Raphson, limited increment, $\delta = 5 \cdot 10^{-4}$	11
Powell Dog Leg, trust region size $\Delta = 10^{-4}$	105
Powell Dog Leg, trust region size $\Delta = 2 \cdot 10^{-4}$	38
Powell Dog Leg, trust region size $\Delta = 3 \cdot 10^{-4}$	16
Powell Dog Leg, trust region size $\Delta = 5 \cdot 10^{-4}$	77
Newton–Raphson, test on stresses	13

Table 4
Comparison of various enhancements to the NEWTON–RAPHSO algorithm.

Implementation	Number of time steps	Number of iterations to reach mechanical equilibrium	Time spend in behaviour integration	Total cpu time
mfront, exact jacobian	60	193	9 min 7 s	38 min 54 s
Powell Dog Leg, $\Delta = 3 \cdot 10^{-4}$	60	193	9 min 12 s	35 min 15 s
Newton–Raphson, limited increment, $\delta = 5 \cdot 10^{-4}$	60	193	10 min 13 s	38 min 30 s
Newton–Raphson, test on stresses	60	193	10 min 13 s	38 min 0 s

As a conclusion, we found that our simplified implementation of the Powell Dog Leg algorithm can significantly improve the robustness of implicit schemes. The use of this algorithm was also considered by Shterenlikht and Alexander [46]. Their conclusion is that the Levenberg–Marquardt algorithm can lead to further improvements. The availability of this algorithm in the future versions of mfront is considered.

4. Application to PWR fuel element simulation: fuel pellet behaviour under irradiation

Under irradiation, the fuel material exhibits several mechanical phenomena: elasticity, viscoplasticity, plasticity and damage through the development of cracks. Those phenomena are described in the framework of the small strain behaviours.

The total deformation $\underline{\varepsilon}^{\text{to}}$ is partitioned into various contributions:

$$\underline{\varepsilon}^{\text{to}} = \underline{\varepsilon}^{\text{el}} + \sum_{i=1}^3 \underline{\varepsilon}_i^{\text{c}} + \underline{\varepsilon}^{\text{p}} + \underline{\varepsilon}^{\text{vis}} + \underline{\varepsilon}^{\text{th}} + \underline{\varepsilon}^{\text{s}} \quad (11)$$

where $\underline{\varepsilon}^{\text{el}}$ is the elastic part of the deformation described in Section 4.1, $\underline{\varepsilon}_i^{\text{c}}$ are three strains associated with crack opening described in Section 4.2, $\underline{\varepsilon}^{\text{p}}$ is the fuel plastic strain described in Section 4.3, $\underline{\varepsilon}^{\text{vis}}$ is the fuel viscoplastic strain described in Section 4.4, $\underline{\varepsilon}^{\text{th}}$ is the thermal strain and $\underline{\varepsilon}^{\text{s}}$ are strain associated to swelling due to fission products. In the following, $\underline{\varepsilon}^{\text{th}}$ and $\underline{\varepsilon}^{\text{s}}$ will not be considered.⁴

A remarkable feature of the behaviour treated in this section shall be emphasised: Eq. (11) couples all the phenomena linearly. It means that we may implement and test each mechanism independently and gather them into a final implementation. This procedure is extremely flexible and compensates the fact that the resulting implementation is numerically sub-optimal. In practice, performances were found satisfying.

4.1. Elastic behaviour

The elastic behaviour is considered isotropic:

$$\underline{\sigma} = \lambda(T, f) \operatorname{tr}(\underline{\varepsilon}^{\text{el}}) \underline{I} + 2 \mu(T, f) \underline{\varepsilon}^{\text{el}} \quad (12)$$

where $\lambda(T, f)$ and $\mu(T, f)$ are respectively the first and second LAMÉ coefficients related to the YOUNG modulus $E(T, f)$ and the constant POISSON ratio ν :

$$\lambda(T, f) = \frac{\nu E(T, f)}{(1 + \nu)(1 - 2\nu)} \quad \mu(T, f) = \frac{E(T, f)}{2(1 + \nu)}.$$

⁴ They are usually handled by mechanical solvers as a boundary condition.

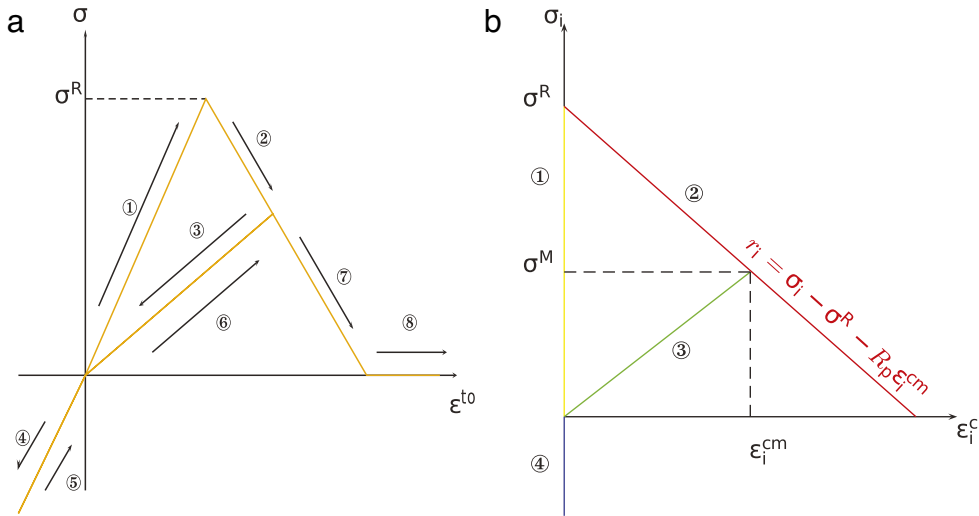


Fig. 11. (a) Stress–strain relationship of the DIFF2 behaviour during a uniaxial tensile test controlled by the deformation. Arrows show how the deformation is controlled at each step of the test. (b) Relation between σ_i and ε_i^c during this test.

The bulk modulus $K(T, f)$ is given by:

$$K(T, f) = \lambda(T, f) + \frac{2}{3}\mu(T, f).$$

The implementation of the YOUNG modulus has been described in Section 2.1. In this paper, the POISSON Ratio is considered constant, equal to 0.3.

4.2. The DDIF2 damage model

A unified description of fuel damage, leading to the formation of a network of cracks for all the fuel materials used within the PLEIADES applications is provided by the DIFF2 model [47]: it introduces a phenomenological anisotropic description of damage through a multi-surface plastic softening law. This description of the fuel damage is consistent with the various modelling hypotheses commonly used to describe the fuel element.⁵

4.2.1. Description of the DIFF2 behaviour in 1D

This section describes the stress–strain relationship obtained during a uniaxial tensile test in traction, compression and traction. The evolution of the stress during the test is illustrated in Fig. 11(a).

The following description can be made:

- a linear elastic response of the material during step ①. The slope is given by the YOUNG modulus.
- a softening of the material once a critical stress σ_R is reached during (step ②). This softening is associated with a growing damage within the material. This damage can be seen as a reduction of maximum stress sustainable by the material and an apparent reduction of the elastic modulus. The post-softening response is chosen linear.
- a linear elastic response on unloading with an apparent modulus lower than the initial one (step ③).
- a linear elastic response in compression (steps ④ and ⑤). The slope given by the initial undamaged YOUNG modulus.
- a linear elastic response with an apparent modulus lower than the initial one in traction (step ⑥).
- a softening of the material (step ⑦).
- the material is completely damaged (step ⑧).

4.2.2. Constitutive equations

The DIFF2 behaviour describes the damage of a material in three orthotropic directions, each represented by a vector \vec{e}_i ($i \in [1, 2, 3]$) defining a crack plane. In three dimensions, those directions can be chosen a priori or be determined during the computations. Once the principal stress is found greater than the critical stress σ_R , the associated eigenvector determines a crack plane which is then fixed for the rest of the computations. For the sake of simplicity, only predetermined crack planes are considered in this paper.

⁵ For example, most fuel performance code uses a monodimensional modelling of the fuel rod [48–50] in which an isotropic damage would not suit [51].

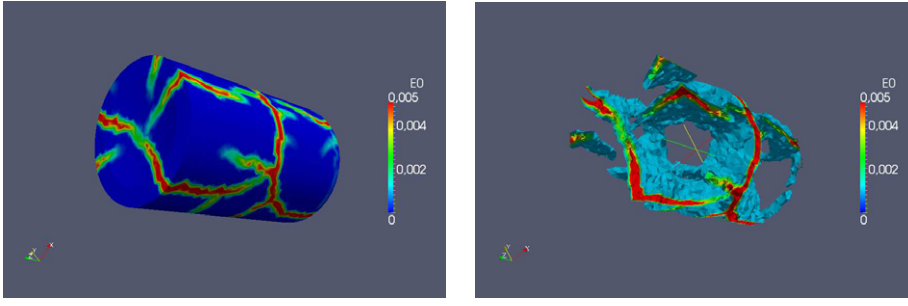


Fig. 12. Fuel pellet fragmentation predicted by the DDIF2 behaviour (computations performed using the [Cast3M](#) finite element solver).

Damage and crack opening in each direction are taken into account through an additional strain $\underline{\varepsilon}_i^c$:

$$\underline{\varepsilon}_i^c = \varepsilon_i^c \underline{n}_i = \varepsilon_i^c \vec{e}_i \otimes \vec{e}_i. \quad (13)$$

The damage level of the material is characterised by the maximum value reached by the variable ε_i^c , noted ε_i^{cm} :

$$\varepsilon_i^{cm} = \max_{\tau \leq t} (\varepsilon_i^c(\tau)).$$

The DIFF2 behaviour relates the strain ε_i^c and the projection $\sigma_i = \underline{\sigma} : \underline{n}_i$ of the stress in direction i through a yield surface:

$$\sigma_i - \sigma_R - R_p \varepsilon_i^{cm} \leq 0 \quad (14)$$

where R_p is a softening modulus.

Eq. (14) defines the evolution of the variable ε_i^c during damage increase, when this variable is equal to its maximum value ε_i^{cm} . During unloading, the ε_i^c is related to the stress σ_i through the following linear relationship:

$$\varepsilon_i^c = \frac{\sigma_i}{\sigma_i^M(\varepsilon_i^{cm})} \varepsilon_i^{cm} \quad 0 < \sigma_i < \sigma_i^M(\varepsilon_i^{cm}) \quad (15)$$

where $\sigma_i^M(\varepsilon_i^{cm})$ is the maximum sustainable stress still sustainable by the material:

$$\sigma_i^M(\varepsilon_i^{cm}) = \sigma_R + R_p \varepsilon_i^{cm}.$$

In compression state ($\sigma_i < 0$), the ε_i^c is equal to 0.

[Fig. 11\(b\)](#) shows the relation between ε^c and the stress σ during the uniaxial tensile test described in the previous section. The integration of those constitutive equations using an implicit scheme is described in depth in [Appendix D](#).

4.2.3. Fuel pellet fragmentation

Alone (without additional plastic and viscoplastic flow), the DIFF2 model is able to describe the fuel pellet fragmentation that occurs during the first power increase, as depicted in [Fig. 12](#): stresses within the fuel are driven by the thermal strain gradient (the temperature as a function of the radius has a parabola-like shape). Due to the local nature of the behaviour, the results are mesh-sensitive. However, if the mesh is sufficiently isotropic (no preferential direction), the crack network pattern and the number of fragments are merely mesh-insensitive and are in accordance with experimental evidence.

4.3. Plastic behaviour

Decohesion of grain boundaries is modelled by a dilatant plastic flow using a DRUCKER–PRAGER yield criterion:

$$f_p(\underline{\sigma}) = A p + \sigma_{eq} - k(T) \leq 0 \quad (16)$$

where p is the hydrostatic pressure $p = \frac{1}{3} \text{tr}(\underline{\sigma})$, and σ_{eq} is the VON-MISES stress.

The plastic strain $\underline{\varepsilon}^p$ evolution is associated:

$$\underline{\dot{\varepsilon}}^p = \dot{p}^p \frac{\partial f}{\partial \underline{\sigma}} = \dot{p}^p \left[\frac{A}{3} \underline{I} + \underline{n} \right] = \dot{p}^p \underline{n}^p$$

where p_p is a state variable and $\underline{n} = \frac{\partial \sigma_{eq}}{\partial \underline{\sigma}}$ (see [Appendix C](#)). \dot{p}^p is such that the Yield Criterion (16) is satisfied at the end of the time step.

The change of volume \dot{p}_v^p associated with the plastic flow $\underline{\dot{\varepsilon}}^p$ is given by:

$$\dot{p}_v^p = \text{tr}(\underline{\dot{\varepsilon}}^p) = A \dot{p}^p$$

\dot{p}_v^p contributes to the porosity evolution \dot{f} described in Section 4.5.

```

1      feel      += dpl*np_;
2      dfeel_ddelel += 2.*mu*theta*dpl*inv_seq*(Stensor4::M()-(n_^n_));
3      dfeel_ddpl = np_;
4      const strain g = (a*pr+seq-k)/young;
5      if ((g>0) || (dpl>epsilon)) {
6          Stensor np_ = a/3*Stensor::Id()+n_;
7          fpl = g;
8          dfpl_ddpl = 0;
9          dfpl_ddelel = (a*K*Stensor::Id()+2*mu*n_)*theta/young;
0      }

```

Fig. 13. Plastic part.

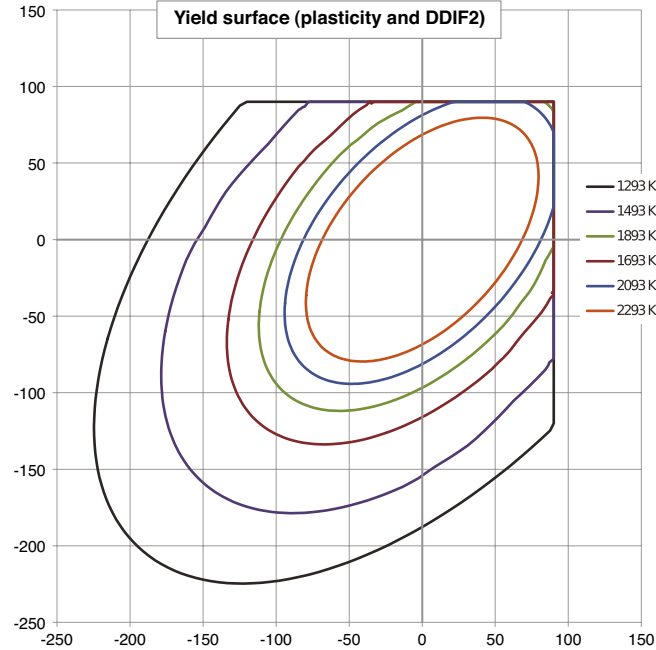


Fig. 14. Stress limits obtained by combining dilatant plasticity and DDIF2. This figure was obtained by tensile tests in all directions of the σ_{xx} vs. σ_{yy} plane using the `mtest`, assuming σ_{zz} is zero.

Implicit resolution. If the value of $f_p(\underline{\sigma})$ is positive, the residual f_{p^p} associated with the state variable p^p is given by normalising the Yield Criterion (16):

$$f_{p^p} = \frac{1}{E} [A p + \sigma_{eq} - k(T)].$$

Given the results of Appendix C, derivatives of f_{p^p} are obvious:

$$\frac{\partial f_{p^p}}{\partial \underline{\varepsilon}^{el}} = \frac{\theta}{E} [A K + 2 \mu \underline{n}] \quad \frac{\partial f_{p^p}}{\partial p^p} = 0.$$

Implementation. The implementation of the plastic part of this behaviour is reported in Fig. 13.

Unit testing. Fig. 14 represents the stress boundaries in a σ_{xx} vs. σ_{yy} plane obtained by combining the DDIF2 cracking model described in the previous section and the Yield Criterion (16).

4.4. Compressible viscoplastic behaviour

The macroscopic viscoplastic behaviour is defined through an effective dissipation potential with a hyperbolic stress-dependency:

$$\underline{\underline{\varepsilon}}^{vis} = \frac{\partial \Phi}{\partial \underline{\underline{\sigma}}} \quad \text{with } \Phi(s_{eq}(p, \sigma_{eq})) = \sigma_0 K \exp\left(-\frac{Q}{RT}\right) \cosh\left(\frac{s_{eq}}{\sigma_0}\right) \quad (17)$$

where σ_0 , K , Q are material parameters identified via compression tests and s_{eq} an equivalent stress described below.

Equivalent stress. Following Monerie and Gatt [52], the equivalent stress $s_{eq}(p, \sigma_{eq}, f)$ is defined as function of the hydrostatic pressure $p = \frac{1}{3} \text{tr}(\sigma)$, the VON-MISES stress σ_{eq} and the porosity f :

$$s_{eq}(p, \sigma_{eq}) = \sqrt{A(f) p^2 + B(f) \sigma_{eq}^2} \quad (18)$$

where $A(f)$ and $B(f)$ are given by the following functions of the porosity:

$$\begin{cases} A(f) = \frac{9}{4} (E (f^{-1/E} - 1))^{-\frac{2E}{E+1}} \\ B(f) = \left(1 + \frac{2}{3}f\right) (1-f)^{-\frac{2E}{E+1}} \end{cases}$$

E is a constant chosen equal to 6.

Usual expression of the viscoplastic flow. Eqs. (17) and (18) lead to an explicit expression of the viscoplastic flow:

$$\underline{\varepsilon}^{vis} = \frac{\partial \phi}{\partial \underline{\sigma}} = \frac{\partial \phi}{\partial s_{eq}} \frac{\partial s_{eq}}{\partial \underline{\sigma}} = \frac{\partial \phi}{\partial s_{eq}} \left[\frac{\partial s_{eq}}{\partial p} \frac{\partial p}{\partial \underline{\sigma}} + \frac{\partial s_{eq}}{\partial \sigma_{eq}} \frac{\partial \sigma_{eq}}{\partial \underline{\sigma}} \right] = \frac{\partial \phi}{\partial s_{eq}} \left[\frac{1}{3} \frac{\partial s_{eq}}{\partial p} \underline{I} + \frac{\partial s_{eq}}{\partial \sigma_{eq}} \underline{n} \right]$$

where \underline{n} is the derivative of the VON MISES stress with respect to the stress (see Appendix C.1).

This expression states that the viscoplastic flow can be decomposed into a volumetric change characterised by the state variable p_v and a deviatoric flow characterised by an equivalent viscoplastic strain p_d :

$$\underline{\varepsilon}^{vis} = \dot{p}_v^{vis} \underline{I} + \dot{p}_d^{vis} \underline{n} \quad \text{with} \quad \begin{cases} \dot{p}_v^{vis} = \frac{\partial \phi}{\partial s_{eq}} \frac{\partial s_{eq}}{\partial p} \\ \dot{p}_d^{vis} = \frac{\partial \phi}{\partial s_{eq}} \frac{\partial s_{eq}}{\partial \sigma_{eq}} \end{cases} \quad (19)$$

The remaining derivatives are given by (see Appendix C for details):

$$\frac{\partial \phi}{\partial s_{eq}} = K \exp\left(-\frac{Q}{RT}\right) \sinh\left(\frac{s_{eq}}{\sigma_0}\right) \quad \frac{\partial s_{eq}}{\partial p} = \frac{1}{s} A(f) p \quad \frac{\partial s_{eq}}{\partial \sigma_{eq}} = \frac{1}{s} B(f) \sigma_{eq}.$$

Implementation. The implicit equations associated with Eq. (19) are:

$$\begin{cases} f_{p_v^{vis}} = \Delta p_v^{vis} - \frac{\partial \phi}{\partial s_{eq}} \bigg|_{t+\theta \Delta t} \frac{\partial s_{eq}}{\partial p} \bigg|_{t+\theta \Delta t} \\ f_{p_d^{vis}} = \Delta p_d^{vis} - \frac{\partial \phi}{\partial s_{eq}} \bigg|_{t+\theta \Delta t} \frac{\partial s_{eq}}{\partial \sigma_{eq}} \bigg|_{t+\theta \Delta t} \end{cases}.$$

The derivatives of those equations with respect to the porosity are neglected. The other jacobian terms associated with those terms, although a bit tedious, do not present any difficulty.

The implementation relative to the viscoplastic part of the behaviour is reported in Fig. 15.

4.5. Porosity evolution

Mass balance can be used to relate the evolution of the porosity to the change of volume associated with the plastic and viscoplastic flow:

$$\dot{f} = (1-f) \text{tr}(\underline{\dot{\varepsilon}}^{vis} + \underline{\dot{\varepsilon}}^p) = (1-f) (\dot{p}_v^{vis} + \dot{p}_v^p). \quad (20)$$

Implementation. The implementation equation associated with Eq. (20) is:

$$f_f = \Delta f (1 + \theta (\Delta p_v^{vis} + \Delta p_v^p)) - (1-f|_t) (\Delta p_v^{vis} + \Delta p_v^p).$$

The implementation of this equation and its derivative is trivial and is not reported here.

4.6. Concluding remarks on this implementation

This section described how a fairly complex behaviour, mixing damage, viscoplasticity and dilatant plasticity can be implemented using an implicit scheme with analytical jacobian: while only the most relevant parts were reported here, the complete `mfront` file is 220 lines long.

The presented implementation was introduced in the Alcyone fuel performance code [47,2] and used to simulate a fuel rod irradiated in the CABRI reactor in the framework of safety studies intended to verify the integrity of high burn-up

```

1  feel      += (dpv/3)*Stensor::Id()+(dpd+def)*n_ ;
2  dfeel_ddeel += 2.*mu*theta*(dpd+def)*inv_seq*(Stensor4::M()-(n_^n_));
3  dfeel_ddpv  = (real(1)/3)*Stensor::Id();
4  dfeel_ddpd  = n_;
5  dfeel_ddef  = n_;
6  // equivalent stress
7  const real Af = pow((pow(f_,-1/E)-1)*E,-2*E/(E+1));
8  const real Bf = (1.+2.*f_/real(3))*pow(1.-f_,-2*E/(E+1));
9  const real B = (Bf+20.*Af)/Bf;
10 const real A = (9./4.)*(Af/Bf)*B;
11 const real s  = sqrt(A*pr*pr+B*seq*seq);
12 if(s>1.e-8*young){
13     const real tmp_e      = exp(s/sig0);
14     const real tmp_s      = 0.5*(tmp_e-(1./tmp_e));
15     const real tmp_c      = 0.5*(tmp_e+(1./tmp_e));
16     const real ds_dpr     = A*pr/s;
17     const real ds_dseq    = B*seq/s;
18     const real d2s_dpr2   = A/s - 1.*A*A*pr*pr/(s*s*s);
19     const real d2s_dseq2  = B/s - 1.*B*B*seq*seq/(s*s*s);
20     const real d2s_dprdseq = -1.*A*B*pr*seq/(s*s*s);
21     const real dphi_ds    = K0 * tmp_s;
22     const real d2phi_ds2  = K0 / sig0 * tmp_c;
23     const real dphi_dpr   = dphi_ds*ds_dpr;
24     const real dphi_dseq  = dphi_ds*ds_dseq;
25     const real d2phi_dpr2 = d2phi_ds2*ds_dpr*ds_dpr +dphi_ds*d2s_dpr2;
26     const real d2phi_dprdseq = d2phi_ds2*ds_dpr*ds_dseq +dphi_ds*d2s_dprdseq;
27     const real d2phi_dseq2 = d2phi_ds2*ds_dseq*ds_dseq+dphi_ds*d2s_dseq2;
28     // hydrostatic part
29     fpv      -= dphi_dpr*dt;
30     dfpv_ddeel = -theta*dt*(d2phi_dpr2*K*Stensor::Id()+2*mu*d2phi_dprdseq*n_);
31     // deviatoric part
32     fpd      -= dphi_dseq*dt;
33     dfpd_ddeel = -theta*dt*(2*mu*d2phi_dseq2*n_+K*d2phi_dprdseq*Stensor::Id());
34     // irradiation creep
35     fef      -= Kf*fii*seq*dt;
36     dfef_ddeel = -2*mu*theta*Kf*fii*dt*n_ ;
37 }

```

Fig. 15. Viscoplastic part.

fuels during Reactivity Initiated Accidents (RIA) in Pressurised Water Reactors (PWR) [54,53]. Fig. 16 shows that the plastic deformation localisation does match the experimental area where grain boundaries decohesion was detected.

5. Conclusions

This paper gives a general overview of a code generator named *mfront* that was recently released in an open-source licence to allow its use by researchers and engineers in solid mechanics. We feel that the creation of a broad community of users would help increasing the quality and robustness of this software.

A set of domain specific languages were introduced to ease the implementations of material properties and mechanical behaviours. Benchmarks show that the generated code performance is on par or better than implementations available in the *Code_Aster* finite element solver.

A set of interfaces allows mechanical behaviours to be shared between various solvers. New interfaces for various solvers are already considered. We feel that supporting de facto standard finite element solvers, such as Abaqus and Ansys, would greatly increase the number of potential users. Any contribution in that sense would be welcomed.

Acknowledgements

This research was conducted in the framework of the PLEIADES project, which was supported financially by the CEA (Commissariat à l'Énergie Atomique et aux Énergies Alternatives), EDF (Électricité de France) and AREVA and in the framework of the Simu-Meca2015 project hold within EDF R&D.

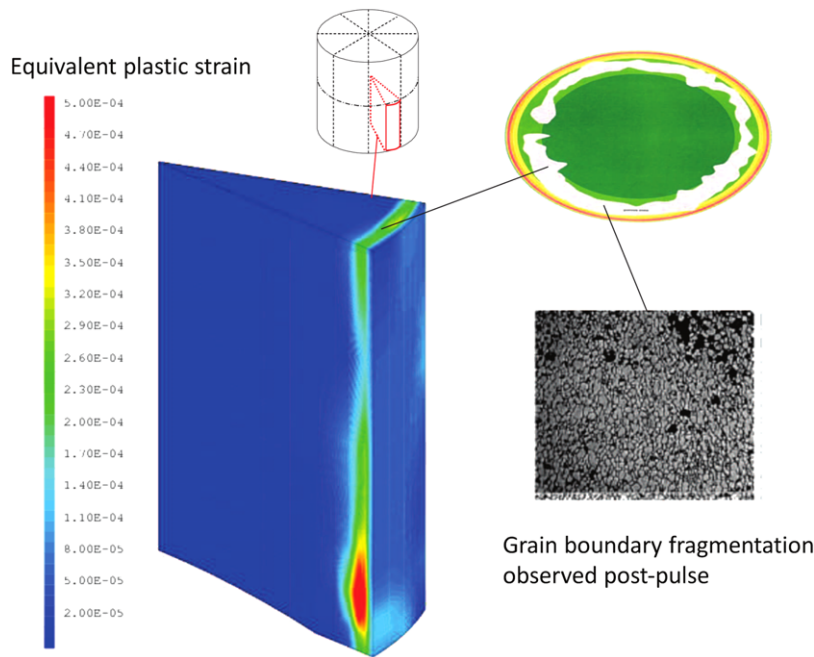


Fig. 16. Results of a 3D simulation performed in the framework of safety studies intended to verify the integrity of high burn-up fuels during Reactivity Initiated Accidents (RIA) in Pressurised Water Reactors (PWR) in the CABRI reactor. Comparison of the equivalent plastic strain to experimental evidences of grain boundaries decohesion. Images at the right were extracted from papers of Schmitz et al. [53] and Frizonnet et al. [54].

Appendix A. The TFEL library

Traditional object oriented approaches classically introduce an abstraction cost that may be leveraged by the use of generic programming [55,56].

This section provides an overview of the programming techniques used in the TFEL mathematical library. Each of those techniques has already been described individually in previous works. The strength of the proposed work is to put them in a coherent library, based on an original implementation of concept on the basis of the C++98 standard as described in Appendix A.5, in order to achieve a high performance library suitable for the implementation of mechanical behaviour, as described in Section 2.

The use of those techniques allows the use of a high level expression of the mechanical behaviours using tensorial formalism while retaining a performance level which are on par or outperform previous implementations, notably those made with the `fortran` language. Those points were treated in depth in Sections 2 and 2.6.

However, those advanced techniques are not suitable for direct use by engineers. The development of the `mfront` code generator, described in Section 2, was partially motivated by the need to hide those technical details: `mfront` provides higher level domain specific languages built on top of the TFEL library and the C++ language.

A.1. Dimensional analysis

Dimensional analysis is a useful tool to prevent standard errors.

The TFEL library introduces the `qt` template class which takes two arguments: a floating point number class and class representing a unit. For a given binary operation, the library provides appropriate metafunctions which guarantee that this operation is valid (one cannot add a stress value and a strain value) and compute the type result. Those checks being made at compilation stage, they do not affect the computational efficiency. The implementation made closely follows the work of D. Abrahams and A. Gurtovoy [57].

A.2. Symmetric tensors

Symmetric tensors are one of the most used mathematical objects in the implementation of a mechanical behaviour. Those objects will be used to illustrate the various programming techniques used in the TFEL library.

Symmetric tensors are implemented by the `stensor` template class which takes two template arguments:

- the spatial dimension. The only valid values are 1, 2 or 3.
- the type of values hold.

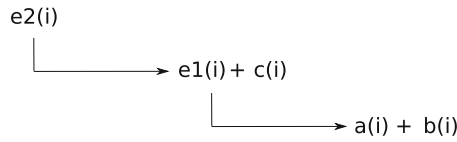


Fig. A.17. Application of the expression templates technique to the addition of three symmetric tensors.

In three dimensions, a symmetric tensor \underline{s} contains 6 independent values and may be represented by an array of values^{6,7}:

$$\underline{s} = (s_{xx}, s_{yy}, s_{zz}, \sqrt{2} s_{xy}, \sqrt{2} s_{xz}, \sqrt{2} s_{yz}). \quad (\text{A.1})$$

In one dimension, a symmetric tensor only contains the 3 diagonal elements.

Those objects can be allocated on the stack, which enables C++ compilers to highly optimise the code and check for illegal memory access at compile-time.

A.3. Expression templates

One may expect that the addition of two symmetric tensors results in a new symmetric tensor. This naive approach may lead to poor performances, due to temporaries objects and data copying [56].

Expression templates is a C++ template metaprogramming technique which introduces additional classes which represent the actions to be performed on some objects and lazily delay the execution of those actions until the result is explicitly requested.

The objects of those classes are placeholders, also called handlers within the library, that are meant to be assigned to an object whose type is the expected result of the operation treated. The distinction between the handler of the operation and the expected result is mandatory to understand how the `ComputeBinaryResult` class, discussed in [Appendix A.5.1](#), works.

To illustrate this technique, let us consider the addition of three symmetric tensors a , b and c and its assignment to a symmetric tensor d :

```
1 d = a + b + c;
```

The addition of the symmetric tensors a and b produces an intermediate object $e1$ of type `Expr1` which keeps a reference to those two symmetric tensors. Similarly, the addition of those three symmetric tensors defines another object $e2$ of type `Expr2` which stands for the addition of $e1$ and the symmetric tensor c .

[Fig. A.17](#) shows how the access operator of $e2$ is implemented. The assignment of an object of type `Expr2` to a symmetric tensor d is implemented as a standard `for` loop:

```
1 for(size_type i=0;i!=d.size();++i){
2   d[i] = e2(i);
3 }
```

The temporary objects $e1$ and $e2$ are meant to be eliminated by the compiler optimisation process. Thanks to function inlining, the compiler is able to produce a code that is equivalent to what would have been obtained with the following instructions:

```
1 for(size_type i=0;i!=d.size();++i){
2   d[i] = a[i] + b[i] + c[i];
3 }
```

A.4. Finite size algorithms

In the case of objects whose size is known at compilation, additional techniques can be used to manually unroll the previous loop. TFEL implements known size version of the algorithms provided by the standard C++ library, following an idea introduced in the Matrix Template Library [59]. In this case, the addition of three symmetric tensors leads to a code equivalent to:

⁶ The $\sqrt{2}$ factor appearing in Eq. (A.1) is introduced so that the contracted product of two symmetric tensors is obtained by the scalar products of the associated arrays of values [58].

⁷ The order of the components follows the conventions introduced in the `Cast3M` and `Code_Aster` finite element codes [7,6].

```

1 d[0] = a[0] + b[0] + c[0];
2 d[1] = a[1] + b[1] + c[1];
3 ...
4 d[N] = a[N] + b[N] + c[N];

```

where N is the size of the symmetric tensors.

Those finite-size algorithms are used whenever it is possible.

A.5. An implementation of concepts based on C++-98 standard

The expression templates technique introduces new classes build upon the mathematical operations made. As mentioned in [Appendix A.3](#), objects of those classes are usually meant to be eliminated by the compiler optimisation process. However, it is convenient that those objects act like the mathematical object they stand for. For example, one may expect to be able to compute the trace of the addition of two symmetric tensor s_1 and s_2 , i.e. the following code shall be valid:

```

1 trace(s1+s2)

```

and this code shall be equivalent to⁸:

```

1 s1(0)+s2(0)+s1(1)+s2(1)+s1(2)+s2(2)

```

The `trace` function shall then accept any argument that stands for a symmetric tensor. The TFEL library introduces an implementation of concepts, based on the C++98 standard, which precisely defines the meaning of the previous sentence.

The concept term was introduced in the Standard Template Library (STL) [60] as a simple description of the requirements for a particular type, usually being a template parameter. Concepts are at the core of generic programming and are used to specify the abstractions that facilitate the use of generic libraries [57]. Despite their importance, concepts are not explicitly supported in C++ [61]. There was a proposal to add concepts as an explicit language feature in the C++11 standard, though it was rejected.

The implementation of concepts used in the TFEL library is limited by the C++ 98 standard and is based on a classical C++ idiom, called the curiously recurring template pattern (CRTP), in which a class X derives from a class template instantiation using X itself as template argument [62].

Using CRTP, the base class have access to its (uniq) child. For our purpose, this idiom allows to the base class to put constraints on the derivate class enforcing some formal properties of the mathematical object it stands for by using static assertion and metafunctions [57].

The `stensor<N, real>` class briefly introduced in [Appendix A.2](#) inherits from the `StensorConcept<stensor<N, real>>` concept class. One of the requirements of the `StensorConcept` concept is that the `stensor<N, real>` class must provide an access operator which takes an unsigned short argument.

We have introduced a metafunction named `Implements` which takes two template arguments: a type name and a concept. This metafunction returns true (contains a static boolean named `cond`) if the given type implements the given concept, false otherwise:

```

1 template<typename T, template<typename> class concept>
2 struct TFEL_VISIBILITY_LOCAL Implements
3 {
4     static const bool cond = tfel::meta::IsSuperClassOf<concept<T>,T>::cond;
5 };

```

A.5.1. Concept based disambiguation

The implementation of concepts proposed in the previous section can be used to disambiguate functions or operators definition. This is how the addition operator between two symmetric tensor is defined⁹:

```

1 template<typename T1,typename T2>
2 TFEL_MATH_INLINE
3 typename tfel::meta::EnableIf<
4     tfel::meta::Implements<T1,StensorConcept>::cond&&
5     tfel::meta::Implements<T2,StensorConcept>::cond&&
6     !tfel::typetraits::IsValid<typename ComputeBinaryResult<T1,T2,OpPlus>::Result>::cond,
7     typename ComputeBinaryResult<T1,T2,OpPlus>::Handle
8     >::type
9 operator + (const T1&,const T2&);

```

⁸ The representation of symmetric tensors as an array of values given by Eq. (A.1) shows that the trace is computed by adding the three first values of this array.

⁹ Though unusual within C++ 98 standard, this operator definition is reasonably similar to what would have been using the concepts lite extension envisaged for the next C++ standard [63].

The typename `tfel::meta::EnableIf` is equivalent to the `std::enable_if` class introduced in the C++11 standard [64]. It takes two template arguments:

- a boolean. If the given boolean is false, the function definition is invalidated using the Substitution Failure Is Not An Error (SFINAE) principle [65].
- a returned type. This returned type is only taken into account if the boolean given as the first template parameter is true.

In the case of the addition operator, this definition is only valid if the following requirements are met by the two types T1 and T2:

- T1 and T2 both implement the `Stensor` concept;
- the result of the addition of the object of types T1 and T2 is valid. The validity of this operation is assessed by the `ComputeBinaryResult`.

The `ComputeBinaryResult` metafunction is one key part of the `tfel` library. In `TFEL`, the addition of two symmetric tensors of type T1 and T2 is valid if:

- T1 and T2 stand for symmetric tensors having the same spatial dimension. For example, one cannot add a one-dimensional symmetric tensor and a three-dimensional one;
- it is valid to add the values hold by this two tensors. For example, one cannot add a stress tensor and a strain tensor.

The `ComputeBinaryResult` metafunction has two outputs:

- `Handle` which is the type of the returned value by the operator, following the expression templates technique described in [Appendix A.3](#);
- `Result` which is the type for which the `Handle` type is a placeholder.

A.5.2. Traits based disambiguation and static polymorphism

Another technique used in the `TFEL` library is based on type traits [66,67,57].

The `StensorTraits` is a template class whose argument fulfils the requirements of the symmetric tensor concept. It contains:

- the spatial dimension of the symmetric tensor represented by its template argument;
- the type of values held by its template argument;

Those information are used to provide specialised algorithms for specific cases. For example, the determinant of a symmetric tensor is always computed using a function named `det` but the specific implementation used is determined at compile-time:

- in one dimension, the value of the determinant is obtained by multiplying the diagonal elements;
- in three dimensions, this determinant is computed by the rule of SARRUS.

Whenever relevant (computations of eigenvalues and/or eigenvectors), specialised algorithms are provided based on the dimension used.

A.6. Portability

An important effort has been put to assess that the `TFEL` library has production quality. Portability is a hint.

`TFEL` supports four different compilers: `gcc` [68], `clang`, `icpc`, `PathScale`.

`TFEL` is cross-platform and available on Windows and various systems based on unix, for example `FreeBSD` and `Solaris`. The main developing platform is `Linux`.

Appendix B. Classical results on implicit resolution for small strain behaviour

This appendix considers small strains behaviours. We make the following assumptions:

- the equation associated with the elastic strain reflects the strain partitioning hypothesis:

$$f_{\varepsilon^{\text{el}}} = \Delta \varepsilon^{\text{el}} + \dots - \Delta \varepsilon^{\text{to}} \quad (\text{B.1})$$

- the strain increment $\Delta \varepsilon^{\text{to}}$ does not appear in any other equation of the implicit system than Eq. (B.1).

For the sake of simplicity, we also make the assumption that the stresses are related to the elastic strains by the Hooke law through a constant elastic stiffness $\underline{\underline{D}}$:

$$\underline{\underline{\sigma}} = \underline{\underline{D}} : \underline{\underline{\varepsilon}}^{\text{el}}. \quad (\text{B.2})$$

The results given in this appendix can easily be extended to a more general elasticity law.

```

1  @TangentOperator{
2      if ( (smt==ELASTIC) || (smt==SECANTOPERATOR) ) {
3          Dt = D;
4      } else if (smt==CONSISTENTTANGENTOPERATOR) {
5          Stensor4 iJe;
6          getPartialJacobianInvert(iJe);
7          Dt = D*iJe;
8      } else {
9          return false;
10     }
11 }

```

Fig. B.18. A standard implementation of the consistent tangent operator computation based on Eq. (B.5).

B.1. Consistent tangent operator computations

The previous assumptions can be used to derive the tangent consistent operator $\frac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\varepsilon}^{to}}$ from the jacobian matrix of the implicit system. Using Eq. (B.2), we have:

$$\frac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\varepsilon}^{to}} = \frac{\partial \underline{\sigma}|_{t+\Delta t}}{\partial \Delta \underline{\varepsilon}^{to}} = \underline{D} : \frac{\partial \underline{\varepsilon}^{el}|_{t+\Delta t}}{\partial \Delta \underline{\varepsilon}^{to}} = \underline{D} : \frac{\partial \Delta \underline{\varepsilon}^{el}}{\partial \Delta \underline{\varepsilon}^{to}}.$$

To compute $\frac{\partial \Delta \underline{\varepsilon}^{el}}{\partial \Delta \underline{\varepsilon}^{to}}$, let us consider a variation $\delta \Delta \underline{\varepsilon}^{to}$ of the total strain increment. Differentiation of the implicit system:

$$f(\Delta \underline{\varepsilon}^{el}(\delta \Delta \underline{\varepsilon}^{to}), \Delta Z_i(\delta \Delta \underline{\varepsilon}^{to}), \delta \Delta \underline{\varepsilon}^{to}) = 0 \quad (B.3)$$

using the chain rule and the assumptions made below leads to the following relation:

$$J \begin{pmatrix} \delta \Delta \underline{\varepsilon}^{el} \\ \delta \Delta Z_i \end{pmatrix} = \begin{pmatrix} \delta \Delta \underline{\varepsilon}^{to} \\ 0 \end{pmatrix} \Rightarrow \delta \Delta \underline{\varepsilon}^{el} = J_{\underline{\varepsilon}^{el}}^{-1} \delta \Delta \underline{\varepsilon}^{to} \quad (B.4)$$

where $J_{\underline{\varepsilon}^{el}}^{-1}$ is the $N \times N$ left upper part of the inverse of the jacobian, N being size of symmetric tensors for the spatial dimension considered. Eq. (B.4) being true for all perturbation $\delta \Delta \underline{\varepsilon}^{to}$, we conclude that the derivative $\frac{\partial \Delta \underline{\varepsilon}^{el}}{\partial \Delta \underline{\varepsilon}^{to}}$ is equal to $J_{\underline{\varepsilon}^{el}}^{-1}$.

The consistent tangent operator is thus equal to:

$$\frac{\partial \Delta \underline{\sigma}}{\partial \Delta \underline{\varepsilon}^{to}} = \underline{D} : J_{\underline{\varepsilon}^{el}}^{-1}. \quad (B.5)$$

If the computation of the stresses involves other states variables, Eq. (B.4) can be generalised to obtain their derivatives relative to the increment of the total strain $\Delta \underline{\varepsilon}^{to}$: those derivatives are read in the N th first column of the inverse of the jacobian J .

Several methods generated by `mfront` allow the computations of $J_{\underline{\varepsilon}^{el}}^{-1}$ and any other state variable derivatives if necessary. The computation of consistent tangent operator is generally close to the source code reported in Fig. B.18. Although there is no general answer to this question whether using Eq. (B.4) is more efficient than computing the consistent tangent operator analytically, we shall emphasise the following points:

- the analytic computation may require complex operations involving special mathematical functions;
- In `mfront`, the computation of the consistent tangent operator occurs after the last `NEWTON` step, so a LU decomposition of the jacobian J has already been performed. Therefore, computing $J_{\underline{\varepsilon}^{el}}^{-1}$ is a cheap operation. However, the quality of the consistent tangent operator greatly depends on the criterion value used to stop the `NEWTON` algorithm, which must generally be smaller (typically 10^{-11}) than what is usually used for determining the state variable increments (typically 10^{-9}). This is due to the fact that Eq. (B.3), which is a crucial point in the method presented here, is only approximately satisfied (up to the criterion value). In practice, this generally means that at least one another iteration and one more LU decomposition of the jacobian must be performed.

B.2. Plane stress modelling hypothesis

The plane stress modelling hypothesis generally requires specific integration algorithms to be implemented. However, using the assumptions made at the beginning of this appendix, the plane strain implementation of the behaviour can

```

1  @Integrator<PlaneStress , Append , AtEnd>{
2      fetozz    = 1./D(2,2)*(eel(2)+deel(2)+D(0,2)*(eel(0)+deel(0))+D(1,2)*(eel(1)+deel(1)));
3      // modification of the partition of strain
4      feel(2)  -= detozz;
5      // jacobian terms
6      dfeel_ddetozz(2)=-1;
7      dfetozz_ddetozz  = real(0);
8      dfetozz_ddeel(2) = 1.;
9      dfetozz_ddeel(0) = D(0,2)/D(2,2);
10     dfetozz_ddeel(1) = D(1,2)/D(2,2);
11 }

```

Fig. B.19. Additional terms for the generic treatment of the plane stress modelling hypothesis.

be reused without modification. To achieve this, the axial strain $\varepsilon_{zz}^{\text{to}}$ is introduced as supplementary state variable and introduction in Eq. (B.1):

$$f_{\varepsilon^{\text{el}}} = \Delta \varepsilon^{\text{el}} + \dots - \Delta \varepsilon^{\text{to}} - \Delta \varepsilon_{zz}^{\text{to}} \vec{e}_z \otimes \vec{e}_z.$$

The additional derivative is obvious:

$$\frac{\partial f_{\varepsilon^{\text{el}}}}{\partial \Delta \varepsilon_{zz}^{\text{to}}} = \vec{e}_z \otimes \vec{e}_z.$$

The associated implicit equation $f_{\varepsilon_{zz}^{\text{to}}}$ is:

$$f_{\varepsilon_{zz}^{\text{to}}} = \frac{1}{E'} \sigma_{zz}|_{t+\Delta t}$$

where E' is an appropriate normalisation factor.

Using Eq. (B.2) in the case of an orthotropic behaviour leads to the following equation:

$$f_{\varepsilon_{zz}^{\text{to}}} = \frac{1}{D_{22}} [\sigma|_t + \Delta \varepsilon_{zz}^{\text{el}} + D_{02} \varepsilon_{xx}^{\text{el}} + D_{12} \varepsilon_{yy}^{\text{el}}].$$

The associated derivatives are also obvious:

$$\frac{\partial f_{\varepsilon_{zz}^{\text{to}}}}{\partial \Delta \varepsilon_{zz}^{\text{to}}} = 0 \quad \frac{\partial f_{\varepsilon_{zz}^{\text{to}}}}{\partial \Delta \varepsilon^{\text{el}}} = \frac{1}{D_{22}} \begin{pmatrix} D_{02} \\ D_{12} \\ D_{22} \\ 0 \end{pmatrix}.$$

Those modifications can be gathered in a block of code that will only be considered when treating the plane stress modelling hypothesis. An example of such treatment is provided in Fig. B.19.

One another remarkable feature of this procedure is that the procedure for computing the consistent tangent operator described in the previous section is still valid.

Appendix C. Isotropic behaviour

This section briefly discusses various topics used to formulate isotropic behaviours. We only recall classical results and definitions without elaborating on their classical physical meaning.

C.1. The equivalent VON MISES stress

We will now make the assumption that the plastic or viscoplastic behaviour is associated and governed by a VON MISES stress.

The VON MISES is given by:

$$\sigma_{\text{eq}} = \sqrt{\underline{\underline{\sigma}} : \underline{\underline{M}} : \underline{\underline{\sigma}}} \quad (\text{C.1})$$

where $\underline{\underline{M}}$ is the fourth order tensor defined by:

$$\underline{\underline{M}} = \frac{3}{2} \left[\underline{\underline{I}} - \frac{1}{3} \underline{\underline{I}} \otimes \underline{\underline{I}} \right]. \quad (\text{C.2})$$

C.2. Flow direction

Many flow directions are given by the normal tensor \underline{n} :

$$\underline{n} = \frac{\partial \sigma_{eq}}{\partial \underline{\sigma}} = \frac{1}{\sigma_{eq}} \underline{M} : \underline{\sigma}. \quad (C.3)$$

The derivative of \underline{n} with respect to $\underline{\sigma}$ is given by:

$$\frac{\partial \underline{n}}{\partial \underline{\sigma}} = \frac{\partial}{\partial \underline{\sigma}} \left(\frac{1}{\sigma_{eq}} \underline{M} : \underline{\sigma} \right) = \frac{1}{\sigma_{eq}} \underline{M} - \frac{1}{\sigma_{eq}} \underline{n} \otimes \left[\underline{M} : \underline{\sigma} \right] = \frac{1}{\sigma_{eq}} \left[\underline{M} - \underline{n} \otimes \underline{n} \right]. \quad (C.4)$$

If the elastic behaviour is linear and isotropic, the derivative of \underline{n} with respect to the elastic strain is:

$$\frac{\partial \underline{n}}{\partial \underline{\varepsilon}^{el}} = \frac{2\mu}{\sigma_{eq}} \left[\underline{M} - \underline{n} \otimes \underline{n} \right].$$

Appendix D. Implicit implementation of the DIFF2 damage behaviour

In this appendix, no viscoplastic nor plastic flow is considered. The state variable of the behaviour is the elastic strain $\underline{\varepsilon}^{el}$, the ε_i^c strains, their maximum values ε_i^{cm} .

The $\varepsilon_i^{cm}|_{t+\Delta t}$ can easily be eliminated from the system and be computed once the other values are known. In front such variables are called auxiliary state variables.

The numerical integration of the DIFF2 behaviour is done using a classical implicit scheme.¹⁰

The Hooke law (12) allows the definition at the stresses $\underline{\sigma}|_{t+\theta \Delta t}$ and $\underline{\sigma}|_{t+\Delta t}$.

$$\begin{cases} \underline{\sigma}|_{t+\theta \Delta t} = \lambda|_{t+\theta \Delta t} \text{tr} \left(\underline{\varepsilon}^{el}|_{t+\theta \Delta t} \right) + 2\mu|_{t+\theta \Delta t} \underline{\varepsilon}^{el}|_{t+\theta \Delta t} \\ \underline{\sigma}|_{t+\Delta t} = \lambda|_{t+\Delta t} \text{tr} \left(\underline{\varepsilon}^{el}|_{t+\Delta t} \right) + 2\mu|_{t+\Delta t} \underline{\varepsilon}^{el}|_{t+\Delta t}. \end{cases} \quad (D.1)$$

Discretisation of Eq. (11) leads to the equation associated with the elastic strain:

$$f_{\underline{\varepsilon}^{el}} = \Delta \underline{\varepsilon}^{el} + \sum_{i=1}^3 \Delta \varepsilon_i^c \underline{n}_i^c.$$

The associated jacobian terms are:

$$\frac{\partial f_{\underline{\varepsilon}^{el}}}{\partial \Delta \underline{\varepsilon}^{el}} = \underline{I} \quad \frac{\partial f_{\underline{\varepsilon}^{el}}}{\partial \Delta \underline{\varepsilon}^{vis}} = \underline{I} \quad \frac{\partial f_{\underline{\varepsilon}^{el}}}{\partial \Delta \varepsilon_i^c} = \underline{n}_i^c.$$

The evolution of the crack strains ε_i^c is treated in a purely implicit manner. Depending on the state of the material in the stress–crack strain graph depicted in Fig. 11(b), Eq. (14) or Eq. (15) governs this evolution:

$$\begin{cases} f_{\varepsilon_i^c} = \sigma_i|_{t+\Delta t} - \sigma^R - R_p \left(\varepsilon_i^c + \Delta \varepsilon_i^c \right) & \text{(case ②, Eq. (14))} \\ f_{\varepsilon_i^c} = \frac{1}{E} \left(\sigma_i|_{t+\Delta t} - \frac{\sigma^R - R_p \varepsilon_i^{cm}}{\varepsilon_i^{cm}} \left(\varepsilon_i^c + \Delta \varepsilon_i^c \right) \right) & \text{(case ③, Eq. (15))} \\ f_{\varepsilon_i^c} = \varepsilon_i^c + \Delta \varepsilon_i^c & \text{otherwise.} \end{cases}$$

In each case, the equation involved is linear and its derivatives can be computed in a straightforward manner using the following result:

$$\frac{\partial \sigma_i|_{t+\Delta t}}{\partial \Delta \underline{\varepsilon}^{el}} = \underline{n}_i; \quad \frac{\partial \sigma_i|_{t+\Delta t}}{\partial \Delta \underline{\varepsilon}^{el}} = \lambda|_{t+\Delta t} \underline{I} + 2\mu|_{t+\Delta t} \underline{n}_i \quad (D.2)$$

where we used the fact that $\underline{n}_i : \underline{I} \otimes \underline{I} = \underline{I}$, or equivalently $\text{tr}(\underline{n}_i) = 1$.

¹⁰ When considered, the viscoplastic flow are treated in a semi-implicit manner while crack strains are treated using a fully implicit scheme.

References

- [1] V. Bouineau, M. Lainet, A. Courcelle, M. Pelletier, Toward an improved GERMINAL v2 code to model oxide fuel for sodium fast reactor, in: Structural Mechanics in Reactor Technology, New Delhi, India, 2011.
- [2] V. Marelle, J. Sercombe, B. Michel, R. Tawizgant, Thermo-mechanical modeling of PWR fuel with ALCYONE, in: Water Reactor Fuel Performance Meeting, Chengdu, China, 2011.
- [3] G. Thouvenin, D. Baron, N. Largenton, R. Largenton, P. Thevenin, EDF CYRANO3 code, recent innovations, in: LWR Fuel Performance Meeting/ TopFuel/WRFPM, Orlando, Florida, USA, 2010.
- [4] V. Marelle, J. Gatt, MTR plates 3D mechanical modeling with MAIA, in: 31st International Meeting on Reduced Enrichment for Research and Test Reactors, Beijing, China, 2009.
- [5] T. Helfer, E. Brunon, E. Castelier, A. Ravenet, N. Chauvin, The fuel performance code celaeno, conception and simulation of fuel elements for gas-cooled fast reactor, in: Proceedings of GLOBAL 2009 Conference on Advanced Nuclear Fuel, 2009.
- [6] EDF, Code_Aster web site, 2013. URL: <http://www.code-aster.org>.
- [7] CEA, Cast3M web site, 2013. URL: <http://www-cast3m.cea.fr/>.
- [8] B. Stroustrup, C. Eberhardt, Le Langage C++, Pearson Education, Paris, 2004.
- [9] D. Martin, The elastic constants of polycrystalline UO₂ and (U, Pu) mixed oxides: a review and recommendations, High Temp. - High Press. 21 (1989) 13–24.
- [10] G. Von Rossum, Python Library Reference, 2007, URL: <http://docs.python.org>.
- [11] R. Foerch, J. Besson, G. Cailletaud, P. Pilvin, Polymorphic constitutive equations in finite element codes, Comput. Methods Appl. Mech. Engrg. 141 (1997) 355–372.
- [12] J. Besson, R. Leriche, R. Foerch, G. Cailletaud, Object-oriented programming applied to the finite element method part II. Application to material behaviors, Rev. Eur. Eléments 7 (1998) 567–588.
- [13] Northwest Numerics and Modeling, Inc., Zebfront, 2014. URL: <http://www.nwnumerics.com/Z-mat/ZebFront>.
- [14] O. Zienkiewicz, The Finite Element Method, McGraw-Hill, 1977.
- [15] J. Besson, G. Cailletaud, J.-L. Chaboche, Mécanique non linéaire des matériaux, Hermès, Paris, 2001.
- [16] EDF, Algorithme non linéaire quasi-statique STAT_NON_LINE, Référence du Code Aster R5.03.01 révision : 10290, EDF-R&D/AMA, 2013, URL: <http://www.code-aster.org>.
- [17] J.C. Simo, R.L. Taylor, Consistent tangent operators for rate-independent elastoplasticity, Computer Methods in Applied Mechanics and Engineering 48 (1985) 101–118.
- [18] J.C. Simo, T.J.R. Hughes, Computational Inelasticity, Springer, New York, 1998.
- [19] J.-L. Chaboche, J. Lemaitre, A. Benallal, R. Desmorat, Mécanique des matériaux Solides, Dunod, Paris, 2009.
- [20] C. Miehe, N. Apel, M. Lambrecht, Anisotropic additive plasticity in the logarithmic strain space: modular kinematic formulation and implementation based on incremental minimization principles for standard materials, Computer Methods in Applied Mechanics and Engineering 191 (2002) 5383–5425.
- [21] EDF, Loi de comportement en grandes rotations et petites déformations, Référence du Code Aster R5.03.22 révision : 11536, EDF-R&D/AMA, 2013, URL: <http://www.code-aster.org>.
- [22] I. Doghri, Mechanics of Deformable Solids: Linear, Nonlinear, Analytical, and Computational Aspects, Springer, Berlin, New York, 2000.
- [23] EDF, Modèles de grandes déformations GDEF_LOG et GDEF_HYPO_ELAS, Référence du Code Aster R5.03.24 révision : 10464, EDF-R&D/AMA, 2013, URL: <http://www.code-aster.org>.
- [24] J. Simo, C. Miehe, Associative coupled thermoplasticity at finite strains: Formulation, numerical analysis and implementation, Computer Methods in Applied Mechanics and Engineering 98 (1992) 41–104.
- [25] V. Tvergaard, effect of fibre debonding in a whisker reinforced metal, Mater. Sci. Eng. A125 (1990) 203–213.
- [26] A. Fortin, Analyse numérique pour ingénieurs, Presses internationales Polytechnique, [Montréal], 2001.
- [27] J. Besson, D. Desmorat, Numerical implementation of constitutive models, in: J. Besson (Ed.), Local Approach to Fracture, Ecole des Mines de Paris-les presses, 2004.
- [28] H.-S. Chen, M.A. Stadtherr, A modification of powell's dogleg method for solving systems of nonlinear equations, Comput. Chem. Eng. 5 (1981) 143–150.
- [29] EDF, Macro-commande SIMU_POINT_MAT, Référence du Code Aster U4.51.12 révision 9069, EDF-R&D/AMA, 2013. URL: <http://www.code-aster.org>.
- [30] P. Pilvin, SiDoLo Version 2.4495 Notice d'utilisation, Laboratoire Génie Mécanique et Matériaux, Université de Bretagne Sud, 2003.
- [31] Salome, ADAO, a SALOME module for data assimilation and optimization, 2014. URL: <http://www.salome-platform.org/>.
- [32] Salome, The open source integration platform for numerical simulation, 2014. URL: <http://www.salome-platform.org/>.
- [33] R. Mustata, D.R. Hayhurst, Creep constitutive equations for a 0.5Cr 0.5 mo 0.25 V ferritic steel in the temperature range 565°C–675°C, Int. J. Press. Vessels Pip. 82 (2005) 363–372.
- [34] EDF, Comportement viscoplastique avec endommagement de Hayhurst, Référence du Code Aster R5.03.13 révision : 8886, EDF-R&D/AMA, 2012, URL: <http://www.code-aster.org>.
- [35] EDF, Modèle d'endommagement de Mazars, Référence du Code Aster R7.01.08 révision : 10461, EDF-R&D/AMA, 2013. URL: <http://www.code-aster.org>.
- [36] L. Méric, G. Cailletaud, Single crystal modelling for structural calculations, J. Eng. Mater. Technol. 113 (1991) 171–182.
- [37] EDF, Comportements élastoviscoplastiques mono et polycristallins, Référence du Code Aster R5.03.11 révision : 10623, EDF-R&D/AMA, 2013, URL: <http://www.code-aster.org>.
- [38] G. Monnet, S. Naamane, B. Devincere, Orowan strengthening at low temperatures in bcc materials studied by dislocation dynamics simulations, Acta Mater. 59 (2011) 451–461.
- [39] M. Berveiller, A. Zaoui, An extension of the self-consistent scheme to plastically-flowing polycrystals, J. Mech. Phys. Solids 26 (1978) 325–344.
- [40] EDF, Modèle de comportement élasto-visqueux META.LEMA.ANI avec prise en compte de la métallurgie pour les tubes de gaine du crayon combustible, Documentation du Code-Aster R4.04.05 EDF-R&D/AMA, 2013. URL: <http://www.code-aster.org>.
- [41] J. Mazars, F. Hamon, S. Grange, A new 3D damage model for concrete under monotonic, cyclic and dynamic loadings, Mater. Struct. (2014) 1–15.
- [42] J. Chaboche, G. Cailletaud, Integration methods for complex plastic constitutive equations, Comput. Methods Appl. Mech. Engrg. 133 (1996) 125–155.
- [43] R.P. Brent, Algorithms for Minimization Without Derivatives, Dover Publications, 1973.
- [44] N. Rupin, J.-M. Proix, F. Latourte, G. Monnet, Simulation de la réponse mécanique d'un acier inoxydable austénitique à l'aide de calculs cristallins, in: Actes du congrès CSMA, 2011.
- [45] X. Han, B. Tanguy, J. Besson, S. Forest, A micromechanical modeling for irradiated austenitic stainless steels, in: IUTAM Symposium on Advanced Materials Modelling for Structures, Paris, 2012.
- [46] A. Shterenlikht, N.A. Alexander, Levenberg-marquardt vs powell's dogleg method for gursun-tvergaard-needleman plasticity model, Computer Methods in Applied Mechanics and Engineering 237–240 (2012) 1–9.
- [47] B. Michel, J. Sercombe, G. Thouvenin, R. Chatelet, 3D fuel cracking modelling in pellet cladding mechanical interaction, Eng. Fract. Mech. 75 (2008) 3581–3598.
- [48] K. Lassmann, The structure of fuel element codes, Nucl. Eng. Des. 57 (1980) 17–39.
- [49] K. Lassmann, TRANSURANUS: a fuel rod analysis code ready for use, J. Nucl. Mater. 188 (1992) 295–302.
- [50] P. Garcia, C. Struzik, M. Agard, V. Louche, Mono-dimensional mechanical modelling of fuel rods under normal and off-normal operating conditions, Nucl. Eng. Des. 216 (2002) 183–201.
- [51] T. Helfer, Etude de l'impact de la fissuration des combustibles nucléaires oxyde sur le comportement normal et incidentel des crayons combustible (Ph.D. thesis), Ecole Centrale de Lyon, 2006.
- [52] Y. Monerie, J.-M. Gatt, Overall viscoplastic behavior of non-irradiated porous nuclear ceramics, Mech. Mater. 38 (2006) 608–619.

- [53] F. Schmitz, J. Papin, High burnup effects on fuel behaviour under accident conditions: the tests CABRI REP-Na, *J. Nucl. Mater.* 270 (1999) 55–64.
- [54] J. Frizonnet, J. Breton, H. Rigat, J. Papin, The main outcomes from the interpretation of the cabri rep-na experiments from ria study, in: *Proceedings of the International Topical Meeting on Light Water Reactor Fuel Performance*, Portland, Oregon, 1997.
- [55] K. Driesen, U. Hölzle, The direct cost of virtual function calls in c++, in: *ACM Sigplan Notices*, vol. 31, 1996, pp. 306–323.
- [56] T. Veldhuizen, *Techniques for Scientific C++*, 1999.
- [57] D. Abrahams, A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and beyond*, Addison-Wesley, Boston, 2004.
- [58] M. Bornert, T. Bretheau, P. Gilormini, Homogénéisation en mécanique des matériaux 1: matériaux aléatoires élastiques et milieux périodiques (traité MIM série alliages métalliques, alliages métalliques), in: *Hermes Science*, Paris, 2001.
- [59] J.G. Siek, A. Lumsdaine, The matrix template library: A generic programming approach to high performance numerical linear algebra, in: *Computing in Object-Oriented Parallel Environments*, Springer, 1998, pp. 59–70.
- [60] A. Stepanov, M. Lee, The Standard Template Library, Technical Report, WG21/N0482, ISO Programming Language C++ Project, 1994.
- [61] D. Gregor, B. Stroustrup, J. Järvi, G.D. Reis, J. Siek, A. Lumsdaine, Concepts: Linguistic support for generic programming in c, in: *SIGPLAN Notices*, ACM Press, 2006, pp. 291–310.
- [62] J.O. Coplien, Curiously recurring template patterns, *C++ Report 7* (1995) 24–27.
- [63] B. Stroustrup, G.D. Reis, A. Sutton, *Concepts Lite: Constraining Templates with Predicates*, 2013.
- [64] J. Järvi, J. Willcock, A. Lumsdaine, Concept-controlled polymorphism, in: Frank Pfennig, Yannis Smaragdakis (Eds.), *Generative Programming and Component Engineering*, in: LNCS, vol. 2830, Springer Verlag, 2003, pp. 228–244.
- [65] D. Vandevoorde, N. Josuttis, *C++ Templates: The Complete Guide*, Addison Wesley, 2002.
- [66] N.C. Myers, Traits: a new and useful template technique, *C++ Report 7* (1995) 32–35.
- [67] A. Alexandrescu, *Modern C++ Design: Applied Generic Programming and Design Patterns*, Addison-Wesley, Boston, MA, London, 2000.
- [68] Free Software Foundation, GNU compiler collection (GCC), 2014. URL: <http://gcc.gnu.org/>.